A Spider User's Guide

Norman Ramsey
Department of Computer Science
Princeton University

July 1989

Introduction

Donald Knuth developed the `WEB` system of structured documentation as part of the TEX project [Knuth 84]. `WEB` enables a programmer to divide his or her program into chunks (called *modules*), to associate text with each chunk, and to present the chunks in in any order. In Knuth's implementation, the chunks are pieces of PASCAL programs, and the chunks are formatted using TEX.

The `WEB` idea suggests a way of combining *any* programming language with *any* document formatting language, but until recently there was no software support for writing anything but PASCAL programs using `WEB`. In 1987, Silvio Levy rewrote the `WEB` system in C for C, while retaining TEX as the formatting language [Levy 87]. I have has modified Levy's implementation by removing the parts that make C the target programming language, and I have added a third tool, Spider, which complements `WEAVE` and `TANGLE`. Spider reads a description of a programming language, and writes source code for a `WEAVE` and `TANGLE` which support that language. Using Spider, a C compiler, and an Awk interpreter, an experienced systems programmer can generate a `WEB` system for an Algol-like language in a few hours.

This document explains how to use Spider to generate a `WEB` system for any programming language. (The choice of programming language is limited only by the lexical structure built into Spidery `WEB`, as we shall see.) You should consult the companion document, "The Spidery `WEB` system of structured documentation," to learn how to use the generated `WEB` system.

Prerequisites If you are going to use Spider to build a WEB system, you should be comfortable using WEB. To get an idea how WEB works, you should have read Knuth's introductory article on WEB [Knuth 84], as well as the WEB users' manual. (The WEB user's manual is pretty heavy going, so you may want to consult the Bibliography for more introductory material on WEB. Wayne Sewell's *Weaving a Program: Literate Programming in* WEB may be helpful [Sewell 89].)

In what follows we will assume that you know what WEAVE and TANGLE are, what input they expect, and what output they produce.

Plan of this guide We'll begin with a review of weaving and tangling, so that we can get an idea what is necessary to build a language-independent WEB. Then we'll present a discussion of the features of Spider that tell WEB about the programming language. We'll define these in detail and give some examples, and then we'll close with a complete description of the Spider language and tools.

How WEAVE and TANGLE see the world

Both WEAVE and TANGLE operate on the same input, a WEB file. WEAVE must examine this input and produce a TEX text, while TANGLE must produce a program text from the same input. The input consists of TEX parts, definition parts, and code parts. The TEX parts are the easiest to consider: WEAVE just copies them and TANGLE throws them away. The definition parts are a bit more complicated: WEAVE's job is to typeset them, while TANGLE must remember the definitions and expand them at the proper time. The code parts are the most complex of all: WEAVE must prettyprint them, and TANGLE must rearrange them into a coherent program text.

Lexical analysis in WEB Both WEAVE and TANGLE interpret the code parts as a stream of *tokens*. Since not all programming languages have the same tokens, it is Spider's job to tell WEAVE and TANGLE how to tokenize the input.[1] A Spidery WEB system can recognize the following kinds of tokens:

- identifiers
- numeric and string constants

[1] The current implementation of WEB's lexical analysis is limited. It should be replaced with something using regular expressions.

- newlines
- "pseudo-semicolons" (the token `@;`)
- reserved words
- non-alphanumeric tokens

`TANGLE` rearranges these tokens into one long program text, then writes out the program text token by token. Normally, `TANGLE` puts no white space between tokens, but it will put blanks between adjacent identifier, reserved word, and numeric constant tokens. Thus the input

```
if 0 > x-y then z := -1;
```

will be written out as

```
if 0>x-y then z:=-1;
```

and not

```
if0>x-ythenz:=-1;
```

which wouldn't parse. When it is desirable to have `TANGLE` translate the tokens differently, each token can be given a `tangleto` attribute, which specifies what program text is printed out for that token. For example, the `spider` file used to generate C `WEB` forces the `=` tokento be printed out as the string `"= "`, because in C the string `"=-"` can be ambiguous.

`WEAVE` must turn the token stream into a TEX text that will cause the code to be prettyprinted. It does so in three steps:

1. `WEAVE` turns each token into a *scrap*. A scrap has two important properties: its syntactic *category* and its *translation*. The categories are symbols in a prettyprinting grammar; that grammar tells `WEAVE` how to combine the scraps with prettyprinting instructions. The translations are the TEX texts that will tell TEX exactly how to print the scraps.

2. `WEAVE` reduces the scrap stream by combining scraps according to the productions of its prettyprinting grammar. (`WEAVE` does a kind of shift-reduce parsing of program fragments.) While combining the translations, `WEAVE` adds TEX text that will cause indenting, outdenting, line breaking, and so on.

3. Ideally, `WEAVE` keeps reducing scraps until it has a single scrap with a very long translation, but perhaps it will end up with an irreducible sequence of scraps. In any case, after no more reductions can be done, the translations of the remaining scraps are output one at a time.

Using Spider to tell `WEB` how to tokenize

Spider divides tokens into two classes; reserved words and other. The reserved words as specified using the `reserved` and `ilk` commands; the other tokens are specified using the `token` command. (This somewhat unusual setup is dictated by the way `WEAVE` works; its advantage is that is is easy to define a whole group of reserved words that will be treated identically.) Here's how it works: the `reserved` command designates a particular identifier as a reserved word, and says what *ilk* it belongs to. The `token` and `ilk` commands tell `WEAVE` and `TANGLE` what to do with a particular token, or with all the reserved words of a particular ilk. For each token or ilk one can specify the *tangleto* field, the token's *mathness* (whether it has to be typeset in math mode), and its *category* and *translation* (for conversion to scraps). All but the category can have defaults, set with the `defaults` command. Choice of category names is up to the user.

We will discuss the tokenization commands more later when we present the syntax of Spider in detail. Meanwhile, here are some example tokenization commands from the `spider` file for C:

```
token + category unorbinop
token - category unorbinop
token * category unorbinop
token = category equals translation <"\\leftarrow"> tangleto <"="-space>
token ~ category unop translation <"\\TI">
token & category unorbinop translation <"\\amp">
token ^ translation <"\\^"> category binop
token ? translation <"\\?"> category question
token % translation <"\\%"> category binop
token # translation <"\\#"> category sharp
token ! category unop translation <"\\neg">
token ( category lpar
token ) category rpar
token [ category lpar
token ] category rpar
```

```
token { translation <"\\{"> category lbrace
token } translation <"\\}"> category rbrace
token ++ category unop translation <"\\PP">
token -- category unop translation <"\\MM">
token != translation <"\\I"> category binop
token == translation <"\\S"> category binop
token && translation <"\\W"> category binop

ilk case_like category case
ilk int_like category int

reserved auto ilk int_like
reserved break ilk case_like
reserved case ilk case_like
reserved char ilk int_like
```

These show the definitions of some of the tokens used in C. Notice the `tangleto` option is almost always left to default, and the `translation` option is often left to default.

