A New Language-Independent Prettyprinting Algorithm*

William W. Pugh†
Steven J. Sinofsky

TR 87-808

January 1987

Department of Computer Science
Cornell University
Ithaca, NY 14853

* Publication of the report was partially supported by ONR and NSF under grant DCR8514862.

† AT&T Bell Laboratories Scholar.

A New Language-Independent Prettyprinting Algorithm*

William W. Pugh†
Steven J. Sinofsky

Computer Science Department
Cornell University
Upson Hall
Ithaca, New York 14853

January, 1987

Abstract

An algorithm for prettyprinting using word wrapping is described that is independent of the language being formatted and is substantially simpler than other published algorithms. The algorithm makes use of a simple model with a small set of primitives to direct the prettyprinting of text. For an input string of length n, and an output device m characters wide the algorithm runs in $O(n)$ time and requires $O(m)$ space. The algorithm can be restarted from an intermediate point and is therefore well suited for incremental prettyprinting of text. This algorithm is now being used in the Cornell Synthesizer Generator [2]. The algorithm is compared with and contrasted to the previously published algorithm by Oppen [1].

---

*Publication of the report was partially supported by ONR and NSF under grant DCR 8514862.

†AT&T Bell Laboratories Scholar.

A New Language-Independent Prettyprinting Algorithm

I. Introduction

A prettyprinter takes as input a stream of characters and prints them in an aesthetically pleasing and functional way by displaying the text with appropriate line breaks and indentation. As an example consider a Pascal statement such as:

```
if error then return(errorFlag);
```

which we might prefer to have appear on one line if the output device is wide enough. If the statement could not fit on a single line we might prefer:

```
if error then
    return(errorFlag);
```

but we would certainly not like to see:

```
if error then return(
    errorFlag);
```

We shall describe a language-independent algorithm that permits the specification of preferred locations within a string where word wrapping and/or indenting should occur. In order that the algorithm be language independent, the algorithm is oblivious to the textual structure of the strings it prints; prettyprinting commands embedded in the string indicate which segments of text form a logical grouping and where line breaks may be inserted.

One of the more recent and novel uses of prettyprinters has been in structure editors for block-structured languages such as Pascal and PL/1 [2, 3, 4, 5]. Traditionally, these environments have provided one or more schemes for the fixed (that is, independent of the length of the string and width of the output device) unparsing of text. If a statement is too long to fit on a single line, the user is required to manually select an unparsing scheme that will break the text onto multiple lines appropriately.

While the subject of prettyprinting block-structured languages is relatively well developed, prettyprinting can also be useful in clarifying other textual objects. The Cornell Synthesizer Generator [2] is now being used to develop less conventional editors such as a formal logic proof editor and an editor for program verification. Such editors generate logical goals and verification conditions that can be hundreds

characters long; to be understandable these logical formulae must be displayed on several lines in a way that reflects their structure.

Comparison with Oppen's algorithm. Our algorithm is similar to that of Derek Oppen [1], but is somewhat simpler and is readily adapted to incremental prettyprinting. Oppen's analysis focuses on defining groups of text that should not be broken and initially assumes text can be split at any blank. In our algorithm we allow the user to restrict the locations at which the line may be split, giving the user greater control over the prettyprinting process. For example, the user can specify that in an expression a binary operator should always appear on the same line as the second operand and that it is not permissible to split the line immediately after the operator. Oppen defines extensions to achieve roughly the same functionality as our algorithm; however, this complicates his algorithm and the changes required in his implementation are not explained. We feel that our approach leads to a substantially simpler algorithm that will make it easier for others to adapt our algorithm for their own uses.

Additionally, in our algorithm we can restart the prettyprinting in mid-stream by restoring only a constant amount of information, while it was not clear how to restart Oppen's algorithm. We need this property because the algorithm is used in the Cornell Synthesizer Generator [2]. For efficiency reasons, editors created by the Synthesizer Generator do not unparse the entire edited object to update the screen, but rather restart the unparsing from the line above the top of the screen, stopping once the bottom of the screen is reached. We handle indentation differently than Oppen. Oppen lines up any breaks in a group with the column where the group started and we provide commands to increase to decrease the indentation level. While the merits of either scheme could be argued, we found that ours worked better for incremental prettyprinting.

