CS39003 Compilers Laboratory
Autumn 2025

A short LEX tutorial

Lexical Analysis

- A program consists of tokens.
- Tokens in C
  - Key words (Examples: `int`, `for`, `case`, `struct`, `typedef`, `return`)
  - Constants (Examples: `-55`, `6.023e+23`, `'a'`, `"x = %d\n"`, `NULL`)
  - Identifiers (Variable and function names)
  - Operators (Arithmetic, logical, bitwise, referencing/dereferencing)
  - Item selector (`[...]`, `.`, `->`)
  - Statement terminator and separator (semicolon, comma)
  - Block delimiters (opening and closing braces)
  - Other punctuation symbols (like opening and closing parentheses)
  - Comments
  - White spaces (spaces, tabs, new-line characters)
  - `...`

Lexical Analysis

- A lexical analyzer breaks down the input program into a sequence of tokens.
- These tokens are fed to the next stage of the compilation process (parsing).
- Tokens are specified by
  - Literal strings (like `float`, `struct`, `NULL`)
  - Patterns or regular expressions (like number constants, variable names)
  - Punctuation symbols (like semicolon, opening brace, operators)
  - White spaces and comments are typically ignored

- Lex (or flex in Linux) is a lexical analyzer (also called lexer).
- Lex may be used as a standalone utility.
- The killer application of lex is to supply the sequence of tokens to a parser (like yacc or bison).

How to Run Lex

- A lex file is (usually) given the extension `.l`.
- Run `lex` on your input lex file.

```
$ lex mypatterns.l
$ flex mypatterns.l
```

- If lex succeeds, a C source file `lex.yy.c` will be created.
- If you write a `main()` function in `mypatterns.l`, then `lex.yy.c` can be compiled as a self-sufficient C file. If needed, include the library `-ll` or `-lfl` during compilation.
- If your `main()` function appears in a separate file, include `lex.yy.c` from it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lex.yy.c"
. . .
```

```
int main ()
{
    yylex();
}
```

- You may generate object files (using `gcc -c`) from your source files (including `lex.yy.c`), and join the object files later.

Parts of a Lex Program

A lex program consists of three sections.

Definitions section
%%
Rules section
%%
(*Optional*) *User functions* (*may include* `main()`)

The second %% and the subsequent third section are not mandatory.

Definitions Section

This section is for writing the initial headers for `lex.yy.c`, and for defining named patterns (regular expressions).

```
%{
    This part should consist of valid C code.
    This code will go verbatim to the top of lex.yy.c.
    Include header files, define global variables, data types, macros, and so on.
    These are needed for the successful compilation of lex.yy.c to an executable or
    an object file.
%}
Define patterns (regular expressions)
```

Regular Expressions

Reg Exp	Meaning	Examples
`x`	Literal `x`	`a`, `abcd`
`.`	Any character (except new line)	`abc.xyz`
`\.`	Literal `.`	`prog\.c`
`[]`	Character class	`[abcd123_]`
`[]`	Character class with range	`[a-zA-Z0-9_]`
`[^]`	Negated character class	`[^A-Z\n]`
`{-}`	Set difference	`[A-Z]{-}[AEIOU]`
`*`	Zero or more occurrences	`[A-Za-z]*`
`+`	One or more occurrences	`[A-Za-z]+`
`?`	Zero or one occurrence	`[+-]?[0-9]+`
`{m,n}`	Number of occurrence in the range `[m,n]`	`.{10,99}`
`{m,}`	At least `m` occurrences	`[a-z]{5,}`
`{m}`	Exactly `m` occurrences	`.{20}`
`{name}`	A regular expression defined as `name`	`{letter_}`

Regular Expressions

Reg Exp	Meaning	Examples
`()`	Grouping to alter precedence	`(ab|cde|f)(1|22|333|4444)`
`rs`	Concatenation of `r` and `s`	`[a-zA-Z]+[1-9][0-9][0-9]`
`r|s`	Either `r` or `s`	`[a-z]+|[A-Z]+`
`r/s`	`r` only if followed by `s`	`.+/\..+` (works in Rules Section only)
`^r`	Match of `r` at the beginning of a line	`^[^ \t]+`
`r$`	Match of `r` at the end of a line	`\.$` (not the same as `\.\n`)
`<<EOF>>`	End of file	

Predefined character classes

```
[:alnum:]  [:alpha:]  [:blank:]
[:cntrl:]  [:digit:]  [:graph:]
[:lower:]  [:print:]  [:punct:]
[:space:]  [:upper:]  [:xdigit:]
```

Naming Patterns

Supply a name followed by a (non-empty) sequence of white spaces and then by a regular expression. Do not use unnecessary spaces anywhere else (not even at the beginning of a definition line).

