The \LaTeX\ preprocessor

(Version 0.99d—June 11, 2023)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>The main program</td>
<td>10</td>
</tr>
<tr>
<td>The character set</td>
<td>21</td>
</tr>
<tr>
<td>Input and output</td>
<td>36</td>
</tr>
<tr>
<td>String handling</td>
<td>48</td>
</tr>
<tr>
<td>The hash table</td>
<td>64</td>
</tr>
<tr>
<td>Scanning an input line</td>
<td>80</td>
</tr>
<tr>
<td>Getting the top-level auxiliary file name</td>
<td>97</td>
</tr>
<tr>
<td>Reading the auxiliary file(s)</td>
<td>109</td>
</tr>
<tr>
<td>Reading the style file</td>
<td>146</td>
</tr>
<tr>
<td>Style-file commands</td>
<td>163</td>
</tr>
<tr>
<td>Reading the database file(s)</td>
<td>218</td>
</tr>
<tr>
<td>Executing the style file</td>
<td>290</td>
</tr>
<tr>
<td>The built-in functions</td>
<td>331</td>
</tr>
<tr>
<td>Cleaning up</td>
<td>455</td>
</tr>
<tr>
<td>System-dependent changes</td>
<td>467</td>
</tr>
<tr>
<td>Index</td>
<td>479</td>
</tr>
</tbody>
</table>
1* Introduction. \texttt{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 \texttt{.bbl} file that a user can edit to add any finishing touches \texttt{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 \texttt{BibTeX} works. It takes as input (a) an \texttt{.aux} file produced by \LaTeX on an earlier run; (b) a \texttt{.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 \texttt{BibTeX} documentation—see the file \texttt{btxdoc.tex}; and (c) \texttt{.bib} file(s) constituting a database of all reference-list entries the user might ever hope to use. \texttt{BibTeX} chooses from the \texttt{.bib} file(s) only those entries specified by the \texttt{.aux} file (that is, those given by \LaTeX’s \texttt{\cite} or \texttt{\nocite} commands), and creates as output a \texttt{.bbl} file containing these entries together with the formatting commands specified by the \texttt{.bst} file (\texttt{BibTeX} also creates a \texttt{.blg} log file, which includes any error or warning messages, but this file isn’t used by any program). \LaTeX will use the \texttt{.bbl} file, perhaps edited by the user, to produce the reference list.

Many modules of \texttt{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 \texttt{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 \texttt{banner} string defined here should be changed whenever \texttt{BibTeX} gets modified.

\begin{verbatim}
define my_name ≡ `bibtex'
define banner ≡ `This is BibTeX, Version 0.99d'  \{ printed when the program starts \}
\end{verbatim}

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

\begin{verbatim}
define term_out ≡ standard_output
define term_in ≡ standard_input
\end{verbatim}

(\texttt{Globals in the outer block 2*} ≡ \texttt{standard_input, standard_output: text;})


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_`.

```plaintext
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;
```


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 Bib\TeX is being installed or when system wizards are fooling around with Bib\TeX 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 Bib\TeX’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.

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

define stat ≡ ifdef(``NO_BIBTEX_STAT``)
define tats ≡ endif(``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.

```c
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; litstk_size ← LIT_STK_SIZE;
\{Process a possible command line 102*\} 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);
    gb_str_ptr ← XTALLOC(max_glob_strs, str_number);
    global_strs ← XTALLOC(max_glob_strs * (glob_str_size + 1), ASCII_code);
    gb_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(litstk_size + 1, integer); litstk_type ← XTALLOC(litstk_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_in(version_string); end;
log_pr_in("Capacity:\max_strings=\max_strings : 1, \hash_size=\hash_size : 1, \hash_prime=\hash_prime : 1); (Read the .aux file 110*);
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_in(term_out, bad : 0, ´is_a_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 \LaTeX{}’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 = 60; { 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*.
These parameters can also be changed at compile time, but they’re needed to define some \TeX\ numeric macros so they must be so defined themselves.

\begin{verbatim}
define HASH_SIZE = 5000  \{ minimum value for hash_size \}
define file_name_size \equiv \text{maxint}  \{ file names have no arbitrary maximum length \}
    \{ For dynamic allocation. \}
define x_entry_strs_tail(#) \equiv (#) 
#define x_entry_strs(#) \equiv entry_strs[ (#) \ast (ent_str_size + 1) + x_entry_strs_tail
#define x_global_strs(#) \equiv global_strs[ (#) \ast (glob_str_size + 1) + x_global_strs_tail
\end{verbatim}

In case somebody has inadvertently made bad settings of the “constants,” \Bibtex\ checks them using a global variable called \texttt{bad}.

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

\begin{verbatim}
\langle \text{Globals in the outer block 2*} \rangle +\equiv
pool_size: integer;
max_print_line: integer;
max_bib_files: integer;
max_cites: 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? \}
\end{verbatim}

Each digit-value of \texttt{bad} has a specific meaning.

\begin{verbatim}
\langle \text{Check the “constant” values for consistency 17*} \rangle \equiv
bad \leftarrow 0;
if (min_print_line < 3) then bad \leftarrow 1;
if (max_print_line \leq min_print_line) then bad \leftarrow 10 \ast bad + 2;
if (max_print_line \geq buf_size) then bad \leftarrow 10 \ast bad + 3;
if (hash_prime < 128) then bad \leftarrow 10 \ast bad + 4;
if (hash_prime > hash_size) then bad \leftarrow 10 \ast bad + 5;
if (hash_base \neq 1) then bad \leftarrow 10 \ast bad + 6;
if (max_strings > hash_size) then bad \leftarrow 10 \ast bad + 7;
if (max_cites > max_strings) then bad \leftarrow 10 \ast bad + 8;  \{ well, almost each \}
\end{verbatim}

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 \texttt{ASCII code}, which is a subrange of the integers.

\[
\text{Types in the outer block 22*} \equiv \text{ASCII code } = 0 \ldots 255; \{ \text{eight-bit numbers} \}
\]

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 \texttt{char} for the characters in text files, even though there now are more than 64 such characters, while other PASCALs consider \texttt{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 \texttt{text_char} to stand for the data type of the characters that are converted to and from \texttt{ASCII code} when they are input and output. We shall also assume that \texttt{text_char} consists of the elements \texttt{chr(first_text_char)} through \texttt{chr(last_text_char)}, inclusive. The following definitions should be adjusted if necessary.

\[
\text{define text_char } \equiv \text{ASCII code} \{ \text{the data type of characters in text files} \}
\]

\[
\text{define first_text_char } = 0 \{ \text{ordinal number of the smallest element of text_char} \}
\]

\[
\text{define last_text_char } = 255 \{ \text{ordinal number of the largest element of text_char} \}
\]

(\text{Local variables for initialization 23*}) \equiv

\[
i: \text{integer};
\]

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 \texttt{xchr[1 \ldots 37]}, but the safest policy is to blank everything out by using the code shown below.

However, other settings of \texttt{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

\[
\text{for } i \leftarrow 1 \text{ to } 37 \text{ do } \texttt{xchr[i]} \leftarrow \texttt{chr(i)};
\]

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 \texttt{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.

(\text{Set initial values of key variables 20} ) \equiv

\[
\text{for } i \leftarrow 0 \text{ to } 37 \text{ do } \texttt{xchr[i]} \leftarrow \texttt{chr(i)};
\]

\[
\text{for } i \leftarrow 177 \text{ to } 377 \text{ do } \texttt{xchr[i]} \leftarrow \texttt{chr(i)};
\]
28* This system-independent code makes the \texttt{xord} array contain a suitable inverse to the information in \texttt{xchr}. Note that if \texttt{xchr[i] = xchr[j]} where \texttt{i < j < 177}, the value of \texttt{xord[xchr[i]]} will turn out to be \texttt{j} or more; hence, standard ASCII code numbers will be used instead of codes below \texttt{'40} in case there is a coincidence.

\begin{verbatim}
( Set initial values of key variables 20 ) +≡
  for i ← first_text_char to last_text_char do xord[xchr[i]] ← i;
\end{verbatim}

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

\begin{verbatim}
( Set initial values of key variables 20 ) +≡
  for i ← 0 to 177 do lex_class[i] ← other_lex;
  for i ← 200 to 377 do lex_class[i] ← alpha;
  for i ← 0 to 37 do lex_class[i] ← illegal;
  lex_class[invalid_code] ← illegal; lex_class[tab] ← white_space; lex_class[13] ← white_space;
  lex_class[space] ← white_space; lex_class[tie] ← sep_char; lex_class[hyphen] ← sep_char;
  for i ← 60 to 71 do lex_class[i] ← numeric;
  for i ← 101 to 132 do lex_class[i] ← alpha;
  for i ← 141 to 172 do lex_class[i] ← alpha;
\end{verbatim}

33* And now the \texttt{id_class} array.

\begin{verbatim}
( Set initial values of key variables 20 ) +≡
  for i ← 0 to 377 do id_class[i] ← legal_id_char;
  for i ← 0 to 37 do id_class[i] ← illegal_id_char;
  id_class[space] ← illegal_id_char; id_class[tab] ← illegal_id_char; id_class[double_quote] ← illegal_id_char;
  id_class[single_quote] ← illegal_id_char; id_class[comment] ← illegal_id_char;
  id_class[right_paren] ← illegal_id_char; id_class[left_paren] ← illegal_id_char;
  id_class[equals_sign] ← illegal_id_char; id_class[comma] ← illegal_id_char;
  id_class[right_brace] ← illegal_id_char;
\end{verbatim}
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.

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

function bib_makecstring(s : str_number): cstring;
var cstr: cstring; i: pool_pointer;
begincstr ← 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;
```

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.

Input from text files is read one line at a time, using a routine called *input_in*. 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.)

```
buf_type = ↑ASCII_code;  { for various buffers}
```

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.
§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*) \(\equiv\)

**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_XRETLALLOC_NOSET(‘buffer’, buffer, ASCII_code, buf_size, buf_size + BUF_SIZE);
BIB_XRETLALLOC_NOSET(‘sv_buffer’, sv_buffer, ASCII_code, buf_size, buf_size + BUF_SIZE);
BIB_XRETLALLOC_NOSET(‘exbuf’, exbuf, ASCII_code, buf_size, buf_size + BUF_SIZE);
BIB_XRETLALLOC_NOSET(‘outbuf’, outbuf, ASCII_code, buf_size, buf_size + BUF_SIZE);
BIB_XRETLALLOC_NOSET(‘nametok’, nametok, buf_pointer, buf_size, buf_size + BUF_SIZE);
BIB_XRETLALLOC(‘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 ← 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 BIB\TeX returns only a weaker restriction: If eof occurs in the middle of a line, the system function eoln should return a true result (even though \(f^\dagger\) will be undefined).

(Procedures and functions for all file I/O, error messages, and such 3*) \(\equiv\)

**function input Ln(\textbf{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 \(\geq\) buf_size) then buffer_overflow;
buffer[last] ← xord[getc(\(f\))]; incr(last);
end;
vggetc(\(f\)); \{ 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. \TeX{} uses variable-length strings of seven-bit characters. Since PASCAL does not have a well-developed string mechanism, \TeX{} does all its string processing by home-grown (predominantly \TeX{}’s) methods. Unlike \TeX{}, however, \BIB{} 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 \texttt{str_pool} contains all the (seven-bit) ASCII codes in all the strings \TEX{} must ever search for (generally identifiers names), and the array \texttt{str_start} contains indices of the starting points of each such string. Strings are referred to by integer numbers, so that string number \texttt{s} comprises the characters \texttt{str_pool[j]} for \texttt{str_start[s] \leq j < str_start[s + 1]}.

Additional integer variables \texttt{pool_ptr} and \texttt{str_ptr} indicate the number of entries used so far in \texttt{str_pool} and \texttt{str_start}; locations \texttt{str_pool[pool_ptr]} and \texttt{str_start[str_ptr]} are ready for the next string to be allocated. Location \texttt{str_start[0]} is unused so that hashing will work correctly.

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

\begin{verbatim}
(Global in the outer block 2*) +≡
\texttt{str_pool: \uparrow ASCII_code;} { the characters }
\texttt{str_start: \uparrow pool_pointer;} { the starting pointers }
\texttt{pool_ptr: pool_pointer;} { first unused position in \texttt{str_pool} }
\texttt{str_ptr: str_number;} { start of the current string being created }
\texttt{str_num: str_number;} { start of the current string being created }
\texttt{p_ptr1, p_ptr2: pool_pointer;} { several procedures use these locally }
\end{verbatim}

49* Where \texttt{pool_pointer} and \texttt{str_number} are pointers into \texttt{str_pool} and \texttt{str_start}.

\begin{verbatim}
(Types in the outer block 22*) +≡
\texttt{pool_pointer = integer;} { for variables that point into \texttt{str_pool} }
\texttt{str_number = integer;} { for variables that point into \texttt{str_start} }
\end{verbatim}

50* These macros send a string in \texttt{str_pool} to an output file.

\begin{verbatim}
define max_pop = 3 { —see the \texttt{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(#)
\end{verbatim}

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

To test if there is room to append \texttt{l} more characters to \texttt{str_pool}, we shall write \texttt{str_room(l)}, which aborts \TeX{} and gives an error message if there isn’t enough room.

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

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

\begin{verbatim}
procedure pool_overflow;
  begin BIB_XRETLALLOC(\texttt{“str_pool”, str_pool, ASCII_code, pool_size, pool_size + \texttt{POOL_SIZE}});
  end;
\end{verbatim}
§58  This procedure copies file name \texttt{file_name} into the beginning of \texttt{name_of_file}, if it will fit. It also sets the global variable \texttt{name_length} to the appropriate value.

\begin{verbatim}
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;
\end{verbatim}

See also sections 60* and 61*.

This code is used in section 12.

59*  Yet another complaint-before-quitting.

\begin{verbatim}
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;
\end{verbatim}

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

\begin{verbatim}
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;
\end{verbatim}

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

\begin{verbatim}
procedure add_area(area : str_number);
  var p_ptr: pool_pointer;  \{ running index \}
  begin name_ptr ← name_length + 1; p_ptr ← str_start[area];
    while (p_ptr < str_start[area + 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(area); name_of_file[name_length + 1] ← 0;
  end;
\end{verbatim}

This code is used in section 12.
The hash table. All static strings that \LaTeX{} 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 \texttt{str_pool} array.

