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␣3.64)\n"
  ⟨ Include files 28 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Common code for CWEAVE and CTANGLE 0 ⟩
  ⟨ Typedef declarations 8 ⟩
  ⟨ Global variables 7 ⟩
  ⟨ Predeclaration of procedures 2 ⟩
```

2. We predeclare several standard system functions here instead of including their system header files, because the names of the header files are not as standard as the names of the functions. (For example, some C environments have <string.h> where others have <strings.h>.)

```
⟨ Predeclaration of procedures 2 ⟩ ≡
  extern int strlen( );   /* length of string */
  extern int strcmp( );   /* compare strings lexicographically */
  extern char *strcpy( );   /* copy one string to another */
  extern int strncmp( );   /* compare up to n string characters */
  extern char *strncpy( );   /* copy up to n string characters */
```

See also sections 24, 29, 45, 49, 52, 54, 64, 73, 81, 104, 171, 184, 195, 202, 211, 215, 227, and 236.

This code is used in section 1.

3. 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(ac, av)
      int ac;   /* argument count */
      char **av;   /* argument values */
  {
    argc = ac;
    argv = av;
    program = cweave;
    make_xrefs = force_lines = make_pb = 1;   /* controlled by command-line options */
    common_init( );
    ⟨ Set initial values 10 ⟩;
    if (show_banner) printf(banner);   /* print a "banner line" */
    ⟨ Store all the reserved words 18 ⟩;
    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 */
    return wrap_up( );   /* and exit gracefully */
  }
```

4. The following parameters were sufficient in the original `WEAVE` to handle TeX, so they should be sufficient for most applications of `CWEAVE`. If you change *max_bytes*, *max_names*, *hash_size*, or *buf_size* you have to change them also in the file `"common.w"`.

```
#define max_bytes 90000   /* the number of bytes in identifiers, index entries, and section names */
#define max_names 4000   /* number of identifiers, strings, section names; must be less than 10240;
          used in "common.w" */
#define max_sections 2000   /* greater than the total number of sections */
#define hash_size 353   /* should be prime */
#define buf_size 100   /* maximum length of input line, plus one */
#define longest_name 10000   /* section names and strings shouldn't be longer than this */
#define long_buf_size (buf_size + longest_name)
#define line_length 80
          /* lines of TeX output have at most this many characters; should be less than 256 */
#define max_refs 20000   /* number of cross-references; must be less than 65536 */
#define max_toks 20000   /* number of symbols in C texts being parsed; must be less than 65536 */
#define max_texts 4000   /* number of phrases in C texts being parsed; must be less than 10240 */
#define max_scraps 2000   /* number of tokens in C texts being parsed */
#define stack_size 400   /* number of simultaneous output levels */
```

5. 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 needs to be updated when `"common.w"` changes.

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

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

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

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

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

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

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

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

7. 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.

```
⟨ Global variables 7 ⟩ ≡
  boolean change_exists; /* has any section changed? */
```

See also sections 9, 15, 21, 27, 31, 33, 48, 58, 63, 67, 87, 94, 98, 158, 177, 181, 197, 206, 217, 219, 223, 225, and 234.

This code is used in section 1.

8. 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*→*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* = *p*→*xref* is a pointer into *xmem*, the value of *x*→*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*→*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 8 ⟩ ≡
  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 14, 93, and 176.

This code is used in section 1.

9.
```
⟨ Global variables 7 ⟩ +≡
  xref_info xmem[max_refs];    /* contains cross-reference information */
  xref_pointer xmem_end = xmem + max_refs − 1;
  xref_pointer xref_ptr;    /* the largest occupied position in xmem */
  sixteen_bits xref_switch, section_xref_switch;    /* either zero or def_flag */
```

10. 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 10 ⟩ ≡
  xref_ptr = xmem;
  name_dir→xref = (char ∗) xmem;
  xref_switch = 0;
  section_xref_switch = 0;
  xmem→num = 0;    /* sentinel value */
```

See also sections 16, 22, 42, 70, 72, 88, 95, 178, 224, and 226.

This code is used in section 3.

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

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

```
#define append_xref(c)
          if (xref_ptr ≡ xmem_end) overflow("cross-reference");
          else (++xref_ptr)→num = c;
#define no_xref (flags['x'] ≡ 0)
#define make_xrefs flags['x']      /* should cross references be output? */
#define is_tiny(p) ((p + 1)→byte_start ≡ (p)→byte_start + 1)
#define unindexed(a) (a < res_wd_end ∧ a→ilk ≥ custom)
            /* tells if uses of a name are to be indexed */

  void new_xref(p)name_pointerp;
  {
    xref_pointer q;     /* pointer to previous cross-reference */
    sixteen_bitsm, 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;
    p→xref = (char *) xref_ptr;
  }
```

12. 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.

```
void new_section_xref(p)name_pointerp;
{
  xref_pointer q, r;     /* pointers to previous cross-references */
  q = (xref_pointer) p→xref;
  r = xmem;
  if (q > xmem)
    while (q→num > section_xref_switch) {
      r = q;
      q = q→xlink;
    }
  if (r→num ≡ section_count + section_xref_switch) return;     /* don't duplicate entries */
  append_xref(section_count + section_xref_switch);
  xref_ptr→xlink = q;
  section_xref_switch = 0;
  if (r ≡ xmem) p→xref = (char *) xref_ptr;
  else r→xlink = xref_ptr;
}
```

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

```
void set_file_flag(p)name_pointerp;
{
  xref_pointer q;
  q = (xref_pointer) p→xref;
  if (q→num ≡ file_flag) return;
  append_xref(file_flag);
  xref_ptr→xlink = q;
  p→xref = (char *) xref_ptr;
}
```

14. 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 8 ⟩ +≡
  typedef sixteen_bits token;
  typedef token *token_pointer;
  typedef token_pointer *text_pointer;
```

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

```
⟨ Global variables 7 ⟩ +≡
  token tok_mem[max_toks];   /∗ tokens ∗/
  token_pointer tok_mem_end = tok_mem + max_toks − 1;   /∗ end of tok_mem ∗/
  token_pointer tok_start[max_texts];   /∗ directory into tok_mem ∗/
  token_pointer tok_ptr;   /∗ first unused position in tok_mem ∗/
  text_pointer text_ptr;   /∗ first unused position in tok_start ∗/
  text_pointer tok_start_end = tok_start + max_texts − 1;   /∗ end of tok_start ∗/
  token_pointer max_tok_ptr;   /∗ largest value of tok_ptr ∗/
  text_pointer max_text_ptr;   /∗ largest value of text_ptr ∗/
```

16. ⟨ Set initial values 10 ⟩ +≡

```
  tok_ptr = tok_mem + 1;
  text_ptr = tok_start + 1;
  tok_start[0] = tok_mem + 1;
  tok_start[1] = tok_mem + 1;
  max_tok_ptr = tok_mem + 1;
  max_text_ptr = tok_start + 1;
```

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

```
  int names_match(p, first, l, t)name_pointerp;   /∗ points to the proposed match ∗/
  char ∗first;   /∗ position of first character of string ∗/
  int l;   /∗ length of identifier ∗/
  eight_bitst;   /∗ desired ilk ∗/
  {
    if (length(p) ≠ l) return 0;
    if (p→ilk ≠ t ∧ ¬(t ≡ normal ∧ abnormal(p))) return 0;
    return ¬strncmp(first, p→byte_start, l);
  }
  void init_p(p, t)name_pointerp;
  eight_bitst;
  {
    p→ilk = t;
    p→xref = (char ∗) xmem;
  }
  void init_node(p)name_pointerp;
  {
    p→xref = (char ∗) xmem;
  }
```

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

⟨ Store all the reserved words 18 ⟩ ≡

```
  id_lookup("and", Λ, alfop);
  id_lookup("and_eq", Λ, alfop);
  id_lookup("asm", Λ, sizeof_like);
  id_lookup("auto", Λ, int_like);
  id_lookup("bitand", Λ, alfop);
  id_lookup("bitor", Λ, alfop);
  id_lookup("bool", Λ, raw_int);
  id_lookup("break", Λ, case_like);
  id_lookup("case", Λ, case_like);
  id_lookup("catch", Λ, catch_like);
  id_lookup("char", Λ, raw_int);
  id_lookup("class", Λ, struct_like);
  id_lookup("clock_t", Λ, raw_int);
  id_lookup("compl", Λ, alfop);
  id_lookup("const", Λ, const_like);
  id_lookup("const_cast", Λ, raw_int);
  id_lookup("continue", Λ, case_like);
  id_lookup("default", Λ, case_like);
  id_lookup("define", Λ, define_like);
  id_lookup("defined", Λ, sizeof_like);
  id_lookup("delete", Λ, delete_like);
  id_lookup("div_t", Λ, raw_int);
  id_lookup("do", Λ, do_like);
  id_lookup("double", Λ, raw_int);
  id_lookup("dynamic_cast", Λ, raw_int);
  id_lookup("elif", Λ, if_like);
  id_lookup("else", Λ, else_like);
  id_lookup("endif", Λ, if_like);
  id_lookup("enum", Λ, struct_like);
  id_lookup("error", Λ, if_like);
  id_lookup("explicit", Λ, int_like);
  id_lookup("export", Λ, int_like);
  id_lookup("extern", Λ, int_like);
  id_lookup("FILE", Λ, raw_int);
  id_lookup("float", Λ, raw_int);
  id_lookup("for", Λ, for_like);
  id_lookup("fpos_t", Λ, raw_int);
  id_lookup("friend", Λ, int_like);
  id_lookup("goto", Λ, case_like);
  id_lookup("if", Λ, if_like);
  id_lookup("ifdef", Λ, if_like);
  id_lookup("ifndef", Λ, if_like);
  id_lookup("include", Λ, if_like);
  id_lookup("inline", Λ, int_like);
  id_lookup("int", Λ, raw_int);
  id_lookup("jmp_buf", Λ, raw_int);
  id_lookup("ldiv_t", Λ, raw_int);
  id_lookup("line", Λ, if_like);
```

```
id_lookup("long", Λ, raw_int);
id_lookup("mutable", Λ, int_like);
id_lookup("namespace", Λ, struct_like);
id_lookup("new", Λ, new_like);
id_lookup("not", Λ, alfop);
id_lookup("not_eq", Λ, alfop);
id_lookup("NULL", Λ, custom);
id_lookup("offsetof", Λ, raw_int);
id_lookup("operator", Λ, operator_like);
id_lookup("or", Λ, alfop);
id_lookup("or_eq", Λ, alfop);
id_lookup("pragma", Λ, if_like);
id_lookup("private", Λ, public_like);
id_lookup("protected", Λ, public_like);
id_lookup("ptrdiff_t", Λ, raw_int);
id_lookup("public", Λ, public_like);
id_lookup("register", Λ, int_like);
id_lookup("reinterpret_cast", Λ, raw_int);
id_lookup("return", Λ, case_like);
id_lookup("short", Λ, raw_int);
id_lookup("sig_atomic_t", Λ, raw_int);
id_lookup("signed", Λ, raw_int);
id_lookup("size_t", Λ, raw_int);
id_lookup("sizeof", Λ, sizeof_like);
id_lookup("static", Λ, int_like);
id_lookup("static_cast", Λ, raw_int);
id_lookup("struct", Λ, struct_like);
id_lookup("switch", Λ, for_like);
id_lookup("template", Λ, template_like);
id_lookup("this", Λ, custom);
id_lookup("throw", Λ, case_like);
id_lookup("time_t", Λ, raw_int);
id_lookup("try", Λ, else_like);
id_lookup("typedef", Λ, typedef_like);
id_lookup("typeid", Λ, raw_int);
id_lookup("typename", Λ, struct_like);
id_lookup("undef", Λ, if_like);
id_lookup("union", Λ, struct_like);
id_lookup("unsigned", Λ, raw_int);
id_lookup("using", Λ, int_like);
id_lookup("va_dcl", Λ, decl);    /* Berkeley's variable-arg-list convention */
id_lookup("va_list", Λ, raw_int);    /* ditto */
id_lookup("virtual", Λ, int_like);
id_lookup("void", Λ, raw_int);
id_lookup("volatile", Λ, const_like);
id_lookup("wchar_t", Λ, raw_int);
id_lookup("while", Λ, for_like);
id_lookup("xor", Λ, alfop);
id_lookup("xor_eq", Λ, alfop);
res_wd_end = name_ptr;
id_lookup("TeX", Λ, custom);
id_lookup("make_pair", Λ, func_template);
```

This code is used in section 3.

19. 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.

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

```
#define ignore °0   /* control code of no interest to CWEAVE */
#define verbatim °2   /* takes the place of extended ASCII α */
#define begin_short_comment °3   /* C++ short comment */
#define begin_comment '\t'   /* tab marks will not appear */
#define underline '\n'   /* this code will be intercepted without confusion */
#define noop °177   /* takes the place of ASCII delete */
#define xref_roman °203   /* control code for '@^' */
#define xref_wildcard °204   /* control code for '@:' */
#define xref_typewriter °205   /* control code for '@.' */
#define TeX_string °206   /* control code for '@t' */
  format TeX_string TeX
#define 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 '@*' */
```

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

```
⟨ Global variables 7 ⟩ +≡
  eight_bits ccode[256];   /* meaning of a char following @ */
```

22. ⟨ Set initial values 10 ⟩ +≡

```
{
  int c;
  for (c = 0; c < 256; c++) ccode[c] = 0;
}
ccode['␣'] = ccode['\t'] = ccode['\n'] = ccode['\v'] = ccode['\r'] = ccode['\f'] = ccode['*'] =
    new_section;
ccode['@'] = '@';     /* ‘quoted’ at sign */
ccode['='] = verbatim;
ccode['d'] = ccode['D'] = definition;
ccode['f'] = ccode['F'] = ccode['s'] = ccode['S'] = format_code;
ccode['c'] = ccode['C'] = ccode['p'] = ccode['P'] = begin_C;
ccode['t'] = ccode['T'] = TeX_string;
ccode['l'] = ccode['L'] = translit_code;
ccode['q'] = ccode['Q'] = noop;
ccode['h'] = ccode['H'] = output_defs_code;
ccode['&'] = join;
ccode['<'] = ccode['('] = section_name;
ccode['!'] = underline;
ccode['^'] = xref_roman;
ccode[':'] = xref_wildcard;
ccode['.'] = xref_typewriter;
ccode[','] = thin_space;
ccode['|'] = math_break;
ccode['/'] = line_break;
ccode['#'] = big_line_break;
ccode['+'] = no_line_break;
ccode[';'] = pseudo_semi;
ccode['['] = macro_arg_open;
ccode[']'] = macro_arg_close;
ccode['\''] = ord;
⟨ Special control codes for debugging 23 ⟩
```

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

⟨ Special control codes for debugging 23 ⟩ ≡

```
  ccode['0'] = ccode['1'] = ccode['2'] = trace;
```

This code is used in section 22.

24. 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 2 ⟩ +≡

```
  void skip_limbo( );
```

25.

```
void skip_limbo( )
{
  while (1) {
    if (loc > limit ∧ get_line( ) ≡ 0) 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 61⟩;
    }
  }
}
```

26. 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
unsigned skip_TEX( )      /∗ skip past pure TEX code ∗/
{
  while (1) {
    if (loc > limit ∧ get_line( ) ≡ 0) return (new_section);
    ∗(limit + 1) = '@';
    while (∗loc ≠ '@' ∧ ∗loc ≠ '|') loc++;
    if (∗loc++ ≡ '|') return ('|');
    if (loc ≤ limit) return (ccode[(eight_bits) ∗ (loc++)]);
  }
}
```

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

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

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

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

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

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

xref_roman, *xref_wildcard*, *xref_typewriter*, *TEX_string*, *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 */
⟨ Global variables 7 ⟩ +≡
  name_pointer cur_section;   /* name of section just scanned */
  char cur_section_char;   /* the character just before that name */
```

28. ⟨ Include files 28 ⟩ ≡

```
#include <ctype.h>   /* definition of isalpha, isdigit and so on */
#include <stdlib.h>   /* definition of exit */
```

This code is used in section 1.