```
letter_       [a-zA-Z_]
letterdig_    [a-zA-Z0-9_]
validname     {letter_}{letterdig_}*

digits        [0-9]
integer       [+-]?{digits}+
fracpart      \.{digits}*
realnum       {integer}{fracpart}
number        ({integer}|{realnum})([eE]{integer})?

ws            [ \t]+
wsnl          [ \t\n]+
```

Rules Section

Each rule is of the following form.

pattern *action*

Here, *pattern* is a regular expression (may include names from the Definitions Section), and *action* is a C code. If the C code contains multiple statements, enclose them inside braces. Each action goes verbatim to the body of the function `yylex()`. The *action* must begin at the same line as *pattern*.

A do-nothing action can be specified by `{ }` or by a single `;` (but not by writing nothing).

Whenever a match of a *pattern* is found in the input, the corresponding *action* is taken by `yylex()`.

The lexeme that matches is stored in the global string `yytext`.

- **Leftmost** match in the input.
- **Longest** match in the input.
- If multiple patterns match the same lexeme, action will be taken for the matching pattern that is declared the **earliest**.
- You should first declare the literal patterns for the key words, and then a pattern for valid names.

User-Code Section

- Pattern matching starts by the call `yylex()`.
- The patterns are matched one by one, and the corresponding actions are taken.
- A character not matched by any of the patterns in the Rules Section will be echoed to the output. Use the following last rule to prevent this. You may have a separate action for `\n`.

```
.|\n            { }
```

- `yylex()` continues pattern matching until it encounters end-of-file (EOF) in the input.
- But reaching the EOF does not let `yylex()` return immediately.
- It instead calls a function `yywrap()` to decide whether it should wrap up.
- If `yywrap()` returns 1, then `yylex()` stops after returning 0. If `yywrap()` returns 0, then `yylex()` continues matching more patterns (from where?).
- You need to define `yywrap()` explicitly. Sometimes compiling `lex.yy.c` with `-ll` or `-lfl` introduces a default implementation of `yywrap()`, but that is not guaranteed.

```
int yywrap ( ) { return 1; }
```

A Pattern-Matching Example

- We want to find all function references (definitions and calls) in a C program.
- It is difficult (indeed not fully possible) to do this by pattern matching alone.
- Our example lex program works reasonably well subject to certain assumptions.
- A function definition is of the form `type fname (...)` or `type *fname (...)`.
- A function reference is of the form `fname (...)`, and is not preceded by a type.
- We assume that there is no line break between the function name and the opening parenthesis (argument list), and between the return type and the function name. The return type may contain multiple words (like `struct node` or `unsigned long long int`, but we match only the last word in the type.
- We take `type` and `fname` as any valid C identifier, with the following two exceptions.
  - The keywords `if`, `for`, `while`, `switch`, and `return` cannot be valid `fname`.
  - The keywords `else` and `do` are not valid as `type`.
- We will not guard against the function patterns appearing inside comments or literal strings.
- Parameterized macros are treated like function calls.

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>

int fdefn = 0, fcall = 0, lineno = 1;
%}

whitespace      [ \t]+
optwhitespace   [ \t]*
validname       [a-zA-Z_][a-zA-Z0-9_]*
invalidname     (if|for|while|switch|return)
invalidtype     (else|do)
leftparen       {optwhitespace}\(
ftypename       {validname}{whitespace}\*?{optwhitespace}{validname}

%%
```

Rules Section

```
{invalidtype}{whitespace}{invalidname}/{leftparen} { }
{invalidtype}{whitespace}{validname}/{leftparen} {
      char *p;
      p = yytext;
      while ((*p != ' ') && (*p != '\t')) ++p;  /* Skip the keyword */
      while ((*p == ' ') || (*p == '\t')) ++p;  /* Skip spaces after the keyword */
      printf("\t\tFunction call at line %d: \"%s\"\n", lineno, p);
      ++fcall;
   }
{ftypename}/{leftparen} {
      printf("\tFunction definition at line %d: \"%s\"\n", lineno, yytext);
      ++fdefn;
   }
{invalidname}/{leftparen}  { }
{validname}/{leftparen} {
      printf("\t\tFunction call at line %d: \"%s\"\n", lineno, yytext);
      ++fcall;
   }
\n { ++lineno; }
.  { }

%%
```

User Routines

```
int yywrap ()
{
   return 1;
}

int main ()
{
   yylex();
   printf("+++ %d function definitions found\n", fdefn);
   printf("+++ %d function calls found\n", fcall);
   exit(0);
}
```

