CTWILL (Version 4.2 [TeX Live]) ⸺

1. Introduction. This is the CTWILL program by D. E. Knuth, based on CWEAVE by Silvio Levy and D. E. Knuth. It is also based on TWILL, a private WEB program that Knuth wrote to produce Volumes B and D of *Computers & Typesetting* in 1985. CTWILL was hacked together hastily in June, 1992, to generate pages for Knuth's book about the Stanford GraphBase, and updated even more hastily in March, 1993 to generate final copy for that book. The main idea was to extend CWEAVE so that "mini-indexes" could appear. No time was available to make CTWILL into a refined or complete system, nor even to fully update the program documentation below. Subsequent changes were made only to maintain compatibility with CWEAVE. Further information can be found in Knuth's article "Mini-indexes for literate programs," reprinted in *Digital Typography* (1999), 225–245.

The "banner line" defined here should be changed whenever CTWILL is modified. The version number parallels the corresponding version of CWEAVE.

Editor's Note: *Although CTWILL is based on* `cweave.w`, *new and modified material is incorporated all over the place, without taking special care to keep the original section numbering intact.*

Moreover, this heavily redacted version of `ctwill.pdf` *had to meddle with the section numbering even more, spreading tabular material over several sections and splitting long sections into smaller chunks in order to fix overful pages—both horizontally and vertically—, to make the overall appearance of the* CTWILL *documentation most pleasing to the readers' eyes.*

Please do not try to compare this `ctwill.pdf` *to the one created by* CWEAVE *instead of* CTWILL; *the section numbering will be even more "off" from* `cweave.w`. *Care has been taken to give a faithful overall rendering of* CTWILL's *code, though.*   *—Enjoy!*

#define *banner* `"This␣is␣CTWILL,␣Version␣4.2"`
            ▷ will be extended by the TeX Live *versionstring* ◁

⟨ Include files 7 ⟩
⟨ Preprocessor definitions ⟩
⟨ Common code for CWEAVE and CTANGLE 6 ⟩
⟨ Typedef declarations 25 ⟩
⟨ Private variables 26 ⟩
⟨ Predeclaration of procedures 11 ⟩

2.   CTWILL user manual.   Here is a sort of user manual for CTWILL—which is exactly like CWEAVE except that it produces much better documentation, for which you must work harder. As with CWEAVE, input comes from a source file `foo.w` and from an optional (but now almost mandatory) change file `foo.ch`; output goes to `foo.tex`, `foo.idx`, and `foo.scn`. Unlike CWEAVE, there is an additional output file, `foo.aux`, which records all nonexternal definitions. The `.aux` file also serves as an input file on subsequent runs. You should run CTWILL twice, once to prime the pump and once to get decent answers.

Moreover, you must run the output twice through TEX. (This double duplicity suggested the original name TWILL.) After 'tex foo' you will have output that looks like final pages except that the entries of mini-indexes won't be alphabetized. TEX will say 'This is the first pass', and it will produce a weird file called `foo.ref`. Say

<div align="center"><code>refsort < foo.ref > foo.sref</code></div>

and then another 'tex foo' will produce alphabetized output. While TEX runs it emits messages filled with numeric data, indicating how much space is consumed by each program section. If you can decipher these numbers (see `ctwimac.tex`), you can use them to fine-tune the page layout. You might be tempted to do fine tuning by editing `foo.tex` directly, but it's better to incorporate all changes into `foo.ch`.

The mini-indexes list identifiers that are used but not defined on each two-page spread. At the end of each section, CTWILL gives TEX a list of identifiers used in that section and information about where they are defined. The macros in `ctwimac.tex` figure out which identifiers should go in each mini-index, based on how the pages break. (Yes, those macros are pretty hairy.)

The information that CTWILL concocts from `foo.w` is not always correct. Sometimes you'll use an identifier that you don't want indexed; for example, your exposition might talk about $f(x)$ when you don't mean to refer to program variables f or x. Sometimes you'll use an identifier that's defined in a header file, unknown to CTWILL. Sometimes you'll define a single identifier in several different places, and CTWILL won't know which definition to choose. But all is not lost. CTWILL guesses right most of the time, and you can give it the necessary hints in other places via your change file.

If you think it's easy to write a completely automatic system that doesn't make CTWILL's mistakes and doesn't depend so much on change files, please do so.

CTWILL uses a very simple method to generate mini-index info. By understanding this method, you will understand how to fix it when things go wrong. Every identifier has a current "meaning," consisting of its abstract type and the number of the section in which it was most recently defined. For example, if your C program says '**char** ∗s' in section 3, the meaning of s gets changed to '**char** ∗, §3' while CTWILL is processing that section. If you refer to s in section 10, and if s hasn't been redefined in the meantime, and if section 10 doesn't wind up on the same two-page spread as section 3, the mini-index generated by section 10 will say "s: **char** ∗, §3."

---

versionstring, `<lib/lib.h>`.

3. The current meaning of every identifier is initially '`\uninitialized`'. Then `CTWILL` reads the `.aux` file for your job, if any; this `.aux` file contains all definitions of new meanings in the previous run, so it tells `CTWILL` about definitions that will be occurring in the future. If all identifiers have a unique definition, they will have a unique and appropriate meaning in the mini-indexes.

But some identifiers, like parameters to procedures, may be defined several times. Others may not be defined at all, because they are defined elsewhere and mentioned in header files included by the C preprocessor. To solve this problem, `CTWILL` provides mechanisms by which the current meaning of an identifier can be temporarily or permanently changed.

For example, the operation

```
@$s {FOO}3 \&{char} $*$@>
```

changes the current meaning of *s* to the TEX output of '`\&{char} $*$`' in section 3 of program FOO. All entries in the `.aux` file are expressed in the form of this `@$` operator; therefore you can use a text editor to paste such entries into a `.ch` file, whenever you want to tell `CTWILL` about definitions that are out of order or from other programs.

Before reading the `.aux` file, `CTWILL` actually looks for a file called `system.bux`, which will be read if present. And after `foo.aux`, a third possibility is `foo.bux`. The general convention is to put definitions of system procedures such as *printf* into `system.bux`, and to put definitions found in specifically foo-ish header files into `foo.bux`. Like the `.aux` files, `.bux` files should contain only `@$` specifications; this rule corresponds to the fact that 'bux' is the plural of '$'. The `.bux` files may also contain `@i` includes.

A companion operation `@%` signifies that all `@$` specifications from the present point to the beginning of the next section will define *temporary* meanings instead of permanent ones. Temporary meanings are placed into the mini-index of the current section; the permanent (current) meaning of the identifier will not be changed, nor will it appear in the mini-index of the section. If several temporary meanings are assigned to the same identifier in a section, all will appear in the mini-index. Each `@%` toggles the temporary/permanent convention; thus, after an even number of `@%` operations in a section, meanings specified by `@$` are permanent.

The operation `@-` followed by an identifier followed by `@>` specifies that the identifier should not generate a mini-index entry in the current section (unless, of course, a temporary meaning is assigned).

If `@-foo@>` appears in a section where a new permanent meaning is later defined by the semantics of C, the current meaning of *foo* will not be redefined; moreover, this current meaning, which may have been changed by `@$foo ...@>`, will also be written to the `.aux` file. Therefore you can control what `CTWILL` outputs; you can keep it from repeatedly contaminating the `.aux` file with things you don't like.

The meaning specified by `@$...@>` generally has four components: an identifier (followed by space), a program name (enclosed in braces), a section number (followed by space), and a TEX part. The TEX part must have fewer than 50 characters.

#define *max_tex_chars* 50 ▷ limit on the TEX part of a meaning ◁

4.   If the TEX part starts with '=', the mini-index entry will contain an equals sign instead of a colon; for example,

                    @$buf_size {PROG}10 =\T{200}@>

generates either '*buf_size* = 200, §10' or '*buf_size* = 200, PROG§10', depending on whether 'PROG' is or isn't the title of the current program. If the TEX part is '\zip', the mini-index entry will contain neither colon nor equals, just a comma. The program name and section number can also be replaced by a string. For example,

                    @$printf "<stdio.h>" \zip@>

will generate a mini-index entry like '*printf*, <stdio.h>.'.

   A special "proofmode" is provided so that you can check CTWILL's conclusions about cross-references. Run CTWILL with the flag +P, and TEX will produce a specially formatted document (*without* mini-indexes) in which you can check that your specifications are correct. You should always do this before generating mini-indexes, because mini-indexes can mask errors if page breaks are favorable but the errors might reveal themselves later after your program has changed. The proofmode output is much easier to check than the mini-indexes themselves.

   The control code @r or @R causes CTWILL to emit the TEX macro '\shortpage' just before starting the next section of the program. This causes the section to appear at the top of a right-hand page, if it would ordinarily have appeared near the bottom of a left-hand page and split across the pages. (The \shortpage macro is fragile and should be used only in cases where it will not mess up the output; insert it only when fine-tuning a set of pages.) If the next section is a starred section, the behavior is slightly different (but still fragile): The starred section will either be postponed to a left-hand page, if it normally would begin on a right-hand page, or vice versa. In other words, @r@* inverts the left/right logic.

   CTANGLE does not recognize the operations @$, @%, @-, and @r, which are unique to CTWILL. But that is no problem, since you use them only in change files set up for book publishing, which are quite different from the change files you set up for tangling.

   (End of user manual. We now resume the program for CWEAVE, with occasional outbursts of new code.)

---

printf, <stdio.h>.   *s*: **char** *, §2.

5. Introduction (continued). CWEAVE has a fairly straightforward outline. It operates in three phases: First it inputs the source file and stores cross-reference data, then it inputs the source once again and produces the TEX output file, finally it sorts and outputs the index.

Please read the documentation for common, the set of routines common to CTANGLE and CWEAVE, before proceeding further.

```
int main(int ac,       ▷ argument count ◁
    char **av)       ▷ argument values ◁
{
  argc ← ac; argv ← av; program ← ctwill; ⟨Set initial values 29⟩
  common_init(); ⟨Start TEX output 100⟩
  if (show_banner) cb_show_banner();       ▷ print a "banner line" ◁
  ⟨Store all the reserved words 44⟩
  phase_one();     ▷ read all the user's text and store the cross-references ◁
  phase_two();     ▷ read all the text again and translate it to TEX form ◁
  phase_three();     ▷ output the cross-reference index ◁
  if (tracing ≡ 2 ∧ ¬show_progress) new_line;
  return wrap_up();       ▷ and exit gracefully ◁
}
```

6. The next few sections contain stuff from the file "common.w" that must be included in both "ctangle.w" and "cweave.w". It appears in file "common.h", which is also included in "common.w" to propagate possible changes from this COMMON interface consistently.

First comes general stuff:

⟨Common code for CWEAVE and CTANGLE 6⟩ ≡
```
  typedef bool boolean;
  typedef uint8_t eight_bits;
  typedef uint16_t sixteen_bits;
  typedef enum {
    ctangle, cweave, ctwill
  } cweb;
  extern cweb program;     ▷ CTANGLE or CWEAVE or CTWILL? ◁
  extern int phase;     ▷ which phase are we in? ◁
```
See also sections 8, 9, 10, 12, 13, 15, 17, 18, and 310.

This code is used in section 1.

7. You may have noticed that almost all "strings" in the CWEB sources are placed in the context of the '_' macro. This is just a shortcut for the '*gettext*' function from the "GNU gettext utilities." For systems that do not have this library installed, we wrap things for neutral behavior without internationalization.

#define _(S) gettext(S)

⟨Include files 7⟩ ≡
#ifndef HAVE_GETTEXT
#define HAVE_GETTEXT 0
#endif
#if HAVE_GETTEXT

```
#include <libintl.h>
#else
#define gettext(A)   A
#endif
```

#include <ctype.h>   ▷ definition of *isalpha*, *isdigit* and so on ◁
#include <stdbool.h>   ▷ definition of **bool**, *true* and *false* ◁
#include <stddef.h>   ▷ definition of **ptrdiff_t** ◁
#include <stdint.h>   ▷ definition of **uint8_t** and **uint16_t** ◁
#include <stdlib.h>   ▷ definition of *getenv* and *exit* ◁
#include <stdio.h>   ▷ definition of *printf* and friends ◁
#include <string.h>   ▷ definition of *strlen*, *strcmp* and so on ◁

This code is used in section 1.

8.   Code related to the character set:

#define *and_and*   °4   ▷ '&&'; corresponds to MIT's ∧ ◁
#define *lt_lt*   °20   ▷ '<<'; corresponds to MIT's ⊂ ◁
#define *gt_gt*   °21   ▷ '>>'; corresponds to MIT's ⊃ ◁
#define *plus_plus*   °13   ▷ '++'; corresponds to MIT's ↑ ◁
#define *minus_minus*   °1   ▷ '--'; corresponds to MIT's ↓ ◁
#define *minus_gt*   °31   ▷ '->'; corresponds to MIT's → ◁
#define *non_eq*   °32   ▷ '!='; corresponds to MIT's ≠ ◁
#define *lt_eq*   °34   ▷ '<='; corresponds to MIT's ≤ ◁
#define *gt_eq*   °35   ▷ '>='; corresponds to MIT's ≥ ◁
#define *eq_eq*   °36   ▷ '=='; corresponds to MIT's ≡ ◁
#define *or_or*   °37   ▷ '||'; corresponds to MIT's ∨ ◁
#define *dot_dot_dot*   °16   ▷ '...'; corresponds to MIT's ⧚ ◁
#define *colon_colon*   °6   ▷ '::'; corresponds to MIT's ∈ ◁
#define *period_ast*   °26   ▷ '.*'; corresponds to MIT's ⊗ ◁
#define *minus_gt_ast*   °27   ▷ '->*'; corresponds to MIT's ⇆ ◁

⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
    extern char *section_text*[ ];   ▷ text being sought for ◁
    extern char *∗section_text_end*;   ▷ end of *section_text* ◁
    extern char *∗id_first*;   ▷ where the current identifier begins in the buffer ◁
    extern char *∗id_loc*;   ▷ just after the current identifier in the buffer ◁

---

argc: **int**, COMMON.W §73.
argv: **char** ∗∗, COMMON.W §73.
bool, <stdbool.h>.
cb_show_banner: **void** ( ),
    COMMON.W §99.
common_init: **void** ( ),
    COMMON.W §20.
ctangle: **int**, COMMON.W §18.
ctwill: **int**, COMMON.W §18.
cweave: **int**, COMMON.W §18.
exit, <stdlib.h>.
false, <stdbool.h>.
getenv, <stdlib.h>.
gettext, <libintl.h>.
id_first: **char** ∗,

COMMON.W §21.
id_loc: **char** ∗, COMMON.W §21.
isalpha, <ctype.h>.
isdigit, <ctype.h>.
new_line = macro, §18.
phase: **int**, COMMON.W §19.
phase_one: **static void** ( ), §80.
phase_three: **static void** ( ),
    §282.
phase_two: **static void** ( ),
    §255.
printf, <stdio.h>.
program: **int**, COMMON.W §18.
ptrdiff_t, <stddef.h>.

section_text: **char** [ ][ ],
    COMMON.W §21.
section_text_end: **char** ∗,
    COMMON.W §21.
show_banner = macro, §17.
show_progress = macro, §17.
strcmp, <string.h>.
strlen, <string.h>.
tracing: **static int**, §211.
true, <stdbool.h>.
uint16_t, <stdint.h>.
uint8_t, <stdint.h>.
wrap_up: **int** ( ),
    COMMON.W §68.

9.   Code related to input routines:

#define $xisalpha(c)$ $(isalpha((\textbf{eight\_bits})\ c) \wedge ((\textbf{eight\_bits})\ c < °200))$
#define $xisdigit(c)$ $(isdigit((\textbf{eight\_bits})\ c) \wedge ((\textbf{eight\_bits})\ c < °200))$
#define $xisspace(c)$ $(isspace((\textbf{eight\_bits})\ c) \wedge ((\textbf{eight\_bits})\ c < °200))$
#define $xislower(c)$ $(islower((\textbf{eight\_bits})\ c) \wedge ((\textbf{eight\_bits})\ c < °200))$
#define $xisupper(c)$ $(isupper((\textbf{eight\_bits})\ c) \wedge ((\textbf{eight\_bits})\ c < °200))$
#define $xisxdigit(c)$ $(isxdigit((\textbf{eight\_bits})\ c) \wedge ((\textbf{eight\_bits})\ c < °200))$

⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
  extern char $buffer[\,]$;   ▷ where each line of input goes ◁
  extern char $*buffer\_end$;   ▷ end of $buffer$ ◁
  extern char $*loc$;   ▷ points to the next character to be read from the buffer ◁
  extern char $*limit$;   ▷ points to the last character in the buffer ◁

10.   Code related to file handling:

  format $line$ x ▷ make $line$ an unreserved word ◁
#define $max\_include\_depth$ 10   ▷ maximum number of source files open simultaneously,
          not counting the change file ◁
#define $max\_file\_name\_length$ 1024
#define cur_file $file[include\_depth]$ ▷ current file ◁
#define cur_file_name $file\_name[include\_depth]$ ▷ current file name ◁
#define cur_line $line[include\_depth]$ ▷ number of current line in current file ◁
#define web_file $file[0]$ ▷ main source file ◁
#define web_file_name $file\_name[0]$ ▷ main source file name ◁

⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
  extern int $include\_depth$;   ▷ current level of nesting ◁
  extern FILE $*file[\,]$;   ▷ stack of non-change files ◁
  extern FILE $*change\_file$;   ▷ change file ◁
  extern char $file\_name[\,][max\_file\_name\_length]$;   ▷ stack of non-change file names ◁
  extern char $change\_file\_name[\,]$;   ▷ name of change file ◁
  extern char $check\_file\_name[\,]$;   ▷ name of $check\_file$ ◁
  extern int $line[\,]$;   ▷ number of current line in the stacked files ◁
  extern int $change\_line$;   ▷ number of current line in change file ◁
  extern int $change\_depth$;   ▷ where @y originated during a change ◁
  extern boolean $input\_has\_ended$;   ▷ if there is no more input ◁
  extern boolean $changing$;   ▷ if the current line is from $change\_file$ ◁
  extern boolean web_file_open;   ▷ if the web file is being read ◁

11.   ⟨ Predeclaration of procedures 11 ⟩ ≡
  extern boolean $get\_line(\textbf{void})$;   ▷ inputs the next line ◁
  extern void $check\_complete(\textbf{void})$;   ▷ checks that all changes were picked up ◁
  extern void $reset\_input(\textbf{void})$;   ▷ initialize to read the web file and change file ◁

See also sections 14, 16, 19, 35, 43, 52, 57, 76, 80, 82, 94, 97, 101, 106, 109, 120, 140, 144, 148, 206, 214, 219, 226, 235, 239, 255, 263, 273, 279, 282, 292, and 301.

This code is used in section 1.

12.   Code related to section numbers:

⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
  extern sixteen_bits $section\_count$;   ▷ the current section number ◁
  extern boolean $changed\_section[\,]$;   ▷ is the section changed? ◁

    extern boolean *change_pending*;   ▷ is a decision about change still unclear? ◁
    extern boolean *print_where*;   ▷ tells CTANGLE to print line and file info ◁

13.   Code related to identifier and section name storage:

#define *length*(*c*)   (**size_t**)((*c* + 1)⃗*byte_start* − (*c*)⃗*byte_start*)
          ▷ the length of a name ◁
#define *print_id*(*c*)   *term_write*((*c*)⃗*byte_start*, *length*((*c*)))   ▷ print identifier ◁
#define *llink*   *link*   ▷ left link in binary search tree for section names ◁
#define *rlink*   *dummy*.*Rlink*   ▷ right link in binary search tree for section names ◁
#define *root*   *name_dir*⃗*rlink*   ▷ the root of the binary search tree for section names ◁
⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
  typedef struct name_info {
    char ∗*byte_start*;   ▷ beginning of the name in *byte_mem* ◁
    struct name_info ∗*link*;
    union {
      struct name_info ∗*Rlink*;   ▷ right link in binary search tree for section names ◁
      char *Ilk*;   ▷ used by identifiers in CWEAVE only ◁
    } *dummy*;
    void ∗*equiv_or_xref*;   ▷ info corresponding to names ◁
  } **name_info**;   ▷ contains information about an identifier or section name ◁
  typedef name_info ∗**name_pointer**;   ▷ pointer into array of **name_infos** ◁
  typedef name_pointer ∗**hash_pointer**;
  extern char *byte_mem*[ ];   ▷ characters of names ◁
  extern char ∗*byte_mem_end*;   ▷ end of *byte_mem* ◁
  extern char ∗*byte_ptr*;   ▷ first unused position in *byte_mem* ◁
  extern name_info *name_dir*[ ];   ▷ information about names ◁
  extern name_pointer *name_dir_end*;   ▷ end of *name_dir* ◁
  extern name_pointer *name_ptr*;   ▷ first unused position in *name_dir* ◁
  extern name_pointer *hash*[ ];   ▷ heads of hash lists ◁
  extern hash_pointer *hash_end*;   ▷ end of *hash* ◁
  extern hash_pointer *h*;   ▷ index into hash-head array ◁

---

boolean = **bool**, §6.
buffer: **char** [ ],
  COMMON.W §22.
buffer_end: **char** ∗,
  COMMON.W §22.
byte_mem: **char** [ ],
  COMMON.W §43.
byte_mem_end: **char** ∗,
  COMMON.W §43.
byte_ptr: **char** ∗,
  COMMON.W §44.
change_depth: **int**,
  COMMON.W §25.
change_file: **FILE** ∗,
  COMMON.W §25.
change_file_name: **char** [ ],
  COMMON.W §25.
change_line: **int**,
  COMMON.W §25.
change_pending: **boolean**,
  COMMON.W §37.
changed_section: **boolean** [ ],
  COMMON.W §37.
changing: **boolean**,
  COMMON.W §25.

check_complete: **void** ( ),
  COMMON.W §42.
check_file: **FILE** ∗,
  COMMON.W §83.
check_file_name: **char** [ ],
  COMMON.W §73.
eight_bits = **uint8_t**, §6.
file: **FILE** ∗ [ ], COMMON.W §25.
file_name: **char** [ ][ ],
  COMMON.W §25.
get_line: **boolean** ( ),
  COMMON.W §38.
h: **hash_pointer**,
  COMMON.W §46.
hash: **name_pointer** [ ],
  COMMON.W §46.
hash_end: **hash_pointer**,
  COMMON.W §46.
include_depth: **int**,
  COMMON.W §25.
input_has_ended: **boolean**,
  COMMON.W §25.
isalpha, <ctype.h>.
isdigit, <ctype.h>.

islower, <ctype.h>.
isspace, <ctype.h>.
isupper, <ctype.h>.
isxdigit, <ctype.h>.
limit: **char** ∗, COMMON.W §22.
line: **int** [ ], COMMON.W §25.
loc: **char** ∗, COMMON.W §22.
name_dir: **name_info** [ ],
  COMMON.W §43.
name_dir_end: **name_pointer**,
  COMMON.W §43.
name_ptr: **name_pointer**,
  COMMON.W §44.
print_where: **boolean**,
  COMMON.W §37.
reset_input: **void** ( ),
  COMMON.W §35.
section_count: **sixteen_bits**,
  COMMON.W §37.
sixteen_bits = **uint16_t**, §6.
size_t, <stddef.h>.
term_write = macro ( ), §18.
web_file_open: **boolean**,
  COMMON.W §25.

14.   ⟨ Predeclaration of procedures 11 ⟩ +≡
　extern boolean *names_match*(**name_pointer**, **const char** ∗, **size_t**, **eight_bits**);
　extern **name_pointer** *id_lookup*(**const char** ∗, **const char** ∗, **char**);
　　▷ looks up a string in the identifier table ◁
　extern **name_pointer** *section_lookup*(**char** ∗, **char** ∗, **int**);　　▷ finds section name ◁
　extern **void** *init_node*(**name_pointer**);
　extern **void** *init_p*(**name_pointer**, **eight_bits**);
　extern **void** *print_prefix_name*(**name_pointer**);
　extern **void** *print_section_name*(**name_pointer**);
　extern **void** *sprint_section_name*(**char** ∗, **name_pointer**);

15.   Code related to error handling:

#define *spotless*　0　　▷ *history* value for normal jobs ◁
#define *harmless_message*　1　　　▷ *history* value when non-serious info was printed ◁
#define *error_message*　2　　　▷ *history* value when an error was noted ◁
#define *fatal_message*　3　　　▷ *history* value when we had to stop prematurely ◁
#define *mark_harmless*
　　　　{
　　　　　if (*history* ≡ *spotless*) *history* ← *harmless_message*;
　　　　}
#define *mark_error*　*history* ← *error_message*
#define *confusion*(*s*)　*fatal*(_("!␣This␣can't␣happen:␣"), *s*)
⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
　extern **int** *history*;　　▷ indicates how bad this run was ◁

16.   ⟨ Predeclaration of procedures 11 ⟩ +≡
　extern **int** *wrap_up*(**void**);　　▷ indicate *history* and exit ◁
　extern **void** *err_print*(**const char** ∗);　　▷ print error message and context ◁
　extern **void** *fatal*(**const char** ∗, **const char** ∗);　　▷ issue error message and die ◁
　extern **void** *overflow*(**const char** ∗);　　▷ succumb because a table has overflowed ◁

17.   Code related to command line arguments:

#define *show_banner*　*flags*['b']　　▷ should the banner line be printed? ◁
#define *show_progress*　*flags*['p']　　▷ should progress reports be printed? ◁
#define *show_stats*　*flags*['s']　　▷ should statistics be printed at end of run? ◁
#define *show_happiness*　*flags*['h']　　▷ should lack of errors be announced? ◁
#define *temporary_output*　*flags*['t']　　▷ should temporary output take precedence? ◁
#define *make_xrefs*　*flags*['x']　　▷ should cross references be output? ◁
⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
　extern **int** *argc*;　　▷ copy of *ac* parameter to *main* ◁
　extern **char** ∗∗*argv*;　　▷ copy of *av* parameter to *main* ◁
　extern **char** *C_file_name*[ ];　　▷ name of *C_file* ◁
　extern **char** *tex_file_name*[ ];　　▷ name of *tex_file* ◁
　extern **char** *idx_file_name*[ ];　　▷ name of *idx_file* ◁
　extern **char** *scn_file_name*[ ];　　▷ name of *scn_file* ◁
　extern **boolean** *flags*[ ];　　▷ an option for each 7-bit code ◁
　extern **const char** ∗*use_language*;　　▷ prefix to ctwimac.tex in TEX output ◁

18.   Code related to output:

#define *update_terminal*   *fflush*(*stdout*)   ▷ empty the terminal output buffer ◁
#define *new_line*   *putchar*(*'*\n*'*)
#define *putxchar*   *putchar*
#define *term_write*(*a*, *b*)   *fflush*(*stdout*), *fwrite*(*a*, **sizeof**(**char**), *b*, *stdout*)
#define *C_printf*(*c*, *a*)   *fprintf*(*C_file*, *c*, *a*)
#define *C_putc*(*c*)   *putc*(*c*, *C_file*)   ▷ isn't C wonderfully consistent? ◁

⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
   extern FILE ∗*C_file*;   ▷ where output of CTANGLE goes ◁
   extern FILE ∗*tex_file*;   ▷ where output of CWEAVE goes ◁
   extern FILE ∗*idx_file*;   ▷ where index from CWEAVE goes ◁
   extern FILE ∗*scn_file*;   ▷ where list of sections from CWEAVE goes ◁
   extern FILE ∗*active_file*;   ▷ currently active file for CWEAVE output ◁
   extern FILE ∗*check_file*;   ▷ temporary output file ◁

19.   The procedure that gets everything rolling:

⟨ Predeclaration of procedures 11 ⟩ +≡
   extern void *common_init*(**void**);
   extern void *print_stats*(**void**);
   extern void *cb_show_banner*(**void**);

20.   The following parameters were sufficient in the original WEB to handle TEX, so they should be sufficient for most applications of CWEB.

#define *max_bytes*   1000000
         ▷ the number of bytes in identifiers, index entries, and section names ◁
#define *max_toks*   1000000   ▷ number of bytes in compressed C code ◁
#define *max_names*   10239
         ▷ number of identifiers, strings, section names; must be less than 10240 ◁
#define *max_sections*   4000   ▷ greater than the total number of sections ◁
#define *max_texts*   10239   ▷ number of replacement texts, must be less than 10240 ◁
#define *longest_name*   10000
         ▷ file and section names and section texts shouldn't be longer than this ◁
#define *stack_size*   500   ▷ number of simultaneous levels of macro expansion ◁
#define *buf_size*   1000   ▷ maximum length of input line, plus one ◁
#define *long_buf_size*   (*buf_size* + *longest_name*)   ▷ for CWEAVE ◁

21.   End of COMMON interface.

---

22. The following parameters were sufficient in the original `WEAVE` to handle TEX, so they should be sufficient for most applications of `CWEAVE`.

#define *line_length* 80
     ▷ lines of TEX output have at most this many characters; should be less than 256 ◁
#define *max_refs* 65535   ▷ number of cross-references; must be less than 65536 ◁
#define *max_texts* 10239
     ▷ number of phrases in C texts being parsed; must be less than 10240 ◁
#define *max_scraps* 10000   ▷ number of tokens in C texts being parsed ◁

23.   Data structures exclusive to CWEAVE.   As explained in common.w, the field of a **name_info** structure that contains the *rlink* of a section name is used for a completely different purpose in the case of identifiers. It is then called the *ilk* of the identifier, and it is used to distinguish between various types of identifiers, as follows:

normal and *func_template* identifiers are part of the C program that will appear in italic type (or in typewriter type if all uppercase).

custom identifiers are part of the C program that will be typeset in special ways.

roman identifiers are index entries that appear after @^ in the CWEB file.

wildcard identifiers are index entries that appear after @: in the CWEB file.

typewriter identifiers are index entries that appear after @. in the CWEB file.

alfop, . . . , *template_like* identifiers are C or C++ reserved words whose *ilk* explains how they are to be treated when C code is being formatted.

#**define** *ilk*   *dummy.Ilk*
#**define** *normal*   0   ▷ ordinary identifiers have *normal* ilk ◁
#**define** *roman*   1   ▷ normal index entries have *roman* ilk ◁
#**define** *wildcard*   2   ▷ user-formatted index entries have *wildcard* ilk ◁
#**define** *typewriter*   3   ▷ 'typewriter type' entries have *typewriter* ilk ◁
#**define** *abnormal*(*a*)   (*a*→*ilk* > *typewriter*)   ▷ tells if a name is special ◁
#**define** *func_template*   4   ▷ identifiers that can be followed by optional template ◁
#**define** *custom*   5   ▷ identifiers with user-given control sequence ◁
#**define** *alfop*   22   ▷ alphabetic operators like **and** or **not_eq** ◁
#**define** *else_like*   26   ▷ **else** ◁
#**define** *public_like*   40   ▷ **public**, **private**, **protected** ◁
#**define** *operator_like*   41   ▷ **operator** ◁
#**define** *new_like*   42   ▷ **new** ◁
#**define** *catch_like*   43   ▷ **catch** ◁
#**define** *for_like*   45   ▷ **for**, **switch**, **while** ◁
#**define** *do_like*   46   ▷ **do** ◁
#**define** *if_like*   47   ▷ **if**, **ifdef**, **endif**, **pragma**, . . . ◁
#**define** *delete_like*   48   ▷ **delete** ◁
#**define** *raw_ubin*   49   ▷ '&' or '*' when looking for **const** following ◁
#**define** *const_like*   50   ▷ **const**, **volatile** ◁
#**define** *raw_int*   51   ▷ **int**, **char**, . . . ; also structure and class names ◁
#**define** *int_like*   52   ▷ same, when not followed by left parenthesis or :: ◁
#**define** *case_like*   53   ▷ **case**, **return**, **goto**, **break**, **continue** ◁
#**define** *sizeof_like*   54   ▷ **sizeof** ◁
#**define** *struct_like*   55   ▷ **struct**, **union**, **enum**, **class** ◁
#**define** *typedef_like*   56   ▷ **typedef** ◁
#**define** *define_like*   57   ▷ **define** ◁
#**define** *template_like*   58   ▷ **template** ◁

---

dummy: **union**, §13.   **name_info** = **struct**   *rlink* = macro, §13.
Ilk: **char**, §13.   **name_info**, §13.

24.   We keep track of the current section number in *section_count*, which is the total number of sections that have started. Sections which have been altered by a change file entry have their *changed_section* flag turned on during the first phase—NOT!

25.   The other large memory area in CWEAVE keeps the cross-reference data. All uses of the name p are recorded in a linked list beginning at $p \rightarrow xref$, which points into the *xmem* array. The elements of *xmem* are structures consisting of an integer, *num*, and a pointer *xlink* to another element of *xmem*. If $x \leftarrow p \rightarrow xref$ is a pointer into *xmem*, the value of $x \rightarrow num$ is either a section number where p is used, or *cite_flag* plus a section number where p is mentioned, or *def_flag* plus a section number where p is defined; and $x \rightarrow xlink$ points to the next such cross-reference for p, if any. This list of cross-references is in decreasing order by section number. The next unused slot in *xmem* is *xref_ptr*. The linked list ends at $\& xmem[0]$.

The global variable *xref_switch* is set either to *def_flag* or to zero, depending on whether the next cross-reference to an identifier is to be underlined or not in the index. This switch is set to *def_flag* when @! or @d is scanned, and it is cleared to zero when the next identifier or index entry cross-reference has been made. Similarly, the global variable *section_xref_switch* is either *def_flag* or *cite_flag* or zero, depending on whether a section name is being defined, cited or used in C text.

⟨ Typedef declarations 25 ⟩ ≡
  typedef struct xref_info {
    sixteen_bits *num*;   ▷ section number plus zero or *def_flag* ◁
    struct xref_info ∗*xlink*;   ▷ pointer to the previous cross-reference ◁
  } **xref_info**;
  typedef xref_info ∗**xref_pointer**;

See also sections 27, 39, 136, and 232.

This code is used in section 1.

26.   ⟨ Private variables 26 ⟩ ≡
  static xref_info *xmem*[*max_refs*];   ▷ contains cross-reference information ◁
  static xref_pointer *xmem_end* ← *xmem* + *max_refs* − 1;
  static xref_pointer *xref_ptr*;   ▷ the largest occupied position in *xmem* ◁
  static sixteen_bits *xref_switch*, *section_xref_switch*;   ▷ either zero or *def_flag* ◁

See also sections 28, 40, 46, 49, 55, 58, 60, 78, 87, 92, 96, 117, 137, 143, 211, 233, 238, 253, 256, 261, 267, 284, 287, 289, and 298.

This code is used in section 1.

27.   CTWILL also has special data structures to keep track of current and temporary meanings. These structures were not designed for maximum efficiency; they were designed to be easily grafted into CWEAVE's existing code without major surgery.

#**define** $max\_meanings$ 100   ▷ max temporary meanings per section ◁
#**define** max_titles 100   ▷ max distinct program or header names in meanings ◁

⟨ Typedef declarations 25 ⟩ +≡
  typedef struct {
    name_pointer id;   ▷ identifier whose meaning is being recorded ◁
    sixteen_bits $prog\_no$;   ▷ title of program or header in which defined ◁
    sixteen_bits sec_no;   ▷ section number in which defined ◁
    char $tex\_part[max\_tex\_chars]$;   ▷ TEX part of meaning ◁
  } **meaning_struct**;

28.   ⟨ Private variables 26 ⟩ +≡
  struct perm_meaning {
    meaning_struct $perm$;   ▷ current meaning of an identifier ◁
    int $stamp$;   ▷ last section number in which further output suppressed ◁
    struct perm_meaning $*link$;   ▷ another meaning to output in this section ◁
  } $cur\_meaning[max\_names]$;   ▷ the current "permanent" meanings ◁
  static struct perm_meaning $*top\_usage$;   ▷ first meaning to output in this section ◁
  static meaning_struct $temp\_meaning\_stack[max\_meanings]$;
    ▷ the current "temporary" meanings ◁
  static meaning_struct $*temp\_meaning\_ptr$;
    ▷ first available slot in $temp\_meaning\_stack$ ◁
  static meaning_struct $*max\_temp\_meaning\_ptr$;   ▷ its maximum value so far ◁
  static name_pointer $title\_code[max\_titles]$;   ▷ program names seen so far ◁
  static name_pointer $*title\_code\_ptr$;   ▷ first available slot in $title\_code$ ◁
  static char $ministring\_buf[max\_tex\_chars]$;   ▷ TEX code being generated ◁
  static char $*ministring\_ptr$;   ▷ first available slot in $ministring\_buf$ ◁
  static boolean ms_mode;   ▷ are we outputting to $ministring\_buf$? ◁

29.   ⟨ Set initial values 29 ⟩ ≡
  $max\_temp\_meaning\_ptr \leftarrow temp\_meaning\_stack$; $title\_code\_ptr \leftarrow title\_code$;
  $ms\_mode \leftarrow false$;
See also sections 34, 41, 50, 69, 103, 118, 138, 185, 229, 234, 288, 290, 305, 306, and 311.
This code is used in section 5.

$boolean =$ **bool**, §6.
$changed\_section$: **boolean** [ ],
  COMMON.W §37.
$cite\_flag = 10240$, §34.
$def\_flag =$ macro, §34.

$false$, `<stdbool.h>`.
$max\_names = 10239$, §20.
$max\_refs = 65535$, §22.
$max\_tex\_chars = 50$, §3.
name_pointer = **name_info**

$*$, §13.
$section\_count$: **sixteen_bits**,
  COMMON.W §37.
sixteen_bits = **uint16_t**, §6.
$xref =$ macro, §34.

30.   Here's a routine that converts a program title from the buffer into an internal number for the *prog_no* field of a meaning. It advances *loc* past the title found.

```
static sixteen_bits title_lookup(void)
{
    char *first, *last;      ▷ boundaries ◁
    int balance;       ▷ excess of left over right ◁
    register name_pointer *p;

    first ← loc;
    if (*loc ≡ '"') {
        while (++loc ≤ limit ∧ *loc ≠ '"')
            if (*loc ≡ '\\') loc++;
    }
    else if (*loc ≡ '{') {
        balance ← 1;
        while (++loc ≤ limit) {
            if (*loc ≡ '}' ∧ −−balance ≡ 0) break;
            if (*loc ≡ '{') balance++;
        }
    }
    else err_print(_("! ␣Title␣should␣be␣enclosed␣in␣braces␣or␣doublequotes"));
    last ← ++loc;
    if (last > limit) err_print(_("! ␣Title␣name␣didn't␣end"));
    if (title_code_ptr ≡ &title_code[max_titles]) overflow(_("titles"));
    *title_code_ptr ← id_lookup(first, last, title);
    for (p ← title_code; ; p++)
        if (*p ≡ *title_code_ptr) break;
    if (p ≡ title_code_ptr) title_code_ptr++;
    return p − title_code;
}
```

31.   ⟨ Give a default title to the program, if necessary 31 ⟩ ≡
```
if (title_code_ptr ≡ title_code) {       ▷ no \def\title found in limbo ◁
    char *saveloc ← loc, *savelimit ← limit;

    loc ← limit + 1;  limit ← loc;  *limit++ ← '{';
    memcpy(limit, tex_file_name, strlen(tex_file_name) − 4);
    limit += strlen(tex_file_name) − 4;  *limit++ ← '}';  title_lookup();  loc ← saveloc;
    limit ← savelimit;
}
```
This code is used in section 79.

32.   The *new_meaning* routine changes the current "permanent meaning" when an identifier is redeclared. It gets the *tex_part* from *ministring_buf*.

> **static void** *new_meaning*(**name_pointer** *p*)
> {
>   **struct perm_meaning** *∗q ← p − name_dir + cur_meaning*;
>   *ms_mode ← false*;
>   **if** (*q⃗stamp ≠ section_count*) {
>   **if** (∗(*ministring_ptr* − 1) ≡ '␣') *ministring_ptr* −−;
>   **if** (*ministring_ptr ≥* &*ministring_buf*[*max_tex_chars*])
>   *strcpy*(*ministring_buf*, "\\zip");   ▷ ignore *tex_part* if too long ◁
>   **else** ∗*ministring_ptr ←* '\0';
>   *q⃗perm.prog_no ←* 0;   ▷ *q⃗perm.id ← p* ◁
>   *q⃗perm.sec_no ← section_count*; *strcpy*(*q⃗perm.tex_part*, *ministring_buf*);
>   }
>   ⟨Write the new meaning to the .aux file 33⟩;
> }

33.   ⟨Write the new meaning to the .aux file 33⟩ ≡
> { **int** *n ← q⃗perm.prog_no*;
>   *fprintf*(*aux_file*, "@$%.*s␣%.*s", (**int**)((*p* + 1)⃗*byte_start − p⃗byte_start*), *p⃗byte_start*,
>   (**int**)((*title_code*[*n*] + 1)⃗*byte_start − title_code*[*n*]⃗*byte_start*),
>   *title_code*[*n*]⃗*byte_start*);
>   **if** (∗(*title_code*[*n*]⃗*byte_start*) ≡ '{') *fprintf*(*aux_file*, "%d", *q⃗perm.sec_no*);
>   *fprintf*(*aux_file*, "␣%s@>\n", *q⃗perm.tex_part*);
> }

This code is used in section 32.

34.   A section that is used for multi-file output (with the @(feature) has a special first cross-reference whose *num* field is *file_flag*.

#define *file_flag*   (3 ∗ *cite_flag*)
#define *def_flag*   (2 ∗ *cite_flag*)
#define *cite_flag*   10240   ▷ must be strictly larger than *max_sections* ◁
#define *xref*   *equiv_or_xref*
⟨Set initial values 29⟩ +≡
> *xref_ptr ← xmem*; *init_node*(*name_dir*); *xref_switch ←* 0; *section_xref_switch ←* 0;
> *xmem⃗num ←* 0;   ▷ sentinel value ◁

␣ = macro ( ), §7.
aux_file: **static FILE** ∗, §253.
byte_start: **char** ∗, §13.
cur_meaning: **struct**
  perm_meaning[ ], §28.
equiv_or_xref: **void** ∗, §13.
err_print: **void** ( ),
  COMMON.W §66.
false, <stdbool.h>.
fprintf, <stdio.h>.
id: **name_pointer**, §27.
id_lookup: **name_pointer** ( ),
  COMMON.W §48.
init_node: **void** ( ), §42.
limit: **char** ∗, COMMON.W §22.
loc: **char** ∗, COMMON.W §22.
max_sections = 4000, §20.
max_tex_chars = 50, §3.
max_titles = 100, §27.
memcpy, <string.h>.

ministring_buf: **static char**
  [ ], §28.
ministring_ptr: **static char** ∗,
  §28.
ms_mode: **static boolean**,
  §28.
name_dir: **name_info** [ ],
  COMMON.W §43.
name_pointer = **name_info**
  ∗, §13.
num: **sixteen_bits**, §25.
overflow: **void** ( ),
  COMMON.W §71.
perm: **meaning_struct**, §28.
perm_meaning: **struct**, §28.
prog_no: **sixteen_bits**, §27.
sec_no: **sixteen_bits**, §27.
section_count: **sixteen_bits**,
  COMMON.W §37.
section_xref_switch: **static**

sixteen_bits, §26.
sixteen_bits = **uint16_t**, §6.
stamp: **int**, §28.
strcpy, <string.h>.
strlen, <string.h>.
tex_file_name: **char** [ ],
  COMMON.W §73.
tex_part: **char** [ ], §27.
title = 63, §117.
title_code: **static**
  name_pointer [ ], §28.
title_code_ptr: **static**
  name_pointer ∗, §28.
xmem: **static xref_info** [ ],
  §26.
xref_ptr: **static xref_pointer**,
  §26.
xref_switch: **static**
  sixteen_bits, §26.

