1
Introduction

A *compiler* translates source code to assembler or object code for a target machine. A *retargetable* compiler has multiple targets. Machine-specific compiler parts are isolated in modules that are easily replaced to target different machines.

This book describes `lcc`, a retargetable compiler for ANSI C; it focuses on the implementation. Most compiler texts survey compiling algorithms, which leaves room for only a toy compiler. This book leaves the survey to others. It tours most of a practical compiler for full ANSI C, including code generators for three target machines. It gives only enough compiling theory to explain the methods that it uses.

1.1 Literate Programs

This book not only describes the implementation of `lcc`, it *is* the implementation. The `noweb` system for literate programming generates both the book and the code for `lcc` from a single source. This source consists of interleaved prose and labelled code *fragments*. The fragments are written in the order that best suits describing the program, namely the order you see in this book, not the order dictated by the C programming language. The program `noweave` accepts the source and produces the book's typescript, which includes most of the code and all of the text. The program `notangle` extracts all of the code, in the proper order for compilation.

Fragments contain source code and references to other fragments. Fragment definitions are preceded by their labels in angle brackets. For example, the code

```
⟨a fragment label 1⟩≡                                          2▼
  sum = 0;
  for (i = 0; i < 10; i++) ⟨increment sum 1⟩

⟨increment sum 1⟩≡                                              1
  sum += x[i];
```

sums the elements of `x`. Fragment uses are typeset as illustrated by the use of ⟨*increment* `sum`⟩ in the example above. Several fragments may have the same name; `notangle` concatenates their definitions to produce a single fragment. `noweave` identifies this concatenation by using $+\equiv$ instead of $\equiv$ in continued definitions:

⟨*a fragment label* 1⟩+≡ ▲1
```
  printf("%d\n", sum);
```

Fragment definitions are like macro definitions; `notangle` extracts a program by expanding one fragment. If its definition refers to other fragments, they are themselves expanded, and so on.

Fragment definitions include aids to help readers navigate among them. Each fragment name ends with the number of the page on which the fragment's definition begins. If there's no number, the fragment isn't defined in this book, but its code does appear on the companion diskette. Each continued definition also shows the previous definition, and the next continued definition, if there is one. ▲14 is an example of a previous definition that appears on page 14, and 31▼ says the definition is continued on page 31. These annotations form a doubly linked list of definitions; the up arrow points to the previous definition in the list and down arrow points to the next one. The previous link on the first definition in a list is omitted, and the next link on the last definition is omitted. These lists are complete: If some of a fragment's definitions appear on the same page with each other, the links refer to the page on which they appear.

Most fragments also show a list of pages on which the fragment is used, as illustrated by the number 1 to the right of the definition for ⟨*increment* `sum`⟩, above. These unadorned use lists are omitted for root fragments, which define modules, and for some fragments that occur too frequently, as detailed below.

`notangle` also implements one extension to C. A long string literal can be split across several lines by ending the lines to be continued with underscores. `notangle` removes leading white space from continuation lines and concatenates them to form a single string. The first argument to `error` on page 119 is an example of this extension.

1.2 How to Read This Book

Read this book front-to-back. A few variants are possible.

- Chapter 5 describes the interface between the front end and back ends of the compiler. This chapter has been made as self-contained as possible.
- Chapters 13 18 describe the back ends of the compiler. Once you know the interface, you can read these chapters with few excursions back into their predecessors. Indeed, people have *replaced* the front end and the back ends without reading, much less understanding, the other half.

- Chapters 16 18 describe the modules that capture all information about the three targets the MIPS, SPARC, and Intel 386 and successor architectures. Each of these chapters is independent, so you may read any subset of them. If you read more than one, you may notice some repetition, but it shouldn't be too irritating because most code common to all three targets has been factored out into Chapters 13 15.

Some parts of the book describe `lcc` from the bottom up. For example, the chapters on managing storage, strings, and symbol tables describe functions that are at or near the ends of call chains. Little context is needed to understand them.

Other parts of the book give a top-down presentation. For example, the chapters on parsing expressions, statements, and declarations begin with the top-level constructs. Top-down material presents some functions or fragments well after the code that uses them, but material near the first use tells enough about the function or fragment to understand what's going on in the interim.

Some parts of the book alternate between top-down and bottom-up presentations. A less variable explanation order would be nice, but it's unattainable. Like most compilers, `lcc` includes mutually recursive functions, so it's impossible to describe all callees before all callers or all callers before all callees.

Some fragments are easier to explain before you see the code. Others are easier to explain afterward. If you need help with a fragment, don't struggle before scanning the text just before *and* after the fragment.

Most of the code for `lcc` appears in the text, but a few fragments are used but not shown. Some of these fragments hold code that is omitted to save space. Others implement language extensions, optional debugging aids, or repetitious constructs. For example, once you've seen the code that handles C's for statement, the code that handles the do-while statement adds little. The only wholesale omission is the explanation of how `lcc` processes C's initializers, which we skipped because it is long, not very interesting, and not needed to understand anything else. Fragments that are used but not defined are easy to identify: no page number follows the fragment name.

Also omitted are assertions. `lcc` includes hundreds of assertions. Most assert something that the code assumes about the value of a parameter or data structure. One is `assert(0)`, which guarantees a diagnostic and thus identifies states that are not supposed to occur. For example, if a switch is supposed to have a bona fide case for all values of the switch expression, then the default case might include `assert(0)`.

The companion diskette is complete. Even the assertions and fragments that are omitted from the text appear on the diskette. Many of them are easily understood once the documented code nearby is understood.

A mini-index appears in the middle of the outside margin of many pages. It lists each program identifier that appears on the page and the page number on which the identifier is defined in code or explained in text. These indices not only help locate definitions, but highlight circularities: Identifiers that are used before they are defined appear in the mini-indices with page numbers that follow the page on which they are used. Such circularities can be confusing, but they are inevitable in any description of a large program. A few identifiers are listed with more than one definition; these name important identifiers that are used for more than one purpose or that are defined by both code and prose.

6
Lexical Analysis

The *lexical analyzer* reads source text and produces *tokens*, which are the basic lexical units of the language. For example, the expression `*ptr = 56;` contains 10 characters or five tokens: `*`, `ptr`, `=`, `56`, and `;`. For each token, the lexical analyzer returns its *token code* and zero or more *associated values*. The token codes for single-character tokens, such as operators and separators, are the characters themselves. Defined constants (with values that do not collide with the numeric values of significant characters) are used for the codes of the tokens that can consist of one or more characters, such as identifiers and constants.

