Prettyprinting without losing comments

M.G.J. van den Brand
Programming Research Group, University of Amsterdam
P.O. box 41882, NL-1009 DB Amsterdam, The Netherlands
email: markvdb@fwi.uva.nl

Abstract

The problem of prettyprinting in generated programming environments is addressed. A practical solution for the prettyprinting process itself as well as for the handling of comments is given.

A new box-language is presented for specifying prettyprint rules in an algebraic way. Both the syntax and the semantics of this box-language are formalized. Instead of designing and implementing a formatting machinery in for example C, the transformation from box-language to (prettyprinted) text is formally specified. This specification can be executed and its performance is surprisingly good.

The box-language and its formatting algorithms suggest a general mechanism to prettyprint text with comments without losing the comments in the resulting text. There are no restrictions on the form of comments and places where comments may occur in the text. The method is not based on adaptations of the abstract syntax tree, but on modifications of the intermediate box-term that is built when a program is prettyprinted.

1991 CR Categories: D.2.1 [Software Engineering]: Requirements/Specifications - Languages D.2.3 [Software Engineering]: Coding - Pretty printers D.3.2 [Programming Languages]: Language Classifications - Specialized application languages

General terms: Design, Languages

Additional keywords and phrases: Comments, formatting algorithms

1 Introduction

Most programming environment generators have a special-purpose formalism for specifying prettyprinting rules, for example, both the Synthesizer Generator [RT89] and the PSG-system [BS86] have such a formalism. The CENTAUR-system [BCD+89] uses the prettyprint specification language PPML [Bor89]. The ASF+SDF meta-environment [Kli91], or more precisely the ASF+SDF formalism [BHK89, HHKR89], offers no explicit mechanism to specify prettyprint rules. Instead, the prettyprint rules are derived from the context-free grammar of the language in question. The advantage of this approach is that the specification writer does not have to bother to define the prettyprinting rules of a language, since a 'default' prettyprinter is always generated automatically by the system. The disadvantage is the lack of facilities to influence this default prettyprinter. Of course, we could introduce a separate prettyprint specification formalism to achieve this. However, we dislike this idea for two reasons:

- many semantic concepts from the specification language (like, e.g., pattern matching, tree traversal, substitution) will also be needed in the prettyprint formalism;
- this duplication of semantic concepts will also become manifest in a duplication of implementation effort of the prettyprint formalism.

The question now is: how can the specification writer influence prettyprinting without resorting to a separate prettyprint specification formalism? Our solution is to *identify* the two formalisms. Specifying the prettyprint rules for a language amounts then to writing an algebraic specification.

We have tackled the problem of prettyprinting in a larger context. First, we have developed a new and simple box-language (see Section 2) and we have written an algebraic specification of its semantics, viz. the transformation of expressions in the box-language to text (see Section 3). Furthermore, we present an elegant solution of prettyprinting text including comments, this is described in Section 4.

The box-language and the transformation algorithms are not enough for solving the remaining prettyprint problems but may serve as a starting point. We developed a generator which takes a context-free grammar of an arbitrary language and generates a prettyprinter for this language. The resulting prettyprinter consists of a set of rules which use the box-language. The specification writer can specify prettyprint rules which are combined with the generated prettyprinter and overrule its default behaviour. Both approaches, default prettyprinting and tailored prettyprinting, are described in a forthcoming paper.

2 Box-language

In most prettyprint specification languages the objects that have to be prettyprinted are represented by boxes. The terminal symbols in the language are basic boxes. These basic boxes can be glued together in various ways to form bigger ones (non-basic boxes), corresponding to the syntactic constructs in the source language to be prettyprinted. The various kinds of boxes together with the operators for composing them form a *box-language*. Formatting[1] amounts to transforming an expression in the box-language (a *box-term*) into text. Formatting proceeds in two steps. First, position and size information is added to the (non-)basic boxes. Next, this information is used to produce the text. We are only interested in the position information of the basic boxes, which is used to calculate the layout between terminal symbols and to update the position information in the abstract syntax tree that is used by other tools. The latter is not further discussed in this paper.

Our box-language is based on PPML [Bor89] the prettyprint specification language of the CENTAUR-system [BCD$^+$89] and Pretty [Vos90].

