Prettyprinting

DEREK C. OPPEN
Stanford University

An algorithm for prettyprinting is given. For an input stream of length n and an output device with linewidth m, the algorithm requires time $O(n)$ and space $O(m)$. The algorithm is described in terms of two parallel processes: the first scans the input stream to determine the space required to print logical blocks of tokens; the second uses this information to decide where to break lines of text; the two processes communicate by means of a buffer of size $O(m)$. The algorithm does not wait for the entire stream to be input, but begins printing as soon as it has received a full line of input. The algorithm is easily implemented.

Key Words and Phrases: prettyprinting, formating, paragraphing, text editing, program manipulation
CR Category: 4.12

1. INTRODUCTION

A *prettyprinter* takes as input a stream of characters and prints them with aesthetically appropriate indentations and line breaks. As an example, consider the following stream:

```
var x:integer; y:char; begin x := 1; y := 'a' end
```

If our line width is 40 characters, we might want it printed as follows:

```
var x:integer; y:char;
  begin x := 1; y := 'a' end
```

If our line width is 30 characters, we might want it printed as follows:

```
var x:integer;
    y:char;
begin
  x := 1;
  y := 'a';
end
```

But under no circumstances do we want to see

```
var x:integer; y:
char; begin x := 1;
y := 'a'; end
```


This work was supported by the National Science Foundation under Grant MCS 78-02835.
Author's address: Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

Prettyprinters are common components of LISP environments, where trees or *s*-expressions are data objects which are interactively manipulated and have to be displayed on a screen or on the printed page. Since the main delimiters in LISP are parentheses and spaces, a LISP program or *s*-expression is visually intolerable unless prettyprinted. (See [2] or [3] for descriptions of some prettyprinters for LISP.)

Prettyprinters for block-structured languages appear less commonly, perhaps because "programming environments" for such languages did not exist until recently. (See the references and bibliography for descriptions of some implemented prettyprinters.) Happily, this situation is changing fast. Prettyprinters are integral components of any programming environment tool. For example, editors for block-structured languages benefit enormously from a prettyprinter—as the user interactively makes changes to his program text, the modified program is pleasingly displayed. Not only does this make it easier for the user to read his program text, but it makes it easier for him to notice such common programming errors as missing **ends**. Compilers should use prettyprinters to print out error messages in which program text is displayed; this would make the error much more understandable. Prettyprinters are useful in any system which prints or displays messages or other output to the user.

Prettyprinters have traditionally been implemented by rather ad hoc pieces of code directed toward specific languages. We instead give a language-independent prettyprinting algorithm. The algorithm is easy to implement and quite fast. It is not, however, as sophisticated as it might be, and certainly cannot compete with typesetting systems (such as TEX [4]) for preparing text for publication. However, it seems to strike a reasonable balance between sophistication and simplicity, and to be appropriate as a subcomponent of editors and the like.

We do not discuss in detail the question of how to interface the prettyprinter described here with any specific language. In general, the prettyprinter requires a front-end processor, which knows the syntax of the language, to handle questions about where best to break lines (that is, questions about the inherent block or indenting structure of the language) and questions such as whether blanks are redundant. We describe in Section 6 two approaches we have taken to implementing a preprocessor for prettyprinting.

2. BASIC NOTIONS

The basic idea of how a prettyprinter works is well established in the folklore, and the algorithms of which the author is aware all provide roughly the same set of primitives, which the algorithm described here also provides.

A prettyprinter expects as input a stream of characters. A character may be a printable character such as "a", "3", "&", or ",", or it may be a *delimiter* such as blank, carriage-return, linefeed, or formfeed. A contiguous sequence of printable characters (that is, not delimiters) is called a *string*. The prettyprinter may break a line between strings but not within a string.

We differentiate between several types of delimiters. The first type of delimiter is the blank (carriage returns, formfeeds, and linefeeds are treated as blanks). The next two types correspond to special starting and ending delimiters for logically contiguous blocks of strings. We denote the delimiters ⟦ and ⟧, respec-

tively. The algorithm will try to break onto different lines as few blocks as possible. For instance, suppose we wish to print $f(a, b, c, d) + g(a, b, c, d)$ on a display which is only 20 characters wide. We might want this expression printed as

$$f(a, b, c, d)$$
$$\quad + g(a, b, c, d)$$

or as

$$f(a, b, c, d) +$$
$$\quad g(a, b, c, d)$$

but definitely not as

$$f(a, b, c, d) + g(a,$$
$$b,$$
$$c,$$
$$d)$$

We can avoid the third possibility by making $f(a, b, c, d)$ and $g(a, b, c, d)$ logically contiguous blocks, that is, by surrounding each by ⟦ and ⟧. In fact, since this expression undoubtedly appears within some other text, we should include logical braces around the whole expression as well:

$$⟦⟦ f(a, b, c, d)⟧ + ⟦ g(a, b, c, d)⟧⟧$$

(You might be asking at this point why the algorithm does not recognize that parentheses are delimiters and thus that $g(a, b, c, d)$ should not be broken if possible. But the prettyprinting algorithm given here is a general-purpose algorithm providing primitives for prettyprinting and is not tailored to any particular language. The example could have been written just as easily with two **begin** ... **end** blocks.)