We achieve the same time bound, $O(\textit{length of input})$, and space bound, $O(\textit{width of output})$, as Oppen's algorithm.

II. Using the Prettyprinter

The algorithm presented transfers an input string to an output device one line at a time with no line wider than the width of the output device. Each character of the input string is either a printable character or a format control character indicating a prettyprinter action to be taken. In this paper, format control characters will be

printed in *italics*; in practice, we use normal ASCII characters preceded by a special escape character such as %. The control characters follow in the table below, and are explained in the remainder of this section:

{ and *}*	- grouping symbols
t and *b*	- indent and outdent following text
n	- unconditional line break
o	- optional conditional line break (called *inconsistent* in [1])
c	- connected conditional line break (called *consistent* in [1])

Briefly, *grouping* symbols are used to group segments of text that should be kept on a single line if possible. The *indent* and *outdent* commands change the left margin for all following lines. An *unconditional* line break indicates the the following text should be placed on a new line. A *conditional* line break indicates a location where the text can be broken onto two lines if it will not fit on a single line. A *connected* conditional line break (referred to as a connected break) has the special property that if a group containing connected breaks will not fit on a single line, the text will be split at *all* of the connected breaks at that level. Connected breaks are used to separate items of a list that should each be placed on a separate line if they cannot be all placed on a single line. Conditional breaks that are not connected are referred to as *optional* line breaks.

Grouping Symbols. The prettyprinter tries to divide the input stream into separate lines such that a minimum number of groupings are broken between lines. A grouping is a segment of text surrounded by *{* and *}*, the open grouping symbol and close grouping symbol, respectively. We assume these grouping symbols to be balanced. As an example, consider the prettyprinting of the Pascal **if** statement. We might prefer to break the line after the **then** rather than breaking the line in the middle of the conditional expression. Given such a statement, which we desire to output on a device with lines that are 25 characters wide, we might have the following output:

```
if x < 0 then x := -x
```

or with an output device 15 characters wide we might prefer:

```
if x < 0 then
   x := -x
```

To obtain the desired output in either case using grouping symbols and an optional line break, the input to the prettyprinter might look like:

*t{***if** *t{*x *o*< 0*}b* **then** *ot{*x *o*:= -x*}b}b*

or more generally

t{**if** *t*{<condition>}*b* **then** *ot*{<statement>}*b*}*b*

The entire statement is surrounded by grouping symbols, because presumably the statement occurs within a list of other statements. The <condition> and <statement> are also enclosed in grouping symbols, because we would prefer to break the line at the optional break after the **then** token rather than at any optional breaks that might occur inside the <condition> or <statement>.

Indent and Outdent. The indent and outdent commands, represented by *t* and *b* respectively, are used to change the left margin, which is the column where the first character of the next line will appear. Typically an indent command is paired with an open grouping command and an outdent command is paired with a close grouping command; this allows a user to visually group lines according to their indentation. The current left margin at the time a conditional break is seen determines the indentation of the text after the break if the prettyprinter takes that break.

We considered defining grouping symbols to include the effects of indent and outdent commands; however, this might not be desired in all situations. Since prettyprinting strings are created and parsed without being seen by the user, we decided that flexibility was more important than readability of internal prettyprinting strings.

Unconditional Line Breaks. An unconditional line break, represented as *n*, always causes the following text to begin on a new line. For example, we might wish a **begin** and **end** to always appear on distinct lines, in which case the keywords will be preceded by an unconditional line break. One also might desire that an **else** clause always appear on a line distinct from the **if** and the **then**.

Optional Line Breaks. The above example makes use of one type of conditional line break, the optional line break, represented as *o*. This indicates where the prettyprinting algorithm can insert a line break if necessary. The prettyprinter chooses the optional line break that breaks the fewest existing groups. Any optional breaks inside of the <condition> in the above example will not be taken unless the <condition> itself will not fit on one line, since there is an optional break at an outer grouping level. If there are two optional breaks within the same group, the rightmost one is taken.