We distinguish five different *operators* in our box-language:

- `#H` (horizontal composition)
- `#V` (vertical composition)
- `#HV` (horizontal composition and vertical composition)
- `#HOV` (horizontal composition or vertical composition)

[1] We will use the word *formatting* for the transformation from box-term to text. The word *prettyprinting* is used when we refer to the transformation of an abstract syntax tree to text. Formatting is thus the last phase of prettyprinting.

- `#I` (indentation)

An operator works on zero or more boxes and specifies only the layout between (but not inside) the argument boxes. The terminal symbols are the most elementary boxes, for example '`begin`' is a box.

Each operator has a number of *options*. These options are useful for modifying the behaviour of the operators. If the options are not given, default values are used instead. The following options can be specified:

- `hs` (horizontal space between boxes)
- `vs` (vertical space between boxes)
- `is` (indentation space)

The `#H`-operator places boxes horizontally. The beginning of a box is on the same line as the ending of the preceding one, without paying attention to the right margin of the screen or window. The term `#H ["begin" "statements" "end"]` will be formatted as:

```
begin statements end
```

The `#H`-operator may only have the option `hs` to adjust the horizontal space between the boxes. For example, `#H hs=5 ["begin" "statements" "end"]` results in:

```
begin     statements     end
```

The default value of `hs` is 1.

The `#V`-operator places boxes vertically. Each box starts in the same column. The term `#V ["begin" "statements" "end"]` is formatted as:

```
begin
statements
end
```

The `#V`-operator may only have the option `vs` to adjust the vertical space between the boxes. For example, `#V vs=2 ["begin" "statements" "end"]` results in:

```
begin

statements

end
```

The default value of `vs` is 1.

The `#HV`-operator resembles the `#H`-operator. The boxes are put horizontally next to each other. If a box does not fit on the remaining space of a line, it will be placed on the next line, starting at the column in which the first box started. The `#HV`-operator may have the options `hs` and `vs`. The first is used to adjust the horizontal space between the boxes. The second one is used to adjust the vertical space if boxes are printed on a newline. It is not necessary to give both options, if only one is to be changed.

The `#HOV`-operator operates either as a `#H`-operator or as a `#V`-operator depending on two criteria. First, if the boxes fit on the remaining space of a line the operator acts as a `#H`-operator. However, if the boxes do not fit, the operator acts as a `#V`-operator. Second, if within one of the boxes a #V-operator is used the operator will act as a #V-operator as well. The #HOV-operator may have the options hs and vs. Construction the text for a box-term containing a #HOV-operator may be expensive since two passes may be necessary to distinguish the above two cases.

The #I-operator indents a box-term. It has only one box as argument and it may have the option is. The purpose of this operator is to shift the box horizontally over is spaces. The term #V ["begin" #I is=4 ["statements"] "end"] is prettyprinted as:

```
begin
    statements
end
```

The default value of is is 2.

The context-free syntax of our box-language is summarized in the following SDF definition:

```
context-free syntax
 "#H"                          -> OPERATOR
 "#V"                          -> OPERATOR
 "#HV"                         -> OPERATOR
 "#HOV"                        -> OPERATOR
 "#I"                          -> OPERATOR

 "hs"                          -> OPTION-SYMBOL
 "vs"                          -> OPTION-SYMBOL
 "is"                          -> OPTION-SYMBOL
 OPTION-SYMBOL "=" INT         -> OPTION

 STRING                        -> BOX
 OPERATOR OPTION* "[" BOX* "]" -> BOX
 BOX*                          -> BOX-LIST
```

The non-terminal STRING represents string values, for example the lexical symbols. The non-terminal INT represents integer values. The non-terminals OPTION* and BOX* describe both a list of zero or more boxes and options, respectively. SDF allows the specification of lists with separators as well, for example {OUT-BOX ","}*. Lists may also contain at least one element, this is expressed by writing '+' instead of '*'.

