# ForML - a Pretty-Printing Facility for SML

Ekkehard Rohwedder

25 January 1993

**Abstract**

This is the documentation accompanying Release 0.6 of ForML, a prettyprinting module written in SML that supports the formatting of concrete syntax in languages or language-development tools.

## 1 Introduction

A number of languages and language-development tools (e.g. CAML, Ppml) offer prettyprinting facilities to support the unparsing and formatting of abstract syntax. ForML, a formatter written in SML of New Jersey, adresses some of these issues for the SML language.

After examining basic goals involved in the design of a pretty printer, we will introduce the ForML formatting primitives and give examples of their use. Next we describe the entire ForML interface, and finally we give a more involved example of ForML's use.

The ForML ideas are taken from the CAML pretty-printing primitives [1], which in turn are based on the Ppml pretty-printing facility of the Centaur system [2]. In contrast to the pretty printing algorithm by Oppen [3], the ForML formatting is not based on a stream interface: an intermediate formatting structure is generated in memory before the final output is printed.

### Unparsing and Formatting

Lexer and parser transform input text into trees containing the abstract syntax of the input language. In language-development environments, however, one often wants to take the reverse route producing nicely formatted output from some abstract syntax trees. It is advantageous to structure this process into two steps that are interfaced by a formatting language:

- Unparsing — precedences and associativity in the abstract syntax tree are disabiguated (e.g. by inserting "(" and ")" or other bracketing constructs), and the appropriate output functions for the subtrees are selected (allowing, for example, the elision of deeply nested structures).
- A device-independent formatting language interfaces the unparser and the formatter.
- Formatting — the breakpoints in the output are determined, and the mapping to the output device(s) is performed.

ForML does *not* take you all of the way: the unparsing of abstract syntax trees will have to be implemented by hand. However, ForML offers a formatter-language interface and output routines for a monospaced device.[1]

Processing with ForML is efficient, allowing its use in an interactive environment, and its formatting language can express common formating demands directly and easily, while relying on perspicuous and usable concepts. It has been successfully employed as the formatting tool for the Elf-language [4].

In the next section we describe the primitives provided by ForML to build formats and how they affect the final output. SML program text will be set in `teletype font`, and we also assume that the `Formatter` structure has been opened, making its primitives directly available.

[1] Future versions will support other output devices, like LaTeX or PostScript. Please contact the author at `er@cs.cmu.edu` if you need to produce output for other than a monospaced device.

# 2 ForML Formats

## 2.1 Boxes

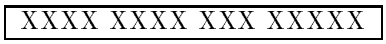
A `format` is either a string (`String "`*a string*`"`), a space `Space`, a breakpoint `Break`, or a box containing a list of formats. Breakpoints indicate mandatory or potential locations where a new line (and possibly increased indentation) is to be inserted into the output.

Boxes come in various formats, and —depending upon the flavor of the box in which they are situated and upon the available printwidth— the `Break`s inside of them are interpreted differently. To illustrate we will fill various boxes with the following list of formats

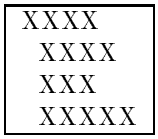
```
[String "XXXX", Break, String "XXXX", Break, String "XXX", Break, String "XXXXX"]
```

and observe the respective outputs[2].

- Horizontal boxes `Hbox` *never* break at their breakpoints. `Break`s are simply converted to whitespace.

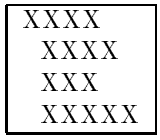



- Vertical boxes `Vbox` *always* break at all their breakpoints. The `Break`s are transformed into newlines, and a certain amount of indentation from the left border of the box is inserted.

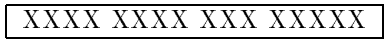



- Horizontal-or-vertical boxes `HOVbox` can behave as a horizontal box if the available pagewith accomodates the width of the box, otherwise they behave exactly like vertical boxes.