Once the tokens are specified, and each has a `tangleto` string, we can almost construct a `TANGLE` for the language. Before we can construct a `WEAVE`, we have to tell it how to combine and reduce scraps.

Using Spider to tell `WEAVE` how to reduce scraps

The most intricate part of `WEAVE` is its mechanism for converting programming language code into TEX code. `WEAVE` uses a simple bottom-up parsing algorithm, since it must deal with fragmentary constructions whose overall "part of speech" is not known.

The input is represented as a sequence of *scraps*, where each scrap of information consists of two parts, its *category* and its *translation*. The category is essentially a syntactic class, and the translation represents TEX code. Rules of syntax and semantics tell us how to combine adjacent scraps into larger ones, and if we are lucky an entire program text that starts out as hundreds of small scraps will join together into one gigantic scrap whose translation is the desired TEX code. If we are unlucky, we will be left with several scraps that don't combine; their translations will simply be output, one by one.

The combination rules are given as context-sensitive productions that are applied from left to right. Suppose that we are currently working on the sequence of scraps $s_1\, s_2 \ldots s_n$. We try first to find the longest production that applies to an initial substring $s_1\, s_2 \ldots$; but if no such productions exist, we find to find the longest production applicable to the next substring $s_2\, s_3 \ldots$; and if that fails, we try to match $s_3\, s_4 \ldots$, et cetera.

A production applies if the category codes have a given pattern. For example, if one of the productions is

```
open [ math semi <"\\,"-opt-5> ] --> open math
```

then it means that three consecutive scraps whose respective categories are `open`, `math`, and `semi` are converted to two scraps whose categories are `open` and `math`. The `open` scrap has not changed, while the string `<"\\,"-opt-5>` indicates that the new `math` scrap has a translation composed of the translation of the original `math` scrap followed by the translation of the `semi` scrap followed by '`\,`' followed by '`opt`' followed by '`5`'. (In the TEX file, this will specify an additional thin space after the semicolon, followed by an optional line break with penalty 50.) Translations are enclosed in angle brackets, and may contain quoted strings (using the C conventions to escape backslashes and so on), or may contain special keywords.

Before giving examples of useful productions, we'll break to give the detailed syntax of the Spider subset covered so far.

Syntax of `spider` files

Spider is an Awk program which converts a description of a language into C code for `WEAVE` and `TANGLE`. Since Spider is an Awk program, its input is a sequence of lines, and all Spider commands must fit on one line.

Comments and blank lines Because *any* character sequence can be a token of a programming language, we can't just designate a particular sequence as a "begin comment" marker. So in Spider there are no comments, only *comment lines*. A comment line is one whose first character is "`#`". The Spider processor ignores comment lines and blank lines.

Fields Each command in the `spider` file consists of a sequence of *fields*. These are just the Awk fields, and they are separated by white space. This

feature of Spider (inherited from Awk) forbids the use of white space within a field.

Translations

Most fields in a Spider file are simple identifiers, or perhaps strings of non-alphanumeric characters. The major exception is *translations*. Translations are always surrounded by angle brackets (`<>`), and consist of a (possibly empty) list of translation pieces. The pieces on a list are separated by dashes (`-`). A piece is one of:

- A quoted string. This string may contain embedded quotes escaped by "\", but it *must not* contain embedded white space or an embedded dash.
- The "self" marker, "`*`", refers to the sequence of characters making up the token being translated. The self marker is permitted only in certain contexts, and its precise meaning depends on the context.
- A digit.
- A key word. The key words known to Spider are

  `space` Stands for one space (`" "`).

  `dash` Stands for a dash (`"-"`).

  The other key words are passed on to `WEAVE`.

  `WEAVE` recognizes the following key words:

  `break_space` denotes an optional line break or an en space;

  `force` denotes a line break;

  `big_force` denotes a line break with additional vertical space;

  `opt` denotes an optional line break (with the continuation line indented two ems with respect to the normal starting position)—this code is followed by an integer n, and the break will occur with penalty $10n$;

  `backup` denotes a backspace of one em;

  `cancel` obliterates any `break_space` or `force` or `big_force` tokens that immediately precede or follow it and also cancels any `backup` tokens that follow it;

`indent` causes future lines to be indented one more em;

`outdent` causes future lines to be indented one less em.

`math_rel` translates to `\mathrel{`

`math_bin` translates to `\mathbin{`

`math_op` translates to `\mathop{`

The *only* key words that will work properly in math mode are `indent` and `outdent`, so when you're defining the translations of tokens you must use `mathness no` if your translations contain other key words. You may use any recognized key words in the translations of a production; there the mathness is automatically taken care of for you.

Here are some example translations:

```
<"\\"-space>
<indent-force>
<"{\\let\\\\=\\bf"-space>
<"}"-indent-"{}"-space>
```

Restricted translations In some cases, notably for a `tangleto` description, translations are *restricted*. A restricted translation is never converted to typesetting code, but is always converted to an ASCII string, usually for output by `TANGLE`, but sometimes for other things. A restricted translation may contain only *quoted strings* and the keywords `space` and `dash`.

`token` commands

The syntax of the `token` command is:

⟨*command*⟩ ::= `token` ⟨*token-designator*⟩ ⟨*token-descriptions*⟩

Where ⟨*token-descriptions*⟩ is a (possibly empty) list of token descriptions.

Token descriptions The token descriptions are

- `tangleto` ⟨*restricted translation*⟩

  The ⟨*restricted translation*⟩ tells `TANGLE` what program text to write out for this token. The only kinds of translation pieces valid in a restricted translation are quoted strings and the special words `space` and `dash`. If no `tangleto` description is present, `TANGLE` just writes out the sequence of characters that constitute the token.

- `translation` ⟨*translation*⟩

  Tells `WEAVE` what translation to assign when making this token into a scrap. The self marker (`*`) stands for the sequence of characters that were read in to make up the token. The translation often defaults to `translation <*>`; Spider is set up to have this default initially.

