Compiler course

Chapter 1

Lexical Analyzer Generator

AMRITA VISHWA VIDYAPEETHAM School of Engineering,- Dept .of Computer Science & Engineering

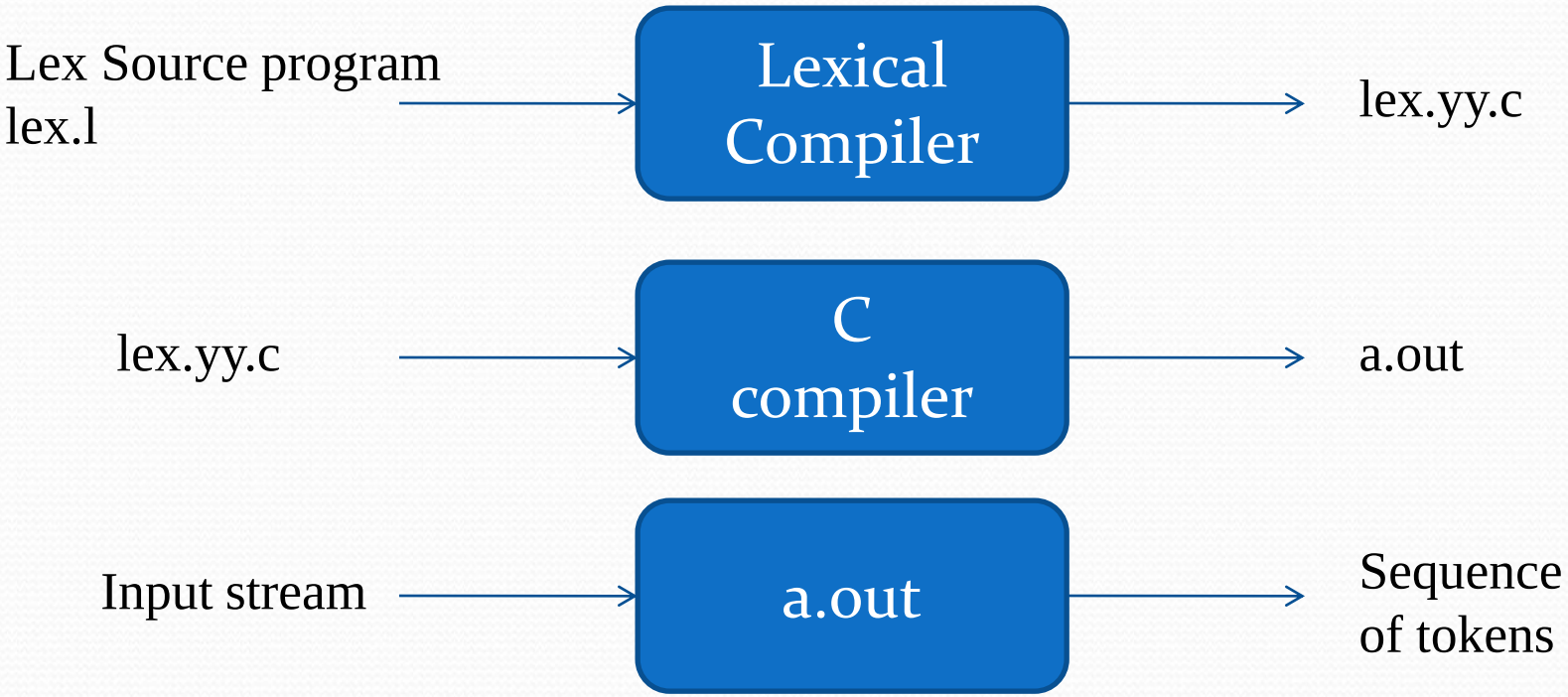
Lexical Analyzer Generator - Lex

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex

- Firstly lexical analyzer creates a program lex.l in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations
%%
translation rules → Pattern {Action}
%%
auxiliary functions

- **Declarations**
  - Include declarations of constant, variable and regular definitions.
- **Transition Rules**
  - Define the statement of form p1 {action1} p2 {action2}....pn {action}.
  - Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.
- **User subroutines**
  - Are auxiliary procedures needed by the actions.
  - The subroutine can be loaded with the lexical analyzer and compiled separately.

