A Quick Start Introduction to Flex and Bison

Rushikesh K Joshi
Department of Computer Science and Engineering
IIT Bombay

1 Flex

1.1 What is flex

Flex is a tool that generates yylex(), given a scannar description file. yylex() is the scanner routine corresponding to this input file. The routine is contained in output file lex.yy.c. File yylex() uses a flex runtime library, which should be linked with to produce executable code from yylex().

1.2 Format of a flex file

Inside the scannar description file, regular expression patterns and the corresponding actions to be taken can be written. The following format is followed in the input file for flex.

```
%{Text to be placed into the output file
%}
List of names and their definititions
Start conditions
%%
Rules and actions
%%
User code
```

The top section includes definitions, the middle section includes the rules, and the bottom section includes the user code. The user code is optional.

1.3 Example

An example Flex input file is developed below. The program extracts words from a text file. It ignores newline characters, and also ignores anything else that is not a word composed of characters from a-z and A-Z.

The user code includes function main() which calls yylex(). The name of the file from which yylex() reads its input is taken in from the command line.

```
%{
/* code to keep track of count of words in a text file */
        #include  <iostream>
        #include  <string>
        using namespace std;
        int count=0;
                int pick (char *text) {
                        count ++; cout << text << endl;
                }
%}
ID      [a-zA-Z][A-Za-z]*
%%
{ID} { pick(yytext);}
[\n] {}
.    {}
%%
#include<stdio.h>
int main (int argc, char *argv[]) {
yyin=fopen(argv[1],"r");

 yylex();

cout << "total " << count << " words found\n";
}
```

The definitions block in this example contains a name definition for name ID, and also text block which will be copied verbatim into the output file.

The text block is being used to define a function. This function uses the string library. The corresponding include and using statements have been placed into the top block. All this code will be part of the output file. As shown, comments can also be added to this code.

The name definition gets used in the Flex file in the rules. The code that we defined in the top block and text block is also for use in the rules.

Start conditions activate the rules. Start conditions are explained later.

1.4 Rules and Actions

The following patterns are used to express the rules.

Pattern	Meaning
x	character
.	any char except newline
[*abc*]	any of a,b,c
[*a* − *zA* − *Z*]	any from range a-z or from range A-Z
[^a-z]	any char except a to z
[^12a\n]	any char except 1, 2, a, and newline
r*	0 or more of r, r is a regexp
r+	1 or more of regexp r
r?	0 or 1 r, which is r is optional
r{2,5}	anywhere between 2 or 5 repetitions of r

r{2,}	2 or more repetitions of r
r{2}	2 repetitions of r
{name}	regexp defined by 'name' in definitions section
"[xyz] \"p"	literal string [xyz]"foo
\ 0	ASCII null char 0
\xff	hexadecimal value FF
\71	octal value 71
(r)	regexp r
(?$o_1 - o_2$:r)	apply option o_1 and omit option o_2 from pattern r. The options: i or -i (case sensitivity or not), s or -s (to include or not to include ' n' in .) x (ignore comments and whitespaces in r– see manual for more details)
(?# comment)	omit everything in (), the first) ends the pattern, it can be composed of many lines
rs	concatenation
r|s	either r or s
r/s	match r if next exp is s. s is returned to input (trailing context)
^r	r only if it is in the beginning of newline
r$	r only if it is at the end of line
$< s > r$	r with start condition s
$< s1, s2, s3 > r$	r with any of start conditions s1, s2, s3
$< * > r\ n$	r in any start condition
$<< EOF >>$	end of file
$< s1, s2 >< < EOF >>$	when it is eof and start condition is either s1 or s2

[e]	where e can use a character class expression a character class expression 'e' can be [:alnum:] [:alpha:] [:blank:] [:cntrol:] [:digit] [:graph:] [:lower:] [:print] [:punct:] [:space:] [:upper:] [:xdigit:] e.g. [[:alphanum:]] and [a-zA-Z[:digit:]] are equivalent

1.5 Actions

A rule pattern ends at the first non-escaped white space. From here, the action can continue till the end of that line. Actions can be included in { } if they span more lines. Following are special directives which can be included in actions. For more directives, refer to manual.

