Generation of language independent modular prettyprinters

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

An algebraic approach to address the problem of generating prettyprinters is described. An algebraic prettyprint generator is defined which produces an algebraic prettyprinter for a language given its context-free grammar. Adaptation of the generated prettyprinter is possible by customizing the generated prettyprint rules. The prettyprinter transforms the abstract syntax tree of the text to be prettyprinted into a box-language expression. This expression, the so-called box-term, is processed by a generic formatter (also algebraically specified). The generated prettyprinter is able to deal with various forms of comments. The context-free grammar may be split into modules and, instead of generating one big prettyprinter for such a context-free grammar, a corresponding prettyprint module is generated for each syntax module. The collection of generated modules form the prettyprinter for the specified language. Prettyprinters for various languages have been generated. Among these are PASCAL, a simplified version of LISP, and Action Semantics.

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

General terms: Design, Languages

Additional keywords and phrases: Formatting algorithms

1 Introduction

Prettyprinters play an important rôle in generated programming environments, such as the Synthesizer Generator [RT89] and the PSG-system [BS86]. The generation of these prettyprinters is usually based on some kind of special-purpose prettyprint formalism. These formalisms are either part of a more general formalism such as the SSL formalism [RT89] or stand-alone such as the PPML formalism [Bor89] used in the CENTAUR-system [BCD+89]. The ASF+SDF Meta-environment [Kli93], or more precisely the ASF+SDF formalism [BHK89, HHKR89], offers no explicit mechanism for specifying prettyprint rules. In this paper we describe how to generate prettyprinters given a context-free grammar, without any prettyprint annotation, and how to incorporate these generated prettyprinters in the ASF+SDF Meta-environment.

The input to the generated prettyprinters are abstract syntax trees and their output is a readable version of the text represented by the tree. The application areas of the generated prettyprinters are quite diverse, the abstract syntax tree to be prettyprinted can be constructed either manually by means of some editor or automatically by some tool which generates output,

e.g. error messages, lisp code, etc.. The prettyprinter traverses the tree and prints each node according to the prettyprint rules available for it. The generation of a prettyprinter amounts to the generation of these prettyprint rules.

Given a context-free grammar of an arbitrary language a prettyprinter for this language is generated, see Section 3. This prettyprinter consists of a set of rewrite rules using the box-language introduced in Section 2. The availability of the prettyprint rules raised the issue of user customizable prettyprint rules, see Section 4. The combination of user-specified rules with generated rules is described in Section 5.

2 Box-language

In most prettyprint specification languages the objects that have to be prettyprinted are represented as 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). The various kinds of boxes together with the operators for composing boxes form a *box-language*. Formatting[1] amounts to transforming an expression in the box-language (a *box-term*) into text. It 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 the terminal symbols. The formatting algorithms are only informally described here, a more detailed description can be found in [Bra93].

2.1 Functionality of the box operators

We will give a short introduction to the box operators used and the syntactic structure of the box-language. Our box-language is inspired by PPML [Bor89] the prettyprint specification language of the CENTAUR-system [BCD+89].

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)
- `#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 such 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)

[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 complete transformation process from an abstract syntax tree to text. Formatting is thus the last phase of prettyprinting.

- `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 box, without paying attention to the right margin of the screen or window. The `#H`-operator may have one option `hs` to adjust the horizontal space between the boxes. The term `#H hs=5 ["begin" "declarations" "statements" "end"]` will be prettyprinted as:

```
begin     declarations     statements     end
```

The default value is 1.

The `#V`-operator places boxes vertically. Each box starts in the same column. The `#V`-operator may also have only one option `vs` to adjust the vertical space between the boxes. The term `#V vs=2 ["begin" "declarations" "statements" "end"]` is prettyprinted as:

```
begin

declarations

statements

end
```

The default value of this option is 1.

The `#HV`-operator resembles the `#H`-operator. The boxes are put horizontally after each other. If a box does not fit on the remaining space of a line bounded by the right, it will be prettyprinted on the next line, starting at the column in which the first box started increased by the value of the option `is`. The `#HV`-operator may have also the options `hs` and `vs`. The first is used to adjust the horizontal space between the boxes, whereas the second is used to adjust the vertical space if boxes are printed on a newline. It is not necessary to specify all options, if only one is to be changed. The default value of the `is` option is 2.

