The CWEAVE processor

(Version 4.3)

1.   Introduction.   This is the `CWEAVE` program by Silvio Levy and Donald E. Knuth, based on `WEAVE` by Knuth. We are thankful to Steve Avery, Nelson Beebe, Hans-Hermann Bode (to whom the original C++ adaptation is due), Klaus Guntermann, Norman Ramsey, Tomas Rokicki, Joachim Schnitter, Joachim Schrod, Lee Wittenberg, Saroj Mahapatra, Cesar Augusto Rorato Crusius, and others who have contributed improvements.

The "banner line" defined here should be changed whenever `CWEAVE` is modified.

#**define** *banner* `"This␣is␣CWEAVE␣(Version␣4.3)"`

⟨Include files 4⟩
⟨Preprocessor definitions⟩
⟨Common code for `CWEAVE` and `CTANGLE` 3⟩
⟨Typedef declarations 22⟩
⟨Private variables 21⟩
⟨Predeclaration of procedures 8⟩

2.   `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 ← cweave;
  ⟨Set initial values 24⟩
  common_init( );
  ⟨Start TEX output 89⟩
  if (show_banner) puts(banner);      /* print a "banner line" */
  ⟨Store all the reserved words 34⟩
  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 ≡ fully ∧ ¬show_progress) new_line;
  return wrap_up( );      /* and exit gracefully */
}
```

3.   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:

#**define** *ctangle false*
#**define** *cweave true*

⟨Common code for `CWEAVE` and `CTANGLE` 3⟩ ≡
  typedef bool boolean;
  typedef uint8_t eight_bits;
  typedef uint16_t sixteen_bits;
  extern boolean *program*;   /* CWEAVE or CTANGLE? */
  extern int *phase*;   /* which phase are we in? */

See also sections 5, 6, 7, 9, 10, 12, 14, and 15.

This code is used in section 1.

4.   Interface to the standard C library:

⟨ Include files 4 ⟩ ≡
#**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.

5.   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 ⇆ ∗/

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

⟨ Common code for CWEAVE and CTANGLE 3 ⟩ +≡
  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 ∗/

6.   Code related to input routines:

#**define** *xisalpha*(*c*) (*isalpha*((**eight_bits**)(*c*)) ∧ ((**eight_bits**)(*c*) < °200))
#**define** *xisdigit*(*c*) (*isdigit*((**eight_bits**)(*c*)) ∧ ((**eight_bits**)(*c*) < °200))
#**define** *xisspace*(*c*) (*isspace*((**eight_bits**)(*c*)) ∧ ((**eight_bits**)(*c*) < °200))
#**define** *xislower*(*c*) (*islower*((**eight_bits**)(*c*)) ∧ ((**eight_bits**)(*c*) < °200))
#**define** *xisupper*(*c*) (*isupper*((**eight_bits**)(*c*)) ∧ ((**eight_bits**)(*c*) < °200))
#**define** *xisxdigit*(*c*) (*isxdigit*((**eight_bits**)(*c*)) ∧ ((**eight_bits**)(*c*) < °200))
#**define** *isxalpha*(*c*) ((*c*) ≡ '_' ∨ (*c*) ≡ '$')   /∗ non-alpha characters allowed in identifier ∗/
#**define** *ishigh*(*c*) ((**eight_bits**)(*c*) > °177)

⟨ Common code for CWEAVE and CTANGLE 3 ⟩ +≡
  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 ∗/

7.   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 3 ⟩ +≡
   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 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 ∗/

8.   ⟨ Predeclaration of procedures 8 ⟩ ≡
   extern boolean *get_line*(**void**);   /∗ inputs the next line ∗/
   extern void *check_complete*(**void**);   /∗ checks that all changes were picked up ∗/
   extern void *reset_input*(**void**);   /∗ initialize to read the web file and change file ∗/

See also sections 11, 13, 16, 25, 33, 40, 45, 65, 69, 71, 83, 86, 90, 95, 98, 109, 117, 120, 123, 182, 190, 195, 202, 211, 215, 229,
    236, 245, 249, 260, and 269.

This code is used in section 1.

9.   Code related to section numbers:

⟨ Common code for CWEAVE and CTANGLE 3 ⟩ +≡
   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 ∗/

10.   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 3 ⟩ +≡
  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_info**s ∗/
  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 ∗/

11.   ⟨ Predeclaration of procedures 8 ⟩ +≡
  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** ∗, **boolean**);   /∗ 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**);

12.   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 3 ⟩ +≡
  extern int *history*;   /∗ indicates how bad this run was ∗/

13.   ⟨Predeclaration of procedures 8⟩ +≡
  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 ∗/

14.   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_happiness* *flags*['h']   /∗ should lack of errors be announced? ∗/
#**define** *show_stats* *flags*['s']   /∗ should statistics be printed at end of run? ∗/
#**define** *make_xrefs* *flags*['x']   /∗ should cross references be output? ∗/
⟨Common code for CWEAVE and CTANGLE 3⟩ +≡
  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 ∗/

15.   Code related to output:
#**define** *update_terminal* *fflush*(*stdout*)   /∗ empty the terminal output buffer ∗/
#**define** *new_line* *putchar*('\n')
#**define** *term_write*(*a, b*) *fflush*(*stdout*), *fwrite*(*a*, **sizeof**(**char**), *b*, *stdout*)
⟨Common code for CWEAVE and CTANGLE 3⟩ +≡
  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 ∗/

16.   The procedure that gets everything rolling:
⟨Predeclaration of procedures 8⟩ +≡
  extern void *common_init*(**void**);
  extern void *print_stats*(**void**);

17.   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 ∗/

18.   End of `COMMON` interface.

19.   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_scraps* 10000 /∗ number of tokens in C texts being parsed ∗/

20.   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, ..., *attr* 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** ∗/
#**define** *alignas_like* 59   /∗ **alignas** ∗/
#**define** *using_like* 60   /∗ **using** ∗/
#**define** *default_like* 61   /∗ **default** ∗/
#**define** *attr* 62   /∗ **noexcept** and attributes ∗/

21.   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.

⟨ Private variables 21 ⟩ ≡
  static boolean *change_exists*;   /∗ has any section changed? ∗/

See also sections 23, 30, 37, 43, 46, 48, 67, 76, 81, 85, 106, 114, 119, 187, 209, 214, 230, 239, 250, 252, 255, 257, and 266.

This code is used in section 1.

22.   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 22 ⟩ ≡
  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 29, 113, and 208.

This code is used in section 1.

23.   ⟨ Private variables 21 ⟩ +≡
  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* ∗/

24.   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 24 ⟩ ≡
  xref_ptr ← *xmem*;
  init_node(*name_dir*);
  xref_switch ← *section_xref_switch* ← 0;
  xmem{\rightarrow}*num* ← 0;   /∗ sentinel value ∗/

See also sections 31, 38, 61, 92, 107, 115, 156, 205, 210, 256, and 258.

This code is used in section 2.

25.   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 115 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) (*length*(p) ≡ 1)
#**define** *unindexed*(a) (a < *res_wd_end* ∧ a→*ilk* ≥ *custom*)
    /∗ tells if uses of a name are to be indexed ∗/

⟨ Predeclaration of procedures 8 ⟩ +≡
  static void *new_xref* (**name_pointer**);
  static void *new_section_xref* (**name_pointer**);
  static void *set_file_flag* (**name_pointer**);

26.   **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**) p→*xref*;
   if (q ≠ *xmem*) {
    n ← q→*num*;
    if (n ≡ m ∨ n ≡ m + *def_flag*) **return**;
    else if (m ≡ n + *def_flag*) {
     q→*num* ← m;
     return;
    }
   }
   append_xref (m);
   xref_ptr→*xlink* ← q;
   update_node (p);
  }

27.   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 ← (xref_pointer) p⃗xref ;
  xref_pointer r ← xmem;      /∗ pointers to previous cross-references ∗/
  if (q > r)
    while (q⃗num > section_xref_switch) {
      r ← q;
      q ← q⃗xlink ;
    }
  if (r⃗num ≡ section_count + section_xref_switch) return;      /∗ don't duplicate entries ∗/
  append_xref (section_count + section_xref_switch);
  xref_ptr⃗xlink ← q;
  section_xref_switch ← 0;
  if (r ≡ xmem) update_node (p);
  else r⃗xlink ← xref_ptr ;
}
```

28.   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 (name_pointer p)
{
  xref_pointer q ← (xref_pointer) p⃗xref ;
  if (q⃗num ≡ file_flag) return;
  append_xref (file_flag);
  xref_ptr⃗xlink ← q;
  update_node (p);
}
```

29.   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 22 ⟩ +≡
  typedef sixteen_bits token;
  typedef token ∗token_pointer;
  typedef token_pointer ∗text_pointer;

30.   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 $*text\_ptr \equiv tok\_ptr$.

⟨ Private variables 21 ⟩ +≡
  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* ∗/

31.   ⟨ Set initial values 24 ⟩ +≡
  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;

32.   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*)
  {
    p⃗*ilk* ← *t*;
    init_node(*p*);
  }
  void *init_node*(**name_pointer** *p*)
  {
    p⃗*xref* ← (**void** ∗) *xmem*;
  }
  static void *update_node*(**name_pointer** *p*)
  {
    p⃗*xref* ← (**void** ∗) *xref_ptr*;
  }

33.   ⟨ Predeclaration of procedures 8 ⟩ +≡ **static void** *update_node*(**name_pointer** *p*);

34. We have to get C's and 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. An ever growing list of C++ keywords can be found here: https://en.cppreference.com/w/cpp/keyword.)

⟨ Store all the reserved words 34 ⟩ ≡
  id_lookup("alignas", Λ, *alignas_like*);
  id_lookup("alignof", Λ, *sizeof_like*);
  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("char8_t", Λ, *raw_int*);
  id_lookup("char16_t", Λ, *raw_int*);
  id_lookup("char32_t", Λ, *raw_int*);
  id_lookup("class", Λ, *struct_like*);
  id_lookup("clock_t", Λ, *raw_int*);
  id_lookup("compl", Λ, *alfop*);
  id_lookup("concept", Λ, *int_like*);
  id_lookup("const", Λ, *const_like*);
  id_lookup("consteval", Λ, *const_like*);
  id_lookup("constexpr", Λ, *const_like*);
  id_lookup("constinit", Λ, *const_like*);
  id_lookup("const_cast", Λ, *raw_int*);
  id_lookup("continue", Λ, *case_like*);
  id_lookup("co_await", Λ, *case_like*);
  id_lookup("co_return", Λ, *case_like*);
  id_lookup("co_yield", Λ, *case_like*);
  id_lookup("decltype", Λ, *sizeof_like*);
  id_lookup("default", Λ, *default_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*);
  id_lookup("extern", Λ, *int_like*);
  id_lookup("FILE", Λ, *raw_int*);

$id\_lookup\,("\texttt{false}", \Lambda, normal\,);$
$id\_lookup\,("\texttt{float}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{for}", \Lambda, for\_like\,);$
$id\_lookup\,("\texttt{fpos\_t}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{friend}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{goto}", \Lambda, case\_like\,);$
$id\_lookup\,("\texttt{if}", \Lambda, if\_like\,);$
$id\_lookup\,("\texttt{ifdef}", \Lambda, if\_like\,);$
$id\_lookup\,("\texttt{ifndef}", \Lambda, if\_like\,);$
$id\_lookup\,("\texttt{include}", \Lambda, if\_like\,);$
$id\_lookup\,("\texttt{inline}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{int}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{jmp\_buf}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{ldiv\_t}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{line}", \Lambda, if\_like\,);$
$id\_lookup\,("\texttt{long}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{mutable}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{namespace}", \Lambda, struct\_like\,);$
$id\_lookup\,("\texttt{new}", \Lambda, new\_like\,);$
$id\_lookup\,("\texttt{noexcept}", \Lambda, attr\,);$
$id\_lookup\,("\texttt{not}", \Lambda, alfop\,);$
$id\_lookup\,("\texttt{not\_eq}", \Lambda, alfop\,);$
$id\_lookup\,("\texttt{NULL}", \Lambda, custom\,);$
$id\_lookup\,("\texttt{nullptr}", \Lambda, custom\,);$
$id\_lookup\,("\texttt{offsetof}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{operator}", \Lambda, operator\_like\,);$
$id\_lookup\,("\texttt{or}", \Lambda, alfop\,);$
$id\_lookup\,("\texttt{or\_eq}", \Lambda, alfop\,);$
$id\_lookup\,("\texttt{pragma}", \Lambda, if\_like\,);$
$id\_lookup\,("\texttt{private}", \Lambda, public\_like\,);$
$id\_lookup\,("\texttt{protected}", \Lambda, public\_like\,);$
$id\_lookup\,("\texttt{ptrdiff\_t}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{public}", \Lambda, public\_like\,);$
$id\_lookup\,("\texttt{register}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{reinterpret\_cast}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{requires}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{restrict}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{return}", \Lambda, case\_like\,);$
$id\_lookup\,("\texttt{short}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{sig\_atomic\_t}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{signed}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{size\_t}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{sizeof}", \Lambda, sizeof\_like\,);$
$id\_lookup\,("\texttt{static}", \Lambda, int\_like\,);$
$id\_lookup\,("\texttt{static\_assert}", \Lambda, sizeof\_like\,);$
$id\_lookup\,("\texttt{static\_cast}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{struct}", \Lambda, struct\_like\,);$
$id\_lookup\,("\texttt{switch}", \Lambda, for\_like\,);$
$id\_lookup\,("\texttt{template}", \Lambda, template\_like\,);$
$id\_lookup\,("\texttt{this}", \Lambda, custom\,);$
$id\_lookup\,("\texttt{thread\_local}", \Lambda, raw\_int\,);$
$id\_lookup\,("\texttt{throw}", \Lambda, case\_like\,);$

$id\_lookup(\texttt{"time\_t"}, \Lambda, raw\_int)$;
$id\_lookup(\texttt{"true"}, \Lambda, normal)$;
$id\_lookup(\texttt{"try"}, \Lambda, else\_like)$;
$id\_lookup(\texttt{"typedef"}, \Lambda, typedef\_like)$;
$id\_lookup(\texttt{"typeid"}, \Lambda, sizeof\_like)$;
$id\_lookup(\texttt{"typename"}, \Lambda, struct\_like)$;
$id\_lookup(\texttt{"undef"}, \Lambda, if\_like)$;
$id\_lookup(\texttt{"union"}, \Lambda, struct\_like)$;
$id\_lookup(\texttt{"unsigned"}, \Lambda, raw\_int)$;
$id\_lookup(\texttt{"using"}, \Lambda, using\_like)$;
$id\_lookup(\texttt{"va\_dcl"}, \Lambda, decl)$;   /∗ Berkeley's variable-arg-list convention ∗/
$id\_lookup(\texttt{"va\_list"}, \Lambda, raw\_int)$;   /∗ ditto ∗/
$id\_lookup(\texttt{"virtual"}, \Lambda, int\_like)$;
$id\_lookup(\texttt{"void"}, \Lambda, raw\_int)$;
$id\_lookup(\texttt{"volatile"}, \Lambda, const\_like)$;
$id\_lookup(\texttt{"wchar\_t"}, \Lambda, raw\_int)$;
$id\_lookup(\texttt{"while"}, \Lambda, for\_like)$;
$id\_lookup(\texttt{"xor"}, \Lambda, alfop)$;
$id\_lookup(\texttt{"xor\_eq"}, \Lambda, alfop)$; $res\_wd\_end \leftarrow name\_ptr$;
$id\_lookup(\texttt{"TeX"}, \Lambda, custom)$;
$id\_lookup(\texttt{"complex"}, \Lambda, int\_like)$;
$id\_lookup(\texttt{"imaginary"}, \Lambda, int\_like)$;
$id\_lookup(\texttt{"make\_pair"}, \Lambda, func\_template)$;

This code is used in section 2.

35.   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.

36.   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 ASCII `STX` ∗/
#**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 `DEL` ∗/
#**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** *ord* °*207*   /∗ control code for '`@'`' ∗/
#**define** *join* °*210*   /∗ control code for '`@&`' ∗/
#**define** *thin_space* °*211*   /∗ control code for '`@,`' ∗/
#**define** *math_break* °*212*   /∗ control code for '`@|`' ∗/
#**define** *line_break* °*213*   /∗ control code for '`@/`' ∗/
#**define** *big_line_break* °*214*   /∗ control code for '`@#`' ∗/
#**define** *no_line_break* °*215*   /∗ control code for '`@+`' ∗/
#**define** *pseudo_semi* °*216*   /∗ control code for '`@;`' ∗/
#**define** *macro_arg_open* °*220*   /∗ control code for '`@[`' ∗/
#**define** *macro_arg_close* °*221*   /∗ control code for '`@]`' ∗/
#**define** *trace* °*222*   /∗ control code for '`@0`', '`@1`' and '`@2`' ∗/
#**define** *translit_code* °*223*   /∗ control code for '`@l`' ∗/
#**define** *output_defs_code* °*224*   /∗ control code for '`@h`' ∗/
#**define** *format_code* °*225*   /∗ control code for '`@f`' and '`@s`' ∗/
#**define** *definition* °*226*   /∗ control code for '`@d`' ∗/
#**define** *begin_C* °*227*   /∗ control code for '`@c`' ∗/
#**define** *section_name* °*230*   /∗ control code for '`@<`' ∗/
#**define** *new_section* °*231*   /∗ control code for '`@␣`' and '`@*`' ∗/

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

⟨ Private variables 21 ⟩ +≡
  static eight_bits *ccode*[256];   /∗ meaning of a char following `@` ∗/

38.   ⟨Set initial values 24⟩ +≡
  {
    int c;   /∗ must be **int** so the **for** loop will end ∗/
    for $(c \leftarrow 0;\ c < 256;\ c{++})\ ccode[c] \leftarrow ignore$;
  }
  $ccode['\textrm{\textvisiblespace}'] \leftarrow ccode['\backslash t'] \leftarrow ccode['\backslash n'] \leftarrow ccode['\backslash v'] \leftarrow ccode['\backslash r'] \leftarrow ccode['\backslash f'] \leftarrow ccode['*'] \leftarrow$
      $new\_section$;
  $ccode['@'] \leftarrow '@'$;   /∗ 'quoted' at sign ∗/
  $ccode['='] \leftarrow verbatim$;
  $ccode['d'] \leftarrow ccode['D'] \leftarrow definition$;
  $ccode['f'] \leftarrow ccode['F'] \leftarrow ccode['s'] \leftarrow ccode['S'] \leftarrow format\_code$;
  $ccode['c'] \leftarrow ccode['C'] \leftarrow ccode['p'] \leftarrow ccode['P'] \leftarrow begin\_C$;
  $ccode['t'] \leftarrow ccode['T'] \leftarrow T_{\!E}X\_string$;
  $ccode['l'] \leftarrow ccode['L'] \leftarrow translit\_code$;
  $ccode['q'] \leftarrow ccode['Q'] \leftarrow noop$;
  $ccode['h'] \leftarrow ccode['H'] \leftarrow output\_defs\_code$;
  $ccode['\&'] \leftarrow join$;
  $ccode['<'] \leftarrow ccode['('] \leftarrow section\_name$;
  $ccode['!'] \leftarrow underline$;
  $ccode['\verb|^|'] \leftarrow xref\_roman$;
  $ccode[':'] \leftarrow xref\_wildcard$;
  $ccode['.'] \leftarrow xref\_typewriter$;
  $ccode[','] \leftarrow thin\_space$;
  $ccode['|'] \leftarrow math\_break$;
  $ccode['/'] \leftarrow line\_break$;
  $ccode['\#'] \leftarrow big\_line\_break$;
  $ccode['+'] \leftarrow no\_line\_break$;
  $ccode[';'] \leftarrow pseudo\_semi$;
  $ccode['['] \leftarrow macro\_arg\_open$;
  $ccode[']'] \leftarrow macro\_arg\_close$;
  $ccode['\backslash''] \leftarrow ord$;
  ⟨Special control codes for debugging 39⟩

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

⟨Special control codes for debugging 39⟩ ≡
  $ccode['0'] \leftarrow ccode['1'] \leftarrow ccode['2'] \leftarrow trace$;
This code is used in section 38.

40.   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 8⟩ +≡
  static void $skip\_limbo$(**void**);
  static eight_bits $skip\_T_{\!E}X$(**void**);

41.   **static void** *skip_limbo*(**void**)
{
  while (*true*) {
    if (*loc* > *limit* ∧ *get_line*( ) ≡ *false*) **return**;
    ∗(*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 79⟩
    }
  }
}

42.   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**)   /∗ skip past pure TEX code ∗/
  {
    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*++)];
    }
  }

43.   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*, T_EX_string, *verbatim*: The values of *id_first* and *id_loc* will have been set to the beginning 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 `@(` instead 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 21 ⟩ +≡
   static name_pointer *cur_section*;   /∗ name of section just scanned ∗/
   static char *cur_section_char*;   /∗ the character just before that name ∗/

44.   As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise.

