To reuse or to be reused. Techniques for component composition and construction

de Jonge, M.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Pretty-Printing for Software Reengineering

Pretty-printing forms an integral part of Language-Centered Software Engineering (LCSE). To facilitate reuse, generic (i.e., language-independent) and customisable pretty-print technology is needed. In this chapter we discuss such pretty-print technology in the context of software reengineering.

A typical application domain of LCSE is software reengineering. Software reengineering puts strong requirements on pretty-print technology. These include layout and comment preservation, as well as customisable format definitions. From a maintenance perspective, software reengineering requires reusability of format engines and of format definitions.

In this chapter we present the Generic Pretty-Printer \( \text{GPP} \) and discuss the pretty-print techniques that it uses to fulfill the requirements for software reengineering. \( \text{GPP} \) forms a generally reusable pretty-print component in our language-centered architecture and is part of the \( \text{XT} \) bundle discussed in the previous chapter. Applications, such as COBOL reengineering and SDL documentation generation (which will be discussed in the next chapter) show that our pretty-print techniques are feasible and successful. The work presented in this chapter was published earlier as [80].

4.1 Introduction

Software reengineering is concerned with changing and repairing existing software systems. It is often language-dependent and customer-specific.

For instance, Dutch banks have to standardize their bank account numbers
before the second quarter of 2004 [105]. To that end, a restructuring reengineering [43] might be implemented for a particular Dutch bank to reengineer his COBOL-85 dialect, by changing account numbers from 9 to 10 digits while preserving his specific coding conventions. Although the reengineering itself is of general use for all Dutch banks, this specific implementation is hard to reuse.

When a reengineering company wants to develop such reengineerings for different customers and different language dialects (for instance to support the bank account number reengineering for some other of the 300 existing COBOL dialects [95]), problematic reuse may easily lead to a significant maintenance effort. A reengineering company would therefore benefit when reuse of reengineerings could be improved, such that reengineerings for new customers or language dialects can be developed rapidly from existing ones and time to market can be decreased [114].

Developing reusable reengineerings requires advanced language technology to easily deal with multiple customers and language dialects. The literature contains many articles addressing flexible parsing and processing techniques. Flexible, reusable pretty-printing techniques are not very well addressed and are the subject of this chapter.

Pretty-printing in the area of software reengineering serves two purposes. Firstly, for automatic software reengineering pretty-printing is used for source (re-) generation, to transform the abstract representation of a reengineered program back to human readable textual form. Usually, a pretty-printer is then the last phase in a reengineering pipeline. Programs are first parsed, then reengineered (for instance by transformation), and finally pretty-printed.

Secondly, for semi-automatic (or manual) software reengineering pretty-printing is used for documentation generation [57]. In this case, the reengineering process requires user intervention and documentation generation is used to make programs easily accessible. To that end, a pretty-printer is used to format programs nicely and can be combined with additional program understanding techniques to enable analysis and inspection of programs to determine where and how reengineering steps are required. Examples are web-site generation and document generation.

The area of software reengineering introduces challenging functional requirements for pretty-printing:

- To support source generation and documentation generation, a pretty-printer should be able to produce multiple output formats. We define formatings independently of such output formats in pretty-print rules [106].

- The modifications made during software reengineering to the original source text should be minimal [136]. Only the parts that need repairing should be modified. This implies that comments and existing layout should be preserved. To that end, we propose conservative pretty-printing.
Customer-specific format conventions should be respected. To yield a program text that looks like the original one, a pretty-printer is required that produces a customer-specific formatting. We propose customizable pretty-printing to meet such specific conventions.

In order to cost-effectively develop pretty-printers for various programming languages, tailored towards different customer-specific format conventions, maintenance effort should be minimal. To reduce maintenance effort, reuse across different pretty-printers should be promoted. To that end, we formulate a number of technical requirements:

- A formatting definition should be reusable for a language, its dialects, and for defining customized formatings. We propose to group pretty-print rules in modular pretty-print tables, which allow a formatting to be defined as a composition of new and existing pretty-print tables.

- Adding support for a new language should be easy. We propose using generic formatting engines, which can interpret pretty-print rules of arbitrary languages. Moreover, we simplify creating new formatting definitions with pretty-print table generation.

- Software reengineering can be applied to different representations of programs each requiring specific pretty-print techniques. We address two such representations (i.e., parse-trees and abstract syntax trees), and we propose to reduce maintenance cost by sharing formatting definitions between the corresponding formatting engines.

In contrast to the existing literature, which usually concentrates on a specific aspect of pretty-printing [79], this chapter strives to give a complete discussion of pretty-printing in the area of software reengineering. We start with a discussion of conceptual foundations of pretty-printing for software reengineering. Then we describe the generic pretty-print framework GPP. Finally, we cover existing reengineering projects conducted with GPP, including COBOL reengineering and SDL documentation generation.