For example, the statement `*ptr = 56;` yields the token stream shown on the left below; the associated values, if there are any, are shown on the right.

stringn 30

```
'*'
ID     "ptr"   symbol-table entry for "ptr"
'='
ICON   "56"    symbol-table entry for 56
```

The token codes for the operators `*` and `=` are the operators themselves, i.e., the numeric values of `*` and `=`, respectively, and they do not have associated values. The token code for the identifier `ptr` is the value of the defined constant `ID`, and the associated values are the saved copy of the identifier string itself, i.e., the string returned by `stringn`, and a symbol-table entry for the identifier, if there is one. Likewise, the integer constant `56` returns `ICON`, and the associated values are the string `"56"` and a symbol-table entry for the integer constant 56.

Keywords, such as for, are assigned their own token codes, which distinguish them from identifiers.

The lexical analyzer also tracks the *source coordinates* for each token. These coordinates, defined in Section 3.1, give the file name, line number, and character index within the line of the first character of the token. Coordinates are used to pinpoint the location of errors and to remember where symbols are defined.

The lexical analyzer is the only part of the compiler that looks at each character of the source text. It is not unusual for lexical analysis to account for half the execution time of a compiler. Hence, speed is important. The lexical analyzer s main activity is moving characters, so minimizing the amount of character movement helps increase speed. This is done by dividing the lexical analyzer into two tightly coupled modules.

The input module, `input.c`, reads the input in large chunks into a buffer, and the recognition module, `lex.c`, examines the characters to recognize tokens.

6.1 Input

In most programming languages, input is organized in lines. Although in principle, there is rarely a limit on line length, in practice, line length is limited. In addition, tokens cannot span line boundaries in most languages, so making sure complete lines are in memory when they are being examined simplifies lexical analysis at little expense in capability. String literals are the one exception in C, but they can be handled as a special case.

The input module reads the source in large chunks, usually much larger than individual lines, and it helps arrange for complete tokens to be present in the input buffer when they are being examined, except identifiers and string literals. To minimize the overhead of accessing the input, the input module exports pointers that permit direct access to the input buffer:

⟨*input.c exported data*⟩+≡ 97 104

```
extern unsigned char *cp;
extern unsigned char *limit;
```

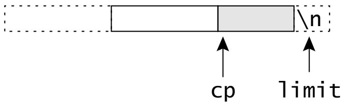
106 `fillbuf`
106 `nextline`

`cp` points to the current input character, so `*cp` is that character. `limit` points one character past the end of the characters in the input buffer, and `*limit` is always a newline character and acts as a sentinel. These pointers reference unsigned characters so that `*cp`, for example, won t sign-extend a character whose value is greater than 127.

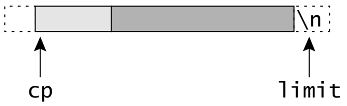
The important consequence of this design is that most of the input characters are accessed by `*cp`, and many characters are never moved. Only identifiers (excluding keywords) and string literals that appear in executable code are copied out of the buffer into permanent storage. Function calls are required only at line boundaries, which occur infrequently when compared to the number of characters in the input. Specifically, the lexical analyzer can use `*cp++` to read a character and increment `cp`. If `*cp++` is a newline character, however, it must call `nextline`, which might reset `cp` and `limit`. After calling `nextline`, if `cp` is equal to `limit`, the end of file has been reached.

Since `*limit` is always a newline, and `nextline` must be called after reading a newline, it is rarely necessary for the lexical analyzer to check if `cp` is less than `limit`. `nextline` calls `fillbuf` when the newline is the character pointed to by `limit`. The lexical analyzer can also call `fillbuf` explicitly if, for example, it wishes to ensure that an entire token is present in the input buffer. Most tokens are short, less than 32 characters, so the lexical analyzer might call `fillbuf` whenever `limit-cp` is less than 32.

This protocol is necessary in order for `fillbuf` to properly handle lines that span input buffers. In general, each input buffer ends with a partial line. To maintain the illusion of contiguous lines, and to reduce unnecessary searching, `fillbuf` moves the `limit-cp` characters of the partial line to the memory locations *preceding* the characters in the input buffer so that they will be concatenated with the characters in the trailing portion of the line when the input buffer is refilled. An example clarifies this process: Suppose the state of the input buffer is

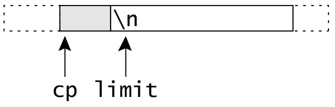


where shading depicts the characters that have yet to be consumed and `\n` represents the newline. If `fillbuf` is called, it slides the unconsumed tail of the input buffer down and refills the buffer. The resulting state is

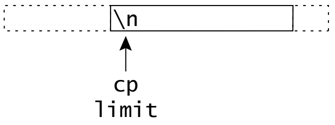


`fillbuf`	106
`limit`	103
`nextline`	106

where the darker shading differentiates the newly read characters from those moved by `fillbuf`. When a call to `fillbuf` reaches the end of the input, the buffer s state becomes



Finally, when `nextline` is called for the last sentinel at `*limit`, `fillbuf` sets `cp` equal to `limit`, which indicates end of file (after the first call to `nextline`). This final state is



The remaining global variables exported by `input.c` are:

⟨*input.c exported data*⟩+≡ 103

```
  extern int infd;
  extern char *firstfile;
  extern char *file;
  extern char *line;
  extern int lineno;
```

Input is read from the file descriptor given by `infd`; the default is zero, which is the standard input. `file` is the name of the current input file; `line` gives the location of the beginning of the current line, if it were to fit in the buffer; and `lineno` is the line number of the current line. The coordinates f, x, y of the token that begins at `cp`, where f is the file name, are thus given by `file`, `cp-line`, and `lineno`, where characters in the line are numbered beginning with zero. `line` is used only to compute the x coordinate, which counts tabs as single characters. `firstfile` gives the name of the first source file encountered in the input; it s used in error messages.

The input buffer itself is hidden inside the input module:

⟨*input.c exported macros*⟩≡
```
#define  MAXLINE  512
#define  BUFSIZE 4096
```

⟨*input.c data*⟩≡
```
static int bsize;
static unsigned char buffer[MAXLINE+1 + BUFSIZE+1];
```

`BUFSIZE` is the size of the input buffer into which characters are read, and `MAXLINE` is the maximum number of characters allowed in an unconsumed tail of the input buffer. `fillbuf` must not be called if `limit-cp` is greater than `MAXLINE`. The standard specifies that compilers need not handle lines that exceed 509 characters; `lcc` handles lines of arbitrary length, but, except for identifiers and string literals, insists that tokens not exceed 512 characters.

104	`file`
106	`fillbuf`
104	`firstfile`
104	`infd`
103	`limit`
104	`line`
104	`lineno`
106	`nextline`

The value of `bsize` encodes three different input states: If `bsize` is less than zero, no input has been read or a read error has occurred; if `bsize` is zero, the end of input has been reached; and `bsize` is greater than zero when `bsize` characters have just been read. This rather complicated encoding ensures that `lcc` is initialized properly and that it never tries to read past the end of the input.