Connected Line Breaks. The addition of a second type of conditional line break, the connected break, represented as *c*, further enhances the prettyprinter. A connected line break is connected with all of the other breaks at the same grouping level in the sense that either they are all taken or none of them are taken. Such a break would typically be used to separate items of a list where the entire list should be put on a single line, if possible; otherwise each item should be on a separate line. An example would be the parameter list and accompanying declarations of a Pascal procedure:

```
procedure f(a:integer; b: boolean; c: real);
```

which we might prefer to print on a narrow output device as

```
procedure f(a: integer;
            b: boolean;
            c: real;)
```

rather than

```
procedure f(a:integer;
   b:boolean; c: real);
```

Interactions. An unconditional break inside a group will cause all connected breaks at that level to be taken. If connected and optional breaks exist at the same grouping level, the system will attempt to take just the connected breaks. We make this choice because taking the connected breaks may mean that we do not have to break the text at the optional break. If we break the text at the optional break the group containing the connected breaks will not be on a single line and the connected breaks will need to be taken anyway.

Problems. As with all prettyprinters we are aware of, we have been unable to devise an elegant technique to use when there are no conditional line breaks within a string that is too long to display on one line. This can occur if the display is very narrow, the left margin has been indented so that it is close to the right margin, long identifiers or strings occur in the object to be prettyprinted, or if not enough conditional line breaks have been specified. We are seeking user advice on which of several inelegant schemes would be preferred.

III. The Prettyprinting Algorithm

Code for the algorithm is presented in the appendix in pseudo-Pascal. Briefly, the algorithm scans the prettyprinting string and places each character in a buffer. If a required line break is seen the contents of the buffer are sent to the display. If the

text now in the buffer will not fit on a single line, the algorithm chooses a conditional line break that occurs within the text currently in the buffer and sends to the display the characters that occurred before the line break chosen. The algorithm is described in detail below to assist those who might wish to adapt it.

Data Structures. The algorithm makes use of a single abstract data type, the double ended queue or DEQUEUE [6, 7], to handle all of the information required for prettyprinting. This structure is an unbounded list that allows insertion at either the right end or the left end, through the operations of right_enqueue and left_enqueue respectively. Elements can be removed from either the right or left end with the operations right_dequeue and left_dequeue. Several other operations are also defined on the structure: dq_right and dq_left return, without removing, the rightmost and the leftmost elements of the DEQUEUE; dq_size returns the number of elements in the DEQUEUE (or 0 if there are none); dq_empty returns true if there are no elements in the DEQUEUE; and the operation make_null initializes a DEQUEUE.

We will make use of two distinct DEQUEUEs: the *prettyprinting buffer* and the *conditional break* DEQUEUE. We assume polymorphic operations on DEQUEUEs. Each element of the prettyprinting buffer contains a character and an associated grouping level. The character is either a printable character, a space, or one of the special symbols MARKER, NEWLINE, INDENT, or OUTDENT. The grouping level is only relevant for the MARKER character, which indicates a placeholder for a connected line break; otherwise the level is ∞.