### 4.2 Pretty-printing

Pretty-printing is concerned with transforming abstract representations of programs to human readable form. Pretty-printing is like unparsing, but additionally, is concerned with producing nicely (pretty) formatted output. The result of pretty-printing can be plain text which is directly human readable, or a document in some markup language such as HTML.

In this chapter we consider two types of abstract representations: (full) parse trees (see Figure 4.1) and abstract syntax trees (see Figure 4.2). Full parse trees (also called concrete syntax trees) contain all lexical information, including comments and ordinary layout.
Pretty-Printing for Software Reengineering  

Chapter 4

Figure 4.1 Example of a full parse tree containing layout and comments. Layout nodes are denoted by '.', comments start with '%'.

A parse tree (PT) can be pretty-printed progressively, which means that all layout will be generated. For the PT of Figure 4.1 this implies that the comment and '.' nodes are discarded and replaced by newly generated layout strings. Generation of layout can also be less progressive by preserving (some or all) of the existing layout nodes. We call this conservative pretty-printing.

An abstract syntax tree (AST) must always be pretty-printed progressively, since it does not contain layout nodes. An extra challenge of AST pretty-printing is that literals (i.e., the keywords of the program) should be reconstructed.

Pretty-printing consists of two phases [113]. First, the abstract representation is transformed to an intermediate format containing formatting instructions. Then, the formatting instructions are transformed to a desired output format.

The intermediate format that is obtained during the first phase of pretty-printing can be represented as a tree (see Figure 4.3). The nodes of this format tree correspond to format operators and denote how to layout the underlying leaves (for example, H for horizontal, and V for vertical formatting). This phase is thus basically a tree transformation in which an AST or PT is transformed to a format tree. During the second phase, the format tree is used to produce the corresponding layout in the desired output format.

We propose to define the transformation to a format tree as a mapping from language constructs (grammar productions) to corresponding format constructs. As we will see in Section 4.4, such mappings can be shared for transforming PTs and ASTs. This makes pretty-printing of both tree types, based on a single transformation definition, possible. We will also see in Section 4.4 how the construction of such language-specific mappings is simplified by generating them from corresponding grammars.
4.3 Pretty-printing for software reengineering

This section addresses requirements for pretty-printing in the context of software reengineering as well as corresponding solutions.

4.3.1 Multiple output formats

Pretty-printing for software reengineering serves two purposes: i) as back-end of an automated reengineering process; ii) as part of a documentation generator. In the first case a pretty-printer is used to transform a reengineered program to plain text, such that it can be further developed, compiled etc. In the latter case, it is used to produce a visually attractive representation of programs for program understanding purposes.

Both purposes demand for different output formats: plain text in case the pretty-printer serves as back-end, and a high-quality format (such as \LaTeX{}, HTML, or PDF) for a documentation generator.

To limit maintenance cost of a pretty-printer due to code duplication we divide a pretty-printer in two separate components, a format tree producer and a format tree consumer. The first produces a language-specific formatting represented as format tree, while the latter transforms such a tree to an output format. This division makes a producer independent of the output format and a consumer independent of the input language.

A pretty-printer for input language \(i\) and output format \(o\) now consists of the composition:

\[
pp_i^o = \text{format tree producer}_i + \text{format tree consumer}_o
\]

By developing different format tree consumers, a format tree can be transformed to multiple output formats. In Section 4.4 we discuss the implementation of three such consumers which produce plain text, HTML, and \LaTeX{}.

This architecture reduces maintenance effort because each format tree consumer can be reused as-is in all pretty-printers. Once the pretty-print rules for
4.3.2 Layout preservation

An important function of layout is to improve the understandability of programs. Such layout is inserted by developers and does not always follow strict format conventions. It may contain slight adaptations, for instance to group certain lines of code together, or to make a statement fit nicely on a single line.

With standard (progressive) pretty-print techniques such formattings will disappear. Consequently, layout occurring in program fragments that are not even touched by the actual program reengineering will not survive.

Clearly, for serious software reengineering, it is essential that the formatting of unaffected program parts will be preserved [136]. Only the affected parts should be formatted automatically using a customized pretty-printer. Therefore the use of conservative pretty-printing is inevitable.

Conservative pretty-printing produces a format tree which may contain original layout nodes (such as the node ‘\ldots’ in Figure 4.1). Conservative pretty-printing therefore only works on full parse trees because ASTs, in general, do not contain layout nodes.

