noweb Source Code*

Norman Ramsey†
nr@eecs.harvard.edu

June 2, 2000

Preface

This isn't a real literate program—there's just enough documentation here to get the job done. One of my observations is that the cost of creating a high-quality, well-documented literate program adds 1–3 times the amount of effort it took to create the program in the first place. Most programs I write aren't worth that kind of effort, and noweb is no exception. Don't expect too much.

Contents

† Section 3.2 is the generous contribution of Preston Briggs

1 Converting a `noweb` file to tool markup

1.1 Main program

This is the initial filter in the pipeline. Syntax from `markup.ow`. Output may contain

```
@[begin|end] [code|docs] nnn ...
@text <line of text>
@quote
@endquote
@defn <module name>
@use <module name>
@nl
@file <filename>
@line nnn
@index [defn|use] <indexed string>
@index nl
@literal <literal text for weave or other tool>
```

Here's the code that writes the output. We have one function to print the state, which takes care of `@begin` and `@end` and remembering what kind of chunk we have. Another function prints `@name value` pairs. If the `value` has a trailing newline, the pair is followed by `@nl`.

If the only way index entries were created was by tools, life would be easy. Permitting index definitions in the input causes two complications. First, it introduces newlines that don't belong to any chunk, but that must be counted. Second, it introduces a special state, `CodeIndex` in which a code chunk is about to be terminated but won't be until there are no more index definitions.

2 ⟨*state and printing definitions* 2⟩≡ (4)

```
typedef enum state {Code=1, Docs=2, CodeIndex=3} State;
typedef enum mark {Begin=1, End=2} Mark;
typedef enum index {Defn=1, Use=2, Newline=3} Index;

static char *states[]  = { "bad state", "code", "docs", "code" };
static char *marks[]   = { "bad mark", "begin", "end" };
static char *indices[] = { "bad index", "defn", "use", "nl" };
static char low_at_sign = '@';

static void print_state(FILE *out, Mark be, State state, int count) {
    fprintf(out, "%c%s %s %d\n", low_at_sign, marks[be], states[state], count);
}

static void print_index(FILE *out, Index idx, char *arg) {
    if (arg)
        fprintf(out, "%cindex %s %s\n", low_at_sign, indices[idx], arg);
    else
        fprintf(out, "%cindex %s\n",    low_at_sign, indices[idx]);
}

static void print_pair(FILE *out, char *name, char *value) {
    if (value) {
        int last=strlen(value)-1;
        if (last>=0 && value[last]=='\n')
            fprintf(out, "%c%s %s%cnl\n", low_at_sign, name, value, low_at_sign);
        else
            fprintf(out, "%c%s %s\n", low_at_sign, name, value);
    } else
        fprintf(out, "%c%s\n", low_at_sign, name);
}
```

Defines:
- `index`, used in chunks 28, 32c, and 34d.
- `mark`, used in chunks 4 and 19–22.
- `print_index`, used in chunk 15a.
- `print_pair`, used in chunks 5, 6b, 10f, 12a, and 14.
- `print_state`, used in chunk 5.
- `state`, used in chunks 4, 5, 9–13, 57, 58a, and 60–62.

Uses `strlen` 55c.

4 ⟨* 4⟩≡ 16a ▷

```
static char rcsid[] = "$Id: markmain.nw,v 2.15 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#include "errors.h"
#include "markup.h"
#include "getline.h"
#include "columns.h"

⟨state and printing definitions 2⟩

void markup (FILE* in, FILE *out, char *filename) {
    State state = Docs;         /* what we are reading */
    int quoting = 0;            /* currently quoting code? */
    int count = 0;              /* number of current chunk, from 0 */
    int missing_newline;        /* was last line missing a trailing \n? */
    int lineno = 0;             /* number of lines read */
    int last_open_quote;        /* line number of last unmatched open quote */

    char *line;                 /* part of line up to mark (or end) */
    char *mark;                 /* potentially marked part of line */
#define MAX_MODNAME 255
    char modname[MAX_MODNAME+1] = ""; /* name of module currently being read,
                                         [[""]] if no module is being read */

    ⟨copy all lines from in to out, adding markup 5⟩
}
```

Defines:
`markup`, used in chunks 16a, 17, and 21c.

Uses `getline` 65a, `mark` 2, `module` 38b, and `state` 2.

A *noweb* file consists of a sequence of *chunks,* which may appear in any order. *noweb* supports two kinds of chunks: documentation chunks and code chunks. Documentation chunks begin with an at sign (@) at the beginning of a line, followed by a space (or with an at sign on a line by itself). They have no names. Code chunks begin with

`<<chunk name>>=`

on a line by itself. The left double angle bracket (`<<`) must be in the first column. Chunks are terminated by the beginning of another chunk, or by end of file. If the first line in the file does not signal the beginning of a documentation or code chunk, the first chunk is assumed to be a documentation chunk.

We implement this by just looping through the document, looking for the beginnings of chunks. We use `state` to keep track of what's ending. We add a little extra markup for file name and module names Starting code and starting documentation are notably different: code is started by a module definition, which must appear on a line by itself. Documentation is started by a character sequence, and arbitrary documentation text can follow on the same line.

5 ⟨*copy all lines from* `in` *to* `out`*, adding markup* 5⟩≡ (4)

```
print_pair(out, "file", filename);
print_state(out, Begin, state, count);
while ((line = getline_expand(in)) != NULL) {
    lineno++;
    ⟨remember if a newline was missing from the last line 6a⟩
    if (starts_code(line, filename, lineno)) {
        ⟨complain and correct unmatched open quote, if any 15b⟩
        print_state(out, End, state, count);
        count++;
        state = Code;
        print_state(out, Begin, state, count);
        getmodname(modname,MAX_MODNAME,line);
        print_pair(out,"defn",modname);
        print_pair(out,"nl",0);     /* could be implicit but this is better */
    } else if (is_def(line)) {
        ⟨emit index definitions and index newline 15a⟩
        if (state == Code)
            state = CodeIndex;
    } else {
        if (starts_doc(line) || state == CodeIndex) {
            ⟨complain and correct unmatched open quote, if any 15b⟩
            print_state(out, End, state, count);
            count++;
            state = Docs;       /* always reading docs after a stop */
            print_state(out, Begin, state, count);
            if (starts_doc(line))
                line = first_doc_line(line);
        }
```

```
        {   ⟨declarations for conversion engine 6c⟩
            ⟨macro definitions for conversion engine 8a⟩
            ⟨initialize conversion engine 6d⟩
            ⟨if the line begins with two ESCs, append one 7a⟩
            if (state == Code || quoting)
                goto convert_code;
            else
                goto convert_docs;
            ⟨conversion state machine 9a⟩
          done_converting:
            (void)0;
        }
    }
}
⟨complain and correct unmatched open quote, if any 15b⟩
⟨add a trailing newline if one was missing 6b⟩
print_state(out, End, state, count);
```

Uses `first_doc_line` 18d, `getline_expand` 65b, `getmodname` 20c, `is_def` 19c, `print_pair` 2, `print_state` 2, `starts_code` 20c, `starts_doc` 18d, and `state` 2.

We make it safe to assume henceforth that every chunk ends with newline, even if the file didn't end with newline. If the file didn't end with newline, we add a newline to the last line before printing the final `@end`.

6a ⟨*remember if a newline was missing from the last line* 6a⟩≡ (5)

```
missing_newline = line[strlen(line)-1] != '\n';
```

Uses `strlen` 55c.

6b ⟨*add a trailing newline if one was missing* 6b⟩≡ (5)

```
if (missing_newline) print_pair(out, "nl",0);
```

Uses `print_pair` 2.

The conversion engine copies the input `line` to the output `buf`, using the pointers `s` on the left for the destination and `t` on the right for the source. `c` holds the current input character, and `p` points to the current piece of text about to be emitted.

6c ⟨*declarations for conversion engine* 6c⟩≡ (5)

```
char *p, *s, *t, c;
static char *buf;
static int buflen = 0;
```

6d ⟨*initialize conversion engine* 6d⟩≡ (5)

```
⟨make buf big enough to hold all of line, plus two more characters 14c⟩
p = s = buf+2;
t = line;
c = *t++;
```

Using `@@` in the first column to escape an at sign was added after the state machine was designed, in response to a plea from Dirk Verworner:

> We at the Humboldt-University of Berlin use your `noweb` programs for our project. We've a simple description language, which sometimes an at sign on a line by itself requires. So I've a problem.

It might be that I ought to add a special beginning-of-line state to the state machine, so that we could recognize with there, but I'd have to replicate it for both code and documentation. Because I only pay the cost once per input line, I don't mind using the following Band-Aid:

7a ⟨*if the line begins with two* `ESC`*s, append one* 7a⟩≡ (5)
```
if (c == ESC && *t == ESC) { *s++ = ESC; c = *++t; ++t; }
```

To convert documentation lines, we have to track the quoting brackets. Within quoting brackets, we use the same engine as for within code lines (except for our treatment of square brackets). We have the state machine from hell. We can have ordinary text, quoted text in `[[...]]`, or a use in `[[`⟨*...* (never defined)⟩`]]`. Ordinary text and quoted text can span lines, but uses can't. So, the four main states of our lexer are:

`t`	Ordinary text
`c`	Code text
`u`	A use
`uc`	Quoted code within a use

That last state covers the case where we have `[[`⟨*........* (never defined)⟩`]]`. BTW, it's no coincidence that if we've just consumed a newline (i.e., we're at the start of a line), we must be in one of those four states. Moreover, since a newline in states `u` and `uc` means it's not really a use[1], those states fall back to state `c` (while prepending the opening `<<` that appeared to signal a use). So `c` and `t` are the major states. When scanning documentation, they correspond to `quoting` and `!quoting`.

By the way, one reason for writing a state machine by hand instead of using a lexer generator is so I can test the machine using the wonderful χATAC tool developed by Bob Horgan, Saul London, and Hira Agrawal. For more information send email to `jrh@bellcore.com`.

We don't have to treat a newline specially, because `print_pair` inserts `@nl` if needed.

7b ⟨*emit text and goto* `done_converting` *if at end of line* 7b⟩≡ (9a 11a)
```
if (c == '\0') {
  ⟨emit p to s as text 14d⟩
  goto done_converting;
}
```

[1] Because uses may not span lines.

Markup is defined as follows:

`LA1 LA2`	open a chunk
`RA1 RA2`	close a chunk
`LS1 LS2`	open a quote
`RS1 RS2`	close a quote
`ESC`	quotes the sequences above

By default we use angles for chunks and squares for quotes, at sign for escape:

8a ⟨*macro definitions for conversion engine* 8a⟩≡ (5) 8b ▷

```
#define LA1 '<'
#define LA2 '<'
#define RA1 '>'
#define RA2 '>'
#define LS1 '['
#define LS2 '['
#define RS1 ']'
#define RS2 ']'
#define ESC '@'
```

Usually when we change state we advance the input pointer. Sometimes we also copy the current character.

8b ⟨*macro definitions for conversion engine* 8a⟩+≡ (5) ◁ 8a 8c ▷

```
#define next(L) do {           c = *t++; goto L; } while (0)
#define copy(L) do { *s++ = c; c = *t++; goto L; } while (0)
```

Normally we don't check assertions in the lexer. To check them, we would `#define lexassert assert`.

8c ⟨*macro definitions for conversion engine* 8a⟩+≡ (5) ◁ 8b

```
#define lexassert(E) ((void)0)
```

From here on I intercalate the code that implements the state machine with the dot code that draws a picture of it. The picture can be found in file `markup.ps`; it's too big to be useful as part of this document. The arcs in the picture are labelled with *char/output*, but the output may be omitted. In the code, we see `goto` when the picture has a `-` sign (denoting character not consumed), `next` when we're emitting some literal, and `copy` when we emit `c` (the next character).

8d ⟨*dot arcs* 8d⟩≡ (14a) 9b ▷

```
t -> at [label="@"]
t -> la [label="<"]
t -> ls [label="["]
t -> t  [label="c/c"]
```

9a ⟨*conversion state machine* 9a⟩≡ (5) 9c ▷
```
  convert_docs:
  t: lexassert(state == Docs && !quoting);
     if (c == ESC) next(at);
     if (c == LA1) next(la);
     if (c == LS1) next(ls);
     ⟨emit text and goto done_converting if at end of line 7b⟩
     copy(t);
```
Uses `state` 2.