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 any of the boxes uses the `#V`-operator, the operator will act as a `#V`-operator as well. The `#HOV`-operator uses the options `hs` and `vs`. This operator may cause quite a lot of overhead during the transformation of the box-term into text: two passes may be necessary to distinguish the above two cases.

The `#I`-operator differs from all the other operators given earlier. It has at least one box as argument and it may have the option `is`. The purpose of this operator is to shift the boxes horizontally over `is` spaces. The term `#V ["begin" #I ["declarations"] #I is=4 ["statements"] "end"]` is prettyprinted as:

```
begin
  declarations
    statements
end
```

The default value of this option is 2.

The context-free syntax of the box-language is defined in SDF [HK89, HHKR92]:

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

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

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

The non-terminal `STRING` represents string values, such as the lexical symbols. The non-terminal `NAT` represents naturals. The non-terminals `OPTION*` and `BOX*` describe both a list of zero or more options and boxes, respectively. The specification of lists with separators is also possible in SDF. For example {`OUT-BOX ","`}`*` describes a list of out-boxes separated by commas, the list does not end with a comma. Lists may also contain at least one element, this is expressed by "`+`" instead of "`*`".

2.2 From box-term to text

The formatting process consists of two phases. In the first phase the box-term is transformed into some intermediate representation consisting of a list of strings, viz. the lexical symbols, with absolute position information connected to it. 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 is used to calculate the actual 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 of the first phase 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 representation built for the argument boxes as specified by the operator.

The operators `#H`, `#V`, and `#I` place their argument boxes in a specific manner and can do this in one pass. The operators `#HV` and `#HOV` are more complex. The operators `#H`, `#V`, and `#I` do 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.

The method for dealing with `#HV` and `#HOV`-operators, is based on backtracking but re-uses the results of the horizontal formatting of a box-term. This is achieved by replacing the subbox-term by the formatted result. This is based on the observation that a subbox-term, which fitted on a line when some surrounding construct was printed horizontally, will surely fit if the same construct is printed vertically. The only difference is the position information. Therefore it is necessary to adapt the position information in the already formatted subbox-term.

Given the intermediate representation which consists of strings with position information we are able to derive the text to be printed. The algorithm for the second phase generates layout consisting of spaces and newlines, using the positions connected to the strings. The algorithm keeps track of which position the last prettyprinted string ended, and compares this position with the position information of the next string to be printed. 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, such as an editor.

3 Prettyprinter generator

The ultimate goal is to derive a prettyprinter for an arbitrary language given its context-free grammar. In this section we will show how to derive prettyprint rules from a context-free grammar. We will first give a short description of the generator itself together with the structure of the generated prettyprinter and then concentrate on a number of problems, such as lists, priority and associativity definitions, and the generation of rules for prettyprinting comments.

3.1 Generator

The process of generating a prettyprinter for an arbitrary language consists of three phases. The first phase determines the prettyprint-type for each context-free grammar rule. This is the most crucial step in the generator. The prettyprint-types are used to generate the prettyprint functions as well as their definitions, these are the second and third phase of the generator respectively.

3.1.1 First phase

A set of heuristic rules is applied to determine how each language construct specified should be prettyprinted. The heuristic rules only look at the structure of a context-free grammar rule: the alternation of terminals and non-terminals is important as well as the number of terminals and non-terminals in the context-free grammar rule. The heuristic rules do not consider the place where a rule is applied. Refinements are, for example, looking at the length of terminals as well. The structure of the context-free grammar rules is expressed by the following prettyprint-types:

- `BRACKET`

  For example: `"(" EXPR ")" -> EXPR {bracket}`

- `LEFT`, `RIGHT`, `NON-ASSOC`

  For example: `EXPR "+" EXPR -> EXPR {left}`

- `EMPTY`

  For example: `-> EXPR`

- `INJECTION`

  For example: `TERM -> EXPR`

- `SMALL-PREFIX`

  For example: `"bool" ARG -> EXPR`

- `PREFIX`

  For example: `equal(INT,INT) -> EXPR`

- `INDENTED`

  For example: `"while" EXPR "do" SERIES "od" -> STATEMENT`

- `SIMPLE`

  All rules which are not of one of the preceding types.

