Stanford Verification Group
Report No. 13

Computer Science Department
Report No. STAN-CS-79-770

October 1979

PRETTY PRINTING

by

Derek C. Oppen

Research sponsored by

National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Verification Group
Report No. 13

Computer Science Department
Report No. STAN-CS-79-770

October 1979

PRETTY PRINTING

by

Derek C. Oppen

ABSTRACT

An algorithm for pretty printing is given. For an input stream of length n and an output device with margin width 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 linefull of input. The algorithm is easily implemented.

This research was supported by the National Science Foundation under Contract NSF MCS 78-02835. The views and conclusions contained in this document ate those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

Pretty Printing

Derek C. Oppen
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

An algorithm for pretty printing is given. For an input stream of length n and an output device with margin width 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 linefull of input. The algorithm is easily implemented.

1. Introduction

Althqugh the art of parsing is a well-researched area, its dual — "unparsing" and "pretty printing" — has not received like attention. A pretty printer 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 margin width is 40, we might want it printed as follows:

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

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

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

This research was supported by the National Science Foundation under contract MCS 78-02835.

But under no circumstances do we want to see

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

Pretty printers are common components of Lisp environments, where trees or s-expressions are data objects which are interactively manipulated and which 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 pretty printed, (See [Goldstein 1973] or [Hearn and Norman 1979) for descriptions of some pretty printers for Lisp.)

Pretty printers have generally not been very common for block-structured languages, perhaps because, until recently, "programming environments" for such languages did not exist. (See [McKeeman 1965], [Hueras and Ledgard 1977], [Huet et al 1978] or [Hearn and Norman 1979] for descriptions of some implemented pretty printers.) Happily, this situation is fast changing. Pretty printers are integral components of any programming environment tool. Editors, for example, for block-structured languages benefit enormously from a pretty printer — 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 pretty printers to print out error messages in which program text is displayed; this would make the error much more understandable. Pretty printers are useful in any system which prints or displays messages or other output to the user.

Pretty printers have traditionally been implemented by rather ad hoc pieces of code directed towards specific languages. We will instead give a language-independent pretty printing 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 [Knuth 1979]) 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 will not discuss in detail the question of how to interface the pretty printer described here with any specific language. In general, the pretty printer 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 to handle questions such as whether blanks are redundant. We shall describe in section 6 two approaches we have taken to implementing a preprocessor *for* pretty printing.

2. Basic Notions

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

A pretty printer expects as input a stream of characters. A character may be a printable character such as "a" or "3" or "&" 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 pretty printer may break a line between strings but not within a string.

We will differentiate between several types of delimiters. The first type of delimiter is the blank (carriage returns, formfeeds and linefeeds arc treated as blanks). The next two types correspond to special starting and ending delimiters for logically-contiguous blocks of strings. We will denote the delimiters $[\![$ and $]\!]$ respectively. The algorithm will try to break onto different lines as few blocks as possible. For instance, suppose we wish to print out $f(a, b, c, d) + g(a, b, c, d)$ on a display which is only 20 characters wide. We might want this printed as

$$\begin{array}{l} f(a,b,c,d) \\ \quad +g(a,b,c,d) \end{array}$$

or as

$$\begin{array}{l} f(a,b,c,d)+ \\ \quad g(a,b,c,d) \end{array}$$

but definitely not as

$$\begin{array}{ll} f(a,b,c,d)+g(& a, \\ & b, \\ & c, \\ & d) \end{array}$$

We can avoid this 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 doesn't recognize that parentheses are delimiters and thus that $g(a, b, c, d)$ shouldn't be broken if possible. But the pretty printing algorithm given here is a general purpose algorithm providing primitives for pretty printing, and is not tailored to any particular language. The example could have been written just as easily with two **begin** . . . **end** blocks.)

We will later allow refinements to the above set of delimiters, but for the moment we will 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 <blank> s_2 <blank> . . . <blank> s_k⟧ is a stream.

As we shall see later, this definition of an "allowable" stream is a little too restrictive in practice, but makes describing the basic algorithm easier. 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 linewidth 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 pretty printed. $Scan()$ 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.

$Print()$ uses the length information associated with each token 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 At 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).