We later allow refinements to the above set of delimiters, but for the moment we describe the algorithm using just these three. We assume that the algorithm is to accept as input a "stream" of *tokens*, where a token is a string, a blank, or one of the delimiters ⟦ and ⟧. A *stream* is recursively defined as follows:

(1) A string is a stream.
(2) If $s_1, \ldots, s_k$ are streams, then $⟦ s_1 \langle\text{blank}\rangle s_2 \langle\text{blank}\rangle \cdots \langle\text{blank}\rangle s_k ⟧$ is a stream.

As we see later, this definition of an "allowable" stream is a little too restrictive in practice but makes it easier to describe the basic algorithm. We make one additional assumption to simplify discussion of the space and time required by the basic algorithm: No string is of length greater than the line width of the output medium.

3. AN INEFFICIENT BUT SIMPLE ALGORITHM

We first describe an algorithm which uses too much storage, but which should be fairly easy to understand. The algorithm uses functions *scan*( ) and *print*( ).

The input to *scan*( ) is the stream to be prettyprinted; *scan*( ) successively adds the tokens of the stream to the right end of a buffer. Associated with each token in the buffer is an integer computed by *scan*( ) as follows. Associated with each string is the space needed to print it (the length of the string). Associated with each ⟦ is the space needed to print the block it begins (the sum of the lengths of the strings in the block plus the number of blanks in the block). Associated with each ⟧ is the integer 0. Associated with each blank is the amount of space needed to print the blank and the next block in the stream (1 + the length of the next block).

In order to compute these lengths, *scan*( ) must "look ahead" in the stream; it uses the buffer *stream* to store the tokens it has already seen. When *scan*( ) has computed the length l for the token x at the left end of the buffer, it calls *print*(x, l) and removes x and l from the buffer. The buffer is therefore a first-in-first-out buffer.

The length information associated with each token is used by *print*( ) to decide how to print it. If *print*( ) receives a string, it prints it immediately. If *print*( ) receives a ⟦, it pushes the current indentation on a stack but prints nothing. If it receives a ⟧, it pops the stack. If *print*( ) receives a blank, it checks to see if the next block can fit on the present line. If so, it prints a blank; if not, it skips to a new line and indents by the indentation stored on the top of the stack plus an arbitrary offset (in this case, 2).

We describe *print*( ) first, because it is the simpler routine. It uses auxiliary functions *output*(x), which prints x on the output device, and *indent*(x), which starts a new line and indents x spaces. *print*( ) also uses a local stack S with operations *push*( ), *pop*( ), and *top*( ) (the latter returns the top of the stack without popping it). It also uses the constant *margin*, which is the line width, and a variable *space* (initially equal to *margin*), which stores the number of spaces left on the present line.

```
print(x, l):
cases
  x:string ⇒ output(x); space := space − l;
  x:⟦ ⇒ push(S, space);
  x:⟧ ⇒ pop(S);
  x:blank ⇒ if l > space
      then space := top(S) − 2; indent(margin − space);
      else output(x); space := space − 1;
```

Now we are ready for *scan*( ). It successively receives tokens from *receive*( ) and stores each at the right of the buffer *stream*. It uses a second buffer *size* for storing the lengths associated with tokens as described above. It uses variables *left* and *right* for pointing at the left and right ends of these buffers (the buffers are assumed to be of arbitrary length). It uses a local stack S with operations *push*( ), *pop*( ), and *top*( ), and a local variable x. Finally, it uses a variable *rightotal* to store the total number of spaces needed to print all elements of the buffer from *stream*[1] through *stream*[*right*].

```
scan( ):local x;
forever x := receive( );
  cases
    x:eof ⇒ halt;
    x:[[ ⇒
      cases S:empty ⇒ left := right := rightotal := 1;
            otherwise ⇒ right := right + 1;
      stream[right] := x;
      size[right] := −rightotal;
      push(S, right);
    x:]] ⇒
      right := right + 1;
      stream[right] := x;
      size[right] := 0;
      x := pop(S);
      size[x] := rightotal + size[x];
      if stream[x]:blank then x := pop(S); size[x] := rightotal + size[x];
      if S:empty
        then until left > right do
          print(stream[left], size[left]);
          left := left + 1;
    x:blank ⇒
      right := right + 1;
      x := top(S);
      if stream[x]:blank then size[pop(S)] := rightotal + size[x];
      stream[right] := x;
      size[right] := −rightotal;
      push(S, right);
      rightotal := rightotal + 1;
    x:string ⇒
      cases S:empty ⇒ print(x, length(x));
            otherwise ⇒
                right := right + 1;
                stream[right] := x;
                size[right] := length(x);
                rightotal := rightotal + length(x);
```

To keep track of occurrences of delimiters, *scan*( ) uses the stack. If it receives a [[, it stores the [[in *stream*[*right*] and −*rightotal* in *size*[*right*]; when it receives the corresponding]], it computes the space needed for this block—it is (the current value of) *rightotal* + *size*[*right*]. If *scan*( ) receives a]], the top of the stack is either the index of the [[starting the block (if the block contained no blanks) or the index of the previous blank in this block and underneath that the index of the [[starting the block. In the former case, *scan*( ) computes the length associated with the [[; in the latter, it computes the lengths associated with the [[and the blank. If *scan*( ) receives a blank, the top of the stack contains either the index to the start of the block or the index to the previous blank in the block. If the latter, *scan*( ) computes the length associated with the previous blank.

