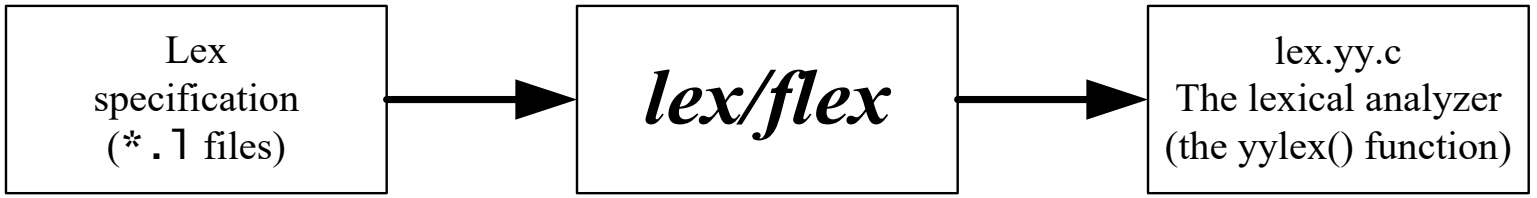
Compiler

An Introduction to Lex

What is Lex

- A lexical analyzer generator
  - Read characters and translate to token
  - Read rules from a specification file (*.l file)
  - Generate the lexical analyzer in C language



The First Impression

- The structure of a lex program – three sections
  - Definition section: any initial C/C++ codes or lex definitions
  - Rules section: pattern + action, separated by whitespaces
  - Subroutine section: any legal C or C++ codes
- Two sections are separated by a “%%” line

```
… Definition Section …
%%
… Rules Section …
%%
… Subroutine Section …
```

The Simplest Lex Program

- The lex code

```
%{
unsigned int charCount=0, wordCount=0, lineCount=0;
%}
word     [^ \t\n]+
eol      \n
%%
{word}   { wordCount++; charCount += yyleng; }
{eol}    { charCount++; lineCount++; }
.        charCount++;
%%
int main(int argc, char *argv[]) {
         yylex();
         printf("%u %u %u\n", lineCount, wordCount, charCount);
         return(0);
}
```

Practice 1

- Using nano editor to create input.txt with following content.

```
This is an apple.
```

Practice 1

- Compile and Run the Program
- How to compile?

```
$ flex -o lex.yy.c wordcount.l
$ gcc -o wordcount lex.yy.c -lfl
```

- Run the program

```
$ cat input.txt | ./ wordcount
```

```
$ ./ wordcount < input.txt
```

The Definition Section

- Define **C/C++ codes** and **Abbreviations**

Initial C or C++ codes enclosed with %{ and %}

```
%{
unsigned int charCount=0, wordCount=0, lineCount=0;
%}
word     [^ \t\n]+
eol      \n
%%
...
%%
...
}
```

Lex definitions :
Token name and corresponding REs that separated by whitespaces

The Rules Section

- The rules used by `yylex()` function

Separated by whitespaces

Action:
C/C++ statement or lex macro

```
…
%%
{word}    { wordCount++; charCount += yyleng; }
{eol}     { charCount++; lineCount++; }
.         charCount++;
%%
…
```

