Report writing on
FLEX

Badhan Biswas **(R.N: 11000120019)**
Department of Computer Science and Engineering,
Government College of Engineering and Textile Technology, Serampore
badhan@outlook.in

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

I. Introduction

Before 1975 writing a compiler was a very time-consuming process. Then Lesk (1975) and Johnson (1975) published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho-Ullman[1].

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and yacc and produces faster code. Bison produces parser from the input file provided by the user. The function *yylex()* is automatically generated by the flex when it is provided with a .l file and this *yylex()* function is expected by the parser to call to retrieve tokens from current/this token stream. The function *yylex()* is the main flex function that runs the Rule Section and extension (.l) is the extension used to save the programs.

II. Detailed Explanations

A. *Flex*

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to flex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to flex has an associated action. Typically, an action returns a token that represents the matched string for subsequent use by the parser.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Flex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

```
letter(letter|digit )*
```

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

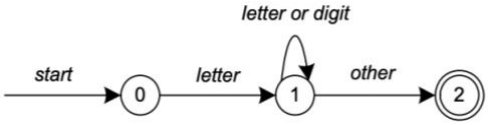

Fig. 1. Finite State Automaton

In Figure 1, state 0 is the start state and state 2 is the accepting state. As characters are read, we make a transition from one state to another. When the first letter is read, we transition to state 1.

We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit, we transition to accepting state 2. Any FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start: goto state0
state0: read c
         if c = letter goto state1
         goto state0
state1: read c
           if c = letter goto state1
           if c = digit goto state1
           goto state2
state2: accept string
```

This is the technique used by flex. Regular expressions are translated by flex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of flex's limitations. For example, flex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we

push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However, flex only has states and transitions between states. Since it has no stack, it is not well suited for parsing nested structures.

B. Practice

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class normal operators lose their meaning.

TABLE 1. PATTERN MATCHING PRIMITIVES

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 2. PATTERN MATCHING EXAMPLE

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

Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

III. APPLICATIONS

To use flex first we need to install flex on our machine. Here we are using Ubuntu22 OS. To install flex, we need to execute this command in the terminal.

```
$sudo apt-get update
$sudo apt-get install flex
```

Note: If the Update command is not run on the machine for a while, it's better to run it first so that a newer version is installed as an older version might not work with the other packages installed or may not be present now.

A. How to use

There are only three step to use flex.

1. An input file describes the lexical analyzer to be generated named *lex.l* is written in lex language. The flex compiler transforms *lex.l* to C program, in a file that is always named *lex.yy.c*.
2. The C compiler compile *lex.yy.c* file into an executable file called *a.out*.
3. The output file *a.out* take a stream of input characters and produce a stream of tokens.

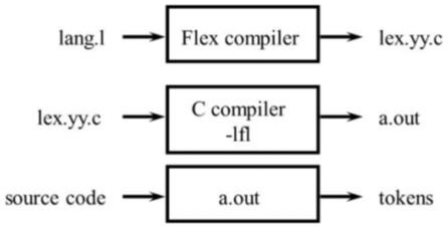

Fig. 2. Steps in flex

B. Format of the Input File

The flex input file consists of three sections, separated by a line containing only '%%'.

```
definitions
%%
rules
%%
user code
```

C. Format of the Definition Section

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section. Name definitions have the form:

```
Name definition
```

The 'name' is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non whitespace character following the name and continuing to the end of the line. The definition can subsequently be referred to using '{name}', which will expand to '(definition)'. For example,

.

```
DIGIT [0-9]
ID [a-z][a-z0-9]*
```

Defines 'DIGIT' to be a regular expression which matches a single digit, and 'ID' to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to {DIGIT}+"."{DIGIT}* is identical to ([0-9])+"."([0-9])* and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

An unindended comment (i.e., a line beginning with '/*') is copied verbatim to the output up to the next '*/'.

Any indented text or text enclosed in '%{' and '%}' is also copied verbatim to the output (with the %{ and %} symbols removed). The %{ and %} symbols must appear unindented on lines by themselves.

A %top block is similar to a '%{' ... '%}' block, except that the code in a %top block is relocated to the top of the generated file, before any flex definitions. The %top block is useful when you want certain preprocessor macros to be defined or certain files to be included before the generated code. The single characters, '{' and '}' are used to delimit the %top block, as show in the example below:

