南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 2

Yepang Liu

liuyp1@sustech.edu.cn

Agenda

- Recognizing tokens using transition diagram
- Introduction to Flex

Recognition of Tokens

- Lexical analyzer examines the input string and finds a prefix that matches one of the token patterns

- The first thing when building a lexical analyzer is to define the patterns of tokens using regular definitions

- A special token: $\texttt{ws} \rightarrow (\texttt{blank} \mid \texttt{tab} \mid \texttt{newline})^{+}$
  - When the lexical analyzer recognizes a whitespace token, it does not return it to the parser, but restarts from the next character

Example: Patterns and Tokens

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

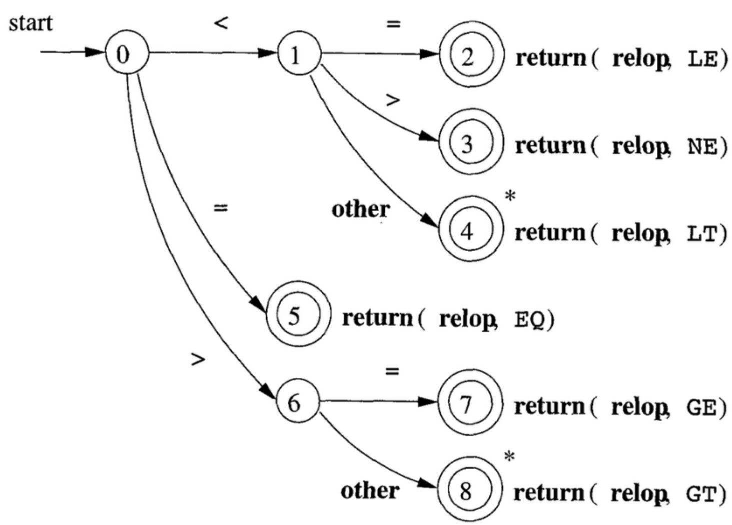
Patterns for tokens

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

Lexemes, tokens, and attribute values

Transition Diagrams (状态转换图)

- An important step in constructing a lexical analyzer is to convert patterns into "transition diagrams"
- Transition diagrams have a collection of nodes, called *states* (状态) and *edges* (边) directed from one node to another

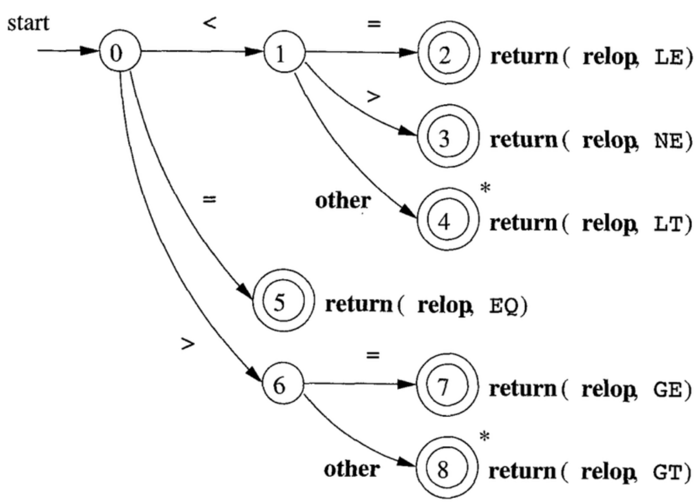


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

The transition diagram in the left recognizes **relop** tokens

States

- Represent conditions that could occur during the process of scanning (i.e., what characters we have seen)
- The *start state* (开始状态), or *initial state*, is indicated by an edge labeled "start", which enters from nowhere
- Certain states are said to be *accepting* (接受状态), or *final*, indicating that a lexeme has been found