9b ⟨*dot arcs* 8d⟩+≡ (14a) ◁ 8d 9d ▷
```
  at -> atls [label="["]
  at -> atla [label="<"]
  at -> atrs [label="]"]
  at -> atra [label=">"]
  at -> t    [label="c-/@"]
```

9c ⟨*conversion state machine* 9a⟩+≡ (5) ◁ 9a 9e ▷
```
  at: if (c == LA1) next(atla);
      if (c == LS1) next(atls);
      if (c == RA1) next(atra);
      if (c == RS1) next(atrs);
      *s++ = ESC; goto t;
```

9d ⟨*dot arcs* 8d⟩+≡ (14a) ◁ 9b 9f ▷
```
  atls -> t [label="[/[["]
  atls -> t [label="c-/@["]
```

9e ⟨*conversion state machine* 9a⟩+≡ (5) ◁ 9c 9g ▷
```
  atls: if (c == LS2) { *s++ = LS1; *s++ = LS2; next(t); }
        *s++ = ESC; *s++ = LS1; goto t;
```

9f ⟨*dot arcs* 8d⟩+≡ (14a) ◁ 9d 9h ▷
```
  atla -> t [label="</<<"]
  atla -> t [label="c-/@<"]
```

9g ⟨*conversion state machine* 9a⟩+≡ (5) ◁ 9e 9i ▷
```
  atla: if (c == LA2) { *s++ = LA1; *s++ = LA2; next(t); }
         *s++ = ESC; *s++ = LA1; goto t;
```

9h ⟨*dot arcs* 8d⟩+≡ (14a) ◁ 9f 10a ▷
```
  atrs -> t [label="]/]]"]
  atrs -> t [label="c-/@]"]
```

9i ⟨*conversion state machine* 9a⟩+≡ (5) ◁ 9g 10b ▷
```
  atrs: if (c == RS2) { *s++ = RS1; *s++ = RS2; next(t); }
        *s++ = ESC; *s++ = RS1; goto t;
```

10a ⟨*dot arcs* 8d⟩+≡ (14a) ◁9h 10c▷
```
atra -> t [label=">/>>"]
atra -> t [label="c-/@>"]
```

10b ⟨*conversion state machine* 9a⟩+≡ (5) ◁9i 10d▷
```
atra: if (c == RA2) { *s++ = RA1; *s++ = RA2; next(t); }
      *s++ = ESC; *s++ = RA1; goto t;
```

10c ⟨*dot arcs* 8d⟩+≡ (14a) ◁10a 10e▷
```
la  -> t [label="</<< (error)"]
la  -> t [label="c-/<"]
```

10d ⟨*conversion state machine* 9a⟩+≡ (5) ◁10b 10f▷
```
la: if (c == LA2) {
      ⟨complain of unescaped @¡¡ in documentation 14f⟩
      *s++ = LA1; *s++ = LA2; next(t);
    }
    *s++ = LA1; goto t;
```

When we've seen [[, we don't yet know whether it's the start of a quote or the start of a use—but we can emit the text so far.

10e ⟨*dot arcs* 8d⟩+≡ (14a) ◁10c 10g▷
```
ls  -> c [label="[ (emit text; quote)"]
ls  -> t [label="c-/["]
```

10f ⟨*conversion state machine* 9a⟩+≡ (5) ◁10d 11a▷
```
ls: lexassert(state == Docs);
    if (c == LS2) {
      ⟨emit p to s as text 14d⟩
      quoting = 1; last_open_quote = lineno; print_pair(out, "quote", 0);
      next(c);
    }
    *s++ = LS1;
    goto t;
```
Uses `print_pair` 2 and `state` 2.

Here's the ordinary quote state.

10g ⟨*dot arcs* 8d⟩+≡ (14a) ◁10e 11b▷
```
/*{ rank=same cat catla catra }*/
c -> crs [label="] (docs only)"]
c -> cla [label="<"]
c -> cat [label="@"]
c -> c   [label="c/c"]
```

11a ⟨*conversion state machine* 9a⟩+≡ (5) ◁10f 11c▷
```
convert_code:
c: lexassert(state == Code || quoting);
   if (c == RS1 && quoting) next(crs);
   if (c == LA1) next(cla);
   if (c == ESC) next(cat);
   ⟨emit text and goto done_converting if at end of line 7b⟩
   copy(c);
```
Uses `state` 2.

In code, quoting square brackets is neither necessary nor permitted.

11b ⟨*dot arcs* 8d⟩+≡ (14a) ◁10g 11d▷
```
cat -> catla [label="<"]
cat -> catra [label=">"]
cat -> c     [label="c-/@"]
```

11c ⟨*conversion state machine* 9a⟩+≡ (5) ◁11a 11e▷
```
cat: if (c == LA1) next(catla);
     if (c == RA1) next(catra);
     *s++ = ESC; goto c;
```

11d ⟨*dot arcs* 8d⟩+≡ (14a) ◁11b 11f▷
```
catla -> c [label="</<<"]
catla -> c [label="c-/@<"]
```

11e ⟨*conversion state machine* 9a⟩+≡ (5) ◁11c 11g▷
```
catla: if (c == LA2) { *s++ = LA1; *s++ = LA2; next(c); }
       *s++ = ESC; *s++ = LA1; goto c;
```

11f ⟨*dot arcs* 8d⟩+≡ (14a) ◁11d 11h▷
```
catra -> c [label=">/>>"]
catra -> c [label="c-/@>"]
```

11g ⟨*conversion state machine* 9a⟩+≡ (5) ◁11e 11i▷
```
catra: if (c == RA2) { *s++ = RA1; *s++ = RA2; next(c); }
       *s++ = ESC; *s++ = RA1; goto c;
```

We can't end the quote until we've consumed all the closing `]`'s.

11h ⟨*dot arcs* 8d⟩+≡ (14a) ◁11f 11j▷
```
crs -> ce [label="]"]
crs -> c [label="c-/]"]
```

11i ⟨*conversion state machine* 9a⟩+≡ (5) ◁11g 12a▷
```
crs: if (c == RS2) next(ce);
     *s++ = RS1; goto c;
```

11j ⟨*dot arcs* 8d⟩+≡ (14a) ◁11h 12b▷
```
ce -> ce [label="]/]"]
ce -> t  [label="c- (emit text; endquote)"]
```

12a ⟨*conversion state machine* 9a⟩+≡ (5) ◁11i 12c▷
```
ce: lexassert(quoting);
    if (c == RS2) copy(ce);
    quoting = 0; ⟨emit p to s as text 14d⟩ print_pair(out, "endquote", 0);
    goto t;
```
Uses `print_pair` 2.

Now's here where we might be seeing a use.

12b ⟨*dot arcs* 8d⟩+≡ (14a) ◁11j 12d▷
```
cla -> u [label="< (emit text)"]
cla -> c [label="c-/<"]
```

12c ⟨*conversion state machine* 9a⟩+≡ (5) ◁12a 12e▷
```
cla: if (c == LA2) { ⟨emit p to s as text 14d⟩ next(u); }
     *s++ = LA1; goto c;
```

When processing a use, we make sure `[[...]]` hide any closing angle brackets.

12d ⟨*dot arcs* 8d⟩+≡ (14a) ◁12b 12f▷
```
u -> uls [label="[/["]
u -> urs [label="] (docs only)"]
u -> ura [label=">"]
u -> c   [label="eol- (prepend <<)"]
u -> u   [label="c/c"]
```

12e ⟨*conversion state machine* 9a⟩+≡ (5) ◁12c 12g▷
```
u: lexassert(state == Code || quoting);
   if (c == LS1) copy(uls);
   if (c == RS1 && quoting) next(urs);
   if (c == RA1) next(ura);
   if (c == '\0') { /* premature end --- it's not a use after all */
     ⟨prepend opening angles to p 14b⟩
     goto c;
   }
   copy(u);
```
Uses `state` 2.

12f ⟨*dot arcs* 8d⟩+≡ (14a) ◁12d 13a▷
```
urs -> ce [label="] (not use; prepend <<)"]
urs -> u  [label="c-/]"]
```

12g ⟨*conversion state machine* 9a⟩+≡ (5) ◁12e 13b▷
```
urs: lexassert(quoting);
     if (c == RS2) { /* premature end --- it's not a use after all */
       ⟨prepend opening angles to p 14b⟩
       next(ce);
     }
     *s++ = RS1; goto u;
```

13a ⟨*dot arcs* 8d⟩+≡ (14a) ◁12f 13c▷
```
uls -> uc [label="[/["]
uls -> u  [label="c-"]
```

13b ⟨*conversion state machine* 9a⟩+≡ (5) ◁12g 13d▷
```
uls: if (c == LS2) copy(uc);
     goto u;
```

13c ⟨*dot arcs* 8d⟩+≡ (14a) ◁13a 13e▷
```
uc -> ucrs [label="]/]"]
uc -> c    [label="eol- (prepend <<)"]
uc -> uc   [label="c/c"]
```

13d ⟨*conversion state machine* 9a⟩+≡ (5) ◁13b 13f▷
```
uc: lexassert(quoting || state == Code);
    if (c == RS1) copy(ucrs);
    if (c == '\0') {
      ⟨prepend opening angles to p 14b⟩
      goto c;
    }
    copy(uc);
```
Uses `state` 2.

13e ⟨*dot arcs* 8d⟩+≡ (14a) ◁13c 13g▷
```
ucrs -> uce [label="]/]"]
ucrs -> uc [label="c-"]
```

13f ⟨*conversion state machine* 9a⟩+≡ (5) ◁13d 13h▷
```
ucrs: if (c == RS2) copy(uce);
      goto uc;
```

13g ⟨*dot arcs* 8d⟩+≡ (14a) ◁13e 13i▷
```
uce -> uce [label="]/]"]
uce -> u  [label="c-"]
```

13h ⟨*conversion state machine* 9a⟩+≡ (5) ◁13f 13j▷
```
uce: if (c == RS2) copy(uce);
     goto u;
```

13i ⟨*dot arcs* 8d⟩+≡ (14a) ◁13g
```
ura -> c [label="> (emit use)"]
ura -> u [label="c-/>"]
```

13j ⟨*conversion state machine* 9a⟩+≡ (5) ◁13h
```
ura: if (c == RA2) { ⟨emit p to s as use 14e⟩ next(c); }
     *s++ = RA1; goto u;
```

14a ⟨*markup.dot* 14a⟩≡
```
digraph lexer {
  rotate=90
  size="10,7.5"
  ratio=fill
  rankdir=LR
  node [shape=circle]
  ⟨dot arcs 8d⟩
}
```

14b ⟨*prepend opening angles to* `p` 14b⟩≡ (12 13d)
```
lexassert(p == buf + 2);
p -= 2;
p[0] = LA1;
p[1] = LA2;
```

14c ⟨*make* `buf` *big enough to hold all of* `line`*, plus two more characters* 14c⟩≡ (6d)
```
if (buf == NULL)
  checkptr(buf = (char *) malloc(buflen = 128));
if (buflen < strlen(line) + 1 + 2) {
  while (buflen < strlen(line) + 1 + 2)
    buflen *= 2;
  checkptr(buf = (char *) realloc(buf, buflen));
}
```
Uses `checkptr` 70a, `malloc` 55d, and `strlen` 55c.

14d ⟨*emit* `p` *to* `s` *as text* 14d⟩≡ (7b 10f 12)
```
if (s > p) {
  *s = 0;
  print_pair(out, "text", p);
}
s = p = buf + 2;
```
Uses `print_pair` 2.

emit the empty use regardless

14e ⟨*emit* `p` *to* `s` *as use* 14e⟩≡ (13j)
```
*s = 0;
print_pair(out, "use", p);
s = p = buf + 2;
```
Uses `print_pair` 2.

14f ⟨*complain of unescaped @¡¡ in documentation* 14f⟩≡ (10d)
```
errorat(filename, lineno, Error, "unescaped << in documentation chunk");
```
Uses `errorat` 70d.

15a ⟨*emit index definitions and index newline* 15a⟩≡ (5)
```
line = remove_def_marker(line);
while (*line && isspace(*line)) line++;
while (*line) {
  char tmp;
  char *s = line+1;
  while (*s && !isspace(*s)) s++;
  tmp = *s; *s = 0;
  print_index(out, Defn, line);
  *s = tmp;
  while (isspace(*s)) s++;
  line = s;
}
print_index(out, Newline, 0);
```
Uses `print_index` 2 and `remove_def_marker` 19c.

15b ⟨*complain and correct unmatched open quote, if any* 15b⟩≡ (5)
```
if (quoting) {
    errorat(filename, last_open_quote, Warning, "open quote `[[' never closed");
    quoting = 0;
}
```
Uses `errorat` 70d.

Finally, we give the main program, which opens each file and passes it to `markup`. Like many main programs, it's based on K & R `cat`.

16a ⟨* 4⟩+≡ ◃4 17▹

```
main(int argc, char **argv) {
    FILE *fp;
    int i;

    progname = argv[0];
    for (i = 1; i < argc && argv[i][0] == '-' && argv[i][1] != 0; i++)
        switch(argv[i][1]) {
            case 't': ⟨handle -t 16b⟩ break;
            default : errormsg(Error, "%s: unknown option -%c", progname, argv[i][1]);
                      break;
        }
    if (i < argc)
        for (; i < argc; i++) {
            if (!strcmp(argv[i], "-")) {
                markup(stdin,stdout,"");
            } else
                if ((fp=fopen(argv[i],"r"))==NULL)
                    errormsg(Error, "%s: couldn't open file %s", progname, argv[i]);
                else {
                    markup(fp,stdout,argv[i]);
                    fclose(fp);
                }
        }
    else
        markup(stdin,stdout,"");
    nowebexit(NULL);
    return errorlevel;          /* slay warning */
}
```

Uses `errorlevel` 69b, `errormsg` 70a 70b, and `markup` 4.

16b ⟨*handle* -t 16b⟩≡ (16a)

```
if (isdigit(argv[i][2]))
    tabsize = atoi(argv[i]+2);
else if (argv[i][2]==0)
    tabsize = 0;                /* no tabs */
else
    errormsg(Error, "%s: ill-formed option %s", progname, argv[i]);
```

Uses `errormsg` 70a 70b and `tabsize` 67a.

1.2 Identifying the special `noweb` control sequences

This file is supposed to contain all the definitions of how different things are distinguished in the source text.

Documentation is introduced only by an at sign (@) on a line by itself (optionally followed by spaces). It continues up to new documentation or a module definition. A module definition is a module name, followed by one equals sign (=), possibly followed by white space, on a line by itself. A module name is any test enclosed in double angle brackets (`<<...>>`). The "unnamed" module is called * (that is, `<<*>>`).

Double angle brackets may be escaped in source by preceding them with the at sign. No other character, not even the at sign, needs to be escaped. This makes for more readable source, but makes it impossible to put an at sign immediately before a module name. If we liked we could try to hack this by creating a predefined module that expands to an at sign, but I won't bother.

17 ⟨*4⟩+≡ ◁16a 23▷

```
static char rcsid[] = "$Id: markup.nw,v 2.10 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#include "markup.h"
#include "strsave.h"
#include "errors.h"

⟨data definitions 18a⟩

⟨C functions 18d⟩
```

Uses `markup` 4.

