Generation of Formatters for Context-free Languages

M.G.J. van den Brand
E. Visser
Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
email: {markvdb,visser}@fwi.uva.nl

Abstract

Good documentation is important for the production of reusable and maintainable software. For the production of accurate documentation it is necessary that the original program text is not copied manually to obtain a typeset version. Aside of being tedious, this will invariably lead to errors. Tools that support the production of legible and accurate documentation are essential in software engineering. The production of such tools is a software engineering challenge in itself.

We present an algebraic approach to the generation of tools that produce typographically effective presentations of computer programs. A specification of a formatter is generated from the context-free grammar of a programming language. These genererated formatters translate abstract syntax trees of programs into box expressions. These box expressions are translated by language independent interpreters of the box language into ASCII or TEX. The formatting rules that are generated can easily be fine tuned in order to get the desired exact formatting of programs.

A practical solution for the problem of formatting comments, which occur in the original text, is also presented. Furthermore, two examples of generated formatters, for Cobol and for a process specifcation language, are discussed in some detail.

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: Formatting algorithms

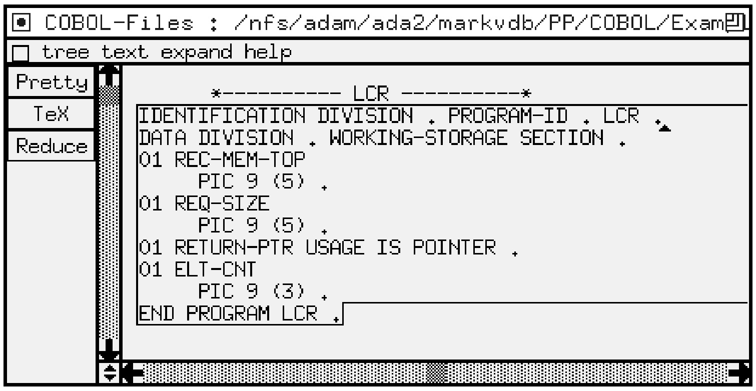
1 Introduction

Software engineering and software documentation are almost synonymous. A program that is not documented will soon become a unintelligible black box that is hard to maintain and extend. Typesetters and pretty printers help to visualize the structure of programs and are essential in producing legible and accurate documentation. The production of such tools can be a large investment, especially for short lived, or new and experimental languages the production of formatters is too expensive.

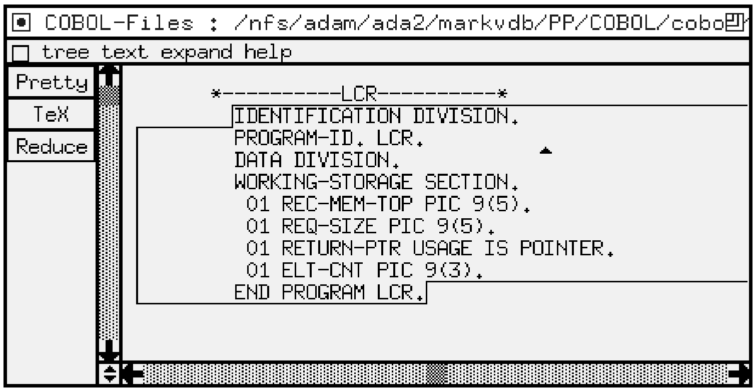
Refer to this document as: M.G.J. van den Brand and E. Visser. Generation of Formatters for Context-free Languages. Technical Report P9506, Programming Research Group, University of Amsterdam, July 1995. Available as ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1995/P9506.ps.Z

In this paper we describe a program generator that simplifies the task of building such tools considerably and makes it feasible to construct documentation tools for a language under construction. The generator produces an executable specification of a formatter for a (programming) language from its context-free grammar. A generated formatter first translates the abstract syntax tree for a program into a box expression which is a declarative description of the intended layout of the program. This language independent box expression is subsequently translated into ASCII text, TEX code or some other input format for a displaying device. These language independent box formatters do the hard work of producing a legible text. They are reused for every language for which a formatter is generated.

The generator can not possibly produce formatters that realize the desired formatting for an arbitrary language, as there is no generic criterium for good typesetting. However, the rules of the generated formatter can be adapted easily without disturbing the overall structure of its specification.



(a)



(b)

Figure 1: Structure editors (a) with unformatted and (b) with formatted Cobol text.

We concentrate in this paper on the generation of a language specific front-end for formatters. These front-ends produce declarative descriptions of the layout of a program in the box language BOX. This paper is a continuation of preliminary work described in the technical reports [Bra93a] and [Bra93b]. The translation of boxes to ASCII and TEX is described in [BV94].

IDENTIFICATION DIVISION.
PROGRAM-ID. *LCR*.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 *REC-MEM-TOP* **PIC** 9(5).
01 *REQ-SIZE* **PIC** 9(5).
01 *RETURN-PTR* **USAGE IS POINTER**.
01 *ELT-CNT* **PIC** 9(3).
END PROGRAM *LCR*.

Figure 2: TEX output for a part of the Cobol program.

Application Given the context-free grammar of the language Cobol [SS88], a (structure) editor for this language may offer a "`Pretty`" button to format a part of the text selected by the user, and a "`TeX`" button to generate TEX code for the selected part. Figure 1a shows an instantiation of such an editor with these two buttons. Pushing the "`Pretty`" button causes the selected text to be formatted as is shown in Figure 1b.

Pushing the "`TeX`" button in Figure 1a produces TEX code which yields the text shown in Figure 2. Note that keywords and variable names are set in different fonts.

Architecture Given a context-free grammar of a language L the formatter generator generates a formatter for L. This formatter traverses an abstract syntax tree of an L program and produces a box term. This box term can either be transformed to ASCII text, or it can be translated to TEX code. This process is illustrated in Figure 3.

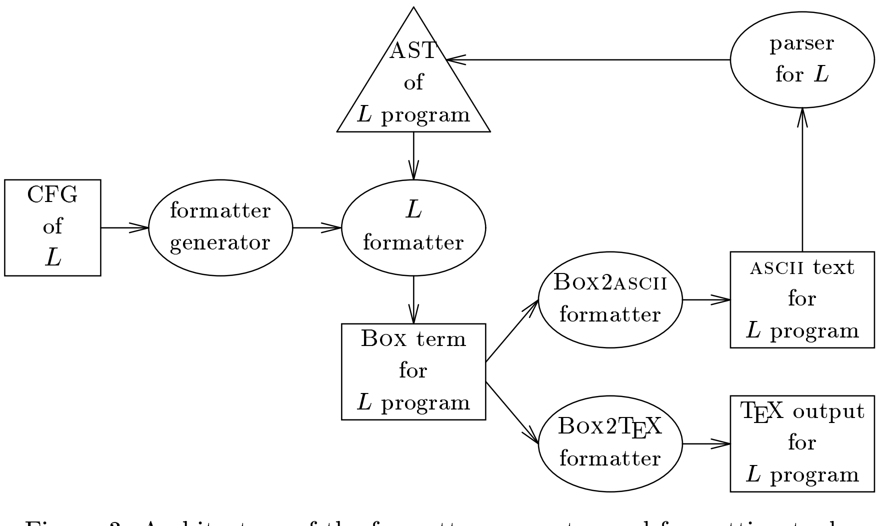


Figure 3: Architecture of the formatter generator and formatting tools.

In this paper we describe a formatter generator that produces ASF+SDF specifications [BHK89, HHKR89]. The underlying ideas are equally well applicable to imperative or functional implementations and can be expressed in any language which supports recursive functions and data structures, e.g., C. The only requirements are that the generated formatter has (direct or indirect) access to the abstract syntax tree and that it generates box terms that can be recognized by the various back-ends.

Overview We start in the next section with the generation of *unparsers* — formatters that directly translate abstract syntax trees to strings without consideration for layout. Unparsers are then extended to deal with lists. In Section 3 priority and associativity declarations are used to place brackets in the generated string in order to keep a correct correspondence with the abstract syntax tree in case of ambiguities.

In Section 4 we give a short overview of the box language and present a formatter that produces boxes. Some extensions of the box language are described, such as box operators for fonts and alignments and in Section 5 we show how these extensions can be used to adapt the generated formatter.

In Section 6 we present an elegant and flexible solution for the classical problem for formatting comments occurring in the source text.

In Section 7 formatters for the languages Cobol [SS88] and the process specification formalism PSF [MV90] are discussed as case studies of this formatting mechanism.

2 Unparsers

An unparser translates an abstract syntax tree representation of a program into a textual representation without considering layout. A parser for language L parses some input string and builds a parse tree if the string is syntactically correct. Unparsing a parse tree is simple, because all information concerning keywords, priority and associativity conflicts, etc. is explicitly available. The yield (frontier) of a parse tree gives in fact a perfect unparsing. However, a parser normally does not return a parse tree but an abstract syntax tree which contains less information. An unparser which traverses such an abstract syntax tree must know for each node which keywords to generate, and whether brackets are needed to correctly represent the priorities and associativities of operators.