3 From box-term to text

The formatting process consists of two phases. In the first phase the box-term is transformed into an intermediate representation consisting of a list of strings, viz. the lexical symbols, with absolute position information attached to them. For example,

```
#V ["begin"
    #I [#V ["declare" #H hs=0 [#I [#H ["x" ":" "natural"]] ";"]]]
    #I [#HV ["x" ":=" "1"]]
    "end"]
```

is transformed into the list:

```
"begin" at (0,0),
"declare" at (2,1),
"x" at (4,2),
```

```
":" at (6,2),
"natural" at (8,2),
";" at (15,2),
"x" at (2,3),
":=" at (4,3),
"1" at (7,3),
"end" at (0,4)
```

In the second phase, this list with absolute position information is used to determine the layout between the lexical symbols. Given the above list, the result of the second phase would be:

```
begin
  declare
    x : natural;
  x := 1
end
```

The algorithm for doing the first transformation step, from box-term to intermediate representation, is the most interesting one. The algorithm interprets its input, viz. the box-term, and transforms it gradually into the intermediate representation. For each operator the algorithm recursively transforms the argument boxes and combines the intermediate presentation built for them in the way specified by the operator. We start with a global description of the various data structures used, we restrict ourselves to the most essential parts.

The first phase produces a list of strings with attached position information which has the following form:

```
context-free syntax
  STRING at POINT -> OUT-BOX
  {OUT-BOX ","}*  -> OUT-BOX-LIST
```

The box operators insert an amount of space between boxes; this information is given either by the options of the operator or by default values stored in a data structure called `FRAME`. The left and right margin may be used or even modified by the operators, therefore their values are stored in `FRAME` as well. The `FRAME` data structure also contains the start-point of the next box.

```
context-free syntax
  point (NAT,NAT)                    -> POINT
  frame (POINT,NAT,NAT,NAT,NAT,NAT) -> FRAME
```

A number of access-functions are defined on `FRAME`s:

```
context-free syntax
  set start-point POINT in FRAME     -> POINT
  get start-point FRAME              -> POINT
  get rightmargin FRAME              -> POINT
  get hs value from OPTION* or FRAME -> INT
```

The function '`get hs value from OPTION* or FRAME`' will first try to find the value of `hs` in the (possible empty) option list and if not found get the default value from the current frame. Similar functions exist for retrieving the value of `vs` and `is`.

3.1 From box-term to intermediate representation

For all the operators in the box-language the start-point of a box always depends on the end-point of the preceding one. Both start-point and end-point consist of a line and a column value, the initial value of the start-point is `(0,0)`. The effect of the box operators becomes thus mainly visible via the layout inserted between boxes.

The operators `#H`, `#V`, and `#I` place their argument boxes in a fixed manner and can do this in one pass. We will only give the rewrite rules for the `#H`-operator. In all the following rewrite rules the variable `Boxl` describes a list of boxes consisting of at least one element.

```
context-free syntax
  place BOX* with FRAME -> OUT-BOX-LIST

equations
  [1] place #H Options [Box] with Frame = place Box with Frame

  [2] place Box with Frame1 = Out-box-list1,
      get end-point Out-box-list1 = (X1,Y1),
      get hs value from Options or Frame1 = Hs,
      set start-point (X1+Hs,Y1) in Frame1 = Frame2,
      place #H Options [Boxl] with Frame2 = Out-box-list2
      ==================================================
      place #H Options [Box Boxl] with Frame1 = combine(Out-box-list1,Out-box-list2)
```

The `#V`-operator can be specified in a similar way. The `#I`-operator, however, works completely different: it only defines a point where extra horizontal offset is to be inserted and needs not process its list of argument boxes.

```
equations
  [1]  get is value from Options or Frame1 = Is,
       get start-point Frame1 = (X1,Y1),
       set start-point (X1+Is,Y1) in Frame1 = Frame2
       ============================================
       place #I Options [Box] with Frame1 = place Box with Frame2
```

The operators `#HV` and `#HOV` are more complex. The operators `#H`, `#V`, and `#I` did not take into account whether a box fits on a line or not, whereas the operators `#HV` and `#HOV` strongly depend on this fact. Furthermore, the functionality of the `#HOV`-operator changes if a `#V`-operator is used in one of its argument boxes.

There are two approaches: either we calculate the size of each of the argument boxes and use this information to decide how the argument boxes should be formatted, or we start placing the boxes and backtrack if it turns out that one of the boxes does not fit. The disadvantage of the first approach is that an accurate calculation of the size of a box-term amounts to formatting the box-term, so doing the job twice. The second approach is inefficient as well, since also double work is being done: first a box-term is formatted in horizontal mode and later on it may be necessary to reformat it in vertical mode.