Conservative pretty-printing operates on PTs in which the layout nodes that should be preserved are marked. We defined an algorithm for conservative pretty-printing that consists of two steps. First, a PT is mapped to a format tree using progressive pretty-printing. Second, the nodes that were marked in the original PT are inserted in the format tree. To combine these layout nodes with non-layout nodes, we introduce a special empty format operator $\epsilon$. This operator joins its sub-trees without producing any layout. For each layout node that needs to be preserved, the insertion process is defined as follows:

1. The terminal symbol occurring left to a layout node is removed from the format tree (e.g., the second 'x' in Figure 4.3 in case the layout string '---' between the symbols 'x' and ':=' in Figure 4.1 has to be preserved).
Section 4.3  Pretty-printing for software reengineering

Figure 4.4 Format tree in which existing layout (\ldots) is preserved.

2. The format tree right to that terminal symbol is determined and removed (the node `\texttt{:=}' in Figure 4.3).

3. The terminal symbol (\texttt{'}x\texttt{'}), the layout node (\texttt{'}\ldots\texttt{'}), and the format tree (\texttt{'}\ldots\texttt{'}) are then inserted into a new sub-tree which has the empty format operator \(\epsilon\) as root.

4. This tree is inserted in the original format tree, at the location of the terminal symbol that was removed at step 1.

Applying these steps to the format tree of Figure 4.3 yields the tree as depicted in Figure 4.4.

To transform a format tree with layout nodes to text, we use a format function \(f\) that operates on format expressions and produces text. It is defined as follows:

\[
\begin{align*}
f(\epsilon(t_1, \ldots, t_n)) &= f(t_1) \cdot \ldots \cdot f(t_n) \\
f(\phi(t_1, \ldots, t_n)) &= f(t_1) \cdot l_\phi \cdot f(t_2) \cdot \ldots \cdot l_\phi \cdot f(t_n) \\
&\text{for each format operator } \phi \\
f(s) &= s \text{ for each non-terminal symbol } s \\
f(w) &= w \text{ for each layout string } w
\end{align*}
\]

The operator `\cdot' denotes string concatenation, \(l_\phi\) denotes the layout string as generated by the format operator \(\phi\). This definition states that sub-trees of \(\epsilon\) are only concatenated, while sub-trees of other operators are also separated by layout strings.

For example, when we define \(l_H = \omega\) and \(l_V = \backslash n\), then we can translate the format tree of Figure 4.4 to text using the following derivation:

\[
\begin{align*}
f(V(H(\text{if }, x, \text{then } x \leftarrow \text{:= } 0 \text{ fi })), H(\epsilon(x, \ldots, \text{:= } 0), \text{fi }))) &= \\
f(H(\text{if }, x, \text{then } x \leftarrow \text{:= } 0)) \cdot l_V \cdot f(H(\epsilon(x, \ldots, \text{:= } 0)), \text{fi }) \cdot l_V \cdot \text{fi } &=
\end{align*}
\]
Since the \( \epsilon \) operator produces no layout, the string that separates the two adjacent terminal symbols 'x' and ':=’ is exactly the layout string that had to be preserved. All other layout strings are generated according to the (customerspecific) pretty-print rules.

### 4.3.3 Comment preservation

Like layout, comments also serve to improve the understandability of programs. Such information, which might include important descriptions and instructions to developers, should in all cases be preserved. Since an ordinary progressive pretty-printer would destroy this information, a comment preserving pretty-printer is required for software reengineering.

Comment preservation is similar to layout preservation and is also only defined on full parse trees. The \( \epsilon \) operator is used to insert a comment into a format tree (like for inserting ordinary white space). In addition, we introduce a new format operator \( \gamma \) to mark comments in format trees. This operator serves documentation generation and literate programming [89]. It allows comments to be formatted explicitly in non-text formats (for instance in a separate font).

The format tree that is obtained from Figure 4.3 by inserting '%a comment' is depicted in Figure 4.5.

### 4.3.4 Customizability

When performing automatic software reengineering for a customer, the resulting programs should have a similar formatting as the original ones. When different customers are served, this requires the availability of several format engines, each producing the formatting of a particular customer. Developing
Section 4.3  Pretty-printing for software reengineering

each of them from scratch is a lot of work and easily leads to undesired maintenance effort. Instead, reusing existing pretty-print engines and customizing their behavior is preferable.

To make pretty-printers easily customizable, we advocate pretty-printing using pretty-print rules [106]. Pretty-print rules are mappings from language constructs to formatting constructs. Each mapping defines how a language construct should be formatted. Pretty-print rules are defined declaratively and interpreted by a formatting engine. Constructing a format tree from a PT or AST now consists of a tree transformation in which the exact transformation is defined by a set of pretty-print rules. Customization is achieved by supplying different rules to a formatting engine.

By using this interpreted approach, the same formatting engine can be used for every language and all customers. Only pretty-print rules have to be defined to develop a pretty-printer for a language. Furthermore, existing rules can be redefined to customize a formatting definition.

4.3.5  Modularity

Software reengineering requires pretty-print technology that makes dealing with language dialects, embedded languages, and customer-specific formatings easy.