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

```
#define isxalpha(c) ((c) ≡ '_' ∨ (c) ≡ '$')   /* non-alpha characters allowed in identifier */
#define ishigh(c) ((eight_bits)(c) > °177)
⟨ Predeclaration of procedures 2 ⟩ +≡
  eight_bits get_next( );
```

30.

```
eight_bits get_next( )     /* produces the next input token */
{ eight_bits c;     /* the current character */
  while (1) {
    ⟨ Check if we're at the end of a preprocessor command 35 ⟩;
    if (loc > limit ∧ get_line( ) ≡ 0) return (new_section);
    c = *(loc++);
    if (xisdigit(c) ∨ c ≡ '.') ⟨ Get a constant 38 ⟩
    else if (c ≡ '\'' ∨ c ≡ '"' ∨ (c ≡ 'L' ∧ (*loc ≡ '\'' ∨ *loc ≡ '"'))
          ∨ (c ≡ '<' ∧ sharp_include_line ≡ 1)) ⟨ Get a string 39 ⟩
    else if (xisalpha(c) ∨ isxalpha(c) ∨ ishigh(c)) ⟨ Get an identifier 37 ⟩
    else if (c ≡ '@') ⟨ Get control code and possible section name 40 ⟩
    else if (xisspace(c)) continue;     /* ignore spaces and tabs */
    if (c ≡ '#' ∧ loc ≡ buffer + 1) ⟨ Raise preprocessor flag 32 ⟩;
  mistake: ⟨ Compress two-symbol operator 36 ⟩
    return (c);
  }
}
```

31. 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 */
```

```
⟨ Global variables 7 ⟩ +≡
  boolean preprocessing = 0;     /* are we scanning a preprocessor command? */
```

32. ⟨ Raise preprocessor flag 32 ⟩ ≡

```
{
  preprocessing = 1;
  ⟨ Check if next token is include 34 ⟩;
  return (left_preproc);
}
```

This code is used in section 30.

33. 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.

```
⟨ Global variables 7 ⟩ +≡
  boolean sharp_include_line = 0;     /* are we scanning a # include line? */
```

34. ⟨ Check if next token is **include** 34 ⟩ ≡

```
while (loc ≤ buffer_end − 7 ∧ xisspace(*loc)) loc++;
if (loc ≤ buffer_end − 6 ∧ strncmp(loc, "include", 7) ≡ 0) sharp_include_line = 1;
```

This code is used in section 32.

35. 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 35 ⟩ ≡
  while (loc ≡ limit − 1 ∧ preprocessing ∧ ∗loc ≡ '\\')
    if (get_line() ≡ 0) return (new_section);    /∗ still in preprocessor mode ∗/
  if (loc ≥ limit ∧ preprocessing) {
    preprocessing = sharp_include_line = 0;
    return (right_preproc);
  }
```

This code is used in section 30.

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

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

⟨ Compress two-symbol operator 36 ⟩ ≡
  switch (c) {
  case '/':
    if (*loc ≡ '*') {
      compress(begin_comment);
    }
    else if (*loc ≡ '/') compress(begin_short_comment);
    break;
  case '+':
    if (*loc ≡ '+') compress(plus_plus);
    break;
  case '-':
    if (*loc ≡ '-') {
      compress(minus_minus);
    }
    else if (*loc ≡ '>')
      if (*(loc + 1) ≡ '*') {
        loc++;
        compress(minus_gt_ast);
      }
      else compress(minus_gt);
    break;
  case '.':
    if (*loc ≡ '*') {
      compress(period_ast);
    }
    else if (*loc ≡ '.' ∧ *(loc + 1) ≡ '.') {
      loc++;
      compress(dot_dot_dot);
    }
    break;
  case ':':
    if (*loc ≡ ':') compress(colon_colon);
    break;
  case '=':
    if (*loc ≡ '=') compress(eq_eq);
    break;
  case '>':
    if (*loc ≡ '=') {
      compress(gt_eq);
    }
    else if (*loc ≡ '>') compress(gt_gt);
    break;
  case '<':
    if (*loc ≡ '=') {
      compress(lt_eq);
    }
    else if (*loc ≡ '<') compress(lt_lt);
    break;
  case '&':
    if (*loc ≡ '&') compress(and_and);
    break;
  case '|':
    if (*loc ≡ '|') compress(or_or);
    break;
  case '!':
    if (*loc ≡ '=') compress(not_eq);
    break;
  }
```

This code is used in section 30.

37. ⟨ Get an identifier 37 ⟩ ≡

```
  {
    id_first = --loc;
    while (isalpha(*++loc) ∨ isdigit(*loc) ∨ isxalpha(*loc) ∨ ishigh(*loc)) ;
    id_loc = loc;
    return (identifier);
  }
```

This code is used in section 30.

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

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

⟨ Get a constant 38 ⟩ ≡

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

This code is used in section 30.

39. 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 39 ⟩ ≡
  {
    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') {      /* wide character constant */
      delim = *loc++;
      *++id_loc = delim;
    }
    if (delim ≡ '<') delim = '>';      /* for file names in # include lines */
    while (1) {
      if (loc ≥ limit) {
        if (*(limit − 1) ≠ '\\') {
          err_print("!␣String␣didn't␣end");
          loc = limit;
          break;
        }
        if (get_line() ≡ 0) {
          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) {
      printf("\n!␣String␣too␣long:␣");
      term_write(section_text + 1, 25);
      printf("...");
      mark_error;
    }
    id_loc++;
    return (string);
  }
```

This code is used in sections 30 and 40.

40. 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 40 ⟩ ≡
  {
    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:
      c = ccode[c];
      skip_restricted( );
      return (c);
    case section_name: ⟨ Scan the section name and make cur_section point to it 41 ⟩;
    case verbatim: ⟨ Scan a verbatim string 47 ⟩;
    case ord: ⟨ Get a string 39 ⟩;
    default: return (ccode[(eight_bits)c]);
    }
  }
```

This code is used in section 30.

41. 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 41 ⟩ ≡
  {
    char ∗k;      /∗ pointer into section_text ∗/
    cur_section_char = ∗(loc − 1);
    ⟨ Put section name into section_text 43 ⟩;
    if (k − section_text > 3 ∧ strncmp(k − 2, "...", 3) ≡ 0)
      cur_section = section_lookup(section_text + 1, k − 3, 1);      /∗ 1 indicates a prefix ∗/
    else cur_section = section_lookup(section_text + 1, k, 0);
    xref_switch = 0;
    return (section_name);
  }
```

This code is used in section 40.

42. 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 10 ⟩ +≡
  section_text[0] = '␣';
```

43. ⟨ Put section name into *section_text* 43 ⟩ ≡

```
k = section_text;
while (1) {
  if (loc > limit ∧ get_line( ) ≡ 0) {
    err_print("!␣Input␣ended␣in␣section␣name");
    loc = buffer + 1;
    break;
  }
  c = *loc;
  ⟨ If end of name or erroneous control code, break 44 ⟩;
  loc++;
  if (k < section_text_end) k++;
  if (xisspace(c)) {
    c = '␣';
    if (*(k − 1) ≡ '␣') k−−;
  }
  *k = c;
}
if (k ≥ section_text_end) {
  printf("\n!␣Section␣name␣too␣long:␣");
  term_write(section_text + 1, 25);
  printf("...");
  mark_harmless;
}
if (*k ≡ '␣' ∧ k > section_text) k−−;
```

This code is used in section 41.

44. ⟨ If end of name or erroneous control code, **break** 44 ⟩ ≡

```
if (c ≡ '@') {
  c = *(loc + 1);
  if (c ≡ '>') {
    loc += 2;
    break;
  }
  if (ccode[(eight_bits) c] ≡ new_section) {
    err_print("!␣Section␣name␣didn't␣end");
    break;
  }
  if (c ≠ '@') {
    err_print("!␣Control␣codes␣are␣forbidden␣in␣section␣name");
    break;
  }
  *(++k) = '@';
  loc++;     /* now c ≡ *loc again */
}
```

This code is used in section 43.

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

⟨ Predeclaration of procedures 2 ⟩ +≡

```
  void skip_restricted( );
```

46.

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

47. 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 47 ⟩ ≡

```
{
    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 40.

48. 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.

```
⟨ Global variables 7 ⟩ +≡
  eight_bits next_control;      /∗ control code waiting to be acting upon ∗/
```

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

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_one( );
```

```
50.  void phase_one( )
  {
    phase = 1;
    reset_input( );
    section_count = 0;
    skip_limbo( );
    change_exists = 0;
    while (¬input_has_ended) ⟨ Store cross-reference data for the current section 51 ⟩;
    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 66 ⟩;
  }
```

```
51.  ⟨ Store cross-reference data for the current section 51 ⟩ ≡
  {
    if (++section_count ≡ max_sections) overflow("section␣number");
    changed_section[section_count] = changing;     /∗ it will become 1 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 56 ⟩;
    ⟨ Store cross-references in the definition part of a section 59 ⟩;
    ⟨ Store cross-references in the C part of a section 62 ⟩;
    if (changed_section[section_count]) change_exists = 1;
  }
```

This code is used in section 50.

52. 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., $\mathit{next\_control} \geq \mathit{format\_code}$). If $\mathit{next\_control} \geq \mathit{format\_code}$ when *C_xref* is called, nothing will happen; but if $\mathit{next\_control} \equiv$ `'|'` 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 $\mathit{spec\_ctrl} \equiv \mathit{ignore}$, which triggers the default processing described above. If $\mathit{spec\_ctrl} \equiv \mathit{section\_name}$, 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 $\mathit{xref\_roman} \equiv \mathit{identifier} + \mathit{roman}$ and $\mathit{xref\_wildcard} \equiv \mathit{identifier} + \mathit{wildcard}$ and $\mathit{xref\_typewriter} \equiv \mathit{identifier} + \mathit{typewriter}$, as well as $\mathit{normal} \equiv 0$.

```
⟨Predeclaration of procedures 2⟩ +≡
  void C_xref( );
```

53.

```
void C_xref(spec_ctrl)      /* 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 "@" 57⟩
        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;
    }
  }
```

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

```
⟨Predeclaration of procedures 2⟩ +≡
  void outer_xref( );
```

55.

```
void outer_xref( )    /* 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 */
      }
    }
}
```

56. 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 56⟩ ≡
  while (1) {
    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 "@" 57⟩
        new_xref(id_lookup(id_first, id_loc, next_control − identifier));
      }
      break;
    }
    if (next_control ≥ format_code) break;
  }
```

This code is used in section 51.

57. ⟨ Replace "@@" by "@" 57 ⟩ ≡

```
{
  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 53 and 56.

58. 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.

⟨ Global variables 7 ⟩ +≡

```
name_pointer lhs, rhs;    /* pointers to byte_start for format identifiers */
name_pointer res_wd_end;    /* pointer to the first nonreserved identifier */
```

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

⟨ Store cross-references in the definition part of a section 59 ⟩ ≡

```
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 60 ⟩;
  outer_xref();
}
```

This code is used in section 51.

60. 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 60 ⟩ ≡
  {
    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 = (char *) q→xlink;
            else r = q;
        }
        next_control = get_next();
      }
    }
  }
```

This code is used in section 59.

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

```
⟨ Process simple format in limbo 61 ⟩ ≡
  {
    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 25.

62. Finally, when the TEX and definition parts have been treated, we have *next_control* ≥ *begin_C*.

⟨ Store cross-references in the C part of a section 62 ⟩ ≡

```
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 51.

63. 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.

⟨ Global variables 7 ⟩ +≡

```
xref_pointer cur_xref;     /* temporary cross-reference pointer */
boolean an_output;     /* did file_flag precede cur_xref? */
```

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

⟨ Predeclaration of procedures 2 ⟩ +≡

```
void section_check();
```

65.
```
void section_check(p)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 = 1;
      cur_xref = cur_xref→xlink;
    }
    else an_output = 0;
    if (cur_xref→num < def_flag) {
      printf("\n!␣Never␣defined:␣<");
      print_section_name(p);
      putchar('>');
      mark_harmless;
    }
    while (cur_xref→num ≥ cite_flag) cur_xref = cur_xref→xlink;
    if (cur_xref ≡ xmem ∧ ¬an_output) {
      printf("\n!␣Never␣used:␣<");
      print_section_name(p);
      putchar('>');
      mark_harmless;
    }
    section_check(p→rlink);
  }
}
```

66.
```
⟨Print error messages about unused or undefined section names 66⟩ ≡
  section_check(root)
```

This code is used in section 50.

67. 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.

```
⟨ Global variables 7 ⟩ +≡
  char out_buf[line_length + 1];      /* assembled characters */
  char *out_ptr;      /* just after last character in out_buf */
  char *out_buf_end = out_buf + line_length;      /* end of out_buf */
  int out_line;      /* number of next line to be output */
```

68. 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 1 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 0, 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, c)

  void flush_buffer(b, per_cent, carryover)
        char *b;      /* outputs from out_buf + 1 to b,where b ≤ out_ptr */
  booleanper_cent, carryover;
  {
    char *j;

    j = b;      /* pointer into out_buf */
    if (¬per_cent)      /* remove trailing blanks */
      while (j > out_buf ∧ *j ≡ '␣') j−−;
    c_line_write(j − out_buf);
    if (per_cent) tex_putc('%');
    tex_new_line;
    out_line++;
    if (carryover)
      while (j > out_buf)
        if (*j−− ≡ '%' ∧ (j ≡ out_buf ∨ *j ≠ '\\')) {
          *b−− = '%';
          break;
        }
    if (b < out_ptr) strncpy(out_buf + 1, b + 1, out_ptr − b);
    out_ptr −= b − out_buf;
  }
```

69. 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.

```
void finish_line()    /* do this at the end of a line */
{
  char *k;    /* pointer into buffer */
  if (out_ptr > out_buf) flush_buffer(out_ptr, 0, 0);
  else {
    for (k = buffer; k ≤ limit; k++)
      if (¬(xisspace(*k))) return;
    flush_buffer(out_buf, 0, 0);
  }
}
```

70. 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`'.

```
⟨ Set initial values 10 ⟩ +≡
  out_ptr = out_buf + 1;
  out_line = 1;
  active_file = tex_file;
  *out_ptr = 'c';
  tex_printf("\\input␣cwebma");
```

71. 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;
        }

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

72. 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 10 ⟩ +≡
  out_buf[0] = '\\';
```

73. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void break_out();
```

74.

```
void break_out( )    /* finds a way to break the output line */
{
  char *k = out_ptr;    /* pointer into out_buf */
  while (1) {
    if (k ≡ out_buf) ⟨Print warning message, break the line, return 75⟩;
    if (*k ≡ '␣') {
      flush_buffer(k, 0, 1);
      return;
    }
    if (*(k−−) ≡ '\\' ∧ *k ≠ '\\') {    /* we've decreased k */
      flush_buffer(k, 1, 1);
      return;
    }
  }
}
```

75. 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 75⟩ ≡
{
  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, 1, 1);
  return;
}
```

This code is used in section 74.

76. 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.

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

77. The *out_name* procedure is used to output an identifier or index entry, enclosing it in braces.

```
void out_name(p, quote_xalpha)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('}');
}
```

78. Routines that copy TEX material. During phase two, we use subroutines *copy_limbo*, *copy_TEX*, and *copy_comment* in place of the analogous *skip_limbo*, *skip_TEX*, and *skip_comment* that were used in phase one. (Well, *copy_comment* was actually written in such a way that it functions as *skip_comment* 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.

```
void copy_limbo( )
{
  char c;

  while (1) {
    if (loc > limit ∧ (finish_line( ), get_line( ) ≡ 0)) 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('@');
      }
    }
  }
}
```

79. 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.

80. format *copy_TeX* *TeX*

```
eight_bits copy_TEX()
{
  char c;    /* current character being copied */
  while (1) {
    if (loc > limit ∧ (finish_line(), get_line() ≡ 0)) 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++)]);
  }
}
```

81. 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;
        }
⟨ Predeclaration of procedures 2 ⟩ +≡
  int copy_comment();
```

82.