Given a context-free grammar the generator derives an executable algebraic specification of the text formatter. Each production in the grammar is translated to an equation in the generated formatter. This approach is later extended to the more sophisticated solution we have used in our generator.

2.1 Correctness of unparsers

The unparser traverses an abstract syntax tree to obtain its yield. Such a tree is either built by a parser, or constructed automatically by program transformations or program generators.

A parser is a function

$$\texttt{parse}_L \;:\; \texttt{STRING} \rightarrow \texttt{TREE}$$

which takes a string and builds an abstract syntax tree if this string belongs to L, and an unparser is a function

$$\texttt{pp}_L \;:\; \texttt{TREE} \rightarrow \texttt{STRING}$$

which takes an abstract syntax tree constructed for a string over L and transforms it back into a string. We can not require

$$\texttt{pp}_L\texttt{(}\texttt{parse}_L\texttt{(}s\texttt{))} = s$$

as the unparser may modify the layout in the original string s. An unparser $\mathtt{pp}_L$ is correct with respect to $\mathtt{parse}_L$ if, given any abstract syntax tree $\triangle$ over language L,

$$\mathtt{parse}_L(\mathtt{pp}_L(\triangle)) = \triangle$$

If the parser would produce a parse tree instead of an abstract syntax tree the correctness is quite obvious, because in that case it is sufficient for the unparser to concatenate all leaves. In case of an abstract syntax tree, however, it is necessary to show that the unparser does yield a text which is syntactically correct and represents the original tree.

2.2 Derivation of unparsers

The process of generating a text formatter for an arbitrary language consists of translating context-free grammar rules into text formatting rules. Given the abstract syntax tree of some program, the text formatter produces an unparsing which consists of the yield of the corresponding parse tree. All terminals are concatenated separated by a space character.

The text formatter is generated using the context-free grammar $G = (V_N, V_T, S, P)$ of language L, where V_N is the set of nonterminals, V_T is the set of terminals, $V_N \cap V_T = \emptyset$, $S \in V_N$, and P is a finite subset of $V_N \times (V_N \cup V_T)^*$.

For each nonterminal N the text formatter contains a function

$$\mathtt{pp}N\ (N)\ \rightarrow\ \mathtt{STRING}$$

Each alternative of a production rule for a nonterminal is transformed into an equation. Consider the production "$N ::= \alpha$", all nonterminals X in α are replaced by variables, $\mathtt{var}X_i$, of the sort[1] X. The i is needed to ensure the uniqueness of the variable names in case a nonterminal occurs more than once in α. The terminals t in α are not modified.

The context-free grammar rule "$N ::= X_1 \ldots X_n$" is translated into the equation

$$\begin{aligned}\mathtt{pp}N\ (\overline{X}_1 \ldots \overline{X}_n) \ &= \ s_1 \cdot \texttt{"␣"} \cdot \ldots \cdot \texttt{"␣"} \cdot s_n \\ &\ \ \textbf{where} \\ &\ \ \overline{X}_i \ = \ \begin{cases} X_i & \text{if } X_i \in V_T, \\ \mathtt{var}X_i & \text{if } X_i \in V_N, \text{ and} \end{cases} \\ &\ \ s_i \ = \ \begin{cases} \texttt{"}X_i\texttt{"} & \text{if } X_i \in V_T, \\ \mathtt{pp}X_i(\overline{X}_i) & \text{if } X_i \in V_N \end{cases}\end{aligned}$$

Right hand sides of equations are constructed from constant strings and recursive applications of unparsing functions separated by spaces and concatenated by a binary concatenation operator $(\cdot)$ on strings.

Example 1 The production rule

```
Program ::= "begin" Decls Series "end"
```

is transformed into the specification given in Figure 4. The indices of the variable names are suppressed; *varDecls* is used instead of *varDecls*$_1$ because the nonterminal `Decls` occurred only once in the production rule. □

Before we continue our description of the unparser a few words on the interpretation of equations.

[1]When context-free grammars are used to denote algebraic signatures nonterminals are conventionally called sorts. We will use both notions when appropriate.

context-free syntax

ppProgram(Program) → String
ppDecls(Decls) → String
ppSeries(Series) → String

equations

[1] *ppProgram*(*begin varDecls varSeries end*) =
`"begin" . "␣" .` *ppDecls*(*varDecls*) `. "␣" .` *ppSeries*(*varSeries*) `. "␣" . "end"`

Figure 4: Generated text formatter.

2.3 Interpretation of equations

The ASF+SDF formalism allows the use of general user-definable syntax instead of prefix functions only. The argument, *begin varDecls varSeries end*, of the function *ppProgram* corresponds with the abstract syntax tree in Figure 5.

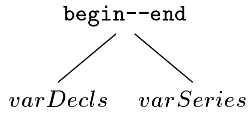


Figure 5: The abstract syntax tree for *begin varDecls varSeries end.*

Applying the function *ppProgram* to *begin varDecls varSeries end* corresponds to applying the function *ppProgram* to the abstract syntax tree of Figure 5. *ppProgram*(*begin varDecls varSeries end*) rewrites to the right hand side of Figure 6. The variable *varDecls* is matched with the abstract syntax tree with the nonterminal *Decls* as root. The function *ppDecls* applied to this abstract syntax tree can now be rewritten as well. This process is repeated for all nonterminals in the abstract syntax tree.

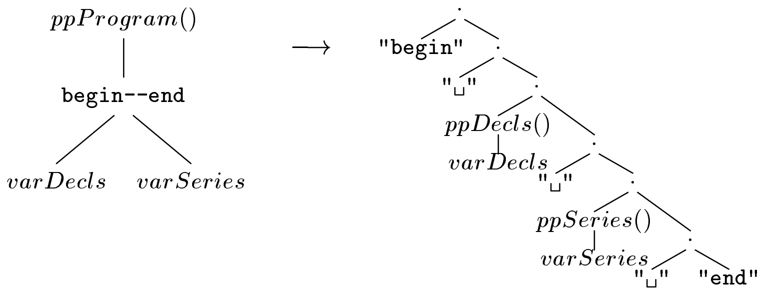


Figure 6: Interpretation of an equation.

The equations are interpreted as rewrite rules on the abstract syntax trees. Variables in the left hand side and right hand side of equations are consistently substituted with abstract syntax trees. A detailed description of this process can be found in [Wal91].

2.4 Lists

Most extended BNF notations support a notion for *lists*. In SDF [HHKR89] there are four types of lists: with and without a separator, and with zero or more, or one or more elements. Lists without a separator are denoted by $N\oplus$, where $\oplus$ = `*`, or `+`, respectively. An equivalent BNF definition of such lists is

$$
\begin{array}{lll@{\qquad\qquad}lll}
N* & ::= & & N+ & ::= & N \\
N* & ::= & N+ & N+ & ::= & N\ N+
\end{array}
$$

Lists with a separator are denoted by $\{N\ "t"\}\oplus$ with $t \in V_T$. The BNF definition of this type of list is

$$
\begin{array}{lll@{\qquad\qquad}lll}
\{N\ "t"\}* & ::= & & \{N\ "t"\}+ & ::= & N \\
\{N\ "t"\}* & ::= & \{N\ "t"\}+ & \{N\ "t"\}+ & ::= & N\ "t"\ \{N\ "t"\}+
\end{array}
$$

Although these rules for lists suggest a cons list structure, the corresponding node in the abstract syntax tree has the structure as shown in Figure 7.

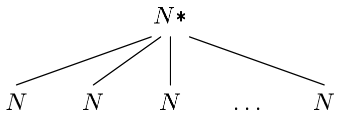


Figure 7: List node in abstract syntax tree for a list without separators.

Formatting rules for lists For each list $\{N\ "t"\}*$ the text formatter contains a function

$$\texttt{"pp}N t\texttt{0" (}\{N\ "t"\}*\texttt{)} \rightarrow \texttt{STRING}$$

defined by

$$
\begin{array}{rcl}
\texttt{pp}Nt\texttt{0()} & = & \texttt{""} \\
\texttt{pp}Nt\texttt{0(var}N\texttt{)} & = & \texttt{pp}N\texttt{(var}N\texttt{)} \\
\texttt{pp}Nt\texttt{0(var}N\ t\ \mathrm{var}Nt\texttt{+)} & = & \texttt{pp}N\texttt{(var}N\texttt{)."␣"."}t\texttt{"."␣".pp}Nt\texttt{0(var}Nt\texttt{+)}
\end{array}
$$

For each list $\{N\ "t"\}+$ the text formatter contains a function

$$\texttt{"pp}N t\texttt{1" (}\{N\ "t"\}+\texttt{)} \rightarrow \texttt{STRING}$$

defined by the equations

$$
\begin{array}{rcl}
\texttt{pp}Nt\texttt{1(var}N\texttt{)} & = & \texttt{pp}N\texttt{(var}N\texttt{)} \\
\texttt{pp}Nt\texttt{1(var}N\ t\ \mathrm{var}Nt\texttt{+)} & = & \texttt{pp}N\texttt{(var}N\texttt{)."␣"."}t\texttt{"."␣".pp}Nt\texttt{1(var}Nt\texttt{+)}
\end{array}
$$

For lists without separators the text formatter contains similar functions and equations. The list separator `"t"` and the list iterators, `*` and `+`, are used to compose the variable names which represent lists.

