JWEB — Literate Programming for Java

Boštjan Slivnik

Faculty of Computer and Information Science
University of Ljubljana, Slovenia
Email: bostjan.slivnik@fri.uni-lj.si

Abstract

JWEB, a system for literate programming in Java programming language, is described. It has been made by replacing code for processing C and C++ (and to some extend Java as well) in CWEB with code tailored specifically for processing Java. Thus, CWEB's tools cweave and ctangle were replaced by tools jweave and jtangle in JWEB, but a large amount of code remains the same. This paper is a report on the main design decisions that had to be taken during JWEB development. These include, among others, keeping the C preprocessor a part of JWEB, resolving certain typographical details like typesetting class names and section headers. Furthermore, the construction of context-sensitive grammar for parsing chunks of Java code is discussed.

1 Forewarning

Literate programming has been invented by D. E. Knuth somewhere before 1981 [1, 2, 3]. It is therefore remarkably older than many bizarre phenomena computer science and software engineering have been littered with since those early ages... and yes, the Python programming language (1991) is precisely such peculiar rubbish an enlightened computer programmer reading these lines should first think of. It is also older than the latest reincarnation of the UEFA Champions League (1992) and it appeared even before Sir Alex Ferguson took over ManUtd (1986)... ohh, all those glorious years. However, it is perhaps worth mentioning that literate programming is nevertheless younger than smileys as they were first introduced by English poet Robert Herrick in his poem *To Fortune* in 1648. In a few words, the idea of literate programming emerged approximately at the same time as a small subalpine tribe started claiming its thousand years dream of becoming a sovereign nation. Nowadays, many would argue that the idea of literate programming is as widespread as is the belief that the particular tribe has demonstrated its aptitude for ruling itself in any respectful way. You have been warned...

2 On Literate Programming

Knuth envisaged literate programming as a paradigm stimulating the programmer to concentrate on the explanation of a program, how it is constructed and also why it is constructed as it is and not perhaps otherwise. The resulting artefact is thus not merely a program but an essay containing both the explanation of the program's logic as well as the program's code. These two ingredients should be interwoven in the same document so that the text follows the programmer's line of thought and not the program's structure as imposed by the programming language used.

Technically, the programmer writes a programming essay in a single file containing the explanation written in TEX and code written in a selected programming language. The original literate programming tool WEB [1] combining TEX and Pascal consists of two tools, namely weave and tangle. If the entire essay is stored in file `foo.w`, then

- weave is used to produce the file `foo.tex` containing the printable copy of the essay intended to be read by humans... well, programmers;
- tangle is used to produce the file `foo.pas` containing the program's Pascal code intended to be compiled using the Pascal compiler.

The two by far the most important pieces of software ever written using WEB (or literate programming in general) are Knuth's TEX [4] and METAFONT [5].

WEB lead to the creation of a number of similar tools. It was rewritten for other programming languages, e.g., CWEB [6] for C / C++ or FWEB [7] for C / C++ / Fortran 77 / Fortran 90 / Ratfor, but language agnostic tools are also available. For instance, nuweb combines LATEX and any programming language of programmer's choice [8]. As no programming language is built into nuweb, it cannot perform any prettyprinting and does not produce the index of identifiers. However, it can also generate an HTML version of the documentation. Likewise, noweb, a slight simplification of Knuth's literate programming, provides no prettyprinting [9]. On the other hand, SpiderWeb is a tool for replacing Pascal in the original WEB system with another programming language [10]. At present, no specialised literate programming tool for Java able to typeset fragments of Java source code is known to the author.

Finally, literate programming must not be confused with other approaches for creating source code and API documentation [11, 12, 13], e.g., JavaDoc, Jadeite or Doxygen. These tools mostly produce (API) documentation

1. Some code is explained below, in Section 2.

```
public class Main {
  public static void main(String[]args)
  {
    ⟨ print banner 2 ⟩
  }
}
```

2. This is inserted into the code of Section 1.

⟨ print banner 2 ⟩ ≡

```
System.out.printf("Hello world.");
```

This code is used in section 1.

Figure 1: A simple Java program typeset by CWEB illustrating how the program written in CWEB might be split into sections.

focusing of *what* each part of the source code does rather than *how* and *why* it does what it does.

3 CWEB in Particular

The logical first step in writing literate Java programs is to use CWEB [6] because

> *CWEB is a version of WEB for documenting C, C++, and Java programs.*
>
> — Knuth's home page.

Thus, a simple "Hello world" program might be written and typeset as shown in Figure 1. The example illustrates how a program written in CWEB is split into sections where each section documents a small part of code. cweave retains the ordering of sections while ctangle produces the valid Java source by inserting the code of "named" sections into the code of other sections. The sections in Figure 1 might have been written in different order if the programmer decided otherwise. cweave would respect the ordering, ctangle would produce the same output.

