An Architecture for Context-sensitive Formatting*
Extended Abstract

M.G.J. van den Brand, A.T. Kooiker, J.J. Vinju
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Mark.van.den.Brand, Taeke.Kooiker, Jurgen.Vinju}@cwi.nl

N.P. Veerman
Department of Computer Science, Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
nveerman@cs.vu.nl

1 Introduction

The layout and style of program source code are crucial for the time required to understand and thus modify it [5]. In fact, these aspects are too integral to the coding aspect to be changed effectively later [4, p399]. However, a system may be initiated with a proper coding standard but years of evolution causes the code to deviate from it. For instance, some programmers format code to a preferred style by hand before making changes. The result is a mixture of different programming styles, which increases the time for maintenance.

A solution to these problems is to format source code with automatic tools. Automatic formatting improves the productivity of maintenance programmers, but pretty-printers hardly implement the company specific conventions. There exist some language-specific formatting tools that can be used off-the-shelf, but these tools suffer from a profusion of command line arguments required to deal with the highly variable formatting conventions that users of the tool will have. When a language-specific formatting tool is not available for a certain legacy or domain-specific language, generic formatting (e.g. tools [1, 6, 9]) can be applied. Such tools allow a formatting convention for any language to be defined and can then generate the formatting tool from this definition. They can even provide a more flexible alternative to existing language-specific formatters. However, one of the limitations of the generic formatters mentioned above is that these can not use context information, which can be required when formatting according to industrial coding standards. The goal of this research is to rapidly obtain a formatter for any language that implements arbitrary formatting requirements.

Contributions We present an architecture for context-sensitive formatting, and demonstrate it in a formatting case with an industrial Cobol system. We claim that our architecture can handle all kinds of unexpected formatting conventions in any programming language. This architecture needs no default formatter to be generated and allows arbitrary computational power while mapping language constructs to formatting instructions. More detailed information, as well as a discussion on related work, can be found in [7].

An industrial case with context-sensitive formatting requirements In the context of a software renovation research project, we were asked by a company to enforce a corporate formatting standard to the source code of a Cobol system covering nearly 80 thousand lines of code.

We obtained the company specific conventions for formatting a number of Cobol language constructs. These are summarized in Table 1. For several constructs, the assigned starting columns and indentation depths are displayed. The requested standard is illustrated by a code example in Figure 1, containing column numbers, data declarations and a number of statements. The code example shows a record structure for storing a date. Level `03` sub-records are used to store the day, month and year, and a level `88` condition entry (a flag variable) is declared to check if a date exists. Then in the `PROCEDURE DIVISION`, if no date is present in the record, the date is retrieved from the system clock. In addition to this, the date is stored in another record.

Many formatting conventions are geared to clarify the

*The research was supported by the Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018 *CALCE: Computer-aided Life Cycle Enabling of Software Assets*.

Table 1. Required layout standard.

Description	Column
Start of divisions, sections, declarations, paragraphs	01
Start of PIC and REDEFINES clauses	41
Start of VALUE, COMP and OCCURS clauses	51
Start of statements	09
Second part of statements (e.g. MOVE Id TO ...)	25
Third part of Statements (e.g. MOVE Id TO ...)	49
Fourth part of Statements (e.g. MOVE ... TO Id)	53
Description	**Indent**
A nested data declaration (record) should indent with respect to its associated group variable	4
A level 88 data declaration should indent with respect to its preceding variable	4
In a data declaration, a level number and a variable name are separated	2
Indentation of nested statements	4

logical structure of the source code. For example, the statements inside an `IF` are indented to clearly indicate that their execution is subject to the conditional. However, this particular layout standard also contains some more subjective rules. Figure 1 shows how the `FROM` part of the `ACCEPT` statement is aligned with the `TO` parts of `MOVE` statements that are *outside* the conditional. This is an example of alignment that crosscuts the logical structure of the program. When using a logical structure, the `FROM` part of the `ACCEPT` statement would be indented relative to the beginning of the entire statement. Instead, the standard dictates that we must indent the first `ACCEPT` part according to the context, and put the `FROM` part at an absolute column, regardless of the current indentation level. This requires context-sensitive information.

Another construct which requires context information are the data declarations. The general scheme is that declarations with a higher number should be indented more. However, the level `88` condition entry field is an exception, since it always appears directly under its *preceding* variable, whereas a regular level number (e.g. `01`, `03`, ..) is indented with respect to its associated *group* variable. It is a formatting exception that assigns a particular meaning to the number `88`, requiring context information.

So both the statements and the data declarations in this layout standard require context-sensitive formatting.

2 Context-sensitive formatting architecture

Our industrial case illustrates that formatting is a process which heavily depends on specific user requirements, requiring context information in some cases. The application of formatting source codes is bound by strict, but possibly irregular rules given by the owner of the code base. Both the used language, as well as the corporate conventions may be unique.