To facilitate this, development and maintenance time should be decreased by allowing new pretty-printers to be constructed from existing ones. In Section 4.3.4, we already pointed out how this can be achieved by separating pretty-print rules and formatting engines. Pretty-print engines can be reused for all different customers, only pretty-print rules have to be defined for each of them.

However, only a small portion of an existing set of pretty-print rules needs to be changed usually, when adding support for a new customer-specific format or language dialect. Pretty-printer construction would therefore be further simplified when the unchanged pretty-print rules could also be reused. With modular pretty-print tables this is achieved.

We therefore group pretty-print rules in pretty-print tables. By prioritizing each table, pretty-print rules in a table with higher priority will override the rules in tables of lower priority. The set of pretty-print rules \( \rho \) obtained by combining two tables \( t_1 \) and \( t_2 \) (where \( t_1 \) has highest priority), is defined as:

\[
\rho = t_1 \cup (t_2 \setminus t_1)
\]

Definition of formatings with modular pretty-print tables works as follows:

Language dialects  Pretty-print rules corresponding to new or affected language constructs are defined in a new set of pretty-print tables \( T_{new} \). The complete formatting definition is obtained by merging \( T_{new} \) with the pretty-print tables of the existing (base) language. Thus, only the pretty-print rules in \( T_{new} \) have to be defined, the rest can be reused.
Embedded languages A formatting definition for a language \( L \) which embeds \( L_e \) is obtained by combining the format rules of \( L \) with the format rules of \( L_e \). For instance, suppose you already have pretty-print rules for the language COBOL and SQL, then building a pretty-printer for COBOL with embedded SQL, amounts to combining both sets of pretty-print rules.

Customer specificity A customer-specific formatting definition for a language \( L \) can be defined by combining customer-specific pretty-print rules and standard pretty-print rules. Only the customer-specific rules have to be defined which override the reused default rules.

An arbitrary number of pretty-print tables can be merged this way, which makes extensive reuse of pretty-print rules possible. For instance, to define a customer-specific formatting definition for an embedded language dialect.

### 4.4 GPP: a generic pretty-printer

GPP is a generic pretty-printer that implements the ideas and techniques discussed in Section 4.2 and 4.3. GPP's architecture is shown in Figure 4.6. It

---

1See Appendix A for information about the availability of GPP.
Section 4.4  

GPP: a generic pretty-printer

context-free syntax

<table>
<thead>
<tr>
<th>Production</th>
<th>Syntax Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;if&quot; EXP &quot;then&quot; STAT &quot;fi&quot;</td>
<td>STAT {cons(&quot;IF&quot;)}</td>
</tr>
<tr>
<td>ID := EXP</td>
<td>STAT {cons(&quot;ASSIGN&quot;)}</td>
</tr>
<tr>
<td>ID</td>
<td>EXP {cons(&quot;ID&quot;)}</td>
</tr>
<tr>
<td>NAT-CON</td>
<td>EXP {cons(&quot;NAT&quot;)}</td>
</tr>
<tr>
<td>STR-CON</td>
<td>EXP {cons(&quot;STR&quot;)}</td>
</tr>
</tbody>
</table>

Figure 4.7  Grammar productions in the Syntax Definition Formalism SDF, corresponding to the PT and AST of Figure 4.1 and 4.2.²

consists of the pretty-print table generator ppgen, two format tree producers pt2box and ast2box (see Section 4.4.2), and the format tree consumers box2text, box2html, and box2latex (see Section 4.4.3). Format tree producers and consumers are implemented as separate components which exchange format trees represented in the markup language BOX. This architecture can easily be extended by additional tools that produce or consume BOX format trees (see Section 4.5).

4.4.1 Format definition

The first phase of pretty-printing consists of transforming a PT or AST to a format tree. This section discusses how the transformation can be defined as mapping from language productions in SDF to format constructs in BOX.

Syntax definition  We use the Syntax Definition Formalism SDF [68, 137] to define language productions (see Figure 4.7 for some examples). SDF allows modular, purely declarative syntax definition. In combination with generalized LR (GLR) parser generation, the full class of context-free grammars is supported.

SDF allows concise syntax definitions with syntax operators in arbitrary nested productions. For instance, using the ‘(’ and ‘?’ operators which denote sequences and optionals, respectively, the if-construct of Figure 4.7 can be extended with an optional else-branch as depicted in Figure 4.8.

Together with concrete syntax, corresponding abstract syntax can also be defined in SDF. This is achieved by defining the constructor names of the abstract syntax as annotations to the concrete syntax productions (the cons attributes in Figure 4.7 and 4.8). These define the node names of abstract syntax trees. Chapter 2, “Grammars as Contracts”, discusses generation of abstract syntax from concrete syntax in more detail.