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

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

\begin{verbatim}
⟨Types in the outer block \texttt{22}\*⟩ \equiv
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 \}
\end{verbatim}

\begin{verbatim}
⟨Globals in the outer block \texttt{2}\*⟩ \equiv
hash_next: \uparrow hash_pointer; \{ coalesced-list link \}
hash_text: \uparrow str_number; \{ pointer to a string \}
hash_ilk: \uparrow str_ilk; \{ the type of string \}
ilk_info: \uparrow integer; \{ ilk-specific info \}
hash_used: integer; \{ allocation pointer for hash table \}
hash_found: boolean; \{ set to \texttt{true} if it’s already in the hash table \}
dummy_loc: hash_loc; \{ receives \texttt{str_lookup} value whenever it’s useless \}
\end{verbatim}
§68 THE HASH TABLE

68* Here is the subroutine that searches the hash table for a \( \langle \text{string}, \text{strilk} \rangle \) pair, where the string is of length \( l \geq 0 \) and appears in \( \text{buffer}[j \ldots (j + l - 1)] \). If it finds the pair, it returns the corresponding hash-table location and sets the global variable \( \text{hash\_found} \) to true. Otherwise it sets \( \text{hash\_found} \) to false, and if the parameter \( \text{insert\_it} \) is true, it inserts the pair into the hash table, inserts the string into \( \text{str\_pool} \) if not previously encountered, and returns its location. Note that two different pairs can have the same string but different \( \text{str\_ilks} \), in which case the second pair encountered, if \( \text{insert\_it} \) were true, would be inserted into the hash table though its string wouldn’t be inserted into \( \text{str\_pool} \) because it would already be there.

\begin{verbatim}
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;

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 \leq h < hash\_prime \} \nhash\_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 \( (\text{hash\_next}[p] = \text{empty}) \) then  \{ location \( p \) may or may not be empty \}
      begin if \( \neg \text{insert\_it} \) then goto str_not_found;
          \{ Insert pair into hash table and make \( p \) point to it 71 \};
          goto str_found;
      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 \( \text{str\_ilk} \) doesn’t match.

\begin{verbatim}
begin \{ Process the string if we’ve already encountered it 70 \} +≡

begin if \( (\text{hash\_text}[p] > 0) \) then  \{ there’s something here \}
  if \( (\text{str\_eq\_buf}(\text{hash\_text}[p], \text{buf}, j, l)) \) then  \{ it’s the right string \}
    begin if \( \text{hash\_ilk}[p] = \text{ilk} \) then  \{ it’s the right \text{str\_ilk} \}
        begin hash_found ← true; goto str_found;
        end
      else begin str_num ← hash\_text[p];
        end;

end

This code is used in section 68*.\end{verbatim}
This code inserts the pair into the appropriate unused location.  
\langle Insert pair into hash table and make \( p \) point to it \rangle 71*  
\[ \begin{aligned} 
\text{begin} & \text{if (} \text{hash}_\text{text}[p] > 0) \text{ then } \{ \text{location } p \text{ isn't empty} \} \\
& \text{begin repeat if (} \text{hash}_\text{is full}) \text{ then } \text{overflow('hash}\_\text{size', hash size}); \\
& \text{decr(} \text{hash}_\text{used}) ; \\
& \text{until (} \text{hash}_\text{text}[\text{hash}_\text{used}] = 0) ; \{ \text{search for an empty location} \} \\
& \text{hash}_\text{next}[p] \leftarrow \text{hash}_\text{used} ; \; p \leftarrow \text{hash}_\text{used} ; \text{end} ; \{ \text{now location } p \text{ is empty} \} \\
\text{if (} \text{str}_\text{num} > 0) \text{ then } \{ \text{it's an already encountered string} \} \\
& \text{hash}_\text{text}[p] \leftarrow \text{str}_\text{num} \\
\text{else begin } \{ \text{it's a new string} \} \\
& \text{\text{str}_\text{room}(l)} ; \{ \text{make sure it'll fit in \text{str}_\text{pool}} \} \\
& \; k \leftarrow j ; \\
& \text{while } (k < j + l) \text{ do } \{ \text{not a for loop in case } j = l = 0 \} \\
& \; \text{begin append_char(buf[k]); incr(k);} \\
& \text{end} ; \\
& \text{hash}_\text{text}[p] \leftarrow \text{make_string} ; \{ \text{and make it official} \} \\
& \text{end} ; \\
& \text{hash}_\text{ilk}[p] \leftarrow \text{ilk} ; \\
\text{end} \end{aligned} \]

This code is used in section 68*.

The longest pre-defined string determines type definitions used to insert the pre-defined strings into \text{str}_\text{pool}.

\[ \begin{aligned} 
\text{define longest}_\text{pds} = 12 \; \{ \text{the length of 'change\.case'} \} \\
\text{\langle Types in the outer block 22* } \rangle \equiv \\
\text{pds}_\text{loc} = 1 \ldots \text{longest}_\text{pds}; \; \text{pds}_\text{len} = 0 \ldots \text{longest}_\text{pds}; \; \text{pds}_\text{type} = \text{const.cstring}; \\
\end{aligned} \]

This procedure initializes a pre-defined string of length at most \text{longest}_\text{pds}.

\[ \begin{aligned} 
\langle \text{Procedures and functions for handling numbers, characters, and strings 54* } \rangle \equiv \\
\text{procedure pre\_define(pds : pds\_type; len : pds\_len; ilk : str\_ilk);} \\
& \text{var i: pds\_len;} \\
& \text{begin for } i \leftarrow 1 \text{ to } \text{len} \text{ do } \text{buffer}[i] \leftarrow \text{xord[ucharcast(pds[i-1])];} \\
& \text{pre\_def\_loc } \leftarrow \text{str\_lookup(buffer, 1, len, ilk, do\_insert);} \\
& \text{end} ; \\
\end{aligned} \]
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.

```c
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*)

```c
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*)

```c
procedure get_the_top_level_aux_file_name;
    label aux_found, aux_not_found;
    begin
        { 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*)

```c
parse_arguments;
```

This code is used in section 10*.

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

(Add extensions and open files 106*)

```c
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;
```

This code is used in section 103.
Print the name of the current .aux file, followed by a newline.

\begin{verbatim}
\procedure{print_aux_name;
  \begin{verbatim}
    print_pool_str(cur_aux_str); print_newline;
  \end{verbatim}
\end{verbatim}

\procedure{log_pr_aux_name;
  \begin{verbatim}
    log_pr_pool_str(cur_aux_str); log_pr_newline;
  \end{verbatim}
\end{verbatim}
\end{verbatim}
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*) $\equiv$

