SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(9), 839–856 (SEPTEMBER 1989)

A Language-independent Prettyprinter

MATTI O. JOKINEN
Department of Computer Science, University of Turku, SF-20500 Turku, Finland

SUMMARY

A general-purpose prettyprinting program is presented. The input to the prettyprinter is a linear stream composed of visible symbols and elementary formatting instructions. The user can specify a set of alternative formats and the precedence of the alternatives. The prettyprinter attempts to select automatically the 'best' layout that fits on the bounded horizontal space available on the output medium. The prettyprinter is implemented as a library module, which makes it more flexible than many traditional prettyprinters that are written as main programs. For instance, formatted expressions can be mixed with plain text or displayed interactively on the user's terminal, and the application program can use several prettyprinters simultaneously.

KEY WORDS Prettyprinting

INTRODUCTION

A prettyprinter takes clauses of a formal language as its input and prints them in a format that is easy to read. There are many prettyprinters designed for particular languages[1–8] but the problem is rather similar in most programming languages and even in mathematical and logical calculi. Several language-independent algorithms have been published.[9–14]

The simplest prettyprinters merely indent source lines without adding or deleting any new lines. Others break lines unconditionally at certain syntactic positions, e.g. immediately before keywords such as begin, if and while. In either case the details of the layout must be processed manually. This is not a big problem in program development because programmers know how to fill the details when the program is typed into a computer, and defects in the layout can be fixed during the debugging process. However, if the program was originally written by another programmer who had a different view of a good layout, or worse still, if the program was synthesized by a computer, then the prettyprinter should solve all details of the layout automatically.

To fit deeply nested clauses into a limited horizontal space, conditional formatting rules must be used. In a typical case the prettyprinter must decide whether the clause is printed on a single line or folded on several lines. Potential folding points may be logically connected. For instance, if a conditional statement does not fit on one line, then each branch should start on a new line. Similarly, the layout of a subclause may affect the format of the enclosing clause and vice versa. Thus the prettyprinting problem can be reduced to a constrained optimization problem: the readability of

0038–0644/89/090839–18$09.00

Received 18 September 1987
Revised 24 January 1989

the text should be maximized without exceeding the available horizontal space. The ordering of possible layouts by their readability must be defined by the formatting rules.

There has been much discussion about the best layout for programs.[15–23] There appears to be an agreement about certain basic principles, and experimental evidence has been found in favour of some rules.[24,25] Nevertheless, many details remain a matter of taste. Therefore the rules should not be fixed when a prettyprinter is implemented. A *programmable prettyprinter* gets the formatting rules as part of its input data.

The prettyprinting package presented in this paper can process a wide variety of formatting rules. The package was written in CLU[26] and it has been implemented as an abstract data type. The actual program is rather complicated because of many details added for flexibility and reliability. Therefore we shall not present the program as such but describe the basic ideas in a simplified form. The original program and its documents[27] are available in a computer-readable form upon request.

FORMULATION OF PRETTYPRINTING INSTRUCTIONS

Several authors have proposed formalisms for specifying prettyprinting rules.[3,8,10,11,13,14] The formalisms can be divided into two classes. In one approach programs are fed to the prettyprinter in the form of parse trees, and the rules are functions that map subtrees into formatted text. There need not be any obvious similarity between the parse tree and the output, a fact that makes this approach suitable not only to programs but also to abstract data structures. In the other approach the input is regarded as a mixed stream of formatting instructions and pieces of visible text. Formatting instructions can be considered as terminal symbols and they can be included in the grammar of the language. Although the expressive power of this method is more limited than that of the first approach, it is sufficient in practice. A stream-oriented prettyprinter can have a simple interface, which makes it a good general-purpose library module. This is the main reason why we chose the stream-oriented approach.

The stream should contain three kinds of items:

1. *Visible symbols*, or ordinary terminals of the language.
2. Items that specify spaces and newlines to be inserted between visible symbols. We use a term familiar from TEX[27] and call these items *glue*.
3. *Group parentheses* that indicate the subclause structure of the text. It need not be identical with the parsing structure. Group parentheses themselves are invisible in the output but they mark the current left margin in the folding process: indentation is always relative to the start of the smallest enclosing subclause. Parentheses also affect the processing of glue, as will be described below. In this paper group parentheses are denoted by braces {}. We shall introduce later another pair of parentheses, called *list parentheses*, but first we must discuss the nature of glue in more detail. A clause as a whole (e.g. a complete program) must always be enclosed in group or list parentheses.

When the stream is processed, each glue item is evaluated into a sequence of newlines and spaces. Conditional folding is indicated by glue that may be evaluated in more than one way. Most authors have limited the number of alternatives to two: one with a newline and the other without it. The prettyprinter selects the proper alternative depending on the length of the clause and the available horizontal space.