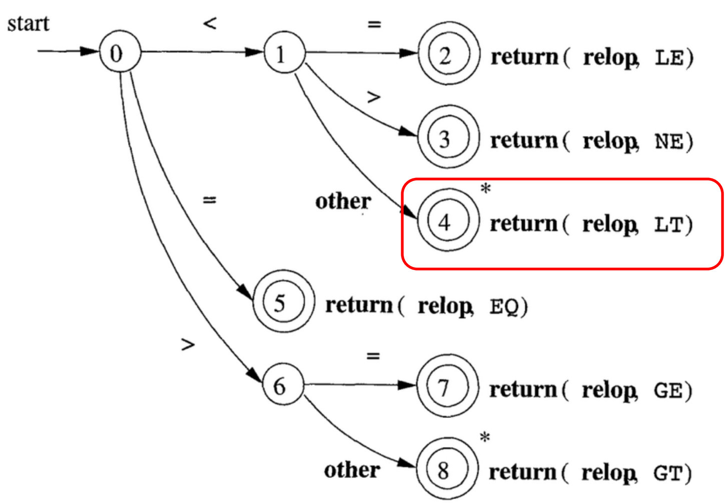


States 2-8 are accepting. They return a pair (token name, attribute value).

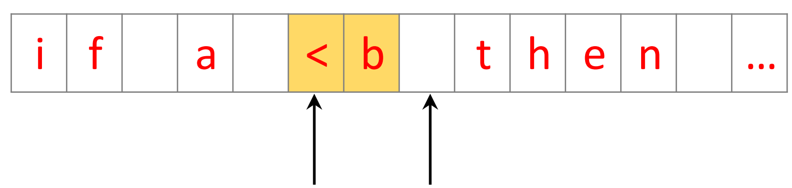
By convention, we indicate accepting states by double circles

The Retract Action

- At certain accepting states, the found lexeme may not contain all characters that we have seen from the start state (such states are annotated with *)
- When entering * states, it is necessary to retract (回退) the `forward` pointer, which points to the next char in the input string