35. A new cross-reference for an identifier is formed by calling *new_xref*, which discards duplicate entries and ignores non-underlined references to one-letter identifiers or C's reserved words.

If the user has sent the *no_xref* flag (the `-x` option of the command line), it is unnecessary to keep track of cross-references for identifiers. If one were careful, one could probably make more changes around section 100 to avoid a lot of identifier looking up.

#define *append_xref*(c)
      if (*xref_ptr* ≡ *xmem_end*) *overflow*(_("cross-reference"));
      else (++*xref_ptr*)⃗*num* ← c;
#define *no_xref* (¬*make_xrefs*)
#define *is_tiny*(p) ((p + 1)⃗*byte_start* ≡ (p)⃗*byte_start* + 1)
#define *unindexed*(a) (a < *res_wd_end* ∧ a⃗*ilk* ≥ *custom*)
      ▷ tells if uses of a name are to be indexed ◁

⟨Predeclaration of procedures 11⟩ +≡
  static void *new_xref*(**name_pointer**);
  static void *new_section_xref*(**name_pointer**);
  static void *set_file_flag*(**name_pointer**);

36.   **static void** *new_xref*(**name_pointer** p)
  {
    xref_pointer q;   ▷ pointer to previous cross-reference ◁
    sixteen_bits m, n;   ▷ new and previous cross-reference value ◁
    if (*no_xref*) **return**;
    if ((*unindexed*(p) ∨ *is_tiny*(p)) ∧ *xref_switch* ≡ 0) **return**;
    m ← *section_count* + *xref_switch*; *xref_switch* ← 0; q ← (**xref_pointer**) p⃗*xref*;
    if (q ≠ *xmem*) {
      n ← q⃗*num*;
      if (n ≡ m ∨ n ≡ m + *def_flag*) **return**;
      else if (m ≡ n + *def_flag*) {
        q⃗*num* ← m; **return**;
      }
    }
    append_xref(m); *xref_ptr*⃗*xlink* ← q; *update_node*(p);
  }

37. The cross-reference lists for section names are slightly different. Suppose that a section name is defined in sections $m_1, \ldots, m_k$, cited in sections $n_1, \ldots, n_l$, and used in sections $p_1, \ldots, p_j$. Then its list will contain $m_1 + def\_flag, \ldots, m_k + def\_flag$, $n_1 + cite\_flag, \ldots, n_l + cite\_flag, p_1, \ldots, p_j$, in this order.

Although this method of storage takes quadratic time with respect to the length of the list, under foreseeable uses of CWEAVE this inefficiency is insignificant.

  static void *new_section_xref*(**name_pointer** p)
  {
    xref_pointer q, r;   ▷ pointers to previous cross-references ◁
    q ← (**xref_pointer**) p⃗*xref*; r ← *xmem*;
    if (q > *xmem*)
      while (q⃗*num* > *section_xref_switch*) {

$$r \leftarrow q; \quad q \leftarrow q\text{-}xlink;$$
$$\}$$
if $(r\text{-}num \equiv section\_count + section\_xref\_switch)$ **return**;
▷ don't duplicate entries ◁
$append\_xref\,(section\_count + section\_xref\_switch\,); \quad xref\_ptr\text{-}xlink \leftarrow q;$
$section\_xref\_switch \leftarrow 0;$
if $(r \equiv xmem)$ $update\_node\,(p);$
else $r\text{-}xlink \leftarrow xref\_ptr;$
$\}$

38.   The cross-reference list for a section name may also begin with *file_flag*. Here's how that flag gets put in.

static void $set\_file\_flag(\mathbf{name\_pointer}\ p)$
$\{$
xref_pointer $q;$

$q \leftarrow (\mathbf{xref\_pointer})\ p\text{-}xref;$
if $(q\text{-}num \equiv file\_flag)$ **return**;
$append\_xref\,(file\_flag); \quad xref\_ptr\text{-}xlink \leftarrow q; \quad update\_node\,(p);$
$\}$

39.   A third large area of memory is used for sixteen-bit 'tokens', which appear in short lists similar to the strings of characters in *byte_mem*. Token lists are used to contain the result of C code translated into TₑX form; further details about them will be explained later. A **text_pointer** variable is an index into *tok_start*.

⟨ Typedef declarations 25 ⟩ +≡
typedef sixteen_bits token;
typedef token ∗token_pointer;
typedef token_pointer ∗text_pointer;

---

40.   The first position of *tok_mem* that is unoccupied by replacement text is called
tok_ptr, and the first unused location of *tok_start* is called *text_ptr*. Thus, we usually
have *∗text_ptr ≡ tok_ptr*.

⟨ Private variables 26 ⟩ +≡
  static token *tok_mem*[*max_toks*];   ▷ tokens ◁
  static token_pointer *tok_mem_end* ← *tok_mem* + *max_toks* − 1;   ▷ end of *tok_mem* ◁
  static token_pointer *tok_ptr*;   ▷ first unused position in *tok_mem* ◁
  static token_pointer *max_tok_ptr*;   ▷ largest value of *tok_ptr* ◁
  static token_pointer *tok_start*[*max_texts*];   ▷ directory into *tok_mem* ◁
  static text_pointer *tok_start_end* ← *tok_start* + *max_texts* − 1;   ▷ end of *tok_start* ◁
  static text_pointer *text_ptr*;   ▷ first unused position in *tok_start* ◁
  static text_pointer *max_text_ptr*;   ▷ largest value of *text_ptr* ◁

41.   ⟨ Set initial values 29 ⟩ +≡
  tok_ptr ← *max_tok_ptr* ← *tok_mem* + 1;
  tok_start[0] ← *tok_start*[1] ← *tok_mem* + 1;
  text_ptr ← *max_text_ptr* ← *tok_start* + 1;

42.   Here are the three procedures needed to complete *id_lookup*:

  boolean *names_match*(**name_pointer** *p*,   ▷ points to the proposed match ◁
      const char *∗first*,   ▷ position of first character of string ◁
      size_t *l*,   ▷ length of identifier ◁
      eight_bits *t*)   ▷ desired *ilk* ◁
  {
    if (*length*(*p*) ≠ *l*) **return** *false*;
    if (*p↱ilk* ≠ *t* ∧ ¬(*t* ≡ *normal* ∧ *abnormal*(*p*))) **return** *false*;
    return ¬*strncmp*(*first*, *p↱byte_start*, *l*);
  }

  void *init_p*(**name_pointer** *p*, **eight_bits** *t*)
  {
    struct perm_meaning *∗q* ← *p* − *name_dir* + *cur_meaning*;
    p↱ilk ← *t*; *init_node*(*p*); *q↱stamp* ← 0; *q↱link* ← Λ; *q↱perm.id* ← *p*;
    q↱perm.prog_no ← *q↱perm.sec_no* ← 0; *strcpy*(*q↱perm.tex_part*, "\\uninitialized");
  }

  void *init_node*(**name_pointer** *p*)
  {
    p↱xref ← (**void** ∗) *xmem*;
  }

  static void *update_node*(**name_pointer** *p*)
  {
    p↱xref ← (**void** ∗) *xref_ptr*;
  }

43.   ⟨ Predeclaration of procedures 11 ⟩ +≡ **static void** *update_node*(**name_pointer** *p*);

44. We have to get C's reserved words into the hash table, and the simplest way to do this is to insert them every time CWEAVE is run. Fortunately there are relatively few reserved words. (Some of these are not strictly "reserved," but are defined in header files of the ISO Standard C Library.)

⟨ Store all the reserved words 44 ⟩ ≡
  id_lookup("and", Λ, *alfop*); *id_lookup*("and_eq", Λ, *alfop*);
  id_lookup("asm", Λ, *sizeof_like*); *id_lookup*("auto", Λ, *int_like*);
  id_lookup("bitand", Λ, *alfop*); *id_lookup*("bitor", Λ, *alfop*);
  id_lookup("bool", Λ, *raw_int*); *id_lookup*("break", Λ, *case_like*);
  id_lookup("case", Λ, *case_like*); *id_lookup*("catch", Λ, *catch_like*);
  id_lookup("char", Λ, *raw_int*); *id_lookup*("class", Λ, *struct_like*);
  id_lookup("clock_t", Λ, *raw_int*); *id_lookup*("compl", Λ, *alfop*);
  id_lookup("const", Λ, *const_like*); *id_lookup*("const_cast", Λ, *raw_int*);
  id_lookup("continue", Λ, *case_like*); *id_lookup*("default", Λ, *case_like*);
  id_lookup("define", Λ, *define_like*); *id_lookup*("defined", Λ, *sizeof_like*);
  id_lookup("delete", Λ, *delete_like*); *id_lookup*("div_t", Λ, *raw_int*);
  id_lookup("do", Λ, *do_like*); *id_lookup*("double", Λ, *raw_int*);
  id_lookup("dynamic_cast", Λ, *raw_int*); *id_lookup*("elif", Λ, *if_like*);
  id_lookup("else", Λ, *else_like*); *id_lookup*("endif", Λ, *if_like*);
  id_lookup("enum", Λ, *struct_like*); *id_lookup*("error", Λ, *if_like*);
  id_lookup("explicit", Λ, *int_like*); *id_lookup*("export", Λ, *int_like*);
  ext_loc ← *id_lookup*("extern", Λ, *int_like*) − *name_dir*; *id_lookup*("FILE", Λ, *raw_int*);
  id_lookup("float", Λ, *raw_int*); *id_lookup*("for", Λ, *for_like*);
  id_lookup("fpos_t", Λ, *raw_int*); *id_lookup*("friend", Λ, *int_like*);

See also section 45.

This code is used in section 5.

---

abnormal = macro ( ), §23.
alfop = 22, §23.
boolean = **bool**, §6.
byte_start: **char** ∗, §13.
case_like = 53, §23.
catch_like = 43, §23.
const_like = 50, §23.
cur_meaning: **struct**
  perm_meaning [ ], §28.
define_like = 57, §23.
delete_like = 48, §23.
do_like = 46, §23.
eight_bits = **uint8_t**, §6.
else_like = 26, §23.
ext_loc: **static sixteen_bits**,
  §46.
false, <stdbool.h>.
for_like = 45, §23.
id: **name_pointer**, §27.

id_lookup: **name_pointer** ( ),
  COMMON.W §48.
if_like = 47, §23.
ilk = macro, §23.
int_like = 52, §23.
length = macro ( ), §13.
link: **struct perm_meaning**
  ∗, §28.
max_texts = 10239, §22.
max_toks = 1000000, §20.
name_dir: **name_info** [ ],
  COMMON.W §43.
name_pointer = **name_info**
  ∗, §13.
normal = 0, §23.
perm: **meaning_struct**, §28.
perm_meaning: **struct**, §28.
prog_no: **sixteen_bits**, §27.

raw_int = 51, §23.
sec_no: **sixteen_bits**, §27.
size_t, <stddef.h>.
sizeof_like = 54, §23.
stamp: **int**, §28.
strcpy, <string.h>.
strncmp, <string.h>.
struct_like = 55, §23.
tex_part: **char** [ ], §27.
text_pointer = **token_pointer**
  ∗, §39.
token = **sixteen_bits**, §39.
token_pointer = **token** ∗, §39.
xmem: **static xref_info** [ ],
  §26.
xref = macro, §34.
xref_ptr: **static xref_pointer**,
  §26.

45. ⟨Store all the reserved words 44⟩ +≡
  id_lookup("goto", Λ, *case_like*); *id_lookup*("if", Λ, *if_like*);
  id_lookup("ifdef", Λ, *if_like*); *id_lookup*("ifndef", Λ, *if_like*);
  id_lookup("include", Λ, *if_like*); *id_lookup*("inline", Λ, *int_like*);
  int_loc ← *id_lookup*("int", Λ, *raw_int*) − *name_dir*; *id_lookup*("jmp_buf", Λ, *raw_int*);
  id_lookup("ldiv_t", Λ, *raw_int*); *id_lookup*("line", Λ, *if_like*);
  id_lookup("long", Λ, *raw_int*); *id_lookup*("mutable", Λ, *int_like*);
  id_lookup("namespace", Λ, *struct_like*); *id_lookup*("new", Λ, *new_like*);
  id_lookup("not", Λ, *alfop*); *id_lookup*("not_eq", Λ, *alfop*); *id_lookup*("NULL", Λ, *custom*);
  id_lookup("offsetof", Λ, *raw_int*); *id_lookup*("operator", Λ, *operator_like*);
  id_lookup("or", Λ, *alfop*); *id_lookup*("or_eq", Λ, *alfop*); *id_lookup*("pragma", Λ, *if_like*);
  id_lookup("private", Λ, *public_like*); *id_lookup*("protected", Λ, *public_like*);
  id_lookup("ptrdiff_t", Λ, *raw_int*); *id_lookup*("public", Λ, *public_like*);
  id_lookup("register", Λ, *int_like*); *id_lookup*("reinterpret_cast", Λ, *raw_int*);
  id_lookup("return", Λ, *case_like*); *id_lookup*("short", Λ, *raw_int*);
  id_lookup("sig_atomic_t", Λ, *raw_int*); *id_lookup*("signed", Λ, *raw_int*);
  id_lookup("size_t", Λ, *raw_int*); *id_lookup*("sizeof", Λ, *sizeof_like*);
  id_lookup("static", Λ, *int_like*); *id_lookup*("static_cast", Λ, *raw_int*);
  id_lookup("struct", Λ, *struct_like*); *id_lookup*("switch", Λ, *for_like*);
  id_lookup("template", Λ, *template_like*); *id_lookup*("this", Λ, *custom*);
  id_lookup("throw", Λ, *case_like*); *id_lookup*("time_t", Λ, *raw_int*);
  id_lookup("try", Λ, *else_like*); *id_lookup*("typedef", Λ, *typedef_like*);
  id_lookup("typeid", Λ, *raw_int*); *id_lookup*("typename", Λ, *struct_like*);
  id_lookup("undef", Λ, *if_like*); *id_lookup*("union", Λ, *struct_like*);
  id_lookup("unsigned", Λ, *raw_int*); *id_lookup*("using", Λ, *int_like*);
  id_lookup("va_dcl", Λ, *decl*);   ▷ Berkeley's variable-arg-list convention ◁
  id_lookup("va_list", Λ, *raw_int*);   ▷ ditto ◁
  id_lookup("virtual", Λ, *int_like*); *id_lookup*("void", Λ, *raw_int*);
  id_lookup("volatile", Λ, *const_like*); *id_lookup*("wchar_t", Λ, *raw_int*);
  id_lookup("while", Λ, *for_like*); *id_lookup*("xor", Λ, *alfop*);
  id_lookup("xor_eq", Λ, *alfop*); *res_wd_end* ← *name_ptr*; *id_lookup*("TeX", Λ, *custom*);
  id_lookup("make_pair", Λ, *func_template*);

46. ⟨Private variables 26⟩ +≡
  static sixteen_bits *int_loc*, *ext_loc*;   ▷ locations of special reserved words ◁

47. Lexical scanning.   Let us now consider the subroutines that read the CWEB source file and break it into meaningful units. There are four such procedures: One simply skips to the next '@␣' or '@*' that begins a section; another passes over the TEX text at the beginning of a section; the third passes over the TEX text in a C comment; and the last, which is the most interesting, gets the next token of a C text. They all use the pointers *limit* and *loc* into the line of input currently being studied.

48. Control codes in `CWEB`, which begin with '`@`', are converted into a numeric code designed to simplify `CWEAVE`'s logic; for example, larger numbers are given to the control codes that denote more significant milestones, and the code of *new_section* should be the largest of all. Some of these numeric control codes take the place of **char** control codes that will not otherwise appear in the output of the scanning routines.

#**define** *ignore* °*0* ▷ control code of no interest to `CWEAVE` ◁
#**define** *verbatim* °*2* ▷ takes the place of extended ASCII α ◁
#**define** *begin_short_comment* °*3* ▷ C++ short comment ◁
#**define** *begin_comment* '`\t`' ▷ tab marks will not appear ◁
#**define** *underline* '`\n`' ▷ this code will be intercepted without confusion ◁
#**define** *noop* °*177* ▷ takes the place of ASCII delete ◁
#**define** *xref_roman* °*203* ▷ control code for '`@^`' ◁
#**define** *xref_wildcard* °*204* ▷ control code for '`@:`' ◁
#**define** *xref_typewriter* °*205* ▷ control code for '`@.`' ◁
#**define** *TEX_string* °*206* ▷ control code for '`@t`' ◁
  format *TeX_string* *TeX*
#**define** *meaning* °*207* ▷ control code for '`@$`' ◁
#**define** *suppress* °*210* ▷ control code for '`@-`' ◁
#**define** *temp_meaning* °*211* ▷ control code for '`@%`' ◁
#**define** *right_start* °*212* ▷ control code for '`@r`' ◁
#**define** *ord* °*213* ▷ control code for '`@'`' ◁
#**define** *join* °*214* ▷ control code for '`@&`' ◁
#**define** *thin_space* °*215* ▷ control code for '`@,`' ◁
#**define** *math_break* °*216* ▷ control code for '`@|`' ◁
#**define** *line_break* °*217* ▷ control code for '`@/`' ◁
#**define** *big_line_break* °*220* ▷ control code for '`@#`' ◁
#**define** *no_line_break* °*221* ▷ control code for '`@+`' ◁
#**define** *pseudo_semi* °*222* ▷ control code for '`@;`' ◁
#**define** *macro_arg_open* °*224* ▷ control code for '`@[`' ◁
#**define** *macro_arg_close* °*225* ▷ control code for '`@]`' ◁
#**define** *trace* °*226* ▷ control code for '`@0`', '`@1`' and '`@2`' ◁
#**define** *translit_code* °*227* ▷ control code for '`@l`' ◁
#**define** *output_defs_code* °*230* ▷ control code for '`@h`' ◁
#**define** *format_code* °*231* ▷ control code for '`@f`' and '`@s`' ◁
#**define** *definition* °*232* ▷ control code for '`@d`' ◁
#**define** *begin_C* °*233* ▷ control code for '`@c`' ◁
#**define** *section_name* °*234* ▷ control code for '`@<`' ◁
#**define** *new_section* °*235* ▷ control code for '`@␣`' and '`@*`' ◁

49.   Control codes are converted to `CWEAVE`'s internal representation by means of
the table *ccode*.

⟨ Private variables 26 ⟩ +≡
  static eight_bits *ccode*[256];   ▷ meaning of a char following @ ◁

50.   ⟨ Set initial values 29 ⟩ +≡
  {
    int *c*;
    for (*c* ← 0; *c* < 256; *c*++) *ccode*[*c*] ← 0;
  }
  ccode[' '] ← *ccode*['\t'] ← *ccode*['\n'] ← *ccode*['\v'] ← *ccode*['\r'] ← *ccode*['\f'] ←
    ccode['*'] ← *new_section*; *ccode*['@'] ← '@';   ▷ 'quoted' at sign ◁
  ccode['='] ← *verbatim*; *ccode*['d'] ← *ccode*['D'] ← *definition*;
  ccode['f'] ← *ccode*['F'] ← *ccode*['s'] ← *ccode*['S'] ← *format_code*;
  ccode['c'] ← *ccode*['C'] ← *ccode*['p'] ← *ccode*['P'] ← *begin_C*;
  ccode['t'] ← *ccode*['T'] ← *TEX_string*; *ccode*['l'] ← *ccode*['L'] ← *translit_code*;
  ccode['q'] ← *ccode*['Q'] ← *noop*; *ccode*['h'] ← *ccode*['H'] ← *output_defs_code*;
  ccode['&'] ← *join*; *ccode*['<'] ← *ccode*['('] ← *section_name*; *ccode*['!'] ← *underline*;
  ccode['^'] ← *xref_roman*; *ccode*[':'] ← *xref_wildcard*; *ccode*['.'] ← *xref_typewriter*;
  ccode[','] ← *thin_space*; *ccode*['|'] ← *math_break*; *ccode*['/'] ← *line_break*;
  ccode['#'] ← *big_line_break*; *ccode*['+'] ← *no_line_break*; *ccode*[';'] ← *pseudo_semi*;
  ccode['['] ← *macro_arg_open*; *ccode*[']'] ← *macro_arg_close*; *ccode*['\''] ← *ord*;
  ccode['$'] ← *meaning*; *ccode*['%'] ← *temp_meaning*; *ccode*['-'] ← *suppress*;
  ccode['r'] ← *ccode*['R'] ← *right_start*; ⟨ Special control codes for debugging 51 ⟩

51.   Users can write `@2`, `@1`, and `@0` to turn tracing fully on, partly on, and off,
respectively.

⟨ Special control codes for debugging 51 ⟩ ≡
  ccode['0'] ← *ccode*['1'] ← *ccode*['2'] ← *trace*;

This code is used in section 50.

52.   The *skip_limbo* routine is used on the first pass to skip through portions of
the input that are not in any sections, i.e., that precede the first section. After this
procedure has been called, the value of *input_has_ended* will tell whether or not a
section has actually been found.

    There's a complication that we will postpone until later: If the `@s` operation appears
in limbo, we want to use it to adjust the default interpretation of identifiers.

⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *skip_limbo*(**void**);
  static eight_bits *skip_TEX*(**void**);

---

eight_bits = **uint8_t**, §6.   *input_has_ended*: **boolean**,   COMMON.W §25.

53. We look for a clue about the program's title, because this will become part of all meanings.

```
static void skip_limbo(void)
{
  while (true) {
    if (loc > limit ∧ get_line( ) ≡ false) return;
    if (loc ≡ buffer ∧ strncmp(buffer, "\\def\\title{", 11) ≡ 0) {
      loc ← buffer + 10;  title_lookup( );      ▷ this program's title will be code zero ◁
    }
    *(limit + 1) ← '@';
    while (*loc ≠ '@') loc ++;        ▷ look for '@', then skip two chars ◁
    if (loc ++ ≤ limit) {
      int c ← ccode[(eight_bits) *loc ++];

      if (c ≡ new_section) return;
      if (c ≡ noop) skip_restricted( );
      else if (c ≡ format_code) ⟨Process simple format in limbo 90⟩
    }
  }
}
```

54. The *skip_TEX* routine is used on the first pass to skip through the TEX code at the beginning of a section. It returns the next control code or '|' found in the input. A *new_section* is assumed to exist at the very end of the file.

```
format  skip_TeX   TeX
static eight_bits skip_TEX(void)
{
  while (true) {
    if (loc > limit ∧ get_line( ) ≡ false) return new_section;
    *(limit + 1) ← '@';
    while (*loc ≠ '@' ∧ *loc ≠ '|') loc ++;
    if (*loc ++ ≡ '|') return '|';
    if (loc ≤ limit) return ccode[(eight_bits) *(loc ++)];
  }
}
```

55. Inputting the next token. As stated above, CWEAVE's most interesting
lexical scanning routine is the *get_next* function that inputs the next token of C
input. However, *get_next* is not especially complicated.

The result of *get_next* is either a **char** code for some special character, or it is a
special code representing a pair of characters (e.g., '`!=`'), or it is the numeric value
computed by the *ccode* table, or it is one of the following special codes:

identifier: In this case the global variables *id_first* and *id_loc* will have been set to
the beginning and ending-plus-one locations in the buffer, as required by the
id_lookup routine.

string: The string will have been copied into the array *section_text*; *id_first* and
id_loc are set as above (now they are pointers into *section_text*).

constant: The constant is copied into *section_text*, with slight modifications; *id_first*
and *id_loc* are set.

Furthermore, some of the control codes cause *get_next* to take additional actions:

xref_roman, *xref_wildcard*, *xref_typewriter*, TEX_string, *meaning*, *suppress*, and
verbatim: The values of *id_first* and *id_loc* will have been set to the begin-
ning and ending-plus-one locations in the buffer.

section_name: In this case the global variable *cur_section* will point to the
byte_start entry for the section name that has just been scanned. The value
of *cur_section_char* will be '`(`' if the section name was preceded by `@(` in-
stead of `@<`.

If *get_next* sees '`@!`' it sets *xref_switch* to *def_flag* and goes on to the next token.

#define *constant* °*200*   ▷ C constant ◁
#define *string* °*201*   ▷ C string ◁
#define *identifier* °*202*   ▷ C identifier or reserved word ◁

⟨Private variables 26⟩ +≡
  static name_pointer *cur_section*;   ▷ name of section just scanned ◁
  static char *cur_section_char*;   ▷ the character just before that name ◁

56. As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise. C allows underscores to appear in identifiers, and some C compilers even allow the dollar sign.

#define *isxalpha*(*c*) $((c) \equiv \text{'\_'} \vee (c) \equiv \text{'\$'})$
▷ non-alpha characters allowed in identifier ◁
#define *ishigh*(*c*) $((\textbf{eight\_bits})(c) > °177)$

   static eight_bits *get_next*(**void**)   ▷ produces the next input token ◁
   {
      eight_bits *c*;   ▷ the current character ◁

      while (*true*) {
        ⟨ Check if we're at the end of a preprocessor command 62 ⟩
        if (*loc* > *limit* ∧ *get_line*( ) ≡ *false*) **return** *new_section*;
        c ← *(*loc*++);
        if (*xisdigit*(*c*) ∨ *c* ≡ '.') ⟨ Get a constant 65 ⟩
        else if (*c* ≡ '\'' ∨ *c* ≡ '"'
              ∨ ((*c* ≡ 'L' ∨ *c* ≡ 'u' ∨ *c* ≡ 'U') ∧ (**loc* ≡ '\'' ∨ **loc* ≡ '"'))
              ∨ ((*c* ≡ 'u' ∧ **loc* ≡ '8') ∧ (*(*loc* + 1) ≡ '\'' ∨ *(*loc* + 1) ≡ '"'))
              ∨ (*c* ≡ '<' ∧ *sharp_include_line* ≡ *true*)) ⟨ Get a string 66 ⟩
        else if (*xisalpha*(*c*) ∨ *isxalpha*(*c*) ∨ *ishigh*(*c*)) ⟨ Get an identifier 64 ⟩
        else if (*c* ≡ '@') ⟨ Get control code and possible section name 67 ⟩
        else if (*xisspace*(*c*)) **continue**;   ▷ ignore spaces and tabs ◁
        if (*c* ≡ '#' ∧ *loc* ≡ *buffer* + 1) ⟨ Raise preprocessor flag 59 ⟩
      mistake: ⟨ Compress two-symbol operator 63 ⟩
        return *c*;
      }
   }

57. ⟨ Predeclaration of procedures 11 ⟩ +≡ **static eight_bits** *get_next*(**void**);

58. Because preprocessor commands do not fit in with the rest of the syntax of C, we have to deal with them separately. One solution is to enclose such commands between special markers. Thus, when a # is seen as the first character of a line, *get_next* returns a special code *left_preproc* and raises a flag *preprocessing*.

   We can use the same internal code number for *left_preproc* as we do for *ord*, since *get_next* changes *ord* into a string.

#define *left_preproc* *ord*   ▷ begins a preprocessor command ◁
#define *right_preproc* °223   ▷ ends a preprocessor command ◁

⟨ Private variables 26 ⟩ +≡
   static boolean *preprocessing* ← *false*;   ▷ are we scanning a preprocessor command? ◁

59. ⟨ Raise preprocessor flag 59 ⟩ ≡
   {
      preprocessing ← *true*; ⟨ Check if next token is **include** 61 ⟩
      return *left_preproc*;
   }
This code is used in section 56.

60.   An additional complication is the freakish use of < and > to delimit a file name in lines that start with #include. We must treat this file name as a string.

⟨Private variables 26⟩ +≡
    static boolean *sharp_include_line* ← *false*;   ▷ are we scanning a #**include** line? ◁

61.   ⟨Check if next token is **include** 61⟩ ≡
    while (*loc* ≤ *buffer_end* − 7 ∧ *xisspace*(∗*loc*)) *loc* ++;
    if (*loc* ≤ *buffer_end* − 6 ∧ *strncmp*(*loc*, "include", 7) ≡ 0) *sharp_include_line* ← *true*;
This code is used in section 59.

62.   When we get to the end of a preprocessor line, we lower the flag and send a code *right_preproc*, unless the last character was a \.

⟨Check if we're at the end of a preprocessor command 62⟩ ≡
    while (*loc* ≡ *limit* − 1 ∧ *preprocessing* ∧ ∗*loc* ≡ '\\')
        if (*get_line*( ) ≡ *false*) **return** *new_section*;   ▷ still in preprocessor mode ◁
    if (*loc* ≥ *limit* ∧ *preprocessing*) {
        preprocessing ← *sharp_include_line* ← *false*; **return** *right_preproc*;
    }
This code is used in section 56.

---

boolean = **bool**, §6.
buffer: **char** [ ],
    COMMON.W §22.
buffer_end: **char** ∗,
    COMMON.W §22.
eight_bits = **uint8_t**, §6.

false, <stdbool.h>.
get_line: **boolean** ( ),
    COMMON.W §38.
limit: **char** ∗, COMMON.W §22.
loc: **char** ∗, COMMON.W §22.
new_section = °235, §48.

ord = °213, §48.
strncmp, <string.h>.
true, <stdbool.h>.
xisalpha = macro ( ), §9.
xisdigit = macro ( ), §9.
xisspace = macro ( ), §9.

63. The following code assigns values to the combinations `++`, `--`, `->`, `>=`, `<=`, `==`, `<<`, `>>`, `!=`, `||`, and `&&`, and to the C++ combinations `...`, `::`, `.*` and `->*`. The compound assignment operators (e.g., `+=`) are treated as separate tokens.

#define *compress*(*c*) **if** (*loc* ++ ≤ *limit*) **return** *c*

⟨Compress two-symbol operator 63⟩ ≡

```
switch (c) {
case '/':
  if (∗loc ≡ '∗') {
    compress(begin_comment);
  }
  else if (∗loc ≡ '/') compress(begin_short_comment);
  break;
case '+':
  if (∗loc ≡ '+') compress(plus_plus);
  break;
case '−':
  if (∗loc ≡ '−') {
    compress(minus_minus);
  }
  else {
    if (∗loc ≡ '>') {
      if (∗(loc + 1) ≡ '∗') {
        loc ++;  compress(minus_gt_ast);
      }
      else compress(minus_gt);
    }
  }
  break;
case '.':
  if (∗loc ≡ '∗') {
    compress(period_ast);
  }
  else if (∗loc ≡ '.' ∧ ∗(loc + 1) ≡ '.') {
    loc ++;  compress(dot_dot_dot);
  }
  break;
case ':':
  if (∗loc ≡ ':') compress(colon_colon);
  break;
case '=':
  if (∗loc ≡ '=') compress(eq_eq);
  break;
case '>':
  if (∗loc ≡ '=') {
    compress(gt_eq);
  }
  else if (∗loc ≡ '>') compress(gt_gt);
  break;
```

```
    case '<':
      if (*loc ≡ '=') {
        compress(lt_eq);
      }
      else if (*loc ≡ '<') compress(lt_lt);
      break;
    case '&':
      if (*loc ≡ '&') compress(and_and);
      break;
    case '|':
      if (*loc ≡ '|') compress(or_or);
      break;
    case '!':
      if (*loc ≡ '=') compress(non_eq);
      break;
    }
```

This code is used in section 56.

64. ⟨Get an identifier 64⟩ ≡
```
  {
    id_first ← −−loc;
    while (isalpha((eight_bits) *++loc) ∨ isdigit((eight_bits) *loc)
          ∨ isxalpha((eight_bits) *loc) ∨ ishigh((eight_bits) *loc)) ;
    id_loc ← loc;  return identifier;
  }
```

This code is used in section 56.

---

and_and = °4 , §8.
begin_comment = macro, §48.
begin_short_comment = °3 , §48.
colon_colon = °6 , §8.
dot_dot_dot = °16 , §8.
eight_bits = **uint8_t**, §6.
eq_eq = °36 , §8.
gt_eq = °35 , §8.
gt_gt = °21 , §8.
id_first: **char** ∗,

COMMON.W §21.
id_loc: **char** ∗, COMMON.W §21.
identifier = °202 , §55.
isalpha, <ctype.h>.
isdigit, <ctype.h>.
ishigh = macro ( ), §56.
isxalpha = macro ( ), §56.
limit: **char** ∗, COMMON.W §22.
loc: **char** ∗, COMMON.W §22.

lt_eq = °34 , §8.
lt_lt = °20 , §8.
minus_gt = °31 , §8.
minus_gt_ast = °27 , §8.
minus_minus = °1 , §8.
non_eq = °32 , §8.
or_or = °37 , §8.
period_ast = °26 , §8.
plus_plus = °13 , §8.

65. Different conventions are followed by TEX and C to express octal and hexadecimal numbers; it is reasonable to stick to each convention within its realm. Thus the C part of a CWEB file has octals introduced by `0` and hexadecimals by `0x`, but CWEAVE will print with TEX macros that the user can redefine to fit the context. In order to simplify such macros, we replace some of the characters.

Notice that in this section and the next, *id_first* and *id_loc* are pointers into the array *section_text*, not into *buffer*.

⟨ Get a constant 65 ⟩ ≡
  {
    id_first ← *id_loc* ← *section_text* + 1;
    if (∗(*loc* − 1) ≡ '0') {
      if (∗*loc* ≡ 'x' ∨ ∗*loc* ≡ 'X') {
        ∗*id_loc*++ ← '^'; *loc*++;
        while (*xisxdigit*(∗*loc*)) ∗*id_loc*++ ← ∗*loc*++;
      }   ▷ hex constant ◁
      else if (*xisdigit*(∗*loc*)) {
        ∗*id_loc*++ ← '~';
        while (*xisdigit*(∗*loc*)) ∗*id_loc*++ ← ∗*loc*++;
      }   ▷ octal constant ◁
      else goto *dec*;   ▷ decimal constant ◁
    }
    else {   ▷ decimal constant ◁
      if (∗(*loc* − 1) ≡ '.' ∧ ¬*xisdigit*(∗*loc*)) **goto** *mistake*;   ▷ not a constant ◁
    dec: ∗*id_loc*++ ← ∗(*loc* − 1);
      while (*xisdigit*(∗*loc*) ∨ ∗*loc* ≡ '.') ∗*id_loc*++ ← ∗*loc*++;
      if (∗*loc* ≡ 'e' ∨ ∗*loc* ≡ 'E') {   ▷ float constant ◁
        ∗*id_loc*++ ← '_'; *loc*++;
        if (∗*loc* ≡ '+' ∨ ∗*loc* ≡ '-') ∗*id_loc*++ ← ∗*loc*++;
        while (*xisdigit*(∗*loc*)) ∗*id_loc*++ ← ∗*loc*++;
      }
    }
    while (∗*loc* ≡ 'u' ∨ ∗*loc* ≡ 'U' ∨ ∗*loc* ≡ 'l' ∨ ∗*loc* ≡ 'L' ∨ ∗*loc* ≡ 'f' ∨ ∗*loc* ≡ 'F') {
      ∗*id_loc*++ ← '$'; ∗*id_loc*++ ← *toupper*((**eight_bits**) ∗*loc*); *loc*++;
    }
    return *constant*;
  }

This code is used in section 56.

66. C strings and character constants, delimited by double and single quotes, respectively, can contain newlines or instances of their own delimiters if they are protected by a backslash. We follow this convention, but do not allow the string to be longer than *longest_name*.

⟨ Get a string 66 ⟩ ≡
  { **char** *delim* ← *c*;   ▷ what started the string ◁
    id_first ← *section_text* + 1; *id_loc* ← *section_text*;
    if (*delim* ≡ '\'' ∧ ∗(*loc* − 2) ≡ '@') {
      ∗++*id_loc* ← '@'; ∗++*id_loc* ← '@';
    }

```
  *++id_loc ← delim;
  if (delim ≡ 'L' ∨ delim ≡ 'u' ∨ delim ≡ 'U') {          ▷ wide character constant ◁
    if (delim ≡ 'u' ∧ *loc ≡ '8') {
      *++id_loc ← *loc ++;
    }
    delim ← *loc ++;  *++id_loc ← delim;
  }
  if (delim ≡ '<') delim ← '>';          ▷ for file names in #include lines ◁
  while (true) {
    if (loc ≥ limit) {
      if (*(limit − 1) ≠ '\\') {
        err_print(_("!␣String␣didn't␣end"));  loc ← limit;  break;
      }
      if (get_line() ≡ false) {
        err_print(_("!␣Input␣ended␣in␣middle␣of␣string"));  loc ← buffer;  break;
      }
    }
    if ((c ← *loc ++) ≡ delim) {
      if (++id_loc ≤ section_text_end) *id_loc ← c;
      break;
    }
    if (c ≡ '\\') {
      if (loc ≥ limit) continue;
      else {
        if (++id_loc ≤ section_text_end) {
          *id_loc ← '\\';  c ← *loc ++;
        }
      }
    }
    if (++id_loc ≤ section_text_end) *id_loc ← c;
  }
  if (id_loc ≥ section_text_end) {
    fputs(_("\n!␣String␣too␣long:␣"), stdout);  term_write(section_text + 1, 25);
    printf("...");  mark_error;
  }
  id_loc ++;  return string;
}
```

This code is used in sections 56 and 67.

---

67. After an @ sign has been scanned, the next character tells us whether there is more work to do.

⟨ Get control code and possible section name 67 ⟩ ≡
```
  {
    c ← ∗loc ++;
    switch (ccode[(eight_bits) c]) {
    case translit_code: err_print(_("! Use @l in limbo only")); continue;
    case underline: xref_switch ← def_flag; continue;
    case temp_meaning: temp_switch ← true − temp_switch; continue;
    case right_start: right_start_switch ← true; continue;
    case trace: tracing ← c − '0'; continue;
    case xref_roman: case xref_wildcard: case xref_typewriter: case noop:
      case meaning: case suppress: case TₑX_string: c ← ccode[(eight_bits) c];
      skip_restricted( ); return c;
    case section_name: ⟨ Scan the section name and make cur_section point to it 68 ⟩
    case verbatim: ⟨ Scan a verbatim string 77 ⟩
    case ord: ⟨ Get a string 66 ⟩
    default: return ccode[(eight_bits) c];
    }
  }
```
This code is used in section 56.

68. The occurrence of a section name sets *xref_switch* to zero, because the section name might (for example) follow **int**.

⟨ Scan the section name and make *cur_section* point to it 68 ⟩ ≡
```
  {
    char ∗k;      ▷ pointer into section_text ◁
    cur_section_char ← ∗(loc − 1); ⟨ Put section name into section_text 70 ⟩
    if (k − section_text > 3 ∧ strncmp(k − 2, "...", 3) ≡ 0)
      cur_section ← section_lookup(section_text + 1, k − 3, 1);      ▷ 1 indicates a prefix ◁
    else cur_section ← section_lookup(section_text + 1, k, 0);
    xref_switch ← 0; return section_name;
  }
```
This code is used in section 67.

69. Section names are placed into the *section_text* array with consecutive spaces, tabs, and carriage-returns replaced by single spaces. There will be no spaces at the beginning or the end. (We set *section_text*[0] ← '␣' to facilitate this, since the *section_lookup* routine uses *section_text*[1] as the first character of the name.)

⟨ Set initial values 29 ⟩ +≡
  section_text[0] ← '␣';

70. ⟨Put section name into *section_text* 70⟩ ≡
  $k \leftarrow section\_text$;
  while (*true*) {
    if (*loc* > *limit* ∧ *get_line*( ) ≡ *false*) {
      err_print(("!␣Input␣ended␣in␣section␣name")); *loc* ← *buffer* + 1; **break**;
    }
    $c \leftarrow *loc$; ⟨If end of name or erroneous control code, **break** 71⟩
    loc++;
    if ($k < section\_text\_end$) k++;
    if (*xisspace*(*c*)) {
      $c \leftarrow$ '␣';
      if (*(k − 1) ≡ '␣') k−−;
    }
    $*k \leftarrow c$;
  }
  if ($k \geq section\_text\_end$) {
    fputs(("\n!␣Section␣name␣too␣long:␣"), *stdout*); *term_write*(*section_text* + 1, 25);
    printf("..."); *mark_harmless*;
  }
  if (*k ≡ '␣' ∧ $k > section\_text$) k−−;
This code is used in section 68.

71. ⟨If end of name or erroneous control code, **break** 71⟩ ≡
  if ($c ≡$ '@') {
    $c \leftarrow *(loc + 1)$;
    if ($c ≡$ '>') {
      loc += 2; **break**;
    }
    if (*ccode*[(**eight_bits**) *c*] ≡ *new_section*) {
      err_print(("!␣Section␣name␣didn't␣end")); **break**;
    }
    if ($c \neq$ '@') {
      err_print(("!␣Control␣codes␣are␣forbidden␣in␣section␣name")); **break**;
    }
    *(++k) ← '@'; *loc*++;   ▷ now $c ≡ *loc$ again ◁
  }
This code is used in section 70.

_ = macro ( ), §7.
buffer: **char** [], COMMON.W §22.
c: **int**, §53.
ccode: **static eight_bits** [], §49.
cur_section: **static name_pointer**, §55.
cur_section_char: **static char**, §55.
def_flag = macro, §34.
eight_bits = uint8_t, §6.
err_print: **void** ( ), COMMON.W §66.
false, <stdbool.h>.
fputs, <stdio.h>.
get_line: **boolean** ( ), COMMON.W §38.
limit: **char** *, COMMON.W §22.
loc: **char** *, COMMON.W §22.

mark_harmless = macro, §15.
meaning = °207, §48.
new_section = °235, §48.
noop = °177, §48.
ord = °213, §48.
printf, <stdio.h>.
right_start = °212, §48.
right_start_switch: **static boolean**, §256.
section_lookup: **name_pointer** ( ), COMMON.W §59.
section_name = °234, §48.
section_text: **char** [][], COMMON.W §21.
section_text_end: **char** *, COMMON.W §21.
skip_restricted: **static void** ( ), §76.
stdout, <stdio.h>.

strncmp, <string.h>.
suppress = °210, §48.
temp_meaning = °211, §48.
temp_switch: **static boolean**, §256.
term_write = macro ( ), §18.
TEX_string = °206, §48.
trace = °226, §48.
tracing: **static int**, §211.
translit_code = °227, §48.
true, <stdbool.h>.
underline = macro, §48.
verbatim = °2, §48.
xisspace = macro ( ), §9.
xref_roman = °203, §48.
xref_switch: **static sixteen_bits**, §26.
xref_typewriter = °205, §48.
xref_wildcard = °204, §48.