Deeply-nested clauses tend to run off the right edge of the output page even if folding occurs at all allowed positions. The user should have an opportunity to specify secondary folding points for such clauses. One solution is to add folding priorities to glues.[13] A more general solution is to allow each glue to specify several alternative formats with an increasing number of folding points and decreasing amount of indentation.

A glue item is represented as a sequence of *separators*. Each separator defines a possible sequence of newlines and/or blank spaces. A separator is represented as a record with two integer components:

```
separator = record[newlines,spaces: int]
```

The component *newlines* indicates the number of newlines to be printed. If it is zero, the component *spaces* denotes the number of spaces to be printed in place of the glue. If *newlines* is positive, *spaces* indicates the indentation of the continuation line relative to the current left margin. In the latter case the *spaces* component may even be negative.

In a glue item separators are ordered from the best to the worst and some of them may be identical. The prettyprinter will frequently need the lowest index i such that all separators i, $i+1$, . . . are similar. Efficiency can be increased by storing this index into each glue object:

```
glue = record[elems: array of separator,
              similar: int]
```

The index of a separator in the *elems* list is called a *style*. For simplicity we assume that the number of styles is fixed to N. Relaxation of this restriction does not essentially change the idea of the implementation but adds a number of uninteresting details into data structures and routines. In the actual CLU module the number of styles is unlimited.

To simplify the representation we define a shorthand notation for glues. The list of separators is enclosed in square brackets. The component *spaces* of each separator is written as a decimal integer and the value of the component *newlines* is indicated by the number of asterisks in front of the integer. Thus *2 denotes a separator with *newlines* = 1 and *spaces* = 2.

Example

Let us consider the printing of expressions of a simple language. Visible symbols appear in double quotes in the grammar.

```
expr       ::=  { sum }
sum        ::=  term | sum [1,*4,*2,*0] "+ " term
term       ::=  id | call
call       ::=  { id [0,0,*2,*1] "(" {expr_list } ")" }
                | { call [1,1,*2,*1] "(" { expr_list } ")" }
expr_list  ::=  expr | expr_list "," [1,*0,*0,*0] expr
```

These rules state that the preferred layout for expressions is to print everything on one line and insert one blank space after each comma, on both sides of an addition operator, and after a closing parenthesis when an opening parenthesis follows. In the second best style sums and argument lists are folded; the second and consecutive terms of a sum are indented four columns relative to the beginning of the first term and the elements of argument lists are left-aligned. In style 3 the indentation is reduced to two columns in sums and an additional folding point with a two-column indentation appears between a function and its argument list. In style 4 indentation is further reduced but there are no additional folding points. The space that always follows '+' was included in the symbol itself. The symbol '+' followed by the glue item [1,1,1,1] would have the same effect.

To design the glue items for a given language, decide first which separators are used in clauses which are so short that the page width is no problem. These separators determine the style number 1. Then assume that the clause exceeds slightly the maximum width and decide which changes you are willing to make to fit the clause on the output. You have then defined the style number 2. Continue until you feel that there are enough alternatives for clauses that are expected to appear normally. Programming languages typically require four or five different styles. It is *not* advisable to define emergency styles for extremely crowded clauses which are unlikely to occur in practice (see the beginning of the section on 'Implementation' for an explanation). The prettyprinter can handle bad input decently. If a clause does not fit on the output page in any style, one or more overwide lines will be printed, but the prettyprinter attempts to minimize their length by using the last style.

The reader may wonder whether an acceptable default format could be constructed automatically from the (unformatted) grammar of a language. The above example demonstrates that it is not easy. For instance, it may appear natural to enclose the right-hand side of each production rule in group parentheses, but the example shows that this strategy does not work well unless the representation of the grammar is selected with special care. Selection of separators is even more delicate: a newline can precede an opening parenthesis but not a closing parenthesis or a comma. Thus the construction of a default format would require a lot of knowledge about the habitual usage of various symbols.

ACCEPTABLE COMBINATIONS

It is not necessary to use the same style throughout a clause. We shall now derive a set of rules that determine how styles can be combined and which combinations should be used preferentially. The first rule is obvious:

Rule 1

If glue items G_1 and G_2 are enclosed by the same pair of group parentheses and neither of them is enclosed by an inner pair of parentheses, then G_1 and G_2 must be printed in the same style.

This rule excludes formats such as

```
f(a, b, c,
  d)
```

We shall write *style*(G) to denote the style of a glue item G in the output. If E is a subclause enclosed by group parentheses, then *style*(E) denotes the common style of those glue items in E that are not enclosed by inner parentheses.