For the determination of the prettyprint-type a kind of finite automaton is used, developed by Jan Rekers. These prettyprint-types are used to generate a prettyprinter. Extending the set of heuristic rules amounts to refining the generated prettyprinter. For example, the following context-free grammar rule of the programming language PASCAL

```
"var" { VAR-DECL ";" }+ ";" -> VARS
```

is typed as `INDENTED`, but it could be refined further.

3.1.2 Second phase

During this phase for each context-free grammar rule, a prettyprint equation is generated. For each syntax rule such an equation consists of a functionname with zero or more arguments corresponding to the number of non-terminals in the rule. The righthand side of this equation consists of a box-term formed by box-operators, terminal symbols, and other function calls for the non-terminals in the syntax rule.

For each non-terminal in the context-free grammar a prettyprint function is generated with a unique name. The reason for having unique prettyprint functions is that we may get an ambiguous specification if we have one general prettyprint function. If a non-terminal has more than one alternative a prettyprint equation is generated for each alternative. All these equations have the same functionname.

For example, the prettyprint equation for the rule

```
"if" EXP "then" SERIES "else" SERIES "fi" -> STATEMENT
```

is:

```
equations
  [1] ppSTATEMENT(if varEXP then varSERIES1 else varSERIES2 fi) =
        #V ["if"
            #I [ppEXP(varEXP)]
            "then"
            #I [ppSERIES(varSERIES1)]
            "else"
            #I [ppSERIES(varSERIES2)]
            "fi"]
```

The generated prettyprinter uses the abstract syntax tree built for a program to derive a box-term for the corresponding program. The fact that the generated prettyprint rules can use concrete syntax is based on a characteristic feature of the ASF+SDF formalism [Kli93].

3.1.3 Third phase

The generator generates also definitions for the prettyprint equations. These definitions consist of function names together with their arguments and their result sort. For example, the rule

```
"if" EXP "then" SERIES "else" SERIES "fi" -> STATEMENT
```

results in:

```
context-free syntax
  ppSTATEMENT ( STATEMENT ) -> BOX
```

All variables used in the equations should also be declared. Therefore a list of variable names with their types is generated. For example:

```
variables
  varEXP[0-9]*    -> EXP
  varSERIES[0-9]* -> SERIES
```

The definitions, variables, and prettyprint equations themselves form the prettyprinter. The generator is in fact a straightforward transducer from context-free grammar to prettyprinter. There are, however, a few characteristics of both the context-free grammar as well as the prettyprinter which make the generator more complex.

3.2 Modular structure of the generated prettyprinter

The SDF-formalism allows the specification of the context-free grammar to be split into several modules. The generator generates for each of these modules a separate prettyprint module. The import graph for the generated prettyprint modules is the same as for the context-free grammar. This allows an incremental approach to the generation of the prettyprinter. If one of the modules of the context-free grammar changes, a new prettyprint module needs to be generated only for this module.

3.3 Lists

Lists are used, among others, to squeeze left- and right-recursive rules into a more compact notation. There are two different types of lists: lists with a separator and lists without a separator. Furthermore, a list may consist of zero or more ("*") or one or more ("+") elements. The problem of prettyprinting lists is deciding whether its elements should be printed vertically or horizontally. We print lists with separators vertically. For the context-free rule

```
begin DECLS {STATEMENT ";"}* end -> PROGRAM
```