Print() is the simpler routine so we describe it first. It uses auxillary functions *Output*(x), which prints x on the output device, and Indent(z), 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 margin width, and a variable space 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 3
        cases S : empty ⇒ Print(x, length(x));
                otherwirc ⇒
                    right := right + 1;
                    stream[right] := x;
                    size[right] := length(x);
                    rightotal := rightotal + length(x);
```

Scan() uses the stack to keep track of occurrences of delimiters. 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), and otherwise 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.

Scan() has the nice property that it requires time linear in the length of the stream (as does *Print*()). It has the undesirable property that it also requires space linear in the length of the stream. For 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 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 the linewidth 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 fhe 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*() needn't 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. So, our buffer size can be linear in m, and we never need look ahead more than 3m tokens before being able to print *something.*

And 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. So we don't 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 a close relative 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 auxillary function *Advanceleft*(). We implement *stream* and size as two arrays of size *arraysize*, a constant equal to $3m$, say. The variables *left* and *right* arc initially 1, pointing to the start of the arrays,

```
$Scan()$ : **local** $x$;
**forever** $x := Receive()$;
  **cases**
    $x$ : eof ⇒ halt;
    $x$ : ⟦ ⇒
      **cases** S : empty ⇒ $left := right := lefttotal := righttotal := 1$;
              **otherwise** ⇒ $right :=$ **if** right = $arraysize$ then 1 **else** $right + 1$;
      $stream[right] := x$;
      $size[right] := -righttotal$;
      $Push(S, right)$;
    $x$ : ⟧ ⇒
      **cases** S : **empty** 3 $print(x, 0)$;
              otherwire ⇒
                  $right :=$ if $right = arraysize$ then 1 elre $right + 1$;
                  $stream[right] := x$;
                  $size[right] := 0$;
                  $x := Pop(S)$;
                  $size[x] := righttotal + size[x]$;
                  **if** $stream[x]$ : blank and $\neg S$ : empty
                      **then** $x :=$ Pop(S); $size[x] := righttotal + size[x]$;
                  if S : empty **then** $Advanceleft(stream[left], size[left])$;
    $x$ : blank ⇒
      **cases** S : empty ⇒ $left := right := righttotal := 1$;
              otherwire ⇒
                  $right :=$ if $right = arraysize$ then 1 **else** $right + 1$;
                  $x :=$ Top(S);
                  **if** $stream[x]$ : blank then $size[Pop(S)] := righttotal + size[x]$;
      $stream[right] := x$;
      $size[right] := -righttotal$;
      $Push(S, right)$;
      $righttotal := righttotal + 1$;
    $x$ : **string** ⇒
      **cases** S : **empty** ⇒ $Print(x, length(x))$;
              otherwise ⇒
                  $right :=$ if $right = arraysize$ then 1 **else** $right + 1$;
                  $stream[right] := x$;
                  $size[right] := length(x)$;
                  $righttotal := righttotal + length(x)$;
                  while $righttotal - lefttotal > space$ do
                     $size[Popbottom()] :=$ 999999;
                     $Advanceleft( stream[left], size[left]$;
Advanceleft(x, l):
if $l \geq 0$ then
   Print(x, l);
   cases x : blank ⇒ lefttotal := lefttotal + 1;
         x : string ⇒ lefttotal := lefttotal + 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 = arraysite 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 appendix A.

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 linewidth 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 sub-block 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 linewidth 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 pretty printer is used as the output device for an unparser. Consider the following stream for printing out f (g(x, y)) (<blank> denotes a blank):

```
    [[ f ( [[ g(x, <blank> y ) ]] <blank> ) ]]
```