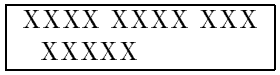

or



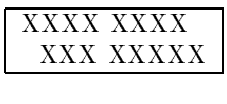

- Horizontal-vertical boxes `HVbox` allow each breakpoint *individually* to behave as if it was in a horizontal box or in a vertical box, depending upon whether the material after the break (and up to the next break) would still fit into the available pagewidth. Thus —depending on the page width— the above example could be displayed quite differently:



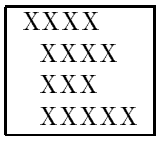

or




or




or




## 2.2 Example: Pretty-Printer for $\lambda$-Calculus

Let us delve into a concrete example and put these primitives to work by writing a formatter for a $\lambda$-calculus language presented with the following concrete grammar:

$$\begin{array}{lcl} exp & ::= & \lambda x.exp \mid let\ x = exp\ in\ exp \mid exp_A \\ exp_A & ::= & exp_A\ exp_B \mid exp_B \\ exp_B & ::= & var \mid \text{“(”}exp\text{“)”} \end{array}$$

[2] The suggestive frame drawn around the boxes is *not* part of the actual output.

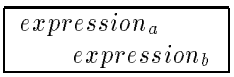
An SML datatype `exp` which captures the abstract syntax of such $\lambda$-expressions might look as follows:

```
datatype exp =
         Var of string
       | Lam of string * exp
       | App of exp * exp
       | Let of string * exp * exp
```

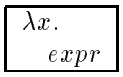
### 2.2.1 A first cut

Let us first design the "boxes" for the different language elements:

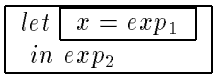
- variable names will be output directly
- applications $e_1\ e_2$ may be split if they do not fit into a line:




- no breaks are allowed immediately after "(" or immediately before ")"
- $\lambda$-abstractions allow a break after the ".":




- `let`-expressions also allow a break:




  Additionally we want —if necessary— "$x = exp_1$" to break after "=".

Armed with this design specification, we now can write a simple pretty-printer for `lambda`-expressions along the concrete syntax as follows:

```
fun format (Let(ID,exp1,exp2)) =
           HOVbox[String "let", Space,
                  HOVbox[String ID, String "=", Break, format exp1],
                  Break, String "in", Space, format exp2]
  | format (Lam(ID,exp)) =
           HOVbox[ String "\\", String ID, String ".", Break, format exp]
  | format exp = format_A exp
and format_A (App(expa,expb)) = HOVbox[ format_A expa, Break, format_B expb ]
  | format_A exp = format_B exp
and format_B (Var ID) = String ID
  | format_B exp = HOVbox[ String "(", format exp, String ")" ]
```

Here is sample output from this formatter with the `Pagewidth` 20:

```
\x.
   \y.
      \z.
         let x= y
            in y (x y)
                  (z y)
```

### 2.2.2 Some improvements

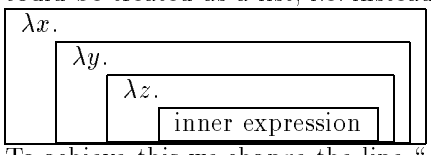
There are a number of points about this particular formatter that we are not quite satisfied with.

- We do not like that `let x= exp in ...` inserts a white space after `=` in horizontal output mode. However, we are in luck: ForML allows us to fine-tune breaks by specifying:
  - how many spaces to output for the break in horizontal mode (default: 1)
  - how many spaces to indent from the left border of the box, when the break actually is taken (default: 3)

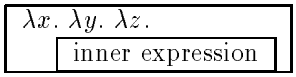
  The fine-tuning break primitive is called `Break0` and takes these two parameters as arguments. Thus in the `format (Let ...)`-clause of the above `format`-function we need to change the first `Break` to `Break0 0 3` to get the desired effect.

- Similarly, we might want to indent the `in` by one rather than by three spaces to align it under the `let`. We can achieve this by changing the second `Break` in the `Let`-clause to read `Break0 1 1`.

- A long series of $\lambda$-abstractions may lead to breaks in nested boxes, although it may be nicer if the could be treated as a list, i.e. instead of




we would like to have




To achieve this we change the line "`| format (Lam ...)`" to read "`| format (e as Lam(ID,exp)) = format_lam e nil`", and we add two clauses to define the auxiliary function `format_lam`:

```
and format_lam (Lam(ID,exp)) absl =
                format_lam exp (ll @ [String("\\" ^ ID ^ "."), Break])
  | format_lam exp ll = HVbox(ll @ [format exp])
```

With these improvements in place, the formatter now produces:

```
\x. \y. \z.
   let x=y
    in y (x y) (z y)
```

## 2.3 The signature FORMATTER

We now explain the interface to the formatter by describing the components of its `FORMATTER` signature in `formatter.sig`:

### 2.3.1 Defaults and Switches

Several references contain default values which may be changed by the user:

```
val Indent : int ref
val Blanks : int ref
val Skip   : int ref
```