- `category` ⟨*category-name*⟩

  Tells `WEAVE` what category to assign when making this token into a scrap. If you're writing a Spider file, you may choose any category names you like, subject only to the restriction that they not conflict with other names known to Spider (e.g. predefined key words, names of ilks, and so on). Using category names that are identical to reserved words of the target programming language (or reserved words of C) is not only supported, it is strongly encouraged, for clarity. Also, when we get to the sample grammars later on, you will see some other conventions we use for category names.

- `mathness` ⟨*mathness-indicator*⟩

  where ⟨*mathness-indicator*⟩ is `yes`, `no`, or `maybe`. This indicates to `WEAVE` whether the translation for this token needs to be typeset in TEX's math mode or not, or whether it doesn't matter. When firing productions, `WEAVE` will place math shift characters (`$`) in the TEX text that guarantee the placement of tokens in the correct modes. Tokens with the *empty translation* (`<>`) should always have `mathness maybe`, lest they cause `WEAVE` to place two consecutive math shift characters.

- `name` ⟨*token-name*⟩

  This should only be necessary in debugging Spider or `WEB`. It causes the specified name to be attached to the token, so that a programmer can search for that name in the C code generated by Spider.

Token designators Spider recognizes the following token designators:

`identifier` A `token` command using this designator tells `WEAVE` and `TANGLE` what to do with identifier tokens. Unfortunately it is not possible to specify with Spider just what an identifier is; that definition is hard-wired into `WEAVE` and `TANGLE`. An identifier is the longest string match-

ing this regular expression[2]:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

`number` In the current implementation of Spider and `WEAVE`, a `token` command using this designator covers the treatment of both numeric constants and string constants. Like the identifiers, the definitions of what constitutes a numeric or string constant cannot be changed. A numeric constant is the longest string matching[3]:

```
[0-9]+(\.[0-9]*)?
```

A string constant is the longest string matching

```
\"([^"]*\\\")*[^"]*\"|'[^@\]'|'\\.'|'@@'
```

Carriage returns may appear in string constants if escaped by a backslash (\).

`newline` A `token` command using this descriptor tells `WEAVE` and `TANGLE` how to treat a newline. We'll see later how to make `WEAVE` ignore newlines.

`pseudo_semi` A `token` command using this descriptor tells `WEAVE` what to do with the `WEB` control sequence `@;`. This control sequence is always ignored by `TANGLE`.

⟨*characters*⟩ where none of the characters is alphanumeric. A `token` command using this descriptor defines the sequence of characters as a token, and tells `WEAVE` and `TANGLE` what to do with that token. A token may be a prefix of another token; `WEAVE` and `TANGLE` will prefer the longer token to the shorter. Thus, in a C `WEB`, `==` will be read as a single `==` token, not as two `=` tokens.

[2]The reader unfamiliar with the Unix notation for regular expressions should consult the *ed(1)* man page.

[3]There ought to be some kind of `WEB` control sequence to support floating point notation for those languages that have it.

Reserved word tokens

Reserved words are attached to a particular *ilk* using the `reserved` command.

`reserved` ⟨*reserved-word*⟩ [`ilk` ⟨*ilk-name*⟩]

If you're writing a Spider file, you may choose any ilk names you like, subject only to the restriction that they not conflict with other names known to Spider (e.g. predefined key words, names of categories, and so on). The convention, however, is to use ilk `with_like` for a reserved word `with`, and so on.[4]

The `ilk` and `token` commands have nearly identical syntax. The syntax of the `ilk` command is:

⟨*command*⟩ ::= `ilk` ⟨*ilk-name*⟩ ⟨*token-descriptions*⟩

In translations that appear in `ilk` commands, the self marker (`*`) designates the string of characters making up the reserved word, surrounded by `\&{...}`, which makes the reserved words appear in bold face.

Syntax of the prettyprinting grammar

Defining the tokens of a language is somewhat tedious, but it is essentially straightforward, and the definition usually does not need fine tuning. When developing a new `WEB` with Spider, you will spend most of your time writing the grammar that tells `WEAVE` how to reduce scraps. The grammar is defined as a sequence of context-sensitive productions. Each production has the form:

⟨*left context*⟩ `[` ⟨*firing instructions*⟩ `]` ⟨*right context*⟩
`-->` ⟨*left context*⟩ ⟨*target category*⟩ ⟨*right context*⟩

where the left and right contexts are (possibly empty) sequences of scrap designators, the firing instructions are a sequence of scrap designators and translations (containing at least one scrap designator), and the target category is a category designator. If the left and right contexts are both empty,

[4] The existence of this convention seduced me into adding a pernicious feature to Spider—if you omit the ilk from a `reserved` command, Spider will make an ilk name by appending `_like` to the name of the reserved word. Furthermore, if that ilk doesn't already exist, Spider will construct one. Don't use this feature.

the square brackets (`[]`) can be omitted, and the production is context free. The left and right contexts must be the same on both sides of the `-->`.

What does the production mean? Well, `WEAVE` is trying to reduce a sequence of scraps. So what `WEAVE` does is look at the sequence, to find out whether the left hand side of some production matches an initial subsequence of the scraps. `WEAVE` picks the first matching production, and *fires* it, reducing the scraps described in the firing instructions to a single scrap, and it gives the new scrap the *target category*. The translation of the new scrap is formed by concatenating the translations in the *firing instructions*, where a scrap designator stands for the translation of the designated scrap.

Here is the syntax that describes contexts, firing instructions, scrap designators, and so on.

⟨*left context*⟩ ::= ⟨*scrap designators*⟩
⟨*right context*⟩ ::= ⟨*scrap designators*⟩
⟨*firing instruction*⟩ ::= ⟨*scrap designator*⟩
⟨*firing instruction*⟩ ::= ⟨*translation*⟩
⟨*scrap designator*⟩ ::= `?`
⟨*scrap designator*⟩ ::= [`!`]⟨*category name*⟩[`*`]
⟨*scrap designator*⟩ ::= [`!`]⟨*category alternatives*⟩[`*`]
⟨*category alternatives*⟩ ::= `(`⟨*optional alternatives*⟩⟨*category name*⟩`)`
⟨*optional alternative*⟩ ::= ⟨*category name*⟩`|`
⟨*target category*⟩ ::= `#`⟨*integer*⟩
⟨*target category*⟩ ::= ⟨*category name*⟩

Matching the left hand side of a production When does a sequence of scraps match the left hand side of a production? For matching purposes, we can ignore the translations and the square brackets (`[]`), and look at the left hand side just as a sequence of scrap designators. A sequence of scraps matches a sequence of scrap designators if and only if each scrap on the sequence matches the corresponding scrap designator. Here are the rules for matching scrap designators (we can ignore starring[5]):

- Every scrap matches the designator `?`.
- A scrap matches ⟨*marked category*⟩ if and only if its category is the same as the category of the designator.

