Building a Language-Independent `WEB`*

Norman Ramsey[†]
Odyssey Research Associates

November 18, 1998

In the fall of 1987 I started planning the implementation of a suite of tools for building verified Ada programs [Ramsey 89]. The first tool to be built was a verification condition generator, which was to be formally defined using the typed lambda calculus. I was eager to include the definition with the code so that it would be easy to check the code's correctness. Using `WEB` would have made this easy, but, unfortunately, the target programming language was SSL (a language for specifying structure editors), and the only languages for which `WEB` implementations were available were Pascal and C.

Writing a new `WEB` from scratch didn't make sense, so I decided to modify Silvio Levy's implementation of `WEB` in C [Levy 87], to get a `WEB` that would be written in C, but would read and write SSL code. From my previous experiences modifying `WEB`, I knew that the most time-consuming job would be fine-tuning the grammar that `WEAVE` uses to prettyprint code. I believed I could make debugging that grammar a lot less painful if, instead of trying to make dozens of small modifications by hand, I wrote a simple program, perhaps an AWK script, that would read a description of the grammar and generate C code for `WEAVE`. That AWK script became `SPIDER`, a program that turns language descriptions into C code for `TANGLE` and `WEAVE`. I have used `SPIDER` to generate `WEB`s for C, AWK, SSL, Ada, and a couple of other languages. I won't go into the details of `SPIDER`; instead, I'll try to describe what `SPIDER` does to accomplish its mission, or how to take the "essence of `WEB`" and make it language-independent.

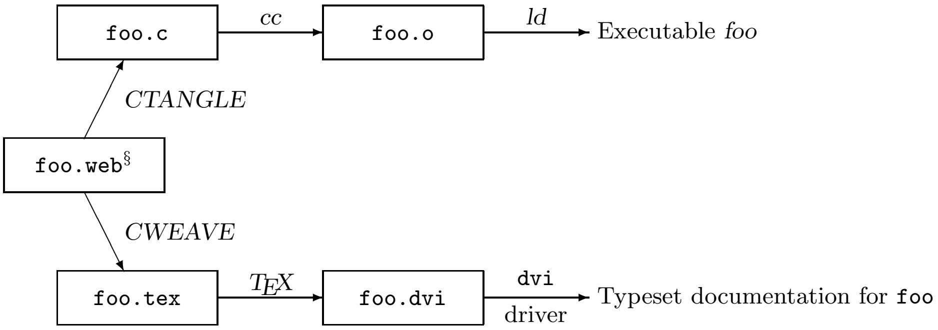
When using `WEB`, a programmer writes a single source file, `foo.web`,

---

[*] This research has been sponsored in part by the USAF, Rome Air Development Center, under contract number F30602–86–C–0071.

[†] Current address: Department of Computer Science, Princeton University, Princeton, New Jersey 08544