To quote from the man page:

> A *noweb* file consists of a sequence of *chunks,* which may appear in any order. *noweb* supports two kinds of chunks: documentation chunks and code chunks. Documentation chunks begin with an at sign (@) at the beginning of a line, followed by a space (or with an at sign on a line by itself). They have no names. Code chunks begin with
>
> `<<`*chunk name*`>>=`
>
> on a line by itself. The left double angle bracket (`<<`) must be in the first column. Chunks are terminated by the beginning of another chunk, or by end of file. If the first line in the file does not signal the beginning of a documentation or code chunk, the first chunk is assumed to be a documentation chunk.

18a ⟨*data definitions* 18a⟩≡ (17) 19a ▷

```
char at_sign = '@';             /* should be the only place '@' is mentioned */
```

Defines:
`at_sign`, used in chunks 18 and 19b.

18b ⟨*header* 18b⟩≡ (55a) 18c ▷

```
extern char at_sign;                            /* the at sign */
```

Uses `at_sign` 18a.

Here's how we recognize the start of a documentation chunk, and extract the first line of documentation from what follows the @ sign.

18c ⟨*header* 18b⟩+≡ (55a) ◁ 18b 18e ▷

```
int starts_doc(char *);
char *first_doc_line(char *);
```

Uses `first_doc_line` 18d and `starts_doc` 18d.

18d ⟨*C functions* 18d⟩≡ (17 39b 64a 69a) 19b ▷

```
int starts_doc(char *line) {
    return (*line==at_sign && (line[1]=='\0' || isspace(line[1])));
}

char *first_doc_line(char *line) {
    if (line[1]!='\0' && line[1] !='\n') return line+2;
    else return line+1;
}
```

Defines:
`first_doc_line`, used in chunks 5 and 18c.
`starts_doc`, used in chunks 5 and 18c.
Uses `at_sign` 18a.

An index line is a special case of starting documentation.

18e ⟨*header* 18b⟩+≡ (55a) ◁ 18c 19d ▷

```
int is_def(char *);
char *remove_def_marker(char *);
```

Uses `is_def` 19c and `remove_def_marker` 19c.

19a ⟨*data definitions* 18a⟩+≡ (17) ◁18a
```
static char def_marker[] = " %def ";
#define def_length (6)
```

19b ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁18d 20a▷
```
int is_def(char *line) {
    ⟨first time through, assert we got the length right 19c⟩
    return (*line==at_sign && !strncmp(line+1, def_marker, def_length));
}

char *remove_def_marker(char *line) {
    return line+1+def_length;
}
```
Uses `at_sign` 18a, `is_def` 19c, and `remove_def_marker` 19c.

19c ⟨*first time through, assert we got the length right* 19c⟩≡ (19b)
```
{ static int checked = 0;
  if (!checked) {
    assert(strlen(def_marker) == def_length);
    checked = 1;
  }
}
```
Defines:
`is_def`, used in chunks 5, 18e, and 19b.
`remove_def_marker`, used in chunks 15a, 18e, and 19b.
Uses `strlen` 55c.

Recognizing module names (which introduce code chunks) and extracting them is the mission of the `mod_start` and `mod_end` functions.

The basic idea is to find an instance of `<<`, while skipping every instance of `@<<`. That's the start of a module name. We find the end the same way. A function called `find_escaped` does the dirty work. If it succeeds, it returns a pointer to the first character after the special sequence, in this case the first character of the module name. (If it fails, it returns a null pointer.) If its `mark` paramter is set, it will put a `'\0'` in place of the special sequence, splitting the string into pieces. These two properties make it perfect for dissecting lines with module names.

`mod_start` and `mod_end` just store the special sequences.

19d ⟨*header* 18b⟩+≡ (55a) ◁18e 20b▷
```
char *mod_start (char *s, int mark);    /* find the first module name */
char *mod_end (char *s, int mark);              /* find the end of module name */
```
Uses `mark` 2, `mod_end` 20a, `mod_start` 20a, and `module` 38b.

20a ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁19b 20c▷
```
char *mod_start(char *s, int mark) {
    return find_escaped(s,"<<","@<<", mark);
}
char *mod_end(char *s, int mark) {
    return find_escaped(s,">>","@>>", mark);
}
```
Defines:
`mod_end`, used in chunks 19d and 20c.
`mod_start`, used in chunks 19d and 20c.
Uses `find_escaped` 22 and `mark` 2.

Once we know how to `mod_start` and `mod_end`, recognizing the beginning of a code chunk is easy.

20b ⟨*header* 18b⟩+≡ (55a) ◁19d 21c▷
```
int starts_code (char *line, char *filename, int lineno);
                                        /* does this line start module defn? */
void getmodname(char *dest, int size, char *source);
                                        /* extract module name and put in dest */
```
Uses `getmodname` 20c, `module` 38b, and `starts_code` 20c.

20c ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁20a 22▷
```
int starts_code (char *line, char *filename, int lineno) {
    char *tail;
    if (mod_start(line,0) != line+2) return 0;
    tail = mod_end(line+2,0);
    if (tail == NULL) ⟨error exit from starts_code 21a⟩
    if (*tail++ != '=') return 0;
    while (isspace(*tail)) tail++;
    return (*tail == '\0');
}

void getmodname(char *dest, int size, char *source) {
    /* excess characters in the module name are ignored */
    char *p = strsave(source);
    char *q = mod_start(p,1);

    if (q==NULL) ⟨Error exit from getmodname 21b⟩
    if (mod_end(q,1)==NULL) ⟨Error exit from getmodname 21b⟩
    strncpy(dest,q,size-1);
    dest[size-1] = '\0';
    free(p);
}
```
Defines:
`getmodname`, used in chunks 5 and 20b.
`starts_code`, used in chunks 5 and 20b.
Uses `free` 55d, `mod_end` 20a, `mod_start` 20a, and `module` 38b.

21a ⟨*error exit from* `starts_code` 21a⟩≡ (20c)
```
{
    errorat(filename, lineno, Error, "Module name doesn't end", line);
    return 0;
}
```
Uses `errorat` 70d and `Module` 38b.

21b ⟨*Error exit from getmodname* 21b⟩≡ (20c)
```
{
    free(p);
    impossible
        ("I couldn't manage to extract a module name, but I'm sure I saw one");
}
```
Uses `free` 55d, `impossible` 70a, and `module` 38b.

Here we do the magic `find_escaped`. Find an arbitrary character seqence, skipping an escaped sequence (if any), replacing the beginning with `'\0'` if `mark` is set.

If no `search` sequence in `s`, returns `NULL`. Otherwise, returns a pointer to the first character after the `search` sequence, and sets the first character to `'\0'`, so that `s` will point to the part of the string before the `search` sequence. The `escape` sequence (if any) is always skipped, even if the `search` sequence is a substring of it.

`escape` will be ignored if it is either the `NULL` pointer or the null string (`""`).

21c ⟨*header* 18b⟩+≡ (55a) ◁20b 26a▷
```
char *find_escaped(register char *s, char *search, char *escape, int mark);
                /* find escaped strings. See markup.nw for details */
```
Uses `find_escaped` 22, `mark` 2, and `markup` 4.

22 ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁20c 39a▷

```
char *find_escaped(register char *s, char *search, char *escape, int mark) {
    register char first = *search;
    register char first_escape = (escape != NULL ? *escape : '\0');
    int searchlen = strlen(search);
    int escapelen = (escape != NULL ? strlen(escape) : 0);

    do {
        while (*s && *s != first && *s != first_escape) s++;
        if (!*s) break;
        if (first_escape && !strncmp(s,escape,escapelen)) {
                s += escapelen;
                continue;
        }
        if (!strncmp(s,search,searchlen)) break;
        s++;
    } while (*s != '\0');
    /* here either s is empty or it points to the first unescaped [[search]] */
    if (*s == '\0') return NULL;
    if (mark) *s = '\0';
    return s+searchlen;
}
```

Defines:
`find_escaped`, used in chunks 20a and 21c.
Uses `mark` 2 and `strlen` 55c.

2 Tangling

2.1 Main program

23 ⟨* 4⟩+≡ ◁17 24a▷

```
static char rcsid[] = "$Id: main.nw,v 2.11 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "notangle.h"
#include "errors.h"
#include "columns.h"
#include "modules.h"
#include "modtrees.h"
```

Uses `notangle` 26c.

The main program makes one pass through the arguments, using options to set global state, and counting any roots it sees. It then makes a second pass to emit all the roots, or if there are no roots on the command line, it emits the module (chunk) named ⟨* 4⟩.

24a ⟨* 4⟩+≡ ◁23 26b▷

```
main(int argc, char **argv) {
    int i;
    char *locformat = "";
    char *Clocformat = "#line %L \"%F\"%N";
    int root_options_seen = 0;

    tabsize = 0;  /* default for nt is not to use tabs */
    progname = argv[0];
    finalstage = 1;

    for (i=1; i<argc; i++) {
        ⟨process argument i—first pass 24b⟩
    }

    read_defs(stdin);                        /* read all the definitions */
    apply_each_module(remove_final_newline); /* pretty up the module texts */
    if (root_options_seen == 0)
      emit_module_named(stdout, "*", locformat);
    else
      for (i=1; i<argc; i++)
        if (argv[i][0] == '-' && argv[i][1] == 'R')
          emit_module_named(stdout, argv[i]+2, locformat);

    nowebexit(NULL);
    return errorlevel;  /* slay warning */
}
```

Uses `apply_each_module` 49b, `errorlevel` 69b, `module` 38b, `read_defs` 28, `remove_final_newline` 46, and `tabsize` 67a.

24b ⟨*process argument* i—*first pass* 24b⟩≡ (24a)

```
if (*argv[i]=='-') {
    ⟨handle option in argument i 25a⟩
} else {
    ⟨handle argument in argument i 25b⟩
}
```

The following options are good on the command line:

`-Rname`	make name the name of the root module. Default is `<<*>>`.
`-L[format]`	Use the given format to write `#line` numbers. If -L is given alone, use ANSI C format. If not given, write no `#line` numbers.
`-t[tabsize]`	Fiddle with tabs

25a ⟨*handle option in argument* `i` 25a⟩≡ (24b)

```
switch (argv[i][1]) {
    case 't': /* set tab size or turn off */
        if (isdigit(argv[i][2]))
            tabsize = atoi(argv[i]+2);
        else if (argv[i][2]==0)
            tabsize = 0;            /* no tabs */
        else
            errormsg(Error, "%s: ill-formed option %s\n", argv[0], argv[i]);
        break;
    case 'R': /* change name of root module */
        root_options_seen++;
        break;
    case 'L': /* have a #line number format */
        locformat = argv[i]+2;
        if (!*locformat) locformat = Clocformat;
        break;
    default:
        errormsg(Warning, "Ignoring unknown option -%s", argv[i]);
 }
```

Uses `errormsg` 70a 70b, `module` 38b, `root` 57c, and `tabsize` 67a.

25b ⟨*handle argument in argument* `i` 25b⟩≡ (24b)

```
errormsg(Warning,
    "I can't handle arguments yet, so I'll just ignore `%s'",argv[i]);
```

Uses `errormsg` 70a 70b.

2.2 Tangling a single file

A cheap imitation of tangle. The idea is a very cheap implementation of tangle. The markup of the file is described in file markup.ow.

The structure of the program is as follows: first we accumulate all of the module definitions, then we write out the definition of the root module (normally "*"). The module definition information will be stored statically in a table supplied by `modtrees.h`; we'll cover the details later.

26a ⟨*header* 18b⟩+≡ (55a) ◁21c 27▷

```
void emit_module_named (FILE *out, char *rootname, char *locformat);
```

26b ⟨* 4⟩+≡ ◁24a 26c▷

```
static char rcsid[] = "$Id: notangle.nw,v 2.11 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#define MAX_MODNAME 255
```

Uses `notangle` 26c.