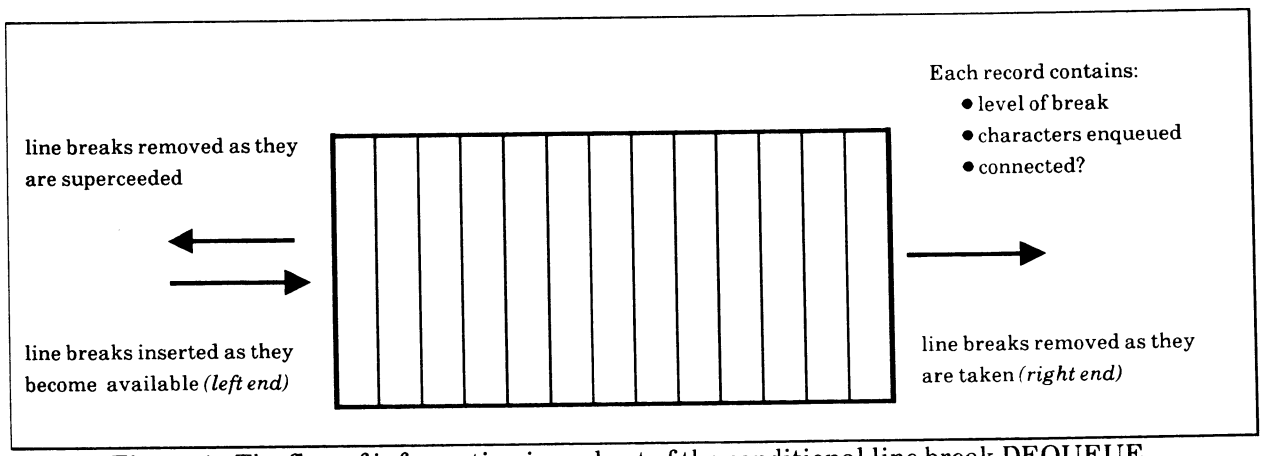

Figure 1. The flow of information in and out of the conditional line break DEQUEUE

A record stored in the conditional break DEQUEUE contains three fields: chars_enqueued, which is the number of actual characters to the left of the break in the entire string (including special characters the prettyprinter may insert into the string); the grouping level of the break, which is the number of open grouping

symbols minus the number of close grouping symbols preceding the break; and connected, which is true if the break is a connected break. Figure 1 shows the flow of information in and out of the conditional break DEQUEUE.

Four global variables are used to hold information relevant to the state of the prettyprinting buffer. The algorithm maintains the total number of all characters and the total number of printable characters that have been flushed from the prettyprinting buffer in the variables total_chars_flushed and total_pchars_flushed respectively, as well as the number of characters that have been enqueued in total_chars_enqueued and total_pchars_enqueued. From these values we can compute the number of printable characters in the prettyprinting buffer as total_pchars_enqueued minus total_pchars_flushed.

The routine print_buffer(k) sends the k leftmost characters of the prettyprinting buffer DEQUEUE to the output device (using a generic routine output(c) where c is the character to output). The print_buffer routine interprets the special characters that have been inserted into the buffer. When the special characters INDENT and OUTDENT are seen, the variable device_left_margin is modified. When the special character NEWLINE is seen a linefeed character is sent to the output device which directs the device to move to the column specified by the current device_left_margin of the next line. The special character MARKER is used as the placeholder in the prettyprinting buffer for connected line breaks. When the special character MARKER is seen, the print_buffer routine checks the grouping level associated with a MARKER and outputs a newline if break_level is greater than or equal to the the level of the break; otherwise no action is taken.

Conditional Breaks. As conditional breaks are encountered, we enqueue onto the left end of the break DEQUEUE a record containing the current value of total_chars_enqueued, the grouping level, and a flag which is true if the break is a connected break. Before putting a new element on the break DEQUEUE we first remove from the left end of the DEQUEUE all breaks that are at a greater grouping level (more deeply nested), since we now have a conditional line break that is preferred to the ones we are removing from the DEQUEUE. If the break on the left of the DEQUEUE is at the same grouping level as the new break, it is removed. If the break on the DEQUEUE is a connected break and the new break is an optional break, however, we do not remove that break because we may need both breaks in the future. We then enqueue the new break onto the left end of the DEQUEUE. If the

breakis a connected break, we also insert the special MARKER character into the prettyprinting buffer along with the grouping level of the break.

Unconditional Breaks. When an unconditional line break is encountered, the conditional line break DEQUEUE is flushed, removing all pending conditional breaks. Clearly, once a new line has been started, there is no longer the possibility of taking a conditional line break that appears on a previous line.

Choosing a Break. If we need to take a conditional break, the rightmost element of the break DEQUEUE contains the outermost break available; choosing it will break the fewest number of groupings. We flush all characters from the prettyprinting buffer to the left of the break chosen. If the break is an optional break, we send a newline to the output device; if the break is a connected break, there is already a MARKER character in the buffer that will force an appropriate line break.

Connected Breaks. When printing from the prettyprinting buffer, the variable break_level is equal to the level of the deepest grouping in the prettyprinting buffer that is being broken across multiple lines. The value of break_level indicates that connected breaks at that level or less deeply nested should be taken as line breaks, otherwise they are discarded. If while scanning the input string we encounter a connected break at a level at least as deeply nested as break_level, we immediately take it as a line break.

