Generation of Formatters for Context-Free Languages

MARK VAN DEN BRAND and EELCO VISSER
University of Amsterdam

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. Apart from being tedious, this will invariably introduce errors. The production of tools that support the production of legible and accurate documentation 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 generated formatters translate abstract syntax trees of programs into box expressions. 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 tuned in order to get the desired formatting of programs. We demonstrate this by means of real-life applications. Furthermore, we give a practical solution for the problem of formatting comments, which occur in the original text. The formatter generation approach proposed in this article can be used to generate formatting programs for arbitrary programming environments. Our formatter generation approach can be used to automatically generate formatters that have to be programmed explicitly in other systems.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; D.2.3 [**Software Engineering**]: Coding—*pretty printers*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*documentation*; D.2.m [**Software Engineering**]: Miscellaneous—*rapid prototyping*; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*; I.7.2 [**Text Processing**]: Document Preparation

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: Document preparation, program generators

1. INTRODUCTION

Software engineering and software documentation are almost synonymous. A program that is not documented will soon become an unintelligible black box that is hard to maintain and extend. Improving the textual presenta-

Authors' address: Programming Research Group, University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands; email: {markvdb; visser}@fwi.uva.nl.

tion of programs increases the chance that these are being *read* (and understood) by programmers. However, manually maintaining a program *and* its typeset documentation is time consuming and error prone.

Formatters for programming languages relieve documentation writers from typesetting programs by hand and ensure that documentation is correct and up to date. Formatters can be used to enforce a uniform textual presentation of programs written by different programmers, which eases software maintenance. Formatters can also be used to visualize the structure of programs, which can be of great help when analyzing the program text for reverse engineering of legacy code. Formatters are used in interactive applications, such as (structure) editors, to reformat the program text during editing. Formatters are also used in noninteractive applications to produce high-quality, printed versions of programs, useful for the production of legible and accurate documentation.

Current formatting technology has a number of problems. For many programming languages there are no formatters available, and these would have to be implemented from scratch. The implementation of a formatter for a language under development is an even more time consuming activity, since each time the language is modified the formatter has to be adapted also. Most existing formatters are inflexible and cannot be adapted, although this is sometimes necessary when, for example, the layout of a language influences its semantics. Furthermore, most formatters are geared to one text processing or typesetting system and cannot easily be connected with other systems.

In this article we describe a program generator that considerably simplifies the task of building formatters. The generator produces an executable specification of a formatter for a programming language from its context-free grammar. The formatter thus obtained works as follows. First, it translates the abstract syntax tree of a program into a box expression which is a declarative description of the intended layout of the program. Second, this language-independent box expression is translated into ASCII text, TEX code, or some other input format for a displaying device. These language-independent back-ends do the hard work of producing a legible text. They are reused for every language for which a formatter is generated. So, it will be obvious that our language-independent back-ends form a powerful set of typesetting tools. In combination with our mechanism to restore comments this offers a sound basis for sophisticated documentation tools.

The generator cannot possibly produce formatters that realize the desired formatting for an arbitrary language, as there is no absolute layout criterion. For instance, consider the differences between layout conventions of imperative languages and functional ones. Therefore it is necessary to be able to easily adapt the rules of the generated formatter. The formatters produced by our generator can easily be manipulated by a layman in typesetting.

This research was initiated to improve the formatting facilities of programming environments generated by the ASF+SDF Metaenvironment

[Klint 1993]. These environments had a built-in formatter which could hardly be adapted to satisfy the semantic layout constraints for languages such as Cobol [Stern and Stern 1988]. The need to be able to influence the standard formatter resulted in the box language, Box, and a number of tools, such as the formatter generator and a Box-to-ASCII back-end. Generated formatters in combination with this back-end are used to replace the standard formatting facilities of the environments. Since the structure editors of the generated environments do not support facilities such as colors and more than one font, it was not necessary to support this in the Box-to-ASCII back-end. However, it is easy to implement back-ends that do support those facilities. Currently, we are building a Box-to-HTML back-end.

Our approach of generating formatters given a context-free grammar can be applied in other programming environment generators, such as the Synthesizer Generator [Reps and Teitelbaum 1989] and CENTAUR [Borras 1989; Borras et al. 1989], where currently the formatting rules have to be specified by hand. The advantage of applying our techniques is that the bulk of formatting rules is automatically generated. This saves a considerable amount of time, and the generation of unparsing rules for a given grammar ensures the correctness of the unparsing specification.

1.1 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 1. The triangle in this figure represents a tree. The rectangles stand for text files. The ellipses denote programs. The dashed objects are currently developed. Arrows denote the input/output behavior of the programs. Note that this figure contains a cycle: AST, formatter, Box term, Box2ASCII, ASCII text, parser, and AST again. This cycle expresses that our approach is formal: the parsing of the formatted text of a given abstract syntax tree produces this abstract syntax tree again. This is important as formatters are not allowed to alter the syntactic structure of a program by themselves. For the other two back-ends this is not relevant because these produce output for final versions of the programs to be used as or included in documentation.

In this article we describe a formatter generator that produces ASF+SDF specifications [Bergstra et al. 1989; Heering et al. 1992]. The underlying ideas are equally well applicable to imperative or functional implementations and can be expressed in any language that supports recursive functions and data structures such as C [Kernighan and Ritchie 1978]. 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. A detailed description of the implementation can be found in Section 7.

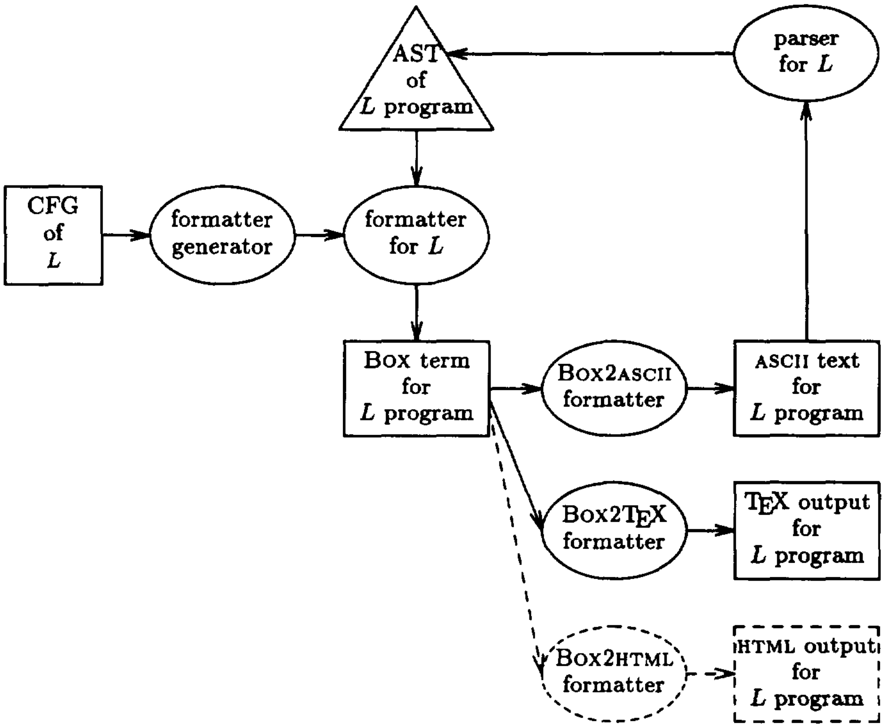


Fig. 1. Architecture of the formatter generator and formatting tools.

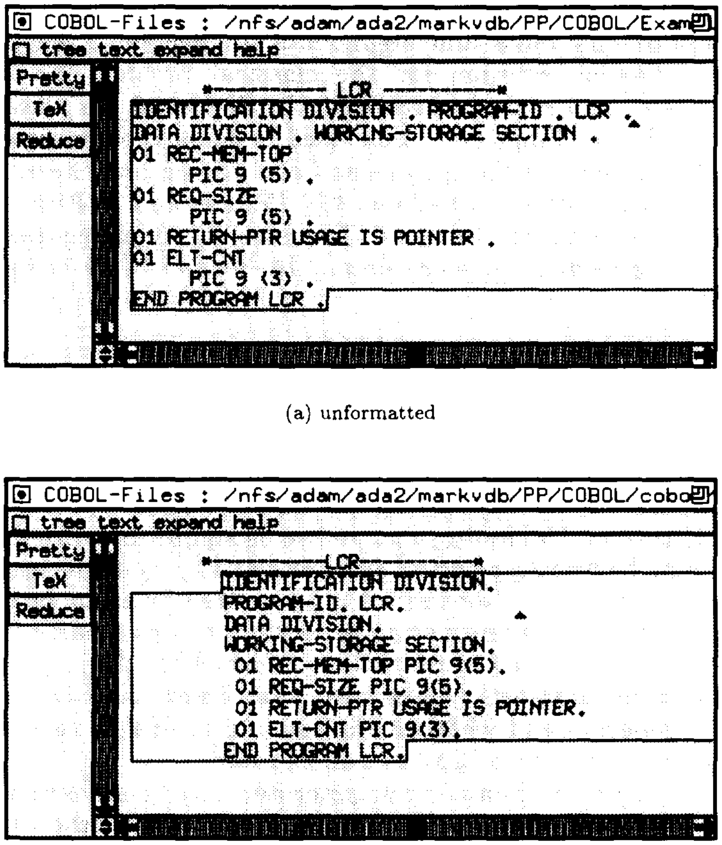
In the next example we discuss the use of a generated and adapted formatter and the Box-to-ASCII and Box-to-TEX back-ends in a structure editor.