`inputInit` initializes the input variables and fills the buffer:

⟨*input.c functions*⟩≡ 106▼
```
void inputInit() {
    limit = cp = &buffer[MAXLINE+1];
    bsize = -1;
    lineno = 0;
    file = NULL;
    ⟨refill buffer 106⟩
    nextline();
}
```

`nextline` is called whenever `*cp++` reads a newline. If `cp` is greater than or equal to `limit`, the input buffer is empty.

```
⟨input.c functions⟩+≡                                          105 106
  void nextline() {
     do {
        if (cp >= limit) {
           ⟨refill buffer 106⟩
           if (cp == limit)
              return;
        } else
           lineno++;
        for (line = (char *)cp; *cp==' ' || *cp=='\t'; cp++)
           ;
     } while (*cp == '\n' && cp == limit);
     if (*cp == '#') {
        resynch();
        nextline();
     }
  }
```

If cp is still equal to limit after filling the buffer, the end of the file has been reached. The do-while loop advances cp to the first nonwhite-space character in the line, treating sentinel newlines as white space. The last four lines of nextline check for resynchronization directives emitted by the preprocessor; see Exercise 6.2. inputInit and nextline call fillbuf to refill the input buffer:

bsize	105
buffer	105
BUFSIZE	105
infd	104
inputInit	105
limit	103
line	104
lineno	104
MAXLINE	105
resynch	125

```
⟨refill buffer 106⟩≡                                           105 106
  fillbuf();
  if (cp >= limit)
     cp = limit;
```

If the input is exhausted, cp will still be greater than or equal to limit when fillbuf returns, which leaves these variables set as shown in the last diagram on page 104. fillbuf does all of the buffer management and the actual input:

```
⟨input.c functions⟩+≡                                              106
  void fillbuf() {
     if (bsize == 0)
        return;
     if (cp >= limit)
        cp = &buffer[MAXLINE+1];
     else
        ⟨move the tail portion 107⟩
     bsize = read(infd, &buffer[MAXLINE+1], BUFSIZE);
     if (bsize < 0) {
        error("read error\n");
        exit(1);
```

```
      }
      limit = &buffer[MAXLINE+1+bsize];
      *limit = '\n';
   }
```

fillbuf reads the BUFSIZE (or fewer) characters into the buffer beginning at position MAXLINE+1, resets limit, and stores the sentinel newline. If the input buffer is empty when fillbuf is called, cp is reset to point to the first new character. Otherwise, the tail limit-cp characters are moved so that the last character is in buffer[MAXLINE], and is thus adjacent to the newly read characters.

⟨*move the tail portion* 107⟩≡ 106

```
   {
      int n = limit - cp;
      unsigned char *s = &buffer[MAXLINE+1] - n;
      line = (char *)s - ((char *)cp - line);
      while (cp < limit)
         *s++ = *cp++;
      cp = &buffer[MAXLINE+1] - n;
   }
```

Notice the computation of line: It accounts for the portion of the current line that has already been consumed, so that cp-line gives the correct index of the character *cp.

105	bsize
105	buffer
105	BUFSIZE
106	fillbuf
103	limit
104	line
105	MAXLINE

6.2 Recognizing Tokens

There are two principal techniques for recognizing tokens: building a *finite automaton* or writing an ad hoc recognizer by hand. The lexical structure of most programming languages can be described by regular expressions, and such expressions can be used to construct a *deterministic finite automaton* that recognizes and returns tokens. The advantage of this approach is that it can be automated. For example, LEX is a program that takes a lexical specification, given as regular expressions, and generates an automaton and an appropriate interpreting program.

The lexical structure of most languages is simple enough that lexical analyzers can be constructed easily by hand. In addition, automatically generated analyzers, such as those produced by LEX, tend to be large and slower than analyzers built by hand. Tools like LEX are very useful, however, for one-shot programs and for applications with complex lexical structures.

For C, tokens fall into the six classes defined by the following EBNF grammar:

token:
  keyword
  identifier
  constant
  string-literal
  operator
  punctuator

punctuator:
  one of `[ ] ( ) { } * , : = ; ...`

White space   blanks, tabs, newlines, and comments   separates some tokens, such as adjacent identifiers, but is otherwise ignored except in string literals.

The lexical analyzer exports two functions and four variables:

⟨*lex.c exported functions*⟩≡
```
extern int getchr ARGS((void));
extern int gettok ARGS((void));
```

⟨*lex.c exported data*⟩≡
```
extern int t;
extern char *token;
extern Symbol tsym;
extern Coordinate src;
```

`Coordinate` 38
`gettok` 111

`gettok` returns the next token. `getchr` returns, but does not consume, the next nonwhite-space character. The values returned by `gettok` are the characters themselves (for single-character tokens), enumeration constants (such as `IF`) for the keywords, and the following defined constants for the others:

`ID`	identifiers
`FCON`	floating constants
`ICON`	integer constants
`SCON`	string constants
`INCR`	`++`
`DECR`	`--`
`DEREF`	`->`
`ANDAND`	`&&`
`OROR`	`||`
`LEQ`	`<=`
`EQL`	`==`
`NEQ`	`!=`
`GEQ`	`>=`
`RSHIFT`	`>>`
`LSHIFT`	`<<`
`ELLIPSIS`	`...`
`EOI`	end of input

These constants are defined by

⟨*lex.c exported types*⟩≡
```
  enum {
  #define xx(a,b,c,d,e,f,g) a=b,
  #define yy(a,b,c,d,e,f,g)
  #include "token.h"
     LAST
  };
```

where `token.h` is a file with 256 lines like

⟨*token.h* 109⟩≡ 109 ▼
```
  yy(0,         0, 0,  0,    0,      0)
  xx(FLOAT,     1, 0,  0,    0,      CHAR,   "float")
  xx(DOUBLE,    2, 0,  0,    0,      CHAR,   "double")
  xx(CHAR,      3, 0,  0,    0,      CHAR,   "char")
  xx(SHORT,     4, 0,  0,    0,      CHAR,   "short")
  xx(INT,       5, 0,  0,    0,      CHAR,   "int")
  xx(UNSIGNED,  6, 0,  0,    0,      CHAR,   "unsigned")
  xx(POINTER,   7, 0,  0,    0,      0,      0)
  xx(VOID,      8, 0,  0,    0,      CHAR,   "void")
  xx(STRUCT,    9, 0,  0,    0,      CHAR,   "struct")
  xx(UNION,    10, 0,  0,    0,      CHAR,   "union")
  xx(FUNCTION, 11, 0,  0,    0,      0,      0)
  xx(ARRAY,    12, 0,  0,    0,      0,      0)
  xx(ENUM,     13, 0,  0,    0,      CHAR,   "enum")
  xx(LONG,     14, 0,  0,    0,      CHAR,   "long")
  xx(CONST,    15, 0,  0,    0,      CHAR,   "const")
  xx(VOLATILE, 16, 0,  0,    0,      CHAR,   "volatile")
```