Example 2 Consider a production rule for a block of statements

context-free syntax
begin Decls {Stat ";"}* end → Program

The generated formatter will contain the generated equations shown in Figure 8. The variable *varStat;** in Figure 8 denotes a list with zero or more elements, whereas *varStat;+* denotes a list with at least one element. □

2.5 Implicit syntax rules

Some tools use syntax rules which are implicitly derived from context-free grammars. For example placeholders used in structure and hybrid editors. Text formatters must be able to deal with these implicitly derived syntax rules.

context-free syntax

“*ppStat;0*”({Stat “;”}*) → String

equations

[2] *ppProgram*(*begin varDecls varStat;** *end*) =
`"begin" . "␣" .` *ppDecls*(*varDecls*) `. "␣" .` *ppStat;0*(*varStat;**) `. "␣" . "end"`

[3] *ppStat;0*() = `""`

[4] *ppStat;0*(*varStat*) = *ppStat*(*varStat*)

[5] *ppStat;0*(*varStat*; *varStat;*$^+$) =
ppStat(*varStat*) `. "␣" . ";" . "␣" .` *ppStat;0*(*varStat;*$^+$)

Figure 8: Generated text formatter for lists.

Example 3 Generic structure editors allow the manipulation of incomplete programs. Placeholders are used by the structure editors to represent holes in the program text. Assume, the textual representation of a placeholder is the name of a nonterminal surrounded by the characters “<” and “>”. A placeholder can only replace syntactic constructs of the equivalent sort. For each nonterminal N the implicit syntax rule N `::= "`<N>`"` for the corresponding placeholder is derived from the grammar. □

Formatting rules for placeholders The text formatter must be able to format programs containing placeholders, therefore for each nonterminal N it contains an equation:

$$\texttt{pp}N\texttt{(<}N\texttt{>)} \;=\; \texttt{"<}N\texttt{>"}$$

For each list {N "t"}* and {N "t"}+ the text formatter contains the equations:

$$\begin{aligned}\texttt{pp}Nt\texttt{0(<}Nt\texttt{*>)} &= \texttt{"<}Nt\texttt{*>"}\\ \texttt{pp}Nt\texttt{1(<}Nt\texttt{+>)} &= \texttt{"<}Nt\texttt{+>"}\end{aligned}$$

For lists without separators the text formatter contains equivalent equations.

3 Priority and associativity

Priority and associativity definitions for (binary) operators are used to disambiguate sentences containing occurrences of these operators. Priority definitions are used to define the (relative) ordering of operators with respect to each other. The priority and associativity of the operators can be overruled by inserting brackets. Priority and associativity of operators introduce the problem whether or not the unparser should insert brackets in the formatted text. Two situations can be distinguished: the abstract syntax tree contains nodes representing brackets or these nodes do not occur in the abstract syntax tree. In the latter case the unparser needs extra information to decide whether brackets should be inserted. This is discussed in this section.

The only way to define priority and associativity in BNF is by means of introducing extra nonterminals. SDF allows a more explicit definition of priority and associativity in a separate priorities section. The priorities are expressed by a partial order on productions. Associativity can be defined for binary operators by means of attributes `left`, `right`, `assoc`, or `non-assoc` which are asociated with context-free grammar rules of the form `SORT "op" SORT -> SORT`. The associativity relation between a group of operators can be defined in the priorities section.

Example 4 Consider the following SDF definitions for arithmetic expressions.

context-free syntax

Nat	$\rightarrow$ Exp	
Exp "+" Exp	$\rightarrow$ Exp	{**left**}
Exp "−" Exp	$\rightarrow$ Exp	{**left**}
Exp "$*$" Exp	$\rightarrow$ Exp	{**left**}
"(" Exp ")"	$\rightarrow$ Exp	{**bracket**}

priorities

{**left**: Exp "+"Exp $\rightarrow$ Exp, Exp "−"Exp $\rightarrow$ Exp} $<$ Exp "$*$"Exp $\rightarrow$ Exp

All the operators are left-associative. The operator "`*`" has higher priority than the operators "`+`" and "`-`", which have the same priority and are left-associative with respect to each other.

If a context-free grammar rule is followed by the attribute `bracket`, *no* node will be created in the abstract syntax tree for an occurrence of such a rule. Given this context-free grammar the parser builds the abstract syntax tree shown in Figure 9 for the sentence `(5+5)*(4-2)`.

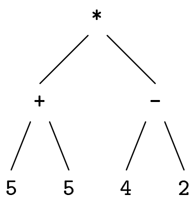


Figure 9: The abstract syntax tree for `(5+5)*(4-2)`.

A naive text formatter will produce the output `5+5*4-2` for this abstract syntax tree. So, it does not put the brackets back, although they are necessary. The abstract syntax tree for `5+5*4-2` is given in Figure 10.

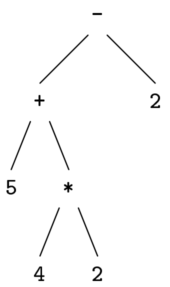


Figure 10: The abstract syntax tree for `5+5*4-2`.

Given the priority and associativity definitions the formatter should contain equations which insert these brackets automatically. □

For each nonterminal occurring in the right hand side of a context-free grammar rule which is used in the priorities section and/or is extended with an associativity attribute the text formatter contains the functions

$$\texttt{non-assoc}N\ \texttt{(}N\texttt{,}N\texttt{)} \quad \rightarrow \quad \texttt{BOOL}$$

context-free syntax

$$\begin{array}{ll} \textit{non-assocExp}(\text{Exp}, \text{Exp}) & \rightarrow \text{BOOL} \\ \textit{rightExp}(\text{Exp}, \text{Exp}) & \rightarrow \text{BOOL} \\ \textit{leftExp}(\text{Exp}, \text{Exp}) & \rightarrow \text{BOOL} \\ \textit{gtrExp}(\text{Exp}, \text{Exp}) & \rightarrow \text{BOOL} \end{array}$$

equations

[6] $\textit{non-assocExp}(\textit{varExp}_0, \textit{varExp}_1) = \textit{false}$ **otherwise**

[7] $\textit{rightExp}(\textit{varExp}_0, \textit{varExp}_1) = \textit{false}$ **otherwise**

[8] $\textit{leftExp}(\textit{varExp}_0 + \textit{varExp}_1, \textit{varExp}_2 + \textit{varExp}_3) = \textit{true}$

[9] $\textit{leftExp}(\textit{varExp}_0 - \textit{varExp}_1, \textit{varExp}_2 - \textit{varExp}_3) = \textit{true}$

[10] $\textit{leftExp}(\textit{varExp}_0 * \textit{varExp}_1, \textit{varExp}_2 * \textit{varExp}_3) = \textit{true}$

[11] $\textit{leftExp}(\textit{varExp}_0 + \textit{varExp}_1, \textit{varExp}_2 - \textit{varExp}_3) = \textit{true}$

[12] $\textit{leftExp}(\textit{varExp}_0 - \textit{varExp}_1, \textit{varExp}_2 + \textit{varExp}_3) = \textit{true}$

[13] $\textit{leftExp}(\textit{varExp}_0, \textit{varExp}_1) = \textit{false}$ **otherwise**

[14] $\textit{gtrExp}(\textit{varExp}_0 * \textit{varExp}_1, \textit{varExp}_2 + \textit{varExp}_3) = \textit{true}$

[15] $\textit{gtrExp}(\textit{varExp}_0 * \textit{varExp}_1, \textit{varExp}_2 - \textit{varExp}_3) = \textit{true}$

[16] $\textit{gtrExp}(\textit{varExp}_0, \textit{varExp}_1) = \textit{false}$ **otherwise**

Figure 11: Generated equations representing a priority and associativity declaration.

$$\begin{array}{rcl} \texttt{right}N\ (N\texttt{,}N) & \rightarrow & \texttt{BOOL} \\ \texttt{left}N\ (N\texttt{,}N) & \rightarrow & \texttt{BOOL} \\ \texttt{gtr}N\ (N\texttt{,}N) & \rightarrow & \texttt{BOOL} \end{array}$$

Example 5 The rules in the SDF definition for the expressions from Example 4 are translated to the equations presented in Figure 11. The generated text formatting rule for the `+`-operator defined in Example 4 is shown in Figure 12. The functions *l_bracketsExp* and *r_bracketsExp* in Figure 12 transform the leftmost and rightmost argument, respectively, into a string and add brackets if needed. □