26c ⟨* 4⟩+≡ ◁26b 28▷

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "strsave.h"
#include "getline.h"
#include "modules.h"
#include "modtrees.h"
#include "errors.h"
#include "match.h"
#include "notangle.h"

⟨Function declarations 30d⟩

void emit_module_named (FILE *out, char *rootname, char *locformat) {
    Module root = NULL; /* ptr to root module */

    root = lookup(rootname);
    ⟨quit if we couldn't find the root 30e⟩
    (void) expand(root,0,0,0,locformat,out);
    putc('\n',out);                     /* make output end with newline */
}
```

Defines:
`notangle`, used in chunks 23, 26b, 33a, and 34d.

Uses `expand` 42a, `getline` 65a, `lookup` 49a, `Module` 38b, `module` 38b, and `root` 57c.

We loop looking for the start of a code chunk. When we find one, we get the name of the module in which the code is to appear. Then we just keep adding lines to that module until we see a terminator. After we see the terminator we start all over again looking for another code chunk.

27 ⟨*header* 18b⟩+≡ (55a) ◁26a 32a▷

```
void read_defs(FILE *in);              /* read module definitions */
```

Uses `module` 38b and `read_defs` 28.

28 ⟨* 4⟩+≡ ◁26c 30c▷

```
void read_defs(FILE *in) {
    char modname[MAX_MODNAME+1] = ""; /* name of module currently being read,
                                         [[""]] if no module is being read */
    Module modptr = NULL;       /* ptr to current module, or NULL */
    char *line = NULL;          /* buffer for input */
    Location loc;

    while ((line = getline(in)) != NULL) {
        if (is_keyword(line, "fatal")) exit(1);
        ⟨track line numbers, then continue unless line is @begin code 29b⟩
        ⟨repeat line = getline(in); until EOF, @defn, or @text 29a⟩
        insist(line,"defn","code chunk had no definition line");
        ⟨copy module name into modname 29d⟩
        warn_dots(modname);       /* names ending in ... aren't like web */
        modptr = insert(modname); /* find or add module in table */

        line = getline(in);
        insist(line,"nl","definition line not followed by newline");
        loc.lineno++;
        line = getline(in);
        while (line != NULL && !is_end(line,"code")) {
            if (is_keyword(line,"nl")) {
                addnewline(modptr);
                loc.lineno++;
            } else if (is_keyword(line,"text")) {
                addstring(modptr,line+1+4+1,loc);
            } else if (is_keyword(line,"use")) {
                warn_dots(line+1+3+5);
                addmodule(modptr,line+1+3+1);
            } else if (is_index(line, "nl")) {
                loc.lineno++;
            ⟨} else if (line contains @file or @line) { adjust loc 29c⟩
            } else if (!is_keyword(line, "index"))
                ⟨complain of botched code chunk 31d⟩
            line = getline(in);
        }
        ⟨if line==NULL die of premature end of file 31b⟩
    }
}
```

Defines:
`read_defs`, used in chunks 24a, 27, and 33b.

Uses `addmodule` 39c, `addnewline` 39c, `addstring` 39c, `getline` 65a, `index` 2, `insert` 48a, `insist` 31a, `is_end` 32c, `is_index` 32c, `is_keyword` 32c, `Location` 38a, `Module` 38b, `module` 38b, and `warn_dots` 30c.

29a ⟨*repeat* `line = getline(in);` *until EOF,* `@defn`*, or* `@text` 29a⟩≡ (28)
```
  do { line = getline(in);
  } while (line != NULL && !is_keyword(line,"defn") && !is_keyword(line,"text"));
```
Uses `getline` 65a and `is_keyword` 32c.

29b ⟨*track line numbers, then* `continue` *unless* `line` *is* `@begin code` 29b⟩≡ (28)
```
  if (is_keyword(line, "nl") || is_index(line, "nl")) {
      loc.lineno++;
  ⟨} else if (line contains @file or @line) { adjust loc 29c⟩
  }
  if (!is_begin(line, "code"))
      continue;
```
Uses `is_begin` 32c, `is_index` 32c, and `is_keyword` 32c.

The only tricky bit with the line numbers is to note that `@line` gives the line number of the *following* line, not of the line on which the `@line` appears. That means `loc.lineno` must be decremented after it is set, so that the next newline will increment it to the correct value.

29c ⟨`} else if (line` *contains* `@file` *or* `@line) {` *adjust* `loc` 29c⟩≡ (28 29b)
```
  } else if (is_keyword(line,"file")) {
      ⟨save name from line (@file ...) into loc.filename 29e⟩
      loc.lineno = 1;
  } else if (is_keyword(line, "line")) {
      ⟨save line number from line (@line ...) into loc.lineno 30a⟩
      loc.lineno--;
```
Uses `is_keyword` 32c.

When copying the module name or a file name, we have to strip the trailing newline.

29d ⟨*copy module name into* `modname` 29d⟩≡ (28)
```
  strcpy(modname,line+strlen("@defn "));
  modname[strlen(modname)-1]='\0';
```
Uses `strcpy` 55c and `strlen` 55c.

29e ⟨*save name from* `line` *(*`@file ...`*) into* `loc.filename` 29e⟩≡ (29c)
```
  { char temp[MAX_MODNAME+1];
    if (strlen(line) >= MAX_MODNAME + strlen("@file "))
      overflow("file name size");
    strcpy(temp,line+strlen("@file "));
    temp[strlen(temp)-1]='\0';
    loc.filename = strsave(temp);
  }
```
Uses `overflow` 70a, `strcpy` 55c, and `strlen` 55c.

30a ⟨*save line number from* `line` *(*`@line ...`*) into* `loc.lineno` 30a⟩≡ (29c)
```
{ char temp[MAX_MODNAME+1];
  if (strlen(line) >= MAX_MODNAME + strlen("@line "))
    overflow("file name size");
  strcpy(temp,line+strlen("@line "));
  temp[strlen(temp)-1]='\0';
  ⟨fail if string temp contains a non-digit 30b⟩
  loc.lineno = atoi(temp);
}
```
Uses `overflow` 70a, `strcpy` 55c, and `strlen` 55c.

30b ⟨*fail if string* `temp` *contains a non-digit* 30b⟩≡ (30a)
```
{ char *p;
  for (p = temp; *p; p++)
    if (!isdigit(*p))
      errormsg(Error, "non-numeric line number in ‘@line %s’", temp);
}
```
Uses `errormsg` 70a 70b.

In WEB, module names ending in "..." may be prefixes for other names. We don't do anything like that here, but we do warn the user about "..." in case he's got a file converted from WEB.

30c ⟨* 4⟩+≡ ◁28 31a▷
```
static
void warn_dots(char *modname) {
  if (!strcmp(modname+strlen(modname)-3,"..."))
    errormsg(Warning, "Module name <<%s>> isn’t completed as in web",
             modname);
}
```
Defines:
`warn_dots`, used in chunks 28 and 30d.
Uses `errormsg` 70a 70b, `Module` 38b, and `strlen` 55c.

30d ⟨*Function declarations* 30d⟩≡ (26c 39b 47b) 31c▷
```
static void warn_dots(char *modname);          /* warn about names ending in ... */
```
Uses `warn_dots` 30c.

Error checking is perenially dull.

30e ⟨*quit if we couldn't find the root* 30e⟩≡ (26c)
```
if (root==NULL) {
    errormsg(Fatal, "The root module <<%s>> was not defined.", rootname);
    return;
}
```
Uses `errormsg` 70a 70b, `module` 38b, and `root` 57c.

31a ⟨*4⟩+≡ ◁30c 32b▷
```
void insist(char *line, char *keyword, char *msg) {
  ⟨if line==NULL die of premature end of file 31b⟩
  if (!is_keyword(line,keyword))
    impossible(msg);
}
```
Defines:
  `insist`, used in chunks 28 and 31c.
Uses `impossible` 70a and `is_keyword` 32c.

31b ⟨*if* `line==NULL` *die of premature end of file* 31b⟩≡ (28 31a)
```
if (line==NULL) {
    impossible("End of file occurred in mid-module");
}
```
Uses `impossible` 70a and `module` 38b.

31c ⟨*Function declarations* 30d⟩+≡ (26c 39b 47b) ◁30d 41a▷
```
void insist(char *line, char *keyword, char *msg);
```
Uses `insist` 31a.

31d ⟨*complain of botched code chunk* 31d⟩≡ (28)
```
errorat(loc.filename, loc.lineno, Error, "botched code chunk `%s'", line);
```
Uses `errorat` 70d.

2.3 Code to recognize noweb markup on input

We recognize the input using matching functions, which can find `@begin`, `@end`, and other useful information.

32a ⟨*header* 18b⟩+≡ (55a) ◁27 38a▷

```
extern int is_keyword(char *line, char *keyword);
extern int is_begin(char *line, char *type);
extern int is_end(char *line, char *type);
extern int is_index(char *line, char *type);
```

Uses `is_begin` 32c, `is_end` 32c, `is_index` 32c, and `is_keyword` 32c.

32b ⟨* 4⟩+≡ ◁31a 32c▷

```
static char rcsid[] = "$Id: match.nw,v 1.7 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <string.h>
#include "match.h"
static int matches(char *line, char *search) {
    return !strncmp(line,search,strlen(search));
}
```

Uses `strlen` 55c.

On 24 March 1996, I changed this to accept zero as a terminator because I strip newlines in the prettyprinter. Too bad if we just got worse at detecting bogus filters.

32c ⟨* 4⟩+≡ ◁32b 33a▷

```
int is_keyword(char *line, char *keyword) {
    char low_at_sign = '@';
    return *line==low_at_sign && matches(line+1,keyword) &&
           (line[strlen(keyword)+1]==' ' || line[strlen(keyword)+1]=='\n' ||
            line[strlen(keyword)+1]=='\0');
}
int is_begin(char *line, char *type) {
    return is_keyword(line,"begin") && matches(line+1+6,type);
}
int is_end(char *line, char *type) {
    return is_keyword(line,"end") && matches (line+1+4,type);
}
int is_index(char *line, char *type) {
    return is_keyword(line,"index") && matches(line+1+6,type);
}
```

Defines:
`is_begin`, used in chunks 29b, 32a, and 53a.
`is_end`, used in chunks 28, 32a, and 53a.
`is_index`, used in chunks 28, 29b, 32a, and 52b.
`is_keyword`, used in chunks 28, 29, 31a, 32a, 52b, and 53a.
Uses `index` 2 and `strlen` 55c.

2.4 Expanding multiple files from a single source

This main program is used to make the monolithic `noweb` script efficient. Tips of the hat to Ross Williams and Preston Briggs.

33a ⟨* 4⟩+≡ ◁32c 34c▷

```
static char rcsid[] = "$Id: mnt.nw,v 1.8 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <ctype.h>
#include "modules.h"
#include "modtrees.h"
#include "notangle.h"
#include "errors.h"
#include "columns.h"
#include "strsave.h"

⟨local prototypes 34b⟩

#define Clocformat "#line %L \"%F\"%N"
static char *locformat = Clocformat;

main(int argc, char **argv) {
    int i;

    tabsize = 0;  /* default for nt is not to use tabs */

    progname = argv[0];
    finalstage = 1;
    ⟨read standard input into tree 33b⟩
    for (i=1; i<argc; i++)
      switch (*argv[i]) {
        case '-': ⟨handle option in argv[i] 37a⟩                    break;
        default:  emitfile(argv[i]);                                break;
      }
    nowebexit(NULL);
    return errorlevel;          /* slay warning */
}
```

Uses `emitfile` 35b, `errorlevel` 69b, `notangle` 26c, and `tabsize` 67a.

33b ⟨*read standard input into tree* 33b⟩≡ (33a)

```
read_defs(stdin);
apply_each_module(remove_final_newline);
```

Uses `apply_each_module` 49b, `read_defs` 28, and `remove_final_newline` 46.

34a ⟨*write out all conforming roots* 34a⟩≡ (37a)

```
apply_each_module(add_uses_to_usecounts);
apply_each_module(emit_if_unused_and_conforming);
```

Uses `add_uses_to_usecounts` 34c, `apply_each_module` 49b, and `emit_if_unused_and_conforming` 34d.

34b ⟨*local prototypes* 34b⟩≡ (33a 50c) 35a ▷

```
void add_uses_to_usecounts(Module mp);
void emit_if_unused_and_conforming(Module mp);
```

Uses `add_uses_to_usecounts` 34c, `emit_if_unused_and_conforming` 34d, and `Module` 38b.

34c ⟨* 4⟩+≡ ◁ 33a 34d ▷

```
void add_uses_to_usecounts(Module mp) {
    Module used;
    struct modpart *p;
    for (p=mp->head; p!=NULL; p=p->next)
        if (p->ptype == MODULE) {
            used = lookup(p->contents);
            if (used != NULL)
                used->usecount++;
        }
}
```

Defines:
`add_uses_to_usecounts`, used in chunk 34.
Uses `lookup` 49a, `modpart` 38a, and `Module` 38b.

A conforming output chunk name has no spaces and no metacharacters. Names with spaces are silently ignored, but names that are otherwise conforming but that contain metacharacters cause complaints.

34d ⟨* 4⟩+≡ ◁ 34c 35b ▷

```
void emit_if_unused_and_conforming(Module mp) {
    char *index;
    if (mp->usecount == 0 && strpbrk(mp->name, " \n\t\v\r\f") == NULL)
        if (index = strpbrk(mp->name, "[](){}!$&<>*?;|^`'\\\""),
            index == NULL || index[0] == '*' && index[1] == 0)
            if (index == mp->name)
                errormsg(Error, "<<*>> is not a good chunk name for noweb; "
                                "use notangle instead");
            else
                emitfile(mp->name);
        else
            errormsg(Error, "<<%s>> cannot be an output chunk; "
                            "it contains a metacharacter", mp->name);
}
```

Defines:
`emit_if_unused_and_conforming`, used in chunk 34.
Uses `emitfile` 35b, `errormsg` 70a 70b, `index` 2, `Module` 38b, and `notangle` 26c.

35a ⟨*local prototypes* 34b⟩+≡ (33a 50c) ◁34b 37b▷

```
static void emitfile(char *modname);
```

Uses `emitfile` 35b.

35b ⟨* 4⟩+≡ ◁34d 39b▷

```
static void emitfile(char *modname) {
  Module root = lookup(modname);
  char *tempname = tempnam(".", 0);
  FILE *fp;
  char *lfmt, *filename;
  ⟨set lfmt and filename from modname 35c⟩
  ⟨complain and return if root == NULL 36c⟩
  fp = fopen(tempname, "w");
  if (fp == NULL) errormsg(Fatal, "Can't open temporary file %s", tempname);
  ⟨expand root onto fp and close the file 35d⟩
  ⟨if file filename is as file tempname, remove tempname and return 36a⟩
  remove(filename);
  if (rename(tempname, filename) != 0) { /* different file systems? (may have to copy) */
    FILE *fp = fopen(filename, "w");
    if (fp == NULL) {remove(tempname); ⟨complain about filename and return 36b⟩}
    ⟨expand root onto fp and close the file 35d⟩
    remove(tempname);
  }
}
```

Defines:
  `emitfile`, used in chunks 33–35.
Uses `errormsg` 70a 70b, `lookup` 49a, `Module` 38b, `root` 57c, and `tempnam` 37b.

35c ⟨*set* `lfmt` *and* `filename` *from* `modname` 35c⟩≡ (35b)

```
{ int n = strlen(modname) - 1;
  if (n >= 0 && modname[n] == '*') {
    lfmt = locformat;
    filename = strsave(modname);
    filename[n] = 0;
  } else {
    lfmt = "";
    filename = modname;
  }
}
```

Uses `strlen` 55c.

35d ⟨*expand* `root` *onto* `fp` *and close the file* 35d⟩≡ (35b)