```
int copy_comment(is_long_comment, bal)      /* 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 (1) {
      if (loc > limit) {
        if (is_long_comment) {
          if (get_line() ≡ 0) {
            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 83⟩;
      if (phase ≡ 2) {
        if (ishigh(c)) app_tok(quoted_char);
        app_tok(c);
      }
      ⟨Copy special things when c ≡ '@', '\\' 84⟩;
      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 85⟩;
  }
```

83. ⟨Check for end of comment 83⟩ ≡

```
  if (c ≡ '*' ∧ *loc ≡ '/') {
    loc++;
    if (bal > 1) err_print("!␣Missing␣}␣in␣comment");
    goto done;
  }
```

This code is used in section 82.

84. ⟨ Copy special things when $c \equiv$ '@', '\\' 84 ⟩ ≡

```
  if (c ≡ '@') {
    if (*(loc++) ≠ '@') {
      err_print("!␣Illegal␣use␣of␣@␣in␣comment");
      loc −= 2;
      if (phase ≡ 2) *(tok_ptr − 1) = '␣';
      goto done;
    }
  }
  else if (c ≡ '\\' ∧ *loc ≠ '@') if (phase ≡ 2) app_tok(*(loc++))
  else loc++;
```

This code is used in section 82.

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

⟨ Clear *bal* and **return** 85 ⟩ ≡

```
  if (phase ≡ 2)
    while (bal−− > 0) app_tok('}');
  return (0);
```

This code is used in section 82.

86. 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

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

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

$$\mathit{exp}\ \mathit{comma}\ \mathit{exp} \rightarrow \mathit{exp} \qquad\qquad E_1C\ \mathit{opt}9\,E_2$$

(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 *int_like cast lbrace*, rule 31 is applied; but if the sequence is *int_like cast* followed by anything other than *lbrace*, rule 32 takes effect.

Translation rules such as '$E_1C\ \mathit{opt}9\,E_2$' above use subscripts to distinguish between translations of scraps whose categories have the same initial letter; these subscripts are assigned from left to right.

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

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

```
⟨ Global variables 7 ⟩ +≡
  char cat_name[256][12];
  eight_bits cat_index;
```

88. ⟨ Set initial values 10 ⟩ +≡

```
for (cat_index = 0; cat_index < 255; cat_index++) strcpy(cat_name[cat_index], "UNKNOWN");
strcpy(cat_name[exp], "exp");
strcpy(cat_name[unop], "unop");
strcpy(cat_name[binop], "binop");
strcpy(cat_name[ubinop], "ubinop");
strcpy(cat_name[cast], "cast");
strcpy(cat_name[question], "?");
strcpy(cat_name[lbrace], "{");
strcpy(cat_name[rbrace], "}");
strcpy(cat_name[decl_head], "decl_head");
strcpy(cat_name[comma], ",");
strcpy(cat_name[lpar], "(");
strcpy(cat_name[rpar], ")");
strcpy(cat_name[prelangle], "<");
strcpy(cat_name[prerangle], ">");
strcpy(cat_name[langle], "\\<");
strcpy(cat_name[colcol], "::");
strcpy(cat_name[base], "\\:");
strcpy(cat_name[decl], "decl");
strcpy(cat_name[struct_head], "struct_head");
strcpy(cat_name[alfop], "alfop");
strcpy(cat_name[stmt], "stmt");
strcpy(cat_name[function], "function");
strcpy(cat_name[fn_decl], "fn_decl");
strcpy(cat_name[else_like], "else_like");
strcpy(cat_name[semi], ";");
strcpy(cat_name[colon], ":");
strcpy(cat_name[tag], "tag");
strcpy(cat_name[if_head], "if_head");
strcpy(cat_name[else_head], "else_head");
strcpy(cat_name[if_clause], "if()");
strcpy(cat_name[lproc], "#{");
strcpy(cat_name[rproc], "#}");
strcpy(cat_name[insert], "insert");
strcpy(cat_name[section_scrap], "section");
strcpy(cat_name[dead], "@d");
strcpy(cat_name[public_like], "public");
strcpy(cat_name[operator_like], "operator");
strcpy(cat_name[new_like], "new");
strcpy(cat_name[catch_like], "catch");
strcpy(cat_name[for_like], "for");
strcpy(cat_name[do_like], "do");
strcpy(cat_name[if_like], "if");
strcpy(cat_name[delete_like], "delete");
strcpy(cat_name[raw_ubin], "ubinop?");
strcpy(cat_name[const_like], "const");
strcpy(cat_name[raw_int], "raw");
strcpy(cat_name[int_like], "int");
strcpy(cat_name[case_like], "case");
strcpy(cat_name[sizeof_like], "sizeof");
strcpy(cat_name[struct_like], "struct");
strcpy(cat_name[typedef_like], "typedef");
strcpy(cat_name[define_like], "define");
strcpy(cat_name[template_like], "template");
strcpy(cat_name[ftemplate], "ftemplate");
strcpy(cat_name[new_exp], "new_exp");
strcpy(cat_name[begin_arg], "@[");
strcpy(cat_name[end_arg], "@]");
strcpy(cat_name[0], "zero");
```

89. This code allows `CWEAVE` to display its parsing steps.