But after a rather quick examination, one finds that CWEB does not work as it probably should for Java programs. For instance, a simple Java program

```
class Main { Main() {} }
```

is typeset by cweave and TEX correctly as shown in Figure 2 while a similar program where it is explicitly specified that class *Main* extends class *Object*, is typeset incorrectly as shown in Figure 3. In terms of literate programming and prettyprinting, the difference is not only that symbol **extends** is not recognised as a keyword but also that symbol *Main* is no longer recognised as a class name. This becomes important later when variables of class *Main* are declared and index is generated.

Note that in this example CWEB does not misunderstands Java because the example includes some syntactic structure that has been introduced into Java after CWEB has been written. After all, **extends** clause have been a part of Java from version 1.0. Many other examples where CWEB misunderstands Java and thus typesets the code incorrectly can be constructed.

Hence, a new tool is needed for Java and to create JWEB, a literate programming tool for Java, a redesign of

1. Derived by default.

```
class Main {
  Main()
  {}
}
```

Figure 2: A program typeset correctly.

1. Derived explicitly.

```
class Main extends Object
{
  Main()
  {}
}
```

Figure 3: A program typeset incorrectly.

CWEB has been chosen instead of implementation from scratch. In the first place because for typesetting C and C++ CWEB is an excellent piece of software that can mostly be reused as demonstrated by SpiderWeb. Furthermore, C and C++ on the one hand and Java on the other are syntactically close enough to Java and thus the modification of the existing software should most likely be rather limited. And finally, all tricks of TEX built into WEB and CWEB by Knuth can be reused easily.

4 JWEB Desing Issues

As described above, JWEB has been made by reusing the code of CWEB. Therefore, JWEB incorporates the same idea and the same overall design as the original WEB and CWEB, i.e., the source file with extension `.w` is processed by jweave and jtangle. Apart from the lexical and syntactic modifications that are necessary for the latest version of Java and which are discussed in the next section, a few general decisions had to be taken.

The C preprocessor. As plain Java does not assume a preprocessor, a question whether to keep the C preprocessor support built into CWEB or not raises up. Both answers, positive and negative, could be argued for. However, it has been decided that the preprocessor support remains. First, literate sources must be processed by one preprocessor, i.e., jtangle, before actual compilation with javac and thus entire compilation process will most likely be automated using make-like tool anyway. Hence, it does not take much to use yet another preprocessor. Second, the original WEB supports simple macros even though no preprocessor support was envisaged for Pascal (but Knuth demonstrated in TEX and METAFONT that macros can be useful in Pascal as well). And finally, if you don't like it, just don't use it.

Typesetting class names. In CWEB a class name must be known to cweave before it is used so that it can be typeset as a type name, i.e., in roman bold typeface, and that template parameters (if present) can be typeset correctly (< and > get replaced by ⟨ and ⟩, respectivelly). If a name is used in a section that preceedes the section where the name is declared, the programmer must use a format-

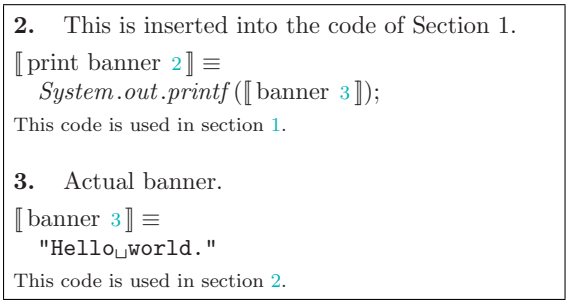

Figure 4: Section headers typeset using ⟦ and ⟧.

ting declaration (`@f` or `@s`) telling cweave explicitly how to typeset it.

With its enormous API Java provides a lot of class names by default and even more once they are imported. To avoid long lists of formatting declarations (placed far away from actual definitions) and in some cases their subsequent cancelations, JWEB typesets all identifiers, including class names, in the same manner.

Typesetting section headers. Java (like C++) uses templates and thus symbols < and > are used not only as (a part of) relation operators but in class names as well. In the latter case, they are typeset as ⟨ and ⟩, but the same symbols are used by WEB and CWEB to enclose the section headers as shown in Figure 1. To avoid confusion, JWEB uses symbols ⟦ and ⟧ instead of ⟨ and ⟩ in section headers. So a section header is typeset as shown in Figure 4.

5 JAVA Grammar Used by JWEB

Apart from small and simple lexical modifications, the major issue in turning CWEB into JWEB is to modify the parser built into cweave. It should no longer support C/C++ but specifically Java.