```
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 }
```
Here’s a procedure we’ll need shortly. It prints the name of the current \texttt{.bib} file, followed by a newline.

\{ Return true if the \texttt{ext} string is at the end of the \texttt{s} string. There are surely far more clever ways to do this, but it doesn’t matter. \}

\begin{verbatim}
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
            \{ ≥ 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 \texttt{\bibdata} command may or may not have the \texttt{.bib} extension. If it does, we don’t want to print \texttt{.bib} twice. \}
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}
Now we add the just-found argument to \texttt{bib\_list} if it hasn’t already been encountered as a \texttt{\bibdata} argument and if, after appending the \texttt{s\_bib\_extension} string, the resulting file name can be opened.

\begin{verbatim}
(Open a \texttt{.bib} file 123*) \equiv
begin if (bib\_ptr = max\_bib\_files) then
    begin if (max\_bib\_files + MAX\_BIB\_FILES) then
        BIB\_XRETAILLOC\_NOSET(`bib\_list`, bib\_list, str\_number, max\_bib\_files,
        max\_bib\_files + MAX\_BIB\_FILES); BIB\_XRETAILLOC\_NOSET(`bib\_file`, bib\_file, alpha\_file,
        max\_bib\_files, max\_bib\_files + MAX\_BIB\_FILES); BIB\_XRETAILLOC(`s\_preamble`, s\_preamble,
        str\_number, max\_bib\_files, max\_bib\_files + MAX\_BIB\_FILES);
    end;
    cur\_bib\_str \leftarrow hash\_text[\texttt{str\_lookup(buffer, buf\_ptr1, token\_len, bib\_file\_ilk, do\_insert)}];
    if (hash\_found) then  \{ already encountered this as a \texttt{\bibdata} argument \}
        open\_bib\_data\_aux\_err(`This\_database\_file\_appears\_more\_than\_once:`);
        start\_name(cur\_bib\_str);
        if (\neg kpse\_in\_name\_ok(stringcast(name\_of\_file + 1)) \lor \neg a\_open\_in(cur\_bib\_file, kpse\_bib\_format)) then
            open\_bib\_data\_aux\_err(`I\_could\_nt\_open\_database\_file`);
            trace trace\_pr\_pool\_str(cur\_bib\_str); trace\_pr\_pool\_str(s\_bib\_extension);
            trace\_pr\_ln(`is\_a\_bib\_data\_file`);
            ecart
            incr(bib\_ptr);
        end
    end
This code is used in section 120.
\end{verbatim}

Now we open the file whose name is the just-found argument appended with the \texttt{s\_bst\_extension} string, if possible.

\begin{verbatim}
(Open the \texttt{.bst} file 127*) \equiv
begin bst\_str \leftarrow hash\_text[\texttt{str\_lookup(buffer, buf\_ptr1, token\_len, bst\_file\_ilk, do\_insert)}];
    if (hash\_found) then  \{ already encountered this as a \texttt{\bstdata} argument \}
        begin trace print\_bst\_name;
            ecart
            confusion(`Already\_encountered\_style\_file`);
        end;
    start\_name(bst\_str);
    if (\neg kpse\_in\_name\_ok(stringcast(name\_of\_file + 1)) \lor \neg a\_open\_in(bst\_file, kpse\_bst\_format)) then
        begin print(`I\_could\_nt\_open\_style\_file`); print\_bst\_name;
            bst\_str \leftarrow 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
This code is used in section 126.
\end{verbatim}
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;

Here we introduce some variables for processing a \citeation 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 }
um_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 \citeation 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 }

Where cite_number is the obvious.

(Types in the outer block 22*) +%≡
cite_number = integer;  { gives the cite_list range }

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_XREALLOC_NOSET(‘cite_list’, cite_list, str_number, max_cites,
max_cites + MAX_CITES);
BIB_XREALLOC_NOSET(‘type_list’, type_list, hash_ptr2, max_cites, max_cites + MAX_CITES);
BIB_XREALLOC_NOSET(‘entry_exists’, entry_exists, boolean, max_cites,
max_cites + MAX_CITES);
BIB_XREALLOC(‘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 }
icr(last_cite);
end;
end;
end;
141* We check that this .aux file can actually be opened, and then open it.

\[
\langle \text{Open this .aux file 141*} \rangle \equiv \\
\begin{align*}
\text{begin } & \text{start\_name(\texttt{cur\_aux\_str}); \{ extension already there for .aux files \}} \\
& \text{name\_ptr} \leftarrow \text{name\_length} + 1; \text{name\_of\_file[name\_ptr]} \leftarrow 0; \\
& \text{if } (\neg \text{kpse\_in\_name\_ok(stringcast(name\_of\_file + 1))} \lor (\neg \text{a\_open\_in(cur\_aux\_file,} \\
& \text{no\_file\_path}) \land \neg \text{a\_open\_in\_with\_dirname(cur\_aux\_file, no\_file\_path, bib\_makecstring(top\_lev\_str)))}) \\
& \quad \text{then} \\
& \quad \text{begin } \text{print(´I´ could´n´t \text{open} auxiliary \text{file};´); print\_aux\_name; decr(aux\_ptr);} \\
& & \quad \text{aux\_err\_return;}
& \quad \text{end;}
& \quad \text{log\_pr(´A´ level\_´, aux\_ptr : 0, ´auxiliary\text{file};´); log\_pr\_aux\_name; cur\_aux\_line} \leftarrow 0;
& \quad \text{end}
\end{align*}
\]

This code is used in section 140.
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 \textbf{begin - end} pair surrounding the entire module.

\begin{verbatim}
⟨Read and execute the .bst file 151*⟩ ≡
  \textbf{if} (bst_str = 0) \textbf{then} \{ there's no .bst file to read \}
  \textbf{goto} no_bst_file; \{ this is a \textbf{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;
  \textbf{begin if} (¬eat_bst_white_space) \textbf{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);
\end{verbatim}

This code is used in section 10*.

Besides the function classes, we have types based on \LaTeX{}'s capacity limitations and one based on what can go into the array \textit{wiz_functions} explained below.

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

Six other arrays give the contents of functions: The array \texttt{wiz\_functions} holds definitions for \texttt{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 \texttt{entry\_ints} contains the current values of \texttt{int\_entry\_vars}; the array \texttt{entry\_strs} contains the current values of \texttt{str\_entry\_vars}; an element of the array \texttt{global\_strs} contains the current value of a \texttt{str\_global\_var} if the corresponding \texttt{glb\_str\_ptr} entry is empty, otherwise the nonempty entry is a pointer to the string; and the array \texttt{field\_info}, for each field of each entry, contains either a pointer to the string or the special value \texttt{missing}.

The array \texttt{global\_strs} isn’t packed (that is, it isn’t \texttt{array \ldots of packed array \ldots}) to increase speed on some systems; however, on systems that are byte-addressable and that have a good compiler, packing \texttt{global\_strs} would save lots of space without much loss of speed.

\begin{verbatim}
define fn_info \equiv ilk_info \{ an alias used with functions \}
define missing = empty \{ a special pointer for missing fields \}

(Globals in the outer block 2*) \equiv 
f\_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: \uparrow\text{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: \uparrow\text{hash\_ptr2};

int\_ent\_ptr: int\_ent\_loc; \{ general int\_entry\_var location \}

entry\_ints: \uparrow\text{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: \uparrow\text{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: \uparrow\text{str\_number};

global\_strs: \uparrow\text{ASCII\_code};

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: \uparrow\text{str\_number};

num\_fields: field\_loc; \{ the number of distinct field names \}

num\_pre\_defined\_fields: field\_loc; \{ so far, just one: \texttt{crossref} \}

crossref\_num: field\_loc; \{ the number given to \texttt{crossref} \}

no\_fields: boolean; \{ used for \texttt{tr\_printing} entry information \}
\end{verbatim}
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.

\(\text{Procedure and functions for input scanning}\) +≡

**procedure scan_fn_def (fn_hash : 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 \neq \text{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;

**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(singl_ptr);
        end

\(\text{Procedures and functions for all file I/O, error messages, and such}\) +≡
This procedure takes the integer \texttt{int}, copies the appropriate \texttt{ASCII} code string into \texttt{int\_buf} starting at \texttt{int\_begin}, and sets the \texttt{var} parameter \texttt{int\_end} to the first unused \texttt{int\_buf} location. The \texttt{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.

\begin{verbatim}
define \texttt{int} ≡ the_int

procedure \texttt{int\_to\_ASCII} (\texttt{int : integer}; \texttt{var int\_buf : buf\_type}; \texttt{int\_begin : buf\_pointer};
               \texttt{var int\_end : buf\_pointer});

  \texttt{var int\_ptr, int\_xptr : buf\_pointer};  \{ pointers into \texttt{int\_buf} \}
  \texttt{int\_tmp\_val : ASCII\_code};  \{ the temporary element in an exchange \}

begin \texttt{int\_ptr} ← \texttt{int\_begin};
if (\texttt{int} < 0) then \{ add the \texttt{minus\_sign} and use the absolute value \}
  begin \texttt{append\_int\_char}(\texttt{minus\_sign}); \texttt{int} ← −\texttt{int};
  end;
\texttt{int\_xptr} ← \texttt{int\_ptr};
repeat \{ copy digits into \texttt{int\_buf} \}
  \texttt{append\_int\_char}("0" + (\texttt{int} mod 10)); \texttt{int} ← \texttt{int div} 10;
until (\texttt{int} = 0);
\texttt{int\_end} ← \texttt{int\_ptr};  \{ set the string length \}
\texttt{decr} (\texttt{int\_ptr});
while (\texttt{int\_xptr} < \texttt{int\_ptr}) do \{ and reorder (flip) the digits \}
  begin \texttt{int\_tmp\_val} ← \texttt{int\_buf}[\texttt{int\_xptr}]; \texttt{int\_buf}[\texttt{int\_xptr}] ← \texttt{int\_buf}[\texttt{int\_ptr}];
    \texttt{int\_buf}[\texttt{int\_ptr}] ← \texttt{int\_tmp\_val}; \texttt{decr} (\texttt{int\_ptr}); \texttt{incr} (\texttt{int\_xptr});
  end
end;
\end{verbatim}

Now we add the \texttt{end\_of\_def} special marker, make sure this function will fit into \texttt{wiz\_functions}, and put it there.

\begin{verbatim}
\langle Complete this function’s definition \texttt{200*} \rangle ≡

begin \texttt{insert\_fn\_loc} (\texttt{end\_of\_def}); \{ add special marker ending the definition \}
while (\texttt{single\_ptr} + \texttt{wiz\_def\_ptr} > \texttt{wiz\_fn\_space}) do
  begin BIB\_XREALALLOC(\texttt{wiz\_functions}\_ptr, \texttt{wiz\_functions}, \texttt{hash\_ptr2}, \texttt{wiz\_fn\_space},
                   \texttt{wiz\_fn\_space} + \texttt{WIZ\_FN\_SPACE});
end;
\texttt{fn\_info}[\texttt{fn\_hash\_loc}] ← \texttt{wiz\_def\_ptr}; \{ pointer into \texttt{wiz\_functions} \}
\texttt{copy\_ptr} ← 0;
while (\texttt{copy\_ptr} < \texttt{single\_ptr}) do \{ make this function official \}
  begin \texttt{wiz\_functions}[\texttt{wiz\_def\_ptr}] ← \texttt{singl\_function}[\texttt{copy\_ptr}]; \texttt{incr} (\texttt{copy\_ptr}); \texttt{incr} (\texttt{wiz\_def\_ptr});
end;
\end{verbatim}

This code is used in section \texttt{187*}. 
216* Here we insert the just found \texttt{str\_global\_var} name into the hash table, record it as a \texttt{str\_global\_var},
set its pointer into \texttt{global\_strs}, and initialize its value there to the null string.

\begin{verbatim}
define end_of_string = invalid_code  { this illegal ASCII code ends a string }
⟨ Insert a \texttt{str\_global\_var} into the hash table 216* ⟩≡
begin trace trace_pr_token; trace_pr_in("\texttt{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] ← \texttt{str\_global\_var};
fn_info[fn_loc] ← num\_glb\_strs;  { pointer into \texttt{global\_strs} }
if (num\_glb\_strs = max\_glob\_strs) then
  begin BIB\_XRETAILLOC\_NOSET(\texttt{glb\_str\_ptr}, glb\_str\_ptr, str\_number, max\_glob\_strs,
             max\_glob\_strs + MAX\_GLOB\_STRS); BIB\_XRETAILLOC\_STRING(\texttt{global\_strs},
             glob\_str\_size, max\_glob\_strs, max\_glob\_strs + MAX\_GLOB\_STRS);
BIB\_XRETAILLOC(\texttt{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 \texttt{str\_global\_vars} empty }
    begin glb\_str\_ptr[\texttt{str\_glb\_ptr}] ← 0; glb\_str\_end[\texttt{str\_glb\_ptr}] ← 0; incr (\texttt{str\_glb\_ptr});
    end;
  end;
  incr (num\_glb\_strs);
end
\end{verbatim}

This code is used in section 215.
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*) \equiv

\begin{align*}
\text{bib_line_num: integer; } & \text{ line number of the .bib file} \\
\text{entry_type_loc: hash_loc; } & \text{ the hash-table location of an entry type} \\
\text{type_list: } & \text{hash_ptr2; } \\
\text{type_exists: boolean; } & \text{ true if this entry type is .bst-defined} \\
\text{entry_exists: } & \text{hash_ptr2; } \\
\text{store_entry: boolean; } & \text{ true if we’re to store info for this entry} \\
\text{field_name_loc: hash_loc; } & \text{ the hash-table location of a field name} \\
\text{field_val_loc: hash_loc; } & \text{ the hash-table location of a field value} \\
\text{store_field: boolean; } & \text{ true if we’re to store info for this field} \\
\text{store_token: boolean; } & \text{ true if we’re to store this macro token} \\
\text{right_outer_delim: ASCII_code; } & \text{ either a right_brace or a right_paren} \\
\text{right_str_delim: ASCII_code; } & \text{ either a right_brace or a double_quote} \\
\text{at_bib_command: boolean; } & \text{ true for a command, false for an entry} \\
\text{cur_macro_loc: hash_loc; } & \text{ macro_loc for a string being defined} \\
\text{cite_info: } & \text{hash_ptr2; } \\
\text{preamble_ptr: bib_number; } & \text{ pointer into the s_preamble array} \\
\text{num_preamble_strings: bib_number; } & \text{ counts the s_preamble strings}
\end{align*}

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

(Read the .bib file(s) 23*)

\begin{align*}
\text{begin } & \text{(Final initialization for .bib processing 224);} \\
\text{read_performed } & \leftarrow \text{true; } \text{bib_ptr } \leftarrow 0; \\
\text{while } & (\text{bib_ptr } < \text{num_bib_files}) \text{ do} \\
\text{begin } & \text{if verbose then} \\
\text{begin } & \text{print(‘Database_file[#’, bib_ptr + 1 : 0, ‘:\:0’); print_bib_name}; \\
\text{end} \\
\text{else begin } & \text{log_pr(‘Database_file[#’, bib_ptr + 1 : 0, ‘:\:0’); log_pr_bib_name}; \\
\text{end; } \\
\text{bib_line_num } & \leftarrow 0; \text{ initialize to get the first input line } \\
\text{buf_ptr2 } & \leftarrow \text{last;} \\
\text{while } & \text{not eof(cur_bib_file)) do get_bib_command_or_entry_and_process; } \\
\text{a_close(cur_bib_file); incr(bib_ptr);} \\
\text{end;} \\
\text{reading_completed } & \leftarrow \text{true;} \\
\text{trace } & \text{trace_pr_in(’Finished_reading_the_database_file(s’);} \\
\text{ecart} & \text{(Final initialization for processing the entries 276);} \\
\text{read_completed } & \leftarrow \text{true;} \\
\text{end}
\end{align*}

This code is used in section 211.
Complain if somebody’s got a field fetish.

(Procedures and functions for all file I/O, error messages, and such 3* ) \equiv

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 \leftarrow max_fields;
      BIB XRETAILLOC ( "field_info", field_info, str_number, max_fields, total_fields + MAX_FIELDS );
      { Initialize to missing. }
      for f_ptr \leftarrow start_fields to max_fields \small{-} 1 do
        begin field_info [ f_ptr ] \leftarrow missing;
        end;
      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 chars. 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* ) \equiv

begin if ( preamble_ptr = max_bib_files ) then
  begin { Keep old value of max_bib_files for the last array. }
    BIB XRETAILLOC_NOSET ( "bib_list", bib_list, str_number, max_bib_files, 
    max_bib_files + MAX_BIB_FILES ); BIB XRETAILLOC_NOSET ( "bib_file", bib_file, alpha_file, 
    max_bib_files, max_bib_files + MAX_BIB_FILES ); BIB XRETAILLOC ( "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 \leftarrow right_brace
  else if ( scan_char = left_paren ) then right_outer_delim \leftarrow 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 \leftarrow true;
  if ( \neg scan_and_store_the_field_value_and_eat_white ) then return;
  if ( scan_char \neq 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.

```latex
\text{define \texttt{copy\_char}(\#) \equiv}
\begin{align*}
&\text{begin} \quad \{ \text{We don’t always increment by 1, so have to check \geq}. \}\}
&\text{if (field\_end \geq buf\_size) then}
&\begin{align*}
&\text{begin log\_pr(} \text{'Field\_filled, up\_at\,\#, \, reallocating.'); log\_pr\_newline; buffer\_overflow; } \quad \{ \text{reallocates all buf\_size buffers} \}
&\text{end;}
&\end{align*}
&\text{field\_vl\_str[field\_end] \leftarrow \#; incr(field\_end);}
\end{align*}
```

263 And here, an entry.

\langle Store the field value for a database entry 263* \rangle \equiv
\begin{align*}
\text{begin field\_ptr \leftarrow entry\_cite\_ptr \ast num\_fields + fn\_info[field\_name\_loc];}
&\text{if (field\_ptr \geq max\_fields) then confusion(} \text{'field\_info, index, is, out\_of\_range.')};
&\text{if (field\_info[field\_ptr] \neq missing) then}
&\begin{align*}
&\text{begin print(} \text{'Warning--I\,\, ignoring\,\,'}; print\_pool\_str(cite\_list[entry\_cite\_ptr]);
&\text{print(} \text{'\,\, extra\,'}; print\_pool\_str(hash\_text[field\_name\_loc]); bib\_warn\_newline(} \text{'field');}
&\text{end}
&\end{align*}
&\text{else begin} \quad \{ \text{the field was empty, store its new value} \}
&\text{field\_info[field\_ptr] \leftarrow hash\_text[field\_val\_loc];}
&\text{if ((fn\_info[field\_name\_loc] = crossref\_num) \land (\,\neg\, all\_entries)) then}
&\begin{align*}
&\text{\langle Add or update a cross reference on cite\_list if necessary 264 \rangle;}
&\end{align*}
&\text{end;}
&\text{end}
\end{align*}
```

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.

\langle Procedures and functions for handling numbers, characters, and strings 54 \rangle +\equiv
\text{procedure add\_database\_cite(var new\_cite : cite\_number);}\begin{align*}
&\text{begin check\_cite\_overflow(new\_cite); } \quad \{ \text{make sure this cite will fit} \}
&\text{check\_field\_overflow(num\_fields \ast (new\_cite + 1)); cite\_list[new\_cite] \leftarrow hash\_text[cite\_loc];}
&\text{ilk\_info[cite\_loc] \leftarrow new\_cite; ilk\_info[lc\_cite\_loc] \leftarrow cite\_loc; incr(new\_cite);}
&\text{end;}
\end{align*}
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.

\[
\text{⟨Add cross-reference information 277*⟩ ≡ }
\]
\[
\text{begin if } ((\text{num_cites} - 1) * \text{num_fields} + \text{crossref}.\text{num} \geq \text{max_fields}) \text{ then }
\]
\[
\text{confusion}('\text{field_info}.\text{index} \text{is out of range'});
\]
\[
cite.\text{ptr} \leftarrow 0;
\]
\[
\text{while } (\text{cite}_.\text{ptr} < \text{num_cites}) \text{ do }
\]
\[
\text{begin field}_.\text{ptr} \leftarrow \text{cite}_.\text{ptr} * \text{num_fields} + \text{crossref}.\text{num};
\]
\[
\text{if } (\text{field}_.\text{info}[\text{field}_.\text{ptr}] \neq \text{missing}) \text{ then }
\]
\[
\text{if } (\text{find_cite_locs_for_this_cite_key(\text{field}_.\text{info}[\text{field}_.\text{ptr}])) \text{ then }
\]
\[
\text{begin cite}_.\text{loc} \leftarrow \text{ilk}_.\text{info}[\text{lc}_.\text{cite}_.\text{loc}]; \text{ field}_.\text{info}[\text{field}_.\text{ptr}] \leftarrow \text{hash_text}[\text{cite}_.\text{loc}];
\]
\[
\text{cite}_.\text{parent}_.\text{ptr} \leftarrow \text{ilk}_.\text{info}[\text{cite}_.\text{loc}]; \text{ field}_.\text{ptr} \leftarrow \text{cite}_.\text{ptr} * \text{num_fields} + \text{num_pre_defined_fields};
\]
\[
\text{field}_.\text{end}_.\text{ptr} \leftarrow \text{field}_.\text{ptr} - \text{num_pre_defined_fields} + \text{num_fields};
\]
\[
\text{field}_.\text{parent}_.\text{ptr} \leftarrow \text{cite}_.\text{parent}_.\text{ptr} * \text{num_fields} + \text{num_pre_defined_fields};
\]
\[
\text{while } (\text{field}_.\text{ptr} < \text{field}_.\text{end}_.\text{ptr}) \text{ do }
\]
\[
\text{begin if } (\text{field}_.\text{info}[\text{field}_.\text{ptr}] = \text{missing}) \text{ then } \text{field}_.\text{info}[\text{field}_.\text{ptr}] \leftarrow \text{field}_.\text{info}[\text{field}_.\text{parent}_.\text{ptr}];
\]
\[
\text{incr(field}_.\text{ptr}); \text{ incr(field}_.\text{parent}_.\text{ptr});
\]
\[
\text{end};
\]
\[
\text{end};
\]
\[
\text{end}
\]
\[
\text{This code is used in section 276.}
\]
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.