Figure 1: Processing a C `web` file

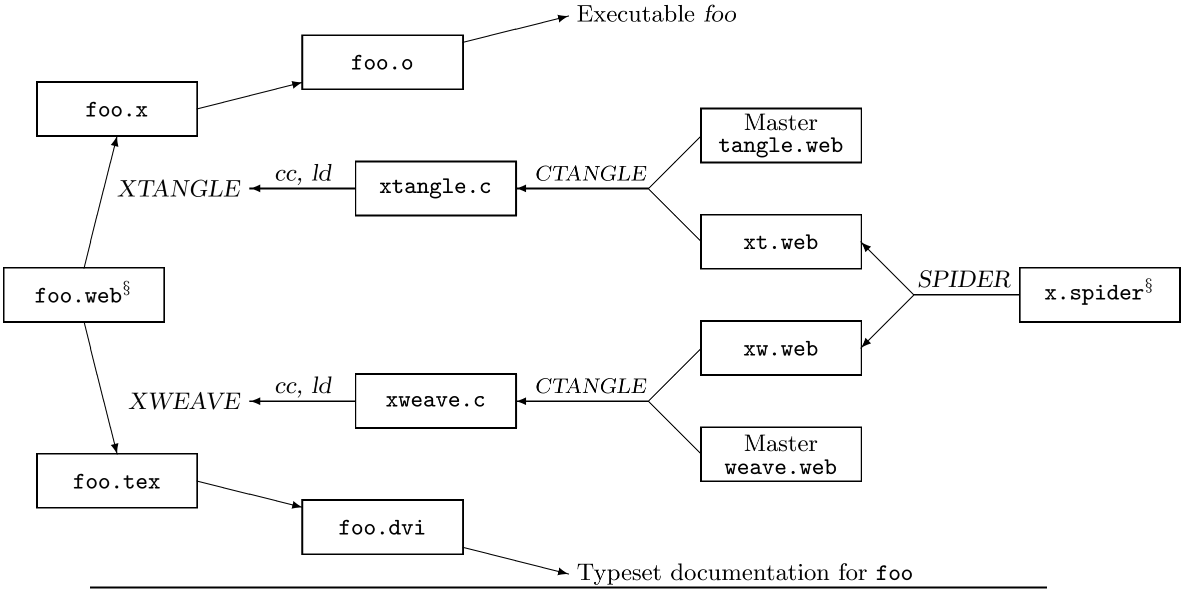


that holds both code and documentation. TANGLE and WEAVE read that file. TANGLE extracts the code from the WEB file and rewrites it in a form suitable for compiling. WEAVE passes the documentation parts to a document formatter (TEX), and prettyprints the code parts. The whole process is shown in Figure 1, for C programs written in WEB. The § represents files that have to be written by hand. *Slant* type is used for the names of executable programs. *CTANGLE* and *CWEAVE* are the C-language versions of TANGLE and WEAVE, *cc* is a C compiler, and *ld* is a loader.

SPIDER is used to construct *instances* of TANGLE and WEAVE, and these instances are used to process programs as shown in Figure 1. Code for the language-dependent parts of these instances is generated by SPIDER when it reads a language description file written by a WEB designer. Figure 2 shows how instances of TANGLE and WEAVE are generated. SPIDER converts a hand-written description of a programming language into C WEB code for the language-dependent parts of TANGLE and WEAVE. In Figure 2 the target programming language is a hypothetical "X," and the description file is called "`x.spider`." CTANGLE combines the code SPIDER writes with the "master copies" of `tangle.web` and `weave.web`, which contain the language-independent parts of TANGLE and WEAVE. The result is C source code for XTANGLE and XWEAVE. After that code is compiled and loaded with WEB's I/O code, the resulting executable versions of XTANGLE and XWEAVE can be used to process X-language programs written in WEB format, as shown around the periphery of Figure 2.

The master copies of `tangle.web` and `weave.web` are about 1800 and

Figure 2: Building and using an instance of `WEB` (for language X)



3200 lines long, respectively. About one third of these lines are comments. To illustrate the other size, suppose X is the Ada programming language. The `ada.spider` file is about 260 lines long, and from it `SPIDER` generates about 1400 lines of `ADATANGLE` and `ADAWEAVE`. About one tenth of these lines are comments. It is typical for `SPIDER` to generate between $5n$ and $6n$ lines of C `WEB` code from an n line language description.

A `WEB` program is a collection of "sections," each of which has a documentation part, a definition part, and a code part. The documentation part describes what the section is supposed to do, and is intended to be processed by a formatter—my `WEB`s use TEX, which is especially convenient for mathematical symbols like those used in writing a formal semantics. The definition part contains macro definitions. Each macro may have parameters or not, as the programmer chooses. The code in the code part is a fragment of the whole program. It is called a "module" and can be named or unnamed. When the module is named, the module name "stands for" that code, just as a macro name stands for the code in its definition. The unnamed module is special; the code in the unnamed module is considered to be "the program."