Sample Run

```
$ lex example1.l
$ gcc lex.yy.c
$ ./a.out < strops.c
        Function definition at line 13: "void rmws"
        Function definition at line 26: "int validname"
        Function definition at line 46: "int STlookup"
                Function call at line 51: "strcmp"
        Function definition at line 56: "char *evalarg"
                Function call at line 61: "strchr"
                Function call at line 63: "atoi"
                Function call at line 65: "STlookup"
                Function call at line 67: "fprintf"
        Function definition at line 78: "int parse"
                Function call at line 84: "strcmp"
                Function call at line 84: "exit"
                Function call at line 85: "strchr"
                Function call at line 87: "fprintf"
                Function call at line 93: "validname"
                Function call at line 94: "fprintf"
                Function call at line 100: "strchr"
                Function call at line 107: "evalarg"
                Function call at line 112: "strlen"
                Function call at line 114: "malloc"
                Function call at line 114: "sizeof"
                Function call at line 117: "sprintf"
                Function call at line 120: "STlookup"
                Function call at line 122: "strcpy"
                Function call at line 123: "malloc"
                Function call at line 123: "sizeof"
```

Sample Run

```
                Function call at line 124: "strcpy"
                Function call at line 127: "free"
                Function call at line 128: "malloc"
                Function call at line 128: "sizeof"
                Function call at line 129: "strcpy"
                Function call at line 131: "printf"
        Function definition at line 135: "int procline"
                Function call at line 137: "rmws"
                Function call at line 138: "parse"
        Function definition at line 142: "int main"
                Function call at line 149: "printf"
                Function call at line 150: "fgets"
                Function call at line 151: "strlen"
                Function call at line 152: "procline"
                Function call at line 155: "exit"
+++ 7 function definitions found
+++ 34 function calls found
$
```

Returning Token Types

- Lex is typically used in tandem with a parser (like yacc). The pair (token_type, token_value) is in totality useful as a token to the parser.
- Only the token value (lexeme) is stored in `yytext`. The parser will have to relook at each matched lexeme to determine what the type of the token is. This is a useless duplication of effort (if possible at all).
- So far, `yylex()` is used to find matches one after another until it encounters EOF, consults `yywrap()`, and (if approved) returns 0 (thereby leaving the pattern-matching loop).
- `yylex()` is a function, and can return from anywhere within its body. The return type is `int`.
- If we define token types as `int` values, `yylex()` can return a token type when a lexeme for that token is found. The token value can be accessed from the global string `yytext`.
- This can be done by introducing `return TOKEN_TYPE;` in the action for that pattern.
- This breaks the pattern-matching loop of `yylex()`. You should call `yylex()` again so that pattern matching resumes from the position where it was aborted by the last `return`.
- Tokens that are to be ignored need not return anything in their actions.

Our Function-Reference Example with `return`: Definitions

```
%{
#include <stdio.h>
#include <stdlib.h>

/* Define token types to return. Yacc reserves the tokens 0-255 for literal characters, and
   the token 256 for internal use, so we start numbering our tokens from 257. This example
   does not use yacc, so this restriction does not apply. Let us still do it that way. */
#define FDEFN 257
#define FCALL 258

int fdefn = 0, fcall = 0, lineno = 1;
char *lexeme;
%}

whitespace      [ \t]+
optwhitespace   [ \t]*
validname       [a-zA-Z_][a-zA-Z0-9_]*
invalidname     (if|for|while|switch|return)
invalidtype     (else|do)
leftparen       {optwhitespace}\(
ftypename       {validname}{whitespace}\*?{optwhitespace}{validname}

%%
```

Our Function-Reference Example with `return`: Actions

```
{invalidtype}{whitespace}{invalidname}/{leftparen} { }
{invalidtype}{whitespace}{validname}/{leftparen} {
      lexeme = yytext;
      while ((*lexeme != ' ') && (*lexeme != '\t')) ++lexeme;
      while ((*lexeme == ' ') || (*lexeme == '\t')) ++lexeme;
      return FCALL;
   }
{ftypename}/{leftparen} {
      lexeme = yytext;
      return FDEFN;
   }
{invalidname}/{leftparen}  { }
{validname}/{leftparen} {
      lexeme = yytext;
      return FCALL;
   }
\n { ++lineno; }
.  { }

%%
```

Our Function-Reference Example with `return`: User Codes

```
int yywrap ()
{
   return 1;
}

int main ()
{
   int nexttoken;

   while ((nexttoken = yylex())) {
      switch (nexttoken) {
         case FCALL: printf("\t\tFunction call at line %d: \"%s\"\n", lineno, lexeme);
                     ++fcall; break;
         case FDEFN: printf("\tFunction definition at line %d: \"%s\"\n", lineno, lexeme);
                     ++fdefn; break;
      }
   }
   printf("+++ %d function definitions found\n", fdefn);
   printf("+++ %d function calls found\n", fcall);
   exit(0);
}
```

Reading from a File

- By default, lex reads from `stdin`.
- We redirected a file to `stdin` by the shell's input-redirection operator <.
- The input of lex is dictated by a global variable `yyin` of type `FILE *`.
- You can set `yyin` to any opened file pointer of your choice.