```
   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 50 ⟩
         if (loc > limit ∧ get_line( ) ≡ false) return new_section;
         c ← ∗(loc ++);
         if (xisdigit(c) ∨ c ≡ '.') ⟨ Get a constant 53 ⟩
         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 57 ⟩
         else if (isalpha(c) ∨ isxalpha(c) ∨ ishigh(c)) ⟨ Get an identifier 52 ⟩
         else if (c ≡ '@') ⟨ Get control code and possible section name 59 ⟩
         else if (xisspace(c)) continue;      /∗ ignore spaces and tabs ∗/
         if (c ≡ '#' ∧ loc ≡ buffer + 1) ⟨ Raise preprocessor flag 47 ⟩
      mistake: ⟨ Compress two-symbol operator 51 ⟩
         return c;
      }
   }
```

45.   ⟨ Predeclaration of procedures 8 ⟩ +≡   **static eight_bits** *get_next*(**void**);

46. 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* °*217*   /∗ ends a preprocessor command ∗/
⟨ Private variables 21 ⟩ +≡
  static boolean *preprocessing* ← *false*;   /∗ are we scanning a preprocessor command? ∗/

47. ⟨ Raise preprocessor flag 47 ⟩ ≡
  {
     preprocessing ← *true*;
     ⟨ Check if next token is **include** 49 ⟩
     return *left_preproc*;
  }
This code is used in section 44.

48. 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 21 ⟩ +≡
  static boolean *sharp_include_line* ← *false*;   /∗ are we scanning a #**include** line? ∗/

49. ⟨ Check if next token is **include** 49 ⟩ ≡
  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 47.

50. 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 50 ⟩ ≡
  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 44.

51.   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.

$\langle$ Compress two-symbol operator $51\,\rangle \equiv$
 switch (c) {
 case '/':
  if $(*loc \equiv$ '*') { $compress(begin\_comment)$; }
  else if $(*loc \equiv$ '/') $compress(begin\_short\_comment)$;
  break;
 case '+':
  if $(*loc \equiv$ '+') $compress(plus\_plus)$;
  break;
 case '−':
  if $(*loc \equiv$ '−') { $compress(minus\_minus)$; }
  else if $(*loc \equiv$ '>') {
   if $(*(loc + 1) \equiv$ '*') {
    $loc\,{+}{+}$; $compress(minus\_gt\_ast)$;
   }
   else $compress(minus\_gt)$;
  }
  break;
 case '.':
  if $(*loc \equiv$ '*') { $compress(period\_ast)$; }
  else if $(*loc \equiv$ '.' $\wedge *(loc + 1) \equiv$ '.') {
   $loc\,{+}{+}$; $compress(dot\_dot\_dot)$;
  }
  break;
 case ':':
  if $(*loc \equiv$ ':') $compress(colon\_colon)$;
  break;
 case '=':
  if $(*loc \equiv$ '=') $compress(eq\_eq)$;
  break;
 case '>':
  if $(*loc \equiv$ '=') { $compress(gt\_eq)$; }
  else if $(*loc \equiv$ '>') $compress(gt\_gt)$;
  break;
 case '<':
  if $(*loc \equiv$ '=') { $compress(lt\_eq)$; }
  else if $(*loc \equiv$ '<') $compress(lt\_lt)$;
  break;
 case '&':
  if $(*loc \equiv$ '&') $compress(and\_and)$;
  break;
 case '|':
  if $(*loc \equiv$ '|') $compress(or\_or)$;
  break;
 case '!':
  if $(*loc \equiv$ '=') $compress(non\_eq)$;
  break;
 }

This code is used in section 44.

52. ⟨Get an identifier 52⟩ ≡
```
  {
    id_first ← −−loc;
    do ++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 44.

53. 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.

On output, the ␣ that replaces ' in C++ literals will become "`\␣`".

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

```
#define gather_digits_while(t)
        while (t ∨ *loc ≡ '\'')
          if (*loc ≡ '\'') {        /* C++-style digit separator */
            *id_loc++ ← '␣';
            loc++;        /* insert a little bit of space */
          } else *id_loc++ ← *loc++;
```
⟨Get a constant 53⟩ ≡
```
  {
    id_first ← id_loc ← section_text + 1;
    if (*(loc − 1) ≡ '.' ∧ ¬xisdigit(*loc)) goto mistake;      /* not a constant */
    if (*(loc − 1) ≡ '0') {
      if (*loc ≡ 'x' ∨ *loc ≡ 'X') ⟨Get a hexadecimal constant 54⟩
      else if (*loc ≡ 'b' ∨ *loc ≡ 'B') ⟨Get a binary constant 55⟩
      else if (xisdigit(*loc)) ⟨Get an octal constant 56⟩
    }
    *id_loc++ ← *(loc − 1);      /* decimal constant */
    gather_digits_while(xisdigit(*loc) ∨ *loc ≡ '.');
  get_exponent:
    if (*loc ≡ 'e' ∨ *loc ≡ 'E') *id_loc++ ← '_';
    else if (*loc ≡ 'p' ∨ *loc ≡ 'P') *id_loc++ ← '%';
    else goto digit_suffix;
    loc++;
    if (*loc ≡ '+' ∨ *loc ≡ '−') *id_loc++ ← *loc++;
    gather_digits_while(xisdigit(*loc));
  digit_suffix:
    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 44.

54. ⟨ Get a hexadecimal constant 54 ⟩ ≡
  {
    $*id\_loc ++ \leftarrow$ '^';
    $loc ++$;
    $gather\_digits\_while\,(xisxdigit\,(*loc) \vee *loc \equiv$ '.'$)$;
    $*id\_loc ++ \leftarrow$ '/';
    goto $get\_exponent$;
  }

This code is used in section 53.

55. ⟨ Get a binary constant 55 ⟩ ≡
  {
    $*id\_loc ++ \leftarrow$ '\\';
    $loc ++$;
    $gather\_digits\_while\,(*loc \equiv$ '0' $\vee *loc \equiv$ '1'$)$;
    $*id\_loc ++ \leftarrow$ '/';
    goto $digit\_suffix$;
  }

This code is used in section 53.

56. ⟨ Get an octal constant 56 ⟩ ≡
  {
    $*id\_loc ++ \leftarrow$ '~';
    $gather\_digits\_while\,(xisdigit\,(*loc))$;
    $*id\_loc ++ \leftarrow$ '/';
    goto $digit\_suffix$;
  }

This code is used in section 53.

57.   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 57 ⟩ ≡
  { **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') ⟨ Get a wide character constant 58 ⟩
    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 44 and 59.

58. ⟨Get a wide character constant 58⟩ ≡
  {
    if (*delim* ≡ 'u' ∧ *loc* ≡ '8') *++*id_loc* ← *loc* ++;
    delim ← *loc* ++;
    *++*id_loc* ← *delim*;
  }
This code is used in section 57.

59.   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 59⟩ ≡
  {
    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 *trace*: *tracing* ← *c* − '0';
      continue;
    case *xref_roman*: **case** *xref_wildcard*: **case** *xref_typewriter*: **case** *noop*: **case** *TEX_string*:
      skip_restricted( );
      return *ccode*[(**eight_bits**) *c*];
    case *section_name*: ⟨Scan the section name and make *cur_section* point to it 60⟩
    case *verbatim*: ⟨Scan a verbatim string 66⟩
    case *ord*: ⟨Get a string 57⟩
    default: **return** *ccode*[(**eight_bits**) *c*];
    }
  }
This code is used in section 44.

60.   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 60⟩ ≡
  {
    char *k*;   /* pointer into *section_text* */
    cur_section_char ← *(loc* − 1);
    ⟨Put section name into *section_text* 62⟩
    if (*k* − *section_text* > 3 ∧ *strncmp*(*k* − 2, "...", 3) ≡ 0)
      cur_section ← *section_lookup*(*section_text* + 1, *k* − 3, *true*);   /* *true* indicates a prefix */
    else *cur_section* ← *section_lookup*(*section_text* + 1, *k*, *false*);
    xref_switch ← 0;
    return *section_name*;
  }
This code is used in section 59.

61.   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 24⟩ +≡
    section_text[0] ← '␣';

62.   ⟨Put section name into *section_text* 62⟩ ≡
  $k \leftarrow section\_text$;
  while (*true*) {
    if ($loc > limit \wedge get\_line() \equiv false$) {
      $err\_print($"!␣Input␣ended␣in␣section␣name"$)$;
      $loc \leftarrow buffer + 1$;
      break;
    }
    $c \leftarrow *loc$;
    ⟨If end of name or erroneous control code, **break** 63⟩
    $loc{+}{+}$;
    if ($k < section\_text\_end$) $k{+}{+}$;
    if ($xisspace(c)$) {
      $c \leftarrow$ '␣';
      if ($*(k - 1) \equiv$ '␣') $k{-}{-}$;
    }
    $*k \leftarrow c$;
  }
  if ($k \geq section\_text\_end$) {
    $fputs($"\n!␣Section␣name␣too␣long:␣"$, stdout)$;
    $term\_write(section\_text + 1, 25)$;
    $printf($"..."$)$;
    $mark\_harmless$;
  }
  if ($*k \equiv$ '␣' $\wedge k > section\_text$) $k{-}{-}$;
This code is used in section 60.

63.   ⟨If end of name or erroneous control code, **break** 63⟩ ≡
  if ($c \equiv$ '@') {
    $c \leftarrow *(loc + 1)$;
    if ($c \equiv$ '>') {
      $loc \mathrel{+}= 2$;
      break;
    }
    if ($ccode[(\textbf{eight\_bits})\, c] \equiv new\_section$) {
      $err\_print($"!␣Section␣name␣didn't␣end"$)$;
      break;
    }
    if ($c \neq$ '@') {
      $err\_print($"!␣Control␣codes␣are␣forbidden␣in␣section␣name"$)$;
      break;
    }
    $*({+}{+}k) \leftarrow$ '@';
    $loc{+}{+}$;   /∗ now $c \equiv *loc$ again ∗/
  }
This code is used in section 62.

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

```
static void skip_restricted(void)
{
    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");
    }
}
```

65. ⟨Predeclaration of procedures 8⟩ +≡ **static void** *skip_restricted*(**void**);

66. 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 66⟩ ≡
```
{
    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 59.

67.   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 21 ⟩ +≡
    static eight_bits *next_control*;   /∗ control code waiting to be acting upon ∗/

68.   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( );
    change_exists ← false;
    while (¬input_has_ended) ⟨ Store cross-reference data for the current section 70 ⟩
    changed_section[section_count] ← change_exists;        /∗ the index changes if anything does ∗/
    phase ← 2;        /∗ prepare for second phase ∗/
    ⟨ Print error messages about unused or undefined section names 84 ⟩
}
```

69.   ⟨ Predeclaration of procedures 8 ⟩ +≡   **static void** *phase_one*(**void**);

70.   ⟨ Store cross-reference data for the current section 70 ⟩ ≡
```
{
    if (++section_count ≡ max_sections) overflow("section␣number");
    changed_section[section_count] ← changing;        /∗ it will become true if any line changes ∗/
    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 74 ⟩
    ⟨ Store cross-references in the definition part of a section 77 ⟩
    ⟨ Store cross-references in the C part of a section 80 ⟩
    if (changed_section[section_count]) change_exists ← true;
}
```
This code is used in section 68.

71.   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* ≥ *format_code*). If *next_control* ≥ *format_code* when *C_xref* is called, nothing will happen; but if *next_control* ≡ '|' 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* ≡ *ignore*, which triggers the default processing described above. If *spec_ctrl* ≡ *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* ≡ *identifier* + *roman* and *xref_wildcard* ≡ *identifier* + *wildcard* and *xref_typewriter* ≡ *identifier* + *typewriter*, as well as *normal* ≡ 0.

⟨ Predeclaration of procedures 8 ⟩ +≡
  static void *C_xref* (**eight_bits**);
  static void *outer_xref* (**void**);

72.   **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 `"@"` 75 ⟩
        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;
    }
  }

73.   The *outer_xref* subroutine is like *C_xref* except that it begins with *next_control* ≠ '|' and ends with *next_control* ≥ *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 */
         }
      }
}
```

74.   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 74⟩ ≡
```
   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 noop: case section_name: loc −= 2;
         next_control ← get_next ();      /* scan to @> */
         if (next_control ≥ xref_roman ∧ next_control ≤ xref_typewriter) {
            ⟨Replace "@@" by "@" 75⟩
            new_xref (id_lookup (id_first, id_loc, next_control − identifier));
         }
         break;
      }
      if (next_control ≥ format_code) break;
   }
```
This code is used in section 70.

75. ⟨Replace "`@@`" by "`@`" 75⟩ ≡
```
  {
    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 72 and 74.

76. 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 21⟩ +≡
```
  static name_pointer lhs, rhs;     /* pointers to byte_start for format identifiers */
  static name_pointer res_wd_end;      /* pointer to the first nonreserved identifier */
```

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

⟨Store cross-references in the definition part of a section 77⟩ ≡
```
  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 78⟩
    outer_xref();
  }
```
This code is used in section 70.

78.   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 78 ⟩ ≡
```
  {
    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 77.

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

⟨ Process simple format in limbo 79 ⟩ ≡
```
  {
    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 41.

80.   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 80 ⟩ ≡
  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 70.

81.   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 21 ⟩ +≡
  static xref_pointer *cur_xref*;   /∗ temporary cross-reference pointer ∗/
  static boolean *an_output*;   /∗ did *file_flag* precede *cur_xref*? ∗/

82.   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*);
    }
  }

83.   ⟨ Predeclaration of procedures 8 ⟩ +≡   **static void** *section_check*(**name_pointer**);

84. ⟨ Print error messages about unused or undefined section names 84 ⟩ ≡
  $section\_check(root)$;

This code is used in section 68.

85.   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 21 ⟩ +≡
  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 ∗/

86.   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 8 ⟩ +≡
  static void *flush_buffer*(**char** ∗, **boolean**, **boolean**);
  static void *finish_line*(**void**);

87.   **static void** *flush_buffer*(**char** ∗*b*,   /∗ outputs from *out_buf* + 1 to *b*, where $b \le$ *out_ptr* ∗/
    boolean *per_cent*, **boolean** *carryover*)
  {
    char ∗*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*;
  }

88.   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);
   }
}
```

89.   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 '`\input cwebmac`'.

⟨ Start TEX output 89 ⟩ ≡
  out_ptr ← *out_buf* + 1;
  out_line ← 1;
  active_file ← *tex_file* ;
  ∗*out_ptr* ← '`c`';
  tex_printf ("`\\input␣cwebma`");

This code is used in section 2.

90.   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 (out_ptr ≥ out_buf_end ) break_out ( );
           ∗(++out_ptr) ← c;
        }
```
⟨ Predeclaration of procedures 8 ⟩ +≡
  static void *out_str*(**const char** ∗);
  static void *break_out*(**void**);

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

92.   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 24 ⟩ +≡
  out_buf [0] ← '`\\`';

93.   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 94⟩
        if (*k ≡ '␣') {
            flush_buffer(k, false, true);
            return;
        }
        if (*(k−−) ≡ '\\' ∧ *k ≠ '\\') {      /* we've decreased k */
            flush_buffer(k, true, true);
            return;
        }
    }
}
```

94.   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** 94⟩ ≡
```
    {
        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 93.

95.   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. If the section is changed, we output '*' just after the number.

⟨Predeclaration of procedures 8⟩ +≡
```
    static void out_section(sixteen_bits);
    static void out_name(name_pointer, boolean);
```

96.   `static void out_section(sixteen_bits n)`
```
    {
        char s[6];
        sprintf(s, "%d", n);
        out_str(s);
        if (changed_section[n]) out_str("\\*");
    }
```

97.   The *out_name* procedure is used to output an identifier or index entry, enclosing 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('}');
}
```

98.   Routines that copy TEX material.   During phase two, we use the subroutines *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 8 ⟩ +≡
  static void *copy_limbo*(**void**);
  static eight_bits *copy_TEX*(**void**);
  static int *copy_comment*(**boolean**, **int**);

99.   **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 ∗/
        default: *err_print*("!␣Double␣@␣should␣be␣used␣in␣limbo");
          out('@');
        }
      }
    }
  }

100.   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 ++)];
    }
}
```

101.   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 102⟩
        if (phase ≡ 2) {
            if (ishigh(c)) app_tok(quoted_char);
            app_tok(c);
        }
        ⟨Copy special things when c ≡ '@', '\\' 103⟩
        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 104⟩
}
```

102.   ⟨ Check for end of comment 102 ⟩ ≡
   if $(c \equiv$ '*' $\wedge *loc \equiv$ '/') {
     $loc{+}{+};$
     if $(bal > 1)$ $err\_print($ "!␣Missing␣}␣in␣comment" $);$
     goto $done;$
   }

This code is used in section 101.