Rule 1 is not adequate in all cases. For example, the label part of a Pascal program should be formatted as

```
label 1,2,3,4,
      5,6,7;
```

rather than

```
label 1,
      2,
      3,
      4,
      5,
      6,
      7;
```

To distinguish between the two cases the layout description language must be extended either with a new class of separators[10] or another pair of parentheses. The author has chosen the latter approach. The additional parentheses are called *list parentheses*. The variant of rule 1 for list parentheses is:

Rule 2

If glue items G_1 and G_2 are enclosed by the same pair of list parentheses and neither of them is enclosed by an inner pair of parentheses, then $|style(G_1) - style(G_2)| \leq 1$.

The style of a subclause limited by list parentheses is defined as the maximum of the styles of its top-level glue items.

Rules 1 and 2 define a horizontal dependency between styles of glues. There are vertical dependencies too. Style $k + 1$ is designed for longer clauses than style k; therefore style $k + 1$ can be used on the top of the clause hierarchy and style k on the bottom, but not inversely:

Rule 3

If E_1 and E_2 are subclauses limited by pairs of group or list parentheses and if E_2 is a part of E_1, then *style*(E_2) should not be greater than *style*(E_1).

This rule excludes formats such as

```
p(x, q(x,
       y,
       z), z)
```

The remaining rules do not exclude any combinations but they define the ordering of legal formats.

Rule 4

For each clause E, select the first style that is legal by rules 1 to 3. Then apply this rule to the subclauses of E.

This rule is applied recursively starting from the top level. It implies that a format with style 3 on the top level and style 2 on the bottom level is better than a format with style 4 on the top and style 1 on the bottom.

Sometimes there is a conflict between styles of disjoint subclauses. For instance, the following formats are legal by rules 1 to 4:

```
p(x, y) (z,
         w)

p(x,
  y) (z, w)
```

In this case the first format is preferable. More generally:

Rule 5

Let $S(E_1),\ldots,S(E_m)$ be the styles of the immediate subclauses of a clause E. If there is a conflict between the styles, assign a lower style to the first subclause.

The rule still leaves some freedom in the choice of styles. The styles of subclauses can be selected in a greedy manner, by choosing the lowest possible style for the first subclause before considering the rest. Alternatively, the lowest style simultaneously available for *all* subclauses can be set as the upper limit before rule 5 is used. The implementation of both variants will be described in this article.

USER'S INTERFACE TO THE PRETTYPRINTING PACKAGE

In this section we shall first describe the interface of the prettyprinting package and then demonstrate its use by an example. Details of implementation are described in the next section.

The package is an abstract data type with eight user-callable procedures:

start = **proc**(*ch*: *channel*, *width*: *int*) **returns**(*prettystream*)

putsymbol = **proc**(*p*: *prettystream*, *s*: *string*)

putglue = **proc**(*p*: *prettystream*, *g*: *glue*)

begingroup = **proc**(*p*: *prettystream*)

endgroup = **proc**(*p*: *prettystream*)

beginlist = **proc**(*p*: *prettystream*)

endlist = **proc**(*p*: *prettystream*)

finish = **proc**(*p*: *prettystream*)

We take an object-oriented view of the system and its data. The output channel, the buffers and other status data are combined into a *prettystream*-typed mutable object.

The procedure *start* must be called once at the beginning of printing; it creates and returns a prettystream object. The output channel is given as the first argument; the channel must already be open for writing. The second argument is the width of the output page. The procedure *finish* flushes the output buffers; it must be called once at the end of printing. The channel is reserved for prettyprinting between the calls of *start* and *finish* but it can be used for other output before *start* is called and again after calling *finish*. The procedure *putsymbol* mutates the stream by adding a visible symbol at the end of the stream. *Putglue* appends a glue item, *begingroup* and *endgroup* append group parentheses, and *beginlist* and *endlist* append list parentheses at the end of the stream.

The package does not care how the elements of the stream were generated. If the unformatted clauses are in a text file, a parser (or at least a scanner) is required as a front end. If the clauses are given as parse trees, the sequence of items is constructed by traversing the trees in the proper order.