```
resetloc();
(void) expand(root, 0, 0, 0, lfmt, fp);
putc('\n', fp);
fclose(fp);
```

Uses `expand` 42a and `root` 57c.

36a ⟨*if file* filename *is as file* tempname, *remove* tempname *and* return 36a⟩≡ (35b)
```
{ FILE *dest, *tmp;
  dest = fopen(filename, "r");
  if (dest != NULL) {
    int x, y;
    tmp = fopen(tempname, "r");
    assert(tmp);
    do {
      x = getc(tmp);
      y = getc(dest);
    } while (x == y && x != EOF);
    fclose(tmp);
    fclose(dest);
    if (x == y) {
      remove(tempname);
      return;
    }
  }
}
```

36b ⟨*complain about* filename *and* return 36b⟩≡ (35b)
```
errormsg(Error, "Can't open output file %s", filename);
return;
```
Uses errormsg 70a 70b.

36c ⟨*complain and* return *if* root == NULL 36c⟩≡ (35b)
```
if (root == NULL) {
  errormsg(Error, "Chunk <<%s>> is undefined", filename);
  return;
}
```
Uses errormsg 70a 70b and root 57c.

37a ⟨*handle option in* argv[i] 37a⟩≡ (33a)
```
      switch (*++argv[i]) {
          case 'a':
              if (strcmp(argv[i], "all"))
                  errormsg(Warning, "Ignoring unknown option -%s", argv[i]);
              else {⟨write out all conforming roots 34a⟩}
              break;
          case 't': /* set tab size or turn off */
              if (isdigit(argv[i][1]))
                  tabsize = atoi(argv[i]+1);
              else if (argv[i][1]==0)
                  tabsize = 0;            /* no tabs */
              else
                  errormsg(Error, "%s: ill-formed option %s\n", argv[0], argv[i]);
              break;
          case 'L': /* have a #line number format */
              locformat = argv[i] + 1;
              if (!*locformat) locformat = Clocformat;
              break;
          default:
              errormsg(Warning, "Ignoring unknown option -%s", argv[i]);
      }
```
Uses `errormsg` 70a 70b and `tabsize` 67a.

2.4.1 Temporary files

Some operating systems provide a `tempnam` call that makes it possible to create a temporary file in the current directory. This makes `mnt` more efficient by making `rename` efficient. If your operating system provides `tempnam`, define `TEMPNAM` for compilation.

37b ⟨*local prototypes* 34b⟩+≡ (33a 50c) ◁35a 51c▷
```
#ifdef TEMPNAM
extern char *tempnam (const char *dir, const char *pfx);        /* temp file in dir */
#else
#define tempnam(DIR,PFX) (strsave(tmpnam(NULL)))
#endif
```
Defines:
`tempnam`, used in chunk 35b.

2.5 The internal structure of modules (chunks)

This code is so old, it dates back to when I used to call "chunks" "modules."

2.5.1 Assembling modules from parts

Modules are linked lists. Each element of the list is either a string or a reference to another module name. During expansion of a module, strings are printed and module references are expanded. The structure of module references must be a dag or else `expand` will complain about a cycle.

38a ⟨*header* 18b⟩+≡ (55a) ◁32a 38b▷

```
typedef struct location {       /* identify lines of source */
    char *filename;
    int lineno;
} Location;

typedef enum parttype {STRING=1, MODULE, NEWLINE} Parttype;

struct modpart {
    Parttype ptype;             /* type of fragment: STRING, MODULE, NEWLINE */
    char *contents;
    Location loc;               /* for String, where's it from ? */
    struct modpart *next;
};
```

Defines:
`Location`, used in chunks 28, 39–41, and 44.
`modpart`, used in chunks 34c, 38b, 40–42, and 46.
`parttype`, never used.

In order to facilitate expansion of all roots (and to identify roots), I keep a use counter in each module.

38b ⟨*header* 18b⟩+≡ (55a) ◁38a 38c▷

```
typedef struct module {
    char *name;
    int usecount;
    struct modpart *head, *tail;
} *Module;
```

Defines:
`Module`, used in chunks 21a, 26c, 28, 30c, 34, 35b, and 38–49.
`module`, used in chunks 4, 19–21, 24–28, 30e, 31b, 38, 39, 41, 42a, and 45–47.
Uses `modpart` 38a.

38c ⟨*header* 18b⟩+≡ (55a) ◁38b 39c▷

```
Module newmodule(char *modname);         /* create a new, blank module */
```

Uses `Module` 38b, `module` 38b, and `newmodule` 39a.

39a ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁22 40a▷

```
Module newmodule (char *modname) {
    Module p = (Module) malloc (sizeof (struct module));
    checkptr(p);
    p->name = strsave(modname);
    p->usecount = 0;
    p->head = p->tail = NULL;
    return p;
}
```

Defines:
  `newmodule`, used in chunks 38c and 48a.
Uses `checkptr` 70a, `malloc` 55d, `Module` 38b, and `module` 38b.

39b ⟨*** 4⟩+≡ ◁35b 47b▷

```
static char rcsid[] = "$Id: modules.nw,v 2.12 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "modules.h"
#include "modtrees.h"
#include "errors.h"
#include "columns.h"
#include "strsave.h"

⟨Function declarations 30d⟩
⟨local data 41c⟩
⟨C functions 18d⟩
```

39c ⟨*header* 18b⟩+≡ (55a) ◁38c 41b▷

```
#define addstring(MP,S,L) add_part(MP,S,STRING,&L)
        /* add a string to a module definition (stripping final newline) */
#define addmodule(MP,S) add_part(MP,S,MODULE,0)
        /* add a module reference to a module definition (stripping final newline) */
#define addnewline(MP) add_part(MP,0,NEWLINE,0)
void add_part (Module mp, char *s, Parttype type, Location *loc);
```

Defines:
  `addmodule`, used in chunk 28.
  `addnewline`, used in chunk 28.
  `addstring`, used in chunk 28.
Uses `add_part` 40a, `Location` 38a, `Module` 38b, and `module` 38b.

40a ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁39a 40b▷

```
void add_part (Module mp, char *s, Parttype type, Location *loc) {
    struct modpart *p = newmodpart(type,s,loc);
    append (mp,p);
}
```

Defines:
`add_part`, used in chunk 39c.
Uses `append` 40c, `Location` 38a, `modpart` 38a, `Module` 38b, and `newmodpart` 40b.

40b ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁40a 40c▷

```
static struct modpart *
newmodpart(int type, char *s, Location *loc) {
    struct modpart *p = (struct modpart *) malloc (sizeof (struct modpart));
    checkptr(p);
    p->ptype = type;
    if (s) {
        p->contents = strsave(s);
        ⟨strip final newline from p-¿contents 40d⟩
    }
    if (loc) p->loc = *loc;
    p->next = NULL;
    return p;
}
```

Defines:
`newmodpart`, used in chunks 40a and 41a.
Uses `checkptr` 70a, `Location` 38a, `malloc` 55d, and `modpart` 38a.

40c ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁40b 41e▷

```
static
void append(Module mp, struct modpart *p) {
    /* append p to mp's list of modparts */
    if (mp->head == NULL) {
        mp->head = mp->tail = p;
    } else {
        mp->tail->next = p;
        mp->tail = p;
    }
}
```

Defines:
`append`, used in chunks 40a and 41a.
Uses `modpart` 38a and `Module` 38b.

40d ⟨*strip final newline from p-¿contents* 40d⟩≡ (40b)

```
{ int k = strlen(p->contents)-1;
  if (p->contents[k] == '\n') p->contents[k] = '\0';
  else impossible("input line doesn't end with newline");
}
```

Uses `impossible` 70a and `strlen` 55c.

41a ⟨*Function declarations* 30d⟩+≡ (26c 39b 47b) ◁31c 43e▷
```
static struct modpart *
newmodpart(int type, char *s, Location *loc);   /* create a new module part */

static
void append(Module mp, struct modpart *p);
```
Uses `append` 40c, `Location` 38a, `modpart` 38a, `Module` 38b, `module` 38b, and `newmodpart` 40b.

2.5.2 Expanding modules

`expand` returns 0 if all is well, nonzero if error. A `struct parent` chain lists modules currently being expanded, so I can detect cycles. `partial_distance` is the size of the line already written out when `expand` is called.

41b ⟨*header* 18b⟩+≡ (55a) ◁39c 41d▷
```
typedef struct parent {
    Module this;
    struct parent *parent;
} *Parent;

int expand (Module mp, int indent, int partial_distance, Parent parent,
            char *locformat, FILE *out);
        /* expand a module, writing to file out */
```
Defines:
  `Parent`, used in chunks 42–44.
Uses `expand` 42a, `Module` 38b, and `module` 38b.

41c ⟨*local data* 41c⟩≡ (39b)
```
static char *lastfilename = 0;
static int lastlineno = -1;
```
Defines:
  `lastfilename`, used in chunks 41e and 44c.
  `lastlineno`, used in chunks 41e, 43a, and 44c.

41d ⟨*header* 18b⟩+≡ (55a) ◁41b 44b▷
```
void resetloc(void);
```

41e ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁40c 42a▷
```
void resetloc(void) {
  lastfilename = 0;
  lastlineno = -1;
}
```
Uses `lastfilename` 41c and `lastlineno` 41c.

indent is the amount by which this chunk should be indented. partial_distance is the width of what has already been written to the current line.

42a ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁41e 44a▷

```
int expand (Module mp, int indent, int partial_distance, Parent parent,
            char *locformat, FILE *out) {
    struct modpart *p;
    Module newmod;
    int error=Normal;
    struct parent thismodule; /* the value only matters when we're expanding a module */

    ⟨set thismodule = {mp, parent} 43c⟩
    ⟨if mp is an ancestor, complain and return 43d⟩

    for (p=mp->head; p!=NULL; p=p->next) {
        switch (p->ptype) {
            case STRING:  ⟨print a string 42b⟩;  break;
            case MODULE:  ⟨expand a module 43b⟩; break;
            case NEWLINE: ⟨print a newline 43a⟩; break;
            default: impossible("bad part type");
        }
    }
    return error;
}
```

Defines:
expand, used in chunks 26c, 35d, 41b, and 43b.
Uses impossible 70a, modpart 38a, Module 38b, module 38b, and Parent 41b.

If the previous line was not partial, this string starts a fresh line and it is necessary to indent. When emitting #line directives, don't indent—instead try to preserve the original column of the information in the source, as well as the line number. This means indenting by partial_distance if a #line directive is emitted, *except* when it's the first line of a module (hack! hack!). printloc returns nonzero when #line is actually emitted.

42b ⟨*print a string* 42b⟩≡ (42a)

```
if (*(p->contents) != '\0') {
    if (*locformat) {
        if (printloc(out,locformat,p->loc,partial_distance) && (p != mp->head))
              indent_for(partial_distance, out);
    } else if (partial_distance == 0) {
        indent_for(indent, out);
        partial_distance = indent;
    }
    fprintf(out,"%s",p->contents);
    partial_distance = limitcolumn(p->contents, partial_distance);
}
```

Uses indent_for 68, limitcolumn 67d, and printloc 44c.

43a ⟨*print a newline* 43a⟩≡ (42a)

```
partial_distance = 0;
putc('\n', out);
lastlineno++;
```

Uses `lastlineno` 41c.

Previously, when we expanded a module, the partial distance became the new indentation level. This was clearly wrong when we were at the beginning of the line, so I just hacked a special case for no `locformat` and zero `partial_distance`. It's essentially what would be done above for a string of length zero if that weren't hacked out as a special case. This change probably blows the case where the module being expanded is empty.

43b ⟨*expand a module* 43b⟩≡ (42a)

```
newmod = lookup(p->contents);
if (newmod==NULL) {
    errormsg (Error, "undefined chunk name: <<%s>>", p->contents);
    error=Error;
} else {
    int retcode;
    if (*locformat == 0 && partial_distance == 0) {
        indent_for(indent, out);
        partial_distance = indent;
    }
    retcode = expand (newmod, partial_distance, partial_distance,
                      &thismodule, locformat, out);
    if (retcode > error) error = retcode;
}
partial_distance = limitcolumn(p->contents, partial_distance + 2) + 2;
                                   /* account for surrounding brackets */
```

Uses `errormsg` 70a 70b, `expand` 42a, `indent_for` 68, `limitcolumn` 67d, and `lookup` 49a.

43c ⟨*set* `thismodule = {mp, parent}` 43c⟩≡ (42a)

```
thismodule.this = mp;
thismodule.parent = parent;
```

43d ⟨*if* `mp` *is an ancestor, complain and return* 43d⟩≡ (42a)

```
if (seekcycle(mp, parent)) {
    errormsg(Error, "<<%s>>", mp->name);
    return Error;
}
```

Uses `errormsg` 70a 70b and `seekcycle` 44a.

43e ⟨*Function declarations* 30d⟩+≡ (26c 39b 47b) ◁41a 48c▷

```
static int seekcycle(Module mp, Parent parent);
```

Uses `Module` 38b, `Parent` 41b, and `seekcycle` 44a.

44a ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁42a 44c▷

```
static int seekcycle(Module mp, Parent parent) {
    if (parent == NULL) {
        return 0;
    } else if (parent->this == mp || seekcycle(mp, parent->parent)) {
        if (parent->this == mp)
            fprintf(stderr, "Cyclic code chunks: ");
        fprintf(stderr, "<<%s>> -> ", parent->this->name);
        return 1;
    } else {
        return 0;
    }
}
```

Defines:
`seekcycle`, used in chunk 43.
Uses `Module` 38b and `Parent` 41b.

Printing locations means emitting `#line` numbers. `printloc` emits a `#line` only if the line emitted is out of sequence. It returns nonzero when `#line` is emitted; zero otherwise. The flag `partial` tells whether the preceding line was partial. If so, then a newline is necessary before `#line` can be emitted.

44b ⟨*header* 18b⟩+≡ (55a) ◁41d 45c▷