\[\text{(Subtract cross-reference information} \ 279^* \ \text{)} \equiv \]
\[
\begin{align*}
\text{begin if } & ((\text{num_cites} - 1) \ast \text{num_fields} + \text{crossref_num} \geq \text{max_fields}) \text{ then} \\
& \text{confusion(‘field_info\_index\_is\_out\_of\_range’); } \\
& \text{cite_ptr} \leftarrow 0; \\
\text{while (cite_ptr < num_cites) do} \\
& \text{begin cite_ptr} \leftarrow \text{cite_ptr} \ast \text{num_fields} + \text{crossref_num;} \\
& \text{if (field_info[cite_ptr] \neq \text{missing) then} } \\
& \text{if (\text{find\_cite\_locs\_for\_this\_cite\_key(field_info[cite_ptr]]) then} } \\
& \text{begin } \{ \text{the parent is not on cite_list} \} \\
& \text{if (\text{cite\_hash\_found) then hash\_cite\_confusion; } } \\
& \text{nonexistent\_cross\_reference\_error; field_info[cite_ptr] \leftarrow missing; } \{ \text{remove the crossref ptr} \} \\
& \text{end} \\
\text{else begin } \{ \text{the parent exists on cite_list} \} \\
& \text{if (\text{cite\_loc} \neq \text{ilk\_info[lc\_cite\_loc]}) then hash\_cite\_confusion; } \\
& \text{cite\_parent\_ptr} \leftarrow \text{ilk\_info[cite\_loc]; } \\
& \text{if (\text{type\_list[cite\_parent\_ptr]} = \text{empty}) then} \\
& \text{begin nonexistent\_cross\_reference\_error; } \\
& \text{field_info[cite_ptr] \leftarrow missing; } \{ \text{remove the crossref ptr} \} \\
& \text{end} \\
\text{else begin } \{ \text{the parent exists in the database too} \} \\
& \text{field\_parent\_ptr} \leftarrow \text{cite\_parent\_ptr} \ast \text{num_fields} + \text{crossref_num;} \\
& \text{if (field_info[field\_parent\_ptr] \neq \text{missing) then} \langle \text{Complain about a nested cross reference} \ 282^* \rangle } \\
& \text{if ((-all\_entries) \land \text{(cite\_parent\_ptr} \geq \text{old\_num\_cites}) \land \text{(\text{cite\_info[cite\_parent\_ptr]} < \text{min\_crossrefs})}} \\
& \text{then} \\
& \text{field_info[cite_ptr] \leftarrow missing; } \{ \text{remove the crossref ptr} \} \\
& \text{end; } \\
& \text{end; } \\
& \text{incr(cite\_ptr);} \\
& \text{end;}
\end{align*}
\]

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.

\[\text{(Slide this cite key down to its permanent spot} \ 285^* \ \text{)} \equiv \]
\[
\begin{align*}
\text{begin if } & ((\text{cite\_xptr} + 1) \ast \text{num_fields} > \text{max_fields}) \text{ then} \\
& \text{confusion(‘field\_info\_index\_is\_out\_of\_range’); } \\
& \text{cite_list[cite\_xptr]} \leftarrow \text{cite_list[cite\_ptr]; } \text{type\_list[cite\_xptr]} \leftarrow \text{type\_list[cite\_ptr]; } \\
& \text{if (\text{find\_cite\_locs\_for\_this\_cite\_key(cite\_list[cite\_ptr]]) then} \text{cite\_key\_disappeared\_confusion; } \\
& \text{if (\text{cite\_hash\_found) \lor (cite\_loc} \neq \text{ilk\_info[lc\_cite\_loc]}) then hash\_cite\_confusion; } \\
& \text{ilk\_info[cite\_loc]} \leftarrow \text{cite\_xptr; } \\
& \text{field\_ptr} \leftarrow \text{cite\_xptr} \ast \text{num_fields; } \text{field\_end\_ptr} \leftarrow \text{field\_ptr} + \text{num_fields; } \text{tmp\_ptr} \leftarrow \text{cite\_ptr} \ast \text{num_fields; } \\
\text{while (field\_ptr < field\_end\_ptr) do} \\
& \text{begin field\_info[field\_ptr]} \leftarrow \text{field\_info[tmp\_ptr]; } \text{incr(field\_ptr);} \text{ incr(tmp\_ptr);} \\
& \text{end; } \\
& \text{end}
\end{align*}
\]

This code is used in section 283.
This module initializes all \texttt{int\_entry\_vars} of all entries to 0, the value to which all integers are initialized.

\begin{verbatim}
\langle\text{Initialize the int\_entry\_vars 287*}\rangle \equiv
\begin{verbatim}
begin entry_ints ← XTALLOC\((\text{num\_ent\_ints} + 1) \times (\text{num\_cites} + 1), \text{integer})\); int\_ent\_ptr ← 0;
while (int\_ent\_ptr < \text{num\_ent\_ints} \times \text{num\_cites}) do
    begin entry_ints\[int\_ent\_ptr\] ← 0; incr\(int\_ent\_ptr); 
    end;
end
\end{verbatim}
\end{verbatim}
This code is used in section 276.

This module initializes all \texttt{str\_entry\_vars} of all entries to the null string, the value to which all strings are initialized.

\begin{verbatim}
\langle\text{Initialize the str\_entry\_vars 288*}\rangle \equiv
\begin{verbatim}
begin entry_strs ← XTALLOC\((\text{num\_ent\_strs} + 1) \times (\text{num\_cites} + 1) \times (\text{ent\_str\_size} + 1), \text{ASCII\_code})\); str\_ent\_ptr ← 0;
while (str\_ent\_ptr < \text{num\_ent\_strs} \times \text{num\_cites}) do
    begin x\_entry\_strs\(str\_ent\_ptr\)(0) ← end\_of\_string; incr\(str\_ent\_ptr); 
    end;
end
\end{verbatim}
\end{verbatim}
This code is used in section 276.
### 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`).

```markdown
( Globals in the outer block )
lit_stack: ↑integer;  { the literal function stack }
lit_stk_type: ↑stk_type;  { their corresponding types }
lit_stk_ptr: lit_stack_loc;  { points just above the top of the stack }
cmd_chr_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 }
```

### Types in the outer block

- `lit_stack_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.

```markdown
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 )
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`$s$ 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_codes` to compare, so it might give “interesting” results when handling nonletters.

```plaintext
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`$s$ 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);
        exit: end;
```
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.

\[\textbf{procedure} \ push\_lit\_stk (\text{push}\_lt : \text{integer}; \text{push}\_\text{type} : \text{stk}\_\text{type});\]

\[\text{trace} \]

\[\text{var} \ dum\_ptr : \text{lit}\_\text{stk}\_\text{loc}; \{ \text{used just as an index variable} \}\]

\[\text{ecart} \]

\[\text{begin} \text{lit}\_\text{stack}[\text{lit}\_\text{stk}\_\text{ptr}] \leftarrow \text{push}\_\lt; \text{lit}\_\text{stk}\_\text{type}[\text{lit}\_\text{stk}\_\text{ptr}] \leftarrow \text{push}\_\text{type}; \]

\[\text{trace for} \ dum\_\text{ptr} \leftarrow 0 \text{ to } \text{lit}\_\text{stk}\_\text{ptr} \text{ do } \text{trace}\_pr(˝\_\_˝); \]

\[\text{trace}\_pr(˝\_\_˝); \]

\[\text{case } (\text{lit}\_\text{stk}\_\text{type}[\text{lit}\_\text{stk}\_\text{ptr}]) \text{ of} \]

\[\text{stk}\_\text{int}: \text{trace}\_pr(˝\"˝); \text{trace}\_pr\_\text{pool}\_\text{str}(\text{lit}\_\text{stack}[\text{lit}\_\text{stk}\_\text{ptr}])); \text{trace}\_pr\_\text{ln}(˝\"˝); \]

\[\text{end}; \]

\[\text{stk}\_\text{str}: \text{begin} \text{trace}\_pr(˝°˝); \text{trace}\_pr\_\text{pool}\_\text{str}(\text{hash}\_\text{text}[\text{lit}\_\text{stack}[\text{lit}\_\text{stk}\_\text{ptr}])); \text{trace}\_pr\_\text{ln}(˝°˝); \]

\[\text{end}; \]

\[\text{stk}\_\text{fn}: \text{begin} \text{trace}\_pr(˝\"˝); \text{trace}\_pr\_\text{pool}\_\text{str}(\text{hash}\_\text{text})); \text{trace}\_pr\_\text{ln}(˝\"˝); \]

\[\text{end}; \]

\[\text{stk}\_\text{field}\_\text{missing}: \text{begin} \text{trace}\_pr(˝\text{missing\_field\_}˝); \text{trace}\_pr\_\text{pool}\_\text{str}(\text{lit}\_\text{stack}[\text{lit}\_\text{stk}\_\text{ptr}])); \text{trace}\_pr\_\text{ln}(˝\"˝); \]

\[\text{end}; \]

\[\text{stk}\_\text{empty}: \text{trace}\_pr\_\text{ln}(˝\text{a\_bad\_literal}--\text{popped}\_\text{from}\_\text{an}\_\text{empty}\_\text{stack}´); \]

\[\text{othercases unknown\_literal\_confusion} \]

\[\text{endcases}; \]

\[\text{ecart} \]

\[\text{if } (\text{lit}\_\text{stk}\_\text{ptr} = \text{lit}\_\text{stk}\_\text{size}) \text{ then} \]

\[\text{begin} \text{BIB}\_\text{XRETA} \text{LLOC\_NOSET} (˝\text{lit}\_\text{stack}´, \text{lit}\_\text{stack}, \text{integer}, \text{lit}\_\text{stk}\_\text{size}, \]

\[\text{lit}\_\text{stk}\_\text{size} + \text{LIT}\_\text{STK}\_\text{SIZE}); \]

\[\text{BIB}\_\text{XRETA} \text{LLOC}(˝\text{lit}\_\text{stk}\_\text{type}´, \text{lit}\_\text{stk}\_\text{type}, \text{stk}\_\text{type}, \text{lit}\_\text{stk}\_\text{size}, \text{lit}\_\text{stk}\_\text{size} + \text{LIT}\_\text{STK}\_\text{SIZE}); \]

\[\text{end}; \]

\[\text{incr}(\text{lit}\_\text{stk}\_\text{ptr}); \]

\[\text{end}; \]

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

This code is used in section 12.
This procedure adds to the output buffer the given string in \textit{str\_pool}. It assumes the global variable \textit{out\_buf\_length} gives the length of the current string in \textit{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 \texttt{.bbl} file. It breaks a line only at a \textit{white\_space} character, and when it does, it adds two \textit{spaces} to the next output line.

\begin{verbatim}
(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\_ptr\_found: boolean;  { a suitable \textit{white\_space} character }
unbreakable\_tail: boolean;  { as it contains no \textit{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[\text{out\_buf\_ptr}] ← str\_pool[\text{p\_ptr1}]; incr(\text{p\_ptr1}); incr(\text{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;

end

end

\end{verbatim}

This code is used in section 325.

This module pushes the string given by the field onto the literal stack unless it’s \textit{missing}, in which case it pushes a special value onto the stack.

\begin{verbatim}
(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\_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

end

This code is used in section 325.

This module adds the string given by a \textit{str\_entry\_var} to \textit{str\_pool} via the execution buffer and pushes it onto the literal stack.

\begin{verbatim}
(Execute a \textit{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.

\end{verbatim}
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).

\[
\langle \text{Execute a } \texttt{str\_global\_var} \texttt{ 330*} \rangle \equiv \\
\begin{array}{l}
\text{begin } \texttt{str\_glb\_ptr } \leftarrow \texttt{fn\_info[ex\_fn\_loc]}; \\
\text{if } (\texttt{glb\_str\_ptr[\texttt{str\_glb\_ptr}] > 0}) \text{ then } \{} \text{ we’re dealing with a static string } \}
\quad \text{push\_lit\_stk(\texttt{glb\_str\_ptr[\texttt{str\_glb\_ptr}], stk\_str})}
\text{else begin } \texttt{str\_room(\texttt{glb\_str\_end[\texttt{str\_glb\_ptr}]})}; \quad \texttt{glob\_chr\_ptr } \leftarrow \text{ 0;}
\quad \text{while } (\texttt{glob\_chr\_ptr < \texttt{glb\_str\_end[\texttt{str\_glb\_ptr}]}) \text{ do } \{} \text{ copy the string } \}
\quad \text{begin } \texttt{append\_char(x\_global\_strs(\texttt{str\_glb\_ptr})(\texttt{glob\_chr\_ptr})); } \quad \texttt{incr(\texttt{glob\_chr\_ptr});}
\quad \text{end;}
\quad \text{push\_lit\_stk(\texttt{make\_string, stk\_str}); } \text{ and push it onto the stack } \}
\text{end;}
\end{array}
\]

This code is used in section 325.
334* It’s time for us to insert more pre-defined strings into \texttt{str_pool} (and thus the hash table) and to insert the \texttt{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 \texttt{longest_pds} characters long. The \texttt{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 \texttt{log_file} hasn’t been opened yet.

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

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

\begin{verbatim}
(Globals in the outer block \texttt{2*}) +≡
   \texttt{s_null: str_number; \{ the null string \}}
   \texttt{s_default: str_number; \{ default.type, for unknown entry types \}}
   \texttt{s_t: str_number; \{ t, for title_lowers case conversion \}}
   \texttt{s_l: str_number; \{ l, for all_lowers case conversion \}}
   \texttt{s_u: str_number; \{ u, for all_uppers case conversion \}}
   \texttt{s_preamble: \texttt{str_number; \{ for the preamble$ built_in function \}}
\end{verbatim}
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.

\textbf{define} name\_buf ≡ sv\_buffer \quad \{ an alias, a buffer for manipulating names \}

( Globals in the outer block 2\*) +≡

\textbf{pop\_lit1, pop\_lit2, pop\_lit3: integer;} \quad \{ stack literals \}

\textbf{pop\_typ1, pop\_typ2, pop\_typ3: stk\_type;} \quad \{ stack types \}

\textbf{sp\_ptr: pool\_pointer;} \quad \{ for manipulating str\_pool strings \}

\textbf{sp\_xptr1, sp\_xptr2: pool\_pointer;} \quad \{ more of the same \}

\textbf{sp\_end: pool\_pointer;} \quad \{ marks the end of a str\_pool string \}

\textbf{sp\_length, sp2\_length: pool\_pointer;} \quad \{ lengths of str\_pool strings \}

\textbf{sp\_brace\_level: integer;} \quad \{ for scanning str\_pool strings \}

\textbf{ex\_buf\_xptr, ex\_buf\_yptr: buf\_pointer;} \quad \{ extra ex\_buf locations \}

\textbf{control\_seq\_loc: hash\_loc;} \quad \{ hash-table loc of a control sequence \}

\textbf{preceding\_white: boolean;} \quad \{ used in scanning strings \}

\textbf{and\_found: boolean;} \quad \{ to stop the loop that looks for an “and” \}

\textbf{num\_names: integer;} \quad \{ for counting names \}

\textbf{name\_bf\_ptr: buf\_pointer;} \quad \{ general name\_buf location \}

\textbf{name\_bf\_xptr, name\_bf\_yptr: buf\_pointer;} \quad \{ and two more \}

\textbf{nm\_brace\_level: integer;} \quad \{ for scanning name\_buf strings \}

\textbf{name\_tok: ↑buf\_pointer;} \quad \{ name-token ptr list \}

\textbf{name\_sep\_char: ↑ASCII\_code;} \quad \{ token-ending chars \}

\textbf{num\_tokens: buf\_pointer;} \quad \{ this counts name tokens \}

\textbf{token\_starting: boolean;} \quad \{ used in scanning name tokens \}

\textbf{alpha\_found: boolean;} \quad \{ used in scanning the format string \}

\textbf{double\_letter, end\_of\_group, to\_be\_written: boolean;} \quad \{ the same \}

\textbf{first\_start: buf\_pointer;} \quad \{ start-ptr into name\_tok for the first name \}

\textbf{first\_end: buf\_pointer;} \quad \{ end-ptr into name\_tok for the first name \}

\textbf{last\_end: buf\_pointer;} \quad \{ end-ptr into name\_tok for the last name \}

\textbf{von\_start: buf\_pointer;} \quad \{ start-ptr into name\_tok for the von name \}

\textbf{von\_end: buf\_pointer;} \quad \{ end-ptr into name\_tok for the von name \}

\textbf{jr\_end: buf\_pointer;} \quad \{ end-ptr into name\_tok for the jr name \}

\textbf{cur\_token, last\_token: buf\_pointer;} \quad \{ name\_tok ptrs for outputting tokens \}

\textbf{use\_default: boolean;} \quad \{ for the inter-token intra-name part string \}

\textbf{num\_commas: buf\_pointer;} \quad \{ used to determine the name syntax \}

\textbf{comma1, comma2: buf\_pointer;} \quad \{ ptrs into name\_tok \}

\textbf{num\_text\_chars: buf\_pointer;} \quad \{ special characters count as one \}
This module removes all leading white space (and sep_chars), and trailing white space (and sep_chars) and commas. It complains for each trailing comma.

(Remove leading and trailing junk, complaining if necessary 388*)

\begin{enumerate}
\item while \((ex\_buf\_ptr > ex\_buf\_xptr)\) do \{ now remove trailing stuff \}
\item case \([\text{lex\_class}\[\text{ex\_buf}\[\text{ex\_buf\_ptr - 1}\]_\text{ex\_buf\_ptr}]\] of \text{white\_space}, \text{sep\_char}: \text{decr}(\text{ex\_buf\_ptr}); \}
\item othercases if \((\text{ex\_buf}[\text{ex\_buf\_ptr - 1}]=\text{comma})\) then \begin{enumerate}
\item begin \(\text{print}(''\text{Name}, pop\_lit2 : 0, ''\text{un}\_\text{in}\_\text{ed}''); \text{print\_pool\_str}(\text{pop\_lit3}); \text{print}(''\text{has} a comma at the end''); \text{bst\_ex\_warn\_print}; \text{decr}(\text{ex\_buf\_ptr}); \}
\item end
\end{enumerate}
\item else goto \text{loop1\_exit}
\item endcases;
\item \text{loop1\_exit}: end
\end{enumerate}
This module finds the substring as described in the last section, and slides it into place in the string pool, if necessary.

\[\text{Form the appropriate substring 438}^* \equiv \]
\[
\begin{align*}
\text{begin if } (\text{pop_lit2} > 0) \text{ then} \\
\quad \text{begin if } (\text{pop_lit1} > \text{sp_length} - (\text{pop_lit2} - 1)) \text{ then } & \text{pop_lit1} \leftarrow \text{sp_length} - (\text{pop_lit2} - 1); \\
\quad & \text{sp_ptr} \leftarrow \text{str_start}[\text{pop_lit3}] + (\text{pop_lit2} - 1); \\
\quad & \text{sp_end} \leftarrow \text{sp_ptr} + \text{pop_lit1}; \\
\quad \text{if } (\text{pop_lit2} = 1) \text{ then} \\
\quad \quad \text{if } (\text{pop_lit3} \geq \text{cmd_str_ptr}) \text{ then} \{ \text{no shifting—merely change pointers} \} \\
\quad \quad \quad \text{begin } \text{str_start}[\text{pop_lit3} + 1] \leftarrow \text{sp_end}; \\
\quad \quad \quad \text{unflush_string}; \\
\quad \quad \quad \text{incr} (\text{lit_stk_ptr}); \\
\quad \quad \text{return}; \\
\quad \text{end}; \\
\quad \text{end}; \\
\text{end} \\
\text{else} \{ -\text{ex_buf_length} \leq \text{pop_lit2} < 0 \} \\
\text{begin } \text{pop_lit2} \leftarrow -\text{pop_lit2}; \\
\text{if } (\text{pop_lit1} > \text{sp_length} - (\text{pop_lit2} - 1)) \text{ then } & \text{pop_lit1} \leftarrow \text{sp_length} - (\text{pop_lit2} - 1); \\
\text{sp_end} \leftarrow & \text{str_start}[\text{pop_lit3} + 1] - (\text{pop_lit2} - 1); \\
\text{sp_ptr} \leftarrow & \text{sp_end} - \text{pop_lit1}; \\
\text{end}; \\
\text{str_room} & (\text{sp_end} - \text{sp_ptr}); \\
\text{while } (\text{sp_ptr} < \text{sp_end}) \text{ do } \{ \text{shift the substring} \} \\
\quad \text{begin } & \text{append_char} (\text{str_pool}[\text{sp_ptr}]); \\
\quad & \text{incr} (\text{sp_ptr}); \\
\quad \text{end}; \\
\text{push_lit_stk} & (\text{make_string}, \text{stk_str}); \{ \text{and push it onto the stack} \}
\end{align*}
\]
This code is used in section 437.

This module finds the prefix as described in the last section, and appends any needed matching \textit{right_braces}.

\[\text{Form the appropriate prefix 444}^* \equiv \]
\[
\begin{align*}
\text{begin } & \text{sp_ptr} \leftarrow \text{str_start}[\text{pop_lit2}]; \\
& \text{sp_end} \leftarrow \text{str_start}[\text{pop_lit2} + 1]; \{ \text{this may change} \} \\
\langle \text{Scan the appropriate number of characters 445} \rangle; \\
& \text{str_room} (\text{sp_brace_level} + \text{sp_end} - \text{sp_ptr}); \\
\text{if } & (\text{pop_lit2} \geq \text{cmd_str_ptr}) \text{ then} \{ \text{no shifting—merely change pointers} \} \\
& \text{pool_ptr} \leftarrow \text{sp_end} \\
\text{else while } (\text{sp_ptr} < \text{sp_end}) \text{ do } \{ \text{shift the substring} \} \\
\quad & \text{begin } \text{append_char} (\text{str_pool}[\text{sp_ptr}]); \\
\quad & \text{incr} (\text{sp_ptr}); \\
\quad \text{end}; \\
\text{while } (\text{sp_brace_level} > 0) \text{ do } \{ \text{add matching \textit{right_braces} } \} \\
\quad & \text{begin } \text{append_char} (\text{right_brace}); \\
\quad & \text{decr} (\text{sp_brace_level}); \\
\quad \text{end}; \\
\text{push_lit_stk} & (\text{make_string}, \text{stk_str}); \{ \text{and push it onto the stack} \}
\end{align*}
\]
This code is used in section 443.
459* This prints information gathered while reading the .bst and .bib files.

\begin{verbatim}
⟨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('has entry strings'); (Print entry strings 460*);
trace_pr('has entry integers'); (Print entry integers 461*);
trace_pr('and has fields'); (Print fields 462*);
end
\end{verbatim}

This code is used in section 458.

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