Example

Let us apply the package to the language defined in the previous chapter. We assume that expressions are given as objects of types *expr* and *term*:

```
printexpr = proc(p: prettystream, e: expr);
    begingroup(p);
    printterm(p, pick_term(e, 1));
    for i: int from 2 to no_of_terms(e) do
        putglue(p, [1, *4, *2, *0]);
        putsymbol(p, "+ ");
        printterm(p, pick_term(e, i))
    od;
    endgroup(p)
end printexpr

printterm = proc(p: prettystream, t: term)
    case t in
        (s: string):
            putsymbol(p, s)
        (c: call):
            begingroup(p);
            printterm(p, function(c));
            if is_id(function(c))
                then putglue(p, [0, 0, *2, *1])
                else putglue(p, [1, 1, *2, *1])
            fi;
            putsymbol(p, "(");
            begingroup(p);
            printexpr(p, pick_arg(c, 1));
            for i: int from 2 to no_of_args(c) do
                putsymbol(p, ",");
                putglue(p, [1, *0, *0, *0]);
                printexpr(p, pick_arg(c, i))
```

```
            od;
            endgroup(p);
            putsymbol(p, ")");
            endgroup(p)
        esac
    end printterm
```

IMPLEMENTATION

In the first version of the prettyprinter the output was first stored in a tree which had one node for each subclause. Sizes of subclauses in each style were computed when the tree was built and the computed values were stored in the tree. When *finish* was called, the tree was traversed, final styles were selected and the text was printed into the output file. Since the tree and its attributes take much more space than the program text, this version could not handle large programs.

Usually the layout of a piece of program depends on items that come later in the stream, but the effect is rather localized. In typical cases fewer than 100 successive items need to be examined to fix the layout. Thus it is unnecessary to save the whole program in the main memory: each clause can be printed as soon as its final format is known.

The maximum size of the buffer depends mainly on the glue items on the outer levels of the clause hierarchy. The final layout of a clause can be computed only when all legal styles are identical with the worst style. Therefore the glue items on the outer levels should not contain separators that are rarely required. For instance, programs composed of nested procedures are usually formatted by indenting each procedure two columns or so relative to the enclosing procedure. If the last style defines an indentation of only one column, it will probably never be selected and the layout of the program cannot be resolved until the whole program is in the buffer.

In practice it suffices to print incomplete clauses only when a line of text is complete (that is, when a freshly-added glue item produces a newline) and this restriction makes the algorithms somewhat simpler.

Data structures

A prettystream object is represented as a record of the following type:

```
prettystream = record[chan: channel,
                      width: int,
                      col: int,
                      active: int,
                      stack: stack of frame]
```

The component *chan* is the output channel, *width* is the maximum line length, *col* is the length of the last, incomplete line in the output, and the component *stack* is a stack of incomplete subclauses. Element i of *stack* represents the immediate subclause of the clause represented by element $i - 1$. An element is pushed on the stack when an opening group or list parenthesis is encountered and it is popped out when the corresponding closing parenthesis is met. At the start (immediately after the call of the

procedure *start*) the stack contains one element, and at the end (immediately before the call of *finish*) it should again contain exactly one element. The component *active* is used by the *flush* procedure to remember which stack components have not been printed. Elements of the stack are of the following type

```
frame = record[clause: clause,
               room: array [1..N] of int,
               lmarg: int,
               best: 1..N,
               similar: 1..N]
```

The jth element of *room* contains the maximum horizontal space available for this clause in style j. The component *best* denotes an optimistic approximation of the style of the clause; the final style may be greater but not less than the approximation. The component *similar* is the first style such that all styles *similar* to N produce identical outputs (*best* $\leq$ *similar*). *Lmarg* is used by the *flush* procedure to remember the left margins of the clauses.

Clauses are represented as records of the following type:

```
clause = record[text: list of item,
                tight: bool,
                printed: int,
                width,last: array [1..N] of int]

item = union(string,glue,clause)
```

The component *text* represents the text of a clause as a list of symbols, glue items and completed subclauses. The component *tight* is true if the glues of the clause are 'tightly connected', that is if the clause is enclosed by group rather than list parentheses. *Width* and *last* are vectors with N elements. The ith element of *width* contains the width of the clause in style i, that is, the length of its longest line measured from the current left margin. Similarly, the ith element of *last* contains the length of the last line in style i. *Width* and *last* are computed assuming that all subclauses are printed in the same style. The component *printed* indicates the width of the initial part that has already been printed and therefore removed from the buffer.

The minimal length of the last and the longest line of a clause c may be smaller than that indicated by *c.width* and *c.last* if some subclause of c is wider in style i than in style $i - 1$. In that case the prettyprinter may select a non-optimal style. It occurs rarely, however, and the output is still a good approximation of the optimum. Therefore it seems inadvisable to pay the extra cost of finding the real optimum.

