, 8

COMPILER CONSTRUCTION (Lexical Analyser Generator : Lex) Fall 2019

Lecture 7, 8: July 29 & 31, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

7.1 Lexical Analyser Generator

FLEX (Fast LEXical analyser generator) is a tool for generating scanners. It was written by Mike Lesk and Eric Schmidt in 1975, for Unix and Unix like operating system. It is now open source and available on Linux and its variants. In stead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language (e.g. words like, Simple, hard, etc.), write specifications of patterns using regular expressions (e.g. DIGIT as [0-9]). Having this specifications FLEX will construct a scanner for you. FLEX is generally used in the manner described below:

1. First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file *lex.yy.c*.

2. Then, lex.yy.c is compiled and linked with the "-lfl" library to produce an executable *a.out*.

3. Finally, the program *a.out* is run that analyses its input stream and transforms it into a sequence of tokens.

Note that program Lex is not only for C compiler, but in general for all the languages. The program *.lex is in the form of: pairs of regular expressions, and C code. The file *lex.yy.c* defines a routine *yylex*() that uses the specification to recognize tokens; and *a.out* is actually the scanner! (see Fig. 7.1).

A Lex program has the following form:

declarations
%%
translation rules
%%
auxiliary functions

The declaration section comprise declarations of variables, *manifest constants* (identifiers to stand for constant, e.g., name of a token), and the regular definitions.

The *translations rules* have the form:

$$Pattern \;\; \{Action\},$$

Each pattern is regular expression that makes use of regular definitions of the declaration section. The actions are fragments of code, usually written in C. The third section of Lex holds additional functions used in the actions. These functions can also be compiled separately and may be loaded along with the lexical analyser.

The following is more specific details of Lex program format:

```
    Input file Format:
          definitions
          %%
          rules
          %%
          user code
  Definitions section:
         "name definition"
   Rules section:
         "pattern-action"
   User code section:
         "yylex() routine"
```

The lexical analyser created by **Lex** is called by the parser as a routine. Once called, it starts reading inputs, character by character, until it finds the longest prefix of the input that matches one of the patterns P_i, and then executes the corresponding actions A_i. These patterns are usually lexemes, which returns a single value, i.e., the token name to the parser. The lexical analyser may use a shared integer variable *yy1val* to pass additional information about the lexeme, if needed. If what is found by lexical analyser is whitespaces, nothing is returned to parser.

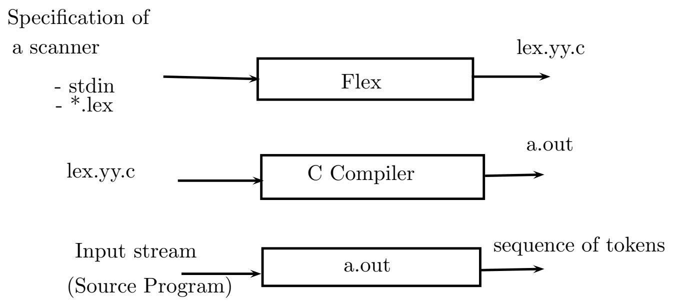

Figure 7.1: Lexical Analyser generator

Consider that "test.lex" is a Lex program for generating a lexical analyser. We compile it using lexical analyser generator (Flex) to generate the lexical analyser lex.yy.c in C language, which is compiled by *GCC* to get *a.out* executable. Following are some command sequences related to lexical analyser generation process (see Fig. 7.1).

```
$ gedit test.lex
$ flex test.lex  # outputs lex.yy.c
$ ls -l
lex.yy.c
....
$ gcc lex.yy.c -lfl ; compile lex.yy.c and link to flex library
$ ./a.out
 Input few lines here, terminate with ^d
```

Example 7.1 *A simple Lexical analyzer to implement the wc command.*

```
$ cat test.lex
/* just like Unix wc (word count command)*/

%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%
/* regular definitions */
[a-zA-Z]+   {words++; chars += strlen(yytext);}
\n          {chars++; lines++;}
.           {chars++;}
%%

/* auxiliary functions */
int main()
{
   yylex();
   printf("%d %d %d\n", lines, words, chars);
   return 0;
}
```

```
$ ./a.out < text.lex
21   33   266
```

□

7.2 Design consideration for Lex

In the declarations section there are pair of special brackets, %{ and %}. Any thing in these brackets is copied directly to the file **lex.yy.c**, and not treated as regular definition. It is common to place there the definitions of the manifest constants, using C #**define** statements to associate unique integer codes with each of the manifest constants. In this example, we have listed in a comment the names of the manifest constants, LT, GT, etc., but we have not shown them defined to be particular integers. Note that declaration section appears between %{...%}.

The regular definitions in the declarations are used for defining other definitions or used in the patterns of the translation rules surrounded by curley braces. For example *delim* is short hand for character class comprising of blank, tab, or new line. Then *ws* is defined a sone or more delimiters, i.e., *delim*+.

Note that, *id* and *number* do not stand for themselves, but for grouping of numbers, but $+, *, or?$ stand for themselves.

In the auxiliary functions section there are two functions, $installID()$ and $installNum()$. Every thing in the auxiliary section is copied directly into the file *lex.yy.c*, but may be used in the actions.

There are other things to be noted in this code. The ws is an identifier declared in the first section, has no action. The reason is that if we come across a white-space, we do not return any thing to the parser, but look for next lexeme. In the case of two letters if, with no letter or digit followed to it, the lexical analyzer returns token IF. To make sure it is if only, the scanner goes beyond letter if, and then returns back to start of lexeme next to if. Similar is case with $then$, $else$.

The next pattern for token is *id*. Note that keywords that are like *id* also match this. This is resolved as follows: *Lex* chooses which ever pattern is listed first, in situations where longest matching prefix matches two or more patterns. When *id* is matched, following action is taken:

1. Function $installID()$ is called to place the lexeme found, into the symbol table.
2. The above function returns a pointer to the symbol table, which is placed in the global variable *yylval*, which can also be used by the parser or later stages. The $installID()$ has two variables, that are set automatically by the lexical analyzer, that Lex egenrates:
   (a) *yytext* is a pointer to the beginning of the lexeme,
   (b) *yyleng* is length of lexeme found.
3. The token name ID is returned to the parser.

Similar is the case with the $installNum()$.

To resolve the conflict, the Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer longer prefix to a shorter prefix,
2. If a longest possible prefix matches to two or more patterns, prefer the pattern listed first in the Lex program.

The first rule says to continue reading letter and digits to find the longest prefix of these characters to group as identifier. It also tells us to treat ">=" as single lexeme and not two. The second rule makes keywords reserved if they are listed before *id* in the program.

The Lex automatically reads one character ahead of the last character that forms the selected lexeme. Then retracts itself so that lexeme only is consumed. Some time, we want the pattern when it is followed by a specific character, e.g., semicolon in C.

Example 7.2 *Lex Program to recognize the tokens in C language.*

```
%{
    /* definitions of constants
    LT, EQ, LE, GT, GE, NE
    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(); retun (NUMBER);}
"<"       {yylval = LT; return(RELOP);}
...

%%

int installID() {/* function to install lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yyleng. */
                 }
int installNum() {/* similar to installID, but puts numerical
                 constants into a separate table*/
                 }
```

□

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho , Monica S. Lam, et al., Sep 10, 2006

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.