103.   ⟨ Copy special things when $c \equiv$ '@', '\\' 103 ⟩ ≡
   if $(c \equiv$ '@') {
     if $(*(loc{+}{+}) \neq$ '@') {
       $err\_print($ "!␣Illegal␣use␣of␣@␣in␣comment" $);$
       $loc \mathrel{-}= 2;$
       if $(phase \equiv 2)$ $*(tok\_ptr - 1) \leftarrow$ '␣';
       goto $done;$
     }
   }
   else {
     if $(c \equiv$ '\\' $\wedge *loc \neq$ '@') {
       if $(phase \equiv 2)$ $app\_tok(*(loc{+}{+}))$
       else $loc{+}{+};$
     }
   }

This code is used in section 101.

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

⟨ Clear bal and **return** 104 ⟩ ≡
   if $(phase \equiv 2)$
     while $(bal{-}{-} > 0)$ $app\_tok($ '}' $);$
   return 0;

This code is used in section 101.

105. 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

$$exp \left\{ \begin{matrix} binop \\ ubinop \end{matrix} \right\} exp \;\rightarrow\; exp$$

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

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

(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.

106.   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 ∗/
#**define** *rpar* 12   /∗ denotes a right parenthesis ∗/
#**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* 63   /∗ *make_pair* ∗/
#**define** *new_exp* 64   /∗ **new** and a following type identifier ∗/
#**define** *begin_arg* 65   /∗ @[∗/
#**define** *end_arg* 66   /∗ @] ∗/
#**define** *lbrack* 67   /∗ denotes a left bracket ∗/
#**define** *rbrack* 68   /∗ denotes a right bracket ∗/
#**define** *attr_head* 69   /∗ denotes beginning of attribute ∗/

⟨ Private variables 21 ⟩ +≡
  static char *cat_name*[256][12];

107.   ⟨ Set initial values 24 ⟩ +≡
  {
    int c;
    for $(c \leftarrow 0;\ c < 256;\ c{+}{+})$ $strcpy(cat\_name[c], \texttt{"UNKNOWN"})$;
  }
  $strcpy(cat\_name[exp], \texttt{"exp"})$;
  $strcpy(cat\_name[unop], \texttt{"unop"})$;
  $strcpy(cat\_name[binop], \texttt{"binop"})$;
  $strcpy(cat\_name[ubinop], \texttt{"ubinop"})$;
  $strcpy(cat\_name[cast], \texttt{"cast"})$;
  $strcpy(cat\_name[question], \texttt{"?"})$;
  $strcpy(cat\_name[lbrace], \texttt{"\{"})$;
  $strcpy(cat\_name[rbrace], \texttt{"\}"})$;
  $strcpy(cat\_name[decl\_head], \texttt{"decl\_head"})$;
  $strcpy(cat\_name[comma], \texttt{","})$;
  $strcpy(cat\_name[lpar], \texttt{"("})$;
  $strcpy(cat\_name[rpar], \texttt{")"})$;
  $strcpy(cat\_name[prelangle], \texttt{"<"})$;
  $strcpy(cat\_name[prerangle], \texttt{">"})$;
  $strcpy(cat\_name[langle], \texttt{"\\<"})$;
  $strcpy(cat\_name[colcol], \texttt{"::"})$;
  $strcpy(cat\_name[base], \texttt{"\\:"})$;
  $strcpy(cat\_name[decl], \texttt{"decl"})$;
  $strcpy(cat\_name[struct\_head], \texttt{"struct\_head"})$;
  $strcpy(cat\_name[alfop], \texttt{"alfop"})$;
  $strcpy(cat\_name[stmt], \texttt{"stmt"})$;
  $strcpy(cat\_name[function], \texttt{"function"})$;
  $strcpy(cat\_name[fn\_decl], \texttt{"fn\_decl"})$;
  $strcpy(cat\_name[else\_like], \texttt{"else\_like"})$;
  $strcpy(cat\_name[semi], \texttt{";"})$;
  $strcpy(cat\_name[colon], \texttt{":"})$;
  $strcpy(cat\_name[tag], \texttt{"tag"})$;
  $strcpy(cat\_name[if\_head], \texttt{"if\_head"})$;
  $strcpy(cat\_name[else\_head], \texttt{"else\_head"})$;
  $strcpy(cat\_name[if\_clause], \texttt{"if()"})$;
  $strcpy(cat\_name[lproc], \texttt{"\#\{"})$;
  $strcpy(cat\_name[rproc], \texttt{"\#\}"})$;
  $strcpy(cat\_name[insert], \texttt{"insert"})$;
  $strcpy(cat\_name[section\_scrap], \texttt{"section"})$;
  $strcpy(cat\_name[dead], \texttt{"@d"})$;
  $strcpy(cat\_name[public\_like], \texttt{"public"})$;
  $strcpy(cat\_name[operator\_like], \texttt{"operator"})$;
  $strcpy(cat\_name[new\_like], \texttt{"new"})$;
  $strcpy(cat\_name[catch\_like], \texttt{"catch"})$;
  $strcpy(cat\_name[for\_like], \texttt{"for"})$;
  $strcpy(cat\_name[do\_like], \texttt{"do"})$;
  $strcpy(cat\_name[if\_like], \texttt{"if"})$;
  $strcpy(cat\_name[delete\_like], \texttt{"delete"})$;
  $strcpy(cat\_name[raw\_ubin], \texttt{"ubinop?"})$;
  $strcpy(cat\_name[const\_like], \texttt{"const"})$;
  $strcpy(cat\_name[raw\_int], \texttt{"raw"})$;
  $strcpy(cat\_name[int\_like], \texttt{"int"})$;

$strcpy(cat\_name[case\_like], \texttt{"case"});$
$strcpy(cat\_name[sizeof\_like], \texttt{"sizeof"});$
$strcpy(cat\_name[struct\_like], \texttt{"struct"});$
$strcpy(cat\_name[typedef\_like], \texttt{"typedef"});$
$strcpy(cat\_name[define\_like], \texttt{"define"});$
$strcpy(cat\_name[template\_like], \texttt{"template"});$
$strcpy(cat\_name[ftemplate], \texttt{"ftemplate"});$
$strcpy(cat\_name[new\_exp], \texttt{"new\_exp"});$
$strcpy(cat\_name[begin\_arg], \texttt{"@["});$
$strcpy(cat\_name[end\_arg], \texttt{"@]"});$
$strcpy(cat\_name[lbrack], \texttt{"["});$
$strcpy(cat\_name[rbrack], \texttt{"]"});$
$strcpy(cat\_name[attr\_head], \texttt{"attr\_head"});$
$strcpy(cat\_name[attr], \texttt{"attr"});$
$strcpy(cat\_name[alignas\_like], \texttt{"alignas"});$
$strcpy(cat\_name[using\_like], \texttt{"using"});$
$strcpy(cat\_name[default\_like], \texttt{"default"});$
$strcpy(cat\_name[0], \texttt{"zero"});$

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

static void $print\_cat($ /∗ symbolic printout of a category ∗/
   eight_bits $c)$
{
   $fputs(cat\_name[c], stdout);$
}

109. ⟨ Predeclaration of procedures 8 ⟩ +≡   **static void** $print\_cat(\textbf{eight\_bits});$

110.   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;

dindent causes future lines to be indented two more ems.

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).

A *dindent* token becomes \1\1. It is equivalent to a pair of *indent* tokens. However, if *dindent* immediately precedes *big_force*, the two tokens are swapped, so that the indentation happens after the line break.

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 ∗/
#**define** *dindent* °*226*   /∗ two more tabs (\1\1) ∗/

111.   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
0b10111	*exp*: \T{\\10111/}	maybe
77	*exp*: \T{77}	maybe
77L	*exp*: \T{77\$L}	maybe
0.1E5	*exp*: \T{0.1_5}	maybe
0x10p3	*exp*: \T{\^10/\%3}	maybe
1'000'000	*exp*: \T{1\⊔000\⊔000}	maybe
+	*ubinop*: +	yes
–	*ubinop*: –	yes
*	*raw_ubin*: *	yes
/	*binop*: /	yes
<	*prelangle*: \langle	yes
=	*binop*: \K	yes
>	*prerangle*: \rangle	yes
.	*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
)	*rpar*:)	maybe
[	*lbrack*: [	maybe

]	*rbrack*:]	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
alignas	*alignas_like*: **	maybe
alignof	*sizeof_like*: **	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
char8_t	*raw_int*: **	maybe
char16_t	*raw_int*: **	maybe
char32_t	*raw_int*: **	maybe
class	*struct_like*: **	maybe
clock_t	*raw_int*: **	maybe
compl	*alfop*: **	yes
complex	*int_like*: **	yes
concept	*int_like*: **	maybe
const	*const_like*: **	maybe
consteval	*const_like*: **	maybe
constexpr	*const_like*: **	maybe
constinit	*const_like*: **	maybe
const_cast	*raw_int*: **	maybe
continue	*case_like*: **	maybe
co_await	*case_like*: **	maybe
co_return	*case_like*: **	maybe
co_yield	*case_like*: **	maybe
decltype	*sizeof_like*: **	maybe
default	*default_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
false	*normal*: **	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
imaginary	*int_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
make_pair	*ftemplate*: \\{make_pair}	maybe
mutable	*int_like*: **	maybe
namespace	*struct_like*: **	maybe
new	*new_like*: **	maybe
noexcept	*attr*: **	maybe
not	*alfop*: **	yes
not_eq	*alfop*: **	yes
NULL	*exp*: \NULL	yes
nullptr	*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
requires	*int_like*: **	maybe
restrict	*int_like*: **	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_assert`	*sizeof_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
`thread_local`	*raw_int*: **	maybe
`throw`	*case_like*: **	maybe
`time_t`	*raw_int*: **	maybe
`try`	*else_like*: **	maybe
`typedef`	*typedef_like*: **	maybe
`typeid`	*sizeof_like*: **	maybe
`typename`	*struct_like*: **	maybe
`undef`	*if_like*: **	maybe
`union`	*struct_like*: **	maybe
`unsigned`	*raw_int*: **	maybe
`using`	*using_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.

112.   Here is a table of all the 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*, *bsp*, and *din* as shorthands for *indent*, *outdent*, *backup*, *break_space*, and *dindent*, respectively.

LHS	$\rightarrow$ RHS	Translation	Example
0 $\left\{\begin{array}{c} any \\ any\ any \\ any\ any\ any \end{array}\right\}$ *insert*	$\rightarrow \left\{\begin{array}{c} any \\ any\ any \\ any\ any\ any \end{array}\right\}$		stmt; /∗ comment ∗/
1 $exp \left\{\begin{array}{c} lbrace \\ int\_like \\ decl \end{array}\right\}$	$\rightarrow fn\_decl \left\{\begin{array}{c} lbrace \\ int\_like \\ decl \end{array}\right\}$	$F = E^* din$	$main()\{$ $main(ac, av)$ **int** $ac;$
2 $exp\ unop$	$\rightarrow exp$		$x{+}{+}$
3 $exp \left\{\begin{array}{c} binop \\ ubinop \end{array}\right\} exp$	$\rightarrow exp$		x/y $x + y$
4 $exp\ comma\ exp$	$\rightarrow exp$	$EC\ opt9\ E$	$f(x, y)$
5 $exp \left\{\begin{array}{c} lpar\ rpar \\ cast \end{array}\right\} colon$	$\rightarrow exp \left\{\begin{array}{c} lpar\ rpar \\ cast \end{array}\right\} base$		**C**( ): **C**(**int** i):
6 $exp\ semi$	$\rightarrow stmt$		$x \leftarrow 0;$
7 $exp\ colon$	$\rightarrow tag$	E^*C	*found*:
8 $exp\ rbrace$	$\rightarrow stmt\ rbrace$		end of **enum** list
9 $exp \left\{\begin{array}{c} lpar\ rpar \\ cast \end{array}\right\} \left\{\begin{array}{c} const\_like \\ case\_like \end{array}\right\}$	$\rightarrow exp \left\{\begin{array}{c} lpar\ rpar \\ cast \end{array}\right\}$	$\left\{\begin{array}{c} R = R_{\sqcup}C \\ C_1 = C_{1\sqcup}C_2 \end{array}\right\}$	$f(\,)$ **const** $f(\textbf{int})$ **throw**
10 $exp \left\{\begin{array}{c} exp \\ cast \end{array}\right\}$	$\rightarrow exp$		$time(\,)$
11 $lpar \left\{\begin{array}{c} exp \\ ubinop \end{array}\right\} rpar$	$\rightarrow exp$		(x) $(*)$
12 $lpar\ rpar$	$\rightarrow exp$	$L\backslash, R$	functions, declarations
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 $lpar \left\{\begin{array}{c} stmt \\ decl \end{array}\right\}$	$\rightarrow lpar$	$\left\{\begin{array}{c} LS_{\sqcup} \\ LD_{\sqcup} \end{array}\right\}$	$(k \leftarrow 5;$ $(\textbf{int}\ k \leftarrow 5;$
16 $unop \left\{\begin{array}{c} exp \\ int\_like \end{array}\right\}$	$\rightarrow exp$		$\neg x$ ∼**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 $binop\ binop$	$\rightarrow binop$	$math\_rel\ \{B_1\}\{B_2\}\}$	≫=

21 $cast \begin{Bmatrix} lpar \\ exp \end{Bmatrix}$ $\rightarrow \begin{Bmatrix} lpar \\ exp \end{Bmatrix}$ $\begin{Bmatrix} CL \\ C_\sqcup E \end{Bmatrix}$ **(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 $int\_like \begin{Bmatrix} int\_like \\ struct\_like \end{Bmatrix}$ $\rightarrow \begin{Bmatrix} int\_like \\ struct\_like \end{Bmatrix}$ $I_\sqcup \begin{Bmatrix} I \\ S \end{Bmatrix}$ **extern char**

26 $int\_like\ exp \begin{Bmatrix} raw\_int \\ struct\_like \end{Bmatrix}$ $\rightarrow int\_like \begin{Bmatrix} raw\_int \\ struct\_like \end{Bmatrix}$ **extern"Ada" int**

27 $int\_like \begin{Bmatrix} exp \\ ubinop \\ colon \end{Bmatrix}$ $\rightarrow decl\_head \begin{Bmatrix} exp \\ ubinop \\ colon \end{Bmatrix}$ $D = I_\sqcup$ **int** x
int $*x$
unsigned :

28 $int\_like \begin{Bmatrix} semi \\ binop \end{Bmatrix}$ $\rightarrow decl\_head \begin{Bmatrix} semi \\ binop \end{Bmatrix}$ **int** x;
int $f(\textbf{int} = 4)$

29 $public\_like\ colon$ $\rightarrow tag$ **private**:

30 $public\_like$ $\rightarrow int\_like$ **private**

31 $colcol \begin{Bmatrix} exp \\ int\_like \end{Bmatrix}$ $\rightarrow \begin{Bmatrix} exp \\ int\_like \end{Bmatrix}$ $qualifier\ C \begin{Bmatrix} E \\ I \end{Bmatrix}$ **C**::x

32 $colcol\ colcol$ $\rightarrow colcol$ **C**::**B**::

33 $decl\_head\ comma$ $\rightarrow decl\_head$ $DC_\sqcup$ **int** $x,$

34 $decl\_head\ ubinop$ $\rightarrow decl\_head$ $D\{U\}$ **int** $*$

†35 $decl\_head\ exp$ $\rightarrow decl\_head$ DE^* **int** x

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

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

38 $decl\_head \begin{Bmatrix} int\_like \\ lbrace \\ decl \end{Bmatrix}$ $\rightarrow fn\_decl \begin{Bmatrix} int\_like \\ lbrace \\ decl \end{Bmatrix}$ $F = D\ din$ **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 $decl \begin{Bmatrix} stmt \\ function \end{Bmatrix}$ $\rightarrow \begin{Bmatrix} stmt \\ function \end{Bmatrix}$ $D\ big\_force \begin{Bmatrix} S \\ F \end{Bmatrix}$ **extern** n; $main\ (\,)\{\ \}$

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

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

44 $struct\_like\ lbrace$ $\rightarrow struct\_head$ $S_\sqcup L$ **struct** {

45 $struct\_like \begin{Bmatrix} exp \\ int\_like \end{Bmatrix} semi$ $\rightarrow decl\_head\ semi$ $S_\sqcup \begin{Bmatrix} E^{**} \\ I^{**} \end{Bmatrix}$ **struct forward**;

46 $struct\_like \begin{Bmatrix} exp \\ int\_like \end{Bmatrix} lbrace$ $\rightarrow struct\_head$ $S_\sqcup \begin{Bmatrix} E^{**} \\ I^{**} \end{Bmatrix}_\sqcup L$ **struct name_info** {

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 $struct\_head \left\{\begin{matrix} decl \\ stmt \\ function \end{matrix}\right\} rbrace \rightarrow int\_like$ $S \; in \; force \left\{\begin{matrix} D \\ S \\ F \end{matrix}\right\} out \; force \; R$ **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 $fn\_decl \; stmt \rightarrow function$ $F \; out \; out \; force \; S$ $main() \ldots$

53 $function \left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\} \rightarrow \left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\}$ $F \; big\_force \left\{\begin{matrix} S \\ D \\ F \end{matrix}\right\}$ outer block

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

55 $lbrace \left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\} rbrace \rightarrow stmt$ $force \; L \; in \; force \; S \; force \; back \; R \; out \; force$ 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** $x \leftarrow 0$;

