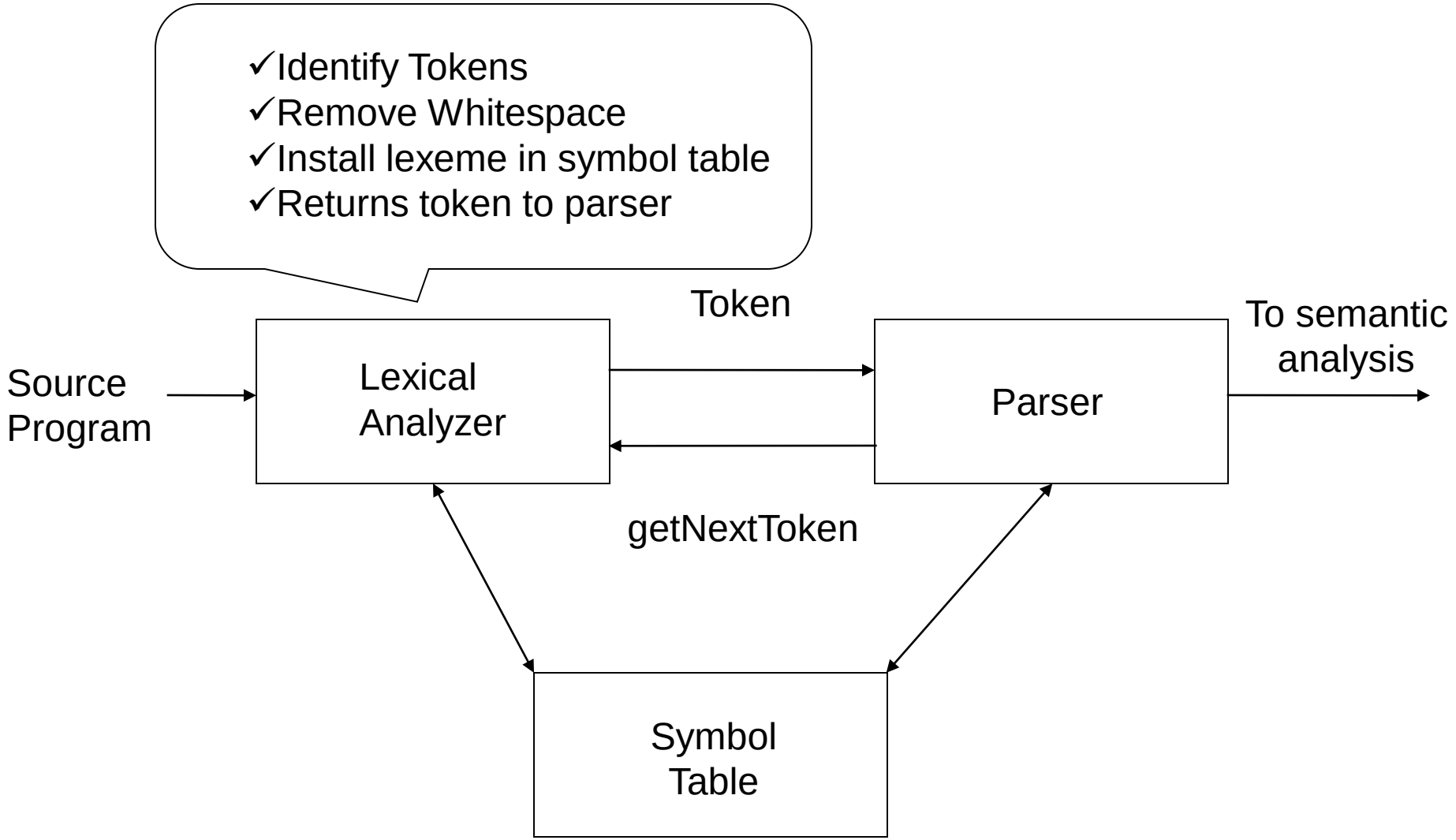
Lexical Analyzer

Using Flex

Lexical Analysis

- First phase of a Compiler
- Also called Scanning
- Scans the character stream of the Source program
- Groups them into meaningful sequences
  - Output: A sequence of token

