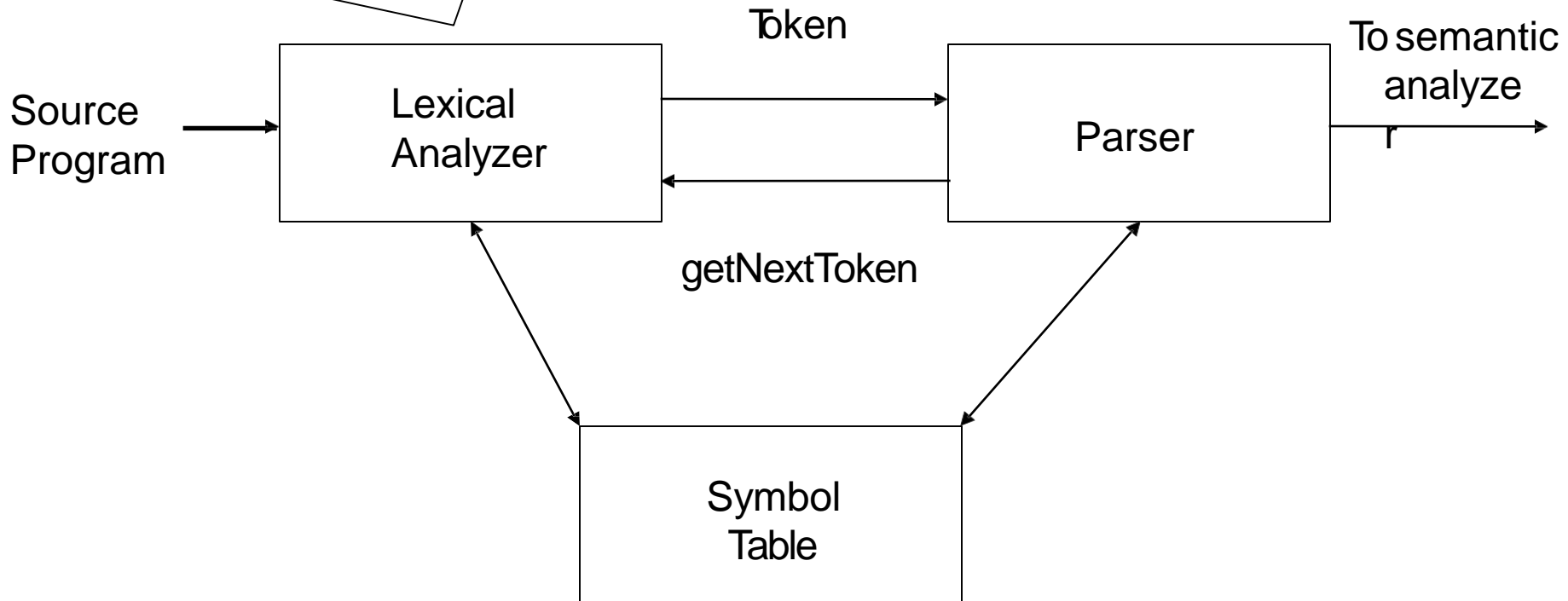# Lexical Analyzer

Using Flex

Lexical Analysis

- First phase of a Compiler

- Also called Scanner

- Scans the character stream of the source program

- Groups them into meaningful sequences
 - Output: A sequence of token

Role of Lexical Analyzer

✓ Identify Tokens
✓ Remove Whitespace
✓ Install lexeme in symbol table
✓ Returns token to parser

Source Program → Lexical Analyzer

Token →

Lexical Analyzer → Parser

getNextToken

Parser → To semantic analyzer

Symbol Table

Flex- The First Lexical Analyzer Generator

- No need to write the code
- Tools that produce the analyzer/ scanner quickly and automatically
- Also known as tokenizer recognizing lexical patterns in text
- Originally written in the C programming language by Vern Paxson in 1987