We have developed a method which is based on backtracking but re-uses the results of the horizontal formatting of the box-term: the subbox-term is simply replaced by the formatted result. This is based on the observation that if a subbox-term fits on a line when some surrounding construct is printed horizontally, it will surely fit when the same construct is printed vertically. The only difference is the position information. Therefore it is only necessary to adapt the position information in the already formatted subbox-term. In order to be able to store this extra information we extend `OUT-BOX`.

```
context-free syntax
  STRING at POINT                          -> OUT-BOX
  BOX-LIST                                 -> OUT-BOX
  OPERATOR OPTION* "[" OUT-BOX-LIST "]" -> OUT-BOX
```

The latter extension of OUT-BOX is necessary since it is possible that the first argument boxes in a list were formatted but that the remaining ones were not, therefore it is necessary to know which operator was applied.

Until now we have given only the rewrite rules for placing BOXes. The rewrite rules for placing OUT-BOXes are similar and are left out, except for the rule:

```
context-free syntax
  place {OUT-BOX ","}* with FRAME -> OUT-BOX-LIST
  get start-point OUT-BOX-LIST    -> POINT
  adjust OUT-BOX-LIST given POINT -> OUT-BOX-LIST

equations
  [1] get start-point Out-box-list = (X1,Y1),
      get start-point Frame = (X2,Y2)
      ==============================
      place Out-box-list with Frame = adjust Out-box-list given (X2-X1,Y2-Y1)
```

where the function 'adjust OUT-BOX-LIST given POINT' traverses the list consisting of symbols and position information and updates the stored position information given the x- and y-values.

After these preparations we can now discuss the treatment of the #HOV-operator. Recall that this operator behaves like a #V-operator if one of its argument boxes runs over the right margin or a #V-operator is applied within one of its argument boxes. First, the horizontal formatting phase is applied. If all argument boxes fit horizontally and no #V-operator was applied, the constructed intermediate representation is returned, see rule [HOV-1]. If either the argument boxes do not fit horizontally or a #V-operator was applied, the formatting must be done in vertical mode re-using as much as possible of the information obtained from the horizontal formatting phase, see rule [HOV-2].

```
context-free syntax
  BOOL & OUT-BOX-LIST  -> BOB-STRUCT
  fits BOX* with FRAME -> BOB-STRUCT

equations
  [HOV-1] fits #H Options [Box Boxl] with Frame = true & Out-box-list
          =========================================================
          place #HOV Options [Box Boxl] with Frame = Out-box-list

  [HOV-2] fits #H Options [Box Boxl] with Frame1 = false & Out-box-list1,
          set leftmargin in Frame1 = Frame2,
          place #V Options [Out-box-list1] with Frame2 = Out-box-list2
          ==========================================================
          place #HOV Options [Box Boxl] with Frame1 = Out-box-list2
```

As soon as a box does not fit anymore the process starts to backtrack, which is expressed by the fact that the boolean value 'false' is returned.

```
context-free syntax
  BOOL & FRAME                                       -> BF-STRUCT
  given BOOL fits INT after OUT-BOX-LIST with FRAME -> BF-STRUCT
  given BOOL fits BOX* with FRAME                    -> BOB-STRUCT
```

```
  combine "(" OUT-BOX-LIST "," OUT-BOX-LIST ")"     -> OUT-BOX-LIST

equations
  [1] get start-point Frame = (X1,Y1),
      length of String+X1 <= get rightmargin Frame = false
      ===================================================
      fits String with Frame = false & String

  [2] fits Box with Frame1 = Bool1 & Out-box-list1,
      get hs value from Options or Frame1 = Hs,
      given Bool1 fits Hs after  Out-box-list1 with Frame1 = Bool2 & Frame2,
      given Bool2 fits #H Options [Boxl] with Frame2 = Bool3 & Out-box-list2
      ======================================================================
      fits #H Options [Box Boxl] with Frame1 = Bool3 & combine(Out-box-list1,Out-box-list2)

  [3] given false fits Hs after Out-box-list with Frame = false & Frame

  [4] get end-point Out-box-list = (X1,Y1),
      set start-point (X1+Hs,Y1) in Frame1 = Frame2
      ==========================================
      given true fits Hs after Out-box-list with Frame1 = true & Frame2

  [5] given false fits #H Options [Boxl] with Frame = false & Boxl

  [6] given true fits #H Options [Boxl] with Frame = fits #H Options [Boxl] with Frame
```

The insertion of the layout between boxes and the formatting of remaining boxes strongly depends on the formatting result of preceding box(es). The intermediate results are stored by the function '`combine(OUT-BOX-LIST,OUT-BOX-LIST)`', where the arguments may be either original box-terms, intermediate representations, or both.