If the formatter contains a context-free grammar rule like

$$\texttt{"("}\ N\ \texttt{")"} \rightarrow N\ \{\texttt{bracket}\}$$

context-free syntax

$\textit{ppExp}(\text{Exp}) \rightarrow \text{String}$

equations

$$[17]\quad \frac{\begin{array}{c}\textit{l\_addbracketsExp}(\textit{varExp}_0, \textit{varExp}_0 + \textit{varExp}_1) = \textit{varString}_0, \\ \textit{r\_addbracketsExp}(\textit{varExp}_1, \textit{varExp}_0 + \textit{varExp}_1) = \textit{varString}_1\end{array}}{\textit{ppExp}(\textit{varExp}_0 + \textit{varExp}_1) = \textit{varString}_0\ .\ \texttt{"␣"}\ .\ \texttt{"+"}\ .\ \texttt{"␣"}\ .\ \textit{varString}_1}$$

Figure 12: Generated equation for unparsing an expression.

and for N there exists also a priority or associativity definition, then the formatter contains a function

$$\texttt{addbrackets}N\ \texttt{(}N\texttt{)} \quad \rightarrow \quad \texttt{STRING}$$

for producing the brackets around an expression and the functions

$$\begin{aligned} \texttt{l\_addbrackets}N\ \texttt{(}N\texttt{,}N\texttt{)} &\quad \rightarrow \quad \texttt{STRING} \\ \texttt{r\_addbrackets}N\ \texttt{(}N\texttt{,}N\texttt{)} &\quad \rightarrow \quad \texttt{STRING} \\ \texttt{m\_addbrackets}N\ \texttt{(}N\texttt{,}N\texttt{)} &\quad \rightarrow \quad \texttt{STRING} \end{aligned}$$

The first argument of these functions is a subexpression of the expression given as second argument. The functions transform the subexpression given as the first argument into a string and put brackets around this string if there is a priority or associativity conflict between the expression and the second argument.

Example 6 The generated equations for restoring the brackets for the grammar presented in Example 4 are given in Figure 13. The functions *l_addbracketsExp* and *r_addbracketsExp* of Figure 12 are defined using the functions of Figure 11. The function *addbracketsExp* transforms its argument into a string and adds the brackets. □

4 Box formatters

The formatters defined by the methods from the previous section only produce a linear "unparsed text" corresponding to an abstract syntax tree. This text is completely unembellished and does, for instance, *not* display structure by indentation. In this section we replace string concatenation by more powerful composition operations to get a better layout.

The formatting rules will be more flexible because they allow

- the use of conditional line breaks and indentation;
- the use of different fonts, sizes, or colors;
- translation to TEX and other formalisms;
- modification of the formatting rules.

The advantage of having more abstract formatting rules based on an intermediate box language is obvious: the formatter depends no longer on a specific output medium. By defining translations from the box language to, for example ASCII, TEX, or any other output language, we are able to re-use the box formatters without modifications.

The box formatters are generated using the context-free grammar of some language. The generated formatting rules can be modified in order to obtain the desired formatting result. It is possible to re-use the modified formatting rules when a new box formatter is generated for the same language after, for instance, modifying the context-free grammar.

4.1 Box: a box language

The most elementary boxes are strings, enclosed by double quotes. Boxes can be composed, and the relative positioning of boxes is defined by operators in the box language. We distinguish 6 basic operators: H (horizontal composition), V (vertical

context-free syntax

$$
\begin{array}{ll}
\mathit{addbracketsExp}(\text{Exp}) & \to \text{String} \\
\mathit{l\_addbracketsExp}(\text{Exp, Exp}) & \to \text{String} \\
\mathit{m\_addbracketsExp}(\text{Exp, Exp}) & \to \text{String} \\
\mathit{r\_addbracketsExp}(\text{Exp, Exp}) & \to \text{String}
\end{array}
$$

equations

[18] $\mathit{addbracketsExp}(\mathit{varExp}_0) \;=\; \texttt{"("}\;.\;\texttt{"␣"}\;.\;\mathit{ppExp}(\mathit{varExp}_0)\;.\;\texttt{"␣"}\;.\;\texttt{")"}$

[19]
$$
\frac{\begin{array}{l} \mathit{non\text{-}assocExp}(\mathit{varExp}_0, \mathit{varExp}_1) \\ \vee\ \mathit{rightExp}(\mathit{varExp}_0, \mathit{varExp}_1) \\ \vee\ \mathit{gtrExp}(\mathit{varExp}_1, \mathit{varExp}_0) \end{array} = \mathit{true}}{\mathit{l\_addbracketsExp}(\mathit{varExp}_0, \mathit{varExp}_1) = \mathit{addbracketsExp}(\mathit{varExp}_0)}
$$

[20] $\mathit{l\_addbracketsExp}(\mathit{varExp}_0, \mathit{varExp}_1) \;=\; \mathit{ppExp}(\mathit{varExp}_0)$ **otherwise**

[21]
$$
\frac{\begin{array}{l} \mathit{non\text{-}assocExp}(\mathit{varExp}_0, \mathit{varExp}_1) \\ \vee\ \mathit{leftExp}(\mathit{varExp}_0, \mathit{varExp}_1) \\ \vee\ \mathit{gtrExp}(\mathit{varExp}_1, \mathit{varExp}_0) \end{array} = \mathit{true}}{\mathit{r\_addbracketsExp}(\mathit{varExp}_0, \mathit{varExp}_1) = \mathit{addbracketsExp}(\mathit{varExp}_0)}
$$

[22] $\mathit{r\_addbracketsExp}(\mathit{varExp}_0, \mathit{varExp}_1) \;=\; \mathit{ppExp}(\mathit{varExp}_0)$ **otherwise**

[23]
$$
\frac{\mathit{gtrExp}(\mathit{varExp}_1, \mathit{varExp}_0) = \mathit{true}}{\mathit{m\_addbracketsExp}(\mathit{varExp}_0, \mathit{varExp}_1) = \mathit{addbracketsExp}(\mathit{varExp}_0)}
$$

[24] $\mathit{m\_addbracketsExp}(\mathit{varExp}_0, \mathit{varExp}_1) \;=\; \mathit{ppExp}(\mathit{varExp}_0)$ **otherwise**

Figure 13: Generated equations for adding brackets.

composition), HV (horizontal and/or vertical composition), HOV (horizontal or vertical composition), I (indentation), and WD (invisible box with the same width as some visible box).

The functionality of the box operators is described by the following diagrams. For simplicity, we only show operators with two or three arguments. Generalization to the case of n arguments is straightforward. The diagrams for the H and V operators are obvious.

$$\mathsf{H}[\ \boxed{B_1}\boxed{B_2}\ ] \ = \ \boxed{B_1}\boxed{B_2}$$

$$\mathsf{V}[\ \boxed{B_1}\boxed{B_2}\ ] \ = \ \begin{array}{c}\boxed{B_1}\\ \boxed{B_2}\end{array}$$

The formatting of the HV operator depends on the amount of space left for the argument boxes. For each argument box is considered whether this box fits in the remaining space or not. In case of three argument boxes this yields, for instance, four different ways of formatting. We express the formatting in terms of H and V operators.

$$\begin{array}{rcl}\mathsf{HV}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] & = & \mathsf{H}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] \text{ or}\\ & & \mathsf{V}[\ \mathsf{H}[\ \boxed{B_1}\boxed{B_2}\ ]\ \boxed{B_3}\ ] \text{ or}\\ & & \mathsf{V}[\ \boxed{B_1}\ \mathsf{H}[\ \boxed{B_2}\boxed{B_3}\ ]\ ] \text{ or}\\ & & \mathsf{V}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ]\end{array}$$

The formatting of the HOV operator also depends on the amount of space left for the boxes. For *all* argument boxes is considered whether they fit in the remaining space or not. We also use the H and V operators to express the functionality of this operator.

$$\begin{array}{rcl}\mathsf{HOV}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] & = & \mathsf{H}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] \text{ or}\\ & & \mathsf{V}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ]\end{array}$$

The functionality of the I operator depends on the surrounding box operator. This operator has only effect in a vertical setting.

$$\mathsf{V}[\ \boxed{B_1}\ \mathsf{I}[\ \boxed{B_2}\ ]\ \boxed{B_3}\ ] \ = \ \begin{array}{l}\boxed{B_1}\\ \quad\boxed{B_2}\\ \boxed{B_3}\end{array}$$

So, if the boxes are formatted horizontally the I operator is ignored.

$$\mathsf{H}[\ \boxed{B_1}\ \mathsf{I}[\ \boxed{B_2}\ ]\ \boxed{B_3}\ ] \ = \ \boxed{B_1}\boxed{B_2}\boxed{B_3}$$

The WD operator translates a box into an empty box with dimensions equal to those of its arguments.

$$\mathsf{V}[\ \boxed{B_1}\ \mathsf{H}[\ \mathsf{WD}[\ \boxed{B_1}\ ]\ \boxed{B_2}\ ]\ ] \ = \ \begin{array}{l}\boxed{B_1}\\ \phantom{\boxed{B_1}}\boxed{B_2}\end{array}$$

The BOX syntax A box term consists of a box operator, zero or more spacing options, and followed by zero or more boxes.

context-free syntax

String	→ Box
Box*	→ Box-List
“H” S-Options “[” Box-List “]”	→ Box
“V” S-Options “[” Box-List “]”	→ Box
“HV” S-Options “[” Box-List “]”	→ Box
“HOV” S-Options “[” Box-List “]”	→ Box
“I” S-Options “[” Box “]”	→ Box
“WD” “[” Box “]”	→ Box

The spacing options are used to adapt the horizontal, vertical, or indentation offset between boxes. Not every combination of spacing option and box operator makes sense. For instance the modification of the vertical offset in combination of the H operator is not useful.

context-free syntax

“*hs*”	→ Spacing-Symbol
“*vs*”	→ Spacing-Symbol
“*is*”	→ Spacing-Symbol
Spacing-Symbol “=” INT	→ S-Option
S-Option*	→ S-Options

If *no* spacing options are specified, default values are used. The default values for *hs*, *vs*, and *is* are 1, 1, and 2, respectively.