Example 1.1.1 Given the context-free grammar of the language Cobol, 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 2(a) 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 2(b). In this situation the formatter is used interactively; only the selected text is replaced by the formatted text.

Pushing the "TeX" button in Figure 2(a) produces TEX code which yields the text shown in Figure 2(c). Here the formatter is used in a batch-oriented manner. The production of TEX code takes more time than the production of plain ASCII text, because a number of transformations is applied, before TEX code can be generated; see van den Brand and Visser [1994] for details. Note that keywords and variable names are set in different fonts.

1.2 Overview

We start in the next section with the generation of *unparsers*—formatters that directly translate abstract syntax trees to strings without consider-


COBOL-Files : /nfs/adam/ada2/markvdb/PP/COBOL/Exam
tree text expand help
Pretty
TeX
Reduce
LCR

```
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 .
```

(a) unformatted

COBOL-Files : /nfs/adam/ada2/markvdb/PP/COBOL/cobo
tree text expand help
Pretty
TeX
Reduce
LCR

```
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.
```

(b) formatted

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*.

(c) typeset

Fig. 2. Structure editor with unformatted (a) and formatted (b) Cobol text and TEX output for a part of the Cobol program (c).

ation for layout. 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 our box language and present a formatter that produces boxes. Some extensions of Box are described, such as box operators for fonts and alignments. In Section 5 we present an elegant and flexible solution for the classical problem for formatting comments occurring in the source text. In Section 6 we show how the extensions of the box language can be used to adapt the generated formatter. Next, formatters for the languages Cobol [Stern and Stern 1988] and the process specification formalism PSF [Mauw and Veltink 1990] are discussed as case studies. Section 7 discusses the implementation of the box formatter generator in more detail. In Section 8 we review related work.

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 unparser. Each production in the grammar is translated to an equation in the generated unparser. 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 for language L is a function

$$\mathsf{parse}_L : \mathsf{STRING} \rightarrow \mathsf{TREE}_L$$

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

$$\mathsf{pp}_L : \mathsf{TREE}_L \rightarrow \mathsf{STRING}$$

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

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

as the unparser may modify the layout in the original string s. An unparser pp_L is correct with respect to parse_L if, given any abstract syntax tree Δ_L over language L, we have

$$\mathsf{parse}_L(\mathsf{pp}_L(\Delta_L)) = \Delta_L.$$

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

The unparser is generated using the context-free grammar $G = \langle V_N, V_T, S, P\rangle$ 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 generated unparser contains a function

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

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, $\mathsf{var}X_i$, 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

$$\mathsf{pp}N(\bar{X}_1 \ldots \bar{X}_n) = s_1 \cdot \text{“}\sqcup\text{”} \cdot \ldots \cdot \text{“}\sqcup\text{”} \cdot s_n$$

where

$$\bar{X}_i = \begin{cases} X_i & \text{if } X_i \in V_T, \\ \mathsf{var}X_i & \text{if } X_i \in V_N \end{cases} \quad \text{and} \quad s_i = \begin{cases} \text{“}X_i\text{”} & \text{if } X_i \in V_T, \\ \mathsf{pp}X_i(\bar{X}_i) & \text{if } X_i \in V_N \end{cases}$$

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.