the following prettyprint rules are generated:

```
context-free syntax
  "ppSTATEMENT;*" ( {STATEMENT ";"}* )  -> BOX
  "ppSTATEMENT;*1" ( {STATEMENT ";"}* ) -> BOX

variables
  "varSTATEMENT"   -> STATEMENT
  "varSTATEMENT;*" -> {STATEMENT ";"}*
  "varSTATEMENT;+" -> {STATEMENT ";"}+

equations
  [1] ppPROGRAM(begin varDECLS varSTATEMENT;* end) =
        #V ["begin"
            #I [ppDECLS(varDECLS)]
            #I [ppSTATEMENT;*(varSTATEMENT;*)]
            "end"]

  [2] ppSTATEMENT;*(varSTATEMENT;*) = #V [ppSTATEMENT;*1(varSTATEMENT;*)]

  [3] ppSTATEMENT;*1() = ""

  [4] ppSTATEMENT;*1(varSTATEMENT) = ppSTATEMENT(varSTATEMENT)

  [5] ppSTATEMENT;*1(varSTATEMENT ; varSTATEMENT;+) =
        #H hs=0 [ppSTATEMENT(varSTATEMENT) ";"]
        ppSTATEMENT;*1(varSTATEMENT;+)
```

The prettyprinting of lists without separators is more problematic. Consider the following context-free grammar rules, the list of S-EXPRs can be printed vertical as well as horizontal. This decision depends mainly on the size of the individual S-EXPRs, but this information is not available during the generation of the prettyprint equations.

```
"(" S-EXPR*  ")" -> S-EXPR
"'" S-EXPR       -> S-EXPR
SYMBOL           -> S-EXPR
```

If the elements themselves are small it may be preferable to print them horizontal by means of the #HOV-operator. The #HOV-operator takes care of a good layout if one of the arguments is too big. However, due to the "inefficiency" of this operator it is advisable to test the number of elements in the list. If there are "many" items (determined by experiment: more than 4) it is unlikely to fit horizontally, thus the #V-operator is used instead. For the S-EXPR the following prettyprint equations are generated.

```
context-free syntax
  countS-EXPR* ( S-EXPR* )  -> INT

equations
  [1] ppS-EXPR( ( varS-EXPR* ) ) = #H ["(" ppS-EXPR*(varS-EXPR*)] ")"]

  [2] countS-EXPR*(varS-EXPR*) > 4 = true
      ====================================
      ppS-EXPR*(varS-EXPR*) = #V [ppS-EXPR*1(varS-EXPR*)]

  [3] countS-EXPR*(varS-EXPR*) > 4 = false
      =====================================
      ppS-EXPR*(varS-EXPR*) = #HOV [ppS-EXPR*1(varS-EXPR*)]
```

The function countS-EXPR* counts the number of elements in a S-EXPR-list.

3.4 Meta-variables

The meta-variables or placeholders are used to obtain structured editing. They represent holes in the program text. The textual representation of a meta-variable is the name of a non-terminal surrounded by the characters "<" and ">". A meta-variable can only replace syntactic constructs of the equivalent sort. A structure editor offers facilities of replacing meta-variables by templates, predefined language constructs consisting of terminals and meta-variables, or by text.

Meta-variables are automatically derived from the context-free grammar. Each non-terminal in the grammar may be used as meta-variable. The generator has to generate prettyprint rules for these meta-variables.

```
equations
  [1] ppEXP(<EXP>) = "<EXP>"
```

Lists as well as sublists can be replaced by meta-variables. For lists of the form "{SORT "separator"}*" the meta-variables "<SORTseparator*>" and "<SORTseparator+>" exist. For lists of the form "{SORT "separator"}+" both meta-variables exist as well. So, the generator has to generate the following prettyprint equations:

```
equations
  [1] ppSTATEMENT;*(<STATEMENT;*>) = "<STATEMENT;*>"

  [2] ppSTATEMENT;*(<STATEMENT;+>) = "<STATEMENT;+>"
```

The same holds for lists without separators.

3.5 Priority and associativity definitions

The priority and associativity relations of the (binary) operators are explicitly specified in ASF+SDF. This frees the specification writer from the need of transforming the grammar in order to disambiguate it. Priorities are defined in a separate `priorities`-section. The priorities of (binary) operators are expressed by a relative ordening on the corresponding context-free grammar rules.