The occurrence of an #V-operator causes the formatting process to backtrack.

```
equations
  [1] fits #V Options [Boxl] with Frame = false & #V Options [Boxl]
```

For operators except `#V` or `#I`, we have rules which resemble the following ones:

```
equations
  [1] fits #H Options [Boxl] with Frame = true & Out-box-list
      ======================================================
      fits #HV Options [Boxl] with Frame = true & Out-box-list

  [2] fits #H Options [Boxl] with Frame = false & Out-box-list
      =======================================================
      fits #HV Options [Boxl] with Frame = false & #HV Options [Out-box-list]
```

3.2 From intermediate representation to text

Given the intermediate representation consisting of strings with position information attached, we are able to derive the text which is to be printed. The algorithm generates a layout consisting of spaces and newlines. The algorithm keeps track of the end point of the last string printed, and compares this position with the position information of the next string to be printed. The difference is translated into spaces and newlines. The result of this process is a list of strings where each element in the list corresponds to one line of text. The resulting prettyprinted text can be used by various tools, for example an editor.

4 Prettyprinting comments

Box-terms are constructed on the basis of an abstract syntax tree and a set of prettyprint rules. The box-terms have no knowledge of the original abstract syntax tree and can be easily modified. This offers us the possibility of prettyprinting comments.

There are two different ways of dealing with comments in a programming language:

- Comments are considered part of the context-free syntax, thus it is specified in the context-free grammar of the language where they may occur. This is, for instance, the case in Eiffel [Mey92]. These comments are included in the abstract syntax tree.

- Comments are considered as a part of the lexical syntax, so they may occur anywhere and are not included in the abstract syntax tree. Comments are considered as layout.

The second form of comments are notoriously difficult to handle during prettyprinting [BS89, Jok89]. Below, we propose a solution for the treatment of this sort of comments. Note that we do *not* consider the case were comments may appear *inside* lexical tokens as in, e.g., Algol68.

The typical scenario is as follows. A parser processes a program text and builds an abstract syntax tree for it. A prettyprinter processes the abstract syntax tree using generated and/or user-defined prettyprint rules and produces text again. The difference between the input text of the parser and the output text of the prettyprinter is the layout. The comments in the input text will be lost in the output text, but this is unacceptable from the users' point of view.

One approach to solve this problem is to attach comments to nodes in the abstract syntax tree [RW81]. During prettyprinting, the comments are regenerated when printing the node in question. This method is also used in the CENTAUR-system [BCD$^+$89, Bor89]. Unfortunately, there is no unique and completely satisfactory method to determine to *which* node the comment should be attached. For instance, in

```
while x > 0 (* as long as x positive *)
do
  ...
od
```

should the comment be attached to the syntax tree for the `0`, the condition, or the while-construct? A wrong choice, may lead to an unexpected placement of the comment in the prettyprinted text.

In our approach, we use the position of a comment in the orginal text to determine its position in the prettyprinted text. After having produced the box-term for a given abstract syntax tree, we 'compare' it with the original text. The differences, thus the comments, are then inserted in the box-term. This approach is based on the following assumptions:

- the 'real' layout consists of spaces and newlines only and will be ignored in the text.

- a comment starts with a begin marker and ends with either a newline or an end marker.

- during prettyprinting *no* lexical symbols are changed, for instance, "`begin`" may not be replaced by "(".