`Indent`, `Blanks`, and `Skip` correspond to the values for added indentation, horizontal spacing, and vertical skip that are used by the default boxes. Their respective values are initially `3`, `1`, and `1`, but they may be changed at any point.

```
val Pagewidth     : int ref
```

`Pagewidth` contains the display width for the page. It is initially set to `80`.

```
val Bailout       : bool ref
val BailoutSpot   : int ref
val BailoutIndent : int ref
```

When you turn the `Bailout` flag on, ForML attempts to fail more gracefully whenever the output overruns the display width: such text will be output with an initial indentation of `BailoutIndent` from the left margin. Since ForML views the text boxes it formats as rectangles rather than octagons, it tends to "panic" when you turn the `Bailout` flag on, and outputs a number of text lines indented flushly by `BailoutIndent`, rather than just continuing normal indentation until the situation on the page really becomes tight again. To ameliorate this situation, set `BailoutSpot` to a value between `BailoutIndent` and `Pagewidth`, and then a Bailout will only be triggered when the leftmost edge of text would be output after the `BailoutSpot`. Sounds like a bailout, doesn't it? Initially, `Bailout` is set to `true`, `BailoutSpot` is `40`, and `BailoutIndent` is `0`.

### 2.3.2 Formats

The abstract datatype `format` holds the actual formats. The minimum and maximum display widths of a format may be inquired with `Width`.

```
type format
val  Width: format -> (int * int)
```

A whole slew of functions are provided to assemble formats, and you have already seen a cross-section of these earlier.

```
val  Break: format
val  Break0: int -> int -> format     (* blanks, indent *)
val  String: string -> format
val  String0: int -> string -> format (* output width *)
val  Space: format
val  Spaces: int -> format
val  Newline: unit -> format
val  Newlines: int -> format
val  Newpage: unit -> format
val  Vbox: format list -> format
val  Vbox0: int -> int -> format list -> format  (* indent, skip *)
val  Hbox: format list -> format
val  Hbox0: int -> format list -> format         (* blanks *)
val  HVbox: format list -> format
val  HVbox0: int -> int -> int -> format list -> format  (* blanks, indent, skip *)
val  HOVbox: format list -> format
val  HOVbox0: int -> int -> int -> format list -> format (* blanks, indent, skip *)
```

The functions ending in `0` take additionally arguments for indentation, horizontal spacing, or vertical skip, whereas their cousins (without the `0`) simply use the default values stored in `Indent`, `Blanks`, and `Skip`. The `String`, `Space`, `Spaces`, `Newline`, `Newlines`, and `Newpage` functions do what you would expect them to. `String0` requires you to additionally enter the width to be used in pagewidth calculations. Be forewarned, though, that ForML is pretty stupid about `Newline(s)` issued inside of boxes: it thinks they are just like any other string, except their length is 0, so you could get some strange formatting results — rather use `Break`s in `Vbox`es if you need to force new lines.

### 2.3.3 Shipping it out

The two functions

```
val  makestring_fmt:        format -> string
val  print_fmt:             format -> unit
```

turn formats into strings and print them out, respectively. If you want to put formatted output into an `outstream`, consider at the following:

```
type fmtstream
val  open_fmt:              outstream -> fmtstream
val  close_fmt:             fmtstream -> outstream
val  output_fmt:            (fmtstream * format) -> unit
val  file_open_fmt:         string -> ( (unit -> unit) * fmtstream )
val  with_open_fmt:         string -> (fmtstream -> 'a) -> 'a
```

The function `open_fmt` creates a "format-stream" `fmtstream` associated with a particular outstream, and `close_fmt` closes it, but leaves the `outstream` open. You use `output_fmt` to print a format on the `fmtstream`. Note that once you have closed an `outstream` associated with a `fmtstream`, any further output to that `fmtstream` will result in an I/O error.

`file_open_format` expects a file name as an argument. It opens the file and returns a pair consisting of a function to later close this file, as well as a `fmtstream` onto which to output formats.

Finally, the function `with_open_format` expects the name of a file, and a function whose argument is a `fmtstream`. It will open the file, call the function with the resulting `fmtstream`, and afterwards close the file and return the result of the function call.