Routines

The procedure *start* creates a new prettystream object. The stack will initially contain one element that represents an empty clause:

```
start = proc(ch: channel, w: int) returns(prettystream)
    c: clause := record{text: empty,
                        tight: true,
                        printed: 0,
```

```
                    width: all zeroes,
                    last: all zeroes}
    s: stack of frame := new stack;
    push(s, record{clause: c, room: all w, lmarg: 0, best: 1, similar: 1})
    return(record{chan: ch, width: w, col: 0, active: 1, stack: s})
end start
```

The procedure *putsymbol* appends a symbol to the clause c in the top frame a and updates the vectors *c.width* and *c.last*. Only the elements *a.best* to N of the vectors need to be updated since the other elements (1 to *a.best* − 1) correspond to styles that have already been rejected:

```
putsymbol = proc(p: prettystream, s: string);
    a: frame := top(p.stack);
    c: clause := a.clause;
    c.text := append(c.text, s);
    for i: int from a.best to N do
        c.last[i] := c.last[i] + length(s);
        c.width[i] := max(c.last[i], c.width[i])
    od
end putsymbol
```

Note the object-oriented representation: the variable c and the *clause* component of the top element of the stack denote the same object, and any changes made in the former are visible in the latter. There is no implicit copying.

The procedure *putglue* resembles *putsymbol* but the length of the glue is style-dependent. The legality of styles is checked here and component *a.best* is increased if necessary. If all styles produce too wide an output, *a.best* is set to N. The addition of a glue item may also affect the *similar* component, which is updated appropriately. Finally the procedure checks whether the contents of the buffer can be printed immediately. That can be done if *a.similar* = *a.best* and the freshly-added glue item produces a newline in the style *a.best*:

```
putglue = proc(p: prettystream, g: glue);
    a: frame := top(p.stack);
    c: clause := a.clause;
    c.text := append(c.text, g);
    for i: int from a.best to N do
        s: separator := g.elems[i];
        if s.newlines > 0
            then c.last[i] := s.spaces
            else c.last[i] := c.last[i] + s.spaces
        fi;
        c.width[i] := max(c.last[i], c.width[i])
    od;
    if c.tight
        then
            while a.best < N and c.width[a.best] > a.room[a.best] do
                a.best := a.best + 1
            od
```

```
        else
            while a.best < N − 1 and c.width[a.best + 1] > a.room[a.best + 1] do
                a.best := a.best + 1
            od
        fi;
        a.similar := max(a.similar, g.similar);
        if a.similar = a.best and g.elems[a.best].newlines > 0 then flush(p) fi
    end putglue
```

The procedure *begingroup* pushes a new element on the stack. The new element represents a subclause that does not yet contain any text. The component *room* of the new frame is derived from the corresponding component of the old top frame:

```
    begingroup = proc(p: prettystream);
        a: frame := top(p.stack);
        c: clause := a.clause;
        room2: array [1..N] of int := new array;
        w: int := a.room[N] − c.last[N];
        for i: int from N by −1 to a.best do
            w := max(w, a.room[i] − c.last[i]);
            room2[i] := w
        od;
        for i: int from 1 to a.best − 1 do room2[i] := w od;
        push(p.stack,
             record{clause: record{text: empty,
                                   tight: true,
                                   printed: 0,
                                   width: all zeroes,
                                   last: all zeroes},
                    room: room2,
                    lmarg: 0,
                    best: 1,
                    similar: a.similar})
    end begingroup
```

The procedure *endgroup* pops an element from the stack and appends it to the next lower element of the stack:

```
    endgroup = proc(p: prettystream);
        a2: frame := pop(p.stack);
        c2: clause := a2.clause;
        if not c2.tight then error fi;
        a1: frame := top(p.stack);
        c1: clause := a1.clause;
        a1.best := max(a1.best, a2.best);
        for i: int from a1.best to N do
            c1.last[i] := c1.last[i] + c2.last[i];
            c1.width[i] := max(c1.width[i], c1.last[i])
```

```
        od;
        c1.text := append(c1.text, c2);
        p.active := min(p.active, size(p.stack))
    end endgroup
```

The procedure *beginlist* is identical with *begingroup* except that the *tight* component is set to false. The procedure *endlist* is identical with *endgroup* except that not is dropped from the fourth line.

The procedure *finish* just prints the contents of the buffer.

```
    finish = proc(p: prettystream);
        if size(p.stack) ≠ 1 then error fi;
        print(p, p.stack[1].clause, 0, p.width)
    end finish
```

The auxiliary procedure *flush* is called from *putglue*. The *active* component of the prettystream is the index of the first frame that may contain something to be printed (frames 1...*p.active*–1 have not been changed since the previous call of *flush*). *Flush* updates the *best* component of all frames in the stack, computes the left margin of each clause and outputs the clauses by calling the *print* procedure. Left margins and *active* are saved for the next call of *flush*:

```
    flush = proc(p: prettystream);
        style: int := 1;
        for i: int from size(p.stack) by −1 to 1 do
            p.stack[i].best := style := max(style, p.stack[i].best)
        od;
        lmarg: int := p.stack[p.active].lmarg;
        for i: int from p.active to size(p.stack) do
            a: frame := p.stack[i];
            a.lmarg := lmarg,
            c: clause := a.clause;
            print(p, c, lmarg, lmarg + c.last[a.best]);
            lmarg := p.col
        od;
        p.active := size(p.stack)
    end flush
```

