

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1102a

August 1989

XP
A Common Lisp Pretty Printing System

by

Richard C. Waters

Abstract

XP provides efficient and flexible support for pretty printing in Common Lisp. Its single greatest advantage is that it allows the full benefits of pretty printing to be obtained when printing data structures, as well as when printing program code.

XP is efficient, because it is based on a linear time algorithm that uses only a small fixed amount of storage. XP is flexible, because users can control the exact form of the output via a set of special `format` directives. XP can operate on arbitrary data structures, because facilities are provided for specifying pretty printing methods for any type of object.

XP also modifies the way abbreviation based on length, nesting depth, and circularity is supported so that they automatically apply to user-defined functions that perform output—e.g., print functions for structures. In addition, a new abbreviation mechanism is introduced that can be used to limit the total number of lines printed.

Copyright © Massachusetts Institute of Technology, 1989

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens corporation, in part by the Microelectronics and Computer Technology Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-88-K-0487. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, neither expressed nor implied, of the sponsors.

Contents

1. Introduction	1
Setting Up XP	2
2. Pretty Printing	3
Additional Printer Control Variables	3
Extensions to Output Functions	5
3. Dynamic Control of the Arrangement of Output	7
Logical Blocks	8
Conditional Newlines	11
Indentation	14
Pretty Printing as Selection	15
Tabbing Within a Section	15
User-Defined Format Directives	16
Abbreviation	18
Functional Interface	20
4. Pretty Printing Types of Objects	24
5. Second Order Details	29
Extended Features	29
Ugly Facts About Portability	30
Doing Things Right	34
Historical Note	37
Bibliography	38
Functional Summary and Index	39

Acknowledgments. A number of people have made important contributions to the development of *XP*. K. Pitman, C. Rich, Y. Feldman, G. Steele, and D. Moon, as well as J. Healy, R. Robbins, P. Anagnostopoulos, D. Chapman, and B. Morrison made suggestions that led to a number of very significant improvements in *XP*. Particular thanks are due to the ad hoc pretty printer subcommittee (S. Haflisch, J. laubsch, S. Loosmore, D. Pierson, and W. van Roggen) of the Common Lisp standardization committee (X3J13), for their efforts in getting *XP* to the point where it could be incorporation into the proposed Common Lisp standard.

1. Introduction

XP is a portable pretty printer for Common Lisp. As a pretty printer for Lisp code, **XP** has the advantage that it is fast and allows the user to easily change the way code is formatted. Beyond this, unlike most pretty printers, **XP** provides comprehensive facilities for pretty printing data structures.

Four levels of use. **XP** can be used at four different levels. At the lowest (weakest) level, you merely have to install **XP** as described below. This will give you the benefits of more attractive and faster pretty printing. If this is all you want to do, you need not read anything more than this introduction.

Moving to a slightly higher level, Section 2 describes several variables that can be used to control **XP** and some simple additional functionalities provided by **XP**.

Section 3 exposes the heart of the approach taken by **XP** and describes how users can control the layout of their output. Two interfaces are provided for this: one via a set of `format` directives and one via a set of functions.

The highest level of using **XP** is discussed in Section 4. Facilities are provided for specifying how particular types of objects should be pretty printed. These facilities can be used to define (or redefine) the way program code and data structures are displayed.

Section 5 begins by describing a couple of advanced features of **XP** not discussed in the first four sections. It then describes a number of shortcomings of **XP** that stem from the fact that it is supported as a portable package of functions and macros rather than being tightly integrated with a particular Common Lisp. A number of things cannot be supported quite as they should be, because Common Lisp does not provide all the necessary hooks. Section 5 concludes by describing how proper integration with a Common Lisp implementation can be achieved.

XP son of **PP** [7] son of **Gprint** [5, 6] son of `#print` is the latest in a line of pretty printers that goes back 12 years. All of these printers use essentially the same basic algorithm and conceptual interface. The algorithm used by `#print`, **Gprint**, **PP**, and **XP** has been independently developed by a number of people [2, 3] in addition to the author. This paper does not go into a detailed discussion of the algorithm since it is fully discussed elsewhere (see [3, 5]). However, it should be noted that the algorithm is an inherently fast linear algorithm that uses very little storage. As a result, pretty printing need not be significantly slower than ordinary printing.

In the Summer of 1989, the Common Lisp standardization committee (X3J13) decided to adopt the **XP** interface as part of their proposed Common Lisp standard. As reflected in this document (in comparison with the version of **XP** released in March, 1989 [8]) a number of relatively minor changes had to be made in **XP**, in order for it to be accepted. The exact interface as accepted by X3J13 is described in Sections 2–4. The extended features in the beginning of Section 5 were not accepted by the committee.

Setting Up XP

To use XP, the file containing it has to be loaded. At the MIT AI Laboratory, XP resides in the file "`b:>lmlib>xp.lisp`". Compiled versions exist for Symbolics and Lucid (Sun) Common Lisp.

The source for XP can be obtained over the INTERNET by using FTP. Connection should be made to the `TRIX.AI/MIT.EDU` machine. Login as 'anonymous' and copy the files shown below. It is advisable to run the tests in `xptest.lisp` after compiling XP for the first time on a new system. A comment at the beginning of the file describes how to run the tests.

```
files on TRIX.AI/MIT.EDU, (INTERNET number 128.52.32.6)
/com/ftp/pub/xp/xp.lisp      source code
/com/ftp/pub/xp/xptest.lisp   tests
/com/ftp/pub/xp/xpdoc.txt    brief documentation
```

As XP is being made available free of charge, it is being distributed *as is*, with no warranty of any kind either expressed or implied including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, and further including no warranty as to conformity with this manual or any other literature that may be issued from time to time. In addition, if you wish to use XP for anything other than your own experimental use, you will have to get a license from MIT. Information about obtaining a non-exclusive, royalty-free license can be obtained by sending a message to "dick@ai.mit.edu".

- `xp::install &key (:pkg *package*) (:macro nil) (:shadow T) (:remove nil) ⇒ T`

Calling this function sets up XP for use in the package :`pkg`. The argument :`pkg` can either be a package, a package name, or a symbol whose name is the name of a package. It defaults to the current package. The examples in this manual assume that the form (`xp::install`) has been evaluated.

The package "XP" is used in :`pkg`. If :`macro` is not `nil`, the # macro character syntax "#..." (p. 29) is set up. If :`shadow` is not `nil`, the symbols `xp::write`, `xp::print`, `xp::prin1`, `xp::princ`, `xp::pprint`, `xp::format`, `xp::terpri`, `xp::fresh-line`, `xp::writeln`, `xp::write-string`, `xp::write-char`, `xp::write-to-string`, `xp::princ-to-string`, `xp::prin1-to-string`, `xp::finish-output`, `xp::force-output`, `xp::clear-output`, and `xp::defstruct` are shadowing imported into :`pkg`. This introduces these functions into :`pkg` in place of the standard printing functions. If :`shadow` is `nil` you will have to refer to these functions with explicit package prefixes whenever you wish to print something using XP.

If :`remove` is not `nil`, the effects of having previously installed XP are undone. In particular, the package is unused and any shadowing is undone. However, any changes to the readable are left in place.

2. Pretty Printing

Additional Printer Control Variables

XP supports all the standard printer control variables (see [4]). In addition, it supports several new control variables that are relevant to its printing method. None of these new variables has any effect when `*print-pretty*` is `nil`.

- **`*print-pprint-dispatch*`** default value causes standard pretty printing

This variable holds a *pprint dispatch table* (see Section 4) that controls the way pretty printing is performed. It is initialized to a value that causes traditional Lisp pretty printing. Altering this table, or setting `*print-pprint-dispatch*` to a different table alters the style of pretty printing.

- **`*print-right-margin*`** default value `nil`

When this variable is `nil`, XP queries the output stream in an attempt to find out how much space is available for printing. If this variable is not `nil`, its value is used as the right margin to be used when pretty printing. By introducing line breaks, XP attempts to prevent any line of output from going beyond `*print-right-margin*`.

The left margin for printing is set to the column position where the output begins (or 0 if this position cannot be determined). Except for unconditional newlines (see page 13) and negative indentation (see page 14), XP will not allow anything but white space to appear to the left of the left margin.

- **`*print-miser-width*`** default value 40

A fundamental problem with pretty printing arises when printing deeply nested structures. As line breaks have to be inserted at deeper and deeper levels of nesting, the indentation gets greater and greater. This causes the line width available for printing to get smaller and smaller until it is no longer possible to print substructures in the space available.

An approach to dealing with this problem, which has been used at least since the original Goldstein pretty printer [1], is to introduce a special compact kind of layout (called ‘miser’ style) and to use this style once the line width begins to get small. The key idea behind miser style is that by trading off readability for compactness, it reduces the width necessary for printing a structure and limits the increase in indentation when further descending into the structure.

XP switches to an extra compact style of output whenever the width available for printing a substructure is less than or equal to `*print-miser-width*`. If `*print-miser-width*` is given the value `nil`, miser style is never used.

A constant default value of 40 for `*print-miser-width*`, irrespective of the right margin, is used because the point at which miser style should be triggered does not depend on the total line width, but rather on the length of the minimal unbreakable units in the output being printed. When pretty printing programs, the size of these units depends on the lengths of the symbols in the program. Experience suggests that

print-miser-width should be set at from two to four times the length of the typical symbol.

- ***print-lines*** default value `nil`

When not `nil`, this variable controls the number of lines that will be printed by a single pretty printing event. If an attempt is made to pretty print more than ***print-lines*** lines, “`...`” is printed at the end of the last line followed by all of the suffixes (closing delimiters) that are pending to be printed. Pretty printing is then halted.

```
(pprint '(progn (setq a 1 b 2 c 3 d 4)))
```

With ***print-lines*** 3 and line width 25 prints:

```
(PROGN (SETQ A 1  
        B 2  
        C 3 ...))
```

(The symbol “`...`” is printed to ensure that a reader error will occur if the output is later read. A symbol different than “`...`” is used to indicate that a different kind of abbreviation has occurred.)

Experience has shown that abbreviation based on the number of lines printed can be much more useful than the traditional depth and length abbreviation mechanisms. This is particularly true when the user wants to limit output to a small space. To do this with depth and length abbreviation, the length and depth have to be limited to very small values such as 3 or 4. This often has the unfortunate effect of producing output that consists almost totally of “`*`”s and “`...`”s grouped in parentheses. In contrast, limiting the total number of lines printed to 3 or 2, or even 1 produces legible output. Seeing the first few lines of output is usually more informative than seeing only the top level skeletal structure of the output. (The function `pp` (p. 29) can be used to reprint an abbreviated object in full.)

- ***print-shared*** default value `nil`

This variable is not part of current Common Lisp, but is slated to become part of the new Common Lisp standard. Inasmuch as this is the case, support for it is provided as part of XP. If ***print-circle*** is not `nil`, then the value of ***print-shared*** is consulted to determine whether `#n#` markers are used to indicate sharing as well as circularities. They are used to indicate sharing if and only if ***print-shared*** is not `nil`.

```
(pprint '(setq a #1=(car x) b #1#))
```

With ***print-circle*** `T` and ***print-shared*** `T` prints:

```
(SETQ A #1=(CAR X) B #1#)
```

With ***print-circle*** `T` and ***print-shared*** `nil` prints:

```
(SETQ A (CAR X) B (CAR X))
```

The variable ***print-shared*** is given a default value of `nil`, because experience suggests that using `#n#` markers to indicate sharing often reduces the readability of output. However, it is interesting to note that this is actually more computationally expensive than running with ***print-shared*** `T`. The best thing of all in most situations is to run with ***print-circle*** `nil` and use ***print-lines*** to prevent runaway printing.

