Lexical Analysis and Lexical Analyzer Generators [Chapter 3 - Part 2]

Software used for the project
lex/flex

Lecture 9

Edited by Dr. Ismail Hababeh
German Jordanian University
Adapted from slides by Dr. Robert A. Engelen

The Lexical Analyzer Generators

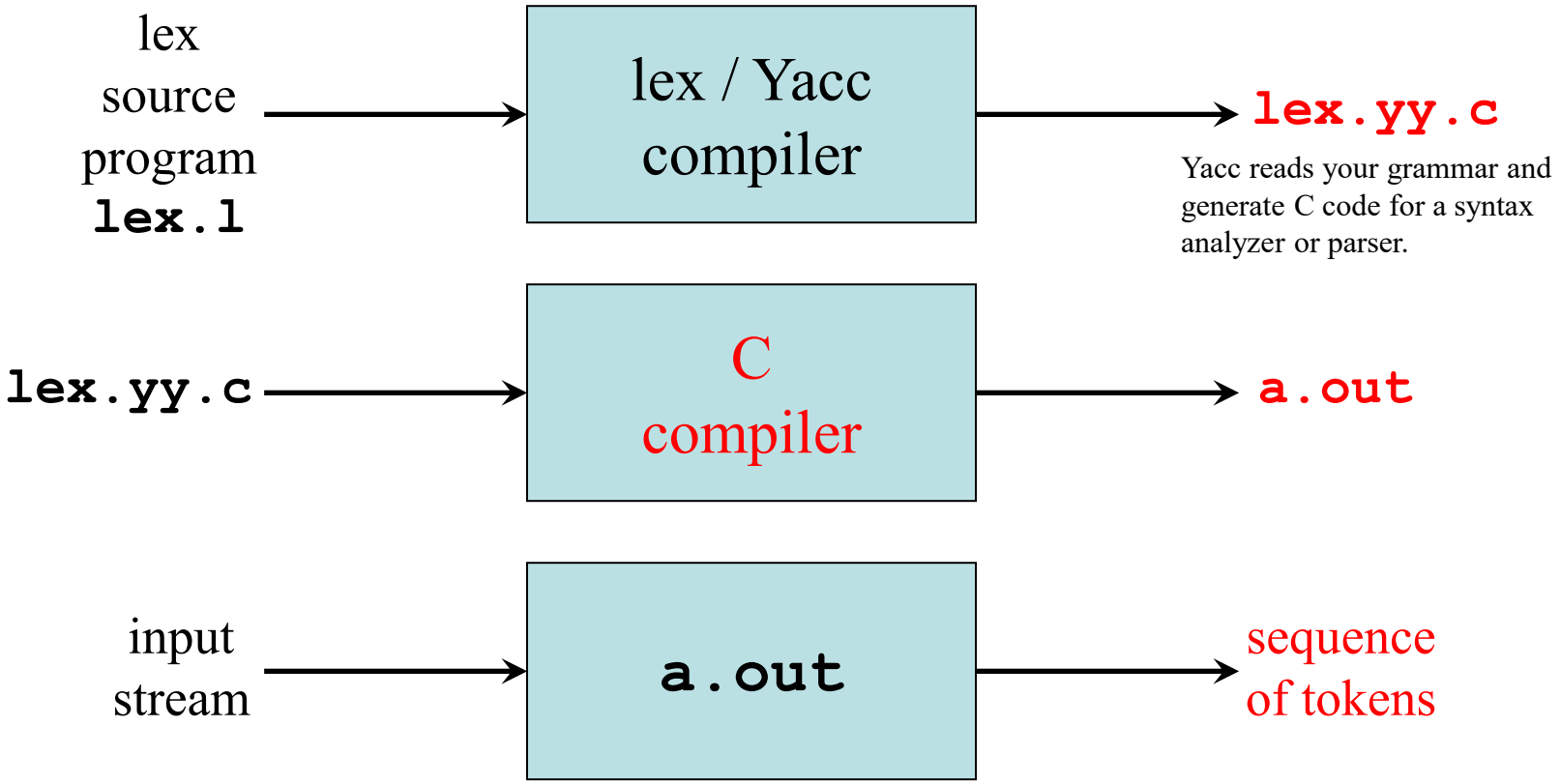
- Lexical Analyzer Generator LAG (scanner /tokenizer) is a program designed to recognize lexical patterns in text.
- LAG is used to identify specific text strings in a structured way that converts the source program into meaningful tokens.
- LAG tools: Lex and Flex.
  - Lex "lexical analyzer generator".
  - Flex "Fast lexical analyzer generator".
- Lex/Flex translates regular definitions into C source code for efficient scanning.

The Parser Generators

- Parser Generator is a program used to produce a parser for a given grammar.
- Yacc/Bison is a grammar parser that generates a parse tree from the tokens generated by Lex/Flex.
- Yacc/Bison reads your own grammar and generates C code for a syntax analyzer or parser.
- The input of Yacc/Bison is the user rules or grammar and the output is a C program.