Figure 3: Table Inversion

```
@ The array |to_ascii| converts an EBCDIC code to
an ASCII code, or to |-1| if there is no ASCII
equivalent to the given code.
@d UNDEFINED_CODE = -1
@<Invert |to_ascii|, producing |to_ebcdic|@>=
    @<Set |to_ebcdic[i]=UNDEFINED_CODE| for all |i|@>@;
    for (i=0; i<256; i++)
        if (to_ascii[i] != UNDEFINED_CODE)
            to_ebcdic[to_ascii[i]]=i;

@ @<Set |to_ebcdic[i]=UNDEFINED_CODE| for all |i|@>=
    for (i=0; i<128; i++) to_ebcdic[i] = UNDEFINED_CODE;
```

Figure 3 shows a fragment of a `WEB` program; the fragment inverts an EBCDIC-to-ASCII table to obtain an ASCII-to-EBCDIC table. The target programming language is C. One module, ⟨Invert *to_ascii*, producing *to_ebcdic*⟩, uses the code defined in the other, ⟨Set $to\_ebcdic[i] \leftarrow UNDEFINED\_CODE$ for all i⟩. The program, `foo`, of which this fragment is a part, can be input to `CTANGLE` and `CWEAVE`, to produce `foo.c` and `foo.tex` respectively, as shown in Figure 1.

`TANGLE`'s job is to take a collection of sections and to produce a compilable program. `TANGLE` reads all the sections, skipping the documentation parts completely, but storing the macro definitions from the definition parts and the module definitions from the code parts. After it has read all the sections, `TANGLE` then writes out the code in the unnamed module. But whenever it encounters a module name in that code, instead of writing out the name, it writes out the code for which this name stands. That code may itself contain module names, which are replaced with the code for which they stand, and so on until `TANGLE` reaches code which contains no occurrences of module names. `TANGLE` processes macros similarly, except that macros may have parameters (modules may not).

As I've described it, the "essence of tangling" is language-independent. In the full implementation of `TANGLE` there are only a few language-dependent details, and almost all of them come up only in lexical analysis. During its input phase, `TANGLE` converts macro definitions and module definitions into token lists. The major kinds of tokens are module name tokens, identifier tokens, and ordinary tokens. Identifier tokens may be expandable (when they are macro names) or unexpandable (when they are programming-language identifiers). Module name tokens are always expandable, and ordinary tokens are never expandable. `TANGLE` uses a stack to write out the token list corresponding to the unnamed module, expanding expandable tokens as it goes. No token is ever expanded until the time comes to write that token on the output.

To build the language-dependent part of `TANGLE`, it is enough to tell `TANGLE` how to tokenize the input and how to write out a token list. `TANGLE` uses a "lowest common denominator" lexical analyzer to tokenize its input. The set of tokens recognized by this lexical analyzer is the union of the sets of legal tokens of many different languages. For example, different ways of delimiting string literals are recognized. Identifiers may have underscores, even though some languages (e.g. Pascal) may not permit underscores in identifiers, and others (e.g. Ada) may not permit consecutive underscores in an identifier. In general, `TANGLE` and `WEAVE` do the right thing with legal programs, but they do not detect illegalities in a program.

`TANGLE`'s lexical analyzer recognizes a fixed set of tokens representing identifiers, string literals, and numeric literals. Any printable ASCII character which is not part of some other token forms a token all by itself. A `WEB` builder can specify that certain strings should be treated as single tokens, and `SPIDER` will convert the specifications into appropriate code for `TANGLE`. For example, when building `WEB` for C, we tell `SPIDER` that the strings `++`, `==`, and `&&` (and many others) should be treated as single tokens, by putting the statements

```
token ++ ...
token == ...
token && ...
```

into the language description file, `c.spider`.

`TANGLE` discards comments, rather than attempting to tokenize them. Comments are assumed to begin with a special string, and to end either with another string or with a newline. We specify C comment conventions by telling `SPIDER`

```
comment begin <"/*"> end <"*/">
```

On output, `TANGLE` converts tokens to text by inverting the process of lexical analysis, so, for example, the token $\boxed{\texttt{++}}$ is written out as "`++`".

TANGLE's output phase inserts white space between adjacent identifiers and numeric literals, but otherwise does not write white space. This can cause problems in some cases; for example, in older C compilers the string "`=-`" is ambiguous. We can solve this problem by telling SPIDER to build a TANGLE that uses the text "`= `" when writing the $\boxed{=}$:

```
token = tangleto <"= ">
```

In sum, we can make TANGLE language-independent with almost no effort. We do this by using a lowest common denominator lexical analyzer whose only parameter is a description of comments, and by providing a way to fine-tune the way TANGLE writes tokens.

Unlike TANGLE, WEAVE does no rearranging of the sections; its job is to translate its input into a prettyprinted program listing. The documentation part of each section is simply copied to the output, but the definition and code parts are prettyprinted. WEAVE's output is handed to a document formatter, which is assumed to implement a prettyprinting algorithm like that described by Oppen [Oppen 80]. Since my WEBs use TeX as the document formatter, the back-end prettyprinting is implemented by TeX macros.

WEAVE copies the documentation parts as texts, but it converts definition and code parts to token lists using the same lexical analyzer used by TANGLE. WEAVE's part of the prettyprinting task (as distinct from TeX's part) is converting these token lists to streams of TeX text, possibly with prettyprinting instructions intercalated between tokens. If you like, WEAVE's job is to produce the input to Oppen's algorithm. For simplicity, we'll discuss only three prettyprinting instructions: *indent* (increase the level of indentation), *outdent* (decrease the level of indentation), and *force* (force a line break).

We tell WEAVE how to convert tokens to TeX text by specifying a *translation* for each token. Suppose we want the C token $\boxed{\texttt{!=}}$ to be printed as "$\neq$", which is produced by the TeX text "`\ne`". Then we write

```
token != translation <"\\ne">
```

(Two backslashes appear in the translation because SPIDER uses C conventions for string literals. The angle brackets `<...>` delimit translations.) The default for translation is just as in TANGLE, so if we want "`+`" on input to produce "$+$" on output we need not specify a translation for the token $\boxed{+}$.

The process of deciding where to put line breaks and indentation is the most complicated part of WEAVE. We have to do this based on the sequence of tokens we have, but the exact details of which token is where usually aren't needed to do prettyprinting. Hence we introduce the *scrap*, which abstracts away from the detail not needed to do prettyprinting. A scrap has two parts: the translation, which we have already seen, and the *category*, which corresponds to a "part of speech" or a symbol in a grammar. `WEAVE` uses categories to decide where to put indentation and line breaks. Since there are usually many different tokens having the same category, prettyprinting is simplified enormously.

`WEAVE` begins processing a program fragment by tokenizing the fragment, then converting each token in the resulting token list into a scrap. It then attempts to reduce the length of the resulting scrap list by combining adjacent scraps into a single scrap, possibly intercalating additional translations, which may include *indent*, *outdent*, and *force* instructions. The scraps are combined according to one of many *reduction rules*. `WEAVE` decides which adjacent scraps are eligible to be reduced based only on the categories of the scraps and a knowledge of the reduction rules. The reduction rules are the productions of the *prettyprinting grammar*. `WEAVE`'s reductions of scraps are like the reductions done in bottom-up parsing.

To take an example, suppose that we want statements to be separated by line breaks. If we can guarantee that any scrap representing a statement has category `stmt`, it will be enough to specify the reduction rule

```
stmt <force> stmt --> stmt
```

which says "two adjacent `stmt` scraps may be reduced to a single `stmt` scrap by intercalating a forced line break between them."

So we tell `WEAVE` how to prettyprint a language by telling how to assign a category to each token and how to combine scraps. Here's another example: the language of C expressions. Let `math` be the category of expressions, `binop` be the category of binary infix operators, and `unop` be the category of both unary prefix and unary postfix operators. Here are some sample tokens:

```
token identifier category math
token + category binop
token - category binop
token = category binop translation <"\\leftarrow">
token == translation <"\\equiv"> category binop
token ( category open
token ) category close
```

Notice we print the $\boxed{\texttt{=}}$ token (assignment) as ←, whereas we print the $\boxed{\texttt{==}}$ token (comparison) as ≡. This makes it a bit easier for us to see when a programmer has mistakenly used $\boxed{\texttt{=}}$ instead of $\boxed{\texttt{==}}$.

The prettyprinting grammar for C expressions is:

```
math binop math --> math
math unop --> math
unop math --> math
open math close --> math
```

Using this grammar, WEAVE can take a long expression consisting of many scraps, and reduce it all to a single scrap of category `math`.

What about an operator like "`*`", which is both binary infix and unary prefix? This does the job:

```
token * category unorbinop
unorbinop math --> math
math unorbinop math --> math
```

There is a mechanism for assigning categories and translations to reserved words as well as to tokens, using slightly different syntax.

To give an idea of the complexity of the grammars, the grammar describing AWK uses 24 categories in 39 productions. The Ada grammar uses 40 categories in 65 productions, and the C grammar uses 54 categories in 129 productions.

SPIDER-generated versions of TANGLE and WEAVE differ subtly from the originals written by Donald Knuth. The most important difference is that SPIDER-generated WEB is not self-contained: where Knuth's Pascal WEB required only a Pascal compiler to bring up, SPIDER would need a C compiler and an AWK interpreter to generate a Pascal WEB, and a Pascal compiler for the resulting WEB to be of any use. Other differences are minor; for example, Knuth's TANGLE does arithmetic on constants at TANGLE time, but SPIDER-generated TANGLEs do not. Knuth's TANGLE provides three different kinds of macros, but none with more than one parameter; SPIDER-generated TANGLEs provide only one kind of macro, but macros of that kind may have from zero to thirty-two parameters.

SPIDER is a WEB generator, akin to parser generators. Both read formal descriptions of some part of a programming language, and both produce code that processes programs written in that language. Since both produce code that is part of the "compiler," using them doesn't introduce any extra steps into the processing of user programs. SPIDER itself is a large AWK script, written as a WEB program. `spider.web` is about 2600 lines long; about a third of these are comments.

The major cost of using SPIDER is the cost of learning yet another language. Learning this language is supposed to substitute for learning how to modify WEB, so it is probably not an exorbitant cost. Some other limitations are the the need for a C compiler and an AWK interpreter, and the need to use a lowest-common-denominator lexical analyzer.

The major benefit of using SPIDER is the ease with which new WEBs can be built. The SPIDER description of a language is much smaller than the WEB implementation generated from that description, and SPIDER descriptions of similar languages are similar. Using SPIDER one can build a WEB without understanding the details of WEB's implementation, and one can easily adjust that WEB to change as a language definition changes.

SPIDER should make one literate programming tool, WEB, available to a much larger audience. I hope that, by separating the language-independent parts of WEAVE and TANGLE, SPIDER will encourage us not just to think about what the essence of tangling and weaving is, but also about what the essence of literate programming is.

I enjoyed many useful discussions of WEB with Charlie Mills. I am grateful to Silvio Levy for providing his CWEB as the basis for the "master copies" of TANGLE and WEAVE, and to Dave Hanson for comments on an earlier version of this paper.

References

[Bentley 86] Jon L. Bentley, "Programming Pearls," *Communications of the ACM* **29:5** (May 1986), 364–368, and **29:6** (June 1986), 471–483.

[Knuth 84] Donald E. Knuth, "Literate Programming," *The Computer Journal* **27:2**(1984), 97-111.

[Levy 87] Silvio Levy, "WEB Adapted to C, Another Approach," *TUGBoat* **8:1**(1987), 12–13.

[Oppen 80] Derek Oppen, "Prettyprinting," TOPLAS **2:4** (October 1980), 465–483.

[Ramsey 89] Norman Ramsey, "Developing Formally Verified Ada Programs," Proceedings, *Fifth International Workshop on Software Specification and Design*, to appear.

[Van Wyk 87] Christopher J. Van Wyk, "Literate Programming," *Communications of the ACM* **30:7** (July 1987), 593–599, and **30:12** (December 1987), 1000–1010.