72.   This function skips over a restricted context at relatively high speed.

```
static void skip_restricted(void)
{
    int c ← ccode[(eight_bits) *(loc − 1)];
    id_first ← loc;  *(limit + 1) ← '@';
false_alarm:
    while (*loc ≠ '@')  loc++;
    id_loc ← loc;
    if (loc++ > limit) {
        err_print(_("! Control text didn't end"));  loc ← limit;
    }
    else {
        if (*loc ≡ '@' ∧ loc ≤ limit) {
            loc++;  goto false_alarm;
        }
        if (*loc++ ≠ '>')
            err_print(_("! Control codes are forbidden in control text"));
        if (c ≡ meaning ∧ phase ≡ 2)  ⟨Process a user-generated meaning 74⟩
        else if (c ≡ suppress ∧ phase ≡ 2)  ⟨Suppress mini-index entry 73⟩;
    }
}
```

73.   ⟨Suppress mini-index entry 73⟩ ≡
```
{
    char *first ← id_first, *last ← id_loc;
    while (xisspace(*first))  first++;
    while (xisspace(*(last − 1)))  last−−;
    if (first < last) {
        struct perm_meaning *q ← id_lookup(first, last, normal) − name_dir + cur_meaning;
        q→stamp ← section_count;        ▷ this is what actually suppresses output ◁
    }
}
```
This code is used in section 72.

74.   ⟨Process a user-generated meaning 74⟩ ≡
```
{
    char *first ← id_first;
    while (xisspace(*first))  first++;
    loc ← first;
    while (xisalpha(*loc) ∨ xisdigit(*loc) ∨ *loc ≡ '_')  loc++;
    if (*loc++ ≠ '␣')
        err_print(_("! Identifier in meaning should be followed by space"));
    else {
        name_pointer p ← id_lookup(first, loc − 1, normal);
        sixteen_bits t;
        int n ← 0;
        t ← title_lookup();
        if (*(loc − 1) ≡ '}')
```

        while $(xisdigit(*loc))$ $n \leftarrow 10 * n + (*loc +\!+) - \texttt{'0'}$;
     if $(*loc +\!+ \neq \texttt{'}_{\sqcup}\texttt{'})$
        $err\_print(\_(\texttt{"!}_\sqcup\texttt{Location}_\sqcup\texttt{in}_\sqcup\texttt{meaning}_\sqcup\texttt{should}_\sqcup\texttt{be}_\sqcup\texttt{followed}_\sqcup\texttt{by}_\sqcup\texttt{space"}))$;
     else   ⟨ Digest the meaning of p, t, n 75 ⟩;
    }
   $loc \leftarrow id\_loc + 2$;
  }

This code is used in section 72.

75.   ⟨ Digest the meaning of p, t, n 75 ⟩ ≡
  {
    meaning_struct $*m$;
    struct perm_meaning $*q \leftarrow p - name\_dir + cur\_meaning$;

    if $(temp\_switch)$ {
      $m \leftarrow temp\_meaning\_ptr +\!+$;
      if $(temp\_meaning\_ptr > max\_temp\_meaning\_ptr)$ {
        if $(temp\_meaning\_ptr > \&temp\_meaning\_stack[max\_meanings])$
          $overflow(\_(\texttt{"temp}_\sqcup\texttt{meanings"}))$;
        $max\_temp\_meaning\_ptr \leftarrow temp\_meaning\_ptr$;
      }
    }
    else $m \leftarrow \&(q\text{-}\!\!>\!perm)$;
    $m\text{-}\!\!>\!id \leftarrow p$; $m\text{-}\!\!>\!prog\_no \leftarrow t$; $m\text{-}\!\!>\!sec\_no \leftarrow n$;
    if $(id\_loc - loc \geq max\_tex\_chars)$ $strcpy(m\text{-}\!\!>\!tex\_part, \texttt{"}\backslash\backslash\texttt{zip"})$;
    else {
      char $*q \leftarrow m\text{-}\!\!>\!tex\_part$;

      while $(loc < id\_loc)$ $*q +\!+ \leftarrow *loc +\!+$;
      $*q \leftarrow \texttt{'}\backslash\texttt{0'}$;
    }
  }

This code is used in section 74.

76.   ⟨ Predeclaration of procedures 11 ⟩ +≡   **static void** $skip\_restricted(\textbf{void})$;

$_ = $ macro ( ), §7.
ccode: **static eight_bits** [ ],
  §49.
cur_meaning: **struct**
  perm_meaning[ ], §28.
eight_bits = **uint8_t**, §6.
err_print: **void** ( ),
  COMMON.W §66.
id: **name_pointer**, §27.
id_first: **char** *,
  COMMON.W §21.
id_loc: **char** *, COMMON.W §21.
id_lookup: **name_pointer** ( ),
  COMMON.W §48.
limit: **char** *, COMMON.W §22.
loc: **char** *, COMMON.W §22.
max_meanings = 100, §27.
max_temp_meaning_ptr: **static**

meaning_struct *, §28.
max_tex_chars = 50, §3.
meaning = °207, §48.
meaning_struct = **struct**,
  §27.
name_dir: **name_info** [ ],
  COMMON.W §43.
name_pointer = **name_info**
  *, §13.
normal = 0, §23.
overflow: **void** ( ),
  COMMON.W §71.
perm: **meaning_struct**, §28.
perm_meaning: **struct**, §28.
phase: **int**, COMMON.W §19.
prog_no: **sixteen_bits**, §27.
sec_no: **sixteen_bits**, §27.
section_count: **sixteen_bits**,

COMMON.W §37.
sixteen_bits = **uint16_t**, §6.
stamp: **int**, §28.
strcpy, <**string.h**>.
suppress = °210, §48.
temp_meaning_ptr: **static**
  meaning_struct *, §28.
temp_meaning_stack: **static**
  meaning_struct [ ], §28.
temp_switch: **static boolean**,
  §256.
tex_part: **char** [ ], §27.
title_lookup: **static**
  sixteen_bits ( ), §30.
xisalpha = macro ( ), §9.
xisdigit = macro ( ), §9.
xisspace = macro ( ), §9.

77. At the present point in the program we have $*(loc - 1) \equiv verbatim$; we set *id_first* to the beginning of the string itself, and *id_loc* to its ending-plus-one location in the buffer. We also set *loc* to the position just after the ending delimiter.

⟨ Scan a verbatim string 77 ⟩ ≡
```
{
    id_first ← loc ++;  *(limit + 1) ← '@';  *(limit + 2) ← '>';
    while (*loc ≠ '@' ∨ *(loc + 1) ≠ '>')  loc ++;
    if (loc ≥ limit)  err_print(_("! Verbatim string didn't end"));
    id_loc ← loc;  loc += 2;  return verbatim;
}
```

This code is used in section 67.

78. Phase one processing. We now have accumulated enough subroutines to make it possible to carry out `CWEAVE`'s first pass over the source file. If everything works right, both phase one and phase two of `CWEAVE` will assign the same numbers to sections, and these numbers will agree with what `CTANGLE` does.

The global variable *next_control* often contains the most recent output of *get_next*; in interesting cases, this will be the control code that ended a section or part of a section.

⟨ Private variables 26 ⟩ +≡
    static eight_bits *next_control*;   ▷ control code waiting to be acting upon ◁

79. The overall processing strategy in phase one has the following straightforward outline.

    static void *phase_one*(**void**)
    {
      phase ← 1; *reset_input*( ); *section_count* ← 0; *skip_limbo*( );
      ⟨ Give a default title to the program, if necessary 31 ⟩;
      while (¬*input_has_ended*) ⟨ Store cross-reference data for the current section 81 ⟩
      phase ← 2;   ▷ prepare for second phase ◁
      ⟨ Print error messages about unused or undefined section names 95 ⟩
    }

80. ⟨ Predeclaration of procedures 11 ⟩ +≡ **static void** *phase_one*(**void**);

81. ⟨ Store cross-reference data for the current section 81 ⟩ ≡
    {
      if (++*section_count* ≡ *max_sections*) *overflow*(_("section␣number"));
      if (∗(*loc* − 1) ≡ '*' ∧ *show_progress*) {
        printf("*%d", *section_count*); *update_terminal*;   ▷ print a progress report ◁
      }
      ⟨ Store cross-references in the TEX part of a section 85 ⟩
      ⟨ Store cross-references in the definition part of a section 88 ⟩
      ⟨ Store cross-references in the C part of a section 91 ⟩
    }
This code is used in section 79.

_ = macro ( ), §7.
eight_bits = uint8_t, §6.
err_print: **void** ( ),
  COMMON.W §66.
get_next: **static eight_bits**
  ( ), §57.
id_first: **char** ∗,
  COMMON.W §21.
id_loc: **char** ∗, COMMON.W §21.

input_has_ended: **boolean**,
  COMMON.W §25.
limit: **char** ∗, COMMON.W §22.
loc: **char** ∗, COMMON.W §22.
max_sections = 4000, §20.
overflow: **void** ( ),
  COMMON.W §71.
phase: **int**, COMMON.W §19.
printf, <stdio.h>.

reset_input: **void** ( ),
  COMMON.W §35.
section_count: **sixteen_bits**,
  COMMON.W §37.
show_progress = macro, §17.
skip_limbo: **static void** ( ),
  §53.
update_terminal = macro, §18.
verbatim = °2 , §48.

82. The *C_xref* subroutine stores references to identifiers in C text material beginning with the current value of *next_control* and continuing until *next_control* is '{' or '|', or until the next "milestone" is passed (i.e., $next\_control \geq format\_code$). If $next\_control \geq format\_code$ when *C_xref* is called, nothing will happen; but if $next\_control \equiv \text{'|'}$ upon entry, the procedure assumes that this is the '|' preceding C text that is to be processed.

The parameter *spec_ctrl* is used to change this behavior. In most cases *C_xref* is called with $spec\_ctrl \equiv ignore$, which triggers the default processing described above. If $spec\_ctrl \equiv section\_name$, section names will be gobbled. This is used when C text in the TEX part or inside comments is parsed: It allows for section names to appear in | ... |, but these strings will not be entered into the cross reference lists since they are not definitions of section names.

The program uses the fact that our internal code numbers satisfy the relations $xref\_roman \equiv identifier + roman$ and $xref\_wildcard \equiv identifier + wildcard$ and $xref\_typewriter \equiv identifier + typewriter$, as well as $normal \equiv 0$.

⟨ Predeclaration of procedures 11 ⟩ +≡
```
static void C_xref (eight_bits);
static void outer_xref (void);
```

83. **static void** *C_xref* (▷ makes cross-references for C identifiers ◁
        eight_bits *spec_ctrl*)
```
{
    name_pointer p;      ▷ a referenced name ◁
    while (next_control < format_code ∨ next_control ≡ spec_ctrl) {
        if (next_control ≥ identifier ∧ next_control ≤ xref_typewriter) {
            if (next_control > identifier) ⟨Replace "@@" by "@" 86⟩
            p ← id_lookup (id_first, id_loc, next_control − identifier);  new_xref (p);
        }
        if (next_control ≡ section_name) {
            section_xref_switch ← cite_flag;  new_section_xref (cur_section);
        }
        next_control ← get_next ( );
        if (next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡
                begin_short_comment) return;
    }
}
```

84. The *outer_xref* subroutine is like *C_xref* except that it begins with *next_control* $\neq$ '|' and ends with $next\_control \geq format\_code$. Thus, it handles C text with embedded comments.

```
static void outer_xref (void)      ▷ extension of C_xref ◁
{
    int bal;      ▷ brace level in comment ◁
    while (next_control < format_code)
        if (next_control ≠ begin_comment ∧ next_control ≠ begin_short_comment)
            C_xref (ignore);
        else {
```

boolean *is_long_comment* ← (*next_control* ≡ *begin_comment*);
bal ← *copy_comment*(*is_long_comment*, 1); *next_control* ← '|';
while (*bal* > 0) {
   C_xref(*section_name*);   ▷ do not reference section names in comments ◁
   if (*next_control* ≡ '|') *bal* ← *copy_comment*(*is_long_comment*, *bal*);
   else *bal* ← 0;   ▷ an error message will occur in phase two ◁
  }
 }
}

85.   In the TₑX part of a section, cross-reference entries are made only for the identifiers in C texts enclosed in | ... |, or for control texts enclosed in @^ ... @> or @. ... @> or @: ... @>.

⟨ Store cross-references in the TₑX part of a section 85 ⟩ ≡
  while (*true*) {
    switch (*next_control* ← *skip_TₑX*( )) {
    case *translit_code*: *err_print*(_("!␣Use␣@l␣in␣limbo␣only")); **continue**;
    case *underline*: *xref_switch* ← *def_flag*; **continue**;
    case *trace*: *tracing* ← ∗(*loc* − 1) − '0'; **continue**;
    case '|': *C_xref*(*section_name*); **break**;
    case *xref_roman*: **case** *xref_wildcard*: **case** *xref_typewriter*: **case** *meaning*:
      case *suppress*: **case** *noop*: **case** *section_name*: *loc* −= 2;
      next_control ← *get_next*( );   ▷ scan to @> ◁
      if (*next_control* ≥ *xref_roman* ∧ *next_control* ≤ *xref_typewriter*) {
        ⟨ Replace "@@" by "@" 86 ⟩
        new_xref(*id_lookup*(*id_first*, *id_loc*, *next_control* − *identifier*));
      }
      break;
    }
    if (*next_control* ≥ *format_code*) **break**;
  }
This code is used in section 81.

_ = macro ( ), §7.
begin_comment = macro, §48.
begin_short_comment = °3 , §48.
boolean = **bool**, §6.
cite_flag = 10240, §34.
copy_comment, §112.
cur_section: **static**
  name_pointer, §55.
def_flag = macro, §34.
eight_bits = **uint8_t**, §6.
err_print: **void** ( ),
  COMMON.W §66.
format_code = °231 , §48.
get_next: **static eight_bits**
  ( ), §57.
id_first: **char** ∗,
  COMMON.W §21.
id_loc: **char** ∗, COMMON.W §21.

id_lookup: **name_pointer** ( ),
  COMMON.W §48.
identifier = °202 , §55.
ignore = °0 , §48.
loc: **char** ∗, COMMON.W §22.
meaning = °207 , §48.
name_pointer = **name_info**
  ∗, §13.
new_section_xref: **static void**
  ( ), §37.
new_xref: **static void** ( ), §36.
next_control: **static**
  eight_bits, §78.
noop = °177 , §48.
normal = 0, §23.
roman = 1, §23.
section_name = °234 , §48.
section_xref_switch: **static**

sixteen_bits, §26.
skip_TₑX: **static eight_bits**
  ( ), §54.
spec_ctrl: **eight_bits**, §218.
suppress = °210 , §48.
trace = °226 , §48.
tracing: **static int**, §211.
translit_code = °227 , §48.
true, <stdbool.h>.
typewriter = 3, §23.
underline = macro, §48.
wildcard = 2, §23.
xref_roman = °203 , §48.
xref_switch: **static**
  sixteen_bits, §26.
xref_typewriter = °205 , §48.
xref_wildcard = °204 , §48.

86.   ⟨ Replace `"@@"` by `"@"` 86 ⟩ ≡
  {
     char *src* ← *id_first*, *dst* ← *id_first*;
     while (*src* < *id_loc*) {
        if (*src* ≡ '@') *src* ++;
        dst ++ ← *src* ++;
     }
     id_loc ← *dst*;
     while (*dst* < *src*) *dst* ++ ← '␣';   ▷ clean up in case of error message display ◁
  }

This code is used in sections 83 and 85.

87.   During the definition and C parts of a section, cross-references are made for all identifiers except reserved words. However, the right identifier in a format definition is not referenced, and the left identifier is referenced only if it has been explicitly underlined (preceded by @!). The TEX code in comments is, of course, ignored, except for C portions enclosed in | . . . |; the text of a section name is skipped entirely, even if it contains | . . . | constructions.

The variables *lhs* and *rhs* point to the respective identifiers involved in a format definition.

⟨ Private variables 26 ⟩ +≡
   static name_pointer *lhs*, *rhs*;   ▷ pointers to *byte_start* for format identifiers ◁
   static name_pointer *res_wd_end*;   ▷ pointer to the first nonreserved identifier ◁

88.   When we get to the following code we have *next_control* ≥ *format_code*.

⟨ Store cross-references in the definition part of a section 88 ⟩ ≡
   while (*next_control* ≤ *definition*) {   ▷ *format_code* or *definition* ◁
     if (*next_control* ≡ *definition*) {
        xref_switch ← *def_flag*;   ▷ implied @! ◁
        next_control ← *get_next*( );
     }
     else ⟨ Process a format definition 89 ⟩
     outer_xref( );
   }

This code is used in section 81.

89.   Error messages for improper format definitions will be issued in phase two. Our job in phase one is to define the *ilk* of a properly formatted identifier, and to remove cross-references to identifiers that we now discover should be unindexed.

⟨ Process a format definition 89 ⟩ ≡
  {
     next_control ← *get_next*( );
     if (*next_control* ≡ *identifier*) {
        lhs ← *id_lookup*(*id_first*, *id_loc*, *normal*); *lhs*→*ilk* ← *normal*;
        if (*xref_switch*) *new_xref*(*lhs*);
        next_control ← *get_next*( );
        if (*next_control* ≡ *identifier*) {
           rhs ← *id_lookup*(*id_first*, *id_loc*, *normal*); *lhs*→*ilk* ← *rhs*→*ilk*;

```
        if (unindexed(lhs)) {          ▷ retain only underlined entries ◁
          xref_pointer q, r ← Λ;
          for (q ← (xref_pointer) lhs→xref; q > xmem; q ← q→xlink)
            if (q→num < def_flag)
              if (r) r→xlink ← q→xlink;
              else lhs→xref ← (void *) q→xlink;
            else r ← q;
        }
        next_control ← get_next();
      }
    }
  }
```
This code is used in section 88.

90.   A much simpler processing of format definitions occurs when the definition is found in limbo.

⟨ Process simple format in limbo 90 ⟩ ≡
```
  {
    if (get_next() ≠ identifier) err_print(_("!␣Missing␣left␣identifier␣of␣@s"));
    else {
      lhs ← id_lookup(id_first, id_loc, normal);
      if (get_next() ≠ identifier) err_print(_("!␣Missing␣right␣identifier␣of␣@s"));
      else {
        rhs ← id_lookup(id_first, id_loc, normal);  lhs→ilk ← rhs→ilk;
      }
    }
  }
```
This code is used in section 53.

_ = macro ( ), §7.
byte_start: **char** *, §13.
def_flag = macro, §34.
definition = °232, §48.
err_print: **void** ( ),
  COMMON.W §66.
format_code = °231, §48.
get_next: **static eight_bits**
  ( ), §57.
id_first: **char** *,
  COMMON.W §21.
id_loc: **char** *, COMMON.W §21.

id_lookup: **name_pointer** ( ),
  COMMON.W §48.
identifier = °202, §55.
ilk = macro, §23.
name_pointer = **name_info**
  *, §13.
new_xref: **static void** ( ), §36.
next_control: **static**
  eight_bits, §78.
normal = 0, §23.
num: **sixteen_bits**, §25.

outer_xref: **static void** ( ),
  §84.
unindexed = macro ( ), §35.
xlink: **struct xref_info** *, §25.
xmem: **static xref_info** [ ],
  §26.
xref = macro, §34.
xref_pointer = **xref_info** *,
  §25.
xref_switch: **static**
  sixteen_bits, §26.

91. Finally, when the TEX and definition parts have been treated, we have $next\_control \geq begin\_C$.

⟨Store cross-references in the C part of a section 91⟩ ≡
  if (*next_control* ≤ *section_name*) {   ▷ *begin_C* or *section_name* ◁
    if (*next_control* ≡ *begin_C*) *section_xref_switch* ← 0;
    else {
      section_xref_switch ← *def_flag*;
      if (*cur_section_char* ≡ '(' ∧ *cur_section* ≠ *name_dir*) *set_file_flag*(*cur_section*);
    }
    do {
      if (*next_control* ≡ *section_name* ∧ *cur_section* ≠ *name_dir*)
        new_section_xref(*cur_section*);
      next_control ← *get_next*( ); *outer_xref*( );
    } **while** (*next_control* ≤ *section_name*);
  }

This code is used in section 81.

92. After phase one has looked at everything, we want to check that each section name was both defined and used. The variable *cur_xref* will point to cross-references for the current section name of interest.

⟨Private variables 26⟩ +≡
  static xref_pointer *cur_xref*;   ▷ temporary cross-reference pointer ◁
  static boolean *an_output*;   ▷ did *file_flag* precede *cur_xref*? ◁

93. The following recursive procedure walks through the tree of section names and prints out anomalies.

  static void *section_check*(**name_pointer** *p*)   ▷ print anomalies in subtree *p* ◁
  {
    if (*p*) {
      section_check(*p*→*llink*); *cur_xref* ← (**xref_pointer**) *p*→*xref*;
      if (*cur_xref*→*num* ≡ *file_flag*) {
        an_output ← *true*; *cur_xref* ← *cur_xref*→*xlink*;
      }
      else *an_output* ← *false*;
      if (*cur_xref*→*num* < *def_flag*) {
        fputs(("\n!␣Never␣defined:␣<"), *stdout*); *print_section_name*(*p*);
        putchar('>'); *mark_harmless*;
      }
      while (*cur_xref*→*num* ≥ *cite_flag*) *cur_xref* ← *cur_xref*→*xlink*;
      if (*cur_xref* ≡ *xmem* ∧ ¬*an_output*) {
        fputs(("\n!␣Never␣used:␣<"), *stdout*); *print_section_name*(*p*); *putchar*('>');
        mark_harmless;
      }
      section_check(*p*→*rlink*);
    }
  }

94. ⟨Predeclaration of procedures 11⟩ +≡ **static void** *section_check*(**name_pointer**);

95. ⟨Print error messages about unused or undefined section names 95⟩ ≡
  section_check(*root*);

This code is used in section 79.

96. Low-level output routines.   The TEX output is supposed to appear in lines at most *line_length* characters long, so we place it into an output buffer. During the output process, *out_line* will hold the current line number of the line about to be output.

⟨ Private variables 26 ⟩ +≡
  static char *out_buf* [*line_length* + 1];   ▷ assembled characters ◁
  static char *∗out_buf_end* ← *out_buf* + *line_length*;   ▷ end of *out_buf* ◁
  static char *∗out_ptr*;   ▷ last character in *out_buf* ◁
  static int *out_line*;   ▷ number of next line to be output ◁

97.   The *flush_buffer* routine empties the buffer up to a given breakpoint, and moves any remaining characters to the beginning of the next line. If the *per_cent* parameter is *true*, a '%' is appended to the line that is being output; in this case the breakpoint *b* should be strictly less than *out_buf_end*. If the *per_cent* parameter is *false*, trailing blanks are suppressed. The characters emptied from the buffer form a new line of output; if the *carryover* parameter is true, a "%" in that line will be carried over to the next line (so that TEX will ignore the completion of commented-out text).

#define *c_line_write*(*c*)   *fflush*(*active_file*), *fwrite*(*out_buf* + 1, **sizeof**(**char**), *c*, *active_file*)
#define *tex_putc*(*c*)   *putc*(*c*, *active_file*)
#define *tex_new_line*   *putc*('\n', *active_file*)
#define *tex_printf*(*c*)   *fprintf*(*active_file*, "%s", *c*)
#define *tex_puts*(*c*)   *fputs*(*c*, *active_file*)

⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *flush_buffer*(**char** ∗, **boolean**, **boolean**);
  static void *finish_line*(**void**);

98.   **static void** *flush_buffer*(**char** *∗b*,
  ▷ outputs from *out_buf* + 1 to *b*, where *b* ≤ *out_ptr* ◁
    boolean *per_cent*, **boolean** *carryover*)
  {
    char *∗j*;
    j ← *b*;   ▷ pointer into *out_buf* ◁
    if (¬*per_cent*)   ▷ remove trailing blanks ◁
      while (*j* > *out_buf* ∧ *∗j* ≡ '␣') *j*−−;
    c_line_write(*j* − *out_buf*);
    if (*per_cent*) *tex_putc*('%');
    tex_new_line; *out_line* ++;
    if (*carryover*)
      while (*j* > *out_buf*)
        if (*∗j*−− ≡ '%' ∧ (*j* ≡ *out_buf* ∨ *∗j* ≠ '\\')) {
          ∗b−− ← '%'; **break**;
        }
    if (*b* < *out_ptr*) *memcpy*(*out_buf* + 1, *b* + 1, (**size_t**)(*out_ptr* − *b*));
    out_ptr −= *b* − *out_buf*;
  }

99.   When we are copying TeX source material, we retain line breaks that occur in the input, except that an empty line is not output when the TeX source line was nonempty. For example, a line of the TeX file that contains only an index cross-reference entry will not be copied. The *finish_line* routine is called just before *get_line* inputs a new line, and just after a line break token has been emitted during the output of translated C text.

```
static void finish_line(void)       ▷ do this at the end of a line ◁
{
   char *k;      ▷ pointer into buffer ◁
   if (out_ptr > out_buf) flush_buffer(out_ptr, false, false);
   else {
      for (k ← buffer; k ≤ limit; k++)
         if (¬(xisspace(*k))) return;
      flush_buffer(out_buf, false, false);
   }
}
```

100.   In particular, the *finish_line* procedure is called near the very beginning of phase two. We initialize the output variables in a slightly tricky way so that the first line of the output file will be dependent of the user language set by the '+l' option and its argument. If you call CTWILL with '+lX' (or '-lX' as well), where 'X' is the (possibly empty) string of characters to the right of 'l', 'X' will be prepended to 'ctwimac.tex', e.g., if you call CTWILL with '+ldeutsch', you will receive the line '\input deutschctwimac'. Without this option the first line of the output file will be '\input ctwimac'. Or, if the user has specified proofing by saying +P on the command line, it's '\input ctproofmac' (resp. \input Xctproofmac with option +lX), a set of macros used when debugging mini-index entries.

#define *proofing*   *flags*['P']

⟨Start TeX output 100⟩ ≡
   out_ptr ← *out_buf* + 1; *out_line* ← 1; *active_file* ← *tex_file*; *∗out_ptr* ← 'c';
   tex_puts("\\input␣"); *tex_printf*(*use_language*);
   tex_puts(*proofing* ? "ctproofma" : "ctwima");

This code is used in section 5.

---

active_file: **FILE** *,
   COMMON.W §83.
b: **eight_bits**, §242.
boolean = **bool**, §6.
buffer: **char** [],
   COMMON.W §22.
false, <stdbool.h>.
fflush, <stdio.h>.
flags: **boolean** [],

COMMON.W §73.
fprintf, <stdio.h>.
fputs, <stdio.h>.
fwrite, <stdio.h>.
get_line: **boolean** ( ),
   COMMON.W §38.
limit: **char** *, COMMON.W §22.
line_length = 80, §22.
memcpy, <string.h>.

putc, <stdio.h>.
size_t, <stddef.h>.
tex_file: **FILE** *,
   COMMON.W §83.
true, <stdbool.h>.
use_language: **const char** *,
   COMMON.W §86.
xisspace = macro ( ), §9.

101. When we wish to append one character c to the output buffer, we write '*out*(c)'; this will cause the buffer to be emptied if it was already full. If we want to append more than one character at once, we say *out_str*(s), where s is a string containing the characters.

A line break will occur at a space or after a single-nonletter TEX control sequence.

#define *out*(c)
```
              {
                  if (ms_mode) {        ▷ outputting to ministring_buf ◁
                    if (ministring_ptr < &ministring_buf[max_tex_chars])
                       *ministring_ptr++ ← c;
                  }
                  else {
                    if (out_ptr ≥ out_buf_end) break_out( );
                    *(++out_ptr) ← c;
                  }
              }
```
⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *out_str*(**const char** ∗);
  static void *break_out*(**void**);

102.   **static void** *out_str*(▷ output characters from s to end of string ◁
      const char ∗s)
  {
    while (∗s) *out*(∗s++);
  }

103. The *break_out* routine is called just before the output buffer is about to overflow. To make this routine a little faster, we initialize position 0 of the output buffer to '\'; this character isn't really output.

⟨ Set initial values 29 ⟩ +≡
  out_buf[0] ← '\\';

104. A long line is broken at a blank space or just before a backslash that isn't preceded by another backslash. In the latter case, a '%' is output at the break.

  static void *break_out*(**void**)   ▷ finds a way to break the output line ◁
  {
    char ∗k ← *out_ptr*;   ▷ pointer into *out_buf* ◁
    while (*true*) {
      if (k ≡ *out_buf*) ⟨ Print warning message, break the line, **return** 105 ⟩
      if (∗k ≡ '␣') {
        flush_buffer(k, *false*, *true*); **return**;
      }
      if (∗(k−−) ≡ '\\' ∧ ∗k ≠ '\\') {   ▷ we've decreased k ◁
        flush_buffer(k, *true*, *true*); **return**;
      }
    }
  }

105.   We get to this section only in the unusual case that the entire output line
consists of a string of backslashes followed by a string of nonblank non-backslashes.
In such cases it is almost always safe to break the line by putting a '%' just before
the last character.

⟨ Print warning message, break the line, **return** 105 ⟩ ≡
  {
      printf (("\n!␣Line␣had␣to␣be␣broken␣(output␣l.␣%d):\n"), *out_line*);
      term_write (*out_buf* + 1, *out_ptr* − *out_buf* − 1); *new_line*; *mark_harmless*;
      flush_buffer (*out_ptr* − 1, *true*, *true*); **return**;
  }

This code is used in section 104.

106.   Here is a macro that outputs a section number in decimal notation. The
number to be converted by *out_section* is known to be less than *def_flag*, so it cannot
have more than five decimal digits.

⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *out_section* (**sixteen_bits**);
  static void *out_name* (**name_pointer**, **boolean**);

107.   **static void** *out_section* (**sixteen_bits** *n*)
  {
      char *s*[6];
      sprintf (*s*, "%d", *n*); *out_str* (*s*);
  }

108.   The *out_name* procedure is used to output an identifier or index entry, enclos-
ing it in braces.

  static void *out_name* (**name_pointer** *p*, **boolean** *quote_xalpha*)
  {
      char ∗*k*, ∗*k_end* ← (*p* + 1)→*byte_start*;   ▷ pointers into *byte_mem* ◁
      out ('{');
      for (*k* ← *p*→*byte_start*; *k* < *k_end*; *k*++) {
        if (*isxalpha* (∗*k*) ∧ *quote_xalpha*) *out* ('\\');
        out (∗*k*);
      }
      out ('}');
  }

_ = macro ( ), §7.
boolean = **bool**, §6.
byte_mem: **char** [],
  COMMON.W §43.
byte_start: **char** ∗, §13.
def_flag = macro, §34.
false, <stdbool.h>.
flush_buffer: **static void** ( ),
  §98.
isxalpha = macro ( ), §56.
mark_harmless = macro, §15.

max_tex_chars = 50, §3.
ministring_buf : **static char**
  [], §28.
ministring_ptr : **static char** ∗,
  §28.
ms_mode: **static boolean**,
  §28.
name_pointer = **name_info**
  ∗, §13.
new_line = macro, §18.
out_buf : **static char** [], §96.

out_buf_end: **static char** ∗,
  §96.
out_line: **static int**, §96.
out_ptr: **static char** ∗, §96.
printf , <stdio.h>.
s: **char** ∗, §2.
sixteen_bits = **uint16_t**, §6.
sprintf , <stdio.h>.
term_write = macro ( ), §18.
true, <stdbool.h>.

109. Routines that copy TEX material. During phase two, we use the sub-routines *copy_limbo* and *copy_TEX* (and *copy_comment*) in place of the analogous *skip_limbo* and *skip_TEX* that were used in phase one.

The *copy_limbo* routine, for example, takes TEX material that is not part of any section and transcribes it almost verbatim to the output file. The use of '@' signs is severely restricted in such material: '@@' pairs are replaced by singletons; '@l' and '@q' and '@s' are interpreted.

⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *copy_limbo*(**void**);
  static eight_bits *copy_TEX*(**void**);
  static int *copy_comment*(**boolean**, **int**);

110.   static void *copy_limbo*(**void**)
  {
    char *c*;
    while (*true*) {
      if (*loc* > *limit* ∧ (*finish_line*( ), *get_line*( ) ≡ *false*)) **return**;
      ∗(*limit* + 1) ← '@';
      while (∗*loc* ≠ '@') *out*(∗(*loc*++));
      if (*loc*++ ≤ *limit*) {
        c ← ∗*loc*++;
        if (*ccode*[(**eight_bits**) *c*] ≡ *new_section*) **break**;
        switch (*ccode*[(**eight_bits**) *c*]) {
        case *translit_code*: *out_str*("\\ATL"); **break**;
        case '@': *out*('@'); **break**;
        case *noop*: *skip_restricted*( ); **break**;
        case *format_code*:
          if (*get_next*( ) ≡ *identifier*) *get_next*( );
          if (*loc* ≥ *limit*) *get_line*( );   ▷ avoid blank lines in output ◁
          break;   ▷ the operands of @s are ignored on this pass ◁
        case *right_start*: *right_start_switch* ← *true*; **break**;
        default: *err_print*(_("!␣Double␣@␣should␣be␣used␣in␣limbo")); *out*('@');
        }
      }
    }
  }

111.   The *copy_TEX* routine processes the TEX code at the beginning of a section; for example, the words you are now reading were copied in this way. It returns the next control code or '|' found in the input. We don't copy spaces or tab marks into the beginning of a line. This makes the test for empty lines in *finish_line* work.

format *copy_TeX*   *TeX*

static eight_bits *copy_TEX*(**void**)
{ **char** *c*;   ▷ current character being copied ◁

   while (*true*) {
      if (*loc* > *limit* ∧ (*finish_line*( ), *get_line*( ) ≡ *false*)) **return** *new_section*;
      ∗(limit + 1) ← '@';
      while ((*c* ← *∗(loc* ++)) ≠ '|' ∧ *c* ≠ '@') {
         out(*c*);
         if (*out_ptr* ≡ *out_buf* + 1 ∧ (*xisspace*(*c*))) *out_ptr* −−;
      }
      if (*c* ≡ '|') **return** '|';
      if (*loc* ≤ *limit*) **return** *ccode*[(**eight_bits**) *∗(loc* ++)];
   }
}

−= macro ( ), §7.
boolean = **bool**, §6.
ccode: **static eight_bits** [ ],
  §49.
eight_bits = **uint8_t**, §6.
err_print: **void** ( ),
  COMMON.W §66.
false, <stdbool.h>.
finish_line: **static void** ( ), §99.
format_code = °*231*, §48.
get_line: **boolean** ( ),
  COMMON.W §38.

get_next: **static eight_bits**
  ( ), §57.
identifier = °*202*, §55.
limit: **char** ∗, COMMON.W §22.
loc: **char** ∗, COMMON.W §22.
new_section = °*235*, §48.
noop = °*177*, §48.
out = macro ( ), §101.
out_buf: **static char** [ ], §96.
out_ptr: **static char** ∗, §96.
out_str, §102.
right_start = °*212*, §48.

right_start_switch: **static**
  boolean, §256.
skip_limbo: **static void** ( ),
  §53.
skip_restricted: **static void**
  ( ), §76.
skip_TEX: **static eight_bits**
  ( ), §54.
translit_code = °*227*, §48.
true, <stdbool.h>.
xisspace = macro ( ), §9.

112.   The *copy_comment* function issues a warning if more braces are opened than closed, and in the case of a more serious error it supplies enough braces to keep TEX from complaining about unbalanced braces. Instead of copying the TEX material into the output buffer, this function copies it into the token memory (in phase two only). The abbreviation *app_tok*(*t*) is used to append token *t* to the current token list, and it also makes sure that it is possible to append at least one further token without overflow.

#define *app_tok*(*c*)
            {
               if (*tok_ptr* + 2 > *tok_mem_end*) *overflow*(_("token"));
               *(*tok_ptr* ++) ← *c*;
            }
   static int *copy_comment*(▷ copies TEX code in comments ◁
         boolean *is_long_comment*,   ▷ is this a traditional C comment? ◁
         int *bal*)   ▷ brace balance ◁
   { **char** *c*;   ▷ current character being copied ◁
      while (*true*) {
         if (*loc* > *limit*) {
            if (*is_long_comment*) {
               if (*get_line*( ) ≡ *false*) {
                  err_print(_("!␣Input␣ended␣in␣mid-comment")); *loc* ← *buffer* + 1;
                  goto *done*;
               }
            }
            else {
               if (*bal* > 1) *err_print*(_("!␣Missing␣}␣in␣comment"));
               goto *done*;
            }
         }
         c ← *(*loc* ++);
         if (*c* ≡ '|') **return** *bal*;
         if (*is_long_comment*) ⟨Check for end of comment 113⟩
         if (*phase* ≡ 2) {
            if (*ishigh*(*c*)) *app_tok*(*quoted_char*);
            app_tok(*c*);
         }
         ⟨Copy special things when *c* ≡ '@', '\\' 114⟩
         if (*c* ≡ '{') *bal* ++;
         else if (*c* ≡ '}') {
            if (*bal* > 1) *bal* --;
            else {
               err_print(_("!␣Extra␣}␣in␣comment"));
               if (*phase* ≡ 2) *tok_ptr* --;
            }
         }
      }
   done: ⟨Clear *bal* and **return** 115⟩
   }

113.   ⟨Check for end of comment 113⟩ ≡
  if $(c \equiv \texttt{'*'} \wedge *loc \equiv \texttt{'/'})$ {
     $loc \mathrel{+}\mathrel{+}$;
     if $(bal > 1)$ $err\_print(\_(\texttt{"!}\_\texttt{Missing}\_\texttt{\}}\_\texttt{in}\_\texttt{comment"}))$;
     goto $done$;
  }

This code is used in section 112.

114.   ⟨Copy special things when $c \equiv \texttt{'@'}, \texttt{'\\'}$ 114⟩ ≡
  if $(c \equiv \texttt{'@'})$ {
     if $(*(loc \mathrel{+}\mathrel{+}) \neq \texttt{'@'})$ {
        $err\_print(\_(\texttt{"!}\_\texttt{Illegal}\_\texttt{use}\_\texttt{of}\_\texttt{@}\_\texttt{in}\_\texttt{comment"}))$; $loc \mathrel{-}= 2$;
        if $(phase \equiv 2)$ $*(tok\_ptr - 1) \leftarrow \texttt{'}\_\texttt{'}$;
        goto $done$;
     }
  }
  else {
     if $(c \equiv \texttt{'\\'} \wedge *loc \neq \texttt{'@'})$ {
        if $(phase \equiv 2)$ $app\_tok(*(loc \mathrel{+}\mathrel{+}))$
        else $loc \mathrel{+}\mathrel{+}$;
     }
  }

This code is used in section 112.

115.   We output enough right braces to keep TEX happy.

⟨Clear bal and **return** 115⟩ ≡
  if $(phase \equiv 2)$
     while $(bal \mathrel{-}\mathrel{-} > 0)$ $app\_tok(\texttt{'\}'})$;
  return 0;

This code is used in section 112.

116. Parsing. The most intricate part of `CWEAVE` is its mechanism for converting C-like code into TEX code, and we might as well plunge into this aspect of the program now. A "bottom up" approach is used to parse the C-like material, since `CWEAVE` must deal with fragmentary constructions whose overall "part of speech" is not known.

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

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

A production applies if the category codes have a given pattern. For example, one of the productions (see rule 3) is

$$ exp \ \left\{ \begin{array}{c} binop \\ ubinop \end{array} \right\} \ exp \ \rightarrow \ exp $$

and it means that three consecutive scraps whose respective categories are *exp*, *binop* (or *ubinop*), and *exp* are converted to one scrap whose category is *exp*. The translations of the original scraps are simply concatenated. The case of

$$ exp \ comma \ exp \ \rightarrow \ exp \qquad E_1 C \ opt 9 \ E_2 $$

(rule 4) is only slightly more complicated: Here the resulting *exp* translation consists not only of the three original translations, but also of the tokens *opt* and 9 between the translations of the *comma* and the following *exp*. In the TEX file, this will specify an optional line break after the comma, with penalty 90.

At each opportunity the longest possible production is applied. For example, if the current sequence of scraps is *if_clause stmt else_like if_like*, rule 63 is applied; but if the sequence is *if_clause stmt else_like* followed by anything other than *if_like*, rule 64 takes effect; and if the sequence is *if_clause stmt* followed by anything other than *else_like*, rule 65 takes effect.

Translation rules such as '$E_1 C \ opt 9 \ E_2$' above use subscripts to distinguish between translations of scraps whose categories have the same initial letter; these subscripts are assigned from left to right.

117.   Here is a list of the category codes that scraps can have. (A few others, like *int_like*, have already been defined; the *cat_name* array contains a complete list.)