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

62 $if\_clause \; lbrace \rightarrow if\_head \; lbrace$ **if** (x) {

63 $if\_clause \; stmt \; else\_like \; if\_like \rightarrow if\_like$ $force \; I \; in \; bsp \; S \; out \; force \; E_{\sqcup}I$ **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 $if\_head \left\{\begin{matrix} stmt \\ exp \end{matrix}\right\} else\_like \; if\_like \rightarrow if\_like \; force \; I \; bsp \; noop \; cancel \; S \; force \; E_{\sqcup}I$ **if** (x) { y; } **else if**

67 $if\_head \left\{\begin{matrix} stmt \\ exp \end{matrix}\right\} else\_like \rightarrow else\_like \; force \; I \; bsp \; noop \; cancel \; S \; force \; E$ **if** (x) { y; } **else**

68 $if\_head \left\{\begin{matrix} stmt \\ exp \end{matrix}\right\} \rightarrow else\_head \left\{\begin{matrix} stmt \\ exp \end{matrix}\right\}$ **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 $case\_like \; exp \rightarrow exp$ $C_{\sqcup}E$ **return** 0

73 $catch\_like \left\{\begin{matrix} cast \\ exp \end{matrix}\right\} \rightarrow fn\_decl$ $C\left\{\begin{matrix} C \\ E \end{matrix}\right\} in \; in$ **catch** $(\ldots)$

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

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

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

77 $semi \rightarrow stmt$ $_{\sqcup}S$ empty statement

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

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

80	$lproc \left\{ \begin{matrix} exp\ [exp] \\ function \end{matrix} \right\} rproc$	$\rightarrow insert$	$I_{\sqcup} \left\{ \begin{matrix} E[_{\sqcup}\backslash 5E] \\ F \end{matrix} \right\}$	#**define** a 1 #**define** a { b; }
81	$section\_scrap\ semi$	$\rightarrow stmt$	$MS\ force$	$\langle$ section name $\rangle$;
82	$section\_scrap$	$\rightarrow exp$		$\langle$ section name $\rangle$
83	$insert\ any$	$\rightarrow any$		|#include|
84	$prelangle$	$\rightarrow binop$	$<$	$<$ not in template
85	$prerangle$	$\rightarrow binop$	$>$	$>$ not in template
86	$langle\ prerangle$	$\rightarrow cast$	$L\backslash, P$	$\langle\rangle$
87	$langle \left\{ \begin{matrix} decl\_head \\ int\_like \\ exp \end{matrix} \right\} prerangle$	$\rightarrow cast$		$\langle$**class C**$\rangle$
88	$langle \left\{ \begin{matrix} decl\_head \\ int\_like \\ exp \end{matrix} \right\} comma$	$\rightarrow langle$	$L \left\{ \begin{matrix} D \\ I \\ E \end{matrix} \right\} C\ opt9$	$\langle$**class C**,
89	$template\_like\ exp\ prelangle$	$\rightarrow template\_like\ exp\ langle$		**template** $a\langle 100 \rangle$
90	$template\_like \left\{ \begin{matrix} exp \\ raw\_int \end{matrix} \right\}$	$\rightarrow \left\{ \begin{matrix} exp \\ raw\_int \end{matrix} \right\}$	$T_{\sqcup} \left\{ \begin{matrix} E \\ R \end{matrix} \right\}$	**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	$new\_exp \left\{ \begin{matrix} int\_like \\ const\_like \end{matrix} \right\}$	$\rightarrow new\_exp$	$N_{\sqcup} \left\{ \begin{matrix} I \\ C \end{matrix} \right\}$	**new const int**
96	$new\_exp\ struct\_like \left\{ \begin{matrix} exp \\ int\_like \end{matrix} \right\}$	$\rightarrow new\_exp$	$N_{\sqcup}S_{\sqcup} \left\{ \begin{matrix} E \\ I \end{matrix} \right\}$	**new struct S**
97	$new\_exp\ raw\_ubin$	$\rightarrow new\_exp$	$N\{R\}$	**new int**$*[2]$
98	$new\_exp \left\{ \begin{matrix} lpar \\ exp \end{matrix} \right\}$	$\rightarrow exp \left\{ \begin{matrix} lpar \\ exp \end{matrix} \right\}$	$E = N \left\{ \begin{matrix} \\ _{\sqcup} \end{matrix} \right\}$	**operator**$[\,](\textbf{int})$ **new int**(2)
†99	new_exp	$\rightarrow exp$		**new int**;
100	$ftemplate\ prelangle$	$\rightarrow ftemplate\ langle$		$make\_pair\langle\textbf{int},\textbf{int}\rangle$
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	$raw\_int\ prelangle$	$\rightarrow raw\_int\ langle$		**C**$\langle$
107	$raw\_int\ colcol$	$\rightarrow colcol$		**C**::
108	$raw\_int\ cast$	$\rightarrow raw\_int$		**C**$\langle$**class T**$\rangle$
109	$raw\_int\ lpar$	$\rightarrow exp\ lpar$		**complex**(x,y)
†110	raw_int	$\rightarrow int\_like$		**complex** z
†111	$operator\_like \left\{ \begin{matrix} binop \\ unop \\ ubinop \end{matrix} \right\}$	$\rightarrow exp$	$O\{ \left\{ \begin{matrix} B \\ U \\ U \end{matrix} \right\} \}$	**operator**$+$
112	$operator\_like \left\{ \begin{matrix} new\_like \\ delete\_like \end{matrix} \right\}$	$\rightarrow exp$	$O_{\sqcup} \left\{ \begin{matrix} N \\ S \end{matrix} \right\}$	**operator delete**
113	$operator\_like\ comma$	$\rightarrow exp$		**operator**,
†114	$operator\_like$	$\rightarrow new\_exp$		**operator char**$*$
115	$typedef\_like \left\{ \begin{matrix} int\_like \\ cast \end{matrix} \right\} \left\{ \begin{matrix} comma \\ semi \end{matrix} \right\} \rightarrow typedef\_like\ exp \left\{ \begin{matrix} comma \\ semi \end{matrix} \right\}$			**typedef int I**,

116 *typedef_like int_like*	→ *typedef_like*	$T_\sqcup I$	**typedef char**
†117 *typedef_like exp*	→ *typedef_like*	$T_\sqcup E^{**}$	**typedef I** @[@] (∗**P**)
118 *typedef_like comma*	→ *typedef_like*	$TC_\sqcup$	**typedef int x**,
119 *typedef_like semi*	→ *decl*		**typedef int x**, **y**;
120 *typedef_like ubinop* $\left\{ \begin{array}{c} cast \\ ubinop \end{array} \right\}$	→ *typedef_like* $\left\{ \begin{array}{c} cast \\ ubinop \end{array} \right\}$	$\left\{ \begin{array}{c} C = \{U\}C \\ U_2 = \{U_1\}U_2 \end{array} \right\}$	**typedef** ∗∗(**CPtr**)
121 *delete_like lpar rpar*	→ *delete_like*	$DL\backslash, R$	**delete**[ ]
122 *delete_like exp*	→ *exp*	$D_\sqcup E$	**delete** p
†123 *question exp* $\left\{ \begin{array}{c} colon \\ base \end{array} \right\}$	→ *binop*		? x : ? $f(\,)$:
124 *begin_arg end_arg*	→ *exp*		@[**char**∗@]
125 *any_other end_arg*	→ *end_arg*		**char**∗@]
126 *alignas_like decl_head*	→ *attr*		**alignas**(**struct** s ∗)
127 *alignas_like exp*	→ *attr*		**alignas**(8)
128 *lbrack lbrack*	→ *attr_head*		attribute begins
129 *lbrack*	→ *lpar*		[elsewhere
130 *rbrack*	→ *rpar*		] elsewhere
131 *attr_head rbrack rbrack*	→ *attr*		[[. . .]]
132 *attr_head exp*	→ *attr_head*		[[*deprecated*
133 *attr_head using_like exp colon*	→ *attr_head*		[[**using** NS :
134 *attr* $\left\{ \begin{array}{c} lbrace \\ stmt \end{array} \right\}$	→ $\left\{ \begin{array}{c} lbrace \\ stmt \end{array} \right\}$	$A_\sqcup \left\{ \begin{array}{c} S \\ L \end{array} \right\}$	[[*likely*]] {
135 *attr tag*	→ *tag*	$A_\sqcup T$	[[*likely*]] **case** 0:
136 *attr semi*	→ *stmt*		[[*fallthrough*]];
137 *attr attr*	→ *attr*	$A_\sqcup A$	**alignas**(x) [[. . .]]
138 *attr decl_head*	→ *decl_head*		[[*nodiscard*]] $f(\,)$
139 *decl_head attr*	→ *decl_head*		(**int** x [[*deprecated*]])
140 *using_like*	→ *int_like*		**using** not in attributes
141 *struct_like attr*	→ *struct_like*	$S_\sqcup A$	**struct** [[*deprecated*]]
142 *exp attr*	→ *attr*	$E_\sqcup A$	**enum** {x [[. . .]]}
143 *attr typedef_like*	→ *typedef_like*	$A_\sqcup T$	[[*deprecated*]] **typedef**
144 *raw_int lbrack*	→ *exp*		**int**[3]
145 *attr_head comma*	→ *attr_head*		[[x, y
146 *if_head attr*	→ *if_head*	$I_\sqcup A$	**if** (x) [[*unlikely*]] {
147 *lbrack lbrack rbrack rbrack*	→ *exp*		[[]]
148 *attr function*	→ *function*	$A_\sqcup F$	attribute and function
149 *default_like colon*	→ *case_like colon*		**default**:
150 *default_like*	→ *exp*		$f(\,) \leftarrow$ **default**;
151 *struct_like struct_like*	→ *struct_like*	$S_\sqcup S$	**enum class**
152 *exp colcol int_like*	→ *int_like*		*std* :: **atomic**
†153 *langle struct_like* $\left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$ *comma*	→ *langle*	$LS \left\{ \begin{array}{c} E^{**} \\ I^{**} \end{array} \right\} C$	⟨**typename** t,
†154 *langle struct_like* $\left\{ \begin{array}{c} exp \\ int\_like \end{array} \right\}$ *prerangle*	→ *cast*	$LS \left\{ \begin{array}{c} E^{**} \\ I^{**} \end{array} \right\} P$	⟨**typename** t⟩
155 *template_like cast struct_like*	→ *struct_like*	$T_\sqcup CS$	**template**⟨. . .⟩ **class**
156 *tag rbrace*	→ *decl rbrace*		**public**: }
157 *fn_decl attr*	→ *fn_decl*	$F_\sqcup A$	**void** $f(\,)$ **noexcept**
158 *alignas_like cast*	→ *attr*		**alignas**(**int**)

†**Notes**

Rule 35: The *exp* must not be immediately followed by *lpar*, *lbrack*, *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 117: The *exp* must not be immediately followed by *lpar*, *exp*, or *cast*.

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

Rules 153, 154: *make_reserved* is called only if the `+t` option is given to CWEAVE.

113.   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 $pp\rightarrow cat$, $(pp+1)\rightarrow cat$, ... 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 22 ⟩ +≡
  typedef struct {
    eight_bits *cat*;
    eight_bits *mathness*;
    union {
      text_pointer *Trans*;
      ⟨ Rest of *trans_plus* union 254 ⟩
    } *trans_plus*;
  } **scrap**;
  typedef scrap *∗**scrap_pointer**;

114.   #**define** *trans   trans_plus.Trans*   /∗ translation texts of scraps ∗/

⟨ Private variables 21 ⟩ +≡
  static scrap *scrap_info*[*max_scraps*];   /∗ memory array for scraps ∗/
  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 ∗/

115.   ⟨ Set initial values 24 ⟩ +≡
  scrap_base ← *scrap_info* + 1;
  max_scr_ptr ← *scrap_ptr* ← *scrap_info*;

116.   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 '| . . . |' ∗/

```
  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 ≥ text_ptr) printf("BAD");
  else
    for (j ← *p; j < *(p + 1); j++) {
      r ← *j % id_flag;
      switch (*j) {
      case id_flag: printf("\\\\{");
        print_id((name_dir + r));
        putchar('}');
        break;
      case res_flag: printf("\\&{");
        print_id((name_dir + r));
        putchar('}');
        break;
      case section_flag: putchar('<');
        print_section_name((name_dir + r));
        putchar('>');
        break;
      case tok_flag: printf("[[%d]]", r);
        break;
      case inner_tok_flag: printf("|[[%d]]|", r);
        break;
      default: ⟨Print token r in symbolic form 118⟩
      }
    }
  update_terminal;
}
```

117.   ⟨Predeclaration of procedures 8⟩ +≡   **static void** *print_text*(**text_pointer** *p*);

118.   ⟨ Print token r in symbolic form 118 ⟩ ≡
  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 *dindent*: *printf* ("[dindent]");
     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: *putchar* (r);
  }
This code is used in section 116.

119.   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 '$exp\ comma\ exp \to exp$' 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, $pp\text{-}trans$ and $(pp + 1)\text{-}trans$, 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.

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** $big\_app1\_insert(p, c)$ $big\_app1(p);\ big\_app(c);\ big\_app1(p + 1)$
#**define** $app(a)$ $*(tok\_ptr +\!+) \leftarrow (\textbf{token})(a)$
#**define** $app1(a)$ $*(tok\_ptr +\!+) \leftarrow (\textbf{token})(tok\_flag + (\textbf{int})((a)\text{-}trans - tok\_start))$

⟨ Private variables 21 ⟩ +≡
   static int $cur\_mathness$, $init\_mathness$;

120.   ⟨ Predeclaration of procedures 8 ⟩ +≡
  static void *app_str*(**const char** ∗);
  static void *big_app*(**token**);
  static void *big_app1*(**scrap_pointer**);

121.   **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*) ∨ *a* ≡ *dindent*)   /∗ 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*));
  }

122.   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** $cat1$ $(pp + 1) \rightarrow cat$
#**define** $cat2$ $(pp + 2) \rightarrow cat$
#**define** $cat3$ $(pp + 3) \rightarrow cat$
#**define** lhs_not_simple $(pp \rightarrow cat \neq public\_like \wedge pp \rightarrow cat \neq semi \wedge pp \rightarrow cat \neq prelangle \wedge pp \rightarrow cat \neq prerangle$
$\wedge\; pp \rightarrow cat \neq template\_like \wedge pp \rightarrow cat \neq new\_like \wedge pp \rightarrow cat \neq new\_exp \wedge pp \rightarrow cat \neq ftemplate$
$\wedge\; pp \rightarrow cat \neq raw\_ubin \wedge pp \rightarrow cat \neq const\_like \wedge pp \rightarrow cat \neq raw\_int \wedge pp \rightarrow cat \neq operator\_like)$
           /∗ not a production with left side length 1 ∗/

⟨ Match a production at pp, or increase pp if there is no match 122 ⟩ ≡
  {
    if $(cat1 \equiv end\_arg \wedge lhs\_not\_simple)$
      if $(pp \rightarrow cat \equiv begin\_arg)$ $squash(pp, 2, exp, -2, 124);$
      else $squash(pp, 2, end\_arg, -1, 125);$
    else if $(pp \rightarrow cat \equiv rbrack)$ $squash(pp, 1, rpar, -3, 130);$
    else if $(pp \rightarrow cat \equiv using\_like)$ $squash(pp, 1, int\_like, -3, 140);$
    else if $(cat1 \equiv insert)$ $squash(pp, 2, pp \rightarrow cat, -2, 0);$
    else if $(cat2 \equiv insert)$ $squash(pp + 1, 2, (pp + 1) \rightarrow cat, -1, 0);$
    else if $(cat3 \equiv insert)$ $squash(pp + 2, 2, (pp + 2) \rightarrow cat, 0, 0);$
    else
      switch $(pp \rightarrow cat)$ {
      case exp: ⟨ Cases for exp 129 ⟩ **break**;
      case $lpar$: ⟨ Cases for $lpar$ 130 ⟩ **break**;
      case $unop$: ⟨ Cases for $unop$ 131 ⟩ **break**;
      case $ubinop$: ⟨ Cases for $ubinop$ 132 ⟩ **break**;
      case $binop$: ⟨ Cases for $binop$ 133 ⟩ **break**;
      case $cast$: ⟨ Cases for $cast$ 134 ⟩ **break**;
      case $sizeof\_like$: ⟨ Cases for $sizeof\_like$ 135 ⟩ **break**;
      case int_like: ⟨ Cases for int_like 136 ⟩ **break**;
      case $public\_like$: ⟨ Cases for $public\_like$ 137 ⟩ **break**;
      case $colcol$: ⟨ Cases for $colcol$ 138 ⟩ **break**;
      case $decl\_head$: ⟨ Cases for $decl\_head$ 139 ⟩ **break**;
      case $decl$: ⟨ Cases for $decl$ 140 ⟩ **break**;
      case $base$: ⟨ Cases for $base$ 141 ⟩ **break**;
      case $struct\_like$: ⟨ Cases for $struct\_like$ 142 ⟩ **break**;
      case $struct\_head$: ⟨ Cases for $struct\_head$ 143 ⟩ **break**;
      case fn_decl: ⟨ Cases for fn_decl 144 ⟩ **break**;
      case $function$: ⟨ Cases for $function$ 145 ⟩ **break**;
      case $lbrace$: ⟨ Cases for $lbrace$ 146 ⟩ **break**;
      case if_like: ⟨ Cases for if_like 147 ⟩ **break**;
      case $else\_like$: ⟨ Cases for $else\_like$ 148 ⟩ **break**;
      case $else\_head$: ⟨ Cases for $else\_head$ 149 ⟩ **break**;
      case if_clause: ⟨ Cases for if_clause 150 ⟩ **break**;
      case if_head: ⟨ Cases for if_head 151 ⟩ **break**;
      case do_like: ⟨ Cases for do_like 152 ⟩ **break**;
      case $case\_like$: ⟨ Cases for $case\_like$ 153 ⟩ **break**;
      case $catch\_like$: ⟨ Cases for $catch\_like$ 154 ⟩ **break**;
      case tag: ⟨ Cases for tag 155 ⟩ **break**;
      case $stmt$: ⟨ Cases for $stmt$ 157 ⟩ **break**;
      case $semi$: ⟨ Cases for $semi$ 158 ⟩ **break**;
      case $lproc$: ⟨ Cases for $lproc$ 159 ⟩ **break**;

```
          case section_scrap: ⟨Cases for section_scrap 160⟩ break;
          case insert: ⟨Cases for insert 161⟩ break;
          case prelangle: ⟨Cases for prelangle 162⟩ break;
          case prerangle: ⟨Cases for prerangle 163⟩ break;
          case langle: ⟨Cases for langle 164⟩ break;
          case template_like: ⟨Cases for template_like 165⟩ break;
          case new_like: ⟨Cases for new_like 166⟩ break;
          case new_exp: ⟨Cases for new_exp 167⟩ break;
          case ftemplate: ⟨Cases for ftemplate 168⟩ break;
          case for_like: ⟨Cases for for_like 169⟩ break;
          case raw_ubin: ⟨Cases for raw_ubin 170⟩ break;
          case const_like: ⟨Cases for const_like 171⟩ break;
          case raw_int: ⟨Cases for raw_int 172⟩ break;
          case operator_like: ⟨Cases for operator_like 173⟩ break;
          case typedef_like: ⟨Cases for typedef_like 174⟩ break;
          case delete_like: ⟨Cases for delete_like 175⟩ break;
          case question: ⟨Cases for question 176⟩ break;
          case alignas_like: ⟨Cases for alignas_like 177⟩ break;
          case lbrack: ⟨Cases for lbrack 178⟩ break;
          case attr_head: ⟨Cases for attr_head 179⟩ break;
          case attr: ⟨Cases for attr 180⟩ break;
          case default_like: ⟨Cases for default_like 181⟩ break;
          }
      pp++;       /* if no match was found, we move to the right */
   }
```

This code is used in section 185.