4.2 Generating boxes

Given the box language we are able to modify the generator presented in Section 2 in such a way that the resulting formatter is no longer based on strings but on boxes. It is not necessary to describe the entire box formatter generator because issues like placeholders and “brackets” remain the same. The process of generating a box formatter for an arbitrary language is done by classifying each context-free grammar rule and by generating the formatting rules in the text formatter for each grammar rule using this classification information.

Classification of context-free grammar rules The box language offers the facility to place the members of a context-free grammar rule in horizontal, vertical, or some mixed way. The way the members are placed depends on a number of circumstances, e.g. the structure of the grammar rule, the surroundings where the rule is applied, etc. A conditional must be formatted in a different way than an expression, e.g.

```
if
  <Exp>
then
  <Series>
else
  <Series>
fi
```

versus

```
<Exp> + <Exp>
```

For the sake of brevity and simplicity we give only a rough classification of context-free grammar rules. Only two different types will be used: *indented* and *non-indented*. The name "indented" stems from the fact that some language constructs are more indented than others to increase the readibility. The name "non-indented" represents *all* other language constructs. The context-free grammar rule is of the type

- indented, if it has one of the following patterns: $t\ N\ (t\ N)*\ t$ or $t\ N\ (t\ N)+$; and
- non-indented, if it does not satisfy one of the patterns given above.

Observe that most control structures in programming languages will be classified as "indented". This classifiation can be decided by means of a simple finite automaton, which scans the production rule and yields its classification as result. This classification is performed by the formatter generator, it uses this information to decide which box operators must be used in the formatting rule of the corresponding context-free grammar rule. Our generator distinguishes more types in the classification and uses also more sophisticated heuristics to classify the context-free grammar rules.

4.2.1 Derivation of box formatting rules

For each nonterminal N in the context-free grammar the formatter contains a function.

$$\mathtt{pp}N\ (N)\ \rightarrow\ \mathtt{BOX}$$

We present the formatting rules for indented and non-indented context-free grammar rules and for lists.

Formatting rules for grammar rules of type "indented" Each grammar rule N `::=` $X_1 \ldots X_n$, which is of type indented, results in an equation

$$\begin{aligned} \mathtt{pp}N\ (\overline{X}_1 \ldots \overline{X}_n) &= \mathsf{V}\ \mathtt{[}b_1\ \ldots\ b_n\mathtt{]} \\ &\ \textbf{where} \\ \overline{X}_i &= \begin{cases} X_i & \text{if } X_i \in V_T, \\ \mathtt{var}X_i & \text{if } X_i \in V_N, \text{ and} \end{cases} \\ b_i &= \begin{cases} \mathtt{"}X_i\mathtt{"} & \text{if } X_i \in V_T, \\ \mathsf{I}\ \mathtt{[pp}X_i\mathtt{(}\overline{X}_i\mathtt{)]} & \text{if } X_i \in V_N \end{cases} \end{aligned}$$

The right hand sides of the equations consist of strings, the most elementary boxes, and (recursive) function calls of $\mathtt{pp}N$ $\mathtt{(var}N\mathtt{)}$ which are indented via the I operator. The boxes in the right hand side are combined into one V box.

Formatting rules for grammar rules of type "non-indented" A grammar rule of type non-indented results in an equation

$$\begin{aligned} \mathtt{pp}N\ (\overline{X}_1 \ldots \overline{X}_n) &= \mathsf{HV}\ \mathtt{[}b_1\ \ldots\ b_n\mathtt{]} \\ &\ \textbf{where} \\ \overline{X}_i &= \begin{cases} X_i & \text{if } X_i \in V_T, \\ \mathtt{var}X_i & \text{if } X_i \in V_N, \text{ and} \end{cases} \\ b_i &= \begin{cases} \mathtt{"}X_i\mathtt{"} & \text{if } X_i \in V_T, \\ \mathtt{pp}X_i\mathtt{(}\overline{X}_i\mathtt{)} & \text{if } X_i \in V_N \end{cases} \end{aligned}$$

The boxes in the right hand side are combined into one HV box.

context-free syntax

$$
\begin{array}{ll}
\textit{ppDecls}(\text{Decls}) & \rightarrow \text{Box} \\
\textit{ppId-Type}(\text{Id-Type}) & \rightarrow \text{Box} \\
\text{“}\textit{ppId-Type,0}\text{”}(\{\text{Id-Type “,”}\}*) & \rightarrow \text{Box} \\
\text{“}\textit{ppId-Type,0e}\text{”}(\{\text{Id-Type “,”}\}*) & \rightarrow \text{Box-List}
\end{array}
$$

equations

[25] $\textit{ppDecls}(\textit{declare varId-Type},^{*};) =$

$\mathsf{V}\ [\texttt{"declare"}$
$\quad \mathsf{I}\ [\textit{ppId-Type,0}(\textit{varId-Type},^{*})]$
$\quad \texttt{";"}]$

[26] $\textit{ppId-Type,0}() = \mathsf{H}\ []$

[27] $\textit{ppId-Type,0}(\textit{varId-Type},^{+}) = \mathsf{V}\ [\textit{ppId-Type,0e}(\textit{varId-Type},^{+})]$

[28] $\textit{ppId-Type,0e}(\textit{varId-Type}) = \textit{ppId-Type}(\textit{varId-Type})$

[29]
$$
\frac{\textit{ppId-Type,0e}(\textit{varId-Type},^{+}) = \textit{varBox-List}}{\begin{array}{l}\textit{ppId-Type,0e}(\textit{varId-Type},\ \textit{varId-Type},^{+}) = \\ \mathsf{H}\ \textit{hs} = 0[\textit{ppId-Type}(\textit{varId-Type})\ \texttt{","}]\ \textit{varBox-List}\end{array}}
$$

Figure 14: Generated box formatter.

Formatting rules for lists For each list $\{N\ \texttt{"}t\texttt{"}\}*$ the formatter contains the functions

$$
\begin{array}{l}
\texttt{"pp}Nt\texttt{0"}\ (\{N\ \texttt{"}t\texttt{"}\}*)\ \rightarrow \texttt{BOX} \\
\texttt{"pp}Nt\texttt{0e"}\ (\{N\ \texttt{"}t\texttt{"}\}*)\ \rightarrow \texttt{BOX}
\end{array}
$$

defined as

$$
\begin{array}{rcl}
\texttt{pp}Nt\texttt{0()} & = & \mathsf{H}\ \texttt{[]} \\
\texttt{pp}Nt\texttt{0(var}Nt\texttt{1)} & = & \mathsf{V}\ \texttt{[pp}Nt\texttt{0e(var}Nt\texttt{1)]} \\
\\
\texttt{pp}Nt\texttt{0e(var}N\texttt{)} & = & \texttt{pp}N\texttt{(var}N\texttt{)} \\
\texttt{pp}Nt\texttt{0e(var}N\ t\ \text{var}Nt\texttt{1)} & = & \mathsf{H}\ \texttt{hs=0 [pp}N\texttt{(var}N\texttt{) "}t\texttt{"] pp}Nt\texttt{0e(var}Nt\texttt{1)}
\end{array}
$$

For the other syntactic forms of lists the formatter contains similar sets of functions and equations.

Example 7 Consider the context-free grammar rule
context-free syntax
declare {Id-Type “,”}∗ “;” → Decls
which is transformed into the specification presented in Figure 14. This rule is classified as type “indented”. □

4.3 Modular structure of the box formatter

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 formatter module. The import graph for these generated modules is the same as for the context-free grammar. This allows an incremental approach to the generation of formatters. If one of the modules of the context-free grammar changes, a formatter needs to be regenerated only for this module.

5 Fine-tuning of generated formatters

Formatters can be adapted by the specification writer in order to obtain results better suited for the language at hand. Some adaptations deal with the layout between terminals and are output medium independent, others are output medium specific, such as font changes. Adapting box formatting rules is demonstrated by giving an example of the desired layout scheme and describing which changes are necessary to obtain this scheme. In general, it may be necessary to extend the kernel box language to obtain the desired result.

It is possible to modify a formatter for a language L in such a way that it produces code for language L'. However we will not describe the possibilities of using the formatter as a kind of "translator", for example to transform Pascal code into C code, since this is not a task for an unparser but for a translator.

