The BibTeX preprocessor

(Version 0.99d—March 17, 2021)

March 17, 2021 at 13:05

1* Introduction. BibTeX is a preprocessor (with elements of postprocessing as explained below) for the LaTeX document-preparation system. It handles most of the formatting decisions required to produce a reference list, outputting a `.bbl` file that a user can edit to add any finishing touches BibTeX isn't designed to handle (in practice, such editing almost never is needed); with this file LaTeX actually produces the reference list.

Here's how BibTeX works. It takes as input (a) an `.aux` file produced by LaTeX on an earlier run; (b) a `.bst` file (the style file), which specifies the general reference-list style and specifies how to format individual entries, and which is written by a style designer (called a wizard throughout this program) in a special-purpose language described in the BibTeX documentation—see the file `btxdoc.tex`; and (c) `.bib` file(s) constituting a database of all reference-list entries the user might ever hope to use. BibTeX chooses from the `.bib` file(s) only those entries specified by the `.aux` file (that is, those given by LaTeX's `\cite` or `\nocite` commands), and creates as output a `.bbl` file containing these entries together with the formatting commands specified by the `.bst` file (BibTeX also creates a `.blg` log file, which includes any error or warning messages, but this file isn't used by any program). LaTeX will use the `.bbl` file, perhaps edited by the user, to produce the reference list.

Many modules of BibTeX were taken from Knuth's TeX and TeXware, with his permission. All known system-dependent modules are marked in the index entry "system dependencies"; Dave Fuchs helped exorcise unwanted ones. In addition, a few modules that can be changed to make BibTeX smaller are marked in the index entry "space savings".

Megathanks to Howard Trickey, for whose suggestions future users and style writers would be eternally grateful, if only they knew.

The *banner* string defined here should be changed whenever BibTeX gets modified.

define *my_name* ≡ `´bibtex´`
define *banner* ≡ `´This␣is␣BibTeX,␣Version␣0.99d´` { printed when the program starts }

2* Terminal output goes to the file *term_out*, while terminal input comes from *term_in*. On our system, these (system-dependent) files are already opened at the beginning of the program, and have the same real name.

define *term_out* ≡ *standard_output*
define *term_in* ≡ *standard_input*

⟨ Globals in the outer block 2* ⟩ ≡
standard_input, *standard_output*: *text*;

See also sections 16*, 19, 24, 30, 34, 37*, 41*, 43, 48*, 65*, 74, 76, 78, 80, 89, 91, 97*, 104, 117*, 124, 129*, 147, 161*, 163, 195, 219*, 247, 290*, 331, 337*, 344*, 365, 469*, and 472*.

This code is used in section 10*.

3* This program uses the term *print* instead of *write* when writing on both the *log_file* and (system-dependent) *term_out* file, and it uses *trace_pr* when in **trace** mode, for which it writes on just the *log_file*. If you want to change where either set of macros writes to, you should also change the other macros in this program for that set; each such macro begins with *print_* or *trace_pr_*.

```
define print(#) ≡
        begin write(log_file,#); write(term_out,#);
        end
define print_ln(#) ≡
        begin write_ln(log_file,#); write_ln(term_out,#);
        end
define print_newline ≡ print_a_newline   { making this a procedure saves a little space }
define trace_pr(#) ≡
        begin write(log_file,#);
        end
define trace_pr_ln(#) ≡
        begin write_ln(log_file,#);
        end
define trace_pr_newline ≡
        begin write_ln(log_file);
        end
define log_pr(#) ≡ trace_pr(#)
define log_pr_ln(#) ≡ trace_pr_ln(#)
define log_pr_newline ≡ trace_pr_newline
```

⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ ≡

```
procedure print_a_newline;
  begin write_ln(log_file); write_ln(term_out);
  end;
```

See also sections 18, 38*, 44, 45, 46*, 47*, 51, 53*, 59*, 82, 95, 96, 98, 99, 108*, 111, 112, 113, 114, 115, 121*, 128*, 137, 138*, 144, 148, 149, 150, 153, 157, 158, 159, 165, 166, 167, 168, 169, 188*, 220, 221, 222, 226*, 229, 230, 231, 232, 233, 234, 235, 240, 271, 280, 281, 284, 293, 294, 295, 310, 311, 313, 321, 356, 368, 373, and 456.

This code is used in section 12.

4* Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when BibTeX is being installed or when system wizards are fooling around with BibTeX without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the codewords '**debug** ... **gubed**', with apologies to people who wish to preserve the purity of English. Similarly, there is some conditional code delimited by '**stat** ... **tats**' that is intended only for use when statistics are to be kept about BibTeX's memory/cpu usage, and there is conditional code delimited by '**trace** ... **ecart**' that is intended to be a trace facility for use mainly when debugging `.bst` files.

```
define debug ≡ ifdef('TEXMF_DEBUG')
define gubed ≡ endif('TEXMF_DEBUG')
format debug ≡ begin
format gubed ≡ end

define stat ≡ ifndef('NO_BIBTEX_STAT')
define tats ≡ endifn('NO_BIBTEX_STAT')
format stat ≡ begin
format tats ≡ end

define trace ≡ ifdef@&('TRACE')
define ecart ≡ endif@&('TRACE')
format trace ≡ begin
format ecart ≡ end
```

10* The main program. This program first reads the `.aux` file that LaTeX produces, (i) determining which `.bib` file(s) and `.bst` file to read and (ii) constructing a list of cite keys in order of occurrence. The `.aux` file may have other `.aux` files nested within. Second, it reads and executes the `.bst` file, (i) determining how and in which order to process the database entries in the `.bib` file(s) corresponding to those cite keys in the list (or in some cases, to all the entries in the `.bib` file(s)), (ii) determining what text to be output for each entry and determining any additional text to be output, and (iii) actually outputting this text to the `.bbl` file. In addition, the program sends error messages and other remarks to the *log_file* and terminal.

define *close_up_shop* = 9998 { jump here after fatal errors }
define *exit_program* = 9999 { jump here if we couldn't even get started }

⟨ Compiler directives 11 ⟩
program *BibTEX*; { all files are opened dynamically }
label *close_up_shop* ⟨ Labels in the outer block 109 ⟩;
const ⟨ Constants in the outer block 14* ⟩
type ⟨ Types in the outer block 22* ⟩
var ⟨ Globals in the outer block 2* ⟩
⟨ Procedures and functions for about everything 12 ⟩
⟨ The procedure *initialize* 13* ⟩
⟨ Define *parse_arguments* 467* ⟩
begin *standard_input* ← *stdin*; *standard_output* ← *stdout*;
pool_size ← *POOL_SIZE*; *buf_size* ← *BUF_SIZE*; *max_bib_files* ← *MAX_BIB_FILES*;
max_glob_strs ← *MAX_GLOB_STRS*; *max_fields* ← *MAX_FIELDS*; *max_cites* ← *MAX_CITES*;
wiz_fn_space ← *WIZ_FN_SPACE*; *lit_stk_size* ← *LIT_STK_SIZE*;
setup_params;
{ Add one to the sizes because that's what bibtex uses. }
bib_file ← *XTALLOC*(*max_bib_files* + 1, *alpha_file*);
bib_list ← *XTALLOC*(*max_bib_files* + 1, *str_number*); *entry_ints* ← **nil**; *entry_strs* ← **nil**;
wiz_functions ← *XTALLOC*(*wiz_fn_space* + 1, *hash_ptr2*);
field_info ← *XTALLOC*(*max_fields* + 1, *str_number*);
s_preamble ← *XTALLOC*(*max_bib_files* + 1, *str_number*);
str_pool ← *XTALLOC*(*pool_size* + 1, *ASCII_code*); *buffer* ← *XTALLOC*(*buf_size* + 1, *ASCII_code*);
sv_buffer ← *XTALLOC*(*buf_size* + 1, *ASCII_code*); *ex_buf* ← *XTALLOC*(*buf_size* + 1, *ASCII_code*);
out_buf ← *XTALLOC*(*buf_size* + 1, *ASCII_code*); *name_tok* ← *XTALLOC*(*buf_size* + 1, *buf_pointer*);
name_sep_char ← *XTALLOC*(*buf_size* + 1, *ASCII_code*);
glb_str_ptr ← *XTALLOC*(*max_glob_strs*, *str_number*);
global_strs ← *XTALLOC*(*max_glob_strs* ∗ (*glob_str_size* + 1), *ASCII_code*);
glb_str_end ← *XTALLOC*(*max_glob_strs*, *integer*);
cite_list ← *XTALLOC*(*max_cites* + 1, *str_number*); *type_list* ← *XTALLOC*(*max_cites* + 1, *hash_ptr2*);
entry_exists ← *XTALLOC*(*max_cites* + 1, *boolean*);
cite_info ← *XTALLOC*(*max_cites* + 1, *str_number*);
str_start ← *XTALLOC*(*max_strings* + 1, *pool_pointer*);
hash_next ← *XTALLOC*(*hash_max* + 1, *hash_pointer*);
hash_text ← *XTALLOC*(*hash_max* + 1, *str_number*); *hash_ilk* ← *XTALLOC*(*hash_max* + 1, *str_ilk*);
ilk_info ← *XTALLOC*(*hash_max* + 1, *integer*); *fn_type* ← *XTALLOC*(*hash_max* + 1, *fn_class*);
lit_stack ← *XTALLOC*(*lit_stk_size* + 1, *integer*); *lit_stk_type* ← *XTALLOC*(*lit_stk_size* + 1, *stk_type*);
compute_hash_prime;
initialize; { This initializes the jmp9998 buffer, which can be used early }
hack0;
if *verbose* **then**
begin *print*(*banner*); *print_ln*(*version_string*);

```
    end
  else begin log_pr(banner); log_pr_ln(version_string);
    end;
  log_pr_ln('Capacity:␣max_strings=', max_strings : 1, ',␣hash_size=', hash_size : 1,
      ',␣hash_prime=', hash_prime : 1); ⟨ Read the .aux file 110* ⟩;
  ⟨ Read and execute the .bst file 151* ⟩;
close_up_shop: ⟨ Clean up and leave 455 ⟩;
  if (history > 1) then uexit(history);
  end.
```

13*. This procedure gets things started properly.

```
⟨ The procedure initialize 13* ⟩ ≡
procedure initialize;
  var ⟨ Local variables for initialization 23* ⟩
  begin ⟨ Check the "constant" values for consistency 17* ⟩;
  if (bad > 0) then
    begin write_ln(term_out, bad : 0, '␣is␣a␣bad␣bad'); uexit(1);
    end;
  ⟨ Set initial values of key variables 20 ⟩;
  pre_def_certain_strings;
  get_the_top_level_aux_file_name;
  end;
```

This code is used in section 10*.

14*. These parameters can be changed at compile time to extend or reduce BibTeX's capacity. They are set to accommodate about 750 cites when used with the standard styles, although *pool_size* is usually the first limitation to be a problem, often when there are 500 cites.

```
⟨ Constants in the outer block 14* ⟩ ≡
  hash_base = empty + 1;  { lowest numbered hash-table location }
  quote_next_fn = hash_base − 1;  { special marker used in defining functions }
  BUF_SIZE = 20000;  { initial maximum number of characters in an input line (or string) }
  min_print_line = 3;  { minimum .bbl line length: must be ≥ 3 }
  max_print_line = 79;  { the maximum: must be > min_print_line and < buf_size }
  aux_stack_size = 20;  { maximum number of simultaneous open .aux files }
  MAX_BIB_FILES = 20;  { initial number of .bib files allowed }
  POOL_SIZE = 65000;  { initial number of characters in strings }
  MAX_STRINGS = 4000;  { minimum value for max_strings }
  MAX_CITES = 750;  { initial number of distinct cite keys; must be ≤ max_strings }
  WIZ_FN_SPACE = 3000;  { initial amount of wiz_defined-function space }
    { min_crossrefs can be set at runtime now. }
  SINGLE_FN_SPACE = 50;  { initial amount for a single wiz_defined-function }
  ENT_STR_SIZE = 100;  { maximum size of a str_entry_var; must be ≤ buf_size }
  GLOB_STR_SIZE = 1000;  { maximum size of a str_global_var; must be ≤ buf_size }
  MAX_GLOB_STRS = 10;  { initial number of str_global_var names }
  MAX_FIELDS = 5000;  { initial number of fields (entries × fields, about 23 ∗ max_cites for consistency) }
  LIT_STK_SIZE = 50;  { initial space for literal functions on the stack }
```

See also section 333.

This code is used in section 10*.

15*. These parameters can also be changed at compile time, but they're needed to define some `WEB` numeric macros so they must be so defined themselves.

hash_size and *hash_prime* are now computed.

define *HASH_SIZE* = 5000 { minimum value for *hash_size* }
define *file_name_size* ≡ *maxint* { file names have no arbitrary maximum length }
{ For dynamic allocation. }
define *x_entry_strs_tail*(#) ≡ (#)]
define *x_entry_strs*(#) ≡ *entry_strs* [(#) ∗ (*ent_str_size* + 1) + *x_entry_strs_tail*
define *x_global_strs_tail*(#) ≡ (#)]
define *x_global_strs*(#) ≡ *global_strs* [(#) ∗ (*glob_str_size* + 1) + *x_global_strs_tail*

16*. In case somebody has inadvertently made bad settings of the "constants," BibTeX checks them using a global variable called *bad*.

This is the first of many sections of BibTeX where global variables are defined.

⟨ Globals in the outer block 2* ⟩ +≡
pool_size: *integer*;
max_bib_files: *integer*;
max_cites: *integer*;
wiz_fn_space: *integer*;
ent_str_size: *integer*;
glob_str_size: *integer*;
max_glob_strs: *integer*;
max_fields: *integer*;
lit_stk_size: *integer*;
max_strings: *integer*;
hash_size: *integer*;
hash_prime: *integer*;
hash_max: *integer*; { highest numbered hash-table location }
end_of_def: *integer*; { another special marker used in defining functions }
undefined: *integer*; { a special marker used for *type_list* }
bad: *integer*; { is some "constant" wrong? }

17*. Each digit-value of *bad* has a specific meaning.