Comments of the style above can specified, for instance, in the following way:

```
"%%" ~[\n]* "\n" -> LAYOUT
"%" ~[%]+ "%"    -> LAYOUT
```

We have completely specified the process of inserting comments in box-terms. We will not present all cases but restrict ourselves to comments of the first form, i.e. "%%" to the end of the line is comment. The text is represented as a list of strings where each element in the list corresponds to one line of text, see Section 3.2.

```
context-free syntax
  STRING*                             -> STRINGS
  BOX* "##" STRINGS                   -> BOX_STRING
  add-comments (BOX*,STRINGS)         -> BOX_STRING
  insert-comments-in-box (BOX*,STRINGS) -> BOX_STRING
  skip-chars (STRING,STRINGS)         -> STRINGS
  skip-spaces (STRINGS)               -> STRINGS
  is-comment (STRING,STRINGS)         -> BOX_STRING
  rest-comment (STRING,STRINGS)       -> BOX_STRING

equations
  [1] insert-comments-in-box (Boxl1,Strings1) = Boxl2 ## Strings2
      ===========================================================
      add-comments (Boxl1,Strings1) = Boxl2

  [2] insert-comments-in-box (Boxl1,Strings1) = Boxl2 ## Strings2
      ===========================================================
      insert-comments-in-box (Oper Options [Boxl1],Strings1) =
        Oper Options [Boxl2] ## Strings2

  [3] insert-comments-in-box (Box1,Strings1) = Box2 ## Strings2,
      insert-comments-in-box (Boxl1,Strings3) = Boxl2 ## Strings3
      ============================================================
      insert-comments-in-box (Box1 Boxl1,Strings1) = Box2 Boxl2 ## Strings3

  [4] skip-spaces (Strings1) = Strings2,
      skip-chars (String1,Strings2) = Strings3,
      skip-spaces (Strings3) = Strings4,
      is-comment (String1,Strings4) = Boxl ## Strings5
      ===============================================
      insert-comments-in-box (String1,Strings1) = Boxl ## Strings5

  [5] skip-spaces (Strings1) = Strings2,
      rest-comment (Strings2) = Boxl ## Strings3
      =========================================
      is-comment (String, "%%" : Chars Strings1) =
        #H [String #V ["%%" : Chars Boxl]] ## Strings3

  [6] Char != "%"
      ===========
      is-comment (String,Char : Chars Strings) = String ## Char : Chars Strings

  [7] skip-spaces (Strings1) = Strings2,
      rest-comment (Strings2) = Boxl ## Strings3
      =========================================
      rest-comment ("%%" : Chars Strings1) = "%%" : Chars Boxl ## Strings3

  [8] Char != "%"
      ===========
      rest-comment (Char : Chars Strings) = ## Char : Chars Strings
```

The program text is stored in a list of strings where each element in the list corresponds to one line of text. The symbol "`:`" is used as concatenation operator. The function '`skip-spaces (STRINGS)`' removes all layout at the beginning of the text. The function '`skip-chars (STRING,` STRINGS)' removes the string specified by the first argument from the list of strings specified by the second argument.

We will demonstrate the functionality of this algorithm by means of a simple example. Suppose the following text has to be prettyprinted:

```
begin %% This is a toy-PICO-program
%% x is declared as a natural and gets the value 1
declare x : natural; x := 1
end
```

The prettyprint rules generate the following box-term:

```
#V ["begin"
    #I [#V ["declare" #H hs=0 [#I [#H ["x" ":" "natural"]] ";"]]]
    #I [#HV ["x" ":=" "1"]]
    "end"]
```

Given the original text and this box-term our algorithm will transform it into:

```
#V [#H ["begin" #V ["%% This is a toy-PICO-program"
                    "%% x is declared as a natural and gets the value 1"]]
    #I [#V ["declare" #H hs=0 [#I [#H ["x" ":" "natural"]] ";"]]]
    #I [#HV ["x" ":=" "1"]]
    "end"]
```

The resulting prettyprinted text is:

```
begin %% This is a toy-PICO-program
      %% x is declared as a natural and gets the value 1
  declare
    x : natural;
  x := 1
end
```

Comments are always related to a preceding terminal symbol. Therefore comments at the beginning of a program have to be treated as a special case.

There are additional concerns when prettyprinting comments. For example, suppose a programmer has written a comment in the following way:

```
x %% x is now declared
: natural
```

Insertion of this comment after the 'x' in the box-term leads eventually to a syntactically incorrect program:

```
x %% x is now declared : natural
```

This can only happen within the argument boxes of a #H-operator or #HV-operator with this form of comments. A possible solution is to split the list of argument boxes, copy the operator and options, and force a line break after the comment string. The box-term

```
#H ["x" ":" "natural"]
```

will be then transformed into:

```
#H [#V [#H ["x" #V ["%% x is now declared"]] #H [":" "natural"]]]
```

This box-term will yield a syntactically correct result. We will not give the rewrite rules for solving this problem.