- The found lexeme: <
- The characters we've seen: <b

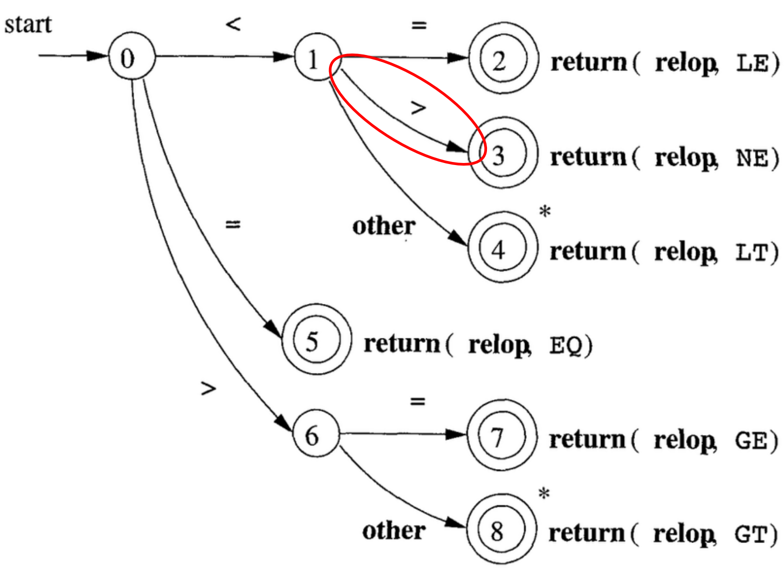


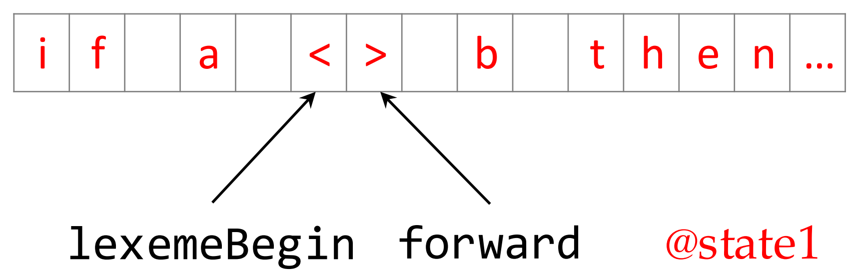
`lexemeBegin` `forward`

We should retract `forward` one step back

Edges

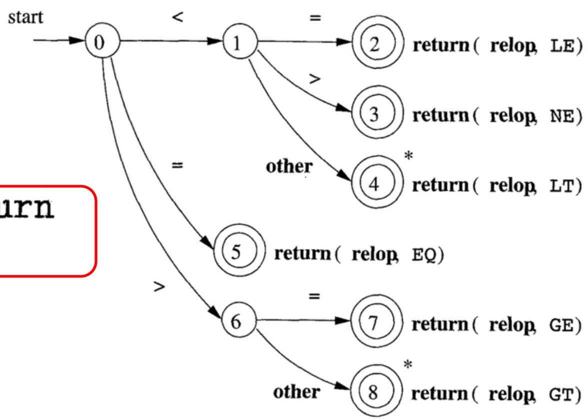
- *Edges* are directed from one state to another
- Each edge is labeled by a symbol or set of symbols





In the above case, we should follow the circled edge to enter state 3 and advance the `forward` pointer

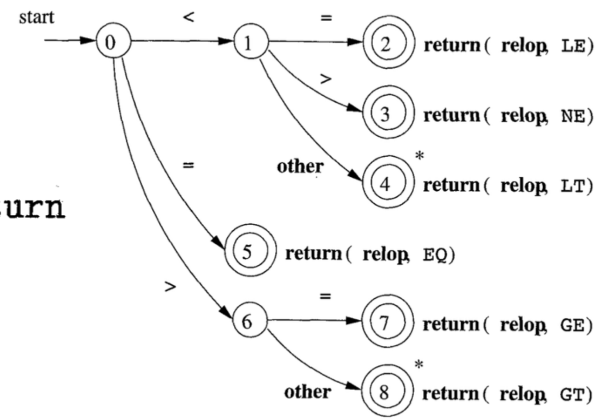
Building a Lexical Analyzer from Transition Diagrams




```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

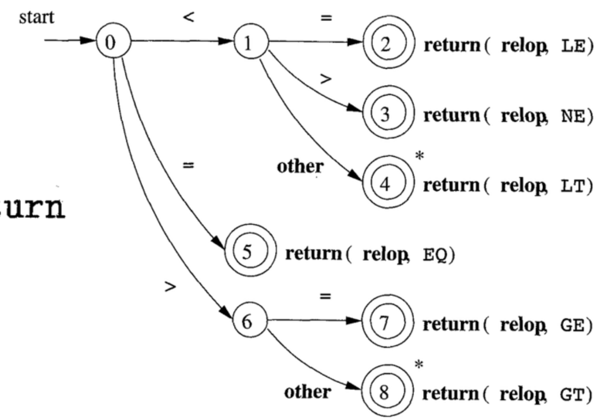



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Use a variable `state` to record the current state

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

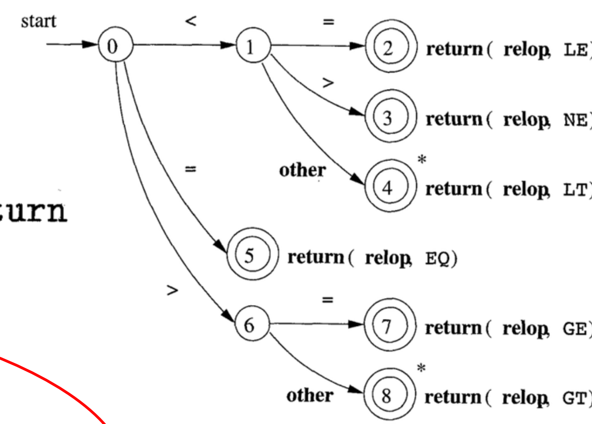



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

A `switch` statement based on the value of `state` takes us to the processing code

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams



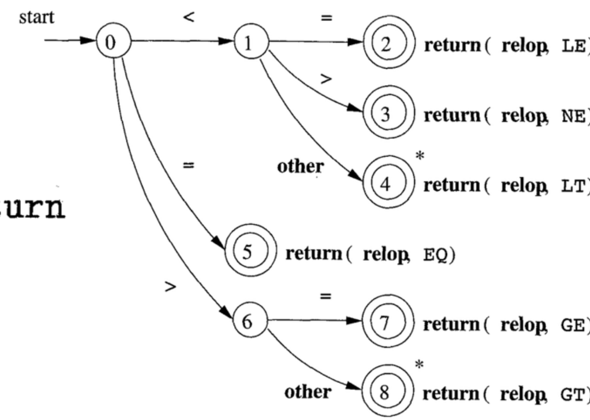

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Sketch implementation of `relop` transition diagram

The code of a normal state:

1. Read the next character
2. Determine the next state
3. If step 2 fails, do error recovery

Building a Lexical Analyzer from Transition Diagrams