A nice property of *scan*( ) is that it requires time linear in the length of the stream (as does *print*( )). An undesirable property is that it also requires space linear in the length of the stream. To see this, suppose the whole stream is delimited by [[and]]. Then *scan*( ) will read the whole stream before it computes

the length of this block. (If all blocks are small, this may be considered an unimportant point.) Another problem with *scan*( ) is that it may have to process large amounts of data before the first character can be printed. This property is undesirable in an interactive environment: we want to start printing characters as soon as possible, if only to give the user positive reinforcement.

We are now ready for the next iteration of the algorithm, which requires space $O(m)$ rather than $O(n)$, that is, space which depends only on the line width of the output medium and not on the length of the input.

4. AN EFFICIENT BUT LESS SIMPLE ALGORITHM

Let us consider again the roles of *scan*( ) and *print*( ). It may be helpful to visualize them as two parallel processes communicating via the buffers *stream* and *size*. *scan*( ) wants to put information into the buffers on the right, while *print*( ) wants to remove information from them on the left. That is, *scan*( ) wants to advance the cursor variable *right*, while *print*( ) wants to advance the cursor variable *left*.

The problem is that *print*( ) cannot use *stream*[*left*] until *size*[*left*] has a positive value. In the algorithm given in the previous section, if *stream*[*left*] is a ⟦ or a blank, *scan*( ) will not fill in *size*[*left*] until it has seen the corresponding ⟧ or next corresponding blank, and this holds up *print*( ) unnecessarily. Since there can only be m characters on a line, it is not necessary for *scan*( ) to compute an exact value for *size*[*left*] if *size*[*left*] is going to be greater than m. As soon as *scan*( ) knows that *size*[*left*] must be greater than m, it may as well make *size*[*left*] equal to ∞. That is, as soon as the sum of the lengths of strings plus the number of blanks between *left* and *right* in *stream* exceeds m, we can let *print*( ) advance.

Thus, *scan*( ) and *print*( ) need not get too far apart in accessing the buffers. Allowing for the fact that *stream* stores occurrences of ⟦ and ⟧ as well as strings and blanks, *right* − *left* need never exceed $3m$. Hence our buffer size can be linear in m, and we never need look ahead more than $3m$ tokens before being able to print *something*.

In fact, we can do even better. At any moment, *print*( ) has printed zero or more characters on a line. All it needs to know in order to make a decision on how to print the next block in the stream is whether or not the block can fit in the remaining space on the line. Consequently, we do not have to test whether the space required by the elements of *stream* between *left* and *right* exceeds m, but rather whether or not it exceeds the present value of *space*—the variable used in *print*( ) to store the number of spaces remaining on the present line.

We are now ready to describe our refined algorithm. It is closely related to our previous algorithm. *print*( ) remains the same. *scan*( ) uses an additional variable *leftotal*, which is the total number of spaces needed to print all elements of the buffer from *stream*[1] through *stream*[*left*] (analogous to *rightotal* which measures from *stream*[1] through *stream*[*right*]). *popbottom*( ) removes the *bottom* element of the stack (so our local stack is no longer a true stack—we can flush elements from its bottom). And when *scan*( ) chooses to force output from the left of the stream, it does so by calling the auxiliary function *advanceleft*( ). We implement *stream* and *size* as two arrays of size *arraysize*, a constant equal to

$3m$, say. The variables *left* and *right* are initially 1, pointing to the start of the arrays.

```
scan( ):local x;
forever x := receive( );
  cases
    x:eof ⇒ halt;
    x:[[ ⇒
      cases S:empty ⇒ left := right := leftotal := rightotal := 1;
            otherwise ⇒ right := if right = arraysize then 1 else right + 1;
      stream[right] := x;
      size[right] := −rightotal;
      push(S, right);
    x:]] ⇒
      cases S:empty ⇒ print(x, 0);
            otherwise ⇒
                right := if right = arraysize then 1 else right + 1;
                stream[right] := x;
                size[right] := 0;
                x := pop(S);
                size[x] := rightotal + size[x];
                if stream[x]:blank and ¬S:empty
                    then x := pop(S); size[x] := rightotal + size[x];
                if S:empty then advanceleft(stream[left], size[left]);
    x:blank ⇒
      cases S:empty left := right := rightotal := 1;
            otherwise ⇒
                right := if right = arraysize then 1 else right + 1;
                x := top(S);
                if stream[x]:blank then size[pop(S)] := rightotal + size[x];
      stream[right] := x;
      size[right] := −rightotal;
      push(S, right);
      rightotal := rightotal + 1;
    x:string ⇒
      cases S:empty ⇒ print(x, length(x));
            otherwise ⇒
                right := if right = arraysize then 1 else right + 1;
                stream[right] := x;
                size[right] := length(x);
                rightotal := rightotal + length(x);
                while rightotal − leftotal > space do
                  size[popbottom( )] := 999999;
                  advanceleft(stream[left], size[left]);

advanceleft(x, l):
if l ≥ 0 then
  print(x, l);
  cases x:blank ⇒ leftotal := leftotal + 1;
        x:string ⇒ leftotal := leftotal + l;
  if left ≠ right then
    left := if left = arraysize then 1 else left + 1;
    advanceleft(stream[left], size[left]);
```