Next let us examine the procedure *print*:

```
    print = proc(p: prettystream, c: clause, lmarg,lastcol: int);
        t: list of item := c.text;
        if size(t) > 0 then
            st1: int := min{k : 1 ≤ k ≤ N
                              and lmarg + c.width[k] ≤ p.width
                              and lmarg + c.last[k] ≤ lastcol};
            i: int := 1;
            while i ≤ size(t) do
                j: int := the index of the last item of the line starting from t[i];
                st2: int := style for the line t[i...j];
                if j = size(t)
```

```
            then printsubseq(p,t,i,j,st2,lastcol);
            else
                printsubseq(p,t,i,j,st2,p.width);
                g: glue := t[j + 1];
                write g.elems[st1].newlines line feeds to p.chan;
                p.col := lmarg + g.elems[st1].spaces;
                write p.col spaces to p.chan
        fi;
        i := j + 2
    od;
    c.text := empty;
    c.printed := p.col - lmarg
  fi
end print
```

The parameter c is the subclause to be printed, *lmarg* is the left margin of c and *lastcol* is the rightmost possible termination column of the last line of c. The parameter *lastcol* is important because the last line often cannot be extended to the right edge of the output page. As an example, consider the subclause h(j(x,y),k(z)) in

```
f(g(h(j(x,y),
      k(z))))
```

The procedure *print* computes first the final style *st1* of the clause. Then the text is divided into subsequences by locating those glue items that produce a newline in style *st1*. At this recursion level each subsequence can be considered as a single line, although it may actually contain multiline subclauses. Computation of j depends on *c.tight*. If *c.tight* is true, then j is the lowest index such that $j \geq i - 1$ and $t[j + 1]$ is a glue item g with $g.elems[st1].newlines > 0$; if there is no such g, then j is set to $size(t)$. If *c.tight* is false, j is the highest index such that

1. either $j = size(t)$, or $t[j + 1]$ is a glue item g with $g.elems[st1].newlines > 0$, and
2. the width of $t[i..j]$ in style *st1* or *st1* − 1 is not greater than the maximum width. The maximum width is determined by *lastcol* for the last line and by *p.width* for other lines.

The value of *st2* also depends on *c.tight*. If *c.tight* is true, then *st2* is always equal to *st1*. If *c.tight* is false, *st2* is equal to *st1* − 1 if the subsequence $t[i..j]$ can be printed in that style; otherwise *st2* is again equal to *st1*.

The subsequences are printed with the procedure *printsubseq*:

```
printsubseq = proc(p: prettystream, t: list of item, low,high,st,lastcol: int)
    lastcolumns: array [low..high] of int := new array;
    r: int := lastcol;
    for i: int from high by -1 to low do
        lastcolumns[i] := r;
        case t[i] in
            (s: string):
                r := r - length(s);
            (g: glue):
                r := r - g.elems[st].spaces;
```

```
            (c: clause):
                r := min(p.width − c.width[st], r − c.last[st])
        esac
    od;
    for i: int from low to high do
        case t[i] in
            (s: string):
                putstring(p.chan, s);
                p.col := p.col + length(s)
            (g: glue):
                n: int := g.elems[st].spaces;
                write n blank spaces to p.chan;
                p.col := p.col + n
            (c: clause):
                print(p, c, p.col − c.printed, lastcolumns[i])
        esac
    od
end printsubseq
```

The glue items in the text segment *t*[*low*. . .*high*] are printed in style *st*. Recall that *low* and *high* have been selected so that the glue items in the segment do not produce newlines in style *st*. All subclauses can also be printed in style *st*, but there may be room to print some or all of them in a better style. To make this space available, *printsubseq* computes the rightmost permissible termination column for each item and passes it as the *lastcol* argument to recursive calls of *print*. Extra space is used in a greedy manner: the first subclause may be printed in style 1 and the last in style 3 even if there is room to print all subclauses in style 2. The styles can be balanced by computing the vector *lastcolumns* for subclause styles 1 to *st* and choosing the first vector that satisfies the constraints.

Performance

The existing version of the prettyprinter can process about 75 lines per second in MicroVAX-II. An obvious source of inefficiency is that the widths of clauses are computed in all styles even though a majority of clauses is printed in style 1. We have implemented another version that computes the widths only when they are needed. This is rather insensitive to the number of styles but the overhead of lazy evaluation is rather high, and the lazy version turns out to be faster only when the number of styles is five or more. Since neither version has been tuned into extremity, these tests do not necessarily express the limits of attainable performances.