²Productions in SDF are reversed with respect to formalisms like BNF: on the right hand side of the arrow is the non-terminal symbol that is produced by the symbols on the left-hand side of the arrow.
context-free syntax

"if" EXP "then" STAT ("else" STAT )? "fi" → STAT {cons("if")}

**Figure 4.8** Extending the if-construct of Figure 4.7 with an optional else-branch using the SDP syntax operators 

As we will see shortly, constructor names also serve to identify productions and to select proper pretty-print rules.

**Format expressions** We use the language BOX [33, 79] for defining format-tings. BOX is a markup language which allows formatting definitions to be expressed as compositions of boxes. Box composition is accomplished using box operators (Table 4.1 lists available operators).

The language distinguishes *positional* and *non-positional* operators. Positional operators are further divided in *conditional* and *non-conditional* operators. Examples of non-conditional, positional operators are the H and V operators, which format their sub-boxes horizontally and vertically, respectively:

\[
H \left[ \begin{array}{c|c|c} B_1 & B_2 & B_3 \end{array} \right] = \begin{array}{c|c|c} B_1 & B_2 & B_3 \end{array}
\]

\[
V \left[ \begin{array}{c|c|c} B_1 & B_2 & B_3 \end{array} \right] = \begin{array}{c|c|c} B_1 & B_2 & B_3 \end{array}
\]

Conditional operators take the available line width into account. An example is the ALT operator:

\[
ALT \left[ \begin{array}{c|c} B_1 & B_2 \end{array} \right] = \begin{array}{c|c} B_1 & B_2 \end{array}
\]

It either formats its first sub-box if sufficient width is available, or its second otherwise.

The exact formatting of positional operators can be controlled using *space options*. For example, to control the amount of horizontal space between boxes, the H operator supports the hs space option:

\[
H_{hs=2} \left[ \begin{array}{c|c|c} B_1 & B_2 & B_3 \end{array} \right] = \begin{array}{c|c|c} B_1 \hspace{2} B_2 \hspace{2} B_3 \end{array}
\]

The non-positional operators of the BOX language are used for cross referencing and for specifying text attributes (such as font and color). They are also used to structure text, for instance by marking parts as comment text, as variable, or as keyword.

BOX does not have an explicit empty format operator \(\epsilon\), which is needed for conservative pretty-printing (see Section 4.3.2). However, this operator can
Table 4.1  Positional and non-positional BOX operators, together with supported space options (hs defines horizontal layout between boxes, vs defines vertical layout between boxes, and is defines left indentation).

be mimicked using the H operator as \( H_{hs=0} \ldots \). The comment operator \( \gamma \) used for comment preservation (see Section 4.3.3) is represented using the C operator in BOX.

Pretty-print rules  We can now define pretty-print rules as mappings from SDF productions to BOX expressions, and pretty-print tables as comma separated lists of pretty-print rules (see Figure 4.9).

BOX expressions in pretty-print tables contain numbered place holders (.1 and .2 in Figure 4.9) which correspond to non-terminal symbols in SDF productions. During pretty-printing, place holders are replaced by BOX-expressions that are generated for these non-terminal symbols.

Figure 4.9  A pretty-print table for the grammar of Figure 4.7.
Pretty-Printing for Software Reengineering

Figure 4.10  Pretty-print table for the nested if-construct of Figure 4.8.

Constructor annotations of SDF productions serve as keys in pretty-print tables. Since they are contained in the parse tree format that we use (see Section 4.4.2) and (as node names) in ASTs (see Figure 4.2), they can be used to format both tree types.

The pretty-print table of Figure 4.9 only contains pretty-print entries for flat (non-nested) SDF productions. To enable the definition of a formatting for an arbitrary nested SDF production, a separate pretty-print rule can be defined for each nested symbol in a production. Such pretty-print rules are identified by the path from the result sort of the production to the nested symbol.

For example, Figure 4.10 contains pretty-print entries for the nested SDF production of Figure 4.8. The first rule has only the constructor name (IF) as key and corresponds to the top-level sequence of symbols of the SDF production (i.e., the sequence of children of the root node of the tree depicted in Figure 4.11). The remaining pretty-print rules correspond to the nested symbols. For each path from the root node in Figure 4.11 to a nested symbol (denoted as grey ellipse), a pretty-print rule is defined.

All path elements contain indexes which are obtained by numbering the non-terminal symbols contained in nested symbols. Symbol names are required as part of path elements for pretty-printing ASTs. This is further discussed in Section 4.4.2.

Pretty-print rule generation  Writing pretty-print tables can be a time-consuming and error-prone process, even for small languages. To simplify pretty-print table construction, we implemented a pretty-print table generator (the component ppgen in Figure 4.6). Given a syntax definition in SDF, this generator produces pretty-print rules for all SDF productions and for all paths to nested symbols.