#define *exp* 1   ▷ denotes an expression, including perhaps a single identifier ◁
#define *unop* 2   ▷ denotes a unary operator ◁
#define *binop* 3   ▷ denotes a binary operator ◁
#define *ubinop* 4
              ▷ denotes an operator that can be unary or binary, depending on context ◁
#define *cast* 5   ▷ denotes a cast ◁
#define *question* 6   ▷ denotes a question mark and possibly the expressions flanking it ◁
#define *lbrace* 7   ▷ denotes a left brace ◁
#define *rbrace* 8   ▷ denotes a right brace ◁
#define *decl_head* 9   ▷ denotes an incomplete declaration ◁
#define *comma* 10   ▷ denotes a comma ◁
#define *lpar* 11   ▷ denotes a left parenthesis or left bracket ◁
#define *rpar* 12   ▷ denotes a right parenthesis or right bracket ◁
#define *prelangle* 13   ▷ denotes '<' before we know what it is ◁
#define *prerangle* 14   ▷ denotes '>' before we know what it is ◁
#define *langle* 15   ▷ denotes '<' when it's used as angle bracket in a template ◁
#define *colcol* 18   ▷ denotes '::' ◁
#define *base* 19   ▷ denotes a colon that introduces a base specifier ◁
#define *decl* 20   ▷ denotes a complete declaration ◁
#define *struct_head* 21   ▷ denotes the beginning of a structure specifier ◁
#define *stmt* 23   ▷ denotes a complete statement ◁
#define *function* 24   ▷ denotes a complete function ◁
#define *fn_decl* 25   ▷ denotes a function declarator ◁
#define *semi* 27   ▷ denotes a semicolon ◁
#define *colon* 28   ▷ denotes a colon ◁
#define *tag* 29   ▷ denotes a statement label ◁
#define *if_head* 30   ▷ denotes the beginning of a compound conditional ◁
#define *else_head* 31   ▷ denotes a prefix for a compound statement ◁
#define *if_clause* 32   ▷ pending **if** together with a condition ◁
#define *lproc* 35   ▷ begins a preprocessor command ◁
#define *rproc* 36   ▷ ends a preprocessor command ◁
#define *insert* 37   ▷ a scrap that gets combined with its neighbor ◁
#define *section_scrap* 38   ▷ section name ◁
#define *dead* 39   ▷ scrap that won't combine ◁
#define *ftemplate* 59   ▷ *make_pair* ◁
#define *new_exp* 60   ▷ **new** and a following type identifier ◁
#define *begin_arg* 61   ▷ @[◁
#define *end_arg* 62   ▷ @] ◁
#define *title* 63   ▷ program name or header name in a "meaning" ◁

⟨ Private variables 26 ⟩ +≡
   static char *cat_name*[256][12];

118. ⟨Set initial values 29⟩ +≡

  {

    int c;

    for $(c \leftarrow 0; \ c < 256; \ c\mathbin{+}\mathbin{+})$ *strcpy*(*cat_name*[c], "UNKNOWN");

  }

  strcpy(*cat_name*[*exp*], "exp"); *strcpy*(*cat_name*[*unop*], "unop");

  strcpy(*cat_name*[*binop*], "binop"); *strcpy*(*cat_name*[*ubinop*], "ubinop");

  strcpy(*cat_name*[*cast*], "cast"); *strcpy*(*cat_name*[*question*], "?");

  strcpy(*cat_name*[*lbrace*], "{"); *strcpy*(*cat_name*[*rbrace*], "}");

  strcpy(*cat_name*[*decl_head*], "decl_head"); *strcpy*(*cat_name*[*comma*], ",");

  strcpy(*cat_name*[*lpar*], "("); *strcpy*(*cat_name*[*rpar*], ")");

  strcpy(*cat_name*[*prelangle*], "<"); *strcpy*(*cat_name*[*prerangle*], ">");

  strcpy(*cat_name*[*langle*], "\\<"); *strcpy*(*cat_name*[*colcol*], "::");

  strcpy(*cat_name*[*base*], "\\:"); *strcpy*(*cat_name*[*decl*], "decl");

  strcpy(*cat_name*[*struct_head*], "struct_head"); *strcpy*(*cat_name*[*alfop*], "alfop");

  strcpy(*cat_name*[*stmt*], "stmt"); *strcpy*(*cat_name*[*function*], "function");

  strcpy(*cat_name*[*fn_decl*], "fn_decl"); *strcpy*(*cat_name*[*else_like*], "else_like");

  strcpy(*cat_name*[*semi*], ";"); *strcpy*(*cat_name*[*colon*], ":");

  strcpy(*cat_name*[*tag*], "tag"); *strcpy*(*cat_name*[*if_head*], "if_head");

  strcpy(*cat_name*[*else_head*], "else_head"); *strcpy*(*cat_name*[*if_clause*], "if()");

  strcpy(*cat_name*[*lproc*], "#{"); *strcpy*(*cat_name*[*rproc*], "#}");

  strcpy(*cat_name*[*insert*], "insert"); *strcpy*(*cat_name*[*section_scrap*], "section");

  strcpy(*cat_name*[*dead*], "@d"); *strcpy*(*cat_name*[*public_like*], "public");

  strcpy(*cat_name*[*operator_like*], "operator"); *strcpy*(*cat_name*[*new_like*], "new");

  strcpy(*cat_name*[*catch_like*], "catch"); *strcpy*(*cat_name*[*for_like*], "for");

  strcpy(*cat_name*[*do_like*], "do"); *strcpy*(*cat_name*[*if_like*], "if");

  strcpy(*cat_name*[*delete_like*], "delete"); *strcpy*(*cat_name*[*raw_ubin*], "ubinop?");

  strcpy(*cat_name*[*const_like*], "const"); *strcpy*(*cat_name*[*raw_int*], "raw");

  strcpy(*cat_name*[*int_like*], "int"); *strcpy*(*cat_name*[*case_like*], "case");

  strcpy(*cat_name*[*sizeof_like*], "sizeof"); *strcpy*(*cat_name*[*struct_like*], "struct");

  strcpy(*cat_name*[*typedef_like*], "typedef"); *strcpy*(*cat_name*[*define_like*], "define");

  strcpy(*cat_name*[*template_like*], "template"); *strcpy*(*cat_name*[*ftemplate*], "ftemplate");

  strcpy(*cat_name*[*new_exp*], "new_exp"); *strcpy*(*cat_name*[*begin_arg*], "@[");

  strcpy(*cat_name*[*end_arg*], "@]"); *strcpy*(*cat_name*[0], "zero");

119. This code allows CWEAVE to display its parsing steps.

  static void *print_cat*(▷ symbolic printout of a category ◁

    eight_bits c)

  {

    fputs(*cat_name*[c], *stdout*);

  }

120. ⟨Predeclaration of procedures 11⟩ +≡ **static void** *print_cat*(**eight_bits**);

121.   The token lists for translated TeX output contain some special control symbols as well as ordinary characters. These control symbols are interpreted by `CWEAVE` before they are written to the output file.

> *break_space* denotes an optional line break or an en space;
>
> *force* denotes a line break;
>
> *big_force* denotes a line break with additional vertical space;
>
> *preproc_line* denotes that the line will be printed flush left;
>
> *opt* denotes an optional line break (with the continuation line indented two ems with respect to the normal starting position)—this code is followed by an integer n, and the break will occur with penalty $10n$;
>
> *backup* denotes a backspace of one em;
>
> *cancel* obliterates any *break_space*, *opt*, *force*, or *big_force* tokens that immediately precede or follow it and also cancels any *backup* tokens that follow it;
>
> *indent* causes future lines to be indented one more em;
>
> *outdent* causes future lines to be indented one less em.

122.   All of these tokens are removed from the TEX output that comes from C text
between |...| signs; *break_space* and *force* and *big_force* become single spaces in
this mode. The translation of other C texts results in TEX control sequences \1, \2,
\3, \4, \5, \6, \7, \8 corresponding respectively to *indent*, *outdent*, *opt*, *backup*,
break_space, *force*, *big_force* and *preproc_line*. However, a sequence of consecutive
'␣', *break_space*, *force*, and/or *big_force* tokens is first replaced by a single token (the
maximum of the given ones).

The token *math_rel* will be translated into \MRL{, and it will get a matching } later.
Other control sequences in the TEX output will be '\\{...}' surrounding identifiers,
'\&{...}' surrounding reserved words, '\.{...}' surrounding strings, '\C{...} *force*'
surrounding comments, and '\X$n$: ... \X' surrounding section names, where n is the
section number.

#define *math_rel*   °206
#define *big_cancel*   °210   ▷ like *cancel*, also overrides spaces ◁
#define *cancel*   °211   ▷ overrides *backup*, *break_space*, *force*, *big_force* ◁
#define *indent*   °212   ▷ one more tab (\1) ◁
#define *outdent*   °213   ▷ one less tab (\2) ◁
#define *opt*   °214   ▷ optional break in mid-statement (\3) ◁
#define *backup*   °215   ▷ stick out one unit to the left (\4) ◁
#define *break_space*   °216   ▷ optional break between statements (\5) ◁
#define *force*   °217   ▷ forced break between statements (\6) ◁
#define *big_force*   °220   ▷ forced break with additional space (\7) ◁
#define *preproc_line*   °221   ▷ begin line without indentation (\8) ◁
#define *quoted_char*   °222   ▷ introduces a character token in the range °200−°377 ◁
#define *end_translation*   °223   ▷ special sentinel token at end of list ◁
#define *inserted*   °224   ▷ sentinel to mark translations of inserts ◁
#define *qualifier*   °225   ▷ introduces an explicit namespace qualifier ◁

123. From raw input to scraps. The raw input is converted into scraps according to the following table, which gives category codes followed by the translations. The symbol '**' stands for '\&{identifier}', i.e., the identifier itself treated as a reserved word. The right-hand column is the so-called *mathness*, which is explained further below.

An identifier c of length 1 is translated as \|c instead of as \\{c}. An identifier CAPS in all caps is translated as \.{CAPS} instead of as \\{CAPS}. An identifier that has become a reserved word via **typedef** is translated with \& replacing \\ and *raw_int* replacing *exp*.

A string of length greater than 20 is broken into pieces of size at most 20 with discretionary breaks in between.

!=	*binop*: \I	yes
<=	*binop*: \Z	yes
>=	*binop*: \G	yes
==	*binop*: \E	yes
&&	*binop*: \W	yes
||	*binop*: \V	yes
++	*unop*: \PP	yes
--	*unop*: \MM	yes
->	*binop*: \MG	yes
>>	*binop*: \GG	yes
<<	*binop*: \LL	yes
::	*colcol*: \DC	maybe
.*	*binop*: \PA	yes
->*	*binop*: \MGA	yes
...	*raw_int*: \,\ldots\,	yes
"string"	*exp*: \.{string with special characters quoted}	maybe
@=string@>	*exp*: \vb{string with special characters quoted}	maybe
@'7'	*exp*: \.{@'7'}	maybe
077 or \77	*exp*: \T{\^77}	maybe
0x7f	*exp*: \T{\^7f}	maybe
77	*exp*: \T{77}	maybe
77L	*exp*: \T{77\$L}	maybe
0.1E5	*exp*: \T{0.1_5}	maybe
+	*ubinop*: +	yes
−	*ubinop*: −	yes
*	*raw_ubin*: *	yes
/	*binop*: /	yes
<	*prelangle*: \langle	yes
=	*binop*: \K	yes
>	*prerangle*: \rangle	yes

---

binop = 3, §117.   *prelangle* = 13, §117.   *raw_ubin* = 49, §23.
colcol = 18, §117.   *prerangle* = 14, §117.   *ubinop* = 4, §117.
exp = 1, §117.   *raw_int* = 51, §23.   *unop* = 2, §117.
mathness: **eight_bits**, §136.

124. Cont.

.	*binop*: .	yes
|	*binop*: \OR	yes
^	*binop*: \XOR	yes
%	*binop*: \MOD	yes
?	*question*: \?	yes
!	*unop*: \R	yes
~	*unop*: \CM	yes
&	*raw_ubin*: \AND	yes
(	*lpar*: (	maybe
[	*lpar*: [	maybe
)	*rpar*:)	maybe
]	*rpar*:]	maybe
{	*lbrace*: {	yes
}	*lbrace*: }	yes
,	*comma*: ,	yes
;	*semi*: ;	maybe
:	*colon*: :	no
# (within line)	*ubinop*: \#	yes
# (at beginning)	*lproc*: *force preproc_line* \#	no
end of # line	*rproc*: *force*	no
identifier	*exp*: \\{identifier with underlines and dollar signs quoted}	maybe
and	*alfop*: **	yes
and_eq	*alfop*: **	yes
asm	*sizeof_like*: **	maybe
auto	*int_like*: **	maybe
bitand	*alfop*: **	yes
bitor	*alfop*: **	yes
bool	*raw_int*: **	maybe
break	*case_like*: **	maybe
case	*case_like*: **	maybe
catch	*catch_like*: **	maybe
char	*raw_int*: **	maybe
class	*struct_like*: **	maybe
clock_t	*raw_int*: **	maybe
compl	*alfop*: **	yes
const	*const_like*: **	maybe
const_cast	*raw_int*: **	maybe
continue	*case_like*: **	maybe

125.   Cont.

default	*case_like*: **	maybe
define	*define_like*: **	maybe
defined	*sizeof_like*: **	maybe
delete	*delete_like*: **	maybe
div_t	*raw_int*: **	maybe
do	*do_like*: **	maybe
double	*raw_int*: **	maybe
dynamic_cast	*raw_int*: **	maybe
elif	*if_like*: **	maybe
else	*else_like*: **	maybe
endif	*if_like*: **	maybe
enum	*struct_like*: **	maybe
error	*if_like*: **	maybe
explicit	*int_like*: **	maybe
export	*int_like*: **	maybe
extern	*int_like*: **	maybe
FILE	*raw_int*: **	maybe
float	*raw_int*: **	maybe
for	*for_like*: **	maybe
fpos_t	*raw_int*: **	maybe
friend	*int_like*: **	maybe
goto	*case_like*: **	maybe
if	*if_like*: **	maybe
ifdef	*if_like*: **	maybe
ifndef	*if_like*: **	maybe
include	*if_like*: **	maybe
inline	*int_like*: **	maybe
int	*raw_int*: **	maybe
jmp_buf	*raw_int*: **	maybe
ldiv_t	*raw_int*: **	maybe
line	*if_like*: **	maybe
long	*raw_int*: **	maybe

126. Cont.

`make_pair`	*ftemplate*: \\{make_pair}	maybe
`mutable`	*int_like*: **	maybe
`namespace`	*struct_like*: **	maybe
`new`	*new_like*: **	maybe
`not`	*alfop*: **	yes
`not_eq`	*alfop*: **	yes
`NULL`	*exp*: \NULL	yes
`offsetof`	*raw_int*: **	maybe
`operator`	*operator_like*: **	maybe
`or`	*alfop*: **	yes
`or_eq`	*alfop*: **	yes
`pragma`	*if_like*: **	maybe
`private`	*public_like*: **	maybe
`protected`	*public_like*: **	maybe
`ptrdiff_t`	*raw_int*: **	maybe
`public`	*public_like*: **	maybe
`register`	*int_like*: **	maybe
`reinterpret_cast`	*raw_int*: **	maybe
`return`	*case_like*: **	maybe
`short`	*raw_int*: **	maybe
`sig_atomic_t`	*raw_int*: **	maybe
`signed`	*raw_int*: **	maybe
`size_t`	*raw_int*: **	maybe
`sizeof`	*sizeof_like*: **	maybe
`static`	*int_like*: **	maybe
`static_cast`	*raw_int*: **	maybe
`struct`	*struct_like*: **	maybe
`switch`	*for_like*: **	maybe
`template`	*template_like*: **	maybe
`TeX`	*exp*: \TeX	yes
`this`	*exp*: \this	yes
`throw`	*case_like*: **	maybe
`time_t`	*raw_int*: **	maybe
`try`	*else_like*: **	maybe

127.   Cont.

typedef	*typedef_like*: **	maybe
typeid	*raw_int*: **	maybe
typename	*struct_like*: **	maybe
undef	*if_like*: **	maybe
union	*struct_like*: **	maybe
unsigned	*raw_int*: **	maybe
using	*int_like*: **	maybe
va_dcl	*decl*: **	maybe
va_list	*raw_int*: **	maybe
virtual	*int_like*: **	maybe
void	*raw_int*: **	maybe
volatile	*const_like*: **	maybe
wchar_t	*raw_int*: **	maybe
while	*for_like*: **	maybe
xor	*alfop*: **	yes
xor_eq	*alfop*: **	yes
@,	*insert*: \\,	maybe
@|	*insert*: *opt* 0	maybe
@/	*insert*: *force*	no
@#	*insert*: *big_force*	no
@+	*insert*: *big_cancel* {} *break_space* {} *big_cancel*	no
@;	*semi*:	maybe
@[	*begin_arg*:	maybe
@]	*end_arg*:	maybe
@&	*insert*: \\J	maybe
@h	*insert*: *force* \\ATH *force*	no
@< section name @>	*section_scrap*: \\X*n*: translated section name\\X	maybe
@(section name @>	*section_scrap*: \\X*n*:\\.{section name with special characters quoted␣}\\X	maybe
/*comment*/	*insert*: *cancel* \\C{translated comment} *force*	no
//comment	*insert*: *cancel* \\SHC{translated comment} *force*	no

The construction @t stuff @> contributes \\hbox{ stuff } to the following scrap.

128. Table of all productions. Each production that combines two or more consecutive scraps implicitly inserts a $ where necessary, that is, between scraps whose abutting boundaries have different *mathness*. In this way we never get double $$.

A translation is provided when the resulting scrap is not merely a juxtaposition of the scraps it comes from. An asterisk* next to a scrap means that its first identifier gets an underlined entry in the index, via the function *make_underlined*. Two asterisks** means that both *make_underlined* and *make_reserved* are called; that is, the identifier's ilk becomes *raw_int*. A dagger † before the production number refers to the notes at the end of this section, which deal with various exceptional cases.

We use *in*, *out*, *back* and *bsp* as shorthands for *indent*, *outdent*, *backup* and *break_space*, respectively.

LHS	→ RHS	Translation	Example
0 $\left\{ \begin{array}{c} any \\ any\ any \\ any\ any\ any \end{array} \right\}$ *insert*	→ $\left\{ \begin{array}{c} any \\ any\ any \\ any\ any\ any \end{array} \right\}$		stmt; /∗ comment ∗/
1 *exp* $\left\{ \begin{array}{c} lbrace \\ int\_like \\ decl \end{array} \right\}$	→ *fn_decl* $\left\{ \begin{array}{c} lbrace \\ int\_like \\ decl \end{array} \right\}$	$F = E^*\ in\ in$	$main()\{$ $main(ac, av)$ **int** $ac;$
2 *exp unop*	→ *exp*		$x{+}{+}$
3 *exp* $\left\{ \begin{array}{c} binop \\ ubinop \end{array} \right\}$ *exp*	→ *exp*		x/y $x + y$
4 *exp comma exp*	→ *exp*	$EC\ opt9\ E$	$f(x, y)$
5 *exp* $\left\{ \begin{array}{c} lpar\ rpar \\ cast \end{array} \right\}$ *colon*	→ *exp* $\left\{ \begin{array}{c} lpar\ rpar \\ cast \end{array} \right\}$ *base*		**C**(): **C int** i):
6 *exp semi*	→ *stmt*		$x \leftarrow 0;$
7 *exp colon*	→ *tag*	E^*C	*found*:
8 *exp rbrace*	→ *stmt rbrace*		end of **enum** list
9 *exp* $\left\{ \begin{array}{c} lpar\ rpar \\ cast \end{array} \right\}$ $\left\{ \begin{array}{c} const\_like \\ case\_like \end{array} \right\}$	→ *exp* $\left\{ \begin{array}{c} lpar\ rpar \\ cast \end{array} \right\}$	$\left\{ \begin{array}{c} R = R_{\sqcup}C \\ C_1 = C_{1\sqcup}C_2 \end{array} \right\}$	$f()$ **const** $f(\textbf{int})$ **throw**
10 *exp* $\left\{ \begin{array}{c} exp \\ cast \end{array} \right\}$	→ *exp*		$time()$
11 *lpar* $\left\{ \begin{array}{c} exp \\ ubinop \end{array} \right\}$ *rpar*	→ *exp*		(x) $(*)$
12 *lpar rpar*	→ *exp*	$L\backslash, R$	functions, declarations

129. Cont.

13 $lpar \left\{ \begin{array}{c} decl\_head \\ int\_like \\ cast \end{array} \right\} rpar$ $\rightarrow cast$ (**char** $*$)

14 $lpar \left\{ \begin{array}{c} decl\_head \\ int\_like \\ exp \end{array} \right\} comma$ $\rightarrow lpar$ $L \left\{ \begin{array}{c} D \\ I \\ E \end{array} \right\} C\, opt9$ (**int**,

15 $lpar \left\{ \begin{array}{c} stmt \\ decl \end{array} \right\}$ $\rightarrow lpar$ $\left\{ \begin{array}{c} LS_\sqcup \\ LD_\sqcup \end{array} \right\}$ $(k \leftarrow 5;$ (**int** $k \leftarrow 5;$

16 $unop \left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$ $\rightarrow exp$ $\neg x$ $\sim$**C**

17 $ubinop\ cast\ rpar$ $\rightarrow cast\ rpar$ $C = \{U\}C$ $*$**CPtr**)

18 $ubinop \left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$ $\rightarrow \left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$ $\{U\}\left\{ \begin{array}{c} E \\ I \end{array} \right\}$ $*x$ $*$**CPtr**

19 $ubinop\ binop$ $\rightarrow binop$ $math\_rel\ U\{B\}\}$ $*=$
20 $binop\ binop$ $\rightarrow binop$ $math\_rel\ \{B_1\}\{B_2\}\}$ $\gg=$

21 $cast \left\{ \begin{array}{c} lpar \\ exp \end{array} \right\}$ $\rightarrow \left\{ \begin{array}{c} lpar \\ exp \end{array} \right\}$ $\left\{ \begin{array}{c} CL \\ C_\sqcup E \end{array} \right\}$ (**double**)$(x+2)$ (**double**) x

22 $cast\ semi$ $\rightarrow exp\ semi$ (**int**);
23 $sizeof\_like\ cast$ $\rightarrow exp$ **sizeof**(**double**)
24 $sizeof\_like\ exp$ $\rightarrow exp$ $S_\sqcup E$ **sizeof** x

25 $int\_like \left\{ \begin{array}{c} int\_like \\ struct\_like \end{array} \right\}$ $\rightarrow \left\{ \begin{array}{c} int\_like \\ struct\_like \end{array} \right\}$ $I_\sqcup \left\{ \begin{array}{c} I \\ S \end{array} \right\}$ **extern char**

26 $int\_like\ exp \left\{ \begin{array}{c} raw\_int \\ struct\_like \end{array} \right\}$ $\rightarrow int\_like \left\{ \begin{array}{c} raw\_int \\ struct\_like \end{array} \right\}$ **extern"Ada" int**

---

130. Cont.

27 *int_like* $\left\{\begin{matrix} exp \\ ubinop \\ colon \end{matrix}\right\}$ $\rightarrow$ *decl_head* $\left\{\begin{matrix} exp \\ ubinop \\ colon \end{matrix}\right\}$ $D = I_\sqcup$ **int** x
int $*x$
unsigned :

28 *int_like* $\left\{\begin{matrix} semi \\ binop \end{matrix}\right\}$ $\rightarrow$ *decl_head* $\left\{\begin{matrix} semi \\ binop \end{matrix}\right\}$ **int** $x;$
int $f(\textbf{int} = 4)$

29 *public_like colon* $\rightarrow$ *tag* **private**:
30 *public_like* $\rightarrow$ *int_like* **private**

31 *colcol* $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ $\rightarrow$ $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ *qualifier* C $\left\{\begin{matrix} E \\ I \end{matrix}\right\}$ $\mathbf{C}::x$

32 *colcol colcol* $\rightarrow$ *colcol* $\mathbf{C}::\mathbf{B}::$
33 *decl_head comma* $\rightarrow$ *decl_head* $DC_\sqcup$ **int** $x,$
34 *decl_head ubinop* $\rightarrow$ *decl_head* $D\{U\}$ **int** $*$
$\dagger 35$ *decl_head exp* $\rightarrow$ *decl_head* DE^* **int** x

36 *decl_head* $\left\{\begin{matrix} binop \\ colon \end{matrix}\right\}$ *exp* $\left\{\begin{matrix} comma \\ semi \\ rpar \end{matrix}\right\}$ $\rightarrow$ *decl_head* $\left\{\begin{matrix} comma \\ semi \\ rpar \end{matrix}\right\}$ $D = D\left\{\begin{matrix} B \\ C \end{matrix}\right\}E$ **int** $f(\textbf{int}\ x = 2)$
int $b : 1$

37 *decl_head cast* $\rightarrow$ *decl_head* **int** $f(\textbf{int})$

38 *decl_head* $\left\{\begin{matrix} int\_like \\ lbrace \\ decl \end{matrix}\right\}$ $\rightarrow$ *fn_decl* $\left\{\begin{matrix} int\_like \\ lbrace \\ decl \end{matrix}\right\}$ $F = D\ in\ in$ **long** *time*( ) {

39 *decl_head semi* $\rightarrow$ *decl* **int** $n;$
40 *decl decl* $\rightarrow$ *decl* $D_1\ force\ D_2$ **int** $n;$ **double** $x;$

41 *decl* $\left\{\begin{matrix} stmt \\ function \end{matrix}\right\}$ $\rightarrow$ $\left\{\begin{matrix} stmt \\ function \end{matrix}\right\}$ $D\ big\_force\ \left\{\begin{matrix} S \\ F \end{matrix}\right\}$ **extern** $n;$ *main* (){ }

42 *base* $\left\{\begin{matrix} int\_like \\ exp \end{matrix}\right\}$ *comma* $\rightarrow$ *base* $B_\sqcup\left\{\begin{matrix} I \\ E \end{matrix}\right\}C\ opt9$: **public A**,
: $i(5),$

43 *base* $\left\{\begin{matrix} int\_like \\ exp \end{matrix}\right\}$ *lbrace* $\rightarrow$ *lbrace* $B_\sqcup\left\{\begin{matrix} I \\ E \end{matrix}\right\}_\sqcup L$ **D** : **public A** {

44 *struct_like lbrace* $\rightarrow$ *struct_head* $S_\sqcup L$ **struct** {

45 *struct_like* $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ *semi* $\rightarrow$ *decl_head semi* $S_\sqcup\left\{\begin{matrix} E^{**} \\ I^{**} \end{matrix}\right\}$ **struct forward**;

46 *struct_like* $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ *lbrace* $\rightarrow$ *struct_head* $S_\sqcup\left\{\begin{matrix} E^{**} \\ I^{**} \end{matrix}\right\}_\sqcup L$ **struct name_info** {

131. Cont.

47 $struct\_like \begin{Bmatrix} exp \\ int\_like \end{Bmatrix} colon$ $\rightarrow struct\_like \begin{Bmatrix} exp \\ int\_like \end{Bmatrix} base$ **class C** :

†48 $struct\_like \begin{Bmatrix} exp \\ int\_like \end{Bmatrix}$ $\rightarrow int\_like$ $S_{\sqcup}\begin{Bmatrix} E \\ I \end{Bmatrix}$ **struct name_info** z;

49 $struct\_head \begin{Bmatrix} decl \\ stmt \\ function \end{Bmatrix} rbrace$ $\rightarrow int\_like$ $S \ in \ force \begin{Bmatrix} D \\ S \\ F \end{Bmatrix} out \ force \ R$ **struct** { declaration }

50 $struct\_head \ rbrace$ $\rightarrow int\_like$ $S\backslash, R$ **class C** { }

51 $fn\_decl \ decl$ $\rightarrow fn\_decl$ $F \ force \ D$ $f(z)$ **double** z;

52 $fn\_decl \ stmt$ $\rightarrow function$ $F \ out \ out \ force \ S$ $main() \dots$

53 $function \begin{Bmatrix} stmt \\ decl \\ function \end{Bmatrix}$ $\rightarrow \begin{Bmatrix} stmt \\ decl \\ function \end{Bmatrix}$ $F \ big\_force \begin{Bmatrix} S \\ D \\ F \end{Bmatrix}$ outer block

54 $lbrace \ rbrace$ $\rightarrow stmt$ $L\backslash, R$ empty statement

55 $lbrace \begin{Bmatrix} stmt \\ decl \\ function \end{Bmatrix} rbrace$ $\rightarrow stmt$ $force \ L \ in \ force \ S \ force \ back \ R \ out \ force$ compound statement

56 $lbrace \ exp \ [comma] \ rbrace$ $\rightarrow exp$ initializer

57 $if\_like \ exp$ $\rightarrow if\_clause$ $I_{\sqcup}E$ **if** (z)

58 $else\_like \ colon$ $\rightarrow else\_like \ base$ **try** :

59 $else\_like \ lbrace$ $\rightarrow else\_head \ lbrace$ **else** {

60 $else\_like \ stmt$ $\rightarrow stmt$ $force \ E \ in \ bsp \ S \ out \ force$ **else** $x \leftarrow 0$;

$base = 19$, §117.
$big\_force = °220$, §122.
$binop = 3$, §117.
$cast = 5$, §117.
$colcol = 18$, §117.
$colon = 28$, §117.
$comma = 10$, §117.
$decl = 20$, §117.
$decl\_head = 9$, §117.
$else\_head = 31$, §117.
$else\_like = 26$, §23.

$exp = 1$, §117.
$fn\_decl = 25$, §117.
$force = °217$, §122.
$function = 24$, §117.
$if\_clause = 32$, §117.
$if\_like = 47$, §23.
$int\_like = 52$, §23.
$lbrace = 7$, §117.
$opt = °214$, §122.
$out = \text{macro}$ ( ), §101.

$public\_like = 40$, §23.
$qualifier = °225$, §122.
$rbrace = 8$, §117.
$rpar = 12$, §117.
$semi = 27$, §117.
$stmt = 23$, §117.
$struct\_head = 21$, §117.
$struct\_like = 55$, §23.
$tag = 29$, §117.
$ubinop = 4$, §117.

132. Cont.

61 *else_head* $\left\{ \begin{array}{c} stmt \\ exp \end{array} \right\}$ $\rightarrow$ *stmt*   *force E bsp noop cancel S bsp*   **else** { $x \leftarrow 0;$ }

62 *if_clause lbrace* $\rightarrow$ *if_head lbrace*   **if** (x) {

63 *if_clause stmt else_like if_like* $\rightarrow$ *if_like*   *force I in bsp S out force E* ␣*I*   **if** (x) *y;* **else if**

64 *if_clause stmt else_like* $\rightarrow$ *else_like*   *force I in bsp S out force E*   **if** (x) *y;* **else**

65 *if_clause stmt* $\rightarrow$ *else_like stmt*   **if** (x)

66 *if_head* $\left\{ \begin{array}{c} stmt \\ exp \end{array} \right\}$ *else_like if_like* $\rightarrow$ *if_like*   *force I bsp noop cancel S force E* ␣*I*   **if** (x) { *y;* } **else if**

67 *if_head* $\left\{ \begin{array}{c} stmt \\ exp \end{array} \right\}$ *else_like* $\rightarrow$ *else_like*   *force I bsp noop cancel S force E*   **if** (x) { *y;* } **else**

68 *if_head* $\left\{ \begin{array}{c} stmt \\ exp \end{array} \right\}$ $\rightarrow$ *else_head* $\left\{ \begin{array}{c} stmt \\ exp \end{array} \right\}$ **if** (x) { *y;* }

69 *do_like stmt else_like semi* $\rightarrow$ *stmt*   *D bsp noop cancel S cancel noop bsp ES*   **do** $f(x);$ **while** $(g(x));$

70 *case_like semi* $\rightarrow$ *stmt*   **return**;

71 *case_like colon* $\rightarrow$ *tag*   **default**:

72 *case_like exp* $\rightarrow$ *exp* C␣E **return** 0

73 *catch_like* $\left\{ \begin{array}{c} cast \\ exp \end{array} \right\}$ $\rightarrow$ *fn_decl* $C\left\{ \begin{array}{c} C \\ E \end{array} \right\}$ *in in*   **catch**(. . .)

74 *tag tag* $\rightarrow$ *tag* T_1 *bsp* T_2 **case** 0: **case** 1:

75 *tag* $\left\{ \begin{array}{c} stmt \\ decl \\ function \end{array} \right\}$ $\rightarrow$ $\left\{ \begin{array}{c} stmt \\ decl \\ function \end{array} \right\}$ *force back T bsp S*   **case** 0: $z \leftarrow 0;$

†76 *stmt* $\left\{ \begin{array}{c} stmt \\ decl \\ function \end{array} \right\}$ $\rightarrow$ $\left\{ \begin{array}{c} stmt \\ decl \\ function \end{array} \right\}$ $S\left\{ \begin{array}{c} force\ S \\ big\_force\ D \\ big\_force\ F \end{array} \right\}$ $x \leftarrow 1;$ $y \leftarrow 2;$

77 *semi* $\rightarrow$ *stmt*   ␣S empty statement

†78 *lproc* $\left\{ \begin{array}{c} if\_like \\ else\_like \\ define\_like \end{array} \right\}$ $\rightarrow$ *lproc*   **#include**   **#else**   **#define**

79 *lproc rproc* $\rightarrow$ *insert*   **#endif**

80 *lproc* $\left\{ \begin{array}{c} exp\ [exp] \\ function \end{array} \right\}$ *rproc* $\rightarrow$ *insert* I␣$\left\{ \begin{array}{c} E[␣\backslash 5E] \\ F \end{array} \right\}$ **#define** a 1   **#define** a { $b;$ }

81 *section_scrap semi* $\rightarrow$ *stmt*   *MS force*   ⟨ section name ⟩;

82 *section_scrap* $\rightarrow$ *exp*   ⟨ section name ⟩

83 *insert any* $\rightarrow$ *any* |#include|

84 *prelangle* $\rightarrow$ *binop*   <   < not in template

85 *prerangle* $\rightarrow$ *binop*   >   > not in template

133. Cont.

86 *langle prerangle* $\rightarrow cast$ $L \backslash, P$ $\langle \rangle$

87 *langle* $\left\{ \begin{array}{c} decl\_head \\ int\_like \\ exp \end{array} \right\}$ *prerangle* $\rightarrow cast$ $\langle$**class C**$\rangle$

88 *langle* $\left\{ \begin{array}{c} decl\_head \\ int\_like \\ exp \end{array} \right\}$ *comma* $\rightarrow langle$ $L \left\{ \begin{array}{c} D \\ I \\ E \end{array} \right\} C \, opt9$ $\langle$**class C,**

89 *template_like exp prelangle* $\rightarrow template\_like\ exp\ langle$ **template** $a\langle 100 \rangle$

90 *template_like* $\left\{ \begin{array}{c} exp \\ raw\_int \end{array} \right\}$ $\rightarrow \left\{ \begin{array}{c} exp \\ raw\_int \end{array} \right\}$ $T_\sqcup \left\{ \begin{array}{c} E \\ R \end{array} \right\}$ **C**::**template** $a(\,)$

91 *template_like* $\rightarrow raw\_int$ **template**$\langle$**class T**$\rangle$
92 *new_like lpar exp rpar* $\rightarrow new\_like$ **new**(*nothrow*)
93 *new_like cast* $\rightarrow exp$ $N_\sqcup C$ **new** (**int** $*$)
†94 *new_like* $\rightarrow new\_exp$ **new C**($\,$)

95 *new_exp* $\left\{ \begin{array}{c} int\_like \\ const\_like \end{array} \right\}$ $\rightarrow new\_exp$ $N_\sqcup \left\{ \begin{array}{c} I \\ C \end{array} \right\}$ **new const int**

96 *new_exp struct_like* $\left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$ $\rightarrow new\_exp$ $N_\sqcup S_\sqcup \left\{ \begin{array}{c} E \\ I \end{array} \right\}$ **new struct S**

97 *new_exp raw_ubin* $\rightarrow new\_exp$ $N\{R\}$ **new int**$*[2]$

98 *new_exp* $\left\{ \begin{array}{c} lpar \\ exp \end{array} \right\}$ $\rightarrow exp \left\{ \begin{array}{c} lpar \\ exp \end{array} \right\}$ $E = N \left\{ \begin{array}{c} \\ \sqcup \end{array} \right\}$ **operator**$[\,](\textbf{int})$
                                                                                                                     new int(2)

†99 *new_exp* $\rightarrow exp$ **new int**;

big_force $= {}^{\circ}220$, §122.
binop $= 3$, §117.
cancel $= {}^{\circ}211$, §122.
case_like $= 53$, §23.
cast $= 5$, §117.
catch_like $= 43$, §23.
colon $= 28$, §117.
comma $= 10$, §117.
const_like $= 50$, §23.
decl $= 20$, §117.
decl_head $= 9$, §117.
define_like $= 57$, §23.
do_like $= 46$, §23.
else_head $= 31$, §117.
else_like $= 26$, §23.

exp $= 1$, §117.
fn_decl $= 25$, §117.
force $= {}^{\circ}217$, §122.
function $= 24$, §117.
if_clause $= 32$, §117.
if_head $= 30$, §117.
if_like $= 47$, §23.
insert $= 37$, §117.
int_like $= 52$, §23.
langle $= 15$, §117.
lbrace $= 7$, §117.
lpar $= 11$, §117.
lproc $= 35$, §117.
new_exp $= 60$, §117.
new_like $= 42$, §23.

noop $= {}^{\circ}177$, §48.
opt $= {}^{\circ}214$, §122.
prelangle $= 13$, §117.
prerangle $= 14$, §117.
raw_int $= 51$, §23.
raw_ubin $= 49$, §23.
rpar $= 12$, §117.
rproc $= 36$, §117.
section_scrap $= 38$, §117.
semi $= 27$, §117.
stmt $= 23$, §117.
struct_like $= 55$, §23.
tag $= 29$, §117.
template_like $= 58$, §23.

134. Cont.

100 *ftemplate prelangle*	$\rightarrow$ *ftemplate langle*		*make_pair* $\langle$**int**, **int**$\rangle$
101 *ftemplate*	$\rightarrow$ *exp*		*make_pair* $(1, 2)$
102 *for_like exp*	$\rightarrow$ *else_like*	$F_{\sqcup}E$	**while** (1)
103 *raw_ubin const_like*	$\rightarrow$ *raw_ubin*	$RC\backslash_{\sqcup}$	$*$**const** x
104 *raw_ubin*	$\rightarrow$ *ubinop*		$* \, x$
105 *const_like*	$\rightarrow$ *int_like*		**const** x
106 *raw_int prelangle*	$\rightarrow$ *raw_int langle*		$\mathbf{C}\langle$
107 *raw_int colcol*	$\rightarrow$ *colcol*		$\mathbf{C}::$
108 *raw_int cast*	$\rightarrow$ *raw_int*		$\mathbf{C}\langle$**class T**$\rangle$
109 *raw_int lpar*	$\rightarrow$ *exp lpar*		**complex**(x, y)
†110 *raw_int*	$\rightarrow$ *int_like*		**complex** z

†111 *operator_like* $\left\{ \begin{array}{c} binop \\ unop \\ ubinop \end{array} \right\}$	$\rightarrow$ *exp*	$O\{\left\{ \begin{array}{c} B \\ U \\ U \end{array} \right\}\}$	**operator**$+$

112 *operator_like* $\left\{ \begin{array}{c} new\_like \\ delete\_like \end{array} \right\}$	$\rightarrow$ *exp*	$O_{\sqcup}\left\{ \begin{array}{c} N \\ S \end{array} \right\}$	**operator delete**

113 *operator_like comma*	$\rightarrow$ *exp*		**operator**,
†114 *operator_like*	$\rightarrow$ *new_exp*		**operator char**$*$
121 *delete_like lpar rpar*	$\rightarrow$ *delete_like*	$DL\backslash, R$	**delete**$[\,]$
122 *delete_like exp*	$\rightarrow$ *exp*	$D_{\sqcup}E$	**delete** p

†123 *question exp* $\left\{ \begin{array}{c} colon \\ base \end{array} \right\}$	$\rightarrow$ *binop*		$? \, x :$ $? \, f(\,) :$

124 *begin_arg end_arg*	$\rightarrow$ *exp*		@[**char**$*$@]
125 *any_other end_arg*	$\rightarrow$ *end_arg*		**char**$*$@]

†200 *typedef_like decl_head* $\left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$	$\rightarrow$ *typedef_like decl_head*	$D = D\left\{ \begin{array}{c} E^{**} \\ I^{**} \end{array} \right\}$	**typedef char ch**;

201 *typedef_like decl_head semi*	$\rightarrow$ *decl*	$T_{\sqcup}D$	**typedef int** x, y;
†202 *typedef_like int_like raw_int*	$\rightarrow$ *typedef_like int_like exp*		**typedef int foo**

135.   †**Notes**

Rule 35: The *exp* must not be immediately followed by *lpar*, *exp*, or *cast*.

Rule 48: The *exp* or *int_like* must not be immediately followed by *base*.

Rule 76: The *force* in the *stmt* line becomes *bsp* if `CWEAVE` has been invoked with the `-f` option.

Rule 78: The *define_like* case calls *make_underlined* on the following scrap.

Rule 94: The *new_like* must not be immediately followed by *lpar*.

Rule 99: The *new_exp* must not be immediately followed by *raw_int*, *struct_like*, or *colcol*.

Rule 110: The *raw_int* must not be immediately followed by *langle*.

Rule 111: The operator after *operator_like* must not be immediately followed by a *binop*.

Rule 114: The *operator_like* must not be immediately followed by *raw_ubin*.

Rule 123: The mathness of the *colon* or *base* changes to 'yes'.

Rule 200: The *exp* must not be immediately followed by *lpar* or *exp*.

Rule 202: The *raw_int* must be immediately followed by *semi* or *comma*.

136.   Implementing the productions.   More specifically, a scrap is a structure consisting of a category *cat* and a **text_pointer** *trans*, which points to the translation in *tok_start*. When C text is to be processed with the grammar above, we form an array *scrap_info* containing the initial scraps. Our production rules have the nice property that the right-hand side is never longer than the left-hand side. Therefore it is convenient to use sequential allocation for the current sequence of scraps. Five pointers are used to manage the parsing:

> *pp* is a pointer into *scrap_info*. We will try to match the category codes $pp\text{-}cat$, $(pp + 1)\text{-}cat$, ... to the left-hand sides of productions.

> *scrap_base*, *lo_ptr*, *hi_ptr*, and *scrap_ptr* are such that the current sequence of scraps appears in positions *scrap_base* through *lo_ptr* and *hi_ptr* through *scrap_ptr*, inclusive, in the *cat* and *trans* arrays. Scraps located between *scrap_base* and *lo_ptr* have been examined, while those in positions $\geq hi\_ptr$ have not yet been looked at by the parsing process.

Initially *scrap_ptr* is set to the position of the final scrap to be parsed, and it doesn't change its value. The parsing process makes sure that $lo\_ptr \geq pp + 3$, since productions have as many as four terms, by moving scraps from *hi_ptr* to *lo_ptr*. If there are fewer than $pp + 3$ scraps left, the positions up to $pp + 3$ are filled with blanks that will not match in any productions. Parsing stops when $pp \equiv lo\_ptr + 1$ and $hi\_ptr \equiv scrap\_ptr + 1$.

Since the *scrap* structure will later be used for other purposes, we declare its second element as a union.

⟨ Typedef declarations 25 ⟩ +≡
      typedef struct {
        eight_bits *cat*;
        eight_bits *mathness*;
        union {
          text_pointer *Trans*;

          ⟨ Rest of *trans_plus* union 286 ⟩
        } *trans_plus*;
      } **scrap**;
      typedef scrap *∗**scrap_pointer**;

137.   #define *trans*   *trans_plus.Trans*   ▷ translation texts of scraps ◁

⟨ Private variables 26 ⟩ +≡
   static scrap *scrap_info*[*max_scraps*];   ▷ memory array for scraps ◁
   static scrap *null_scrap*;   ▷ a scrap with empty translation ◁
   static scrap_pointer *scrap_info_end* ← *scrap_info* + *max_scraps* − 1;
      ▷ end of *scrap_info* ◁
   static scrap_pointer *scrap_base*;   ▷ beginning of the current scrap sequence ◁
   static scrap_pointer *scrap_ptr*;   ▷ ending of the current scrap sequence ◁
   static scrap_pointer *max_scr_ptr*;   ▷ largest value assumed by *scrap_ptr* ◁
   static scrap_pointer *pp*;   ▷ current position for reducing productions ◁
   static scrap_pointer *lo_ptr*;   ▷ last scrap that has been examined ◁
   static scrap_pointer *hi_ptr*;   ▷ first scrap that has not been examined ◁

138.   ⟨ Set initial values 29 ⟩ +≡
   $null\_scrap.trans \leftarrow \&tok\_start[0];$ $scrap\_base \leftarrow scrap\_info + 1;$
   $max\_scr\_ptr \leftarrow scrap\_ptr \leftarrow scrap\_info;$

139. Token lists in *tok_mem* are composed of the following kinds of items for TEX output.

- Character codes and special codes like *force* and *math_rel* represent themselves;
- *id_flag* + *p* represents \\{identifier *p*};
- *res_flag* + *p* represents \&{identifier *p*};
- *section_flag* + *p* represents section name *p*;
- *tok_flag* + *p* represents token list number *p*;
- *inner_tok_flag* + *p* represents token list number *p*, to be translated without line-break controls.