```
%top{
          /* This code goes at the "top" of the
generated file. */
         #include <stdint.h>
         #include <inttypes.h>
         }
```

Multiple %top blocks are allowed, and their order is preserved.

D. *Format of the Rule Section:*

The rules section of the flex input contains a series of rules of the form:

```
pattern  action
```

where the pattern must be unindented and the action must begin on the same line. In the rules section, any indented or %{ %} enclosed text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{ %} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for POSIX compliance.[2]

Any indented text or text enclosed in '%{' and '%}' is copied verbatim to the output(with the %{ and %} symbols removed). The %{ and %} symbols must appear unindented on lines by themselves.

TABLE 3. FLEX PREDEFINED VARIABLES

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

E. *Implementation*

It will be more clear with a example. The following example calculate the number of identifiers in an input string

```
digit           [0-9]
letter          [A-Za-z]
%{
        int count=0;
%}
%option noyywrap

%%
{letter}({letter}|{digit})*        count++;
%%

int main(void) {
    yylex();
    printf("\nNumber of identifiers= %d\n", count);
    return 0;
}
```

F. *Output in the Terminal*

If we follow the steps one by one, we will get the something like this:

```
czr@mackbookpro:-/Desktop$ flex test.l
czr@mackbookpro:-/Desktop$ clang lex.yy.c
czr@mackbookpro:-/Desktop$ ./a.out
Badhan SB BB6 nb7
Number of identifiers= 4
```

G. *Issues with Flex*

I. Time Complexity

A Flex lexical analyzer usually has time complexity in the length of the input. That is, it performs a constant number of operations for each input symbol. This constant is quite low: GCC generates 12 instructions for the DFA match loop. Note that the constant is independent of the length of the token, the length of the regular expression and the size of the DFA. However, using the REJECT macro in a scanner with the potential to match extremely long tokens can cause Flex to generate a scanner with non-linear performance. This feature is optional. In this case, the programmer has explicitly told Flex to "go back and try again" after it has already matched some input. This will cause the DFA to backtrack to find other accepted states. The REJECT feature is not enabled by default, and because of its performance implications its use is discouraged in the Flex manual.[3]

II. Reentrancy

By default, the scanner generated by Flex is not re-entrant. This can cause serious problems for programs that use the generated scanner from different threads. To overcome this issue there are options that Flex provides in order to achieve reentrancy. A detailed description of these options can be found in the Flex manual.[4]

III. Usage under non-Unix environments

Normally the generated scanner contains references to the *unistd.h* header file, which is Unix specific. To avoid generating code that includes *unistd.h*, %option *nounistd* should be used. Another issue is the call to *isatty* (a Unix library function), which can be found in the generated code. The %option never-interactive forces flex to generate code that does not use *isatty*.[5]

IV. Using flex from other languages

Flex can only generate code for C and C++. To use the scanner code generated by flex from other languages a language binding tool such as SWIG can be used.

H. Flex ++

Flex++ is a similar lexical scanner for C++ which is included as part of the flex package. The generated code does not depend on any runtime or external library except for a memory allocator (malloc or a user-supplied alternative) unless the input also depends on it. This can be useful in embedded and similar situations where traditional operating system or C runtime facilities may not be available. The flex++ generated C++ scanner includes the header file FlexLexer.h, which defines the interfaces of the two C++ generated classes.

IV. Conclusion

Flex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers. So, flex is a very important part of Compiler Design. I hope this report will be able to give a vivid idea about flex, its use cases and importances.

References

[1] Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [2006].

[2] Lesk, M. E. and E. Schmidt [1975]. Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey. A PDF version is available at ePaperPress.

[3] "Performance - Lexical Analysis With Flex, for Flex 2.5.37". Flex.sourceforge.net. Archived from the original on 2022-08-28.

[4] "Reentrant - Lexical Analysis With Flex, for Flex 2.5.37". Flex.sourceforge.net. Archived from the original on 2010-11-17.

[5] "Code-Level And API Options - Lexical Analysis With Flex, for Flex 2.5.37". Flex.sourceforge.net. Archived from the original on 2022-08-28. Retrieved 2022-08-28.