```
void print_cat(c)     /* symbolic printout of a category */
eight_bits c;
{
  printf(cat_name[c]);
}
```

90. 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.

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

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

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

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

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

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

!=	*binop*: \I	yes
<=	*binop*: \Z	yes
>=	*binop*: \G	yes
==	*binop*: \E	yes
&&	*binop*: \W	yes
||	*binop*: \V	yes
++	*unop*: \PP	yes
--	*unop*: \MM	yes
->	*binop*: \MG	yes
>>	*binop*: \GG	yes
<<	*binop*: \LL	yes
::	*colcol*: \DC	maybe
.*	*binop*: \PA	yes
->*	*binop*: \MGA	yes
...	*raw_int*: \,\ldots\,	yes
"string"	*exp*: \.{string with special characters quoted}	maybe
@=string@>	*exp*: \vb{string with special characters quoted}	maybe
@’7’	*exp*: \.{@’7’}	maybe
077 or \77	*exp*: \T{\~77}	maybe
0x7f	*exp*: \T{\^7f}	maybe
77	*exp*: \T{77}	maybe
77L	*exp*: \T{77\$L}	maybe
0.1E5	*exp*: \T{0.1_5}	maybe
+	*ubinop*: +	yes
-	*ubinop*: -	yes
*	*raw_ubin*: *	yes
/	*binop*: /	yes
<	*prelangle*: \langle	yes
=	*binop*: \K	yes
>	*prerangle*: \rangle	yes
.	*binop*: .	yes
|	*binop*: \OR	yes
^	*binop*: \XOR	yes
%	*binop*: \MOD	yes
?	*question*: \?	yes
!	*unop*: \R	yes
~	*unop*: \CM	yes
&	*raw_ubin*: \AND	yes
(	*lpar*: (	maybe
[	*lpar*: [	maybe
)	*rpar*:)	maybe
]	*rpar*:]	maybe
{	*lbrace*: {	yes
}	*lbrace*: }	yes

,	*comma*: ,	yes
;	*semi*: ;	maybe
:	*colon*: :	no
# (within line)	*ubinop*: \#	yes
# (at beginning)	*lproc*: *force* *preproc_line* \#	no
end of # line	*rproc*: *force*	no
identifier	*exp*: \\{identifier with underlines and dollar signs quoted}	maybe
and	*alfop*: **	yes
and_eq	*alfop*: **	yes
asm	*sizeof_like*: **	maybe
auto	*int_like*: **	maybe
bitand	*alfop*: **	yes
bitor	*alfop*: **	yes
bool	*raw_int*: **	maybe
break	*case_like*: **	maybe
case	*case_like*: **	maybe
catch	*catch_like*: **	maybe
char	*raw_int*: **	maybe
class	*struct_like*: **	maybe
clock_t	*raw_int*: **	maybe
compl	*alfop*: **	yes
const	*const_like*: **	maybe
const_cast	*raw_int*: **	maybe
continue	*case_like*: **	maybe
default	*case_like*: **	maybe
define	*define_like*: **	maybe
defined	*sizeof_like*: **	maybe
delete	*delete_like*: **	maybe
div_t	*raw_int*: **	maybe
do	*do_like*: **	maybe
double	*raw_int*: **	maybe
dynamic_cast	*raw_int*: **	maybe
elif	*if_like*: **	maybe
else	*else_like*: **	maybe
endif	*if_like*: **	maybe
enum	*struct_like*: **	maybe
error	*if_like*: **	maybe
explicit	*int_like*: **	maybe
export	*int_like*: **	maybe
extern	*int_like*: **	maybe
FILE	*raw_int*: **	maybe
float	*raw_int*: **	maybe
for	*for_like*: **	maybe
fpos_t	*raw_int*: **	maybe
friend	*int_like*: **	maybe
goto	*case_like*: **	maybe
if	*if_like*: **	maybe
ifdef	*if_like*: **	maybe
ifndef	*if_like*: **	maybe
include	*if_like*: **	maybe
inline	*int_like*: **	maybe
int	*raw_int*: **	maybe

`jmp_buf`	*raw_int*: `**`	maybe
`ldiv_t`	*raw_int*: `**`	maybe
`line`	*if_like*: `**`	maybe
`long`	*raw_int*: `**`	maybe
`make_pair`	*ftemplate*: `\\{make\_pair}`	maybe
`mutable`	*int_like*: `**`	maybe
`namespace`	*struct_like*: `**`	maybe
`new`	*new_like*: `**`	maybe
`not`	*alfop*: `**`	yes
`not_eq`	*alfop*: `**`	yes
`NULL`	*exp*: `\NULL`	yes
`offsetof`	*raw_int*: `**`	maybe
`operator`	*operator_like*: `**`	maybe
`or`	*alfop*: `**`	yes
`or_eq`	*alfop*: `**`	yes
`pragma`	*if_like*: `**`	maybe
`private`	*public_like*: `**`	maybe
`protected`	*public_like*: `**`	maybe
`ptrdiff_t`	*raw_int*: `**`	maybe
`public`	*public_like*: `**`	maybe
`register`	*int_like*: `**`	maybe
`reinterpret_cast`	*raw_int*: `**`	maybe
`return`	*case_like*: `**`	maybe
`short`	*raw_int*: `**`	maybe
`sig_atomic_t`	*raw_int*: `**`	maybe
`signed`	*raw_int*: `**`	maybe
`size_t`	*raw_int*: `**`	maybe
`sizeof`	*sizeof_like*: `**`	maybe
`static`	*int_like*: `**`	maybe
`static_cast`	*raw_int*: `**`	maybe
`struct`	*struct_like*: `**`	maybe
`switch`	*for_like*: `**`	maybe
`template`	*template_like*: `**`	maybe
`TeX`	*exp*: `\TeX`	yes
`this`	*exp*: `\this`	yes
`throw`	*case_like*: `**`	maybe
`time_t`	*raw_int*: `**`	maybe
`try`	*else_like*: `**`	maybe
`typedef`	*typedef_like*: `**`	maybe
`typeid`	*raw_int*: `**`	maybe
`typename`	*struct_like*: `**`	maybe
`undef`	*if_like*: `**`	maybe
`union`	*struct_like*: `**`	maybe
`unsigned`	*raw_int*: `**`	maybe
`using`	*int_like*: `**`	maybe
`va_dcl`	*decl*: `**`	maybe
`va_list`	*raw_int*: `**`	maybe
`virtual`	*int_like*: `**`	maybe
`void`	*raw_int*: `**`	maybe
`volatile`	*const_like*: `**`	maybe
`wchar_t`	*raw_int*: `**`	maybe
`while`	*for_like*: `**`	maybe

xor	*alfop*: **	yes
xor_eq	*alfop*: **	yes
@,	*insert*: \,	maybe
@|	*insert*: *opt* 0	maybe
@/	*insert*: *force*	no
@#	*insert*: *big_force*	no
@+	*insert*: *big_cancel* {} *break_space* {} *big_cancel*	no
@;	*semi*:	maybe
@[	*begin_arg*:	maybe
@]	*end_arg*:	maybe
@&	*insert*: \J	maybe
@h	*insert*: *force* \ATH *force*	no
@< section name @>	*section_scrap*: \X*n*:translated section name\X	maybe
@(section name @>	*section_scrap*: \X*n*:\.{section name with special characters quoted␣}\X	maybe
/*comment*/	*insert*: *cancel* \C{translated comment} *force*	no
//comment	*insert*: *cancel* \SHC{translated comment} *force*	no

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

92. 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* and *bsp* as shorthands for *indent*, *outdent*, *backup* and *break_space*, respectively.

	LHS	→ RHS	Translation	Example
0	$\left\{\begin{matrix}\mathit{any}\\ \mathit{any}\ \mathit{any}\\ \mathit{any}\ \mathit{any}\ \mathit{any}\end{matrix}\right\}\ \mathit{insert}$	$\to \left\{\begin{matrix}\mathit{any}\\ \mathit{any}\ \mathit{any}\\ \mathit{any}\ \mathit{any}\ \mathit{any}\end{matrix}\right\}$		stmt; /∗ comment ∗/
1	$\mathit{exp}\ \left\{\begin{matrix}\mathit{lbrace}\\ \mathit{int\_like}\\ \mathit{decl}\end{matrix}\right\}$	$\to \mathit{fn\_decl}\ \left\{\begin{matrix}\mathit{lbrace}\\ \mathit{int\_like}\\ \mathit{decl}\end{matrix}\right\}$	$F = E^*\ \mathit{in}\ \mathit{in}$	*main*( ){ *main*(*ac*, *av*) **int** *ac*;
2	$\mathit{exp}\ \mathit{unop}$	$\to \mathit{exp}$		$x{+}{+}$
3	$\mathit{exp}\ \left\{\begin{matrix}\mathit{binop}\\ \mathit{ubinop}\end{matrix}\right\}\ \mathit{exp}$	$\to \mathit{exp}$		x/y $x+y$
4	$\mathit{exp}\ \mathit{comma}\ \mathit{exp}$	$\to \mathit{exp}$	$EC\ \mathit{opt}9\ E$	$f(x,y)$

5	$\mathit{exp}\ \left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}\ \mathit{colon}$	$\to \mathit{exp}\ \left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}\ \mathit{base}$		**C**( ):
				C **int** *i*):

6	$\mathit{exp}\ \mathit{semi}$	$\to \mathit{stmt}$		$x = 0;$
7	$\mathit{exp}\ \mathit{colon}$	$\to \mathit{tag}$	E^*C	*found*:
8	$\mathit{exp}\ \mathit{rbrace}$	$\to \mathit{stmt}\ \mathit{rbrace}$		end of **enum** list
9	$\mathit{exp}\ \left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}\ \left\{\begin{matrix}\mathit{const\_like}\\ \mathit{case\_like}\end{matrix}\right\}$	$\to \mathit{exp}\ \left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}$	$\left\{\begin{matrix}R = R_{\sqcup}C\\ C_1 = C_{1\sqcup}C_2\end{matrix}\right\}$	*f*( ) **const** *f*(**int**) **throw**
10	$\mathit{exp}\ \left\{\begin{matrix}\mathit{exp}\\ \mathit{cast}\end{matrix}\right\}$	$\to \mathit{exp}$		*time*( )
11	$\mathit{lpar}\ \left\{\begin{matrix}\mathit{exp}\\ \mathit{ubinop}\end{matrix}\right\}\ \mathit{rpar}$	$\to \mathit{exp}$		(x) $(*)$
12	$\mathit{lpar}\ \mathit{rpar}$	$\to \mathit{exp}$	$L\backslash,R$	functions, declarations
13	$\mathit{lpar}\ \left\{\begin{matrix}\mathit{decl\_head}\\ \mathit{int\_like}\\ \mathit{cast}\end{matrix}\right\}\ \mathit{rpar}$	$\to \mathit{cast}$		(**char** ∗)
14	$\mathit{lpar}\ \left\{\begin{matrix}\mathit{decl\_head}\\ \mathit{int\_like}\\ \mathit{exp}\end{matrix}\right\}\ \mathit{comma}$	$\to \mathit{lpar}$	$L\left\{\begin{matrix}D\\ I\\ E\end{matrix}\right\}C\ \mathit{opt}9$	(**int**,
15	$\mathit{lpar}\ \left\{\begin{matrix}\mathit{stmt}\\ \mathit{decl}\end{matrix}\right\}$	$\to \mathit{lpar}$	$\left\{\begin{matrix}LS_{\sqcup}\\ LD_{\sqcup}\end{matrix}\right\}$	($k = 5$; (**int** $k = 5$;
16	$\mathit{unop}\ \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\to \mathit{exp}$		$\neg x$ ∼**C**
17	$\mathit{ubinop}\ \mathit{cast}\ \mathit{rpar}$	$\to \mathit{cast}\ \mathit{rpar}$	$C = \texttt{\{}U\texttt{\}}C$	∗**CPtr**)
18	$\mathit{ubinop}\ \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\to \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\texttt{\{}U\texttt{\}}\left\{\begin{matrix}E\\ I\end{matrix}\right\}$	$*x$ ∗**CPtr**
19	$\mathit{ubinop}\ \mathit{binop}$	$\to \mathit{binop}$	$\mathit{math\_rel}\ U\texttt{\{}B\texttt{\}}\texttt{\}}$	$*=$
20	$\mathit{binop}\ \mathit{binop}$	$\to \mathit{binop}$	$\mathit{math\_rel}\ \texttt{\{}B_1\texttt{\}}\texttt{\{}B_2\texttt{\}}\texttt{\}}$	$\gg=$

21	$\textit{cast}\ \begin{Bmatrix}\textit{lpar}\\ \textit{exp}\end{Bmatrix}$	$\rightarrow \begin{Bmatrix}\textit{lpar}\\ \textit{exp}\end{Bmatrix}$	$\begin{Bmatrix}CL\\ C\sqcup E\end{Bmatrix}$	(**double**)$(x+2)$ (**double**) x
22	$\textit{cast}\ \textit{semi}$	$\rightarrow \textit{exp}\ \textit{semi}$		(**int**);
23	$\textit{sizeof\_like}\ \textit{cast}$	$\rightarrow \textit{exp}$		**sizeof**(**double**)
24	$\textit{sizeof\_like}\ \textit{exp}$	$\rightarrow \textit{exp}$	$S\sqcup E$	**sizeof** x
25	$\textit{int\_like}\ \begin{Bmatrix}\textit{int\_like}\\ \textit{struct\_like}\end{Bmatrix}$	$\rightarrow \begin{Bmatrix}\textit{int\_like}\\ \textit{struct\_like}\end{Bmatrix}$	$I\sqcup\begin{Bmatrix}I\\ S\end{Bmatrix}$	**extern char**
26	$\textit{int\_like}\ \textit{exp}\ \begin{Bmatrix}\textit{raw\_int}\\ \textit{struct\_like}\end{Bmatrix}$	$\rightarrow \textit{int\_like}\ \begin{Bmatrix}\textit{raw\_int}\\ \textit{struct\_like}\end{Bmatrix}$		**extern**`"Ada"` **int**
27	$\textit{int\_like}\ \begin{Bmatrix}\textit{exp}\\ \textit{ubinop}\\ \textit{colon}\end{Bmatrix}$	$\rightarrow \textit{decl\_head}\ \begin{Bmatrix}\textit{exp}\\ \textit{ubinop}\\ \textit{colon}\end{Bmatrix}$	$D = I\sqcup$	**int** x **int** $*x$ **unsigned** :
28	$\textit{int\_like}\ \begin{Bmatrix}\textit{semi}\\ \textit{binop}\end{Bmatrix}$	$\rightarrow \textit{decl\_head}\ \begin{Bmatrix}\textit{semi}\\ \textit{binop}\end{Bmatrix}$		**int** x; **int** f(**int** $= 4$)
29	$\textit{public\_like}\ \textit{colon}$	$\rightarrow \textit{tag}$		**private**:
30	$\textit{public\_like}$	$\rightarrow \textit{int\_like}$		**private**
31	$\textit{colcol}\ \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}$	$\rightarrow \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}$	$\textit{qualifier}\ C\ \begin{Bmatrix}E\\ I\end{Bmatrix}$	**C**::x
32	$\textit{colcol}\ \textit{colcol}$	$\rightarrow \textit{colcol}$		**C**::**B**::
33	$\textit{decl\_head}\ \textit{comma}$	$\rightarrow \textit{decl\_head}$	$DC\sqcup$	**int** x,
34	$\textit{decl\_head}\ \textit{ubinop}$	$\rightarrow \textit{decl\_head}$	$D\{U\}$	**int** $*$
†35	$\textit{decl\_head}\ \textit{exp}$	$\rightarrow \textit{decl\_head}$	DE^*	**int** x
36	$\textit{decl\_head}\ \begin{Bmatrix}\textit{binop}\\ \textit{colon}\end{Bmatrix}\ \textit{exp}\ \begin{Bmatrix}\textit{comma}\\ \textit{semi}\\ \textit{rpar}\end{Bmatrix}$	$\rightarrow \textit{decl\_head}\ \begin{Bmatrix}\textit{comma}\\ \textit{semi}\\ \textit{rpar}\end{Bmatrix}$	$D = D\begin{Bmatrix}B\\ C\end{Bmatrix}E$	**int** f(**int** $x = 2$) **int** b : 1
37	$\textit{decl\_head}\ \textit{cast}$	$\rightarrow \textit{decl\_head}$		**int** f(**int**)
38	$\textit{decl\_head}\ \begin{Bmatrix}\textit{int\_like}\\ \textit{lbrace}\\ \textit{decl}\end{Bmatrix}$	$\rightarrow \textit{fn\_decl}\ \begin{Bmatrix}\textit{int\_like}\\ \textit{lbrace}\\ \textit{decl}\end{Bmatrix}$	$F = D\ \textit{in}\ \textit{in}$	**long** $\textit{time}$( ) {
39	$\textit{decl\_head}\ \textit{semi}$	$\rightarrow \textit{decl}$		**int** n;
40	$\textit{decl}\ \textit{decl}$	$\rightarrow \textit{decl}$	$D_1\ \textit{force}\ D_2$	**int** n; **double** x;
41	$\textit{decl}\ \begin{Bmatrix}\textit{stmt}\\ \textit{function}\end{Bmatrix}$	$\rightarrow \begin{Bmatrix}\textit{stmt}\\ \textit{function}\end{Bmatrix}$	$D\ \textit{big\_force}\ \begin{Bmatrix}S\\ F\end{Bmatrix}$	**extern** n; $\textit{main}$ (){ }
42	$\textit{base}\ \begin{Bmatrix}\textit{int\_like}\\ \textit{exp}\end{Bmatrix}\ \textit{comma}$	$\rightarrow \textit{base}$	$B\sqcup\begin{Bmatrix}I\\ E\end{Bmatrix}C\ \textit{opt}9$	: **public A**, : $i(5)$,
43	$\textit{base}\ \begin{Bmatrix}\textit{int\_like}\\ \textit{exp}\end{Bmatrix}\ \textit{lbrace}$	$\rightarrow \textit{lbrace}$	$B\sqcup\begin{Bmatrix}I\\ E\end{Bmatrix}\sqcup L$	**D** : **public A** {
44	$\textit{struct\_like}\ \textit{lbrace}$	$\rightarrow \textit{struct\_head}$	$S\sqcup L$	**struct** {
45	$\textit{struct\_like}\ \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}\ \textit{semi}$	$\rightarrow \textit{decl\_head}\ \textit{semi}$	$S\sqcup\begin{Bmatrix}E^{**}\\ I^{**}\end{Bmatrix}$	**struct forward**;
46	$\textit{struct\_like}\ \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}\ \textit{lbrace}$	$\rightarrow \textit{struct\_head}$	$S\sqcup\begin{Bmatrix}E^{**}\\ I^{**}\end{Bmatrix}\sqcup L$	**struct name_info** {
47	$\textit{struct\_like}\ \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}\ \textit{colon}$	$\rightarrow \textit{struct\_like}\ \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}\ \textit{base}$		**class C** :
†48	$\textit{struct\_like}\ \begin{Bmatrix}\textit{exp}\\ \textit{int\_like}\end{Bmatrix}$	$\rightarrow \textit{int\_like}$	$S\sqcup\begin{Bmatrix}E\\ I\end{Bmatrix}$	**struct name_info** z;

49	*struct_head* $\left\{\begin{array}{c}\mathit{decl}\\ \mathit{stmt}\\ \mathit{function}\end{array}\right\}$ *rbrace*	→ *int_like*	*S in force* $\left\{\begin{array}{c}D\\ S\\ F\end{array}\right\}$ *out force R*	**struct** { declaration }
50	*struct_head rbrace*	→ *int_like*	*S*\,*R*	**class C** { }
51	*fn_decl decl*	→ *fn_decl*	*F force D*	$f(z)$ **double** z;
52	*fn_decl stmt*	→ *function*	*F out out force S*	*main*() ...
53	*function* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$	→ $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$	*F big_force* $\left\{\begin{array}{c}S\\ D\\ F\end{array}\right\}$	outer block
54	*lbrace rbrace*	→ *stmt*	*L*\,*R*	empty statement
55	*lbrace* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$ *rbrace*	→ *stmt*	*force L in force S force back R out force*	compound statement
56	*lbrace exp* [*comma*] *rbrace*	→ *exp*		initializer
57	*if_like exp*	→ *if_clause*	*I*␣*E*	**if** (z)
58	*else_like colon*	→ *else_like base*		**try** :
59	*else_like lbrace*	→ *else_head lbrace*		**else** {
60	*else_like stmt*	→ *stmt*	*force E in bsp S out force*	**else** $x = 0$;
61	*else_head* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{exp}\end{array}\right\}$	→ *stmt*	*force E bsp noop cancel S bsp*	**else** { $x = 0$; }
62	*if_clause lbrace*	→ *if_head lbrace*		**if** (x) {
63	*if_clause stmt else_like if_like*	→ *if_like*	*force I in bsp S out force E*␣*I*	**if** (x) y; **else if**
64	*if_clause stmt else_like*	→ *else_like*	*force I in bsp S out force E*	**if** (x) y; **else**
65	*if_clause stmt*	→ *else_like stmt*		**if** (x)
66	*if_head* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{exp}\end{array}\right\}$ *else_like if_like*	→ *if_like*	*force I bsp noop cancel S force E*␣*I*	**if** (x) { y; } **else if**
67	*if_head* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{exp}\end{array}\right\}$ *else_like*	→ *else_like*	*force I bsp noop cancel S force E*	**if** (x) { y; } **else**
68	*if_head* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{exp}\end{array}\right\}$	→ *else_head* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{exp}\end{array}\right\}$		**if** (x) { y; }
69	*do_like stmt else_like semi*	→ *stmt*	*D bsp noop cancel S cancel noop bsp ES*	**do** $f(x)$; **while** $(g(x))$;
70	*case_like semi*	→ *stmt*		**return**;
71	*case_like colon*	→ *tag*		**default**:
72	*case_like exp*	→ *exp*	*C*␣*E*	**return** 0
73	*catch_like* $\left\{\begin{array}{c}\mathit{cast}\\ \mathit{exp}\end{array}\right\}$	→ *fn_decl*	$C\left\{\begin{array}{c}C\\ E\end{array}\right\}$ *in in*	**catch**(...)
74	*tag tag*	→ *tag*	T_1 *bsp* T_2	**case** 0: **case** 1:
75	*tag* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$	→ $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$	*force back T bsp S*	**case** 0: $z = 0$;
†76	*stmt* $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$	→ $\left\{\begin{array}{c}\mathit{stmt}\\ \mathit{decl}\\ \mathit{function}\end{array}\right\}$	$S\left\{\begin{array}{c}\mathit{force}\ S\\ \mathit{big\_force}\ D\\ \mathit{big\_force}\ F\end{array}\right\}$	$x = 1$; $y = 2$;
77	*semi*	→ *stmt*	␣*S*	empty statement
†78	*lproc* $\left\{\begin{array}{c}\mathit{if\_like}\\ \mathit{else\_like}\\ \mathit{define\_like}\end{array}\right\}$	→ *lproc*		**#include** **#else** **#define**
79	*lproc rproc*	→ *insert*		**#endif**

80	$\mathit{lproc} \left\{\begin{matrix}\mathit{exp}\ [\mathit{exp}] \\ \mathit{function}\end{matrix}\right\} \mathit{rproc}$	→ *insert*	$I_{\text{␣}}\left\{\begin{matrix}E[\text{␣}\backslash 5E] \\ F\end{matrix}\right\}$	**#define** *a* 1 **#define** *a* { *b*; }
81	*section_scrap semi*	→ *stmt*	*MS force*	⟨ section name ⟩;
82	*section_scrap*	→ *exp*		⟨ section name ⟩
83	*insert any*	→ *any*		`|#include|`
84	*prelangle*	→ *binop*	`<`	< not in template
85	*prerangle*	→ *binop*	`>`	> not in template
86	*langle prerangle*	→ *cast*	$L\backslash,P$	⟨ ⟩
87	$\mathit{langle} \left\{\begin{matrix}\mathit{decl\_head} \\ \mathit{int\_like} \\ \mathit{exp}\end{matrix}\right\} \mathit{prerangle}$	→ *cast*		⟨**class C**⟩
88	$\mathit{langle} \left\{\begin{matrix}\mathit{decl\_head} \\ \mathit{int\_like} \\ \mathit{exp}\end{matrix}\right\} \mathit{comma}$	→ *langle*	$L\left\{\begin{matrix}D \\ I \\ E\end{matrix}\right\} C\ \mathit{opt}9$	⟨**class C**,
89	*template_like exp prelangle*	→ *template_like exp langle*		**template** *a*⟨100⟩
90	$\mathit{template\_like} \left\{\begin{matrix}\mathit{exp} \\ \mathit{raw\_int}\end{matrix}\right\}$	→ $\left\{\begin{matrix}\mathit{exp} \\ \mathit{raw\_int}\end{matrix}\right\}$	$T_{\text{␣}}\left\{\begin{matrix}E \\ R\end{matrix}\right\}$	**C** :: **template** *a*( )
91	*template_like*	→ *raw_int*		**template**⟨**class T**⟩
92	*new_like lpar exp rpar*	→ *new_like*		**new**(*nothrow*)
93	*new_like cast*	→ *exp*	$N_{\text{␣}}C$	**new** (**int** ∗)
†94	*new_like*	→ *new_exp*		**new C**( )
95	$\mathit{new\_exp} \left\{\begin{matrix}\mathit{int\_like} \\ \mathit{const\_like}\end{matrix}\right\}$	→ *new_exp*	$N_{\text{␣}}\left\{\begin{matrix}I \\ C\end{matrix}\right\}$	**new const int**
96	$\mathit{new\_exp}\ \mathit{struct\_like} \left\{\begin{matrix}\mathit{exp} \\ \mathit{int\_like}\end{matrix}\right\}$	→ *new_exp*	$N_{\text{␣}}S_{\text{␣}}\left\{\begin{matrix}E \\ I\end{matrix}\right\}$	**new struct S**
97	*new_exp raw_ubin*	→ *new_exp*	$N\{R\}$	**new int**∗[2]
98	$\mathit{new\_exp} \left\{\begin{matrix}\mathit{lpar} \\ \mathit{exp}\end{matrix}\right\}$	→ $\mathit{exp} \left\{\begin{matrix}\mathit{lpar} \\ \mathit{exp}\end{matrix}\right\}$	$E = N\left\{\begin{matrix} \\ \text{␣}\end{matrix}\right\}$	**operator**[ ](**int**) **new int**(2)
†99	*new_exp*	→ *exp*		**new int**;
100	*ftemplate prelangle*	→ *ftemplate langle*		*make_pair*⟨**int**, **int**⟩
101	*ftemplate*	→ *exp*		*make_pair*(1, 2)
102	*for_like exp*	→ *else_like*	$F_{\text{␣}}E$	**while** (1)
103	*raw_ubin const_like*	→ *raw_ubin*	$RC\backslash_{\text{␣}}$	∗**const** *x*
104	*raw_ubin*	→ *ubinop*		∗ *x*
105	*const_like*	→ *int_like*		**const** *x*
106	*raw_int prelangle*	→ *raw_int langle*		**C**⟨
107	*raw_int colcol*	→ *colcol*		**C** ::
108	*raw_int cast*	→ *raw_int*		**C**⟨**class T**⟩
109	*raw_int lpar*	→ *exp lpar*		**complex**(*x*, *y*)
†110	*raw_int*	→ *int_like*		**complex** *z*
†111	$\mathit{operator\_like} \left\{\begin{matrix}\mathit{binop} \\ \mathit{unop} \\ \mathit{ubinop}\end{matrix}\right\}$	→ *exp*	$O\{\left\{\begin{matrix}B \\ U \\ U\end{matrix}\right\}\}$	**operator**+
112	$\mathit{operator\_like} \left\{\begin{matrix}\mathit{new\_like} \\ \mathit{delete\_like}\end{matrix}\right\}$	→ *exp*	$O_{\text{␣}}\left\{\begin{matrix}N \\ S\end{matrix}\right\}$	**operator delete**
113	*operator_like comma*	→ *exp*		**operator**,
†114	*operator_like*	→ *new_exp*		**operator char**∗
115	$\mathit{typedef\_like} \left\{\begin{matrix}\mathit{int\_like} \\ \mathit{cast}\end{matrix}\right\} \left\{\begin{matrix}\mathit{comma} \\ \mathit{semi}\end{matrix}\right\}$	→ $\mathit{typedef\_like}\ \mathit{exp} \left\{\begin{matrix}\mathit{comma} \\ \mathit{semi}\end{matrix}\right\}$		**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* $\left\{\begin{matrix}\textit{cast}\\ \textit{ubinop}\end{matrix}\right\}$	→ *typedef_like* $\left\{\begin{matrix}\textit{cast}\\ \textit{ubinop}\end{matrix}\right\}$	$\left\{\begin{matrix}C = \{U\}C\\ U_2 = \{U_1\}U_2\end{matrix}\right\}$	**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* $\left\{\begin{matrix}\textit{colon}\\ \textit{base}\end{matrix}\right\}$	→ *binop*		? *x* : ? *f*( ) :
124	*begin_arg end_arg*	→ *exp*		`@[`**char**∗`@]`
125	*any_other end_arg*	→ *end_arg*		**char**∗`@]`

†Notes

Rule 35: The *exp* must not be immediately followed by *lpar*, *exp*, or *cast*.
Rule 48: The *exp* or *int_like* must not be immediately followed by *base*.
Rule 76: The *force* in the *stmt* line becomes *bsp* if `CWEAVE` has been invoked with the `-f` option.
Rule 78: The *define_like* case calls *make_underlined* on the following scrap.
Rule 94: The *new_like* must not be immediately followed by *lpar*.
Rule 99: The *new_exp* must not be immediately followed by *raw_int*, *struct_like*, or *colcol*.
Rule 110: The *raw_int* must not be immediately followed by *langle*.
Rule 111: The operator after *operator_like* must not be immediately followed by a *binop*.
Rule 114: The *operator_like* must not be immediately followed by *raw_ubin*.
Rule 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'.

93. 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 $pp\to cat$, $(pp + 1)\to cat$, ... 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 8 ⟩ +≡
  typedef struct {
    eight_bits cat;
    eight_bits mathness;
    union {
      text_pointer Trans;
      ⟨ Rest of trans_plus union 222 ⟩
    } trans_plus;
  } scrap;
  typedef scrap *scrap_pointer;
```

```
94. #define trans trans_plus.Trans   /* translation texts of scraps */
⟨ Global variables 7 ⟩ +≡
  scrap scrap_info[max_scraps];   /* memory array for scraps */
  scrap_pointer scrap_info_end = scrap_info + max_scraps − 1;   /* end of scrap_info */
  scrap_pointer pp;   /* current position for reducing productions */
  scrap_pointer scrap_base;   /* beginning of the current scrap sequence */
  scrap_pointer scrap_ptr;   /* ending of the current scrap sequence */
  scrap_pointer lo_ptr;   /* last scrap that has been examined */
  scrap_pointer hi_ptr;   /* first scrap that has not been examined */
  scrap_pointer max_scr_ptr;   /* largest value assumed by scrap_ptr */
```

```
95. ⟨ Set initial values 10 ⟩ +≡
  scrap_base = scrap_info + 1;
  max_scr_ptr = scrap_ptr = scrap_info;
```