We have implemented the buffers in the obvious way as ring buffers. *print*( )

follows *scan*( ) around the buffers (that is, *left* follows *right*), and as long as the size of the buffers is at least $3m$, *scan*( ) will not overtake *print*( ).

All that remains is to describe how to implement the local stack S. One way is to implement it also as an array of size *arraysize*, with indexing variables *top* and *bottom* initially equal to 1, and a Boolean variable *stackempty* initially set to true. We implement the test S:*empty* as a test on the value of *stackempty* and the other stack operations as follows:

```
push(S, x):
if stackempty
  then stackempty := false
  else top := if top = arraysize then 1 else top + 1;
S[top] := x;

pop(S):local x;
x := S[top];
if bottom = top
  then stackempty := true
  else top := if top = 1 then arraysize else top - 1;
return x;

top(S):return S[top];

popbottom(S):local x;
x := S[bottom];
if bottom = top
  then stackempty := true
  else bottom := if bottom = arraysize then 1 else bottom + 1;
return x;
```

5. MODIFICATIONS TO THE BASIC ALGORITHM

The algorithm actually implemented by the author is somewhat more sophisticated. The complete algorithm is given in the appendix.

There is one major deficiency in the set of delimiters we chose, and that is that the delimiter *blank* is not subtle enough. It needs at least three associated parameters.

First, we want a variable offset associated with each blank instead of the constant offset 2 used in the algorithm. This allows us to have, for example, the following:

```
cases 1:...
      2:...
      3:...
```

where we have indented six characters to line up the cases. Variable offsets also allow us the option of choosing, say, either of the following ways of indenting **begin** ... **end** blocks (assuming a narrow enough line width to force breaking):

```
begin
  x := f(x);
  y := f(y);
end;
```

```
begin
  x := f(x);
  y := f(y);
  end;
```

Second, we want to differentiate between two types of blanks, which we call *consistent* and *inconsistent* blanks. If a block cannot fit on a line, and the blanks in the block are *consistent* blanks, then each subblock of the block will be placed on a new line. If the blanks in the block are *inconsistent*, then a new line will be forced only if necessary. The reason for this differentiation is that we may prefer

```
begin
  x := f(x);
  y := f(y);
  z := f(z);
  w := f(w);
  end;
```

to

```
begin
  x := f(x); y := f(y);
  z := f(z); w := f(w);
  end;
```

but prefer

```
locals x, y, z, w,
       a, b, c, d;
```

to

```
locals x,
       y,
       z,
       w,
       a,
       b,
       c,
       d;
```

(assuming again that the line width is sufficiently narrow to force breaking). That is, for **begin** . . . **end** blocks we may prefer consistent breaking, but for declaration lists we may prefer inconsistent breaking.

Finally, we want to be able to parameterize the length of each blank. A blank of length zero (that is, an invisible blank) is useful when one wants to insert a possible line break but print nothing otherwise.

There is one other major modification that the author has found useful, especially if this prettyprinter is used as the output device for an unparser. Consider the following stream for printing out $f(g(x, y))$:

$$[\![f([\![g(x, \langle \text{blank} \rangle y)]\!] \langle \text{blank} \rangle)]\!]$$

This may result in the following output:

$$f(g(x, y)$$
$$)$$

given appropriate line width and parameters to the delimiters. We might instead prefer

$$f(g(x,$$
$$y))$$

even though the first is correct according to the algorithm (since it breaks fewer logical blocks). We could try to stop a line break from occurring between the right parentheses by sending the stream

$$[\![f([\![g(x, \langle\text{blank}\rangle y)]\!])]\!]$$

that is, by deleting the ⟨blank⟩ between the parentheses. But this violates the assumptions given in Section 2 on what constitutes a legal stream. The algorithm in the appendix tries to handle in a reasonable fashion any sequence of tokens (if the stream satisfies the assumptions given in Section 2, the output is the same as given by the basic algorithms). It does assume, however, that occurrences of [[and]] are balanced and that the stream begins with a [[(for correct initialization). In particular, it effectively changes (dynamically) each occurrence of]]⟨string⟩ into ⟨string⟩]].

6. A PREPROCESSOR FOR PRETTYPRINTING

Let us briefly consider the question of how to tailor the prettyprinter to some specific language.