Extensions to Output Functions

XP alters the standard printing functions (see the description of `xp::install`) so that they operate via XP when (and only when) `*print-pretty*` is not `nil`. In addition, XP provides some additional functionalities

- **`write object &key :dispatch :right-margin :lines :miser-width ... ⇒ object`**

The list of keywords accepted by the standard output function `write` is extended by adding four more keywords corresponding to the first four control variables above.

- **`~W`**

provides a new `format` directive `~W`, which bares the same basic relationship to `write` that `~S` and `~A` bare to `prin1` and `princ`. In particular, `~W` prints the corresponding argument obeying every printer control variable, rather than overriding the value of `*print-escape*`. In addition, `~W` interacts correctly with depth abbreviation, rather than resetting the depth counter to zero. To get the best results when using XP you should use `~W` whenever possible instead of `~S` or `~A`.

Unlike `~S` and `~A`, `~W` does not accept parameters. If given the colon modifier, `~W` binds `*print-pretty*` to `T`. If given the atsign modifier, `~W` binds `*print-level*` and `*print-length*` to `nil`.

- **`formatter control-string ⇒ function`**

The control strings used by `format` are essentially programs that perform printing. Given that almost all of these strings are constants, there is no reason why they cannot be compiled into highly efficient code. In fact, most languages other than Lisp have always compiled their `format`-like constructs. However, every implementation of Common Lisp known to the author operates on `format` control strings in a purely interpretive manner.

XP supports a macro (`formatter`) that makes it possible to obtain the efficiency of using a function to perform output without loosing the compactness of `format` strings. The `control-string` is identical in every respect to a `format` control string. The macro `formatter` expands into an equivalent function quoted `lambda` that performs the indicated printing. The argument signature of this `lambda` has the following form.

`formatter-function stream &rest args ⇒ unused-args`

The first argument is the stream to send the output to. The remaining arguments are the values to be printed. The function the macro `formatter` expands into returns any unused tail of the arguments it receives as its value.

In support of the above, `format` is extended so that it accepts functions as its second argument as well as strings. The directives `~?` and `~{~}` with no body are also extended so that they accept functions as well as control strings. (This recovers the functionality of the original MacLisp `format` directive `~Q`.)

When a function is provided in any of these situations, it must be a function of the form created by `formatter`. It is called with the appropriate output stream as its first argument and data arguments as its remaining arguments. The function should perform whatever output is necessary and return the unused tail of the arguments (if any). (This

return value is needed for the proper operation of the directives `~@?` and `~{~}` with no body.)

As an example of using `formatter` consider the following. Note also that `formatter` is not limited to appearing only as the second argument to `format`. It can appear anywhere—e.g., passed to some other function that will eventually pass it to `format` or used for some totally different purpose.

```
(format T (formatter "~~%Prices: ~@{~S~~, ~}~") 1.50 3.23 4.50) ≡
(format T #'(lambda (stream &rest args)
  (terpri stream)
  (write-string "Prices: " stream)
  (loop (prin1 (pop args) stream)
        (if (null args) (return nil))
        (write-string ", " stream)))
  args)
 1.50 3.23 4.50)
```

Prints:

Prices: 1.5, 3.23, 4.5

Using compiled functional `format` control strings instead of interpreted ones approximately triples the speed of XP when running on the Symbolics Lisp Machine. You are strongly encouraged to use functional `format` control strings whenever possible. (Using the syntax `#"..."` (p. 29) makes this particularly easy to do.)

3. Dynamic Control of the Arrangement of Output

Through the introduction of several **format** directives, XP allows the user to exercise precise control over what should be done when a piece of output is too large to fit in the line width available for displaying it. The discussion below assumes that the reader has a basic understanding of the function **format** and the standard **format** directives, as described in [4].

Basic concepts. Three concepts underlie the way XP supports the dynamic arrangement of output—*logical blocks*, *conditional newlines*, and *sections*. The positions of logical blocks and conditional newlines are specified by means of **format** directives. Together, these directives divide the output as a whole into a hierarchy of sections within sections.

The first line of Figure 3.1 shows a schematic piece of output. The characters in the output are represented by “-”s. 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 always treated as a logical block and forms one (outermost) section. This section is indicated by the 0’s on the second line of Figure 3.1. Each conditional newline defines two additional 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 as a whole.

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 3.1 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 Figure 3.1, 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-->
000000000000000000000000000000
11 111111111111111111111111111111
    22 222
        333 3333
    44444444444444 44444
```

Figure 3.1: Example of logical blocks, conditional newlines, and sections.

It also makes sense to talk about the section immediately containing a given logical block—i.e., the shortest section containing the logical block. Note that this section immediately contains every conditional newline that is immediately contained in the block. In Figure 3.1, the outermost logical block is immediately contained in the section marked with 0's, the second logical block is immediately contained in the section before the fourth conditional newline, and the innermost logical block is immediately contained in the section after the first conditional newline.

Whenever possible, XP prints the entire contents of a given section on the same line. However, if a section is too long to fit in the line width available, XP inserts line breaks at one or more conditional newline positions within the section—printing the section on more than one line. The pretty printing algorithm uses internal buffering of the output so that it can determine which way to print a section. The algorithm is fast, because the amount of lookahead required is strictly limited by the maximum line width available for printing.

Logical Blocks

If `::> is used to terminate a `<...`>, the directive delimits a logical block. In addition, the directive descends into the corresponding **format** argument (which should be a list).

```
(format T "+ ~<Roads ~<~W, ~:_~W~:> ~:_ Town ~<~W~:>~:> +"
      '((elm cottonwood) (boston)))
```

With line width 50 prints:

```
+ Roads ELM, COTTONWOOD Town BOSTON +
```

With line width 25 prints:

```
+ Roads ELM, COTTONWOOD
  Town BOSTON +
```

With line width 21 prints:

```
+ Roads ELM,
  COTTONWOOD
  Town BOSTON +
```

(As discussed in the next section, the directive `:_` indicates a conditional newline. An instance of `:_` is replaced by a line break when the following section cannot fit on the end of the current line. Whenever line breaks are introduced, indentation is also introduced so that each line in a logical block begins in the same column as the logical block as a whole.)

If the atsign modifier is used with `<...`>, the directive operates on the list of remaining **format** arguments, rather than on just on the next **format** argument. All of the remaining arguments are always consumed by the `<...`>, whether or not they are actually used by the **format** string nested in the directive. The directive `~~` (termination test) can be used to exit from `<...`> if the list being processed has been exhausted.

```
(format T "+ ~@<Roads ~<~W~~, ~:_~W~:> ~:_ Town ~<~W~:>~:> +"
      '((elm) ,(boston)))
```

With line width 21 prints:

```
+ Roads ELM
  Town BOSTON +
```

The portion of a **format** control string enclosed in a `~<...~:>` directive can be divided into segments `~<prefix~;body~;suffix~:>` by `~`; directives. It is an error for the enclosed portion to be divided into more than three segments. If the portion is only divided into two segments, the suffix defaults to the null string. If the portion consists of only a single segment, both the prefix and the suffix default to the null string. The prefix and suffix must both be constant strings. They cannot contain **format** directives. The body can be any arbitrary **format** control string.

When a `~<prefix~;body~;suffix~:>` directive is processed, the prefix is printed out just before the logical block begins and the suffix is printed out just after the logical block ends. This behavior is the same as if the characters in the prefix and suffix simply appeared before and after the `~<...~:>` directive, except for the way error situations are handled and the way `~<...~:>` interacts with depth and circularity abbreviation (see page 18).

```
(format T "+ ~<Roads ~<[~;~W ~:_~W~;]~:> ~:_ Town ~<[~;~W~;]~:>~:> +"
      '((elm cottonwood) boston))
```

With line width 21 prints:

```
+ Roads [ELM
COTTONWOOD]
Town BOSTON +
```

An interesting additional feature of `~<...~:>` is illustrated by the example above. When a `~<...~:>` directive is applied to an argument that is not a list, the directive is ignored and the offending argument is printed using `~W`. Among other things, this means that while the argument is printed, the prefix and suffix are not. The soft failure of `~<...~:>` when presented with non-lists makes it possible to write robust **format** strings that produce reasonable output for a wide range of possible arguments. This is particularly useful in debugging situations.

During the processing of the **format** string nested in `~<...~:>`, arguments are taken one by one from the list passed to `~<...~:>`. If an attempt is made to access an argument at a time when the remaining portion of this argument list is not a cons, then `"."` is inserted in the output, `~W` is used to print out the remaining argument list, and the processing of the logical block is terminated, except for printing the suffix (if any). This makes it easier to write **format** strings that are robust in the face of malformed argument lists. (Note that `~^` exits only when the remaining argument list is `nil`.)

```
(format T "+ ~<Roads ~<[~;~W~~ ~:_~W~;]~:> ~:_ Town ~<[~;~W~;]~:>~:> +"
      '((elm . cottonwood) boston))
```

With line width 21 prints:

```
+ Roads [ELM
. COTTONWOOD]
Town BOSTON +
```

If the colon modifier is used with `~<...~:>`, the prefix and suffix default to "(" and ")" (respectively) instead of to the null string. Note that the prefix and suffix are printed out even when the argument corresponding to `~<...~:>` is an empty list.

```
(format T "+ ~<Roads ~:_~W~:> ~:_ Town ~:<~~~W~:>~:> +"
      '(elm cottonwood) ()))
```

With line width 21 prints:

```
+ Roads (ELM
          COTTONWOOD)
Town () +
```

If the directive `~@;` is used to terminate the prefix in a `~<...~:>` directive, the prefix is treated as a *per-line* prefix. A per-line prefix is printed at the beginning of every line in the logical block, rather than just before the start of the block as a whole. This is done in such a way that the prefixes on subsequent lines are directly below the occurrence of the prefix on the first line.

```
(format T "~<;;; ~@;Roads ~<= ~@;~W, ~:_~W~:> ~:_ Town ~<~W~:>~:>"'
      '(elm cottonwood) (boston)))
```

With line width 50 prints:

```
;;; Roads = ELM, COTTONWOOD Town BOSTON
```

With line width 25 prints:

```
;;; Roads = ELM,
;;;           = COTTONWOOD
;;;   Town BOSTON
```

If a `~<...~:>` directive is terminated with `~:@>`, then a `~:_` 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.