- Let us rewrite the user section of our last example. The other two sections remain exactly the same as in the last example.
- Now, the user has the option of specifying a file name (in the command line), from where lex should read the input.
- If that file name is not specified, then the usual read-from-`stdin` behavior of lex will be used. An input redirection from the shell can still supply a file name for the input.
- If a file name is supplied, that file is opened, and the file pointer is assigned to `yyin`.
- Now, the program can be run in any of the following two equivalent ways.

```
$./a.out < strops.c
$./a.out strops.c
```

Reading form a File: User Codes

```
int yywrap () { return 1; }

int main ( int argc, char *argv[] )
{
   int nexttoken;

   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");

   while ((nexttoken = yylex())) {
      switch (nexttoken) {
         case FCALL: printf("\t\tFunction call at line %d: \"%s\"\n", lineno, lexeme);
                     ++fcall; break;
         case FDEFN: printf("\tFunction definition at line %d: \"%s\"\n", lineno, lexeme);
                     ++fdefn; break;
      }
   }

   fclose(yyin);

   printf("+++ %d function definitions found\n", fdefn);
   printf("+++ %d function calls found\n", fcall);
   exit(0);
}
```

Reading from Multiple Files

- Redirection from multiple files is not possible from the shell.
- Setting `yyin` to a single opened file can process only one file.
- Multiple files supplied as command-line arguments should be processed one after another.
- `yywrap()` can help lex achieve that.
- `yylex()` keeps on consuming the input until it encounters EOF at its `yyin`.
- `yylex()` then calls `yywrap()` for a termination decision.
- If `yywrap()` returns 1 (wrap up), then `yylex()` stops pattern matching, and returns 0.
- If `yywrap()` returns 0 (do not wrap up), then `yylex()` resumes pattern matching.
- Only the code written in `yywrap()` can run between its call from `yylex()` and its returning 0 to `yylex()`. The continuation code written in `yywrap()` should include the following.
  - Finish the work associated with the last opened file.
  - Open the next file specified in the `argv[]` array to redefine `yyin`.
  - Initialize/Update global variables (and do other bookkeeping) before starting to read from the next file.
  - Return 0.

Reading from Multiple Files: Our Example Revisited

- We declare a global string pointer in the Definitions Section (between %{ and %}).

```
char **fname;
```

  This string will run through the entries of the argv[] array. No other change is done in the Definitions Section.

- argv[] is a NULL-terminated array, that is, argv[argc] is NULL.
- When fname reaches that NULL pointer, all files are processed, so yywrap() can return 1.
- Otherwise, the four continuation steps mentioned in the last slide are carried out.
- You can now run the compiled lex.yy.c in all of the following ways.

```
$./a.out < strops.c
$./a.out strops.c
$./a.out file1.c file2.c file3.c
$./a.out ~/compilers/assignments/*.c
```

  In the last example, the shell replaces *.c by all .c files in the directory. There may be any number of them. However, if no file matches the pattern *.c, the shell prints that, and does not run a.out at all.

Reading from Multiple Files: `yywrap()` Rewritten

```
int yywrap ()
{
   /* Finish the work for the last opened file */
   fclose(yyin);

   printf("+++ %d function definitions found\n", fdefn);
   printf("+++ %d function calls found\n", fcall);

   /* No more files to read */
   if (*fname == NULL) return 1;

   /* Open the next file and set yyin accordingly */
   printf("\n\nProcessing input from %s\n", *fname);
   yyin = (FILE *)fopen(*fname,"r");

   /* Bookkeeping before yylex() resumes its pattern-matching loop on the next file */
   ++fname;
   fdefn = fcall = 0; lineno = 1;

   return 0;
}
```