The simplest way is to drive the prettyprinter directly from the parse tree produced by a parser or the parsing component of a compiler. Typically, this component first translates the program (a stream of text) into a tree. For instance, if the grammar for the language contains the production

$$\langle\text{term}\rangle \rightarrow \langle\text{subterm}\rangle\langle\text{operator}\rangle\langle\text{subterm}\rangle$$

the parser may generate, when parsing $a + b$, the subtree consisting of a node with three successors: the subtrees corresponding to a, +, and b. The preprocessor to the prettyprinter then walks this tree in what might be called a "recursive descent unparse." For instance, when faced with our example tree for $a + b$, the unparser may first generate a [[, recursively unparse the first subtree to generate a, generate a blank, unparse the subtree for +, generate another blank, unparse the subtree for b, and finally generate a closing]].

Driving the prettyprinter from the parse tree is relatively straightforward, especially in languages such as LISP where the program *is* a tree. A disadvantage of waiting for the parse tree to be constructed is that prettyprinting is no longer on-line: the whole program must be parsed before prettyprinting can begin. In many situations this is no disadvantage.

Notice that this method makes automatic use of the scanner of the parser to resolve all questions such as whether there are redundant blanks. This is, of course, a double-edged sword; the scanner component of many parsers also

deletes useful information (such as comments). We must modify the scanner to pass this information on and modify the parse tree to save the information.

We have used this "unparsing" approach to write a prettyprinter for formulas produced by the Stanford PASCAL Verifier (with Wolf Polak) and for MESA (with Steve Wood).

Another approach we have used also makes use of a scanner and a parser for a language, but uses the parser to drive the prettyprinter directly, without using the parse tree. For instance, if we use a recursive descent parser, we can add code to the syntax routines of the parser to transmit to the prettyprinter the delimiters [[, ⟨blank⟩, and]], and the other tokens.

If we are using a table-driven parser whose semantic routines are called bottom-up, we can use a slightly different approach. First, notice that the information needed by the prettyprinter can often conveniently be represented directly in the grammar; for instance, in our example production above,

$$\langle \text{term} \rangle \rightarrow [[\langle \text{subterm} \rangle \langle \text{blank} \rangle \langle \text{operator} \rangle \langle \text{blank} \rangle \langle \text{subterm} \rangle]]$$

Suppose we are using a parser generator (to generate a table-driven parser). We modify the grammar of the language to contain prettyprinting information as above, where [[, ⟨blank⟩, and]] are nonterminals mapping only to the empty string. The semantic routines associated with these nonterminals transmit, respectively, [[, ⟨blank⟩, and]] to the prettyprinter. The other semantic routines transmit to the prettyprinter the other tokens in the stream. Because table-driven parsers typically call their semantic routines in a bottom-up fashion, we may have to modify the grammar slightly to ensure that tokens are sent to the prettyprinter in the correct order. For instance, consider the production

$$\langle \text{block} \rangle \rightarrow \textbf{begin}\ \langle \text{statementlist} \rangle\ \textbf{end}$$

We do not want the semantic routine associated with ⟨statementlist⟩ to be called before the semantic routine for ⟨block⟩, because we do not want the tokens corresponding to ⟨statementlist⟩ to be printed before the **begin** is printed. We can correct this by changing this production to

$$\langle \text{block} \rangle \rightarrow \langle \text{begin} \rangle \langle \text{statementlist} \rangle\ \textbf{end}$$
$$\langle \text{begin} \rangle \rightarrow \textbf{begin}$$

so that the semantic routine corresponding to **begin** will be called (and "begin" will be printed) before the semantic routine for ⟨statementlist⟩.

The advantage of this variant is that it is very clean—the prettyprinting information for the language is represented in the grammar instead of being buried in the code. The disadvantage is that the tables for the parser may grow because of the additional productions. (The impact of this can be lessened to acceptable levels by not having explicit nonterminals for [[, ⟨blank⟩, or]], but adding code to the semantic routines for the other nonterminals to drive the prettyprinter directly. For instance, the semantic routine corresponding to the nonterminal ⟨begin⟩ above could emit the three tokens [[, "begin", and ⟨blank⟩.)

A prettyprinter for MESA has been implemented in this fashion by Philip Karlton and the author.

7. OTHER PRETTYPRINTERS

The references give pointers to some of the existing literature on prettyprinters for specific languages. There is little literature on the actual algorithms used for prettyprinting. The following are some exceptions. As before, n denotes the length of the input stream and m denotes the line width of the output device.

Goldstein [2] describes various ways of implementing prettyprinters for LISP and gives several algorithms requiring $O(n)$ time and $O(n)$ space. Dick Waters (private communication) independently discovered the observations given here on how much lookahead is required; he has implemented a prettyprinter for LISP which requires $O(mn)$ time and $O(m)$ space. Greg Nelson (private communication) has a prettyprinting algorithm which requires $O(m)$ space and $O(n)$ time. Jim Morris (private communication) has an algorithm which, like the one described here, conceptually consists of two parallel processors; it requires $O(m)$ space and $O(mn)$ time. Hearn and Norman [3] have independently discovered a similar method; their description is informal and their analysis assumes that line width is constant, but if line width is assumed to be m, their algorithm appears to have the same bounds as Morris' algorithm.