LIMITATIONS AND POSSIBLE EXTENSIONS

In this section we shall discuss briefly certain special problems in prettyprinting. Some of the problems can be solved with our package, others can be solved with relatively trivial extensions and some are difficult in the framework presented in this paper.

Formfeeds

Proper positioning of formfeeds affects the readability of a long program significantly. Formfeed control can be incorporated easily as an extension to the package described above. Each glue item should contain, as an additional component, a penalty of printing a formfeed at that position (penalties could be also attached to individual separators but that would probably be waste of space). Formfeed penalties have no effect on line breaks and indentation, but when the position of a newline is determined the prettyprinter computes the cost of formfeed at that position as a function of the formfeed penalty p and the line number n. The position with the lowest cost is selected. A useful formula for the cost is

$$cost = \begin{cases} p-n, & \text{if } n \text{ is not greater than the page length} \\ \infty & \text{otherwise} \end{cases}$$

Visible text in separators

In some languages line breaks affect the visible text to be printed (for example, in the Unix shell language newlines are equivalent to semicolons). Rose and Welsh[11] define separators so that they may contain visible text instead of a sequence of spaces when the number of newlines is zero. In the general case it should be possible to place visible text also immediately before a sequence of newlines and immediately after it. Addition of such separators to our system does not introduce any fundamental problems.

Folding at lexical level

Breaking of identifiers, keywords or numeric literals onto several lines is illegal in most languages and a need for such an emergency action is unlikely to occur if styles with sufficiently small indentation are defined. String literals, however, may be relatively long and in some languages they can be broken legally by inserting catenation operators between substrings. Actually there is no difference between breaking a clause and breaking a literal, except that in the latter case additional visible text may be needed to join the pieces together. Thus the problem reduces to the inclusion of visible text in separators.

Tables

Sometimes it is reasonable to align clauses by infix operators or some other lexical tokens in the middle of lines. The following statements might appear in a symbol table initialization routine:

```
name[addop]   := "+";   priority[addop]   := 6;
name[equalop] := "=";   priority[equalop] := 8;
name[orop]    := "|";   priority[orop]    := 10;
```

Such formats are not supported by our package and there may not be any simple way to extend it in this direction. In fact specification of the formatting rules may be

rather complicated. For example, how should the above statements be printed if the last statement-pair does not fit on one line? At least the following formats are possible:

```
name[addop]    := "+";   priority[addop]   := 6;
name[equalop]  := "=";   priority[equalop] := 8;
name[orop]     := "|";
priority[orop] := 10;
```

```
name[addop]        := "+";
priority[addop]    := 6;
name[equalop]      := "=";
priority[equalop]  := 8;
name[orop]         := "|";
priority[orop]     := 10;
```

```
name[addop]    := "+";
name[equalop]  := "=";
name[orop]     := "|";
priority[addop]    := 6;
priority[equalop]  := 8;
priority[orop]     := 10;
```

Comments

The fact that comments are usually not part of the syntax complicates prettyprinting of a program when it is given as plain source text. Difficulties arise because it is not always obvious to which clause the comment should be attached. This problem, however, is related to parsing rather than prettyprinting. It is discussed in some detail by Rose and Welsh,[11] who also present a fairly good solution.

Comments are long compared to lexical items and the need to fold them onto several lines occurs frequently. Folding a comment is basically similar to folding a literal.

Short comments are often placed on the right side of the page, clearly separated from the program text that resides on the left side. Such a format is more readable if comments are aligned. This is another example of the need to specify tabular output. It also demonstrates the difficulties in specifying conditional rules for tabular formats: it is clearly undesirable to align comments at any rate but it is not easy to relate the cost of unaligned comments to the cost of less attractive styles in the program text proper.

Context-dependent formatting rules

In a highly-refined print the amount of white space may depend on the structure of nearby subclauses. For example, many programmers use spaces in the innermost subexpressions more sparingly than on outer levels, preferring x * (a + b*y) over x * (a + b * y) and p(q(x,y), r(x,z)) over p(q(x, y), r(x, z)). Context-dependent glue items might

be a possible solution, but we believe that it is not the right way to extend the prettyprinter. Instead, the context dependencies should be handled by the calling program. The grammar can have separate production rules for clauses with different glue items, as illustrated by the rules for 'call' in the example above.

CONCLUSIONS

The prettyprinter described in this paper has the following properties:

1. It can support a wide variety of formatting styles. In particular, satisfactory rules can be specified for deeply-nested expressions.
2. Formatting rules can be represented in a simple formalism.
3. The text is written out on the fly as soon as its final format is known. Only a fraction of the text must be stored in the memory at any time.
4. The prettyprinter is defined as a collection of subroutines rather than as a main program. Thus it can be embedded in application programs as a library module.