Reading from Multiple Files: `main()` Rewritten

```
int main ( int argc, char *argv[] )
{
   int nexttoken;

   if (argc == 1) {                    /* Read from stdin by default */
      printf("\n\nProcessing input from stdin\n");
      fname = argv + 1;
   } else {                            /* Read from the first file */
      printf("\n\nProcessing input from %s\n", argv[1]);
      yyin = (FILE *)fopen(argv[1],"r");
      fname = argv + 2;
   }

   while ((nexttoken = yylex())) {    /* Only one pattern-matching loop */
      switch (nexttoken) {
         case FCALL: printf("\t\tFunction call at line %d: \"%s\"\n", lineno, lexeme);
                     ++fcall; break;
         case FDEFN: printf("\tFunction definition at line %d: \"%s\"\n", lineno, lexeme);
                     ++fdefn; break;
      }
   }

   exit(0);
}
```

Input-Switch Code outside `yywrap()`

- You cannot bypass `yywrap()`. But you can write the input-switch code outside `yywrap()`.
- Even if you handle the pattern `<<EOF>>` yourself, and add the following rule to the Rules Section, `yywrap()` will be called anyway, so this rule is useless.

```
<<EOF>> { return 0; }
```

- We now let `yywrap()` return 1 always, so whenever EOF is reached, `yylex()` can return.

```
int yywrap ( )
{
    return 1;
}
```

- Insert the codes for the following tasks elsewhere (like in `main()`).
  - Finish computation on the last opened file.
  - Set `yyin` to the next opened file.
  - Do the initial bookkeeping for the next file.
  - Call `yylex()` again. This will be a fresh invocation of the lexer. So you lose the context of the earlier run of `yylex()`.

Example: A helper function

Without this function, the following code is to be written twice in `main()`.

```
void procinput ( char *ipname )
{
   int nexttoken;

   printf("\n\nProcessing input from %s\n", ipname);

   fdefn = fcall = 0; lineno = 1;
   while ((nexttoken = yylex())) {
      switch (nexttoken) {
         case FCALL: printf("\t\tFunction call at line %d: \"%s\"\n", lineno, lexeme);
                     ++fcall; break;
         case FDEFN: printf("\tFunction definition at line %d: \"%s\"\n", lineno, lexeme);
                     ++fdefn; break;
      }
   }

   /*** If yywrap() did not return 1, the above loop is not yet broken, so our program can
        proceed no further. ***/

   printf("+++ %d function definitions found\n", fdefn);
   printf("+++ %d function calls found\n", fcall);
}
```

Example: `main()` Rewritten

```
int main ( int argc, char *argv[] )
{
   int i;

   if (argc == 1) {
      procinput("stdin");
   } else {
      for (i=1; i<argc; ++i) {
         yyin = (FILE *)fopen(argv[i],"r");
         procinput(argv[i]);
         fclose(yyin);
      }
   }

   exit(0);
}
```

Note: This solution of "bypassing" `yywrap()` works in flex. If it does not work on your version of lex, or if do not want/afford to lose the context of lex during the exit from the last file, you need to revert to the role of `yywrap()` as the input-switcher.

Matching Multi-Line Patterns

- Examples: Comments in C, literal strings (here documents), argument list of a function, . . .
- Matching a single new line by a regular expression is fine. But including `\n` in a star or a plus operation of a regular expression (like `(.+\n)*`) is a very bad idea.
- Lex has a buffer of limited capacity (like two blocks of size 4KB each). Multi-line patterns longer than that size will cause overflow in the buffer.
- You can increase the buffer size of lex, but multi-line patterns may be even longer.
- It is always recommended to process the lines from the input one by one. As long as a single input line fits in the buffer (like 8192 characters), you are safe.
- We now explain how lex states can help you deal with patterns spanning across multiple lines. (User-defined states can serve the same purpose.)
- Lex states can be used in other situations. For example, a single lex program can switch among multiple pattern-matching rules while processing multiple files having different types of patterns.
- English quotes: “ . . . ” or ‘ . . . ’. German quotes: „ . . . ” or » . . . «. Spanish quotes: “ . . . ” or « . . . ». A single lex program designed to locate quotes in these three languages needs three sets of rules. A language switch may be triggered by a special keyword or by a change of file.

States of Lex

- States are also called *start conditions*. These are integers with names.
- By default, lex works in the state INITIAL (same as the integer 0).
- You can define a new state in the Definitions Section, in one of the two ways.

```
%s INCSTATE1 INCSTATE2 INCSTATE3 ...
%x EXCSTATE1 EXCSTATE2 EXCSTATE3 ...
```

- A state defined by `%s` is called an *inclusive state*.
- A state defined by `%x` is called an *exclusive state*.
- INITIAL is an inclusive state.
- At any point of time, lex can be in exactly one of the available states.
- You can make a state transition in one of the following two ways.

```
BEGIN STATE;
BEGIN(STATE);
```

States of Lex

- Patterns in the Rules Section can be marked by state names.
- A pattern marked by no state name applies to all inclusive states (including INITIAL).

```
pattern { action }
```

- A pattern marked by * applies to all states (both inclusive and exclusive).

```
<*>pattern { action }
```

- A pattern marked by specific state names applies only to those states.

```
<STATE1,STATE2,STATE3>pattern { action }
```

- If lex is in state S, then only the patterns applicable to the state S can be matched.
- The same pattern with different actions can be written for different states.