A disadvantage of our approach is that the box-term and the text must be completely traversed in order to insert the comments. A possible improvement would be to inspect the text while the box-term is being constructed and insert the comments at that time.

5 Related work

Our box-language is strongly influenced by both PPML [Bor89] and PRETTY [Vos90]. PPML is a prettyprinter specification formalism to translate abstract syntax trees into text. A PPML specification is translated to Lisp code which is executed by the PPML formatting machine. PPML uses five types of operators, in PPML terminology separators:

- `<h` d_x`>`
- `<v` d_i,d_y`>`
- `<v1` d_i,d_y`>`
- `<hv` d_x,d_i,d_y`>`
- `<hov` d_x,d_i,d_y`>`

The d_x, d_y, and d_i are the parameters of the separators. d_x stands for the horizontal offset, d_y for the vertical offset, and d_i for the indentation. There is a minor difference between the separators `v` and `v1`. The separator `v` initializes the left hand margin to the column value of the first argument box. The separator `v1` uses the left hand margin of the surrounding box.

There are a number of differences between our box-language and PPML.

- The order in which rules are specified in PPML is relevant, in our box-language there is no order in the rules. For each context-free grammar rule only one prettyprint rule may exist.
- PPML operators may have parameters. If only one of the parameters of a PPML operator is to be changed, all the other parameters must be specified as well. In our box-language it is possible to specify only one of the parameters, this is possible because of the explicit naming of the parameters.
- PPML operators are a mixture of a prefix and infix operators. In our language all the operators are prefix operators.

PPML specifications can also be processed by the formatting machine FIGUE [Has92], which offers possibilities to change font sizes and colors. Furthermore, it offers some extra operators.

Pretty was a first attempt to define the prettyprint process algebraically. The box-language Pretty offers a considerable number of operators. The operators are infix operators over the boxes. Due to the large number of (infix) operators the prettyprint specification tends to become unreadable.

Both the prettyprint language PPML and Pretty offer the possibility of elision, viz. hiding pieces of the prettyprinted text starting at a certain predefined level in order to increase the readability of the prettyprinted text. Our box-language does not yet offer this mechanism.

The specification language SSL of the Synthesizer Generator [RT89] offers a special prettyprint formalism. It is strongly based on attribute grammars and can produce the formatted text in one pass without backtracking. So, operators equivalent to `#HV` or `#HOV` do not exist. There are only operaters for increasing or decreasing the indentation and going to a newline. The formalism offers the facility to place error messages in the prettyprinted text.

The prettyprinting mechanism described in [BS89] makes it possible for the user of an editor to influence the prettyprinting process directly. This is done by allowing the user to adjust the value of a number of parameters, such as LineWidth, Indentation, etc. This prettyprinting mechanism is strongly connected to the languages Pascal and Modula-2. It has no facilities for adapting the prettyprinting for an arbitrary syntactical construct and it cannot be used in generated programming environments. This prettyprinting mechanism is based on Oppen's algorithm [Opp80].

The system we developed is also influenced by the algorithm formulated by Oppen. His mechanism allows the specification writer to insert output indications in the specification, which the algorithm uses to generate a prettyprint when traversing the abstract syntax tree. Oppen's algorithm [Opp80] receives a list of lexical tokens together with special characters to direct the prettyprinting. There are two categories of special characters:

- *bracket characters* to delimit a syntactical construct, such as an assignment or a series;
- *blank characters* to mark a possible line break and/or the number of blanks to be printed between lexical symbols.

The bracket characters consist of an open bracket to denote the start of a syntactical construct and a close bracket to denote its end. The blank characters denote possible break points. There are two different kinds of blank characters, viz. consistent and flexible blank characters. The consistent blank characters have the some functionality as the `#HOV`-operators, whereas the flexible blank characters resemble the `#HV`-operators.