8. IN CONCLUSION

The primitives described in the previous sections seem satisfactory for most purposes. Of course, they are not perfect. For instance, we do not allow offsets which are a function of the next block in the stream. Thus, we may get

```
cases 1:...
      2:...
      3:if x = 1
          then x := f(x)
          else x := g(x);
```

where we might have preferred to indent the cases slightly less, if we knew that this would allow the **if** ... **then** ... **else** statement to fit on one line as follows:

```
cases
  1:...
  2:...
  3:if x = 1 then x := f(x) else x := g(x);
```

Another deficiency of the algorithm is that it can do nothing if there is not room on the line for a string. This might happen if we have indented k spaces and want to print a string of size greater than $margin - k$. The author does not know of any simple and graceful way to solve this problem; two crude solutions are to just wrap around the screen or else forcibly to reduce the indentation just enough to right justify the offending string.

This deficiency illustrates a general drawback of the algorithm—it does only constant space (one line width) lookahead, and its logic is not as sophisticated as it might be. However, we believe that the algorithm with its optional modifications strikes the right balance between simplicity and speed on one hand, and sophistication on the other, to be useful in the applications envisaged. It is perhaps worth repeating one desirable feature of the algorithm—it starts printing more or

less as soon as it has received a full line of input, and printing never lags more than a full line behind the input routine. This we consider an important point in "human engineering." It is also important as more systems begin to take advantage of the notion of "delayed evaluation," where parts of expressions may be output before the entire expression is computed.

APPENDIX

The following is the augmented prettyprinting algorithm implemented by Philip Karlton and the author in MESA (see [1]). Some details have been left out concerning input/output and memory allocation. Comments are preceded by two dashes; numbers are either in octal or in binary (if followed by "B").