A Multi-Line Pattern-Matching Example

- We are given an input file storing plain English text.
- We want to locate all the sentences and all the paragraphs in the text.
- A sentence or a paragraph can span across multiple lines.
- We assume that a sentence ends with one the letters `.` (period), `?` (question mark), and `!` (bang). For simplicity, there will be no quotes in the input text.
- After the end of a sentence, there must be a white space (space, tab, or new line). This guards against false identifications of ends of sentences (like `Dr.H.M.Hazra` or `12.345`).
- A sequence of one or more empty lines (lines without any non-white-space letter) marks a *single* paragraph break.
- We use three states for this purpose.

`INITIAL`	This is the default state of lex. The program will be in this state so long as it is not inside a paragraph (also at the beginning of the program).
`INPARAGRAPH`	The program is inside a paragraph, and is going to read the next sentence.
`INSENTENCE`	The program has started reading a sentence, but has not yet reached the end of the sentence.

State Transitions in Our Example

- An empty line is scanned from the input.
  - If the state is INITIAL, we continue to stay in that state (multiple empty lines are counted only once).
  - If the state is INPARAGRAPH, we switch to the state INITIAL.
  - If the state is INSENTENCE, we print an error message (paragraph ends before sentence finishes), and switch to the state INITIAL.
- A non-empty line is waiting at the input.
  - If the state is INITIAL, we jump to the state INPARAGRAPH (without consuming any sentence).
  - If the state in INPARAGRAPH or INSENTENCE, this rule is not applicable.
- A non-white string ending with a sentence delimiter is read from the input.
  - This situation is not applicable to the state INITIAL.
  - If the state is INPARAGRAPH, we continue to stay in that state.
  - If the state in INSENTENCE, the sentence being currently scanned finishes, so we jump to the state INPARAGRAPH.

State Transitions in Our Example

- A non-white string extending to the end of the line but without a sentence delimiter is read from the input.
  - This situation is not applicable to the state INITIAL.
  - If the state is INPARAGRAPH, then we switch to the state INSENTENCE.
  - If the state in INSENTENCE, we continue to stay in that state.
- We may additionally consume white spaces from the beginning (and end) of sentences.

The Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int lineno = 1, sentno = 0, parno = 0;
%}

%x INPARAGRAPH
%x INSENTENCE

whitespace      [ \t]+
optwhitespace   [ \t]*
emptyline       ^{optwhitespace}\n
sentdelim       [.?!]
sentbegcont     [^ \t\n][^.?!\n]+$
sentend         [^ \t\n][^.?!\n]*{sentdelim}

%%
```

The Rules Section: Begin and End of Paragraphs

```
<INITIAL>{emptyline}       { ++lineno; }
<INPARAGRAPH>{emptyline}   {
      ++lineno;
      BEGIN INITIAL;
   }
<INSENTENCE>{emptyline}    {
      printf("\t*** Paragraph ended before sentence finishes\n");
      ++lineno;
      BEGIN INITIAL;
   }
<*>\n                      { ++lineno; }
<INITIAL>{optwhitespace}/[^ \t\n]     {
      ++parno;
      BEGIN INPARAGRAPH;
      printf("\n+++ Paragraph %d begins at line %d\n", parno, lineno);
   }
```

The Rules Section: Reading Sentences

```
<INPARAGRAPH,INSENTENCE>{optwhitespace} { }
<INPARAGRAPH>{sentbegcont}       {
      ++sentno;
      printf("\tSentence %d at line %d: %s >>\n", sentno, lineno, yytext);
      BEGIN INSENTENCE;
   }
<INSENTENCE>{sentbegcont}        {
      printf("\t\t>> %s >>\n", yytext);
   }
<INPARAGRAPH>{sentend}/[ \t\n]+  {
      ++sentno;
      printf("\tSentence %d at line %d: %s\n", sentno, lineno, yytext);
   }
<INSENTENCE>{sentend}/[ \t\n]+   {
      printf("\t\t>> %s\n", yytext);
      BEGIN INPARAGRAPH;
   }

