Pretty-Printing for Software Reengineering

Merijn de Jonge

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Merijn.de.Jonge@cwi.nl

Abstract

Automatic software reengineerings change or repair existing software systems. They are usually tailor-made for a specific customer and language dependent. Maintaining similar reengineerings for multiple customers and different language dialects might therefore soon become problematic unless advanced language technology is being used.

Generic pretty-printing is part of such technology and is the subject of this paper. We discuss specific pretty-print aspects of software reengineering such as fulfilling customer-specific format conventions, preserving existing layout, and producing multiple output formats. In addition, we describe pretty-print techniques that help to reduce maintenance effort of tailor-made reengineerings supporting multiple language dialects.

Applications, such as COBOL *reengineering and* SDL *documentation generation show that our techniques, implemented in the generic pretty-printer GPP, are feasible.*

1. Introduction

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

For instance, Dutch banks have to standardize their bank account numbers before the second quarter of 2004 [15]. To that end, a restructuring reengineering [3] 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 [14]), 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 [18].

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 paper.

Pretty-printing in the software reengineering area 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 then is 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* [5]. In this case, the reengineering process requires user intervention and documentation generation is used to make programs easy accessible. To that end, a pretty-printer, used to format programs nicely, 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 formattings independently of such output formats in *pretty-print rules* [16].

- The modifications made during software reengineering should be minimal [20]. 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.

Figure 1. Example of a parse tree (a) and an abstract Syntax tree (b). Comments in the parse tree start with '%', layout is denoted by '␣'.

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

- Adding support for a new language should be easy. We propose using generic format engines, which can interpret pretty-print rules of arbitrary languages. Moreover, we simplify creating new formattings 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 formattings between the corresponding pretty-printers.

In contrast to existing literature, which usually concentrates on a specific aspect of pretty-printing [9], this paper 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.

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 paper we consider two types of abstract representations: (full) *parse trees* (see Figure 1(a)) and *abstract syntax trees* (see Figure 1(b)). Full parse trees (also called *concrete syntax trees*) contain all lexical information, including comments and ordinary layout.

A parse tree (PT) can be pretty-printed *progressively*, which means that *all* layout will be generated. For the PT of Figure 1(a) 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. This is called *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 pretty-printing ASTs is that literals (i.e., the keywords of the program) should be reconstructed.

Pretty-printing consists of two phases [17]. 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 2(a)). 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, 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 how the construction of such language-specific mappings is simplified by generating them from corresponding grammars.

3. Pretty-printing for software reengineering

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

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 different output formats: plain text for the pretty-printer 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:

$$pp_i^o = format\ tree\ producer_i + format\ tree\ consumer_o$$

By developing different format tree consumers, a format tree can be transformed to multiple output formats. In Section 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 a language have been defined, programs in that language can be formatted in all available output formats without extra effort.

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 adaptions, 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 [20]. 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 contains (some) original layout nodes (for instance the node '␣␣␣' in Figure 1(a)). 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. Conservative pretty-printing consists of two steps. First, the 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 ε. 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:

- First, the terminal symbol occurring left to a layout node is located in the format tree (e.g., the second 'x' in Figure 2(a) in case the layout string '␣␣␣' has to be preserved between the symbols 'x' and ':=' in Figure 1(a)).

- Then, the format tree right to that terminal symbol is determined and removed (the node ':=' in Figure 2(a)).

- The terminal symbol, the layout node, and the format tree are then inserted into a new sub-tree which has the empty format operator ε as root.

- This tree is inserted at the location of the terminal symbol in the original format tree.

Applying these steps to the format tree of Figure 2(a) yields the tree as depicted in Figure 2(b).

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:

$$f(\varepsilon(t_1,\ldots,t_n)) \quad = \quad f(t_1) \cdot \ldots \cdot f(t_n)$$
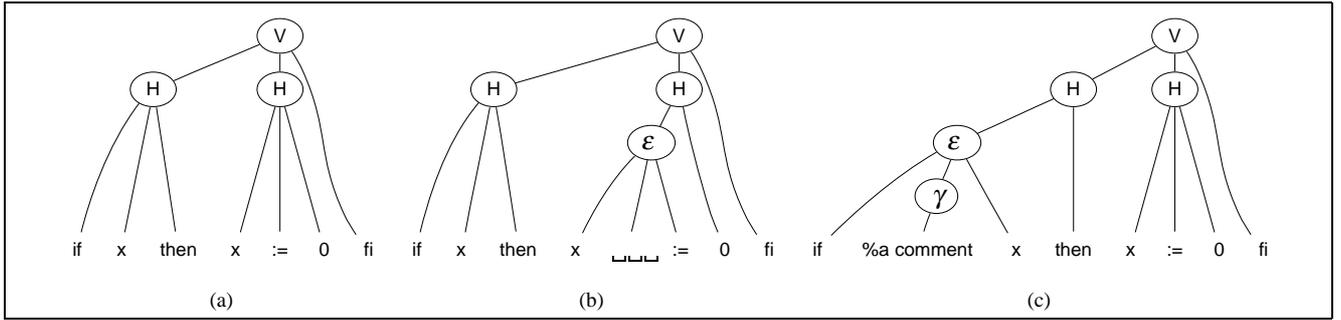
Figure 2. (a) Format tree, (b) format tree in which existing layout (␣␣␣) is preserved, (c) format tree with preserved comments. The nodes in the trees correspond to format operators.