Literals in SDF productions are recognized as keywords and formatted with the KW operator. Several heuristics are used to make a rough estimation about vertical and horizontal formatting. This is achieved by recognizing several structures of SDF productions and generating compositions of H and V boxes accordingly. For most productions however, no explicit formatting definition is generated. That is, a pretty-print rule is generated without positional BOX-operators (such as the last two pretty-print rules in Figure 4.10). This makes generated tables easy to understand and to adapt.

With the generator, a pretty-printer for a new language can be obtained for free. Although the formatting definition thus obtained is minimal, it is di-
Section 4.4  

GPP: a generic pretty-printer

Figure 4.11  Graphical structure of the nested SDF production of Figure 4.8. Grey ellipses denote nested SDF symbols.

rectly usable. To improve the layout, we can benefit from the customizability of pretty-print tables and incrementally redefine the formatting definition. Only pretty-print rules that do not satisfy have to be redefined. Customized pretty-print rules can be grouped in separate tables such that regeneration of tables after a language change does not destroy the customized rules.

4.4.2 Format tree producers

This section discusses pt2box and ast2box which generate format trees from PTs and ASTs, respectively.

Formatting parse trees  Formatting PTs is implemented in pt2box which supports conservative and comment preserving pretty-printing. It operates on a universal PT format, called AsFix (see Chapter 2, “Grammars as Contracts”). This format can represent PTs for arbitrary context-free languages and contains all lexical information including layout and comments. In addition, constructor symbols are also available because each node in an AsFix PT contains the complete grammar production that produced the node (see Figure 2.4 on page 23 for an example of an AsFix PT).

The format tree producer pt2box first collects all layout nodes from a PT. Then it constructs a BOX-expression during a bottom-up traversal of the PT. Finally, it inserts some of the collected layout nodes in the BOX-term. Layout is inserted as follows: comments are always inserted; original non-comment layout is only inserted when conservative pretty-printing is turned on; newly created layout is never inserted (see Section 4.3.2 about layout insertion).

BOX expressions are constructed by obtaining and instantiating pretty-print rules from pretty-print tables. Pretty-print rules can also be generated dynamically because all information required for rule generation is available in AsFix. Part of the functionality of the pretty-print generator ppgei is therefore reused in pt2box to dynamically generate pretty-print rules. This makes pretty-print rules dispensable because rules that are missing are generated on the fly.
Formatting abstract syntax trees  

Generation of format trees from ASTs is implemented in `ast2box`. Given a set of pretty-print rules, `ast2box` produces the same format tree as `pt2box`, except for preserved layout strings. ASTs do not contain layout and consequently, conservative nor comment preserving pretty-printing is supported by `ast2box`.

Abstract syntax is generated from SDF definitions based on the constructor annotations of grammar productions. These constructor annotations define the node names of ASTs. See Chapter 2, “Grammars as Contracts”, for a discussion about abstract syntax generation from concrete syntax definitions. Since constructor names also serve as keys in pretty-print tables, the name of a node in an AST can be used to obtain the corresponding pretty-print rule (see Figures 4.2 and 4.9).

In an AST, certain types of symbols are indistinguishable although they require different formatting. The symbol names in the keys of pretty-print rules serve to be able to always determine the types of symbols, such that the correct formatting can be applied.

For instance, SDF supports comma separated lists which have the same abstract syntax as ordinary lists. The separator symbols are not contained in the abstract syntax and have to be reproduced during pretty-printing. Consequently, comma separated lists require special treatment and should be distinguished from ordinary lists. This can be achieved with the information in the paths of corresponding pretty-print rules.

Keywords are not contained in ASTs and have to be reproduced during pretty-printing. ASTs also lack information to generate pretty-print rules containing these keywords dynamically. In contrast to `pt2box`, it is therefore essential for `ast2box` that pretty-print rules are defined for each constructor in the AST.

4.4.3 Format tree consumers

The last phase of pretty-printing consists of transforming a format tree to an output format. The architecture of GPP allows an arbitrary number of such format tree consumers. We implemented three of them which produce plain text, \texttt{\LaTeX}, and HTML as output format, respectively. PDF can also be generated but indirectly from generated \texttt{\LaTeX} code.

From \texttt{BOX} to text  
The \texttt{box2text} component transforms a format tree to plain text. It fully supports comment preservation and conservative pretty-printing. The transformation consists of two phases. During the normalization phase all non-positional operators (except the C comment operator) are discarded, and positional operators are transformed to H and V boxes. A normalized \texttt{BOX-term} only contains non-conditional operators and absolute, rather than relative offsets as space options. Then, during the second phase, the normalized \texttt{BOX-term} is transformed to text. This amounts to producing non-
terminal symbols and layout strings. The latter are computed based on the absolute space options.