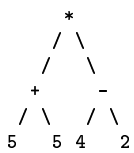
The associativity of the individual binary operators is defined by attributes like {`left`}, {`right`}, {`assoc`}, or {`non-assoc`} connected to context-free grammar rules of the form:

- `SORT "operator" SORT -> SORT {attribute}`
- `SORT OPERATOR SORT -> SORT {attribute}`
- `SORT SORT -> SORT {attribute}`

The associativity relation between different operators with the same priority is defined in the `priorities`-section. For example:

```
context-free syntax
  NAT         -> EXP
  EXP "+" EXP -> EXP {left}
  EXP "-" EXP -> EXP {left}
  EXP "*" EXP -> EXP {left}
  "(" EXP ")" -> EXP {bracket}
priorities
  EXP "*" EXP -> EXP >
  {left: EXP "+" EXP -> EXP,EXP "-" EXP -> EXP}
```

All the operators are left-associative. The operator "*" has a higher priority than the operators "+" and "-", which have the same priority and are left-associative with respect to each other. The rule "`"(" EXP ")" -> EXP`" is followed by the attribute {`bracket`}. For such a rule *no* node will be created in the abstract syntax tree. Given this context-free grammar the parser will built the following abstract syntax tree for the sentence `(5+5)*(4-2)`:



A naive prettyprinter will produce the output `5+5*4-2` for this abstract syntax tree. So, the prettyprinter was not able to put the brackets back, although they were necessary. Given the priority and associativity definitions the generator has to generate equations which print these brackets automatically. We assume that only one pair of brackets is defined per sort. If a sort has more than one pair of brackets, the generated prettyprinter selects one of these pairs and uses it consistently. For the context-free rule "`"(" EXP ")" -> EXP` {`bracket`}" the following function is generated.

```
context-free syntax
  addbracketsEXP (EXP,EXP,ASSOC) -> BOX-LIST
```

The first argument of this function is a direct subexpression of the expression given as second argument. The third argument expresses whether it was the left or right subexpression with respect to the binary operator of the surrounding expression. The function transforms the subexpression in a box-term and surrounds it by brackets *if* there was a priority or associativity conflict with the surrounding expression given as second argument.

```
equations
  [1] prioEXP(varEXP0) != prioEXP(varEXP1),
      (prioEXP(varEXP0) > 0 & prioEXP(varEXP0) < prioEXP(varEXP1)) = true
      ================================================================
      addbracketsEXP(varEXP0,varEXP1,varASSOC) = "(" ppEXP(varEXP0) ")"
```

The generator generates for "`EXP "+" EXP -> EXP`" the following function:

```
equations
  [1] addbracketsEXP(varEXP0,varEXP0+varEXP1,LEFT) = varBOX-LIST0,
      addbracketsEXP(varEXP1,varEXP0+varEXP1,RIGHT) = varBOX-LIST1
      ============================================================
      ppEXP(varEXP0 + varEXP1) = #H [varBOX-LIST0 "+" varBOX-LIST1]
```

In priority definitions non-binary expressions may be used as well. For the context-free grammar rule

```
EXP sub EXP sup EXP -> EXP
```

in:

```
context-free syntax
  EXP sub EXP         -> EXP {left}
  EXP sup EXP         -> EXP {left}
  EXP sub EXP sup EXP -> EXP
priorities
  EXP sub EXP sup EXP -> EXP >
  {left: EXP sub EXP -> EXP,EXP sup EXP -> EXP}
```

the following function will be generated:

```
[1] addbracketsEXP(varEXP0,varEXP0 sub varEXP1 sup varEXP2,NONE) = varBOX-LIST0,
    addbracketsEXP(varEXP1,varEXP0 sub varEXP1 sup varEXP2,NONE) = varBOX-LIST1,
    addbracketsEXP(varEXP2,varEXP0 sub varEXP1 sup varEXP2,NONE) = varBOX-LIST2
    ============================================================================
    ppEXP(varEXP0 sub varEXP1 sup varEXP2) =
      #H [varBOX-LIST0 sub varBOX-LIST1 sup varBOX-LIST2]
```

The use of `NONE` in the third argument indicates that the expression in the second argument is not a binary expression. The function `addbracketsEXP` ignores the associativity definitions and checks only whether there are priority conflicts.