```
int printloc(FILE *fp, char *fmt, Location loc, int partial);
```

Uses `Location` 38a and `printloc` 44c.

44c ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁44a 46▷

```
int printloc(FILE *fp, char *fmt, Location loc, int partial) {
    char *p;
    if (*fmt
    && (loc.filename!=lastfilename || lastlineno != loc.lineno)) {
        if (partial) putc('\n',fp);
        ⟨write out the line number pragma 45a⟩
        lastfilename = loc.filename;
        lastlineno = loc.lineno;
        return 1;
    } else return 0;
}
```

Defines:
`printloc`, used in chunks 42b and 44b.
Uses `lastfilename` 41c, `lastlineno` 41c, and `Location` 38a.

45a ⟨*write out the line number pragma* 45a⟩≡ (44c)
```
  for (p = fmt; *p; p++) {
      if (*p == '%') {
          switch (*++p) {
              case '%': putc('%', fp);                              break;
              case 'N': putc('\n', fp);                             break;
              case 'F': fprintf(fp, "%s", loc.filename);            break;
              case 'L': fprintf(fp, "%d", loc.lineno);              break;
              case '-': case '+':
                          if (isdigit(p[1]) && p[2] == 'L') {
                            fprintf(fp, "%d", *p == '+' ? loc.lineno + (p[1] - '0')
                                                        : loc.lineno - (p[1] - '0'));
                            p += 2;
                          } else
                            ⟨complain about bad format character 45b⟩
                        break;
              default:  ⟨complain about bad format character 45b⟩    break;
          }
      } else putc(*p, fp);
  }
```

45b ⟨*complain about bad format character* 45b⟩≡ (45a)
```
  { static int complained = 0;
    if (!complained) {
      errormsg(Error,"Bad format sequence ``%%%c'' in -L%s",*p,fmt);
      complained = 1;
    }
  }
```
Uses `errormsg` 70a 70b.

2.5.3 Other operations on completed modules

For more elegant-looking output, we provide a primitive that removes the final newline from a module.

We will use `apply_each_module` to remove trailing newlines from each module. This serves two purposes: first, our output looks nicer, and, second, it makes it possible to define a module that doesn't end with newline.

As they are read in, all modules end with newline, by definition.[2] We remove all of the trailing newlines.

45c ⟨*header* 18b⟩+≡ (55a) ◁44b 47a▷
```
  void remove_final_newline (Module mp);
          /* remove trailing newline that must be in module */
```
Uses `Module` 38b, `module` 38b, and `remove_final_newline` 46.

[2]Except of course if this input came from an uncorrected nuweb file, in which case modules needn't end with newlines. Eventually one hopes I'll fix numarkup to insert exactly the newlines that would be removed here.

46 ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁44c 64c▷

```
void remove_final_newline (Module mp) {
        /* remove trailing newline that must be in module */
    if (mp->tail==NULL) /* module has no text */
        return;
    if (mp->tail->ptype != NEWLINE)
        /* do nothing --- this is possible with nuweb front end
              formerly: impossible("Module doesn't end with newline"); */
        ;
    else if (mp->tail == mp->head)
            mp->head = mp->tail = NULL;
    else {
        struct modpart *p = mp->head;
        while (p->next != mp->tail) p = p->next;
        p->next = NULL;
    }
}
```

Defines:
  `remove_final_newline`, used in chunks 24a, 33b, and 45c.
Uses `impossible` 70a, `modpart` 38a, `Module` 38b, and `module` 38b.

2.6 Support for looking up modules by name

Trees of modules. A table of modules with insert and lookup, but no delete. The key is the module name.

47a ⟨*header* 18b⟩+≡ (55a) ◁45c 55b▷

```
Module insert (char *modname);  /* add a module to the world */
Module lookup (char *modname);  /* return ptr to module or NULL */
void apply_each_module(void (*fun)(Module));
     /* apply [[fun]] to each module in the tree */
```

Uses `apply_each_module` 49b, `insert` 48a, `lookup` 49a, `Module` 38b, and `module` 38b.

47b ⟨* 4⟩+≡ ◁39b 48a▷

```
static char rcsid[] = "$Id: modtrees.nw,v 2.10 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "strsave.h"
#include "modules.h"
#include "modtrees.h"
#include "errors.h"

typedef struct tnode {          /* tree node */
  struct tnode *left, *right;
  Module data;
} TNODE;

static struct tnode *root=NULL;

⟨Function declarations 30d⟩
```

Uses `Module` 38b and `root` 57c.

48a ⟨* 4⟩+≡ ◁47b 49a▷

```
Module insert (char *modname) {
    return insert_tree (&root, modname);
}

static Module
insert_tree(TNODE **rootptr, char *modname) {
     if (*rootptr==NULL) {
         ⟨Allocate new tnode in **rootptr 48b⟩
         return (*rootptr)->data = newmodule(modname);
     }
     if (strcmp((*rootptr)->data->name,modname)==0) {
         return (*rootptr)->data;
     } else if (strcmp((*rootptr)->data->name,modname)<0) {
         return insert_tree(&((*rootptr)->left),modname);
     } else /* >0 */ {
         return insert_tree(&((*rootptr)->right),modname);
     }

}
```

Defines:
`insert`, used in chunks 28 and 47a.
Uses `Module` 38b, `newmodule` 39a, and `root` 57c.

Node allocation is perfectly standard.

48b ⟨*Allocate new tnode in **rootptr* 48b⟩≡ (48a)

```
        checkptr(*rootptr=(struct tnode *) malloc (sizeof(struct tnode)));
        (*rootptr)->left = (*rootptr)->right = NULL;
```

Uses `checkptr` 70a and `malloc` 55d.

48c ⟨*Function declarations* 30d⟩+≡ (26c 39b 47b) ◁43e

```
static Module insert_tree(TNODE **rootptr, char *modname);
static Module lookup_tree(TNODE **rootptr, char *modname);
```

Uses `Module` 38b.

49a ⟨* 4⟩+≡ ◁48a 49b▷

```
Module lookup (char *modname) {
    return lookup_tree (&root, modname);
}

static Module
lookup_tree(TNODE **rootptr, char *modname) {
     if (*rootptr==NULL) {
        return NULL;
     }
     if (strcmp((*rootptr)->data->name,modname)==0) {
         return (*rootptr)->data;
     } else if (strcmp((*rootptr)->data->name,modname)<0) {
         return lookup_tree(&((*rootptr)->left),modname);
     } else /* >0 */ {
         return lookup_tree(&((*rootptr)->right),modname);
     }
}
```

Defines:
`lookup`, used in chunks 26c, 34c, 35b, 43b, and 47a.
Uses `Module` 38b and `root` 57c.

`apply_each_module` takes as argument a function and applies it to each module in the tree. This makes it easy to, for example, remove trailing newlines from each module.

49b ⟨* 4⟩+≡ ◁49a 50a▷

```
static
void apply_to_tree(TNODE *p, void (*fun)(Module)) {
    if (p != NULL) {
        apply_to_tree(p->left,fun);
        (*fun)(p->data);
        apply_to_tree(p->right,fun);
    }
}
void apply_each_module(void (*fun)(Module)) {
    apply_to_tree(root,fun);
}
```

Defines:
`apply_each_module`, used in chunks 24a, 33b, 34a, and 47a.
Uses `Module` 38b and `root` 57c.

3 Scanning for uses of identifiers

3.1 Main program

50a ⟨* 4⟩+≡ ◁49b 50b▷
```
static char rcsid[] = "$Id: finduses.nw,v 1.8 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "errors.h"
#include "match.h"
#include "getline.h"
#include "recognize.h"
```
Uses `getline` 65a.

These choices of alphanumerics and symbols seem to work for most languages. Making `@` alphanumeric helps LaTeX, and making `#` alphanumeric helps avoid false hits on C preprocessor directives like `#define` and `#include`.

50b ⟨* 4⟩+≡ ◁50a 50c▷
```
static Recognizer nwindex;
#define ALPHANUM "_'@ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789#"
#define SYMBOLS "!%^&*-+:=|~<>./?`"
/* note $ and \ both delimiters */
```
Defines:
- `ALPHANUM`, used in chunk 51a.
- `SYMBOLS`, used in chunk 51a.

Uses `Recognizer` 56a 58b.

By default, find uses within quoted code (`[[...]]`).

50c ⟨* 4⟩+≡ ◁50b 51a▷
```
static int showquotes = 1;
⟨typedefs 52c⟩
⟨local prototypes 34b⟩
```

51a ⟨* 4⟩+≡ ◁50c 52a▷

```
main(int argc, char **argv) {
    FILE *fp;
    int i;

    progname = argv[0];
    for (i = 1; i < argc && argv[i][0] == '-' && argv[i][1] != 0; i++)
        if (!strcmp(argv[i], "-noquote"))
            showquotes = 0;
        else
            errormsg(Error, "%s: unknown option %s\n", progname, argv[i]);
    nwindex = new_recognizer(ALPHANUM, SYMBOLS);
    if (i == argc) {
       ⟨add uses to stdin, grabbing defns from stdin 52b⟩
    } else {
       ⟨read identifiers to be defined from files named in argv 51b⟩
       stop_adding(nwindex);
       add_use_markers(stdin, stdout);
    }
    nowebexit(NULL);
    return errorlevel;          /* slay warning */
}
```

Uses `add_use_markers` 53a, `ALPHANUM` 50b, `errorlevel` 69b, `errormsg` 70a 70b, `new_recognizer` 55e, `stop_adding` 56b 61, and `SYMBOLS` 50b.

51b ⟨*read identifiers to be defined from files named in* `argv` 51b⟩≡ (51a)

```
for (; i < argc; i++)
    if ((fp=fopen(argv[i],"r"))==NULL)
        errormsg(Error, "%s: couldn't open file %s\n", progname, argv[i]);
    else {
        read_ids(fp);
        fclose(fp);
    }
```

Uses `errormsg` 70a 70b and `read_ids` 52a.

51c ⟨*local prototypes* 34b⟩+≡ (33a 50c) ◁37b 52d▷

```
static void read_ids(FILE *in);
```

Uses `read_ids` 52a.

52a ⟨* 4⟩+≡ ◁51a 53a▷
```
static void read_ids(FILE *in) {
    char *line;
    while ((line = getline(in)) != NULL) {
        if (line[strlen(line)-1] == '\n') line[strlen(line)-1] = 0;
        add_ident(nwindex, line);
    }
}
```
Defines:
  `read_ids`, used in chunk 51.
Uses `add_ident` 56b 59a, `getline` 65a, and `strlen` 55c.

52b ⟨*add uses to stdin, grabbing defns from stdin* 52b⟩≡ (51a)
```
{   FILE *tmp = tmpfile();
    char *line;
    if (tmp == NULL) ⟨complain about opening temp file and exit 54c⟩
    while ((line = getline(stdin)) != NULL) {
        if (fputs(line, tmp) == EOF) ⟨complain about writing temp file and exit 54d⟩
        if (is_index(line, "defn")) {
            if (line[strlen(line)-1] == '\n') line[strlen(line)-1] = 0;
            add_ident(nwindex, line+1+5+1+4+1);
        } else if (is_index(line, "localdefn")) {
            if (line[strlen(line)-1] == '\n') line[strlen(line)-1] = 0;
            add_ident(nwindex, line+1+5+1+9+1);
        } else if (is_keyword(line, "fatal")) {
            exit(1);
        }
    }
    rewind(tmp);
    stop_adding(nwindex);
    add_use_markers(tmp, stdout);
}
```
Uses `add_ident` 56b 59a, `add_use_markers` 53a, `getline` 65a, `is_index` 32c, `is_keyword` 32c,
  `stop_adding` 56b 61, and `strlen` 55c.

52c ⟨*typedefs* 52c⟩≡ (50c)
```
typedef struct line_and_outfile {
    char *line;
    FILE *out;
} LineOut;
```

52d ⟨*local prototypes* 34b⟩+≡ (33a 50c) ◁51c 54b▷
```
static void add_use_markers(FILE *in, FILE *out);
```
Uses `add_use_markers` 53a.

53a ⟨* 4⟩+≡ ◁52a 53b▷

```
static void add_use_markers(FILE *in, FILE *out) {
    char *line;
    int incode = 0;
    LineOut info; info.line = (char *)0; info.out = out;

    while ((line = getline(in)) != NULL) {
        if (is_begin(line, "code") || showquotes && is_keyword(line, "quote"))
            incode = 1;
        else if (is_end(line, "code") || is_keyword(line, "endquote"))
            incode = 0;
        if (is_keyword(line, "text") && incode) {
            info.line = line + 6; /* skip "@text " */
            search_for_ident(nwindex, line, write_index_use, &info);
            if (*info.line && *info.line != '\n')
                fprintf(out, "@text %s", info.line);    /* has newline */
        } else
            fprintf(out, "%s", line);
    }
}
```

Defines:
`add_use_markers`, used in chunks 51 and 52.
Uses `getline` 65a, `is_begin` 32c, `is_end` 32c, `is_keyword` 32c, `search_for_ident` 56c 62a, and `write_index_use` 53b.

We gradually cut out the uses, and the tail of the line is left in `info.line`, to be printed by the code above. There's a tricky bug lurking here—if one identifier is a prefix of another, but both are recognized (as with the C++ `::` separator), we have to avoid writing them both out in full, because that would duplicate text unnecessarily. As a result, we always emit the line in pieces. The function `emit_up_to(f, s, limit)` emits the piece of the string `s` up to but not including `limit`, if any. It returns `limit` or `s`, whichever is greater.

53b ⟨* 4⟩+≡ ◁53a 54a▷