[5] A category name is said to be *starred* if it has the optional `*`.

- A scrap matches `!`⟨*marked category*⟩ if and only if its category is *not* the same as the category of the designator. (The `!` indicates negation.)
- A scrap matches a list of category alternatives if and only if its category is on the list of alternatives.
- A scrap matches a *negated* list of category alternatives if and only if its category is *not* on the list of alternatives.

Firing a production Once a match is found, `WEAVE` fires the production by replacing the subsequence of scraps matching the firing instructions. `WEAVE` replaces this subsequence with a new scrap whose category is the target category, and whose translation is the concatenation of all the translations in the firing instructions. (When the new translation is constructed, the translations of the old scraps are included at the positions of the corresponding scrap designators.) If the target category is not given by name, but rather by number (`#`n), `WEAVE` will take the category of the nth scrap in the subsequence that matches the left hand side of the production, and make that the target category.

Side effects of firing a production When a production fires, `WEAVE` will *underline the index entry* for the first identifier in any *starred* scrap.

If no initial subsequence matches any production If the initial subsequence of scraps does not match the left hand side of any production, `WEAVE` will try to match the subsequence beginning with the second scrap, and so on, until a match is found. Once a match is found, `WEAVE` fires the production, changing its sequence of scraps. It then starts all over again at the beginning of the new sequence, looking for a match.[6] If *no* subsequence of the scraps matches any production, then the sequence of scraps is irreducible, and `WEAVE` writes out the translations of the scraps, one at a time.

Examples of `WEAVE` grammars

This all must seem very intimidating, but it's not really. In this section we present some grammar fragments and explain what's going on.

[6] The implementation is better than that; Spider figures out just how much `WEAVE` must backtrack to get the same effect as returning to the beginning.

Short examples

```
? ignore_scrap --> #1
```

This production should appear in every grammar, because Spidery `WEAVE` expects category `ignore_scrap` to exist with roughly this semantics. (For example, all comments generate scraps of category `ignore_scrap`.) Any scrap of category `ignore_scrap` essentially doesn't affect the reduction of scraps: it is absorbed into the scrap to its left.

```
token newline category newline translation <>
newline --> ignore_scrap
```

This token definition and production, combined with the previous production, causes `WEAVE` to ignore all newlines.

For this next example, from the C grammar, you will need to know that `math` represents a mathematical expression, `semi` a semicolon, and `stmt` a statement or sequence of statements.

```
math semi --> stmt
stmt <force> stmt --> stmt
```

The first production says that a mathematical expression, followed by a semicolon, should be treated as a statement. The second says that two statements can be combined to make a single statement by putting a line break between them.

Expressions This more extended example shows the treatment of expressions in Awk. This is identical to the treatment of expressions in C and in several other languages. We will use the following categories:

math A mathematical expression

binop A binary infix operator

unop A unary prefix or postfix operator

unorbinop An operator that could be binary infix or unary prefix

To show you how these might be used, here are some sample token definitions using these categories:

```
token + category unorbinop
token - category unorbinop
token * category binop
token / category binop
token < category binop
token > category binop
token , category binop translation <",\\,"-opt-3>
token = category binop translation <"\\K">
token != translation <"\\I"> category binop
token == name eq_eq translation <"\\S"> category binop
token ++ name gt_gt category unop translation <"\\uparrow">
token -- name lt_lt category unop translation <"\\downarrow">
```

Notice that the translation for the comma specifies a thin space and an optional line break after the comma. The translations of `=`, `!=`, and `==` produce $\leftarrow$, $\neq$, and $\equiv$.

Here is the grammar for expressions.

```
math (binop|unorbinop) math --> math
(unop|unorbinop) math --> math
math unop --> math
math <"\\"-space> math --> math
```

In Awk there is no concatenation operator; concatenation is by juxtaposition. The last production tells `WEAVE` to insert a space between two juxtaposed expressions.

So far we haven't dealt with parentheses, but that's easily done:

```
token ( category open
token ) category close
token [ category open
token ] category close
open math close --> math
```

Now this grammar just given doesn't handle the Awk or C `+=` feature very well; `x+=1` comes out as $x+ \leftarrow 1$, and `x/=2` is irreducible! Here's the cure; first, we make a new category for assignment:

```
token = category equals translation <"\\K">
```

And then we write productions that reduces assignment (possibly preceded by another operator) to a binary operator:

```
<"\\buildrel"> (binop|unorbinop) <"\\over{"> equals <"}"> --> binop
equals --> binop
```

Notice that, given the rules stated above, the second production can fire only if `equals` is *not* preceded by an operator. On input `x+=1`, the first production fires, and we have the translation $x \stackrel{+}{\leftarrow} 1$.

Conditional statements Here is the grammar for (possibly nested) conditional statements in Awk.

```
if <"\\"-space> math --> ifmath
ifmath lbrace --> ifbrace
ifmath newline --> ifline
ifbrace <force> stmt --> ifbrace
ifbrace <outdent-force> close else <"\\"-space> if --> if
ifbrace <outdent-force> close else lbrace --> ifbrace
ifbrace <outdent-force> close else newline --> ifline
ifbrace <outdent-force> close --> stmt
(ifline|ifmath) <indent-force> stmt <outdent> --> stmt
```

It relies on the following token definitions:

```
ilk if_like category if
reserved if
ilk else_like category else
reserved else
token { translation <"\\;\\{"-indent> category lbrace
token } translation <"\\}\\"-space> category close
token newline category newline translation <>
```

Handling preprocessor directives in C Here is a simplified version of the grammar that handles C preprocessor directives. It puts the directives on the left hand margin, and correctly handles newlines escaped with backslashes. (The full version is also able to distinguish `<...>` bracketing a file name from the use of the same symbols to mean "less than" and "greater than.")

```
# control sequence \8 puts things on the left margin
<"\\8"> sharp <"{\\let\\\\=\\bf"-space> math <"}"-indent-"{}"-space> --> preproc
preproc backslash <force-"\\8\\hskip1em"-space> newline --> preproc
<force> preproc <force-outdent> newline --> ignore_scrap
preproc math --> preproc
newline --> ignore_scrap
```

The `\let` in the first production makes the identifier following the `#` come out in bold face.

Using context-dependent productions

So far we've been able to do a lot without using the context-dependent features of Spider productions. (For example, the entire `spider` file for Awk is written using only context-free productions.) Now we'll show some examples that use the context-dependence.