3.6 Generation of prettyprint functions for comments

The generated prettyprinter is able to deal with comments in a very general way, the mechanism for restoring comments in the prettyprinted text is described in [Bra93]. The mechanism works as follows: both the orginal program text and the constructed box-term are traversed. Spaces and newlines in the program text are ignored and terminal symbols are skipped in both the box-term and the text. In this way the comments are filtered out and can be inserted in the box-term. In order to deal with comments the prettyprinter must know the "syntactic" structure of the comments, this is specified in the lexical syntax. For example:

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

Given these definitions the generator generates the following equations for "scanning" a text (represented as a list of strings) and extracting the comments.

```
context-free syntax
  BOX "##" STRING*     -> BOX_STRING
  is-comment (STRING*) -> BOX_STRING
  no-%-symbol (STRING) -> BOOL

equations
  [1] varChar != "%"
      ==============
      is-comment (varChar : varChars varSTRING*) =
        "" ## varChar : varChars varSTRING*

  [2] is-comment () = "" ## ""

  [3] is-comment ("%"  : "%" : varChars varSTRING*) =
        "%"  : "%" : varChars ## varSTRING*

  [4] no-%-symbol (varChar : varChars1) = true
      ========================================
      is-comment ("%" : varChar :  varChars1 : "%" : varChars2 varSTRING*) =
        "%" : varChar :  varChars1 : "%" ## varChars2 varSTRING*

  [5] varChar != "%"
      ==============
      no-%-symbol (varChar : varChars) =
        no-%-symbol (varChars)

  [6] no-%-symbol ("") = true

  [7] no-%-symbol ("%" : varChars) = false
```

A detailed description of the use of these equations can be found in [Bra93].

4 User-defined prettyprinting

The structure of the generated prettyprinter enables the specification writer to adapt it in an algebraic way. In this section we will show how to define prettyprint rules by means of some typical layout schemes and how they can be specificied.

In the generated prettyprint rules an elaborate naming convention is used with respect to function names and variables. The justification for this convention can be found in Section 5.

4.1 Polishing generated prettyprint rules

In Section 3.1.1 we gave an example of a context-free grammar rule

```
"var" { VAR-DECL ";" }+ ";" -> VARS
```

of the programming language PASCAL. This rule is typed as `INDENTED`, thus the generated prettyprint rule is:

```
equations
  [1] ppVARS(var varVAR-DECL;+ ;) =
        #V ["var"
            #I [ppVAR-DECL;+(varVAR-DECL;+)]
            ";"]
```

The PASCAL program text "`var <VAR-DECL> ;`" is prettyprinted as:

```
var
  <VAR-DECL>
;
```

whereas the following way prettyprinting is more "natural":

```
var
  <VAR-DECL>;
```

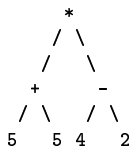
This is achieved by modifying the generated rule in the following way:

```
equations
  [1] ppVARS(var varVAR-DECL;+ ;) =
        #V ["var"
            #I [#H hs=0 [ppVAR-DECL;+(varVAR-DECL;+)] ";"]]
```

The amount of polishing strongly depends on the structure of the language for which the prettyprinter is generated. The number of rules to be adapted is small for Algol-like languages, however for a language like LISP more work has to be done.

4.2 Expressions

Consider the context-free grammar and the abstract syntax tree from Section 3.5:



The specification writer wants to have a 2-dimensional view on his expression like:

```
  (   5
    +
      5 )
*
  (   4
    -
      2 )
```

This can be achieved by a simple alteration of some of the generated rules. The prettyprint functions for the binary expressions must by modified in the following way:

```
equations
  [1] addbracketsEXP(varEXP0,varEXP0+varEXP1,LEFT) = varBOX-LIST0,
      addbracketsEXP(varEXP1,varEXP0+varEXP1,RIGHT) = varBOX-LIST1
      ============================================================
      ppEXP(varEXP0 + varEXP1) = #V [#I [varBOX-LIST0] "+" #I [varBOX-LIST1]]
```

4.3 Columns

Although the box-language only uses 5 operators it is possible to specify other layout schemes. One of them is columns. Suppose we have a list of identifiers with their types, and we want them to be prettyprinted as:

```
intvar    : int,
boolvar   : bool,
stringvar : string;
```