123.   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 overloaded 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 8⟩ +≡
  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**);

124.   **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* ≥ *text_ptr*) *confusion*("`find_first_ident`");
    for (*j* ← *p*; *j* < *(*p* + 1); *j*++) {
      r ← *j* % *id_flag*;
      switch (*j*/*id_flag*) {
      case 2:   /* *res_flag* */
        if (*name_dir*[*r*].*ilk* ≡ *case_like*) **return** *case_found*;
        if (*name_dir*[*r*].*ilk* ≡ *operator_like*) **return** *operator_found*;
        if (*name_dir*[*r*].*ilk* ≠ *raw_int*) **break**;
      case 1: **return** *j*;
      case 4: **case** 5:   /* *tok_flag* or *inner_tok_flag* */
        if ((*q* ← *find_first_ident*(*tok_start* + *r*)) ≠ *no_ident_found*) **return** *q*;
      default: ;   /* char, *section_flag*, fall thru: move on to next token */
        if (*j* ≡ *inserted*) **return** *no_ident_found*;   /* ignore inserts */
        else if (*j* ≡ *qualifier*) *j*++;   /* bypass namespace qualifier */
      }
    }
    return *no_ident_found*;
  }

125.   The scraps currently being parsed must be inspected for any occurrence of the identifier that we're making reserved; hence the **for** loop below.
  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 */
    token_pointer *tok_loc*;   /* pointer to *tok_value* */

    if ((*tok_loc* ← *find_first_ident*(*p⃗trans*)) ≤ *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*;
  }

126.   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 *p→trans* */
>   **scrap_pointer** *p*)
> {
>   **token_pointer** *tok_loc*;   /* where the first identifier appears */
>   **if** ((*tok_loc* ← *find_first_ident*(*p→trans*)) ≤ *operator_found*) **return**;
>   /* this happens, for example, in **case** *found*: */
>   *xref_switch* ← *def_flag*;
>   *underline_xref*(*tok_loc % *id_flag* + *name_dir*);
> }

127.   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 128⟩
> }

128.   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 $p\rightarrow xref$), but rather right before q.

$\langle$ Insert new cross-reference at q, not at beginning of list $128 \rangle \equiv$

   $append\_xref(0);$ /∗ this number doesn't matter ∗/

   $xref\_ptr\rightarrow xlink \leftarrow (\textbf{xref\_pointer})\ p\rightarrow xref\,;$

   $r \leftarrow xref\_ptr\,;$

   $update\_node(p);$

   $\textbf{while}\ (r\rightarrow xlink \neq q)\ \{$

     $r\rightarrow num \leftarrow r\rightarrow xlink\rightarrow num\,;$

     $r \leftarrow r\rightarrow xlink\,;$

   $\}$

   $r\rightarrow num \leftarrow m;$ /∗ everything from q on is left undisturbed ∗/

This code is used in section 127.

129.   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* 129 ⟩ ≡

  if (*cat1* ≡ *lbrace* ∨ *cat1* ≡ *int_like* ∨ *cat1* ≡ *decl*) {

    make_underlined(*pp*);

    big_app(*dindent*);

    big_app1(*pp*);

    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*);

    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_insert(*pp* + 2, '␣');

    reduce(*pp* + 2, 2, *rpar*, 0, 9);

  }

  else if (*cat1* ≡ *cast* ∧ (*cat2* ≡ *const_like* ∨ *cat2* ≡ *case_like*)) {

    big_app1_insert(*pp* + 1, '␣');

    reduce(*pp* + 1, 2, *cast*, 0, 9);

  }

  else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *cast*) *squash*(*pp*, 2, *exp*, −2, 10);

  else if (*cat1* ≡ *attr*) {

    big_app1_insert(*pp*, '␣');

    reduce(*pp*, 2, *exp*, −2, 142);

  }

  else if (*cat1* ≡ *colcol* ∧ *cat2* ≡ *int_like*) *squash*(*pp*, 3, *int_like*, −2, 152);

This code is used in section 122.

130. ⟨Cases for *lpar* 130⟩ ≡
  if $((cat1 \equiv exp \lor cat1 \equiv ubinop) \land cat2 \equiv rpar)$ $squash(pp, 3, exp, -2, 11)$;
  else if $(cat1 \equiv rpar)$ {
    $big\_app1(pp)$;
    $app('\backslash\backslash')$;
    $app(',')$;
    $big\_app1(pp + 1)$;
    $reduce(pp, 2, exp, -2, 12)$;
  }
  else if $((cat1 \equiv decl\_head \lor cat1 \equiv int\_like \lor cat1 \equiv cast) \land cat2 \equiv rpar)$ $squash(pp, 3, cast, -2, 13)$;
  else if $((cat1 \equiv decl\_head \lor cat1 \equiv int\_like \lor cat1 \equiv exp) \land cat2 \equiv comma)$ {
    $big\_app3(pp)$;
    $app(opt)$;
    $app('9')$;
    $reduce(pp, 3, lpar, -1, 14)$;
  }
  else if $(cat1 \equiv stmt \lor cat1 \equiv decl)$ {
    $big\_app2(pp)$;
    $big\_app('\sqcup')$;
    $reduce(pp, 2, lpar, -1, 15)$;
  }
This code is used in section 122.

131. ⟨Cases for *unop* 131⟩ ≡
  if $(cat1 \equiv exp \lor cat1 \equiv int\_like)$ $squash(pp, 2, exp, -2, 16)$;
This code is used in section 122.

132. ⟨Cases for *ubinop* 132⟩ ≡
  if $(cat1 \equiv cast \land cat2 \equiv rpar)$ {
    $big\_app('\{')$;
    $big\_app1\_insert(pp, '\}')$;
    $reduce(pp, 2, cast, -2, 17)$;
  }
  else if $(cat1 \equiv exp \lor cat1 \equiv int\_like)$ {
    $big\_app('\{')$;
    $big\_app1\_insert(pp, '\}')$;
    $reduce(pp, 2, cat1, -2, 18)$;
  }
  else if $(cat1 \equiv binop)$ {
    $big\_app(math\_rel)$;
    $big\_app1\_insert(pp, '\{')$;
    $big\_app('\}')$;
    $big\_app('\}')$;
    $reduce(pp, 2, binop, -1, 19)$;
  }
This code is used in section 122.

133. ⟨ Cases for *binop* 133 ⟩ ≡
  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 122.

134. ⟨ Cases for *cast* 134 ⟩ ≡
  if (*cat1* ≡ *lpar*) *squash*(*pp*, 2, *lpar*, −1, 21);
  else if (*cat1* ≡ *exp*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *exp*, −2, 21);
  }
  else if (*cat1* ≡ *semi*) *squash*(*pp*, 1, *exp*, −2, 22);
This code is used in section 122.

135. ⟨ Cases for *sizeof_like* 135 ⟩ ≡
  if (*cat1* ≡ *cast*) *squash*(*pp*, 2, *exp*, −2, 23);
  else if (*cat1* ≡ *exp*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *exp*, −2, 24);
  }
This code is used in section 122.

136. ⟨ Cases for *int_like* 136 ⟩ ≡
  if (*cat1* ≡ *int_like* ∨ *cat1* ≡ *struct_like*) {
    big_app1_insert(*pp*, '␣');
    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 122.

137. ⟨ Cases for *public_like* 137 ⟩ ≡
  if (*cat1* ≡ *colon*) *squash*(*pp*, 2, *tag*, −1, 29);
  else *squash*(*pp*, 1, *int_like*, −2, 30);
This code is used in section 122.

138. ⟨ Cases for *colcol* 138 ⟩ ≡
  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 122.

139. ⟨ Cases for *decl_head* 139 ⟩ ≡
  if (*cat1* ≡ *comma*) {
    big_app2(*pp*);
    big_app('␣');
    reduce(*pp*, 2, *decl_head*, −1, 33);
  }
  else if (*cat1* ≡ *ubinop*) {
    big_app1_insert(*pp*, '{');
    big_app('}');
    reduce(*pp*, 2, *decl_head*, −1, 34);
  }
  else if (*cat1* ≡ *exp* ∧ *cat2* ≠ *lpar* ∧ *cat2* ≠ *lbrack* ∧ *cat2* ≠ *exp* ∧ *cat2* ≠ *cast*) {
    make_underlined(*pp* + 1);
    squash(*pp*, 2, *decl_head*, −1, 35);
  }
  else if ((*cat1* ≡ *binop* ∨ *cat1* ≡ *colon*) ∧ *cat2* ≡ *exp* ∧ (*cat3* ≡ *comma* ∨ *cat3* ≡ *semi* ∨ *cat3* ≡ *rpar*))
    squash(*pp*, 3, *decl_head*, −1, 36);
  else if (*cat1* ≡ *cast*) *squash*(*pp*, 2, *decl_head*, −1, 37);
  else if (*cat1* ≡ *lbrace* ∨ *cat1* ≡ *int_like* ∨ *cat1* ≡ *decl*) {
    big_app(*dindent*);
    big_app1(*pp*);
    reduce(*pp*, 1, *fn_decl*, 0, 38);
  }
  else if (*cat1* ≡ *semi*) *squash*(*pp*, 2, *decl*, −1, 39);
  else if (*cat1* ≡ *attr*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *decl_head*, −1, 139);
  }
This code is used in section 122.

140. ⟨ Cases for *decl* 140 ⟩ ≡
  if (*cat1* ≡ *decl*) {
    big_app1_insert(*pp*, *force*);
    reduce(*pp*, 2, *decl*, −1, 40);
  }
  else if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *function*) {
    big_app1_insert(*pp*, *big_force*);
    reduce(*pp*, 2, *cat1*, −1, 41);
  }
This code is used in section 122.

141.   ⟨ Cases for *base* 141 ⟩ ≡
  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_insert (*pp*, ' ⌴ ');
      big_app (' ⌴ ');
      big_app1 (*pp* + 2);
      reduce (*pp*, 3, *lbrace*, −2, 43);
    }
  }

This code is used in section 122.

142.   ⟨ Cases for *struct_like* 142 ⟩ ≡
  if (*cat1* ≡ *lbrace*) {
    big_app1_insert (*pp*, ' ⌴ ');
    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);
      big_app1_insert (*pp*, ' ⌴ ');
      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_insert (*pp*, ' ⌴ ');
      reduce (*pp*, 2, *int_like*, −2, 48);
    }
  }
  else if (*cat1* ≡ *attr*) {
    big_app1_insert (*pp*, ' ⌴ ');
    reduce (*pp*, 2, *struct_like*, −3, 141);
  }
  else if (*cat1* ≡ *struct_like*) {
    big_app1_insert (*pp*, ' ⌴ ');
    reduce (*pp*, 2, *struct_like*, −3, 151);
  }

This code is used in section 122.

143.   ⟨Cases for *struct_head* 143⟩ ≡
  if $((\mathit{cat1} \equiv \mathit{decl} \vee \mathit{cat1} \equiv \mathit{stmt} \vee \mathit{cat1} \equiv \mathit{function}) \wedge \mathit{cat2} \equiv \mathit{rbrace})$ {
    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 122.

144.   ⟨Cases for *fn_decl* 144⟩ ≡
  if (*cat1* ≡ *decl*) {
    big_app1_insert (*pp*, *force*);
    reduce (*pp*, 2, *fn_decl*, 0, 51);
  }
  else if (*cat1* ≡ *stmt*) {
    big_app1 (*pp*);
    app (*outdent*);
    app (*outdent*);
    big_app (*force*);
    big_app1 (*pp* + 1);
    reduce (*pp*, 2, *function*, −1, 52);
  }
  else if (*cat1* ≡ *attr*) {
    big_app1_insert (*pp*, '␣');
    reduce (*pp*, 2, *fn_decl*, 0, 157);
  }
This code is used in section 122.

145.   ⟨Cases for *function* 145⟩ ≡
  if (*cat1* ≡ *function* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *stmt*) {
    big_app1_insert (*pp*, *big_force*);
    reduce (*pp*, 2, *cat1*, −1, 53);
  }
This code is used in section 122.

146.   ⟨ Cases for *lbrace* 146 ⟩ ≡
  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 122.

147.   ⟨ Cases for *if_like* 147 ⟩ ≡
  if (*cat1* ≡ *exp*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *if_clause*, 0, 57);
  }
This code is used in section 122.

148.   ⟨ Cases for *else_like* 148 ⟩ ≡
  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 122.

149.   ⟨Cases for *else_head* 149⟩ ≡
  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 122.

150.   ⟨Cases for *if_clause* 150⟩ ≡
  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);
  }
  else if (*cat1* ≡ *attr*) {
    big_app1_insert(*pp*, ' ');
    reduce(*pp*, 2, *if_head*, 0, 146);
  }

This code is used in section 122.

151.   ⟨Cases for *if_head* 151⟩ ≡
  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_insert(*pp* + 1, *force*);
      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 122.

152.   ⟨Cases for *do_like* 152⟩ ≡
  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 122.

153.   ⟨Cases for *case_like* 153⟩ ≡
  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_insert(*pp*, '␣');
    reduce(*pp*, 2, *exp*, −2, 72);
  }

This code is used in section 122.

154.   ⟨Cases for *catch_like* 154⟩ ≡
  if (*cat1* ≡ *cast* ∨ *cat1* ≡ *exp*) {
    big_app1_insert(*pp*, *dindent*);
    reduce(*pp*, 2, *fn_decl*, 0, 73);
  }

This code is used in section 122.

155.   ⟨Cases for *tag* 155⟩ ≡
  if (*cat1* ≡ *tag*) {
    big_app1_insert(*pp*, *break_space*);
    reduce(*pp*, 2, *tag*, −1, 74);
  }
  else if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *function*) {
    big_app(*force*);
    big_app(*backup*);
    big_app1_insert(*pp*, *break_space*);
    reduce(*pp*, 2, *cat1*, −1, 75);
  }
  else if (*cat1* ≡ *rbrace*) *squash*(*pp*, 1, *decl*, −1, 156);
This code is used in section 122.

156.   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 24⟩ +≡
  force_lines ← *true*;

157.   ⟨Cases for *stmt* 157⟩ ≡
  if (*cat1* ≡ *stmt* ∨ *cat1* ≡ *decl* ∨ *cat1* ≡ *function*) {
    big_app1_insert(*pp*, (*cat1* ≡ *function* ∨ *cat1* ≡ *decl*) ? *big_force* : *force_lines* ? *force* : *break_space*);
    reduce(*pp*, 2, *cat1*, −1, 76);
  }
This code is used in section 122.

158.   ⟨Cases for *semi* 158⟩ ≡
  big_app('␣');
  big_app1(*pp*);
  reduce(*pp*, 1, *stmt*, −1, 77);
This code is used in section 122.

159.   ⟨ Cases for *lproc* 159 ⟩ ≡
  if (*cat1* ≡ *define_like*) *make_underlined*(*pp* + 2);
  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_insert(*pp*, '␣');
        app_str("\\5");
        big_app2(*pp* + 2);
        reduce(*pp*, 4, *insert*, −1, 80);
     }
  }
This code is used in section 122.

160.   ⟨ Cases for *section_scrap* 160 ⟩ ≡
  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 122.

161.   ⟨ Cases for *insert* 161 ⟩ ≡
  if (*cat1*) *squash*(*pp*, 2, *cat1*, 0, 83);
This code is used in section 122.

162.   ⟨ Cases for *prelangle* 162 ⟩ ≡
  init_mathness ← *cur_mathness* ← *yes_math*;
  app('<');
  reduce(*pp*, 1, *binop*, −2, 84);
This code is used in section 122.

163.   ⟨ Cases for *prerangle* 163 ⟩ ≡
  init_mathness ← *cur_mathness* ← *yes_math*;
  app('>');
  reduce(*pp*, 1, *binop*, −2, 85);
This code is used in section 122.

164.   #**define** *reserve_typenames* *flags*[′t′]
                /∗ should we treat **typename** in a template like **typedef**? ∗/
⟨ Cases for *langle* 164 ⟩ ≡
   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);
      }
   }
   else if (*cat1* ≡ *struct_like*) {
      if ((*cat2* ≡ *exp* ∨ *cat2* ≡ *int_like*) ∧ (*cat3* ≡ *comma* ∨ *cat3* ≡ *prerangle*)) {
         make_underlined (*pp* + 2);
         if (*reserve_typenames*) *make_reserved* (*pp* + 2);
         big_app2 (*pp*);
         big_app(′␣′);
         big_app2 (*pp* + 2);
         if (*cat3* ≡ *comma*) *reduce*(*pp*, 4, *langle*, 0, 153);
         else *reduce*(*pp*, 4, *cast*, −1, 154);
      }
   }
This code is used in section 122.