In the grammar for Ada, a semicolon is used as a terminator for statements. But semicolons are also used as *separators* in parameter declarations. The first two productions here find the statements, but the third production supersedes them when a semicolon is seen in a parenthesized list.

```
semi --> terminator
math terminator --> stmt
open [ math semi ] --> open math
```

Underlining the index entry for the name of a declared function In SSL, function declarations begin with the type of the function being declared, followed by the name of that function. The following production causes the index entry for that function to be underlined, so that we can look up the function name in the index and easily find the section in which the function is declared:

```
decl simp [ simp* ] --> decl simp math
```

Where we've relied on

```
token identifier category simp mathness yes
```

Conditional expressions Suppose we want to format conditional expressions (for example in C) like this:

⟨*condition*⟩
    ? ⟨*expression*⟩
    : ⟨*expression*⟩

The problem is that it's hard to know when the conditional expression ends. It's essentially a question of precedence, and what we're going to do is look ahead until we see an operator with sufficiently low precedence that it terminates a conditional expression. In SSL a conditional expression can be

terminated by a semicolon, a right parenthesis, a comma, or a colon. We'll use the *right context* to do the lookahead.

```
token ? translation <"\\?"> category question
token : category colon

<indent-force> question math <force> colon --> condbegin
[ condbegin math <outdent> ] (semi|close|comma|colon) --> math (semi|close|comma|colon)
```

Debugging a prettyprinting grammar

`WEAVE` has two tracing modes that can help you debug a prettyprinting grammar. The control sequence `@1` turns on partial tracing, and `@2` turns on a full trace. `@0` turns tracing back off again. In the partial tracing mode, `WEAVE` applies all the productions as many times as possible, and then it prints out the irreducible scraps that remain. If the scraps reduce to a single scrap, no diagnostics are printed.

When a scrap is printed, `WEAVE` prints a leading `+` or `-`, the name of the category of that scrap, and a trailing `+` or `-`. The `+` indicates that TEX should be in math mode, and the `-` that TEX should not be in math mode, at the beginning and end of the scrap's translation, respectively. (You can see the translations by looking at the `.tex` file, since that's where they're written out.)

For beginners, the full trace is more helpful. It prints out the following information every time a production is fired:

- The number of the production just fired (from `productions.list`);
- The sequence of scraps `WEAVE` is now trying to reduce;
- A `*` indicating what subsequence `WEAVE` will try to reduce next.

A good way to understand how prettyprinting grammars work is to take a `productions.list` file, and look at a full trace of the corresponding `WEAVE`. Or, if you prefer, you can simulate by hand the action of `WEAVE` on a sequence of scraps.

The rest of the Spider language

The tokens and the grammar are not quite the whole story. Here's the rest of the truth about what you can do with Spider.

Naming the target language

When a Spidery `WEAVE` or `TANGLE` starts up, it prints the target language for which it was generated, and the date and time of the generation. The `language` command is used to identify the language being targeted. Its syntax is

> `language` ⟨*language-name*⟩ [`extension` ⟨*extension-name*⟩]
> [`version` ⟨*version-name*⟩]

The extension name is the extension used (in place of `.web`) by `TANGLE` to write out the program text for the unnamed module. The extension is also used to construct a language-specific file of TEX macros to be used by `WEAVE`, so different languages should always have different extensions. If the extension is not given it defaults to the language name. If the version information is given, it too will be printed out at startup.

The `c.spider` file I use for Unix has

```
language C extension c
```

Defining TEX macros

In addition to the "kernel" `WEB` macros stored in `webkernel.tex`, you may want to create some TEX macros of your own for use in translations. Any macro definitions you put between lines saying `macros begin` and `macros end` will be included verbatim in the TEX macro file for this language. That macro file will automatically be `\input` by every TEX file generated by this `WEAVE`.

For example, the C grammar includes productions to handle preprocessor directives. These directives may include file names that are delimited by angle brackets. I wanted to use the abbreviations `\LN` and `\RN` for left and right angle brackets, so I included

```
macros begin
\let\LN\langle
\let\RN\rangle
macros end
```

in the `c.spider` file.

Setting default token information

It's possible to set default values for the `translation` and `mathness` properties of tokens, so that they don't have to be repeated. This is done with the `default` command, whose syntax is:

`default` ⟨*token descriptions*⟩

The initial defaults (when Spider begins execution) are `translation <*>` and `mathness maybe`.

Specifying the treatment of modules

`WEB` introduces a new kind of token that isn't in any programming language, and that's the module name (`@<...@>` or `@(...@>`). `TANGLE`'s job is to convert the module names to program text, and when `TANGLE` is finished no module names remain. But `WEAVE` has to typeset the module names, and we need to tell `WEAVE` what category to give a scrap created from a module name. We allow two different categories, one for the definition of the module name (at the beginning of a module), and one for a use of a module name. The syntax of the `module` command is:

`module` [`definition` ⟨*category name*⟩] [`use` ⟨*category name*⟩]

The `c.spider` file contains the line

```
module definition decl use math
```

Determining the at sign

When generating a `WEB` system with Spider, you're not required to use "`@`" as the "magic at sign" that introduces `WEB` control sequences. By convention, however, we use "`@`" unless that is deemed unsuitable. If "`@`" is unsuitable, we use "`#`." Since Spider writes C `WEB` code for `WEAVE` and `TANGLE`, it writes a lot of `@` signs. I didn't when to have to escape each one, so I chose "`#`" for Awk `WEB`'s at sign:

```
at_sign #
```

The at sign defaults to "`@`" if left unspecified.

Changing control sequences Changing the at sign changes the meaning of one or two control sequences. This is more easily illustrated by example than explained. Suppose we change the at sign to `#`. Then in the resulting `WEB` two control sequences have new meanings:

`##` Stands for a `#` in the input, by analogy with `@@` in normal `WEB`. You will need this when defining TEX macros that take parameters.

`#@` This is the new name of the control sequence normally represented by `@#`. You would use `#@` to get a line break followed by vertical white space.

If you change the at sign to something other than `@` or `#`, the above will still hold provided you substitute your at sign for `#`.

Comments in the programming language

We have to tell `WEAVE` and `TANGLE` how to recognize comments in our target programming language, since comment text is treated as TEX text by `WEAVE` and is ignored by `TANGLE`. The syntax of the `comment` command is

`comment begin` ⟨*restricted translation*⟩
`end` (⟨*restricted translation*⟩|`newline`)

The restricted translations can include only quoted strings, `space`, and `dash`. They give the character sequences that begin and end comments. If comments end with newlines the correct incantation is `end newline`.