⟨ Check the "constant" values for consistency 17* ⟩ ≡
bad ← 0;
if (*min_print_line* < 3) **then** *bad* ← 1;
if (*max_print_line* ≤ *min_print_line*) **then** *bad* ← 10 ∗ *bad* + 2;
if (*max_print_line* ≥ *buf_size*) **then** *bad* ← 10 ∗ *bad* + 3;
if (*hash_prime* < 128) **then** *bad* ← 10 ∗ *bad* + 4;
if (*hash_prime* > *hash_size*) **then** *bad* ← 10 ∗ *bad* + 5;
if (*hash_base* ≠ 1) **then** *bad* ← 10 ∗ *bad* + 6;
if (*max_strings* > *hash_size*) **then** *bad* ← 10 ∗ *bad* + 7;
if (*max_cites* > *max_strings*) **then** *bad* ← 10 ∗ *bad* + 8; { well, almost each }

See also section 302.

This code is used in section 13*.

22* Characters of text that have been converted to TEX's internal form are said to be of type *ASCII_code*, which is a subrange of the integers.

⟨ Types in the outer block 22* ⟩ ≡
ASCII_code = 0 .. 255; { eight-bit numbers }

See also sections 31, 36, 42*, 49*, 64*, 73*, 105, 118*, 130*, 160*, 291*, and 332.

This code is used in section 10*.

23* The original PASCAL compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower-case letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for typesetting; so the present specification of TEX has been written under the assumption that the PASCAL compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes *'40* through *'176*; all of these characters are now available on most computer terminals.

Since we are dealing with more characters than were present in the first PASCAL compilers, we have to decide what to call the associated data type. Some PASCALs use the original name *char* for the characters in text files, even though there now are more than 64 such characters, while other PASCALs consider *char* to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters that are converted to and from *ASCII_code* when they are input and output. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *text_char* ≡ *ASCII_code* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 255 { ordinal number of the largest element of *text_char* }

⟨ Local variables for initialization 23* ⟩ ≡
i: *integer*;

See also section 66.

This code is used in section 13*.

27* The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. Appendix C of *The TEXbook* gives a complete specification of the intended correspondence between characters and TEX's internal representation.

If TEX is being used on a garden-variety PASCAL for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in *xchr*[1 .. *'37*], but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make TEX more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '`\ne`'. At MIT, for example, it would be more appropriate to substitute the code

for *i* ← 1 **to** *'37* **do** *xchr*[*i*] ← *chr*(*i*);

TEX's character set is essentially the same as MIT's, even with respect to characters less than *'40*. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of TEX are allowed to have in their input files. It is best to make the codes correspond to the intended interpretations as shown in Appendix C whenever possible; but this is not necessary. For example, in countries with an alphabet of more than 26 letters, it is usually best to map the additional letters into codes less than *'40*.

⟨ Set initial values of key variables 20 ⟩ +≡
for *i* ← 0 **to** *'37* **do** *xchr*[*i*] ← *chr*(*i*);
for *i* ← *'177* **to** *'377* **do** *xchr*[*i*] ← *chr*(*i*);

28* This system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*. Note that if $xchr[i] = xchr[j]$ where $i < j < \acute{}177$, the value of $xord[xchr[i]]$ will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below *´40* in case there is a coincidence.

⟨ Set initial values of key variables 20 ⟩ +≡
for $i \leftarrow$ *first_text_char* **to** *last_text_char* **do** $xord[xchr[i]] \leftarrow i$;

32* Now we initialize the system-dependent *lex_class* array. The *tab* character may be system dependent. Note that the order of these assignments is important here.

⟨ Set initial values of key variables 20 ⟩ +≡
for $i \leftarrow 0$ **to** *´177* **do** $lex\_class[i] \leftarrow other\_lex$;
for $i \leftarrow$ *´200* **to** *´377* **do** $lex\_class[i] \leftarrow alpha$;
for $i \leftarrow 0$ **to** *´37* **do** $lex\_class[i] \leftarrow illegal$;
$lex\_class[invalid\_code] \leftarrow illegal$; $lex\_class[tab] \leftarrow white\_space$; $lex\_class[13] \leftarrow white\_space$;
$lex\_class[space] \leftarrow white\_space$; $lex\_class[tie] \leftarrow sep\_char$; $lex\_class[hyphen] \leftarrow sep\_char$;
for $i \leftarrow$ *´60* **to** *´71* **do** $lex\_class[i] \leftarrow numeric$;
for $i \leftarrow$ *´101* **to** *´132* **do** $lex\_class[i] \leftarrow alpha$;
for $i \leftarrow$ *´141* **to** *´172* **do** $lex\_class[i] \leftarrow alpha$;

33* And now the *id_class* array.

⟨ Set initial values of key variables 20 ⟩ +≡
for $i \leftarrow 0$ **to** *´377* **do** $id\_class[i] \leftarrow legal\_id\_char$;
for $i \leftarrow 0$ **to** *´37* **do** $id\_class[i] \leftarrow illegal\_id\_char$;
$id\_class[space] \leftarrow illegal\_id\_char$; $id\_class[tab] \leftarrow illegal\_id\_char$; $id\_class[double\_quote] \leftarrow illegal\_id\_char$;
$id\_class[number\_sign] \leftarrow illegal\_id\_char$; $id\_class[comment] \leftarrow illegal\_id\_char$;
$id\_class[single\_quote] \leftarrow illegal\_id\_char$; $id\_class[left\_paren] \leftarrow illegal\_id\_char$;
$id\_class[right\_paren] \leftarrow illegal\_id\_char$; $id\_class[comma] \leftarrow illegal\_id\_char$;
$id\_class[equals\_sign] \leftarrow illegal\_id\_char$; $id\_class[left\_brace] \leftarrow illegal\_id\_char$;
$id\_class[right\_brace] \leftarrow illegal\_id\_char$;

37* Most of what we need to do with respect to input and output can be handled by the I/O facilities that are standard in PASCAL, i.e., the routines called *get*, *put*, *eof*, and so on. But standard PASCAL does not allow file variables to be associated with file names that are determined at run time, so it cannot be used to implement BIBTEX; some sort of extension to PASCAL's ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the PASCAL run-time system being used to implement BIBTEX can open a file whose external name is specified by *name_of_file*. BIBTEX does no case conversion for file names.

⟨ Globals in the outer block 2* ⟩ +≡
name_of_file: ↑*text_char*;
name_length: *integer*; { this many characters are relevant in *name_of_file* }
name_ptr: *integer*; { index variable into *name_of_file* }

38* File opening will be done in C. But we want an auxiliary function to change a BIBTEX string into a C string, to keep string pool stuff out of the C code in `lib/openclose.c`.

define *no_file_path* = −1

⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
function *bib_makecstring*(*s* : *str_number*): *cstring*;
var *cstr*: *cstring*; *i*: *pool_pointer*;
begin *cstr* ← *xmalloc_array*(*ASCII_code*, *length*(*s*) + 1);
for *i* ← 0 **to** *length*(*s*) − 1 **do**
begin *cstr*[*i*] ← *str_pool*[*str_start*[*s*] + *i*];
end;
cstr[*length*(*s*)] ← 0; *bib_makecstring* ← *cstr*;
exit: **end**;

39* Files can be closed with the PASCAL-H routine '*close*(*f*)', which should be used when all input or output with respect to *f* has been completed. This makes *f* available to be opened again, if desired; and if *f* was used for output, the *close* operation makes the corresponding external file appear on the user's area, ready to be read.

File closing will be done in C, too.

41* Input from text files is read one line at a time, using a routine called *input_ln*. This function is defined in terms of global variables called *buffer* and *last*. The *buffer* array contains *ASCII_code* values, and *last* is an index into this array marking the end of a line of text. (Occasionally, *buffer* is used for something else, in which case it is copied to a temporary array.)

⟨ Globals in the outer block 2* ⟩ +≡
buf_size: *integer*; { size of buffer }
buffer: *buf_type*; { usually, lines of characters being read }
last: *buf_pointer*; { end of the line just input to *buffer* }

42* The type *buf_type* is used for *buffer*, for saved copies of it, or for scratch work. It's not **packed** because otherwise the program would run much slower on some systems (more than 25 percent slower, for example, on a TOPS-20 operating system). But on systems that are byte-addressable and that have a good compiler, packing *buf_type* would save lots of space without much loss of speed. Other modules that have packable arrays are also marked with a "space savings" index entry.

⟨ Types in the outer block 22* ⟩ +≡
buf_pointer = *integer*; { an index into a *buf_type* }
buf_type = ↑*ASCII_code*; { for various buffers }

46* When a buffer overflows, it's time to complain (and then quit).

```
⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
procedure buffer_overflow;
  begin { These are all the arrays of buf_type or that use buf_pointer, that is, they all depend on the
      buf_size value. Therefore we have to reallocate them all at once, even though only one of them has
      overflowed. The alternative seems worse: even more surgery on the program, to have a separate
      variable for each array size instead of the common buf_size. }
  BIB_XRETALLOC_NOSET('buffer', buffer, ASCII_code, buf_size, buf_size + BUF_SIZE);
  BIB_XRETALLOC_NOSET('sv_buffer', sv_buffer, ASCII_code, buf_size, buf_size + BUF_SIZE);
  BIB_XRETALLOC_NOSET('ex_buf', ex_buf, ASCII_code, buf_size, buf_size + BUF_SIZE);
  BIB_XRETALLOC_NOSET('out_buf', out_buf, ASCII_code, buf_size, buf_size + BUF_SIZE);
  BIB_XRETALLOC_NOSET('name_tok', name_tok, buf_pointer, buf_size, buf_size + BUF_SIZE);
  BIB_XRETALLOC('name_sep_char', name_sep_char, ASCII_code, buf_size, buf_size + BUF_SIZE);
  end;
```

47* The *input_ln* function brings the next line of input from the specified file into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets $last \leftarrow 0$. In general, the *ASCII_code* numbers that represent the next line of the file are input into $buffer[0]$, $buffer[1]$, ..., $buffer[last-1]$; and the global variable *last* is set equal to the length of the line. Trailing *white_space* characters are removed from the line (*white_space* characters are explained in the character-set section—most likely they're blanks); thus, either $last = 0$ (in which case the line was entirely blank) or $lex\_class[buffer[last-1]] \neq white\_space$. An overflow error is given if the normal actions of *input_ln* would make $last > buf\_size$.

Standard PASCAL says that a file should have *eoln* immediately before *eof*, but BibTeX needs only a weaker restriction: If *eof* occurs in the middle of a line, the system function *eoln* should return a *true* result (even though $f\uparrow$ will be undefined).

```
⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
function input_ln(var f : alpha_file): boolean;  { inputs the next line or returns false }
  label loop_exit;
  begin last ← 0;
  if (eof(f)) then input_ln ← false
  else begin while (¬eoln(f)) do
      begin if (last ≥ buf_size) then buffer_overflow;
      buffer[last] ← xord[getc(f)]; incr(last);
      end;
    vgetc(f);  { skip the eol }
    while (last > 0) do  { remove trailing white_space }
      if (lex_class[buffer[last − 1]] = white_space) then decr(last)
      else goto loop_exit;
  loop_exit: input_ln ← true;
    end;
  end;
```

48* String handling. BIBTEX uses variable-length strings of seven-bit characters. Since PASCAL does not have a well-developed string mechanism, BIBTEX does all its string processing by home-grown (predominantly TEX's) methods. Unlike TEX, however, BIBTEX does not use a *pool_file* for string storage; it creates its few pre-defined strings at run-time.

The necessary operations are handled with a simple data structure. The array *str_pool* contains all the (seven-bit) ASCII codes in all the strings BIBTEX must ever search for (generally identifiers names), and the array *str_start* contains indices of the starting points of each such string. Strings are referred to by integer numbers, so that string number s comprises the characters $str\_pool[j]$ for $str\_start[s] \le j < str\_start[s+1]$. Additional integer variables *pool_ptr* and *str_ptr* indicate the number of entries used so far in *str_pool* and *str_start*; locations $str\_pool[pool\_ptr]$ and $str\_start[str\_ptr]$ are ready for the next string to be allocated. Location $str\_start[0]$ is unused so that hashing will work correctly.

Elements of the *str_pool* array must be ASCII codes that can actually be printed; i.e., they must have an *xchr* equivalent in the local character set.

```
⟨ Globals in the outer block 2* ⟩ +≡
str_pool: ↑ASCII_code;   { the characters }
str_start: ↑pool_pointer;   { the starting pointers }
pool_ptr: pool_pointer;   { first unused position in str_pool }
str_ptr: str_number;   { start of the current string being created }
str_num: str_number;   { general index variable into str_start }
p_ptr1, p_ptr2: pool_pointer;   { several procedures use these locally }
```

49* Where *pool_pointer* and *str_number* are pointers into *str_pool* and *str_start*.

```
⟨ Types in the outer block 22* ⟩ +≡
  pool_pointer = integer;   { for variables that point into str_pool }
  str_number = integer;   { for variables that point into str_start }
```

50* These macros send a string in *str_pool* to an output file.

```
define max_pop = 3   { —see the built_in functions section }
define print_pool_str(#) ≡ print_a_pool_str(#)   { making this a procedure saves a little space }
define trace_pr_pool_str(#) ≡
        begin out_pool_str(log_file, #);
        end
define log_pr_pool_str(#) ≡ trace_pr_pool_str(#)
```

53* Strings are created by appending character codes to *str_pool*. The macro called *append_char*, defined here, does not check to see if the value of *pool_ptr* has gotten too high; this test is supposed to be made before *append_char* is used.

To test if there is room to append l more characters to *str_pool*, we shall write $str\_room(l)$, which aborts BIBTEX and gives an error message if there isn't enough room.

```
define append_char(#) ≡   { put ASCII_code # at the end of str_pool }
       begin str_pool[pool_ptr] ← #; incr(pool_ptr);
       end
define str_room(#) ≡   { make sure that the pool hasn't overflowed }
       begin while (pool_ptr + # > pool_size) do pool_overflow;
       end
```

```
⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
procedure pool_overflow;
  begin BIB_XRETALLOC('str_pool', str_pool, ASCII_code, pool_size, pool_size + POOL_SIZE);
  end;
```

58*. This procedure copies file name *file_name* into the beginning of *name_of_file*, if it will fit. It also sets the global variable *name_length* to the appropriate value.

```
⟨ Procedures and functions for file-system interacting 58* ⟩ ≡
procedure start_name(file_name : str_number);
  var p_ptr: pool_pointer;  { running index }
  begin free(name_of_file); name_of_file ← xmalloc_array(ASCII_code, length(file_name) + 1);
  name_ptr ← 1; p_ptr ← str_start[file_name];
  while (p_ptr < str_start[file_name + 1]) do
    begin name_of_file[name_ptr] ← chr(str_pool[p_ptr]); incr(name_ptr); incr(p_ptr);
    end;
  name_length ← length(file_name); name_of_file[name_length + 1] ← 0;
  end;
```

See also sections 60* and 61*.