$$
\begin{aligned}
f(\phi(t_1,\ldots,t_n)) &= f(t_1)\cdot l_\phi\cdot f(t_2)\cdot\ldots\cdot l_\phi\cdot f(t_n) \\
&\quad\text{for each format operator } \phi \\
f(s) &= s \text{ for each non-terminal symbol } s \\
f(w) &= w \text{ for each layout string } w
\end{aligned}
$$

The operator '·' denotes string concatenation, l_ϕ denotes a layout string generated by the operator ϕ. This definition states that for ε only sub-trees are concatenated, while for other operators f also puts layout between sub-trees.

For example, when we define $l_H = \text{␣}$ and $l_V = \backslash n$, then we can translate the format tree of Figure 2(b) to text using the following derivation:

$$
\begin{aligned}
f(V(H(\text{if},\text{x},\text{then}),H(\varepsilon(\text{x},\text{␣␣␣},:=),0),\text{fi})) &= \\
f(H(\text{if},\text{x},\text{then}))\cdot l_V\cdot f(H(\varepsilon(\text{x},\text{␣␣␣},:=),0))\cdot l_V\cdot\text{fi} &= \\
\text{if}\cdot l_H\cdot\text{x}\cdot l_H\cdot\text{then}\cdot l_V\cdot f(\varepsilon(\text{x},\text{␣␣␣},:=))\cdot l_H\cdot 0\cdot l_V\cdot\text{fi} &= \\
\text{if}\cdot l_H\cdot\text{x}\cdot l_H\cdot\text{then}\cdot l_V\cdot\text{x}\cdot\text{␣␣␣}\cdot:=\cdot l_H\cdot 0\cdot l_V\cdot\text{fi} &= \\
\text{if␣x␣then}\backslash n\text{x␣␣␣}:=\text{␣}0\backslash n\text{fi}
\end{aligned}
$$

Since the ε 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 (customer-specific) pretty-print rules.

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 ε 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 γ to mark comments in format trees. This operator serves documentation generation and *literate programming* [13]. 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 2(a) by inserting %a comment is depicted in Figure 2(c).

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 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* [16]. Pretty-print rules are mappings from language constructs to format constructs. Each mapping defines how a language construct should be formatted. Pretty-print rules are defined declaratively and interpreted by a format 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 format engine.

By using this interpreted approach, the same format engine can be reused 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.

3.5. Modularity

Software reengineering requires pretty-print techniques that make dealing with language dialects, embedded languages, and customer-specific formattings easy.

To facilitate this, development and maintenance time should be decreased by allowing new pretty-printers to be constructed from existing ones. In Section 3.4, we already pointed out how this can be achieved by separating pretty-print rules and pretty-print 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 further be 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 ρ obtained by combining two tables t_1 and t_2 (where t_1 has highest priority), is defined as follows:

$$\rho = t_1 \cup (t_2 - t_1)$$

Construction of formattings with modular pretty-print tables works as follows:

Language dialects Pretty-print rules for new or affected language constructs are defined in a new set of pretty-print tables T_{new}. The complete formatting 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 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 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. This makes extensive reuse of pretty-print rules possible. For instance, to define a customer-specific formatting for an embedded language dialect.
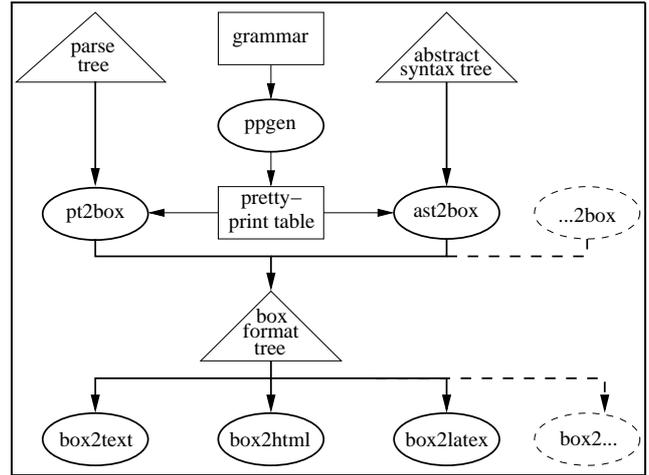
Figure 3. Architecture of the generic pretty-printer GPP. Ellipses denote GPP components, boxes and triangles denote data.