This layout scheme can be obtained by first traversing the list, which is to be prettyprinted, in order to find the maximum length of identifiers which is used when prettyprinting the list.

```
context-free syntax
  ppID-TYPE,* ( {ID-TYPE ","}* )               -> BOX
  ppID-TYPE,*1 ( {ID-TYPE ","}* )              -> BOX
  maxlengthID-TYPE,*1 ( INT , {ID-TYPE ","}* ) -> INT
  length ( ID )                                -> INT

equations
  [1] ppDECLS(declare varID-TYPE,*0 ;) =
        #V ["declare"
            #H hs=0 [#I [ppID-TYPE,*(varID-TYPE,*0)] ";"]]

  [2] ppID-TYPE,*(varID-TYPE,*0) =
        #V [ppID-TYPE,*1(varID-TYPE,*0,
                        maxlengthID-TYPE,*(0,varID-TYPE,*0))]

  [3] ppID-TYPE,*1(,varINT) = ""

  [4] ppID-TYPE,*1(varID : varTYPE,varINT) =
        #H [ppID(varID)
            #I is=varINT-length(varID) [":"]
            ppTYPE(varTYPE)])

  [5] ppID-TYPE,*1(varID : varTYPE , varID-TYPE,+0,varINT) =
        #H hs=0 [#H [ppID(varID)
                     #I is=varINT-length(varID) [":"]
                     ppTYPE(varTYPE)] ","]
        ppID-TYPE,*1(varID-TYPE,+0,varINT)
```

```
[6] maxlengthID-TYPE,*(Int,) = Int

[7] (length(varID) > Int) = true
    ============================
    maxlengthID-TYPE,*(Int,varID : varTYPE , varID-TYPE,*0) =
      maxlengthID-TYPE,*(length(varID),varID-TYPE,*0)

[8] (length(varID) > Int) = false
    ============================
    maxlengthID-TYPE,*(Int,varID : varTYPE , varID-TYPE,*0) =
      maxlengthID-TYPE,*(Int,varID-TYPE,*0)
```

The disadvantage of this solution is that the list of identifiers is traversed twice, once for the determination of the maximum length of the identifiers and once for the generation of the box-term. The computational overhead can be rather significant with large lists.

This mechanism also allows the specification of more than 2 columns. For each individual column the maximum length of the elements must be calculated, which is used during the generation of the box-term. So far we have not considered generalizing this type of specification. It may turn out that the box-language should be extended with an operator to implement columns.

5 Combining the two types of prettyprint functions

The generated prettyprinters are used within programming environments generated by the ASF+SDF Meta-environment [Kli93]. This environment is a collection of editors for the specification and testing of parts of the programming environment to be generated. This characteristic offers us a flexible mechanism for adapting generated prettyprint functions.

Given the context-free grammar of a language the generator generates a prettyprintter for this language. The generated prettyprinter consists of two parts: the generated prettyprint functions and their definitions with the variables used. A modification of the context-free grammar makes it necessary to generate a new prettyprinter, however it is not acceptable that existing user-defined prettyprint functions are lost. It may be necessary to adapt them but the generator may not throw them away. Adapting the generated prettyprinter can be done in two ways. The specification writer can either modify the generated prettyprint functions or specify prettyprint functions which overrule the corresponding generated ones.

The first solution would result in the loss of user-defined functions. The second solution demands the use of so-called "default" equations, which are executed only when no other function with the same name and arguments exists. The generator uses the latter solution, when generating a new prettyprinter for a modified grammar the "old" user-defined functions are kept and automatically included in the new generated prettyprinter. It may be necessary to adapt some of these rules due to the modification of the syntax.

Both solutions demand the specification writer to take into account the naming convention used by the generator. This problem is solved by invoking two editors, one with the generated prettyprint functions, and one with the generated variables and function definitions. The latter one can be used by the specification writer when specifying new prettyprint functions. During the specification of the functions the generated function definitions and variables are visible for the specification writer so that these they can be used (Figure 1) in particular the editor with the name "`ADD-PP-Bool`". Figure 1 shows the user specified rule for prettyprinting a boolean expression, the rule `[PP3]` overrides `[default-3]`.

Figure 1: Example of the editors for a generated prettyprinter

6 Related work

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

- `<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 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 the possibilities to change fontsizes and colors. Furthermore, it offers some extra operators.