Example 8 Consider again the context-free grammar rule
context-free syntax
declare {Id-Type ","}* ";" → Decls

The corresponding box formatting rule is given in Figure 14.

For the program text "`declare x:int;`", the formatter produces the following ASCII result

```
declare
  x : int
;
```

whereas

```
declare
  x : int;
```

is more "natural". This is achieved by changing the formatting rule into

[30] *ppDecls*(*declare varId-Type*,*;) =
V ["declare"
I [H $hs = 0$[*ppId-Type*,*0*(*varId-Type*,*) ";"]]]

□

The amount of polishing strongly depends on the structure of the language. A more elaborate example showing how to tune generated formatters can be found in Section 7.1.

5.1 Columns

Although the box language only uses 6 operators it is possible to specify complex layout schemes, such as columns. First we give an example of how to obtain columns using the plain box language. It turns out that this approach is too tedious for the specification writer and too much directed towards text formatting. It is based on the assumption that *all* characters have the same width. A translation of boxes describing columns to TeX code produces output which does not reflect the desired result. The introduction of special box operators for columns solves these problems.

Example 9 Suppose we have a list of identifiers with their types, and we want them to be formatted as

```
declare
  input  : natural,
  output : natural,
  repnr  : natural,
  rep    : natural
;
```

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

Writing a specification to put things in columns has a number of disadvantages. The first one 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. The computational overhead can be rather significant for large lists. The second disadvantage is the overhead for the specification writer to specify this calculation of columns. In particular, if things must be put in more than two columns. The final disadvantage is that this approach is directed towards text formatting and is not back-end independent.

These disadvantages can be eliminated by introducing special box operators dealing with columns. □

Alignments The introduction of an alignment operator is necessary because during the translation of boxes to TEX we can no longer make the assumption that fonts are non-proportional. Furthermore, from the users viewpoint the use of an alignment operator is much easier than specifying the explicit calculation of columns.

Columns are implemented by means of two box operators: A and R. The arguments of the A operator are R boxes. Each R box represents a row. All R boxes should have the same number of arguments, equal to the desired number of columns. For each column the user can specify whether the alignment should be left, right, or centered. The number of alignment indications should be equal to the number of columns. Each alignment indication can be extended with spacing options, which are transferred to the columns to be formatted.

context-free syntax

"A" A-Options S-Options "[" Box-List "]"	$\rightarrow$ Box
"R" "[" Box-List "]"	$\rightarrow$ Box
"*l*" S-Options	$\rightarrow$ A-Option
"*c*" S-Options	$\rightarrow$ A-Option
"*r*" S-Options	$\rightarrow$ A-Option
"(" {A-Option ","}$*$ ")"	$\rightarrow$ A-Options

The introduction of box operators for alignment requires of course the adaptation of the various back-ends. The text back-end translates the alignment operator to box operators in the plain box language. The problem of traversing the list twice is solved by storing size information in the boxes and by processing the columns one by one. The TEX back-end implements the alignment operator by `\halign`.

Example 10 The use of the alignment operators is demonstrated in Figure 15. □

The declarations of Example 8 are formatted as shown in Figure 16 using the TEX back-end.

context-free syntax

"*ppId-Type,0*"({Id-Type ","}*) → Box

"*ppId-Type,0e*"({Id-Type ","}*) → Box-List

equations

[31] $ppId\text{-}Type{,}0(varId\text{-}Type_{,0}) \;=\; \mathsf{A}\;(l\;,\;c\;,\;l\;)\;[ppId\text{-}Type{,}0e(varId\text{-}Type_{,0})]$

[32] $ppId\text{-}Type{,}0e() \;=\; \texttt{""}$

[33] $ppId\text{-}Type{,}0e(varId : varType) \;=\; \mathsf{R}[ppId(varId)\;\texttt{":"}\;ppType(varType)]$

[34] $$\frac{ppId\text{-}Type{,}0e(varId\text{-}Type_{,1}) = varBox\text{-}List}{\begin{array}{l} ppId\text{-}Type{,}0e(varId : varType,\; varId\text{-}Type_{,1}) = \\ \mathsf{R}[ppId(varId)\;\texttt{":"}\;\mathsf{H}\;hs = 0[ppType(varType)\;\texttt{","}]] \\ varBox\text{-}List \end{array}}$$

Figure 15: Specification of columns using the alignment operators.

declare

input : natural,
output : natural,
repnr : natural,
rep : natural

;

Figure 16: TEX output for the declarations.

5.2 Fonts

The plain box language offers no facilities to manipulate fonts. It is not necessary to have such facilities in case of the ASCII back-end. TEX allows different fonts and therefore we extend the box language with operators to manipulate fonts. We have used the same strategy as with alignments. We extend the plain box language with new box operators for specifying fonts. The font extension is independent of the alignment extension.

Font operators affect only the strings in boxes, they do not influence the structure of boxes.

Fonts are characterized by a number of parameters. We distinguish the following parameters: *fn* (font name), *fm* (font family), *se* (font series), *sh* (font shape), *sz* (font size), and *cl* (font color). Each of these parameters will be translated to the appropiate TEX code.

context-free syntax

Font-Param "=" INT	→ Font-Option
Font-Param "=" ID	→ Font-Option
Font-Option*	→ Font-Options
"F" Font-Options	→ Font-Operator
Font-Operator "[" Box "]"	→ Box
Font-Operator "(" Box-List ")"	→ Box-List
"*fn*"	→ Font-Param
"*fm*"	→ Font-Param
"*se*"	→ Font-Param
"*sh*"	→ Font-Param
"*sz*"	→ Font-Param
"*cl*"	→ Font-Param

A number of frequently used fonts are predefined by means of special font operators. They can be considered as a kind of abbreviation, because they can also be defined using font options. Special font operators are introduced to mark keywords, variables, numbers, mathematical symbols, and comments. The ESC font operator is used as an escape mechanism to obtain special symbols in the formatted text.

context-free syntax

"KW"	→ Font-Operator
"VAR"	→ Font-Operator
"NUM"	→ Font-Operator
"MATH"	→ Font-Operator
"COMM"	→ Font-Operator
"ESC"	→ Font-Operator

The various back-ends have to be adapted in order to deal with the font box operators. The text back-end ignores the font operators and their options, and uses only the string argument to derive plain ASCII text. The TEX back-end translates the font operators into TEX macros to obtain the desired effect in the output.

Some of these font operators are used by the generator of box formatters, for instance, to mark keywords by making them consequently bold.

By using different fonts it is possible to mark defined and undefined variables in a program. This can be achieved by letting the box formatter rules use context sensitive information, such as type checking information, while constructing the box expression for some program in order to decide which font operator to use.

6 Formatting comments

Syntax directed editors in generated programming environments manipulate text as well as the corresponding abstract syntax tree. The problem with text formatting in this type of editors is how to restore comments occurring in the original text, when the abstract syntax tree is transformed into text. A typical scenario is as follows. A parser processes a program text and builds an abstract syntax tree for it. A text formatter processes the abstract syntax tree using generated and/or user-defined formatting rules and produces text again. The difference between the input of the parser and the output of the formatter is layout. Comments in the input text will be lost in the output text if comments are treated as layout, but this is unacceptable from the users' point of view.

There are two different ways of dealing with comments:

- 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 is notoriously difficult to handle during formatting [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.

One approach to solve the problem of restoring comments is to attach them to nodes in the abstract syntax tree [RW81]. During formatting, the comments are regenerated when processing 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 formatted text.

In our approach, the position of a comment in the original text is used to determine its position in the formatted text. Boxes are constructed on the basis of the abstract syntax tree and a set of formatting rules. Boxes have no knowledge of the original abstract syntax tree and can be modified easily. This offers us a possibility for restoring comments. The box term constructed for a given abstract syntax tree is "compared" with the original text. The differences, thus the comments, are then inserted in the box. This approach is based on the following assumptions:

- whitespace consists of spaces and newlines only and will be ignored.

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

- during formatting *no* lexical symbols are changed, for instance, "`begin`" may not be replaced by "`(`". The box contains only original "tokens".

Comments according to this style can be specified, for instance, in the following way

lexical syntax

"%%"~[\n]*"\n" → LAYOUT
"%"~[%\n]+"%" → LAYOUT

The first rule describes comments which start with some begin marker ("%%") and end at the end of line. The second rule describes comments which start with some begin marker ("%") and end also with an end marker, in this case no newlines and "%"-characters are allowed within the comment string. Other forms of comments are also allowed, their processing is similar to the processing of these two forms.

6.1 Comments

The comments occurring in the original text must be restored in the formatted text. The way the comments are formatted depends on the exact place where the comments occur in the original text. We distinguish three different types of comments:

- comments which occur after some symbol (HPAR);
- comments which occur on a line which does not contain any symbol except the comment text, but the comment symbol does not occur in the first column (VPAR); and
- comments which start in the first column of a new line (PAR).