While printing the prettyprinting buffer, break_level is equal to the level of the break that caused the printing to occur (the current level if an unconditional line break is taken, otherwise the level of the conditional break). While scanning input characters, we maintain an invariant that break_level is the level which is currently being broken and so is less than or equal to current_level; when we see a close grouping symbol while break_level is equal to current_level, we know that the level being closed will not be broken anymore, but the level outside of it will, so we decrement break_level .

Space and Time Bounds. The space required for the prettyprinting buffer is equal to the number of printable characters in the buffer plus the number of special characters in the buffer. The number of printable characters in the buffer is limited to the line width of the output device. Assuming no redundant commands are in the prettyprinting string, there can not be more MARKER characters in the prettyprinting buffer than printable characters. Therefore, the space required by the prettyprinting buffer is equal to m, the width of the output, plus i, the maximum

number of indent and outdent commands that occur on a single line. In practice, i is no greater than m. We can eliminate the term i by adding indent and outdent primitives that can change the left margin by an arbitrary amount, in which case there would never need to be more indent and outdent commands than printable characters in the buffer. The space required by the break DEQUEUE is proportional to the number of conditional breaks currently available, and if there are no redundant breaks, this is not more than the number of characters in the prettyprinting buffer. Therefore, the space bound of the algorithm is $O(m)$, where m is the line width of the output device.

The time required is linear in the length of the string to be prettyprinted. The time required by the procedure prettyprint to process each character is constant except when print_buffer is called and in the case where it may have to discard several items from a break DEQUEUE. The time spent in print_buffer can be charged to the cost of putting a character into the prettyprinting buffer since a character put in the prettyprinting buffer can only be printed once. Similarly, an item can only be popped from a break DEQUEUE once, so we can charge the cost of popping elements from a break DEQUEUE to the operation of pushing them onto the DEQUEUE. Therefore, the time required by the algorithm is $O(n)$, where n is the length of the string to be prettyprinted (including prettyprinting commands).

Notes. At initialization both of the DEQUEUEs are empty, current_level, total_chars_enqueued, total_pchars_enqueued, total_chars_flushed and total_pchars_flushed are set to zero, and break_level is initialized to -1 to reflect that no grouping levels have yet been broken.

The observant reader may have noticed that a simplifying assumption has been made in the algorithm. We assume that there will be some kind of line break between two adjacent groupings (e.g. the string "*{*u,cv,cw*}*;*{*x,cy,cz*}*" would not occur because we would expect some kind of line break to be associated with the semicolon). Otherwise, we have trouble distinguishing which connected breaks of the same level belong to the group we are breaking. If this assumption is violated, the algorithm described will behave inconsistently concerning the question of whether connected breaks in those two groups are in fact connected; otherwise it will work as expected. In practice, this situation is unlikely to arise. The problem could be avoided by maintaining the location in the prettyprinting stream of the start of each grouping level; this would not change our time or space bounds, but would complicate the description of the algorithm.

IV. Use in a Structure Editor

In a structure editor, a program (or other structured object) is displayed by traversing the abstract syntax tree of the structure and displaying the unparsing strings associated with each operator. Figure 2 shows how the prettyprinting string for the **if** statement example above could be derived from the abstract syntax tree of the statement. The labels in solid boxes represent syntactic operators, and the