%%
```

The User Codes

```
int yywrap () {
   return 1;
}

int main ( int argc, char *argv[] )
{
   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");

   yylex();

   exit(0);
}
```

Sample Input File (with Line Numbers)

```
 1
 2 Why need one program?
 3
 4         The electronic speed possessed by computers for processing data
 5 is really fabulous. Can you imagine a human prodigy manually multiplying
 6 two thousand digit integers flawlessly in an hour? A computer can perform
 7 that multiplication so fast that you even do not perceive that it has
 8 taken any time at all. It is wise to exploit this instrument to the best
 9 of our benefit. Why not, right?
10
11         However, there are many programs already written by professionals
12 and amateurs.  Why need we bother about writing programs ourselves?
13 If we have to find the roots of a polynomial or invert/multiply matrices
14 or check primality of natural numbers, we can use standard mathematical
15 packages and libraries. If we want to do web browsing, it is not a
16 practical strategy that everyone writes his/her own browser.
17 It is reported that playing chess with the computer could be a really
18 exciting experience, even to world champions like Kasparov. Why should
19 we write our own chess programs then? Thanks to the free (and open-source)
20 software movement, many useful programs are now available in the
21 public domain (often free of cost).
22
23         Still, we have to write programs ourselves! Here are some
24 compelling reasons.
25
26
27 * There are so many problems to solve
28
29         Simple counting arguments suggest that computers can solve
30     infinitely many problems. Given that the age of the universe and the
```

Sample Input File (with Line Numbers)

```
31    human population are finite, we cannot expect every problem to be
32    solved by others. In other words, each of us is expected to encounter
33    problems which are so private that nobody else has even bothered to
34    solve them, let alone making the source-codes or executables freely
35    available for our consumption. Sometimes programs are available for
36    solving some of our problems, but these programs are either too costly
37    to satisfy our budget or so privately solved by others that they don't
38    want to share their programs with us. If we plan to harness the
39    electronic speed of computers, there seems to be no alternative way
40    other than writing the programs ourselves.
41
42        A stupendous example is provided by the proof of the four-color
43    conjecture, a curious mathematical problem that states that, given the
44    map of any country, one can always color the states of the country
45    using only four colors in such a way that no two states that share
46    some boundary receive the same color. That five colors are sufficient
47    was known since long ago, but the four-color conjecture remained unsolved
48    for quite a long time. Mathematicians reduced the problem to checking
49    a list of configurations. But the list was so huge that nobody could
50    even think of hand-calculating for all these instances. A computer
51    program helped them explore all these possibilities. The four-color
52    conjecture finally came out to be true. Conservatives raised a huge
53    hue and cry about such filthy methods of mathematical problem solving.
54    But a problem solved happens to be a problem solved. Let them cry!
55
56        Computers can aid you solving many problems of various flavors
57    ranging from mundane to practical to esoteric to deeply theoretical.
58    Moreover, anybody can benefit from programming computers,
59    irrespective of his/her area of study. It's just your own sweet will
60    whether you plan to exploit this powerful servant.
```

Sample Input File (with Line Numbers)

```
61
62
63 * Hey, we can write better programs than them
64
65         Yes, we often can. Available programs may be too general, and
66     we can solve instances of our interest by specific programs much more
67     efficiently than the general jack-of-all-trades stuff. Moreover,
68     you may occasionally come up with brand-new algorithms that hold
69     the promise of outperforming all previously known algorithms.
70     You would then desire to program your algorithms to see how they
71     perform in reality. Designing algorithms is (usually) a more difficult
72     task than programming the algorithms, but the two may often go
73     hand-in-hand before you jump to a practical conclusion.
```

Output for the Sample Input

```
+++ Paragraph 1 begins at line 2
        Sentence 1 at line 2: Why need one program?

+++ Paragraph 2 begins at line 4
        Sentence 2 at line 4: The electronic speed possessed by computers for processing data >>
                >> is really fabulous.
        Sentence 3 at line 5: Can you imagine a human prodigy manually multiplying >>
                >> two thousand digit integers flawlessly in an hour?
        Sentence 4 at line 6: A computer can perform >>
                >> that multiplication so fast that you even do not perceive that it has >>
                >> taken any time at all.
        Sentence 5 at line 8: It is wise to exploit this instrument to the best >>
                >> of our benefit.
        Sentence 6 at line 9: Why not, right?