Creating a Lexical Analyzer with Lex and Yacc

Lex and Yacc are tools used to generate lexical analyzers and parsers



Lex Program Parts

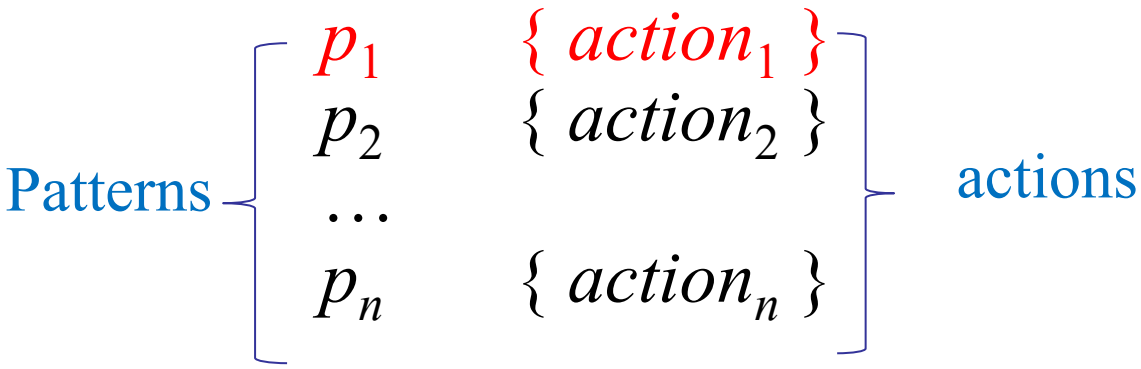
- Declarations
  - Auxiliary declarations
  - Regular definitions
- Rules
  - Patterns followed by actions to be executed
- Auxiliary Procedures

Lex Program Specification

- The *lex program parts specifications*:

```
%{  declarations  %}
 regular definitions
 %%
 translation rules
 %%
 user-defined auxiliary procedures
```

- The *translation rules* are of the form:



Regular Expressions in Lex

x	match the character **x**
\.	match the character **.**
"string"	match contents of string of characters
.	match any character except newline
^	match beginning of a line
$	match the end of a line
[xyz]	match one character **x**, **y**, or **z**
[^xyz]	match any character except **x**, **y**, and **z**
[a-z]	match one character **a** to **z**
*r**	closure (match zero or more occurrences)
r+	positive closure (match one or more occurrences)
r?	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
$r_1 \mid r_2$	match r_1 or r_2 (union)
(*r*)	grouping
$r_1 \backslash r_2$	match r_1 when followed by r_2
{*d*}	match the regular expression defined by *d*

Lex Program – Example

```
// Declarations Part
%{
#include <stdio.h>
int global_variable;
%}
// Regular definitions
number [0-9]+
opr    [-|+|*|/|^|=]
%%
// Rules
%%
// Auxiliary procedures
```

Lex Rules - Example

```
%%
{number}   { printf("number");}
{opr}      { printf("operator");}
%%
```

- LEX generates C code for the user rules and places this code into a single function called yylex().

Lex Auxiliary Procedures

```
int main ()
{
yylex();
}
```

- This part allows to add user code to the *lex.yy.c* file.
- Once the code is written, *lex.yy.c* is generated using the command: >> *lex "filename.l"*
- *lex.yy.c* is compiled using the command:
  >> *gcc lex.yy.c*

Lex Input/Output – Example 1

Based on the previous example:

Input: 12

Output: number

Input: +

Output: operator

Input: 1+2

Output: number operator number

Lex Program – Example 2

Translation rules

```
%{
#include <stdio.h>
%}

%%
CS    printf("Computer science \n");
CE    printf("Computer Engineering\n");
ECE   printf("Electrical & Comm. Eng.\n");
%%
...
```

```
..> lex school.l  //generate lex.yy.c
..> gcc lex.yy.c -ll //generate a.out
..> ./a.out //generate executable file
```

In gcc, -ll is used to tell the compiler to use the lex libraries

Lex Specification – Example 3

Translation rules

Contains the matching lexeme (pointer to the beginning of the lexme)

```
%{
#include <stdio.h>
%}

%%
[0-9]+   { printf("%s\n", yytext); }
.|\n     { }
%%
main()
{ yylex();
}
```

Invokes the lexical analyzer (returns the token)

Lex Specification – Example 5

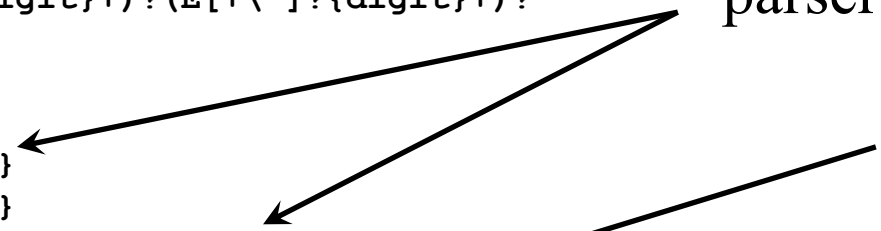
Translation rules

Regular definitions

```
%{
#include <stdio.h>
%}
digit       [0-9]
letter      [A-Za-z]
id          {letter}({letter}|{digit})*
%%
{digit}+    { printf("number: %s\n", yytext); }
{id}        { printf("ident: %s\n", yytext); }
.           { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

Lex Specification – Example 6


```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); 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;}
%%
int install_id(){/* function to install the lexeme whose 1st character
         is pointed to by yytext and whose length is yyleng, into the
         symbol table and return a pointer to it*/}
int install_num(){/* Similar to install_id(), but puts numerical
         contants into a separate table */}
```



the token attribute is stored in the shard variable yylval

yvalue stores the attribute value that is returned to the parser