[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
variables
  varDecls[0-9]*  → Decls
  varSeries[0-9]* → Series
equations
[1]   ppProgram(begin varDecls0 varSeries1 end)
      = "begin" . "␣" . ppDecls(varDecls0) . "␣" . ppSeries(varSeries1) . "␣" . "end"
```

Fig. 3. Generated unparser for a context-free production rule.

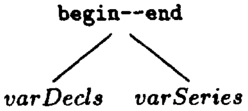


Fig. 4. The abstract syntax tree for *begin varDecls varSeries end*.

Example 2.2.1. The production rule

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

is transformed into the specification given in Figure 3.

2.3 Interpretation of Equations

Before we continue our description of the unparser a few words on the interpretation of equations are appropriate. A description of the structure of ASF+SDF specifications can be found in Section 7.1. 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 4.

Applying the function *ppProgram* to *begin varDecls varSeries end* corresponds to applying the function *ppProgram* to the abstract syntax tree of Figure 4. *ppProgram(begin varDecls varSeries end)* rewrites to the right-hand side of Figure 5. 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.

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 Walters [1991].

2.4 Lists

Most extended BNF notations support a notion for *lists*. In SDF [Heering et al. 1992] there are four types of lists: with and without a separator, and

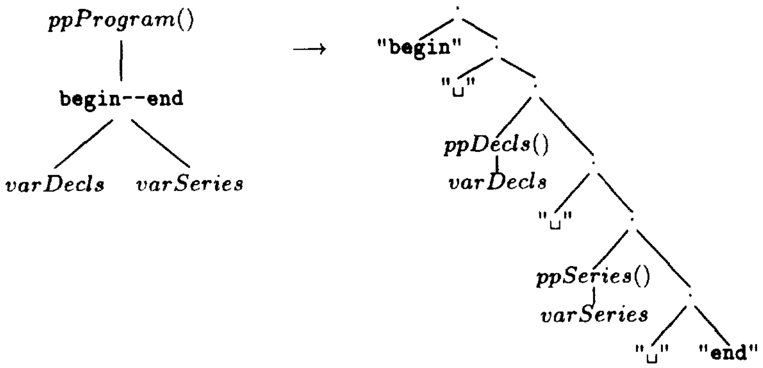


Fig. 5. Interpretation of an equation.

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

$$\begin{array}{ll} N* ::= & N+ ::= N \\ N* ::= N+ & N+ ::= N\ N+ \end{array}$$

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

$$\begin{array}{ll} \{N\ \text{“}t\text{”}\}* ::= & \{N\ \text{“}t\text{”}\}+ ::= N \\ \{N\ \text{“}t\text{”}\}* ::= \{N\ \text{“}t\text{”}\}+ & \{N\ \text{“}t\text{”}\}+ ::= N\ \text{“}t\text{”}\ \{N\ \text{“}t\text{”}\}+ \end{array}$$

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 6.

2.4.1 *Formatting Rules for Lists.* For each list $\{N\ \text{“}t\text{”}\}*$ the generated unparser contains a function

$$\text{“pp}Nt\text{0”}\ (\{N\ \text{“}t\text{”}\}*) \rightarrow \text{STRING}$$

defined by

$$\begin{aligned} \text{pp}Nt\text{0}(\ ) &= \text{“ ”} \\ \text{pp}Nt\text{0}(\text{var}N) &= \text{pp}N(\text{var}N) \\ \text{pp}Nt\text{0}(\text{var}N\ t\ \text{var}Nt+) &= \text{pp}N(\text{var}N) \cdot \text{“␣”} \cdot \text{“}t\text{”} \cdot \text{“␣”} \cdot \text{pp}Nt\text{0}(\text{var}Nt+). \end{aligned}$$

where $\text{var}Nt+$ is a variable of the sort $\{N\ \text{“}t\text{”}\}+$. For each list $\{N\ \text{“}t\text{”}\}+$ the generated unparser contains a function

$$\text{“pp}Nt\text{1”}\ (\{N\ \text{“}t\text{”}\}+) \rightarrow \text{STRING}$$

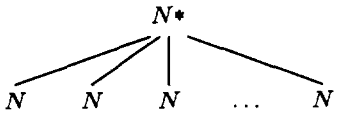


Fig. 6. List node in abstract syntax tree for a list without separators.

defined by the equations

$$\mathsf{pp}Nt1(\mathsf{var}N) = \mathsf{pp}N(\mathsf{var}N)$$

$$\mathsf{p}Nt1(\mathsf{var}N\ t\ \mathsf{var}Nt{+}) = \mathsf{pp}N(\mathsf{var}N) \cdot \text{“}\sqcup\text{”} \cdot \text{“}t\text{”} \cdot \text{“}\sqcup\text{”} \cdot \mathsf{pp}Nt1(\mathsf{var}Nt{+}).$$

For lists without separators the unparser 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 2.4.1.1. Consider a production rule for a block of statements

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

The generated unparser will contain the generated equations shown in Figure 7. The variable *varStat*;* in Figure 7 denotes a list with zero or more elements, whereas *varStat*;$^+$ denotes a list with at least one element.[2]

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. Unparsers must be able to deal with these implicitly derived syntax rules.

Example 2.5.1. 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.

2.5.1 *Formatting Rules for Placeholders.* The unparser must be able to format programs containing placeholders; therefore for each nonterminal N it contains an equation:

$$\mathsf{pp}N(\langle N\rangle) = \text{“}\langle N\rangle\text{”}$$

For each list {N “t”}* and {N “t”}+ the unparser contains the equations:

$$\mathsf{pp}Nt0(\langle Nt*\rangle) = \text{“}\langle Nt*\rangle\text{”}$$

$$\mathsf{pp}Nt1(\langle Nt{+}\rangle) = \text{“}\langle Nt{+}\rangle\text{”}$$

[2] The lifting of the * and + in the variable names is caused by the ASF+SDF typesetter.

context-free syntax
"ppStat;0"({Stat ";"}*) → String
variables
$varDecls[0\text{-}9]*$ → Decls
$varStat[0\text{-}9]*$ → Stat
"varStat; *"$[0\text{-}9]*$ → {Stat ";"}*
"varStat; +"$[0\text{-}9]*$ → {Stat ";"}+
equations

[1] ppProgram(begin $varDecls_0$ $varStat;_1^*$ end)
= "**begin**" . "␣" . ppDecls($varDecls_0$) . "␣" . ppStat;0($varStat;_1^*$) . "␣" . "**end**"

[2] ppStat;0() = ""

[3] ppStat;0($varStat_0$) = ppStat($varStat_0$)

[4] ppStat;0($varStat_0$; $varStat;_1^+$)
= ppStat($varStat_0$) . "␣" . ";" . "␣" . ppStat;0($varStat;_1^+$)

Fig. 7. Generated unparser for lists.

For lists without separators the unparser contains similar equations.

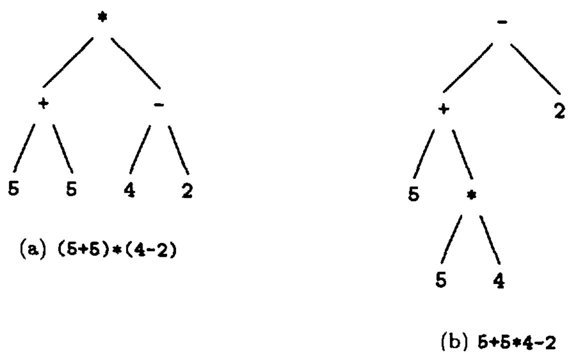
2.5.2 *Variables.* The specification of the unparser contains, besides the functions and equations, a list of variables used in the equations. All variables are directly related to nonterminals in the original context-free grammar; however, for the lists $N*$ and $\{N$ "t"$\}*$ the unparser contains also list variables for $N+$ and $\{N$ "t"$\}+$ respectively. An example of the declaration of variables used in equations can be found in Figure 7.

In the rest of this article all example specifications will be presented without the variable declarations.

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 of 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.

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



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

(b) 5+5*4-2

Fig. 8. Abstract syntax trees for (5 + 5) * (4 − 2) and 5 + 5 * 4 − 2 according to the syntax for expressions in Example 3.1.

which are associated 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 3.1. Consider the following SDF definitions for arithmetic expressions.

context-free syntax
Nat → Exp
Exp "+" Exp → Exp {**left**}
Exp "−" Exp → Exp {**left**}
Exp "*" Exp → Exp {**left**}
"(" Exp ")" → Exp {**bracket**}

priorities
{**left**: Exp "+" Exp → Exp, Exp "−" Exp → Exp}
< Exp "*" Exp → Exp

All the operators are left-associative. The * operator has higher priority than the + and − operators, 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 8(a) for the sentence (5 + 5) * (4 − 2). A naive text formatter might 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 8(b). Given the priority and associativity definitions the formatter should contain equations which insert these brackets automatically.

context-free syntax
non-assocExp(Exp, Exp) → BOOL
rightExp(Exp, Exp) → BOOL
leftExp(Exp, Exp) → BOOL
gtrExp(Exp, Exp) → BOOL
equations

[1] non-assocExp($varExp_0$, $varExp_1$) = false **otherwise**

[2] rightExp($varExp_0$, $varExp_1$) = false **otherwise**

[3] leftExp($varExp_0$ + $varExp_1$, $varExp_2$ + $varExp_3$) = true
[4] leftExp($varExp_0$ − $varExp_1$, $varExp_2$ − $varExp_3$) = true
[5] leftExp($varExp_0$ * $varExp_1$, $varExp_2$ * $varExp_3$) = true
[6] leftExp($varExp_0$ + $varExp_1$, $varExp_2$ − $varExp_3$) = true
[7] leftExp($varExp_0$ − $varExp_1$, $varExp_2$ + $varExp_3$) = true
[8] leftExp($varExp_0$, $varExp_1$) = false **otherwise**

[9] gtrExp($varExp_0$ * $varExp_1$, $varExp_2$ + $varExp_3$) = true
[10] gtrExp($varExp_0$ * $varExp_1$, $varExp_2$ − $varExp_3$) = true
[11] gtrExp($varExp_0$, $varExp_1$) = false **otherwise**

Fig. 9. Generated equations representing priority and associativity declarations.

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

$$\text{non-assoc}N(N, N) \rightarrow \text{BOOL}$$
$$\text{right}N(N, N) \rightarrow \text{BOOL}$$
$$\text{left}N(N, N) \rightarrow \text{BOOL}$$
$$\text{gtr}N(N, N) \rightarrow \text{BOOL}$$

Example 3.2. The rules in the SDF definition for the expressions from Example 3.1 are translated to the equations presented in Figure 9. The generated text-formatting rule for the + operator defined in Example 3.1 is shown in Figure 10. The functions *l_bracketsExp* and *r_bracketsExp* in Figure 10 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

$$\text{“(”}\ N\ \text{“)”} \rightarrow N\ \{\text{bracket}\}$$

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

$$\text{addbrackets}N\ (N) \rightarrow \text{STRING}$$

context-free syntax
ppExp(Exp) → String
equations

$$[1]\quad \frac{\begin{array}{c}\mathrm{l\_addbracketsExp}(\mathit{varExp}_0,\ \mathit{varExp}_0 + \mathit{varExp}_1) = \mathit{varString}_0, \\ \mathrm{r\_addbracketsExp}(\mathit{varExp}_1,\ \mathit{varExp}_0 + \mathit{varExp}_1) = \mathit{varString}_1\end{array}}{\mathrm{ppExp}(\mathit{varExp}_0 + \mathit{varExp}_1) \ = \ \mathit{varString}_0 \ . \ \texttt{"␣"} \ . \ \texttt{"+"} \ . \ \texttt{"␣"} \ . \ \mathit{varString}_1}$$

Fig. 10. Generated equation for unparsing an expression.

for producing the brackets around an expression and the functions

$$\text{l\_addbrackets}N\ (N,\ N) \rightarrow \text{STRING}$$

$$\text{r\_addbrackets}N\ (N,\ N) \rightarrow \text{STRING}$$

$$\text{m\_addbrackets}N\ (N,\ N) \rightarrow \text{STRING}$$

The first argument of these functions is a subexpression of the expression given as the 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 3.3. The generated equations for restoring the brackets for the grammar presented in Example 3.1 are given in Figure 11. The functions *l_addbracketsExp* and *r_addbracketsExp* of Figure 10 are defined using the functions of Figure 9. 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, and 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 ASCII, TEX, or any other output language, we are able to reuse the box formatters without modifications.

```
context-free syntax
  addbracketsExp(Exp)            → String
  l_addbracketsExp(Exp, Exp)     → String
  m_addbracketsExp(Exp, Exp)     → String
  r_addbracketsExp(Exp, Exp)     → String
equations

[1]  addbracketsExp(varExp0) = "(" . "␣" . ppExp(varExp0) . "␣" . ")"

                non-assocExp(varExp0, varExp1)
                ∨ rightExp(varExp0, varExp1)
                ∨ gtrExp(varExp1, varExp0)       = true
[2]  ───────────────────────────────────────────────────────────
     l_addbracketsExp(varExp0, varExp1) = addbracketsExp(varExp0)

[3]  l_addbracketsExp(varExp0, varExp1) = ppExp(varExp0)  otherwise

                non-assocExp(varExp0, varExp1)
                ∨ leftExp(varExp0, varExp1)
                ∨ gtrExp(varExp1, varExp0)       = true
[4]  ───────────────────────────────────────────────────────────
     r_addbracketsExp(varExp0, varExp1) = addbracketsExp(varExp0)

[5]  r_addbracketsExp(varExp0, varExp1) = ppExp(varExp0)  otherwise

                  gtrExp(varExp1, varExp0) = true
[6]  ───────────────────────────────────────────────────────────
     m_addbracketsExp(varExp0, varExp1) = addbracketsExp(varExp0)

[7]  m_addbracketsExp(varExp0, varExp1) = ppExp(varExp0)  otherwise
```

Fig. 11. Generated equations for adding brackets.

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 reuse the modified formatting rules when a new box formatter is generated for the same language after, for instance, modifying the context-free grammar; see Section 6.1.

4.1 Box: A Box Language

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

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

The formatting of the HV operator depends on the amount of space left for the argument boxes. For each argument box it 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.

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

The formatting of the HOV operator also depends on the amount of space left for the boxes. For *all* argument boxes it 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.

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

The functionality of the I operator depends on the surrounding box operator. This operator has an effect only 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.

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

This has the consequence that an I operator in a HV or HOV is only considered when the surrounding box becomes vertical.

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

$$\mathsf{V}[\ \boxed{B_1}\,\mathsf{H}[\mathsf{WD}[\ \boxed{B_1}\ ]\ \boxed{B_2}\ ]] = \begin{array}{ll}\boxed{B_1} & \\ & \boxed{B_2}\end{array}$$

4.2 Box Syntax

A box term consists of a box operator, zero or more spacing options, and is 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 with the H operator is not useful.