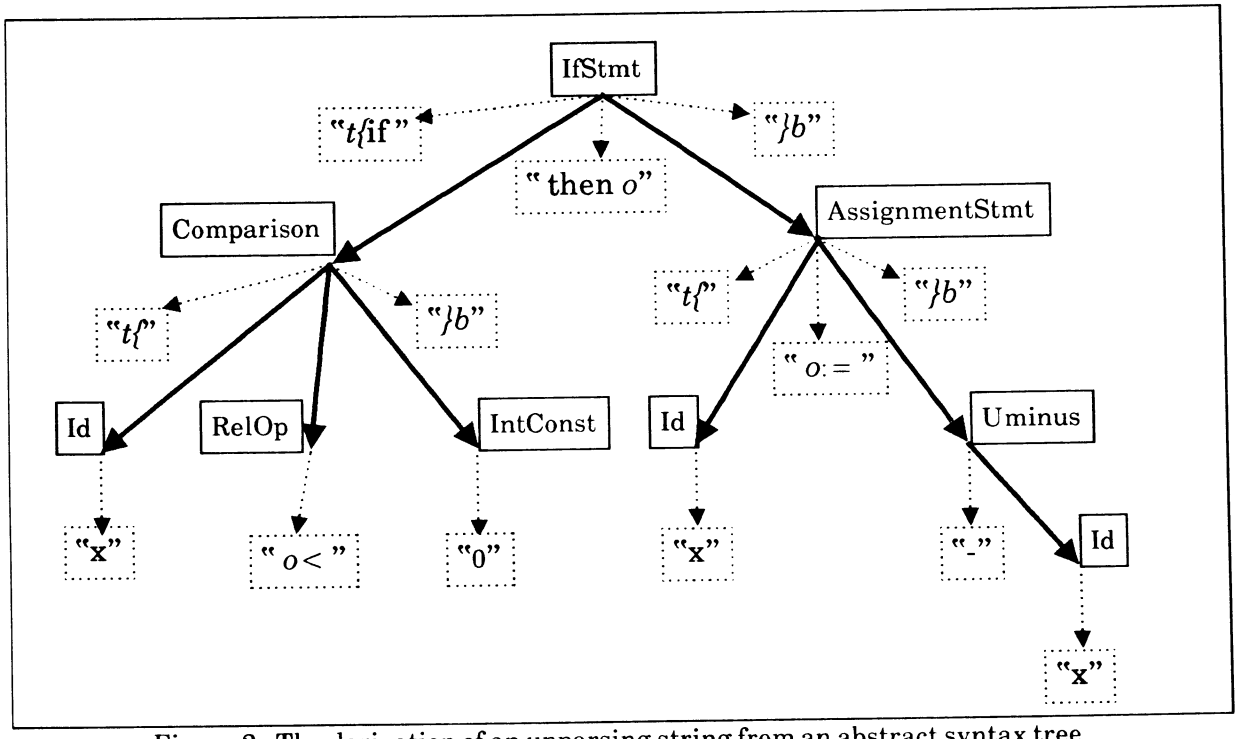

Figure 2. The derivation of an unparsing string from an abstract syntax tree

strings in dashed boxes show each segment of the prettyprinting string and its relation to the tree. The strings are part of the grammar specification supplied to the editor generator. During execution the prettyprinting string need never be assembled in its entirety; instead, each substring may be sent to the prettyprinting routine as the tree is traversed. This approach allows incremental unparsing when traversal is begun at any tree node and the prettyprinter is appropriately initialized. To restart the prettyprinter we have to restore the values of current_level, device_left_margin, break_level, and the row and column of the output device.

V. Conclusion

In addition to enhancing our editors for block-structured languages, this algorithm has allowed us to use the Synthesizer Generator for applications for which it was previously impractical. The use of prettyprinting within editors produces no noticeable increase in the time required to update the display. We hope that the simplicity of the algorithm presented here will encourage others to adapt and expand it.

VI. Acknowledgements

We would like thank Roger Hoover, Carla Marceau and Tim Teitelbaum for their comments and suggestions on this paper.

VII. References

[1] Oppen, Derek C. Prettyprinting. *TOPLAS (ACM)* **2**, 4 (October 1980), 465-483.

[2] Reps, T, and Teitelbaum, T. The Synthesizer Generator. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 23-25, 1984. Appeared as joint issue: *SIGPLAN Notices* (ACM) *19*, 5 (May 1984), and *Soft. Eng. Notes* (ACM) **9**, 3 (May 1984), 42-48.

[3] Reps, T, and Teitelbaum, T. The Cornell program synthesizer: a syntax directed programming environment. *Commun. ACM* **24**, 9 (September 1981), 563-573.

[4] Ellison, Robert J., and Staudt, Barbara J. The Evolution of the GANDALF System. *Journal of Systems and Software* **5**,2 (May 1985), 107-119.

[5] Mikelsons, Martin. Prettyprinting in an Interactive Environment. In Proceedings of the ACM SIGPLAN /SIGOA Symposium on text manipulation, Portland, OR, June 8-10, 1981. Appeared as *SIGOA Newsletter* (ACM) **2**, 1 and 2 (Spring/Summer 1981), 108-116.

[6] Conway, Richard and Gries, David. *An Introduction to Programming: A Structured Approach Using PL/I and PL/C*. Little, Brown, and Company, 1979.

[7] Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

Appendix. The Prettyprinter Algorithm

```
procedure prettyprint(string: array [1..n] of char)
  for i := 1 to n do
    if string[i] = ESCAPE and i < n cand string[i + 1] in ['{','}','t','b','n','o','c'] then
      i := i + 1;
      case string[i] of
        '{' : (* start group *)
          current_level := current_level + 1;
        '}' : (* end group *)
          current_level := current_level - 1;
          if break_level > current_level then
            break_level := current_level;
        't' : (* indent *)
          right_enqueue(buffer, [INDENT, ∞]);
          total_chars_enqueued := total_chars_enqueued + 1;
        'b' : (*outdent*)
          right_enqueue(buffer, [OUTDENT, ∞]);
          total_chars_enqueued := total_chars_enqueued + 1;
        'n' : (*unconditional line break *)
          make_null(break_dq);
          break_level := current_level;
          right_enqueue(buffer, [NEWLINE, ∞]);
          total_chars_enqueued := total_chars_enqueued + 1;
          print_buffer(dq_size(buffer));
        'o' : (* optional line break *)
          while not dq_empty(break_dq) cand
                (dq_left(break_dq).level > current_level
                  or (dq_left(break_dq).level = current_level
                    and not dq_left(break_dq).connected)) do
            (* discard breaks we are no longer interested in *)
            left_dequeue(break_dq);
          left_enqueue(break_dq,
                [total_chars_enqueued, current_level, false]);
        'c' : (* connected line break *)
          if break_level < current_level then
            while not dq_empty(break_dq) cand
                  dq_left(break_dq).level >= current_level do
              (* discard breaks we are no longer interested in *)
              left_dequeue(break_dq);
            right_enqueue(buffer, [MARKER, current_level]);
            total_chars_enqueued := total_chars_enqueued + 1;
```

```
                left_enqueue(break_dq,
                        [total_chars_enqueued, current_level, true]);
            else (* take an immediate line break, break_level = current_level *)
                make_null(break_dq);
                right_enqueue(buffer, [NEWLINE, ∞]);
                total_chars_enqueued : = total_chars_enqueued + 1;
                print__buffer(dq__size(buffer));
    else   (* it is a printable character *)
        if (total_pchars_enqueued - total_pchars_flushed) + device_left_margin
                > = device_output_width then (* must split line *)
            if not dq_empty(break_dq) then (* split line at a break *)
                temp : = right_dequeue(break_dq);
                break_level : = temp.level;
                print_buffer(temp.chars_enqueued - total_chars_flushed);
                if not temp.connected then
                    output('\n');
                break_level : = MIN(break_level, current_level);
            else
                there are no breaks to take
        (* put the current character into the buffer *)
        right_enqueue(buffer, [string[i], ∞]);
        total_chars_enqueued : = total_chars_enqueued + 1;
        total_pchars_enqueued : = total_pchars_enqueued + 1;
end;  (* prettyprint *)

procedure print_buffer(k: integer);
    for i : = 1 to k do
        temp : = left_dequeue(buffer);
        total_chars_flushed : = total_chars_flushed + 1;
        case temp.character of
            MARKER:
                if temp.level < = break_level then
                    output('\n');
            NEWLINE:
                output('\n');
            INDENT:
                device_left_margin : = device_left_margin + INDENT_WIDTH;
            OUTDENT:
                device_left_margin : = device_left_margin - INDENT_WIDTH;
            default:
                total_pchars_flushed : = total_pchars_flushed + 1;
                output(temp.character);
end;  (* print_buffer *)
```