Each of these comment types is represented by a separate box operator. The comment box operators have no space options, instead they have two strings which represent the begin and end marker of the comment.

context-free syntax

"HPAR" String String "[" Box-List "]" → Box
"VPAR" String String "[" Box-List "]" → Box
"PAR" String String "[" Box-List "]" → Box

Back-ends should take care of formatting the comment box in a correct manner.

6.2 Algorithm for restoring comments

We will give only a brief description of the algorithm for restoring comments. Given the box term constructed for some abstract syntax tree and the original text, this text is scanned using the strings in the box term. Whitespace, spaces and newlines, is ignored. As soon as a begin marker of a comment is encountered the comment is scanned and translated into a box term. The selection of the appropriate box operator, HPAR, VPAR, or PAR, depends on the exact place of the comment in the original text.

We will demonstrate the functionality of this algorithm by means of a simple example. Suppose the text

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

has to be formatted. Given the abstract syntax tree of this program the formatter generates the box

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

Given the original text and this box the algorithm transforms it into

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

This results in the following ASCII output:

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

7 Case studies

7.1 Developing a Cobol formatter

We demonstrate the approach of tuning generated formatters by generating a formatter for the programming language Cobol [SS88], and modifying some of the generated box formatting rules. The programming language Cobol has a very simple syntax, but imposes severe constraints on the layout. We will show how to tune a formatter by modifying the box formatting rules generated for the following context-free grammar rules

context-free syntax

ID-DIV OPT-DATA-DIV OPT-PROC-DIV OPT-PROG-END	→ COBOL-PROGRAM
"END" "PROGRAM" ID "."	→ OPT-PROG-END
	→ OPT-PROG-END
"IDENTIFICATION" "DIVISION" "." "PROGRAM-ID" "." ID "."	→ ID-DIV
"DATA" "DIVISION" "." OPT-STORAGE-SEC OPT-LINKAGE-SEC	→ OPT-DATA-DIV
	→ OPT-DATA-DIV
"WORKING-STORAGE" "SECTION" "." DATA-DESCS	→ OPT-STORAGE-SEC
	→ OPT-STORAGE-SEC

The Cobol program in the editor of Figure 17 is an incorrectly formatted Cobol program. This Cobol program is formatted using the automatically generated formatting rules. This editor is extended with a number of buttons. The "Pretty"-button formats the text selected by the user: the old text is replaced by the formatted text. The "TeX"-button translates the text selected by the user to TeX code.

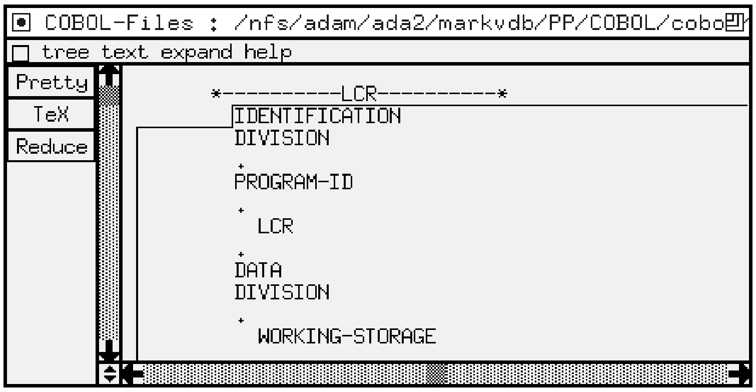


Figure 17: A (incorrectly) formatted Cobol program.

Equation 38 in Figure 18 is changed in equation 43 given in Figure 19, by introducing a few extra H operators with the spacing option `hs=0` and removing the I operator in the right hand side of the equation. It is also necessary to remove the keyword "**otherwise**" in the equation, this keyword marks the equation as a so-called *default* equation. The default equations are tried if none of the other equations succeeds. The modifications in the right hand side of this equation are in fact minor, however pushing the "Pretty"-button clearly shows the differences in Figure 20.

The equations 36, 39, and 41 given in Figure 18 are changed into the equations 44, 45, and 46 given in Figure 21 by introducing some extra H operator (with or without a spacing option `hs=0`) and removing a few I operators. Pushing the "Pretty"-button produces the result shown in Figure 22. The numbered Cobol lines in the Figures 20 and 22 are formatted as well, the modifications to obtain this result will not be discussed.

The Cobol formatter was used as a back-end for a compiler which translated a special-purpose specification formalism for financial applications [Res94] into Cobol code. The output of this compiler could not be processed by the Cobol compiler because of layout conventions. The Cobol formatter was used as a filter to get the layout correct.

7.2 The PSF typesetter

Another application of the box formatter generator was the production of a typesetter for the process specification formalism PSF [MV90]. This typesetter was used to typeset all specifications in [Bru95]. There are a couple of interesting features related to this typesetter not specific for PSF. Firstly, comments occurring in the original specification were not lost in the typeset text, because they were essential for understanding the specifications. Secondly, the author wanted to "hide" (leave out) parts of PSF specifications that were not relevant for his presentation. This can be considered as a kind of *elision*. This latter feature has been implemented by extending the PSF grammar with a new type of comments.

lexical syntax

"..."E-CHAR+"..."	→ LAYOUT
~[.] "."~[.]	→ E-CHAR
~[.]	→ E-CHAR

equations

[35] *ppCOBOL-PROGRAM*($var\_ID\text{-}DIV{:}_0$ $var\_OPT\text{-}DATA\text{-}DIV{:}_1$ $var\_OPT\text{-}PROC\text{-}DIV{:}_2$ $var\_OPT\text{-}PROG\text{-}END{:}_3$) =

HV [*ppID-DIV*($var\_ID\text{-}DIV{:}_0$) **otherwise**
ppOPT-DATA-DIV($var\_OPT\text{-}DATA\text{-}DIV{:}_1$)
ppOPT-PROC-DIV($var\_OPT\text{-}PROC\text{-}DIV{:}_2$)
ppOPT-PROG-END($var\_OPT\text{-}PROG\text{-}END{:}_3$)]

[36] *ppOPT-PROG-END*(END PROGRAM $var\_ID{:}_2$.) =
V [KW["END"] KW["PROGRAM"] I [*ppID*($var\_ID{:}_2$)] "."] **otherwise**

[37] *ppOPT-PROG-END*() = HV [] **otherwise**

[38] *ppID-DIV*(IDENTIFICATION DIVISION . PROGRAM-ID . $var\_ID{:}_5$.) =
V [KW["IDENTIFICATION"] **otherwise**
KW["DIVISION"] "."
KW["PROGRAM-ID"] "."
I [*ppID*($var\_ID{:}_5$)] "."]

[39] *ppOPT-DATA-DIV*(DATA DIVISION . $var\_OPT\text{-}STORAGE\text{-}SEC{:}_3$ $var\_OPT\text{-}LINKAGE\text{-}SEC{:}_4$) =
V [KW["DATA"] KW["DIVISION"] "." **otherwise**
I [*ppOPT-STORAGE-SEC*($var\_OPT\text{-}STORAGE\text{-}SEC{:}_3$)]
I [*ppOPT-LINKAGE-SEC*($var\_OPT\text{-}LINKAGE\text{-}SEC{:}_4$)]]

[40] *ppOPT-DATA-DIV*() = HV [] **otherwise**

[41] *ppOPT-STORAGE-SEC*(WORKING-STORAGE SECTION . $var\_DATA\text{-}DESCS{:}_3$) =
V [KW["WORKING-STORAGE"] **otherwise**
KW["SECTION"] "."
I [*ppDATA-DESCS*($var\_DATA\text{-}DESCS{:}_3$)]]

[42] *ppOPT-STORAGE-SEC*() = HV [] **otherwise**

Figure 18: Some of the generated formatting rules for Cobol.

[43] *ppID-DIV*(IDENTIFICATION DIVISION . PROGRAM-ID . $var\_ID{:}_5$.) =
V [H [KW["IDENTIFICATION"] H $hs = 0$[KW["DIVISION"] "."]]
H [H $hs = 0$[KW["PROGRAM-ID"] "."]
H $hs = 0$[*ppID*($var\_ID{:}_5$) "."]]]

Figure 19: The first modification in the generated formatting rules of Cobol.

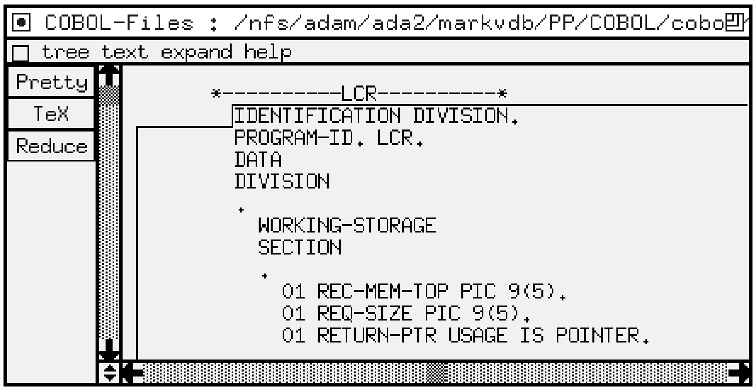


Figure 20: A slightly better formatted Cobol program.