```
context-free syntax
  "hs"                       → Spaced-Symbol
  "vs"                       → Spaced-Symbol
  "is"                       → Spaced-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.

4.3 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 box formatter for each grammar rule using this classification information.

4.3.1 *Classification of Context-Free Grammar Rules.* Box offers the facility to place the members of a context-free grammar rule in a 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 *nonindented.* The name "indented" stems from the fact that some language constructs are more indented than others to increase readibility. The name "nonindented" 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

—nonindented, 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 classification can be decided by means of a simple finite automaton, which scans the production rule and yields its classification as the result. This classification is performed by the formatter generator, which uses this information to decide which box operators must be used in the formatting rule of the corresponding context-free grammar rule. The actual generator distinguishes more types in the classification and uses more sophisticated heuristics to classify the context-free grammar rules.

4.3.2 *Derivation of Box Formatting Rules.* For each nonterminal N in the context-free grammar the generated formatter contains a function:

$$\mathsf{pp}N\ (N) \rightarrow \mathsf{Box}$$

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

4.3.3 *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

$$\mathsf{pp}N\ (\bar{X}_1 \ldots \bar{X}_n) = \mathsf{V}\ [b_1 \ldots b_n]$$

where

$$\bar{X}_i = \begin{cases} X_i & \text{if} \quad X_i \in V_T \\ \mathsf{var}X_i & \text{if} \quad X_i \in V_N \end{cases} \qquad \text{and} \qquad b_i = \begin{cases} \text{“}X_i\text{”} & \text{if} \quad X_i \in V_T \\ \mathsf{I}[\mathsf{pp}X_i(\bar{X}_i)] & \text{if} \quad X_i \in V_N. \end{cases}$$

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

4.3.4 *Formatting Rules for Grammar Rules of Type "Nonindented."* A grammar rule of type nonindented results in an equation

$$\mathsf{pp}N\ (\bar{X}_1 \ldots \bar{X}_n) = \mathsf{HV}\ [b_1 \ldots b_n]$$

where

$$\bar{X}_i = \begin{cases} X_i & \text{if} \quad X_i \in V_T \\ \mathsf{var}X_i & \text{if} \quad X_i \in V_N \end{cases} \qquad \text{and} \qquad b_i = \begin{cases} \text{“}X_i\text{”} & \text{if} \quad X_i \in V_T \\ \mathsf{pp}X_i(\bar{X}_i) & \text{if} \quad X_i \in V_N. \end{cases}$$

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

4.3.5 *Formatting Rules for Lists.* For each list $\{N\ \text{“}t\text{”}\}*$ the formatter contains the functions

$$\text{“}\mathsf{pp}Nt0\text{”}\ (\{N\ \text{“}t\text{”}\}*) \rightarrow \mathsf{Box}$$

$$\text{“}\mathsf{pp}Nt0\mathsf{e}\text{”}\ (\{N\ \text{“}t\text{”}\}*) \rightarrow \mathsf{Box}$$

defined as

$$\mathsf{pp}Nt0(\ ) = \mathsf{H}\ [\ ]$$

$$\mathsf{pp}Nt0(\mathsf{var}Nt1) = \mathsf{V}\ [\mathsf{pp}Nt0\mathsf{e}(\mathsf{var}Nt1)]$$

$$\mathsf{pp}Nt0\mathsf{e}(\mathsf{var}N) = \mathsf{pp}N(\mathsf{var}N)$$

$$\mathsf{pp}Nt0\mathsf{e}(\mathsf{var}N\ t\ \mathsf{var}Nt1) = \mathsf{H}\ \mathsf{hs{=}0}\ [\mathsf{pp}N(\mathsf{var}N)\ \text{“}t\text{”}]\ \mathsf{pp}Nt0e(\mathsf{var}Nt1).$$

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

```
context-free syntax
  ppDecls(Decls)                     → Box
  ppId-Type(Id-Type)                 → Box
  "ppId-Type,0"({Id-Type ","}*)      → Box
  "ppId-Type,0e"({Id-Type ","}*)     → Box-List
equations
[1]  ppDecls(declare varId-Type,*;)
     = V ["declare"
          I [ppId-Type,0(varId-Type,*)]
          ";"]

[2]  ppId-Type,0() = H []

[3]  ppId-Type,0(varId-Type,+) = V [ppId-Type,0e(varId-Type,+)]

[4]  ppId-Type,0e(varId-Type) = ppId-Type(varId-Type)

         ppId-Type,0e(varId-Type,+) = varBox-List
[5]  ----------------------------------------------
      ppId-Type,0e(varId-Type, varId-Type,+) =
      H hs = 0[ppId-Type(varId-Type) ","] varBox-List
```

Fig. 12. Generated box formatter.

Example 4.3.5.1. Consider the context-free grammar rule

context-free syntax
declare {Id-Type ","}* ";" → Decls

which is transformed into the specification presented in Figure 12. This rule is classified as type "indented."

4.4 Modular Structure of the Box Formatter

SDF 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. FORMATTING COMMENTS

A major problem 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 for the "expected" comments in Eiffel [Meyer 1992]. 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 [Blaschek and Sametinger 1989; Jokinen 1989]. Below, we describe a solution for the treatment of this sort of comments. Note that we do *not* consider the case where comments may appear *inside* lexical tokens, e.g., as in Algol68 [Wijngaarden et al. 1976].

One approach to solve the problem of restoring comments is to attach them to nodes in the abstract syntax tree [Rose and Welsh 1981]. During formatting, the comments are regenerated when processing the node in question. This method is also used in the CENTAUR system [Borras 1989; Borras et al. 1989]. 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:

—white space consists of spaces and new lines only and will be ignored;

—a comment starts with a begin marker and ends with either a new line 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 ends at the end of the line. The second rule describes comments which start with some begin marker (“%”) and ends also with an end marker; in this case no new lines 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.

5.1 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.

5.2 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. White space 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 ["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
```

Note that the text of the comments is formatted as well. This was a design decision; a small modification in the back-ends could suppress this formatting.

6. 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. Such a translator can be specified in ASF+SDF: Pascal → C → Box → ASCII.

Example 6.1. Consider again the context-free grammar rule

```
context-free syntax
  declare {Id-Type ","}* ";" → Decls
```

The corresponding box formatting rule is given in Figure 12.

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

```
[1]  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 will be presented in Section 6.4.

6.1 General Usage

The construction of a formatter for some language passes through four phases.

First the context-free syntax of the language in question has to be formally specified. We use SDF to define the syntax in a modular way. The modules defining the syntax of the language can be used as input for the formatter generator.

In the second phase the formatter generator generates an algebraic specification containing formatting functions for all the context-free rules that were defined in the SDF modules. The generator produces for each SDF module M two modules, one containing the generated formatting information and one module that is intended to adapt certain formatting rules to achieve specific effects, such as putting things in columns. We will call these modules the generated module and the modification module. We name these modules GEN-M and MOD-M respectively.

In the third phase the formatter can be adapted by modifying the generated functions. The functions in the generated modules should not be adapted; when the context-free grammar is changed the specific added information would be destroyed. The rules that need more attention should preferably be copied from the GEN-*M* module to the MOD-*M* module. The generator will *never* overwrite the MOD-*M* modules; only the GEN-*M* modules are affected.

The heuristics used in the generator are based on Algol-like languages; therefore, it is sometimes necessary to alter the generated formatting functions. For example, for declarative languages, like Prolog [SICS 1992] or CLEAN [Plasmeijer and van Eekelen 1993], the formatting functions must be adapted to obtain good formatting. Some compilers impose strict layout conventions on the programs to be processed by them, such as Cobol [Stern and Stern 1988].

The final phase of the development of a formatter is to speed up the program. This is done by translating the ensuing algebraic specification into a standalone C program. This is done with the ASF2C compiler [Kamperman and Walters 1993].

The amount of time spent tuning the generated formatting functions depends on the size of the grammar and the structure of the language. The PSF syntax [Mauw and Veltink 1990] consists of 165 context-free syntax rules spread over 10 SDF modules. It took about two days to produce a satisfactory formatter for this language. This was done by a user who was not an expert in making formatting programs.