#define *id_flag* 10240   ▷ signifies an identifier ◁
#define *res_flag* 2 * *id_flag*   ▷ signifies a reserved word ◁
#define *section_flag* 3 * *id_flag*   ▷ signifies a section name ◁
#define *tok_flag* 4 * *id_flag*   ▷ signifies a token list ◁
#define *inner_tok_flag* 5 * *id_flag*   ▷ signifies a token list in '| . . . |' ◁
#if 0
  static void *print_text*(▷ prints a token list for debugging; not used in *main* ◁
      text_pointer *p*)
  {
    token_pointer *j*;   ▷ index into *tok_mem* ◁
    sixteen_bits *r*;   ▷ remainder of token after the flag has been stripped off ◁

    if ($p \geq text\_ptr$) *printf*("BAD");
    else
      for ($j \leftarrow *p$; $j < *(p+1)$; $j{+}{+}$) {
        $r \leftarrow *j \% id\_flag$;
        switch ($*j/id\_flag$) {
        case 1: *printf*("\\\\{"); *print_id*((*name_dir* + *r*)); *printf*("}"); **break**;
            ▷ *id_flag* ◁
        case 2: *printf*("\\&{"); *print_id*((*name_dir* + *r*)); *printf*("}"); **break**;
            ▷ *res_flag* ◁
        case 3: *printf*("<"); *print_section_name*((*name_dir* + *r*)); *printf*(">"); **break**;
            ▷ *section_flag* ◁
        case 4: *printf*("[[%d]]", *r*); **break**;   ▷ *tok_flag* ◁
        case 5: *printf*("|[[%d]]|", *r*); **break**;   ▷ *inner_tok_flag* ◁
        default: ⟨Print token *r* in symbolic form 141⟩
        }
      }
    printf("|\n"); *update_terminal*;
  }
#endif

140. ⟨Predeclaration of procedures 11⟩ +≡
#if 0
  static void *print_text*(**text_pointer** *p*);
#endif

141. ⟨ Print token r in symbolic form 141 ⟩ ≡
```
  switch (r) {
  case math_rel: printf ("\\mathrel{"); break;
  case big_cancel: printf ("[ccancel]"); break;
  case cancel: printf ("[cancel]"); break;
  case indent: printf ("[indent]"); break;
  case outdent: printf ("[outdent]"); break;
  case backup: printf ("[backup]"); break;
  case opt: printf ("[opt]"); break;
  case break_space: printf ("[break]"); break;
  case force: printf ("[force]"); break;
  case big_force: printf ("[fforce]"); break;
  case preproc_line: printf ("[preproc]"); break;
  case quoted_char: j++; printf ("[%o]", (unsigned int) *j); break;
  case end_translation: printf ("[quit]"); break;
  case inserted: printf ("[inserted]"); break;
  default: putxchar (r);
  }
```
This code is used in section 139.

$backup = °215$, §122.
$big\_cancel = °210$, §122.
$big\_force = °220$, §122.
$break\_space = °216$, §122.
$cancel = °211$, §122.
$end\_translation = °223$, §122.
$force = °217$, §122.
$indent = °212$, §122.
$inserted = °224$, §122.
main: **int** ( ), §5.

$math\_rel = °206$, §122.
name_dir: **name_info** [ ],
  COMMON.W §43.
$opt = °214$, §122.
$outdent = °213$, §122.
$preproc\_line = °221$, §122.
print_id = macro ( ), §13.
print_section_name: **void** ( ),
  COMMON.W §52.
printf , <stdio.h>.

putxchar = macro, §18.
$quoted\_char = °222$, §122.
sixteen_bits = **uint16_t**, §6.
text_pointer = **token_pointer**
  ∗, §39.
text_ptr: **static text_pointer**,
  §40.
tok_mem: **static token** [ ], §40.
token_pointer = **token** ∗, §39.
update_terminal = macro, §18.

142. The production rules listed above are embedded directly into `CWEAVE`, since it is easier to do this than to write an interpretive system that would handle production systems in general. Several macros are defined here so that the program for each production is fairly short.

All of our productions conform to the general notion that some k consecutive scraps starting at some position j are to be replaced by a single scrap of some category c whose translation is composed from the translations of the disappearing scraps. After this production has been applied, the production pointer pp should change by an amount d. Such a production can be represented by the quadruple (j, k, c, d). For example, the production '*exp comma exp* $\rightarrow$ *exp*' would be represented by '$(pp, 3, exp, -2)$'; in this case the pointer pp should decrease by 2 after the production has been applied, because some productions with *exp* in their second or third positions might now match, but no productions have *exp* in the fourth position of their left-hand sides. Note that the value of d is determined by the whole collection of productions, not by an individual one. The determination of d has been done by hand in each case, based on the full set of productions but not on the grammar of C or on the rules for constructing the initial scraps.

We also attach a serial number to each production, so that additional information is available when debugging. For example, the program below contains the statement '*reduce*$(pp, 3, exp, -2, 4)$' when it implements the production just mentioned.

Before calling *reduce*, the program should have appended the tokens of the new translation to the *tok_mem* array. We commonly want to append copies of several existing translations, and macros are defined to simplify these common cases. For example, *app2*(pp) will append the translations of two consecutive scraps, $pp \rightarrow trans$ and $(pp + 1) \rightarrow trans$, to the current token list. If the entire new translation is formed in this way, we write '*squash*(j, k, c, d, n)' instead of '*reduce*(j, k, c, d, n)'. For example, '*squash*$(pp, 3, exp, -2, 3)$' is an abbreviation for '*app3*(pp); *reduce*$(pp, 3, exp, -2, 3)$'.

A couple more words of explanation: Both *big_app* and *app* append a token (while *big_app1* to *big_app3* append the specified number of scrap translations) to the current token list. The difference between *big_app* and *app* is simply that *big_app* checks whether there can be a conflict between math and non-math tokens, and intercalates a '`$`' token if necessary. When in doubt what to use, use *big_app*.

The *mathness* is an attribute of scraps that says whether they are to be printed in a math mode context or not. It is separate from the "part of speech" (the *cat*) because to make each *cat* have a fixed *mathness* (as in the original `WEAVE`) would multiply the number of necessary production rules.

The low two bits (i.e. *mathness* % 4) control the left boundary. (We need two bits because we allow cases *yes_math*, *no_math* and *maybe_math*, which can go either way.) The next two bits (i.e. *mathness* /4) control the right boundary. If we combine two scraps and the right boundary of the first has a different mathness from the left boundary of the second, we insert a `$` in between. Similarly, if at printing time some irreducible scrap has a *yes_math* boundary the scrap gets preceded or followed by a `$`. The left boundary is *maybe_math* if and only if the right boundary is.

143.   The code below is an exact translation of the production rules into C, using such macros, and the reader should have no difficulty understanding the format by comparing the code with the symbolic productions as they were listed earlier.

#define *no_math* 2   ▷ should be in horizontal mode ◁
#define *yes_math* 1   ▷ should be in math mode ◁
#define *maybe_math* 0   ▷ works in either horizontal or math mode ◁
#define *big_app2*(*a*)   *big_app1*(*a*); *big_app1*(*a* + 1)
#define *big_app3*(*a*)   *big_app2*(*a*); *big_app1*(*a* + 2)
#define *app*(*a*)   *(*tok_ptr*++) ← (**token**)(*a*)
#define *app1*(*a*)   *(*tok_ptr*++) ← (**token**)(*tok_flag* + (**int**)((*a*)→*trans* − *tok_start*))

⟨ Private variables 26 ⟩ +≡
  static int *cur_mathness*, *init_mathness*;

144.   ⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *app_str*(**const char** ∗);
  static void *big_app*(**token**);
  static void *big_app1*(**scrap_pointer**);

---

big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
cat: **eight_bits**, §136.
comma = 10, §117.
exp = 1, §117.
mathness: **eight_bits**, §136.

reduce: **static void** ( ), §207.
scrap_pointer = **scrap** ∗,
  §136.
squash: **static void** ( ), §208.
tok_flag = macro, §139.
tok_mem: **static token** [ ], §40.

tok_ptr: **static token_pointer**,
  §40.
tok_start: **static**
  token_pointer [ ], §40.
token = **sixteen_bits**, §39.
trans = macro, §137.

145.   **static void** *app_str*(**const char** *∗s*)
{
   while (*∗s*) *app_tok*(*∗*(*s*++));
}
static void *big_app*(**token** *a*)
{
   if (*a* ≡ '␣' ∨ (*a* ≥ *big_cancel* ∧ *a* ≤ *big_force*))   ▷ non-math token ◁
   {
      if (*cur_mathness* ≡ *maybe_math*) *init_mathness* ← *no_math*;
      else if (*cur_mathness* ≡ *yes_math*) *app_str*(`"{}$"`);
      cur_mathness ← *no_math*;
   }
   else {
      if (*cur_mathness* ≡ *maybe_math*) *init_mathness* ← *yes_math*;
      else if (*cur_mathness* ≡ *no_math*) *app_str*(`"${}"`);
      cur_mathness ← *yes_math*;
   }
   app(*a*);
}
static void *big_app1*(**scrap_pointer** *a*)
{
   switch (*a*→*mathness* % 4) {   ▷ left boundary ◁
   case (*no_math*):
      if (*cur_mathness* ≡ *maybe_math*) *init_mathness* ← *no_math*;
      else if (*cur_mathness* ≡ *yes_math*) *app_str*(`"{}$"`);
      cur_mathness ← *a*→*mathness*/4;   ▷ right boundary ◁
      break;
   case (*yes_math*):
      if (*cur_mathness* ≡ *maybe_math*) *init_mathness* ← *yes_math*;
      else if (*cur_mathness* ≡ *no_math*) *app_str*(`"${}"`);
      cur_mathness ← *a*→*mathness*/4;   ▷ right boundary ◁
      break;
   case (*maybe_math*):   ▷ no changes ◁
      break;
   }
   app(*tok_flag* + (**int**)((*a*)→*trans* − *tok_start*));
}

146.   Let us consider the big switch for productions now, before looking at its context. We want to design the program so that this switch works, so we might as well not keep ourselves in suspense about exactly what code needs to be provided with a proper environment.

#define *cat1* $(pp + 1)$→*cat*
#define *cat2* $(pp + 2)$→*cat*
#define *cat3* $(pp + 3)$→*cat*
#define *lhs_not_simple*
          $(pp$→$cat \neq public\_like \wedge pp$→$cat \neq semi \wedge pp$→$cat \neq prelangle \wedge pp$→$cat \neq prerangle$
              $\wedge\ pp$→$cat \neq template\_like \wedge pp$→$cat \neq new\_like \wedge pp$→$cat \neq new\_exp \wedge pp$→$cat \neq$
              ftemplate
              $\wedge\ pp$→$cat \neq raw\_ubin \wedge pp$→$cat \neq const\_like \wedge pp$→$cat \neq raw\_int \wedge pp$→$cat \neq$
              operator_like)   ▷ not a production with left side length 1 ◁

⟨ Match a production at *pp*, or increase *pp* if there is no match 146 ⟩ ≡
   {
      if $(cat1 \equiv end\_arg \wedge lhs\_not\_simple)$
         if $(pp$→$cat \equiv begin\_arg)$ $squash(pp, 2, exp, -2, 124)$;
         else $squash(pp, 2, end\_arg, -1, 125)$;
      else if $(cat1 \equiv insert)$ $squash(pp, 2, pp$→$cat, -2, 0)$;
      else if $(cat2 \equiv insert)$ $squash(pp + 1, 2, (pp + 1)$→$cat, -1, 0)$;
      else if $(cat3 \equiv insert)$ $squash(pp + 2, 2, (pp + 2)$→$cat, 0, 0)$;
      else
         switch $(pp$→$cat)$ {
            ⟨ Cases for *pp*→*cat* 147 ⟩
         }
      pp ++;   ▷ if no match was found, we move to the right ◁
   }
This code is used in section 209.

---

147. ⟨ Cases for *pp→cat* 147 ⟩ ≡
case *exp*: ⟨ Cases for *exp* 158 ⟩ **break**;
case *lpar*: ⟨ Cases for *lpar* 159 ⟩ **break**;
case *unop*: ⟨ Cases for *unop* 160 ⟩ **break**;
case *ubinop*: ⟨ Cases for *ubinop* 161 ⟩ **break**;
case *binop*: ⟨ Cases for *binop* 162 ⟩ **break**;
case *cast*: ⟨ Cases for *cast* 163 ⟩ **break**;
case *sizeof_like*: ⟨ Cases for *sizeof_like* 164 ⟩ **break**;
case *int_like*: ⟨ Cases for *int_like* 165 ⟩ **break**;
case *public_like*: ⟨ Cases for *public_like* 166 ⟩ **break**;
case *colcol*: ⟨ Cases for *colcol* 167 ⟩ **break**;
case *decl_head*: ⟨ Cases for *decl_head* 168 ⟩ **break**;
case *decl*: ⟨ Cases for *decl* 169 ⟩ **break**;
case *base*: ⟨ Cases for *base* 170 ⟩ **break**;
case *struct_like*: ⟨ Cases for *struct_like* 171 ⟩ **break**;
case *struct_head*: ⟨ Cases for *struct_head* 172 ⟩ **break**;
case *fn_decl*: ⟨ Cases for *fn_decl* 173 ⟩ **break**;
case *function*: ⟨ Cases for *function* 174 ⟩ **break**;
case *lbrace*: ⟨ Cases for *lbrace* 175 ⟩ **break**;
case *if_like*: ⟨ Cases for *if_like* 176 ⟩ **break**;
case *else_like*: ⟨ Cases for *else_like* 177 ⟩ **break**;
case *else_head*: ⟨ Cases for *else_head* 178 ⟩ **break**;
case *if_clause*: ⟨ Cases for *if_clause* 179 ⟩ **break**;
case *if_head*: ⟨ Cases for *if_head* 180 ⟩ **break**;
case *do_like*: ⟨ Cases for *do_like* 181 ⟩ **break**;
case *case_like*: ⟨ Cases for *case_like* 182 ⟩ **break**;
case *catch_like*: ⟨ Cases for *catch_like* 183 ⟩ **break**;
case *tag*: ⟨ Cases for *tag* 184 ⟩ **break**;
case *stmt*: ⟨ Cases for *stmt* 186 ⟩ **break**;
case *semi*: ⟨ Cases for *semi* 187 ⟩ **break**;
case *lproc*: ⟨ Cases for *lproc* 188 ⟩ **break**;
case *section_scrap*: ⟨ Cases for *section_scrap* 189 ⟩ **break**;
case *insert*: ⟨ Cases for *insert* 190 ⟩ **break**;
case *prelangle*: ⟨ Cases for *prelangle* 191 ⟩ **break**;
case *prerangle*: ⟨ Cases for *prerangle* 192 ⟩ **break**;
case *langle*: ⟨ Cases for *langle* 193 ⟩ **break**;
case *template_like*: ⟨ Cases for *template_like* 194 ⟩ **break**;
case *new_like*: ⟨ Cases for *new_like* 195 ⟩ **break**;
case *new_exp*: ⟨ Cases for *new_exp* 196 ⟩ **break**;
case *ftemplate*: ⟨ Cases for *ftemplate* 197 ⟩ **break**;
case *for_like*: ⟨ Cases for *for_like* 198 ⟩ **break**;
case *raw_ubin*: ⟨ Cases for *raw_ubin* 199 ⟩ **break**;
case *const_like*: ⟨ Cases for *const_like* 200 ⟩ **break**;
case *raw_int*: ⟨ Cases for *raw_int* 201 ⟩ **break**;
case *operator_like*: ⟨ Cases for *operator_like* 202 ⟩ **break**;
case *typedef_like*: ⟨ Cases for *typedef_like* 203 ⟩ **break**;
case *delete_like*: ⟨ Cases for *delete_like* 204 ⟩ **break**;
case *question*: ⟨ Cases for *question* 205 ⟩ **break**;

This code is used in section 146.

148.   In C, new specifier names can be defined via **typedef**, and we want to make
the parser recognize future occurrences of the identifier thus defined as specifiers.
This is done by the procedure *make_reserved*, which changes the *ilk* of the relevant
identifier.

We first need a procedure to recursively seek the first identifier in a token list,
because the identifier might be enclosed in parentheses, as when one defines a function
returning a pointer.

If the first identifier found is a keyword like '**case**', we return the special value
case_found; this prevents underlining of identifiers in case labels. If the first identifier
is the keyword '**operator**', we give up; users who want to index definitions of over-
loaded C++ operators should say, for example, '`@!@^\&{operator} $+{=}$@>`' (or,
more properly alphabetized, '`@!@:operator+=}{\&{operator} $+{=}$@>`').

#define *no_ident_found*   (**token_pointer**) 0   ▷ distinct from any identifier token ◁
#define *case_found*   (**token_pointer**) 1   ▷ likewise ◁
#define *operator_found*   (**token_pointer**) 2   ▷ likewise ◁

⟨ Predeclaration of procedures 11 ⟩ +≡
    static token_pointer *find_first_ident*(**text_pointer**);
    static void *make_reserved*(**scrap_pointer**);
    static void *make_underlined*(**scrap_pointer**);
    static void *underline_xref*(**name_pointer**);

---

149. **static token_pointer** *find_first_ident* (**text_pointer** *p*)
{
    token_pointer *q*;   ▷ token to be returned ◁
    token_pointer *j*;   ▷ token being looked at ◁
    sixteen_bits *r*;   ▷ remainder of token after the flag has been stripped off ◁

    if ($p \geq text\_ptr$) *confusion*(`"find_first_ident"`);
    for ($j \leftarrow *p;\ j < *(p+1);\ j{+}{+}$) {
      $r \leftarrow *j \% id\_flag$;
      switch ($*j/id\_flag$) {
      case 2:   ▷ *res_flag* ◁
        if ($name\_dir[r].ilk \equiv case\_like$) **return** *case_found*;
        if ($name\_dir[r].ilk \equiv operator\_like$) **return** *operator_found*;
        if ($name\_dir[r].ilk \neq raw\_int$) **break**;
      case 1: **return** *j*;
      case 4: **case** 5:   ▷ *tok_flag* or *inner_tok_flag* ◁
        if (($q \leftarrow find\_first\_ident(tok\_start + r)) \neq no\_ident\_found$) **return** *q*;
      default: ;   ▷ char, *section_flag*, fall thru: move on to next token ◁
        if ($*j \equiv inserted$) **return** *no_ident_found*;   ▷ ignore inserts ◁
        else if ($*j \equiv qualifier$) $j{+}{+}$;   ▷ bypass namespace qualifier ◁
      }
    }
    return *no_ident_found*;
}

150.   The scraps currently being parsed must be inspected for any occurrence of the identifier that we're making reserved; hence the **for** loop below.

We use the fact that *make_underlined* has been called immediately preceding *make_reserved*, hence *tok_loc* has been set.

> **static token_pointer** *tok_loc*;   ▷ where the first identifier appears ◁
>
> **static void** *make_reserved*(▷ make the first identifier in *p⃗trans* like **int** ◁
>   **scrap_pointer** *p*)
> {
>   **sixteen_bits** *tok_value*;   ▷ the name of this identifier, plus its flag ◁
>   **if** (*tok_loc* ≤ *operator_found*) **return**;   ▷ this should not happen ◁
>   *tok_value* ← *∗tok_loc*;
>   **for** (; *p* ≤ *scrap_ptr*; *p* ≡ *lo_ptr* ? *p* ← *hi_ptr* : *p*++) {
>   **if** (*p⃗cat* ≡ *exp*) {
>   **if** (∗∗(*p⃗trans*) ≡ *tok_value*) {
>   *p⃗cat* ← *raw_int*; ∗∗(*p⃗trans*) ← *tok_value* % *id_flag* + *res_flag*;
>   }
>   }
>   }
>   (*name_dir* + (**sixteen_bits**)(*tok_value* % *id_flag*))⃗*ilk* ← *raw_int*;
>   ∗*tok_loc* ← *tok_value* % *id_flag* + *res_flag*;
> }

151.   In the following situations we want to mark the occurrence of an identifier as a definition: when *make_reserved* is just about to be used; after a specifier, as in **char** ∗∗*argv*; before a colon, as in *found*:; and in the declaration of a function, as in *main*(){...;}. This is accomplished by the invocation of *make_underlined* at appropriate times. Notice that, in the declaration of a function, we find out that the identifier is being defined only after it has been swallowed up by an *exp*.

> **static void** *make_underlined*(▷ underline the entry for the first identifier in *p⃗trans* ◁
>   **scrap_pointer** *p*)
> {
>   **if** ((*tok_loc* ← *find_first_ident*(*p⃗trans*)) ≤ *operator_found*) **return**;
>   ▷ this happens, for example, in **case** *found*: ◁
>   *xref_switch* ← *def_flag*; *underline_xref*(∗*tok_loc* % *id_flag* + *name_dir*);
> }

---

152. We cannot use *new_xref* to underline a cross-reference at this point because this would just make a new cross-reference at the end of the list. We actually have to search through the list for the existing cross-reference.

```
static void underline_xref (name_pointer p)
{
    xref_pointer q ← (xref_pointer) p→xref ;
        ▷ pointer to cross-reference being examined ◁
    xref_pointer r;      ▷ temporary pointer for permuting cross-references ◁
    sixteen_bits m;      ▷ cross-reference value to be installed ◁
    sixteen_bits n;      ▷ cross-reference value being examined ◁

    if (no_xref ) return;
    m ← section_count + xref_switch ;
    while (q ≠ xmem ) {
        n ← q→num;
        if (n ≡ m) return;
        else if (m ≡ n + def_flag) {
            q→num ← m;  return;
        }
        else if (n ≥ def_flag ∧ n < m) break;
        q ← q→xlink ;
    }
    ⟨ Insert new cross-reference at q, not at beginning of list  153 ⟩
}
```

153. We get to this section only when the identifier is one letter long, so it didn't get a non-underlined entry during phase one. But it may have got some explicitly underlined entries in later sections, so in order to preserve the numerical order of the entries in the index, we have to insert the new cross-reference not at the beginning of the list (namely, at *p→xref*), but rather right before *q*.

⟨ Insert new cross-reference at q, not at beginning of list 153 ⟩ ≡

```
    append_xref (0);     ▷ this number doesn't matter ◁
    xref_ptr→xlink ← (xref_pointer) p→xref ;  r ← xref_ptr ;  update_node (p);
    while (r→xlink ≠ q) {
        r→num ← r→xlink→num;  r ← r→xlink ;
    }
    r→num ← m;     ▷ everything from q on is left undisturbed ◁
```

This code is used in section 152.

154.   CTWILL needs the following procedure, which appends tokens of a translated text until coming to *tok_loc*, then suppresses text that may appear between parentheses or brackets. The calling routine should set *ident_seen* ← *false* first. (This is admittedly tricky.)

boolean *ident_seen*;
static boolean *app_supp*(**text_pointer** *p*)
{
   token_pointer *j*;
   text_pointer *q*;
   if (*ident_seen* ∧ ∗∗*p* ≥ *tok_flag*) {
      q ← ∗∗*p* − *tok_flag* + *tok_start*;
      if (∗∗*q* ≡ '(') {
         app('('); *app*('\\'); *app*(','); *app*(')'); **goto** *catch14*;
      }
      if (∗∗*q* ≡ '[') {
         app('['); *app*('\\'); *app*(','); *app*(']'); **goto** *catch14*;
      }
   }
   for (*j* ← ∗*p*; *j* < ∗(*p* + 1); *j*++) {
      if (∗*j* < *tok_flag*) {
         if (∗*j* ≡ *inserted*) **return** *false*;
         if (*j* ≡ *tok_loc*) *ident_seen* ← *true*;
         else *app*(∗*j*);
      }
      else if (∗*j* ≥ *inner_tok_flag*) *confusion*(_("inner"));
      else if (*app_supp*(∗*j* − *tok_flag* + *tok_start*)) **goto** *catch14*;
      ;
   }
   return *false*;
catch14:
   if (∗(∗(*p* + 1) − 1) ≡ '9') **return** *true*;   ▷ production 14 was used ◁
   else return *false*;
}

_ = macro ( ), §7.
app = macro ( ), §143.
append_xref = macro ( ), §35.
boolean = **bool**, §6.
confusion = macro ( ), §15.
def_flag = macro, §34.
false, **<stdbool.h>**.
inner_tok_flag = macro, §139.
inserted = °224, §122.
name_pointer = **name_info**
  ∗, §13.
new_xref: **static void** ( ), §36.
no_xref = macro, §35.

num: **sixteen_bits**, §25.
section_count: **sixteen_bits**,
  COMMON.W §37.
sixteen_bits = **uint16_t**, §6.
text_pointer = **token_pointer**
  ∗, §39.
tok_flag = macro, §139.
tok_loc: **static token_pointer**,
  §150.
tok_start: **static**
  token_pointer [], §40.
token_pointer = **token** ∗, §39.
true, **<stdbool.h>**.

update_node: **static void** ( ),
  §43.
xlink: **struct xref_info** ∗, §25.
xmem: **static xref_info** [],
  §26.
xref = macro, §34.
xref_pointer = **xref_info** ∗,
  §25.
xref_ptr: **static xref_pointer**,
  §26.
xref_switch: **static**
  sixteen_bits, §26.

155. The trickiest part of CTWILL is the procedure *make_ministring*(*l*), which tries to figure out a symbolic form of definition after *make_underlined*(*pp* + *l*) has been called. We rely heavily on the existing productions, which force the translated texts to have a structure that's decodable even though the underlying *cat* and *mathness* codes have disappeared.

```
static void make_ministring(int l)      ▷ 0, 1, or 2 ◁
{ text_pointer q, r;
  name_pointer cn;
  token t;
  int ast_count;      ▷ asterisks preceding the expression ◁
  boolean non_ast_seen;      ▷ have we seen a non-asterisk? ◁

  if (tok_loc ≤ operator_found) return;
  cn ← ((∗tok_loc) % id_flag) + name_dir;
  ⟨Append the type of the declaree; return if it begins with extern 156⟩;
  null_scrap.mathness ← (((pp + l)→mathness) % 4) ∗ 5;  big_app1(&null_scrap);
      ▷ now we're ready for the mathness that follows (I think) ◁
      ▷ (without the mod 4 times 5, comments posed a problem) ◁
      ▷ (namely in cases like int a(b, c) followed by comment) ◁
  ident_seen ← false;  app_supp((pp + l)→trans);  null_scrap.mathness ← 10;
  big_app1(&null_scrap);      ▷ now cur_mathness ≡ no_math ◁
  ms_mode ← true;  ministring_ptr ← ministring_buf;
  if (l ≡ 2) ∗ministring_ptr ++ ← '=';
  make_output();      ▷ translate the current text into a ministring ◁
  tok_ptr ← ∗(−−text_ptr);      ▷ delete that text ◁
  new_meaning(cn);  cur_mathness ← maybe_math;      ▷ restore it ◁
}
```

156. Here we use the fact that a *decl_head* comes from *int_like* only in production 27, whose translation is fairly easy to recognize. (Well, production 28 has been added for C++, but we hope that doesn't mess us up.) And we also use other similar facts.

If an identifier is given an **extern** definition, we don't change its current meaning, but we do suppress mini-index entries to its current meaning in other sections.

⟨Append the type of the declaree; **return** if it begins with **extern** 156⟩ ≡
```
  if (l ≡ 0) {
    app(int_loc + res_flag);  app('␣');  cur_mathness ← no_math;
  }
  else {
    q ← (pp + l − 1)→trans;  ast_count ← 0;  non_ast_seen ← false;
    while (true) {
      if (∗(q + 1) ≡ ∗q + 1) {
        r ← q; break;      ▷ e.g. struct; we're doing production 45 or 46 ◁
      }
      if (∗∗q < tok_flag) confusion(_("find␣type"));
      r ← ∗∗q − tok_flag + tok_start;
      if ((t ← ∗(∗(q + 1) − 2)) ≥ tok_flag ∧ ∗∗(t − tok_flag + tok_start) ≡ '∗') {
          ▷ production 34 ◁
        if (¬non_ast_seen) ast_count ++;      ▷ count immediately preceding ∗'s ◁
      }
```

        else $non\_ast\_seen \leftarrow true$;

        if $(*(*q + 1) \equiv$ '␣' $\wedge *(q + 1) \equiv *q + 2)$ **break**;   ▷ production 27 ◁

        if $(*(*q + 1) \equiv$ '{' $\wedge *(*q + 2) \equiv$ '}' $\wedge *(*q + 3) \equiv$ '\$' $\wedge *(*q + 4) \equiv$ '␣'

            $\wedge *(q + 1) \equiv *q + 5)$ **break**;   ▷ production 27 in disguise ◁

        $q \leftarrow r$;

      }

    while $(**r \geq tok\_flag)$ {

        if $(*(r + 1) > *r + 9 \wedge *(*r + 1) \equiv$ '{' $\wedge *(*r + 2) \equiv$ '}' $\wedge *(*r + 3) \equiv$ '\$'

            $\wedge *(*r + 4) \equiv indent)$ $q \leftarrow **r - tok\_flag + tok\_start$;   ▷ production 49 ◁

        $r \leftarrow **r - tok\_flag + tok\_start$;

    }

    if $(**r \equiv ext\_loc + res\_flag)$ **return**;   ▷ **extern** gives no definition ◁

    ⟨ Append tokens for type q 157 ⟩;

  }

This code is used in section 155.

157. ⟨ Append tokens for type q 157 ⟩ $\equiv$

  $cur\_mathness \leftarrow no\_math$;   ▷ it was $maybe\_math$ ◁

  if $(*(q + 1) \equiv *q + 8 \wedge *(*q + 1) \equiv$ '␣' $\wedge *(*q + 3) \equiv$ '␣') {

    $app(**q)$; $app($'␣'$)$; $app(*(*q + 2))$;   ▷ production 46 ◁

  }

  else if $((t \leftarrow *(*(q + 1) - 1)) \geq tok\_flag \wedge **(r \leftarrow t - tok\_flag + tok\_start) \equiv$

        '\\' $\wedge *(*r + 1) \equiv$ '{') $app(**q)$;   ▷ $struct\_like$ identifier ◁

  else $app((q - tok\_start) + tok\_flag)$;

  while (ast_count) {

    $big\_app($'{'$)$; $app($'*'$)$; $app($'}'$)$; ast_count −−;

  }

This code is used in section 156.

158. Now comes the code that tries to match each production starting with a particular type of scrap. Whenever a match is discovered, the *squash* or *reduce* macro will cause the appropriate action to be performed, followed by **goto** *found*.

⟨ Cases for *exp* 158 ⟩ ≡
  if (*cat1* ≡ *lbrace* ∨ *cat1* ≡ *int_like* ∨ *cat1* ≡ *decl*) {
    make_underlined(*pp*); *make_ministring*(0); *big_app1*(*pp*);
    if (*indent_param_decl*) {
      big_app(*indent*); *app*(*indent*);
    }
    reduce(*pp*, 1, *fn_decl*, 0, 1);
  }
  else if (*cat1* ≡ *unop*) *squash*(*pp*, 2, *exp*, −2, 2);
  else if ((*cat1* ≡ *binop* ∨ *cat1* ≡ *ubinop*) ∧ *cat2* ≡ *exp*) *squash*(*pp*, 3, *exp*, −2, 3);
  else if (*cat1* ≡ *comma* ∧ *cat2* ≡ *exp*) {
    big_app2(*pp*); *app*(*opt*); *app*('9'); *big_app1*(*pp* + 2); *reduce*(*pp*, 3, *exp*, −2, 4);
  }
  else if (*cat1* ≡ *lpar* ∧ *cat2* ≡ *rpar* ∧ *cat3* ≡ *colon*) *squash*(*pp* + 3, 1, *base*, 0, 5);
  else if (*cat1* ≡ *cast* ∧ *cat2* ≡ *colon*) *squash*(*pp* + 2, 1, *base*, 0, 5);
  else if (*cat1* ≡ *semi*) *squash*(*pp*, 2, *stmt*, −1, 6);
  else if (*cat1* ≡ *colon*) {
    make_underlined(*pp*);
    if (*tok_loc* > *operator_found*) {
      name_pointer *cn* ← ((*∗tok_loc*) % *id_flag*) + *name_dir*;
      strcpy(*ministring_buf*, "label"); *new_meaning*(*cn*);
    }
    squash(*pp*, 2, *tag*, −1, 7);
  }
  else if (*cat1* ≡ *rbrace*) *squash*(*pp*, 1, *stmt*, −1, 8);
  else if (*cat1* ≡ *lpar* ∧ *cat2* ≡ *rpar* ∧ (*cat3* ≡ *const_like* ∨ *cat3* ≡ *case_like*)) {
    big_app1(*pp* + 2); *big_app*('␣'); *big_app1*(*pp* + 3); *reduce*(*pp* + 2, 2, *rpar*, 0, 9);
  }
  else if (*cat1* ≡ *cast* ∧ (*cat2* ≡ *const_like* ∨ *cat2* ≡ *case_like*)) {
    big_app1(*pp* + 1); *big_app*('␣'); *big_app1*(*pp* + 2); *reduce*(*pp* + 1, 2, *cast*, 0, 9);
  }
  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *cast*) *squash*(*pp*, 2, *exp*, −2, 10);
This code is used in section 147.

159. ⟨Cases for *lpar* 159⟩ ≡
  if ((*cat1* ≡ *exp* ∨ *cat1* ≡ *ubinop*) ∧ *cat2* ≡ *rpar*) *squash*(*pp*, 3, *exp*, −2, 11);
  else if (*cat1* ≡ *rpar*) {
     big_app1(*pp*); *app*('\\'); *app*(','); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 12);
  }
  else if ((*cat1* ≡ *decl_head* ∨ *cat1* ≡ *int_like* ∨ *cat1* ≡ *cast*) ∧ *cat2* ≡ *rpar*)
     squash(*pp*, 3, *cast*, −2, 13);
  else if ((*cat1* ≡ *decl_head* ∨ *cat1* ≡ *int_like* ∨ *cat1* ≡ *exp*) ∧ *cat2* ≡ *comma*) {
     big_app3(*pp*); *app*(*opt*); *app*('9'); *reduce*(*pp*, 3, *lpar*, −1, 14);
  }
  else if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *decl*) {
     big_app2(*pp*); *big_app*('␣'); *reduce*(*pp*, 2, *lpar*, −1, 15);
  }
This code is used in section 147.

160. ⟨Cases for *unop* 160⟩ ≡
  if (*cat1* ≡ *exp* ∨ *cat1* ≡ *int_like*) *squash*(*pp*, 2, *exp*, −2, 16);
This code is used in section 147.

161. ⟨Cases for *ubinop* 161⟩ ≡
  if (*cat1* ≡ *cast* ∧ *cat2* ≡ *rpar*) {
     big_app('{'); *big_app1*(*pp*); *big_app*('}'); *big_app1*(*pp* + 1);
     reduce(*pp*, 2, *cast*, −2, 17);
  }
  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *int_like*) {
     big_app('{'); *big_app1*(*pp*); *big_app*('}'); *big_app1*(*pp* + 1);
     reduce(*pp*, 2, *cat1*, −2, 18);
  }
  else if (*cat1* ≡ *binop*) {
     big_app(*math_rel*); *big_app1*(*pp*); *big_app*('{'); *big_app1*(*pp* + 1); *big_app*('}');
     big_app('}'); *reduce*(*pp*, 2, *binop*, −1, 19);
  }
This code is used in section 147.

162. ⟨Cases for *binop* 162⟩ ≡
  if (*cat1* ≡ *binop*) {
     big_app(*math_rel*); *big_app*('{'); *big_app1*(*pp*); *big_app*('}'); *big_app*('{');
     big_app1(*pp* + 1); *big_app*('}'); *big_app*('}'); *reduce*(*pp*, 2, *binop*, −1, 20);
  }
This code is used in section 147.

---

app = macro ( ), §143.
base = 19, §117.
big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
big_app2 = macro ( ), §143.
big_app3 = macro ( ), §143.
binop = 3, §117.
case_like = 53, §23.
cast = 5, §117.
cat1 = macro, §146.
cat2 = macro, §146.
cat3 = macro, §146.
colon = 28, §117.
comma = 10, §117.
const_like = 50, §23.
decl = 20, §117.
decl_head = 9, §117.
exp = 1, §117.
fn_decl = 25, §117.

found: label, §128.
id_flag = 10240, §139.
indent = °212, §122.
indent_param_decl = macro,
  §305.
int_like = 52, §23.
lbrace = 7, §117.
lpar = 11, §117.
make_ministring: **static void**
  ( ), §155.
make_underlined, §151.
math_rel = °206, §122.
ministring_buf: **static char**
  [], §28.
name_dir: **name_info** [],
  COMMON.W §43.
name_pointer = name_info
  *, §13.

new_meaning: **static void** ( ),
  §32.
operator_found = macro, §148.
opt = °214, §122.
pp: **static scrap_pointer**,
  §137.
rbrace = 8, §117.
reduce: **static void** ( ), §207.
rpar = 12, §117.
semi = 27, §117.
squash: **static void** ( ), §208.
stmt = 23, §117.
strcpy, <string.h>.
tag = 29, §117.
tok_loc: **static token_pointer**,
  §150.
ubinop = 4, §117.
unop = 2, §117.

163. ⟨ Cases for *cast* 163 ⟩ ≡
  if (*cat1* ≡ *lpar*) *squash*(*pp*, 2, *lpar*, −1, 21);
  else if (*cat1* ≡ *exp*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 21);
  }
  else if (*cat1* ≡ *semi*) *squash*(*pp*, 1, *exp*, −2, 22);
This code is used in section 147.

164. ⟨ Cases for *sizeof_like* 164 ⟩ ≡
  if (*cat1* ≡ *cast*) *squash*(*pp*, 2, *exp*, −2, 23);
  else if (*cat1* ≡ *exp*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 24);
  }
This code is used in section 147.

165. ⟨ Cases for *int_like* 165 ⟩ ≡
  if (*cat1* ≡ *int_like* ∨ *cat1* ≡ *struct_like*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *cat1*, −2, 25);
  }
  else if (*cat1* ≡ *exp* ∧ (*cat2* ≡ *raw_int* ∨ *cat2* ≡ *struct_like*))
    squash(*pp*, 2, *int_like*, −2, 26);
  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *ubinop* ∨ *cat1* ≡ *colon*) {
    big_app1(*pp*); *big_app*('␣'); *reduce*(*pp*, 1, *decl_head*, −1, 27);
  }
  else if (*cat1* ≡ *semi* ∨ *cat1* ≡ *binop*) *squash*(*pp*, 1, *decl_head*, 0, 28);
This code is used in section 147.

166. ⟨ Cases for *public_like* 166 ⟩ ≡
  if (*cat1* ≡ *colon*) *squash*(*pp*, 2, *tag*, −1, 29);
  else *squash*(*pp*, 1, *int_like*, −2, 30);
This code is used in section 147.

167. ⟨ Cases for *colcol* 167 ⟩ ≡
  if (*cat1* ≡ *exp* ∨ *cat1* ≡ *int_like*) {
    app(*qualifier*); *squash*(*pp*, 2, *cat1*, −2, 31);
  } **else if** (*cat1* ≡ *colcol*) *squash*(*pp*, 2, *colcol*, −1, 32);
This code is used in section 147.

168. ⟨Cases for $decl\_head$ 168⟩ ≡
  if ($cat1 \equiv comma$) {
    $big\_app2(pp)$; $big\_app(\text{'}\sqcup\text{'})$; $reduce(pp, 2, decl\_head, -1, 33)$;
  }
  else if ($cat1 \equiv ubinop$) {
    $big\_app1(pp)$; $big\_app(\text{'}\{\text{'})$; $big\_app1(pp+1)$; $big\_app(\text{'}\}\text{'})$;
    $reduce(pp, 2, decl\_head, -1, 34)$;
  }
  else if ($cat1 \equiv exp \wedge cat2 \neq lpar \wedge cat2 \neq exp \wedge cat2 \neq cast$) {
    $make\_underlined(pp+1)$; $make\_ministring(1)$; $squash(pp, 2, decl\_head, -1, 35)$;
  }
  else if (($cat1 \equiv binop \vee cat1 \equiv colon) \wedge cat2 \equiv exp \wedge (cat3 \equiv comma \vee cat3 \equiv$
      $semi \vee cat3 \equiv rpar$)) $squash(pp, 3, decl\_head, -1, 36)$;
  else if ($cat1 \equiv cast$) $squash(pp, 2, decl\_head, -1, 37)$;
  else if ($cat1 \equiv lbrace \vee cat1 \equiv int\_like \vee cat1 \equiv decl$) {
    $big\_app1(pp)$;
    if ($indent\_param\_decl$) {
      $big\_app(indent)$; $app(indent)$;
    }
    $reduce(pp, 1, fn\_decl, 0, 38)$;
  }
  else if ($cat1 \equiv semi$) $squash(pp, 2, decl, -1, 39)$;
This code is used in section 147.

169. ⟨Cases for $decl$ 169⟩ ≡
  if ($cat1 \equiv decl$) {
    $big\_app1(pp)$; $big\_app(force)$; $big\_app1(pp+1)$; $reduce(pp, 2, decl, -1, 40)$;
  }
  else if ($cat1 \equiv stmt \vee cat1 \equiv function$) {
    $big\_app1(pp)$;
    if ($order\_decl\_stmt$) $big\_app(big\_force)$;
    else $big\_app(force)$;
    $big\_app1(pp+1)$; $reduce(pp, 2, cat1, -1, 41)$;
  }
This code is used in section 147.

$app = $ macro ( ), §143.
big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
$big_app2 = $ macro ( ), §143.
$big\_force = °220$, §122.
$binop = 3$, §117.
$cast = 5$, §117.
$cat1 = $ macro, §146.
$cat2 = $ macro, §146.
$cat3 = $ macro, §146.
$colcol = 18$, §117.
$colon = 28$, §117.
$comma = 10$, §117.
$decl = 20$, §117.
$decl\_head = 9$, §117.

$exp = 1$, §117.
$fn\_decl = 25$, §117.
$force = °217$, §122.
$function = 24$, §117.
$indent = °212$, §122.
$indent_param_decl = $ macro,
  §305.
$int\_like = 52$, §23.
$lbrace = 7$, §117.
$lpar = 11$, §117.
$make\_ministring$: **static void**
  ( ), §155.
$make\_underlined$, §151.
$order_decl_stmt = $ macro, §306.

pp: **static scrap_pointer**,
  §137.
$public\_like = 40$, §23.
$qualifier = °225$, §122.
$raw\_int = 51$, §23.
$reduce$: **static void** ( ), §207.
$rpar = 12$, §117.
$semi = 27$, §117.
$sizeof\_like = 54$, §23.
$squash$: **static void** ( ), §208.
$stmt = 23$, §117.
$struct\_like = 55$, §23.
$tag = 29$, §117.
$ubinop = 4$, §117.

