Unparsing Continued

CGN8 concluded with two pages about unparsing by means of general-purpose "formatting streams". This note carries this idea one step further. CGN8 and CGN9 should be sufficient introduction for a Cedar programmer to use the formatting routines accessible via

[indigo]<juno>parser>unparserbuffer.mesa

By a <u>code</u> I mean any one of

(1) an ascii character code
(2) either of the two special codes <u>setb</u> and <u>endb</u>
(3) any one of an infinite set of codes called <u>breakpoint codes</u>.

The input to the algorithm described in this note is a sequence of codes; the output is a sequence of ascii character codes. The output duplicates the input except that the setb and endb codes have been removed, and some of the breakpoint codes have been replaced by newline-blank* sequences. (A newline-blank* sequence consists of a newline followed by some number of blanks.) The breakpoint codes that are not replaced by newline-blank* sequences are removed.

A sequence of codes is <u>properly</u> <u>nested</u> if none of its prefixes contains more endb's than setb's. A sequence of codes is a <u>group</u> if it is properly nested, begins with a setb, ends with an endb, and has no proper prefix that satisfies these conditions. A code sequence x is a <u>subgroup</u> of a code sequence y if x is a subsequence of y and x is a group. The subgroups of a properly nested code sequence form a ~~fore~~ forest under the containment relation. (A forest is a disjoint union of trees.) The formatting algorithm obeys the break-from-the-root rule on the trees of this forest. (See CGN8.) More precisely, let us say that a breakpoint code in the input is <u>broken</u> if it is replaced in the output by a newline-blank* sequence, and say that a group in the input is broken if it contains a breakpoint code that is broken and is not contained in any proper subgroup. Then the break-from-the root rule says that no unbroken group contains a broken subgroup.

With each breakpoint code bp we assume there is an associated integer off(bp), the <u>offset</u> of bp, which determines the number of blanks in the newline-blank* sequence that will replace the breakpoint if it is broken. The determining rule is as follows. Consider a group consisting of the sequence of codes setb s_1 bp s_2 endb, where bp is a breakpoint code and s_1 and s_2 are properly-nested code sequences. The output will contain a character for each character in s_1; let c be the output character corresponding to the first character in s_1. Every character of the output occurs in some column — column zero if it immediatly follows a newline code; otherwise a column with a positive index. Let i be the index of the output column of c. Then the index of the output column of the first character of s_2 is $i + off(bp)$. Note that this index is the number of blanks in the newline-blank* sequence that replaces bp.

In other words, off(bp) is the amount to increase the current indentation if bp is selected for a line break. Zero offsets lead to equally-indented lines; positive offsets increase indentation; negative offsets decrease indentation. Note that the offset does not determine the relative indentation between this line and the previous line, but between this line and the first character of the containing group.

Each breakpoint code is either <u>united</u> or <u>ununited</u>. If any breakpoint of a group is broken, then all the group's united breakpoints break in unison. An ununited breakpoint breaks only if the characters between it and the next breakpoint (or the end of the group, if it is the last breakpoint of its group) cannot be fit within the margin. For example, to format a paragraph of text, we put ununited breakpoints between the words, so that the formatter will pack as many words to a line as fit. To format a BEGIN-END block, we use united breakpoints with offset 2 before each interior statement, and a united breakpoint with offset zero before the closing END. This produces the multi-line format

```
BEGIN
  Statement 1;
  Statement 2;
     :
  Statement n
END
```

I have been as precise as I know how to be with words. To be more precise, I will write the rules as a program, using the exception-handling technique of CGN8. The treatment of united breakpoints requires an additional wrinkle.

We use the procedures Read and Write, defined by

$$c : \text{Read} \equiv inp := inp + 1 \; ; \; c := in(inp)$$

$$
\begin{array}{ll}
 & Write(c) \\
\equiv & out(outp) := c \\
; & outp := outp + 1 \\
; & \underline{if}\ c = newline \rightarrow indent := 0 \\
 & \ \ [\!]\ c \neq newline \rightarrow indent := indent + 1\ \underline{fi} \\
; & \underline{if}\ indent \leq margin \rightarrow skip\ \underline{fi}
\end{array}
$$

This will abort if writing c overflows the margin.

For any code x, we define

$$
\begin{array}{rcl}
Char(x) & \equiv & x \text{ is an ascii character code} \\
Bp(x) & \equiv & x \text{ is a breakpoint code} \\
off(x) = n & \equiv & x \text{ is a breakpoint code with offset } n \\
United(x) & \equiv & x \text{ is a united breakpoint code.}
\end{array}
$$

The procedure P1 is used to translate a group using one-line format. It moves the input through exactly one group; that is

$$\{in(inp) = setb\}\ P1\ \{in(inp) = \text{the matching endb}\}$$

It has the side effect of writing all the characters of P1 to the output. If this would overflow the margin, P1 aborts. The code for P1 is:

```
      P1
  ≡  c: Read
  ;  do Char(c) → Write(c); c: Read
     ▯  Bp(c)  →  c: Read
     ▯  c = setb → P1; c: Read od
```

Thus P1 write characters, ignores breakpoints, calls itself recursively to print subgroups, and halts when c = the endb that matched the initial setb.