4. GPP: a generic pretty-printer

GPP is a generic pretty-printer that implements the ideas and techniques discussed in Section 2 and 3. GPP's architecture is shown in Figure 3. It consists of the pretty-print table generator `ppgen`, two format tree producers `pt2box` and `ast2box` (see Section 4.2), and the format tree consumers `box2text`, `box2html`, and `box2latex` (see Section 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 5).

4.1. Format definition

The first phase of pretty-printing consists of transforming an 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 [7, 21] to define language productions (see Figure 4 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 definition 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 can be extended with an optional else-branch as depicted in Figure 5.

```
context-free syntax
  "if" EXP "then" STAT "fi"  → STAT  {cons("IF")}
  ID ":=" EXP                → STAT  {cons("ASSIGN")}
  ID                         → EXP   {cons("ID")}
  NAT-CON                    → EXP   {cons("NAT")}
  STR-CON                    → EXP   {cons("STR")}
```

Figure 4. Grammar productions in SDF corresponding to the PT and AST of Figure 1.[1]

```
context-free syntax
  "if" EXP "then" STAT ( "else" STAT )? "fi" →
        STAT {cons("IF")}
```

Figure 5. Extending the if-construct of Figure 4 with an optional else-branch using the SDF syntax operators '()' and '?'.

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 and 5). These define the node names of abstract syntax trees [12].

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 [2, 9] to express formattings. BOX is a markup language which allows formattings to be expressed as compositions of boxes. Box composition is accomplished using *box 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:

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

$$\mathsf{V}\;[\;\boxed{B_1}\;\boxed{B_2}\;\boxed{B_3}\;] \;=\; \begin{array}{c}\boxed{B_1}\\\boxed{B_2}\\\boxed{B_3}\end{array}$$

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

$$\mathsf{ALT}\;[\;\boxed{B_1}\;\boxed{B_2}\;] \;=\; \boxed{B_1}\quad or\quad \boxed{B_2}$$

---

[1]Productions 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.

```
IF       -- V[H["if" _1 "then"] _2 "fi"],
ASSIGN   -- H[_1 ":=" _2],
ID       -- _1,
NAT      -- _1,
STR      -- _1
```

Figure 6. A pretty-print table for the grammar of Figure 4.

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:

$$\mathsf{H_{hs=2}}\;[\;\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\;] \;=\; \boxed{B_1}\;\sqcup\!\sqcup\;\boxed{B_2}\;\sqcup\!\sqcup\;\boxed{B_3}$$

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 ε, which is needed for conservative pretty-printing (see Section 3.2). However, this operator can be mimicked using the H operator as $\mathsf{H_{hs=0}}\;[\;\ldots\;]$. The comment operator γ used for comment preservation (see Section 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 6).

BOX expressions in pretty-print tables contain numbered place holders (_1 and _2 in Figure 6) which correspond to non-terminal symbols in SDF productions. During pretty-printing they are replaced by the generated BOX-expressions for these non-terminal symbols.

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 4.2) and (as node names) in ASTs (see Figure 1(b)), they can be used to format both tree types.

The pretty-print table of Figure 6 only contains pretty-print entries for flat (non-nested) SDF productions. To enable format definition for arbitrary nested SDF productions, 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 7 contains pretty-print entries for the nested SDF production of Figure 5. The first rule has

```
IF              -- V[H[“if” _1 “then”] _2 _3 “fi”],
IF.3:opt        -- _1,
IF.3:opt.1:seq -- “else” _1
```

Figure 7. Pretty-print table for the nested if-construct of Figure 5.

Figure 8. Graphical structure of the nested SDF **production of Figure 5. Grey ellipses denote nested** SDF **symbols.**

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 8). The remaining pretty-print rules correspond to the nested symbols. For each path from the root node in Figure 8 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 4.2.

Pretty-print rule generation Writing pretty-print tables can, even for small languages, be a time consuming and error-prone process. To simplify pretty-print table construction, we implemented a pretty-print table generator (the component ppgen in Figure 3). 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 is generated. That is, a pretty-print rule is generated without positional BOX-operators (such as the last two pretty-print rules in Figure 7). 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 thus obtained is minimal, it is directly usable. To improve the layout, we can benefit from the customizability of pretty-print tables and incrementally redefine the formatting. 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.