This code is used in section 12.

59*. Yet another complaint-before-quiting.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
```

60*. This procedure copies file extension *ext* into the array *name_of_file* starting at position *name_length*+1. It also sets the global variable *name_length* to the appropriate value.

```
⟨ Procedures and functions for file-system interacting 58* ⟩ +≡
procedure add_extension(ext : str_number);
  var p_ptr: pool_pointer;  { running index }
  begin name_ptr ← name_length + 1; p_ptr ← str_start[ext];
  while (p_ptr < str_start[ext + 1]) do
    begin name_of_file[name_ptr] ← chr(str_pool[p_ptr]); incr(name_ptr); incr(p_ptr);
    end;
  name_length ← name_length + length(ext); name_of_file[name_length + 1] ← 0;
  end;
```

61*. This procedure copies the default logical area name *area* into the array *name_of_file* starting at position 1, after shifting up the rest of the filename. It also sets the global variable *name_length* to the appropriate value.

```
⟨ Procedures and functions for file-system interacting 58* ⟩ +≡
```

64* The hash table. All static strings that BIBTEX might have to search for, generally identifiers, are stored and retrieved by means of a fairly standard hash-table algorithm (but slightly altered here) called the method of "coalescing lists" (cf. Algorithm 6.4C in *The Art of Computer Programming*). Once a string enters the table, it is never removed. The actual sequence of characters forming a string is stored in the *str_pool* array.

The hash table consists of the four arrays *hash_next*, *hash_text*, *hash_ilk*, and *ilk_info*. The first array, *hash_next*[*p*], points to the next identifier belonging to the same coalesced list as the identifier corresponding to *p*. The second, *hash_text*[*p*], points to the *str_start* entry for *p*'s string. If position *p* of the hash table is empty, we have *hash_text*[*p*] = 0; if position *p* is either empty or the end of a coalesced hash list, we have *hash_next*[*p*] = *empty*; an auxiliary pointer variable called *hash_used* is maintained in such a way that all locations *p* ≥ *hash_used* are nonempty. The third, *hash_ilk*[*p*], tells how this string is used (as ordinary text, as a variable name, as an `.aux` file command, etc). The fourth, *ilk_info*[*p*], contains information specific to the corresponding *hash_ilk*—for *integer_ilk*s: the integer's value; for *cite_ilk*s: a pointer into *cite_list*; for *lc_cite_ilk*s: a pointer to a *cite_ilk* string; for *command_ilk*s: a constant to be used in a **case** statement; for *bst_fn_ilk*s: function-specific information; for *macro_ilk*s: a pointer to its definition string; for *control_seq_ilk*s: a constant for use in a **case** statement; for all other *ilk*s it contains no information. This *ilk*-specific information is set in other parts of the program rather than here in the hashing routine.

define *hash_is_full* ≡ (*hash_used* = *hash_base*) { test if all positions are occupied }

define *text_ilk* = 0 { a string of ordinary text }
define *integer_ilk* = 1 { an integer (possibly with a *minus_sign*) }
define *aux_command_ilk* = 2 { an `.aux`-file command }
define *aux_file_ilk* = 3 { an `.aux` file name }
define *bst_command_ilk* = 4 { a `.bst`-file command }
define *bst_file_ilk* = 5 { a `.bst` file name }
define *bib_file_ilk* = 6 { a `.bib` file name }
define *file_ext_ilk* = 7 { one of `.aux`, `.bst`, `.bib`, `.bbl`, or `.blg` }
define *file_area_ilk* = 8 { one of `texinputs:` or `texbib:` }
define *cite_ilk* = 9 { a `\citation` argument }
define *lc_cite_ilk* = 10 { a `\citation` argument converted to lower case }
define *bst_fn_ilk* = 11 { a `.bst` function name }
define *bib_command_ilk* = 12 { a `.bib`-file command }
define *macro_ilk* = 13 { a `.bst` macro or a `.bib` string }
define *control_seq_ilk* = 14 { a control sequence specifying a foreign character }
define *last_ilk* = 14 { the same number as on the line above }

⟨ Types in the outer block 22* ⟩ +≡
hash_loc = *integer*; { a location within the hash table }
hash_pointer = *integer*; { either *empty* or a *hash_loc* }
str_ilk = 0 .. *last_ilk*; { the legal string types }

65* ⟨ Globals in the outer block 2* ⟩ +≡
hash_next: ↑*hash_pointer*; { coalesced-list link }
hash_text: ↑*str_number*; { pointer to a string }
hash_ilk: ↑*str_ilk*; { the type of string }
ilk_info: ↑*integer*; { *ilk*-specific info }
hash_used: *integer*; { allocation pointer for hash table }
hash_found: *boolean*; { set to *true* if it's already in the hash table }
dummy_loc: *hash_loc*; { receives *str_lookup* value whenever it's useless }

68*. Here is the subroutine that searches the hash table for a (string, *str_ilk*) pair, where the string is of length $l \geq 0$ and appears in $buffer[j \,..\, (j+l-1)]$. If it finds the pair, it returns the corresponding hash-table location and sets the global variable *hash_found* to *true*. Otherwise it sets *hash_found* to *false*, and if the parameter *insert_it* is *true*, it inserts the pair into the hash table, inserts the string into *str_pool* if not previously encountered, and returns its location. Note that two different pairs can have the same string but different *str_ilk*s, in which case the second pair encountered, if *insert_it* were *true*, would be inserted into the hash table though its string wouldn't be inserted into *str_pool* because it would already be there.

```
define do_insert ≡ true  { insert string if not found in hash table }
define dont_insert ≡ false  { don't insert string }
define str_found = 40  { go here when you've found the string }
define str_not_found = 45  { go here when you haven't }
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
function str_lookup(var buf : buf_type; j, l : buf_pointer; ilk : str_ilk; insert_it : boolean): hash_loc;
        { search the hash table }
  label str_found, str_not_found;
  var h: integer;  { hash code }
    p: hash_loc;  { index into hash_ arrays }
    k: buf_pointer;  { index into buf array }
    str_num: str_number;  { pointer to an already encountered string }
  begin ⟨ Compute the hash code h 69 ⟩;
  p ← h + hash_base;  { start searching here; note that 0 ≤ h < hash_prime }
  hash_found ← false; str_num ← 0;  { set to > 0 if it's an already encountered string }
  loop
    begin ⟨ Process the string if we've already encountered it 70* ⟩;
    if (hash_next[p] = empty) then  { location p may or may not be empty }
      begin if (¬insert_it) then goto str_not_found;
      ⟨ Insert pair into hash table and make p point to it 71* ⟩;
      goto str_found;
      end;
    p ← hash_next[p];  { old and new locations p are not empty }
    end;
str_not_found: do_nothing;  { don't insert pair; function value meaningless }
str_found: str_lookup ← p;
  end;
```

70*. Here we handle the case in which we've already encountered this string; note that even if we have, we'll still have to insert the pair into the hash table if *str_ilk* doesn't match.

```
⟨ Process the string if we've already encountered it 70* ⟩ ≡
  begin if (hash_text[p] > 0) then  { there's something here }
    if (str_eq_buf(hash_text[p], buf, j, l)) then  { it's the right string }
      if (hash_ilk[p] = ilk) then  { it's the right str_ilk }
        begin hash_found ← true; goto str_found;
        end
      else begin  { it's the wrong str_ilk }
        str_num ← hash_text[p];
        end;
  end
```

This code is used in section 68*.

71*. This code inserts the pair in the appropriate unused location.

```
⟨ Insert pair into hash table and make p point to it 71* ⟩ ≡
  begin if (hash_text[p] > 0) then { location p isn't empty }
    begin repeat if (hash_is_full) then overflow('hash␣size␣', hash_size);
      decr(hash_used);
    until (hash_text[hash_used] = 0); { search for an empty location }
    hash_next[p] ← hash_used; p ← hash_used;
    end; { now location p is empty }
  if (str_num > 0) then { it's an already encountered string }
    hash_text[p] ← str_num
  else begin { it's a new string }
    str_room(l); { make sure it'll fit in str_pool }
    k ← j;
    while (k < j + l) do { not a for loop in case j = l = 0 }
      begin append_char(buf[k]); incr(k);
      end;
    hash_text[p] ← make_string; { and make it official }
    end;
  hash_ilk[p] ← ilk;
  end
```

This code is used in section 68*.

73*. The longest pre-defined string determines type definitions used to insert the pre-defined strings into *str_pool*.

```
  define longest_pds = 12 { the length of 'change.case$' }
⟨ Types in the outer block 22* ⟩ +≡
  pds_loc = 1 .. longest_pds; pds_len = 0 .. longest_pds; pds_type = const_cstring;
```

77*. This procedure initializes a pre-defined string of length at most *longest_pds*.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure pre_define(pds : pds_type; len : pds_len; ilk : str_ilk);
  var i: pds_len;
  begin for i ← 1 to len do buffer[i] ← xord[ucharcast(pds[i − 1])];
  pre_def_loc ← str_lookup(buffer, 1, len, ilk, do_insert);
  end;
```

97* Getting the top-level auxiliary file name. These modules read the name of the top-level `.aux` file. Some systems will try to find this on the command line; if it's not there it will come from the user's terminal. In either case, the name goes into the *char* array *name_of_file*, and the files relevant to this name are opened.

```
define aux_found = 41 { go here when the .aux name is legit }
define aux_not_found = 46 { go here when it's not }
⟨ Globals in the outer block 2* ⟩ +≡
aux_name_length: integer;
```

100* This module and the next two must be changed on those systems using command-line arguments.

```
⟨ Procedures and functions for the reading and processing of input files 100* ⟩ ≡
procedure get_the_top_level_aux_file_name;
  label aux_found, aux_not_found;
  begin ⟨ Process a possible command line 102* ⟩
    { Leave room for the ., the extension, the junk byte at the beginning, and the null byte at the end. }
  name_of_file ← xmalloc_array(ASCII_code, strlen(cmdline(optind)) + 5);
  strcpy(stringcast(name_of_file + 1), cmdline(optind));
  aux_name_length ← strlen(stringcast(name_of_file + 1)); ⟨ Handle this .aux name 103 ⟩;
aux_not_found: uexit(1);
aux_found: { now we're ready to read the .aux file }
  end;
```

See also sections 120, 126, 132, 139, 142, 143, 145, 170, 177, 178, 180, 201, 203, 205, 210, 211, 212, 214, 215, and 217.

This code is used in section 12.

101* The switch *check_cmnd_line* tells us whether we're to check for a possible command-line argument.

102* Here's where we do the real command-line work. Those systems needing more than a single module to handle the task should add the extras to the "System-dependent changes" section.

```
⟨ Process a possible command line 102* ⟩ ≡
  parse_arguments;
```

This code is used in section 100*.

106* We must make sure the (top-level) `.aux`, `.blg`, and `.bbl` files can be opened.

```
⟨ Add extensions and open files 106* ⟩ ≡
  begin name_length ← aux_name_length; { set to last used position }
  if (name_length < 4) ∨ (strcmp(stringcast(name_of_file + 1 + name_length − 4), '.aux') ≠ 0) then
    add_extension(s_aux_extension) { this also sets name_length }
  else aux_name_length ← aux_name_length − 4; { set to length without .aux }
  aux_ptr ← 0; { initialize the .aux file stack }
  if (¬kpse_in_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_in(cur_aux_file, no_file_path)) then
    sam_you_made_the_file_name_wrong;
  name_length ← aux_name_length; add_extension(s_log_extension); { this also sets name_length }
  if (¬kpse_out_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_out(log_file)) then
    sam_you_made_the_file_name_wrong;
  name_length ← aux_name_length; add_extension(s_bbl_extension); { this also sets name_length }
  if (¬kpse_out_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_out(bbl_file)) then
    sam_you_made_the_file_name_wrong;
  end
```

This code is used in section 103.

108*. Print the name of the current `.aux` file, followed by a *newline*.

⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡

```
procedure print_aux_name;
  begin print_pool_str(cur_aux_str); print_newline;
  end;
procedure log_pr_aux_name;
  begin log_pr_pool_str(cur_aux_str); log_pr_newline;
  end;
```

110*. We keep reading and processing input lines until none left. This is part of the main program; hence, because of the *aux_done* label, there's no conventional **begin** - **end** pair surrounding the entire module.

```
⟨ Read the .aux file 110* ⟩ ≡
  if verbose then
    begin print(´The␣top-level␣auxiliary␣file:␣´); print_aux_name;
    end
  else begin log_pr(´The␣top-level␣auxiliary␣file:␣´); log_pr_aux_name;
    end;
  loop
    begin { pop_the_aux_stack will exit the loop }
    incr(cur_aux_line);
    if (¬input_ln(cur_aux_file)) then { end of current .aux file }
      pop_the_aux_stack
    else get_aux_command_and_process;
    end;
  trace trace_pr_ln(´Finished␣reading␣the␣auxiliary␣file(s)´);
  ecart
aux_done: last_check_for_aux_errors;
```

This code is used in section 10*.