Of course, as soon as a user adapts the generated rules for specific effects, their correctness is no longer guaranteed. There is no tool to check the correctness of the tuned rules. However, there is one aspect of the correctness of the formatting rules that can be defined by means of an algorithm. The skeleton of the right-hand side of the box formatting rule of a context-free grammar rule should be equivalent to the right-hand side of the corresponding text-formatting rule. This can be checked by stripping the box expression in the right-hand side of its operators and using string concatenation instead; this should always yield the original text-formatting rule. The semantics of Box guarantees its constructs are placed linearly; none of the box operators switches boxes. Furthermore, the modification of the generated formatting rules is supported by an interactive programming environment; every single change can be visualized immediately, and therefore the most obvious mistakes will become apparent immediately.

6.2 Columns

Although Box only uses six operators it is possible to specify complex layout schemes, such as columns. First we give an example of how to obtain columns using the kernel box language. It turns out that this approach is too tedious for the specification writer and too much directed toward 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 6.2.1. 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 get columns in the formatted text has a number of disadvantages. Such a specification is nondeclarative; it contains formatting information; thus it is not back-end independent. It is quite difficult for the specification writer to specify this calculation of columns, particularly if things must be put in more than two columns. The box formatter contains too much detailed information; only the layout structure should be specified, not the way to obtain this form. Finally, such a specification is difficult to maintain and is error prone. Changing the context-free syntax rules could result in a complete new specification of the column calculation.

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

6.2.1 *Alignments*. The introduction of an alignment operator relieves the user from specifying the explicit calculation of columns. Furthermore, the alignment operator can easily be translated to TEX.

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 “]” → Box
  “R” “[” Box-List “]”                      → Box

  “l” S-Options                             → A-Option
  “c” S-Options                             → A-Option
  “r” S-Options                             → A-Option

  “(” {A-Option “,”}* “)”                   → A-Options
```

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

[1] ppId-Type,0(*varId-Type*$_{,0}$) = A (l , c , l) [ppId-Type,0e(*varId-Type*$_{,0}$)]

[2] ppId-Type,0e() = ""

[3] ppId-Type,0e(*varId* : *varType*) = R[ppId(*varId*) ":" ppType(*varType*)]

[4] ppId-Type,0e(*varId-Type*$_{,1}$) = ***varBox-List***
ppId-Type,0e(*varId* : *varType*, *varId-Type*$_{,1}$) =
R[ppId(*varId*) ":" H hs = 0[ppType(*varType*) ","]]
varBox-List

Fig. 13. Specification of declaration columns using the alignment operators.

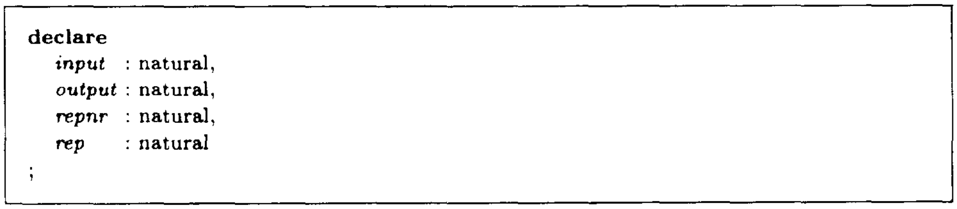

declare
input : natural,
output : natural,
repnr : natural,
rep : natural
;

Fig. 14. TEX output for the declarations.

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 into box operators in the kernel 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 means of its built-in alignment facilities.

Example 6.2.1.1. The use of the alignment operators is demonstrated in Figure 13. The declarations of Example 6.2.1 are formatted as shown in Figure 14 using the TEX back-end.

6.3 Fonts

The kernel 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 Box with operators to manipulate fonts. We have used the same strategy as with alignments. We extend the kernel 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 appropriate 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
```

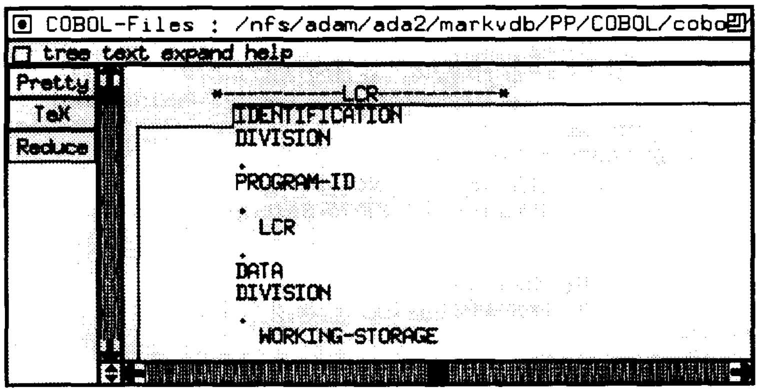
A number of frequently used fonts are predefined by means of special font operators. They can be considered as a kind of abbreviation, because they can also be defined using font options. Special font operators are introduced to mark keywords, variables, numbers, mathematical symbols, and comments. The 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 typechecking information, while constructing the box expression for some program in order to decide which font operator to use.



```
*----------LCR----------*
IDENTIFICATION
DIVISION
.
PROGRAM-ID
.
 LCR
.
DATA
DIVISION
.
 WORKING-STORAGE
```

Fig. 15. An incorrectly formatted Cobol program.

6.4 Developing a Cobol Formatter

We demonstrate the approach of tuning generated formatters by generating a formatter for the programming language Cobol [Stern and Stern 1988] and modifying some of the generated box formatting rules. The programming language Cobol has a simple syntax, but Cobol compilers impose severe constraints on the layout. Cobol compilers do not process programs which do not satisfy these layout constraints. We will show how to tune a formatter by modifying the box formatting rules generated for the context-free grammar rules in Figure 18.

The Cobol program in the editor of Figure 15 is an incorrectly formatted Cobol program which will not be accepted by a Cobol compiler. 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.

Equation 4 in Figure 16 is changed in equation 1 given in Figure 17, 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 19.

The equations 2, 5, and 7 given in Figure 16 are changed into the equations 1, 2, and 3 given in Figure 20 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 21. The numbered Cobol lines in Figures 19 and 21 are now formatted as well. The modifications to obtain this result are not discussed here. The end result is a program which can be processed by a Cobol compiler.

```
equations
[1]  ppCOBOL-PROGRAM(var_ID-DIV:0 var_OPT-DATA-DIV:1
                         var_OPT-PROC-DIV:2 var_OPT-PROG-END:3)
     = HV [ppID-DIV(var_ID-DIV:0)
           ppOPT-DATA-DIV(var_OPT-DATA-DIV:1)
           ppOPT-PROC-DIV(var_OPT-PROC-DIV:2)
           ppOPT-PROG-END(var_OPT-PROG-END:3)]
     otherwise

[2]  ppOPT-PROG-END(END PROGRAM var_ID:2 .)
     = V [KW["END"] KW["PROGRAM"] I [ppID(var_ID:2)] "."]
     otherwise

[3]  ppOPT-PROG-END() = HV []  otherwise

[4]  ppID-DIV(IDENTIFICATION DIVISION . PROGRAM-ID . var_ID:5 .)
     = V [KW["IDENTIFICATION"]
          KW["DIVISION"] "."
          KW["PROGRAM-ID"] "."
          I [ppID(var_ID:5)] "."]
     otherwise

[5]  ppOPT-DATA-DIV(DATA DIVISION . var_OPT-STORAGE-SEC:3
                                    var_OPT-LINKAGE-SEC:4)
     = V [KW["DATA"] KW["DIVISION"] "."
          I [ppOPT-STORAGE-SEC(var_OPT-STORAGE-SEC:3)]
          I [ppOPT-LINKAGE-SEC(var_OPT-LINKAGE-SEC:4)]]
     otherwise

[6]  ppOPT-DATA-DIV() = HV []  otherwise

[7]  ppOPT-STORAGE-SEC(WORKING-STORAGE SECTION .
                                          var_DATA-DESCS:3)
     = V [KW["WORKING-STORAGE"]
          KW["SECTION"] "."
          I [ppDATA-DESCS(var_DATA-DESCS:3)]]
     otherwise

[8]  ppOPT-STORAGE-SEC() = HV []  otherwise
```

Fig. 16. Some of the generated formatting rules for Cobol.

The Cobol formatter was used as a back-end for a compiler which translated a special-purpose specification formalism for financial applications into Cobol code [Res 1994]. 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 obtain compilable input for the Cobol compiler.

6.5 The PSF Formatter

Another application of the box formatter generator was the production of a formatter for the process specification formalism PSF [Mauw and Veltink

```
[1]  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) "."]]]
```

Fig. 17. The first modification in the generated formatting rules of Cobol.