```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

The code of an accepting state:

1. Perform retraction if the state has *
2. Set token attribute values
3. Return the token to parser

Sketch implementation of `relop` transition diagram

Building the Entire Lexical Analyzer

- **Strategy 1:** Try the transition diagram for each type of token sequentially
  - `fail()` resets the pointer `forward` and tries the next diagram
- **Problem:** Not efficient
  - May need to try many irrelevant diagrams whose first edge does not match the first character in the input stream

Building the Entire Lexical Analyzer

- **Strategy 2**: Run transition diagrams in parallel
  - Need to resolve the case where one diagram finds a lexeme and others are still able to process input.
  - **Solution:** take the longest prefix of the input that matches any pattern

- **Problem:** Requires special hardware for parallel simulation, may degenerate into the sequential strategy on certain machines

Building the Entire Lexical Analyzer

- **Strategy 3:** Combining all transition diagrams into one
  - Allow the transition diagram to read input until there is no possible next state
  - Take the longest lexeme that matched any pattern

- This is a commonly-adopted strategy in real-world compiler implementation (efficient & requires no special hardware)

How? Be patient ☺, we will talk about this later.

Agenda

- Recognizing tokens using transition diagram
- Introduction to Flex

The Lexical-Analyzer Generator `Lex`

- `Lex`, or a more recent tool `Flex`, allows one to specify a lexical analyzer by specifying regexps to describe patterns for tokens
- Often used with Yacc/Bison to create the frontend of compiler