```
(format T "~<:(~W~) street goes to ~:(~W~).~:@>" '(main boston)) ≡
(format T "~<:(~W~) ~:_street ~:_goes ~:_to ~:_~:(~W~).~:@>"'
      '(main boston))
```

With line width 12 prints:

```
Main street
goes to
Boston.
```

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, it is an error for any of these directives to be nested within `~<...~>`. (Note that in standard `format`, it makes little sense to have anything that can cause a line break within a `~<...~>` and [4] does not define what would happen if this were the case.)

Beyond this, it is also an error for the `~<...~:_;...~>` form of `~<...~>` to be used at all in conjunction with any of these directives. This is a functionality that is rendered obsolete by per-line prefixes in `~<...~:>`.

Conditional Newlines

The `format` directive `~_` is used to specify conditional newlines. By means of the colon and atsign modifiers, the directive can be used to specify four different criteria for the dynamic insertion of line breaks: *linear-style*, *fill-style*, *miser-style*, and *mandatory-style*.

By default, whenever a line break is inserted by a conditional newline, indentation is also introduced so that the following line begins in the same column as the first character in the immediately containing logical block. This default can be changed by using the directive `~I` (p. 14).

Linear-style conditional newlines. Without any modifiers, `~_` specifies linear-style insertion of line breaks. This style calls for the subsections of a logical block to be printed either all on one line or each on a separate line.

Linear-style conditional newlines are replaced by line breaks if and only if the immediately containing section cannot be printed on one line. As soon as the line width available becomes less than the length of a given section, every linear-style conditional newline in it is replaced by a line break.

```
(format T "(LIST ~@<~W ~_~W ~_~W~:>)" 'first 'second 'third)
```

With line width ≥ 25 prints:

```
(LIST FIRST SECOND THIRD)
```

With line width < 25 prints:

```
(LIST
  FIRST
  SECOND
  THIRD)
```

Miser-style conditional newlines. If the atsign modifier is used with `~_`, the directive specifies miser-style insertion of line breaks. Miser-style conditional newlines are replaced by line breaks if and only if miser style is in effect in the immediately enclosing logical block and the immediately containing section cannot be printed on one line. Miser style is in effect for a given logical block if and only if the starting column of the logical block is less than or equal to `*print-miser-width*` columns from the end of the line. (Note that the “`(`” in the example below is not in the logical block, but rather before it.)

```
(format T "~-:@<LIST ~@_~W ~@_~W ~@_~W~:>" 'first 'second 'third)
```

With line width 10 and `*print-miser-width* < 9` prints:

```
(LIST FIRST SECOND THIRD)
```

With line width 10 and `*print-miser-width* ≥ 9` prints:

```
(LIST
  FIRST
  SECOND
  THIRD)
```

Even in miser style, the pretty printing algorithm is not guaranteed to succeed in keeping the output within the line width available. In particular, line breaks are never inserted except at conditional newline positions. As a result, a given output requires a

certain minimum amount of line width to print it. If the amount of line width available is less than this amount, characters are printed beyond the end of the line.

(Gprint [5] supported an additional mechanism for dealing with deeply nested structures. When indentation reduced the line width to a small percentage of its initial value, major program structures (such as `prog` and `let`) were radically shifted to the left by reducing the indentation to nearly zero. This violated standard Lisp pretty printing style, but significantly increased the line width available for printing. Unfortunately, experience showed that, though this was very useful in some situations, it was, in general, more confusing than helpful.)

Fill-style conditional newlines. If the colon modifier is used with `~_`, the directive specifies fill-style insertion of line breaks. This style calls for the subsections of a section to be printed with as many as possible on each line.

Fill-style conditional newlines are replaced by line breaks if and only if either (a) the following section cannot be printed on the end of the current line, (b) the preceding section was not printed on a single line, or (c) the immediately containing section cannot be printed on one line and miser style is in effect in the immediately containing logical block. If a logical block is broken up into a number of subsections by fill-style conditional newlines, the basic effect is that the logical block is printed with as many subsections as possible on each line. However, if miser style is in effect, fill-style conditional newlines act like linear-style conditional newlines.

For instance, consider the example below. The `format` control string shown uses the standard directives `~@{...~}` (iterate over arguments) and `~^` (terminate iteration) to decompose the list argument into pairs. Each pair in the list is itself decomposed into two parts using a `~<...~:>` directive. A space and a `~:_` are placed after each pair except the last.

```
(format T "(LET ~:~<~@{~:~W ~_~W~:>~^ ~:_~}~:>~_ ...)"
      '((x 4) (*print-length* nil) (z 2) (list nil)))
```

With line width 35 prints:

```
(LET ((X 4) (*PRINT-PRETTY* NIL)
      (Z 2) (LIST NIL))
  ...)
```

With line width 22 prints:

```
(LET ((X 4)
      (*PRINT-LENGTH*
       NIL)
      (Z 2)
      (LIST NIL))
  ...)
```

Note that when the line width is 35, only one line break (before “(z 2)”) has to be introduced. Once this is done, “(z 2)” and “(LIST NIL)” can both fit on the next line. Also note that when the line width is 25, there is a line break after “NIL”, even though “(z 2)” would fit on the end of the previous line. The line break is introduced, due to criteria (b) in the definition above. This criteria is used, because many people judge that output of the following form violates the basic aesthetics of Lisp pretty printing.

```
(LET ((X 4)
      (*PRINT-LENGTH*
      NIL) (Z 2)
      (LIST NIL))
  ...)
```

It is often useful to mix different kinds of conditional newlines together in a single logical block. In general, this works well without any conflicts arising between the ways the various conditional newlines work. However, it is not a good idea to have a miser-style conditional newline immediately after a fill-style conditional newline. The problem is that the miser-style conditional newline will terminate the section following the fill-style conditional newline. As a result, no account will be taken of what follows the miser-style conditional newline when deciding whether or not to insert a line break at the fill-style conditional newline. This can cause the output after the miser-style conditional newline to extend beyond the end of the line.

Mandatory-style conditional newlines. If the colon and atsign modifiers are both used with `~_`, the directive specifies mandatory insertion of a line break. Among other things, this implies that none of the containing sections can be printed on a single line. This will trigger the insertion of line breaks at linear-style conditional newlines in these sections. With regard to indentation, mandatory-style conditional newlines are treated just like any other kind of conditional newline. This makes them different from *unconditional* newlines.

Unconditional newlines. There are at least six ways to introduce a newline into the output without using `~_`. You can use the directives `~%` and `~&`. You can include a newline character in a `format` string. You can call `terpri` or `print`. You can print a string that contains a newline. Each of these methods produces an unconditional newline that must always appear in the output. Like a mandatory conditional newline, this prevents any of the containing sections from being printed on one line.

It is not completely obvious how unconditional newlines should be handled. There are two very suggestive cases, which unfortunately contradict each other. First, suppose a program containing a string constant containing a newline character is printed out and then read back in. To ensure that the result of the read will be `equal` to the original program, it is important that indentation not be inserted after the newline character in the string. On the other hand, suppose that the same program is being printed into a file by a `format` string specifying a per-line prefix of `";;;"`, with the intention that the program appear in the output as a comment. To ensure that this comment will not interfere with subsequent reading from the file, it is important that the prefix be printed when the newline in the string is printed.

In an attempt to satisfy the spirit of both of the cases above, XP applies the following heuristic. Indentation is used only if a newline is created with `~_`.

```
(format T "(LIST ~@<~W ~_~W~:>)" 'first "string on
two lines")
```

With line width 100 prints:

```
(LIST FIRST
      "string on
      two lines")
```

However, any per-line prefixes (and any indentation preceding them) are always printed no matter how a newline originates.

```
(format T "~@<;; ~@;(LIST ~@<~W ~_~W~:>)~:>" 'first "string on
two lines")
```

With line width 100 prints:

```
;;; (LIST FIRST
;;;      "string on
;;;      two lines")
```

Discarding trailing spaces. Conditional newline directives are typically preceded by some amount of blank space. This is done so that the subsections of a section will be visually separated when they are printed on a single line. However, without anything more being said, this would lead to the printing of unnecessary blank spaces at the end of most lines when line breaks are inserted. In the interest of efficiency, XP suppresses the printing of blanks at the end of a line if (and only if) the line break was caused by a `~_` directive. For instance, there are no blanks at the end of any of the lines in the examples above, except that, if there are blanks after the word “on” in the string in the two unconditional newline examples, these blanks appear in the output.

Indentation

By default, the second and subsequent output lines corresponding to a logical block are indented so that they line up vertically under the first character in the block. The `format` directive `~I` makes it possible to specify a different indentation.

The directive `~nI` specifies that the indentation within the immediately containing logical block should be set to the column position of the first character in the block plus `n`. If omitted, `n` defaults to zero. The parameter can be negative, which will have the effect of moving the indentation point to the left of the first character in the block. However, the total indentation cannot be moved left of the beginning of the line or left of the end of the rightmost per-line prefix.

The directive `~n:I` is exactly the same as `~nI` except that it operates relative to the position in the output of the directive itself, rather than relative to the position of the first character in the block.

As an example of using `~I`, consider the following: If the line width available is 25, the `~_` directive is replaced by a line break. The `~1I` directive specifies that the statement in the body of the `defun` should be printed at a relative indentation of 1 in the logical block. If the line width is 15, the `~:_` is also replaced by a line break. The `~:I` directive before the `~W` printing the function name causes the argument list to be lined up under the function name. The column position corresponding to the `~:I` is determined dynamically as the output is printed.

```
(format T "~:<~W ~@_~:I~W ~:_~W ~1I~_~W~:>
      '(defun prod (x y) (* x y)))
```

With line width 50 prints:

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

With line width 25 prints:

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

With line width 15 prints:

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

Changes in indentation caused by a `~I` directive do not take effect until after the next line break. As a result, it is important that the `~I` directives in the example above precede the `~_` directives they are supposed to affect. It should also be noted that, a `~I` directive only affects the indentation in the immediately containing logical block.

In miser style, all `~I` directives are ignored, thereby forcing the lines corresponding to the logical block to line up under the first character in the block.

With line width 15 and `*print-miser-width*` 20 prints:

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

Pretty Printing as Selection

Stepping back a moment, it is useful to reflect on how the `format` directives above interact to support pretty printing. The `~<...~:>` and `~_` directives in a `format` control string divide the output up into a hierarchy of sections within sections. The `~_` and `~I` directives simultaneously specify three ways (on one line, on multiple lines, and in miser style) for printing each section. The job of the pretty printer boils down to selecting (based on the length of the section, the line width available, and the value of `*print-miser-width*`) which of the three ways is appropriate for printing each section.

The `format` directives have been designed so that it is relatively easy to specify three different ways to print a logical block in a single `format` control string. In particular, except for line breaks, white space, (and per-line prefixes), the characters to be printed are exactly the same in each of the three styles. The `~_` and `~I` directives specify how the logical block sections are to be arranged when printed on multiple lines and in miser style.

Many other kinds of pretty printing directives could have been supported—for example, arbitrary sections of text that are output only when a section is printed on multiple lines. XP supports only the limited set of directives above, because experience has shown them to be a good compromise between the requirements of expressive power, easy understandability, and efficiency.

Tabbing Within a Section

The standard `format` directive `~T` is extended so that it supports the colon modifier in addition to the atsign modifier. If the colon modifier is specified, the tabbing computation is done relative to the beginning of the immediately containing section, rather than

with respect to the beginning of the line. (When this computation is performed, any unconditional newlines in the section are ignored.) As an example of using `~:T`, consider the following. Each street name is followed by a `~8:T`, which ensures that the total width taken up will be a multiple of 8. Fill-style conditional newlines are used to put as many streets as possible on each line.