\begin{verbatim}
⟨Print entry strings 460*⟩≡
begin if (num_ent_strs = 0) then trace_pr('undefined');
  else if (¬read_completed) then trace_pr('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(''); incr(str_ent_ptr);
           end;
        end;
  end
end
\end{verbatim}

This code is used in section 459*.

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

\begin{verbatim}
⟨Print fields 462*⟩≡
begin if (¬read_performed) then trace_pr('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('');
         no_fields ← false;
         end;
       end;
     end;
     if (no_fields) then trace_pr('missing');
  end
end
\end{verbatim}

This code is used in section 459*.
§467* System-dependent changes.

\[\text{define } \text{argument}\_\text{is}(\#) \equiv (\text{strcmp}(\text{long}\_\text{options}[\text{option}\_\text{index}], \#) = 0)\]

(Define \text{parse arguments} \ 467*) \equiv

\text{procedure} \ \text{parse_arguments};
\begin{align*}
\text{const} \ n\_\text{options} &= 4; \quad \{\text{Pascal won’t count array lengths for us.}\} \\
\text{var} \ \text{long}\_\text{options} : \text{array}[0..n\_\text{options}] \text{of} \ \text{getopt}\_\text{struct}; \\
\text{getopt}_\text{return}_\text{val} : \text{integer}; \ \text{option}\_\text{index} : \text{c}\_\text{int}\_\text{type}; \ \text{current}\_\text{option} : 0..n\_\text{options};
\end{align*}

\begin{align*}
\text{begin} \quad \langle \text{Initialize the option variables} \ 470* \rangle; \\
\langle \text{Define the option table} \ 468* \rangle;
\end{align*}

\begin{align*}
\text{repeat} \ & \text{getopt}_\text{return}_\text{val} \leftarrow \text{getopt}\_\text{long}\_\text{only}(\text{argc}, \text{argv}, \text{\'\text{-}\text{-}\text{\textperiodcentered},} \text{long}\_\text{options}, \text{address\_of}(\text{option}\_\text{index})); \\
\text{if} \ & \text{getopt}_\text{return}_\text{val} = -1 \text{ then} \\
\begin{align*}
\text{begin} \ & \text{do\_nothing}; \quad \{\text{End of arguments; we exit the loop below.}\} \\
\text{end} \ & \text{else if} \ \text{getopt}_\text{return}_\text{val} = \text{\'?\'} \text{ then} \\
\begin{align*}
\text{begin} \ & \text{usage}(\text{my}_\text{name}); \\
\text{end} \ & \text{else if} \ \text{argument\_is(\text{\textquoteleft min\_crossrefs\textquoteleft})} \text{ then} \\
\begin{align*}
\text{begin} \ & \text{min\_crossrefs} \leftarrow \text{atoi}(\text{optarg}); \\
\text{end} \ & \text{else if} \ \text{argument\_is(\text{\textquoteleft help\textquoteleft})} \text{ then} \\
\begin{align*}
\text{begin} \ & \text{usage\_help}(\text{BIBTEX\_HELP}, \text{nil}); \\
\text{end} \ & \text{else if} \ \text{argument\_is(\text{\textquoteleft version\textquoteleft})} \text{ then} \\
\begin{align*}
\text{begin} \ & \text{print\_version\_and\_exit}(\text{banner}, \text{\'\text{Oren\_Patashnik\textquoteleft}, \text{nil}, \text{nil}); \\
\text{end} \ & \text{until} \ \text{getopt}_\text{return}_\text{val} = -1; \quad \{\text{Now optind is the index of first non-option on the command line. We must have one remaining argument.}\} \\
\text{if} \ (\text{optind} + 1 \neq \text{argc}) \text{ then} \\
\begin{align*}
\text{begin} \ & \text{write\_ln}(\text{stderr, my}_\text{name}, \text{\textquoteleft\text{I\_Need\_exactly\_one\_file\_argument.}\textquoteleft}); \ \text{usage}(\text{my}_\text{name}); \\
\text{end} \\
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}

This code is used in section 10*.

\section*{468*}

Here is the first of the options we allow.

\begin{align*}
\langle \text{Define the option table} \ 468* \rangle \equiv \\
\text{current}\_\text{option} \leftarrow 0; \ \text{long}\_\text{options}[0].\text{name} \leftarrow \text{\textquoteleft terse\textquoteleft}; \ \text{long}\_\text{options}[0].\text{has\_arg} \leftarrow 0; \\
\text{long}\_\text{options}[0].\text{flag} \leftarrow \text{address\_of}(\text{verbose}); \ \text{long}\_\text{options}[0].\text{val} \leftarrow 0; \ \text{incr}(\text{current}\_\text{option});
\end{align*}

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

This code is used in section 467*.

\section*{469*}

The global variable \text{verbose} determines whether or not we print progress information.

\begin{align*}
\langle \text{Globals in the outer block} \ 2* \rangle + \equiv \\
\text{verbose} : \text{c}\_\text{int}\_\text{type};
\end{align*}

\section*{470*}

Start off \text{true}, to match the default behavior.

\begin{align*}
\langle \text{Initialize the option variables} \ 470* \rangle \equiv \\
\text{verbose} \leftarrow \text{true};
\end{align*}

See also section 473*.

This code is used in section 467*.
Here is an option to change the minimum number of cross-refs required for automatic cite_list inclusion.

\[
\text{\begin{verbatim}
⟨Define the option table 468\rangle +≡
long_options[\text{current_option}].name ← 'min-crossrefs'; long_options[\text{current_option}].has_arg ← 1;
long_options[\text{current_option}].flag ← 0; long_options[\text{current_option}].val ← 0; incr(\text{current_option});
\end{verbatim}}
\]

 Globals in the outer block 2 \(\rangle +≡
min_crossrefs: integer;

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

\[
\text{\begin{verbatim}
⟨Initialize the option variables 470\rangle +≡
min_crossrefs ← 2;
\end{verbatim}}
\]

One of the standard options.

\[
\text{\begin{verbatim}
⟨Define the option table 468\rangle +≡
long_options[\text{current_option}].name ← 'help'; long_options[\text{current_option}].has_arg ← 0;
long_options[\text{current_option}].flag ← 0; long_options[\text{current_option}].val ← 0; incr(\text{current_option});
\end{verbatim}}
\]

Another of the standard options.

\[
\text{\begin{verbatim}
⟨Define the option table 468\rangle +≡
long_options[\text{current_option}].name ← 'version'; long_options[\text{current_option}].has_arg ← 0;
long_options[\text{current_option}].flag ← 0; long_options[\text{current_option}].val ← 0; incr(\text{current_option});
\end{verbatim}}
\]

An element with all zeros always ends the list.

\[
\text{\begin{verbatim}
⟨Define the option table 468\rangle +≡
long_options[\text{current_option}].name ← 0; long_options[\text{current_option}].has_arg ← 0;
long_options[\text{current_option}].flag ← 0; long_options[\text{current_option}].val ← 0;
\end{verbatim}}
\]

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.

\[
\text{\begin{verbatim}
define setup_bound_var(#) ≡ bound_default ← #; setup_bound_var_end
define setup_bound_var_end(#) ≡ bound_name ← #; setup_bound_var_end_end
define setup_bound_var_end_end(#) ≡ setup_bound_variable(address_of(#), bound_name, bound_default);
if # < bound_default then # ← bound_default
\end{verbatim}}
\]

(Procedures and functions for about everything 12 \rangle +≡

procedure setup_params;
\text{\begin{verbatim}
var bound_default: integer; \{ for setup \}
bound_name: constcstring; \{ for setup \}
begin kps.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);
setup_bound_var(MAX_PRINT_LINE)(max_print_line)(max_print_line);
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;
\end{verbatim}}
\]
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 \( \geq 128 \)).

\[
\text{define } \text{primes} \equiv \text{hash}._{\text{next}} \quad \{ \text{array holding the first } k \text{ primes} \}
\]

\[
\text{define } \text{mult} \equiv \text{hash}._{\text{text}} \quad \{ \text{array holding odd multiples of the first } o \text{ primes} \}
\]

\(\langle\text{Procedures and functions for about everything} \quad 12\rangle\) +

\section{procedure compute_hash_prime;}
\begin{verbatim}
  var hash\_want: integer;  \{ 85\% of hash\_size \}
    k: integer;  \{ number of prime numbers }\_i in primes \}
    j: integer;  \{ a prime number candidate \}
    o: integer;  \{ number of odd multiples of primes in mult \}
    square: integer;  \{ }^2 \}
    n: integer;  \{ loop index \}
    j\_prime: boolean;  \{ is j a prime? \}
  begin
    hash\_want \leftarrow (hash\_size \div 20) \ast 17;  j \leftarrow 1;  k \leftarrow 1;  hash\_prime \leftarrow 2;  primes[k] \leftarrow hash\_prime;  o \leftarrow 2;  square \leftarrow 9;
    while hash\_prime < hash\_want do
      begin
        repeat
          j \leftarrow j + 2;
        if j = square then
          begin
            mult[o] \leftarrow j;  j \leftarrow j + 2;  incr(o);  square \leftarrow primes[o] \ast primes[o];
          end;
        n \leftarrow 2;  j\_prime \leftarrow true;
        while (n < o) \land j\_prime do
          begin
            while mult[n] < j do  mult[n] \leftarrow mult[n] + 2 \ast primes[n];
            if mult[n] = j then  j\_prime \leftarrow false;
            incr(n);
          end;
        until j\_prime;
        incr(k);  hash\_prime \leftarrow j;  primes[k] \leftarrow hash\_prime;
      end;
    end;
\end{verbatim}

478*
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”.


- help: 474*
- min-crossrefs: 471*
- terse: 468*
- version: 475*
a_close: 142, 151* 223* 455.
a_minus: 331.
a_open_in: 106* 123* 127* 141*.
a_open_in_with_dirname: 141*.
a_open_out: 106*.
add a built-in function: 331, 333, 334*, 341, 342.
add_database_cite: 264, 265* 272.
add_extension: 60* 106* 107.
add_out_pool: 322* 454.
address_of: 467*, 468*, 477*.
ae_width: 35, 453.
alpha_file: 10*, 36, 47, 51, 82, 104, 117*, 123*, 124, 242*.
alpha_found: 344*, 403, 405.
already_seen_function_print: 169.
and_found: 344*, 384, 386.
any_value: 9, 138*, 227.
appeend_int_char: 197, 198*.
area: 61*
argument_is: 467*.
argv: 467*, 477*.
argv1: 301*.
argv2: 301*.
ASCII code: 21.
at_bib_command: 219*, 221, 236, 239, 259, 261.
at_sign: 29, 218, 237, 238.
atoi: 467*.
aux_bib_data_command: 116, 120.
aux_bib_style_command: 116, 126.
aux_citation_command: 116, 132.
aux_command_ik: 64* 79, 116.
aux_done: 109, 110*, 142.
aux_end_err: 144, 145.
aux_end1_err_print: 144.
aux_end2_err_print: 144.
aux_err: 111, 122.
aux_err_ik: 112, 120, 126.
aux_err_ik: 112.
aux_err_no_right_brace: 113, 120, 126, 132, 139.
aux_err_no_right_brace: 113.
aux_err_print: 111.
aux_err_stuff_after_right_brace: 114, 120, 126, 132, 139.
aux_err_stuff_after_right_brace: 114.
aux_err_white_space_in_argument: 115, 120, 126, 132, 139.
aux_err_white_space_in_argument: 115.
aux_extension: 139, 140.
aux_file: 104.
aux_file_ik: 64*, 107, 140.
aux_found: 97*, 100*, 103.
aux_input_command: 116, 139.
aux_ln_stack: 104.
aux_name_length: 97*98, 100*103, 106*107.
aux_not_found: 97*98, 99, 100*.
aux_number: 104, 105.
aux_ptr: 104, 106*140, 141*142.
aux_stack_size: 14*104, 105, 109, 140.

   \@input: 139.
   \bibdata: 120.
   \bibstyle: 126.
   \citation: 132.

b: 331.
b_add_period: 331, 334*.
b_call_type: 331, 334*.
b_change_case: 331, 334*.
b_chr_to_int: 331, 334*.
b_cite: 331, 334*.
b_concatenate: 331, 334*.
b_default: 182, 331, 339, 363.
b_duplicate: 331, 334*.
b_empty: 331, 334*.
b_equal: 331, 334*.
b_formal_name: 331, 334*.
b_gat: 331.
b_gets: 331, 334*.
b_greater_than: 331, 334*.
b_if: 331, 334*.
b_int_to_chr: 331, 334*.
b_int_to_str: 331, 334*.
b_less_than: 331, 334*.
b_minus: 331, 334*.
b_missing: 331, 334*.
b_newline: 331, 334*.
b_num_names: 331, 334*.
b_plus: 331, 334*.
b_pop: 331, 334*.
b_preamble: 331, 334*.
b_purge: 331, 334*.
b_quote: 331, 334*.
b_skip: 331, 334* 339.
b_stack: 331, 334*.
b_substring: 331, 334*.
b_swap: 331, 334*.
b_text_length: 331, 334*.
b_text_prefix: 331, 334*.
b_top_stack: 331, 334*.
b_type: 331, 334*.
b_warning: 331, 334*.
b_while: 331, 334*.
b_width: 331, 334*.
b_write: 331, 334*.

bad: 13*16*17*302.
bad_argument_token: 177, 179, 204, 213.
banner: 1*10*467.
\bbl_file: 104, 106*151*321.
\bbl_line_num: 147, 151*321.
begin: 4*
b_f_ptr: 56, 62, 63, 95.
\bib_brace_level: 247, 253, 254, 255, 256, 257.
\bib_cmd_confusion: 239, 240, 262.
\bib_command_ilk: 64*79, 238.
\bib_equal_sign_expected_err: 231, 246, 275.
\bib_equal_sign_print: 231.
\bib_err_print: 221.
\bib_field_too_long_err: 233.
\bib_field_too_long_print: 233.
\bib_file: 10*117*123*242*.
\bib_file_ilk: 64*123*.
\bib_id_print: 235.
\bib_identifier_scan_check: 235, 238, 244, 259, 275.
\bib_line_num: 219*220, 223*228, 237, 252, 455.
\bib_list: 10*117*118*119, 123*242*.
\bib_ln_num_print: 220, 221, 222.
\bib_makestring: 38*141*.
\bib_number: 117*118*219*.
\bib_one_of_two_expected_err: 230, 242*244, 266, 274.
\bib_one_of_two_print: 230.
\bib_ptr: 117*119, 123*145, 223*457.
\bib_seen: 117*119, 120, 145.
\bib_unbalanced_braces_err: 232, 254, 256.
\bib_unbalanced_braces_print: 232.
\bib_warn: 222.
\bib_warnnewline: 222, 234, 263*273.
\bib_warn_print: 222.
\BIBXRETALLOC: 46*53*123*138*188*200*216*226*242*307*.
\BIBXRETALLOC_NOSET: 46*123*138*216*242*307*.
\BIBXRETALLOC_STRING: 216*.
biblical_procreation: 331.
\bibtex: 10*.

BibTeX capacity exceeded: 44.
    buffer size: 46*47*197, 319, 320, 414, 416, 417.
    file name size: 58*59*60*61*.
    hash size: 71*.
number of `.aux` files: 140.
number of `.bib` files: 123*
number of cite keys: 138*
number of string global-variables: 216*
number of strings: 54.
output buffer size: 322*
pool size: 53*
single function space: 188*
total number of fields: 226*
total number of integer entry-variables: 287*
total number of string entry-variables: 288*
wizard-defined function space: 200*

BibTeX documentation: 1*.

BibTeX: 1*

BIBTEX_HELP: 467*

`blt_in_loc`: 331, 335, 465.
`blt_in_num`: 335.
`blt_in_ptr`: 331, 465.
`blt_in_range`: 331, 332, 335.
`boolean`: 10*47, 56, 57, 65, 66, 83, 84, 85, 86, 87, 88, 92, 93, 94, 117*121, 124, 129*138, 139, 152, 161*163, 177, 219*228, 249, 250, 252, 253, 278, 290*301, 322, 344, 365, 397, 418, 478*.

bottom up: 12.

`bound_default`: 477*.

`bound_name`: 477*.


`brace_bal_one_letters_complaint`: 405, 406.

`brace_unbalanced_complaint`: 367, 368, 369, 402.

`break_pt_found`: 322, 323, 324.

`break_ptr`: 322*, 323.


`bst_command_ilk`: 64*, 79, 154.

`bst_done`: 146, 149, 151*.

`bst_entry_command`: 155, 170.


`bst_err_print_and_look_for_blank_line`: 149.