The prettyprinter receives tokens which are records of various types. A token of type "string" contains a string. A token of type "break" denotes an optional line break; if the prettyprinter outputs a line break, it indents "offset" spaces relative to the indentation of the enclosing block; otherwise it outputs "blankSpace" blanks; these values are defaulted to 0 and 1, respectively. Tokens of type "begin" and "end" correspond to our ⟦ and ⟧, except that the type of breaks is associated with the "begin" rather than with the break itself (the type is defaulted to "inconsistent"), and an offset value may be associated with the "begin" (the offset applies to the whole block and is defaulted to 2). A token of type "eof" initiates cleanup. Finally, a "linebreak" is a distinguished instance of "break" which forces a line break (by setting "blankSpace" to be a very large integer).

```
PrettyPrint: DEFINITIONS =
  BEGIN
-- types
  TokenType: TYPE = {string, break, begin, end, eof};
  Token: TYPE = RECORD[
    SELECT type: TokenType FROM
      string ⇒ [string: string],
      break ⇒ [
        blankSpace: [0. .MaxBlanks] ← 1,   -- number of spaces per blank
        offset: [0. .31] ← 0],   -- indent for overflow lines
      begin ⇒ [
        offset: [0. .127] ← 2,   -- indent for this group
        breakType: Breaks ← inconsistent],   -- default "inconsistent"
      end ⇒ NULL,
      eof ⇒ NULL,
      ENDCASE];

  MaxBlanks: CARDINAL = 127;
  Breaks: TYPE = {consistent, inconsistent};
  LineBreak: break Token = [break[blankSpace: MaxBlanks]];
  END.

PrettyPrinter: PROGRAM
  EXPORTS PrettyPrint =
  BEGIN
  margin, space: INTEGER;
  left, right: INTEGER;
  token: DESCRIPTOR FOR ARRAY OF Token ← DESCRIPTOR[NIL, 0];
  size: DESCRIPTOR FOR ARRAY OF INTEGER ← DESCRIPTOR[NIL, 0];
  leftTotal, rightTotal: INTEGER;
```

```
sizeInfinity: INTEGER = 77777B;
scanStack: DESCRIPTOR FOR ARRAY OF INTEGER ← DESCRIPTOR[NIL, 0];
scanStackEmpty: BOOLEAN;
top, bottom: CARDINAL;
printStack: PrintStack ← CreatePrintStack[63];

PrettyPrintInit: PROCEDURE[lineWidth: CARDINAL ← 75] =
  BEGIN
  n: CARDINAL;
  space ← margin ← lineWidth;
  n ← 3*margin;
  top ← bottom ← 0;
  scanStackEmpty ← TRUE;
  token ← Memory.Get[n*SIZE[Token], n];
  size ← Memory.Get[n*SIZE[INTEGER], n];
  scanStack ← Memory.Get[n*SIZE[CARDINAL], n];
  END;

PrettyPrint: PROCEDURE[tkn: Token] =
  BEGIN
  WITH t: tkn SELECT FROM
    eof ⇒
      BEGIN
      IF ~scanStackEmpty THEN
        BEGIN
        CheckStack[0];
        AdvanceLeft[token[left], size[left]];
        END;
      Indent[0];
      Memory.Free[BASE[token]];
      Memory.Free[BASE[size]];
      Memory.Free[BASE[scanStack]];
      END;
    begin ⇒
      BEGIN
      IF scanStackEmpty THEN
        BEGIN
        leftTotal ← rightTotal ← 1;
        left ← right ← 0;
        END
      ELSE AdvanceRight[ ];
      token[right] ← t;
      size[right] ← −rightTotal;
      ScanPush[right];
      END;
    end ⇒
      BEGIN
      IF scanStackEmpty THEN Print[t, 0]
      ELSE
        BEGIN
        AdvanceRight[ ];
        token[right] ← t;
        size[right] ← − 1;
        ScanPush[right];
        END;
      END;
```

```
    break ⇒
      BEGIN
      IF scanStackEmpty THEN
        BEGIN
        leftTotal ← rightTotal ← 1;
        left ← right ← 0;
        END
      ELSE AdvanceRight[ ];
      CheckStack[0];
      ScanPush[right];
      token[right] ← t;
      size[right] ← −rightTotal;
      rightTotal ← rightTotal + t.blankSpace;
      END;
    string ⇒
      BEGIN
      IF scanStackEmpty THEN Print[t, t.length]
      ELSE
        BEGIN
        AdvanceRight[ ];
        token[right] ← t;
        size[right] ← t.length;
        rightTotal ← rightTotal + t.length;
        CheckStream[ ];
        END;
      END;
    ENDCASE;
  END;

CheckStream: PROCEDURE =
  BEGIN
  IF rightTotal − leftTotal > space THEN
    BEGIN
    IF ~scanStackEmpty THEN
      IF left = scanStack[bottom] THEN
        size[ScanPopBottom[ ]] ← sizeInfinity;
    AdvanceLeft[token[left], size[left]];
    IF ~(left = right) THEN CheckStream[ ];
    END;
  END;

ScanPush: PROCEDURE[x: CARDINAL] =
  BEGIN
  IF scanStackEmpty THEN scanStackEmpty ← FALSE
  ELSE
    BEGIN
    top ← (top + 1) MOD LENGTH[scanStack];
    IF top = bottom THEN ERROR ScanStackFull;
    END;
  scanStack[top] ← x;
  END;

ScanPop: PROCEDURE RETURNS[x: CARDINAL] =
  BEGIN
  IF scanStackEmpty THEN ERROR ScanStackEmpty;
  x ← scanStack[top];
  IF top = bottom THEN scanStackEmpty ← TRUE
```

```
  ELSE top ← (top + LENGTH[scanStack] − 1) MOD LENGTH[scanStack];
  END;

ScanTop: PROCEDURE RETURNS[CARDINAL] =
  BEGIN
  IF scanStackEmpty THEN ERROR ScanStackEmpty;
  RETURN[scanStack[top]]
  END;

ScanPopBottom: PROCEDURE RETURNS[x: CARDINAL] =
  BEGIN
  IF scanStackEmpty THEN ERROR ScanStackEmpty;
  x ← scanStack[bottom];
  IF top = bottom THEN scanStackEmpty ← TRUE
  ELSE bottom ← (bottom + 1) MOD LENGTH[scanStack];
  END;

AdvanceRight: PROCEDURE =
  BEGIN
  right ← (right + 1) MOD LENGTH[scanStack];
  IF right = left THEN ERROR TokenQueueFull;
  END;

AdvanceLeft: PROCEDURE[x: Token, l: INTEGER] = BEGIN
  IF l >= 0 THEN
    BEGIN
    Print[x, l];
    WITH x SELECT FROM
      break ⇒ leftTotal ← leftTotal + blankSpace;
      string ⇒ leftTotal ← leftTotal + l;
      ENDCASE;
    IF left ≠ right THEN BEGIN
      left ← (left + 1) MOD LENGTH[scanStack];
      AdvanceLeft[token[left], size[left]];
      END;
    END;
  END;

CheckStack: PROCEDURE[k: INTEGER] =
  BEGIN
  x: INTEGER;
  IF ~scanStackEmpty THEN
    BEGIN
    x ← ScanTop[ ];
    WITH token[x] SELECT FROM
      begin ⇒
        IF k > 0 THEN
          BEGIN
          size[ScanPop[ ]] ← size[x] + rightTotal;
          CheckStack[k − 1];
          END;
      end ⇒ BEGIN size[ScanPop[ ]] ← 1; CheckStack[k + 1]; END;
      ENDCASE ⇒
        BEGIN
        size[ScanPop[ ]] ← size[x] + rightTotal;
        IF k > 0 THEN CheckStack[k];
        END;
    END;
  END;
```

```
  PrintNewLine: PROCEDURE[amount: CARDINAL] =
    BEGIN
    PutChar[output, CR]; -- output a carriage return
    THROUGH [0..amount) DO PutChar[output, ' ] ENDLOOP;  -- indent
    END;

  Indent: PROCEDURE[amount: CARDINAL] =
    BEGIN
    THROUGH [0..amount) DO PutChar[output, ' ] ENDLOOP;  -- indent
    END;

-- print stack handling
-- We assume push, pop, and top are defined on the stack printStack; printStack is a
-- stack of records; each record contains two fields: the integer "offset" and a flag "break"
-- (which equals "fits" if no breaks are needed (the block fits on the line), or "consistent"
-- or "inconsistent")

  PrintStack: TYPE = POINTER TO PrintStackObject;
  PrintStackObject: TYPE = RECORD[
    index: CARDINAL ← 0,
    length: CARDINAL ← 0,
    items: ARRAY [0..0) OF PrintStackEntry];
  PrintStackEntry: TYPE = RECORD[
    offset: [0..127],
    break: PrintStackBreak];
  PrintStackBreak: TYPE = {fits, inconsistent, consistent};

  Print: PROCEDURE[x: Token, l: INTEGER] =
    BEGIN
    WITH x SELECT FROM
      begin ⇒
        BEGIN
        IF l > space THEN
          Push[[space − offset,
            IF breakType = consistent THEN consistent ELSE inconsistent]]
        ELSE Push[[0, fits]];
      END;
    end ⇒ [ ] ← Pop[ ];
    break ⇒
      BEGIN
      SELECT Top[ ].break FROM
        fits ⇒
          BEGIN
          space ← space − blankSpace;
          Indent[blankSpace];
          END;
        consistent ⇒
          BEGIN
          space ← Top[ ].offset − offset;
          PrintNewLine[margin − space];
          END;
        inconsistent ⇒
          BEGIN
          IF l > space THEN
            BEGIN
            space ← Top[ ].offset − offset;
            PrintNewLine[margin − space];
            END
```

```
          ELSE
            BEGIN
            space ← space − blankSpace;
            Indent[blankSpace];
            END;
          END;
        ENDCASE;
      END;
    string ⇒
      BEGIN
      IF l > space THEN ERROR LineTooLong;
      space ← space − l;
      CharIO.PutString[output, string];
      END;
    ENDCASE ⇒ ERROR;
  END;

END.
```