Declarations

- The declarations section consists of two parts, *regular definitions* and *auxiliary declarations*.
- A regular definition in LEX is of the form :
  - D R
    - where D is the symbol representing the regular expression R.
- The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' .
- It is generally used to declare functions, include header files, or define global variables and constants

Transition Rules

- Rules in a LEX program consists of two parts :
  - The pattern to be matched
  - The corresponding action to be executed
- Example:

```
    /* Declarations*/
%%
 {number} {printf(“ number”);}
{op}        {printf(“ operator”);}
 %%
    /* Auxiliary functions */
```

Auxiliary functions

- LEX generates C code for the rules specified in the Rules section and places this code into a single function called yylex(). In addition to this LEX generated code, the programmer may wish to add his own code to the lex.yy.c file. The auxiliary functions section allows the programmer to achieve this.

- Example:

```
/* Declarations */
%%
/* Rules */
%%
 int main()
{
yylex(); return 1;
}
```

The yy variables

- The following variables are offered by LEX to aid the programmer in designing sophisticated lexical analyzers.
- These variables are accessible in the LEX program and are automatically declared by LEX in *lex.yy.c*.
  - yyin
  - yytext
  - yyleng

yyin

- *yyin* is a variable of the type FILE* and points to the input file.
- *yyin* is defined by LEX automatically.
- If the programmer assigns an input file to *yyin* in the auxiliary functions section, then *yyin* is set to point to that file.
- Otherwise LEX assigns *yyin* to stdin(console input).

yytext

- *yytext* is of type *char** and it contains the *lexeme* currently found.
- A **lexeme** is a sequence of characters in the input stream that matches some pattern in the Rules Section.
- It is the first matching sequence in the input from the position pointed to by *yyin*.
- Each invocation of the function *yylex()* results in *yytext* carrying a pointer to the lexeme found in the input stream by *yylex()*.
- The value of yytext will be overwritten after the next *yylex()* invocation.

yyleng

- *yyleng* is a variable of the type int and it stores the length of the lexeme pointed to by *yytext*.

Methods the values to symbol table.

- installID()
  - funtion 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
- installNum()
  - similar to installID, but puts numerical constants into a separate table.

yyfunctions

yylex()

- *yylex()* is a function of return type int.
- LEX automatically defines *yylex()* in *lex.yy.c* but does not call it.
- The programmer must call yylex() in the Auxiliary functions section of the LEX program.
- LEX generates code for the definition of yylex() according to the rules specified in the Rules section.

yywrap()

- LEX declares the function yywrap() of return-type int in the file *lex.yy.c* .
- LEX does not provide any definition for yywrap().
- yylex() makes a call to yywrap() when it encounters the end of input.
- If yywrap() returns zero (indicating *false*) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin.
- If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. “wraps up”).
- If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting yyin to a new input file in yywrap() and return 0.

- As LEX does not define yywrap() in lex.yy.c file but makes a call to it under yylex(), the programmer must define it in the Auxiliary functions section or provide %option noyywrap in the declarations section.
- This options removes the call to yywrap() in the lex.yy.c file.
- Note that, it is **mandatory** to either define yywrap() or indicate the absence using the %option feature.
- If not, LEX will flag an error

```
%option noyywrap
%{
        #include <stdlib.h>
        #include <stdio.h>
%}
number [0-9]+
letter [a-zA-Z]
 %%
 {number} {printf("Found : %d\n",atoi(yytext));}
{letter} {printf("letter found:%s\n",yytext);}
 %%
int main()
{
FILE *f;
f=fopen("id.txt","r");
yyin=f;
        yylex();
        return 1;
}
```

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}+)?
%%
{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);}
…
```

```
Int installID() {/* funtion 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 numerical
    constants into a separate table */
}
```