```
(format T """:<Roads ~:@_{"W~~~8:T":_}"":>""
      '(elm main maple center))
```

With line width 25 prints:

Roads	ELM	MAIN
	MAPLE	CENTER

The fact that `~:T` operates solely within the immediately containing section means that the number of spaces to insert is independent of whatever indentation is in effect. In the example above, a column spacing of 8 is used, but the entire table is shifted over 6 columns. (Note the way the `~@_` delimits the beginning of the section containing the first road.)

(The fact that `~:T` operates solely within the immediately containing section and ignores unconditional newlines means that the amount of space to insert can be determined before deciding which (if any) conditional newlines have to be replaced with line breaks. This is essential for the efficient operation of the pretty printing algorithm.)

When the normal directive `~T` (without a colon) is processed, tabbing is computed relative to the beginning of the line and all conditional newlines are ignored. (Again, this is important so that the number of spaces to insert can be determined before making any decisions about conditional newlines.)

As a practical matter, one should not use `~T` after a conditional newline nor `~:T` after an unconditional newline.

User-Defined Format Directives

XP provides a mechanism for allowing the user to define new `format` directives. In a somewhat simplified form, this revives a feature of `format` that was left out when Common Lisp was initially developed.

XP allows a `format` string to contain a directive of the form `~/name/`. When this is the case, it is assumed that a function named `name` has been defined. This function must accept the pattern of arguments shown below.

`name stream arg colon? atsign? &rest parameters ⇒ ignored`

The colon modifier, the atsign modifier, and arbitrarily many parameters can be specified with the `~/name/` directive. This information, along with the output stream and one argument from the current argument list, are passed to the function `name`. The input `colon?` is `T` if and only if the colon modifier was specified. The input `atsign?` is `T` if and only if the atsign modifier was specified. The function `name` should perform whatever operations are required to print `arg` into `stream`. Any value returned by the function is ignored.

Packages. A key problem with `~/name/` directives derives from the fact that, as written by the user, *name* is a string, not a symbol. This string has to be converted into a symbol to identify the function. The *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 “:” or “::”, 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 `user` package.

In Common Lisp, symbols without explicit package prefixes are usually placed in the package that is contained in the variable `*package*` at the moment when the symbol is first read. To continue this policy, *name* should be placed in the package that is contained in the variable `*package*` at the moment when the `format` string is first read. Unfortunately, strings do not contain any record of the package they were read into. As a result, a fixed default package has to be used instead.

Special format directives for lists. XP provides three special `format` directives for printing lists. These are accessed using the mechanisms for user-defined `format` directives described above.

The directive `~/pprint-linear/` prints out the elements of a list either all on one line or each on a separate line. parentheses are printed around the list if the colon modifier is specified.

```
(defun pprint-linear (stream list &optional (colon? T) atsign?)
  (declare (ignore atsign?))
  (if colon?
      (format stream "~:@{~W~~ ~_~}~:>" list)
      (format stream "~<~@{~W~~ ~_~}~:>" list)))
  (format T "~:/pprint-linear/" '(one two three))
```

With line width 15 prints:

```
(ONE TWO THREE)
```

With line width 14 prints:

```
(ONE
 TWO
 THREE)
```

The directive `~/pprint-fill/` prints out the elements of a list with as many elements as possible on each line. Except for the fact that it uses `~:_` instead of `~_`, it is identical to `~/pprint-linear/`.

```
(format T "~:/pprint-fill/" '(one two three four five))
```

With line width 25 prints:

```
ONE TWO THREE FOUR FIVE
```

With line width 15 prints:

```
ONE TWO THREE
FOUR FIVE
```

The directive `~/pprint-tabular/` is similar to `~/pprint-fill/`, except that it prints the elements of the list so that they line up in a table. In addition to the colon modifier, `~/pprint-tabular/` takes a parameter (default 16) that specifies the width of columns in the table. (The definition of `pprint-tabular` is shown on page 22.)

```
(format T "^\~8:/pprint-tabular/" '(one two three four five))
```

With line width 20 prints:

```
(ONE      TWO
 THREE    FOUR
 FIVE)
```

Abbreviation

XP supports abbreviation controlled by the variables `*print-level*`, `*print-length*`, `*print-circle*`, and `*print-shared*`. In addition, (see Section 2) XP supports a new abbreviation mechanism that limits the total number of lines printed. All five mechanisms are supported in such a way that they automatically apply to user-defined functions that perform output.

Depth abbreviation. XP obeys `*print-level*` in its internal operation. In addition, it makes it very easy to write `format` control strings that obey `*print-level*`. This is done by basing depth abbreviation on the concept of logical blocks. Whenever a `~<...~:>` directive is encountered at a dynamic nesting depth in logical blocks greater than `*print-level*`, “#” is printed instead of the block. In addition, the argument (or for `~@<...~:>` arguments) that would have been consumed by the directive are skipped.

The following example illustrates how `~<...~:>` supports depth abbreviation. The most important feature of the example is that it shows that depth abbreviation is controlled by the *dynamic* nesting of `~<...~:>` directives, not their *static* nesting. In the second output shown, the statically outermost instance of `~<...~:>` in `~/pprint-linear/` (p. 17) is at a dynamic nesting depth of 3. (Note that since there is an implicit logical block dynamically wrapped around the entire output, the dynamically outermost instance of `~<...~:>` is at a dynamic nesting depth of 1.)

```
(format T "+ ~:<~W ~:<~W ~:/pprint-linear/~:>~:> +" '(1 (2 (3))))
```

With `*print-level*` nil prints:

```
+ (1 (2 (3))) +
```

With `*print-level*` 2 prints:

```
+ (1 (2 #)) +
```

Length abbreviation. XP obeys `*print-length*` in its internal operation. In addition, it makes it very easy to write `format` control strings that obey `*print-length*`. This is done by basing length abbreviation on the concept of logical blocks.

`~<...~:>` provides automatic support for length abbreviation. If `*print-length*` is not `nil`, a count is kept of the number of arguments used within the `~<...~:>`. If this count ever reaches `*print-length*`, “...” is inserted in the output and the processing of the logical block is terminated, except for printing the suffix (if any). As with depth abbreviation, the processing depends on dynamic relationships, not static ones.

```
(format T "+ ~:@{~W~~ ~}~:> +" '(1 2 3 4 5))
With *print-length* nil prints:
+ (1 2 3 4 5) +
With *print-length* 2 prints:
+ (1 2 ...) +
```

Circularity abbreviation. XP obeys ***print-circle*** and ***print-shared*** in its internal operation. In addition, it makes it very easy to write **format** control strings that obey ***print-circle*** and ***print-shared***. This is done by supporting circularity abbreviation through the combined actions of **~W** and **~<...~:>**.

In situations where ***print-circle*** is not **nil**, the following extra processing is performed. When **~W** or **~<...~:>** is applied to an object that has previously been encountered, an appropriate **#n#** marker is printed out instead of printing the argument.

In addition, if an attempt is made to access an argument from the list passed to **~<...~:>**, at a time when the remaining portion of this list has already been encountered during the printing process, **". #n#"** is inserted in the output and the processing of the logical block is terminated, except for printing the suffix (if any). This catches instances of **cdr** circularity in lists. The circularity checking described above is not applied if a **~@<...~:>** directive 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.