PRETTY [Vos90] 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. 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 operations for increasing or decreasing the indentation and going to a newline. The formalism offers the facility to place error messages in the produced text.

The programming environment generator Pregmatic [Bra92] offered the user of a generated environment a tool to modify the prettyprint rule for a specific language construct. This can be considered as a very simple form of "prettyprinting-by-example".

The prettyprint mechanism described in [BS89] makes it possible for the user of the 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 prettyprint mechanism is strongly connected to the languages PASCAL and MODULA-2. It has no facilities for adapting the prettyprinting of an arbitrary syntactical construct and it cannot be used in generated programming environments. The prettyprint 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 character denotes a possible break points. There are two different kinds of blank characters, viz. consistent and flexible blank characters. The consistent blank characters have the same functionality as the `#HOV`-operators, whereas the flexible blank characters resemble the `#HV`-operators.

7 Conclusions and future work

We can summarize our results as follows:

- The main goal of writing a complete algebraic specification of a prettyprinter generator proved successful.
- The specification of the prettyprinting process makes the prettyprinter generator much easier to maintain and offers the possibility of adding new features in the future as well as experimenting with new heuristic rules.
- The generation of modular prettyprinters reduces the generation time, specially after modification of the context-free grammar. It is a first step towards the incremental generation of prettyprinters.
- The box-language proved to be a good starting point for the generation of 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 restore the comments in an ad-hoc manner. We can 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.

The prettyprinter generator described in this paper is used within the ASF+SDF Meta-environment [Kli93], which is a programming environment generator using incremental program generation techniques [HKR91]. Some of the tools generated, such as the lexical scanner [HKR92] and parser [HKR89], are based on this technique. The incremental generation of prettyprinters is still a research issue.

We have used the generator for deriving prettyprinters for several languages for which an ASF+SDF specification exists, such as PASCAL, CLaX [DT92], the box-language itself, simplified LISP code, Action Semantics [Mos92], fSDL [WKD93], COBOL, C, and the definition part of the generated prettyprinters. The generator is compiled using the ASF2C-compiler [KW93] to obtain a fast generator. The generated (and modified) prettyprinter can also be compiled in order to obtain a fast prettyprinter. The performance of the resulting prettyprinter is mainly influenced by the operators used. By reducing the use of `#HV` and `#HOV` operators the performance is improved.

The generated prettyprinter in general produces good results, although in a few cases adapting some of the prettyprint functions yields better results. Specially for languages with an exceptional syntactic structure, such as the definition part of the generated prettyprinters. The prettyprinters for languages with an Algol-like structure need almost no modifications. Other types of languages need more modification unless the `#HOV`-operator is used, but as mentioned before this reduces the performance of the resulting prettyprinter.

Acknowledgements

I want to thank Wilco Koorn, Jasper Kamperman, Pum Walters, Paul Klint and Susan Usküdarli for the numerous discussions on this subject. I thank Paul and Susan also 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.

[Bra92] M.G.J. van den Brand. *Pregmatic, A generator for incremental programming environments.* PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[Bra93] M.G.J. van den Brand. Prettyprinting without losing comments. Report P9315, University of Amsterdam, Amsterdam, 1993. Submitted.

[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.

[DT92] T.B. Dinesh and F. Tip. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as DT92.ps.Z.

[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.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual,* version 6 December, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. Available by *ftp* from ftp.cwi.nl:/pub/gipe as SDFmanual.ps.Z.

[HK89] J. Heering and P. Klint. The syntax definition formalism SDF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 283–297. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 6.

[HKR89] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *SIGPLAN Notices*, 24(7):179–191, 1989.

[HKR91] J. Heering, P. Klint, and J. Rekers. Lazy and incremental program generation. Report CS-R9124, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. To appear in *ACM Transactions on Programming Languages and Systems*.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[KW93] J.F.Th. Kamperman and H.R. Walters. Arm, abstract rewriting machine. In *Conference Proceedings of Computing Science in the Netherlands, CSN'93*. SION, 1993.

[Mos92] P.D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 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.

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

[WKD93] H.R. Walters, J.F.Th. Kamperman, and T.B. Dinesh. An extensible language for the generation of parallel data manipulation and control packages. To appear, 1993.