6 Conclusions

We can summarize the following results:

- The main goal of writing a complete algebraic specification of the prettyprinter proved successful.
- The specification of the prettyprinting process makes the prettyprinter much easier to maintain and offers the possibility of adding new features to the box-language in the future. For example, dealing with different fonts during prettyprinting. Although the current version of our box-language does not offer the use of different fonts it will not be very difficult to integrate them in the future.
- The box-language proved to be a good starting point for the generation of the default prettyprinters for arbitrary languages, as well as for user-defined prettyprint specification.
- We are able to deal with comments in a very general way. It is not necessary to adapt either the context-free syntax of the language, or the abstract syntax tree in order to store the comments in an ad-hoc manner. In this paper we have restricted the form of the comments. However, we are able to deal with any form of comments, since given the definition of the comments, we are able to extract the appropriate set of rewrite rules for restoring them.

We conclude this paper with a few remarks on the performance. Although there is some computational overhead, the performance of the specified prettyprinter is encouraging. First tests with the ASF2C-compiler [WK] proved to be very promising. In tables 1 and 2 we give the results of our measurements of the performance of the compiled formatter. The machine used was a SPARCstation IPX (32MB internal memory) running SunOS 4.1.1. The initial program text was:

```
x := 1; if x
        then x := 1; if x
                     then x := 1; x := 2; x := 3; x := 4
                     else x := 1; x := 2; x := 3
                     fi; x := 1; x := 2; x := 3; x := 4; x := 5; x := 6; x := 7
        else x := 1; x := 2; x := 3
        fi; x := 1; x := 2; x := 3
```

The text was doubled for each measurement.

Box operators	Prettyprinted lines	Time
54	32	0.5 sec.
108	64	1.1 sec.
216	128	2.3 sec.
432	256	5.2 sec.
864	512	10.8 sec.

Table 1: Performance without #HOV-operators.

Box operators	#HOV-operators	Prettyprinted lines	Time
54	7	25	1.1 sec.
108	13	50	2.0 sec.
216	25	100	4.2 sec.
432	49	200	9.7 sec.
864	99	400	21.0 sec.

Table 2: Performance with #HOV-operators.

The values given in the last column of table 1 and 2 are the times used by the formatter to transform a box-term, the number of operators are given in the first column, into prettyprinted text, the number of lines are given in the second column. The values in the last column include the time to perform a complete traversal of the box-term and the text to be prettyprinted in order to restore comments. Both tables suggest that the performance of the formatter is almost linear. This conclusion is not completely justified. The performance is not only influenced by the number of operators but also by the operators used and their combinations, compare table 1 with table 2. The effect of the #HOV-operator on the performance is mainly influenced by nested occurrences of this operator. Looking at SSL examples [RT89] it shows that using #HV-operators and #HOV-operators is not really necessary. Avoiding them leads to a better performance.

There are various possibilities to improve the performance. First, we did some small experiments with incremental prettyprinting. Both the box-language and the incremental rewriting machine [Meu90] offer all possibilities needed for a fully incremental prettyprinting process. A second improvement is to work with relative position information, as described in [Koo92], instead of absolute position information in the intermediate data structure generated by the first phase of the prettyprinting process. This reduces the amount of calculation and more importantly makes the adaptations of the position information in the intermediate data structure unnecessary.

Acknowledgements

We want to thank Paul Klint and Susan Usküdarli for the numerous discussions on this subject and for reading draft versions of this paper.

References

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[Bor89] P. Borras. *PPML - Reference manual & compiler implementation.* INRIA, Sophia-Antipolis, 1989.

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[BS89] G. Blaschek and J. Sametinger. User-adaptable Prettyprinting. *Software—Practice and Experience*, 19(7):687–702, 1989.

[Has92] L. Hascoet. *FIGUE An Incremental Graphic Formatter User's manual for Version 1.* INRIA, Sophia-Antipolis, 1992.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Jok89] M.O. Jokinen. A Language-independent Prettyprinter. *Software—Practice and Experience*, 19(9):839–856, 1989.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. Report P9202, University of Amsterdam, 1992.

[Meu90] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990. Extended abstract in *AMAST'91: Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, Springer-Verlag London (1992), pp 277–286. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Meu90.ps.Z.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice Hall Object-Oriented Series. Prentice Hall, 1992.

[Opp80] D.C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[RW81] G.A. Rose and J. Welsh. Formatted Programming Languages. *Software—Practice and Experience*, 11:651–669, 1981.

[Vos90] K.J. Vos. Pretty for an easy touch of beauty. Master's thesis, University of Amsterdam, 1990.

[WK] H.R. Walters and J.F.Th. Kamperman. A self-fulfilling prophecy. to appear.