1990]. It was used to typeset all specifications in Brunekreef [1995a]. There are a couple of interesting features related to this typesetter not specific for PSF. First, comments occurring in the original specification were not lost in the typeset text, because they were essential for understanding the specifications. Second, the author wanted to "hide" (leave out) parts of PSF specifications that were not relevant for the 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
```

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 22.

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.

7. IMPLEMENTATION

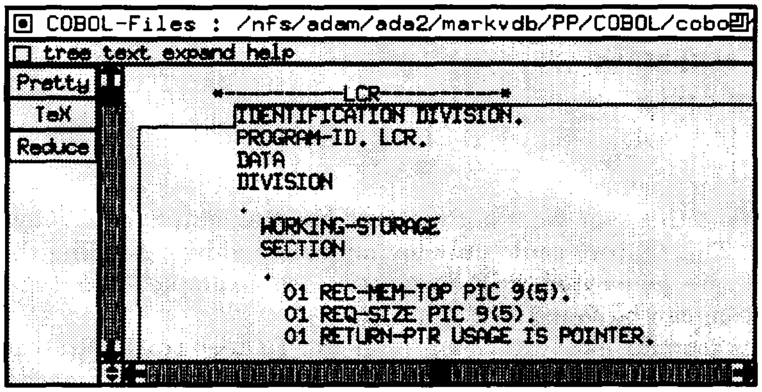
This section discusses the implementation details that played a role in the development of the formatter generator, as well as the generated formatters themselves. We will briefly explain the ASF+SDF Metaenvironment and the two accompanying formalisms. For more details on ASF+SDF we refer to Bergstra et al. [1989] and Heering et al. [1992] and for a description of the Metaenvironment to Klint [1993; 1995]. We will refrain from discussing the technical implementation details of the back-ends of the formatters, i.e., Box-to-ASCII and Box-to-TEX. A detailed description of these back-ends can be found in van den Brand and Visser [1994].

7.1 ASF+SDF

ASF+SDF is a modular algebraic specification formalism for the definition of syntax and semantics of (programming) languages. It is a combination of

```
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
```

Fig. 18. Fragment of the syntax of Cobol.



```
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.
```

Fig. 19. A slightly better formatted Cobol program.

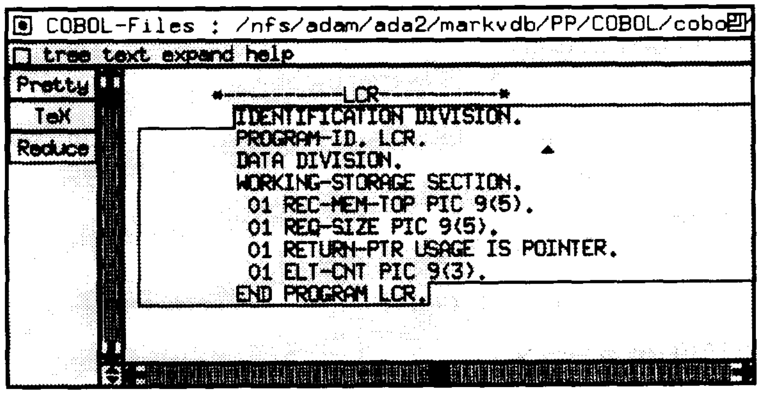
two formalisms: ASF (Algebraic Specification Formalism [Bergstra et al. 1989]) and SDF (Syntax Definition Formalism [Heering et al. 1992]). The ASF+SDF formalism is supported by an interactive programming environment: the ASF+SDF Metaenvironment [Klint 1993]. This system is a metaenvironment because it supports the design and development of programming environments. We used this environment to implement our ideas.

ASF is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. SDF allows the definition of concrete (i.e., lexical and context-free) syntax. Abstract syntax is automatically derived from the concrete syntax rules.

ASF+SDF has already been used for the formal definition of a variety of (programming) languages and for the specification of software engineering problems in diverse areas, such as simulation of hydraulic systems, query optimization, and the specification of a software interconnection bus.

```
[1]  ppOPT-PROG-END(END PROGRAM var_ID:2 .)
     = H [KW["END"] KW["PROGRAM"] H hs = 0[ppID(var_ID:2) "."]]

[2]  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)]

[3]  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)]]
```

Fig. 20. The other modified formatting rules of Cobol.



```
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.
```

Fig. 21. The expected formatted Cobol program.

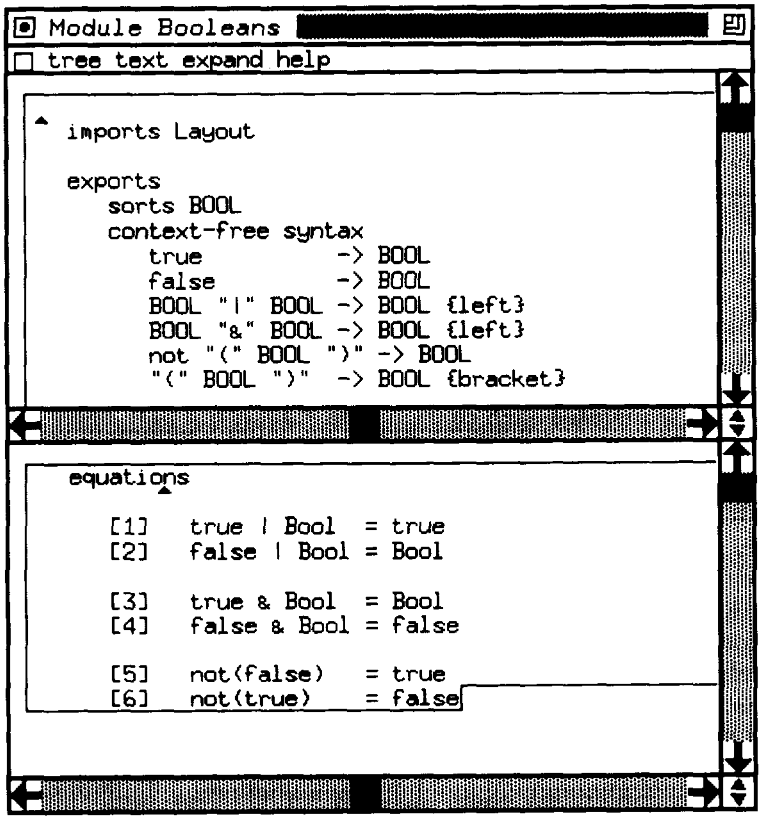
ASF+SDF specifications can be executed by interpretation or by compilation to C using the ASF2C compiler [Kamperman and Walters 1993]. It is also possible to regard the ASF+SDF specification as a *formal* specification and to implement the described functionality in some programming language. This has, for instance, been done for the software interconnection architecture ToolBus of Bergstra and Klint [1995].

The skeleton of an ASF+SDF module consists of an imports section, zero or more exports or hiddens sections (defining sorts, lexical syntax, context-free syntax, and variables), a priority declaration section, and conditional equations; detailed examples can be found in Figures 7 and 23. Physically, an ASF+SDF module consists of two files: the SDF file (everything except the conditional equations) and the ASF file (the conditional equations).

The development of ASF+SDF specifications is supported by the ASF+SDF Metaenvironment. The metaenvironment is an interactive environment in which language definitions can be edited, checked, and compiled. All editing in the environment is performed via syntax-directed

```
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(ˆ0)), lt(length(Q), pcsize)) = true] →
        sum(F in FRset,
            read-SC(cin, F) · P-CHANNEL(cin, cout, enqueue(F, Q)))
    + [or(eq(pcsize, nat(ˆ0)), 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(ˆ0)) = true] → send-CR(cout, serve(Q)) ·
      P-CHANNEL(cin, cout, dequeue(Q))
    + [and(gt(length(Q), nat(ˆ0)), pcerror) = true] → send-CR(cout, ce) ·
      P-CHANNEL(cin, cout, dequeue(Q))
    + [and(gt(length(Q), nat(ˆ0)), pcloss) = true] → lost(cin, cout) ·
      P-CHANNEL(cin, cout, dequeue(Q))
end PointToPointChannel
```

Fig. 22. A typeset PSF specification.