If the comment character is the same as the at sign, it has to be doubled in the `WEB` file to have any effect. For reasons that I've forgotten, Spider is too dumb to figure this out and *you must double the comment character in the Spider file*. This is not totally unreasonable since any at sign that actually appears in a `WEB` file will have to be double to be interpreted correctly.

`WEAVE` uses the macros `\commentbegin` and `\commentend` at the beginning and end of comments, so you can define these to be whatever you like (using the `macros` command) if you don't like Spider's defaults. Spider is smart enough to escape TEX's special characters in coming up with these defaults.

Here's a real-world ugly example of how things really are, from the `spider` file for Awk:

```
comment begin <"##"> end newline
```

```
macros begin
\def\commentbegin{\#} % we don't want \#\#
macros end
```

Controlling line numbering

A compiler doesn't get to see `WEB` files directly; it has to read the output of `TANGLE`. Error messages labelled with line numbers from a tangled file aren't very helpful, so Spidery `TANGLE` does something to improve the situation: it writes `#line` directives into its output, in the manner of the C preprocessor. (`TANGLE` also preserves the line breaking of the `WEB` source, so that the `#line` information will be useful.) For systems like Unix with `cc` and `dbx`, both compile-time and run-time debugging can be done in terms of the `WEB` source, and the intermediate programming language source need never be consulted.

Not all compilers support line numbering with `#line` directives, so Spider provides a `line` command to change the format of the `#line` directives. If your compiler doesn't support `#line`, you can use the `line` command to turn the line number information into a comment.[7] The syntax is:

`line begin` ⟨*restricted translation*⟩ `end` ⟨*restricted translation*⟩

The `begin` translation tells what string to put in front of the file name and line number information; the `end` translation tells what to put afterward. The defaults (which are set for C) are

```
line begin <"#line"> end <"">
```

Here's an example from the Ada Spider file, which makes the line number information into an Ada comment:

```
line begin <"--"-space-"line"> end <"">
```

Showing the date of generation

When Spidery `WEAVE` and `TANGLE` start up, they print the date and time at which their Spider file was processed. This is done through the good offices of Spider's `date` command, which is

`date` ⟨*date*⟩

[7] There should be a command that turns off line numbering.

where ⟨*date*⟩ looks like `"Fri Dec 11 11:31:18 EST 1987"` or some such. Normally you never need to use the `date` command, because one is inserted automatically by the Spider tools, but if you're porting Spider to a non-Unix machine you need to know about it.

Spider's error messages

Spider makes a lot of attempts to detect errors in a Spider specification. Spider's error messages are intended to be self-explanatory, but I don't know how well they succeed. In case you run into trouble, here are the error conditions Spider tries to detect:

- Garbled commands, lines with bad fields in them, or commands with unused fields. Any command with a field Spider can't follow or with an extra field is ignored from the bad field onward, but the earlier fields may have an effect. Any production with a bad field or other error is dropped completely.
- Missing `language` command.
- `macros` or `comment` command before `language` command. Spider uses the `extension` information from the `language` command to determine the name of the file to which the macros will be written, and the `comment` command causes Spider to write macros telling TEX what to do at the beginning and end of comments.
- Contexts don't match on the left and right sides of a production.
- A numbered target token doesn't fall in the range defined by the left hand side of its production.
- Some category is never *appended*. This means there is no way to create a scrap with this category. Spider only looks to see that each category appears at least once as the category of some token or as the category of the target token in some production, so Spider might fail to detect this condition (if there is some production that can never fire).
- Some category is never *reduced*. This means that the category never appears in a scrap designator from the firing instructions of a production. If a category is never reduced, Spider only issues a warning, and does not halt the compilation process with an error.

The append and reduce checks will usually catch you if you misspell a category name.

- You defined more tokens than `WEAVE` and `TANGLE` can handle.
- You forgot token information for identifiers, numeric constants, newlines, pseudo-semicolons (`@;`), module definitions, or module uses.
- Some ilk has no translation, or there is some ilk of which there are no reserved words.

Spider's output files

Spider writes many output files, and you may want to look at them to figure out what's going on. Here is a partial list (you can find a complete list by consulting `spider.web`):

`cycle.test` Used to try to detect potential loops in the grammar. Such loops can cause `WEAVE` to run indefinitely (until it runs out of memory) on certain inputs. Discussed below with the Spider tools.

`spider.slog` A verbose discussion of everything Spider did while it was processing your file. To be consulted when things go very wrong.

`*web.tex` The macros specific to the generated `WEB`.

`productions.list` A numbered list of all the productions. This list is invaluable when you are trying to debug a grammar using Spidery `WEAVE`'s tracing facilities (`@2`).

Using Spider to make `WEB` (the Spider tools)

Many of the Spider tools do error checking, like:

- Check to see there are no duplicate names among the categories, ilks, and translation keywords.
- Check the translation keywords against a list of those recognized by `WEAVE`, and yelps if trouble happens.
- Try to determine whether there is a "production cycle" that could cause `WEAVE` to loop infinitely by firing the productions in the cycle one after another.

I'm not going to say much about how to do all this, or how to make `WEAVE` and `TANGLE`; instead I'm going to show you a `Makefile` and comment on it a little bit. Since right now Spidery `WEB` is available only on Unix systems, chances are you have the `Makefile` and can just type "`make tangle`" or "`make weave`." If not, reading the Makefile is still your best bet to figure out what the tools do.

We assume that you are making `WEAVE` and `TANGLE` in some directory, and that the "master sources" for Spidery `WEB` are kept in some other directory. Some of the `Makefile` macros deserve special mention:

`THETANGLE` Name of the `TANGLE` we will generate.

`THEWEAVE` Name of the `WEAVE` we will generate.

`SPIDER` Name of the Spider input file.

`DEST` The directory in which the executables defined by `$(TANGLE)` and `$(WEAVE)` will be placed.

`WEBROOT` The directory that is the root of the Spidery `WEB` distribution.

`MASTER` The location of the "master sources." This should always be different from the directory in which `make` is called, or havoc will result.

`CTANGLE` The name of the program used to tangle C code.

`AWKTANGLE` The name of the program used to tangle Awk code.

`MACROS` The name of a directory in which to put TEX macro definitions (a `*web.tex` file.

Usually we will only be interested in two commands: "`make weave`" and "`make tangle`." It's safe to use "`make clean`" only if you use the current directory for nothing exception spidering; "`make clean`" is really vicious.

The line that's really of interest is the line showing the dependency for `grammar.web`. First we run Spider. Then we check for bad translation keywords and for potential cycles in the prettyprinting grammar. We check for duplicate names, and then finally (if everything else works), we put the `*web.tex` file in the right place.