170. ⟨Cases for *base* 170⟩ ≡
  if (*cat1* ≡ *int_like* ∨ *cat1* ≡ *exp*) {
    if (*cat2* ≡ *comma*) {
      big_app1(*pp*); *big_app*('␣'); *big_app2*(*pp* + 1); *app*(*opt*); *app*('9');
      reduce(*pp*, 3, *base*, 0, 42);
    }
    else if (*cat2* ≡ *lbrace*) {
      big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *big_app*('␣'); *big_app1*(*pp* + 2);
      reduce(*pp*, 3, *lbrace*, −2, 43);
    }
  }
This code is used in section 147.

171. ⟨Cases for *struct_like* 171⟩ ≡
  if (*cat1* ≡ *lbrace*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *struct_head*, 0, 44);
  }
  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *int_like*) {
    if (*cat2* ≡ *lbrace* ∨ *cat2* ≡ *semi*) {
      make_underlined(*pp* + 1); *make_reserved*(*pp* + 1); *make_ministring*(1);
      big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1);
      if (*cat2* ≡ *semi*) *reduce*(*pp*, 2, *decl_head*, 0, 45);
      else {
        big_app('␣'); *big_app1*(*pp* + 2); *reduce*(*pp*, 3, *struct_head*, 0, 46);
      }
    }
    else if (*cat2* ≡ *colon*) *squash*(*pp* + 2, 1, *base*, 2, 47);
    else if (*cat2* ≠ *base*) {
      big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *int_like*, −2, 48);
    }
  }
This code is used in section 147.

172. ⟨Cases for *struct_head* 172⟩ ≡
  if ((*cat1* ≡ *decl* ∨ *cat1* ≡ *stmt* ∨ *cat1* ≡ *function*) ∧ *cat2* ≡ *rbrace*) {
    big_app1(*pp*); *big_app*(*indent*); *big_app*(*force*); *big_app1*(*pp* + 1); *big_app*(*outdent*);
    big_app(*force*); *big_app1*(*pp* + 2); *reduce*(*pp*, 3, *int_like*, −2, 49);
  }
  else if (*cat1* ≡ *rbrace*) {
    big_app1(*pp*); *app_str*("\\,"); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *int_like*, −2, 50);
  }
This code is used in section 147.

173. ⟨ Cases for *fn_decl* 173 ⟩ ≡
  if (*cat1* ≡ *decl*) {
    big_app1 (*pp*); *big_app*(*force*); *big_app1* (*pp* + 1); *reduce*(*pp*, 2, *fn_decl*, 0, 51);
  }
  else if (*cat1* ≡ *stmt*) {
    big_app1 (*pp*);
    if (*indent_param_decl*) {
      app(*outdent*); *app*(*outdent*);
    }
    big_app(*force*); *big_app1* (*pp* + 1); *reduce*(*pp*, 2, *function*, −1, 52);
  }

This code is used in section 147.

174. ⟨ Cases for *function* 174 ⟩ ≡
  if (*cat1* ≡ *function* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *stmt*) {
    big_app1 (*pp*); *big_app*(*big_force*); *big_app1* (*pp* + 1); *reduce*(*pp*, 2, *cat1*, −1, 53);
  }

This code is used in section 147.

175. ⟨ Cases for *lbrace* 175 ⟩ ≡
  if (*cat1* ≡ *rbrace*) {
    big_app1 (*pp*); *app*('\\'); *app*(','); *big_app1* (*pp* + 1); *reduce*(*pp*, 2, *stmt*, −1, 54);
  }
  else if ((*cat1* ≡ *stmt* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *function*) ∧ *cat2* ≡ *rbrace*) {
    big_app(*force*); *big_app1* (*pp*); *big_app*(*indent*); *big_app*(*force*); *big_app1* (*pp* + 1);
    big_app(*force*); *big_app*(*backup*); *big_app1* (*pp* + 2); *big_app*(*outdent*); *big_app*(*force*);
    reduce(*pp*, 3, *stmt*, −1, 55);
  }
  else if (*cat1* ≡ *exp*) {
    if (*cat2* ≡ *rbrace*) *squash*(*pp*, 3, *exp*, −2, 56);
    else if (*cat2* ≡ *comma* ∧ *cat3* ≡ *rbrace*) *squash*(*pp*, 4, *exp*, −2, 56);
  }

This code is used in section 147.

176. ⟨ Cases for *if_like* 176 ⟩ ≡
  if (*cat1* ≡ *exp*) {
    big_app1 (*pp*); *big_app*('␣'); *big_app1* (*pp* + 1); *reduce*(*pp*, 2, *if_clause*, 0, 57);
  }

This code is used in section 147.

---

app = macro ( ), §143.
app_str: **static void** ( ), §145.
backup = °*215*, §122.
base = 19, §117.
big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
big_app2 = macro ( ), §143.
big_force = °*220*, §122.
cat1 = macro, §146.
cat2 = macro, §146.
cat3 = macro, §146.
colon = 28, §117.
comma = 10, §117.
decl = 20, §117.

decl_head = 9, §117.
exp = 1, §117.
fn_decl = 25, §117.
force = °*217*, §122.
function = 24, §117.
if_clause = 32, §117.
if_like = 47, §23.
indent = °*212*, §122.
indent_param_decl = macro,
  §305.
int_like = 52, §23.
lbrace = 7, §117.
make_ministring: **static void**
  ( ), §155.

make_reserved, §150.
make_underlined, §151.
opt = °*214*, §122.
outdent = °*213*, §122.
pp: **static scrap_pointer**,
  §137.
rbrace = 8, §117.
reduce: **static void** ( ), §207.
semi = 27, §117.
squash: **static void** ( ), §208.
stmt = 23, §117.
struct_head = 21, §117.
struct_like = 55, §23.

177. ⟨Cases for *else_like* 177⟩ ≡
  if (*cat1* ≡ *colon*) *squash*(*pp* + 1, 1, *base*, 1, 58);
  else if (*cat1* ≡ *lbrace*) *squash*(*pp*, 1, *else_head*, 0, 59);
  else if (*cat1* ≡ *stmt*) {
    big_app(*force*); *big_app1*(*pp*); *big_app*(*indent*); *big_app*(*break_space*);
    big_app1(*pp* + 1); *big_app*(*outdent*); *big_app*(*force*); *reduce*(*pp*, 2, *stmt*, −1, 60);
  }

This code is used in section 147.

178. ⟨Cases for *else_head* 178⟩ ≡
  if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *exp*) {
    big_app(*force*); *big_app1*(*pp*); *big_app*(*break_space*); *app*(*noop*); *big_app*(*cancel*);
    big_app1(*pp* + 1); *big_app*(*force*); *reduce*(*pp*, 2, *stmt*, −1, 61);
  }

This code is used in section 147.

179. ⟨Cases for *if_clause* 179⟩ ≡
  if (*cat1* ≡ *lbrace*) *squash*(*pp*, 1, *if_head*, 0, 62);
  else if (*cat1* ≡ *stmt*) {
    if (*cat2* ≡ *else_like*) {
      big_app(*force*); *big_app1*(*pp*); *big_app*(*indent*); *big_app*(*break_space*);
      big_app1(*pp* + 1); *big_app*(*outdent*); *big_app*(*force*); *big_app1*(*pp* + 2);
      if (*cat3* ≡ *if_like*) {
        big_app('␣'); *big_app1*(*pp* + 3); *reduce*(*pp*, 4, *if_like*, 0, 63);
      } **else** *reduce*(*pp*, 3, *else_like*, 0, 64);
    }
    else *squash*(*pp*, 1, *else_like*, 0, 65);
  }

This code is used in section 147.

180. ⟨Cases for *if_head* 180⟩ ≡
  if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *exp*) {
    if (*cat2* ≡ *else_like*) {
      big_app(*force*); *big_app1*(*pp*); *big_app*(*break_space*); *app*(*noop*); *big_app*(*cancel*);
      big_app1(*pp* + 1); *big_app*(*force*); *big_app1*(*pp* + 2);
      if (*cat3* ≡ *if_like*) {
        big_app('␣'); *big_app1*(*pp* + 3); *reduce*(*pp*, 4, *if_like*, 0, 66);
      } **else** *reduce*(*pp*, 3, *else_like*, 0, 67);
    }
    else *squash*(*pp*, 1, *else_head*, 0, 68);
  }

This code is used in section 147.

181. ⟨Cases for *do_like* 181⟩ ≡
  if (*cat1* ≡ *stmt* ∧ *cat2* ≡ *else_like* ∧ *cat3* ≡ *semi*) {
    big_app1(*pp*); *big_app*(*break_space*); *app*(*noop*); *big_app*(*cancel*); *big_app1*(*pp* + 1);
    big_app(*cancel*); *app*(*noop*); *big_app*(*break_space*); *big_app2*(*pp* + 2);
    reduce(*pp*, 4, *stmt*, −1, 69);
  }

This code is used in section 147.

182.   ⟨ Cases for *case_like* 182 ⟩ ≡
  if (*cat1* ≡ *semi*) *squash*(*pp*, 2, *stmt*, −1, 70);
  else if (*cat1* ≡ *colon*) *squash*(*pp*, 2, *tag*, −1, 71);
  else if (*cat1* ≡ *exp*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 72);
  }

This code is used in section 147.

183.   ⟨ Cases for *catch_like* 183 ⟩ ≡
  if (*cat1* ≡ *cast* ∨ *cat1* ≡ *exp*) {
    big_app2(*pp*); *big_app*(*indent*); *big_app*(*indent*); *reduce*(*pp*, 2, *fn_decl*, 0, 73);
  }

This code is used in section 147.

184.   ⟨ Cases for *tag* 184 ⟩ ≡
  if (*cat1* ≡ *tag*) {
    big_app1(*pp*); *big_app*(*break_space*); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *tag*, −1, 74);
  }
  else if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *function*) {
    big_app(*force*); *big_app*(*backup*); *big_app1*(*pp*); *big_app*(*break_space*);
    big_app1(*pp* + 1); *reduce*(*pp*, 2, *cat1*, −1, 75);
  }

This code is used in section 147.

185.   The user can decide at run-time whether short statements should be grouped together on the same line.

#define *force_lines*   *flags*['f']   ▷ should each statement be on its own line? ◁

⟨ Set initial values 29 ⟩ +≡
  force_lines ← *true*;

---

app = macro ( ), §143.
backup = °215, §122.
base = 19, §117.
big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
big_app2 = macro ( ), §143.
break_space = °216, §122.
cancel = °211, §122.
case_like = 53, §23.
cast = 5, §117.
cat1 = macro, §146.
cat2 = macro, §146.
cat3 = macro, §146.
catch_like = 43, §23.

colon = 28, §117.
decl = 20, §117.
do_like = 46, §23.
else_head = 31, §117.
else_like = 26, §23.
exp = 1, §117.
flags: **boolean** [ ],
    COMMON.W §73.
fn_decl = 25, §117.
force = °217, §122.
function = 24, §117.
if_clause = 32, §117.
if_head = 30, §117.

if_like = 47, §23.
indent = °212, §122.
lbrace = 7, §117.
noop = °177, §48.
outdent = °213, §122.
pp: **static scrap_pointer**,
    §137.
reduce: **static void** ( ), §207.
semi = 27, §117.
squash: **static void** ( ), §208.
stmt = 23, §117.
tag = 29, §117.
true, <stdbool.h>.

186. ⟨Cases for *stmt* 186⟩ ≡
  if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *function*) {
    big_app1 (*pp*);
    if (*cat1* ≡ *function*) *big_app* (*big_force*);
    else if (*cat1* ≡ *decl*) *big_app* (*big_force*);
    else if (*force_lines*) *big_app* (*force*);
    else *big_app* (*break_space*);
    big_app1 (*pp* + 1); *reduce* (*pp*, 2, *cat1*, −1, 76);
  }

This code is used in section 147.

187. ⟨Cases for *semi* 187⟩ ≡
  big_app ('␣'); *big_app1* (*pp*); *reduce* (*pp*, 1, *stmt*, −1, 77);

This code is used in section 147.

188. ⟨Cases for *lproc* 188⟩ ≡
  if (*cat1* ≡ *define_like*) {   ▷ #define is analogous to **extern** ◁
    make_underlined (*pp* + 2);
    if (*tok_loc* > *operator_found*) {   ▷ no time to work out this case; I'll handle defines
      by brute force in the aux file, since they usually don't go in mini-index ◁
    }
  }
  if (*cat1* ≡ *else_like* ∨ *cat1* ≡ *if_like* ∨ *cat1* ≡ *define_like*) *squash* (*pp*, 2, *lproc*, 0, 78);
  else if (*cat1* ≡ *rproc*) {
    app (*inserted*); *big_app2* (*pp*); *reduce* (*pp*, 2, *insert*, −1, 79);
  }
  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *function*) {
    if (*cat2* ≡ *rproc*) {
      app (*inserted*); *big_app1* (*pp*); *big_app* ('␣'); *big_app2* (*pp* + 1);
      reduce (*pp*, 3, *insert*, −1, 80);
    }
    else if (*cat2* ≡ *exp* ∧ *cat3* ≡ *rproc* ∧ *cat1* ≡ *exp*) {
      app (*inserted*); *big_app1* (*pp*); *big_app* ('␣'); *big_app1* (*pp* + 1); *app_str* ("␣\\5");
      big_app2 (*pp* + 2); *reduce* (*pp*, 4, *insert*, −1, 80);
    }
  }

This code is used in section 147.

189. ⟨Cases for *section_scrap* 189⟩ ≡
  if (*cat1* ≡ *semi*) {
    big_app2 (*pp*); *big_app* (*force*); *reduce* (*pp*, 2, *stmt*, −2, 81);
  }
  else *squash* (*pp*, 1, *exp*, −2, 82);

This code is used in section 147.

190. ⟨Cases for *insert* 190⟩ ≡
  if (*cat1*) *squash* (*pp*, 2, *cat1*, 0, 83);

This code is used in section 147.

191. ⟨Cases for *prelangle* 191⟩ ≡
  init_mathness ← *cur_mathness* ← *yes_math*; *app*('<'); *reduce*(*pp*, 1, *binop*, −2, 84);
This code is used in section 147.

192. ⟨Cases for *prerangle* 192⟩ ≡
  init_mathness ← *cur_mathness* ← *yes_math*; *app*('>'); *reduce*(*pp*, 1, *binop*, −2, 85);
This code is used in section 147.

193. ⟨Cases for *langle* 193⟩ ≡
  if (*cat1* ≡ *prerangle*) {
    big_app1(*pp*); *app*('\\'); *app*(','); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *cast*, −1, 86);
  }
  else if (*cat1* ≡ *decl_head* ∨ *cat1* ≡ *int_like* ∨ *cat1* ≡ *exp*) {
    if (*cat2* ≡ *prerangle*) *squash*(*pp*, 3, *cast*, −1, 87);
    else if (*cat2* ≡ *comma*) {
      big_app3(*pp*); *app*(*opt*); *app*('9'); *reduce*(*pp*, 3, *langle*, 0, 88);
    }
  }
This code is used in section 147.

194. ⟨Cases for *template_like* 194⟩ ≡
  if (*cat1* ≡ *exp* ∧ *cat2* ≡ *prelangle*) *squash*(*pp* + 2, 1, *langle*, 2, 89);
  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *raw_int*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *cat1*, −2, 90);
  } **else** *squash*(*pp*, 1, *raw_int*, 0, 91);
This code is used in section 147.

195. ⟨Cases for *new_like* 195⟩ ≡
  if (*cat1* ≡ *lpar* ∧ *cat2* ≡ *exp* ∧ *cat3* ≡ *rpar*) *squash*(*pp*, 4, *new_like*, 0, 92);
  else if (*cat1* ≡ *cast*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 93);
  }
  else if (*cat1* ≠ *lpar*) *squash*(*pp*, 1, *new_exp*, 0, 94);
This code is used in section 147.

---

app = macro ( ), §143.
app_str: **static void** ( ), §145.
big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
big_app2 = macro ( ), §143.
big_app3 = macro ( ), §143.
big_force = °*220*, §122.
binop = 3, §117.
break_space = °*216*, §122.
cast = 5, §117.
cat1 = macro, §146.
cat2 = macro, §146.
cat3 = macro, §146.
comma = 10, §117.
cur_mathness: **static int**, §143.
decl = 20, §117.
decl_head = 9, §117.

define_like = 57, §23.
else_like = 26, §23.
exp = 1, §117.
force = °*217*, §122.
force_lines = macro, §185.
function = 24, §117.
if_like = 47, §23.
init_mathness: **static int**, §143.
insert = 37, §117.
inserted = °*224*, §122.
int_like = 52, §23.
langle = 15, §117.
lpar = 11, §117.
lproc = 35, §117.
make_underlined, §151.
new_exp = 60, §117.
new_like = 42, §23.

operator_found = macro, §148.
opt = °*214*, §122.
pp: **static scrap_pointer**, §137.
prelangle = 13, §117.
prerangle = 14, §117.
raw_int = 51, §23.
reduce: **static void** ( ), §207.
rpar = 12, §117.
rproc = 36, §117.
section_scrap = 38, §117.
semi = 27, §117.
squash: **static void** ( ), §208.
stmt = 23, §117.
template_like = 58, §23.
tok_loc: **static token_pointer**, §150.
yes_math = 1, §143.

196. ⟨ Cases for *new_exp* 196 ⟩ ≡
  if (*cat1* ≡ *int_like* ∨ *cat1* ≡ *const_like*) {
    big_app1 (*pp*); *big_app*('␣'); *big_app1* (*pp* + 1); *reduce* (*pp*, 2, *new_exp*, 0, 95);
  }
  else if (*cat1* ≡ *struct_like* ∧ (*cat2* ≡ *exp* ∨ *cat2* ≡ *int_like*)) {
    big_app1 (*pp*); *big_app*('␣'); *big_app1* (*pp* + 1); *big_app*('␣'); *big_app1* (*pp* + 2);
    reduce (*pp*, 3, *new_exp*, 0, 96);
  }
  else if (*cat1* ≡ *raw_ubin*) {
    big_app1 (*pp*); *big_app*('{'); *big_app1* (*pp* + 1); *big_app*('}');
    reduce (*pp*, 2, *new_exp*, 0, 97);
  }
  else if (*cat1* ≡ *lpar*) *squash* (*pp*, 1, *exp*, −2, 98);
  else if (*cat1* ≡ *exp*) {
    big_app1 (*pp*); *big_app*('␣'); *reduce* (*pp*, 1, *exp*, −2, 98);
  }
  else if (*cat1* ≠ *raw_int* ∧ *cat1* ≠ *struct_like* ∧ *cat1* ≠ *colcol*) *squash* (*pp*, 1, *exp*, −2, 99);
This code is used in section 147.

197. ⟨ Cases for *ftemplate* 197 ⟩ ≡
  if (*cat1* ≡ *prelangle*) *squash* (*pp* + 1, 1, *langle*, 1, 121);
  else *squash* (*pp*, 1, *exp*, −2, 122);
This code is used in section 147.

198. ⟨ Cases for *for_like* 198 ⟩ ≡
  if (*cat1* ≡ *exp*) {
    big_app1 (*pp*); *big_app*('␣'); *big_app1* (*pp* + 1); *reduce* (*pp*, 2, *else_like*, −2, 123);
  }
This code is used in section 147.

199. ⟨ Cases for *raw_ubin* 199 ⟩ ≡
  if (*cat1* ≡ *const_like*) {
    big_app2 (*pp*); *app_str* ("\\␣"); *reduce* (*pp*, 2, *raw_ubin*, 0, 103);
  }
  else *squash* (*pp*, 1, *ubinop*, −2, 104);
This code is used in section 147.

200. ⟨ Cases for *const_like* 200 ⟩ ≡
  squash (*pp*, 1, *int_like*, −2, 105);
This code is used in section 147.

201. ⟨ Cases for *raw_int* 201 ⟩ ≡
  if (*cat1* ≡ *prelangle*) *squash* (*pp* + 1, 1, *langle*, 1, 106);
  else if (*cat1* ≡ *colcol*) *squash* (*pp*, 2, *colcol*, −1, 107);
  else if (*cat1* ≡ *cast*) *squash* (*pp*, 2, *raw_int*, 0, 108);
  else if (*cat1* ≡ *lpar*) *squash* (*pp*, 1, *exp*, −2, 109);
  else if (*cat1* ≠ *langle*) *squash* (*pp*, 1, *int_like*, −3, 110);
This code is used in section 147.

202.   ⟨ Cases for *operator_like* 202 ⟩ ≡
  if (*cat1* ≡ *binop* ∨ *cat1* ≡ *unop* ∨ *cat1* ≡ *ubinop*) {
    if (*cat2* ≡ *binop*) **break**;
    big_app1(*pp*); *big_app*('{'); *big_app1*(*pp* + 1); *big_app*('}');
    reduce(*pp*, 2, *exp*, −2, 111);
  }
  else if (*cat1* ≡ *new_like* ∨ *cat1* ≡ *delete_like*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 112);
  }
  else if (*cat1* ≡ *comma*) *squash*(*pp*, 2, *exp*, −2, 113);
  else if (*cat1* ≠ *raw_ubin*) *squash*(*pp*, 1, *new_exp*, 0, 114);
This code is used in section 147.

203.   Here CTWILL deviates from the normal productions introduced in version 3.6, because those productions bypass *decl_head* (thereby confusing *make_ministring*, which depends on the *decl_head* productions to deduce the type). We revert to an older syntax that was less friendly to C++ but good enough for me.

⟨ Cases for *typedef_like* 203 ⟩ ≡
  if (*cat1* ≡ *decl_head*) {
    if ((*cat2* ≡ *exp* ∧ *cat3* ≠ *lpar* ∧ *cat3* ≠ *exp*) ∨ *cat2* ≡ *int_like*) {
      make_underlined(*pp* + 2); *make_reserved*(*pp* + 2); *make_ministring*(2);
      big_app2(*pp* + 1); *reduce*(*pp* + 1, 2, *decl_head*, 0, 200);
    }
    else if (*cat2* ≡ *semi*) {
      big_app1(*pp*); *big_app*('␣'); *big_app2*(*pp* + 1); *reduce*(*pp*, 3, *decl*, −1, 201);
    }
  }
  else if (*cat1* ≡ *int_like* ∧ *cat2* ≡ *raw_int* ∧ (*cat3* ≡ *semi* ∨ *cat3* ≡ *comma*))
    squash(*pp* + 2, 1, *exp*, 1, 202);
This code is used in section 147.

---

app_str: **static void** ( ), §145.
big_app: **static void** ( ), §145.
big_app1: **static void** ( ), §145.
big_app2 = macro ( ), §143.
binop = 3, §117.
cast = 5, §117.
cat1 = macro, §146.
cat2 = macro, §146.
cat3 = macro, §146.
colcol = 18, §117.
comma = 10, §117.
const_like = 50, §23.
decl = 20, §117.
decl_head = 9, §117.

delete_like = 48, §23.
else_like = 26, §23.
exp = 1, §117.
for_like = 45, §23.
ftemplate = 59, §117.
int_like = 52, §23.
langle = 15, §117.
lpar = 11, §117.
make_ministring: **static void** ( ), §155.
make_reserved, §150.
make_underlined, §151.
new_exp = 60, §117.
new_like = 42, §23.

operator_like = 41, §23.
pp: **static scrap_pointer**, §137.
prelangle = 13, §117.
raw_int = 51, §23.
raw_ubin = 49, §23.
reduce: **static void** ( ), §207.
semi = 27, §117.
squash: **static void** ( ), §208.
struct_like = 55, §23.
typedef_like = 56, §23.
ubinop = 4, §117.
unop = 2, §117.

204.   ⟨ Cases for *delete_like* 204 ⟩ ≡
  if (*cat1* ≡ *lpar* ∧ *cat2* ≡ *rpar*) {
    big_app2(*pp*); *app*('\\'); *app*(','); *big_app1*(*pp* + 2);
    reduce(*pp*, 3, *delete_like*, 0, 121);
  }
  else if (*cat1* ≡ *exp*) {
    big_app1(*pp*); *big_app*('␣'); *big_app1*(*pp* + 1); *reduce*(*pp*, 2, *exp*, −2, 122);
  }
This code is used in section 147.

205.   ⟨ Cases for *question* 205 ⟩ ≡
  if (*cat1* ≡ *exp* ∧ (*cat2* ≡ *colon* ∨ *cat2* ≡ *base*)) {
    (*pp* + 2)→*mathness* ← 5 * *yes_math*;   ▷ this colon should be in math mode ◁
    squash(*pp*, 3, *binop*, −2, 123);
  }
This code is used in section 147.

206.   Now here's the *reduce* procedure used in our code for productions.

The '*freeze_text*' macro is used to give official status to a token list. Before saying *freeze_text*, items are appended to the current token list, and we know that the eventual number of this token list will be the current value of *text_ptr*. But no list of that number really exists as yet, because no ending point for the current list has been stored in the *tok_start* array. After saying *freeze_text*, the old current token list becomes legitimate, and its number is the current value of *text_ptr* − 1 since *text_ptr* has been increased. The new current token list is empty and ready to be appended to. Note that *freeze_text* does not check to see that *text_ptr* hasn't gotten too large, since it is assumed that this test was done beforehand.

#define *freeze_text* *(++*text_ptr*) ← *tok_ptr*
⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *reduce*(**scrap_pointer**, **short**, **eight_bits**, **short**, **short**);
  static void *squash*(**scrap_pointer**, **short**, **eight_bits**, **short**, **short**);

207.   **static void** *reduce*(**scrap_pointer** *j*, **short** *k*, **eight_bits** *c*, **short** *d*, **short** *n*)
  {
    scrap_pointer *i*, *i1*;   ▷ pointers into scrap memory ◁

    j→*cat* ← *c*; *j*→*trans* ← *text_ptr*; *j*→*mathness* ← 4 * *cur_mathness* + *init_mathness*;
    freeze_text;
    if (*k* > 1) {
      for (*i* ← *j* + *k*, *i1* ← *j* + 1; *i* ≤ *lo_ptr*; *i*++, *i1*++) {
        i1→*cat* ← *i*→*cat*; *i1*→*trans* ← *i*→*trans*; *i1*→*mathness* ← *i*→*mathness*;
      }
      lo_ptr ← *lo_ptr* − *k* + 1;
    }
    pp ← (*pp* + *d* < *scrap_base* ? *scrap_base* : *pp* + *d*);
    ⟨ Print a snapshot of the scrap list if debugging 212 ⟩
    pp −−;   ▷ we next say *pp* ++ ◁
  }

208.   Here's the *squash* procedure, which takes advantage of the simplification that occurs when $k \equiv 1$.

static void $squash(\textbf{scrap\_pointer}\ j, \textbf{short}\ k, \textbf{eight\_bits}\ c, \textbf{short}\ d, \textbf{short}\ n)$

$\{$

    scrap_pointer i;   ▷ pointers into scrap memory ◁

    if $(k \equiv 1)$ $\{$

        $j{\to}cat \leftarrow c$; $pp \leftarrow (pp + d < scrap\_base\ ?\ scrap\_base : pp + d)$;

        ⟨ Print a snapshot of the scrap list if debugging 212 ⟩

        $pp\,\text{−−}$;   ▷ we next say $pp\,\text{++}$ ◁

        return;

    $\}$

    for $(i \leftarrow j;\ i < j + k;\ i{+}{+})$ $big\_app1(i)$;

    $reduce(j, k, c, d, n)$;

$\}$

---

209. And here now is the code that applies productions as long as possible. Before applying the production mechanism, we must make sure it has good input (at least four scraps, the length of the lhs of the longest rules), and that there is enough room in the memory arrays to hold the appended tokens and texts. Here we use a very conservative test; it's more important to make sure the program will still work if we change the production rules (within reason) than to squeeze the last bit of space from the memory arrays.

#define $safe\_tok\_incr$ 20
#define $safe\_text\_incr$ 10
#define $safe\_scrap\_incr$ 10

⟨ Reduce the scraps using the productions until no more rules apply 209 ⟩ ≡
  while ($true$) {
    ⟨ Make sure the entries pp through $pp + 3$ of cat are defined 210 ⟩
    if ($tok\_ptr + safe\_tok\_incr > tok\_mem\_end$) {
      if ($tok\_ptr > max\_tok\_ptr$) $max\_tok\_ptr \leftarrow tok\_ptr$;
      $overflow(\_("token"))$;
    }
    if ($text\_ptr + safe\_text\_incr > tok\_start\_end$) {
      if ($text\_ptr > max\_text\_ptr$) $max\_text\_ptr \leftarrow text\_ptr$;
      $overflow(\_("text"))$;
    }
    if ($pp > lo\_ptr$) **break**;
    $init\_mathness \leftarrow cur\_mathness \leftarrow maybe\_math$;
    ⟨ Match a production at pp, or increase pp if there is no match 146 ⟩
  }

This code is used in section 213.

210. If we get to the end of the scrap list, category codes equal to zero are stored, since zero does not match anything in a production.

⟨ Make sure the entries pp through $pp + 3$ of cat are defined 210 ⟩ ≡
  if ($lo\_ptr < pp + 3$) {
    while ($hi\_ptr \leq scrap\_ptr \wedge lo\_ptr \neq pp + 3$) {
      $(\mathord{+}\mathord{+}lo\_ptr){\rightarrow}cat \leftarrow hi\_ptr{\rightarrow}cat$; $lo\_ptr{\rightarrow}mathness \leftarrow (hi\_ptr){\rightarrow}mathness$;
      $lo\_ptr{\rightarrow}trans \leftarrow (hi\_ptr\mathord{+}\mathord{+}){\rightarrow}trans$;
    }
    for ($i \leftarrow lo\_ptr + 1$; $i \leq pp + 3$; $i\mathord{+}\mathord{+}$) $i{\rightarrow}cat \leftarrow 0$;
  }

This code is used in section 209.

211. If CWEAVE is being run in debugging mode, the production numbers and current stack categories will be printed out when $tracing$ is set to 2; a sequence of two or more irreducible scraps will be printed out when $tracing$ is set to 1.

⟨ Private variables 26 ⟩ +≡
  static int $tracing$;   ▷ can be used to show parsing details ◁

212. ⟨Print a snapshot of the scrap list if debugging 212⟩ ≡
```
{
    scrap_pointer k_l;      ▷ pointer into scrap_info ◁
    if (tracing ≡ 2) {
        printf("\n%d:", n);
        for (k_l ← scrap_base; k_l ≤ lo_ptr; k_l++) {
            if (k_l ≡ pp) putxchar('*');
            else putxchar('␣');
            if (k_l→mathness % 4 ≡ yes_math) putchar('+');
            else if (k_l→mathness % 4 ≡ no_math) putchar('-');
            print_cat(k_l→cat);
            if (k_l→mathness/4 ≡ yes_math) putchar('+');
            else if (k_l→mathness/4 ≡ no_math) putchar('-');
        }
        if (hi_ptr ≤ scrap_ptr) fputs("...", stdout);      ▷ indicate that more is coming ◁
    }
}
```
This code is used in sections 207 and 208.

213. The *translate* function assumes that scraps have been stored in positions *scrap_base* through *scrap_ptr* of *cat* and *trans*. It applies productions as much as possible. The result is a token list containing the translation of the given sequence of scraps.

After calling *translate*, we will have $text\_ptr + 3 \leq max\_texts$ and $tok\_ptr + 6 \leq max\_toks$, so it will be possible to create up to three token lists with up to six tokens without checking for overflow. Before calling *translate*, we should have $text\_ptr < max\_texts$ and $scrap\_ptr < max\_scraps$, since *translate* might add a new text and a new scrap before it checks for overflow.

```
static text_pointer translate(void)      ▷ converts a sequence of scraps ◁
{
    scrap_pointer i,      ▷ index into cat ◁
    j;      ▷ runs through final scraps ◁
    pp ← scrap_base; lo_ptr ← pp − 1; hi_ptr ← pp;
    ⟨If tracing, print an indication of where we are 217⟩
    ⟨Reduce the scraps using the productions until no more rules apply 209⟩
    ⟨Combine the irreducible scraps that remain 215⟩
}
```

_ = macro ( ), §7.
cat: **eight_bits**, §136.
cur_mathness: **static int**,
  §143.
fputs, <stdio.h>.
hi_ptr: **static scrap_pointer**,
  §137.
i: **scrap_pointer**, §208.
init_mathness: **static int**,
  §143.
lo_ptr: **static scrap_pointer**,
  §137.
mathness: **eight_bits**, §136.
max_scraps = 10000, §22.
max_text_ptr: **static**
  text_pointer, §40.
max_texts = 10239, §22.
max_tok_ptr: **static**

token_pointer, §40.
max_toks = 1000000, §20.
maybe_math = 0, §143.
n: **short**, §208.
no_math = 2, §143.
overflow: **void** ( ),
  COMMON.W §71.
pp: **static scrap_pointer**,
  §137.
print_cat: **static void** ( ), §120.
printf, <stdio.h>.
putchar, <stdio.h>.
putxchar = macro, §18.
scrap_base: **static**
  scrap_pointer, §137.
scrap_info: **static scrap** [],
  §137.
scrap_pointer = **scrap** *,

§136.
scrap_ptr: **static**
  scrap_pointer, §137.
stdout, <stdio.h>.
text_pointer = **token_pointer**
  *, §39.
text_ptr: **static text_pointer**,
  §40.
tok_mem_end: **static**
  token_pointer, §40.
tok_ptr: **static token_pointer**,
  §40.
tok_start_end: **static**
  text_pointer, §40.
trans = macro, §137.
true, <stdbool.h>.
yes_math = 1, §143.

214. ⟨Predeclaration of procedures 11⟩ +≡ **static text_pointer** *translate*(**void**);

215. If the initial sequence of scraps does not reduce to a single scrap, we concatenate the translations of all remaining scraps, separated by blank spaces, with dollar signs surrounding the translations of scraps where appropriate.

⟨Combine the irreducible scraps that remain 215⟩ ≡
```
  {
    ⟨If semi-tracing, show the irreducible scraps 216⟩
    for (j ← scrap_base; j ≤ lo_ptr; j++) {
      if (j ≠ scrap_base) app('␣');
      if (j→mathness % 4 ≡ yes_math) app('$');
      app1(j);
      if (j→mathness/4 ≡ yes_math) app('$');
      if (tok_ptr + 6 > tok_mem_end) overflow(_("token"));
    }
    freeze_text; return text_ptr − 1;
  }
```
This code is used in section 213.

216. ⟨If semi-tracing, show the irreducible scraps 216⟩ ≡
```
  if (lo_ptr > scrap_base ∧ tracing ≡ 1) {
    printf(_("\nIrreducible␣scrap␣sequence␣in␣section␣%d:"), section_count);
    mark_harmless;
    for (j ← scrap_base; j ≤ lo_ptr; j++) {
      printf("␣"); print_cat(j→cat);
    }
  }
```
This code is used in section 215.

217. ⟨If tracing, print an indication of where we are 217⟩ ≡
```
  if (tracing ≡ 2) {
    printf(_("\nTracing␣after␣l.␣%d:\n"), cur_line); mark_harmless;
    if (loc > buffer + 50) {
      printf("..."); term_write(loc − 51, 51);
    }
    else term_write(buffer, loc − buffer);
  }
```
This code is used in section 213.

218.   Initializing the scraps.   If we are going to use the powerful production
mechanism just developed, we must get the scraps set up in the first place, given a
C text. A table of the initial scraps corresponding to C tokens appeared above in
the section on parsing; our goal now is to implement that table. We shall do this
by implementing a subroutine called *C_parse* that is analogous to the *C_xref* routine
used during phase one.

   Like *C_xref*, the *C_parse* procedure starts with the current value of *next_control*
and it uses the operation *next_control* ← *get_next*( ) repeatedly to read C text until
encountering the next '|' or '/*', or until *next_control* ≥ *format_code*. The scraps
corresponding to what it reads are appended into the *cat* and *trans* arrays, and
scrap_ptr is advanced.

```
static void C_parse(      ▷ creates scraps from C tokens ◁
    eight_bits spec_ctrl)
{
  int count;     ▷ characters remaining before string break ◁
  while (next_control < format_code ∨ next_control ≡ spec_ctrl) {
    ⟨ Append the scrap appropriate to next_control 221 ⟩
    next_control ← get_next( );
    if (next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡
        begin_short_comment) return;
  }
}
```

219.   ⟨ Predeclaration of procedures 11 ⟩ +≡   **static void** *C_parse*(**eight_bits**);

220.   The following macro is used to append a scrap whose tokens have just been
appended:

```
#define  app_scrap(c, b)
        {
          (++scrap_ptr)→cat ← (c);  scrap_ptr→trans ← text_ptr;
          scrap_ptr→mathness ← 5 ∗ (b);     ▷ no no, yes yes, or maybe maybe ◁
          freeze_text;
        }
```

---

221. ⟨Append the scrap appropriate to *next_control* 221⟩ ≡
  ⟨Make sure that there is room for the new scraps, tokens, and texts 222⟩
  switch (*next_control*) {
  case *section_name*: *app*(*section_flag* + (**int**)(*cur_section* − *name_dir*));
    app_scrap(*section_scrap*, *maybe_math*); *app_scrap*(*exp*, *yes_math*); **break**;
  case *string*: **case** *constant*: **case** *verbatim*: ⟨Append a string or constant 224⟩ **break**;
  case *identifier*: *app_cur_id*(*true*); **break**;
  case *TEX_string*: ⟨Append a TEX string, without forming a scrap 225⟩ **break**;
  case ʼ/ʼ: **case** ʼ.ʼ: *app*(*next_control*); *app_scrap*(*binop*, *yes_math*); **break**;
  case ʼ<ʼ: *app_str*("\\langle"); *app_scrap*(*prelangle*, *yes_math*); **break**;
  case ʼ>ʼ: *app_str*("\\rangle"); *app_scrap*(*prerangle*, *yes_math*); **break**;
  case ʼ=ʼ: *app_str*("\\K"); *app_scrap*(*binop*, *yes_math*); **break**;
  case ʼ|ʼ: *app_str*("\\OR"); *app_scrap*(*binop*, *yes_math*); **break**;
  case ʼ^ʼ: *app_str*("\\XOR"); *app_scrap*(*binop*, *yes_math*); **break**;
  case ʼ%ʼ: *app_str*("\\MOD"); *app_scrap*(*binop*, *yes_math*); **break**;
  case ʼ!ʼ: *app_str*("\\R"); *app_scrap*(*unop*, *yes_math*); **break**;
  case ʼ~ʼ: *app_str*("\\CM"); *app_scrap*(*unop*, *yes_math*); **break**;
  case ʼ+ʼ: **case** ʼ-ʼ: *app*(*next_control*); *app_scrap*(*ubinop*, *yes_math*); **break**;
  case ʼ*ʼ: *app*(*next_control*); *app_scrap*(*raw_ubin*, *yes_math*); **break**;
  case ʼ&ʼ: *app_str*("\\AND"); *app_scrap*(*raw_ubin*, *yes_math*); **break**;
  case ʼ?ʼ: *app_str*("\\?"); *app_scrap*(*question*, *yes_math*); **break**;
  case ʼ#ʼ: *app_str*("\\#"); *app_scrap*(*ubinop*, *yes_math*); **break**;
  case *ignore*: **case** *xref_roman*: **case** *xref_wildcard*: **case** *meaning*: **case** *suppress*:
    case *xref_typewriter*: **case** *noop*: **break**;
  case ʼ(ʼ: **case** ʼ[ʼ: *app*(*next_control*); *app_scrap*(*lpar*, *maybe_math*); **break**;
  case ʼ)ʼ: **case** ʼ]ʼ: *app*(*next_control*); *app_scrap*(*rpar*, *maybe_math*); **break**;
  case ʼ{ʼ: *app_str*("\\{"); *app_scrap*(*lbrace*, *yes_math*); **break**;
  case ʼ}ʼ: *app_str*("\\}"); *app_scrap*(*rbrace*, *yes_math*); **break**;
  case ʼ,ʼ: *app*(ʼ,ʼ); *app_scrap*(*comma*, *yes_math*); **break**;
  case ʼ;ʼ: *app*(ʼ;ʼ); *app_scrap*(*semi*, *maybe_math*); **break**;
  case ʼ:ʼ: *app*(ʼ:ʼ); *app_scrap*(*colon*, *no_math*); **break**;
  ⟨Cases involving nonstandard characters 223⟩
  case *thin_space*: *app_str*("\\,"); *app_scrap*(*insert*, *maybe_math*); **break**;
  case *math_break*: *app*(*opt*); *app_str*("0"); *app_scrap*(*insert*, *maybe_math*); **break**;
  case *line_break*: *app*(*force*); *app_scrap*(*insert*, *no_math*); **break**;
  case *left_preproc*: *app*(*force*); *app*(*preproc_line*); *app_str*("\\#");
    app_scrap(*lproc*, *no_math*); **break**;
  case *right_preproc*: *app*(*force*); *app_scrap*(*rproc*, *no_math*); **break**;
  case *big_line_break*: *app*(*big_force*); *app_scrap*(*insert*, *no_math*); **break**;
  case *no_line_break*: *app*(*big_cancel*); *app*(*noop*); *app*(*break_space*); *app*(*noop*);
    app(*big_cancel*); *app_scrap*(*insert*, *no_math*); **break**;
  case *pseudo_semi*: *app_scrap*(*semi*, *maybe_math*); **break**;
  case *macro_arg_open*: *app_scrap*(*begin_arg*, *maybe_math*); **break**;
  case *macro_arg_close*: *app_scrap*(*end_arg*, *maybe_math*); **break**;
  case *join*: *app_str*("\\J"); *app_scrap*(*insert*, *no_math*); **break**;
  case *output_defs_code*: *app*(*force*); *app_str*("\\ATH"); *app*(*force*);
    app_scrap(*insert*, *no_math*); **break**;
  default: *app*(*inserted*); *app*(*next_control*); *app_scrap*(*insert*, *maybe_math*); **break**;

    }
This code is used in section 218.

222.   ⟨Make sure that there is room for the new scraps, tokens, and texts 222⟩ ≡
    if ($scrap\_ptr + safe\_scrap\_incr > scrap\_info\_end \lor tok\_ptr + safe\_tok\_incr > tok\_mem\_end$
            $\lor\ text\_ptr + safe\_text\_incr > tok\_start\_end$) {
        if ($scrap\_ptr > max\_scr\_ptr$) $max\_scr\_ptr \leftarrow scrap\_ptr$;
        if ($tok\_ptr > max\_tok\_ptr$) $max\_tok\_ptr \leftarrow tok\_ptr$;
        if ($text\_ptr > max\_text\_ptr$) $max\_text\_ptr \leftarrow text\_ptr$;
        $overflow(\_("scrap/token/text"))$;
    }
This code is used in sections 221 and 230.

---

223. Some nonstandard characters may have entered CWEAVE by means of standard ones. They are converted to TEX control sequences so that it is possible to keep CWEAVE from outputting unusual **char** codes.