As each section contains just a chunk of a source code taken completely out of the context (as some chunks might be included for explanation only and are not a part of the final program at all), Knuth decided to use a very simple bottom-up parser for a context-sensitive grammar of Pascal in WEB and C/C++ in CWEB. In short, the parser starts with a sequence of symbols the chunk of code consists of and applies the context-sensitive productions one after another, left to right, until no production can be applied any more. Each time a production is applied, TEX macros for formatting (indentation, change of typeface, some extra whitespace, ...) are inserted.

Although Knuth and Levy wrote cweave and ctangle in CWEB and provide a good explanation about how these two programs work, no explanation is given *why* context-sensitive productions build into cweave are as they are. Nevertheless, the following basic principles, nowhere explicitly listed in [1, 6], can be provided:

- The grammar can describe the superset of Java allowing some syntactic structures not found in Java if this makes the grammar smaller or better suited for typesetting (javac, not cweave, must check that Java code is fullly standard compliant).
- The grammar does not need to identify "classic" syntactic structures which syntax directed translation is based upon as typesetting is a much simpler process.
- Each production should describe a reduction which if it can it will be performed. However, the productions are numbered and a production with lower index takes precedence over a production with a higher index.

Finally, it must be emphasised that the grammar underlying the typesetting affects it strongly: some grammar will reduce the chunk of code into a single symbol and thus recognise a chunk of code as a single entity while some other grammar will reduce the same code ending up with several symbols. However, we must accept that no grammar works in all cases.

Due to its size the entire Java grammar that is built into JWEB cannot be presented here, let alone augmented with comments. Therefore only two pieces of it are given, with a short explanation, to give a reader a brief insight.

Annotations and annotation types. As the first example let us consider handling of annotations and annotation types as they both start with `@`. Reductions (written as $\beta \Rightarrow \alpha$ instead of as a production $\alpha \longrightarrow \beta$ where $|\alpha| \leq |\beta|$) handling these syntactic structures are

$$at\_sign\ class\_like \Rightarrow class\_like \quad (1)$$

$$at\_sign\ exp\ dot\ exp \Rightarrow at\_sign\ exp \quad (2)$$

$$at\_sign\ exp\ lpar \Rightarrow ann\_head\ lpar \quad (3)$$

$$at\_sign\ exp \Rightarrow annotation \quad (4)$$

$$ann\_head\ parexp \Rightarrow annotation \quad (5)$$

As token *class_like* denotes keywords **class**, **interface**, and **enum**, reduction (1) makes jweave accept not only `@interface` but also `@enum` and even, if applied twice, `@@class`. As explained above, it is javac's responsibility to reject such code, not jweave's.

Reductions (2), (3), and (4) handle annotations: when annotation name is a compound name, when an annotation has some parameters or when it does not have any parameters, respectively. The last production transforms an annotation head consisting of a (compound) name and its parameters designated as *parexp*, i.e., an expression in parentheses, into an annotation.

Template class names. A simple name, consisting of a single identifier, is tokenised as *exp*. Hence, a chunk of code `A<B` can either be a comparison of two variables named A and B or a template class name where the final > might be missing (as it is a part of another chunk in another section).

As all expressions, declatators, type names and similar syntactic structure sooner or later reduce to *exp*, an important insight by itself, symbols < and > cannot be treated as a binary operator but specifically as separate symbols *prelangle* and *prerangle*. Thus, apart from other reductions for *exp*, there are two special reductions:

$$exp\ prelangle\ exp\ prerangle \Rightarrow exp \quad (6)$$

$$exp\ prelangle\ exp\ \Lambda \Rightarrow exp \quad (7)$$

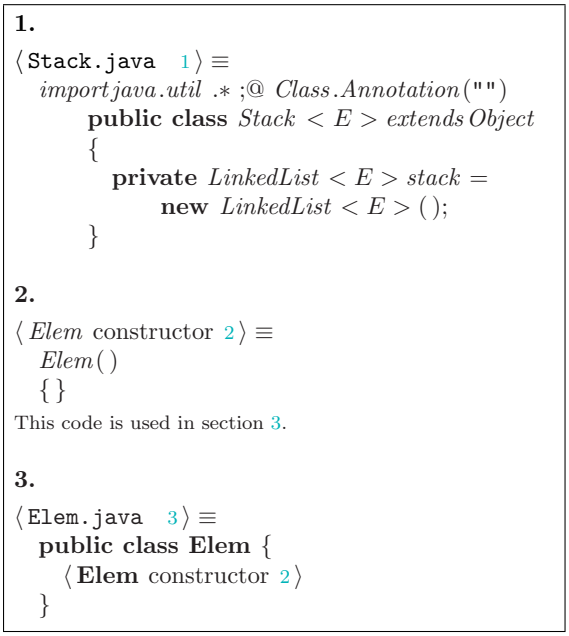

Figure 5: Typeset with CWEB – compare with Figure 6.