`bst_err_print_and_look_for_blank_line_return`: 149, 169, 177.


`bst_ex_warn_print`: 293, 312, 388*389.

`bst_execute_command`: 155, 178.

`bst_file`: 124, 127*, 149, 151*.

`bst_file_ilk`: 64*, 127*.


`bst_function_command`: 155, 180.

`bst_get_and_check_left_brace`: 167, 171, 173, 175, 178, 180, 181, 201, 203, 206, 208, 212, 215.

`bst_get_and_check_right_brace`: 168, 178, 181, 203, 206, 208, 212.

`bst_id_print`: 166.

`bst_identifier_scan`: 166, 171, 173, 175, 178, 181, 201, 203, 206, 212, 215.

`bst_integers_command`: 155, 201.

`bst_iterate_command`: 155, 203.

`bst_left_brace_print`: 167.

`bst_line_num`: 147, 148, 149, 151*.

`bst_lin_num_print`: 148, 149, 150, 183, 293.

`bst_macro_command`: 155, 205.

`bst_mild_ex_warn`: 294, 368.

`bst_mild_ex_warn_print`: 294, 356.

`bst_read_command`: 155, 211.

`bst_reverse_command`: 155, 212.

`bst_right_brace_print`: 168.

`bst_seen`: 124, 125, 126, 145.

`bst_sort_command`: 155, 214.

`bst_str`: 124, 125, 127*128*145, 151*.

`bst_string_size_exceeded`: 356, 357*.

`bst_strings_command`: 155, 215.

`bst_warn`: 150, 170, 294.

`bst_warn_print`: 150.

`bst_1print_string_size_exceeded`: 356.

`bst_1print_string_size_exceeded`: 356.

`buf`: 56, 62, 63, 68*69, 70*, 71*.

`buf_pointer`: 10*41, 42*43, 46*, 56, 62, 63, 68*, 80, 82, 95, 187*198*290*, 322*, 344*418.


`BUF_SIZE`: 10*14*46*.

`buf_size`: 10*14*17*41*46*47*, 197, 217, 224, 231, 232*322*, 414, 416, 417.

`buf_type`: 41*42*43, 46*, 56, 62, 63, 68*, 198*290*.


bunk, history: 466.
c_int_type: 467, 469.
case mismatch: 132.
case mismatch errors: 135, 273.
case_conversion_confusion: 372, 373, 375, 376.
case_difference: 62, 63.
Casey Stengel would be proud: 401.
char: 23, 97.
char_ptr: 301.
char_value: 91, 92, 93.
char_width: 34, 35, 450, 451, 452, 453.
character set dependencies: 23, 25, 26, 27, 32, 33, 35.
char1: 83, 84, 85, 86, 87, 90, 230, 301.
char2: 85, 86, 87, 90, 230, 301.
char3: 87, 90.
check_cite_overflow: 136, 138, 265.
check_cmd_line: 101.
check_command_execution: 296, 297, 298, 317.
check_field_overflow: 225, 226, 265.
check_for_and_compress_bib_white_space: 252, 253, 256, 257.
child entry: 277.
chr: 23, 24, 27, 58, 60.
citation_seen: 129, 131, 132, 145.
cite_already_set: 236, 272.
cite_found: 129.
cite_hash_found: 219, 278, 279, 285.
cite_ilk: 64, 135, 136, 264, 269, 272, 273, 278.
cite_key_disappeared_confusion: 270, 271, 285.
cite_number: 129, 130, 138, 161, 265, 290, 300, 301, 303.
cite_parent_ptr: 161, 277, 279, 282.
hash_size: 10*15*16*17*69, 71*477*478*
HASH_SIZE: 15*477*
hash_text: 10*64*65*67, 70*71*75, 107, 123*127*
136, 140, 169, 182, 194, 207, 209, 245, 261, 262,
263*265*269, 277*297, 298, 307*311, 313, 325,
327*339, 447, 459*463, 465, 478*
hash_used: 64*65*67, 71*
hash_want: 478*
history: 10*18, 19, 20, 466.
hyphen: 20, 32*
i: 51, 56, 62, 63, 77*82, 121*
id_class: 30, 33*, 90.
id_null: 89, 90, 166, 235.
id_scanning_confusion: 165, 166, 235.
id_type: 30, 31.
ifdef: 4*
ifndef: 4*
ilk: 64*65*, 68*, 70*, 71*, 77*.
ilk_info: 10*64*65*67, 78, 79, 116, 135, 136, 154,
161*207, 209, 238, 245, 260, 262, 264, 265*267,
illegal: 31, 32*.
illegal_id_char: 31, 33*, 90.
illegal_literal_confusion: 310, 311, 312, 313.
impl_fn_loc: 187*, 194*.
impl_fn_num: 194, 195, 196.
incred: 9, 18, 47*, 53*, 54, 55, 56, 57, 58*, 60*, 69, 71*
82, 83, 84, 85, 86, 88, 90, 92, 93, 94, 95,
98, 99, 107, 110*, 120, 123*126, 132, 133, 136,
138*,139, 140, 149, 152, 162, 167, 168, 171, 172,
173, 174, 175, 176, 187*, 188*, 190, 191, 192, 194,
197, 198*, 200*, 201, 209, 211, 215, 216*, 223*, 225,
227, 228, 237, 238, 242*, 244, 246, 249, 251*252,
253, 254, 255, 256, 257, 258, 260, 262, 264, 265*,
266, 267, 270, 274, 275, 277*, 279*, 283*, 285*,
286, 287*, 288*, 289, 297, 301*, 306, 307*, 308, 318,
319, 320, 321, 322*, 323, 324, 326, 330*, 340, 341,
351, 352, 353, 357*, 359*, 362, 370, 371, 374, 379,
381, 383, 384, 385, 389, 390, 391, 392, 393,
394, 396, 397, 398, 400, 402, 403, 404, 405,
411, 412, 413, 414, 415, 416, 417, 418, 419,
427, 429, 431, 432, 433, 438*, 440, 442, 444*,
445, 451, 452, 457, 458, 460*, 461, 462*, 463,
init_command_execution: 296, 297, 298, 316.
initialize: 10*12, 13*336.
n Innocent bystander: 300.
input_in: 41*, 47*, 80, 110*, 149, 152, 228, 237, 252.
insert_fn_loc: 188*, 190, 191, 193, 194, 199, 200*.
insert_it: 68*
insert_ptr: 303, 304.

int: 198*
int_begin: 198*
int_buf: 197, 198*
int_end: 198*
int_ent_loc: 160*, 161*
int_ent_ptr: 161*, 287*, 461.
int_entry_var: 156, 158, 159, 160*, 161*, 162, 170,
int_global_var: 156, 158, 159, 201, 202, 325,
331, 340, 354.
int literal: 29, 156, 158, 159, 189, 190, 325.
int_ptr: 197, 198*.
int_tmp_val: 198*.
int_to_ASCII: 194, 197, 198*423.
int_zptr: 198*.
integer: 10*16*, 19, 23*, 34, 37*, 41*, 42*, 43, 49*, 64*,
65*, 68*, 78, 91, 128*, 121*, 130*, 147,
287*, 290*, 291*, 307*, 309, 311, 312, 313, 314,
331, 343, 447*, 473*, 477*, 478*.
integer_ilk: 64*, 156, 190.
invalid code: 26, 32*, 216*.
j: 56, 68*, 478*.
j_prime: 478*.
jk: 66, 68*, 478*.
kpse_bib_format: 123*.
kpse_bst_format: 127*.
kpse_in_name_ok: 106*, 123*, 127*, 141*.
kpse_out_name_ok: 106*.
kpse_set_program_name: 477*.
l: 68*.
last: 41*, 47*, 80, 83, 84, 85, 86, 87, 88, 90, 92,
93, 94, 95, 120, 126, 132, 139, 149, 151*, 190,
191, 211, 223*, 252.
last_check_for_aux_errors: 110*, 145.
last_cite: 138*.
last_end: 344*, 395, 396, 401, 409, 410.
last_fn_class: 156, 160*.
last_ilk: 64*.
last_lex: 31*.
last_lit_type: 201*.
last_text_char: 23*, 28*.
last_token: 344*, 407, 408, 409, 410, 413, 417.
\LaTeX*: 1*, 10*, 132*.
le cite ilk: 64*, 133, 264, 267, 270, 278.
le cite loc: 129*, 133, 135, 136, 264, 265*, 267, 268,
269, 272, 277*, 278, 279*285*.
le xcite loc: 129*, 268, 270.
left: 303, 305, 306.
n bst_execute: 78, 79, 155.
n bst_function: 78, 79, 155.
n bst_integers: 78, 79, 155.
n bst_iterate: 78, 79, 155.
n bst_macro: 78, 79, 155.
n bst_read: 78, 79, 155.
n bst_reverse: 78, 79, 155.
n bst_sort: 78, 79, 155.
n bst_strings: 78, 79, 155.
n call_type: 333, 334*, 341.
n change_case: 333, 334*, 341.
n chr_to_int: 333, 334*, 341.
n cite: 333, 334*, 341.
n concatenate: 333, 334*, 341.
n duplicate: 333, 334*, 341.
n empty: 333, 334*, 341.
n equals: 333, 334*, 341.
n format_name: 333, 334*, 341.
n gets: 333, 334*, 341.
n greater_than: 333, 334*, 341.
n if: 333, 334*, 341.
n int_to_chr: 333, 334*, 341.
n int_to_str: 333, 334*, 341.
n less_than: 333, 334*, 341.
n minus: 333, 334*, 341.
n missing: 333, 334*, 341.
n newline: 333, 334*, 341.
n num_names: 333, 334*, 341.
n options: 467*.
n plus: 333, 334*, 341.
n pop: 333, 334*, 341.
n preamble: 333, 334*, 341.
n purify: 333, 334*, 341.
n quote: 333, 334*, 341.
n skip: 333, 334*, 341.
n stack: 333, 334*.
n substring: 333, 334*, 341.
n swap: 333, 334*, 341.
n text_length: 333, 334*, 341.
n text_prefix: 333, 334*, 341.
n top_stack: 333, 334*, 341.
n type: 333, 334*, 341.
n warning: 333, 334*, 341.
n while: 333, 334*, 341.
n width: 333, 334*, 341.
n write: 333, 334*, 341.
n name: 467*, 468*, 471*, 474*, 475*, 476*.
n name_ptr: 344*, 396, 397, 398, 400, 401, 414, 415, 416.
n name_upptr: 344*, 398.
n name_buf: 43, 344*, 387, 390, 394, 397, 398, 400, 414, 415, 416.
n length: 37*, 58*, 60*, 61*, 99, 106*, 107, 141*.
n name_file: 37*, 58*, 60*, 61*, 97*, 98, 99, 100*, 106*, 107, 123*, 127*, 141*.
n name_ptr: 37*, 58*, 60*, 98, 99, 107, 141*.
n name_scan_and: 383, 384, 427.
n negative: 93.
n nested cross references: 277*.
n new_cite: 265*.
n newline: 108*, 121*, 128*.
n next_cite: 132, 134.
n next_insert: 303, 304.
n next_token: 183, 184, 185, 186, 187*.
n nil: 9.
n num_brace_level: 344*, 397, 398, 400, 416.
n no_file: 146, 151*.
n no_files: 161*, 462*.
n no_file_path: 39*, 106*, 141*.
n nonexistent_cross_reference_error: 279*, 281*.
n null_code: 26.
n num_bib_fns: 332, 333, 335, 465.
n num_gbl_strs: 161*, 162, 216*.
n num_names: 344*, 383, 426, 427.
n num_pre_defined_fields: 161*, 170, 277*, 340.
n num_preamble_strings: 219*, 276, 429.
n num_tokens: 344*, 387, 389, 390, 391, 392, 393, 394, 395.
n number_sign: 29, 33*, 189, 190.
print_recursion_illegal: 184.
print_skipping_whatsoever_remains: 96, 111, 221.
print_stk_lit: 311, 312, 313, 345, 380, 424.
print_token: 81, 135, 140, 154, 177, 184, 185, 207, 234, 273.
print_version_and_exit: 467*
push the literal stack: 308, 351, 352, 353, 361, 379, 437, 438*, 444*.
push_lit_stack: 308.
push_lt: 307*
push_type: 307*
push the literal stack: 308, 351, 352, 353, 361, 379, 437, 438*, 444*.
push_lit_stack: 308.
put: 37*, 40.
question_mark: 20, 360, 361.
r_pop_l1: 343, 449.
r_pop_l2: 343, 449.
r_pop_l3: 343, 449.
r_pop_l4: 343, 449.
raise: 278.
read_performed: 163, 164, 223*, 455, 458, 462*.
read_recent: 163, 164, 178, 203, 205, 211, 212, 214.
reading_completed: 163, 164, 223*, 455.
repush_string: 308, 361, 379, 437.
reset: 37*.
return: 6, 9.
return_von_found: 397, 398, 399.
rewrite: 37*.
right: 303, 304, 305, 306.
right_end: 302, 303, 304, 305, 306.
right_outer_delim: 219*, 242*, 244, 246, 259, 266, 274.
right_paren: 20, 33*, 219*, 242*, 244, 266.
right_str_delim: 219*, 250, 253, 254, 255, 256.
s: 51, 56, 121*, 280, 284.
s_: 74, 337*.
s_aux_extension: 74, 75, 103, 106*, 107, 139, 140.
s_bib_extension: 74, 75, 103, 106*.
s_bib_area: 74, 75.
s_bib_extension: 74, 75, 121*, 123*, 457.
s_bst_area: 74, 75.
s_bst_extension: 74, 75, 127*, 128*, 457.
s_default: 182, 337*, 339.
s_l: 337*.
s_log_extension: 74, 75, 103, 106*.
s_t: 337*.
s_u: 337*.
sam_made_the_file_name_print: 99.
sam_made_the_file_name: 99.
sam_made_the_file_name_too_long: 99, 103.
sam_made_the_file_name_wrong: 99, 106*.
save_space: 42*, 161*.
s_and_store_field_value_and_eat_white: 249, 250.
s_and_store_field_value: 249, 250.

secret agent man: 172.
seen_fn_loc: 169.
setup_bound_var: 477*.
setup_bound_var_end: 477*. 
stringcast:

strats
swap2:

string pool:

str_in:

str_out:

The \TeXbook: 27*

text:

text_char:

text_ilk:

text_int:

this can't happen: 45, 479*

A cite key disappeared: 270, 271, 285.*

A digit disappeared: 258.

Already encountered auxiliary file: 107.

Already encountered implicit function: 194.

Already encountered style file: 127.*

An at-sign disappeared: 238.

Cite hash error: 136, 137, 264, 272, 279,*285.*

Control-sequence hash error: 399.

Duplicate sort key: 301.*

History is bunk: 466.

Identifier scanning error: 165, 166, 235.

Illegal auxiliary-file command: 112.

Illegal literal type: 310, 311, 312, 313.

Illegal number of commas: 395.

Illegal string number: 51.

Nonempty empty string stack: 317.

Nonempty top of string stack: 309.

The cite list is messed up: 268.

Unknown auxiliary-file command: 116.

Unknown built-in function: 341.


Unknown function class: 157, 158, 159, 325.

Unknown literal type: 307,310, 311, 312, 313.

Unknown style-file command: 155.

Unknown type of case conversion: 372, 373, 375, 376.

tie: 29, 32,*396, 401, 411, 417, 419.


tmp_end_ptr: 43, 260, 270, 278.


to_be_written: 344,403, 405, 407, 408, 409, 410.

token_starting: 344,387, 389, 390, 391, 392, 393, 394.

token_value: 91, 92, 93, 190.

top_level_str: 104, 107, 141.*

total_ex_count: 331, 465.

total_fields: 226*

\textbf{tr print:} 161*

trace: 3, 4*
trace_and_stat_printing: 455, 456.
trace_pr: 3* 133, 159, 190, 191, 192, 193, 199, 209, 261, 297, 298, 307* 325, 457, 458, 459* 460* 461, 462* 463, 464, 465.
trace_pr: 3*
trace_pr_fn_class: 159, 193, 199.
trace_pr_newline: 3* 136, 184, 193, 199, 297, 298, 457, 458, 461.

Tuesdays: 325, 401.
turn out lights: 455.
type_exists: 219* 238, 273.
ucharcast: 77*.
uxexit: 10* 13* 100*.
unbreakable_tail: 322* 324.
defined: 16* 219*, 273, 363, 447, 459*, 477*.
unflush_string: 55, 308, 351, 352, 438*, 439*.
unknown_function_class_confusion: 157, 158, 159, 325.
unknown_literal_confusion: 307* 310, 311, 312, 313.
upper_awe_width: 35, 453.
upper_awe_width: 35, 453.
usage: 467*.
usage_help: 467*.
use_default: 344*, 412, 417.
user abuse: 98, 99, 393, 416.
val: 468*, 471*, 474*, 475*, 476*.
verbose: 10* 110*, 127*, 223*, 468*, 469*, 470*.
version_string: 10*.
vgetc: 47*.
von_end: 344*, 396, 401, 408, 409.
von_found: 382, 396.

von_name_ends_and_last_name_starts_stuff: 395, 396, 401.
von_start: 344*, 395, 396, 401, 408.
von_token_found: 396, 397, 401.
warning_message: 18, 19, 20, 150, 293, 294, 246.
WEB: 52, 69.
white Adjacent: 89, 90, 166, 235.
whole database inclusion: 132.

windows: 325.
wiz_fn_loc: 160* 161*, 325.
wiz_fn_ptr: 161*, 463.
wiz_fn_space: 10* 16* 200*.
WIZ_FN_SPACE: 10*, 14*, 200*.
wiz_functions: 10* 160*, 161*, 188*, 190, 191, 193, 194, 199, 200*, 325, 326, 463.
wiz_loc: 161*, 180, 182, 189, 193, 199.
wiz_ptr: 325, 326.
wizard: 1*.
write: 3* 51, 82, 98, 99, 321.
write_la: 3* 13* 98, 99, 321, 467*.
x_add_period: 341, 360.
x_change_case: 341, 364.
x_chr_to_int: 341, 377.
x_cite: 341, 378.
x_concatenate: 341, 350.
x_duplicate: 341, 379.
x_empty: 341, 380.
x_entry_strs: 15* 288*, 301*, 329*, 357*, 460*.
x_entry_strs_tail: 15*.
x_equal: 341, 345.
x_format_name: 341, 382, 420.
x_gets: 341, 354.
x_global_strs: 15*, 330*, 359*.
x_global_strs_tail: 15*.
x_greater_than: 341, 346.
x_int_to_chr: 341, 422.
x_int_to_str: 341, 423.
x_less_than: 341, 347.
x_minus: 341, 349.
x_missing: 341, 424.
x_num_names: 341, 426.
x_plus: 341, 428.
x_preamble: 341, 429.
x_purify: 341, 430.
x_quote: 341, 434.
x_substring: 341, 437.
x_swap: 341, 439.
x_text_length: 341, 441.
x_text_prefix: 341, 443.
x_type: 341, 447.
x_warning: 341, 448.
x_width: 341, 450.
x_write: 341, 454.


xclause: 9.

xmalloc_array: 38*, 58*, 100*.


XTALLOC: 10*, 187*, 287*, 288*.

Yogi: 455.
(Add cross-reference information 277*) Used in section 276.
(Add extensions and open files 106*) Used in section 103.
(Add or update a cross reference on cite_list if necessary 264) Used in section 263*.
(Add the period (it’s necessary) and push 362) Used in section 361.
(Add the period, if necessary, and push 361) Used in section 360.