**From Box to \LaTeX** For the translation to \LaTeX we defined \LaTeX environments which provide the same formatting primitives of Box in \LaTeX [78]. The box2latex consumer translates Box operators to corresponding environments, latex is then used to do the real formatting. An additional feature is the ability to improve the final output by defining translations from Box strings to native \LaTeX code. This feature can be used for example, to introduce mathematical symbols that were not available in the original source text (e.g., to introduce the symbol \( \phi \) for the word phi). All code examples in this thesis (such as Figures 4.7–4.10) are generated with GPP using the box consumer box2latex.

**From Box to HTML** Boxes are translated by box2html to a nested sequence of HTML tables. Representing Box in HTML is difficult because HTML only contains primitives to structure text logically (as title, heading, paragraph etc.), not as composition of horizontal and vertical boxes. Only with HTML tables (where rows correspond to horizontal boxes and tables to vertical boxes) we can correctly represent Box operators in HTML. Figure 2.10 on page 32 and Figure 5.5 on page 85 contain examples of HTML pages as produced by box2html.

## 4.5 Applications

This section addresses several applications of the generic pretty-printer GPP in practice.

### 4.5.1 The Grammar Base

The Grammar Base (GB) is a collection of reusable Open Source language definitions in SDF. This collection includes grammars for XML, C, COBOL, JAVA, FORTRAN, SDL, and YACC.

In addition to language definitions, GB also contains pretty-print tables for each language. Together with generated parsers, GB thus offers a large collection of format tree producers and consumers for software reengineering systems for free.

The pretty-printer generator is used to generate initial pretty-print tables for new languages. By using this generator, pretty-print support can be guaranteed for each language without any effort. The generator is integrated in the build process of GB to automatically build a pretty-print table for a language unless a customized table exists.

We initiated the Online Grammar Base to make the grammars in GB accessible and browsable via Internet. GPP is used to produce the web pages by formatting all language definitions and representing them in HTML. The screen
dump in Figure 2.10 on page 32 gives an impression of the Online Grammar Base and the use of GPP as formatting engine.

### 4.5.2 Integration of GPP and GB in XT

GPP and GB are highly integrated in XT (see Chapter 3, “XT: a Bundle of Program Transformation Tools”), by the general pretty-print tool pp. This tool combines all pretty-print components from GPP, with all pretty-print tables from GB. The result is a tool that can pretty-print any language contained in GB (currently more than 30 languages and 10 dialects) in any format supported by GPP (currently 3). The tool can format either PTs or ASTs.

With this tool, pretty-printing reduces to selecting an input language, an input format (plain text, PT, or AST), and an output format (plain text, HTML, or \( \text{\LaTeX} \)).

### 4.5.3 A documentation generator for SDL

In the next chapter, we will discuss the development of a documentation generator for the Specification and Description Language (SDL) in corporation with Lucent Technologies. This documentation generator produces a web-site that provides different views on SDL programs and formed an application of GPP in industry.

We used ppgen to obtain an initial pretty-print table for SDL and added pretty-print rules to customize this generated formatting definition. We used pt2box to obtain a BOX format tree for SDL programs, containing labels and references for several language constructs. Depending on the view that was being constructed, a separate program connects the labels and references of relevant language constructs, for instance state transitions to corresponding state definitions. All HTML pages are produced by box2html.

We also developed utilities that produce BOX-terms for data types, such as lists. These tools made it very easy to transform such data types to HTML without the extra effort of developing an abstract syntax and pretty-print tables first. This illustrates that besides the format tree producers described thus far, also additional ones can easily be integrated with GPP.

The documentation generator also integrates other views of SDL programs, such as a graphical view on state transitions. These views can be accessed from the generated HTML representation of SDL programs and \textit{vice versa}.

### 4.5.4 Pretty-printing COBOL

GPP is currently being used in an industrial experiment concerned with COBOL reengineering. The goal of this experiment is to bring the results of academic research into practice to build automated reengineering systems. The experiment combines rewriting with layout [32] and conservative, comment preserving pretty-printing to leave the layout of transformed programs in tact.
The transformations in this project are performed on full PTs and implemented in ASF [15, 54], an algebraic specification formalism based on rewriting. Transformation rules are interpreted by an engine that preserves layout during rewriting. Layout nodes that are created during rewriting are marked as new. The result is a PT that contains original layout nodes wherever possible, and nodes marked as new, where layout nodes had to be created. Consequently, only the parts of a COBOL program that were affected by the transformation are re-formatted by GPP, the rest of a program remains unchanged.

To use GPP in this experiment, we first generated a pretty-print table from the COBOL grammar using ppgen. Then we customized the generated pretty-print rules. Since pretty-printing occurs conservatively and the transformations make only local changes, few layout nodes have to be generated by the pretty-printer. Consequently, only a few pretty-print rules had to be customized to obtain a proper formatting of the transformed program parts.

Thus, although the COBOL grammar is large (about 350 nested productions) and the corresponding generated pretty-print table is huge (about 1200 pretty-print rules), only a few pretty-print rules had to be customized by hand (3 in the experiment that we performed) in order to use GPP successfully in this COBOL reengineering system.