This may result in the following output:

```
    f(g(x, y)
     )
```

given appropriate margin 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 linebreak from occurring between the right parentheses by sending the stream:

```
    [[ f ( [[ g(x, <blank> 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 appendix 1 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 Pretty Printing

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

The simplest way is to drive the pretty printer 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

<term> → <subterm> <operator> <subterm>

the parser may generate, when parsing a + 6, the subtree consisting of a node with three successors: the subtrees corresponding to a, + and 6. 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 + 6, 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 6, and finally generate a closing ⟧.

Driving the pretty printer 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 pretty printing is no longer online: the whole program must be parsed before pretty printing 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 such questions 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 pretty printer 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 pretty printer 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 pretty printer 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 pretty printer can often conveniently be represented directly in the grammar; for instance, in our example production above:

<term> → ⟦<subterm> <blank> <operator> <blank> <subterm>⟧

Suppose we are using a parser generator (to generate a table driven parser). We modify the grammar of the language to contain pretty printing 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 pretty printer. The other semantic routines transmit to the pretty printer 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 pretty printer in the correct order. For instance, consider the production:

<block > → begin <statementlist> 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:

<block > → <begin> < statementlist > end
<begin> → 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 pretty printing 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 pretty printer directly. For instance, the semantic routine corresponding to the nonterminal <begin> above could emit the three tokens ⟦, "begin" and <blank >.)

A pretty printer for Mesa has been implemented in this fashion by Philip Karlton and the author.

7. Other Pretty Printers.

As mentioned in the introduction, pretty printers are common in Lisp environments and therefore have been fairly widely implemented, but rarely analyzed. The following is a list of those algorithms known to the author; the list has been growing and is undoubtedly incomplete. With a few exceptions, the analyses given below are the author's. As before, n denotes the length of the input stream and m denotes the linewidth of the output device.

Goldstein [1973] describes various ways of implementing pretty printers for Lisp, and gives several algorithms requiring O(n) time and O(n) space. Whit Diffle (private communication) has an algorithm for Lisp pretty printing which uses the notion of variable glue to put together boxes of text. Mentor, a structure-oriented editor for Pascal, contains a pretty printer for Pascal ([Donzeau-Gouge et al 1975], [Huet et al 1978]). Dick Waters (private communication) independently discovered the observations given here on how much lookahead is required; he has implemented a pretty printer for Lisp which requires $O(mn)$ time and O(m) space. Hueras and Ledgard [1977] describe a formatting program for Pascal; their program appears to require O(n) time and space. Greg Nelson (private communication) has a pretty-printing 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, Tony Hearn and A. C. Norman [1979] have independently discovered a similar method; their description is informal and their analysis assumes that linewidth is constant, but if margin width is assumed to be m, their algorithm appears to have the same bounds as Morris' algorithm. Don Knuth (unpublished memorandum) has written a pre-processor Blaise for Pascal programs which pretty prints them using his text processor TEX.

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 5 := j(z)
              else x := Q(Z);
```

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 reduce the indentation just enough to right justify the offending string.

This illustrates a general drawback of the algorithm — it does only constant space (one linewidth) lookahead and its logic is not as sophisticated as it might be.

But hopefully 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 linefull of input, and printing never lags more than a linefull behind the input routine. This we consider an inportant 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.

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 pretty printing. In particular, I collaborated with Philip Karlton, Wolf Polak and Steve Wood on three different pretty printers.

References

[Donzeau-Gouge et al 1975] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J. J. Levy, A structure-oriented program editor: a first step towards computer assisted programming, Proceedings of International Computing Symposium, Antibes.

[Goldstein 1973] I. Goldstein, Pretty-printing, Converting List to Linear Structure, MIT A. I. Lab memo No. 279.

[Hearn and Norman 1979] A. C. Hearn and A. C. Norman, A One-Pass Prettyprinter, University of Utah Report UUCS-79-112.

[Hueras and Lcdgard 1977] J. Hcuras and H. Lcdgard, An Automatic Formatting Program for Pascal, Sigplan Notices 12, pp. 82-84.

[Huet et al 1978] G. Huet, G. Kahn, B. Lang, The MENTOR Program Manipulation System, Unpublished manuscript.

[Knuth 1978] D. E. Knuth, Tau Epsilon Chi — A System for Technical Text, Report STAN-CS-78-675, Computer Science Department, Stanford University.

Appendix

The following is the augmented pretty printing algorithm implemented by Philip Karlton and the author in Mesa (some details have bccn left out concerning input/output and memory allocation). Comments are prcccdcd by two dashes; numbers are either in octal or in binary (if followed by b).