117*. Here we introduce some variables for processing a `\bibdata` command. Each element in *bib_list* (except for *bib_list*[*max_bib_files*], which is always unused) is a pointer to the appropriate *str_pool* string representing the `.bib` file name. The array *bib_file* contains the corresponding PASCAL **file** variables.

```
  define cur_bib_str ≡ bib_list[bib_ptr] { shorthand for current .bib file }
  define cur_bib_file ≡ bib_file[bib_ptr] { shorthand for current bib_file }
⟨ Globals in the outer block 2* ⟩ +≡
bib_list: ↑str_number; { the .bib file list }
bib_ptr: bib_number; { pointer for the current .bib file }
num_bib_files: bib_number; { the total number of .bib files }
bib_seen: boolean; { true if we've already seen a \bibdata command }
bib_file: ↑alpha_file; { corresponding file variables }
```

118*. Where *bib_number* is the obvious.

```
⟨ Types in the outer block 22* ⟩ +≡
  bib_number = integer; { gives the bib_list range }
```

121*. Here's a procedure we'll need shortly. It prints the name of the current `.bib` file, followed by a *newline*.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
    { Return true if the ext string is at the end of the s string. There are surely far more clever ways to do
      this, but it doesn't matter. }
function str_ends_with(s : str_number; ext : str_number): boolean;
  var i: integer; str_idx, ext_idx: integer; str_char, ext_char: ASCII_code;
  begin str_ends_with ← false;
  if (length(ext) > length(s)) then return; { if extension is longer, they don't match }
  str_idx ← length(s) − 1; ext_idx ← length(ext) − 1;
  while (ext_idx ≥ 0) do
    begin { ≥ so we check the '.' char. }
    str_char ← str_pool[str_start[s] + str_idx]; ext_char ← str_pool[str_start[ext] + ext_idx];
    if (str_char ≠ ext_char) then return;
    decr(str_idx); decr(ext_idx);
    end;
  str_ends_with ← true;
exit: end; { The above is needed because the file name specified in the \bibdata command may or may
      not have the .bib extension. If it does, we don't want to print .bib twice. }
procedure print_bib_name;
  begin print_pool_str(cur_bib_str);
  if ¬str_ends_with(cur_bib_str, s_bib_extension) then print_pool_str(s_bib_extension);
  print_newline;
  end;
procedure log_pr_bib_name;
  begin log_pr_pool_str(cur_bib_str);
  if ¬str_ends_with(cur_bib_str, s_bib_extension) then log_pr_pool_str(s_bib_extension);
  log_pr_newline;
  end;
```

123*. Now we add the just-found argument to *bib_list* if it hasn't already been encountered as a `\bibdata` argument and if, after appending the *s_bib_extension* string, the resulting file name can be opened.

```
⟨ Open a .bib file 123* ⟩ ≡
  begin if (bib_ptr = max_bib_files) then
    begin { Keep old value of max_bib_files for the last array. }
    BIB_XRETALLOC_NOSET('bib_list', bib_list, str_number, max_bib_files,
        max_bib_files + MAX_BIB_FILES); BIB_XRETALLOC_NOSET('bib_file', bib_file, alpha_file,
        max_bib_files, max_bib_files + MAX_BIB_FILES); BIB_XRETALLOC('s_preamble', s_preamble,
        str_number, max_bib_files, max_bib_files + MAX_BIB_FILES);
    end;
  cur_bib_str ← hash_text[str_lookup(buffer, buf_ptr1, token_len, bib_file_ilk, do_insert)];
  if (hash_found) then { already encountered this as a \bibdata argument }
    open_bibdata_aux_err('This␣database␣file␣appears␣more␣than␣once:␣');
  start_name(cur_bib_str);
  if (¬kpse_in_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_in(cur_bib_file, kpse_bib_format)) then
    open_bibdata_aux_err('I␣couldn''t␣open␣database␣file␣');
  trace trace_pr_pool_str(cur_bib_str); trace_pr_pool_str(s_bib_extension);
  trace_pr_ln('␣is␣a␣bibdata␣file');
  ecart
  incr(bib_ptr);
  end
```

This code is used in section 120.

127*. Now we open the file whose name is the just-found argument appended with the *s_bst_extension* string, if possible.

```
⟨ Open the .bst file 127* ⟩ ≡
  begin bst_str ← hash_text[str_lookup(buffer, buf_ptr1, token_len, bst_file_ilk, do_insert)];
  if (hash_found) then
    begin trace print_bst_name;
    ecart
    confusion('Already␣encountered␣style␣file');
    end;
  start_name(bst_str);
  if (¬kpse_in_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_in(bst_file, kpse_bst_format)) then
    begin print('I␣couldn''t␣open␣style␣file␣'); print_bst_name;
    bst_str ← 0; { mark as unused again }
    aux_err_return;
    end;
  if verbose then
    begin print('The␣style␣file:␣'); print_bst_name;
    end
  else begin log_pr('The␣style␣file:␣'); log_pr_bst_name;
    end;
  end
```

This code is used in section 126.

128* Print the name of the `.bst` file, followed by a *newline*.

⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
procedure *print_bst_name*;
  begin *print_pool_str*(*bst_str*); *print_pool_str*(*s_bst_extension*); *print_newline*;
  end;
procedure *log_pr_bst_name*;
  begin *log_pr_pool_str*(*bst_str*); *log_pr_pool_str*(*s_bst_extension*); *log_pr_newline*;
  end;

129* Here we introduce some variables for processing a `\citation` command. Each element in *cite_list* (except for *cite_list*[*max_cites*], which is always unused) is a pointer to the appropriate *str_pool* string. The cite-key list is kept in order of occurrence with duplicates removed.

  define *cur_cite_str* ≡ *cite_list*[*cite_ptr*] { shorthand for the current cite key }

⟨ Globals in the outer block 2* ⟩ +≡
cite_list: ↑*str_number*; { the cite-key list }
cite_ptr: *cite_number*; { pointer for the current cite key }
entry_cite_ptr: *cite_number*; { cite pointer for the current entry }
num_cites: *cite_number*; { the total number of distinct cite keys }
old_num_cites: *cite_number*; { set to a previous *num_cites* value }
citation_seen: *boolean*; { *true* if we've seen a `\citation` command }
cite_loc: *hash_loc*; { the hash-table location of a cite key }
lc_cite_loc: *hash_loc*; { and of its lower-case equivalent }
lc_xcite_loc: *hash_loc*; { a second *lc_cite_loc* variable }
cite_found: *boolean*; { *true* if we've already seen this cite key }
all_entries: *boolean*; { *true* if we're to use the entire database }
all_marker: *cite_number*; { we put the other entries in *cite_list* here }

130* Where *cite_number* is the obvious.

⟨ Types in the outer block 22* ⟩ +≡
  cite_number = *integer*; { gives the *cite_list* range }

138* Complain if somebody's got a cite fetish. This procedure is called when were about to add another cite key to *cite_list*. It assumes that *cite_loc* gives the potential cite key's hash table location.

⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
procedure *check_cite_overflow*(*last_cite* : *cite_number*);
  begin if (*last_cite* = *max_cites*) **then**
    begin *BIB_XRETALLOC_NOSET*(´`cite_list`´, *cite_list*, *str_number*, *max_cites*,
        max_cites + *MAX_CITES*);
    BIB_XRETALLOC_NOSET(´`type_list`´, *type_list*, *hash_ptr2*, *max_cites*, *max_cites* + *MAX_CITES*);
    BIB_XRETALLOC_NOSET(´`entry_exists`´, *entry_exists*, *boolean*, *max_cites*,
        max_cites + *MAX_CITES*);
    BIB_XRETALLOC(´`cite_info`´, *cite_info*, *str_number*, *max_cites*, *max_cites* + *MAX_CITES*);
    while (*last_cite* < *max_cites*) **do**
      begin *type_list*[*last_cite*] ← *empty*;
      cite_info[*last_cite*] ← *any_value*; { to appease PASCAL's boolean evaluation }
      incr(*last_cite*);
      end;
    end;
  end;

141*. We check that this `.aux` file can actually be opened, and then open it.

```
⟨ Open this .aux file 141* ⟩ ≡
  begin start_name(cur_aux_str);  { extension already there for .aux files }
  name_ptr ← name_length + 1; name_of_file[name_ptr] ← 0;
  if (¬kpse_in_name_ok(stringcast(name_of_file + 1)) ∨ (¬a_open_in(cur_aux_file,
        no_file_path) ∧ ¬a_open_in_with_dirname(cur_aux_file, no_file_path, bib_makecstring(top_lev_str))))
        then
    begin print('I couldn''t open auxiliary file '); print_aux_name; decr(aux_ptr);
    aux_err_return;
    end;
  log_pr('A level-', aux_ptr : 0, ' auxiliary file: '); log_pr_aux_name; cur_aux_line ← 0;
  end
```

This code is used in section 140.

151*. Here's the outer loop for reading the `.bst` file—it keeps reading and processing `.bst` commands until none left. This is part of the main program; hence, because of the *bst_done* label, there's no conventional **begin - end** pair surrounding the entire module.

```
⟨ Read and execute the .bst file 151* ⟩ ≡
  if (bst_str = 0) then { there's no .bst file to read }
    goto no_bst_file; { this is a goto so that bst_done is not in a block }
  bst_line_num ← 0; { initialize things }
  bbl_line_num ← 1; { best spot to initialize the output line number }
  buf_ptr2 ← last; { to get the first input line }
  hack1;
  begin if (¬eat_bst_white_space) then { the end of the .bst file }
    hack2;
  get_bst_command_and_process;
  end;
bst_done: a_close(bst_file);
no_bst_file: a_close(bbl_file);
```

This code is used in section 10*.

160*. Besides the function classes, we have types based on BIBTEX's capacity limitations and one based on what can go into the array *wiz_functions* explained below.

```
⟨ Types in the outer block 22* ⟩ +≡
  fn_class = 0 .. last_fn_class; { the .bst function classes }
  wiz_fn_loc = integer; { wiz_defined-function storage locations }
  int_ent_loc = integer; { int_entry_var storage locations }
  str_ent_loc = integer; { str_entry_var storage locations }
  str_glob_loc = integer; { str_global_var storage locations }
  field_loc = integer; { individual field storage locations }
  hash_ptr2 = quote_next_fn .. end_of_def; { a special marker or a hash_loc }
```

161*. We store information about the `.bst` functions in arrays the same size as the hash-table arrays and in locations corresponding to their hash-table locations. The two arrays *fn_info* (an alias of *ilk_info* described earlier) and *fn_type* accomplish this: *fn_type* specifies one of the above classes, and *fn_info* gives information dependent on the class.

Six other arrays give the contents of functions: The array *wiz_functions* holds definitions for *wiz_defined* functions—each such function consists of a sequence of pointers to hash-table locations of other functions (with the two special-marker exceptions above); the array *entry_ints* contains the current values of *int_entry_var*s; the array *entry_strs* contains the current values of *str_entry_var*s; an element of the array *global_strs* contains the current value of a *str_global_var* if the corresponding *glb_str_ptr* entry is empty, otherwise the nonempty entry is a pointer to the string; and the array *field_info*, for each field of each entry, contains either a pointer to the string or the special value *missing*.

The array *global_strs* isn't packed (that is, it isn't **array** ... **of packed array** ...) to increase speed on some systems; however, on systems that are byte-addressable and that have a good compiler, packing *global_strs* would save lots of space without much loss of speed.

define *fn_info* ≡ *ilk_info* { an alias used with functions }

define *missing* = *empty* { a special pointer for missing fields }

⟨ Globals in the outer block 2* ⟩ +≡
fn_loc: *hash_loc*; { the hash-table location of a function }
wiz_loc: *hash_loc*; { the hash-table location of a wizard function }
literal_loc: *hash_loc*; { the hash-table location of a literal function }
macro_name_loc: *hash_loc*; { the hash-table location of a macro name }
macro_def_loc: *hash_loc*; { the hash-table location of a macro definition }
fn_type: ↑*fn_class*;
wiz_def_ptr: *wiz_fn_loc*; { storage location for the next wizard function }
wiz_fn_ptr: *wiz_fn_loc*; { general *wiz_functions* location }
wiz_functions: ↑*hash_ptr2*;
int_ent_ptr: *int_ent_loc*; { general *int_entry_var* location }
entry_ints: ↑*integer*; { dynamically-allocated array }
num_ent_ints: *int_ent_loc*; { the number of distinct *int_entry_var* names }
str_ent_ptr: *str_ent_loc*; { general *str_entry_var* location }
entry_strs: ↑*ASCII_code*; { dynamically-allocated array }
num_ent_strs: *str_ent_loc*; { the number of distinct *str_entry_var* names }
str_glb_ptr: *integer*; { general *str_global_var* location }
glb_str_ptr: ↑*str_number*;
global_strs: ↑*ASCII_code*;
glb_str_end: ↑*integer*; { end markers }
num_glb_strs: *integer*; { number of distinct *str_global_var* names }
field_ptr: *field_loc*; { general *field_info* location }
field_parent_ptr, *field_end_ptr*: *field_loc*; { two more for doing cross-refs }
cite_parent_ptr, *cite_xptr*: *cite_number*; { two others for doing cross-refs }
field_info: ↑*str_number*;
num_fields: *field_loc*; { the number of distinct field names }
num_pre_defined_fields: *field_loc*; { so far, just one: `crossref` }
crossref_num: *field_loc*; { the number given to `crossref` }
no_fields: *boolean*; { used for *tr_print*ing entry information }

187* This recursive function reads and stores the list of functions (separated by *white_space* characters or ends-of-line) that define this new function, and reads a *right_brace*.

```
⟨ Procedures and functions for input scanning 83 ⟩ +≡
procedure scan_fn_def(fn_hash_loc : hash_loc);
  label next_token, exit;
  type fn_def_loc = integer;  { for a single wiz_defined-function }
  var singl_function: ↑hash_ptr2; single_fn_space: integer;
        { space allocated for this singl_function instance }
    single_ptr: fn_def_loc;  { next storage location for this definition }
    copy_ptr: fn_def_loc;  { dummy variable }
    end_of_num: buf_pointer;  { the end of an implicit function's name }
    impl_fn_loc: hash_loc;  { an implicit function's hash-table location }
  begin single_fn_space ← SINGLE_FN_SPACE;
  singl_function ← XTALLOC(single_fn_space + 1, hash_ptr2); eat_bst_white_and_eof_check('function');
  single_ptr ← 0;
  while (scan_char ≠ right_brace) do
    begin ⟨ Get the next function of the definition 189 ⟩;
  next_token: eat_bst_white_and_eof_check('function');
    end;
  ⟨ Complete this function's definition 200* ⟩;
  incr(buf_ptr2);  { skip over the right_brace }
exit: libc_free(singl_function);
  end;
```

188* This macro inserts a hash-table location (or one of the two special markers *quote_next_fn* and *end_of_def*) into the *singl_function* array, which will later be copied into the *wiz_functions* array.

```
  define insert_fn_loc(#) ≡
          begin singl_function[single_ptr] ← #;
          if (single_ptr = single_fn_space) then
            begin BIB_XRETALLOC('singl_function', singl_function, hash_ptr2, single_fn_space,
                single_fn_space + SINGLE_FN_SPACE);
            end;
          incr(single_ptr);
          end
```

⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡

198*. This procedure takes the integer *int*, copies the appropriate *ASCII_code* string into *int_buf* starting at *int_begin*, and sets the **var** parameter *int_end* to the first unused *int_buf* location. The ASCII string will consist of decimal digits, the first of which will be not be a 0 if the integer is nonzero, with a prepended minus sign if the integer is negative.