```
(format T "+ ~:@{~W~~ ~}~:> +" '#1=(1 2 #1# 3 . #1#))
With *print-circle* T prints:
+ #1=(1 2 #1# 3 . #1#) +
With *print-circle* T and *print-length* 2 prints:
+ (1 2 ...) +
```

Circularity detection is an inherently slow process. In particular, two entire passes have to be made over the output: one to determine what **#n=** markers should be printed and another to perform the actual printing. All and all, setting ***print-circle*** to **T** more than doubles the time required for printing using XP and should be avoided unless strictly necessary. In the interest of efficiency, XP does not print circularity abbreviation markers in situations where other abbreviation methods hide the circularity. This is illustrated in the last part of the example above.

For a **format** string to correctly support circularity abbreviation, every part of the object being printed must be seen by an occurrence of **~W** or **~<...~:>**. (If some part is skipped, XP will fail to detect circularities involving that part.) (The above criteria are also required for depth and length abbreviation to be handled in a completely correct way.)

In addition to the arguments of **~W** and **~<...~:>** the arguments of the standard printing functions such as **write**, **print**, **pprint**, **print1**, and **pprint**, as well as the arguments of the standard **format** directives such as, **~A**, and **~S** are all checked (when necessary) for circularity. However, such checking is not applied to the arguments of the functions **write-line**, **write-string**, and **write-char** or to the literal text output by **format**. A consequence of this is that you must use one of the latter functions if you want to print some literal text in the output that is not supposed to be checked for circularity or sharing.

Functional Interface

The primary interface to XP's operations for dynamically determining the arrangement of output is provided through `format`. This is done, because `format` strings are typically the most convenient way of interacting with XP. However, XP's operations have nothing inherently to do with `format` *per se*. In particular, they can also be accessed via the six functions and macros below.

- `pprint-logical-block (stream-symbol list &key :prefix :per-line-prefix :suffix)`
`&body body` ⇒ `nil`

In the manner of `~<...~:>`, this macro causes printing to be grouped into a logical block. The value `nil` is always returned.

The argument `stream-symbol` must be a symbol. If it is `nil`, it is treated the same as if it were `*standard-output*`. If it is `T`, it is treated the same as if it were `*terminal-io*`. The run-time value of `stream-symbol` must be a stream (or `nil` meaning `*standard-output*` or `T` meaning `*terminal-io*`). The logical block is printed into this destination stream.

Within the `body`, `stream-symbol` is bound to a *pretty printing stream* that supports dynamic decisions about the arrangement of output and then forwards the output to the destination stream. All and only the output sent to `stream-symbol` is treated as being in the logical block. (It is an error to send any output directly to the underlying destination stream.)

The `:suffix`, `:prefix`, and `:per-line-prefix` must all be expressions that (at run time) evaluate to strings. The argument `:suffix` (which defaults to the null string) specifies a suffix that is printed just after the logical block. The argument `:prefix` specifies a prefix to be printed before the beginning of the logical block. If the argument `:per-line-prefix` is supplied, it specifies a prefix that is printed before the block and at the beginning of each new line in the block. It is an error for `:prefix` and `:per-line-prefix` to both be supplied. If neither is supplied, a `:prefix` of the null string is assumed.

The argument `list` is interpreted as being a list that the `body` is responsible for printing (see `pprint-exit-if-list-exhausted` and `pprint-pop`). If `list` is not a list, it is printed using `write` on `stream-symbol` and the `body` is skipped along with the printing of the prefix and suffix. If `*print-circle*` is not `nil` and `list` is a cons that has already been printed by or within a dynamically containing logical block, then an appropriate `#n#` marker is printed on `stream-symbol` and the `body` is skipped along with the printing of the prefix and suffix. (If the `body` is not responsible for printing a list, then the behavior above can be turned off by supplying `nil` for the `list` argument.)

If `*print-level*` is not `nil` and the logical block is at a dynamic nesting depth of greater than `*print-level*` in logical blocks, “#” is printed on `stream-symbol` and the `body` is skipped along with the printing of the prefix and suffix.

The `body` can contain any arbitrary Lisp forms. All the standard printing functions (e.g., `write`, `princ`, `terpri`) can be used to print output into `stream-symbol`. Within a logical block, these functions interact correctly with `*print-circle*` and `*print-depth*`.

A consequence of the two-pass approach to the detection of circularity and sharing is

that the *body* of a **pprint-logical-block** must not perform any side-effects on the surrounding environment. This includes not modifying any variables that are bound outside of its scope. Obeying this restriction is facilitated by using **pprint-pop**, instead of an ordinary **pop** when traversing a list being printed by the *body* of a **pprint-logical-block**.)

From the discussion above, it can be seen that **pprint-logical-block** supports all of the functionality of `~<...~:>` except for the automatic introduction of fill-style conditional newlines supported by `~<...~:@>`. This feature is omitted, because it is a transformation on **format** strings rather than a printing operation.

- **pprint-exit-if-list-exhausted** \Rightarrow nil

The function **pprint-exit-if-list-exhausted** tests whether or not the *list* passed to **pprint-logical-block** has been exhausted (see **pprint-pop**). If this list has been reduced to **nil**, **pprint-exit-if-list-exhausted** terminates the execution of the immediately containing **pprint-logical-block** except for the printing of the suffix. Otherwise **pprint-exit-if-list-exhausted** returns **nil**. An error message is issued if **pprint-exit-if-list-exhausted** is used anywhere other than syntactically nested within a call on **pprint-logical-block**. It is undefined what happens if **pprint-exit-if-list-exhausted** is executed outside of the dynamic extent of this **pprint-logical-block**.

- **pprint-pop** \Rightarrow item

The function **pprint-pop** pops elements one at a time off the *list* passed to **pprint-logical-block** obeying ***print-length***, ***print-circle***, and ***print-shared***. An error message is issued if it is used anywhere other than syntactically nested within a call on **pprint-logical-block**. It is undefined what happens if **pprint-pop** is executed outside of the dynamic extent of this **pprint-logical-block**.

Each time **pprint-pop** is called, it pops the next value off the *list* passed to **pprint-logical-block** and returns it. However, before doing this, it performs three tests. If the remaining list is not a list (i.e., a **cons** or **nil**), “**.**” is printed followed by the remaining list. (This makes it easier to write printing functions that are robust in the face of malformed arguments.) If ***print-length*** is **nil** and **pprint-pop** has already been called ***print-length*** times within the immediately containing logical block, “**...**” is printed. (This makes it easy to write printing functions that properly handle ***print-length***.) If ***print-circle*** (and possibly ***print-shared***) is not **nil**, and the remaining list is a circular (or shared) reference, then “**.**” is printed followed by an appropriate **#n#** marker. (This catches instances of **cdr** circularity and sharing in lists.)

If either of the three conditions above occurs, the indicated output is printed on the pretty printing stream created by the immediately containing **pprint-logical-block** and the execution of the immediately containing **pprint-logical-block** is terminated except for the printing of the suffix.

If **pprint-logical-block** is given a *list* argument of **nil**—because it is not processing a list—**pprint-pop** can still be used to obtain support for ***print-length*** (see the example function **pprint-vector** below). In this situation, the first and third tests above are disabled and **pprint-pop** always returns **nil**.

- **pprint-newline kind &optional (stream *standard-output*)** \Rightarrow nil

The function **pprint-newline** supports the functionality of `~_`. The *stream* argument

(which defaults to ***standard-output***) follows the standard conventions for stream arguments to printing functions (i.e., `nil` can be used to mean ***standard-output*** and `T` can be used to mean ***terminal-io***). If `stream` is a pretty print stream bound by **within-logical-block**, a conditional newline is sent to `stream`. Otherwise, **conditional-newline** has no effect. The value `nil` is always returned.

The `kind` argument specifies the style of conditional newline. It must be one of `:linear` for linear-style (`~:_`), `:fill` for fill-style (`~:_`), `:miser` for miser style (`~@:_`), or `:mandatory` for mandatory style (`~:@:_`).

- **pprint-indent** *relative-to n &optional (stream *standard-output*)* \Rightarrow `nil`

The function **pprint-indent** supports the functionality of `~I`. The `stream` argument (which defaults to ***standard-output***) follows the standard conventions for stream arguments to printing functions. If `stream` is a pretty print stream bound by **within-logical-block**, **pprint-indent** specifies the indentation within the innermost enclosing logical block. Otherwise, **pprint-indent** has no effect. The value `nil` is always returned.

The argument `n` specifies the amount of indentation. If `relative-to` is `:block`, this indentation is relative to the start of the enclosing block (as for `~I`). Alternatively, if `relative-to` is `:current`, the indentation is relative to the current output position in the immediately containing section (as for `~:I`). It is an error for `relative-to` to take on any other value.

- **pprint-tab** *kind colnum colinc &optional (stream *standard-output*)* \Rightarrow `nil`

The function **pprint-tab** supports the functionality of `~T`. The `stream` argument (which defaults to ***standard-output***) follows the standard conventions for stream arguments to printing functions. If `stream` is a pretty print stream bound by **within-logical-block**, tabbing is performed. Otherwise, **pprint-tab** has no effect. The value `nil` is always returned.

The arguments `colnum` and `colinc` correspond to the two numeric parameters to `~T`. The `kind` argument specifies the style of tabbing. It must be one of `:line` (tab as by `~T`), `:section` (tab as by `~:T`), `:line-relative` (tab as by `~@T`), or `:section-relative` (tab as by `~:@T`).

As an example of using the functions above, consider that **pprint-tabular** is defined as follows. Using **pprint-tab** in the definition makes it easy to communicate the parameter **tabsize** to the algorithm controlling the dynamic arrangement of output. By means of the `list` argument of **pprint-logical-block** and the macro **pprint-pop**, the definition is robust in the face of malformed lists and supports ***print-length***, ***print-level***, and ***print-circle***.

```
(defun pprint-tabular (s list &optional (colon? T) atsign?
  (declare (ignore atsign?))
  (if (null tabsize) (setq tabsize 16))
  (pprint-logical-block (s list :prefix (if colon? "(" ""))
    :suffix (if colon? ")" ""))
  (pprint-exit-if-list-exhausted)
  (loop (write (pprint-pop) :stream s)
    (pprint-exit-if-list-exhausted)
    (write-char #\space s)
    (pprint-tab :section-relative 0 tabsize s)
    (pprint-newline :fill s))))
```

The function below prints a vector using #(...) notation. A *list* argument of *nil* for **pprint-logical-block** is used, because the structure being printed is not a list. Here the functional interface to XP is appropriate, because **format** control strings do not provide any way to traverse a vector.

```
(defun pprint-vector (v *standard-output*)
  (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))))))
```

- **pprint-fill** *stream list &optional (colon? T) atsign?* ⇒ *nil*
- **pprint-linear** *stream list &optional (colon? T) atsign?* ⇒ *nil*
- **pprint-tabular** *stream list &optional (colon? T) atsign? (tabsize 16)* ⇒ *nil*

The directives **/pprint-fill/**, **/pprint-linear/**, and **/pprint-tabular/** (p. 17) are supported by the three functions above. These functions can also be called directly by the user. Each function prints parentheses around the output if and only if *colon?* (default *T*) is not *nil*. Each function ignores its *atsign?* argument and returns *nil*. Each function handles abbreviation and circularity detection correctly, and uses **write** to print *list* when given a non-list argument.

The function **pprint-linear** prints a list either all on one line, or with each element on a separate line. The function **pprint-fill** prints a list with as many elements as possible on each line. The function **pprint-tabular** is the same as **pprint-fill** except that it prints the elements so that they line up in columns. This function takes an additional argument *tabsize* (default 16) that specifies the column spacing.

supported by XP. However, in order to make sure that merely installing XP will not break any currently running code, XP converts `format` strings to functions only if they contain one or more of XP's special directives. If a `format` string does not contain any of XP's special directives, it is left as a string and the standard function `format` is used to process it.

Third, the dual approach of using XP for some `format` strings and standard `format` for others has some implications with regard to the directive `~?` and the usage `~{~}` with no body. If these forms exist in a `format` string that does not contain any of XP's special directives, then the control arguments they receive must be `format` strings rather than functions. On the other hand, if they exist in a `format` string that contains any of XP's special directives then the control arguments they receive must either be functions or `format` strings that can be successfully converted to functions by XP.

Beyond the problems above, there are several points where the documentation in [4] is not entirely clear, and about which different implementations of Common Lisp seem to disagree. XP may not be doing the right thing in these situations. In particular:

How exactly does `~@*` act in a `~{...~}` and `~@{...~}`? Is it relative to the arguments being processed by the whole loop, or relative to the arguments being process by the current step of the loop? XP assumes the former.

There is no detailed grammar given for how a directive can be specified. In particular, can a colon or atsign modifier be specified before all the parameters have been specified? XP assumes not.

What is supposed to happen to the argument list when a cycle of `~{...~}` is prematurely terminated by a `~~` directive? In particular, are the arguments that have been processed supposed to have been removed or not? In the interest of simplicity, XP assumes they should be removed.

Is `~~` supposed to operate identically when accessed via `~{~}` with no body and a `~@?` directive? XP assumes that it is. (It would be quite difficult for XP to support things any other way.)

Assumptions about the read table. It was possible to more than double the speed of XP by assuming that the characters “`a-z`”, “`A-Z`”, “`*`”, “`+`”, “`<`”, “`>`”, and “`-`” always have the same syntax as defined in the initial read table. This assumption would not be necessary if Common Lisp provided any quick way to determine what the syntax of a character is.

The delay caused by buffering. As part of its operation, XP buffers up output characters before actually printing them into the appropriate stream. The fundamental source of the efficiency of the pretty printing algorithm is that things are designed so that the buffer never has to contain more than one line width worth of output. The algorithm sends output to the underlying stream one line width at a time. The buffer is not guaranteed to be completely empty until the printing is completed. Thus there is typically a delay between the time characters are put in the buffer (e.g., by a call on some printing function in the pretty printing function for some type of object) and the time they appear in the output stream. This can be confusing if a process which is performing pretty printing is interrupted (e.g., during the debugging of a pretty printing function).

The functions `finish-output` and `force-output` can be used to force the internal

4. Pretty Printing Types of Objects

As discussed in Section 2, the pretty printing performed by XP is directed by the value of `*print-pprint-dispatch*`. The value of this variable is a *pprint dispatch table*. This table is initialized with a number of entries that specify how to pretty print all the built-in Common Lisp macros and special forms. You can tailor the pretty printer to your own needs by adding new entries into the table and/or replacing existing entries. Multiple styles of pretty printing can be supported by constructing several tables and switching between them.

Pprint dispatch tables are mappings from keys to pairs of values. The keys are type specifiers. The values are functions and numerical priorities. Basic insertion and retrieval is done based on the keys with the equality of keys being tested by `equal`. The function to use when pretty printing an object is chosen by finding the highest priority function from `*print-pprint-dispatch*` that is associated with a type specifier that matches the object.

- `copy-pprint-dispatch &optional (table *print-pprint-dispatch*) ⇒ copy`

A copy is made of `table`, which defaults to the current pprint dispatch table. If `table` is `nil`, a copy is made of the standard pprint dispatch table initially defined by XP.

- `pprint-dispatch object &optional (table *print-pprint-dispatch*)`
 \Rightarrow `function in-table-p`

This retrieves the highest priority function from a pprint table that is associated with a type specifier in the table that matches `object`. The function is chosen by finding all the type specifiers in `table` that match the object and selecting the highest priority function associated with any of these type specifiers. If there is more than one highest priority function, an arbitrary choice is made. If no type specifiers match the object, a function is returned that prints `object` with `*print-pretty*` bound to `nil`.

As a second return value, `pprint-dispatch` returns a flag that is `T` if a matching type specifier was found in `table` and `nil` if not.

The `table` argument (which defaults to `*print-pprint-dispatch*`) must be a pprint dispatch table. The `table` argument can also be `nil`, in which case retrieval is done in the initial value of `*print-pprint-dispatch*`.

When `*print-pretty*` is `T`, `(write object :stream s)` is equivalent to