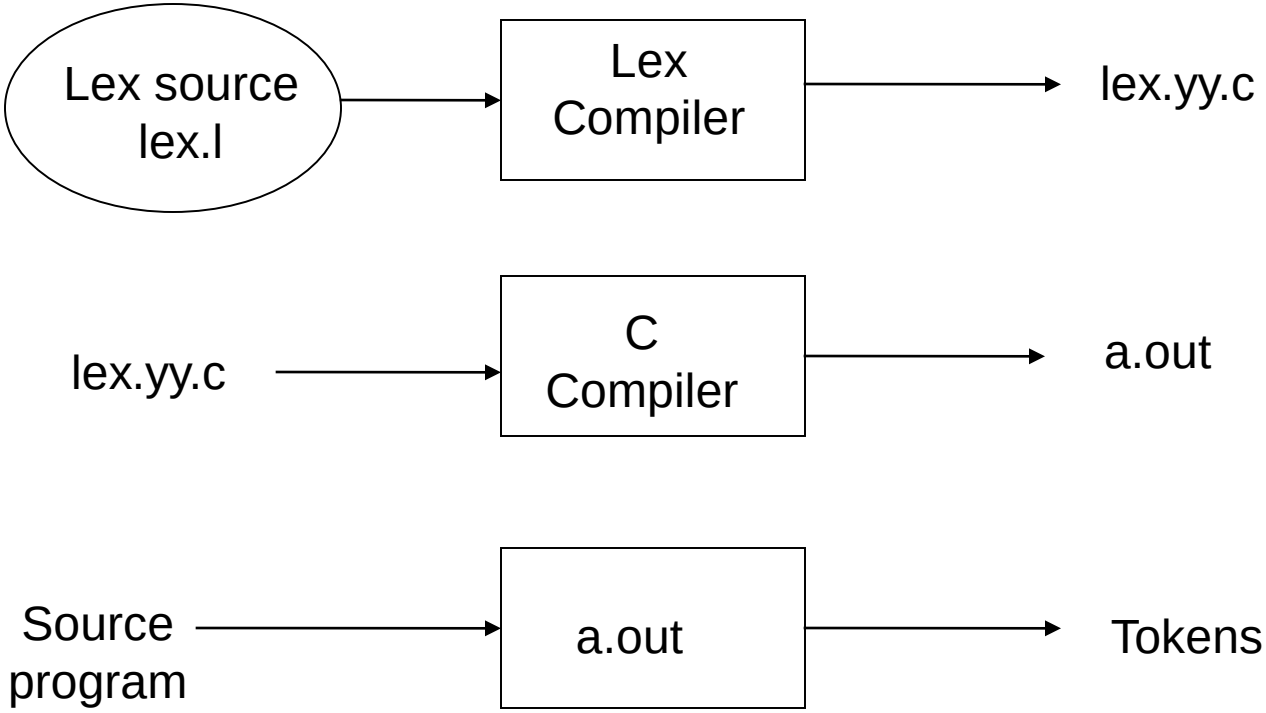
Role of Lexical Analyzer



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:

http://gnuwin32.sourceforge.net/packages/flex.htm

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
%%
Transition rules
%%
auxiliary functions
```

// regular definitions

//token matching & actions

// 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(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

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>, %d",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
}

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

Assignment

- Write a lexical analyzer for C.
  - Ignore white space
  - Match all keywords
  - Match all Identifiers(variables)
  - Match all numbers
  - Match parentheses, curly braces
  - Match Comma, Semicolon, Colon etc.
  - Count line numbers
  - Ignore single line comment

Assignment

- Variables start with a letter or underscore (_)
  - Ex : a, a9bc, _abc but not 8cde.
- Numbers may contain optional fraction or exponent
  - Ex: 3, 3.056, 3.45E5, 3.45E-2, 3E+2
- Single Line Comment is anything preceded by //

Assignment

- Arithmetic operators are +, -, *, /, %
- Relational operators are ==,!=, < , <=, >=, >
- Logical Operators &&, || , !
- Other tokens to match

  = (assignment operator, token name is ASSIGNOP)

  ++ (Increment operator , token name is INC)

  -- (Decrement operator , token name is DEC)

Assignment

- Keywords to match
  - if
  - else
  - else if
  - switch
  - case
  - while
  - for
  - int
  - break
  - default
  - main
  - printf

Token name for parser is keyword name with capital letter

Assignment

For each token print the corresponding token name and line no of occurring.

Take input from file and give output to file

See Sample Input/Output For Further Reference

Question?