192	addtree
149	AND
193	cmptree
149	OR

⟨*token.h* 109⟩+≡ ▲ 109
```
  yy(0,        42, 13, MUL,  multree,ID,     "*")
  yy(0,        43, 12, ADD,  addtree,ID,     "+")
  yy(0,        44, 1,  0,    0,      ',',    ",")
  yy(0,        45, 12, SUB,  subtree,ID,     "-")
  yy(0,        46, 0,  0,    0,      '.',    ".")
  yy(0,        47, 13, DIV,  multree,'/',    "/")
  xx(DECR,     48, 0,  SUB,  subtree,ID,     "--")
  xx(DEREF,    49, 0,  0,    0,      DEREF,  "->")
  xx(ANDAND,   50, 5,  AND,  andtree,ANDAND, "&&")
  xx(OROR,     51, 4,  OR,   andtree,OROR,   "||")
  xx(LEQ,      52, 10, LE,   cmptree,LEQ,    "<=")
```

`token.h` uses macros to collect everything about each token or symbolic constant into one place. Each line in `token.h` gives seven values of interest for the token as arguments to either `xx` or `yy`. The token codes are given by the values in the second column. token.h is read to define symbols, build arrays indexed by token, and so forth, and using it guarantees that such definitions are synchronized with one another. This technique is common in assembler language programming.

Single-character tokens have yy lines and multicharacter tokens and other definitions have xx lines. The first column in xx is the enumeration identifier. The other columns give the identifier or character value, the precedence if the token is an operator (Section 8.3), the generic operator (Section 5.5), the tree-building function (Section 9.4), the token s set (Section 7.6), and the string representation.

These columns are extracted for different purposes by defining the xx and yy macros and including token.h *again*. The enumeration definition above illustrates this technique; it defines xx so that each expansion defines one member of the enumeration. For example, the xx line for DECR expands to

```
DECR=48,
```

and thus defines DECR to an enumeration constant with the value 48. yy is defined to have no replacement, which effectively ignores the yy lines.

The global variable t is often used to hold the current token, so most calls to gettok use the idiom

DECR	109
gettok	111
src	108
Symbol	37
token.h	109
token	108
tsym	108

```
t = gettok();
```

token, tsym, and src hold the values associated with the current token, if there are any. token is the source text for the token itself, and tsym is a Symbol for some tokens, such as identifiers and constants. src is the source coordinate for the current token.

gettok could return a structure containing the token code and the associated values, or a pointer to such a structure. Since most calls to gettok examine only the token code, this kind of encapsulation does not add significant capability. Also, gettok is the most frequently called function in the compiler; a simple interface makes the code easier to read.

gettok recognizes a token by switching on its first character, which classifies the token, and consuming subsequent characters that make up the token. For some tokens, these characters are given by one or more of the sets defined by map. map[c] is a mask that classifies character c as a member of one or more of six sets:

⟨*lex.c types*⟩≡
```
enum { BLANK=01,  NEWLINE=02, LETTER=04,
       DIGIT=010, HEX=020,    OTHER=040 };
```

⟨*lex.c data*⟩≡ 117
```
static unsigned char map[256] = { ⟨map initializer⟩ };
```

map[c]&BLANK is nonzero if c is a white-space character other than a newline. Newlines are excluded because hitting one requires gettok to call nextline. The other values identify other subsets of characters: NEWLINE is the set consisting of just the newline character, LETTER is the set of upper- and lowercase letters, DIGIT is the set of digits 0–9, HEX is the set of digits 0–9, a–f, and A–F, and OTHER is the set that holds the rest of the ASCII characters that are in the source and execution character sets specified by the standard. If map[c] is zero, c is not guaranteed to be acceptable to all ANSI C compilers, which, somewhat surprisingly, is the case for $, @, and '.

gettok is a large function, but the switch statement that dispatches on the token s first character divides it into manageable pieces:

```
⟨lex.c macros⟩≡
  #define MAXTOKEN 32

⟨lex.c functions⟩≡                                        117
  int gettok() {
     for (;;) {
        register unsigned char *rcp = cp;
        ⟨skip white space 112⟩
        if (limit - rcp < MAXTOKEN) {
           cp = rcp;
           fillbuf();
           rcp = cp;
        }
        src.file = file;
        src.x = (char *)rcp - line;
        src.y = lineno;
        cp = rcp + 1;
        switch (*rcp++) {
        ⟨gettok cases 112⟩
        default:
           if ((map[cp[-1]]&BLANK) == 0)
              ⟨illegal character⟩
        }
     }
  }
```

110	BLANK
110	DIGIT
104	file
106	fillbuf
110	HEX
110	LETTER
103	limit
104	line
104	lineno
110	map
110	NEWLINE
106	nextline
110	OTHER

gettok begins by skipping over white space and then checking that there is at least one token in the input buffer. If there isn t, calling fillbuf ensures that there is. MAXTOKEN applies to all tokens except identifiers, string literals, and numeric constants; occurrences of these tokens that are longer than MAXTOKEN characters are handled explicitly in the code for those tokens. The standard permits compilers to limit string literals to 509 characters and identifiers to 31 characters. lcc increases these limits to 4,096 (`BUFSIZE`) and 512 (`MAXLINE`) to accommodate *programs* that emit C programs, because these emitted programs may contain long identifiers.

Instead of using `cp` as suggested in Section 6.1, `gettok` copies `cp` to the register variable `rcp` upon entry, and uses `rcp` in token recognition. `gettok` copies `rcp` back to `cp` before it returns, and before calls to `nextline` and `fillbuf`. Using `rcp` improves performance and makes scanning loops compact and fast. For example, white space is elided by

```
⟨skip white space 112⟩≡                                    111
  while (map[*rcp]&BLANK)
     rcp++;
```

Using a register variable to index `map` generates efficient code where it counts. These kinds of scans examine every character in the input, and they examine characters by accessing the input buffer directly. Some optimizing compilers can make similar improvements locally, but not across potentially aliased assignments and calls to other, irrelevant functions.