```
(funcall (pprint-dispatch object) s object).
```

- `set-pprint-dispatch type-specifier function`
 $\&optional (priority 0) (table *print-pprint-dispatch*) \Rightarrow nil$

This puts an entry into a pprint dispatch table and returns `nil`. The `type-specifier` must be a valid type specifier and is the key of the entry. The first action of `set-pprint-dispatch` is to remove any pre-existing entry associated with `type-specifier`. This guarantees that there will never be two entries associated with the same type specifier in a given pprint dispatch table. Equality of type specifiers is tested by `equal`. Two values are

```
(setq *print-pretty* (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)) prints: (#.(/ 1 3) #.(- (/ 2 3)))
```

The following forms illustrate the definition of pretty printing functions for types of Lisp code. The first form specifies that lists beginning with `cond` are printed the same way as function calls except that the clauses are always printed in linear style, rather than in the format suggested by their cars. The second form specifies that lists beginning with `setq` are printed with two arguments on each line. The third form illustrates how to specify the traditional method for printing quoted objects using “,” syntax. Note the care taken to ensure that data lists that happen to begin with `quote` will be printed readably. The last 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 cond))
  (formatter "":<~W^~ ^:I~@_~@~/pprint-linear/~^ ~_~~:>"))
(set-pprint-dispatch '(cons (member setq))
  (formatter "":<~W^~ ^:I~@_~@~W^~ ^:_~W^~ ~_~~:>"))
(set-pprint-dispatch '(cons (member quote))
  #'(lambda (s list)
    (if (and (consp (cdr list)) (null (cddr list)))
        (format s ",~W" (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, as shown in the definition of `pprint-tabular` (p. 22), `pprint-linear`, `pprint-fill`, and `pprint-tabular` are all defined with optional `colon?` and `atsign?` 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)
(pprint '(0 b c d e f g h i j k))
```

With `*print-pretty*` T and line width 9 prints:

```
(0 b c d
 e f g h
 i j k)
```

Pretty printing structures. An important use of XP is to print data structures. In fact, typical Lisp interactions call for much more printing of data than printing of programs. Pretty printing can do just as much to enhance the readability of this output as it can to enhance the readability of code. As shown below, pretty printing functions for structures that have been defined without the `:type` option can be specified with reference to their types.

```
(defstruct family mom kids)
(set-pprint-dispatch 'family
 #'(lambda (s f)
     (format s "~@#~;~W and ~2I~_~:/pprint-fill/~;~:@~"
             (family-mom f) (family-kids f))))
(write (list 'principal-family
              (make-family :mom "Lucy"
                            :kids '#1=("Mark" "Bob" #1# "Bill" . "Dan"))))
```

With `*print-pretty*` T, line width 23, and `*print-lines*` 3 prints:

```
(PRINCIPAL-FAMILY
 #<"Lucy" and
 ("Mark" "Bob" ...))
```

With `*print-pretty*` T, `*print-level*` 3, and `*print-length*` 3 prints:

```
(PRINCIPAL-FAMILY #<"Lucy" and ("Mark" "Bob" # ...)))
```

With `*print-pretty*` T, `*print-escape*` nil, and `*print-circle*` T prints:

```
(PRINCIPAL-FAMILY #<Lucy and #1=(Mark Bob #1# Bill . Dan)))
```

A key thing to notice about the pretty printing function above is that without the programmer having to take any explicit action, it tolerates a malformed `kids` list and correctly follows the printer control variables `*print-lines*`, `*print-level*`, `*print-length*`, `*print-escape*` and `*print-circle*`. This should be contrasted with Common Lisp's current support for structure print self functions, where it is difficult to handle `*print-level*` and `*print-length*` correctly and impossible to handle `*print-circle*` correctly.

There is clearly a close relationship between XP's pretty printing functions for structures and the standard concept of a print function for a structure. However, there is a fundamental difference in approach. XP stores the function in a pprint dispatch table rather than directly with the structure. This makes it possible to simultaneously support several different styles of printing by maintaining several different dispatch tables and to switch rapidly between them. However, it has the disadvantage that pretty printing functions are only used when `*print-pretty*` is not `nil`. This could have the effect of forcing you to define a pretty printing function and a print function for the same structure merely to ensure that the structure is always printed the same way. To avoid this problem, XP uses the print function for a structure when no pretty printing function is available.

Efficiency. Given only what is said above, the process of determining the printing functions to be used for the various parts of an object to be printed would be horrendously inefficient, because every part of the object would have to be compared against every entry

in the pprint dispatch table. XP avoids this problem by speeding up the selection process in two ways.

A hash table is used to very rapidly compare an object against every entry with a type specifier in the pprint dispatch table that has the form (`cons (member symbol)`). A second hash table is used to rapidly compare objects with type specifiers that are the types of structures defined without the `:type` option. It is advisable for you to make as many pprint dispatching entries as possible fit into these two categories.

Another thing to note is that XP has been designed so that `pprint-dispatch` will be very efficient, since it is called on every part of every object printed. In particular, whenever trade-offs are possible, the efficiency of `set-pprint-dispatch` is sacrificed to improve the efficiency of `pprint-dispatch`. One way this is done is to convert each type-specifier passed to `set-pprint-dispatch` into an equivalent function which is then compiled. This allows `pprint-dispatch` to compare objects against type specifiers by calling compiled functions rather than relying on `typep`. However, the need to perform these transformations can make `set-pprint-dispatch` very slow. You can ensure that a special function will not have to be created or compiled by using type specifiers of the two types described in the last paragraph, or of the form (`satisfies pred`), where `pred` is already a compiled function.

5. Second Order Details

This section begins by describing some features of XP that are not part of the proposed Common Lisp standard. It then discusses some of the problems involved with supporting XP in a truly portable fashion. The discussion above deliberately glosses over these problems on the theory that there is no fundamental need for them to exist and they would not exist if XP were implemented as part of Common Lisp, rather than as a separate package. The last subsection discusses how these problems could be corrected if some time were taken to integrate XP more tightly with a given Common Lisp implementation.

Extended Features

Reprinting an abbreviated object. XP keeps track of the last pretty printing event that lead to abbreviation due to `*print-level*`, `*print-length*`, or `*print-lines*`. A hook, is provided for obtaining this information. Using this hook, mechanisms can easily be implemented for reprinting abbreviated objects in full.

- **`*last-abbreviated-printing*`**

This variable records the last printing event where abbreviation occurred. Funcalling its value (e.g., after turning off abbreviation) causes the printing to happen again.

- **`pp &optional object (stream *standard-output*) ⇒`**

This function pretty prints *object* on *stream* with `*print-circle*` bound to `T` and with `*print-lines*`, `*print-length*`, `*print-level*`, `*print-shared*`, and `*print-shared*` bound to `nil`. It is convenient to use occasionally when operating in an environment where one or more of these abbreviation variables have non-null values. If *object* is omitted, it defaults to the object saved in `*last-abbreviated-printing*`. No values are returned.

In Symbolics Common Lisp, `xp::install` sets up the key sequence `<function>` `<resume>` so that it triggers the reprinting in full of the last abbreviated printing (via `pp`). This turns out to be very convenient. A similar mechanism might be useful in other environments as well.

- **`*default-right-margin* default value 70`**

When `*default-right-margin*` is `nil`, XP queries the output stream, to determine the width available for printing. However, there are some situations (e.g., printing to a string) where the stream has no inherent width limits. In this situation, the right margin is set to the value of `*default-right-margin*`, which must be an integer.

- **`#"... " ⇒ function`**

The `#` reader macro character syntax `#..."` provides a compact syntax for the macro `formatter` as shown below. For this syntax to be enabled, you must call `xp::install` with a non-null `:macro` argument.

```
#..." ≡ (formatter ...)
```

In addition, `#"..."` supports a special feature not supported by `formatter`. If `"/name/"` appears in `#"..."`, and no explicit package prefix is specified, the package defaults to the value of `*package*` at the moment the `#"..."` was read, rather than to the user package. This is in line with the way the packages for symbols default in general. For instance,

```
foo:(f #'^A~/f/) ≡ (foo:f (formatter "^A~/foo:f/"))
```

while

```
foo:(f (formatter "^A~/f/")) ≡ (foo:f (formatter "^A~/user:f/"))
```

Ugly Facts About Portability

XP is implemented in fully portable Common Lisp. However, a number of compromises had to be made for this to be true. This section discusses these problems in detail and explains how they have been dealt with in Symbolics Common Lisp [9]. It is hoped that they can be dealt with as easily in other implementations of Common Lisp.

Insufficient integration with non-pretty printing. XP never comes into play unless `*print-pretty*` is not nil, `(formatter "...")` is encountered, or a `format` string is evaluated that contains one of xp's special `format` directives. This is done as a matter of safety and so that XP will operate purely as an add-on system. However, it has drawbacks. For example, the variable `*print-right-margin*` only has an effect when XP is in operation. Similarly, `*last-abbreviated-printing*` only gets set when XP is in operation. If XP were combined into a Common Lisp implementation, it would be natural to combine it directly into the standard output routines, and support variables like `*print-right-margin*` and `*last-abbreviated-printing*` all of the time.

Getting XP to take effect. By far the biggest problem is that Common Lisp has no standard mechanism for allowing a new pretty printer to be specified. The function `xp::install` uses shadowing to redefine the standard Common Lisp printing functions. However, this is of somewhat limited utility for several reasons.

First, shadowing fundamental functions like `print` and `defstruct` is a dangerous practice. In particular, while it can work when it is done by one subsystem, it is almost never going to work if two subsystems try to do it.

Second, shadowing only effects programs that are read into the package where XP is installed after XP has been installed. Among other things, this means that it will not change the printing that is initiated by the Lisp system itself. For example, it will not change the printing done by the top level read-eval-print loop. You can change this easily enough, but that leaves a host of other places where the system initiates output, such as various things printed by the debugger.

You could try to install XP more firmly by altering the function cells of the standard printing functions. However, this is an exceptionally dangerous thing to do and is quite likely to break the system. (To start with, it will break XP.)

In any event, clobbering these function cells would not fix the problem, because many Lisp implementations do output by calling primitive output functions that are not part of

the standard set of Common Lisp output functions. As a result, clobbering the standard functions still would not fix all output.

A better answer is to have a hook in the Lisp system that is prepared to accept a new pretty printer. Symbolics Common Lisp has such a hook in the form of the variable `scl:*print-pretty-printer*`. The symbolics Common Lisp version of the function `xp::install` sets this variable to a value that causes XP to be used for all pretty printing.

- **`xp::*allow-errors*`** default value `T`

An interesting aspect of the function installed on `scl:*print-pretty-printer*` is that it traps any errors that occur when printing is done. This is very useful when such errors are happening while you are trying to debug something else. However, trapping such errors can be very annoying when it is a `set-pprint-dispatch` function or something like that that is being debugged. You can turn off the error trapping feature by setting the variable `xp::*allow-errors*` to `T`.

Obtaining information from output streams. To operate as intended, XP needs to be able to get two pieces of information from an output stream before starting to print into it. This information is obtained by calling the following two functions.

- **`xp::output-width &optional (output-stream *standard-output*)` ⇒ `width`**

Returns the maximum number of characters that can be printed on a single line without causing truncation or wraparound when printing to `output-stream`, or `nil` if this cannot be computed.