Lex Tool

Lex source lex.l → Lex Compiler → lex.yy.c

lex.yy.c → C Compiler → a.out

Source program → a.out → Tokens
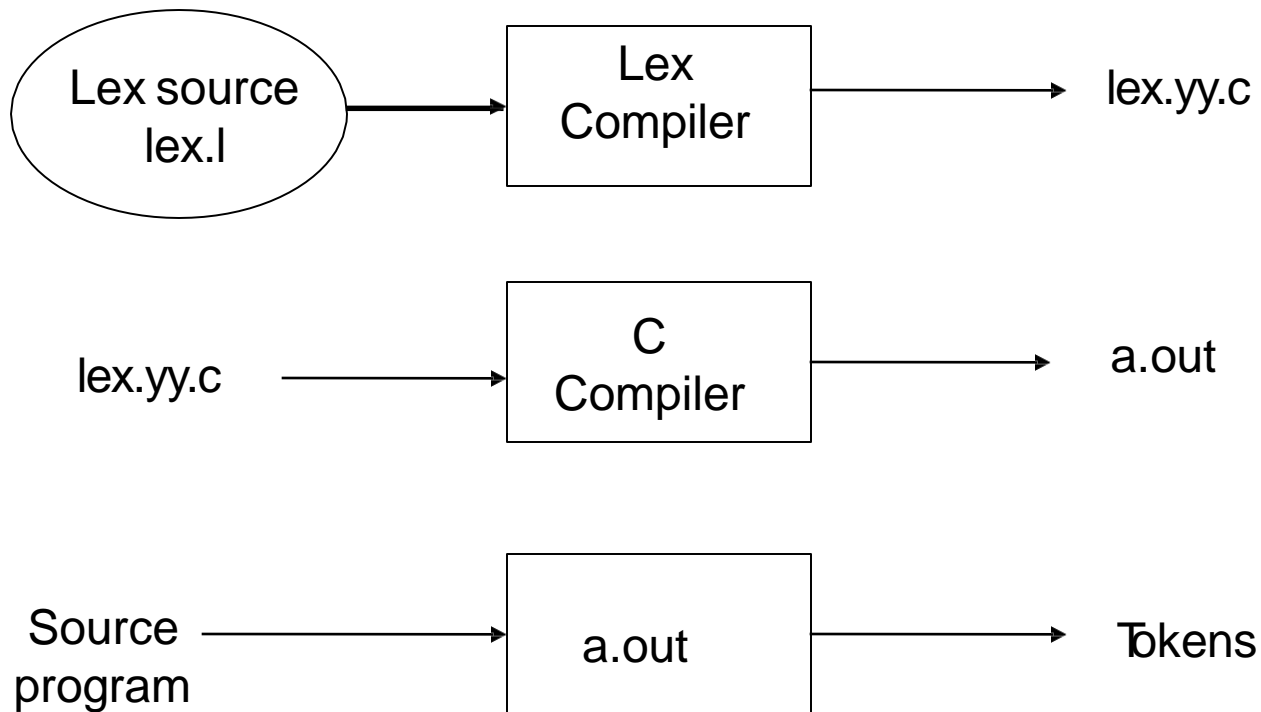
Download link

Flex:

Download from here

Token, Pattern, Lexeme

- Token: Set of strings that represent a particular construct in source language

- Pattern: Rules that describe that string set
  - It matches each string in the set

- Lexeme: Sequence of characters that is matched by a pattern for a token

Example

Token	Sample Lexemes	Pattern Description
WHILE	while	while
RELOP	<, <=, >, >=, <>, ==	< or <= or > or >= or <> or ==
ID	count, account, flag2	letter followed by letters and digits
C comment	/* any comment*/	anything between /* and */
NUM	3.14, 3.2E+5, 5.9E-2	sequence of digits having fraction and exponent

Structure of Lex Programs

```
%{
    #include<stdio.h>
    int Word_count;
%}
```
// anything here is directly copied to lex.yy.c
//include header files and global variables

Declarations
// regular definitions
```
%%
```
Transition rules
//token matching & actions
```
%%
```
auxiliary functions
// any other functions

Regular Expressions

- Specifies a set of strings to match
- One expression for each token pattern
- Some expression
  - [\t\n] //for delimiter
  - [\t\n]+ // for white space
  - a(b)*   //a followed by zero or more occurrence of b
    //a, ab, abb, abbb

Regular Expressions

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Regular Expressions

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

Table 2: Pattern Matching Examples

Transition rules

- Pattern   { Action }

Regular expressions
to
Match the **token**

C code
to
Do the **functions**

Actions

- Specify what to do if a rule matches a token
- Basically C code
- Examples

```
%%
[a-zA-z]     {
                    printf("I found a letter");
             }
[0-9]        {
                    printf("I found a digit");
             }
[  \t\n]     {
                    // actually I do nothing
             }
%%
```

Structure of Lex Programs

```
%{
    #include<stdio.h>
    int Word_count;

%}

    // regular definitions Declarations

%%

    [0-9]           {
                        printf("I found a digit");
                    }
%%
auxiliary functions          // any other functions
```

Regular Definitions

- Give symbolic name to regular expressions
- Examples

delim	[\t\n]
ws	{delim}+
digit	[0-9]
number	{digit}+

Lex Predefined Variables

Name	Function
char *yytext	Pointer to matched string
int yyleng	Length of matched string
FILE *yyin	Input stream pointer
FILE *yyout	Output stream pointer
int yylex()	Invoke lex
int yywrap	Wrapup, return 1 if done, otherwise 0

Complete Lex Source

```
%option noyywrap
%{
    #include<stdio.h>
    int word_count = 0;
%}
delim        [  \t\n]
digit        [0-9]
letter       [A-Za-z]
id           {letter}+
%%
if           {printf("<KEYWORD, %s>\n",yytext);}
exit         {return 1;}
{id}         {printf("<ID, %s>",yytext); word_count++;}
{delim}+     {    }        //no action
{digit}+     {  printf("Here I found a digit");    word_count++; }
%%
int main()
{
yylex();
printf("Total Count: %d",word_count);
}
```

Compilation Code

Run Cmd Prompt and change directory to the folder where the lexfile   is saved using cd Command.

Run the following command to compile the lex file

Command:

flex filename.l

This command will generate lex.yy.c

Command:

flex -t filename.l > filename.c

This command will generate filename.c instead of lex.yy.c

Now run the .c file using C compiler give input and get the token as output.

File Input/Output

Modify main() function of .l file to use file i/o

```
int main()
{
yyin=fopen("in.txt","r");          //opening a file in read mode and passing the
                                        pointer to yyin to take input from file

yyout=fopen("out.txt","w");        //opening a file in write mode and passing the
                                        pointer to yyout to give output to file

yylex();                           //invoking lexer program
fprintf(yyout,"Total word count %d \n",word_count); //output to file using fprintf
fclose(yyin);                      //closing file
fclose(yyout);
}

// input file must be in the same folder where .l file exist
```

Question?