Each of the sections below describes one of the cases in ⟨`gettok` *cases*⟩. The cases omitted from this book are

BLANK	110
BUFSIZE	105
fillbuf	106
gettok	111
limit	103
map	110
MAXLINE	105
nextline	106
rcp	111
tsym	108

```
⟨gettok cases 112⟩≡                                  112▼   111
  case '/': ⟨comment or /⟩
  case 'L': ⟨wide-character constants⟩
  ⟨cases for two-character operators⟩
  ⟨cases for one-character operators and punctuation⟩
```

`gettok` calls `nextline` when it trips over a newline or one of its synonyms:

```
⟨gettok cases 112⟩+≡                             ▲112 113▼   111
  case '\n': case '\v': case '\r': case '\f':
     nextline();
     if (⟨end of input 112⟩) {
        tsym = NULL;
        return EOI;
     }
     continue;
```

```
⟨end of input 112⟩≡                                    112 124
  cp == limit
```

When control reaches this case, `cp` points to the character that *follows* the newline; when `nextline` returns, `cp` still points to that character, and `cp` is less than `limit`. End of file is the exception: here, `cp` equals `limit`. Testing for this condition is rarely needed, because `*cp` will always be a newline, which terminates the scans for most tokens.

The sections below describe the remaining cases. Recognizing the tokens themselves is relatively straightforward; computing the associated values for some token is what complicates each case.

6.3 Recognizing Keywords

There are 28 keywords:

keyword: one of

```
auto       double   int      struct
break      else     long     switch
char       extern   return   union
const      float    short    unsigned
continue   for      signed   void
default    goto     sizeof   volatile
do         if       static   while
```

Keywords could be recognized through a look-up in a table in which each keyword entry carries its token code and each built-in type entry carries its type. Instead, keywords are recognized by a hard-coded decision tree, which is faster than searching a table and nearly as simple. The cases for the lowercase letters that begin keywords make explicit tests for the keywords, which are possible because the entire token must appear in the input buffer. For example, the case for i is

```
110 DIGIT
109 INT
110 LETTER
110 map
111 rcp
108 tsym
```

⟨gettok *cases* 112⟩+≡ 112 114 111

```
  case 'i':
     if (rcp[0] == 'f'
     && !(map[rcp[1]]&(DIGIT|LETTER))) {
        cp = rcp + 1;
        return IF;
     }
     if (rcp[0] == 'n'
     &&  rcp[1] == 't'
     && !(map[rcp[2]]&(DIGIT|LETTER))) {
        cp = rcp + 2;
        tsym = inttype->u.sym;
        return INT;
     }
     goto id;
```

id labels the code in the next section that scans identifiers. If the token is if or int, cp is updated and the appropriate token code is returned; otherwise, the token is an identifier. For int, tsym holds the symbol-table entry for the type int. The cases for the characters abcdefglrsuvw are similar, and were generated automatically by a short program.

The code generated for these fragments is short and fast. For example, on most machines, `int` is recognized by less than a dozen instructions, many fewer than are executed when a table is searched for keywords, even if perfect hashing is used.

6.4 Recognizing Identifiers

The syntax for identifiers is

identifier:
  nondigit { *nondigit* | *digit* }

digit:
  one of `0 1 2 3 4 5 6 7 8 9`

nondigit:
  one of `_`
```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

DIGIT	110
LETTER	110
map	110
MAXLINE	105
MAXTOKEN	111
rcp	111
stringn	30
token	108

The code echoes this syntax, but must also cope with the possibility of identifiers that are longer than `MAXTOKEN` characters and thus might be split across input buffers.

```
⟨gettok cases 112⟩+≡                                    113 116   111
  case 'h': case 'j': case 'k': case 'm': case 'n': case 'o':
  case 'p': case 'q': case 'x': case 'y': case 'z':
  case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
  case 'G': case 'H': case 'I': case 'J': case 'K':
  case 'M': case 'N': case 'O': case 'P': case 'Q': case 'R':
  case 'S': case 'T': case 'U': case 'V': case 'W': case 'X':
  case 'Y': case 'Z': case '_':
  id:
     ⟨ensure there are at least MAXLINE characters 115⟩
     token = (char *)rcp - 1;
     while (map[*rcp]&(DIGIT|LETTER))
        rcp++;
     token = stringn(token, (char *)rcp - token);
     ⟨tsym ← type named by token 115⟩
     cp = rcp;
     return ID;
```

All identifiers are saved in the string table. At the entry to this and all cases, both `cp` and `rcp` have been incremented past the first character of the token. If the input buffer holds less than `MAXLINE` characters, `cp` is backed up one character to point to the identifier s first character, `fillbuf` is called to replenish the input buffer, and `cp` and `rcp` are adjusted to point to the identifier s second character as before:

⟨*ensure there are at least* `MAXLINE` *characters* 115⟩≡ 114 116 120
```
  if (limit - rcp < MAXLINE) {
     cp = rcp - 1;
     fillbuf();
     rcp = ++cp;
  }
```

A typedef makes an identifier a synonym for a type, and these names are installed in the `identifiers` table. `gettok` thus sets `tsym` to the symbol-table entry for `token`, if there is one:

⟨`tsym` ← *type named by* `token` 115⟩≡ 114
```
  tsym = lookup(token, identifiers);
```

If `token` names a type, `tsym` is set to the symbol-table entry for that type, and `tsym->sclass` will be equal to `TYPEDEF`. Otherwise, `tsym` is null or the identifier isn t a type name. The macro

⟨*lex.c exported macros*⟩≡
```
  #define istypename(t,tsym) (kind[t] == CHAR \
     || t == ID && tsym && tsym->sclass == TYPEDEF)
```

encapsulates testing if the current token is a type name: A type name is either one of the keywords that names a type, such as int, or an identifier that is a typedef for a type. The global variables `t` and `tsym` are the only valid arguments to `istypename`.

```
109 CHAR
106 fillbuf
111 gettok
 41 identifiers
143 kind
103 limit
 45 lookup
105 MAXLINE
111 rcp
108 token
108 tsym
```

6.5 Recognizing Numbers

There are four kinds of numeric constants in ANSI C:

constant:
  floating-constant
  integer-constant
  enumeration-constant
  character-constant

enumeration-constant:
  identifier

The code for identifiers shown in the previous section handles enumeration constants, and the code in Section 6.6 handles character constants. The lexical analyzer returns the token code `ID` and sets `tsym` to the symbol-table entry for the enumeration constant. The caller checks for an enumeration constant and uses the appropriate integer in its place; the code in Section 8.8 is an instance of this convention.

There are three kinds of integer constants:

integer-constant:
  decimal-constant [*integer-suffix*]
  octal-constant [*integer-suffix*]
  hexadecimal-constant [*integer-suffix*]

integer-suffix:
  unsigned-suffix [*long-suffix*]
  long-suffix [*unsigned-suffix*]

unsigned-suffix: `u` | `U`

long-suffix: `l` | `L`

The first few characters of the integer constant help identify its kind.

```
⟨gettok cases 112⟩+≡                                    ▲114 119▼   111
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9': {
      unsigned int n = 0;
      ⟨ensure there are at least MAXLINE characters 115⟩
      token = (char *)rcp - 1;
      if (*token == '0' && (*rcp == 'x' || *rcp == 'X')) {
          ⟨hexadecimal constant⟩
      } else if (*token == '0') {
          ⟨octal constant⟩
      } else {
          ⟨decimal constant 117⟩
      }
      return ICON;
  }
```

MAXLINE	105
rcp	111
token	108

As for identifiers, this case begins by insuring that the input buffer holds at least `MAXLINE` characters, which permits the code to look ahead, as the test for hexadecimal constants illustrates.

The fragments for the three kinds of integer constant set `n` to the value of the constant. They must not only recognize the constant, but also ensure that the constant is within the range of representable integers.

Recognizing decimal constants illustrates this processing. The syntax for decimal constants is:

decimal-constant:
  nonzero-digit { *digit* }

nonzero-digit:
  one of `1 2 3 4 5 6 7 8 9`

The code accumulates the decimal value in `n` by repeated multiplications:

⟨*decimal constant* 117⟩≡ 116
```
int overflow = 0;
for (n = *token - '0'; map[*rcp]&DIGIT; ) {
   int d = *rcp++ - '0';
   if (n > ((unsigned)UINT_MAX - d)/10)
      overflow = 1;
   else
      n = 10*n + d;
}
⟨check for floating constant 117⟩
cp = rcp;
tsym = icon(n, overflow, 10);
```

At each step, overflow will occur if $10 * \texttt{n} + \texttt{d} > \texttt{UINT\_MAX}$, where `UINT_MAX` is the value of the largest representable unsigned number. Rearranging this equation gives the test shown above, which looks before it leaps into computing the new value of `n`. `overflow` is set to one if the constant overflows. `icon` handles the optional suffixes.

A decimal constant is the prefix of a floating constant if the next character is a period or an exponent indicator:

⟨*check for floating constant* 117⟩≡ 117
```
if (*rcp == '.' || *rcp == 'e' || *rcp == 'E') {
   cp = rcp;
   tsym = fcon();
   return FCON;
}
```

110	DIGIT
120	fcon
111	gettok
110	map
111	rcp
37	symbol
108	token
108	tsym
57	unsignedlong

`fcon` is similar to `icon`; it recognizes the suffix of a floating constant. `overflow` will be one when a floating constant has a whole part that exceeds `UINT_MAX`, but neither `n` nor `overflow` is passed to `fcon`, which reexamines `token` to check for *floating* overflow.

`icon` recognizes the optional `U` and `L` suffixes (in either case), warns about values that overflow, initializes a symbol to the appropriate type and value, and returns a pointer to the symbol

⟨*lex.c data*⟩+≡ ▲110
```
static struct symbol tval;
```

`tval` serves only to provide the type and value of a constant to `gettok` s caller. The caller must lift the relevant data before the next call to `gettok`.

⟨*lex.c functions*⟩+≡ ▲111 119▼
```
static Symbol icon(n, overflow, base)
unsigned n; int overflow, base; {
   if ((*cp=='u'||*cp=='U') && (cp[1]=='l'||cp[1]=='L')
   ||  (*cp=='l'||*cp=='L') && (cp[1]=='u'||cp[1]=='U')) {
      tval.type = unsignedlong;
```

```
            cp += 2;
        } else if (*cp == 'u' || *cp == 'U') {
            tval.type = unsignedtype;
            cp += 1;
        } else if (*cp == 'l' || *cp == 'L') {
            if (n > (unsigned)LONG_MAX)
                tval.type = unsignedlong;
            else
                tval.type = longtype;
            cp += 1;
        } else if (base == 10 && n > (unsigned)LONG_MAX)
            tval.type = unsignedlong;
        else if (n > (unsigned)INT_MAX)
            tval.type = unsignedtype;
        else
            tval.type = inttype;
        if (overflow) {
            warning("overflow in constant '%S'\n", token,
                (char*)cp - token);
            n = LONG_MAX;
        }
        ⟨set tval s value 118⟩
        ppnumber("integer");
        return &tval;
    }
```

isunsigned	60
longtype	57
ppnumber	119
%S	99
token	108
tval	117
unsignedlong	57
unsignedtype	58

If both U and L appear, n is an unsigned long, and if only U appears, n is an unsigned. If only L appears, n is a long unless it s too big, in which case it s an unsigned long. n is also an unsigned long if it s an unsuffixed decimal constant and it s too big to be a long. Unsuffixed octal and hexadecimal constants are ints unless they re too big, in which case they re unsigneds. The format code %S prints a string like printf s %s, but consumes an additional argument that specifies the length of the string. It can thus print strings that aren t terminated by a null character.

The types int, long, and unsigned are different types, but lcc insists that they all have the same size. This constraint simplifies the tests shown above and the code that sets tval s value:

```
⟨set tval s value 118⟩≡                                          118
    if (isunsigned(tval.type))
        tval.u.c.v.u = n;
    else
        tval.u.c.v.i = n;
```

Relaxing this constraint would complicate this code and the tests above. For example, the standard specifies that the type of an unsuffixed decimal constant is int, long, or unsigned long, depending on its value. In `lcc`, ints and longs can accommodate the same range of integers, so an unsuffixed decimal constant is either int or unsigned.

A numeric constant is formed from a *preprocessing number*, which is the numeric constant recognized by the C preprocessor. Unfortunately, the standard specifies preprocessing numbers that are a superset of the integer and floating constants; that is, a valid preprocessing number may *not* be a valid numeric constant. `123.4.5` is an example. The preprocessor deals with such numbers too, but it may pass them on to the compiler, which must treat them as single tokens and thus must catch preprocessing numbers that aren t valid constants.

The syntax of a preprocessing number is

pp-number:
  [`.`] *digit* { *digit* | `.` | *nondigit* | `E` *sign* | `e` *sign* }

sign: `-` | `+`

Valid numeric constants are prefixes of preprocessing numbers, so the processing in `icon` and `fcon` might conclude successfully without consuming the complete preprocessing number, which is an error. `ppnumber` is called from `icon`, and `fcon` and checks for this case.

⟨*lex.c functions*⟩+≡ 117 120