```
imports Layout

exports
   sorts BOOL
   context-free syntax
      true          -> BOOL
      false         -> BOOL
      BOOL "|" BOOL -> BOOL {left}
      BOOL "&" BOOL -> BOOL {left}
      not "(" BOOL ")" -> BOOL
      "(" BOOL ")"  -> BOOL {bracket}
```

```
equations

   [1]   true | Bool   = true
   [2]   false | Bool  = Bool

   [3]   true & Bool   = Bool
   [4]   false & Bool  = false

   [5]   not(false)    = true
   [6]   not(true)     = false
```

Fig. 23. A module editor.

editors which allow textual as well as structural editing [Koorn 1994]. An ASF+SDF module can be edited via a module editor, which is a combination of two syntax-directed editors: one for the SDF part and one for the ASF part; see Figure 23.

7.2 Generator

Given a context-free grammar in SDF the formatter generator generates a modular and adaptable formatter. The generator is a standalone tool that processes a module and, optionally, all its directly and indirectly imported modules. The generator preserves 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.

A pictorial representation of the implementation of the generator can be found in Figure 24. Observe that the generator has been bootstrapped: it uses its own formatting technology to produce output. The parser processes the SDF modules. The parser is implemented in LEX+YACC [Johnsson 1986; Lesk and Schmidt 1986]. The context-free grammar in SDF is translated into an intermediate representation in the Graph Exchange

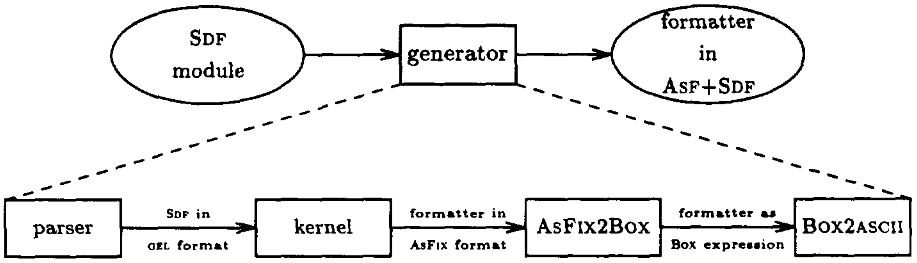


Fig. 24. Architecture of the implementation of the generator.

Language [Kamperman 1994], which we use for the concise encoding of trees and graphs. This intermediate format is input for the next phase, the kernel, which translates each context-free rule in the SDF specification into one or more formatting functions. These functions are in another intermediate format, called AsFix [Klint 1994], which we use for the internal representation of ASF+SDF specifications. The next phase, the AsFix2Box unparser, produces the Box representation of the formatting functions. In this phase the AsFix representation is "translated" to an ASF+SDF specification. The last phase is the Box2ASCII back-end which translates this Box representation into text. The kernel, AsFix2Box and Box2ASCII back-end are algebraic specifications in ASF+SDF. The ASF2C compiler translates these specifications into C programs. These programs can be compiled to obtain a standalone executable.

The performance of these compiled specifications is so good that we felt no need to reimplement them directly in C. The performance of the generator depends mainly on the size of the modules to be processed.

7.3 Generated Formatters

The formatters produced by the generator consist of two parts: the functions to transform an abstract syntax tree into a box expression and the functions which take the original text together with the constructed box expression and insert the comments in the text in the box expression. These two parts are separate because when the formatter is used as the back-end of some code generation process the restoration of comments is not needed.

The generated formatter is an ASF+SDF specification. The modular structure of the underlying context-free grammar is clearly visible in the formatter. Each module in the context-free grammar results in two modules in the generated formatter. In Section 6.1 we already gave an abstract description of the generation of these modules.

The generated formatters can also be compiled to C. Such a compiled formatter can then be used to replace the standard formatter in the programming environments generated by the ASF+SDF Metaenvironment.

In our syntax-directed editors we use a nonconventional approach: rather than regenerating the source text from the abstract syntax tree after each modification, we maintain the original source text as well. Unparsing is thus not part of the innermost time-critical editing loop. As a result, generated formatters can be used in interactive editors, although they are currently nonincremental. The development of incremental formatters is interesting, however, to improve performance when formatting large programs. The elision mechanism should also be improved, since this is important when large programs are inspected or manipulated. A flexible elision mechanism could reduce the size of program code to be formatted and thus influence performance as well. Currently we are investigating an elision mechanism similar to the one described in Cordy et al. [1990].

8. RELATED WORK

Various approaches are developed to tackle the formatting of text, formulas, or programs. Some of these approaches are based on a box-like language, e.g., TEX [Knuth 1984] or PPML [Morcos-Chounet and Conchon 1986], others use a "lexical" approach, e.g., Oppen [1980].

TEX uses a box language to typeset formulas. A formalization of this box language in a functional programming language can be found in Heckmann and Wilhelm [1995]. PPML, a pretty-print metalanguage, uses a box language to format programs. The PPML box language and our box language are quite similar. The formatting language of Pretty [Vos 1990], a first attempt to define formatting algebraically, offered a considerable number of (infix) operators on boxes.

Oppen's formatting mechanism is based on an algorithm that receives lists of lexical tokens with escape characters to direct the pretty-printing process. These lists of lexical tokens are called blocks in Oppen [1980]. A block can contain two categories of escape characters:

- *bracket characters* to delimit a syntactic 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 functionality of the brackets is to delimit syntactic constructs. The blank characters come in two flavors: consistent and flexible ones. Blocks are comparable to our boxes. The functionality of the escape characters in Oppen's approach is expressed by the operators in Box. We think that the semantics of operators is more clear than the semantics of a queue of blank characters. The consistent blank characters represent the functionality of our HOV operator, whereas the flexible ones resemble our HV operator. The unparsing rules in the specification language SSL [Reps and Teitelbaum 1989] of the Synthesizer Generator [Reps and Teitelbaum 1989] use also a set of lexical tokens — e.g., %t (move the left margin one indentation unit to the right), %b (move the left margin one indentation unit to the left), or %n (break the line and return to the current left margin) — to direct the unparsing.

There exist several programming environments that can be parameterized by a programming language definition: Synthesizer Generator [Reps and Teitelbaum 1989], CENTAUR [Borras 1989; Borras et al. 1989], PSG

[Bahlke and Snelting 1986], Mjølner/ORM Environment [Magnusson et al. 1990], Pregmatic [van den Brand 1992], and the ASF+SDF Metaenvironment [Klint 1993]. Each of these systems has its own way of implementing unparsers. SSL [Reps and Teitelbaum 1989] of the Synthesizer Generator offers facilities to specify unparsing rules. CENTAUR on the other hand uses a special-purpose language PPML [Morcos-Chounet and Conchon 1986] to specify unparsing. These two approaches have in common that the unparsers have to be specified manually.

In this article we explained how to automate the process of writing formatters. We used the ASF+SDF Metaenvironment to specify a formatter *generator* to achieve that goal. The difference between our approach and the other approaches is that it becomes possible to specify a generator to generate the unparsing rules in SSL format or PPML programs, rather than writing them by hand as is currently done in CENTAUR and the Synthesizer Generator.

We based our box language on the boxes of PPML. The basic ideas of Box originate from Knuth's TEX, but we follow the approach of PPML. The basic operators in Box and those of PPML differ only in technical details. One of the disadvantages of PPML is that it can only be used within the CENTAUR system whereas in our approach it is possible and feasible to generate a standalone program outside the context of the ASF+SDF Metaenvironment. A PPML program is interpreted by a PPML formatting engine or by the FIGUE formatting engine [Hascoet 1992]. These formatting engines are similar to the back-ends for Box. In PPML the operators are hardwired in the language, whereas Box offers the possibility to extend the set of operators. The construction of new back-ends for Box is also possible.

The VIZ/UAL unparsing mechanism described in Garlan [1985] and the SbyS editor [Minör 1990] of the Mjølner/ORM environment [Magnusson et al. 1990] offer facilities to have different views of the same object in different windows. This is a feature which is not supported by our current box formatters. The elision mechanisms of the above-mentioned systems and of the language-based editor UQ2 [Welsh et al. 1991] also go beyond the elision mechanism in our generated formatters. Finally, the UQ2 editor [Welsh et al. 1991] offers an interesting feature with respect to the production of documentation—namely, the production of program documentation can be done in parallel with developing the program text. This is achieved via editors with two editable views: one for the documentation and one for the program text. This is complementary to our treatment of comments: our system literally restores comments at their original positions, whereas in the UQ2 editor comments are treated in a more restricted way.

9. CONCLUSIONS

In this article we have presented a method for generating formatting tools for arbitrary languages based on the definition of their syntax. The generated formatter is an algebraic specification of a family of functions that compositionally map the constructs of a language to box expressions. A box expression is a declarative, language-independent description of the layout of programs. It can be translated by various back-ends to ASCII, TEX, or other text representations. The formatters reconstruct brackets and comments that are usually thrown away by the parser. Extensions of the box language provide powerful constructs for fonts and alignments.