Existing generic pretty-printing, apart from [9], do not cope well with unexpected formatting conventions that require more elaborate analysis of the source code. Extra information such as nesting depth, specific identifiers, or relative positions between several constructs in a language is often important; this is context-sensitive information.

```
Column   10        20        30        40        50        60        70
1234567890123456789012345678901234567890123456789012345678901234567890

DATA DIVISION.
WORKING-STORAGE SECTION.

01  WD_DATE                                        VALUE ZERO.
    88  WD_NO_DATE                                 VALUE ZERO.
    03  WD_DD                              PIC 9(02).
    03  WD_MM                              PIC 9(02).
    03  WD_JJ                              PIC 9(02).

PROCEDURE DIVISION.
INITIALIZE_DATE SECTION.
INIT_00.
        IF              WD_NO_DATE
            ACCEPT      WD_DATE                 FROM DATE
        END-IF
        MOVE            WD_DD                   TO  WR_DD
        MOVE            WD_MM                   TO  WR_MM
        MOVE            WD_JJ                   TO  WR_JJ.
```

Figure 1. Layout according to Table 1.

When language constructs are mapped during pretty-printing, we should allow more elaborate user-defined computation. Nevertheless, *default* pretty-printing is very practical because it automates the major part of creating a formatting tool. Therefore, the formatting process in our architecture is split into three stages, see Figure 2.

Stage 1: user-defined mapping The input for *Stage 1* is a parse tree. The user-defined mapping that is applied to this parse tree consists of transforming particular language constructs to Box [9] constructs. Box is a special purpose language for formatting. The language constructs that are transformed, depend on the formatting requirements that are not handled as desired by the default mapping in *Stage 2*. Any programming language or tool can be used to map certain selected language constructs to Box constructs. For our Cobol case we used ASF+SDF [10], because its application domain is in these kind of language transformations.

Applying a user-defined mapping on the input parse tree results in a *hybrid* parse tree containing both source language constructs and Box constructs. Figure 3 illustrates such an hybrid tree, where Box language operators can have programming language constructs as children, and vice versa. The borders between the source code language formalism and Box formalism are guarded by encapsulating nodes which are marked by two special node attributes: `from-box` and `to-box`. The outermost pyramid shows a Cobol parse tree that is partially formatted. It has one child that has been transformed to Box constructs. The transition from Cobol to Box is guarded by a `from-box` node. Although this part of the program is not formatted completely, it does contain an unformatted Cobol part again. The transition from Box back to Cobol is guarded by a `to-box` node. This guarantees the type safe merging of host and Box language. Furthermore, *Stage 1* contains a tool that correlates the hybrid tree with the original parse tree to ensure syntax safety.

Figure 3. The hybrid tree

Figure 2. The multistage formatting architecture

Stage 2: default mapping In *Stage 2* the default formatting engine applies default pretty-print rules to the hybrid parse tree result of *Stage 1*. It is constructed in such a way that it guarantees *syntax safety*. All Cobol constructs that are left in the hybrid tree are then mapped to Box operators. The algorithm used by the formatting engine skips over all Box expressions that are between a `from-box` and a `to-box` node, since they have been formatted already (see Figure 3). The resulting tree contains only Box expressions. It is guaranteed that all source language constructs have been transformed into Box constructs.

It is surprising that in general more nodes are formatted in *Stage 2* than in *Stage 1*. Programming languages share typical syntactic idioms that can be formatted in a similar way. The most obvious example is the block structure: a syntax rule that begins and ends with a literal, and has a list of other constructs in the middle. There is an easy opportunity for reuse. *Stage 2* benefits from these similarities by using some smart heuristics. It extracts information from parse trees to identify syntactical idioms, and maps them to Box expressions. We reuse the default mapping that was proposed in [9], but now we implement it on the hybrid tree instead of generating a default implementation that the user needs to adapt. The benefit is twofold: the user can choose the technology he prefers to use for the user-defined formatter and we avoid common maintenance problems with generated code altogether.

Stage 3: Box back-ends For *Stage 3* several reusable Box back-ends are available [1, 9] that can be reused to output formatted programs. The Box tree from *Stage 2* contains different formatting operators. The H and V operators simply output text horizontally or vertically. However, if the text in a HV or HOV box does not fit in the horizontal space, then it is split horizontally between two or more lines or printed vertically. Other operators work in a similar way; if the text contained in the operand does not fit the sizes of the surrounding box, the operator will format the text accordingly.

3 Case study

In this project, we were driven by an industrial case: the layout of a DEC Cobol system of 78 thousand lines of code must be standardized according to specific layout standard. This standard is presented in the Introduction (Table 1). We describe what kind of effort was needed to create a formatter that meets the requirements using the above described formatting architecture. To illustrate the approach, we give a user-defined rule for a specific language construct that was formatted.