96. 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 '| ... |' */

  void print_text(p)   /* 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/id_flag) {
        case 1: printf("\\\\{");
          print_id((name_dir + r));
          printf("}");
          break;   /* id_flag */
        case 2: printf("\\&{");
          print_id((name_dir + r));
          printf("}");
          break;   /* res_flag */
        case 3: printf("<");
          print_section_name((name_dir + r));
          printf(">");
          break;   /* section_flag */
        case 4: printf("[[%d]]", r);
          break;   /* tok_flag */
        case 5: printf("|[[%d]]|", r);
          break;   /* inner_tok_flag */
        default: ⟨Print token r in symbolic form 97⟩;
        }
      }
    fflush(stdout);
  }
```

97. ⟨Print token *r* in symbolic form 97⟩ ≡

```
  switch (r) {
  case math_rel: printf("\\mathrel{");
    break;
  case big_cancel: printf("[ccancel]");
    break;
  case cancel: printf("[cancel]");
    break;
  case indent: printf("[indent]");
    break;
  case outdent: printf("[outdent]");
    break;
  case backup: printf("[backup]");
    break;
  case opt: printf("[opt]");
    break;
  case break_space: printf("[break]");
    break;
  case force: printf("[force]");
    break;
  case big_force: printf("[fforce]");
    break;
  case preproc_line: printf("[preproc]");
    break;
  case quoted_char: j++;
    printf("[%o]", (unsigned) *j);
    break;
  case end_translation: printf("[quit]");
    break;
  case inserted: printf("[inserted]");
    break;
  default: putxchar(r);
  }
```

This code is used in section 96.

98. 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* → *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*→*trans* and $(pp + 1)$→*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_app4* 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_app4(a) big_app3(a); big_app1(a + 3)
#define app(a) *(tok_ptr++) = a
#define app1(a) *(tok_ptr++) = tok_flag + (int)((a)→trans − tok_start)

⟨ Global variables 7 ⟩ +≡
  int cur_mathness, init_mathness;
```

```
99. void app_str(s)
      char *s;
  {
    while (*s) app_tok(*(s++));
  }
  void big_app(a)
      token a;
  {
    if (a ≡ '␣' ∨ (a ≥ big_cancel ∧ a ≤ big_force))   /* non-math token */
    {
      if (cur_mathness ≡ maybe_math) init_mathness = no_math;
      else if (cur_mathness ≡ yes_math) app_str("{}$");
      cur_mathness = no_math;
    }
    else {
      if (cur_mathness ≡ maybe_math) init_mathness = yes_math;
      else if (cur_mathness ≡ no_math) app_str("${}");
      cur_mathness = yes_math;
    }
    app(a);
  }
  void big_app1(a)
      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));
  }
```

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

```
#define cat1 (pp + 1)→cat
#define cat2 (pp + 2)→cat
#define cat3 (pp + 3)→cat
#define lhs_not_simple
          (pp→cat ≠ public_like ∧ pp→cat ≠ semi ∧ pp→cat ≠ prelangle ∧ pp→cat ≠ prerangle ∧ pp→cat ≠
              template_like ∧ pp→cat ≠ new_like ∧ pp→cat ≠ new_exp ∧ pp→cat ≠ ftemplate ∧ pp→cat ≠
              raw_ubin ∧ pp→cat ≠ const_like ∧ pp→cat ≠ raw_int ∧ pp→cat ≠ operator_like)
            /* not a production with left side length 1 */

⟨ Match a production at pp, or increase pp if there is no match 100 ⟩ ≡
  {
    if (cat1 ≡ end_arg ∧ lhs_not_simple)
      if (pp→cat ≡ begin_arg) squash(pp, 2, exp, −2, 124);
      else squash(pp, 2, end_arg, −1, 125);
    else if (cat1 ≡ insert) squash(pp, 2, pp→cat, −2, 0);
    else if (cat2 ≡ insert) squash(pp + 1, 2, (pp + 1)→cat, −1, 0);
    else if (cat3 ≡ insert) squash(pp + 2, 2, (pp + 2)→cat, 0, 0);
    else
      switch (pp→cat) {
      case exp: ⟨ Cases for exp 107 ⟩; break;
      case lpar: ⟨ Cases for lpar 108 ⟩; break;
      case unop: ⟨ Cases for unop 109 ⟩; break;
      case ubinop: ⟨ Cases for ubinop 110 ⟩; break;
      case binop: ⟨ Cases for binop 111 ⟩; break;
      case cast: ⟨ Cases for cast 112 ⟩; break;
      case sizeof_like: ⟨ Cases for sizeof_like 113 ⟩; break;
      case int_like: ⟨ Cases for int_like 114 ⟩; break;
      case public_like: ⟨ Cases for public_like 115 ⟩; break;
      case colcol: ⟨ Cases for colcol 116 ⟩; break;
      case decl_head: ⟨ Cases for decl_head 117 ⟩; break;
      case decl: ⟨ Cases for decl 118 ⟩; break;
      case base: ⟨ Cases for base 119 ⟩; break;
      case struct_like: ⟨ Cases for struct_like 120 ⟩; break;
      case struct_head: ⟨ Cases for struct_head 121 ⟩; break;
      case fn_decl: ⟨ Cases for fn_decl 122 ⟩; break;
      case function: ⟨ Cases for function 123 ⟩; break;
      case lbrace: ⟨ Cases for lbrace 124 ⟩; break;
      case if_like: ⟨ Cases for if_like 125 ⟩; break;
      case else_like: ⟨ Cases for else_like 126 ⟩; break;
      case else_head: ⟨ Cases for else_head 127 ⟩; break;
      case if_clause: ⟨ Cases for if_clause 128 ⟩; break;
      case if_head: ⟨ Cases for if_head 129 ⟩; break;
      case do_like: ⟨ Cases for do_like 130 ⟩; break;
      case case_like: ⟨ Cases for case_like 131 ⟩; break;
      case catch_like: ⟨ Cases for catch_like 132 ⟩; break;
      case tag: ⟨ Cases for tag 133 ⟩; break;
      case stmt: ⟨ Cases for stmt 134 ⟩; break;
      case semi: ⟨ Cases for semi 135 ⟩; break;
      case lproc: ⟨ Cases for lproc 136 ⟩; break;
      case section_scrap: ⟨ Cases for section_scrap 137 ⟩; break;
      case insert: ⟨ Cases for insert 138 ⟩; break;
      case prelangle: ⟨ Cases for prelangle 139 ⟩; break;
      case prerangle: ⟨ Cases for prerangle 140 ⟩; break;
      case langle: ⟨ Cases for langle 141 ⟩; break;
      case template_like: ⟨ Cases for template_like 142 ⟩; break;
      case new_like: ⟨ Cases for new_like 143 ⟩; break;
      case new_exp: ⟨ Cases for new_exp 144 ⟩; break;
      case ftemplate: ⟨ Cases for ftemplate 145 ⟩; break;
      case for_like: ⟨ Cases for for_like 146 ⟩; break;
      case raw_ubin: ⟨ Cases for raw_ubin 147 ⟩; break;
      case const_like: ⟨ Cases for const_like 148 ⟩; break;
      case raw_int: ⟨ Cases for raw_int 149 ⟩; break;
      case operator_like: ⟨ Cases for operator_like 150 ⟩; break;
      case typedef_like: ⟨ Cases for typedef_like 151 ⟩; break;
      case delete_like: ⟨ Cases for delete_like 152 ⟩; break;
      case question: ⟨ Cases for question 153 ⟩; break;
      }
    pp++;     /* if no match was found, we move to the right */
  }
```

This code is used in section 156.

101. 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 */

  token_pointer find_first_ident(p)
      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;
  }
```

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

```
void make_reserved(p)     /∗ make the first identifier in p→trans 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(p→trans)) ≤ operator_found) return;     /∗ this should not happen ∗/
  tok_value = ∗tok_loc;
  for ( ; p ≤ scrap_ptr; p ≡ lo_ptr ? p = hi_ptr : p++) {
    if (p→cat ≡ exp) {
      if (∗∗(p→trans) ≡ tok_value) {
        p→cat = raw_int;
        ∗∗(p→trans) = 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;
}
```

103. 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*.

```
void make_underlined(p)     /∗ 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);
}
```

104. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void underline_xref( );
```

105. **void** *underline_xref* (*p*) *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 106 ⟩;
}
```

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

```
⟨ Insert new cross-reference at q, not at beginning of list 106 ⟩ ≡
  append_xref(0);    /* this number doesn't matter */
  xref_ptr→xlink = (xref_pointer) p→xref;
  r = xref_ptr;
  p→xref = (char *) xref_ptr;
  while (r→xlink ≠ q) {
    r→num = r→xlink→num;
    r = r→xlink;
  }
  r→num = m;    /* everything from q on is left undisturbed */
```

This code is used in section 105.

107. 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 107 ⟩ ≡
  if (cat1 ≡ lbrace ∨ cat1 ≡ int_like ∨ cat1 ≡ decl) {
    make_underlined(pp);
    big_app1(pp);
    big_app(indent);
    app(indent);
    reduce(pp, 1, fn_decl, 0, 1);
  }
  else if (cat1 ≡ unop) squash(pp, 2, exp, −2, 2);
  else if ((cat1 ≡ binop ∨ cat1 ≡ ubinop) ∧ cat2 ≡ exp) squash(pp, 3, exp, −2, 3);
  else if (cat1 ≡ comma ∧ cat2 ≡ exp) {
    big_app2(pp);
    app(opt);
    app('9');
    big_app1(pp + 2);
    reduce(pp, 3, exp, −2, 4);
  }
  else if (cat1 ≡ lpar ∧ cat2 ≡ rpar ∧ cat3 ≡ colon) squash(pp + 3, 1, base, 0, 5);
  else if (cat1 ≡ cast ∧ cat2 ≡ colon) squash(pp + 2, 1, base, 0, 5);
  else if (cat1 ≡ semi) squash(pp, 2, stmt, −1, 6);
  else if (cat1 ≡ colon) {
    make_underlined(pp);
    squash(pp, 2, tag, −1, 7);
  }
  else if (cat1 ≡ rbrace) squash(pp, 1, stmt, −1, 8);
  else if (cat1 ≡ lpar ∧ cat2 ≡ rpar ∧ (cat3 ≡ const_like ∨ cat3 ≡ case_like)) {
    big_app1(pp + 2);
    big_app('␣');
    big_app1(pp + 3);
    reduce(pp + 2, 2, rpar, 0, 9);
  }
  else if (cat1 ≡ cast ∧ (cat2 ≡ const_like ∨ cat2 ≡ case_like)) {
    big_app1(pp + 1);
    big_app('␣');
    big_app1(pp + 2);
    reduce(pp + 1, 2, cast, 0, 9);
  }
  else if (cat1 ≡ exp ∨ cat1 ≡ cast) squash(pp, 2, exp, −2, 10);
```

This code is used in section 100.

108. ⟨ Cases for *lpar* 108 ⟩ ≡

```
  if ((cat1 ≡ exp ∨ cat1 ≡ ubinop) ∧ cat2 ≡ rpar) squash(pp, 3, exp, −2, 11);
  else if (cat1 ≡ rpar) {
    big_app1(pp);
    app('\\');
    app(',');
    big_app1(pp + 1);
    reduce(pp, 2, exp, −2, 12);
  }
  else if ((cat1 ≡ decl_head ∨ cat1 ≡ int_like ∨ cat1 ≡ cast) ∧ cat2 ≡ rpar) squash(pp, 3, cast, −2, 13);
  else if ((cat1 ≡ decl_head ∨ cat1 ≡ int_like ∨ cat1 ≡ exp) ∧ cat2 ≡ comma) {
    big_app3(pp);
    app(opt);
    app('9');
    reduce(pp, 3, lpar, −1, 14);
  }
  else if (cat1 ≡ stmt ∨ cat1 ≡ decl) {
    big_app2(pp);
    big_app('␣');
    reduce(pp, 2, lpar, −1, 15);
  }
```

This code is used in section 100.

109. ⟨ Cases for *unop* 109 ⟩ ≡

```
  if (cat1 ≡ exp ∨ cat1 ≡ int_like) squash(pp, 2, exp, −2, 16);
```

This code is used in section 100.