```
static void write_index_use(void *closure, char *id, char *instance) {
  LineOut *info = (LineOut *) closure;
  info->line = emit_up_to(info->out, info->line, instance);
  fprintf(info->out, "@index use %s\n", id);
  info->line = emit_up_to(info->out, info->line, instance + strlen(id));
}
```

Defines:
`write_index_use`, used in chunks 53a and 54b.
Uses `strlen` 55c.

54a ⟨* 4⟩+≡ ◁53b 55a▷
```
static char *emit_up_to(FILE *f, char *s, char *limit) {
  if (s < limit) {
    char saved = *limit;
    *limit = 0;
    fprintf(f, "@text %s\n", s);
    *limit = saved;
    return limit;
  } else {
    return s;
  }
}
```

54b ⟨*local prototypes* 34b⟩+≡ (33a 50c) ◁52d
```
static void write_index_use(void *closure, char *id, char *instance);
static char *emit_up_to(FILE *f, char *s, char *limit);
```
Uses `write_index_use` 53b.

54c ⟨*complain about opening temp file and exit* 54c⟩≡ (52b)
```
errormsg(Fatal, "%s: couldn't open temporary file\n", progname);
```
Uses `errormsg` 70a 70b.

54d ⟨*complain about writing temp file and exit* 54d⟩≡ (52b)
```
errormsg(Fatal, "%s: error writing temporary file\n", progname);
```
Uses `errormsg` 70a 70b.

3.2 An Efficient String Matcher (by Preston Briggs)

3.2.1 Introduction

The obvious approach to this problem would be quite expensive for large documents; however, there is an interesting paper describing an efficient solution [1].

Boilerplate

55a ⟨* 4⟩+≡ ◁54a 64a▷

```
static char rcsid[] = "$Id: recognize.nw,v 1.10 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
⟨Include files 55c⟩
⟨header 18b⟩
⟨Type definitions 56e⟩
⟨Prototypes 63b⟩
⟨Function definitions 58a⟩
```

55b ⟨*header* 18b⟩+≡ (55a) ◁47a 64b▷

```
⟨Exported type definitions 56a⟩
⟨Exported prototypes 55e⟩
```

55c ⟨*Include files* 55c⟩≡ (55a) 55d▷

```
#include <string.h>
```

Defines:
`strchr`, used in chunk 63a.
`strcpy`, used in chunks 29, 30a, 60c, and 72b.
`strlen`, used in chunks 2, 6a, 14c, 19c, 22, 29, 30, 32, 35c, 40d, 52, 53b, 60c, 62c, 63a, 65a, and 72b.

55d ⟨*Include files* 55c⟩+≡ (55a) ◁55c

```
#include <stdlib.h>
```

Defines:
`calloc`, used in chunks 58–60.
`free`, used in chunks 20c, 21b, and 60b.
`malloc`, used in chunks 14c, 39a, 40b, 48b, 60, 61, 66a, and 72b.

External Interface The externally visible interface was designed by Norman Ramsey.

We assume that `alphanum` and `symbols` point to constant strings; *i.e.*, we don't bother to copy them into separately allocated space.

55e ⟨*Exported prototypes* 55e⟩≡ (55b) 56b▷

```
Recognizer new_recognizer(char *alphanum, char *symbols);
```

Defines:
`new_recognizer`, used in chunks 51a and 58b.
Uses `Recognizer` 56a 58b.

56a ⟨*Exported type definitions* 56a⟩≡ (55b) 56d ▷

```
typedef struct recognizer *Recognizer;
```

Defines:
`Recognizer`, used in chunks 50b, 55, 56, 59a, and 61–63.
Uses `recognizer` 57c.

A copy is made of the string pointed to by `id`. It won't hurt to add the same identifier multiple times to a given recognizer.

56b ⟨*Exported prototypes* 55e⟩+≡ (55b) ◁ 55e 56c ▷

```
void add_ident(Recognizer r, char *id);
void stop_adding(Recognizer r);
```

Defines:
`add_ident`, used in chunk 52.
`stop_adding`, used in chunks 51a and 52b.
Uses `Recognizer` 56a 58b.

56c ⟨*Exported prototypes* 55e⟩+≡ (55b) ◁ 56b

```
void search_for_ident(Recognizer r, char *input, Callback f, void *closure);
```

Defines:
`search_for_ident`, used in chunk 53a.
Uses `Callback` 56d and `Recognizer` 56a 58b.

`instance` is a pointer to the place within `input` that we saw the identifier.

56d ⟨*Exported type definitions* 56a⟩+≡ (55b) ◁ 56a

```
typedef void (*Callback) (void *closure, char *id, char *instance);
```

Defines:
`Callback`, used in chunks 56c and 62a.

3.2.2 Defining the Automata

56e ⟨*Type definitions* 56e⟩≡ (55a) 56f ▷

```
typedef struct goto_node Goto_Node;
typedef struct move_node Move_Node;
```

Defines:
`Goto_Node`, used in chunks 57–62.
`Move_Node`, used in chunks 57, 58a, 60a, and 61.
Uses `goto_node` 57b and `move_node` 57a.

56f ⟨*Type definitions* 56e⟩+≡ (55a) ◁ 56e 57a ▷

```
typedef struct name_node {
  struct name_node *next; /* points to the next name on the output list */
  char *name;
} Name_Node;
```

Defines:
`Name_Node`, used in chunks 57b and 60–62.

57a ⟨*Type definitions* 56e⟩+≡ (55a) ◁56f 57b▷
```
struct move_node {
  Move_Node *next;      /* points to the next node on the move list */
  Goto_Node *state;     /* the next state for this character */
  unsigned char c;
};
```
Defines:
`move_node`, used in chunk 56e.
Uses `Goto_Node` 56e, `Move_Node` 56e, and `state` 2.

57b ⟨*Type definitions* 56e⟩+≡ (55a) ◁57a 57c▷
```
struct goto_node {
  Name_Node *output;    /* list of words ending in this state */
  Move_Node *moves;     /* list of possible moves */
  Goto_Node *fail;      /* and where to go when no move fits */
  Goto_Node *next;      /* next goto node with same depth */
};
```
Defines:
`goto_node`, used in chunk 56e.
Uses `Goto_Node` 56e, `Move_Node` 56e, `Name_Node` 56f, and `state` 2.

57c ⟨*Type definitions* 56e⟩+≡ (55a) ◁57b
```
struct recognizer {
  Goto_Node *root[256]; /* might want 128, depending on the character set */
  char *alphas;
  char *syms;
  int max_depth;
  Goto_Node **depths; /* an array, max_depth long, of lists of goto_nodes,
                         created while adding ids, used while building
                         the failure functions */
};
```
Defines:
`alphas`, used in chunks 58b and 63a.
`depths`, used in chunks 58–61.
`max_depth`, used in chunks 58b, 60, and 61.
`recognizer`, used in chunks 56a and 58b.
`root`, used in chunks 25a, 26c, 30e, 35, 36c, 47–49, 59, 61, and 62b.
`syms`, used in chunks 58b and 63a.
Uses `Goto_Node` 56e.

A Utility Function We need a function that, given the current state and a character, will return the next state as directed by the "goto table." If there is no defined entry in the table, the function returns NULL.

58a ⟨*Function definitions* 58a⟩≡ (55a) 58b ▷

```
static Goto_Node *goto_lookup(unsigned char c, Goto_Node *g)
{
  Move_Node *m = g->moves;
  while (m && m->c != c)
    m = m->next;
  return m ? m->state : NULL;
}
```

Defines:
goto_lookup, used in chunks 59a, 61, and 62b.
Uses Goto_Node 56e, Move_Node 56e, and state 2.

3.2.3 Building the Automata

The max_depth should be initialized to be at least 2.

58b ⟨*Function definitions* 58a⟩+≡ (55a) ◁ 58a 59a ▷

```
Recognizer new_recognizer(char *alphanum, char *symbols)
{
  Recognizer r = (Recognizer) calloc(1, sizeof(struct recognizer));
  r->alphas = alphanum;
  r->syms = symbols;
  r->max_depth = 10;
  r->depths = (Goto_Node **) calloc(r->max_depth, sizeof(Goto_Node *));
  return r;
}
```

Defines:
Recognizer, used in chunks 50b, 55, 56, 59a, and 61–63.
Uses alphas 57c, calloc 55d, depths 57c, Goto_Node 56e, max_depth 57c, new_recognizer 55e, recognizer 57c, and syms 57c.

Building the Goto Table We assume `id` is at least 1 character long.

59a ⟨*Function definitions* 58a⟩+≡ (55a) ◁58b 61▷

```
void add_ident(Recognizer r, char *id)
{
  int depth = 2;
  char *p = id;
  unsigned char c = *p++;
  Goto_Node *q = r->root[c];
  if (!q)
    ⟨Create an entry for root[c] 59b⟩
  c = *p++;
  while (c) {
    Goto_Node *new = goto_lookup(c, q);
    if (!new)
      ⟨Create a new goto entry and attach to q's move list 60a⟩
    q = new;
    depth++;
    c = *p++;
  }
  ⟨Set q->output to id (if not already present) 60c⟩
}
```

Defines:
`add_ident`, used in chunk 52.
Uses `goto_lookup` 58a, `Goto_Node` 56e, `Recognizer` 56a 58b, and `root` 57c.

59b ⟨*Create an entry for* `root[c]` 59b⟩≡ (59a)

```
{
  q = (Goto_Node *) calloc(1, sizeof(Goto_Node));
  r->root[c] = q;
  q->next = r->depths[1];
  r->depths[1] = q;
}
```

Uses `calloc` 55d, `depths` 57c, `Goto_Node` 56e, and `root` 57c.

60a ⟨*Create a new goto entry and attach to* q*'s move list* 60a⟩≡ (59a)

```
{
  Move_Node *new_move = (Move_Node *) malloc(sizeof(Move_Node));
  new = (Goto_Node *) calloc(1, sizeof(Goto_Node));
  new_move->state = new;
  new_move->c = c;
  new_move->next = q->moves;
  q->moves = new_move;
  if (depth == r->max_depth)
    ⟨Double the size of the depths array 60b⟩
  new->next = r->depths[depth];
  r->depths[depth] = new;
}
```

Uses `calloc` 55d, `depths` 57c, `Goto_Node` 56e, `malloc` 55d, `max_depth` 57c, `Move_Node` 56e, and `state` 2.

60b ⟨*Double the size of the* depths *array* 60b⟩≡ (60a)

```
{
  int i;
  Goto_Node **new_depths = (Goto_Node **) calloc(2*depth, sizeof(Goto_Node *));
  r->max_depth = 2 * depth;
  for (i=0; i<depth; i++)
    new_depths[i] = r->depths[i];
  free(r->depths);
  r->depths = new_depths;
}
```

Uses `calloc` 55d, `depths` 57c, `free` 55d, `Goto_Node` 56e, and `max_depth` 57c.

60c ⟨*Set* q->output *to* id *(if not already present)* 60c⟩≡ (59a)

```
if (!q->output) {
  char *copy = malloc(strlen(id) + 1);
  strcpy(copy, id);
  q->output = (Name_Node *) malloc(sizeof(Name_Node));
  q->output->next = NULL;
  q->output->name = copy;
}
```

Uses `malloc` 55d, `Name_Node` 56f, `strcpy` 55c, and `strlen` 55c.

Building the Failure Functions After all the strings have been added to the goto table, we can construct the failure functions. It's going to be hard to explain this one.

61 ⟨*Function definitions* 58a⟩+≡ (55a) ◁59a 62a▷

```
void stop_adding(Recognizer r)
{
  int depth;
  for (depth=1; depth<r->max_depth; depth++) {
    Goto_Node *g = r->depths[depth];
    while (g) {
      Move_Node *m = g->moves;
      while (m) {
        unsigned char a = m->c;
        Goto_Node *s = m->state;
        Goto_Node *state = g->fail;
        while (state && !goto_lookup(a, state))
          state = state->fail;
        if (state)
          s->fail = goto_lookup(a, state);
        else
          s->fail = r->root[a];
        if (s->fail) {
          Name_Node *p = s->fail->output;
          while (p) {
            Name_Node *q = (Name_Node *) malloc(sizeof(Name_Node));
            q->name = p->name; /* depending on memory deallocation
                                  strategy, we may need to copy this */
            q->next = s->output;
            s->output = q;
            p = p->next;
          }
        }
        m = m->next;
      }
      g = g->next;
    }
  }
}
```

Defines:
  `stop_adding`, used in chunks 51a and 52b.

Uses `depths` 57c, `goto_lookup` 58a, `Goto_Node` 56e, `malloc` 55d, `max_depth` 57c, `Move_Node` 56e, `Name_Node` 56f, `Recognizer` 56a 58b, `root` 57c, and `state` 2.

3.2.4 Using the Automata

62a ⟨*Function definitions* 58a⟩+≡ (55a) ◁61 63a▷
```
void search_for_ident(Recognizer r, char *input, Callback f, void *closure)
{
  Goto_Node *state = NULL;
  char *current = input;
  unsigned char c = (unsigned char) *current++;
  while (c) {
    ⟨Goto the next state 62b⟩
    ⟨Perform the callback for any outputs 62c⟩
    c = *current++;
  }
}
```
Defines:
  `search_for_ident`, used in chunk 53a.
Uses `Callback` 56d, `Goto_Node` 56e, `Recognizer` 56a 58b, and `state` 2.

This is all complicated by my use of `NULL` to indicate the initial state. However, we get a nice speedup by using the `root` array instead of walking down the move list for every character.

62b ⟨*Goto the next state* 62b⟩≡ (62a)
```
{
  while (state && !goto_lookup(c, state))
    state = state->fail;
  state = state ? goto_lookup(c, state) : r->root[c];
}
```
Uses `goto_lookup` 58a, `root` 57c, and `state` 2.

We walk down the output list, calling `f` with each name that is not rejected (see the next section).

62c ⟨*Perform the callback for any outputs* 62c⟩≡ (62a)
```
{
  if (state) {
    Name_Node *p = state->output;
    while (p) {
      if (!reject_match(r, p->name, input, current))
        f(closure, p->name, current - strlen(p->name));
      p = p->next;
    }
  }
}
```
Uses `Name_Node` 56f, `reject_match` 63a, `state` 2, and `strlen` 55c.

Rejecting Matches A problem with simple substring matching is that the string "he" would match longer strings like "she" and "her." Norman Ramsey suggested examining the characters occuring immediately before and after a match and rejecting the match if it appears to be part of a longer token. Of course, the concept of *token* is language-dependent, so we may be occasionally mistaken. For the present, we'll consider the mechanism an experiment.

63a ⟨*Function definitions* 58a⟩+≡ (55a) ◁62a

