Unparsing

Let L be a context-free language and P a parser for L: for any string σ in L, $P(\sigma)$ is a parse tree. An unparser for L is a map UP from parse trees to strings that satisfies

(1) For any string σ in L, $P(UP(P(\sigma))) = P(\sigma)$

(2) The strings produced by UP are nicely formatted and easy to read.

Unparsers are often called prettyprinters, formatters, and (in the MACLISP community) grinders.

The implementor of an unparser is faced with many difficulties. There are first of all many practical difficulties such as integrating the unparser with the text editor, handling comments, highlighting in the output designated subtrees of the tree being unparsed (for example, to identify the location of an error), and last but not least, choosing a precise meaning for (2) that will satisfy more than one programmer. I will not discuss these practical difficulties in this note. Suppose them solved: then what remains is a difficulty of another sort: a pure

programming problem. Some kind of backtracking search will be required to choose a readable format that fits within the margin. The details of this programming problem depend on the precise version of (2) that is chosen, but for the reasonable choices that I know of, the programming problem is far from trivial. The purpose of this note is to record one solution to this problem, which I find interesting if for no other reason that it is the only program that I can recall writing that uses "exception-handling" in a non-trivial way.

We suppose that every node of the parse tree can be printed in one of two formats, which we call the one-line format and the multi-line format. The one-line format is produced by recursively printing the node in the obvious way, without breaking the output into lines. We take the view that the one-line format is always preferable to the multi-line format, but may be unfeasible since for nodes near the root of the parse tree, the one-line format will generally not fit between the margins. Informally, a multi-line format is specified by drawing a picture of the output, using boxes to designate the unparsings of the subnodes. For example, here is one of several plausible multi-line

output formats for infix operators:

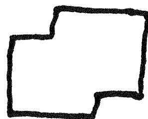
```
        A
    op  B                    (3)
```

Many people prefer to place the operator at the end of the first line rather than the beginning of the second, and many who put the operator on the second line prefer to indent it to the right of A rather than to the left of A. However, format (3) has some advantages, and it is the one I generally use.

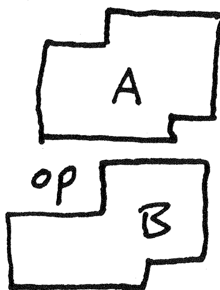
If a node is assigned one-line format, then all its descendents must also be assigned one-line formats. This is called the "break from the root rule": to break a node means to assign it the multi-line format. According to this rule, if the whole parse tree fits on one line, we print it on one line; if not, we break the root node and continue recursively with the children of the root.

It is important to keep in mind that in format (3), A and B may span several lines. Also, we have not yet made precise how format (3) is to fit into the layout of the parent of itself. To do this I will

fall back on a picture. The unparsed text of a broken node will be a sequence of lines, not necessarily beginning or ending with a newline character. The first character of the first line will be added to the previous output, and will not generally go into the ~~right margin~~ leftmost column. Thus it is convenient to draw pictures like



to indicate the unparsed text of a broken node. With this notation, format rule ~~IE~~ (3) can be drawn for broken nodes A, B as follows:



That is, the first character of A is printed into the output buffer; its indentation is determined by the context of the node. After A is unparsed, a newline is printed, followed by enough blanks so that printing the infix operator followed by a blank will restore the

indentation that was current when A was unparsed. Finally, B is unparsed.

For example, here is Euclid's algorithm expressed as a guarded command and unparsed according to the above scheme, first to a half-page margin, then a quarter-page margin, then an eighth-page margin:

```
do n<m → m := m-n
 □ m<n → n := n-m
od
```

```
do n<m
→ m:=m-n
□ m<n
→ n:=n-m
od
```

```
do n
 < m
 → m
 := m
 - n
 □ m
 < n
 → n
 := n
 - m
od
```

(Assuming the multi-line format of do-od is

```
do A
od
```
)

We would never use a format like

```
do n<m → m
           := m - n
[] m<n → n
           := n - m
od
```

because it violates the break-from-the-root rule: the nodes of the form ... → ... were printed in one-line format, while their subnodes of the form ... := ... were printed in multi-line format.

Now let's code it. Characters will be sent to the output via the procedure Write, defined by

```
   out, outp : Write (c)
≡  out (outp) := c
;  outp := outp + 1
;  if c = newline → indent := 0
   [] c ≠ newline → indent := indent + 1 fi
;  if indent > margin → Abort
   [] indent ≤ margin → Skip fi
```

This procedure maintains the following invariant:

(4) The sequence of characters that have been written is exactly out(i) for $i \in [0, outp)$, and if this sequence were printed, no line would contain more than margin characters, and the last line would contain indent characters.

I will let "Write(c)" denote a call to this procedure, although sometimes I think "out, outp, indent: Write(c)" would be better style. (It was an error that I omitted "indent" from the list of var parameters on line -7 of page 6.)

Note that Write(c) aborts if (4) cannot be maintained because of margin overflow. Since if-fi aborts if none of its guards is true, I could have omitted the first guarded command (not the second!), but I wrote it by accident and didn't want to cross it out. I should have just written if indent ≤ margin → Skip fi.