```
  static void ppnumber(which) char *which; {
     unsigned char *rcp = cp--;

     for ( ; (map[*cp]&(DIGIT|LETTER)) || *cp == '.'; cp++)
        if ((cp[0] == 'E' || cp[0] == 'e')
        &&  (cp[1] == '-' || cp[1] == '+'))
           cp++;
     if (cp > rcp)
        error("'%S' is a preprocessing number but an _
           invalid %s constant\n", token,
           (char*)cp-token, which);
  }
```

47	constant
110	DIGIT
111	gettok
110	LETTER
110	map
111	rcp
108	token

`ppnumber` backs up one character and skips over the characters that may comprise a preprocessing number; if it scans past the end of the numeric token, there s an error.

`fcon` recognizes the suffix of floating constants and is called in two places. One of the calls is shown above in ⟨*check for floating constant*⟩. The other call is from the `gettok` case for ‘`.` :

⟨`gettok` *cases* 112⟩+≡ 116 122 111

```
  case '.':
     if (rcp[0] == '.' && rcp[1] == '.') {
        cp += 2;
        return ELLIPSIS;
        }
        if ((map[*rcp]&DIGIT) == 0)
            return '.';
        ⟨ensure there are at least MAXLINE characters 115⟩
        cp = rcp - 1;
        token = (char *)cp;
        tsym = fcon();
        return FCON;
```

The syntax for floating constants is

floating-constant:
  fractional-constant [*exponent-part*] [*floating-suffix*]
  digit-sequence exponent-part [*floating-suffix*]

fractional-constant:
  [*digit-sequence*] `.` *digit-sequence*
  digit-sequence `.`

exponent-part:
  `e` [*sign*] *digit-sequence*
  `E` [*sign*] *digit-sequence*

digit-sequence:
  digit { *digit* }

floating-suffix:
  one of `f l F L`

DIGIT	110
map	110
ppnumber	119
rcp	111
token	108
tsym	108
tval	117

`fcon` recognizes a *floating-constant*, converts the token to a double value, and determines `tval` s type and value:

⟨*lex.c functions*⟩+≡ 119

```
    static Symbol fcon() {
        ⟨scan past a floating constant 121⟩
        errno = 0;
        tval.u.c.v.d = strtod(token, NULL);
        if (errno == ERANGE)
            ⟨warn about overflow 120⟩
        ⟨set tval s type and value 121⟩
        ppnumber("floating");
        return &tval;
    }
```

⟨*warn about overflow* 120⟩≡ 120 121

```
    warning("overflow in floating constant '%S'\n", token,
        (char*)cp - token);
```

`strtod` is a C library function that interprets its first string argument as a floating constant and returns the corresponding double value. If the constant is out of range, `strtod` sets the global variable `errno` to `ERANGE` as stipulated by the ANSI C specification for the C library.

A floating constant follows the syntax shown above, and is recognized by:

```
⟨scan past a floating constant 121⟩≡                                120
  if (*cp == '.')
     ⟨scan past a run of digits 121⟩
  if (*cp == 'e' || *cp == 'E') {
     if (*++cp == '-' || *cp == '+')
        cp++;
     if (map[*cp]&DIGIT)
        ⟨scan past a run of digits 121⟩
     else
        error("invalid floating constant '%S'\n", token,
           (char*)cp - token);
  }
```

```
⟨scan past a run of digits 121⟩≡                                    121
  do
     cp++;
  while (map[*cp]&DIGIT);
```

As dictated by the syntax, an exponent indicator must be followed by at least one digit.

A floating constant may have an `F` or `L` suffix (but not both); these specify the types `float` and `long double`, respectively.

```
⟨set tval s type and value 121⟩≡                                    120
  if (*cp == 'f' || *cp == 'F') {
     ++cp;
     if (tval.u.c.v.d > FLT_MAX)
        ⟨warn about overflow 120⟩
     tval.type = floattype;
     tval.u.c.v.f = tval.u.c.v.d;
  } else if (*cp == 'l' || *cp == 'L') {
     cp++;
     tval.type = longdouble;
  } else
     tval.type = doubletype;
```

110	DIGIT
57	doubletype
57	floattype
57	longdouble
110	map
108	token
117	tval

6.6 Recognizing Character Constants and Strings

Recognizing character constants and string literals is complicated by escape sequences like `\n`, `\034`, `\xFF`, and `\"`, and by wide-character constants. `lcc` implements so-called wide characters as normal ASCII characters, and thus uses unsigned char for the type `wchar_t`. The syntax is

> *character-constant:*
> [`L`] `'`*c-char* { *c-char* }`'`
>
> *c-char:*
> any character except `'`, `\`, or newline
> *escape-sequence*
>
> *escape-sequence:*
> one of `\' \" \? \\ \a \b \f \n \r \t \v`
> `\` *octal-digit* [*octal-digit* [*octal-digit*]]
> `\x` *hexadecimal-digit* { *hexadecimal-digit* }
>
> *string-literal:*
> [`L`] `"`{ *s-char* }`"`
>
> *s-char:*
> any character except `"`, `\`, or newline
> *escape-sequence*

String literals can span more than one line if a backslash immediately precedes the newline. Adjacent string literals are automatically concatenated together to form a single literal. In a proper ANSI C implementation, this line splicing and string literal concatenation is done by the preprocessor, and the compiler sees only single, uninterrupted string literals. `lcc` implements line splicing and concatenation for string literals anyway, so that it can be used with pre-ANSI preprocessors.

`BUFSIZE`	105
`limit`	103
`MAXLINE`	105
`nextline`	106

Implementing these features means that string literals can be longer than `MAXLINE` characters, so ⟨*ensure there are at least* `MAXLINE` *characters*⟩ cannot be used to ensure that a sequence of adjacent entire string literals appears in the input buffer. Instead, the code must detect the newline at `limit` and call `nextline` explicitly, and it must copy the literal into a private buffer.

```
⟨gettok cases 112⟩+≡                                          ▲119   111
  scon:
  case '\'': case '"': {
     static char cbuf[BUFSIZE+1];
     char *s = cbuf;
     int nbad = 0;
     *s++ = *--cp;
     do {
        cp++;
        ⟨scan one string literal 123⟩
        if (*cp == cbuf[0])
           cp++;
        else
```

```
            error("missing %c\n", cbuf[0]);
      } while (cbuf[0] == '"' && getchr() == '"');
      *s++ = 0;
      if (s >= &cbuf[sizeof cbuf])
         error("%s literal too long\n",
            cbuf[0] == '"' ? "string" : "character");
      ⟨warn about non-ANSI literals⟩
      ⟨set tval and return ICON or SCON 123⟩
   }
```

The outer do-while loop gathers up adjacent string literals, which are identified by their leading double quote character, into `cbuf`, and reports those that are too long. The leading character also determines the type of the associated value and `gettok` s return value:

```
⟨set tval and return ICON or SCON 123⟩≡                                123
   token = cbuf;
   tsym = &tval;
   if (cbuf[0] == '"') {
      tval.type = array(chartype, s - cbuf - 1, 0);
      tval.u.c.v.p = cbuf + 1;
      return SCON;
   } else {
      if (s - cbuf > 3)
         warning("excess characters in multibyte character _
            literal '%S' ignored\n", token, (char*)cp-token);
      else if (s - cbuf <= 2)
         error("missing '\n");
      tval.type = inttype;
      tval.u.c.v.i = cbuf[1];
      return ICON;
   }
```

61	array
57	chartype
111	gettok
103	limit
110	map
110	NEWLINE
167	primary
29	string
30	stringn
108	token
108	tsym
117	tval

String literals can contain null characters as the result of the escape sequence `\0`, so the length of the literal is given by its type: An n-character literal has the type `(ARRAY` n `(CHAR))` (n does not include the double quotes). `gettok` s callers, such as `primary`, call `stringn` when they want to save the string literal referenced by `tval`.

The code below, which scans a string literal or character constant, copes with four situations: newlines at `limit`, escape sequences, non-ANSI characters, and literals that exceed the size of `cbuf`.