It is the combination of these properties that makes the program different from other prettyprinters published hitherto.

ACKNOWLEDGEMENTS

The author wants to thank Jorma Sajaniemi, Jukka Teuhola, Kai Koskimies and Ulla Solin for their helpful comments, and Ville Leppänen for assistance in testing.

REFERENCES

1. K. Conrow and R. G. Smith, 'NEATER2: a PL/I source statement reformatter', *Comm. ACM*, **13**, 669–675 (1970).
2. J. Hueras and H. Ledgard, 'An automatic formatting program for PASCAL', *SIGPLAN Notices*, **12**,(7), 82–84 (1977).
3. I. Goldstein, 'Pretty printing: converting list to linear structure', *Memo 279*, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts 1973.
4. R. Bond, 'Another note on Pascal indentation', *SIGPLAN Notices*, **14**,(12), 47–49 (1979).
5. M. Jackel, 'A formatting parser for Pascal programs', *SIGPLAN Notices*, **15**,(7&8), 58–63 (1980).
6. R. M. Bates, 'A Pascal prettyprinter with a different purpose', *SIGPLAN Notices*, **16**,(3), 10–17 (1981).
7. D. Norris, 'An Ada prettyprinter', *Journal of Pascal and Ada*, **3**,(4), 29–48 (1984).
8. D. Waters, 'User format control in a LISP prettyprinter', *ACM Trans. Prog. Lang. Syst.*, **5**, 513–531 (1983).
9. A. C. Hearn and A. C. Norman, 'A one-pass pretty printer', *SIGPLAN Notices*, **14**,(12), 50–58 (1979).
10. D. Oppen, 'Prettyprinting', *ACM Trans. Prog. Lang. Syst.*, **2**, 465–483 (1980).
11. G. A. Rose and J. Welsh, 'Formatted programming languages', *Software—Practice and Experience*, **11**, 651–669 (1981).
12. M. Mikelsons, 'Prettyprinting in an interactive programming environment', *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon; *SIGPLAN Notices*, **16**,(6), 108–116 (1981).
13. L. F. Rubin, 'Syntax-directed pretty printing—a first step towards a syntax-directed editor', *IEEE Trans. Softw. Eng.*, **SE-9**, 111–127 (1983).
14. E. Morcos-Chounet and A. Conchon, 'PPML: a general formalism to specify prettyprinting', *Information processing*, **86**, 583–590 (1986).
15. J. L. Peterson, 'On the formatting of Pascal programs', *SIGPLAN Notices*, **12**,(12), 83–86 (1977).
16. M. H. Clifton, 'A technique for making structured programs more readable', *SIGPLAN Notices*, **13**,(4), 58–63 (1978).

17. P. R. Mohilner, 'Prettyprinting Pascal programs', *SIGPLAN Notices,* **13**,(7), 34–40 (1978).
18. J. E. Crider, 'Structured formatting of Pascal programs', *SIGPLAN Notices,* **13**,(11), 15–22 (1978).
19. P. Grogono, 'On layout, identifiers and semicolons in Pascal programs', *SIGPLAN Notices,* **14**,(4), 35–40 (1979).
20. G. G. Gustafson, 'Some practical experiences formatting Pascal programs', *SIGPLAN Notices,* **14**,(9), 42–49 (1979).
21. J. Ramsdell, 'Prettyprinting structured programs with connector lines', *SIGPLAN Notices,* **14**,(9), 74–75 (1979).
22. G. T. Leavens, 'Prettyprinting styles for various languages', *SIGPLAN Notices,* **19**,(2), 75–79 (1984).
23. R. R. Baldwin, 'Systematic indentation in PL/I: minimizing the reduction in horizontal space', *SIGPLAN Notices,* **21**,(9), 22–26 (1986).
24. J. R. Miara, J. A. Musselman, J. A. Navarro and B. Schneidermann, 'Program indentation and comprehensibility', *Comm. ACM,* **26**, 861–867 (1983).
25. J. Hansen and B. Sands, 'Some design considerations for a "C" source code pretty printer', *Sigsmall/PC Notes,* **11**,(2), 16–22 (1985).
26. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffer, R. Scheifler and A. Snyder, *CLU Reference Manual*, Springer-Verlag, 1981.
27. M. O. Jokinen, 'Prettystream—an abstract data type for prettyprinting', *Report A52*, Department of Computer Science, University of Turku, Finland, 1988.
28. D. E. Knuth, *The TEXbook*, Addison-Wesley, 1984.