165.   ⟨ Cases for *template_like* 165 ⟩ ≡
   if (*cat1* ≡ *exp* ∧ *cat2* ≡ *prelangle*) *squash*(*pp* + 2, 1, *langle*, 2, 89);
   else if (*cat1* ≡ *exp* ∨ *cat1* ≡ *raw_int*) {
      big_app1_insert (*pp*, ′␣′);
      reduce(*pp*, 2, *cat1*, −2, 90);
   }
   else if (*cat1* ≡ *cast* ∧ *cat2* ≡ *struct_like*) {
      big_app1_insert (*pp*, ′␣′);
      reduce(*pp*, 2, *struct_like*, 0, 155);
   }
   else *squash*(*pp*, 1, *raw_int*, 0, 91);
This code is used in section 122.

166.   ⟨Cases for *new_like* 166⟩ ≡
  if (*cat1* ≡ *lpar* ∧ *cat2* ≡ *exp* ∧ *cat3* ≡ *rpar*) *squash*(*pp*, 4, *new_like*, 0, 92);
  else if (*cat1* ≡ *cast*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *exp*, −2, 93);
  }
  else if (*cat1* ≠ *lpar*) *squash*(*pp*, 1, *new_exp*, 0, 94);
This code is used in section 122.

167.   ⟨Cases for *new_exp* 167⟩ ≡
  if (*cat1* ≡ *int_like* ∨ *cat1* ≡ *const_like*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *new_exp*, 0, 95);
  }
  else if (*cat1* ≡ *struct_like* ∧ (*cat2* ≡ *exp* ∨ *cat2* ≡ *int_like*)) {
    big_app1_insert(*pp*, '␣');
    big_app('␣');
    big_app1(*pp* + 2);
    reduce(*pp*, 3, *new_exp*, 0, 96);
  }
  else if (*cat1* ≡ *raw_ubin*) {
    big_app1_insert(*pp*, '{');
    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 122.

168.   ⟨Cases for *ftemplate* 168⟩ ≡
  if (*cat1* ≡ *prelangle*) *squash*(*pp* + 1, 1, *langle*, 1, 100);
  else *squash*(*pp*, 1, *exp*, −2, 101);
This code is used in section 122.

169.   ⟨Cases for *for_like* 169⟩ ≡
  if (*cat1* ≡ *exp*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *else_like*, −2, 102);
  }
This code is used in section 122.

170.   ⟨ Cases for *raw_ubin* 170 ⟩ ≡
   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 122.

171.   ⟨ Cases for *const_like* 171 ⟩ ≡
   squash (*pp*, 1, *int_like*, −2, 105);
This code is used in section 122.

172.   ⟨ Cases for *raw_int* 172 ⟩ ≡
   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* ≡ *lbrack*) *squash* (*pp*, 1, *exp*, −2, 144);
   else if (*cat1* ≠ *langle*) *squash* (*pp*, 1, *int_like*, −3, 110);
This code is used in section 122.

173.   ⟨ Cases for *operator_like* 173 ⟩ ≡
   if (*cat1* ≡ *binop* ∨ *cat1* ≡ *unop* ∨ *cat1* ≡ *ubinop*) {
     if (*cat2* ≡ *binop*) **break**;
     big_app1_insert (*pp*, '{');
     big_app ('}');
     reduce (*pp*, 2, *exp*, −2, 111);
   }
   else if (*cat1* ≡ *new_like* ∨ *cat1* ≡ *delete_like*) {
     big_app1_insert (*pp*, '⎵');
     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 122.

174. ⟨ Cases for *typedef_like* 174 ⟩ ≡
 if ((*cat1* ≡ *int_like* ∨ *cat1* ≡ *cast*) ∧ (*cat2* ≡ *comma* ∨ *cat2* ≡ *semi*)) *squash*(*pp* + 1, 1, *exp*, −1, 115);
 else if (*cat1* ≡ *int_like*) {
  big_app1_insert(*pp*, '␣');
  reduce(*pp*, 2, *typedef_like*, 0, 116);
 }
 else if (*cat1* ≡ *exp* ∧ *cat2* ≠ *lpar* ∧ *cat2* ≠ *exp* ∧ *cat2* ≠ *cast*) {
  make_underlined(*pp* + 1);
  make_reserved(*pp* + 1);
  big_app1_insert(*pp*, '␣');
  reduce(*pp*, 2, *typedef_like*, 0, 117);
 }
 else if (*cat1* ≡ *comma*) {
  big_app2(*pp*);
  big_app('␣');
  reduce(*pp*, 2, *typedef_like*, 0, 118);
 }
 else if (*cat1* ≡ *semi*) *squash*(*pp*, 2, *decl*, −1, 119);
 else if (*cat1* ≡ *ubinop* ∧ (*cat2* ≡ *ubinop* ∨ *cat2* ≡ *cast*)) {
  big_app('{');
  big_app1_insert(*pp* + 1, '}');
  reduce(*pp* + 1, 2, *cat2*, 0, 120);
 }
This code is used in section 122.

175. ⟨ Cases for *delete_like* 175 ⟩ ≡
 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_insert(*pp*, '␣');
  reduce(*pp*, 2, *exp*, −2, 122);
 }
This code is used in section 122.

176. ⟨ Cases for *question* 176 ⟩ ≡
 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 122.

177. ⟨ Cases for *alignas_like* 177 ⟩ ≡
 if (*cat1* ≡ *decl_head*) *squash*(*pp*, 2, *attr*, −1, 126);
 else if (*cat1* ≡ *exp*) *squash*(*pp*, 2, *attr*, −1, 127);
 else if (*cat1* ≡ *cast*) *squash*(*pp*, 2, *attr*, −1, 158);
This code is used in section 122.

178.   ⟨ Cases for *lbrack* 178 ⟩ ≡
  if (*cat1* ≡ *lbrack*)
    if (*cat2* ≡ *rbrack* ∧ *cat3* ≡ *rbrack*) *squash*(*pp*, 4, *exp*, −2, 147);
    else *squash*(*pp*, 2, *attr_head*, −1, 128);
  else *squash*(*pp*, 1, *lpar*, −1, 129);
This code is used in section 122.

179.   ⟨ Cases for *attr_head* 179 ⟩ ≡
  if (*cat1* ≡ *rbrack* ∧ *cat2* ≡ *rbrack*) *squash*(*pp*, 3, *attr*, −1, 131);
  else if (*cat1* ≡ *exp*) *squash*(*pp*, 2, *attr_head*, 0, 132);
  else if (*cat1* ≡ *using_like* ∧ *cat2* ≡ *exp* ∧ *cat3* ≡ *colon*) {
    big_app2(*pp*);
    big_app('␣');
    big_app2(*pp* + 2);
    big_app('␣');
    reduce(*pp*, 4, *attr_head*, 0, 133);
  }
  else if (*cat1* ≡ *comma*) *squash*(*pp*, 2, *attr_head*, 0, 145);
This code is used in section 122.

180.   ⟨ Cases for *attr* 180 ⟩ ≡
  if (*cat1* ≡ *lbrace* ∨ *cat1* ≡ *stmt*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *cat1*, −2, 134);
  }
  else if (*cat1* ≡ *tag*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *tag*, −1, 135);
  }
  else if (*cat1* ≡ *semi*) *squash*(*pp*, 2, *stmt*, −2, 136);
  else if (*cat1* ≡ *attr*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *attr*, −1, 137);
  }
  else if (*cat1* ≡ *decl_head*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *decl_head*, −1, 138);
  }
  else if (*cat1* ≡ *typedef_like*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *typedef_like*, 0, 143);
  }
  else if (*cat1* ≡ *function*) {
    big_app1_insert(*pp*, '␣');
    reduce(*pp*, 2, *function*, −1, 148);
  }
This code is used in section 122.

181.   ⟨ Cases for *default_like* 181 ⟩ ≡
  if (*cat1* ≡ *colon*) *squash*(*pp*, 1, *case_like*, −3, 149);
  else *squash*(*pp*, 1, *exp*, −2, 150);
This code is used in section 122.

182.   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 8 ⟩ +≡
   static void *reduce*(**scrap_pointer**, **short**, **eight_bits**, **short**, **short**);
   static void *squash*(**scrap_pointer**, **short**, **eight_bits**, **short**, **short**);

183.   **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 188 ⟩
    pp −−;   /∗ we next say pp++ ∗/
  }

184.   Here's the *squash* procedure, which takes advantage of the simplification that occurs when $k ≡ 1$.
  static void *squash*(**scrap_pointer** j, **short** k, **eight_bits** c, **short** d, **short** n)
  {
    scrap_pointer i;   /∗ pointers into scrap memory ∗/
    if ($k ≡ 1$) {
      j→*cat* ← c;
      pp ← ($pp + d < scrap\_base$? $scrap\_base$: $pp + d$);
      ⟨ Print a snapshot of the scrap list if debugging 188 ⟩
      pp −−;   /∗ we next say pp++ ∗/
      return;
    }
    for ($i ← j$; $i < j + k$; i++) *big_app1*(i);
    reduce(j, k, c, d, n);
  }

185.   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 185 ⟩ ≡
  while (*true*) {
    ⟨ Make sure the entries *pp* through *pp* + 3 of *cat* are defined 186 ⟩
    if (*tok_ptr* + *safe_tok_incr* > *tok_mem_end*) {
      if (*tok_ptr* > *max_tok_ptr*) *max_tok_ptr* ← *tok_ptr*;
      overflow("token");
    }
    if (*text_ptr* + *safe_text_incr* > *tok_start_end*) {
      if (*text_ptr* > *max_text_ptr*) *max_text_ptr* ← *text_ptr*;
      overflow("text");
    }
    if (*pp* > *lo_ptr*) **break**;
    init_mathness ← *cur_mathness* ← *maybe_math*;
    ⟨ Match a production at *pp*, or increase *pp* if there is no match 122 ⟩
  }

This code is used in section 189.

186.   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 186 ⟩ ≡
  if (*lo_ptr* < *pp* + 3) {
    while (*hi_ptr* ≤ *scrap_ptr* ∧ *lo_ptr* ≠ *pp* + 3) {
      (++*lo_ptr*)→*cat* ← *hi_ptr*→*cat*;
      lo_ptr→*mathness* ← (*hi_ptr*)→*mathness*;
      lo_ptr→*trans* ← (*hi_ptr*++)→*trans*;
    }
    for (*i* ← *lo_ptr* + 1; *i* ≤ *pp* + 3; *i*++) *i*→*cat* ← 0;
  }

This code is used in section 185.

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

#**define** *off* 0
#**define** *partly* 1
#**define** *fully* 2

⟨ Private variables 21 ⟩ +≡
  static int *tracing* ← *off*;   /* can be used to show parsing details */

188.   ⟨ Print a snapshot of the scrap list if debugging 188 ⟩ ≡
　if (*tracing* ≡ *fully*) {
　　scrap_pointer *k_l*;　　/∗ pointer into *scrap_info* ∗/
　　printf ("\n%d:", *n*);
　　for (*k_l* ← *scrap_base*; *k_l* ≤ *lo_ptr*; *k_l*++) {
　　　if (*k_l* ≡ *pp*) *putchar* ('*');
　　　else *putchar* ('␣');
　　　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*) *printf* ("...");　　/∗ indicate that more is coming ∗/
　}

This code is used in sections 183 and 184.

189.   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 ≤ *max_texts* and *tok_ptr* + 6 ≤ *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* ∗/
　　scrap_pointer *j*;　　/∗ runs through final scraps ∗/
　　pp ← *scrap_base*;
　　lo_ptr ← *pp* − 1;
　　hi_ptr ← *pp*;
　　⟨ If tracing, print an indication of where we are 193 ⟩
　　⟨ Reduce the scraps using the productions until no more rules apply 185 ⟩
　　⟨ Combine the irreducible scraps that remain 191 ⟩
　}

190.   ⟨ Predeclaration of procedures 8 ⟩ +≡ **static text_pointer** *translate*(**void**);

191.   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 191 ⟩ ≡
  {
    ⟨ If semi-tracing, show the irreducible scraps 192 ⟩
    for $(j \leftarrow scrap\_base;\ j \leq lo\_ptr;\ j\texttt{++})$ {
      if $(j \neq scrap\_base)$ $app(\texttt{'}_{\sqcup}\texttt{'})$;
      if $(j\rightarrow mathness \ \% \ 4 \equiv yes\_math)$ $app(\texttt{'\$'})$;
      $app1(j)$;
      if $(j\rightarrow mathness / 4 \equiv yes\_math)$ $app(\texttt{'\$'})$;
      if $(tok\_ptr + 6 > tok\_mem\_end)$ $overflow(\texttt{"token"})$;
    }
    $freeze\_text$;
    return $text\_ptr - 1$;
  }

This code is used in section 189.

192.   ⟨ If semi-tracing, show the irreducible scraps 192 ⟩ ≡
  if $(lo\_ptr > scrap\_base \wedge tracing \equiv partly)$ {
    $printf(\texttt{"\textbackslash nIrreducible}_{\sqcup}\texttt{scrap}_{\sqcup}\texttt{sequence}_{\sqcup}\texttt{in}_{\sqcup}\texttt{section}_{\sqcup}\texttt{\%d:"}, section\_count)$;
    $mark\_harmless$;
    for $(j \leftarrow scrap\_base;\ j \leq lo\_ptr;\ j\texttt{++})$ {
      $putchar(\texttt{'}_{\sqcup}\texttt{'})$;
      $print\_cat(j\rightarrow cat)$;
    }
  }

This code is used in section 191.

193.   ⟨ If tracing, print an indication of where we are 193 ⟩ ≡
  if $(tracing \equiv fully)$ {
    $printf(\texttt{"\textbackslash nTracing}_{\sqcup}\texttt{after}_{\sqcup}\texttt{l.}_{\sqcup}\texttt{\%d:\textbackslash n"}, cur\_line)$;
    $mark\_harmless$;
    if $(loc > buffer + 50)$ {
      $printf(\texttt{"..."})$;
      $term\_write(loc - 51, 51)$;
    }
    else $term\_write(buffer, loc - buffer)$;
  }

This code is used in section 189.

194.   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*)
> {
>   **while** (*next_control* < *format_code* ∨ *next_control* ≡ *spec_ctrl*) {
>   ⟨Append the scrap appropriate to *next_control* 197⟩
>   *next_control* ← *get_next*( );
>   **if** (*next_control* ≡ '|' ∨ *next_control* ≡ *begin_comment* ∨ *next_control* ≡ *begin_short_comment*)
>   **return**;
>   }
> }

195.   ** ⟨Predeclaration of procedures 8⟩ +≡   **static void *C_parse*(**eight_bits**);

**196.   ** 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*;
>   }

197.   ⟨Append the scrap appropriate to *next_control* 197⟩ ≡
  ⟨Make sure that there is room for the new scraps, tokens, and texts 198⟩
  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 200⟩ **break**;
  case *identifier*: *app_cur_id*(*true*); **break**;
  case *TEX_string*: ⟨Append a TEX string, without forming a scrap 201⟩ **break**;
  case '/': **case** '.': *app*(*next_control*); *app_scrap*(*binop*, *yes_math*); **break**;
  case '<': *app_str*("\\langle"); *app_scrap*(*prelangle*, *yes_math*); **break**;
  case '>': *app_str*("\\rangle"); *app_scrap*(*prerangle*, *yes_math*); **break**;
  case '=': *app_str*("\\K"); *app_scrap*(*binop*, *yes_math*); **break**;
  case '|': *app_str*("\\OR"); *app_scrap*(*binop*, *yes_math*); **break**;
  case '^': *app_str*("\\XOR"); *app_scrap*(*binop*, *yes_math*); **break**;
  case '%': *app_str*("\\MOD"); *app_scrap*(*binop*, *yes_math*); **break**;
  case '!': *app_str*("\\R"); *app_scrap*(*unop*, *yes_math*); **break**;
  case '~': *app_str*("\\CM"); *app_scrap*(*unop*, *yes_math*); **break**;
  case '+': **case** '−': *app*(*next_control*); *app_scrap*(*ubinop*, *yes_math*); **break**;
  case '*': *app*(*next_control*); *app_scrap*(*raw_ubin*, *yes_math*); **break**;
  case '&': *app_str*("\\AND"); *app_scrap*(*raw_ubin*, *yes_math*); **break**;
  case '?': *app_str*("\\?"); *app_scrap*(*question*, *yes_math*); **break**;
  case '#': *app_str*("\\#"); *app_scrap*(*ubinop*, *yes_math*); **break**;
  case *ignore*: **case** *xref_roman*: **case** *xref_wildcard*: **case** *xref_typewriter*: **case** *noop*: **break**;
  case '(': *app*(*next_control*); *app_scrap*(*lpar*, *maybe_math*); **break**;
  case ')': *app*(*next_control*); *app_scrap*(*rpar*, *maybe_math*); **break**;
  case '[': *app*(*next_control*); *app_scrap*(*lbrack*, *maybe_math*); **break**;
  case ']': *app*(*next_control*); *app_scrap*(*rbrack*, *maybe_math*); **break**;
  case '{': *app_str*("\\{"); *app_scrap*(*lbrace*, *yes_math*); **break**;
  case '}': *app_str*("\\}"); *app_scrap*(*rbrace*, *yes_math*); **break**;
  case ',': *app*(','); *app_scrap*(*comma*, *yes_math*); **break**;
  case ';': *app*(';'); *app_scrap*(*semi*, *maybe_math*); **break**;
  case ':': *app*(':'); *app_scrap*(*colon*, *no_math*); **break**;
  ⟨Cases involving nonstandard characters 199⟩
  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 194.

198. ⟨ Make sure that there is room for the new scraps, tokens, and texts 198 ⟩ ≡
   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 197 and 206.

199. 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 199 ⟩ ≡
   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 197.

200.   The following code must use *app_tok* instead of *app* in order to protect against overflow. Note that *tok_ptr* + 1 ≤ *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 200 ⟩ ≡
```
{ int count ← −1;      /* characters remaining before string break */

   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('}');
   app_scrap(exp, maybe_math);
}
```
This code is used in section 197.

201.   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 201 ⟩ ≡
  app_str("\\hbox{");
  while (*id_first* < *id_loc*)
    if ((**eight_bits**)(∗*id_first*) > °*177*) {
      app_tok(*quoted_char*);
      app_tok((**eight_bits**)(∗*id_first*++));
    }
    else {
      if (∗*id_first* ≡ '@') *id_first*++;
      app_tok(∗*id_first*++);
    }
  app('}');
This code is used in section 197.

202.   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 8 ⟩ +≡
  static void *app_cur_id*(**boolean**);
  static text_pointer *C_translate*(**void**);
  static void *outer_parse*(**void**);

203.   **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 (*p*→*ilk* ≤ *custom*) {   /∗ not a reserved word ∗/
      app(*id_flag* + (**int**)(*p* − *name_dir*));
      if (*scrapping*)
        app_scrap(*p*→*ilk* ≡ *func_template* ? *ftemplate* : *exp*, *p*→*ilk* ≡ *custom* ? *yes_math* : *maybe_math*);
    }
    else {
      app(*res_flag* + (**int**)(*p* − *name_dir*));
      if (*scrapping*) {
        if (*p*→*ilk* ≡ *alfop*) *app_scrap*(*ubinop*, *yes_math*)
        else *app_scrap*(*p*→*ilk*, *maybe_math*);
      }
    }
  }

204.   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* ← *scrap_base*;   /∗ holds original value of *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*;
> }

205.   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`. Although `cwebmac` 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 24 ⟩ +≡
  make_pb ← *true*;

206.   **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 198⟩
        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*));
          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 ∗/
      }
  }

207.   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'.

208.   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 22 ⟩ +≡
  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**;

209.   #**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 21 ⟩ +≡
  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* ∗/

210.   ⟨ Set initial values 24 ⟩ +≡
  max_stack_ptr ← *stack*;

211.   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 8 ⟩ +≡
  static void *push_level*(**text_pointer**);
  static void *pop_level*(**void**);

212.   **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);
  }

213.   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*;
  }

214.   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 21 ⟩ +≡
  static name_pointer *cur_name*;

215.   #**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 8 ⟩ +≡
  static eight_bits *get_output*(**void**);
  static void *output_C*(**void**);
  static void *make_output*(**void**);

216.   **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 ← *(*cur_tok*++);
    if (*a* ≥ °*400*) {
      cur_name ← *a* % *id_flag* + *name_dir*;
      switch (*a*/*id_flag*) {
      case 2: **return** *res_word*;   /* *a* ≡ *res_flag* + *cur_name* */
      case 3: **return** *section_code*;   /* *a* ≡ *section_flag* + *cur_name* */
      case 4: *push_level*(*a* % *id_flag* + *tok_start*);
        goto *restart*;   /* *a* ≡ *tok_flag* + *cur_name* */
      case 5: *push_level*(*a* % *id_flag* + *tok_start*);
        cur_mode ← *inner*;
        goto *restart*;   /* *a* ≡ *inner_tok_flag* + *cur_name* */
      default: **return** *identifier*;   /* *a* ≡ *id_flag* + *cur_name* */
      }
    }
    return (**eight_bits**) *a*;
  }

217.   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* ← *tok_ptr*;
    text_pointer *save_text_ptr* ← *text_ptr*;
    sixteen_bits *save_next_control* ← *next_control*;   /* values to be restored */
    text_pointer *p*;   /* translation of the C text */
    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 */
  }

218.   Here is CWEAVE's major output handler.

```
static void make_output(void)      /* outputs the equivalents of tokens */
{
    eight_bits a ← 0;       /* current output byte */
    eight_bits 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 */
    boolean dindent_pending ← false;        /* should a dindent be output? */

    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 219⟩
            break;
        case section_code: ⟨Output a section name 223⟩
            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 222⟩
            goto reswitch;
        case dindent: a ← get_output();
            if (a ≠ big_force) {
                out_str("\\1\\1");
                goto reswitch;
            }
            else dindent_pending ← true;      /* fall through */
        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 220⟩
            break;
        case quoted_char: out(*(cur_tok++));
```

```
      case qualifier: break;
      default: out(a);      /* otherwise a is an ordinary character */
      }
    }
  }
```

219.   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 219 ⟩ ≡
```
  out('\\');
  if (a ≡ identifier) {
    if (cur_name→ilk ≡ custom ∧ ¬doing_format) {
    custom_out:
      for (p ← cur_name→byte_start; p < (cur_name + 1)→byte_start; p++)
        out(*p ≡ '_' ? 'x' : *p ≡ '$' ? 'X' : *p);
      break;
    }
    else if (is_tiny(cur_name)) out('|')
    else {
      delim ← '.';
      for (p ← cur_name→byte_start; p < (cur_name + 1)→byte_start; p++)
        if (xislower(*p)) {      /* not entirely uppercase */
          delim ← '\\';
          break;
        }
      out(delim);
    }
  }
  else if (cur_name→ilk ≡ alfop) {
    out('X');
    goto custom_out;
  }
  else out('&');      /* a ≡ res_word */
  if (is_tiny(cur_name)) {
    if (isxalpha((cur_name→byte_start)[0])) out('\\');
    out((cur_name→byte_start)[0]);
  }
  else out_name(cur_name, true);
```
This code is used in section 218.

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

$\langle$ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 220 $\rangle \equiv$
  if $(a < break\_space \lor a \equiv preproc\_line)$ {
    if $(cur\_mode \equiv outer)$ {
      $out(\text{'}\backslash\backslash\text{'})$;
      $out(a - cancel + \text{'0'})$;
      if $(a \equiv opt)$ {
        $b \leftarrow get\_output()$;   /∗ *opt* is followed by a digit ∗/
        if $(b \neq \text{'0'} \lor force\_lines \equiv false)$ $out(b)$
        else $out\_str(\texttt{"\{-1\}"})$;   /∗ *force_lines* encourages more @| breaks ∗/
      }
    }
    else if $(a \equiv opt)$ $b \leftarrow get\_output()$;   /∗ ignore digit following *opt* ∗/
  }
  else $\langle$ Look ahead for strongest line break, **goto** *reswitch* 221 $\rangle$
This code is used in section 218.

221.   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* 221 ⟩ ≡
```
  {
    b ← a;
    save_mode ← cur_mode;
    if (dindent_pending) {
      c ← 2;
      dindent_pending ← false;
    }
    else  c ← 0;
    while (true) {
      a ← get_output();
      if (a ≡ inserted) continue;
      if (a ≡ cancel ∨ a ≡ big_cancel) {
        ⟨ Output saved indent or outdent tokens 222 ⟩
        goto reswitch;       /∗ cancel overrides everything ∗/
      }
      if ((a ≠ '␣' ∧ a < indent) ∨ a ≡ backup ∨ a > big_force) {
        if (save_mode ≡ outer) {
          if (out_ptr > out_buf + 3 ∧ strncmp(out_ptr − 3, "\\Y\\B", 4) ≡ 0) goto reswitch;
          ⟨ Output saved indent or outdent tokens 222 ⟩
          out('\\');
          out(b − cancel + '0');
          if (a ≠ end_translation) finish_line();
        }
        else if (a ≠ end_translation ∧ cur_mode ≡ inner) out('␣');
        goto reswitch;
      }
      if (a ≡ indent) c++;
      else if (a ≡ outdent) c−−;
      else if (a ≡ opt) a ← get_output();
      else if (a > b) b ← a;      /∗ if a ≡ '␣' we have a < b ∗/
    }
  }
```
This code is used in section 220.

222.   ⟨ Output saved *indent* or *outdent* tokens 222 ⟩ ≡
```
  for ( ; c > 0; c−−)  out_str("\\1");
  for ( ; c < 0; c++)  out_str("\\2");
```
This code is used in sections 218 and 221.

223.   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 223 ⟩ ≡
```
  {
    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 224 ⟩
    if (an_output) out_str("␣}");
    out_str("\\X");
  }
```
This code is used in section 218.

224.   ⟨Output the text of the section name 224⟩ ≡
  $sprint\_section\_name(scratch, cur\_name)$;
  $k \leftarrow scratch$;
  $k\_limit \leftarrow scratch + strlen(scratch)$;
  $cur\_section\_name \leftarrow cur\_name$;
  while $(k < k\_limit)$ {
    $b \leftarrow *(k{+}{+})$;
    if $(b \equiv \text{'@'})$ ⟨Skip next character, give error if not '@' 225⟩
    if (an_output)
      switch (b) {
      case '␣': **case** '\\': **case** '#': **case** '%': **case** '$': **case** '^': **case** '{': **case** '}': **case** '~':
        case '&': **case** '_': $out(\text{'\\'})$;   /∗ falls through ∗/
      default: $out(b)$;
      }
    else if $(b \neq \text{'|'})$ $out(b)$
    else {
      ⟨Copy the C text into the *buffer* array 226⟩
      $save\_loc \leftarrow loc$;
      $save\_limit \leftarrow limit$;
      $loc \leftarrow limit + 2$;
      $limit \leftarrow j + 1$;
      $*limit \leftarrow \text{'|'}$;
      $output\_C(\,)$;
      $loc \leftarrow save\_loc$;
      $limit \leftarrow save\_limit$;
    }
  }
This code is used in section 223.

225.   ⟨Skip next character, give error if not '@' 225⟩ ≡
  if $(*k{+}{+} \neq \text{'@'})$ {
    $fputs(\text{"\\n!␣Illegal␣control␣code␣in␣section␣name:␣<"}, stdout)$;
    $print\_section\_name(cur\_section\_name)$;
    $printf(\text{">␣"})$;
    $mark\_error$;
  }
This code is used in section 224.

226.   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 226 ⟩ ≡
   $j \leftarrow limit + 1$;
   $*j \leftarrow$ '|';
   $delim \leftarrow 0$;
   while (*true*) {
     if ($k \geq k\_limit$) {
       fputs("\n!␣C␣text␣in␣section␣name␣didn't␣end:␣<", *stdout*);
       print_section_name(*cur_section_name*);
       printf(">␣");
       mark_error;
       break;
     }
     $b \leftarrow *(k{+}{+})$;
     if ($b \equiv$ '@' $\lor$ ($b \equiv$ '\\' $\land delim \neq 0$)) ⟨ Copy a quoted character into the buffer 227 ⟩
     else {
       if ($b \equiv$ '\'' $\lor b \equiv$ '"') {
         if ($delim \equiv 0$) $delim \leftarrow b$;
         else if ($delim \equiv b$) $delim \leftarrow 0$;
       }
       if ($b \neq$ '|' $\lor delim \neq 0$) {
         if ($j > buffer + long\_buf\_size - 3$) *overflow*("buffer");
         $*({+}{+}j) \leftarrow b$;
       }
       else break;
     }
   }

This code is used in section 224.

227.   ⟨ Copy a quoted character into the buffer 227 ⟩ ≡
  {
    if ($j > buffer + long\_buf\_size - 4$) *overflow*("buffer");
    $*({+}{+}j) \leftarrow b$;
    $*({+}{+}j) \leftarrow *(k{+}{+})$;
  }

This code is used in section 226.

228.   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*);
   section_count ← 0;
   format_visible ← *true*;
   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 231⟩
}

229.   ⟨Predeclaration of procedures 8⟩ +≡   **static void** *phase_two*(**void**);

230.   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 21⟩ +≡
  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? ∗/

231.   ⟨Translate the current section 231⟩ ≡
{
   section_count ++;
   ⟨Output the code for the beginning of a new section 232⟩
   save_position;
   ⟨Translate the TeX part of the current section 233⟩
   ⟨Translate the definition part of the current section 234⟩
   ⟨Translate the C part of the current section 240⟩
   ⟨Show cross-references to this section 243⟩
   ⟨Output the code for the end of a section 247⟩
}
This code is used in section 228.

232.   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 232 ⟩ ≡
  if (∗(*loc* − 1) ≠ '*') *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");
    { **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 ('{');
  out_section (*section_count*);
  out ('}');
This code is used in section 231.

233.   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 233 ⟩ ≡
  do {
    switch (*next_control* ← *copy_TₑX* ( )) {
    case '|': *init_stack*;
      output_C ( );
      break;
    case '@': *out* ('@');
      break;
    case *TₑX_string*: **case** *noop*: **case** *xref_roman*: **case** *xref_wildcard*: **case** *xref_typewriter*:
      case *section_name*: *loc* −= 2;
      next_control ← *get_next* ( );   /∗ skip to @> ∗/
      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* ("! You can't do that in TeX text");
      break;
    }
  } **while** (*next_control* < *format_code*);
This code is used in section 231.

234.   When we get to the following code we have $next\_control \geq format\_code$, and the token memory is in its initial empty state.

⟨ Translate the definition part of the current section 234 ⟩ ≡
  $space\_checked \leftarrow false$;
  while ($next\_control \leq definition$) {   /∗ $format\_code$ or $definition$ ∗/
    $init\_stack$;
    if ($next\_control \equiv definition$) ⟨ Start a macro definition 237 ⟩
    else ⟨ Start a format definition 238 ⟩
    $outer\_parse(\ )$;
    $finish\_C(format\_visible)$;
    $format\_visible \leftarrow true$;
    $doing\_format \leftarrow false$;
  }

This code is used in section 231.

235.   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 $visible$)   /∗ $true$ if we should produce TEX output ∗/
  {
    text_pointer p;   /∗ translation of the scraps ∗/
    if ($visible$) {
      $out\_str("\backslash\backslash B")$;
      $app\_tok(force)$;
      $app\_scrap(insert, no\_math)$;
      $p \leftarrow translate(\ )$;
      $app(tok\_flag + (\textbf{int})(p - tok\_start))$;
      $make\_output(\ )$;   /∗ output the list ∗/
      if ($out\_ptr > out\_buf + 1$)
        if ($*(out\_ptr - 1) \equiv '\backslash\backslash'$) {
          if ($*out\_ptr \equiv '6'$) $out\_ptr\ -= 2$;
          else if ($*out\_ptr \equiv '7'$) $*out\_ptr \leftarrow 'Y'$;
        }
      $out\_str("\backslash\backslash par")$;
      $finish\_line(\ )$;
    }
    if ($text\_ptr > max\_text\_ptr$) $max\_text\_ptr \leftarrow text\_ptr$;
    if ($tok\_ptr > max\_tok\_ptr$) $max\_tok\_ptr \leftarrow tok\_ptr$;
    if ($scrap\_ptr > max\_scr\_ptr$) $max\_scr\_ptr \leftarrow scrap\_ptr$;
    $tok\_ptr \leftarrow tok\_mem + 1$;
    $text\_ptr \leftarrow tok\_start + 1$;
    $scrap\_ptr \leftarrow scrap\_info$;   /∗ forget the tokens and the scraps ∗/
  }

236.   ⟨ Predeclaration of procedures 8 ⟩ +≡   **static void** $finish\_C$(**boolean**);

237.   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 237 ⟩ ≡
```
  {
    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 {
      app_cur_id(false);
      if (∗loc ≡ '(') {
        app('$');
      reswitch:
        switch (next_control ← get_next()) {
        case '(': case ',': app(next_control);
          goto reswitch;
        case identifier: app_cur_id(false);
          goto reswitch;
        case ')': app(next_control);
          next_control ← get_next();
          break;
        case dot_dot_dot: app_str("\\,\\ldots\\,");
          app_scrap(raw_int, no_math);
          if ((next_control ← get_next()) ≡ ')') {
            app(next_control);
            next_control ← get_next();
            break;
          }    /∗ otherwise fall through ∗/
        default: err_print("!␣Improper␣macro␣definition");
          break;
        }
        app('$');
      }
      else next_control ← get_next();
      app(break_space);
      app_scrap(dead, no_math);      /∗ scrap won't take part in the parsing ∗/
    }
  }
```
This code is used in section 234.

238. ⟨ Start a format definition 238 ⟩ ≡
  {
    $doing\_format \leftarrow true$;
    if $(*(loc - 1) \equiv \text{'s'} \vee *(loc - 1) \equiv \text{'S'})$ $format\_visible \leftarrow false$;
    if $(\neg space\_checked)$ {
      $emit\_space\_if\_needed$;
      $save\_position$;
    }
    $app\_str(\text{"\\\\F"})$;   /* this will produce '**format** ' */
    $next\_control \leftarrow get\_next(\ )$;
    if $(next\_control \equiv identifier)$ {
      $app(id\_flag + (\textbf{int})(id\_lookup(id\_first, id\_loc, normal) - name\_dir))$;
      $app(break\_space)$;   /* this is syntactically separate from what follows */
      $next\_control \leftarrow get\_next(\ )$;
      if $(next\_control \equiv identifier)$ {
        $app(id\_flag + (\textbf{int})(id\_lookup(id\_first, id\_loc, normal) - name\_dir))$;
        $app\_scrap(exp, maybe\_math)$;
        $app\_scrap(semi, maybe\_math)$;
        $next\_control \leftarrow get\_next(\ )$;
      }
    }
    if $(scrap\_ptr \neq scrap\_info + 2)$ $err\_print(\text{"!␣Improper␣format␣definition"})$;
  }

This code is used in section 234.

239. Finally, when the TEX and definition parts have been treated, we have $next\_control \geq begin\_C$. We will make the global variable *this_section* point to the current section name, if it has a name.

⟨ Private variables 21 ⟩ +≡
  static name_pointer *this_section*;   /* the current section name, or zero */

240. ⟨ Translate the C part of the current section 240 ⟩ ≡
  $this\_section \leftarrow name\_dir$;
  if $(next\_control \leq section\_name)$ {
    $emit\_space\_if\_needed$;
    $init\_stack$;
    if $(next\_control \equiv begin\_C)$ $next\_control \leftarrow get\_next(\ )$;
    else {
      $this\_section \leftarrow cur\_section$;
      ⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section
          definition 241 ⟩
    }
    while $(next\_control \leq section\_name)$ {
      $outer\_parse(\ )$;
      ⟨ Emit the scrap for a section name if present 242 ⟩
    }
    $finish\_C(true)$;
  }

This code is used in section 231.

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

$\langle$ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition $241 \rangle \equiv$
  do $next\_control \leftarrow get\_next(\,)$; **while** $(next\_control \equiv$ '+'$)$;   /* allow optional '+=' */
  if $(next\_control \neq$ '='$\,\wedge next\_control \neq eq\_eq)$
    $err\_print($"! You need an = sign after the section name"$)$;
  else $next\_control \leftarrow get\_next(\,)$;
  if $(out\_ptr > out\_buf + 1 \wedge *out\_ptr \equiv$ 'Y' $\wedge *(out\_ptr - 1) \equiv$ '\\'$)$ $app(backup)$;
    /* 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 $(cur\_xref \rightarrow num \equiv file\_flag)$ $cur\_xref \leftarrow cur\_xref \rightarrow xlink$;
  $app\_str($"\${}"$)$;
  if $(cur\_xref \rightarrow num \neq section\_count + def\_flag)$ {
    $app\_str($"\\mathrel+"$)$;   /* section name is multiply defined */
    $this\_section \leftarrow name\_dir$;   /* so we won't give cross-reference info here */
  }
  $app\_str($"\\E"$)$;   /* output an equivalence sign */
  $app\_str($"{}\$"$)$;
  $app(force)$;
  $app\_scrap(dead, no\_math)$;   /* this forces a line break unless '@+' follows */
This code is used in section 240.

242. $\langle$ Emit the scrap for a section name if present $242 \rangle \equiv$
  if $(next\_control < section\_name)$ {
    $err\_print($"! 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 240.

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

$\langle$ Show cross-references to this section $243 \rangle \equiv$
  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 231.

244.   The *footnote* procedure gives cross-reference information about multiply defined section names (if
the *flag* parameter is *def_flag*), or about references to a section 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 ← cur_xref;      /* cross-reference pointer variable */
    if (q→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 246⟩
    out('.');
}
```

245.   ⟨Predeclaration of procedures 8⟩ +≡   **static void** *footnote*(**sixteen_bits**);

246.   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* 246⟩ ≡
```
    if (q→xlink→num > flag) out('s');      /* plural */
    while (true) {
        out_section(cur_xref→num − flag);
        cur_xref ← cur_xref→xlink;      /* point to the next cross-reference to output */
        if (cur_xref→num ≤ flag) break;
        if (cur_xref→xlink→num > flag) out_str(",␣");      /* not the last */
        else {
            out_str("\\ET");      /* the last */
            if (cur_xref ≠ q→xlink) out('s');      /* the last of more than two */
        }
    }
```
This code is used in section 244.

247.   ⟨Output the code for the end of a section 247⟩ ≡
```
    out_str("\\fi");
    finish_line();
    flush_buffer(out_buf, false, false);      /* insert a blank line, it looks nice */
```
This code is used in section 231.

248.   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");
    finish_line( );
  }
  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);
    if (change_exists) {
      ⟨Tell about changed sections 251⟩
      finish_line( );
      finish_line( );
    }
    out_str("\\inx");
    finish_line( );
    active_file ← idx_file;      /* change active file to the index file */
    ⟨Do the first pass of sorting 253⟩
    ⟨Sort and output the index 261⟩
    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 270⟩
    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);
  }
  if (show_happiness) {
    if (show_progress) new_line;
    fputs("Done.", stdout);
  }
  check_complete( );      /* was all of the change file used? */
}
```

249.   ⟨Predeclaration of procedures 8⟩ +≡   **static void** *phase_three*(**void**);

250. Just before the index comes a list of all the changed sections, including the index section itself.

⟨ Private variables 21 ⟩ +≡
  static sixteen_bits *k_section*;   /∗ runs through the sections ∗/

251. ⟨ Tell about changed sections 251 ⟩ ≡
  {   /∗ remember that the index is already marked as changed ∗/
    k_section ← 0;
    while (¬*changed_section*[++*k_section*]) ;
    out_str("\\ch␣");
    out_section(*k_section*);
    while (*k_section* < *section_count*) {
      while (¬*changed_section*[++*k_section*]) ;
      out_str(",␣");
      out_section(*k_section*);
    }
    out('.');
  }

This code is used in section 248.

252. 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 21 ⟩ +≡
  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 ∗/

253. 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 253 ⟩ ≡
  {
    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 248.

254.   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 jth 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 254 ⟩ ≡
    name_pointer *Head*;

This code is used in section 113.

255.   #**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 ∗/
⟨ Private variables 21 ⟩ +≡
    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* ∗/

256.   ⟨ Set initial values 24 ⟩ +≡
    max_sort_ptr ← *scrap_info*;

257.   The desired alphabetic order is specified by the *collate* array; namely, *collate*[0] < *collate*[1] < ⋯ < *collate*[100].

⟨ Private variables 21 ⟩ +≡
    static **eight_bits** *collate*[101 + 128];   /∗ collation order ∗/

258.   We use the order null $<$ ␣ $<$ other characters $<$ `_` $<$ `A` $=$ `a` $< \cdots <$ `Z` $=$ `z` $<$ `0` $< \cdots <$ `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 24 ⟩ $+\equiv$

 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 ∗/

259.   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*→*depth* $\leftarrow$ *infinity*;
        else *sort_ptr*→*depth* $\leftarrow$ *d*;
        sort_ptr→*head* $\leftarrow$ *bucket*[*collate*[*c*]];
        bucket[*collate*[*c*]] $\leftarrow$ Λ;
      }
  }

260.   ⟨Predeclaration of procedures 8⟩ +≡   **static void** *unbucket*(**eight_bits**);

261.   ⟨Sort and output the index 261⟩ ≡
  sort_ptr $\leftarrow$ *scrap_info*;
  unbucket(1);
  while (*sort_ptr* > *scrap_info*) {
    cur_depth $\leftarrow$ *sort_ptr*→*depth*;
    if (*blink*[*sort_ptr*→*head* − *name_dir*] $\equiv$ 0 $\vee$ *cur_depth* $\equiv$ *infinity*)
      ⟨Output index entries for the list at *sort_ptr* 263⟩
    else ⟨Split the list at *sort_ptr* into further lists 262⟩
  }
This code is used in section 248.

262. ⟨Split the list at *sort_ptr* into further lists 262⟩ ≡
  {
    eight_bits *c*;

    next_name ← *sort_ptr→head*;
    do {
      cur_name ← *next_name*;
      next_name ← *blink*[*cur_name* − *name_dir*];
      cur_byte ← *cur_name→byte_start* + *cur_depth*;
      if (*cur_byte* ≡ (*cur_name* + 1)*→byte_start*) *c* ← 0;   /∗ hit end of the name ∗/
      else {
        c ← (**eight_bits**) ∗*cur_byte*;
        if (*xisupper*(*c*)) *c* ← *tolower*(*c*);
      }
      blink[*cur_name* − *name_dir*] ← *bucket*[*c*];
      bucket[*c*] ← *cur_name*;
    } **while** (*next_name*);
    −−*sort_ptr*;
    unbucket(*cur_depth* + 1);
  }

This code is used in section 261.

263. ⟨Output index entries for the list at *sort_ptr* 263⟩ ≡
  {
    cur_name ← *sort_ptr→head*;
    do {
      out_str("\\I");
      ⟨Output the name at *cur_name* 264⟩
      ⟨Output the cross-references at *cur_name* 265⟩
      cur_name ← *blink*[*cur_name* − *name_dir*];
    } **while** (*cur_name*);
    −−*sort_ptr*;
  }

This code is used in section 261.

264. ⟨Output the name at *cur_name* 264⟩ ≡
  switch (*cur_name*→*ilk*) { **char** ∗*j*;
  case *normal*: **case** *func_template*:
    if (*is_tiny*(*cur_name*)) *out_str*("\\|");
    else {
      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 *wildcard*: *out_str*("\\9"); **goto** *not_an_identifier*;
  case *typewriter*: *out_str*("\\.");
  case *roman*: *not_an_identifier*: *out_name*(*cur_name*, *false*);
    goto *name_done*;
  case *custom*: *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 263.

265. Section numbers that are to be underlined are enclosed in '\[...]'.

⟨Output the cross-references at *cur_name* 265⟩ ≡
  ⟨Invert the cross-reference list at *cur_name*, making *cur_xref* the head 267⟩
  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 263.

266. 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 21⟩ +≡
  static xref_pointer *next_xref*, *this_xref*;   /∗ pointer variables for rearranging a list ∗/

267.   ⟨Invert the cross-reference list at *cur_name*, making *cur_xref* the head 267⟩ ≡
  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 265.

268.   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*);
    }
  }

269.   ⟨Predeclaration of procedures 8⟩ +≡   **static void** *section_print*(**name_pointer**);

270.   ⟨Output all the section names 270⟩ ≡
  section_print(*root*);
This code is used in section 248.

271.   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*);
    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*);
  }

272.   Index.   If you have read and understood the code for Phase III above, you know what is in this index and how it got here. All sections in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in section names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages, control sequences put into the output, and a few other things like "recursion" are indexed here too.

⟨ Append a TEX string, without forming a scrap 201 ⟩   Used in section 197.
⟨ Append a string or constant 200 ⟩   Used in section 197.
⟨ Append the scrap appropriate to *next_control* 197 ⟩   Used in section 194.
⟨ Cases for *alignas_like* 177 ⟩   Used in section 122.
⟨ Cases for *attr_head* 179 ⟩   Used in section 122.
⟨ Cases for *attr* 180 ⟩   Used in section 122.
⟨ Cases for *base* 141 ⟩   Used in section 122.
⟨ Cases for *binop* 133 ⟩   Used in section 122.
⟨ Cases for *case_like* 153 ⟩   Used in section 122.
⟨ Cases for *cast* 134 ⟩   Used in section 122.
⟨ Cases for *catch_like* 154 ⟩   Used in section 122.
⟨ Cases for *colcol* 138 ⟩   Used in section 122.
⟨ Cases for *const_like* 171 ⟩   Used in section 122.
⟨ Cases for *decl_head* 139 ⟩   Used in section 122.
⟨ Cases for *decl* 140 ⟩   Used in section 122.
⟨ Cases for *default_like* 181 ⟩   Used in section 122.
⟨ Cases for *delete_like* 175 ⟩   Used in section 122.
⟨ Cases for *do_like* 152 ⟩   Used in section 122.
⟨ Cases for *else_head* 149 ⟩   Used in section 122.
⟨ Cases for *else_like* 148 ⟩   Used in section 122.
⟨ Cases for *exp* 129 ⟩   Used in section 122.
⟨ Cases for *fn_decl* 144 ⟩   Used in section 122.
⟨ Cases for *for_like* 169 ⟩   Used in section 122.
⟨ Cases for *ftemplate* 168 ⟩   Used in section 122.
⟨ Cases for *function* 145 ⟩   Used in section 122.
⟨ Cases for *if_clause* 150 ⟩   Used in section 122.
⟨ Cases for *if_head* 151 ⟩   Used in section 122.
⟨ Cases for *if_like* 147 ⟩   Used in section 122.
⟨ Cases for *insert* 161 ⟩   Used in section 122.
⟨ Cases for *int_like* 136 ⟩   Used in section 122.
⟨ Cases for *langle* 164 ⟩   Used in section 122.
⟨ Cases for *lbrace* 146 ⟩   Used in section 122.
⟨ Cases for *lbrack* 178 ⟩   Used in section 122.
⟨ Cases for *lpar* 130 ⟩   Used in section 122.
⟨ Cases for *lproc* 159 ⟩   Used in section 122.
⟨ Cases for *new_exp* 167 ⟩   Used in section 122.
⟨ Cases for *new_like* 166 ⟩   Used in section 122.
⟨ Cases for *operator_like* 173 ⟩   Used in section 122.
⟨ Cases for *prelangle* 162 ⟩   Used in section 122.
⟨ Cases for *prerangle* 163 ⟩   Used in section 122.
⟨ Cases for *public_like* 137 ⟩   Used in section 122.
⟨ Cases for *question* 176 ⟩   Used in section 122.
⟨ Cases for *raw_int* 172 ⟩   Used in section 122.
⟨ Cases for *raw_ubin* 170 ⟩   Used in section 122.
⟨ Cases for *section_scrap* 160 ⟩   Used in section 122.
⟨ Cases for *semi* 158 ⟩   Used in section 122.
⟨ Cases for *sizeof_like* 135 ⟩   Used in section 122.
⟨ Cases for *stmt* 157 ⟩   Used in section 122.
⟨ Cases for *struct_head* 143 ⟩   Used in section 122.
⟨ Cases for *struct_like* 142 ⟩   Used in section 122.
⟨ Cases for *tag* 155 ⟩   Used in section 122.
⟨ Cases for *template_like* 165 ⟩   Used in section 122.

⟨ Cases for *typedef_like* 174 ⟩   Used in section 122.
⟨ Cases for *ubinop* 132 ⟩   Used in section 122.
⟨ Cases for *unop* 131 ⟩   Used in section 122.
⟨ Cases involving nonstandard characters 199 ⟩   Used in section 197.
⟨ Check for end of comment 102 ⟩   Used in section 101.
⟨ Check if next token is **include** 49 ⟩   Used in section 47.
⟨ Check if we're at the end of a preprocessor command 50 ⟩   Used in section 44.
⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 241 ⟩
      Used in section 240.
⟨ Clear *bal* and **return** 104 ⟩   Used in section 101.
⟨ Combine the irreducible scraps that remain 191 ⟩   Used in section 189.
⟨ Common code for CWEAVE and CTANGLE 3, 5, 6, 7, 9, 10, 12, 14, 15 ⟩   Used in section 1.
⟨ Compress two-symbol operator 51 ⟩   Used in section 44.
⟨ Copy a quoted character into the buffer 227 ⟩   Used in section 226.
⟨ Copy special things when *c* ≡ '@', '\\' 103 ⟩   Used in section 101.
⟨ Copy the C text into the *buffer* array 226 ⟩   Used in section 224.
⟨ Do the first pass of sorting 253 ⟩   Used in section 248.
⟨ Emit the scrap for a section name if present 242 ⟩   Used in section 240.
⟨ Get a binary constant 55 ⟩   Used in section 53.
⟨ Get a constant 53 ⟩   Used in section 44.
⟨ Get a hexadecimal constant 54 ⟩   Used in section 53.
⟨ Get a string 57 ⟩   Used in sections 44 and 59.
⟨ Get a wide character constant 58 ⟩   Used in section 57.
⟨ Get an identifier 52 ⟩   Used in section 44.
⟨ Get an octal constant 56 ⟩   Used in section 53.
⟨ Get control code and possible section name 59 ⟩   Used in section 44.
⟨ If end of name or erroneous control code, **break** 63 ⟩   Used in section 62.
⟨ If semi-tracing, show the irreducible scraps 192 ⟩   Used in section 191.
⟨ If tracing, print an indication of where we are 193 ⟩   Used in section 189.
⟨ Include files 4 ⟩   Used in section 1.
⟨ Insert new cross-reference at *q*, not at beginning of list 128 ⟩   Used in section 127.
⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 267 ⟩   Used in section 265.
⟨ Look ahead for strongest line break, **goto** *reswitch* 221 ⟩   Used in section 220.
⟨ Make sure that there is room for the new scraps, tokens, and texts 198 ⟩   Used in sections 197 and 206.
⟨ Make sure the entries *pp* through *pp* + 3 of *cat* are defined 186 ⟩   Used in section 185.
⟨ Match a production at *pp*, or increase *pp* if there is no match 122 ⟩   Used in section 185.
⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 220 ⟩   Used in section 218.
⟨ Output a section name 223 ⟩   Used in section 218.
⟨ Output all the section names 270 ⟩   Used in section 248.
⟨ Output all the section numbers on the reference list *cur_xref* 246 ⟩   Used in section 244.
⟨ Output an identifier 219 ⟩   Used in section 218.
⟨ Output index entries for the list at *sort_ptr* 263 ⟩   Used in section 261.
⟨ Output saved *indent* or *outdent* tokens 222 ⟩   Used in sections 218 and 221.
⟨ Output the code for the beginning of a new section 232 ⟩   Used in section 231.
⟨ Output the code for the end of a section 247 ⟩   Used in section 231.
⟨ Output the cross-references at *cur_name* 265 ⟩   Used in section 263.
⟨ Output the name at *cur_name* 264 ⟩   Used in section 263.
⟨ Output the text of the section name 224 ⟩   Used in section 223.
⟨ Predeclaration of procedures 8, 11, 13, 16, 25, 33, 40, 45, 65, 69, 71, 83, 86, 90, 95, 98, 109, 117, 120, 123, 182, 190, 195,
      202, 211, 215, 229, 236, 245, 249, 260, 269 ⟩   Used in section 1.
⟨ Print a snapshot of the scrap list if debugging 188 ⟩   Used in sections 183 and 184.
⟨ Print error messages about unused or undefined section names 84 ⟩   Used in section 68.

⟨ Print token r in symbolic form 118 ⟩   Used in section 116.

⟨ Print warning message, break the line, **return** 94 ⟩   Used in section 93.

⟨ Private variables 21, 23, 30, 37, 43, 46, 48, 67, 76, 81, 85, 106, 114, 119, 187, 209, 214, 230, 239, 250, 252, 255, 257, 266 ⟩
      Used in section 1.

⟨ Process a format definition 78 ⟩   Used in section 77.

⟨ Process simple format in limbo 79 ⟩   Used in section 41.

⟨ Put section name into *section_text* 62 ⟩   Used in section 60.

⟨ Raise preprocessor flag 47 ⟩   Used in section 44.

⟨ Reduce the scraps using the productions until no more rules apply 185 ⟩   Used in section 189.

⟨ Replace "@@" by "@" 75 ⟩   Used in sections 72 and 74.

⟨ Rest of *trans_plus* union 254 ⟩   Used in section 113.

⟨ Scan a verbatim string 66 ⟩   Used in section 59.

⟨ Scan the section name and make *cur_section* point to it 60 ⟩   Used in section 59.

⟨ Set initial values 24, 31, 38, 61, 92, 107, 115, 156, 205, 210, 256, 258 ⟩   Used in section 2.

⟨ Show cross-references to this section 243 ⟩   Used in section 231.

⟨ Skip next character, give error if not '@' 225 ⟩   Used in section 224.

⟨ Sort and output the index 261 ⟩   Used in section 248.

⟨ Special control codes for debugging 39 ⟩   Used in section 38.

⟨ Split the list at *sort_ptr* into further lists 262 ⟩   Used in section 261.

⟨ Start TEX output 89 ⟩   Used in section 2.

⟨ Start a format definition 238 ⟩   Used in section 234.

⟨ Start a macro definition 237 ⟩   Used in section 234.

⟨ Store all the reserved words 34 ⟩   Used in section 2.

⟨ Store cross-reference data for the current section 70 ⟩   Used in section 68.

⟨ Store cross-references in the C part of a section 80 ⟩   Used in section 70.

⟨ Store cross-references in the TEX part of a section 74 ⟩   Used in section 70.

⟨ Store cross-references in the definition part of a section 77 ⟩   Used in section 70.

⟨ Tell about changed sections 251 ⟩   Used in section 248.

⟨ Translate the C part of the current section 240 ⟩   Used in section 231.

⟨ Translate the TEX part of the current section 233 ⟩   Used in section 231.

⟨ Translate the current section 231 ⟩   Used in section 228.

⟨ Translate the definition part of the current section 234 ⟩   Used in section 231.

⟨ Typedef declarations 22, 29, 113, 208 ⟩   Used in section 1.