- **`xp::output-position &optional (output-stream *standard-output*)` ⇒ `position`**

Returns the number of characters printed so far on the current output line in `output-stream`, or `nil` if this cannot be computed.

Unfortunately, although every implementation of Common Lisp probably supports internal functions providing this information, there are currently no standard Common Lisp functions yielding this information. XP contains appropriate definitions of the functions above for several different implementations of Common Lisp; however, in other implementations it is reduced to using default (useless) definitions of these two functions that always return `nil`. If you are operating in one of these other implementations (you can tell by looking at the beginning of the XP file) you should provide better implementations for these functions.

Imperfect integration with structures. To operate as intended, XP needs to be able to determine which types are structure types. This is done by calling the following function.

- **`xp::structure-type-p type` ⇒ `boolean`**

Returns `non-nil` if and only if `type` is a structure type defined by `defstruct` without the `:type` option, and `nil` otherwise.

Unfortunately, although every implementation of Common Lisp probably supports an internal function providing this information, there is currently no standard Common Lisp

function yielding this information. XP contains appropriate definitions of the functions above for several different implementations of Common Lisp; however, in other implementations it is reduced to pessimistically assuming that the only structures are ones defined using `xp::defstruct` (which is used to shadow `lisp:defstruct` if `xp::install` is called with `:shadow T`). If you are operating in one of these other implementations (you can tell by looking at the beginning of the XP file) you should provide an implementation for `xp::structure-type-p`.

Another potential problem is that XP assumes that if a structure is defined using XP's shadowed version of `defstruct`, then the structure's print function (if any) is defined using the XP's shadowed versions of the various printing functions. As a result, XP does not hesitate to call such a print function with one of its pretty printing streams as an argument. Since it is possible for the assumption to be false, this can lead to problems.

Limitations on the definition of new type specifiers. Due to the extreme restrictions Common Lisp places on the ways complex type specifiers can be constructed, there is no implementation independent way to support the extended definition of the type specifier `cons` as a first class type specifier, even though it does not violate the 'spirit' of what can and cannot be a type specifier. As a consequence of this limitation, the extend form of `cons` can only be used in conjunction with `set-pprint-dispatch`. This could easily be remedied if XP were incorporated directly into Common Lisp.

Imperfect integration with `format`. XP supports 99% of the functionality of `format`, but not all of it. In particular, XP takes pains to fully support `format` as described in [4]. However, there is one place where XP falls short of this goal.

As discussed above, the standard `format` directive `~<...~>` is more or less incompatible with `~<...~:>` and the other pretty printing directives. However, it is permissible to have a garden variety instance of `~<...~>` nested in a `format` string that also contains some pretty printing directives. In this situation, XP uses the standard function `format` to process the part of the `format` string containing the `~<...~>`. Unfortunately, this only works when it can be determined exactly how many arguments will be used by the `~<...~>`. As a result, XP is forced to require that `~<...~>` cannot contain complex directives like `~@{...~}`, `~~`, and `~*` or anything similar. This problem could be straightforwardly fixed if XP duplicated all of the code in the standard function `format` that supports `~<...~>` instead of merely using the standard function `format`.

Another area of difficulty concerns the fact that XP is oriented around supporting formating functions (e.g., created by `(formatter "...")`) rather than `format` control strings. Nevertheless, in the interest of upward compatibility, XP allows `format` strings to be used. However, there are three complications with this.

First, to avoid having to implement an interpreter for `format` strings as well as a compiler for `(formatter "...")`, XP converts each `format` string that contains any of XP's special directives into a function the first time it is encountered. This works well as long as `format` strings are not modified by side-effect. The caching of converted `format` control strings can be turned off by setting the variable `xp::*format-string-cache*` to `nil`.

Second, some implementations of Common Lisp support `format` directives beyond the ones defined in [4] or support additional features of the standard `format` directives. No attempt is made to support this functionality in conjunction with the special directives

supported by XP. However, in order to make sure that merely installing XP will not break any currently running code, XP converts `format` strings to functions only if they contain one or more of XP's special directives. If a `format` string does not contain any of XP's special directives, it is left as a string and the standard function `format` is used to process it.

Third, the dual approach of using XP for some `format` strings and standard `format` for others has some implications with regard to the directive `~?` and the usage `~{~}` with no body. If these forms exist in a `format` string that does not contain any of XP's special directives, then the control arguments they receive must be `format` strings rather than functions. On the other hand, if they exist in a `format` string that contains any of XP's special directives then the control arguments they receive must either be functions or `format` strings that can be successfully converted to functions by XP.

Beyond the problems above, there are several points where the documentation in [4] is not entirely clear, and about which different implementations of Common Lisp seem to disagree. XP may not be doing the right thing in these situations. In particular:

How exactly does `~@*` act in a `~{...~}` and `~@{...~}`? Is it relative to the arguments being processed by the whole loop, or relative to the arguments being process by the current step of the loop? XP assumes the former.

There is no detailed grammar given for how a directive can be specified. In particular, can a colon or atsign modifier be specified before all the parameters have been specified? XP assumes not.

What is supposed to happen to the argument list when a cycle of `~{...~}` is prematurely terminated by a `~~` directive? In particular, are the arguments that have been processed supposed to have been removed or not? In the interest of simplicity, XP assumes they should be removed.

Is `~~` supposed to operate identically when accessed via `~{~}` with no body and a `~@?` directive? XP assumes that it is. (It would be quite difficult for XP to support things any other way.)

Assumptions about the read table. It was possible to more than double the speed of XP by assuming that the characters “`a-z`”, “`A-Z`”, “`*`”, “`+`”, “`<`”, “`>`”, and “`-`” always have the same syntax as defined in the initial read table. This assumption would not be necessary if Common Lisp provided any quick way to determine what the syntax of a character is.

The delay caused by buffering. As part of its operation, XP buffers up output characters before actually printing them into the appropriate stream. The fundamental source of the efficiency of the pretty printing algorithm is that things are designed so that the buffer never has to contain more than one line width worth of output. The algorithm sends output to the underlying stream one line width at a time. The buffer is not guaranteed to be completely empty until the printing is completed. Thus there is typically a delay between the time characters are put in the buffer (e.g., by a call on some printing function in the pretty printing function for some type of object) and the time they appear in the output stream. This can be confusing if a process which is performing pretty printing is interrupted (e.g., during the debugging of a pretty printing function).

The functions `finish-output` and `force-output` can be used to force the internal

buffer to empty out. However, to maintain internal consistency in the pretty printing algorithm, all of the logical blocks that have been started but not yet completed are printed as if they are not able to fit on a single line. As a result, the output may not look the same as it would if the buffer were not prematurely forced to empty out.

Taking full advantage of information about formatting special forms. Symbolics Common Lisp contains a large number of special forms that have to be pretty printed in special ways in addition to the standard Common Lisp special forms. In Symbolics Common Lisp, XP takes advantage of the fact that the ZWIE editor maintains information about these forms in order to determine how to pretty print them. A similar sharing of information between XP and the Lisp editor might be useful in other environments as well.

No support for font variations. The pretty printing algorithm depends on extensive calculations about how much space strings of characters will take up when displayed. These calculations are greatly simplified by assuming that every character will have the same fixed width when displayed. Only newlines are treated specially.

It should be noted that (except for `~T`) the standard `format` directives all make the same simplifying assumption. However, this assumption can lead to problems in some situations. For example, it is inadvisable to use literal tab characters when pretty printing and the output produced by XP looks quite strange when it is displayed using a variable width font.

The above notwithstanding, the fundamental algorithms used by XP could be extended to handle characters of variable width and characters whose width depends on the position where they are displayed. In addition, the interface has been designed as much as possible to be independent of this issue.

The only user-visible things that refer to actual lengths are the variables `*print-right-margin*`, `*default-right-margin*`, `*print-miser-width*`, and the numeric arguments to `~T`, `~I`, and `~/pprint-tabular/` and their functional counterparts. All of these measurements must be in the same units, but it does not matter a great deal what the units are. A good choice would be something like the length of an “m” in the current font. This will work out right for fixed width fonts and pretty well for variable width ones. Programmers should be advised to avoid explicit lengths—i.e., they should rely on streams knowing how wide they are and use `~0:I` whenever possible to indicate indentations.

Doing Things Right

XP is the kind of program that cannot really be supported in a totally portable way in Common Lisp. This is true both due to the various problems outlined above and because there are a number of things where portability has only been achieved at the significant sacrifice of efficiency. The right thing to do when incorporating XP into a Common Lisp is not to merely load the system and use it, but rather to totally integrate it with the way printing is done.

Places where XP needs to be more tightly integrated with the primitive printing facilities. There are a number of places where XP falls back on using the standard printing facilities. The standard function `write` is used to print objects for

which there is no special printing function in `*print-pprint-dispatch*`. The standard function `format` is used to support complex `format` directives like `~R`, `~C`, and `~F`. In both cases, this is done by having the standard functions output into a string and then copying the string into XP's internal buffer.

This is effective, but quite slow. As a result, printing with XP is noticeably slower than printing with `*print-pretty* nil`. This is unfortunate, because as demonstrated by PP [7], the algorithms underlying XP are sufficiently efficient that it is possible for pretty printing to be virtually as fast as non-pretty printing. The only thing that is missing is proper integration with the printing subprimitives.

To a certain extent, superior integration could have been achieved by duplicating more of the basic printing code as part of XP. However, it would not be possible to achieve perfect integration in a portable way, because Common Lisp does not provide any way to get information out of the read table. As a result, `write` must be used to print symbols. (As discussed above, XP gets around this problem to some extent by making a few conservative assumptions about the read table.)

The right thing to do when incorporating XP into a Common Lisp is to modify XP so that it directly calls the appropriate printing sub-primitives and modify the sub-primitives so that they put their output directly into XP's internal buffer.

Places where the primitive printing facilities need to be more tightly integrated with XP. To get XP to really take over for all pretty printing, it needs to be installed in such a way that it is always used. The right way to do this is to insert a call to it deep in the standard printing code at the point where the variable `*print-pretty*` is tested.

In addition, all of the functions that make use of `format` strings (e.g., `error`) should be extended so that they can make use of `(formatter "...")` and the special pretty printing functions.

Beyond this, there is a more subtle problem. Internally, XP operates in two stages. The first stage supports dispatching through `*print-pprint-dispatch*` along with various kinds of abbreviation. This dispatching is accessed via the directive `~W` and the function `write`. The second stage performs the actual dynamic formatting decisions.

The second stage essentially operates as a special kind of output stream. This stream receives output characters and commands related to logical blocks and conditional new-lines. After deciding where to insert line breaks, the output is sent on to the ordinary stream that is the eventual destination of the output. This organization is largely hidden. However, it becomes apparent in one key situation.

When writing special printing functions (i.e., to be used with `set-pprint-dispatch`, `defstruct`, or `~/.../`) it is permissible to use any kind of printing function. However, these functions are called with special XP streams as arguments rather than ordinary streams. (This is essential, because XP must be able to catch all output before it gets to the real output stream.) As a result, all of the standard printing functions (e.g., `print`, `terpri`, `force-output`) have to be modified so that they will operate correctly when passed a special XP stream.