Implementation of the formatting architecture The parser and the user-defined formatting were implemented using the language specification formalism ASF+SDF [10]. This is a formal language that is well-equipped for transformations of source code. Using SDF grammar productions, we defined the syntax of DEC Cobol. The SDF grammar was derived from the online IBM VS Cobol II grammar [2, 3] and adapted to be able to parse DEC Cobol-specific constructs. The generated parser outputs a parse tree that is used as input for *Stage 1* (see Figure 2).

In case the default mapping was different from the layout standard, ASF rewrite rules were defined to implement the mapping of Cobol constructs to Box language constructs. These rewrite rules are applied to the parse tree in *Stage 1*. A rule may have complex matching patterns, and by defining parameterized functions, rules can also receive context information to guide a transformation. The rules use the concrete syntax of the manipulated language on both sides and in the conditions. The mapping of Cobol constructs to Box constructs is therefore immediately recognizable as such.

For efficiency reasons, *Stage 2* and *3* are linked together in a new tool called Pandora. Pandora is distributed with the ASF+SDF Meta-Environment [8] and can be used with and without a user-defined mapping. In our industrial case we need most operators of the Box language, and one important extension: tab stops. Tab stops support placing of constructs at fixed columns independent of the current indentation level. The crosscutting concern where parts of a language construct have to be placed at fixed columns has inspired this extension.

Implementation of user-defined rules The implementation of a formatter involves specifying at least one rewrite rule for each construct that the standardization document describes, unless the default mapping (*Stage 2*) coincides with the standard. We present a Cobol construct with its implementation: the `MOVE` statement. The user-defined rule for this construct is illustrated in Figure 4.

The `MOVE` statement, with keywords `MOVE` and `TO`, is formatted by a single rewrite rule (equation). The context-free syntax is defined by a production rule, and meta variables are defined for the sorts `IdOrLit` (identifier or literal) and `Id-list` (list of identifiers) for use in the

```
context-free syntax
 "MOVE" IdOrLit "TO" Id-list -> Move-stat
context-free syntax
 from-box( Box )      -> Move-stat {from-box}
 to-box  ( IdOrLit ) -> Box       {to-box}
 to-box  ( Id-list ) -> Box       {to-box}
variables
 "IdOrLit" -> IdOrLit
 "Id-list" -> Id-list
equations
[move-statement]
 MOVE IdOrLit TO Id-list
 =
 from-box(
 H [ "MOVE"
     H ts=25 [to-box(IdOrLit)]
     H ts=49 ["TO"]
     H ts=53 [to-box(Id-list)]
   ])
```

Figure 4. User-defined mapping for MOVE.

rewrite rule. The left-hand side of the rewrite rule tagged `[move-statement]` matches all instances of this construct. On the right-hand side, we replace the construct by a Box expression. The `from-box` and `to-box` constructors must be defined for the involved sorts, and mark the borders between the Cobol and Box formalisms, similar to the hybrid tree in Figure 2. All parts of the construct are formatted horizontally using a H box, and three individual members are placed on fixed positions using tabstops (e.g. `ts=25`). This is in accordance with the layout standard from Table 1. The result of applying this user-defined rule to a `MOVE` statement can be seen in Figure 1.

Results We were able to develop about 50 user-defined rules for constructs according to the layout standard. We measured the performance of application of our implementation to 78 thousand lines of Cobol code. Parsing 78 KLOC was done in 420 seconds. *Stage 1*, using compiled rewrite rules, took only 22 seconds, while Pandora took 74 seconds to perform *Stage 2* and *3*, of which 32 seconds are spent by the *default* mapping. The above measurements show that formatting 78 thousand lines of code using this architecture is feasible.

4 Conclusions

We have taken a fixed set of formatting requirements for a Cobol system as spelled out in a standardization document, and applied generic formatting technology to implement them. It appeared that corporate conventions can dictate alignment that crosscuts the logical structure of a program, and can even dictate indentation that is dynamically computed from context information.

We have developed and implemented a formatting architecture that allows arbitrary computational power for mapping language constructs to the Box language. The enabling feature is a hybrid format that merges Box expressions with parse trees. Much of the boilerplate part of formatting can still be automated by a default mapping to Box. Absolute tab stops, an important feature which is not found in many Box back-ends, is used extensively in our case study.

References

[1] M. de Jonge. Pretty-printing for software reengineering. In *Proceedings of ICSM 2002*, pages 550–559. IEEE Computer Society Press, Oct. 2002.

[2] R. Lämmel and C. Verhoef. VS Cobol II Grammar, 1999. http://www.cs.vu.nl/grammars/vs-cobol-ii/.

[3] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[4] S. McConnel. *Code Complete*. Microsoft Press, 1993.

[5] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *ACM*, 26(11):861–867, 1983.

[6] D. C. Oppen. Prettyprinting. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.

[7] M. G. J. van den Brand, A. T. Kooiker, N. P. Veerman, and J. J. Vinju. An industrial application of context-sensitive formatting. Technical Report SEN-R0510, Centrum voor Wiskunde en Informatica, June 2005.

[8] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC '01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[9] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, 1996.

[10] 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.