```
int reject_match(Recognizer r, char *id, char *input, char *current)
{
  int len = strlen(id);
  char first = id[0];
  char last = id[len - 1];
  char next = *current;
  char prev = '\0';
  current = current - len - 1;
  if (input <= current)
    prev = *current;
  if (prev && strchr(r->alphas, first) && strchr(r->alphas, prev)) return 1;
  if (next && strchr(r->alphas, last ) && strchr(r->alphas, next)) return 1;
  if (prev && strchr(r->syms,   first) && strchr(r->syms,   prev)) return 1;
  if (next && strchr(r->syms,   last ) && strchr(r->syms,   next)) return 1;
  return 0;
}
```

Defines:
`reject_match`, used in chunks 62c and 63b.
Uses `alphas` 57c, `Recognizer` 56a 58b, `strchr` 55c, `strlen` 55c, and `syms` 57c.

Note we never reject a zero `prev` or `next`, since some implementations of `strchr` always return true when searching for a zero character.

We need a prototype for `reject_match`, since it's referenced before its definition.

63b ⟨*Prototypes* 63b⟩≡ (55a)

```
int reject_match(Recognizer r, char *id, char *input, char *current);
```

Uses `Recognizer` 56a 58b and `reject_match` 63a.

4 Utility functions

4.1 Line-oriented input

`getline` is reads in the next line, no matter how long it is, expanding tabs. I allocate more and more space for the line, until I've read a whole line or until I run out of space, whichever comes first. The buffer grows in size exponentially, so the cost in overhead is roughly the logarithm of the length of the longest line.

`getline` reuses the same internal buffers over and over, so subsequent calls destroy the results of previous calls.

64a ⟨* 4⟩+≡ ◁55a 67b▷

```
static char rcsid[] = "$Id: getline.nw,v 2.10 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#define START_SIZE 128                  /* initial buffer size */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "columns.h"
#include "errors.h"
#include "getline.h"

static char *buf1 = NULL, *buf2 = NULL; /* lines without, with tabs expanded */
static buf_size = START_SIZE;           /* size of both buffers if non-NULL */

⟨C functions 18d⟩
```

Defines:
`buf1`, used in chunks 64–66.
`buf2`, used in chunks 64–66.
`buf_size`, used in chunks 64–66.
`START_SIZE`, never used.
Uses `getline` 65a.

64b ⟨*header* 18b⟩+≡ (55a) ◁55b 67a▷

```
char *getline_expand (FILE *fp);
  /* grab a line in buffer, return new buffer or NULL for eof
     tabs in line are expanded according to tabsize */
char *getline (FILE *fp);
  /* grab a line in the buffer, return a new buffer or NULL for eof
     no expansion of tabs */
```

Uses `getline` 65a, `getline_expand` 65b, and `tabsize` 67a.

64c ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁46 65a▷

```
void new_buffers(void) {
    checkptr(buf1 = (char *) realloc(buf1, buf_size));
    checkptr(buf2 = (char *) realloc(buf2, buf_size));
}
```

Uses `buf1` 64a, `buf2` 64a, `buf_size` 64a, and `checkptr` 70a.

65a ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁64c 65b▷

```
char *getline (FILE *fp) {

    ⟨if buf1==NULL, allocate initial buffers 66a⟩

    buf1=fgets(buf1, buf_size, fp);
    if (buf1==NULL) return buf1; /* end of file */
    while (buf1[strlen(buf1)-1] != '\n') { /* failed to get whole line */
        buf_size *= 2;
        new_buffers();
        if (fgets(buf1+strlen(buf1),buf_size-strlen(buf1),fp)==NULL)
            return buf1; /* end of file */
    }
    return buf1;
}
```

Defines:
`getline`, used in chunks 4, 26c, 28, 29a, 50a, 52, 53a, 64, and 65b.
Uses `buf1` 64a, `buf_size` 64a, and `strlen` 55c.

65b ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁65a 69c▷

```
char *getline_expand (FILE *fp) {
    char *s, *t;
    int width;

    if (getline(fp)==NULL) return NULL;
    ⟨if columnwidth(buf1) > buf_size - 1, allocate new buffers 66b⟩
    s = buf1; t = buf2; width=0;
    while (*s)
        if (*s=='\t' && tabsize > 0) {
            do {
                *t++ = ' '; width++;
            } while (width % tabsize != 0);
            s++;
        } else {
            *t++ = *s++; width++;
        }
    *t='\0';
    return buf2;
}
```

Defines:
`getline_expand`, used in chunks 5 and 64b.
Uses `buf1` 64a, `buf2` 64a, `getline` 65a, and `tabsize` 67a.

66a ⟨*if* buf1==NULL*, allocate initial buffers* 66a⟩≡ (65a)
```
if (buf1==NULL) {
    checkptr(buf1 = (char *) malloc (buf_size));
    checkptr(buf2 = (char *) malloc (buf_size));
}
```
Uses buf1 64a, buf2 64a, buf_size 64a, checkptr 70a, and malloc 55d.

66b ⟨*if* columnwidth(buf1) > buf_size - 1*, allocate new buffers* 66b⟩≡ (65b)
```
if (columnwidth(buf1) > buf_size - 1) {
    while (columnwidth(buf1) > buf_size - 1) buf_size *= 2;
    new_buffers();
}
```
Uses buf1 64a, buf_size 64a, and columnwidth 67c.

4.2 Managing indentation and columns

These functions are used to help accumulate indentations across module boundaries by measuring the widths of various strings in columns. The size of things depends on `tabsize` as defined in columns.h. If `tabsize == 0`, tabs shouldn't be touched on input and won't be generated on output.

67a ⟨*header* 18b⟩+≡ (55a) ◁64b 69b▷

```
extern int tabsize;
extern int columnwidth (char *s);      /* number of columns string occupies */
extern int limitcolumn (char *s, int startcolumn);
                                   /* width of startcolumn blanks plus s */
extern void indent_for (int width, FILE *fp);
                                   /* indent to width; next char -> width+1 */
```

Defines:
  `tabsize`, used in chunks 16b, 24a, 25a, 33a, 37a, 64b, 65b, 67, and 68.
Uses `columnwidth` 67c, `indent_for` 68, and `limitcolumn` 67d.

67b ⟨* 4⟩+≡ ◁64a 67c▷

```
static char rcsid[] = "$Id: columns.nw,v 2.10 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include "columns.h"

int tabsize = 8;
```

Uses `tabsize` 67a.

67c ⟨* 4⟩+≡ ◁67b 67d▷

```
int columnwidth (char *s) {              /* width of a string in columns */
  return limitcolumn(s, 0);
}
```

Defines:
  `columnwidth`, used in chunks 66b and 67a.
Uses `limitcolumn` 67d.

67d ⟨* 4⟩+≡ ◁67c 68▷

```
int limitcolumn (char *s, int col) {
    while (*s) {
        col++;
        if (*s=='\t' && tabsize > 0) while (col % tabsize != 0) col++;
        s++;
    }
    return col;
}
```

Defines:
  `limitcolumn`, used in chunks 42b, 43b, and 67.
Uses `tabsize` 67a.

68 ⟨* 4⟩+≡ ◁67d 69a▷

```
  void indent_for (int width, FILE *fp) {
                                /* write whitespace [[width]] columns wide */
  /*fprintf(fp,"<%2d>",width); if (width>4) {fprintf(fp,"    "); width -= 4;}*/
      if (tabsize > 1)
          while (width >= tabsize) {
              putc('\t', fp);
              width -= tabsize;
          }
      while (width > 0) {
          putc(' ', fp);
          width--;
      }
  }
```

Defines:
`indent_for`, used in chunks 42b, 43b, and 67a.
Uses `tabsize` 67a.

4.3 Error messages

Here is a big set of functions that other places can call to issue error messages.

It might be worth implementing these as macros so they could indicate the file name and line number in the source of where they were called.

`errorlevel` is an integer that says how bad the worst error so far has been. `newerrorlevel` will update it conditionally. Some day we should hide it and give the world a function call that queries it.

69a ⟨* 4⟩+≡ ◁68 72b▷

```
static char rcsid[] = "$Id: errors.nw,v 2.12 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "errors.h"

⟨C functions 18d⟩
```

I track degrees of error as they occur.

69b ⟨*header* 18b⟩+≡ (55a) ◁67a 69d▷

```
enum errorlevel { Normal=0, Warning, Error, Fatal, Impossible };
extern enum errorlevel errorlevel;
extern int finalstage; /* set nonzero if this main() is a final stage */
extern char *progname; /* set to name of program if main() is a filter */
```

Defines:
`errorlevel`, used in chunks 16a, 24a, 33a, 51a, 69, and 70.

69c ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁65b 69e▷

```
enum errorlevel errorlevel = Normal;
int finalstage = 0;
char *progname = NULL;
```

Uses `errorlevel` 69b.

If an error occurred, one can exit with the proper notice on the filter:

69d ⟨*header* 18b⟩+≡ (55a) ◁69b 70a▷

```
extern void nowebexit(char *optional_msg);
```

69e ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁69c 70b▷

```
void nowebexit(char *msg) {
  if (!finalstage && errorlevel > Normal)
    printf("@fatal %s %s\n", progname != NULL ? progname : "a-noweb-filter",
           msg != NULL ? msg : "an unspecified error occurred");
  exit(errorlevel);
}
```

Uses `errorlevel` 69b.

The rest is just varying flavors of error messages.

70a ⟨*header* 18b⟩+≡ (55a) ◁69d 70c▷

```
void errormsg(enum errorlevel level, char *s, ...);
#define overflow(S) errormsg(Fatal,"Capacity exceeded: %s", S)
#define impossible(S) errormsg(Impossible, "This can't happen: %s", S)
#define checkptr(P) do { if (!(P)) overflow("memory"); } while (0)
```

Defines:
`checkptr`, used in chunks 14c, 39a, 40b, 48b, 64c, 66a, and 72b.
`errormsg`, used in chunks 16, 25, 30, 34–37, 43, 45b, 51, and 54.
`impossible`, used in chunks 21b, 31, 40d, 42a, and 46.
`overflow`, used in chunks 29e and 30a.
Uses `errorlevel` 69b.

70b ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁69e 70d▷

```
void errormsg(enum errorlevel level, char *s,...) {
    va_list args;                       /* see K&R, page 174 */
    va_start(args,s);
    ⟨set errorlevel and print s with args 70e⟩
    va_end(args);
    if (level >= Fatal)
        nowebexit(s);
}
```

Defines:
`errormsg`, used in chunks 16, 25, 30, 34–37, 43, 45b, 51, and 54.
Uses `errorlevel` 69b.

70c ⟨*header* 18b⟩+≡ (55a) ◁70a 72a▷

```
void errorat(char *filename, int lineno, enum errorlevel level, char *s, ...);
```

Uses `errorat` 70d and `errorlevel` 69b.

70d ⟨*C functions* 18d⟩+≡ (17 39b 64a 69a) ◁70b

```
void errorat(char *filename, int lineno, enum errorlevel level, char *s, ...) {
    va_list args;                       /* see K&R, page 174 */
    va_start(args,s);
    fprintf(stderr, "%s:%d: ", filename, lineno);
    ⟨set errorlevel and print s with args 70e⟩
    va_end(args);
    if (level >= Fatal)
        nowebexit(s);
}
```

Defines:
`errorat`, used in chunks 14f, 15b, 21a, 31d, and 70c.
Uses `errorlevel` 69b.

70e ⟨*set* `errorlevel` *and print* `s` *with* `args` 70e⟩≡ (70)

```
if (level > errorlevel)
    errorlevel = level;
vfprintf(stderr, s, args);
fprintf(stderr,"\n");
```

Uses `errorlevel` 69b.

Some older machines don't do `vfprintf`; here's an implementation:

71 ⟨*crock for non-ANSI machines* 71⟩≡

```
extern int _doprnt(const char *format, va_list args, FILE *stream);

int vfprintf(FILE *stream, const char *format, va_list arg) {
    return _doprnt(format,arg,stream);
}
```

72a ⟨*header* 18b⟩+≡ (55a) ◁70c

```
char *strsave (char *s);        /* returns a pointer to a fresh copy of s */
```

72b ⟨* 4⟩+≡ ◁69a

```
static char rcsid[] = "$Id: strsave.nw,v 2.7 2000/06/03 02:48:21 nr Exp nr $";
static char rcsname[] = "$Name: v2_9b $";
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "strsave.h"
#include "errors.h"

char *strsave (char *s) {
    char *t = malloc (strlen(s)+1);
    checkptr(t);
    strcpy(t,s);
    return t;
}
```

Uses `checkptr` 70a, `malloc` 55d, `strcpy` 55c, and `strlen` 55c.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

5 List of all chunks from all files

6 Index