ACKNOWLEDGMENTS

I am indebted to Philip Karlton, Don Knuth, Jim Morris, Greg Nelson, Wolf Polak, Ed Satterthwaite, Dick Waters, and Steve Wood for many stimulating conversations on prettyprinting.

REFERENCES

1. GESCHKE, C., MORRIS, J., AND SATTERTHWAITE, E. Early experience with Mesa. *Commun. ACM 20,* 8 (Aug. 1977), 540–553.
2. GOLDSTEIN, I. Pretty-printing, converting list to linear structure. Artificial Intelligence Laboratory Memo. No. 279, M.I.T., Cambridge, Mass., 1973.
3. HEARN, A.C., AND NORMAN, A.C. A one-pass prettyprinter. Rep. UUCS-79-112, University of Utah, Salt Lake City, Utah, 1979.
4. KNUTH, D.E. Tau epsilon chi—A system for technical text. Rep. STAN-CS-78-675, Computer Science Dep., Stanford Univ., Stanford, Calif., 1978.

BIBLIOGRAPHY

CONROW, K., AND SMITH, R.G. NEATER2: A PL/I source program reformatter. *Commun. ACM 13,* 11 (Nov. 1970), 669–675.

DONZEAU-GOUGE, V., HUET, G., KAHN, G., LANG, R., AND LEVY, J.J. A structure-oriented program editor; A first step towards computer assisted programming. Proc. Inter. Computing Symp., Antibes, 1975.

GORDON, H. Paragraphing computer programs. M.Sc. Thesis, Univ. of Toronto, Toronto, Ontario, Canada, 1975.

HEURAS, J., AND LEDGARD, H. An automatic formatting program for Pascal. *SIGPLAN Notices 12* (1977), 82–84.

HUET, G., KAHN, G., AND LANG, B. The MENTOR Program Manipulation System. Unpublished manuscript, 1978.

JOY, W. Berkeley Pascal PXP implementation notes. Manuscript, Dep. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif., 1979.

KNUTH, D.E. BLAISE—A TEX preprocessor for Pascal. To appear.

MCKEEMAN, W. Algorithm 268, Algol-60 reference language editor [R2]. *Commun. ACM 8,* 11 (Nov. 1965), 667–669.

MOHILNER, P.R. Prettyprinting PASCAL programs. *SIGPLAN Notices 13,* 7 (1978), 34–40.

PETERSON, J.L. On the formatting of Pascal programs. *SIGPLAN Notices 12,* 12 (1977), 83–86.

SCOWEN, R., ALLIN, D., HILLMAN, A., AND SHINELL, M. SOAP—A program which documents and edits Algol60 programs. *Comput. J. 14,* 2 (1971), 133–135.

TEITLEBAUM, T. The Cornell program synthesizer. Tech. Rep. 79-370, Dep. of Computer Science, Cornell Univ., 1979.

Received September 1979; revised May and June 1980; accepted July 1980