⟨ Cases involving nonstandard characters 223 ⟩ ≡
case *non_eq*: *app_str*("\\I"); *app_scrap*(*binop*, *yes_math*); **break**;
case *lt_eq*: *app_str*("\\Z"); *app_scrap*(*binop*, *yes_math*); **break**;
case *gt_eq*: *app_str*("\\G"); *app_scrap*(*binop*, *yes_math*); **break**;
case *eq_eq*: *app_str*("\\E"); *app_scrap*(*binop*, *yes_math*); **break**;
case *and_and*: *app_str*("\\W"); *app_scrap*(*binop*, *yes_math*); **break**;
case *or_or*: *app_str*("\\V"); *app_scrap*(*binop*, *yes_math*); **break**;
case *plus_plus*: *app_str*("\\PP"); *app_scrap*(*unop*, *yes_math*); **break**;
case *minus_minus*: *app_str*("\\MM"); *app_scrap*(*unop*, *yes_math*); **break**;
case *minus_gt*: *app_str*("\\MG"); *app_scrap*(*binop*, *yes_math*); **break**;
case *gt_gt*: *app_str*("\\GG"); *app_scrap*(*binop*, *yes_math*); **break**;
case *lt_lt*: *app_str*("\\LL"); *app_scrap*(*binop*, *yes_math*); **break**;
case *dot_dot_dot*: *app_str*("\\,\\ldots\\,"); *app_scrap*(*raw_int*, *yes_math*); **break**;
case *colon_colon*: *app_str*("\\DC"); *app_scrap*(*colcol*, *maybe_math*); **break**;
case *period_ast*: *app_str*("\\PA"); *app_scrap*(*binop*, *yes_math*); **break**;
case *minus_gt_ast*: *app_str*("\\MGA"); *app_scrap*(*binop*, *yes_math*); **break**;
This code is used in section 221.

224. The following code must use *app_tok* instead of *app* in order to protect against overflow. Note that $tok\_ptr + 1 \leq max\_toks$ after *app_tok* has been used, so another *app* is legitimate before testing again.

Many of the special characters in a string must be prefixed by '\' so that TEX will print them properly.

⟨ Append a string or constant 224 ⟩ ≡
  count ← −1;
  if (*next_control* ≡ *constant*) *app_str*("\\T{");
  else if (*next_control* ≡ *string*) {
    count ← 20; *app_str*("\\.{");
  }
  else *app_str*("\\vb{");
  while (*id_first* < *id_loc*) {
    if (*count* ≡ 0) {   ▷ insert a discretionary break in a long string ◁
      app_str("}\\)\\.{"); *count* ← 20;
    }
    if ((**eight_bits**)(∗*id_first*) > °177) {
      app_tok(*quoted_char*); *app_tok*((**eight_bits**)(∗*id_first*++));
    }
    else {
      switch (∗*id_first*) {
      case '␣': **case** '\\': **case** '#': **case** '%': **case** '$': **case** '^': **case** '{':
        case '}': **case** '~': **case** '&': **case** '_': *app*('\\'); **break**;
      case '@':
        if (∗(*id_first* + 1) ≡ '@') *id_first*++;
        else *err_print*(_("! ␣Double␣@␣should␣be␣used␣in␣strings"));
      }

$app\_tok(*id\_first + \! +);$
$\}$
$count - \! -;$
$\}$
$app(\mathtt{'\}'});\;\; app\_scrap(exp, maybe\_math);$

This code is used in section 221.

225.   We do not make the TeX string into a scrap, because there is no telling what the user will be putting into it; instead we leave it open, to be picked up by the next scrap. If it comes at the end of a section, it will be made into a scrap when *finish_C* is called.

There's a known bug here, in cases where an adjacent scrap is *prelangle* or *prerangle*. Then the TeX string can disappear when the \langle or \rangle becomes < or >. For example, if the user writes |x<@ty@>|, the TeX string \hbox{y} eventually becomes part of an *insert* scrap, which is combined with a *prelangle* scrap and eventually lost. The best way to work around this bug is probably to enclose the @t...@> in @[...@] so that the TeX string is treated as an expression.

⟨ Append a TeX string, without forming a scrap 225 ⟩ ≡
$app\_str(\mathtt{"\backslash\backslash hbox\{"});$
while $(id\_first < id\_loc)$
  if $((\mathbf{eight\_bits})(*id\_first) > °177)\;\{$
    $app\_tok(quoted\_char);\;\; app\_tok((\mathbf{eight\_bits})(*id\_first + \! +));$
  $\}$
  else $\{$
    if $(*id\_first \equiv \mathtt{'@'})\; id\_first + \! +;$
    $app\_tok(*id\_first + \! +);$
  $\}$
$app(\mathtt{'\}'});$

This code is used in section 221.

---

226.   The function *app_cur_id* appends the current identifier to the token list; it also builds a new scrap if *scrapping* ≡ *true*.

⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *app_cur_id*(**boolean**);
  static text_pointer *C_translate*(**void**);
  static void *outer_parse*(**void**);

227.   **static void** *app_cur_id*(**boolean** *scrapping*)
    ▷ are we making this into a scrap? ◁
  {
    name_pointer *p* ← *id_lookup*(*id_first*, *id_loc*, *normal*);
    if (*p⃗ilk* ≤ *custom*) {   ▷ not a reserved word ◁
      app(*id_flag* + (**int**)(*p* − *name_dir*));
      if (*scrapping*) *app_scrap*(*p⃗ilk* ≡ *func_template* ? *ftemplate* : *exp*,
              p⃗ilk ≡ *custom* ? *yes_math* : *maybe_math*);
    }
    else {
      app(*res_flag* + (**int**)(*p* − *name_dir*));
      if (*scrapping*) {
        if (*p⃗ilk* ≡ *alfop*) *app_scrap*(*ubinop*, *yes_math*)
        else *app_scrap*(*p⃗ilk*, *maybe_math*);
      }
    }
    ⟨ Flag the usage of this identifier, for the mini-index 276 ⟩;
  }

228.   When the '|' that introduces C text is sensed, a call on *C_translate* will return a pointer to the TEX translation of that text. If scraps exist in *scrap_info*, they are unaffected by this translation process.

  static text_pointer *C_translate*(**void**)
  {
    text_pointer *p*;   ▷ points to the translation ◁
    scrap_pointer *save_base*;   ▷ holds original value of *scrap_base* ◁

    save_base ← *scrap_base*; *scrap_base* ← *scrap_ptr* + 1; *C_parse*(*section_name*);
        ▷ get the scraps together ◁
    if (*next_control* ≠ '|') *err_print*(_("!␣Missing␣'|'␣after␣C␣text"));
    app_tok(*cancel*); *app_scrap*(*insert*, *maybe_math*);
        ▷ place a *cancel* token as a final "comment" ◁
    p ← *translate*();   ▷ make the translation ◁
    if (*scrap_ptr* > *max_scr_ptr*) *max_scr_ptr* ← *scrap_ptr*;
    scrap_ptr ← *scrap_base* − 1; *scrap_base* ← *save_base*;   ▷ scrap the scraps ◁
    return *p*;
  }

229. The *outer_parse* routine is to *C_parse* as *outer_xref* is to *C_xref*: It constructs a sequence of scraps for C text until *next_control* ≥ *format_code*. Thus, it takes care of embedded comments.

The token list created from within '| ... |' brackets is output as an argument to \PB, if the user has invoked CTWILL with the +e flag. Although ctwimac ignores \PB, other macro packages might use it to localize the special meaning of the macros that mark up program text.

#define *make_pb* *flags*['e']

⟨ Set initial values 29 ⟩ +≡
  make_pb ← *true*;

---

230. **static void** *outer_parse*(**void**)   ▷ makes scraps from C tokens and comments ◁
  {
    int *bal*;   ▷ brace level in comment ◁
    text_pointer *p*, *q*;   ▷ partial comments ◁
    while (*next_control* < *format_code*)
      if (*next_control* ≠ *begin_comment* ∧ *next_control* ≠ *begin_short_comment*)
        C_parse(*ignore*);
      else {
        boolean *is_long_comment* ← (*next_control* ≡ *begin_comment*);

        ⟨ Make sure that there is room for the new scraps, tokens, and texts 222 ⟩
        app(*cancel*); *app*(*inserted*);
        if (*is_long_comment*) *app_str*("\\C{");
        else *app_str*("\\SHC{");
        bal ← *copy_comment*(*is_long_comment*, 1); *next_control* ← *ignore*;
        while (*bal* > 0) {
          p ← *text_ptr*; *freeze_text*; *q* ← *C_translate*( );
            ▷ at this point we have *tok_ptr* + 6 ≤ *max_toks* ◁
          app(*tok_flag* + (**int**)(*p* − *tok_start*)); *app*(*inserted*);
          if (*make_pb*) *app_str*("\\PB{");
          app(*inner_tok_flag* + (**int**)(*q* − *tok_start*));
          if (*make_pb*) *app_tok*('}');
          if (*next_control* ≡ '|') {
            bal ← *copy_comment*(*is_long_comment*, *bal*); *next_control* ← *ignore*;
          }
          else *bal* ← 0;   ▷ an error has been reported ◁
        }
        app(*force*); *app_scrap*(*insert*, *no_math*);   ▷ the full comment becomes a scrap ◁
      }
  }

231. Output of tokens.   So far our programs have only built up multi-layered
token lists in `CWEAVE`'s internal memory; we have to figure out how to get them into
the desired final form. The job of converting token lists to characters in the TEX
output file is not difficult, although it is an implicitly recursive process. Four main
considerations had to be kept in mind when this part of `CWEAVE` was designed. (a)
There are two modes of output: *outer* mode, which translates tokens like *force* into
line-breaking control sequences, and *inner* mode, which ignores them except that
blank spaces take the place of line breaks. (b) The *cancel* instruction applies to
adjacent token or tokens that are output, and this cuts across levels of recursion since
'*cancel*' occurs at the beginning or end of a token list on one level. (c) The TEX
output file will be semi-readable if line breaks are inserted after the result of tokens
like *break_space* and *force*. (d) The final line break should be suppressed, and there
should be no *force* token output immediately after '\Y\B'.

---

232. The output process uses a stack to keep track of what is going on at different "levels" as the token lists are being written out. Entries on this stack have three parts:

   end_field is the *tok_mem* location where the token list of a particular level will end;

   tok_field is the *tok_mem* location from which the next token on a particular level will be read;

   mode_field is the current mode, either *inner* or *outer*.

The current values of these quantities are referred to quite frequently, so they are stored in a separate place instead of in the *stack* array. We call the current values *cur_end*, *cur_tok*, and *cur_mode*.

   The global variable *stack_ptr* tells how many levels of output are currently in progress. The end of output occurs when an *end_translation* token is found, so the stack is never empty except when we first begin the output process.

#define *inner* 0   ▷ value of *mode* for C texts within TEX texts ◁
#define *outer* 1   ▷ value of *mode* for C texts in sections ◁

⟨ Typedef declarations 25 ⟩ +≡
  typedef int mode;
  typedef struct {
    token_pointer *end_field*;   ▷ ending location of token list ◁
    token_pointer *tok_field*;   ▷ present location within token list ◁
    boolean *mode_field*;   ▷ interpretation of control tokens ◁
  } **output_state**;
  typedef output_state ∗stack_pointer;

233.   **#define** *cur_end* *cur_state.end_field*   ▷ current ending location in *tok_mem* ◁
#define *cur_tok* *cur_state.tok_field*   ▷ location of next output token in *tok_mem* ◁
#define *cur_mode* *cur_state.mode_field*   ▷ current mode of interpretation ◁
#define *init_stack* *stack_ptr* ← *stack*; *cur_mode* ← *outer*   ▷ initialize the stack ◁

⟨ Private variables 26 ⟩ +≡
  static output_state *cur_state*;   ▷ *cur_end*, *cur_tok*, *cur_mode* ◁
  static output_state *stack*[*stack_size*];   ▷ info for non-current levels ◁
  static stack_pointer *stack_end* ← *stack* + *stack_size* − 1;   ▷ end of *stack* ◁
  static stack_pointer *stack_ptr*;   ▷ first unused location in the output state stack ◁
  static stack_pointer *max_stack_ptr*;   ▷ largest value assumed by *stack_ptr* ◁

234.   ⟨ Set initial values 29 ⟩ +≡
  max_stack_ptr ← *stack*;

235.   To insert token-list p into the output, the *push_level* subroutine is called; it saves the old level of output and gets a new one going. The value of *cur_mode* is not changed.

⟨ Predeclaration of procedures 11 ⟩ +≡
  static void *push_level*(**text_pointer**);
  static void *pop_level*(**void**);

236.   **static void** *push_level*(▷ suspends the current level ◁
      text_pointer *p*)
  {
     if (*stack_ptr* ≡ *stack_end*) *overflow*(_("stack"));
     if (*stack_ptr* > *stack*) {   ▷ save current state ◁
        stack_ptr�*end_field* ← *cur_end*; *stack_ptr*→*tok_field* ← *cur_tok*;
        stack_ptr→*mode_field* ← *cur_mode*;
     }
     stack_ptr ++;
     if (*stack_ptr* > *max_stack_ptr*) *max_stack_ptr* ← *stack_ptr*;
     cur_tok ← ∗*p*; *cur_end* ← ∗(*p* + 1);
  }

237.   Conversely, the *pop_level* routine restores the conditions that were in force when the current level was begun. This subroutine will never be called when *stack_ptr* ≡ 1.

  static void *pop_level*(**void**)
  {
     cur_end ← (−− *stack_ptr*)→*end_field*; *cur_tok* ← *stack_ptr*→*tok_field*;
     cur_mode ← *stack_ptr*→*mode_field*;
  }

238.   The *get_output* function returns the next byte of output that is not a reference to a token list. It returns the values *identifier* or *res_word* or *section_code* if the next token is to be an identifier (typeset in italics), a reserved word (typeset in boldface), or a section name (typeset by a complex routine that might generate additional levels of output). In these cases *cur_name* points to the identifier or section name in question.

⟨ Private variables 26 ⟩ +≡
  static name_pointer *cur_name*;

239.   **#define** *res_word* ∘201   ▷ returned by *get_output* for reserved words ◁
#define *section_code* ∘200   ▷ returned by *get_output* for section names ◁

⟨ Predeclaration of procedures 11 ⟩ +≡
  static eight_bits *get_output*(**void**);
  static void *output_C*(**void**);
  static void *make_output*(**void**);

_ = macro ( ), §7.
boolean = **bool**, §6.
eight_bits = **uint8_t**, §6.
end_translation = ∘223 , §122.
get_output: **static eight_bits** ( ), §240.

identifier = ∘202 , §55.
name_pointer = **name_info** ∗, §13.
overflow: **void** ( ), COMMON.W §71.
p: **text_pointer**, §230.

stack_size = 500, §20.
text_pointer = **token_pointer** ∗, §39.
tok_mem: **static token** [ ], §40.
token_pointer = **token** ∗, §39.

240.   **static eight_bits** *get_output*(**void**)   ▷ returns the next token of output ◁
{
    sixteen_bits a;   ▷ current item read from *tok_mem* ◁
restart:
  while (*cur_tok* ≡ *cur_end*) *pop_level*( );
  $a \leftarrow *(cur\_tok \mathbin{+}\mathbin{+})$;
  if ($a \geq °400$) {
    cur_name ← a % *id_flag* + *name_dir*;
    switch (a/*id_flag*) {
    case 2: **return** *res_word*;   ▷ $a \equiv res\_flag + cur\_name$ ◁
    case 3: **return** *section_code*;   ▷ $a \equiv section\_flag + cur\_name$ ◁
    case 4: *push_level*(a % *id_flag* + *tok_start*); **goto** *restart*;
        ▷ $a \equiv tok\_flag + cur\_name$ ◁
    case 5: *push_level*(a % *id_flag* + *tok_start*); *cur_mode* ← *inner*; **goto** *restart*;
        ▷ $a \equiv inner\_tok\_flag + cur\_name$ ◁
    default: **return** *identifier*;   ▷ $a \equiv id\_flag + cur\_name$ ◁
    }
  }
  return (**eight_bits**) a;
}

241.   The real work associated with token output is done by *make_output*. This procedure appends an *end_translation* token to the current token list, and then it repeatedly calls *get_output* and feeds characters to the output buffer until reaching the *end_translation* sentinel. It is possible for *make_output* to be called recursively, since a section name may include embedded C text; however, the depth of recursion never exceeds one level, since section names cannot be inside of section names.

    A procedure called *output_C* does the scanning, translation, and output of C text within '| . . . |' brackets, and this procedure uses *make_output* to output the current token list. Thus, the recursive call of *make_output* actually occurs when *make_output* calls *output_C* while outputting the name of a section.

  static void *output_C*(**void**)   ▷ outputs the current token list ◁
  {
    token_pointer *save_tok_ptr*;
    text_pointer *save_text_ptr*, p;   ▷ translation of the C text ◁
    sixteen_bits *save_next_control*;   ▷ values to be restored ◁

    save_tok_ptr ← *tok_ptr*; *save_text_ptr* ← *text_ptr*; *save_next_control* ← *next_control*;
    next_control ← *ignore*; p ← *C_translate*( ); *app*(*inner_tok_flag* + (**int**)(p − *tok_start*));
    if (*make_pb*) {
      out_str("\\PB{"); *make_output*( ); *out*('}');
    } **else** *make_output*( );   ▷ output the list ◁
    if (*text_ptr* > *max_text_ptr*) *max_text_ptr* ← *text_ptr*;
    if (*tok_ptr* > *max_tok_ptr*) *max_tok_ptr* ← *tok_ptr*;
    text_ptr ← *save_text_ptr*; *tok_ptr* ← *save_tok_ptr*;   ▷ forget the tokens ◁
    next_control ← *save_next_control*;   ▷ restore *next_control* to original state ◁
  }

242. Here is `CWEAVE`'s major output handler.

```
static void make_output(void)      ▷ outputs the equivalents of tokens ◁
{
    eight_bits a ← 0,      ▷ current output byte ◁
    b;      ▷ next output byte ◁
    int c;      ▷ count of indent and outdent tokens ◁
    char scratch[longest_name + 1];      ▷ scratch area for section names ◁
    char *k, *k_limit;      ▷ indices into scratch ◁
    char *j;      ▷ index into buffer ◁
    char *p;      ▷ index into byte_mem ◁
    char delim;      ▷ first and last character of string being copied ◁
    char *save_loc, *save_limit;      ▷ loc and limit to be restored ◁
    name_pointer cur_section_name;      ▷ name of section being output ◁
    boolean save_mode;      ▷ value of cur_mode before a sequence of breaks ◁

    app(end_translation);      ▷ append a sentinel ◁
    freeze_text; push_level(text_ptr − 1);
    while (true) {
        a ← get_output();
    reswitch:
        switch (a) {
        case end_translation: return;
        case identifier: case res_word: ⟨Output an identifier 243⟩
            break;
        case section_code: ⟨Output a section name 247⟩
            break;
        case math_rel: out_str("\\MRL{");
        case noop: case inserted: break;
        case cancel: case big_cancel: c ← 0; b ← a;
            while (true) {
                a ← get_output();
                if (a ≡ inserted) continue;
                if ((a < indent ∧ ¬(b ≡ big_cancel ∧ a ≡ '␣')) ∨ a > big_force) break;
                if (a ≡ indent) c++;
                else if (a ≡ outdent) c−−;
                else if (a ≡ opt) a ← get_output();
            }
            ⟨Output saved indent or outdent tokens 246⟩
            goto reswitch;
        case indent: case outdent: case opt: case backup: case break_space: case force:
            case big_force: case preproc_line:
            ⟨Output a control, look ahead in case of line breaks, possibly goto reswitch 244⟩
            break;
        case quoted_char: out(*(cur_tok ++));
        case qualifier: break;
        default: out(a);      ▷ otherwise a is an ordinary character ◁
        }
    }
}
```

243.   An identifier of length one does not have to be enclosed in braces, and it looks
slightly better if set in a math-italic font instead of a (slightly narrower) text-italic
font. Thus we output '`\|a`' but '`\\{aa}`'.

⟨ Output an identifier 243 ⟩ ≡
    out(`'\\'`);
    if ($a \equiv identifier$) {
        if ($cur\_name{\rightarrow}ilk \equiv custom \wedge \neg doing\_format$) {
        $custom\_out$:
            for ($p \leftarrow cur\_name{\rightarrow}byte\_start$; $p < (cur\_name + 1){\rightarrow}byte\_start$; $p{+}{+}$)
                out($*p \equiv$ `'_'` ? `'x'` : $*p \equiv$ `'$'` ? `'X'` : $*p$);
            break;
        }
        else if ($is\_tiny(cur\_name)$) out(`'|'`)
        else {
            $delim \leftarrow$ `'.'`;
            for ($p \leftarrow cur\_name{\rightarrow}byte\_start$; $p < (cur\_name + 1){\rightarrow}byte\_start$; $p{+}{+}$)
                if ($xislower(*p)$) {   ▷ not entirely uppercase ◁
                    $delim \leftarrow$ `'\\'`; **break**;
                }
            out($delim$);
        }
    } **else if** ($cur\_name{\rightarrow}ilk \equiv alfop$) {
        out(`'X'`); **goto** $custom\_out$;
    } **else** out(`'&'`);   ▷ $a \equiv res\_word$ ◁
    if ($is\_tiny(cur\_name)$) {
        if ($isxalpha((cur\_name{\rightarrow}byte\_start)[0])$) out(`'\\'`);
        out(($cur\_name{\rightarrow}byte\_start$)[0]);
    }
    else $out\_name(cur\_name, true)$;
This code is used in section 242.

244. The current mode does not affect the behavior of `CWEAVE`'s output routine except when we are outputting control tokens.

⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 244 ⟩ ≡
  if $(a < break\_space \lor a \equiv preproc\_line)$ {
    if $(cur\_mode \equiv outer)$ {
      $out('\backslash\backslash')$; $out(a - cancel + '0')$;
      if $(a \equiv opt)$ {
        $b \leftarrow get\_output()$;   ▷ *opt* is followed by a digit ◁
        if $(b \neq '0' \lor force\_lines \equiv false)$ $out(b)$
        else $out\_str("\{-1\}")$;   ▷ *force_lines* encourages more @| breaks ◁
      }
    }
    else if $(a \equiv opt)$ $b \leftarrow get\_output()$;   ▷ ignore digit following *opt* ◁
  }
  else {⟨ Look ahead for strongest line break, **goto** *reswitch* 245 ⟩}
This code is used in section 242.

245. If several of the tokens *break_space*, *force*, *big_force* occur in a row, possibly mixed with blank spaces (which are ignored), the largest one is used. A line break also occurs in the output file, except at the very end of the translation. The very first line break is suppressed (i.e., a line break that follows '\Y\B').

⟨ Look ahead for strongest line break, **goto** *reswitch* 245 ⟩ ≡
  $b \leftarrow a$; $save\_mode \leftarrow cur\_mode$; $c \leftarrow 0$;
  while $(true)$ {
    $a \leftarrow get\_output()$;
    if $(a \equiv inserted)$ **continue**;
    if $(a \equiv cancel \lor a \equiv big\_cancel)$ {
      ⟨ Output saved *indent* or *outdent* tokens 246 ⟩
      goto *reswitch*;   ▷ *cancel* overrides everything ◁
    }
    if $((a \neq '\sqcup' \land a < indent) \lor a \equiv backup \lor a > big\_force)$ {
      if $(save\_mode \equiv outer)$ {
        if $(out\_ptr > out\_buf + 3 \land strncmp(out\_ptr - 3, "\backslash\backslash Y\backslash\backslash B", 4) \equiv 0)$ **goto** *reswitch*;
        ⟨ Output saved *indent* or *outdent* tokens 246 ⟩
        $out('\backslash\backslash')$; $out(b - cancel + '0')$;
        if $(a \neq end\_translation)$ $finish\_line()$;
      }
      else if $(a \neq end\_translation \land cur\_mode \equiv inner)$ $out('\sqcup')$;
      goto *reswitch*;
    }
    if $(a \equiv indent)$ $c\!+\!+$;
    else if $(a \equiv outdent)$ $c\!-\!-$;
    else if $(a \equiv opt)$ $a \leftarrow get\_output()$;
    else if $(a > b)$ $b \leftarrow a$;   ▷ if $a \equiv '\sqcup'$ we have $a < b$ ◁
  }
This code is used in section 244.

246. ⟨Output saved *indent* or *outdent* tokens 246⟩ ≡
  for (; $c > 0$; $c--$) out_str("\\1");
  for (; $c < 0$; $c++$) out_str("\\2");
This code is used in sections 242 and 245.

247. The remaining part of *make_output* is somewhat more complicated. When we output a section name, we may need to enter the parsing and translation routines, since the name may contain C code embedded in | ... | constructions. This C code is placed at the end of the active input buffer and the translation process uses the end of the active *tok_mem* area.

⟨Output a section name 247⟩ ≡
  {
    out_str("\\X"); cur_xref ← (**xref_pointer**) cur_name→$xref$;
    if (cur_xref→num ≡ $file\_flag$) {
      an_output ← $true$; cur_xref ← cur_xref→$xlink$;
    }
    else an_output ← $false$;
    if (cur_xref→num ≥ def_flag) {
      $out\_section$(cur_xref→num − def_flag);
      if ($phase$ ≡ 3) {
        cur_xref ← cur_xref→$xlink$;
        while (cur_xref→num ≥ def_flag) {
          out_str(",␣"); $out\_section$(cur_xref→num − def_flag);
          cur_xref ← cur_xref→$xlink$;
        }
      }
    }
    else out('0');   ▷ output the section number, or zero if it was undefined ◁
    out(':');
    if (an_output) out_str("\\.{");
    ⟨Output the text of the section name 248⟩
    if (an_output) out_str("␣}");
    out_str("\\X");
  }
This code is used in section 242.

---

a: **eight_bits**, §242.
an_output: **static boolean**, §92.
b: **eight_bits**, §242.
$backup$ = °215, §122.
big_cancel = °210, §122.
big_force = °220, §122.
$break\_space$ = °216, §122.
c: **int**, §242.
$cancel$ = °211, §122.
cur_mode = macro, §233.
cur_name: **static name_pointer**, §238.
cur_xref: **static xref_pointer**, §92.
def_flag = macro, §34.
$end\_translation$ = °223, §122.

$false$, <stdbool.h>.
$file\_flag$ = macro, §34.
$finish\_line$: **static void** ( ), §99.
$force$ = °217, §122.
$force\_lines$ = macro, §185.
get_output: **static eight_bits** ( ), §240.
$indent$ = °212, §122.
$inner$ = 0, §232.
$inserted$ = °224, §122.
$make\_output$: **static void** ( ), §242.
num: **sixteen_bits**, §25.
opt = °214, §122.
out = macro ( ), §101.
out_buf: **static char** [], §96.
out_ptr: **static char** ∗, §96.

$out\_section$: **static void** ( ), §107.
out_str, §102.
$outdent$ = °213, §122.
$outer$ = 1, §232.
$phase$: **int**, COMMON.W §19.
$preproc\_line$ = °221, §122.
$reswitch$: label, §242.
$save\_mode$: **boolean**, §242.
$strncmp$, <string.h>.
tok_mem: **static token** [], §40.
$true$, <stdbool.h>.
$xlink$: **struct xref_info** ∗, §25.
$xref$ = macro, §34.
xref_pointer = **xref_info** ∗, §25.

248. ⟨Output the text of the section name 248⟩ ≡
  sprint_section_name(*scratch*, *cur_name*); *k* ← *scratch*;
  k_limit ← *scratch* + *strlen*(*scratch*); *cur_section_name* ← *cur_name*;
  while (*k* < *k_limit*) {
    b ← ∗(*k*++);
    if (*b* ≡ '@') ⟨Skip next character, give error if not '@' 249⟩
    if (*an_output*)
      switch (*b*) {
      case '␣': **case** '\\': **case** '#': **case** '%': **case** '$': **case** '^': **case** '{':
        case '}': **case** '~': **case** '&': **case** '_': *out*('\\'); ▷ falls through ◁
      default: *out*(*b*);
      }
    else {
      if (*b* ≠ '|') *out*(*b*)
      else {
        ⟨Copy the C text into the *buffer* array 250⟩
        save_loc ← *loc*; *save_limit* ← *limit*; *loc* ← *limit* + 2; *limit* ← *j* + 1;
        ∗*limit* ← '|'; *output_C*(); *loc* ← *save_loc*; *limit* ← *save_limit*;
      }
    }
  }
This code is used in section 247.

249. ⟨Skip next character, give error if not '@' 249⟩ ≡
  if (∗*k*++ ≠ '@') {
    fputs(_("\n!␣Illegal␣control␣code␣in␣section␣name:␣<"), *stdout*);
    print_section_name(*cur_section_name*); *printf*(">␣"); *mark_error*;
  }
This code is used in section 248.

250.   The C text enclosed in | . . . | should not contain '|' characters, except within strings. We put a '|' at the front of the buffer, so that an error message that displays the whole buffer will look a little bit sensible. The variable *delim* is zero outside of strings, otherwise it equals the delimiter that began the string being copied.

⟨ Copy the C text into the *buffer* array 250 ⟩ ≡
   j ← *limit* + 1; ∗*j* ← '|'; *delim* ← 0;
   while (*true*) {
     if (*k* ≥ *k_limit*) {
       fputs(_("\n!␣C␣text␣in␣section␣name␣didn't␣end:␣<"), *stdout*);
       print_section_name(*cur_section_name*); *printf*(">␣"); *mark_error*; **break**;
     }
     b ← ∗(*k*++);
     if (*b* ≡ '@' ∨ (*b* ≡ '\\' ∧ *delim* ≠ 0)) ⟨ Copy a quoted character into the buffer 251 ⟩
     else {
       if (*b* ≡ '\'' ∨ *b* ≡ '"') {
         if (*delim* ≡ 0) *delim* ← *b*;
         else if (*delim* ≡ *b*) *delim* ← 0;
       }
       if (*b* ≠ '|' ∨ *delim* ≠ 0) {
         if (*j* > *buffer* + *long_buf_size* − 3) *overflow*(_("buffer"));
         ∗(++*j*) ← *b*;
       }
       else break;
     }
   }

This code is used in section 248.

251.   ⟨ Copy a quoted character into the buffer 251 ⟩ ≡
   {
     if (*j* > *buffer* + *long_buf_size* − 4) *overflow*(_("buffer"));
     ∗(++*j*) ← *b*; ∗(++*j*) ← ∗(*k*++);
   }

This code is used in section 250.

_ = macro ( ), §7.
an_output: **static boolean**, §92.
b: **eight_bits**, §242.
buffer: **char** [], COMMON.W §22.
cur_name: **static name_pointer**, §238.
cur_section_name: **name_pointer**, §242.
delim: **char**, §242.
fputs, <stdio.h>.

j: **char** ∗, §242.
k: **char** ∗, §242.
k_limit: **char** ∗, §242.
limit: **char** ∗, COMMON.W §22.
loc: **char** ∗, COMMON.W §22.
long_buf_size = macro, §20.
mark_error = macro, §15.
out = macro ( ), §101.
output_C: **static void** ( ), §241.
overflow: **void** ( ), COMMON.W §71.

print_section_name: **void** ( ), COMMON.W §52.
printf, <stdio.h>.
save_limit: **char** ∗, §242.
save_loc: **char** ∗, §242.
scratch: **char** [], §242.
sprint_section_name: **void** ( ), COMMON.W §53.
stdout, <stdio.h>.
strlen, <string.h>.
true, <stdbool.h>.

252. Phase two processing.   We have assembled enough pieces of the puzzle in order to be ready to specify the processing in CWEAVE's main pass over the source file. Phase two is analogous to phase one, except that more work is involved because we must actually output the TEX material instead of merely looking at the CWEB specifications.

> **static void** *phase_two*(**void**)
> {
>   *reset_input*( );
>   **if** (*show_progress*) *fputs*($ $("\nWriting␣the␣output␣file..."), *stdout*);
>   *temp_switch* ← *false*; *temp_meaning_ptr* ← *temp_meaning_stack*;
>   ⟨Read the .aux file, if present; then open it for output 254⟩;
>   *section_count* ← 0; *format_visible* ← *true*; *right_start_switch* ← *false*; *copy_limbo*( );
>   *finish_line*( ); *flush_buffer*(*out_buf*, *false*, *false*);   ▷ insert a blank line, it looks nice ◁
>   **while** (¬*input_has_ended*) ⟨Translate the current section 257⟩
> }

253. ⟨Private variables 26⟩ +≡
>   **static FILE** *∗aux_file*;
>   **static char** *aux_file_name*[*max_file_name_length*];   ▷ name of .aux file ◁

254. ⟨Read the .aux file, if present; then open it for output 254⟩ ≡
>   *memcpy*(*aux_file_name*, *tex_file_name*, *strlen*(*tex_file_name*) − 4);
>   *strcat*(*aux_file_name*, ".bux"); *include_depth* ← 1;   ▷ we simulate @i ◁
>   *strcpy*(*cur_file_name*, *aux_file_name*);   ▷ first in, third out ◁
>   **if** ((*cur_file* ← *fopen*(*cur_file_name*, "r"))) {
>   *cur_line* ← 0; *include_depth* ++;
>   }
>   *strcpy*(*aux_file_name* + *strlen*(*aux_file_name*) − 4, ".aux");
>   *strcpy*(*cur_file_name*, *aux_file_name*);   ▷ second in, second out ◁
>   **if** ((*cur_file* ← *fopen*(*cur_file_name*, "r"))) {
>   *cur_line* ← 0; *include_depth* ++;
>   }
>   *strcpy*(*cur_file_name*, "system.bux");   ▷ third in, first out ◁
>   **if** ((*cur_file* ← *fopen*(*cur_file_name*, "r"))) *cur_line* ← 0;
>   **else** *include_depth* −−;
>   **if** (*include_depth*) {   ▷ at least one new file was opened ◁
>   **while** (*get_next*( ) ≡ *meaning*) ;   ▷ new meaning is digested ◁
>   **if** (*include_depth*) *err_print*($ $("!␣Only␣@$␣is␣allowed␣in␣aux␣and␣bux␣files"));
>   *finish_line*( ); *loc* ← *buffer*;   ▷ now reading beginning of line 1 ◁
>   }
>   **if** ((*aux_file* ← *fopen*(*aux_file_name*, "wb")) ≡ Λ)
>   *fatal*($ $("!␣Cannot␣open␣aux␣output␣file␣"), *aux_file_name*);
> This code is used in section 252.

255. ⟨Predeclaration of procedures 11⟩ +≡   **static void** *phase_two*(**void**);

256.   The output file will contain the control sequence `\Y` between non-null sections of a section, e.g., between the TEX and definition parts if both are nonempty. This puts a little white space between the parts when they are printed. However, we don't want `\Y` to occur between two definitions within a single section. The variables *out_line* or *out_ptr* will change if a section is non-null, so the following macros '*save_position*' and '*emit_space_if_needed*' are able to handle the situation:

#define *save_position*   *save_line* ← *out_line*; *save_place* ← *out_ptr*
#define *emit_space_if_needed*
      if (*save_line* ≠ *out_line* ∨ *save_place* ≠ *out_ptr*) *out_str*("\\Y");
      space_checked ← *true*;

⟨ Private variables 26 ⟩ +≡
  static int *save_line*;   ▷ former value of *out_line* ◁
  static char *∗save_place*;   ▷ former value of *out_ptr* ◁
  static int *sec_depth*;   ▷ the integer, if any, following `@*` ◁
  static boolean *space_checked*;   ▷ have we done *emit_space_if_needed*? ◁
  static boolean *format_visible*;   ▷ should the next format declaration be output? ◁
  static boolean *doing_format* ← *false*;   ▷ are we outputting a format declaration? ◁
  static boolean *group_found* ← *false*;   ▷ has a starred section occurred? ◁
  static boolean *right_start_switch*;   ▷ has '`@r`' occurred recently? ◁
  static boolean *temp_switch*;   ▷ has '`@%`' occurred recently? ◁

257.   **#define** *usage_sentinel* (**struct perm_meaning** ∗) 1

⟨ Translate the current section 257 ⟩ ≡
  {
    section_count ++; *temp_switch* ← *false*; *temp_meaning_ptr* ← *temp_meaning_stack*;
    top_usage ← *usage_sentinel*; ⟨ Output the code for the beginning of a new section 258 ⟩
    save_position; ⟨ Translate the TEX part of the current section 259 ⟩
    ⟨ Translate the definition part of the current section 260 ⟩
    ⟨ Translate the C part of the current section 268 ⟩
    ⟨ Show cross-references to this section 271 ⟩
    ⟨ Output the code for the end of a section 275 ⟩
  }
This code is used in section 252.

_ = macro ( ), §7.
boolean = **bool**, §6.
buffer: **char** [ ],
  COMMON.W §22.
copy_limbo: **static void** ( ),
  §110.
cur_file = macro, §10.
cur_file_name = macro, §10.
cur_line = macro, §10.
err_print: **void** ( ),
  COMMON.W §66.
false, `<stdbool.h>`.
fatal: **void** ( ), COMMON.W §70.
finish_line: **static void** ( ), §99.
flush_buffer: **static void** ( ),
  §98.
fopen, `<stdio.h>`.
fputs, `<stdio.h>`.

get_next: **static eight_bits**
  ( ), §57.
include_depth: **int**,
  COMMON.W §25.
input_has_ended: **boolean**,
  COMMON.W §25.
loc: **char** ∗, COMMON.W §22.
max_file_name_length = 1024,
  §10.
meaning = °207, §48.
memcpy, `<string.h>`.
out_buf: **static char** [ ], §96.
out_line: **static int**, §96.
out_ptr: **static char** ∗, §96.
out_str, §102.
perm_meaning: **struct**, §28.
reset_input: **void** ( ),

COMMON.W §35.
section_count: **sixteen_bits**,
  COMMON.W §37.
show_progress = macro, §17.
stdout, `<stdio.h>`.
strcat, `<string.h>`.
strcpy, `<string.h>`.
strlen, `<string.h>`.
temp_meaning_ptr: **static**
  meaning_struct ∗, §28.
temp_meaning_stack: **static**
  meaning_struct [ ], §28.
tex_file_name: **char** [ ],
  COMMON.W §73.
top_usage: **static struct**
  perm_meaning ∗, §28.
true, `<stdbool.h>`.

258.   Sections beginning with the `CWEB` control sequence '`@␣`' start in the output with the TEX control sequence '`\M`', followed by the section number. Similarly, '`@*`' sections lead to the control sequence '`\N`'. In this case there's an additional parameter, representing one plus the specified depth, immediately after the `\N`. If the section has changed, we put `\*` just after the section number.

⟨ Output the code for the beginning of a new section 258 ⟩ ≡
```
  if (*(loc − 1) ≠ '*') {
    if (right_start_switch) {
      out_str("\\shortpage\n"); right_start_switch ← false;
    }
    out_str("\\M");
  }
  else {
    while (*loc ≡ '␣') loc ++;
    if (*loc ≡ '*') {        ▷ "top" level ◁
      sec_depth ← −1; loc ++;
    }
    else {
      for (sec_depth ← 0; xisdigit(*loc); loc ++)
        sec_depth ← sec_depth * 10 + (*loc) − '0';
    }
    while (*loc ≡ '␣') loc ++;        ▷ remove spaces before group title ◁
    group_found ← true; out_str("\\N");
    if (right_start_switch) {
      out_str("N"); right_start_switch ← false;
    }
    { char s[32]; sprintf(s, "{%d}", sec_depth + 1); out_str(s); }
    if (show_progress) printf("*%d", section_count);
    update_terminal;        ▷ print a progress report ◁
  }
  out_str("{"); out_section(section_count); out_str("}"); flush_buffer(out_ptr, false, false);
```
This code is used in section 257.

259.   In the TEX part of a section, we simply copy the source text, except that index entries are not copied and C text within `| . . . |` is translated.

⟨ Translate the TEX part of the current section 259 ⟩ ≡
```
  do { next_control ← copy_TEX();
    switch (next_control) {
    case '|': init_stack; output_C(); break;
    case '@': out('@'); break;
    case temp_meaning: temp_switch ← true − temp_switch; break;
    case right_start: right_start_switch ← true; break;
    case TEX_string: case noop: case xref_roman: case xref_wildcard:
      case xref_typewriter: case meaning: case suppress: case section_name:
      loc −= 2; next_control ← get_next();        ▷ reprocess ◁
      if (next_control ≡ TEX_string)
        err_print(_("! TeX string should be in C text only"));
      break;
```

  case *thin_space*: **case** *math_break*: **case** *ord*: **case** *line_break*: **case** *big_line_break*:
   case *no_line_break*: **case** *join*: **case** *pseudo_semi*: **case** *macro_arg_open*:
   case *macro_arg_close*: **case** *output_defs_code*:
   err_print($\lrcorner$("!␣You␣can't␣do␣that␣in␣TeX␣text")); **break**;
  }
 } **while** (*next_control* < *format_code*);

This code is used in section 257.

260. When we get to the following code we have *next_control* ≥ *format_code*, and
the token memory is in its initial empty state.

⟨ Translate the definition part of the current section 260 ⟩ ≡
 space_checked ← *false*;
 while (*next_control* ≤ *definition*) { ▷ *format_code* or *definition* ◁
  init_stack;
  if (*next_control* ≡ *definition*) ⟨ Start a macro definition 264 ⟩
  else ⟨ Start a format definition 266 ⟩
  outer_parse( );
  if (*is_macro*) ⟨ Make ministring for a new macro 265 ⟩;
  finish_C(*format_visible*); *format_visible* ← *true*; *doing_format* ← *false*;
 }

This code is used in section 257.

261. ⟨ Private variables 26 ⟩ +≡
 static boolean *is_macro*; ▷ it's a macro def, not a format def ◁
 static int *def_diff*; ▷ 0 iff the current macro has parameters ◁
 static name_pointer *id_being_defined*; ▷ the definee ◁

---

262. The *finish_C* procedure outputs the translation of the current scraps, preceded by the control sequence '\B' and followed by the control sequence '\par'. It also restores the token and scrap memories to their initial empty state.

A *force* token is appended to the current scraps before translation takes place, so that the translation will normally end with \6 or \7 (the TEX macros for *force* and *big_force*). This \6 or \7 is replaced by the concluding \par or by \Y\par.