The formatter generator presented in this article is, in combination with the formatters for boxes [van den Brand and Visser 1994], a practical support tool for the construction of documentation tools. It has been applied to generate parts of programming environments for various languages. Some of these parts are: error messages for a SEAL type checker [Koorn 1994]; a typesetter for the specification language μCRL [Hillebrand 1996]; a formatter for Prolog [SICS 1992] used in a transformation system for Prolog programs [Brunekreef 1995b]; a formatter for a subset of the functional language CLEAN [Plasmeijer and van Eekelen 1993] used in a transformation system for CLEAN programs [van den Brand et al. 1995]; a pretty-printer for Cobol used as a back-end of a Cobol generator [Res 1994]; a typesetter for the process specification formalism PSF used to typeset the specifications in Brunekreef [1995a]. The latter two tools were discussed in Sections 6.4 and 6.5. The generated formatters save a considerable amount of document preparation time, and the generation of the formatters saves a considerable amount of tool specification and implementation time. This is especially important when applied to new and experimental languages.

Both the formatter generator itself and the generated formatters are specified algebraically using ASF+SDF. Because of the high level of specification and the support provided by the ASF+SDF Metaenvironment, the generator and generated tools are easy to maintain, document, and adapt.

Our approach for generating formatters can also be applied to systems, such as the Synthesizer Generator [Reps and Teitelbaum 1989] and CENTAUR [Borras 1989; Borras et al. 1989], where the formatting rules have to be specified by hand.

The techniques described in this article are a starting point for generating more advanced formatters. Currently the formatting tools are used to experiment with visualization of programs in the context of reverse engineering. Some of the issues involved are the automatic generation of comments to emphasize peculiarities in a program and the use of an elision mechanism to clarify the structure of certain constructs. It is interesting to investigate the combination of elision, color, and hypertext to display the results of semantic analysis of programs. These techniques are crucial for a better understanding of poorly documented programs.

ACKNOWLEDGMENTS

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 4.3. We thank Paul Klint, Emma van der Meulen, Susan

Üsküdarlı, and Chris Verhoef for reading draft versions of this article. Finally, we thank the referees for their useful comments.

REFERENCES

BAHLKE, R. AND SNELTING, G. 1986. Context-sensitive editing with PSG environments. In *Proceedings of the International Workshop on Advanced Programming Environments*, R. Conradi, T. Didriksen, and D. Wanvik, Eds. Lecture Notes in Computer Science, vol. 244. Springer-Verlag, Berlin, 26–38.

BERGSTRA, J. A. AND KLINT, P. 1995. The discrete time ToolBus. Tech. Rep. P9502, Programming Research Group, Univ. of Amsterdam, Netherlands. Available as ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1995/P9502.ps.Z.

BERGSTRA, J. A., HEERING, J., AND KLINT, P. 1989. The algebraic specification formalism ASF. In *Algebraic Specification*, J. A. Bergstra, J. Heering, and P. Klint, Eds. Addison-Wesley, Reading, Mass., 1–66.

BLASCHEK, G. AND SAMETINGER, J. 1989. User-adaptable prettyprinting. *Softw. Pract. Exp. 19,* 7, 687–702.

BORRAS, P. 1989. *PPML—Reference Manual and Compiler Implementation.* INRIA, Sophia-Antipolis, France.

BORRAS, P., CLEMENT, D., DESPEYROUX, T., INCERPI, J., LANG, B., AND PASCUAL, V. 1989. CENTAUR: The system. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. *SIGPLAN Not. 14,* 2, 14–24.

BRUNEKREEF, J. J. 1995a. On modular algebraic protocol specification. Ph.D. thesis, Univ. of Amsterdam, Netherlands.

BRUNEKREEF, J. J. 1995b. TransLog, an interactive tool for transformation of logic programs. Tech. Rep. P9512, Programming Research Group, Univ. of Amsterdam, Netherlands. Available as ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1995/P9512.ps.Z.

CORDY, J. R., ELIOT, N. L., AND ROBERTSON, M. G. 1990. Turingtool: A user interface to aid in the software maintenance task. *IEEE Trans. Softw. Eng. 16,* 3, 294–301.

GARLAN, D. 1985. Flexible unparsing in a structure editing environment. Tech. Rep. CMU-CS-85-129, Carnegie-Mellon Univ., Pittsburgh, Pa.

HASCOET, L. 1992. *FIGUE An Incremental Graphic Formatter User's Manual for Version 1.* INRIA, Sophia-Antipolis, France.

HECKMANN, R. AND WILHELM, R. 1995. Formula layout. Tech. Rep. A 07/95, FB 14 Informatik, Universität des Saarlandes, Saarbrücken, Germany.

HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. 1992. *The Syntax Definition Formalism SDF—Reference Manual.* Version 6. CWI, Amsterdam, Netherlands. Available as ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z. Dec. Earlier version appeared in *SIGPLAN Not. 24,* 11 (1989), 43–75.

HILLEBRAND, J. A. 1996. A small language for the specification of grid protocols. Tech. Rep., Programming Research Group, Univ. of Amsterdam, Netherlands. To appear.

JOHNSSON, T. 1986. Target code generation from G-machine code. In *Graph Reduction,* J. F. Fasel and R. M. Keller, Eds. Lecture Notes in Computer Science, vol. 279. Springer-Verlag, Berlin, 119–159.

JOKINEN, M. O. 1989. A Language-independent prettyprinter. *Softw. Pract. Exp. 19,* 9, 839–856.

KAMPERMAN, J. F. T. 1994. GEL, a graph exchange language. Tech. Rep. CS-R9440, CWI, Amsterdam, Netherlands. Available as ftp://ftp.cwi.nl/pub/gipe/reports/Kam94.ps.Z.

KAMPERMAN, J. F. T. AND WALTERS, H. R. 1993. ARM, abstract rewriting machine. Tech. Rep. CS-9330, CWI, Amsterdam, Netherlands. Available as ftp://ftp.cwi.nl/pub/gipe/reports/KW93.ps.Z.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1978. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

KLINT, P. 1993. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Meth. 2,* 2, 176–201.

Klint, P. 1994. Writing meta-level specifications in ASF+SDF. CWI, Amsterdam, Netherlands.

Klint, P. 1995. The ASF+SDF *meta-environment—user's guide*. CWI, Amsterdam, Netherlands. Available as ftp://ftp.cwi.nl/pub/gipe/reports/SysManual.ps.Z.

Knuth, D. E. 1984. *The TEXbook*. Vol. A, *Computers and Typesetting*. Addison-Wesley, Reading, Mass. (Ninth printing, revised, October 1989).

Koorn, J. W. C. 1994. Generating uniform user-interfaces for interactive programming environments. Ph.D. thesis, ILLC dissertation series 1994-2, Univ. of Amsterdam, Netherlands.

Lesk, M. E. and Schmidt, E. 1986. LEX—A lexical analyzer generator. UNIX Programmer's Supplementary Documents, volume 1 (PS1). Bell Laboratories, Murray Hill, N.J.

Magnusson, B., Bengtsson, M., Dahlin, L.-O., Fries, G., Gustavsson, A., Hedin, G., Minor, S., Oscarsson, D., and Taube, M. 1990. An Overview of the Mjølner/ORM Environment: Incremental language and software development. In *Proceedings of TOOLS '90*. Prentice-Hall, Englewood Cliffs, N.J., 635–646.

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

Meyer, B. 1992. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, N.J.

Minör, S. 1990. On structure-oriented editing. Ph.D. thesis, Lund Univ., Lund, Sweden.

Morcos-Chounet, E. and Conchon, A. 1986. PPML: A general formalism to specify prettyprinting. In *Information Processing 86,* H.-J. Kugler, Ed. Elsevier, Amsterdam, 583–590.

Oppen, D. C. 1980. Prettyprinting. *ACM Trans. Program. Lang. Syst. 2,* 4, 465–483.

Plasmeijer, M. J. and van Eekelen, M. C. J. D. 1993. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, Mass.

Reps, T. and Teitelbaum, T. 1989. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, Berlin.

Res, M. 1994. A generated programming environment for RISLA, a specification language for defining financial products. M.S. thesis, Programming Research Group, Univ. of Amsterdam, Netherlands.

Rose, G. A. and Welsh, J. 1981. Formatted Programming Languages. *Softw. Pract. Exp. 11,* 651–669.

SICS. 1992. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, Kista, Sweden.

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

van den Brand, M. G. J. 1992. Pregmatic, a generator for incremental programming environments. Ph.D. thesis, Katholieke Universiteit Nijmegen, Netherlands.

van den Brand, M. G. J., Eijkelkamp, S. M., Geluk, D. K. A., Meijer, H., Osborne, H. R., and Polling, M. J. F. 1995. Program transformations using ASF+SDF. In *ASF+SDF '95: A Workshop on Generating Tools from Algebraic Specifications*, M. G. J. van den Brand, A. van Deursen, T. B. Dinesh, J. F. T. Kamperman, and E. Visser, Eds. Programming Research Group, Univ. of Amsterdam, Netherlands, 29–52. Also Tech. Rep. P9504, Programming Research Group, Univ. of Amsterdam. Available as http://www.fwi.uva.nl/research/prog/reports/.

van den Brand, M. G. J. and Visser, E. 1994. From Box to TEX: An algebraic approach to the construction of documentation tools. Tech. Rep. P9420, Programming Research Group, Univ. of Amsterdam, Netherlands. Available as ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1994/P9420.ps.Z.

Vos, K. J. 1990. Pretty for an easy touch of beauty. M.S. thesis, Programming Research Group, Univ. of Amsterdam, Netherlands.

Walters, H. R. 1991. On equal terms, implementing algebraic specifications. Ph.D. thesis, Univ. of Amsterdam, Netherlands. Available as ftp://ftp.cwi.nl:/pub/gipe/reports/Wal91.ps.Z.

Welsh, J., Broom, B., and Kiong, D. 1991. A design rationale for a language-based editor. *Softw. Pract. Exp. 21,* 9, 923–948.

Wijngaarden, A., Mailloux, B., Peck, J., Koster, C., Sintzoff, M., Lindsey, C., Meertens, L., and Fisker, R. 1976. *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag, Berlin.

Received July 1995; revised December 1995; accepted January 1996