Pattern:
Can be a token or RE
Predefine token enclosed with { and }

Following two are different:

```
%%
.|\n      ECHO;
%%
```

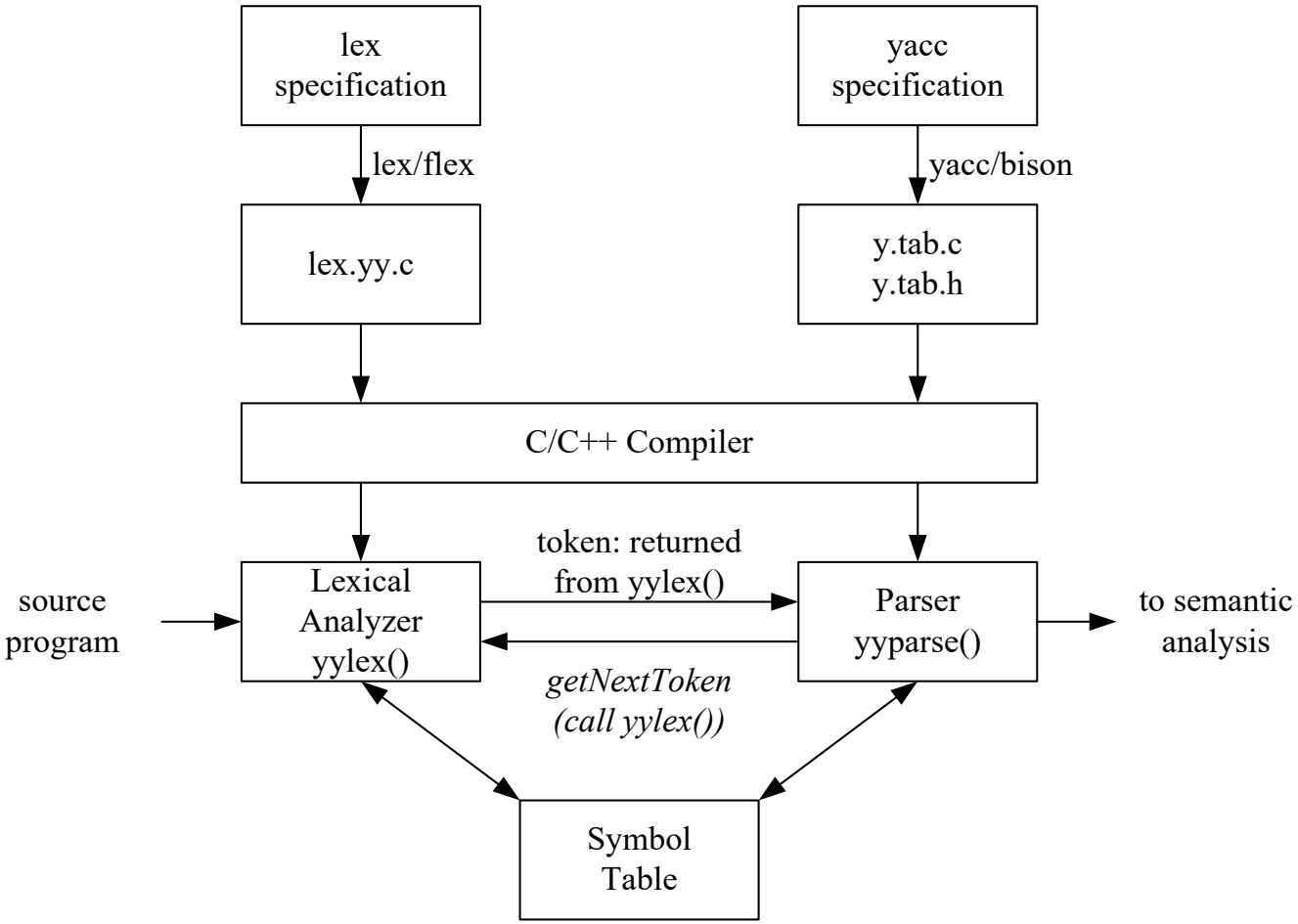
```
%%
.         |
\n        ECHO;
%%
```

The Rule Section (Cont'd)

- If C/C++ codes are more than one statements or span multiple lines, it can be enclosed in braces { }.
  - .|\n printf("rule1"); char_count++;
  - .|\n { printf("rule1"); char_count++; }

  Same
- If a pattern matches, the associated actions are executed
- We are allowed to use *return* in actions
  - Interrupt the `yylex()` analyzer with the returned value
  - The analyzer resumes when `yylex()` is called next time

The Use of yylex()



Lex Matching Rules

- Match the token with longest match
  - Input: abc
  - Rule: [a-z]+
  - The “abc” will be treated as a token
- Use the first applicable rule
  - Input: test
  - Rules:

    test { printf(“rule 1”); }
    [a-z]+ { printf(“rule 2”); }
  - It will print “rule 1”

Lex Matching Pattern – REs

RE	Purpose
.	Matches any single character except \n.
*	Matches zero or more occurrence of the preceding RE. Bo* -> B, Bo
+	Matches one or more occurrence of the preceding RE Bo+ -> Bo, Boo, Booo
?	Matches zero or one occurrence of the preceding RE Bo? -> B, Bo
[]	Matches a character listed in brackets. If the first character is a circumflex (^), it inverses the meaning to match a not listed character. This is also called a character class. [abc] -> a, b, c [_a-z] -> a, b, …, z [^abc] -> 非a, b, c
^	Matches the beginning of a line
$	Matches the end of a line

Lex Matching Pattern – REs (Cont'd)

RE	Purpose
|	Matches either the preceding RE or the following RE a|b -> a, b (a|b is equal to [ab])
\	The escape character. \. -> .
{}	Indicate how many times the preceding pattern is allowed to match. {times-lower-bound[,times-upper-bound]} ok{2} -> okok ok{2,} -> okokok ok{2,3} -> okokok
"…"	Interprets everything within the quotation marks literally "int" -> int
()	Group a series of REs together into a new RE (abc|ABC) -> abc, ABC (abc|ABC) + -> abcABCabc