Here's `$(MASTER)/WebMakefile`:

```
# Copyright 1989 by Norman Ramsey and Odyssey Research Associates.
# Not to be sold, but may be used freely for any purpose.
# For more information, see file COPYRIGHT in the parent directory.

HOME=/u/nr# # Make no longer inherits environment vars
THETANGLE=tangle
THEWEAVE=weave
SPIDER=any.spider
#
DVI=dvi
CFLAGS=-DDEBUG -g -DSTAT

# CPUTYPE is a grim hack that attempts to solve the problem of multiple
# cpus sharing a file system.  In my environment I have to have different
# copies of object and executable for vax, sun3, next, iris, and other
# cpu types.  If you will be using Spidery WEB in a homogeneous processor
# environment, you can just set CPUTYPE to a constant, or eliminate it
# entirely.
#
# In my environment, the 'cputype' program returns a string that
# describes the current processor.  That means that the easiest thing
# for you to do is to define a 'cputype' program that does something
# sensible.  A shell script that says 'echo "vax"' is fine.

CPUTYPE=‘cputype‘

# Change the following three directories to match your installation
#
# the odd placement of # is to prevent any trailing spaces from slipping in

WEBROOT=$(HOME)/web/src#  # root of the WEB source distribution
DEST=$(HOME)/bin/$(CPUTYPE)#   # place where the executables go
MACROS=$(HOME)/tex/macros#  # place where the macros go

MASTER=$(WEBROOT)/master#  # master source directory
OBDIR=$(MASTER)/$(CPUTYPE)# # common object files

TANGLESRC=tangle
CTANGLE=ceetangle -I$(MASTER)
CWEAVE=ceeweave -I$(MASTER)
AWKTANGLE=awktangle -I$(MASTER)
COMMONOBJS=$(OBDIR)/common.o $(OBDIR)/pathopen.o
COMMONC=$(MASTER)/common.c $(MASTER)/pathopen.c
COMMONSRC=$(COMMONC) $(MASTER)/spider.awk
```

```
# Our purpose is to make tangle and weave

web: tangle weave

tangle: $(COMMONOBJS) $(TANGLESRC).o
cc $(CFLAGS) -o $(DEST)/$(THETANGLE) $(COMMONOBJS) $(TANGLESRC).o

weave: $(COMMONOBJS) weave.o
cc $(CFLAGS) -o $(DEST)/$(THEWEAVE) $(COMMONOBJS) weave.o


source: $(TANGLESRC).c $(COMMONSRC) # make tangle.c and common src, then clean
if [ -f WebMakefile ]; then exit 1; fi  # don't clean the master!
if [ -f spiderman.tex ]; then exit 1; fi # don't clean the manual
-rm -f tangle.web weave.* common.* # remove links that may be obsolete
-rm -f *.unsorted *.list grammar.web outtoks.web scraps.web
-rm -f cycle.test spider.slog
-rm -f *.o *.tex *.toc *.dvi *.log *.makelog *~ *.wlog *.printlog

# Here is how we make the common stuff

$(MASTER)/common.c: $(MASTER)/common.web # no change file
$(CTANGLE) $(MASTER)/common

$(OBDIR)/common.o: $(MASTER)/common.c
cc $(CFLAGS) -c $(MASTER)/common.c
mv common.o $(OBDIR)


$(MASTER)/pathopen.c: $(MASTER)/pathopen.web # no change file
$(CTANGLE) $(MASTER)/pathopen
mv pathopen.h $(MASTER)

$(OBDIR)/pathopen.o: $(MASTER)/pathopen.c
cc $(CFLAGS) -c $(MASTER)/pathopen.c
mv pathopen.o $(OBDIR)


## Now we make the tangle and weave source locally

$(TANGLESRC).c: $(MASTER)/$(TANGLESRC).web $(MASTER)/common.h grammar.web
```

```
-/bin/rm -f $(TANGLESRC).web
ln $(MASTER)/$(TANGLESRC).web $(TANGLESRC).web
# chmod -w $(TANGLESRC).web
$(CTANGLE) $(TANGLESRC)

weave.c: $(MASTER)/weave.web $(MASTER)/common.h grammar.web
-/bin/rm -f weave.web
ln $(MASTER)/weave.web weave.web
# chmod -w weave.web
$(CTANGLE) weave

## Here's where we run SPIDER to create the source

grammar.web: $(MASTER)/cycle.awk $(MASTER)/spider.awk $(SPIDER)
echo "date" `date` | cat - $(SPIDER) | awk -f $(MASTER)/spider.awk
cat $(MASTER)/transcheck.list trans_keys.unsorted | awk -f $(MASTER)/transcheck.awk
awk -f $(MASTER)/cycle.awk < cycle.test
sort *.unsorted | awk -f $(MASTER)/nodups.awk
mv *web.tex $(MACROS)

## We might have to make spider first.

$(MASTER)/spider.awk: $(MASTER)/spider.web
$(AWKTANGLE) $(MASTER)/spider
mv cycle.awk nodups.awk transcheck.awk $(MASTER)
rm junk.list


# $(MASTER)/cycle.awk: $(MASTER)/cycle.web # making spider also makes cycle
#  $(AWKTANGLE) $(MASTER)/cycle


# This cleanup applies to every language

clean:
if [ -f WebMakefile ]; then exit 1; fi # don't clean the master!
if [ -f spiderman.tex ]; then exit 1; fi # don't clean the manual
-rm -f tangle.* weave.* common.* # remove links that may be obsolete
-rm -f *.unsorted *.list grammar.web outtoks.web scraps.web
-rm -f cycle.test spider.slog
-rm -f *.c *.o *.tex *.toc *.dvi *.log *.makelog *~ *.wlog *.printlog
```

```
# booting the new distribution
boot:
cd ../master; rm -f *.o; for i in $(COMMONC); do \
cc $(CFLAGS) -c $$i; \
mv *.o $(OBDIR) ; \
done; cd ../c
cc $(CFLAGS) -c $(TANGLESRC).c; \
cc $(CFLAGS) -o $(DEST)/$(THETANGLE) $(COMMONOBJS) $(TANGLESRC).o
```

Getting your own Spidery WEB

At this time, Spidery WEB has been tested only on Unix machines. It should be easy to port to any machine having a C compiler and an Awk interpreter, but undoubtedly some changes will be necessary. The full Spider distribution, including this manual, is available by anonymous ftp from princeton.edu: ftp/pub/spiderweb.tar.Z. You should pick a directory to install Spider in, untar the distribution, and follow the directions in the README file. The directory you have picked becomes WEBROOT.