```
⟨scan one string literal 123⟩≡                                          122
   while (*cp != cbuf[0]) {
      int c;
      if (map[*cp]&NEWLINE) {
         if (cp < limit)
```

```
            break;
         cp++;
         nextline();
         if (⟨end of input 112⟩)
            break;
         continue;
      }
      c = *cp++;
      if (c == '\\') {
         if (map[*cp]&NEWLINE) {
            if (cp < limit)
               break;
            cp++;
            nextline();
         }
         if (limit - cp < MAXTOKEN)
            fillbuf();
         c = backslash(cbuf[0]);
      } else if (map[c] == 0)
         nbad++;
      if (s < &cbuf[sizeof cbuf] - 2)
         *s++ = c;
   }
```

backslash	126
fillbuf	106
limit	103
map	110
MAXTOKEN	111
NEWLINE	110
nextline	106

If `*limit` is a newline, it serves only to terminate the buffer, and is thus ignored unless there s no more input. Other newlines (those for which `cp` is less than `limit`) and the one at the end of file terminate the while loop without advancing `cp`. `backslash` interprets the escape sequences described above; see Exercise 6.10. `nbad` counts the number of non-ANSI characters that appear in the literal; `lcc` s `-A -A` option causes warnings about literals that contain such characters or that are longer than ANSI s 509-character guarantee.

Further Reading

The input module is based on the design described by Waite (1986). The difference is that Waite s algorithm moves one partial line instead of potentially several partial lines or tokens, and does so after scanning the *first* newline in the buffer. But this operation overwrites storage before the buffer when a partial line is longer than a fixed maximum. The algorithm above avoids this problem, but at the per-token cost of comparing `limit-cp` with `MAXTOKEN`.

Lexical analyzers can be generated from a regular-expression specification of the lexical structure of the language. LEX (Lesk 1975), which is available on UNIX, is perhaps the best known example. Schreiner and

Friedman (1985) use LEX in their sample compilers, and Holub (1990) details an implementation of a similar tool. More recent generators, such as `flex`, `re2c` (Bumbulis and Cowan 1993), and ELI s scanner generator (Gray et al. 1992; Heuring 1986), produce lexical analyzers that are much faster and smaller than those produced by LEX. On some computers, ELI and `re2c` produce lexical analyzers that are faster than `lcc` s. ELI originated some of the techniques used in `lcc` s `gettok`.

A perfect hash function is one that maps each word from a known set into a different hash number (Cichelli 1980; Jaeschke and Osterburg 1980; Sager 1985). Some compilers use perfect hashing for keywords, but the hashing itself usually takes more instructions than `lcc` uses to recognize keywords.

`lcc` relies on the library function `strtod` to convert the string representation of a floating constant to its corresponding double value. Doing this conversion as accurately as possible is complicated; Clinger (1990) shows that it may require arithmetic of arbitrary precision in some cases. Many implementations of `strtod` are based on Clinger s algorithm. The opposite problem   converting a double to its string representation   is just as laborious. Steele and White (1990) give the gory details.

Exercises

105	`BUFSIZE`
104	`file`
111	`gettok`
104	`lineno`
106	`nextline`

6.1 What happens if a line longer than `BUFSIZE` characters appears in the input? Are zero-length lines handled properly?

6.2 The C preprocessor emits lines of the form

```
# n "file"
#line n "file"
#line n
```

These lines are used to reset the current line number and file name to *n* and *file*, respectively, so that error messages refer to the correct file. In the third form, the current file name remains unchanged. `resynch`, called by `nextline`, recognizes these lines and resets `file` and `lineno` accordingly. Implement `resynch`.

6.3 In many implementations of C, the preprocessor runs as a separate program with its output passed along as the input to the compiler. Implement the preprocessor as an integral part of `input.c`, and measure the resulting improvement. Be warned: Writing a preprocessor is a big job with many pitfalls. The only definitive specification for the preprocessor is the ANSI standard.

6.4 Implement the fragments omitted from `gettok`.

6.5 What happens when `lcc` reads an identifier longer than `MAXLINE` characters?

6.6 Implement `int getchr(void)`.

6.7 Try perfect hashing for the keywords. Does it beat the current implementation?

6.8 The syntax for octal constants is

octal-constant:
  `0` { *octal-digit* }

octal-digit:
  one of `0 1 2 3 4 5 6 7`

Write ⟨*octal constant*⟩. Be careful; an octal constant is a valid prefix of a floating constant, and octal constants can overflow.

6.9 The syntax for hexadecimal constants is

hexadecimal-constant:
  (`0x` | `0X`) *hexadecimal-digit* { *hexadecimal-digit* }

hexadecimal-digit:
  one of `0 1 2 3 4 5 6 7 a b c d e f A B C D E F`

Write ⟨*hexadecimal constant*⟩. Don t forget to handle overflow.

getchr 108
icon 117
MAXLINE 105

6.10 Implement

⟨*lex.c prototypes*⟩≡
```
static int backslash ARGS((int q));
```

which interprets a single escape sequence beginning at `cp`. `q` is either a single or double quote, and thus distinguishes between character constants and string literals.

6.11 Implement the code for ⟨*wide-character constants*⟩. Remember that `wchar_t` is unsigned char, so the value of the constant `L'\377'` is 255, not −1.

6.12 Reimplement the lexical analyzer using LEX or an equivalent program generator, and compare the two implementations. Which is faster? Smaller? Which is easier to understand? Modify?

6.13 How many instructions is ⟨*skip white space*⟩ on your machine? How many would it be if it used `cp` instead of `rcp`?

6.14 Write a program to generate the ⟨`gettok` *cases*⟩ for the C keywords.

6.15 `lcc` assumes that int and long (signed and unsigned) have the same size. Revise `icon` to remove this regrettable assumption.