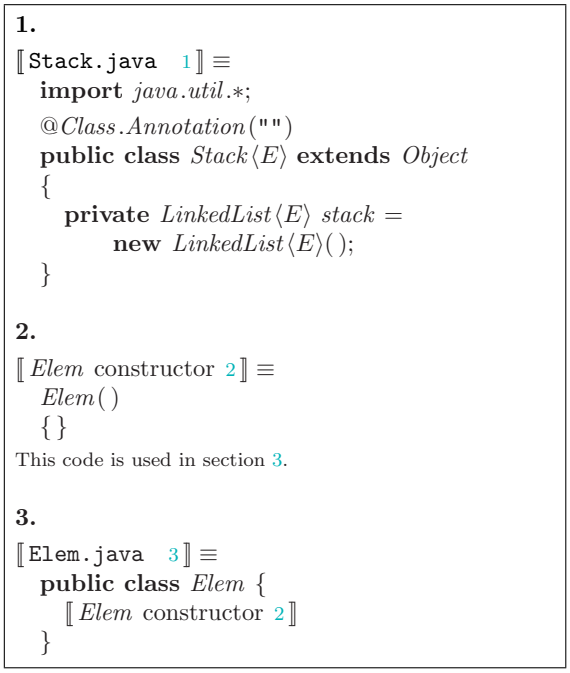

Figure 6: Typeset with JWEB – compare with Figure 5.

where Λ denotes any symbol than cannot extend *exp*. Reduction (6) typesets *prelangle* and *prerangle* as ⟨ and ⟩ while reduction (7) typesets them as < and >.

Finally, let us look at Figures 5 and 6 to see the difference between CWEB and JWEB — to fit within a two column document, a line break `@/` was inserted in both cases after annotation, right before **public**, and a line break `@|` was inserted right after =. Although the differences might seem minor at the first glance, the code in Figure 6 is typeset more properly: the import declaration is recognised, annotation is recognised, the name of the template class is typeset correctly, and most importantly, each occurrence of a class name is typeset in the same manner (see *Elem* in sections 2 and 3, in the code as well as in the section header).

6 Conclusion

As noted above, the three major design issues explained in Section 4 allow different decisions, and the grammar briefly outlined in Section 5 is neither the only one appropriate nor proven to be the optimal one. Yet some decision simply must be taken, for bad or for worse.

Once JWEB is fully tested, the authors of CWEB will be asked for permission to publicly release CWEB-based JWEB; if permission is not granted, only patch files will be released. It is estimated that this should happen in late 2016.

References

[1] D. E. Knuth, *The WEB System of Structured Documentation*, available at Comprehensive TEX Archive Network (ctan.org), 1981.

[2] D. E. Knuth, *The WEB System of Structured Documentation*, Technical report, Stanford University, Stanford, CA, USA, 1983.

[3] D. E. Knuth, *Literate Programming*, CSLI Lecture Notes, Center for the Study of Language and Information, Stanford, CA, USA, no 27, 1984.

[4] D. E. Knuth, *TEX: The program*, Computers & Typesetting, Vol. B, Addison-Wesley, Reading, MA, USA, 1986.

[5] D. E. Knuth, *METAFONT: The program*, Computers & Typesetting, Vol. D, Addison-Wesley, Reading, MA, USA, 1986.

[6] D. E. Knuth, Silvio Levy, *The CWEB System of Structured Documentation*, Addison-Wesley, Reading, MA, USA, 1993.

[7] J. A. Krommes, *The WEB System of Structured Software Design and Documentation for C, C++, Fortran, Ratfor, and TEX*, available at Comprehensive TEX Archive Network (ctan.org), 1993.

[8] P. Briggs, J. D. Ramsdell, M. W. Mengel, S. Wright, K. Harwood, *Nuweb Version 1.57 — A Simple Literate Programming Tool*, available at nuweb.sourceforge.net.

[9] N. Ramsey *Literate programming simplified*, IEEE Software, 11(5), 97–105, 1994.

[10] N. Ramsey, *Literate programming: Weaving a language-independent WEB*, Communications of the ACM, 32(9), 1051-1055, 1989.

[11] D. Kramer, *API Documentation from Source Code Comments: A Case Study of Javadoc*, Proceedings of the 17th Annual International Conference on Computer Documentation (SIGDOC'99), New Orleans, LA, USA, 147–153, 1999.

[12] J. Stylos, B. A. Myers, Z. Yang, *Jadeite: Improving API Documentation Using Usage Information*, Extended Abstracts on Human Factors in Computing Systems (CHI EA'09), Boston, MA, USA, 4429–4434, 2009.

[13] G. Dubochet, D. Malayeri, *Improving API Documentation for Java-like Languages*, Evaluation and Usability of Programming Languages and Tools (PLATEAU'10), Reno, NV, USA, 3:1–3:1, 2010.