Procedure P2 is similar to P1, in that it uses one-line output format, but instead of printing the text between a setb and its matching endb, it prints the text between a breakpoint and the next breakpoint of the same group, or the end of the group, whichever comes first:

```
      P2
  ≡  c: Read
  ;  do Char(c) → Write(c); c: Read
     ▯  c = setb → P1; c: Read od
```

Eventually we will write a procedure PP that prints a group according to the break-from-the-root rule, just as P1 outputs the group in one-line format. Assuming the availability of PP, we can code P3, which prints a single group in multi-line format.

```
    P3
≡  let n : n = indent
   in  c : Read
   ;  do  Char(c) → Write(c); c : Read
       []  c = setb → PP; c : Read
       []  United(c) → Break(n+off(c)); c : Read
       []  Bp(c) ∧ ¬United(c)
       →  let m1, m2, m3
            : m1 = inp ∧ m2 = outp ∧ m3 = indent
          in  P2
          err  T
          →  inp, outp, indent := m1, m2, m3
           ;  Break(n + off(in(inp)))
           ;  c : Read
          end
          end
      od
  end
```

Comments about the notation. I am writing

$$\underline{\text{let}}\ v : P\ \underline{\text{in}}\ S\ \underline{\text{end}}$$

where v is a variable or list of variables, P is a predicate, and S is a command, as a more palatable version of the formally-equivalent

$$\underline{\text{if}}\ v : P \rightarrow S\ \underline{\text{fi}},$$

which introduces local variables v satisfying the predicate P, executes S, and then removes the local variables.

Thus P3 saves the indentation in the local variable n; this is necessary since all subsequent breakpoints will produce an indentation that is relative to n. Thereafter P3 writes characters, recursively pretty-prints subgroups, breaks united breakpoints, and treats ununited breakpoints with care as follows. To process an ununited breakpoint, P3 saves inp, outp, and indent in three local variables and calls P2, attempting to print everything up to the next breakpoint in one-line format. If this succeeds, P3 continues; otherwise the error trap "err $T \rightarrow$" gets control, the variables inp, outp, and indent are restored, the breakpoint is replaced by an appropriate newline–blank* combination, and P3 continues.

In CGN8 as well as here, I am writing

$$A \ \underline{\text{err}} \ T \rightarrow B \ \underline{\text{end}}$$

to indicate the command: execute A, then execute B if and only if A aborted. Since I have already come to dislike this notation, and found another that serves the same purpose and is more to my taste, I won't explain it further.

It remains to code Break and PP. We have

$$\begin{aligned} & \text{Break}(n) \\ \equiv \ & \text{Write}(\text{newline}) \\ ; \ & \underline{\text{for}} \ i : i \in [0, n) \rightarrow \text{Write}(\text{blank}) \ \underline{\text{end}} \end{aligned}$$

This writes a newline followed by n blanks.

The procedure PP prints a group in either single- or multi-line format. First it tries single-line format by calling P1; if this aborts, it uses multi-line format by calling P3. Naturally the important state variables must be saved and restored:

PP

$\equiv$ <u>let</u> $m1, m2, m3 : m1 = inp \wedge m2 = outp \wedge m3 = indent$
<u>in</u> $P1$ <u>err</u> $T \rightarrow inp, outp, indent := m1, m2, m3 ; P3$ <u>end</u>
<u>end</u>

Finally we can write a loop that transduces and infinite stream of input codes into an infinite stream of output characters. We need only decide on the interpretation of breakpoint codes that are not contained within groups. This is most easily finessed by assuming the existence of an initial setb with no matching endb. With this assumption, a call to PP will transduce the input code stream to an appropriately formatted output stream.

These programs have ignored one important aspect of reality: that input and output are bufferred. Because we have used only two formats per node, a margin overflow will back up the input over no more than <u>margin</u> input ascii characters; and in practice this will not require backing the input up over more than, say, 2 * margin codes. (In the worst case there is no reason that arbitrarily many setb and breakpoint codes may not exhaust any finite buffer, but that doesn't seem like it would be a problem in practice.) Therefore, it seems natural to buffer input and output in FIFO queues somewhat larger than one line. It strikes me as an interesting problem to modify the code above to handle this extra constraint, but I haven't time for it now.

My actual Cedar program is not coded like the one above at all: I didn't think of it soon enough. It has no recursion or error-traps: the state stored in the program counter and recursion stack of the algorithm described in this note is stored in explicit program variables of my Cedar program. For a description of a program very similar to my Cedar code, see Derek Oppen's paper "Prettyprinting", ACM Trans. Program. Lang. Syst. 2, 4 (Oct. 1980), 465-83.

— Greg Nelson, 6 Dec 83

Errors in CGN9

page 2 line 6: "subsequence" → "substring"

page 2 line 4: "ends with an endb," → "ends with an endb, has the same number of setbs and endbs,"

page 4 line −3: "united" → "ununited".