Alternatively, the fundamental concept of what an output stream is can be altered so that every stream is capable of supporting the operations of the second stage of XP. This approach was taken by PP, and worked very well. A particular advantage

of this is that it allows proper integration of XP with functions like `with-open-file` and `with-output-to-string`. Unfortunately, it is impossible to create a new kind of stream in a portable manner, because Common Lisp does not provide any appropriate primitives.

Using XP to the full. Because the capabilities of XP go way beyond typical pretty printers, XP can be used in many ways that typical pretty printers cannot. As a result. It is useful to extend a Common Lisp so that it takes better advantage of pretty printing. To start with, since (when properly integrated) XP is just as fast as a non-pretty printer their is no reason not to have the default value of `*print-pretty*` be T.

Beyond this, many kinds of output done by the system itself should be upgraded to take advantage of XP. As an example, in Symbolics Common Lisp, the `trace` facilities can be used to print out information about the arguments a function is called with whenever it is called. This output is produced using standard `format` control strings and always prints all the arguments on one line. If the arguments are large, this output ends up being more or less unreadable. The Symbolics version of `xp::install` replaces the trace printer with a new function that takes full advantage of xp. There are dozens of other places where such changes could profitably be made.

Historical Note

The original **#print** system was written in MacLisp in 1977. The primary motivation behind **#print** was producing a pretty printer significantly faster than the Goldstein pretty printer [1] then in use. By means of the same basic algorithms that are still in use in XP, **#print** succeeded in being almost as fast as ordinary printing. In addition to using fundamentally the same algorithms as XP, **#print** followed the same basic approach of having pretty printing control strings for specifying how to control the dynamic layout of output and mechanisms for associating pretty printing functions with types of objects. However, the interface was markedly different in two respects.

First, the pretty printing control strings used by **#print** were developed before **format** came into wide use. Although fundamentally similar to **format** control strings, they looked very different, because they treated unmarked characters as directives instead of literal characters to be printed out. Literal output had to be specified by enclosing it in apostrophes. The pretty printing control strings were also described in a confusing way that exposed unnecessarily much of the underlying algorithm. Second, the mechanisms used by **#print** to associate pretty printing functions with types of objects were significantly more cumbersome and less powerful than those supported by XP.

#print was released for general use in the MIT AI Laboratory in January of 1978. However, probably because satisfactory documentation was never produced, **#print** was not extensively used by anyone other than the author.

In early 1980, **#print** was cleaned up and re-released under the name **Gprint**. The primary change was that, **Gprint** extended the power of (and further complicated) the mechanisms for associating pretty printing functions with types of objects. In late 1981, full documentation was prepared [5] and **Gprint** began to reach a wide audience.

In the spring and summer of 1982, **Gprint** was converted to run on the Symbolics Lisp Machine. After a delay of a year or so, **Gprint** was adopted as the standard pretty printer on the Symbolics Lisp Machine, in which role it is still being used today. However, Symbolics decided not to publicize the interface that can be used to define new ways of pretty printing things. In the summer of 1983, DEC converted **Gprint** into their Common Lisp and adopted it and its interface as official parts of their Common Lisp.

In 1984, **Gprint** was totally rewritten in Symbolics Lisp Machine Lisp and re-emerged as **PP** [7]. The key advance was that **PP** unified the concepts of **format** control strings and pretty printing control strings, recasting everything in **format**'s syntax. From the point of view of people who understand **format**, this simplified things tremendously. **PP** also somewhat simplified the mechanisms for associating pretty printing functions with types of objects by eliminating the least used features. In the fall of 1985, DEC upgraded their Common Lisp to include **PP** and its interface instead of **Gprint**.

In 1988, **PP** was totally rewritten in completely portable Common Lisp and re-emerged as **XP** [8]. **XP**'s major contribution is that, by taking an entirely different approach, it greatly simplifies the mechanisms for associating pretty printing functions with types of objects and makes them even more powerful than the mechanisms supported by **Gprint**. Since September 1988, **XP** has been in experimental use as the pretty printer in CMU Common Lisp.

Starting in the winter of 1988-89, discussions began with the Common Lisp stan-

dardization committee (X3J13) about the possibility of incorporating XP as part of their proposed standard. A number of relatively minor abjections were raised by various members of the committee that necessitated changes in XP. However, in the summer of 1989, XP was accepted as part of the new standard.

The relevance of XP is not restricted to Common Lisp. Many of the ideas discussed in this paper have a wider area of applicability. For instance, the mechanisms for allowing the user to exercise control over the dynamic arrangement of output could be incorporated into almost any programming language (e.g., into the formatted output statements of Fortran or PL/I). Similarly, the mechanisms for linking special pretty printing functions with various types of objects could be introduced into almost any programming environment, given the same kind of type information from the compiler that a good debugger requires.

Bibliography

- [1] Goldstein I., “Pretty Printing, Converting List to Linear Structure”, MIT/AIM-279, February 1973.
- [2] Hearn A.C. and Norman A.C., *A One-Pass Pretty Printer*, Report UUCS-79-112, Univ of Utah, Salt Lake City Utah, 1979.
- [3] Oppen D., “Prettyprinting”, *ACM TOPLAS*, 2(4):465–483, October 1980.
- [4] Steele G.L.Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.
- [5] Waters R.C., *Gprint: A Lisp Pretty Printer Providing Extensive User Format-Control Mechanisms*, MIT/AIM-611, October 1981.
(Revised version MIT/AIM-611a, September 1982.)
- [6] Waters R.C., “User Format Control in a Lisp Prettyprinter”, *ACM TOPLAS*, 5(4):513–531, October 1983.
- [7] Waters R.C., *PP: A Lisp Pretty Printing System*, MIT/AIM-816, December 1984.
- [8] Waters R.C., *XP: A Common Lisp Pretty Printing System*, MIT/AIM-1102, March 1989.
- [9] *Lisp Machine Documentation for Genera 7.0*, Symbolics, Cambridge MA, 1986.

Functional Summary and Index

```

copy-pprint-dispatch &optional (table *print-pprint-dispatch*) ⇒ copy
  p. 24 Copies a pprint dispatch table.

*default-right-margin* default value 70
  p. 29 Default line width to use when pretty printing.

formatter control-string ⇒ function
  p. 5 Macro that transforms a format control string into a function.

xp::install &key (:pkg *package*) (:macro nil) (:shadow T) (:remove nil) ⇒ T
  p. 2 Makes XP ready for use.

*last-abbreviated-printing*
  p. 29 Variable recording last printing event that was abbreviated.

pp &optional object (stream *standard-output*) ⇒
  p. 29 Pretty prints an object without abbreviation.

pprint-fill stream list &optional (colon? T) atsign? ⇒ nil
  p. 23 Function underlying ~/pprint-fill/.

pprint-dispatch object &optional table ⇒ function in-table-p
  p. 24 Retrieves the pprint function for object from a pprint dispatch table.

pprint-exit-if-list-exhausted ⇒ nil
  p. 21 Supports length and circularity abbreviation.

pprint-indent relative-to n &optional (stream *standard-output*) ⇒ nil
  p. 22 Functional interface to ^I.

pprint-linear stream list &optional (colon? T) atsign? ⇒ nil
  p. 23 Function underlying ~/pprint-linear/.

pprint-logical-block (s l &key :prefix :per-line-prefix :suffix) &body b ⇒ nil
  p. 20 Functional interface to ^<...^:>.

pprint-newline kind &optional (stream *standard-output*) ⇒ nil
  p. 21 Functional interface to ^_.

pprint-pop ⇒ item
  p. 21 Supports length and circularity abbreviation.

pprint-tab kind colnum colinc &optional (stream *standard-output*) ⇒ nil
  p. 22 Functional interface to ^T.

pprint-tabular stream list &optional (colon? T) atsign? (tabsize 16) ⇒ nil
  p. 23 Function underlying ~/pprint-tabular/.

*print-lines* default value nil
  p. 4 Variable limiting the total number of lines pretty printed.

*print-miser-width* default value 40
  p. 3 Variable specifying when pretty printing should switch to space saving mode.

*print-pprint-dispatch* default value causes standard pretty printing
  p. 3 Variable containing the current pprint dispatch table controlling pretty printing.

*print-right-margin* default value nil
  p. 3 Variable specifying the line width to use when pretty printing.

set-pprint-dispatch type function &optional priority table ⇒ nil
  p. 24 Sets a pprint dispatch table entry.

```

`#"..."` (p. 29) Functional `format` control string.

The directive `~W` (write object p. 5) uses the function `write` to output the corresponding `format` argument without forcing the setting of any output control variables.

`~W` (p. 5) Prints an argument following all output control variables.

`~:W` (p. 5) Forces pretty printing.

`~@W` (p. 5) Suppresses length and depth abbreviation.

There are three special directives for printing lists. Each of them prints parentheses around the output when used with the colon modifier.

`~/pprint-fill/` (p. 17) Prints as many elements as possible on each line.

`~/pprint-linear/` (p. 17) Prints elements all on one line or one to a line.

`~c/pprint-tabular/` (p. 18) Prints elements in a table with column spacing `c`.

The directive `~<prefix~;body~;suffix~:>` (logical block, p. 8) iterates over a list argument using `body` to print the elements of the list in a logical block. The `prefix` and `suffix` are printed before and after the block respectively.

`~<...~:>` (p. 8) Denotes a logical block and descends into a list argument.

`~@<...~:>` (p. 8) Operates on all the remaining arguments.

`~:<body~:>` (p. 10) Prefix and suffix default to "(" and ")" respectively.

`~<body~:>` (p. 10) Body printed to fill the line width.

`~<prefix~@;...~:>` (p. 10) Prefix printed on each line.

The indentation in a logical block is initially set to the column position of the first character in the block. The directive `~I` (set indentation, p. 14) is used to alter the indentation within a logical block. If omitted, the parameter defaults to zero. When a logical block is printed in miser style, all instances of `~I` are ignored.

`~nI` (p. 14) Indentation set to position of first character in block plus `n`.

`~n:I` (p. 14) Indentation set to position of directive plus `n`.

The directive `~_` (conditional newline, p. 11) specifies a place where a newline can be inserted in a logical block. For a discussion of line breaks inserted by other means than `~_`, see page 13.

`~_` (p. 11) Linear-style conditional newline.

`~:_` (p. 12) Fill-style conditional newline.

`~@_` (p. 11) Miser-style conditional newline.

`~:@_` (p. 13) Mandatory-style conditional newline.

The directive `~T` has augmented capabilities.

`~:T` (p. 15) Tab relative to containing section.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI Memo 1102	2. GOVT ACCESSION NO. <i>AD-A208215</i>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) XP: A Common Lisp Pretty Printing System		5. TYPE OF REPORT & PERIOD COVERED memorandum
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard C. Waters		8. CONTRACT OR GRANT NUMBER(s) N00014-88-K-0487
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE March 1989
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		13. NUMBER OF PAGES 40
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution is unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) LISP formatted output pretty printing abbreviated output		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) XP provides efficient and flexible support for pretty printing in Common Lisp. Its single greatest advantage is that it allows the full benefits of pretty printing to be obtained when printing data structures, as well as when printing program code. XP is efficient, because it is based on a linear time algorithm that uses only a small fixed amount of storage. XP is flexible, because users can control the		

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives**, using funds from the **Advanced Research Projects Agency of the United states Government** under Grant: **MDA972-92-J1029**.

The scanning agent for this project was the **Document Services** department of the **M.I.T. Libraries**. Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences**.