110. ⟨ Cases for *ubinop* 110 ⟩ ≡

```
  if (cat1 ≡ cast ∧ cat2 ≡ rpar) {
    big_app('{');
    big_app1(pp);
    big_app('}');
    big_app1(pp + 1);
    reduce(pp, 2, cast, −2, 17);
  }
  else if (cat1 ≡ exp ∨ cat1 ≡ int_like) {
    big_app('{');
    big_app1(pp);
    big_app('}');
    big_app1(pp + 1);
    reduce(pp, 2, cat1, −2, 18);
  }
  else if (cat1 ≡ binop) {
    big_app(math_rel);
    big_app1(pp);
    big_app('{');
    big_app1(pp + 1);
    big_app('}');
    big_app('}');
    reduce(pp, 2, binop, −1, 19);
  }
```

This code is used in section 100.

111. ⟨ Cases for *binop* 111 ⟩ ≡

```
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 100.

112. ⟨ Cases for *cast* 112 ⟩ ≡

```
if (cat1 ≡ lpar) squash(pp, 2, lpar, −1, 21);
else if (cat1 ≡ exp) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, exp, −2, 21);
}
else if (cat1 ≡ semi) squash(pp, 1, exp, −2, 22);
```

This code is used in section 100.

113. ⟨ Cases for *sizeof_like* 113 ⟩ ≡

```
if (cat1 ≡ cast) squash(pp, 2, exp, −2, 23);
else if (cat1 ≡ exp) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, exp, −2, 24);
}
```

This code is used in section 100.

114. ⟨ Cases for *int_like* 114 ⟩ ≡

```
if (cat1 ≡ int_like ∨ cat1 ≡ struct_like) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, cat1, −2, 25);
}
else if (cat1 ≡ exp ∧ (cat2 ≡ raw_int ∨ cat2 ≡ struct_like)) squash(pp, 2, int_like, −2, 26);
else if (cat1 ≡ exp ∨ cat1 ≡ ubinop ∨ cat1 ≡ colon) {
    big_app1(pp);
    big_app('␣');
    reduce(pp, 1, decl_head, −1, 27);
}
else if (cat1 ≡ semi ∨ cat1 ≡ binop) squash(pp, 1, decl_head, 0, 28);
```

This code is used in section 100.

115. ⟨ Cases for *public_like* 115 ⟩ ≡

```
if (cat1 ≡ colon) squash(pp, 2, tag, −1, 29);
else squash(pp, 1, int_like, −2, 30);
```

This code is used in section 100.

116. ⟨ Cases for *colcol* 116 ⟩ ≡

```
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 100.

117. ⟨ Cases for *decl_head* 117 ⟩ ≡

```
if (cat1 ≡ comma) {
  big_app2(pp);
  big_app('␣');
  reduce(pp, 2, decl_head, −1, 33);
}
else if (cat1 ≡ ubinop) {
  big_app1(pp);
  big_app('{');
  big_app1(pp + 1);
  big_app('}');
  reduce(pp, 2, decl_head, −1, 34);
}
else if (cat1 ≡ exp ∧ cat2 ≠ lpar ∧ 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_app1(pp);
  big_app(indent);
  app(indent);
  reduce(pp, 1, fn_decl, 0, 38);
}
else if (cat1 ≡ semi) squash(pp, 2, decl, −1, 39);
```

This code is used in section 100.

118. ⟨ Cases for *decl* 118 ⟩ ≡

```
if (cat1 ≡ decl) {
  big_app1(pp);
  big_app(force);
  big_app1(pp + 1);
  reduce(pp, 2, decl, −1, 40);
}
else if (cat1 ≡ stmt ∨ cat1 ≡ function) {
  big_app1(pp);
  big_app(big_force);
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −1, 41);
}
```

This code is used in section 100.

119. ⟨ Cases for *base* 119 ⟩ ≡

```
if (cat1 ≡ int_like ∨ cat1 ≡ exp) {
  if (cat2 ≡ comma) {
    big_app1(pp);
    big_app('␣');
    big_app2(pp + 1);
    app(opt);
    app('9');
    reduce(pp, 3, base, 0, 42);
  }
  else if (cat2 ≡ lbrace) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    big_app('␣');
    big_app1(pp + 2);
    reduce(pp, 3, lbrace, −2, 43);
  }
}
```

This code is used in section 100.

120. ⟨ Cases for *struct_like* 120 ⟩ ≡

```
if (cat1 ≡ lbrace) {
   big_app1(pp);
   big_app('␣');
   big_app1(pp + 1);
   reduce(pp, 2, struct_head, 0, 44);
}
else if (cat1 ≡ exp ∨ cat1 ≡ int_like) {
   if (cat2 ≡ lbrace ∨ cat2 ≡ semi) {
      make_underlined(pp + 1);
      make_reserved(pp + 1);
      big_app1(pp);
      big_app('␣');
      big_app1(pp + 1);
      if (cat2 ≡ semi) reduce(pp, 2, decl_head, 0, 45);
      else {
         big_app('␣');
         big_app1(pp + 2);
         reduce(pp, 3, struct_head, 0, 46);
      }
   }
   else if (cat2 ≡ colon) squash(pp + 2, 1, base, 2, 47);
   else if (cat2 ≠ base) {
      big_app1(pp);
      big_app('␣');
      big_app1(pp + 1);
      reduce(pp, 2, int_like, −2, 48);
   }
}
```

This code is used in section 100.

121. ⟨ Cases for *struct_head* 121 ⟩ ≡

```
if ((cat1 ≡ decl ∨ cat1 ≡ stmt ∨ cat1 ≡ function) ∧ cat2 ≡ rbrace) {
   big_app1(pp);
   big_app(indent);
   big_app(force);
   big_app1(pp + 1);
   big_app(outdent);
   big_app(force);
   big_app1(pp + 2);
   reduce(pp, 3, int_like, −2, 49);
}
else if (cat1 ≡ rbrace) {
   big_app1(pp);
   app_str("\\,");
   big_app1(pp + 1);
   reduce(pp, 2, int_like, −2, 50);
}
```

This code is used in section 100.

122. ⟨ Cases for *fn_decl* 122 ⟩ ≡

```
if (cat1 ≡ decl) {
    big_app1(pp);
    big_app(force);
    big_app1(pp + 1);
    reduce(pp, 2, fn_decl, 0, 51);
}
else if (cat1 ≡ stmt) {
    big_app1(pp);
    app(outdent);
    app(outdent);
    big_app(force);
    big_app1(pp + 1);
    reduce(pp, 2, function, −1, 52);
}
```

This code is used in section 100.

123. ⟨ Cases for *function* 123 ⟩ ≡

```
if (cat1 ≡ function ∨ cat1 ≡ decl ∨ cat1 ≡ stmt) {
    big_app1(pp);
    big_app(big_force);
    big_app1(pp + 1);
    reduce(pp, 2, cat1, −1, 53);
}
```

This code is used in section 100.

124. ⟨ Cases for *lbrace* 124 ⟩ ≡

```
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 100.

125. ⟨ Cases for *if_like* 125 ⟩ ≡

```
if (cat1 ≡ exp) {
   big_app1(pp);
   big_app('␣');
   big_app1(pp + 1);
   reduce(pp, 2, if_clause, 0, 57);
}
```

This code is used in section 100.

126. ⟨ Cases for *else_like* 126 ⟩ ≡

```
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 100.

127. ⟨ Cases for *else_head* 127 ⟩ ≡

```
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 100.

128. ⟨ Cases for *if_clause* 128 ⟩ ≡

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

This code is used in section 100.

129. ⟨ Cases for *if_head* 129 ⟩ ≡

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

This code is used in section 100.

130. ⟨ Cases for *do_like* 130 ⟩ ≡

```
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 100.

131. ⟨ Cases for *case_like* 131 ⟩ ≡

```
if (cat1 ≡ semi) squash(pp, 2, stmt, −1, 70);
else if (cat1 ≡ colon) squash(pp, 2, tag, −1, 71);
else if (cat1 ≡ exp) {
   big_app1(pp);
   big_app('␣');
   big_app1(pp + 1);
   reduce(pp, 2, exp, −2, 72);
}
```

This code is used in section 100.

132. ⟨ Cases for *catch_like* 132 ⟩ ≡

```
if (cat1 ≡ cast ∨ cat1 ≡ exp) {
   big_app2(pp);
   big_app(indent);
   big_app(indent);
   reduce(pp, 2, fn_decl, 0, 73);
}
```

This code is used in section 100.

133. ⟨ Cases for *tag* 133 ⟩ ≡

```
if (cat1 ≡ tag) {
   big_app1(pp);
   big_app(break_space);
   big_app1(pp + 1);
   reduce(pp, 2, tag, −1, 74);
}
else if (cat1 ≡ stmt ∨ cat1 ≡ decl ∨ cat1 ≡ function) {
   big_app(force);
   big_app(backup);
   big_app1(pp);
   big_app(break_space);
   big_app1(pp + 1);
   reduce(pp, 2, cat1, −1, 75);
}
```

This code is used in section 100.

134. 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? */
⟨ Cases for stmt 134 ⟩ ≡
  if (cat1 ≡ stmt ∨ cat1 ≡ decl ∨ cat1 ≡ function) {
    big_app1(pp);
    if (cat1 ≡ function) big_app(big_force);
    else if (cat1 ≡ decl) big_app(big_force);
    else if (force_lines) big_app(force);
    else big_app(break_space);
    big_app1(pp + 1);
    reduce(pp, 2, cat1, −1, 76);
  }
```

This code is used in section 100.

135. ⟨ Cases for *semi* 135 ⟩ ≡

```
  big_app('␣');
  big_app1(pp);
  reduce(pp, 1, stmt, −1, 77);
```

This code is used in section 100.

136. ⟨ Cases for *lproc* 136 ⟩ ≡

```
  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(pp);
      big_app('␣');
      big_app1(pp + 1);
      app_str("␣\\5");
      big_app2(pp + 2);
      reduce(pp, 4, insert, −1, 80);
    }
  }
```

This code is used in section 100.

137. ⟨ Cases for *section_scrap* 137 ⟩ ≡

```
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 100.

138. ⟨ Cases for *insert* 138 ⟩ ≡

```
if (cat1) squash(pp, 2, cat1, 0, 83);
```

This code is used in section 100.

139. ⟨ Cases for *prelangle* 139 ⟩ ≡

```
init_mathness = cur_mathness = yes_math;
app('<');
reduce(pp, 1, binop, −2, 84);
```

This code is used in section 100.

140. ⟨ Cases for *prerangle* 140 ⟩ ≡

```
init_mathness = cur_mathness = yes_math;
app('>');
reduce(pp, 1, binop, −2, 85);
```

This code is used in section 100.

141. ⟨ Cases for *langle* 141 ⟩ ≡

```
if (cat1 ≡ prerangle) {
   big_app1(pp);
   app('\\');
   app(',');
   big_app1(pp + 1);
   reduce(pp, 2, cast, −1, 86);
}
else if (cat1 ≡ decl_head ∨ cat1 ≡ int_like ∨ cat1 ≡ exp) {
   if (cat2 ≡ prerangle) squash(pp, 3, cast, −1, 87);
   else if (cat2 ≡ comma) {
      big_app3(pp);
      app(opt);
      app('9');
      reduce(pp, 3, langle, 0, 88);
   }
}
```

This code is used in section 100.

142. ⟨ Cases for *template_like* 142 ⟩ ≡

```
if ($cat1 \equiv exp \land cat2 \equiv prelangle$) squash(pp + 2, 1, langle, 2, 89);
else if ($cat1 \equiv exp \lor cat1 \equiv raw\_int$) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, cat1, −2, 90);
} else squash(pp, 1, raw_int, 0, 91);
```

This code is used in section 100.

143. ⟨ Cases for *new_like* 143 ⟩ ≡

```
if ($cat1 \equiv lpar \land cat2 \equiv exp \land cat3 \equiv rpar$) squash(pp, 4, new_like, 0, 92);
else if ($cat1 \equiv cast$) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, exp, −2, 93);
}
else if ($cat1 \neq lpar$) squash(pp, 1, new_exp, 0, 94);
```

This code is used in section 100.

144. ⟨ Cases for *new_exp* 144 ⟩ ≡

```
if ($cat1 \equiv int\_like \lor cat1 \equiv const\_like$) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, new_exp, 0, 95);
}
else if ($cat1 \equiv struct\_like \land (cat2 \equiv exp \lor cat2 \equiv int\_like)$) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    big_app('␣');
    big_app1(pp + 2);
    reduce(pp, 3, new_exp, 0, 96);
}
else if ($cat1 \equiv raw\_ubin$) {
    big_app1(pp);
    big_app('{');
    big_app1(pp + 1);
    big_app('}');
    reduce(pp, 2, new_exp, 0, 97);
}
else if ($cat1 \equiv lpar$) squash(pp, 1, exp, −2, 98);
else if ($cat1 \equiv exp$) {
    big_app1(pp);
    big_app('␣');
    reduce(pp, 1, exp, −2, 98);
}
else if ($cat1 \neq raw\_int \land cat1 \neq struct\_like \land cat1 \neq colcol$) squash(pp, 1, exp, −2, 99);
```

This code is used in section 100.

145. ⟨ Cases for *ftemplate* 145 ⟩ ≡

```
if (cat1 ≡ prelangle) squash(pp + 1, 1, langle, 1, 100);
else squash(pp, 1, exp, −2, 101);
```

This code is used in section 100.

146. ⟨ Cases for *for_like* 146 ⟩ ≡

```
if (cat1 ≡ exp) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, else_like, −2, 102);
}
```

This code is used in section 100.

147. ⟨ Cases for *raw_ubin* 147 ⟩ ≡

```
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 100.

148. ⟨ Cases for *const_like* 148 ⟩ ≡

```
squash(pp, 1, int_like, −2, 105);
```

This code is used in section 100.

149. ⟨ Cases for *raw_int* 149 ⟩ ≡

```
if (cat1 ≡ prelangle) squash(pp + 1, 1, langle, 1, 106);
else if (cat1 ≡ colcol) squash(pp, 2, colcol, −1, 107);
else if (cat1 ≡ cast) squash(pp, 2, raw_int, 0, 108);
else if (cat1 ≡ lpar) squash(pp, 1, exp, −2, 109);
else if (cat1 ≠ langle) squash(pp, 1, int_like, −3, 110);
```

This code is used in section 100.

150. ⟨ Cases for *operator_like* 150 ⟩ ≡

```
if (cat1 ≡ binop ∨ cat1 ≡ unop ∨ cat1 ≡ ubinop) {
  if (cat2 ≡ binop) break;
  big_app1(pp);
  big_app('{');
  big_app1(pp + 1);
  big_app('}');
  reduce(pp, 2, exp, −2, 111);
}
else if (cat1 ≡ new_like ∨ cat1 ≡ delete_like) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, exp, −2, 112);
}
else if (cat1 ≡ comma) squash(pp, 2, exp, −2, 113);
else if (cat1 ≠ raw_ubin) squash(pp, 1, new_exp, 0, 114);
```

This code is used in section 100.

151. ⟨ Cases for *typedef_like* 151 ⟩ ≡

```
if ((cat1 ≡ int_like ∨ cat1 ≡ cast) ∧ (cat2 ≡ comma ∨ cat2 ≡ semi)) squash(pp + 1, 1, exp, −1, 115);
else if (cat1 ≡ int_like) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  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(pp);
  big_app('␣');
  big_app1(pp + 1);
  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(pp + 1);
  big_app('}');
  big_app1(pp + 2);
  reduce(pp + 1, 2, cat2, 0, 120);
}
```

This code is used in section 100.

152. ⟨ Cases for *delete_like* 152 ⟩ ≡

```
if (cat1 ≡ lpar ∧ cat2 ≡ rpar) {
  big_app2(pp);
  app('\\');
  app(',');
  big_app1(pp + 2);
  reduce(pp, 3, delete_like, 0, 121);
}
else if (cat1 ≡ exp) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, exp, −2, 122);
}
```

This code is used in section 100.

153. ⟨ Cases for *question* 153 ⟩ ≡

```
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 100.

154. 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
  void reduce(j, k, c, d, n)
      scrap_pointer j;
  eight_bitsc;
  short k, d, 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 159⟩;
    pp−−;      /* we next say pp++ */
  }
```

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

```
  void squash(j, k, c, d, n)
      scrap_pointer j;
  eight_bitsc;
  short k, d, 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 159⟩;
      pp−−;      /* we next say pp++ */
      return;
    }
    for (i = j; i < j + k; i++) big_app1(i);
    reduce(j, k, c, d, n);
  }
```

156. 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 156⟩ ≡
  while (1) {
    ⟨Make sure the entries pp through pp + 3 of cat are defined 157⟩;
    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 100⟩;
  }
```

This code is used in section 160.

157. 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 157⟩ ≡
  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 156.

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

```
⟨Global variables 7⟩ +≡
  int tracing;    /* can be used to show parsing details */
```

159. ⟨ Print a snapshot of the scrap list if debugging 159 ⟩ ≡

```
{
  scrap_pointer k;      /* pointer into scrap_info */
  if (tracing ≡ 2) {
    printf("\n%d:", n);
    for (k = scrap_base; k ≤ lo_ptr; k++) {
      if (k ≡ pp) putxchar('*');
      else putxchar('␣');
      if (k→mathness % 4 ≡ yes_math) putchar('+');
      else if (k→mathness % 4 ≡ no_math) putchar('-');
      print_cat(k→cat);
      if (k→mathness/4 ≡ yes_math) putchar('+');
      else if (k→mathness/4 ≡ no_math) putchar('-');
    }
    if (hi_ptr ≤ scrap_ptr) printf("...");      /* indicate that more is coming */
  }
}
```

This code is used in sections 154 and 155.

160. 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 $text\_ptr + 3 \le max\_texts$ and $tok\_ptr + 6 \le max\_toks$, 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.

```
text_pointer translate()      /* converts a sequence of scraps */
{
  scrap_pointer i,      /* index into cat */
  j;      /* runs through final scraps */
  pp = scrap_base;
  lo_ptr = pp - 1;
  hi_ptr = pp;
  ⟨ If tracing, print an indication of where we are 163 ⟩;
  ⟨ Reduce the scraps using the productions until no more rules apply 156 ⟩;
  ⟨ Combine the irreducible scraps that remain 161 ⟩;
}
```

161. 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 161 ⟩ ≡
  {
    ⟨ If semi-tracing, show the irreducible scraps 162 ⟩;
    for (j = scrap_base; j ≤ lo_ptr; j++) {
      if (j ≠ scrap_base) app('␣');
      if (j→mathness % 4 ≡ yes_math) app('$');
      app1(j);
      if (j→mathness/4 ≡ yes_math) app('$');
      if (tok_ptr + 6 > tok_mem_end) overflow("token");
    }
    freeze_text;
    return (text_ptr − 1);
  }
```

This code is used in section 160.

162. ⟨ If semi-tracing, show the irreducible scraps 162 ⟩ ≡

```
  if (lo_ptr > scrap_base ∧ tracing ≡ 1) {
    printf("\nIrreducible␣scrap␣sequence␣in␣section␣%d:", section_count);
    mark_harmless;
    for (j = scrap_base; j ≤ lo_ptr; j++) {
      printf("␣");
      print_cat(j→cat);
    }
  }
```

This code is used in section 161.

163. ⟨ If tracing, print an indication of where we are 163 ⟩ ≡

```
  if (tracing ≡ 2) {
    printf("\nTracing␣after␣l.␣%d:\n", cur_line);
    mark_harmless;
    if (loc > buffer + 50) {
      printf("...");
      term_write(loc − 51, 51);
    }
    else term_write(buffer, loc − buffer);
  }
```

This code is used in section 160.

164. 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.

```
void C_parse(spec_ctrl)      /* creates scraps from C tokens */
eight_bits spec_ctrl;
{
  int count;      /* characters remaining before string break */
  while (next_control < format_code ∨ next_control ≡ spec_ctrl) {
    ⟨Append the scrap appropriate to next_control 166⟩;
    next_control = get_next( );
    if (next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡ begin_short_comment)
      return;
  }
}
```

165. 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;
        }
```

166. ⟨ Append the scrap appropriate to *next_control* 166 ⟩ ≡

```
⟨ Make sure that there is room for the new scraps, tokens, and texts 167 ⟩;
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 169 ⟩; break;
case identifier: app_cur_id(1); break;
case TeX_string: ⟨ Append a TeX string, without forming a scrap 170 ⟩; 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 '(': case '[': app(next_control);
  app_scrap(lpar, maybe_math); break;
case ')': case ']': app(next_control);
  app_scrap(rpar, 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 168 ⟩
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 164.

167. ⟨Make sure that there is room for the new scraps, tokens, and texts 167⟩ ≡

```
  if (scrap_ptr + safe_scrap_incr > scrap_info_end ∨ tok_ptr + safe_tok_incr > tok_mem_end
          ∨ text_ptr + safe_text_incr > tok_start_end) {
    if (scrap_ptr > max_scr_ptr) max_scr_ptr = scrap_ptr;
    if (tok_ptr > max_tok_ptr) max_tok_ptr = tok_ptr;
    if (text_ptr > max_text_ptr) max_text_ptr = text_ptr;
    overflow("scrap/token/text");
  }
```

This code is used in sections 166 and 174.

168. 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 168 ⟩ ≡

```
case not_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 166.

169. The following code must use *app_tok* instead of *app* in order to protect against overflow. Note that $\mathit{tok\_ptr} + 1 \le \mathit{max\_toks}$ 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 169 ⟩ ≡
  count = −1;
  if (next_control ≡ constant) app_str("\\T{");
  else if (next_control ≡ string) {
    count = 20;
    app_str("\\.{");
  }
  else app_str("\\vb{");
  while (id_first < id_loc) {
    if (count ≡ 0) {      /∗ insert a discretionary break in a long string ∗/
      app_str("}\\)\\.{");
      count = 20;
    }
    if ((eight_bits)(∗id_first) > °177) {
      app_tok(quoted_char);
      app_tok((eight_bits)(∗id_first++));
    }
    else {
      switch (∗id_first) {
      case '␣': case '\\': case '#': case '%': case '$': case '^': case '{': case '}': case '~':
        case '&': case '_': app('\\');
        break;
      case '@':
        if (∗(id_first + 1) ≡ '@') id_first++;
        else err_print("!␣Double␣@␣should␣be␣used␣in␣strings");
      }
      app_tok(∗id_first++);
    }
    count−−;
  }
  app('}');
  app_scrap(exp, maybe_math);
```

This code is used in section 166.

170. 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 170 ⟩ ≡

```
  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 166.

171. The function *app_cur_id* appends the current identifier to the token list; it also builds a new scrap if *scrapping* ≡ 1.

⟨ Predeclaration of procedures 2 ⟩ +≡

```
  void app_cur_id();
```

172.

```
void app_cur_id(scrapping)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);
      }
    }
  }
```

173. 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.

```
text_pointer C_translate()
{
  text_pointer p;    /* points to the translation */
  scrap_pointer save_base;    /* holds original value of scrap_base */
  save_base = scrap_base;
  scrap_base = scrap_ptr + 1;
  C_parse(section_name);    /* get the scraps together */
  if (next_control ≠ '|') err_print("!␣Missing␣'|'␣after␣C␣text");
  app_tok(cancel);
  app_scrap(insert, maybe_math);    /* place a cancel token as a final "comment" */
  p = translate();    /* make the translation */
  if (scrap_ptr > max_scr_ptr) max_scr_ptr = scrap_ptr;
  scrap_ptr = scrap_base − 1;
  scrap_base = save_base;    /* scrap the scraps */
  return (p);
}
```

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

The token list created from within '`|` ... `|`' brackets is output as an argument to `\PB`, if the user has invoked `CWEAVE` with the `+e` flag. 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']
  void outer_parse( )      /* 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 {
        booleanis_long_comment = (next_control ≡ begin_comment);
        ⟨ Make sure that there is room for the new scraps, tokens, and texts 167 ⟩;
        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 */
      }
  }
```

175. 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`'.

176. 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 8 ⟩ +≡
  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;
```

177.

```
#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 */
⟨ Global variables 7 ⟩ +≡
  output_state cur_state;    /* cur_end, cur_tok, cur_mode */
  output_state stack[stack_size];    /* info for non-current levels */
  stack_pointer stack_ptr;    /* first unused location in the output state stack */
  stack_pointer stack_end = stack + stack_size − 1;    /* end of stack */
  stack_pointer max_stack_ptr;    /* largest value assumed by stack_ptr */
```

178. ⟨ Set initial values 10 ⟩ +≡

```
  max_stack_ptr = stack;
```

179. 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.

```
void push_level(p)      /∗ 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);
}
```

180. 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.

```
void pop_level( )
{
  cur_end = (−− stack_ptr)→end_field;
  cur_tok = stack_ptr→tok_field;
  cur_mode = stack_ptr→mode_field;
}
```

181. 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.

```
⟨ Global variables 7 ⟩ +≡
  name_pointer cur_name;
```

```
182. #define res_word °201    /* returned by get_output for reserved words */
#define section_code °200    /* returned by get_output for section names */
  eight_bits get_output( )    /* 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 (a);
  }
```

183. 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.

```
void output_C()    /* outputs the current token list */
{
  token_pointer save_tok_ptr;
  text_pointer save_text_ptr;
  sixteen_bits save_next_control;    /* values to be restored */
  text_pointer p;    /* translation of the C text */
  save_tok_ptr = tok_ptr;
  save_text_ptr = text_ptr;
  save_next_control = next_control;
  next_control = ignore;
  p = C_translate();
  app(inner_tok_flag + (int)(p − tok_start));
  if (make_pb) {
    out_str("\\PB{");
    make_output();
    out('}');
  } else make_output();    /* output the list */
  if (text_ptr > max_text_ptr) max_text_ptr = text_ptr;
  if (tok_ptr > max_tok_ptr) max_tok_ptr = tok_ptr;
  text_ptr = save_text_ptr;
  tok_ptr = save_tok_ptr;    /* forget the tokens */
  next_control = save_next_control;    /* restore next_control to original state */
}
```

184. Here is CWEAVE's major output handler.

```
⟨Predeclaration of procedures 2⟩ +≡
  void make_output();
```

185.

```
void make_output( )    /* outputs the equivalents of tokens */
{
  eight_bits a,    /* current output byte */
  b;    /* next output byte */
  int c;    /* count of indent and outdent tokens */
  char scratch[longest_name];    /* scratch area for section names */
  char *k, *k_limit;    /* indices into scratch */
  char *j;    /* index into buffer */
  char *p;    /* index into byte_mem */
  char delim;    /* first and last character of string being copied */
  char *save_loc, *save_limit;    /* loc and limit to be restored */
  name_pointer cur_section_name;    /* name of section being output */
  boolean save_mode;    /* value of cur_mode before a sequence of breaks */
  app(end_translation);    /* append a sentinel */
  freeze_text;
  push_level(text_ptr − 1);
  while (1) {
    a = get_output( );
  reswitch:
    switch (a) {
    case end_translation: return;
    case identifier: case res_word: ⟨Output an identifier 186⟩;
      break;
    case section_code: ⟨Output a section name 190⟩;
      break;
    case math_rel: out_str("\\MRL{");
    case noop: case inserted: break;
    case cancel: case big_cancel: c = 0;
      b = a;
      while (1) {
        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 189⟩;
      goto reswitch;
    case indent: case outdent: case opt: case backup: case break_space: case force: case big_force:
      case preproc_line:
      ⟨Output a control, look ahead in case of line breaks, possibly goto reswitch 187⟩;
      break;
    case quoted_char: out(*(cur_tok++));
    case qualifier: break;
    default: out(a);    /* otherwise a is an ordinary character */
    }
  }
}
```

186. 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 186 ⟩ ≡

```
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, 1);
```

This code is used in section 185.

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

⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 187 ⟩ ≡

```
if (a < break_space ∨ a ≡ preproc_line) {
  if (cur_mode ≡ outer) {
    out('\\');
    out(a − cancel + '0');
    if (a ≡ opt) {
      b = get_output();    /* opt is followed by a digit */
      if (b ≠ '0' ∨ force_lines ≡ 0) out(b)
      else out_str("{-1}");    /* force_lines encourages more @| breaks */
    }
  }
  else if (a ≡ opt) b = get_output();    /* ignore digit following opt */
}
else ⟨ Look ahead for strongest line break, goto reswitch 188 ⟩
```

This code is used in section 185.

188. 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 188 ⟩ ≡
  {
    b = a;
    save_mode = cur_mode;
    c = 0;
    while (1) {
      a = get_output();
      if (a ≡ inserted) continue;
      if (a ≡ cancel ∨ a ≡ big_cancel) {
        ⟨ Output saved indent or outdent tokens 189 ⟩;
        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 189 ⟩;
          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 187.

189. ⟨ Output saved *indent* or *outdent* tokens 189 ⟩ ≡

```
  for ( ; c > 0; c−−) out_str("\\1");
  for ( ; c < 0; c++) out_str("\\2");
```

This code is used in sections 185 and 188.

190. 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 190 ⟩ ≡
  {
    out_str("\\X");
    cur_xref = (xref_pointer) cur_name→xref;
    if (cur_xref→num ≡ file_flag) {
      an_output = 1;
      cur_xref = cur_xref→xlink;
    }
    else an_output = 0;
    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 191 ⟩;
    if (an_output) out_str("␣}");
    out_str("\\X");
  }
```

This code is used in section 185.

191. ⟨ Output the text of the section name 191 ⟩ ≡

```
sprint_section_name(scratch, cur_name);
k = scratch;
k_limit = scratch + strlen(scratch);
cur_section_name = cur_name; while (k < k_limit) { b = *(k++);
if (b ≡ '@') ⟨ Skip next character, give error if not '@' 192 ⟩;
if (an_output)
  switch (b) {
  case '␣': case '\\': case '#': case '%': case '$': case '^': case '{': case '}': case '~':
    case '&': case '_': out('\\');     /* falls through */
  default: out(b);
  }
else if (b ≠ '|') out(b)
else {
  ⟨ Copy the C text into the buffer array 193 ⟩;
  save_loc = loc;
  save_limit = limit;
  loc = limit + 2;
  limit = j + 1;
  *limit = '|';
  output_C( );
  loc = save_loc;
  limit = save_limit;
}
}
```

This code is used in section 190.

192. ⟨ Skip next character, give error if not '@' 192 ⟩ ≡

```
if (*k++ ≠ '@') {
  printf("\n!␣Illegal␣control␣code␣in␣section␣name:␣<");
  print_section_name(cur_section_name);
  printf(">␣");
  mark_error;
}
```

This code is used in section 191.

193. 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 193 ⟩ ≡
  j = limit + 1;
  *j = '|';
  delim = 0; while (1) {
  if (k ≥ k_limit) {
    printf("\n!␣C␣text␣in␣section␣name␣didn't␣end:␣<");
    print_section_name(cur_section_name);
    printf(">␣");
    mark_error;
    break;
  }
  b = *(k++); if (b ≡ '@' ∨ (b ≡ '\\' ∧ delim ≠ 0)) ⟨ Copy a quoted character into the buffer 194 ⟩
  else {
    if (b ≡ '\'' ∨ b ≡ '"')
      if (delim ≡ 0) delim = b;
      else if (delim ≡ b) delim = 0;
    if (b ≠ '|' ∨ delim ≠ 0) {
      if (j > buffer + long_buf_size − 3) overflow("buffer");
      *(++j) = b;
    }
    else break;
  }
  }
```

This code is used in section 191.

194. ⟨ Copy a quoted character into the buffer 194 ⟩ ≡

```
  {
    if (j > buffer + long_buf_size − 4) overflow("buffer");
    *(++j) = b;
    *(++j) = *(k++);
  }
```

This code is used in section 193.

195. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_two( );
```

196. **void** *phase_two*( )

```
  {
    reset_input( );
    if (show_progress) printf("\nWriting␣the␣output␣file...");
    section_count = 0;
    format_visible = 1;
    copy_limbo( );
    finish_line( );
    flush_buffer(out_buf, 0, 0);     /* insert a blank line, it looks nice */
    while (¬input_has_ended) ⟨ Translate the current section 198 ⟩;
  }
```

197. 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 = 1

⟨ Global variables 7 ⟩ +≡
  int save_line;     /* former value of out_line */
  char *save_place;     /* former value of out_ptr */
  int sec_depth;     /* the integer, if any, following @* */
  boolean space_checked;     /* have we done emit_space_if_needed? */
  boolean format_visible;     /* should the next format declaration be output? */
  boolean doing_format = 0;     /* are we outputting a format declaration? */
  boolean group_found = 0;     /* has a starred section occurred? */
```

198. ⟨ Translate the current section 198 ⟩ ≡

```
  {
    section_count++;
    ⟨ Output the code for the beginning of a new section 199 ⟩;
    save_position;
    ⟨ Translate the TEX part of the current section 200 ⟩;
    ⟨ Translate the definition part of the current section 201 ⟩;
    ⟨ Translate the C part of the current section 207 ⟩;
    ⟨ Show cross-references to this section 210 ⟩;
    ⟨ Output the code for the end of a section 214 ⟩;
  }
```

This code is used in section 196.

199. 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 199 ⟩ ≡

```
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 = 1;
  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_str("{");
out_section(section_count);
out_str("}");
```

This code is used in section 198.

200. 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 200 ⟩ ≡

```
do {
  next_control = copy_TEX();
  switch (next_control) {
  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 198.

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

```
⟨ Translate the definition part of the current section 201 ⟩ ≡
  space_checked = 0;
  while (next_control ≤ definition) {     /* format_code or definition */
    init_stack;
    if (next_control ≡ definition) ⟨ Start a macro definition 204 ⟩
    else ⟨ Start a format definition 205 ⟩;
    outer_parse( );
    finish_C(format_visible);
    format_visible = 1;
    doing_format = 0;
  }
```

This code is used in section 198.

202. 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`.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void finish_C( );
```

203.

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

204. 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 204⟩ ≡
  {
    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('$');
      app_cur_id(0);
      if (*loc ≡ '(')
      reswitch:
        switch (next_control = get_next()) {
        case '(': case ',': app(next_control);
          goto reswitch;
        case identifier: app_cur_id(0);
          goto reswitch;
        case ')': app(next_control);
          next_control = get_next();
          break;
        default: err_print("!␣Improper␣macro␣definition");
          break;
        }
      else  next_control = get_next();
      app_str("$␣");
      app(break_space);
      app_scrap(dead, no_math);      /* scrap won't take part in the parsing */
    }
  }
```

This code is used in section 201.

205. ⟨ Start a format definition 205 ⟩ ≡

```
{
  doing_format = 1;
  if (*(loc − 1) ≡ 's' ∨ *(loc − 1) ≡ 'S') format_visible = 0;
  if (¬space_checked) {
    emit_space_if_needed;
    save_position;
  }
  app_str("\\F");    /* this will produce 'format ' */
  next_control = get_next();
  if (next_control ≡ identifier) {
    app(id_flag + (int)(id_lookup(id_first, id_loc, normal) − name_dir));
    app('␣');
    app(break_space);    /* this is syntactically separate from what follows */
    next_control = get_next();
    if (next_control ≡ identifier) {
      app(id_flag + (int)(id_lookup(id_first, id_loc, normal) − name_dir));
      app_scrap(exp, maybe_math);
      app_scrap(semi, maybe_math);
      next_control = get_next();
    }
  }
  if (scrap_ptr ≠ scrap_info + 2) err_print("!␣Improper␣format␣definition");
}
```

This code is used in section 201.

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

⟨ Global variables 7 ⟩ +≡

```
  name_pointer this_section;    /* the current section name, or zero */
```

207. ⟨ Translate the C part of the current section 207 ⟩ ≡

```
  this_section = name_dir;
  if (next_control ≤ section_name) {
    emit_space_if_needed;
    init_stack;
    if (next_control ≡ begin_C) next_control = get_next();
    else {
      this_section = cur_section;
      ⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section
          definition 208 ⟩;
    }
    while (next_control ≤ section_name) {
      outer_parse();
      ⟨ Emit the scrap for a section name if present 209 ⟩;
    }
    finish_C(1);
  }
```

This code is used in section 198.

208. The title of the section and an ≡ or +≡ are made into a scrap that should not take part in the parsing.

⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 208 ⟩ ≡

```
do next_control = get_next( ); while (next_control ≡ '+');    /* allow optional '+=' */
if (next_control ≠ '=' ∧ next_control ≠ eq_eq)
  err_print("!␣You␣need␣an␣=␣sign␣after␣the␣section␣name");
else next_control = get_next( );
if (out_ptr > out_buf + 1 ∧ *out_ptr ≡ 'Y' ∧ *(out_ptr − 1) ≡ '\\') app(backup);
    /* the section name will be flush left */
app(section_flag + (int)(this_section − name_dir));
cur_xref = (xref_pointer) this_section→xref;
if (cur_xref→num ≡ file_flag) cur_xref = cur_xref→xlink;
app_str("${}");
if (cur_xref→num ≠ section_count + def_flag) {
  app_str("\\mathrel+");    /* section name is multiply defined */
  this_section = name_dir;    /* so we won't give cross-reference info here */
}
app_str("\\E");    /* output an equivalence sign */
app_str("{}$");
app(force);
app_scrap(dead, no_math);    /* this forces a line break unless '@+' follows */
```

This code is used in section 207.

209. ⟨ Emit the scrap for a section name if present 209 ⟩ ≡

```
if (next_control < section_name) {
  err_print("!␣You␣can't␣do␣that␣in␣C␣text");
  next_control = get_next( );
}
else if (next_control ≡ section_name) {
  app(section_flag + (int)(cur_section − name_dir));
  app_scrap(section_scrap, maybe_math);
  next_control = get_next( );
}
```

This code is used in section 207.

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

⟨ Show cross-references to this section 210 ⟩ ≡

```
if (this_section > name_dir) {
  cur_xref = (xref_pointer) this_section→xref;
  if (cur_xref→num ≡ file_flag) {
    an_output = 1;
    cur_xref = cur_xref→xlink;
  }
  else an_output = 0;
  if (cur_xref→num > def_flag) cur_xref = cur_xref→xlink;    /* bypass current section number */
  footnote(def_flag);
  footnote(cite_flag);
  footnote(0);
}
```

This code is used in section 198.

211. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void footnote( );
```

```
212.  void footnote(flag)      /∗ outputs section cross-references ∗/
        sixteen_bits flag;
  {
    xref_pointer q;      /∗ cross-reference pointer variable ∗/
    if (cur_xref→num ≤ flag) return;
    finish_line( );
    out('\\');
    out(flag ≡ 0 ? 'U' : flag ≡ cite_flag ? 'Q' : 'A');
    ⟨ Output all the section numbers on the reference list cur_xref 213 ⟩;
    out('.');
  }
```

213. 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 213 ⟩ ≡
  q = cur_xref;
  if (q→xlink→num > flag) out('s');      /∗ plural ∗/
  while (1) {
    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 212.

```
214.  ⟨ Output the code for the end of a section 214 ⟩ ≡
  out_str("\\fi");
  finish_line( );
  flush_buffer(out_buf, 0, 0);      /∗ insert a blank line, it looks nice ∗/
```

This code is used in section 198.

215. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_three( );
```

216. **void** *phase_three*( )

```
  {
    if (no_xref) {
      finish_line( );
      out_str("\\end");
      finish_line( );
    }
    else {
      phase = 3;
      if (show_progress) printf("\nWriting␣the␣index...");
      finish_line( );
      if ((idx_file = fopen(idx_file_name, "w")) ≡ Λ)
        fatal("!␣Cannot␣open␣index␣file␣", idx_file_name);
      if (change_exists) {
        ⟨ Tell about changed sections 218 ⟩;
        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 220 ⟩;
      ⟨ Sort and output the index 229 ⟩;
      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, "w")) ≡ 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 238 ⟩;
      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) printf("\nDone.");
    check_complete( );   /* was all of the change file used? */
  }
```

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

⟨ Global variables 7 ⟩ +≡

```
  sixteen_bits k_section;    /* runs through the sections */
```

218. ⟨ Tell about changed sections 218 ⟩ ≡

```
  {     /* 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 216.

219. 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 102 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.

⟨ Global variables 7 ⟩ +≡

```
  name_pointer bucket[256];
  name_pointer next_name;    /* successor of cur_name when sorting */
  name_pointer blink[max_names];    /* links in the buckets */
```

220. 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 220 ⟩ ≡

```
  {
    int c;
    for (c = 0; c ≤ 255; 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 ≠ (char *) 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 216.

221. 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*$[j] \equiv k$.

222. ⟨ Rest of *trans_plus* union 222 ⟩ ≡

```
  name_pointer Head;
```

This code is used in section 93.

223.

```
#define depth cat   /* reclaims memory that is no longer needed for parsing */
#define head trans_plus.Head   /* ditto */
  format sort_pointer int
#define sort_pointer scrap_pointer   /* ditto */
#define sort_ptr scrap_ptr   /* ditto */
#define max_sorts max_scraps   /* ditto */
```

⟨ Global variables 7 ⟩ +≡

```
  eight_bits cur_depth;   /* depth of current buckets */
  char *cur_byte;   /* index into byte_mem */
  sixteen_bits cur_val;   /* current cross-reference number */
  sort_pointer max_sort_ptr;   /* largest value of sort_ptr */
```

224. ⟨ Set initial values 10 ⟩ +≡

```
  max_sort_ptr = scrap_info;
```

225. The desired alphabetic order is specified by the *collate* array; namely, *collate*$[0] <$ *collate*$[1] < \cdots <$ *collate*$[100]$.

⟨ Global variables 7 ⟩ +≡

```
  eight_bits collate[102 + 128];   /* collation order */
```

226. We use the order null < ␣ < other characters < `_` < `A` = `a` < ··· < `Z` = `z` < `0` < ··· < `9`. Warning: The collation mapping needs to be changed if ASCII code is not being used.

We initialize *collate* by copying a few characters at a time, because some C compilers choke on long strings.

```
⟨Set initial values 10⟩ +≡
  collate[0] = 0;
  strcpy(collate + 1, "␣\1\2\3\4\5\6\7\10\11\12\13\14\15\16\17");      /* 16 characters + 1 = 17 */
  strcpy(collate + 17, "\20\21\22\23\24\25\26\27\30\31\32\33\34\35\36\37");
    /* 16 characters + 17 = 33 */
  strcpy(collate + 33, "!\42#$%&'()*+,-./:;<=>?@[\\]^`{|}~_");      /* 32 characters + 33 = 65 */
  strcpy(collate + 65, "abcdefghijklmnopqrstuvwxyz0123456789");
    /* (26 + 10) characters + 65 = 101 */
  strcpy(collate + 101, "\200\201\202\203\204\205\206\207\210\211\212\213\214\215\216\217");
    /* 16 characters + 101 = 117 */
  strcpy(collate + 117, "\220\221\222\223\224\225\226\227\230\231\232\233\234\235\236\237");
    /* 16 characters + 117 = 133 */
  strcpy(collate + 133, "\240\241\242\243\244\245\246\247\250\251\252\253\254\255\256\257");
    /* 16 characters + 133 = 149 */
  strcpy(collate + 149, "\260\261\262\263\264\265\266\267\270\271\272\273\274\275\276\277");
    /* 16 characters + 149 = 165 */
  strcpy(collate + 165, "\300\301\302\303\304\305\306\307\310\311\312\313\314\315\316\317");
    /* 16 characters + 165 = 181 */
  strcpy(collate + 181, "\320\321\322\323\324\325\326\327\330\331\332\333\334\335\336\337");
    /* 16 characters + 181 = 197 */
  strcpy(collate + 197, "\340\341\342\343\344\345\346\347\350\351\352\353\354\355\356\357");
    /* 16 characters + 197 = 213 */
  strcpy(collate + 213, "\360\361\362\363\364\365\366\367\370\371\372\373\374\375\376\377");
    /* 16 characters + 213 = 229 */
```

227. 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) */
⟨Predeclaration of procedures 2⟩ +≡
  void unbucket();
```

228.

```
void unbucket(d)      /* 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 = 100 + 128; c ≥ 0; c−−)
      if (bucket[collate[c]]) {
        if (sort_ptr ≥ scrap_info_end) overflow("sorting");
        sort_ptr++;
        if (sort_ptr > max_sort_ptr) max_sort_ptr = sort_ptr;
        if (c ≡ 0) sort_ptr→depth = infinity;
        else sort_ptr→depth = d;
        sort_ptr→head = bucket[collate[c]];
        bucket[collate[c]] = Λ;
      }
  }
```

229. ⟨ Sort and output the index 229 ⟩ ≡

```
sort_ptr = scrap_info;
unbucket(1);
while (sort_ptr > scrap_info) {
  cur_depth = sort_ptr→depth;
  if (blink[sort_ptr→head − name_dir] ≡ 0 ∨ cur_depth ≡ infinity)
    ⟨ Output index entries for the list at sort_ptr 231 ⟩
  else ⟨ Split the list at sort_ptr into further lists 230 ⟩;
}
```

This code is used in section 216.

230. ⟨ Split the list at sort_ptr into further lists 230 ⟩ ≡

```
{
  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 229.

231. ⟨ Output index entries for the list at sort_ptr 231 ⟩ ≡

```
{
  cur_name = sort_ptr→head;
  do {
    out_str("\\I");
    ⟨ Output the name at cur_name 232 ⟩;
    ⟨ Output the cross-references at cur_name 233 ⟩;
    cur_name = blink[cur_name − name_dir];
  } while (cur_name);
  −− sort_ptr;
}
```

This code is used in section 229.

232. ⟨ Output the name at *cur_name* 232 ⟩ ≡

```
  switch (cur_name→ilk) {
  case normal: case func_template:
    if (is_tiny(cur_name)) out_str("\\|");
    else {
      char *j;

      for (j = cur_name→byte_start; j < (cur_name + 1)→byte_start; j++)
        if (xislower(*j)) goto lowcase;
      out_str("\\.");
      break;
    lowcase: out_str("\\\\");
    }
    break;
  case wildcard: out_str("\\9"); goto not_an_identifier;
  case typewriter: out_str("\\.");
  case roman: not_an_identifier: out_name(cur_name, 0);
    goto name_done;
  case custom:
    {
      char *j;

      out_str("$\\");
      for (j = cur_name→byte_start; j < (cur_name + 1)→byte_start; j++)
        out(*j ≡ '_' ? 'x' : *j ≡ '$' ? 'X' : *j);
      out('$');
      goto name_done;
    }
  default: out_str("\\&");
  }
  out_name(cur_name, 1);
name_done:
```

This code is used in section 231.

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

⟨ Output the cross-references at *cur_name* 233 ⟩ ≡

```
  ⟨ Invert the cross-reference list at cur_name, making cur_xref the head 235 ⟩;
  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 231.

234. 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.

```
⟨ Global variables 7 ⟩ +≡
  xref_pointer next_xref, this_xref;    /* pointer variables for rearranging a list */
```

235. ⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 235 ⟩ ≡

```
  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 233.

236. The following recursive procedure walks through the tree of section names and prints them.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void section_print( );
```

237.

```
void section_print(p)    /* print all section names in subtree p */
  name_pointerp;
  {
    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);
    }
  }
```

238. ⟨ Output all the section names 238 ⟩ ≡

```
  section_print(root)
```

This code is used in section 216.

239. Because on some systems the difference between two pointers is a **long** rather than an **int**, we use `%ld` to print these quantities.

```
void print_stats( )
{
  printf("\nMemory␣usage␣statistics:\n");
  printf("%ld␣names␣(out␣of␣%ld)\n", (long)(name_ptr − name_dir), (long) max_names);
  printf("%ld␣cross-references␣(out␣of␣%ld)\n", (long)(xref_ptr − xmem), (long) max_refs);
  printf("%ld␣bytes␣(out␣of␣%ld)\n", (long)(byte_ptr − byte_mem), (long) max_bytes);
  printf("Parsing:\n");
  printf("%ld␣scraps␣(out␣of␣%ld)\n", (long)(max_scr_ptr − scrap_info), (long) max_scraps);
  printf("%ld␣texts␣(out␣of␣%ld)\n", (long)(max_text_ptr − tok_start), (long) max_texts);
  printf("%ld␣tokens␣(out␣of␣%ld)\n", (long)(max_tok_ptr − tok_mem), (long) max_toks);
  printf("%ld␣levels␣(out␣of␣%ld)\n", (long)(max_stack_ptr − stack), (long) stack_size);
  printf("Sorting:\n");
  printf("%ld␣levels␣(out␣of␣%ld)\n", (long)(max_sort_ptr − scrap_info), (long) max_scraps);
}
```

240. 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.

The CWEAVE processor

(Version 3.64)