ECHO;	copies yytext to output
BEGIN	used in start conditions as explained above
REJECT;	this rule is skipped and the scanner proceeds to the second best matching rule
yymore();	the next value of yytext is appended to the current value of yytext
yyless(n);	keep only first n characters and return the rest back into input stream and will be scanned next
unput(c);	puts character c back into the input stream, and will be the next char scanned
input(); or yyinput();	in case of c++: returns the next characters from the input stream
yyterminate();	terminates the scanner and returns 0 to scanner's caller

yyrestart(FILE *); scans from this new file by changing *yyin*.
int yywrap(); called at the end of input. returns 0 if scanning must continue with a different input file, which is set inside yywrap, else returns non-zero (default) to terminate scanning.

Values accessible:
yytext: char * pointing to matched token, with size #define YYLMAX.
yyout: scanner writes ECHOs on this
yyin: input file
YYSTART or YYSTATE: current value of start condition

1.6 Start conditions (An advanced feature)

A start condition is specified as inclusive or exclusive start condition. For example, the following code defines condition *state*1 as inclusive, while condition *statex* as exclusive start condition.

%s state1
%x statex

Rules can be prefixed with start conditons as follows:

<state1>pattern action

In the above rule, the pattern is matched only when the scannar is in a start condition called *state*1.

Start conditions can be activated by a $BEGIN(< condition >)$ statement in the action blocks as given below:

BEGIN (state1);

When a new exclusive condition begins, all old active condition are deactivated, including even the rules that have no start conditions. When an inclusive start condition begins, all old start active start conditions are deactiviated except the rules that have no start condition.

1.7 Generated scanner

The output file: **lex.yy.c**
contains: **int yylex () { .... }**

- whenever yylex() is called, it scans tokens from **yyin** and it continues till eof or a return executed from an action. The default yyin is set to stdin. yylex() returns 0 when it reaches end of file. It also returns with a 0 when one of the actions make a call to return.
- The generated scanner picks the longest match when multiple strings can be matched. The ties are broken by picking the first matching rule.

- The matched text is available as **char * yytext**. The length of this char string is stored in **int yyleng**. But alternative character array representation is also available, which can be specified with directive **%array**. However the default **%pointer** is faster.
- If no match is found the default rule is executed. By default, any text not matched gets copied to the output.
- e.g. empty flex file (just the %%) copies all input to output.
- The scannar uses block reads from the input file, but it can be changed.

1.8 Interface with Yacc

- yacc generated parser expects a call to yylex() to find the next input token.
- **yacc -d** generates **y.tab.h** containing definitions of all **%tokens** appearing in yacc input file. This file is included in flex scannar. The token numbers from this included file can be used for returning them from flex actions.
- **yywrap()** This function is called by the scannar when it sees end of file. If yywrap() returns 0, it indicates that inside yywrap() code, yyin has been set to another valid input file and scanning should continue. Compiling flex input file with *-lfl* option brings in a default implementation of yywrap() which returns 1. Another way is to provide directive **%option nowrap** to indicate that no implementation of yywrap is being supplied by the user, and default should be used automatically without having to give the *-lfl* option.

2 Bison

2.1 What is Bison

Bison is a parser generator. It takes input as a grammar with actions and generates parser source code that implements a parser for the grammar specified. The grammar is specified in terms of tokens which can come from a lexical anayzer *yylex()* program generated using flex. Bison generates function **yyparse()** in file **filename.tab.h** for input grammar file *file.y*. Additional function **yyerror()** to handle errors needs to be specified. Other additional functions *yylex()* and *main()* specified in the grammar file are also copied in the output file filename.tab.h.

2.2 Format of input file to Bison

```
%{ definitions of types and variables that are used in actions
preprocessor macro definitions, header include declarations
declaration of yylex and yyerror
%} Declarations
%%
Grammar Rules and actions
%%
User code
```