If the Makefile in the distribution differs from the one given above, the one in the distribution should be considered the correct one.

A real Spider file

I have tried to use real examples to illustrate the use of Spider. I include here, as an extended example, the complete Spider file for the Awk language. Those who cannot easily study the distribution may find it useful to study this.

```
# Copyright 1989 by Norman Ramsey and Odyssey Research Associates.
# Not to be sold, but may be used freely for any purpose.
# For more information, see file COPYRIGHT in the parent directory.

language AWK extension awk

at_sign #

module definition stmt use stmt
# use as stmt is unavoidable since tangle introduces line breaks
```

```
comment begin <"##"> end newline
macros begin
\def\commentbegin{\#} % we don't want \#\#
macros end

line begin <"#line"> end <"">

default translation <*> mathness yes

token identifier category math mathness yes
token number category math mathness yes
token newline category newline translation <> mathness maybe
token pseudo_semi category ignore_scrap mathness no translation <opt-0>

token \ category backslash translation <> mathness maybe
token + category unorbinop
token - category unorbinop
token * category binop
token / category binop
token < category binop
token > category binop
token >> category binop translation <"\\GG">
token = category equals translation <"\\K">
token ~ category binop translation <"\\TI">
token !~ category binop translation <"\\not\\TI">
token & category binop translation <"\\amp">
token % translation <"\\%"> category binop
token ( category open
token [ category lsquare
token ) category close
token ] category close
token { translation <"\\;\\{"-indent> category lbrace
token } translation <"\\}\\"-space> category close
token , category binop translation <",\\,"-opt-3>

token ; category semi translation <";"-space-opt-2> mathness no
# stuff with semi can be empty in for statements
open semi --> open
semi semi --> semi
semi close --> close
semi --> binop

# token : category colon
# token | category bar
```

```
token != name not_eq translation <"\\I"> category binop
token <= name lt_eq translation <"\\L"> category binop
token >= name gt_eq translation <"\\G"> category binop
token == name eq_eq translation <"\\S"> category binop
token && name and_and translation <"\\W"> category binop
token || name or_or translation <"\\V"> category binop
# token -> name minus_gt translation <"\\MG"> category binop
token ++ name gt_gt category unop translation <"\\uparrow">
token -- name lt_lt category unop translation <"\\downarrow">
# preunop is for unary operators that are prefix only
token $ category preunop translation <"\\DO"> mathness yes

default mathness yes translation <*>

ilk pattern_like category math
reserved BEGIN ilk pattern_like
reserved END ilk pattern_like

ilk if_like category if
reserved if
ilk else_like category else
reserved else

ilk print_like category math
# math forces space between this and other math...
reserved print ilk print_like
reserved printf ilk print_like
reserved sprintf ilk print_like


ilk functions category unop mathness yes
reserved length ilk functions
reserved substr ilk functions
reserved index ilk functions
reserved split ilk functions
reserved sqrt ilk functions
reserved log  ilk functions
reserved exp ilk functions
reserved int ilk functions

ilk variables category math mathness yes
reserved NR ilk variables
reserved NF ilk variables
reserved FS ilk variables
```

```
reserved RS ilk variables
reserved OFS ilk variables
reserved ORS ilk variables

ilk for_like category for
reserved for ilk for_like
reserved while ilk for_like

ilk in_like category binop translation <math_bin-*-"}"> mathness yes
# translation <"\\"-space-*-"\\"-space>
reserved in ilk in_like

ilk stmt_like category math
reserved break ilk stmt_like
reserved continue ilk stmt_like
reserved next ilk stmt_like
reserved exit ilk stmt_like


backslash newline --> math
# The following line must be changed to make a backslash
backslash <"\\backslash"> --> math

math (binop|unorbinop) math --> math
<"\\buildrel"> (binop|unorbinop) <"\\over{"> equals <"}"> --> binop
equals --> binop
(unop|preunop|unorbinop) math --> math
# unorbinop can only act like unary op as *prefix*, not postfix
math unop --> math
math <"\\"-space> math --> math
# concatenation

math newline --> stmt
newline --> ignore_scrap

stmt <force> stmt --> stmt

(open|lsquare) math close --> math

math lbrace --> lbrace
lbrace <force> stmt --> lbrace
lbrace <outdent-force> close --> stmt

if <"\\"-space> math --> ifmath
```

```
ifmath lbrace --> ifbrace
ifmath newline --> ifline
ifbrace <force> stmt --> ifbrace
ifbrace <outdent-force> close else <"\\"-space> if --> if
ifbrace <outdent-force> close else lbrace --> ifbrace
ifbrace <outdent-force> close else newline --> ifline
ifbrace <outdent-force> close --> stmt
(ifline|ifmath) <indent-force> stmt <outdent-force> else <"\\"-space> if --> if
(ifline|ifmath) <indent-force> stmt <outdent-force> else lbrace --> ifbrace
(ifline|ifmath) <indent-force> stmt <outdent-force> else newline --> ifline
(ifline|ifmath) <indent-force> stmt <outdent-force> else --> ifmath
(ifline|ifmath) <indent-force> stmt <outdent> --> stmt

for <"\\"-space> math --> formath
formath lbrace --> forbrace
formath newline --> forline
forbrace <force> stmt --> forbrace
forbrace <outdent-force> close --> stmt
(forline|formath) <indent-force> stmt <outdent> --> stmt


? ignore_scrap --> #1
```

Bibliography

References

[Bentley 87] Jon L. Bentley, "Programming Pearls," *Communications of the ACM* **29:5**(May 1986), 364–?, and **29:6**(June 1986), 471–483. Two columns on literate programming. The first is an introduction, and the second is an extended example by Donald Knuth, with commentary by Douglas MacIlroy.

[Knuth 83] Donald E. Knuth, "The `WEB` system of structured documentation" Technical Report 980, Stanford Computer Science, Stanford, California, September 1983. The manual for the original `WEB`.

[Knuth 84] Donald E. Knuth, "Literate Programming," *The Computer Journal* **27:2**(1984), 97–111. The original introduction to literate programming with `WEB`.

[Levy 87] Silvio Levy, "Web Adapted to C, Another Approach" *TUGBoat* **8:2**(1987), 12–13. A short note about the C implementation of `WEB`, from which Spidery `WEB` is descended.

[Sewell 89] Wayne Sewell, "Weaving a program: Literate programming in `WEB`," Van Nostrand Reinhold, 1989.