The specification for the unparser is that in any state in which (4) is true, the call UP(A), where A is a parse tree, will maintain (4) and

extend the output with the unparsed version of A. We can imagine it implemented like this:

UP(A)
≡ **if** Oracle(A, indent) = oneline → UP1(A)
▯ Oracle(A, indent) = manyline → UP2(A) **fi**

that is we decide on which format to use by consulting an oracle. I will try to be precise about the specifications of UP1 and UP2. We have:

{(4)}
UP1(A)
{(4) ∧ the sequence out(i) for $i \in [outp_0, outp)$ is the one-line format unparsed version of A. UP1 aborts only if it is impossible to meet this spec; i.e. if the one-line format would overflow the margin}

The coding of UP1 will be very straightforward, since it can call itself recursively to unparse the subnodes of its argument.

There are no surprises in the specs for UP2:

{ (4) }
UP2(A)
{ (4) $\wedge$ the sequence out(i) for $i \in [out_0, outp)$
is the multi-line format unparsed version of A.
UP2 aborts only if it is impossible to meet this spec }

The coding of UP2 will reflect the multi-line format rules. Since the format of the children of a broken node may be either 1-line or multi-line, the recursive calls that UP2 makes will be to UP, not to UP1 or UP2. For example, suppose that (3) is the format for broken infix operators, and that the node A is for B op C, and (to simplify things) that op is a single character. Then UP2(A) is equivalent to

```
      temp := indent
    ; UP(B)
    ; Write(newline)
    ; for i : i ∈ [0, temp-2) → Write(blank) end
    ; Write(op)
    ; Write(blank)
    ; UP(C)
```

This code saves the current indentation in temp, unparses the left argument, advances to a newline, indents sufficiently that after printing op and a blank, the saved indentation will be restored, and finally unparses the right argument.

It remains to code Oracle. For any fixed set of formatting rules, this will not be difficult, but it does not appeal to me to have to change the code for Oracle everytime the code for UP1 or UP2 is changed. Furthermore, Oracle is logically redundant, since if many-line format is called for, we will discover this when UP1 aborts. Thus instead of coding Oracle we replace the code for UP by

~~UP(A)~~
~~≡ if a, b, c : [in~~

```
   UP(A)
≡  if a, b : (outp, indent) = (a, b)
   → UP1(A)
   err T → outp, indent := a, b ; UP2(A) end
   fi
```

That is, UP saves the output pointer and indentation in a and b, which are local variables for the current invocation of UP. Then UP calls UP1. If UP1 returns normally, thats the end of it. But if an error occurs, (and we assume that the only errors are the Aborts inside of the procedure Write), UP restores the output to what it was before the call to UP1, restores the indentation as well, and calls UP2.

That's all there is to this program.

Notice that each time UP1 aborts, a node is changed from single-line format to multi-line format. Thus the total number of aborts from UP1 is bounded by the final number of multi-line formats, which can generally be bounded by the final number of lines. The cost of an abort from UP1 is the time required to overflow the margin, which is proportional to the margin itself. Thus the time required to unparse a tree is O(margin · #lines printed); for practical languages this will be of the same order of magnitude as the number of characters printed.

For many languages, satisfactory unparsing cannot be obtained with just two possible node formats. For example, here are the three formats used by the MACLISP grinder for a short COND:

```
(cond (p a) (q b))              one-line

(cond (p a)                     standard
      (q b))                    multi-line

(cond
  (p a)                         miser-mode
  (q b))
```

Miser-mode is so-called because it is a miser about indentation: the standard multi-line format squanders room by indenting too much. If we used only the one-line and standard multi-line formats, too many programs would be impossible to fit between the margins. If we used only the one-line and miser-mode formats, programmers would complain that the results were ugly. The solution is to generalize the method described above to three-formats per node; it is easy to see how to do this. We just modify UP

to try three formats instead of 2. The timing analysis used for the 2-format case is invalid for the 3-format case, since it may take much more than one line of output to discover the need to back up from the 2nd format to the 3rd.

I believe that the algorithm I have described is known to many people; at least to Glen Holloway and Bill Gosper. I have heard from both of them about the uses of exception-handling for this problem, but I have never seen the method written down, and didn't understand it clearly until I thought about it last night.

The reason I was thinking about it last night is that I wanted to document the Juno unparser, which, perhaps regrettably, does not use the method described in this note, though the methods have more in common than I used to realize. I took care in designing Juno's syntax to avoid any constructs that could not be accomodated by simple two-format methods — for example, no IF-THEN-ELSE, which is horrible to unparse.

The Juno unparser traverses the parse tree that is its argument in symmetric order, and sends a stream of information to a formatting program, when produces the final stream of output characters. The stream that the unparser sends to the formatter contains characters, "Begin Group" codes, "End group" codes, and "breakpoint" codes. The begin/end group codes should be properly nested; thus they specify a kind of tree structure over the stream. The breakpoint codes indicate desirable places to break; effectively they determine the multi-line formats for the group that contains them. One implementation of the formatting program would be to read all its input before sending any output, build a tree of nested groups, and apply the method described in this note. But this implementation is not acceptable, since we want to transduce streams continuously rather than in large chunks. Thus we must ~~inpu~~ implement the method of this note as a coroutine, giving it input one character at a time, and recording the state of its stack of local variables in an explicit data structure. This seems less and less appealing to me, but there are at least two reasons to do it. (1) It is already implemented

that way; in fact I already have a client in the person of Mike Spreitzer, who is using the formatter to implement formatted IO streams & ~~am~~ a pretty-printer for arbitrary Cedar Typed Variables. (2) It may be instructive to translate this algorithm into a coroutine just to see how ugly it becomes as the state that was implicit in the program counter and recursion stack is accounted for with explicit program variables.

— Greg Nelson, 29 Nov 83