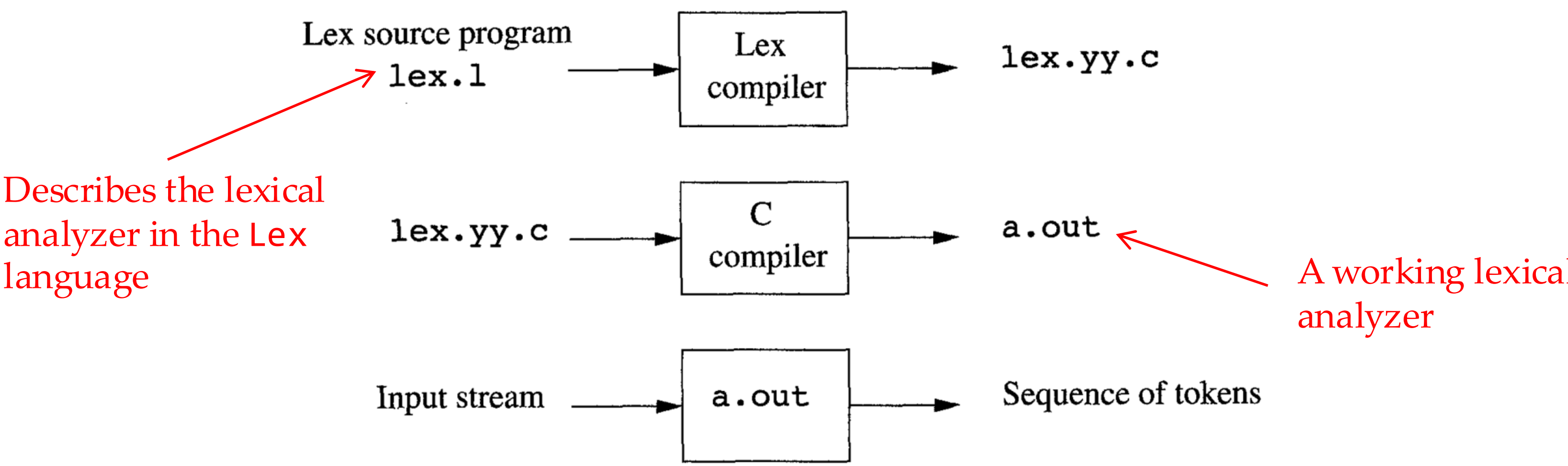


Figure 3.22: Creating a lexical analyzer with `Lex`

Structure of Lex Programs

- **A Lex program has three sections separated by %%**
  - Declaration (声明)
    - Variables, constants (e.g., token names)
    - Regular definitions
  - Translation rules (转换规则) in the form "Pattern {Action}"
    - Each pattern (模式) is a regexp (may use the regular definitions of the declaration section)
    - Actions (动作) are fragments of code, typically in C, which are executed when the pattern is matched
  - Auxiliary functions section (辅助函数)
    - Additional functions that can be used in the actions

Lex Program Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

Anything in between `%{` and `}%` is copied directly to `lex.yy.c`.

In the example, there is only a comment, not real C code to define manifest constants

Regular definitions that can be used in translation rules

Section separator

Lex Program Example Cont.

Continue to recognize other tokens

Return token name to the parser

Place the lexeme found in the symbol table

Literal strings*

```
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}

%%
```

A global variable that stores a pointer to the symbol table entry for the lexeme. Can be used by the parser or a later component of the compiler.

* The characters inside have no special meaning (even if it is a special one such as *).

Lex Program Example Cont.

- Everything in the auxiliary function section is copied directly to the file `lex.yy.c`
- Auxiliary functions may be used in actions in the translation rules

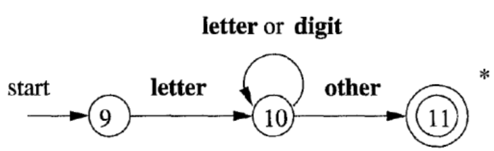
```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Variables defined and set automatically by the lexical analyzer `Lex` generates

Conflict Resolution

- When the generated lexical analyzer runs, it analyzes the input looking for prefixes that match any of its patterns. *
- **Rule 1:** If it finds multiple such prefixes, it takes the longest one
  - The analyzer will treat <= as a single lexeme, rather than < as one lexeme and = as the next
- **Rule 2:** If it finds a prefix matching different patterns, the pattern listed first in the `Lex` program is chosen.
  - Identifier pattern can also match keywords



  - If the keyword patterns are listed before identifier pattern, the lexical analyzer will not recognize keywords as identifiers

* See Flex manual for details (Chapter 8: How the input is matched) at http://dinosaur.compilertools.net/flex/

Flex

- Flex的前身是Lex。Lex是1975年由Mike Lesk和当时还在贝尔实验室做暑期实习的Eric Schmidt（前谷歌CEO），共同完成的一款基于Unix环境的词法分析程序生成工具。虽然Lex很出名并被广泛使用，但它的低效和诸多问题也使其颇受诟病。

- 1987年伯克利实验室（隶属美国能源部的国家实验室）的Vern Paxson使用C语言重写Lex，并将这个新程序命名为Flex（Fast Lexical Analyzer Generator）。无论从效率上还是稳定性上，Flex都远远好于它的前辈Lex。

*我们在Linux下使用的是`Flex`在BSD License下的版本（和Bison不同，Flex不属于GNU计划）。

An Example Flex Program

- A word-counting program (the Flex source file can be found on our blackboard course site)

- Build the program with the following commands
  - `flex lex.l`
    - After running the command, you will see a lex.yy.c file generated
  - `gcc lex.yy.c -lfl -o wc.out`
    - After running the command, you will see an executable file wc.out generated

Note: Install gcc first if it is not available on your machine: `sudo apt install gcc`

An Example Flex Program