```
define int ≡ the_int
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure int_to_ASCII(int : integer; var int_buf : buf_type; int_begin : buf_pointer;
        var int_end : buf_pointer);
  var int_ptr, int_xptr: buf_pointer;  { pointers into int_buf }
    int_tmp_val: ASCII_code;  { the temporary element in an exchange }
  begin int_ptr ← int_begin;
  if (int < 0) then  { add the minus_sign and use the absolute value }
    begin append_int_char(minus_sign); int ← −int;
    end;
  int_xptr ← int_ptr;
  repeat  { copy digits into int_buf }
    append_int_char("0" + (int mod 10)); int ← int div 10;
  until (int = 0);
  int_end ← int_ptr;  { set the string length }
  decr(int_ptr);
  while (int_xptr < int_ptr) do  { and reorder (flip) the digits }
    begin int_tmp_val ← int_buf[int_xptr]; int_buf[int_xptr] ← int_buf[int_ptr];
    int_buf[int_ptr] ← int_tmp_val; decr(int_ptr); incr(int_xptr);
    end
  end;
```

200*. Now we add the *end_of_def* special marker, make sure this function will fit into *wiz_functions*, and put it there.

```
⟨ Complete this function's definition 200* ⟩ ≡
  begin insert_fn_loc(end_of_def);  { add special marker ending the definition }
  while (single_ptr + wiz_def_ptr > wiz_fn_space) do
    begin BIB_XRETALLOC('wiz_functions', wiz_functions, hash_ptr2, wiz_fn_space,
        wiz_fn_space + WIZ_FN_SPACE);
    end;
  fn_info[fn_hash_loc] ← wiz_def_ptr;  { pointer into wiz_functions }
  copy_ptr ← 0;
  while (copy_ptr < single_ptr) do  { make this function official }
    begin wiz_functions[wiz_def_ptr] ← singl_function[copy_ptr]; incr(copy_ptr); incr(wiz_def_ptr);
    end;
  end
```

This code is used in section 187*.

216*. Here we insert the just found *str_global_var* name into the hash table, record it as a *str_global_var*, set its pointer into *global_strs*, and initialize its value there to the null string.

```
define end_of_string = invalid_code { this illegal ASCII_code ends a string }
⟨ Insert a str_global_var into the hash table 216* ⟩ ≡
  begin trace trace_pr_token; trace_pr_ln('␣is␣a␣string␣global-variable');
  ecart
  lower_case(buffer, buf_ptr1, token_len); { ignore case differences }
  fn_loc ← str_lookup(buffer, buf_ptr1, token_len, bst_fn_ilk, do_insert);
  check_for_already_seen_function(fn_loc); fn_type[fn_loc] ← str_global_var;
  fn_info[fn_loc] ← num_glb_strs; { pointer into global_strs }
  if (num_glb_strs = max_glob_strs) then
    begin BIB_XRETALLOC_NOSET('glb_str_ptr', glb_str_ptr, str_number, max_glob_strs,
        max_glob_strs + MAX_GLOB_STRS); BIB_XRETALLOC_STRING('global_strs', global_strs,
        glob_str_size, max_glob_strs, max_glob_strs + MAX_GLOB_STRS);
    BIB_XRETALLOC('glb_str_end', glb_str_end, integer, max_glob_strs,
        max_glob_strs + MAX_GLOB_STRS); str_glb_ptr ← num_glb_strs;
    while (str_glb_ptr < max_glob_strs) do { make new str_global_vars empty }
      begin glb_str_ptr[str_glb_ptr] ← 0; glb_str_end[str_glb_ptr] ← 0; incr(str_glb_ptr);
      end;
    end;
  incr(num_glb_strs);
  end
```

This code is used in section 215.

219* These global variables are used while reading the `.bib` file(s). The elements of *type_list*, which indicate an entry's type (book, article, etc.), point either to a *hash_loc* or are one of two special markers: *empty*, from which *hash_base* = *empty* + 1 was defined, means we haven't yet encountered the `.bib` entry corresponding to this cite key; and *undefined* means we've encountered it but it had an unknown entry type. Thus the array *type_list* is of type *hash_ptr2*, also defined earlier. An element of the boolean array *entry_exists* whose corresponding entry in *cite_list* gets overwritten (which happens only when *all_entries* is *true*) indicates whether we've encountered that entry of *cite_list* while reading the `.bib` file(s); this information is unused for entries that aren't (or more precisely, that have no chance of being) overwritten. When we're reading the database file, the array *cite_info* contains auxiliary information for *cite_list*. Later, *cite_info* will become *sorted_cites*, and this dual role imposes the (not-very-imposing) restriction *max_strings* ≥ *max_cites*.

```
⟨ Globals in the outer block 2* ⟩ +≡
bib_line_num: integer;   { line number of the .bib file }
entry_type_loc: hash_loc;   { the hash-table location of an entry type }
type_list: ↑hash_ptr2;
type_exists: boolean;   { true if this entry type is .bst-defined }
entry_exists: ↑boolean;
store_entry: boolean;   { true if we're to store info for this entry }
field_name_loc: hash_loc;   { the hash-table location of a field name }
field_val_loc: hash_loc;   { the hash-table location of a field value }
store_field: boolean;   { true if we're to store info for this field }
store_token: boolean;   { true if we're to store this macro token }
right_outer_delim: ASCII_code;   { either a right_brace or a right_paren }
right_str_delim: ASCII_code;   { either a right_brace or a double_quote }
at_bib_command: boolean;   { true for a command, false for an entry }
cur_macro_loc: hash_loc;   { macro_loc for a string being defined }
cite_info: ↑str_number;   { extra cite_list info }
cite_hash_found: boolean;   { set to a previous hash_found value }
preamble_ptr: bib_number;   { pointer into the s_preamble array }
num_preamble_strings: bib_number;   { counts the s_preamble strings }
```

223* For all *num_bib_files* database files, we keep reading and processing `.bib` entries until none left.

```
⟨ Read the .bib file(s) 223* ⟩ ≡
  begin ⟨ Final initialization for .bib processing 224 ⟩;
  read_performed ← true; bib_ptr ← 0;
  while (bib_ptr < num_bib_files) do
    begin if verbose then
      begin print('Database file #', bib_ptr + 1 : 0, ': '); print_bib_name;
      end
    else begin log_pr('Database file #', bib_ptr + 1 : 0, ': '); log_pr_bib_name;
      end;
    bib_line_num ← 0;   { initialize to get the first input line }
    buf_ptr2 ← last;
    while (¬eof(cur_bib_file)) do get_bib_command_or_entry_and_process;
    a_close(cur_bib_file); incr(bib_ptr);
    end;
  reading_completed ← true;
  trace trace_pr_ln('Finished reading the database file(s)');
  ecart
  ⟨ Final initialization for processing the entries 276 ⟩;
  read_completed ← true;
  end
```

This code is used in section 211.

226* Complain if somebody's got a field fetish.

```
⟨ Procedures and functions for all file I/O, error messages, and such 3* ⟩ +≡
procedure check_field_overflow(total_fields : integer);
  var f_ptr: field_loc; start_fields: field_loc;
  begin if (total_fields > max_fields) then
    begin start_fields ← max_fields;
    BIB_XRETALLOC('field_info', field_info, str_number, max_fields, total_fields + MAX_FIELDS);
        { Initialize to missing. }
    for f_ptr ← start_fields to max_fields − 1 do
      begin field_info[f_ptr] ← missing;
      end;
    end;
  end;
```

242* The `preamble` command lets a user have TEX stuff inserted (by the standard styles, at least) directly into the `.bbl` file. It is intended primarily for allowing TEX macro definitions used within the bibliography entries (for better sorting, for example). One `preamble` command per `.bib` file should suffice.

A `preamble` command has either braces or parentheses as outer delimiters. Inside is the preamble string, which has the same syntax as a field value: a nonempty list of field tokens separated by *concat_char*s. There are three types of field tokens—nonnegative numbers, macro names, and delimited strings.

This module does all the scanning (that's not subcontracted), but the `.bib`-specific scanning function *scan_and_store_the_field_value_and_eat_white* actually stores the value.

```
⟨ Process a preamble command 242* ⟩ ≡
  begin if (preamble_ptr = max_bib_files) then
    begin    { Keep old value of max_bib_files for the last array. }
    BIB_XRETALLOC_NOSET('bib_list', bib_list, str_number, max_bib_files,
        max_bib_files + MAX_BIB_FILES); BIB_XRETALLOC_NOSET('bib_file', bib_file, alpha_file,
        max_bib_files, max_bib_files + MAX_BIB_FILES); BIB_XRETALLOC('s_preamble', s_preamble,
        str_number, max_bib_files, max_bib_files + MAX_BIB_FILES);
    end;
  eat_bib_white_and_eof_check;
  if (scan_char = left_brace) then right_outer_delim ← right_brace
  else if (scan_char = left_paren) then right_outer_delim ← right_paren
    else bib_one_of_two_expected_err(left_brace, left_paren);
  incr(buf_ptr2);    { skip over the left-delimiter }
  eat_bib_white_and_eof_check; store_field ← true;
  if (¬scan_and_store_the_field_value_and_eat_white) then return;
  if (scan_char ≠ right_outer_delim) then
    bib_err('Missing␣"', xchr[right_outer_delim], '"␣in␣preamble␣command');
  incr(buf_ptr2);    { skip over the right_outer_delim }
  return;
  end
```

This code is used in section 239.

251* Now we come to the stuff that actually accumulates the field value to be stored. This module copies a character into *field_vl_str* if it will fit; since it's so low level, it's implemented as a macro.

```
define copy_char(#) ≡
    begin { We don't always increment by 1, so have to check ≥. }
    if (field_end ≥ buf_size) then
      begin log_pr('Field␣filled␣up␣at␣', #, ',␣reallocating.'); log_pr_newline;
      buffer_overflow; { reallocates all buf_size buffers }
      end;
    field_vl_str[field_end] ← #; incr(field_end);
    end
```

263* And here, an entry.

```
⟨ Store the field value for a database entry 263* ⟩ ≡
  begin field_ptr ← entry_cite_ptr * num_fields + fn_info[field_name_loc];
  if (field_ptr ≥ max_fields) then confusion('field_info␣index␣is␣out␣of␣range');
  if (field_info[field_ptr] ≠ missing) then
    begin print('Warning--I''m␣ignoring␣'); print_pool_str(cite_list[entry_cite_ptr]);
    print('''s␣extra␣"'); print_pool_str(hash_text[field_name_loc]); bib_warn_newline('"␣field');
    end
  else begin { the field was empty, store its new value }
    field_info[field_ptr] ← hash_text[field_val_loc];
    if ((fn_info[field_name_loc] = crossref_num) ∧ (¬all_entries)) then
      ⟨ Add or update a cross reference on cite_list if necessary 264 ⟩;
    end;
  end
```

This code is used in section 261.

265* This procedure adds (or restores) to *cite_list* a cite key; it is called only when *all_entries* is *true* or when adding cross references, and it assumes that *cite_loc* and *lc_cite_loc* are set. It also increments its argument.

```
⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡
procedure add_database_cite(var new_cite : cite_number);
  begin check_cite_overflow(new_cite); { make sure this cite will fit }
  check_field_overflow(num_fields * (new_cite + 1)); cite_list[new_cite] ← hash_text[cite_loc];
  ilk_info[cite_loc] ← new_cite; ilk_info[lc_cite_loc] ← cite_loc; incr(new_cite);
  end;
```

277*. Now we update any entry (here called a *child* entry) that cross referenced another (here called a *parent* entry); this cross referencing occurs when the child's `crossref` field (value) consists of the parent's database key. To do the update, we replace the child's *missing* fields by the corresponding fields of the parent. Also, we make sure the `crossref` field contains the case-correct version. Finally, although it is technically illegal to nest cross references, and although we give a warning (a few modules hence) when someone tries, we do what we can to accommodate the attempt.

```
⟨ Add cross-reference information 277* ⟩ ≡
  begin if ((num_cites − 1) ∗ num_fields + crossref_num ≥ max_fields) then
    confusion(´field_info␣index␣is␣out␣of␣range´);
  cite_ptr ← 0;
  while (cite_ptr < num_cites) do
    begin field_ptr ← cite_ptr ∗ num_fields + crossref_num;
    if (field_info[field_ptr] ≠ missing) then
      if (find_cite_locs_for_this_cite_key(field_info[field_ptr])) then
        begin cite_loc ← ilk_info[lc_cite_loc]; field_info[field_ptr] ← hash_text[cite_loc];
        cite_parent_ptr ← ilk_info[cite_loc]; field_ptr ← cite_ptr ∗ num_fields + num_pre_defined_fields;
        field_end_ptr ← field_ptr − num_pre_defined_fields + num_fields;
        field_parent_ptr ← cite_parent_ptr ∗ num_fields + num_pre_defined_fields;
        while (field_ptr < field_end_ptr) do
          begin if (field_info[field_ptr] = missing) then field_info[field_ptr] ← field_info[field_parent_ptr];
          incr(field_ptr); incr(field_parent_ptr);
          end;
        end;
    incr(cite_ptr);
    end;
  end
```

This code is used in section 276.

279*. Here we remove the `crossref` field value for each child whose parent was cross referenced too few times. We also issue any necessary warnings arising from a bad cross reference.