# 3 An extended example: Mini-ML

We now extend the previous example to a subset of an ML-like language with the following abstract syntax:

$$\begin{array}{rcl} exp & ::= & x \mid n \mid \texttt{true} \mid \texttt{false} \mid exp_1\ exp_2 \mid \lambda x.exp \\ & & \mid \texttt{let}\ x = exp_1\ \texttt{in}\ exp_2 \mid \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \\ & & \mid [exp_1, \ldots, exp_n] \mid exp_1\ op\ exp_2 \end{array}$$

where $x \in Var$, $n \in Nat$, $\{\texttt{true}, \texttt{false}\} \in Bool$, $op = \{=, <, +, -, *, /, \ldots\}$, and $[\ldots]$ denotes a list.

A corresponding SML datatype for abstract syntax trees might be as follows (purely for convenience we also decorate `Op`-nodes with the operator precedence):

```
datatype mexp = Var of string | Int of int | Bool of bool
              | App of mexp * mexp
              | Lam of string * mexp
              | Let of string * mexp * mexp
              | If of mexp * mexp * mexp
              | List of mexp list
              | Op of string * int * mexp * mexp
```

The formatter for `mexp`s now also has to provide unparsing, i.e. it needs to take operator precedences into account. Below is an example of a formatting function for this language. Rather than mimicking some suitable concrete grammar, it deals with nested `App`s similarly as the second version of the `format`-function did with nested `Lam`s. The reader may want to reconstruct the basic box designs.

Before we start, however, we need to define an auxiliary function:

```
infixr 5 @@

fun (l1 @@ nil) = nil
  | (l1 @@ l2) = l1 @ l2
```

Now on to the main formatting routine:

```
fun mformat (Var s) = String s
  | mformat (Int n) = String (makestring n)
  | mformat (Bool b) = String (if b then "true" else "false")
```

```
  | mformat (e as Lam _) = mformat_lam e []
  | mformat (e as App (e1,e2)) = mformat_app e1 [mformat_op 50 e2]
  | mformat (Let(s,e1,e2)) =
          HOVbox[String "let", Space, HOVbox[String s, String "=", Break0 3 1,
                                             mformat e1], Break0 1 1,
                 String "in", Space, mformat e2]
  | mformat (If(e1,e2,e3)) = HOVbox[ String "if", Space, mformat e1, Break,
                                     String "then", Space, mformat e2, Break,
                                     String "else", Space, mformat e3]
  | mformat (List l) =
          Hbox[ String "[",
                HVbox (fold (fn (el, fmtl) =>
                                [ mformat el ] @
                                      [String ",", Break0 1 0] @@ fmtl)
                            l
                            nil),
                String "]" ]
  | mformat (e as Op(_,p,_,_)) = mformat_op p e
```

In formatting `Apps` above (and also below), we force surrounding parentheses by calling the formatting routine for operators with the highest priority for the "outside" operator. Finally, expressions consisting of nested `Lams`, `Apps`, and `Ops` are formatted by the auxiliaray functions `mformat_lam`, `mformat_app`, and `mformat_op` respectively:

```
and mformat_lam (Lam(s,e)) l =
                mformat_lam e (l @ [String("\\" ^ s ^ "."), Break])
  | mformat_lam e l = HVbox(l @ [mformat e])
and mformat_app (App(e1,e2)) l = mformat_app e1 ((mformat_op 50 e2)::Break::l)
  | mformat_app e l = HVbox((mformat_op 50 e)::Break::l)
and mformat_op p' (Op(s,p,e1,e2)) =
    HOVbox( (if p'>p then [String "("] else [])
            @ [mformat_op p e1] @ [ Break0 0 1, String s]
            @ [mformat_op p e2] @
            (if p'>p then [String ")"] else []) )
  | mformat_op p e = mformat e
```

A sample output with a `Pagewidth` of 20 might look as follows:

```
            \x. \y. \z.
               let silly=
                    \x. \y. \z.
                       if y+z>0
                          then x
                          else y
                in silly
                      [1, ~1, 0,
                       true,
                       false,
                       \x. x+1]
                      (1+(2+x)*3
                          /(4-5)
                       -6*7+8)
                      15
```

# References


[1] Pierre Weis et. al. The CAML reference manual. INRIA-ENS, March 1989.

[2] N.N. The Ppml manual. Sema Group, Fontenay-sous-Bois Cedex, France.

[3] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages*, 2(4):465–483, October 1980.

[4] Frank Pfenning. An implementation of the Elf core language in Standard ML. Available via ftp over the Internet, September 1991. Send mail to elf-request@cs.cmu.edu for further information.