### 4.6 Discussion

We only support conservative pretty-printing for PTs because AST do not contain layout. However, defining a tree format based on ASTs with layout, and to implement conservative pretty-printing for it, would not be difficult.\(^3\)

We use pretty-print rule interpreters to transform PTs and ASTs into format trees. This has the advantage that the interpreters are language-independent and that pretty-printers can be customized at any time. Alternatively, language-specific format tree producers can be generated from pretty-print rules. This would increase performance at the cost of reduced flexibility.

In our approach, the rules that transform a PT or AST to a format tree are defined per language production. This transformation can also be based on pattern matching in which case the tree structure is taken into account to determine a proper formatting of program fragments [33, 106]. This gives more expressiveness because it is context sensitive. However, reuse of format definitions for formatting PTs and ASTs would be complicated due to the usually different tree structures.

---

3See for instance [148], which discusses layout preserving re-engineerings defined on abstract syntax trees with layout.
4.7 Concluding remarks

Related work We refer to [79] for a general overview of existing work in pretty-printing. An early approach to comment preserving pretty-printing was discussed in [26]. In this approach, the original program text is needed during pretty-printing in order to retrieve comments and to determine the location where to insert comments in BOX-like formatting expressions.

Van De Vanter emphasizes the importance of comments and white space for the comprehensibility and maintainability of source code [136]. Like we, he advocates the need to preserve this crucial aspect. He describes the difficulties involved in preserving this documentary structure of source code, which is largely orthogonal to the formal linguistic structure, but he does not provide practical solutions.

In [125] another conservative pretty-print approach is described which only adjusts code fragments that do not satisfy a set of language-specific format constraints. This approach is useful for improving the formatting of source code by correcting format errors. However, it cannot handle completely unformatted (generated) text and does not support comment preservation. Furthermore, it adapts existing layout (in case format errors are detected), which makes it not useful in general for software reengineering.

Pretty-printing as tree transformation is also discussed by [67, 106]. They do not address typical requirements for software reengineering such as customization and reuse of transformation rules, and layout preservation. Also transforming different types of trees and producing different output formats is not addressed.

Components and Reuse Figure 4.12 displays the gpp package, its constituent components, and the components it reuses (see Section 3.6 on page 42 for information about component diagrams). The gpp package implements 8 components and reuses 6 components from 6 different packages. The reused components are part of XT and are described in more detail in Chapter 3, “XT: a Bundle of Program Transformation Tools”.

Table 4.2 depicts component sizes and reuse levels of the gpp package. The table shows that the implementation consists of approximately 12,100 lines of code, of which more than 8,400 lines are reused. This yields a reuse level between 69% and 85%. Section 3.6 justifies these numbers and describes how they are obtained by analyzing component implementations.

Contributions In this chapter, we discussed pretty-printing in the context of software reengineering and described techniques that make pretty-printing generally applicable in this area. These techniques help to decrease development and maintenance effort of reengineering systems, tailored towards particular customer needs, and supporting multiple language dialects. The tech-
Figure 4.12 Components used by the GPP package. This picture will be filled in the remaining chapters with additional software packages. This leads to a complete picture of component usage, which is depicted in Figure 8.2 on page 145.
Component | Non-transitive reuse | Transitive reuse
| LOC | RS1 | Reuse% | LOC | Reuse%
abox2html | 1,436 | 783 | 54% | 1,436 | 54%
abox2latex | 1,782 | 895 | 50% | 1,782 | 50%
abox2text | 1,226 | 805 | 65% | 1,226 | 65%
abox-format | 755 | 676 | 89% | 755 | 89%
asfix2abox | 3,617 | 2,883 | 79% | 5,325 | 86%
ast2abox | 1,714 | 1,382 | 80% | 3,422 | 90%
ppgen | 1,415 | 984 | 69% | 1,415 | 69%
tohtml | 233 | 0 | 0% | 10,988 | 97%

Total: | 12,178 | 8,408 | 69% | 26,349 | 85%

Table 4.2  Reuse table for the gpp package. The table shows that for this package, a total of 3,770 new lines of code had to be written.

Techniques addressed in this chapter include i) customizable pretty-printing to adapt a pretty-printer to customer-specific format conventions, ii) conservative pretty-printing to preserve layout and comments during pretty-printing, iii) modular pretty-print tables to make formattings reusable, and iv) pretty-printing of PTs and ASTs using a single format definition. Furthermore, we implemented these techniques in the generic pretty-printer GPP which transforms PTs and ASTs to plain text, HTML, and \TeX, using a single format definition. Finally, we demonstrated that these techniques are feasible by using GPP for a number of non-trivial applications, including COBOL reengineering and SDL documentation generation.

Acknowledgments  We thank Arie van Deursen, Jan Heering, and Paul Klint for reading drafts of the chapter.