```
⟨ Subtract cross-reference information 279* ⟩ ≡
  begin if ((num_cites − 1) ∗ num_fields + crossref_num ≥ max_fields) then
    confusion(´field_info␣index␣is␣out␣of␣range´);
  cite_ptr ← 0;
  while (cite_ptr < num_cites) do
    begin field_ptr ← cite_ptr ∗ num_fields + crossref_num;
    if (field_info[field_ptr] ≠ missing) then
      if (¬find_cite_locs_for_this_cite_key(field_info[field_ptr])) then
        begin    { the parent is not on cite_list }
        if (cite_hash_found) then hash_cite_confusion;
        nonexistent_cross_reference_error; field_info[field_ptr] ← missing;    { remove the crossref ptr }
        end
      else begin    { the parent exists on cite_list }
        if (cite_loc ≠ ilk_info[lc_cite_loc]) then hash_cite_confusion;
        cite_parent_ptr ← ilk_info[cite_loc];
        if (type_list[cite_parent_ptr] = empty) then
          begin nonexistent_cross_reference_error;
          field_info[field_ptr] ← missing;    { remove the crossref ptr }
          end
        else begin    { the parent exists in the database too }
          field_parent_ptr ← cite_parent_ptr ∗ num_fields + crossref_num;
          if (field_info[field_parent_ptr] ≠ missing) then ⟨ Complain about a nested cross reference 282 ⟩;
          if ((¬all_entries) ∧ (cite_parent_ptr ≥ old_num_cites) ∧ (cite_info[cite_parent_ptr] < min_crossrefs))
              then
            field_info[field_ptr] ← missing;    { remove the crossref ptr }
          end;
        end;
    incr(cite_ptr);
    end;
  end
```

This code is used in section 276.

285*. We have to move to its final resting place all the entry information associated with the exact location in *cite_list* of this cite key.

```
⟨ Slide this cite key down to its permanent spot 285* ⟩ ≡
  begin if ((cite_xptr + 1) ∗ num_fields > max_fields) then
    confusion(´field_info␣index␣is␣out␣of␣range´);
  cite_list[cite_xptr] ← cite_list[cite_ptr]; type_list[cite_xptr] ← type_list[cite_ptr];
  if (¬find_cite_locs_for_this_cite_key(cite_list[cite_ptr])) then cite_key_disappeared_confusion;
  if ((¬cite_hash_found) ∨ (cite_loc ≠ ilk_info[lc_cite_loc])) then hash_cite_confusion;
  ilk_info[cite_loc] ← cite_xptr;
  field_ptr ← cite_xptr ∗ num_fields; field_end_ptr ← field_ptr + num_fields; tmp_ptr ← cite_ptr ∗ num_fields;
  while (field_ptr < field_end_ptr) do
    begin field_info[field_ptr] ← field_info[tmp_ptr]; incr(field_ptr); incr(tmp_ptr);
    end;
  end
```

This code is used in section 283.

287* This module initializes all *int_entry_vars* of all entries to 0, the value to which all integers are initialized.

```
⟨ Initialize the int_entry_vars 287* ⟩ ≡
  begin entry_ints ← XTALLOC((num_ent_ints + 1) * (num_cites + 1), integer); int_ent_ptr ← 0;
  while (int_ent_ptr < num_ent_ints * num_cites) do
    begin entry_ints[int_ent_ptr] ← 0; incr(int_ent_ptr);
    end;
  end
```

This code is used in section 276.

288* This module initializes all *str_entry_vars* of all entries to the null string, the value to which all strings are initialized.

```
⟨ Initialize the str_entry_vars 288* ⟩ ≡
  begin entry_strs ← XTALLOC((num_ent_strs + 1) * (num_cites + 1) * (ent_str_size + 1), ASCII_code);
  str_ent_ptr ← 0;
  while (str_ent_ptr < num_ent_strs * num_cites) do
    begin x_entry_strs(str_ent_ptr)(0) ← end_of_string; incr(str_ent_ptr);
    end;
  end
```

This code is used in section 276.

290* Executing the style file. This part of the program produces the output by executing the `.bst`-file commands `execute`, `iterate`, `reverse`, and `sort`. To do this it uses a stack (consisting of the two arrays *lit_stack* and *lit_stk_type*) for storing literals, a buffer *ex_buf* for manipulating strings, and an array *sorted_cites* for holding pointers to the sorted cite keys (*sorted_cites* is an alias of *cite_info*).

⟨ Globals in the outer block 2* ⟩ +≡
lit_stack: ↑*integer*; { the literal function stack }
lit_stk_type: ↑*stk_type*; { their corresponding types }
lit_stk_ptr: *lit_stk_loc*; { points just above the top of the stack }
cmd_str_ptr: *str_number*; { stores value of *str_ptr* during execution }
ent_chr_ptr: 0 .. *ent_str_size*; { points at a *str_entry_var* character }
glob_chr_ptr: 0 .. *glob_str_size*; { points at a *str_global_var* character }
ex_buf: *buf_type*; { a buffer for manipulating strings }
ex_buf_ptr: *buf_pointer*; { general *ex_buf* location }
ex_buf_length: *buf_pointer*; { the length of the current string in *ex_buf* }
out_buf: *buf_type*; { the `.bbl` output buffer }
out_buf_ptr: *buf_pointer*; { general *out_buf* location }
out_buf_length: *buf_pointer*; { the length of the current string in *out_buf* }
mess_with_entries: *boolean*; { *true* if functions can use entry info }
sort_cite_ptr: *cite_number*; { a loop index for the sorted cite keys }
sort_key_num: *str_ent_loc*; { index for the *str_entry_var* `sort.key$` }
brace_level: *integer*; { the brace nesting depth within a string }

291* Where *lit_stk_loc* is a stack location, and where *stk_type* gives one of the three types of literals (an integer, a string, or a function) or a special marker. If a *lit_stk_type* element is a *stk_int* then the corresponding *lit_stack* element is an integer; if a *stk_str*, then a pointer to a *str_pool* string; and if a *stk_fn*, then a pointer to the function's hash-table location. However, if the literal should have been a *stk_str* that was the value of a field that happened to be *missing*, then the special value *stk_field_missing* goes on the stack instead; its corresponding *lit_stack* element is a pointer to the field-name's string. Finally, *stk_empty* is the type of a literal popped from an empty stack.

define *stk_int* = 0 { an integer literal }
define *stk_str* = 1 { a string literal }
define *stk_fn* = 2 { a function literal }
define *stk_field_missing* = 3 { a special marker: a field value was missing }
define *stk_empty* = 4 { another: the stack was empty when this was popped }
define *last_lit_type* = 4 { the same number as on the line above }

⟨ Types in the outer block 22* ⟩ +≡
lit_stk_loc = *integer*; { the stack range }
stk_type = 0 .. *last_lit_type*; { the literal types }

301* The function *less_than* compares the two `sort.key$`s indirectly pointed to by its arguments and returns *true* if the first argument's `sort.key$` is lexicographically less than the second's (that is, alphabetically earlier). In case of ties the function compares the indices *arg1* and *arg2*, which are assumed to be different, and returns *true* if the first is smaller. This function uses *ASCII_code*s to compare, so it might give "interesting" results when handling nonletters.

```
define compare_return(#) ≡
    begin { the compare is finished }
    less_than ← #; return;
    end
```

⟨ Procedures and functions for handling numbers, characters, and strings 54 ⟩ +≡

```
function less_than(arg1, arg2 : cite_number): boolean;
  label exit;
  var char_ptr: 0 .. ent_str_size; { character index into compared strings }
    ptr1, ptr2: str_ent_loc; { the two sort.key$ pointers }
    char1, char2: ASCII_code; { the two characters being compared }
  begin ptr1 ← arg1 * num_ent_strs + sort_key_num; ptr2 ← arg2 * num_ent_strs + sort_key_num;
  char_ptr ← 0;
  loop
    begin char1 ← x_entry_strs(ptr1)(char_ptr); char2 ← x_entry_strs(ptr2)(char_ptr);
    if (char1 = end_of_string) then
      if (char2 = end_of_string) then
        if (arg1 < arg2) then compare_return(true)
        else if (arg1 > arg2) then compare_return(false)
          else { arg1 = arg2 }
      confusion('Duplicate␣sort␣key')
  else { char2 ≠ end_of_string }
    compare_return(true)
  else { char1 ≠ end_of_string }
    if (char2 = end_of_string) then compare_return(false)
    else if (char1 < char2) then compare_return(true)
      else if (char1 > char2) then compare_return(false);
    incr(char_ptr);
    end;
exit: end;
```

307* Ok, that's it for sorting; now we'll play with the literal stack. This procedure pushes a literal onto the stack, checking for stack overflow.

```
⟨ Procedures and functions for style-file function execution 307* ⟩ ≡
procedure push_lit_stk(push_lt : integer; push_type : stk_type);
  trace
  var dum_ptr: lit_stk_loc;  { used just as an index variable }
  ecart
  begin lit_stack[lit_stk_ptr] ← push_lt; lit_stk_type[lit_stk_ptr] ← push_type;
  trace for dum_ptr ← 0 to lit_stk_ptr do trace_pr('␣␣');
  trace_pr('Pushing␣');
  case (lit_stk_type[lit_stk_ptr]) of
  stk_int: trace_pr_ln(lit_stack[lit_stk_ptr] : 0);
  stk_str: begin trace_pr('"'); trace_pr_pool_str(lit_stack[lit_stk_ptr]); trace_pr_ln('"');
    end;
  stk_fn: begin trace_pr('`'); trace_pr_pool_str(hash_text[lit_stack[lit_stk_ptr]]); trace_pr_ln('''');
    end;
  stk_field_missing: begin trace_pr('missing␣field␣`'); trace_pr_pool_str(lit_stack[lit_stk_ptr]);
    trace_pr_ln('''');
    end;
  stk_empty: trace_pr_ln('a␣bad␣literal--popped␣from␣an␣empty␣stack');
  othercases unknwn_literal_confusion
  endcases;
  ecart
  if (lit_stk_ptr = lit_stk_size) then
    begin BIB_XRETALLOC_NOSET('lit_stack', lit_stack, integer, lit_stk_size,
        lit_stk_size + LIT_STK_SIZE);
    BIB_XRETALLOC('lit_stk_type', lit_stk_type, stk_type, lit_stk_size, lit_stk_size + LIT_STK_SIZE);
    end;
  incr(lit_stk_ptr);
  end;
```

See also sections 309, 312, 314, 315, 316, 317, 318, 320, 322*, and 342.

This code is used in section 12.

322* This procedure adds to the output buffer the given string in *str_pool*. It assumes the global variable *out_buf_length* gives the length of the current string in *out_buf*, and thus also gives the location for the next character. If there are enough characters present in the output buffer, it writes one or more lines out to the `.bbl` file. It breaks a line only at a *white_space* character, and when it does, it adds two *space*s to the next output line.

```
⟨ Procedures and functions for style-file function execution 307* ⟩ +≡
procedure add_out_pool(p_str : str_number);
  label loop1_exit, loop2_exit;
  var break_ptr: buf_pointer;  { the first character following the line break }
    end_ptr: buf_pointer;  { temporary end-of-buffer pointer }
    break_pt_found: boolean;  { a suitable white_space character }
    unbreakable_tail: boolean;  { as it contains no white_space character }
  begin p_ptr1 ← str_start[p_str]; p_ptr2 ← str_start[p_str + 1];
  while (out_buf_length + (p_ptr2 − p_ptr1) > buf_size) do buffer_overflow;
  out_buf_ptr ← out_buf_length;
  while (p_ptr1 < p_ptr2) do
    begin  { copy characters into the buffer }
    out_buf[out_buf_ptr] ← str_pool[p_ptr1]; incr(p_ptr1); incr(out_buf_ptr);
    end;
  out_buf_length ← out_buf_ptr; unbreakable_tail ← false;
  while ((out_buf_length > max_print_line) ∧ (¬unbreakable_tail)) do ⟨ Break that line 323 ⟩;
  end;
```

327* This module pushes the string given by the field onto the literal stack unless it's *missing*, in which case it pushes a special value onto the stack.

```
⟨ Execute a field 327* ⟩ ≡
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else begin field_ptr ← cite_ptr * num_fields + fn_info[ex_fn_loc];
    if (field_ptr ≥ max_fields) then confusion(´field␣info␣index␣is␣out␣of␣range´);
    if (field_info[field_ptr] = missing) then push_lit_stk(hash_text[ex_fn_loc], stk_field_missing)
    else push_lit_stk(field_info[field_ptr], stk_str);
    end
  end
```

This code is used in section 325.

329* This module adds the string given by a *str_entry_var* to *str_pool* via the execution buffer and pushes it onto the literal stack.

```
⟨ Execute a str_entry_var 329* ⟩ ≡
  begin if (¬mess_with_entries) then bst_cant_mess_with_entries_print
  else begin str_ent_ptr ← cite_ptr * num_ent_strs + fn_info[ex_fn_loc];
    ex_buf_ptr ← 0;  { also serves as ent_chr_ptr }
    while (x_entry_strs(str_ent_ptr)(ex_buf_ptr) ≠ end_of_string) do  { copy characters into the buffer }
      append_ex_buf_char(x_entry_strs(str_ent_ptr)(ex_buf_ptr));
    ex_buf_length ← ex_buf_ptr; add_pool_buf_and_push;  { push this string onto the stack }
    end;
  end
```

This code is used in section 325.

330*. This module pushes the string given by a *str_global_var* onto the literal stack, but it copies the string to *str_pool* (character by character) only if it has to—it *doesn't* have to if the string is static (that is, if the string isn't at the top, temporary part of the string pool).

```
⟨ Execute a str_global_var 330* ⟩ ≡
  begin str_glb_ptr ← fn_info[ex_fn_loc];
  if (glb_str_ptr[str_glb_ptr] > 0) then { we're dealing with a static string }
    push_lit_stk(glb_str_ptr[str_glb_ptr], stk_str)
  else begin str_room(glb_str_end[str_glb_ptr]); glob_chr_ptr ← 0;
    while (glob_chr_ptr < glb_str_end[str_glb_ptr]) do { copy the string }
      begin append_char(x_global_strs(str_glb_ptr)(glob_chr_ptr)); incr(glob_chr_ptr);
      end;
    push_lit_stk(make_string, stk_str); { and push it onto the stack }
    end;
  end
```

This code is used in section 325.

334* It's time for us to insert more pre-defined strings into *str_pool* (and thus the hash table) and to insert the *built_in* functions into the hash table. The strings corresponding to these functions should contain no upper-case letters, and they must all be exactly *longest_pds* characters long. The *build_in* routine (to appear shortly) does the work.

Important note: These pre-definitions must not have any glitches or the program may bomb because the *log_file* hasn't been opened yet.

⟨ Pre-define certain strings 75 ⟩ +≡
build_in(´=␣␣␣␣␣␣␣␣␣␣´, 1, *b_equals*, *n_equals*);
build_in(´>␣␣␣␣␣␣␣␣␣␣´, 1, *b_greater_than*, *n_greater_than*);
build_in(´<␣␣␣␣␣␣␣␣␣␣´, 1, *b_less_than*, *n_less_than*); *build_in*(´+␣␣␣␣␣␣␣␣␣␣´, 1, *b_plus*, *n_plus*);
build_in(´-␣␣␣␣␣␣␣␣␣␣´, 1, *b_minus*, *n_minus*);
build_in(´*␣␣␣␣␣␣␣␣␣␣´, 1, *b_concatenate*, *n_concatenate*); *build_in*(´:=␣␣␣␣␣␣␣␣␣␣´, 2, *b_gets*, *n_gets*);
build_in(´add.period$␣´, 11, *b_add_period*, *n_add_period*);
build_in(´call.type$␣␣´, 10, *b_call_type*, *n_call_type*);
build_in(´change.case$´, 12, *b_change_case*, *n_change_case*);
build_in(´chr.to.int$␣´, 11, *b_chr_to_int*, *n_chr_to_int*); *build_in*(´cite$␣␣␣␣␣␣␣´, 5, *b_cite*, *n_cite*);
build_in(´duplicate$␣␣´, 10, *b_duplicate*, *n_duplicate*); *build_in*(´empty$␣␣␣␣␣␣´, 6, *b_empty*, *n_empty*);
build_in(´format.name$´, 12, *b_format_name*, *n_format_name*); *build_in*(´if$␣␣␣␣␣␣␣␣␣´, 3, *b_if*, *n_if*);
build_in(´int.to.chr$␣´, 11, *b_int_to_chr*, *n_int_to_chr*);
build_in(´int.to.str$␣´, 11, *b_int_to_str*, *n_int_to_str*);
build_in(´missing$␣␣␣␣´, 8, *b_missing*, *n_missing*); *build_in*(´newline$␣␣␣␣´, 8, *b_newline*, *n_newline*);
build_in(´num.names$␣␣´, 10, *b_num_names*, *n_num_names*); *build_in*(´pop$␣␣␣␣␣␣␣␣´, 4, *b_pop*, *n_pop*);
build_in(´preamble$␣␣␣´, 9, *b_preamble*, *n_preamble*); *build_in*(´purify$␣␣␣␣␣´, 7, *b_purify*, *n_purify*);
build_in(´quote$␣␣␣␣␣␣´, 6, *b_quote*, *n_quote*); *build_in*(´skip$␣␣␣␣␣␣␣´, 5, *b_skip*, *n_skip*);
build_in(´stack$␣␣␣␣␣␣´, 6, *b_stack*, *n_stack*); *build_in*(´substring$␣␣´, 10, *b_substring*, *n_substring*);
build_in(´swap$␣␣␣␣␣␣␣´, 5, *b_swap*, *n_swap*); *build_in*(´text.length$´, 12, *b_text_length*, *n_text_length*);
build_in(´text.prefix$´, 12, *b_text_prefix*, *n_text_prefix*);
build_in(´top$␣␣␣␣␣␣␣␣´, 4, *b_top_stack*, *n_top_stack*); *build_in*(´type$␣␣␣␣␣␣␣´, 5, *b_type*, *n_type*);
build_in(´warning$␣␣␣␣´, 8, *b_warning*, *n_warning*); *build_in*(´while$␣␣␣␣␣´, 6, *b_while*, *n_while*);
build_in(´width$␣␣␣␣␣´, 6, *b_width*, *n_width*); *build_in*(´write$␣␣␣␣␣´, 6, *b_write*, *n_write*);

337* These variables all begin with *s_* and specify the locations in *str_pool* of certain often-used strings that the `.bst` commands need. The *s_preamble* array is big enough to allow an average of one `preamble$` command per `.bib` file.

⟨ Globals in the outer block 2* ⟩ +≡
s_null: *str_number*; { the null string }
s_default: *str_number*; { `default.type`, for unknown entry types }
s_t: *str_number*; { `t`, for *title_lowers* case conversion }
s_l: *str_number*; { `l`, for *all_lowers* case conversion }
s_u: *str_number*; { `u`, for *all_uppers* case conversion }
s_preamble: ↑*str_number*; { for the `preamble$` *built_in* function }

344* These are nonrecursive variables that *execute_fn* uses. Declaring them here (instead of in the previous module) saves execution time and stack space on most machines.

define *name_buf* ≡ *sv_buffer* { an alias, a buffer for manipulating names }

⟨ Globals in the outer block 2* ⟩ +≡
pop_lit1, *pop_lit2*, *pop_lit3*: *integer*; { stack literals }
pop_typ1, *pop_typ2*, *pop_typ3*: *stk_type*; { stack types }
sp_ptr: *pool_pointer*; { for manipulating *str_pool* strings }
sp_xptr1, *sp_xptr2*: *pool_pointer*; { more of the same }
sp_end: *pool_pointer*; { marks the end of a *str_pool* string }
sp_length, *sp2_length*: *pool_pointer*; { lengths of *str_pool* strings }
sp_brace_level: *integer*; { for scanning *str_pool* strings }
ex_buf_xptr, *ex_buf_yptr*: *buf_pointer*; { extra *ex_buf* locations }
control_seq_loc: *hash_loc*; { hash-table loc of a control sequence }
preceding_white: *boolean*; { used in scanning strings }
and_found: *boolean*; { to stop the loop that looks for an "and" }
num_names: *integer*; { for counting names }
name_bf_ptr: *buf_pointer*; { general *name_buf* location }
name_bf_xptr, *name_bf_yptr*: *buf_pointer*; { and two more }
nm_brace_level: *integer*; { for scanning *name_buf* strings }
name_tok: ↑*buf_pointer*; { name-token ptr list }
name_sep_char: ↑*ASCII_code*; { token-ending chars }
num_tokens: *buf_pointer*; { this counts name tokens }
token_starting: *boolean*; { used in scanning name tokens }
alpha_found: *boolean*; { used in scanning the format string }
double_letter, *end_of_group*, *to_be_written*: *boolean*; { the same }
first_start: *buf_pointer*; { start-ptr into *name_tok* for the first name }
first_end: *buf_pointer*; { end-ptr into *name_tok* for the first name }
last_end: *buf_pointer*; { end-ptr into *name_tok* for the last name }
von_start: *buf_pointer*; { start-ptr into *name_tok* for the von name }
von_end: *buf_pointer*; { end-ptr into *name_tok* for the von name }
jr_end: *buf_pointer*; { end-ptr into *name_tok* for the jr name }
cur_token, *last_token*: *buf_pointer*; { *name_tok* ptrs for outputting tokens }
use_default: *boolean*; { for the inter-token intra-name part string }
num_commas: *buf_pointer*; { used to determine the name syntax }
comma1, *comma2*: *buf_pointer*; { ptrs into *name_tok* }
num_text_chars: *buf_pointer*; { special characters count as one }

357* This module checks that what we're about to assign is really a string, and then assigns.

```
⟨ Assign to a str_entry_var 357* ⟩ ≡
  begin if (pop_typ2 ≠ stk_str) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str)
  else begin str_ent_ptr ← cite_ptr * num_ent_strs + fn_info[pop_lit1]; ent_chr_ptr ← 0;
    sp_ptr ← str_start[pop_lit2]; sp_xptr1 ← str_start[pop_lit2 + 1];
    if (sp_xptr1 − sp_ptr > ent_str_size) then
      begin bst_string_size_exceeded(ent_str_size : 0, ´,␣the␣entry´); sp_xptr1 ← sp_ptr + ent_str_size;
      end;
    while (sp_ptr < sp_xptr1) do
      begin { copy characters into entry_strs }
      x_entry_strs(str_ent_ptr)(ent_chr_ptr) ← str_pool[sp_ptr]; incr(ent_chr_ptr); incr(sp_ptr);
      end;
    x_entry_strs(str_ent_ptr)(ent_chr_ptr) ← end_of_string;
    end
  end
```

This code is used in section 354.

359* This module checks that what we're about to assign is really a string, and then assigns.

```
⟨ Assign to a str_global_var 359* ⟩ ≡
  begin if (pop_typ2 ≠ stk_str) then print_wrong_stk_lit(pop_lit2, pop_typ2, stk_str)
  else begin str_glb_ptr ← fn_info[pop_lit1];
    if (pop_lit2 < cmd_str_ptr) then glb_str_ptr[str_glb_ptr] ← pop_lit2
    else begin glb_str_ptr[str_glb_ptr] ← 0; glob_chr_ptr ← 0; sp_ptr ← str_start[pop_lit2];
      sp_end ← str_start[pop_lit2 + 1];
      if (sp_end − sp_ptr > glob_str_size) then
        begin bst_string_size_exceeded(glob_str_size : 0, ´,␣the␣global´); sp_end ← sp_ptr + glob_str_size;
        end;
      while (sp_ptr < sp_end) do
        begin { copy characters into global_strs }
        x_global_strs(str_glb_ptr)(glob_chr_ptr) ← str_pool[sp_ptr]; incr(glob_chr_ptr); incr(sp_ptr);
        end;
      glb_str_end[str_glb_ptr] ← glob_chr_ptr;
      end;
    end
  end
```

This code is used in section 354.

388* This module removes all leading *white_space* (and *sep_char*s), and trailing *white_space* (and *sep_char*s) and *comma*s. It complains for each trailing *comma*.

```
⟨ Remove leading and trailing junk, complaining if necessary 388* ⟩ ≡
  begin while (ex_buf_ptr > ex_buf_xptr) do { now remove trailing stuff }
    case (lex_class[ex_buf[ex_buf_ptr − 1]]) of
    white_space, sep_char: decr(ex_buf_ptr);
    othercases if (ex_buf[ex_buf_ptr − 1] = comma) then
        begin print(´Name␣´, pop_lit2 : 0, ´␣in␣"´); print_pool_str(pop_lit3);
        print(´"␣has␣a␣comma␣at␣the␣end´); bst_ex_warn_print; decr(ex_buf_ptr);
        end
      else goto loop1_exit
    endcases;
loop1_exit: end
```

This code is used in section 387.

438*. This module finds the substring as described in the last section, and slides it into place in the string pool, if necessary.

```
⟨ Form the appropriate substring 438* ⟩ ≡
  begin if (pop_lit2 > 0) then
    begin if (pop_lit1 > sp_length − (pop_lit2 − 1)) then pop_lit1 ← sp_length − (pop_lit2 − 1);
    sp_ptr ← str_start[pop_lit3] + (pop_lit2 − 1); sp_end ← sp_ptr + pop_lit1;
    if (pop_lit2 = 1) then
      if (pop_lit3 ≥ cmd_str_ptr) then { no shifting—merely change pointers }
        begin str_start[pop_lit3 + 1] ← sp_end; unflush_string; incr(lit_stk_ptr); return;
        end;
    end
  else { −ex_buf_length ≤ pop_lit2 < 0 }
  begin pop_lit2 ← −pop_lit2;
  if (pop_lit1 > sp_length − (pop_lit2 − 1)) then pop_lit1 ← sp_length − (pop_lit2 − 1);
  sp_end ← str_start[pop_lit3 + 1] − (pop_lit2 − 1); sp_ptr ← sp_end − pop_lit1;
  end; str_room(sp_end − sp_ptr);
  while (sp_ptr < sp_end) do { shift the substring }
    begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
    end;
  push_lit_stk(make_string, stk_str); { and push it onto the stack }
  end
```

This code is used in section 437.

444*. This module finds the prefix as described in the last section, and appends any needed matching *right_brace*s.

```
⟨ Form the appropriate prefix 444* ⟩ ≡
  begin sp_ptr ← str_start[pop_lit2]; sp_end ← str_start[pop_lit2 + 1]; { this may change }
  ⟨ Scan the appropriate number of characters 445 ⟩;
  str_room(sp_brace_level + sp_end − sp_ptr);
  if (pop_lit2 ≥ cmd_str_ptr) then { no shifting—merely change pointers }
    pool_ptr ← sp_end
  else while (sp_ptr < sp_end) do { shift the substring }
      begin append_char(str_pool[sp_ptr]); incr(sp_ptr);
      end;
  while (sp_brace_level > 0) do { add matching right_braces }
    begin append_char(right_brace); decr(sp_brace_level);
    end;
  push_lit_stk(make_string, stk_str); { and push it onto the stack }
  end
```

This code is used in section 443.

459*. This prints information gathered while reading the `.bst` and `.bib` files.

```
⟨ Print entry information 459* ⟩ ≡
  begin trace_pr(´␣entry-type␣´);
  if (type_list[cite_ptr] = undefined) then trace_pr(´unknown´)
  else if (type_list[cite_ptr] = empty) then trace_pr(´---␣no␣type␣found´)
    else trace_pr_pool_str(hash_text[type_list[cite_ptr]]);
  trace_pr_ln(´␣has␣entry␣strings´); ⟨ Print entry strings 460* ⟩;
  trace_pr(´␣␣has␣entry␣integers´); ⟨ Print entry integers 461 ⟩;
  trace_pr_ln(´␣␣and␣has␣fields´); ⟨ Print fields 462* ⟩;
  end
```

This code is used in section 458.

460*. This prints, for the current entry, the strings declared by the `entry` command.

```
⟨ Print entry strings 460* ⟩ ≡
  begin if (num_ent_strs = 0) then trace_pr_ln(´␣␣␣␣undefined´)
  else if (¬read_completed) then trace_pr_ln(´␣␣␣␣uninitialized´)
    else begin str_ent_ptr ← cite_ptr * num_ent_strs;
      while (str_ent_ptr < (cite_ptr + 1) * num_ent_strs) do
        begin ent_chr_ptr ← 0; trace_pr(´␣␣␣␣"´);
        while (x_entry_strs(str_ent_ptr)(ent_chr_ptr) ≠ end_of_string) do
          begin trace_pr(xchr[x_entry_strs(str_ent_ptr)(ent_chr_ptr)]); incr(ent_chr_ptr);
          end;
        trace_pr_ln(´"´); incr(str_ent_ptr);
        end;
      end;
  end
```

This code is used in section 459*.

462*. This prints the fields stored for the current entry.

```
⟨ Print fields 462* ⟩ ≡
  begin if (¬read_performed) then trace_pr_ln(´␣␣␣␣uninitialized´)
  else begin field_ptr ← cite_ptr * num_fields; field_end_ptr ← field_ptr + num_fields;
    if (field_end_ptr > max_fields) then confusion(´field_info␣index␣is␣out␣of␣range´);
    no_fields ← true;
    while (field_ptr < field_end_ptr) do
      begin if (field_info[field_ptr] ≠ missing) then
        begin trace_pr(´␣␣␣␣"´); trace_pr_pool_str(field_info[field_ptr]); trace_pr_ln(´"´);
        no_fields ← false;
        end;
      incr(field_ptr);
      end;
    if (no_fields) then trace_pr_ln(´␣␣␣␣missing´);
    end;
  end
```

This code is used in section 459*.

467* System-dependent changes.

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨ Define parse_arguments 467* ⟩ ≡
procedure parse_arguments;
  const n_options = 4;  { Pascal won't count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
    getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
  begin ⟨ Initialize the option variables 470* ⟩;
  ⟨ Define the option table 468* ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;  { End of arguments; we exit the loop below. }
      end
    else if getopt_return_val = "?" then
        begin usage(my_name);
        end
      else if argument_is(´min-crossrefs´) then
          begin min_crossrefs ← atoi(optarg);
          end
        else if argument_is(´help´) then
            begin usage_help(BIBTEX_HELP, nil);
            end
          else if argument_is(´version´) then
              begin print_version_and_exit(banner, ´Oren␣Patashnik´, nil, nil);
              end;  { Else it was a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. We
        must have one remaining argument. }
  if (optind + 1 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣exactly␣one␣file␣argument.´); usage(my_name);
    end;
  end;
```

This code is used in section 10*.

468* Here is the first of the options we allow.

```
⟨ Define the option table 468* ⟩ ≡
  current_option ← 0; long_options[0].name ← ´terse´; long_options[0].has_arg ← 0;
  long_options[0].flag ← address_of(verbose); long_options[0].val ← 0; incr(current_option);
```

See also sections 471*, 474*, 475*, and 476*.

This code is used in section 467*.

469* The global variable *verbose* determines whether or not we print progress information.

```
⟨ Globals in the outer block 2* ⟩ +≡
verbose: c_int_type;
```

470* Start off *true*, to match the default behavior.

```
⟨ Initialize the option variables 470* ⟩ ≡
  verbose ← true;
```

See also section 473*.

This code is used in section 467*.

471*. Here is an option to change the minimum number of cross-refs required for automatic *cite_list* inclusion.

```
⟨ Define the option table 468* ⟩ +≡
  long_options[current_option].name ← ´min-crossrefs´; long_options[current_option].has_arg ← 1;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

472*. ⟨ Globals in the outer block 2* ⟩ +≡

```
min_crossrefs: integer;
```

473*. Set *min_crossrefs* to two by default, so we match the documentation (`btxdoc.tex`).

```
⟨ Initialize the option variables 470* ⟩ +≡
  min_crossrefs ← 2;
```

474*. One of the standard options.

```
⟨ Define the option table 468* ⟩ +≡
  long_options[current_option].name ← ´help´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

475*. Another of the standard options.

```
⟨ Define the option table 468* ⟩ +≡
  long_options[current_option].name ← ´version´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

476*. An element with all zeros always ends the list.

```
⟨ Define the option table 468* ⟩ +≡
  long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```

477*. Determine *ent_str_size*, *glob_str_size*, and *max_strings* from the environment, configuration file, or default value. Set *hash_size* ← *max_strings*, but not less than *HASH_SIZE*.

setup_bound_var stuff adapted from `tex.ch`.

```
  define setup_bound_var(#) ≡ bound_default ← #; setup_bound_var_end
  define setup_bound_var_end(#) ≡ bound_name ← #; setup_bound_var_end_end
  define setup_bound_var_end_end(#) ≡ setup_bound_variable(address_of(#), bound_name, bound_default);
      if # < bound_default then # ← bound_default

⟨ Procedures and functions for about everything 12 ⟩ +≡
procedure setup_params;
  var bound_default: integer; { for setup }
    bound_name: const_cstring; { for setup }
  begin kpse_set_program_name(argv[0], ´bibtex´);
  setup_bound_var(ENT_STR_SIZE)(´ent_str_size´)(ent_str_size);
  setup_bound_var(GLOB_STR_SIZE)(´glob_str_size´)(glob_str_size);
  setup_bound_var(MAX_STRINGS)(´max_strings´)(max_strings);
  hash_size ← max_strings;
  if hash_size < HASH_SIZE then hash_size ← HASH_SIZE;
  hash_max ← hash_size + hash_base − 1; end_of_def ← hash_max + 1; undefined ← hash_max + 1;
  end;
```

478*. We use the algorithm from Knuth's `primes.web` to compute *hash_prime* as the smallest prime number not less than 85% of *hash_size* (and ≥ 128).

```
define primes ≡ hash_next   { array holding the first k primes }
define mult ≡ hash_text   { array holding odd multiples of the first o primes }
⟨ Procedures and functions for about everything 12 ⟩ +≡
procedure compute_hash_prime;
  var hash_want: integer;   { 85% of hash_size }
    k: integer;   { number of prime numbers p_i in primes }
    j: integer;   { a prime number candidate }
    o: integer;   { number of odd multiples of primes in mult }
    square: integer;   { p_o^2 }
    n: integer;   { loop index }
    j_prime: boolean;   { is j a prime? }
  begin hash_want ← (hash_size div 20) * 17; j ← 1; k ← 1; hash_prime ← 2; primes[k] ← hash_prime;
  o ← 2; square ← 9;
  while hash_prime < hash_want do
    begin repeat j ← j + 2;
      if j = square then
        begin mult[o] ← j; j ← j + 2; incr(o); square ← primes[o] * primes[o];
        end;
      n ← 2; j_prime ← true;
      while (n < o) ∧ j_prime do
        begin while mult[n] < j do mult[n] ← mult[n] + 2 * primes[n];
        if mult[n] = j then j_prime ← false;
        incr(n);
        end;
    until j_prime;
    incr(k); hash_prime ← j; primes[k] ← hash_prime;
    end;
  end;
```

479* Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. All references are to section numbers instead of page numbers.

This index also lists a few error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing TeX in a new operating environment. A list of various things that can't happen appears under "this can't happen".

The following sections were changed by the change file: 1, 2, 3, 4, 10, 13, 14, 15, 16, 17, 22, 23, 27, 28, 32, 33, 37, 38, 39, 41, 42, 46, 47, 48, 49, 50, 53, 58, 59, 60, 61, 64, 65, 68, 70, 71, 73, 77, 97, 100, 101, 102, 106, 108, 110, 117, 118, 121, 123, 127, 128, 129, 130, 138, 141, 151, 160, 161, 187, 188, 198, 200, 216, 219, 223, 226, 242, 251, 263, 265, 277, 279, 285, 287, 288, 290, 291, 301, 307, 322, 327, 329, 330, 334, 337, 344, 357, 359, 388, 438, 444, 459, 460, 462, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479.

⟨ Do a full brace-balanced scan 256 ⟩ Used in section 253.
⟨ Do a full scan with *bib_brace_level* > 0 257 ⟩ Used in section 256.
⟨ Do a quick brace-balanced scan 254 ⟩ Used in section 253.
⟨ Do a quick scan with *bib_brace_level* > 0 255 ⟩ Used in section 254.
⟨ Do a straight insertion sort 304 ⟩ Used in section 303.
⟨ Do the partitioning and the recursive calls 306 ⟩ Used in section 303.
⟨ Draw out the median-of-three partition element 305 ⟩ Used in section 303.
⟨ Execute a field 327* ⟩ Used in section 325.
⟨ Execute a *built_in* function 341 ⟩ Used in section 325.
⟨ Execute a *str_entry_var* 329* ⟩ Used in section 325.
⟨ Execute a *str_global_var* 330* ⟩ Used in section 325.
⟨ Execute a *wiz_defined* function 326 ⟩ Used in section 325.
⟨ Execute an *int_entry_var* 328 ⟩ Used in section 325.
⟨ Figure out how to output the name tokens, and do it 412 ⟩ Used in section 411.
⟨ Figure out the formatted name 402 ⟩ Used in section 420.
⟨ Figure out what this letter means 405 ⟩ Used in section 403.
⟨ Figure out what tokens we'll output for the 'first' name 407 ⟩ Used in section 405.
⟨ Figure out what tokens we'll output for the 'jr' name 410 ⟩ Used in section 405.
⟨ Figure out what tokens we'll output for the 'last' name 409 ⟩ Used in section 405.
⟨ Figure out what tokens we'll output for the 'von' name 408 ⟩ Used in section 405.
⟨ Final initialization for `.bib` processing 224 ⟩ Used in section 223*.
⟨ Final initialization for processing the entries 276 ⟩ Used in section 223*.
⟨ Finally format this part of the name 411 ⟩ Used in section 403.
⟨ Finally output a full token 414 ⟩ Used in section 413.
⟨ Finally output a special character and exit loop 416 ⟩ Used in section 415.
⟨ Finally output an abbreviated token 415 ⟩ Used in section 413.
⟨ Finally output the inter-token string 417 ⟩ Used in section 413.
⟨ Finally output the name tokens 413 ⟩ Used in section 412.
⟨ Find the lower-case equivalent of the *cite_info* key 270 ⟩ Used in section 268.
⟨ Find the parts of the name 395 ⟩ Used in section 382.
⟨ Form the appropriate prefix 444* ⟩ Used in section 443.
⟨ Form the appropriate substring 438* ⟩ Used in section 437.
⟨ Format this part of the name 403 ⟩ Used in section 402.
⟨ Get the next field name 275 ⟩ Used in section 274.
⟨ Get the next function of the definition 189 ⟩ Used in section 187*.
⟨ Globals in the outer block 2*, 16*, 19, 24, 30, 34, 37*, 41*, 43, 48*, 65*, 74, 76, 78, 80, 89, 91, 97*, 104, 117*, 124, 129*, 147, 161*, 163, 195, 219*, 247, 290*, 331, 337*, 344*, 365, 469*, 472* ⟩ Used in section 10*.
⟨ Handle a discretionary *tie* 419 ⟩ Used in section 411.
⟨ Handle this `.aux` name 103 ⟩ Used in section 100*.
⟨ Handle this accented or foreign character (and **return**) 399 ⟩ Used in section 398.
⟨ Initialize the option variables 470*, 473* ⟩ Used in section 467*.
⟨ Initialize the *field_info* 225 ⟩ Used in section 224.
⟨ Initialize the *int_entry_var*s 287* ⟩ Used in section 276.
⟨ Initialize the *sorted_cites* 289 ⟩ Used in section 276.
⟨ Initialize the *str_entry_var*s 288* ⟩ Used in section 276.
⟨ Initialize things for the *cite_list* 227 ⟩ Used in section 224.
⟨ Insert a *field* into the hash table 172 ⟩ Used in section 171.
⟨ Insert a *str_entry_var* into the hash table 176 ⟩ Used in section 175.
⟨ Insert a *str_global_var* into the hash table 216* ⟩ Used in section 215.
⟨ Insert an *int_entry_var* into the hash table 174 ⟩ Used in section 173.
⟨ Insert an *int_global_var* into the hash table 202 ⟩ Used in section 201.
⟨ Insert pair into hash table and make *p* point to it 71* ⟩ Used in section 68*.

⟨ The scanning function *scan_balanced_braces* 253 ⟩ Used in section 248.
⟨ Types in the outer block 22*, 31, 36, 42*, 49*, 64*, 73*, 105, 118*, 130*, 160*, 291*, 332 ⟩ Used in section 10*.
⟨ *execute_fn* itself 325 ⟩ Used in section 342.
⟨ *execute_fn*(`*`) 350 ⟩ Used in section 342.
⟨ *execute_fn*(`+`) 348 ⟩ Used in section 342.
⟨ *execute_fn*(`-`) 349 ⟩ Used in section 342.
⟨ *execute_fn*(`:=`) 354 ⟩ Used in section 342.
⟨ *execute_fn*(`<`) 347 ⟩ Used in section 342.
⟨ *execute_fn*(`=`) 345 ⟩ Used in section 342.
⟨ *execute_fn*(`>`) 346 ⟩ Used in section 342.
⟨ *execute_fn*(`add.period$`) 360 ⟩ Used in section 342.
⟨ *execute_fn*(`call.type$`) 363 ⟩ Used in section 341.
⟨ *execute_fn*(`change.case$`) 364 ⟩ Used in section 342.
⟨ *execute_fn*(`chr.to.int$`) 377 ⟩ Used in section 342.
⟨ *execute_fn*(`cite$`) 378 ⟩ Used in section 342.
⟨ *execute_fn*(`duplicate$`) 379 ⟩ Used in section 342.
⟨ *execute_fn*(`empty$`) 380 ⟩ Used in section 342.
⟨ *execute_fn*(`format.name$`) 382 ⟩ Used in section 342.
⟨ *execute_fn*(`if$`) 421 ⟩ Used in section 341.
⟨ *execute_fn*(`int.to.chr$`) 422 ⟩ Used in section 342.
⟨ *execute_fn*(`int.to.str$`) 423 ⟩ Used in section 342.
⟨ *execute_fn*(`missing$`) 424 ⟩ Used in section 342.
⟨ *execute_fn*(`newline$`) 425 ⟩ Used in section 341.
⟨ *execute_fn*(`num.names$`) 426 ⟩ Used in section 342.
⟨ *execute_fn*(`pop$`) 428 ⟩ Used in section 341.
⟨ *execute_fn*(`preamble$`) 429 ⟩ Used in section 342.
⟨ *execute_fn*(`purify$`) 430 ⟩ Used in section 342.
⟨ *execute_fn*(`quote$`) 434 ⟩ Used in section 342.
⟨ *execute_fn*(`skip$`) 435 ⟩ Used in section 341.
⟨ *execute_fn*(`stack$`) 436 ⟩ Used in section 341.
⟨ *execute_fn*(`substring$`) 437 ⟩ Used in section 342.
⟨ *execute_fn*(`swap$`) 439 ⟩ Used in section 342.
⟨ *execute_fn*(`text.length$`) 441 ⟩ Used in section 342.
⟨ *execute_fn*(`text.prefix$`) 443 ⟩ Used in section 342.
⟨ *execute_fn*(`top$`) 446 ⟩ Used in section 341.
⟨ *execute_fn*(`type$`) 447 ⟩ Used in section 342.
⟨ *execute_fn*(`warning$`) 448 ⟩ Used in section 342.
⟨ *execute_fn*(`while$`) 449 ⟩ Used in section 341.
⟨ *execute_fn*(`width$`) 450 ⟩ Used in section 342.
⟨ *execute_fn*(`write$`) 454 ⟩ Used in section 342.