> **static void** *finish_C*(▷ finishes a definition or a C part ◁
>   **boolean** *visible*)   ▷ nonzero if we should produce TEX output ◁
> {
>   **text_pointer** *p*;   ▷ translation of the scraps ◁
>   **if** (*visible*) {
>   *out_str*("\\B"); *app_tok*(*force*); *app_scrap*(*insert*, *no_math*); *p* ← *translate*( );
>   *app*(*tok_flag* + (**int**)(*p* − *tok_start*)); *make_output*( );   ▷ output the list ◁
>   **if** (*out_ptr* > *out_buf* + 1) {
>   **if** (∗(*out_ptr* − 1) ≡ '\\') {
>   **if** (∗*out_ptr* ≡ '6') *out_ptr* −= 2;
>   **else if** (∗*out_ptr* ≡ '7') ∗*out_ptr* ← 'Y';
>   }
>   }
>   *out_str*("\\par"); *finish_line*( );
>   }
>   **if** (*text_ptr* > *max_text_ptr*) *max_text_ptr* ← *text_ptr*;
>   **if** (*tok_ptr* > *max_tok_ptr*) *max_tok_ptr* ← *tok_ptr*;
>   **if** (*scrap_ptr* > *max_scr_ptr*) *max_scr_ptr* ← *scrap_ptr*;
>   *tok_ptr* ← *tok_mem* + 1; *text_ptr* ← *tok_start* + 1; *scrap_ptr* ← *scrap_info*;
>   ▷ forget the tokens and the scraps ◁
> }

263. ⟨Predeclaration of procedures 11⟩ +≡ **static void** *finish_C*(**boolean**);

264. Keeping in line with the conventions of the C preprocessor (and otherwise contrary to the rules of CWEB) we distinguish here between the case that '(' immediately follows an identifier and the case that the two are separated by a space. In the latter case, and if the identifier is not followed by '(' at all, the replacement text starts immediately after the identifier. In the former case, it starts after we scan the matching ')'.

⟨Start a macro definition 264⟩ ≡
>   {
>   *is_macro* ← *true*;
>   **if** (*save_line* ≠ *out_line* ∨ *save_place* ≠ *out_ptr* ∨ *space_checked*) *app*(*backup*);
>   **if** (¬*space_checked*) {
>   *emit_space_if_needed*; *save_position*;
>   }
>   *app_str*("\\D");   ▷ this will produce '**define** ' ◁
>   **if** ((*next_control* ← *get_next*( )) ≠ *identifier*)
>   *err_print*(_("! ⌴Improper⌴macro⌴definition"));
>   **else** {
>   *id_being_defined* ← *id_lookup*(*id_first*, *id_loc*, *normal*); *app*('$'); *app_cur_id*(*false*);
>   *def_diff* ← ∗*loc* − '(';

if $(*loc \equiv$ '(')
reswitch:
  switch $(next\_control \leftarrow get\_next())$ {
  case '(': **case** ',': $app(next\_control)$; **goto** *reswitch*;
  case *identifier*: $app\_cur\_id(false)$; **goto** *reswitch*;
  case ')': $app(next\_control)$; $next\_control \leftarrow get\_next()$; **break**;
  default: $err\_print(\_("!\_Improper\_macro\_definition"))$; **break**;
  }
else $next\_control \leftarrow get\_next()$;
$app\_str("\$\_")$; $app(break\_space)$; $app\_scrap(dead, no\_math)$;
  ▷ scrap won't take part in the parsing ◁
  }
 }

This code is used in section 260.

265. ⟨Make ministring for a new macro 265⟩ ≡
```
  {
     ms_mode ← true;  ministring_ptr ← ministring_buf;  *ministring_ptr ++ ← ’=’;
     if (def_diff) {        ▷ parameterless ◁
        scrap_pointer s ← scrap_ptr;
        text_pointer t;
        token_pointer j;

        while (s⃗cat ≡ insert) s −−;
        if ((s − 1)⃗cat ≡ dead ∧ s⃗cat ≡ exp ∧ **(t ← s⃗trans) ≡ ’\\’ ∧ *(*t + 1) ≡ ’T’)
              ▷ it's just a constant ◁
           for (j ← *t; j < *(t + 1); j++) *ministring_ptr ++ ← *j;
        else  out_str("macro");
     }
     else  out_str("macro␣(\\,)");
     new_meaning(id_being_defined);
  }
```
This code is used in section 260.

266. ⟨Start a format definition 266⟩ ≡
```
  {
     doing_format ← true;  is_macro ← false;
     if (*(loc − 1) ≡ ’s’ ∨ *(loc − 1) ≡ ’S’) format_visible ← false;
     if (¬space_checked) {
        emit_space_if_needed;  save_position;
     }
     app_str("\\F");        ▷ this will produce ‘format ’ ◁
     next_control ← get_next( );
     if (next_control ≡ identifier) {
        app(id_flag + (int)(id_lookup(id_first, id_loc, normal) − name_dir));  app(’␣’);
        app(break_space);        ▷ this is syntactically separate from what follows ◁
        next_control ← get_next( );
        if (next_control ≡ identifier) {
           app(id_flag + (int)(id_lookup(id_first, id_loc, normal) − name_dir));
           app_scrap(exp, maybe_math);  app_scrap(semi, maybe_math);
           next_control ← get_next( );
        }
     }
     if (scrap_ptr ≠ scrap_info + 2)  err_print(_("! ␣Improper␣format␣definition"));
  }
```
This code is used in section 260.

267.   Finally, when the TEX and definition parts have been treated, we have
next_control ≥ *begin_C*. We will make the global variable *this_section* point to the
current section name, if it has a name.

⟨Private variables 26⟩ +≡
```
  static name_pointer this_section;        ▷ the current section name, or zero ◁
```

268. ⟨Translate the C part of the current section 268⟩ ≡
  this_section ← *name_dir* ;
  if (*next_control* ≤ *section_name*) {
    emit_space_if_needed ; *init_stack* ;
    if (*next_control* ≡ *begin_C*) *next_control* ← *get_next*( );
    else {
      this_section ← *cur_section* ; ⟨Check that '=' or '==' follows this section name, and
          emit the scraps to start the section definition 269⟩
    }
    while (*next_control* ≤ *section_name*) {
      outer_parse( ); ⟨Emit the scrap for a section name if present 270⟩
    }
    finish_C(*true*);
  }
This code is used in section 257.

_ = macro ( ), §7.
app = macro ( ), §143.
app_scrap = macro ( ), §220.
app_str: **static void** ( ), §145.
begin_C = °*233* , §48.
break_space = °*216* , §122.
cat: **eight_bits**, §136.
cur_section: **static**
  name_pointer, §55.
dead = 39, §117.
def_diff: **static int**, §261.
doing_format: **static boolean**,
  §256.
emit_space_if_needed = macro,
  §256.
err_print: **void** ( ),
  COMMON.W §66.
exp = 1, §117.
false, <stdbool.h>.
finish_C: **static void** ( ), §263.
format_visible: **static**
  boolean, §256.
get_next: **static eight_bits**
  ( ), §57.
id_being_defined: **static**

name_pointer, §261.
id_first: **char** *,
  COMMON.W §21.
id_flag = 10240, §139.
id_loc: **char** *, COMMON.W §21.
id_lookup: **name_pointer** ( ),
  COMMON.W §48.
identifier = °*202* , §55.
init_stack = macro, §233.
insert = 37, §117.
is_macro: **static boolean**,
  §261.
loc: **char** *, COMMON.W §22.
maybe_math = 0, §143.
ministring_buf: **static char**
  [ ], §28.
ministring_ptr: **static char** *,
  §28.
ms_mode: **static boolean**,
  §28.
name_dir: **name_info** [ ],
  COMMON.W §43.
name_pointer = **name_info**
  *, §13.

new_meaning: **static void** ( ),
  §32.
next_control: **static**
  eight_bits, §78.
normal = 0, §23.
out_str, §102.
outer_parse: **static void** ( ),
  §230.
save_position = macro, §256.
scrap_info: **static scrap** [ ],
  §137.
scrap_pointer = **scrap** *,
  §136.
scrap_ptr: **static**
  scrap_pointer, §137.
section_name = °*234* , §48.
semi = 27, §117.
space_checked: **static**
  boolean, §256.
text_pointer = **token_pointer**
  *, §39.
token_pointer = **token** *, §39.
trans = macro, §137.
true, <stdbool.h>.

269. The title of the section and an $\equiv$ or $+\equiv$ are made into a scrap that should not take part in the parsing.

⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 269 ⟩ ≡
  do $next\_control \leftarrow get\_next(\,)$; **while** $(next\_control \equiv \texttt{'+'})$; ▷ allow optional '+=' ◁
  if $(next\_control \neq \texttt{'='} \wedge next\_control \neq eq\_eq)$
    $err\_print(\text{\_}(\texttt{"!\_You\_need\_an\_=sign\_after\_the\_section\_name"}));$
  else $next\_control \leftarrow get\_next(\,)$;
  if $(out\_ptr > out\_buf + 1 \wedge *out\_ptr \equiv \texttt{'Y'} \wedge *(out\_ptr - 1) \equiv \texttt{'\backslash\backslash'})$ $app(backup)$;
    ▷ the section name will be flush left ◁
  $app(section\_flag + (\textbf{int})(this\_section - name\_dir))$;
  $cur\_xref \leftarrow (\textbf{xref\_pointer})\ this\_section \rightarrow xref$;
  if $(cur\_xref \rightarrow num \equiv file\_flag)$ $cur\_xref \leftarrow cur\_xref \rightarrow xlink$;
  $app\_str(\texttt{"\$\{\}"})$;
  if $(cur\_xref \rightarrow num \neq section\_count + def\_flag)$ {
    $app\_str(\texttt{"\backslash\backslash mathrel+"})$; ▷ section name is multiply defined ◁
    $this\_section \leftarrow name\_dir$; ▷ so we won't give cross-reference info here ◁
  }
  $app\_str(\texttt{"\backslash\backslash E"})$; ▷ output an equivalence sign ◁
  $app\_str(\texttt{"\{\}\$"})$; $app(force)$; $app\_scrap(dead, no\_math)$;
    ▷ this forces a line break unless '@+' follows ◁
This code is used in section 268.

270. ⟨ Emit the scrap for a section name if present 270 ⟩ ≡
  if $(next\_control < section\_name)$ {
    $err\_print(\text{\_}(\texttt{"!\_You\_can't\_do\_that\_in\_C\_text"}))$; $next\_control \leftarrow get\_next(\,)$;
  }
  else if $(next\_control \equiv section\_name)$ {
    $app(section\_flag + (\textbf{int})(cur\_section - name\_dir))$;
    $app\_scrap(section\_scrap, maybe\_math)$; $next\_control \leftarrow get\_next(\,)$;
  }
This code is used in section 268.

271. Cross references relating to a named section are given after the section ends.

⟨ Show cross-references to this section 271 ⟩ ≡
  if $(this\_section > name\_dir)$ {
    $cur\_xref \leftarrow (\textbf{xref\_pointer})\ this\_section \rightarrow xref$;
    if $(cur\_xref \rightarrow num \equiv file\_flag)$ {
      $an\_output \leftarrow true$; $cur\_xref \leftarrow cur\_xref \rightarrow xlink$;
    }
    else $an\_output \leftarrow false$;
    if $(cur\_xref \rightarrow num > def\_flag)$ $cur\_xref \leftarrow cur\_xref \rightarrow xlink$;
        ▷ bypass current section number ◁
    $footnote(def\_flag)$; $footnote(cite\_flag)$; $footnote(0)$;
  }
This code is used in section 257.

272.   The *footnote* procedure gives cross-reference information about multiply de-
fined section names (if the *flag* parameter is *def_flag*), or about references to a sec-
tion name (if *flag* ≡ *cite_flag*), or to its uses (if *flag* ≡ 0). It assumes that *cur_xref*
points to the first cross-reference entry of interest, and it leaves *cur_xref* pointing
to the first element not printed. Typical outputs: '\A101.'; '\Us 370\ET1009.';
'\As 8, 27*\ETs64.'.

Note that the output of CWEAVE is not English-specific; users may supply new
definitions for the macros \A, \As, etc.

```
static void footnote (     ▷ outputs section cross-references ◁
     sixteen_bits flag )
{
   xref_pointer q;      ▷ cross-reference pointer variable ◁

   if (cur_xref→num ≤ flag ) return;
   finish_line ( );  out('\\');  out(flag ≡ 0 ? 'U' : flag ≡ cite_flag ? 'Q' : 'A');
   ⟨Output all the section numbers on the reference list cur_xref  274⟩
   out('.');
}
```

273.   ⟨Predeclaration of procedures 11⟩ +≡   **static void** *footnote*(**sixteen_bits**);

_ = macro ( ), §7.
an_output: **static boolean**,
   §92.
app = macro ( ), §143.
app_scrap = macro ( ), §220.
app_str: **static void** ( ), §145.
backup = °215 , §122.
cite_flag = 10240, §34.
cur_section: **static**
   name_pointer, §55.
cur_xref: **static xref_pointer**,
   §92.
dead = 39, §117.
def_flag = macro, §34.
eq_eq = °36 , §8.
err_print: **void** ( ),

COMMON.W §66.
false, <stdbool.h>.
file_flag = macro, §34.
finish_line: **static void** ( ), §99.
force = °217 , §122.
get_next: **static eight_bits**
   ( ), §57.
maybe_math = 0, §143.
name_dir: **name_info** [ ],
   COMMON.W §48.
next_control: **static**
   eight_bits, §78.
no_math = 2, §143.
num: **sixteen_bits**, §25.
out = macro ( ), §101.

out_buf: **static char** [ ], §96.
out_ptr: **static char** ∗, §96.
section_count: **sixteen_bits**,
   COMMON.W §37.
section_flag = macro, §139.
section_name = °234 , §48.
section_scrap = 38, §117.
sixteen_bits = **uint16_t**, §6.
this_section: **static**
   name_pointer, §267.
true, <stdbool.h>.
xlink: **struct xref_info** ∗, §25.
xref = macro, §34.
xref_pointer = **xref_info** ∗,
   §25.

274. The following code distinguishes three cases, according as the number of cross-references is one, two, or more than two. Variable q points to the first cross-reference, and the last link is a zero.

⟨ Output all the section numbers on the reference list *cur_xref* 274 ⟩ ≡
   $q \leftarrow cur\_xref$;
   if $(q \rightarrow xlink \rightarrow num > flag)$ $out('s')$;   ▷ plural ◁
   while $(true)$ {
      $out\_section(cur\_xref \rightarrow num - flag)$; $cur\_xref \leftarrow cur\_xref \rightarrow xlink$;
        ▷ point to the next cross-reference to output ◁
      if $(cur\_xref \rightarrow num \leq flag)$ **break**;
      if $(cur\_xref \rightarrow xlink \rightarrow num > flag)$ $out\_str(",_{\sqcup}")$;   ▷ not the last ◁
      else {
         $out\_str("\backslash\backslash ET")$;   ▷ the last ◁
         if $(cur\_xref \neq q \rightarrow xlink)$ $out('s')$;   ▷ the last of more than two ◁
      }
   }
This code is used in section 272.

275. ⟨ Output the code for the end of a section 275 ⟩ ≡
  $finish\_line()$; $out\_str("\backslash\backslash mini")$; $finish\_line()$;
  ⟨ Output information about usage of id's defined in other sections 277 ⟩;
  $out\_str("\}\backslash\backslash FI")$; $finish\_line()$; $flush\_buffer(out\_buf, false, false)$;
    ▷ insert a blank line, it looks nice ◁
This code is used in section 257.

276. The following code is performed for each identifier parsed during a section. Variable *top_usage* is always nonzero; it has the sentinel value 1 initially, then it points to each variable scheduled for possible citation. A variable is on this list if and only if its *link* field is nonzero. All variables mentioned in the section are placed on the list, unless they are reserved and their current TEX meaning is uninitialized.

⟨ Flag the usage of this identifier, for the mini-index 276 ⟩ ≡
  {
    struct perm_meaning $*q \leftarrow p - name\_dir + cur\_meaning$;
    if $(\neg(abnormal(p)) \vee strcmp(q \rightarrow perm.tex\_part, "\backslash\backslash uninitialized") \neq 0)$
      if $(q \rightarrow link \equiv 0)$ {
        $q \rightarrow link \leftarrow top\_usage$; $top\_usage \leftarrow q$;
      }
  }
This code is used in section 227.

277. ⟨ Output information about usage of id's defined in other sections 277 ⟩ ≡
  {
    struct perm_meaning $*q$;
    while $(temp\_meaning\_ptr > temp\_meaning\_stack)$ {
      $out\_mini(-- temp\_meaning\_ptr)$;
      $q \leftarrow temp\_meaning\_ptr \rightarrow id - name\_dir + cur\_meaning$; $q \rightarrow stamp \leftarrow section\_count$;
        ▷ suppress output from "permanent" data ◁
    }

```
    while (top_usage ≠ usage_sentinel) {
        q ← top_usage;  top_usage ← q→link;  q→link ← Λ;
        if (q→stamp ≠ section_count)  out_mini(&(q→perm));
    }
}
```

This code is used in section 275.

278.   **static void** *out_mini*(**meaning_struct** *∗m*)
```
{
    char s[60];
    name_pointer cur_name ← m→id;
    if (m→prog_no ≡ 0) {          ▷ reference within current program ◁
        if (m→sec_no ≡ section_count) return;       ▷ defined in current section ◁
        sprintf(s, "\\[%d", m→sec_no);
    }
    else {
        name_pointer n ← title_code[m→prog_no];
        if (∗(n→byte_start) ≡ '{')  sprintf(s, "\\]%.*s%d",
                (int)((n + 1)→byte_start − n→byte_start), n→byte_start, m→sec_no);
        else  sprintf(s, "\\]%.*s", (int)((n + 1)→byte_start − n→byte_start), n→byte_start);
    }
    out_str(s);  out('␣');  ⟨Mini-output the name at cur_name 280⟩;
    out('␣');  out_str(m→tex_part);  finish_line();
}
```

279.   ⟨Predeclaration of procedures 11⟩ +≡   **static void** *out_mini*(**meaning_struct** *∗*);

---

abnormal = macro (), §23.
byte_start: **char** *∗*, §13.
cur_meaning: **struct**
   perm_meaning[ ], §28.
cur_xref: **static xref_pointer**,
   §92.
false, <stdbool.h>.
finish_line: **static void** (), §99.
flag: **sixteen_bits**, §272.
flush_buffer: **static void** (),
   §98.
id: **name_pointer**, §27.
link: **struct perm_meaning**
   ∗, §28.
meaning_struct = **struct**,
   §27.
name_dir: **name_info** [ ],

COMMON.W §43.
name_pointer = **name_info**
   ∗, §13.
num: **sixteen_bits**, §25.
out = macro (), §101.
out_buf: **static char** [ ], §96.
out_section: **static void** (),
   §107.
out_str, §102.
p: **text_pointer**, §262.
perm: **meaning_struct**, §28.
perm_meaning: **struct**, §28.
prog_no: **sixteen_bits**, §27.
q: **xref_pointer**, §272.
sec_no: **sixteen_bits**, §27.
section_count: **sixteen_bits**,

COMMON.W §37.
sprintf, <stdio.h>.
stamp: **int**, §28.
strcmp, <string.h>.
temp_meaning_ptr: **static**
   meaning_struct *∗*, §28.
temp_meaning_stack: **static**
   meaning_struct [ ], §28.
tex_part: **char** [ ], §27.
title_code: **static**
   name_pointer [ ], §28.
top_usage: **static struct**
   perm_meaning *∗*, §28.
true, <stdbool.h>.
usage_sentinel = macro, §257.
xlink: **struct xref_info** *∗*, §25.

280. ⟨ Mini-output the name at *cur_name* 280 ⟩ ≡
  switch (*cur_name→ilk*) {
  case *normal*: **case** *func_template*:
    if (*length*(*cur_name*) ≡ 1) *out_str*("\\|");
    else {
      char *j;
      for (*j* ← *cur_name→byte_start*; *j* < (*cur_name* + 1)→*byte_start*; *j*++)
        if (*xislower*(*j*)) **goto** *lowcase*;
      out_str("\\."); **break**;
    lowcase: *out_str*("\\\\");
    }
    break;
  case *roman*: **break**;
  case *wildcard*: *out_str*("\\9"); **break**;
  case *typewriter*: *out_str*("\\."); **break**;
  case *custom*:
    {
      char *j;
      out_str("$\\");
      for (*j* ← *cur_name→byte_start*; *j* < (*cur_name* + 1)→*byte_start*; *j*++)
        out(*j* ≡ '_' ? 'x' : *j* ≡ '$' ? 'X' : *j*);
      out('$'); **goto** *name_done*;
    }
  default: *out_str*("\\&");
  }
  out_name(*cur_name*, *true*); *name_done*:
This code is used in section 278.

281. Phase three processing. We are nearly finished! `CWEAVE`'s only remaining task is to write out the index, after sorting the identifiers and index entries.

If the user has set the *no_xref* flag (the `-x` option on the command line), just finish off the page, omitting the index, section name list, and table of contents.

> **static void** *phase_three*(**void**)
> {
>   **if** (*no_xref*) {
>   *finish_line*(); *out_str*("\\end"); *active_file* ← *tex_file*;
>   }
>   **else** {
>   *phase* ← 3;
>   **if** (*show_progress*) *fputs*(_("\nWriting␣the␣index..."), *stdout*);
>   *finish_line*();
>   **if** ((*idx_file* ← *fopen*(*idx_file_name*, "wb")) ≡ Λ)
>   *fatal*(_("!␣Cannot␣open␣index␣file␣"), *idx_file_name*);
>   *out_str*("\\inx"); *finish_line*(); *active_file* ← *idx_file*;
>   ▷ change active file to the index file ◁
>   ⟨ Do the first pass of sorting 285 ⟩
>   ⟨ Sort and output the index 293 ⟩
>   *finish_line*(); *fclose*(*active_file*);   ▷ finished with *idx_file* ◁
>   *active_file* ← *tex_file*;   ▷ switch back to *tex_file* for a tic ◁
>   *out_str*("\\fin"); *finish_line*();
>   **if** ((*scn_file* ← *fopen*(*scn_file_name*, "wb")) ≡ Λ)
>   *fatal*(_("!␣Cannot␣open␣section␣file␣"), *scn_file_name*);
>   *active_file* ← *scn_file*;   ▷ change active file to section listing file ◁
>   ⟨ Output all the section names 302 ⟩
>   *finish_line*(); *fclose*(*active_file*);   ▷ finished with *scn_file* ◁
>   *active_file* ← *tex_file*;
>   **if** (*group_found*) *out_str*("\\con"); **else** *out_str*("\\end");
>   }
>   *finish_line*(); *fclose*(*active_file*); *active_file* ← Λ;
>   ⟨ Update the result when it has changed 307 ⟩
>   **if** (*show_happiness*) {
>   **if** (*show_progress*) *new_line*;
>   *fputs*(_("Done."), *stdout*);
>   }
>   *check_complete*();   ▷ was all of the change file used? ◁
> }

282. ⟨ Predeclaration of procedures 11 ⟩ +≡ **static void** *phase_three*(**void**);

283. Just before the index comes a list of all the changed sections, including the index section itself—NOT!

284. A left-to-right radix sorting method is used, since this makes it easy to adjust the collating sequence and since the running time will be at worst proportional to the total length of all entries in the index. We put the identifiers into different lists based on their first characters. (Uppercase letters are put into the same list as the corresponding lowercase letters, since we want to have '*t* < *TeX* < **to**'.) The list for character *c* begins at location *bucket*[*c*] and continues through the *blink* array.

⟨ Private variables 26 ⟩ +≡
  static name_pointer *bucket*[256];
  static name_pointer *next_name*;   ▷ successor of *cur_name* when sorting ◁
  static name_pointer *blink*[*max_names*];   ▷ links in the buckets ◁

285.   To begin the sorting, we go through all the hash lists and put each entry
having a nonempty cross-reference list into the proper bucket.

⟨ Do the first pass of sorting 285 ⟩ ≡
  {
    int *c*;
    for (*c* ← 0; *c* < 256; *c*++) *bucket*[*c*] ← Λ;
    for (*h* ← *hash*; *h* ≤ *hash_end*; *h*++) {
      next_name ← ∗*h*;
      while (*next_name*) {
        cur_name ← *next_name*; *next_name* ← *cur_name*→*link*;
        if (*cur_name*→*xref* ≠ (**void** ∗) *xmem*) {
          c ← (**eight_bits**)((*cur_name*→*byte_start*)[0]);
          if (*xisupper*(*c*)) *c* ← *tolower*(*c*);
          blink[*cur_name* − *name_dir*] ← *bucket*[*c*]; *bucket*[*c*] ← *cur_name*;
        }
      }
    }
  }

This code is used in section 281.

286.   During the sorting phase we shall use the *cat* and *trans* arrays from CWEAVE's
parsing algorithm and rename them *depth* and *head*. They now represent a stack of
identifier lists for all the index entries that have not yet been output. The variable
sort_ptr tells how many such lists are present; the lists are output in reverse order
(first *sort_ptr*, then *sort_ptr* − 1, etc.). The *j*th list starts at *head*[*j*], and if the first
k characters of all entries on this list are known to be equal we have *depth*[*j*] ≡ *k*.

⟨ Rest of *trans_plus* union 286 ⟩ ≡
  name_pointer *Head*;

This code is used in section 136.

_ = macro ( ), §7.
active_file: **FILE** ∗, COMMON.W §83.
byte_start: **char** ∗, §13.
cat: **eight_bits**, §136.
check_complete: **void** ( ), COMMON.W §42.
cur_name: **name_pointer**, §278.
depth = macro, §287.
eight_bits = **uint8_t**, §6.
fatal: **void** ( ), COMMON.W §70.
fclose, <stdio.h>.
finish_line: **static void** ( ), §99.
fopen, <stdio.h>.
fputs, <stdio.h>.
group_found: **static boolean**, §256.
h: **hash_pointer**,

COMMON.W §46.
hash: **name_pointer** [], COMMON.W §46.
hash_end: **hash_pointer**, COMMON.W §46.
head = macro, §287.
idx_file: **FILE** ∗, COMMON.W §83.
idx_file_name: **char** [], COMMON.W §73.
link: **struct perm_meaning** ∗, §28.
max_names = 10239, §20.
name_dir: **name_info** [], COMMON.W §43.
name_pointer = **name_info** ∗, §13.
new_line = macro, §18.
no_xref = macro, §35.

out_str, §102.
phase: **int**, COMMON.W §19.
scn_file: **FILE** ∗, COMMON.W §83.
scn_file_name: **char** [], COMMON.W §73.
show_happiness = macro, §17.
show_progress = macro, §17.
sort_ptr = macro, §287.
stdout, <stdio.h>.
tex_file: **FILE** ∗, COMMON.W §83.
tolower, <ctype.h>.
trans = macro, §137.
trans_plus: **union**, §136.
xisupper = macro ( ), §9.
xmem: **static xref_info** [], §26.
xref = macro, §34.

287.   #**define** *depth* *cat*   ▷ reclaims memory that is no longer needed for parsing ◁
#**define** *head* *trans_plus.Head*   ▷ ditto ◁
   format *sort_pointer* *int*
#**define** **sort_pointer** **scrap_pointer**   ▷ ditto ◁
#**define** *sort_ptr* *scrap_ptr*   ▷ ditto ◁
#**define** *max_sorts* *max_scraps*   ▷ ditto ◁

⟨ Private variables 26 ⟩ +≡
   static eight_bits *cur_depth*;   ▷ depth of current buckets ◁
   static char *∗cur_byte*;   ▷ index into *byte_mem* ◁
   static sixteen_bits *cur_val*;   ▷ current cross-reference number ◁
   static sort_pointer *max_sort_ptr*;   ▷ largest value of *sort_ptr* ◁

288.   ⟨ Set initial values 29 ⟩ +≡
   max_sort_ptr ← *scrap_info*;

289.   The desired alphabetic order is specified by the *collate* array; namely, *collate*[0]
< *collate*[1] < · · · < *collate*[100].

⟨ Private variables 26 ⟩ +≡
   static eight_bits *collate*[101 + 128];   ▷ collation order ◁

290.   We use the order null $< _\sqcup <$ other characters $< \_ < A = a < · · · < Z = z <$
$0 < · · · < 9$. Warning: The collation mapping needs to be changed if ASCII code is
not being used.
   We initialize *collate* by copying a few characters at a time, because some C compilers
choke on long strings.

⟨ Set initial values 29 ⟩ +≡
   collate[0] ← 0;
   memcpy((**char** ∗) *collate* + 1, "␣\1\2\3\4\5\6\7\10\11\12\13\14\15\16\17", 16);
      ▷ 16 characters + 1 = 17 ◁
   memcpy((**char** ∗) *collate* + 17,
      "\20\21\22\23\24\25\26\27\30\31\32\33\34\35\36\37", 16);
      ▷ 16 characters + 17 = 33 ◁
   memcpy((**char** ∗) *collate* + 33, "!\42#$%&'()∗+,−./:;<=>?@[\\]^'{|}~_", 32);
      ▷ 32 characters + 33 = 65 ◁
   memcpy((**char** ∗) *collate* + 65, "abcdefghijklmnopqrstuvwxyz0123456789", 36);
      ▷ (26 + 10) characters + 65 = 101 ◁
   memcpy((**char** ∗) *collate* + 101,
      "\200\201\202\203\204\205\206\207\210\211\212\213\214\215\216\217", 16);
      ▷ 16 characters + 101 = 117 ◁
   memcpy((**char** ∗) *collate* + 117,
      "\220\221\222\223\224\225\226\227\230\231\232\233\234\235\236\237", 16);
      ▷ 16 characters + 117 = 133 ◁
   memcpy((**char** ∗) *collate* + 133,
      "\240\241\242\243\244\245\246\247\250\251\252\253\254\255\256\257", 16);
      ▷ 16 characters + 133 = 149 ◁
   memcpy((**char** ∗) *collate* + 149,
      "\260\261\262\263\264\265\266\267\270\271\272\273\274\275\276\277", 16);
      ▷ 16 characters + 149 = 165 ◁

$memcpy(($**char** $*)$ $collate + 165,$
        "\300\301\302\303\304\305\306\307\310\311\312\313\314\315\316\317", 16$);$
    ▷ 16 characters + 165 = 181 ◁
$memcpy(($**char** $*)$ $collate + 181,$
        "\320\321\322\323\324\325\326\327\330\331\332\333\334\335\336\337", 16$);$
    ▷ 16 characters + 181 = 197 ◁
$memcpy(($**char** $*)$ $collate + 197,$
        "\340\341\342\343\344\345\346\347\350\351\352\353\354\355\356\357", 16$);$
    ▷ 16 characters + 197 = 213 ◁
$memcpy(($**char** $*)$ $collate + 213,$
        "\360\361\362\363\364\365\366\367\370\371\372\373\374\375\376\377", 16$);$
    ▷ 16 characters + 213 = 229 ◁

291.   Procedure *unbucket* goes through the buckets and adds nonempty lists to the stack, using the collating sequence specified in the *collate* array. The parameter to *unbucket* tells the current depth in the buckets. Any two sequences that agree in their first 255 character positions are regarded as identical.

#**define** *infinity* 255   ▷ ∞ (approximately) ◁
  static void *unbucket* (▷ empties buckets having depth d ◁
      eight_bits $d)$
  {
    int $c;$
      ▷ index into *bucket*; cannot be a simple **char** because of sign comparison below ◁

    for $(c \leftarrow 100 + 128;\ c \geq 0;\ c\!-\!-)$
      if $(bucket[collate[c]])$ {
        if $(sort\_ptr \geq scrap\_info\_end)$ $overflow\,($ ("sorting"));
        $sort\_ptr\!+\!+;$
        if $(sort\_ptr > max\_sort\_ptr)$ $max\_sort\_ptr \leftarrow sort\_ptr;$
        if $(c \equiv 0)$ $sort\_ptr\text{→}depth \leftarrow infinity;$
        else $sort\_ptr\text{→}depth \leftarrow d;$
        $sort\_ptr\text{→}head \leftarrow bucket[collate[c]];$ $bucket[collate[c]] \leftarrow \Lambda;$
      }
  }

292.   ⟨ Predeclaration of procedures 11 ⟩ +≡ **static void** *unbucket* (**eight_bits**);

---

293. ⟨Sort and output the index 293⟩ ≡
  $sort\_ptr \leftarrow scrap\_info$; $unbucket(1)$;
  while ($sort\_ptr > scrap\_info$) {
    $cur\_depth \leftarrow sort\_ptr{\rightarrow}depth$;
    if ($blink[sort\_ptr{\rightarrow}head - name\_dir] \equiv 0 \lor cur\_depth \equiv infinity$)
      ⟨Output index entries for the list at $sort\_ptr$ 295⟩
    else ⟨Split the list at $sort\_ptr$ into further lists 294⟩
  }

This code is used in section 281.

294. ⟨Split the list at $sort\_ptr$ into further lists 294⟩ ≡
  {
    eight_bits c;

    $next\_name \leftarrow sort\_ptr{\rightarrow}head$; **do** {
      $cur\_name \leftarrow next\_name$; $next\_name \leftarrow blink[cur\_name - name\_dir]$;
      $cur\_byte \leftarrow cur\_name{\rightarrow}byte\_start + cur\_depth$;
      if ($cur\_byte \equiv (cur\_name + 1){\rightarrow}byte\_start$) $c \leftarrow 0$;   ▷ hit end of the name ◁
      else {
        $c \leftarrow$ (**eight_bits**) $*cur\_byte$;
        if ($xisupper(c)$) $c \leftarrow tolower(c)$;
      }
      $blink[cur\_name - name\_dir] \leftarrow bucket[c]$; $bucket[c] \leftarrow cur\_name$;
    } **while** ($next\_name$); $-\!\!-sort\_ptr$; $unbucket(cur\_depth + 1)$;
  }

This code is used in section 293.

295. ⟨Output index entries for the list at $sort\_ptr$ 295⟩ ≡
  {
    $cur\_name \leftarrow sort\_ptr{\rightarrow}head$; **do** {
      $out\_str("\backslash\backslash I")$; ⟨Output the name at cur_name 296⟩
      ⟨Output the cross-references at cur_name 297⟩
      $cur\_name \leftarrow blink[cur\_name - name\_dir]$;
    } **while** (cur_name); $-\!\!-sort\_ptr$;
  }

This code is used in section 293.

296. We don't format the index completely; the `twinx` program does the rest of the job.

⟨Output the name at cur_name 296⟩ ≡
  switch ($cur\_name{\rightarrow}ilk$) {
  case $normal$:
    if ($is\_tiny(cur\_name)$) $out\_str("\backslash\backslash|")$;
    else { **char** $*j$;
      for ($j \leftarrow cur\_name{\rightarrow}byte\_start$; $j < (cur\_name + 1){\rightarrow}byte\_start$; $j{+}{+}$)
        if ($xislower(*j)$) **goto** $lowcase$;
      $out\_str("\backslash\backslash.")$; **break**;
    $lowcase$: $out\_str("\backslash\backslash\backslash\backslash")$;
    }
    break;

```
case roman: out_str("␣␣"); goto not_an_identifier;
case wildcard: out_str("\\9"); goto not_an_identifier;
case typewriter: out_str("\\.");
not_an_identifier: out_name(cur_name, false); goto name_done;
case custom: out_str("\\$"); break;
default: out_str("\\&");
}
if (proofing) out_name(cur_name, true);
else {
  out('{');
  {
    char *j;
    for (j ← cur_name→byte_start; j < (cur_name + 1)→byte_start; j++) out(*j);
  }
  out('}');
}
name_done:
```

This code is used in section 295.

297. Section numbers that are to be underlined are enclosed in '\[...]'.

⟨Output the cross-references at *cur_name* 297⟩ ≡
  ⟨Invert the cross-reference list at *cur_name*, making *cur_xref* the head 299⟩
```
  do {
    out_str(",␣"); cur_val ← cur_xref→num;
    if (cur_val < def_flag) out_section(cur_val);
    else {
      out_str("\\["); out_section(cur_val − def_flag); out(']');
    }
    cur_xref ← cur_xref→xlink;
  } while (cur_xref ≠ xmem); out('.'); finish_line();
```
This code is used in section 295.

---

298.   List inversion is best thought of as popping elements off one stack and pushing them onto another. In this case *cur_xref* will be the head of the stack that we push things onto.

⟨ Private variables 26 ⟩ +≡
  static xref_pointer *next_xref*, *this_xref*;   ▷ pointer variables for rearranging a list ◁

299.   ⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 299 ⟩ ≡
  this_xref ← (**xref_pointer**) *cur_name*→*xref*; *cur_xref* ← *xmem*; **do** {
    next_xref ← *this_xref*→*xlink*; *this_xref*→*xlink* ← *cur_xref*; *cur_xref* ← *this_xref*;
    this_xref ← *next_xref*;
  } **while** (*this_xref* ≠ *xmem*);
This code is used in section 297.

300.   The following recursive procedure walks through the tree of section names and prints them.
  static void *section_print*(▷ print all section names in subtree *p* ◁
    name_pointer *p*)
  {
    if (*p*) {
      section_print(*p*→*llink*); *out_str*("\\I"); *tok_ptr* ← *tok_mem* + 1;
      text_ptr ← *tok_start* + 1; *scrap_ptr* ← *scrap_info*; *init_stack*;
      app(*p* − *name_dir* + *section_flag*); *make_output*( ); *footnote*(*cite_flag*); *footnote*(0);
        ▷ *cur_xref* was set by *make_output* ◁
      finish_line( );
      section_print(*p*→*rlink*);
    }
  }

301.   ⟨ Predeclaration of procedures 11 ⟩ +≡   **static void** *section_print*(**name_pointer**);

302.   ⟨ Output all the section names 302 ⟩ ≡
  section_print(*root*);
This code is used in section 281.

303. Because on some systems the difference between two pointers is a **ptrdiff_t**
rather than an **int**, we use %ld to print these quantities.

void *print_stats* (**void**)
{
 puts (("\nMemory␣usage␣statistics:"));
 printf (("%ld␣names␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*name_ptr* − *name_dir*), (**long**)
  max_names); *printf* (("%ld␣cross-references␣(out␣of␣%ld)\n"),
  (**ptrdiff_t**)(*xref_ptr* − *xmem*), (**long**) *max_refs*);
 printf (("%ld␣bytes␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*byte_ptr* − *byte_mem*), (**long**)
  max_bytes); *printf* (("%ld␣temp␣meanings␣(out␣of␣%ld)\n"),
  (**ptrdiff_t**)(*max_temp_meaning_ptr* − *temp_meaning_stack*), (**long**) *max_meanings*);
 printf (("%ld␣titles␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*title_code_ptr* − *title_code*), (**long**)
  max_titles); *puts* (("Parsing:"));
 printf (("%ld␣scraps␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*max_scr_ptr* − *scrap_info*), (**long**)
  max_scraps);
 printf (("%ld␣texts␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*max_text_ptr* − *tok_start*), (**long**)
  max_texts);
 printf (("%ld␣tokens␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*max_tok_ptr* − *tok_mem*), (**long**)
  max_toks);
 printf (("%ld␣levels␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*max_stack_ptr* − *stack*), (**long**)
  stack_size); *puts* (("Sorting:"));
 printf (("%ld␣levels␣(out␣of␣%ld)\n"), (**ptrdiff_t**)(*max_sort_ptr* − *scrap_info*), (**long**)
  max_scraps);
}

304. Extensions to CWEB.   The following sections introduce new or improved features that have been created by numerous contributors over the course of a quarter century.

305. Formatting alternatives.   CWEAVE indents declarations after old-style func-
tion definitions. With the `-i` option they will come out flush left. You won't see any
difference if you use ANSI-style function definitions.

#define *indent_param_decl*   *flags*['i']
            ▷ should formal parameter declarations be indented? ◁

⟨ Set initial values 29 ⟩ +≡
  indent_param_decl ← *true*;

306.   The original manual described the `-o` option for CWEAVE, but this was not yet
present. Here is a simple implementation. The purpose is to suppress the extra space
between local variable declarations and the first statement in a function block.

#define *order_decl_stmt*   *flags*['o']
            ▷ should declarations and statements be separated? ◁

⟨ Set initial values 29 ⟩ +≡
  order_decl_stmt ← *true*;

---

flags: **boolean** [],   COMMON.W §73.   *true*, `<stdbool.h>`.

307. Output file update. Most C projects are controlled by a `Makefile` that automatically takes care of the temporal dependecies between the different source modules. It is suitable that `CWEB` doesn't create new output for all existing files, when there are only changes to some of them. Thus the `make` process will only recompile those modules where necessary. The idea and basic implementation of this mechanism can be found in the program `NUWEB` by Preston Briggs, to whom credit is due.

⟨ Update the result when it has changed 307 ⟩ ≡
  if $((\textit{tex\_file} \leftarrow \textit{fopen}(\textit{tex\_file\_name}, \texttt{"r"})) \neq \Lambda)$ {
    char $x[\texttt{BUFSIZ}]$, $y[\texttt{BUFSIZ}]$;
    int x_size, y_size, $comparison \leftarrow false$;
    if $((\textit{check\_file} \leftarrow \textit{fopen}(\textit{check\_file\_name}, \texttt{"r"})) \equiv \Lambda)$
      $\textit{fatal}(\_(\texttt{"!}\_\texttt{Cannot}\_\texttt{open}\_\texttt{output}\_\texttt{file}\_\texttt{"}), \textit{check\_file\_name})$;
    if $(\textit{temporary\_output})$ ⟨ Compare the temporary output to the previous output 308 ⟩
    $\textit{fclose}(\textit{tex\_file})$; $\textit{tex\_file} \leftarrow \Lambda$; $\textit{fclose}(\textit{check\_file})$; $\textit{check\_file} \leftarrow \Lambda$;
    ⟨ Take appropriate action depending on the comparison 309 ⟩
  }
  else $\textit{rename}(\textit{check\_file\_name}, \textit{tex\_file\_name})$;   ▷ This was the first run ◁
  $\textit{strcpy}(\textit{check\_file\_name}, \texttt{""})$;   ▷ We want to get rid of the temporary file ◁
This code is used in section 281.

308. We hope that this runs fast on most systems.

⟨ Compare the temporary output to the previous output 308 ⟩ ≡
  do {
    $x\_size \leftarrow \textit{fread}(x, 1, \texttt{BUFSIZ}, \textit{tex\_file})$; $y\_size \leftarrow \textit{fread}(y, 1, \texttt{BUFSIZ}, \textit{check\_file})$;
    $comparison \leftarrow (x\_size \equiv y\_size)$;   ▷ Do not merge these statements! ◁
    if $(comparison)$ $comparison \leftarrow \neg \textit{memcmp}(x, y, x\_size)$;
  } **while** $(comparison \wedge \neg \textit{feof}(\textit{tex\_file}) \wedge \neg \textit{feof}(\textit{check\_file}))$;
This code is used in section 307.

309. Note the superfluous call to *remove* before *rename*. We're using it to get around a bug in some implementations of *rename*.

⟨ Take appropriate action depending on the comparison 309 ⟩ ≡
  if $(comparison)$ $\textit{remove}(\textit{check\_file\_name})$;   ▷ The output remains untouched ◁
  else {
    $\textit{remove}(\textit{tex\_file\_name})$; $\textit{rename}(\textit{check\_file\_name}, \textit{tex\_file\_name})$;
  }
This code is used in section 307.

310.   Put "version" information in COMMON**.**   Don't do this at home, kids! Push
our local macro to the variable in COMMON for printing the *banner* and the *versionstring*
from there.

#**define** *max_banner*   50

⟨ Common code for CWEAVE and CTANGLE 6 ⟩ +≡
   extern char *cb_banner* [ ];

311.   ⟨ Set initial values 29 ⟩ +≡
   strncpy (*cb_banner* , *banner* , *max_banner* − 1);