[44] *ppOPT-PROG-END*(END PROGRAM *var_ID*:$_2$.) =
H [KW["END"] KW["PROGRAM"] H $hs = 0$[*ppID*(*var_ID*:$_2$) "."]]

[45] *ppOPT-DATA-DIV*(DATA DIVISION . *var_OPT-STORAGE-SEC*:$_3$ *var_OPT-LINKAGE-SEC*:$_4$) =
V [H [KW["DATA"] H $hs = 0$[KW["DIVISION"] "."]]
ppOPT-STORAGE-SEC(*var_OPT-STORAGE-SEC*:$_3$)
ppOPT-LINKAGE-SEC(*var_OPT-LINKAGE-SEC*:$_4$)]

[46] *ppOPT-STORAGE-SEC*(WORKING-STORAGE SECTION . *var_DATA-DESCS*:$_3$) =
V [H [KW["WORKING-STORAGE"] H $hs = 0$[KW["SECTION"] "."]]
I $is = 1$[*ppDATA-DESCS*(*var_DATA-DESCS*:$_3$)]]

Figure 21: The other modified formatting rules of Cobol.

Comments of this type were not restored but translated to "..." by the TEX back-end. This feature could only be implemented by extending the specification of the generated PSF formatter. An example of a typeset PSF specification can be found in Figure 23.

PSF is a modular specification formalism. To keep track of the dependencies between the various modules a cross referencing mechanism is added to the PSF typesetter. This was implemented by extending the box language.

8 Related work

The Box language we developed is influenced by the work of Oppen [Opp80]. Oppen's algorithm receives lists of lexical tokens with special characters to direct the pretty printing. These lists of lexical tokens, blocks in Oppen's terminology, are comparable to boxes. There are two categories of special characters:

- *bracket characters* to delimit a syntactic construct, such as an assignment or a series;

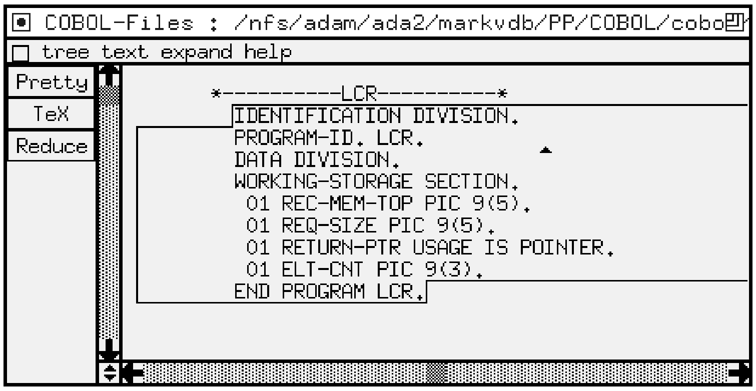


Figure 22: The expected formatted Cobol program.

- *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 syntactic construct and a close bracket to denote its end. The blank character denotes a possible break point. 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.

The Box language is based on PPML [MCC86]. PPML is a programming language for constructing pretty printers. A PPML progam is interpreted by a PPML formatting machine. PPML distinguishes 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 Box and PPML.

- The order in which rules are specified in PPML is relevant, in Box there is only one formatting rule for each context-free grammar rule.
- PPML operators 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 Box it is possible to specify only one of the parameters.
- PPML operators are a mixture of prefix and infix operators. In Box all the operators are prefix operators.

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

```
process module PointToPointChannel
begin
  parameters
    SettingsParameter
    begin
      functions
        pcerror : → BOOLEAN
        pcloss  : → BOOLEAN
        pcsize  : → NATURAL
    end SettingsParameter
  ...
  imports
    Sides,
    Queues
      {Queue-Parameters bound by [Q-ELEMENT → FR, queue-nil → ce]
         to GenericFrames}
  ...
  variables
    Q          : → QUEUE
    cin, cout  : → SIDE
  definitions
  -- initialisation
  P-CHANNEL(cin, cout) = P-CHANNEL(cin, cout, empty-queue)
  P-CHANNEL(cin, cout, Q) =
      [or(eq(pcsize, nat(^0)), lt(length(Q), pcsize)) = true] →
        sum(F in FRset,
            read-SC(cin, F) · P-CHANNEL(cin, cout, enqueue(F, Q)))
    + [or(eq(pcsize, nat(^0)), lt(length(Q), pcsize)) = false] →
        sum(F in FRset,
            read-SC(cin, F) · P-CHANNEL(cin, cout, enqueue(F, dequeue(Q))))
    + [gt(length(Q), nat(^0)) = true] → send-CR(cout, serve(Q)) ·
      P-CHANNEL(cin, cout, dequeue(Q))
    + [and(gt(length(Q), nat(^0)), pcerror) = true] → send-CR(cout, ce) ·
      P-CHANNEL(cin, cout, dequeue(Q))
    + [and(gt(length(Q), nat(^0)), pcloss) = true] → lost(cin, cout) ·
      P-CHANNEL(cin, cout, dequeue(Q))
end PointToPointChannel
```

Figure 23: A typeset PSF specification.

Pretty [Vos90] was a (first) attempt to define formatting algebraically. The formatting language Pretty offered a considerable number of (infix) operators over boxes.

Both formatting languages PPML and Pretty offer the possibility of elision, viz. hiding pieces of the formatted text starting at a certain predefined level in order to increase the readability. Elision can be implemented in several ways in Box, one possible manner was discussed in Section 7.2.

The specification language SSL of the Synthesizer Generator [RT89] offers a special pretty print formalism. It is strongly based on attribute grammars and can produce the formatted text in one pass without backtracking. 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] offers the user of a generated environment a tool to modify the pretty print rule for a specific language construct. This can be considered as a form of "pretty-printing-by-example".

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

9 Conclusions

In this paper we have presented a method for generating formatting tools for arbitrary languages based on their description by means of a context-free grammar. A generated formatter is expressed by an algebraic specification of a family of functions that compositionally map the constructs of a language to box expressions. The box expression resulting from translating a program is a declarative description of the layout of that program independent of the programming language. It can be translated by various back-end formatters to ASCII, TeX or some other displaying device. The formatters reconstruct brackets and comments that are thrown away by the parser. Extensions of the Box language provide powerful constructs for fonts and alignments.

The formatter generator presented in this paper, in combination with the formatters for boxes [BV94], is a practical support tool for the construction of documentation tools. It has been applied to generated parts of programming environments for various languages. To name a few: error messages for a SEAL type checker [Koo94]; a formatter for the specification language μCRL; a pretty printer for Cobol used as a back-end of a Cobol generator [Res94]; a typesetter for the process specification formalism PSF used to typeset the specifications in [Bru95]. The latter two tools were discussed in Section 7. The generated formatters save a considerable amount of typesetting time and the generation of the formatters save a considerable amount of tool specification time. This is especially important for the evaluation of new and experimental languages.

Both formatter generator and the generated formatters are algebraically specified in the algebraic specification formalism ASF+SDF [BHK89, HHKR89]. Because of the high level of specification and the support provided by the interactive ASF+SDF Meta-environment [Kli93], the generator and generated tools are easy to maintain, document, and adapt. The generator respects the modular structure of the input grammar and after generating an initial version, incrementally updates only those modules for which the original grammar is changed.

The program generation techniques presented in this paper can also be used to produce generators of (schemata of) other syntax directed translators.

Acknowledgements

We thank Jacob Brunekreef, Joris Hillebrand, Wilco Koorn, and Martijn Res for their remarks and suggestions based on their experience with the formatter generator. Jan Rekers developed (a more sophisticated version of) the algorithm for classifying context-free grammar rules that we presented in Section 4.2. We thank Paul Klint, Emma van der Meulen, and Susan Üsküdarlı 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* 24(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.

[Bra93a] M. G. J. van den Brand. Generation of language independent prettyprinters. Technical Report P9327, programming Research Group, University of Amsterdam, Amsterdam, October 1993.

[Bra93b] M.G.J. van den Brand. Prettyprinting without losing comments. Report P9315, University of Amsterdam, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Bra93.ps.Z.

[Bru95] J.J. Brunekreef. *On Modular Algebraic Protocol Specification*. PhD thesis, University of Amsterdam, 1995.

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

[BV94] Mark van den Brand and Eelco Visser. From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, July 1994. Postscript version available by anonymous ftp as `ftp.fwi.uva.nl:/pub/prg/reports/1994/P9420.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.

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

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

[Koo94] J. W. C. Koorn. *Generating uniform user-interfaces for interactive programming environments.* PhD thesis, University of Amsterdam, 1994. ILLC Dissertation series 1994-2.

[MCC86] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

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

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, 12:85–139, 1990.

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

[Res94] M. Res. A generated programming environment for RISLA, a specification language for defining financial products. Master's thesis, Programming Research Group, University of Amsterdam, 1994.

[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.

[SS88] N. Stern and R. Stern. *Structured COBOL Programming.* Wiley, New York, 1988.

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

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Wal91.ps.Z.