(Add up the char_widths in this string 451) Used in section 450.
(Assign to a str_entry_var 357*) Used in section 354.
(Assign to a str_global_var 359*) Used in section 354.
(Assign to an int_entry_var 355) Used in section 354.
(Assign to an int_global_var 358) Used in section 354.
(Break that line 323) Used in section 322*.
(Break that unbreakably long line 324) Used in section 323.
( Check and insert the quoted function 193) Used in section 192.
( Check for a database key of interest 267) Used in section 266.
( Check for a duplicate or crossref-matching database key 268) Used in section 267.
( Check for entire database inclusion (and thus skip this cite key) 134) Used in section 133.
( Check the execute-command argument token 179) Used in section 178.
( Check the iterate-command argument token 204) Used in section 203.
( Check the reverse-command argument token 213) Used in section 212.
( Check the “constant” values for consistency 17*, 302) Used in section 13*.
( Check the cite key 133) Used in section 132.
( Check the macro name 207) Used in section 206.
( Check the special character (and return) 398) Used in section 397.
( Check the wiz_defined function name 182) Used in section 181.
( Cite seen, don’t add a cite key 135) Used in section 133.
( Cite unseen, add a cite key 136) Used in section 133.
( Clean up and leave 455) Used in section 10*.
( Compiler directives 11) Used in section 10*.
( Complain about a nested cross reference 282) Used in section 279*.
( Complain about missing entries whose cite keys got overwritten 286) Used in section 283.
( Complete this function’s definition 200*) Used in section 187*.
( Compute the hash code h 69) Used in section 68*.
( Concatenate the two strings and push 351) Used in section 350.
 (Concatenate them and push when pop_lit1, pop_lit2 < cmd_str_ptr 353) Used in section 352.
( Concatenate them and push when pop_lit2 < cmd_str_ptr 352) Used in section 351.
( Constants in the outer block 14*, 333) Used in section 10*.
( Convert a noncontrol sequence 375) Used in section 371.
( Convert a special character 371) Used in section 370.
( Convert a brace_level = 0 character 376) Used in section 370.
( Convert the accented or foreign character, if necessary 372) Used in section 371.
( Convert, then remove the control sequence 374) Used in section 372.
( Copy name and count commas to determine syntax 387) Used in section 382.
( Copy the macro string to field_nul_str 260) Used in section 259.
( Count the text characters 442) Used in section 441.
( Declarations for executing built_in functions 343) Used in section 325.
( Define the option table 468*, 471*, 474*, 475*, 476*) Used in section 467*.
( Define parse_arguments 467*) Used in section 10*.
( Determine the case-conversion type 366) Used in section 364.
( Determine the number of names 427) Used in section 426.
( Determine the width of this accented or foreign character 453) Used in section 452.
( Determine the width of this special character 452) Used in section 451.
( Determine where the first name ends and von name starts and ends 396) Used in section 395.

_NAMES OF THE SECTIONS_ 61
Do a quick brace-balanced scan \[256\] Used in section \[253\].
Do a full scan with \texttt{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 \texttt{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 \texttt{built in} function \[341\] Used in section \[325\].
Execute a \texttt{str entry var} \[329\] Used in section \[325\].
Execute a \texttt{str global var} \[330\] Used in section \[325\].
Execute a \texttt{viz defined} function \[326\] Used in section \[325\].
Execute an \texttt{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 \texttt{.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 \texttt{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 \[187\].
Handle a discretionary \texttt{tie} \[419\] Used in section \[411\].
Handle this \texttt{.aux} name \[103\] Used in section \[100\].
Handle this accented or foreign character (and \texttt{return}) \[399\] Used in section \[398\].
Initialize the option variables \[470*, 473\] Used in section \[467\].
Initialize the \texttt{field info} \[225\] Used in section \[224\].
Initialize the \texttt{int entry vars} \[287\] Used in section \[276\].
Initialize the \texttt{sorted cites} \[289\] Used in section \[276\].
Initialize the \texttt{str entry vars} \[288\] Used in section \[276\].
Initialize things for the \texttt{cite list} \[227\] Used in section \[224\].
Insert a \texttt{field} into the hash table \[172\] Used in section \[171\].
Insert a \texttt{str entry var} into the hash table \[176\] Used in section \[175\].
Insert a \texttt{str global var} into the hash table \[216\] Used in section \[215\].
Insert an \texttt{int entry var} into the hash table \[174\] Used in section \[173\].
Insert an \texttt{int global var} into the hash table \[202\] Used in section \[201\].
Insert pair into hash table and make \texttt{p} point to it \[71\] Used in section \[68\].
Isolate the desired name 383  Used in section 382.
Labels in the outer block 109, 146  Used in section 10*.
Local variables for initialization 23*, 66  Used in section 13*.
Make sure this entry is ok before proceeding 273  Used in section 267.
Make sure this entry’s database key is on cite_list 269  Used in section 268.
Name-process a comma 389  Used in section 387.
Name-process a left_brace 390  Used in section 387.
Name-process a right_brace 391  Used in section 387.
Name-process a sep_char 393  Used in section 387.
Name-process a white_space 392  Used in section 387.
Name-process some other character 394  Used in section 387.
Open a .bib file 123*  Used in section 120.
Open the .bst file 127*  Used in section 126.
Open this .aux file 141*  Used in section 140.
Perform a reverse command 298  Used in section 212.
Perform a sort command 299  Used in section 214.
Perform an execute command 296  Used in section 178.
Perform an iterate command 297  Used in section 203.
Perform the case conversion 370  Used in section 364.
Perform the purification 431  Used in section 430.
Pre-define certain strings 75, 79, 334*, 339, 340  Used in section 336.
Print all .bib- and .bst-file information 457  Used in section 456.
Print all cite_list and entry information 458  Used in section 456.
Print entry information 459*  Used in section 458.
Print entry integers 461  Used in section 459*.
Print entry strings 460*  Used in section 459*.
Print fields 462*  Used in section 459*.
Print the job history 466  Used in section 455.
Print the string pool 464  Used in section 456.
Print the wiz_defined functions 463  Used in section 456.
Print usage statistics 465  Used in section 456.
Procedures and functions for about everything 12, 477*, 478*  Used in section 10*.
Procedures and functions for file-system interacting 58*, 60*, 61*  Used in section 12.
Procedures and functions for handling numbers, characters, and strings 54, 56, 57, 62, 63, 68*, 77*, 198*, 265*, 278, 300, 301*, 303, 335, 336  Used in section 12.
Procedures and functions for input scanning 83, 84, 85, 86, 87, 88, 90, 92, 93, 94, 152, 183, 184, 185, 186, 187*, 228, 248, 249  Used in section 12.
Procedures and functions for name-string processing 367, 369, 384, 397, 401, 404, 406, 418, 420  Used in section 12.
Procedures and functions for the reading and processing of input files 100*, 120, 126, 132, 139, 142, 143, 145, 170, 177, 178, 180, 201, 203, 205, 210, 211, 212, 214, 215, 217  Used in section 12.
Process a .bib command 239  Used in section 238.
Process a comment command 241  Used in section 239.
Process a preamble command 242*  Used in section 239.
Process a string command 243  Used in section 239.
Process a possible command line 102*  Used in section 10*.
Process the appropriate \texttt{.bst} command 155 \ Used in section 154.
Process the string if we’ve already encountered it 70* \ Used in section 68*.
Purify a special character 432 \ Used in section 431.
Purify this accented or foreign character 433 \ Used in section 432.
Push 0 if the string has a non\texttt{white_space} char, else 1 381 \ Used in section 380.
Push the \texttt{.aux} stack 140 \ Used in section 139.
Put this cite key in its place 272 \ Used in section 267.
Put this name into the hash table 107 \ Used in section 103.
Read and execute the \texttt{.bst} file 151* \ Used in section 10*.
Read the \texttt{.aux} file 110* \ Used in section 10*.
Read the \texttt{.bib} file(s) 223* \ Used in section 211.
Remove leading and trailing junk, complaining if necessary 388* \ Used in section 387.
Remove missing entries or those cross referenced too few times 283 \ Used in section 276.
Scan a macro name 259 \ Used in section 250.
Scan a number 258 \ Used in section 250.
Scan a quoted function 192 \ Used in section 189.
Scan a \texttt{str\_literal} 191 \ Used in section 189.
Scan an already-defined function 199 \ Used in section 189.
Scan an \texttt{int\_literal} 190 \ Used in section 189.
Scan for and process a \texttt{.bib} command or database entry 236 \ Used in section 210.
Scan for and process a \texttt{.bst} command 154 \ Used in section 217.
Scan for and process an \texttt{.aux} command 116 \ Used in section 143.
Scan the appropriate number of characters 445 \ Used in section 444*.
Scan the entry type or scan and process the \texttt{.bib} command 238 \ Used in section 236.
Scan the entry’s database key 266 \ Used in section 236.
Scan the entry’s list of fields 274 \ Used in section 236.
Scan the list of \texttt{fields} 171 \ Used in section 170.
Scan the list of \texttt{int\_entry\_vars} 173 \ Used in section 170.
Scan the list of \texttt{str\_entry\_vars} 175 \ Used in section 170.
Scan the macro definition-string 209 \ Used in section 208.
Scan the macro name 206 \ Used in section 205.
Scan the macro’s definition 208 \ Used in section 205.
Scan the string’s definition field 246 \ Used in section 243.
Scan the string’s name 244 \ Used in section 243.
Scan the \texttt{wiz\_defined} function name 181 \ Used in section 180.
See if we have an “and” 386 \ Used in section 384.
Set initial values of key variables 20, 25, 27*, 28*, 32*, 33*, 35, 67, 72, 119, 125, 131, 162, 164, 196, 292 \ Used in section 13*.
Skip over \texttt{ex\_buf} stuff at \texttt{brace\_level} > 0 385 \ Used in section 384.
Skip over \texttt{name\_buf} stuff at \texttt{nm\_brace\_level} > 0 400 \ Used in section 397.
Skip to the next database entry or \texttt{.bib} command 237 \ Used in section 236.
Slide this cite key down to its permanent spot 285* \ Used in section 283.
Start a new function definition 194 \ Used in section 189.
Store the field value for a command 262 \ Used in section 261.
Store the field value for a database entry 263* \ Used in section 261.
Store the field value string 261 \ Used in section 249.
Store the string’s name 245 \ Used in section 244.
Subtract cross-reference information 279* \ Used in section 276.
Swap the two strings (they’re at the end of \texttt{str\_pool}) 440 \ Used in section 439.
The procedure \texttt{initialize} 13* \ Used in section 10*.
The scanning function \texttt{compress\_bib\_white} 252 \ Used in section 248.
The scanning function \texttt{scan\_a\_field\_token\_and\_eat\_white} 250 \ Used in section 248.
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*

\(\text{execute}_\text{fn} \text{ itself} 325\) Used in section 342.
\(\text{execute}_\text{fn}(*) 350\) Used in section 342.
\(\text{execute}_\text{fn}(+) 348\) Used in section 342.
\(\text{execute}_\text{fn}(-) 349\) Used in section 342.
\(\text{execute}_\text{fn}(:) 354\) Used in section 342.
\(\text{execute}_\text{fn}(\langle) 347\) Used in section 342.
\(\text{execute}_\text{fn}(=) 345\) Used in section 342.
\(\text{execute}_\text{fn}(>) 346\) Used in section 342.
\(\text{execute}_\text{fn}(\text{add.period$}) 360\) Used in section 342.
\(\text{execute}_\text{fn}(\text{call.type$}) 363\) Used in section 341.
\(\text{execute}_\text{fn}(\text{change.case$}) 364\) Used in section 342.
\(\text{execute}_\text{fn}(\text{chr.to.int$}) 377\) Used in section 342.
\(\text{execute}_\text{fn}(\text{cite$}) 378\) Used in section 342.
\(\text{execute}_\text{fn}(\text{duplicate$}) 379\) Used in section 342.
\(\text{execute}_\text{fn}(\text{empty$}) 380\) Used in section 342.
\(\text{execute}_\text{fn}(\text{format.name$}) 382\) Used in section 342.
\(\text{execute}_\text{fn}(\text{if$}) 421\) Used in section 341.
\(\text{execute}_\text{fn}(\text{int.to.chr$}) 422\) Used in section 342.
\(\text{execute}_\text{fn}(\text{int.to.str$}) 423\) Used in section 342.
\(\text{execute}_\text{fn}(\text{missing$}) 424\) Used in section 342.
\(\text{execute}_\text{fn}(\text{newline$}) 425\) Used in section 341.
\(\text{execute}_\text{fn}(\text{num.names$}) 426\) Used in section 342.
\(\text{execute}_\text{fn}(\text{pop$}) 428\) Used in section 341.
\(\text{execute}_\text{fn}(\text{preamble$}) 429\) Used in section 342.
\(\text{execute}_\text{fn}(\text{purify$}) 430\) Used in section 342.
\(\text{execute}_\text{fn}(\text{quote$}) 434\) Used in section 342.
\(\text{execute}_\text{fn}(\text{skip$}) 435\) Used in section 341.
\(\text{execute}_\text{fn}(\text{stack$}) 436\) Used in section 341.
\(\text{execute}_\text{fn}(\text{substring$}) 437\) Used in section 342.
\(\text{execute}_\text{fn}(\text{swap$}) 439\) Used in section 342.
\(\text{execute}_\text{fn}(\text{text.length$}) 441\) Used in section 342.
\(\text{execute}_\text{fn}(\text{text.prefix$}) 443\) Used in section 342.
\(\text{execute}_\text{fn}(\text{top$}) 446\) Used in section 341.
\(\text{execute}_\text{fn}(\text{type$}) 447\) Used in section 342.
\(\text{execute}_\text{fn}(\text{warning$}) 448\) Used in section 342.
\(\text{execute}_\text{fn}(\text{while$}) 449\) Used in section 341.
\(\text{execute}_\text{fn}(\text{width$}) 450\) Used in section 342.
\(\text{execute}_\text{fn}(\text{write$}) 454\) Used in section 342.