+++ Paragraph 3 begins at line 11
        Sentence 7 at line 11: However, there are many programs already written by professionals >>
                >> and amateurs.
        Sentence 8 at line 12: Why need we bother about writing programs ourselves?
        Sentence 9 at line 13: If we have to find the roots of a polynomial or invert/multiply matrices >>
                >> or check primality of natural numbers, we can use standard mathematical >>
                >> packages and libraries.
        Sentence 10 at line 15: If we want to do web browsing, it is not a >>
                >> practical strategy that everyone writes his/her own browser.
        Sentence 11 at line 17: It is reported that playing chess with the computer could be a really >>
                >> exciting experience, even to world champions like Kasparov.
        Sentence 12 at line 18: Why should >>
                >> we write our own chess programs then?
        Sentence 13 at line 19: Thanks to the free (and open-source)  >>
                >> software movement, many useful programs are now available in the >>
                >> public domain (often free of cost).
```

Output for the Sample Input

```
+++ Paragraph 4 begins at line 23
        Sentence 14 at line 23: Still, we have to write programs ourselves!
        Sentence 15 at line 23: Here are some >>
                >> compelling reasons.

+++ Paragraph 5 begins at line 27
        Sentence 16 at line 27: * There are so many problems to solve >>
        *** Paragraph ended before sentence finishes

+++ Paragraph 6 begins at line 29
        Sentence 17 at line 29: Simple counting arguments suggest that computers can solve >>
                >> infinitely many problems.
        Sentence 18 at line 30: Given that the age of the universe and the >>
                >> human population are finite, we cannot expect every problem to be >>
                >> solved by others.
        Sentence 19 at line 32: In other words, each of us is expected to encounter >>
                >> problems which are so private that nobody else has even bothered to >>
                >> solve them, let alone making the source-codes or executables freely >>
                >> available for our consumption.
        Sentence 20 at line 35: Sometimes programs are available for >>
                >> solving some of our problems, but these programs are either too costly >>
                >> to satisfy our budget or so privately solved by others that they don't >>
                >> want to share their programs with us.
        Sentence 21 at line 38: If we plan to harness the >>
                >> electronic speed of computers, there seems to be no alternative way >>
                >> other than writing the programs ourselves.
```

Output for the Sample Input

```
+++ Paragraph 7 begins at line 42
        Sentence 22 at line 42: A stupendous example is provided by the proof of the four-color >>
                >> conjecture, a curious mathematical problem that states that, given the >>
                >> map of any country, one can always color the states of the country >>
                >> using only four colors in such a way that no two states that share >>
                >> some boundary receive the same color.
        Sentence 23 at line 46: That five colors are sufficient >>
                >> was known since long ago, but the four-color conjecture remained unsolved >>
                >> for quite a long time.
        Sentence 24 at line 48: Mathematicians reduced the problem to checking >>
                >> a list of configurations.
        Sentence 25 at line 49: But the list was so huge that nobody could >>
                >> even think of hand-calculating for all these instances.
        Sentence 26 at line 50: A computer >>
                >> program helped them explore all these possibilities.
        Sentence 27 at line 51: The four-color >>
                >> conjecture finally came out to be true.
        Sentence 28 at line 52: Conservatives raised a huge >>
                >> hue and cry about such filthy methods of mathematical problem solving.
        Sentence 29 at line 54: But a problem solved happens to be a problem solved.
        Sentence 30 at line 54: Let them cry!

+++ Paragraph 8 begins at line 56
        Sentence 31 at line 56: Computers can aid you solving many problems of various flavors >>
                >> ranging from mundane to practical to esoteric to deeply theoretical.
        Sentence 32 at line 58: Moreover, anybody can benefit from programming computers, >>
                >> irrespective of his/her area of study.
        Sentence 33 at line 59: It's just your own sweet will >>
                >> whether you plan to exploit this powerful servant.
```

Output for the Sample Input

```
+++ Paragraph 9 begins at line 63
        Sentence 34 at line 63: * Hey, we can write better programs than them >>
        *** Paragraph ended before sentence finishes

+++ Paragraph 10 begins at line 65
        Sentence 35 at line 65: Yes, we often can.
        Sentence 36 at line 65: Available programs may be too general, and >>
                >> we can solve instances of our interest by specific programs much more >>
                >> efficiently than the general jack-of-all-trades stuff.
        Sentence 37 at line 67: Moreover, >>
                >> you may occasionally come up with brand-new algorithms that hold >>
                >> the promise of outperforming all previously known algorithms.
        Sentence 38 at line 70: You would then desire to program your algorithms to see how they >>
                >> perform in reality.
        Sentence 39 at line 71: Designing algorithms is (usually) a more difficult >>
                >> task than programming the algorithms, but the two may often go >>
                >> hand-in-hand before you jump to a practical conclusion.
```