```
cs323@deb-cs323-compilers:~/cs323/lab2$ ls -l
total 12
-rw-r--r-- 1 cs323 cs323 6525 Sep 14 05:25 inferno3.txt
-rw-r--r-- 1 cs323 cs323 1242 Sep 14 05:25 lex.l
cs323@deb-cs323-compilers:~/cs323/lab2$ flex lex.l
cs323@deb-cs323-compilers:~/cs323/lab2$ ls -l
total 60
-rw-r--r-- 1 cs323 cs323  6525 Sep 14 05:25 inferno3.txt
-rw-r--r-- 1 cs323 cs323  1242 Sep 14 05:25 lex.l
-rw-rw-r-- 1 cs323 cs323 45322 Sep 14 05:42 lex.yy.c
cs323@deb-cs323-compilers:~/cs323/lab2$ gcc lex.yy.c -lfl -o wc.out
cs323@deb-cs323-compilers:~/cs323/lab2$ ls -l
total 88
-rw-r--r-- 1 cs323 cs323  6525 Sep 14 05:25 inferno3.txt
-rw-r--r-- 1 cs323 cs323  1242 Sep 14 05:25 lex.l
-rw-rw-r-- 1 cs323 cs323 45322 Sep 14 05:42 lex.yy.c
-rwxrwxr-x 1 cs323 cs323 27432 Sep 14 05:43 wc.out
cs323@deb-cs323-compilers:~/cs323/lab2$ ./wc.out inferno3.txt
#lines  #words  #chars  file path
162     1088    6525    inferno3.txt
cs323@deb-cs323-compilers:~/cs323/lab2$ ./wc.out lex.l
#lines  #words  #chars  file path
40      167     1242    lex.l
cs323@deb-cs323-compilers:~/cs323/lab2$ ./wc.out lex.yy.c
#lines  #words  #chars  file path
1776    6735    45322   lex.yy.c
```

A Closer Look

```
%{
    // just let you know you have macros!
    // C macro tutorial in Chinese: http://c.biancheng.net/view/446.html
    #define EXIT_OK 0
    #define EXIT_FAIL 1

    // global variables
    int chars = 0;
    int words = 0;
    int lines = 0;
%}
letter [a-zA-Z]

%%
{letter}+ { words++; chars+=strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }

%%
int main(int argc, char **argv){
    char *file_path;
    if(argc < 2){
        fprintf(stderr, "Usage: %s <file_path>\n", argv[0]);
        return EXIT_FAIL;
    } else if(argc == 2){
        file_path = argv[1];
        if(!(yyin = fopen(file_path, "r"))){
            perror(argv[1]);
            return EXIT_FAIL;
        }
        yylex();
        printf("%-8s%-8s%-8s%s\n", "#lines", "#words", "#chars", "file path");
        printf("%-8d%-8d%-8d%s\n", lines, words, chars, file_path);
        return EXIT_OK;
    } else{
        fputs("Too many arguments! Expected: 2.\n", stderr);
        return EXIT_FAIL;
    }
}
```

The structure is the same as in a Lex program:

1. Declaration
2. Translation rules
3. Auxiliary functions

More on Flex patterns (self-study)

Flex supports a rich set of conveniences:

Character classes	`[0-9]`	This means alternation of the characters in the range listed (in this case: **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**). More than one range may be specified, e.g. `[0-9A-Za-z]` as well as specifying individual characters, as with `[aeiou0-9]`.
Character exclusion	`^`	The first character in a character class may be `^` to indicate the complement of the set of characters specified. For example, `[^0-9]` matches any non-digit character.
Arbitrary character	`.`	The period matches any single character **except newline**.
Single repetition	`x?`	0 or 1 occurrence of `x`.

More on Flex patterns (self-study)

Non-zero repetition	**x+**	**x** repeated one or more times; equivalent to **xx*.**
Specified repetition	**x{n,m}**	**x** repeated between **n** and **m** times.
Beginning of line	^x	Match **x** at beginning of line only.
End of line	**x$**	Match **x** at end of line only.
Context-sensitivity	**ab/cd**	Match **ab** but only when followed by **cd.** The lookahead characters are left in the input stream to be read for the next token.
Literal strings	**"x"**	This means **x** even if **x** would normally have special meaning. Thus, **"x*"** may be used to match **x** followed by an asterisk. You can turn off the special meaning of just one character by preceding it with a backslash, .e.g. **\\.** matches exactly the period character and nothing more.
Definitions	**{name}**	Replace with the earlier defined pattern called **name**. This kind of substitution allows you to re-use pattern pieces and define more readable patterns.

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf