

interprets `x=-1` to mean “subtract 1 from x ,” which is very annoying. (The current idiom for this instruction is `x-=1`, and has been for over a decade.)

In order to “understand” input code and typeset it correctly, WEAVE’s parser transforms it into a sequence of *scraps*. Each scrap has a *category* (or *cat*, in the lingo), which is like its part of speech; when several scraps with the right cats are found in sequence, they “fire” a *production rule*; for this reason I also call them *sparks*, a quasi-anagram of scraps. It turns out that C’s syntax is different enough from Pascal’s that I needed to rewrite the production rules from scratch. For example, WEAVE should distinguish between the use of ‘*’ and ‘&’ as unary or binary operators: in the common idiom `char **argv`; both *’s “belong” to *argv*, so the output should look something like

```
char **argv;
```

Here’s what the T_EX output of my version of WEAVE looks like:

```
\&{char} ${*}{*}\{argv};$
```

(I’m thankful to Guntermann and Schrod for pointing out that this makes T_EX treat the asterisk as an Ord atom, not as a unary operator; but then I tried making them Op symbols, and the output didn’t look as good to me. Op symbols are meant for large operators, and things like log.)

Following the syntax definition of C (appendix A of Kernighan and Ritchie’s *The C Programming Language*), I wrote a relatively small set of rules (fewer than in the original WEB) that correctly parses all C constructs, including variable and function definitions. (It can fail spectacularly when module names or macro names are used in unusual ways; then manual formatting is called for.) In addition all variables being defined automatically get an underlined entry in the index. This means that it is no longer necessary to insert @! by hand when certain variables are being defined; I only use @! in special circumstances.

In C, when you say `typedef double foo`, the identifier `foo` can no longer be used to hold the value of a variable and it becomes syntactically equivalent to `double`. Thus WEAVE must give it the same syntactic treatment as a reserved word like `double`, and should also give it the same typographical treatment. Furthermore this should preferably be done automatically. Currently my version of WEAVE takes care of this by changing the *ilk* of the identifier at parsing time. This is not very elegant, and doesn’t work if the `typedef` definition is in a separate file; but then one can use WEB’s @f control sequence. There is also a new

control sequence @i which works like `#include`, but actually does include the file into WEAVE’s input.

```
Thanks to these changes, if I write
double inner_prod(vec1,vec2)
double vec1[dim],vec2[dim];
```

the variables *inner_prod*, *vec1* and *vec2* automatically get an underlined reference in the index; and if I write

```
typedef double vector[dim];
```

the word **vector** will from now on appear in boldface, and its “part of speech” becomes the same as that of **double**.

The last addition I made to WEAVE doesn’t show in the output, but it simplifies the grammar a lot. In the original WEB sparks of certain cats can be printed in math mode only, others in either mode and others in non-math mode only. With the relatively more complex grammar of C this scheme would imply a great increase in the number of cats and of production rules. Guntermann and Schrod’s solution (letter of December 11, 1986) was to typeset everything in math mode, and have the T_EX macros for the various output tokens switch back to non-math when necessary (using the `\ifmode` primitive). My solution is somewhat different: my sparks have a new attribute, their *mathness*, which is independent of their cat. When a production rule is fired, there is a special bit of code that inserts a ‘\$’ between sparks of different mathness, but the grammar itself doesn’t have to contain any mathness information. This makes WEAVE run about 2% slower, but T_EX’ing the WEAVE’d file is faster because T_EX doesn’t have to check the modes for lots of control sequences.

In conclusion, I am quite happy with CWEB, and do all my programming in it. CWEB itself is written in CWEB. Although I still consider the program experimental, I’m distributing it to interested people, and looking forward to comments and suggestions for improvements.

My heartfelt thanks to Klaus Guntermann and Joachim Schrod, for their helpful correspondence, and to Helmut Jürgensen and Barbara Beeton for inviting me to send this paper.