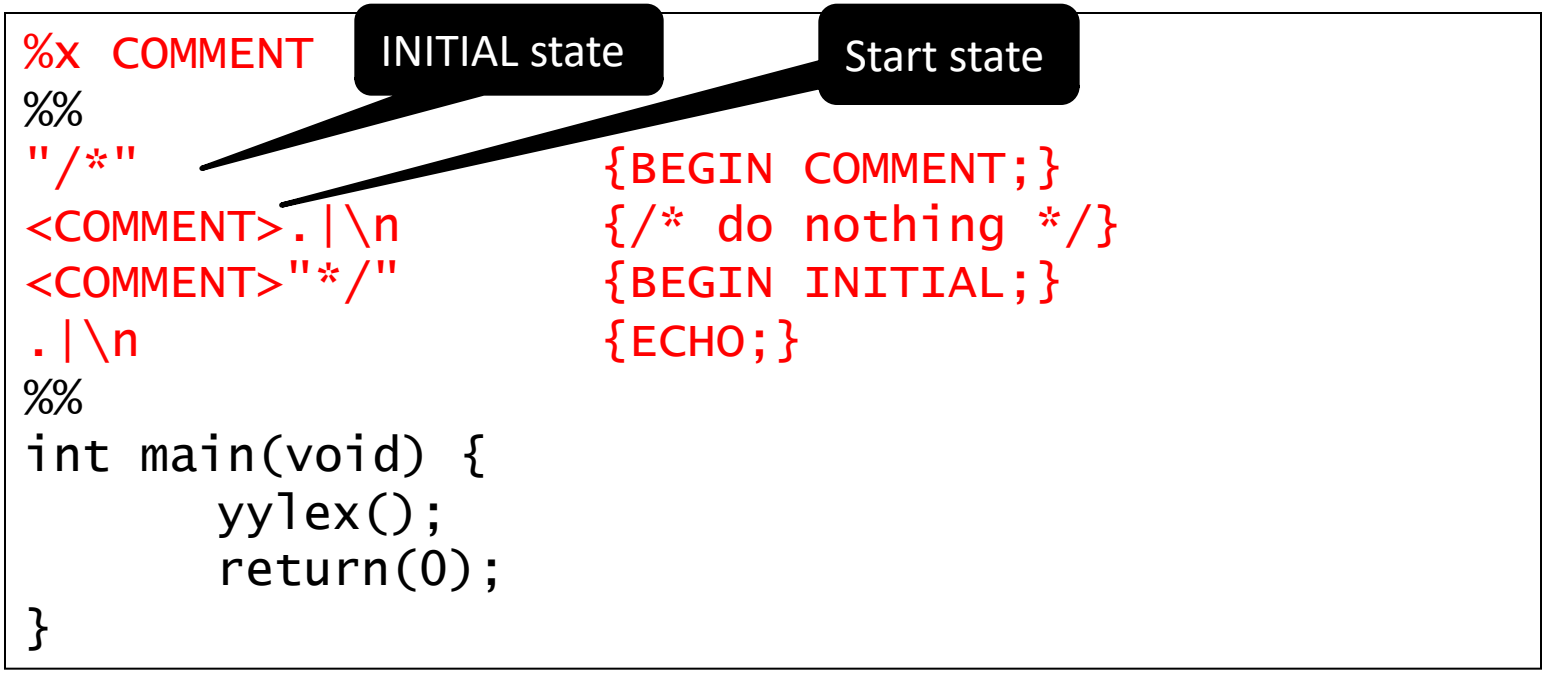
Practice 2

```
%{
unsigned int charCount=0, wordCount=0, lineCount=0;
%}
digit    [0-9]
%%
{digit}  { ECHO; printf(" is digit.\n"); }
%%
int main(int argc, char *argv[]) {
        yylex();
        return(0);
}
```

- digits [0-9]+ , {digit}+
- integer [+-]?{digits}
- float {digits}\.{digits}
- letter [a-zA-Z_]
- identifier {letter}({letter}|{digit})*

How to write comments rule

- Identify C-like comments /* … */




```
%x COMMENT
%%
"/*"                {BEGIN COMMENT;}
<COMMENT>.|\n       {/* do nothing */}
<COMMENT>"*/"       {BEGIN INITIAL;}
.|\n                {ECHO;}
%%
int main(void) {
        yylex();
        return(0);
}
```

Match with Start States

- Limit the scope of rules
  - A rule can be limited to be available only in a *start state*
- Start states are defined in the definition section
  - %s *state-name*: Define a regular start state
  - %x *state-name*: Define a exclusive start state

More on Start States

- Lex starts with a regular INITIAL start state
  - start state = INITIAL
- Start state can be switched using the *BEGIN* macro in actions
- When lex is in a regular start state
  - Rules with the INITIAL state or the matched state are both used for matching
- When lex is in an exclusive start state
  - Only rules with the matched state are used for matching

Common Lex Variables

- yyout
  - The FILE* used for output, it is equivalent to stdout by default
- yyin
  - The FILE* used for read input, it is equivalent to stdin by default
- yytext
  - Store the matched token
- yyleng
  - Store the length of the matched token

Common Lex Macros

- ECHO
  - It is equivalent to `fprintf(yyout, "%s", yytext)`
  - An example: ECHO;
- BEGIN
  - Switch the start state
  - An example: BEGIN INITIAL;

Practice2

- Add ‘comment’ to “word_count.l”

- Compile and run the analyzer

```
$ flex -o lex.yy.c word_count.l
$ gcc -o simple lex.yy.c -lfl

$ ./simple < /etc/passwd
```