4.2. Format tree producers

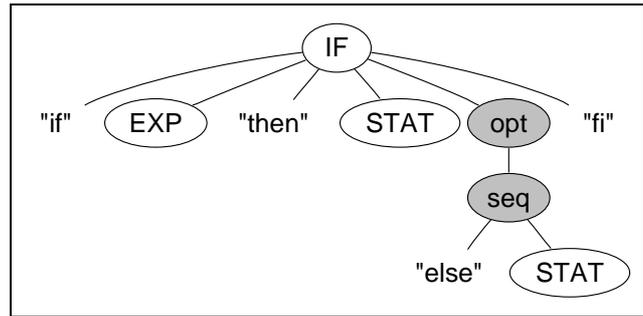This section discusses pt2box and ast2box which generate format trees from PTs and ASTs, respectively.

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 [21]. 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.

This format tree producer 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 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 ppgen 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. We refer to [12] for a discussion of 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 1(b) and 6).

In an AST, certain types of symbols are indistinguishable although they require to be formatted differently. 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 form 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.

4.3. 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, LaTeX, and HTML as output format, respectively.

From BOX to text The 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 BOX-term only contains non-conditional operators and absolute, rather then relative offsets as space options. Then, during the second phase, the normalized 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 corresponding to BOX-operators [8]. The box2latex consumer translates BOX-operators to corresponding environments, LaTeX is then used to do the real formatting. As an example, Figures 4–7 are all generated by 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.

5. Applications

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

5.1. The Grammar Base

The Grammar Base (GB) is a collection of reusable Open Source language definitions in SDF.[2] 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.[3] GPP is used to produce the web pages by formatting all language definitions and representing them in HTML.

5.2. Integration of GPP and GB in XT

XT [11] is a bundle of program transformation tools, initiated to simplify component-based development of program transformations.[4]

GPP and GB are highly integrated in XT 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 then 30 languages and 10 dialects) in any format supported by GPP (currently 3). The tool can format either PTs or ASTs.

---

[2] See http://www.cwi.nl/~mdejonge/package-base/
[3] See http://www.cwi.nl/~mdejonge/grammar-base/
[4] See http://www.program-transformation.org/xt/

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 LaTeX).

5.3. A documentation generator for SDL

The development of a documentation generator for the Specification and Description Language (SDL) in corporation with Lucent Technologies was an application of GPP in industry [10]. The documentation generator produces a web-site that provides different views on SDL programs.

We used ppgen to obtain an initial pretty-print table for SDL and added pretty-print rules to customize this generated formatting.

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 vice versa.

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* [1] 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 [4], 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 SDF COBOL grammar using pp-gen. 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.

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.

We use pretty-print rule interpreters for transforming PTs and ASTs to format trees. This has the advantage that the interpreters are language independent and that a pretty-printer 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 [2, 16]. 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.

7. Concluding remarks

Related work   We refer to [9] for a general overview of existing work in pretty-printing.

Van De Vanter emphasizes the importance of comments and white space for the comprehensibility and maintainability of source code [20]. 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 [19] 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 formating 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 using transformation rules is also discussed by [6, 16]. 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.

Contributions We discussed pretty-printing in the context of software reengineering and described techniques that make pretty-printing general 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 techniques addressed in this paper 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 LaTeX, 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.

Availability GPP is Free Software and can be obtained from `http://www.cwi.nl/~mdejonge/package-base/`.

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

References

[1] M. G. J. van den Brand and J. J. Vinju. Rewriting with layout. In *First Int. Workshop on Rule-Based Programming (RULE'2000)*, 2000.

[2] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1 – 41, 1996.

[3] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[4] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[5] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.

[6] M. A. Harrison and V. Maverick. Presentation by tree transformation. In *Proceeedings of 42nd IEEE International Computer Conference*. IEEE, 1997.

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

[8] M. de Jonge. boxenv.sty: A LaTeX style file for formatting BOX expressions. Technical Report SEN-R9911, CWI, 1999.

[9] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[10] M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings of the International Conference on Software Maintenance (ICSM 2001)*, pages 240 – 249. IEEE, 2001.

[11] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. van den Brand and D. Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[12] M. de Jonge and J. Visser. Grammars as contracts. In G. Butler and S. Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *LNCS*, Erfurt, Germany, 2001. Springer.

[13] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, 1992.

[14] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, Nov./Dec. 2001.

[15] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.

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

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

[18] H. Ossher, W. Harrison, and P. Tarr. Software engineering tools and environments. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 261–278. ACM Press, 2000.

[19] M. Ruckert. Conservative Pretty-Printing. *SIGPLAN Notices*, 23(2):39–44, 1996.

[20] M. Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*. IEEE, 2001.

[21] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.