The pretty printer 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 pretty printer 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 assocated 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 linebreak (by setting blankSpace to be a very large integer).

```
PrettyPrint: DEFINITIONS =
  BEGIN
-- typos
  TokenType: TYPE = {string, break, begin, end, eof};
  Tokon: TYPE = RECORD[
    SELECT type: TokenType FROM
      string => [string: string],
      break => [
        blankSpace: [0..MaxBlanks] ← 1,    -- number of spaces per blank
        offset: [0..31] ← 03,              -- Indent for overflow lines
      begin => [
        offset: [0..127] ← 2,              -- indent for this group
        breakType: Break8 t inconsistent], -- default ''inconsistent''
      and => NULL,
      oof => NULL,
      ENDCASE];

  MaxBlanks: CARDINAL = 127;
  Breaks: TYPE = {consistent, inconsistent};
  LineBreak: break Token = [break[blankSpace: MaxBlanks]];
  END.
PrettyPrinter: PROGRAM
  EXPORT6 PrettyPrint=
  BEGIN
  margin, space: INTEGER;
  left, right: INTEGER;
  token: DESCRIPTOR FOR ARRAY OF Token ← DESCRIPTORCNIL, 0];
  size: DESCRIPTOR FOR ARRAY OF INTEGER ← DESCRIPTOR[NIL, 0];
  leftTotal, rightTotal: INTEGER;
  sizeInfinity: INTEGER = 777778;
  scanStack: DESCRIPTOR FOR ARRAY OF INTEGER ← DESCRIPTORCNIL, 0];
  scanStackEmpty: BOOLEAN;
  top, bottom: CARDINAL;
  printStack: PrintStack ← CreatePrintStack[63];

  PrettyPrintInit: PROCEDURE[lineWidth: CARDINAL ← 75] =
    BEGIN
    n: CARDINAL;
    space ← margin ← lineWidth;
    n t 3*margin;
    top t bottom ← 0;
    scanStackEmpty← TRUE;
    token ← Memory.Get[n*SIZE[Token], n];
    size ← Memory.Get[n*SIZE[INTEGER], n];
    scanStack t 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[loft],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 t rightTotal ← 1;
          left t right t 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 t;
          size[right] ← -1;
          ScanPush[right];
          END;
        END;
      break =>
        BEGIN
        IF scanStackEmpty THEN
          BEGIN
          leftTotal t rightTotal t 1;
          left t right ← 0;
          END
        ELSE AdvanceRight[];
        CheckStack[0];
        ScanPush[right];
        token[right] t t;
        size[right]← -rightTotal;
        rightTotal← rightTotal + t.blankSpace;
        END;
      string =>
        BEGIN
        IF scanStackEmpty THEN Print[t, t.length]
        ELSE
          BEGIN
          AdvanceRight[];
          token[right] t t;
          size[right] t t. length;
          rightTotal ← rightTotal + t.length;
          CheckStream[];
          END;
        END;
     ENDCASE;
  END;

 CheckStream: PROCEDURE =
BEGIN ·
· IF  rightTotal - leftTotal > space THEN
    BEGIN
    IF -8canStackEmpty THEN
      IF loft = scanStack[bottom] THEN
        size[ScanPopBottom[]] ← 900;
    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] t  x ;
  END;

ScanPop  :  PROCEDURE  RETURN6 [x : CARDINAL] =
  BEGIN
  IF scanStackEmpty THEN ERROR ScanStackEmpty;
  x ← scanStack[top];
  IF top = bottom THEN scanStackEmpty t 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 t scanStack[bottom];
  IF top = bottom THEN scanStackEmpty t 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, 1: INTEGER] = BEGIN
  IF 1 >= 0 THEN
    BEGIN
    Print[x, 1];
    WITH x SELECT FROM
      break => leftTotal t leftTotal + blankSpace;
      string => leftTotal ← leftTotal + 1;
      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[]] t size[x] + rightTotal;
           CheckStack[k  - 11;
           END;
       end => BEGIN size[ScanPop[]] + 1; CheckStack[k + 1]; END;
       ENDCASE =>
         BEGIN
         size[ScanPop []] t 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 [O..amount) DO PutChar[output,.I ENDLOOP; -- indent
   END;

 Indent: PROCEDURE[amount: CARDINAL] =
   BEGIN
   THROUGH [O..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 equal8 "fits"
-- if no break8 are needed (the block fits on the line), or
-- "coneietent" or ''inconsistent'')

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

 Print: PROCEDURE[x: Token, 1: INTEGER1 =
   BEGIN
```

```
      WITH x SELECT FROM
        begin =>
          BEGIN
          IF 1 > space THEN
            Push[[space-offset,
              IF breakTypo = consistent THEN conelstent ELSE inconsistent]]
          ELSE Push[[0, fits]];
          END;
        end => Cl ← Pop[];
        break =>
          BEGIN
          SELECT Top[].break FROM
            fit8 =>
              BEGIN
              space t space-blankspace;
              Indent[blankSpace];
              END;
            conelstent =>
              BEGIN
              epace t Top[] .offset - offset;
              PrintNewLine[margin-space];
              END;
            inconsistent =>
              BEGIN
              IF 1 > epace THEN
                BEGIN
                space ← Top[].offset - offset;
                PrintNewLine[margin-space];
                END
              ELSE
                BEGIN
                space ← space-blankSpace;
                Indent[blankSpace];
                END;
              END;
            ENDCASE;
          END;
        string =>
          BEGIN
          IF 1 > epace THEN ERROR LineTooLong;
          space t epace - 1;
          CharIO.PutString[output, string];
          END;
        ENDCASE => ERROR;
      END;

    END.
```