



UNIVERSIDAD PANAMERICANA

Compilers
COMPUTER GRAPHICS AND SYSTEMS ENGINEERING

The Lexical-Analyzer Generator **Lex**

semester

2022-2

Students:

(0197495@up.edu.mx) Mariana Ávalos Arce
(0219634@up.edu.mx) Karen Leticia Bocardo Aranda

Professor:

Blanca Estela Villalvazo Flores

wednesday, march 30th, 2022
Guadalajara, Jalisco

1 Introduction

We present a tool called **Lex** or **Flex** (recent implementation), that allows a person to specify a **lexical analyzer** by specifying **regular expressions** to describe patterns for tokens. The input notation for the Lex tool is called the *Lex language* and the tool is the *Lex compiler*.

The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c` that simulates this transition diagram. The mechanics is basically a translation of regular expressions into transition diagrams.

First, we need to provide an input file, say `lex.l`, written in the *Lex language* and describing the lexical analyzer to be generated. The Lex compiler transforms the file `lex.l` into a C program, in a file that is always named `lex.yy.c` and that needs to be compiled by the C compiler into a file called `a.out`, as any object file in the C language.

Then, the file `a.out` compiled is used as a subroutine of the parser: it is a function in C that returns an integer, which is a code for one of the possible token names. The attribute value (a numeric code, a pointer to the symbol table or nothing) is stored in a global variable `yylval`. This variable is shared between the lexical and syntax analyzer (parser), so that it is easier to return both the name and the attribute value of a token.

1.1 Structure of a Lex Program

A Lex program has the form:

```
%{  
to copy into lex.yy.c  
%}  
declarations  
%%  
translation rules  
%%  
auxiliary functions
```

1.2 Declarations

This section in Lex includes declarations of variables, manifest constants (identifiers declared to stand for a constant, i.e., the name of a token) and regular definitions. This section is framed by the special brackets `%{` and `%}`, and anything within these brackets is copied directly to the file `lex.yy.c` and it's not treated as a regular definition. It is common to place there the definitions of the manifest constants, using C `#define` statements to associate unique integers with each of the manifest constants.

After these, follow the regular definitions: these use the extended notation for regular expressions. Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Example:

```
delim  [ \t\n]  
ws     {delim}+
```

Parentheses are used as grouping metasympols and do not stand for themselves. If we wish to use one of Lex metasympols to stand for itself, we need to add a backslash, i.e., for the definition of **number** we would use `\.` to represent the dot, since the dot means otherwise *any character*.

1.3 Translation Rules

The translation rules each have the form:

```
pattern { action }
```

Where each pattern is a regular expression, which may use the regular definitions of the above section. The actions are fragments of code, typically written in C.

1.4 Auxiliary Functions

This section holds whatever additional functions are used in the actions. This functions can be compiled separately and loaded with the lexical analyzer. Everything in this section is copied as it is directly to `lex.yy.c` file, but may be used in the actions.

1.5 Lexical Analyzer & The Parser

The lexical analyzer created by `Lex` and the parser work together in the following way: when the parser calls the analyzer, the latter reads its remaining input char by char, until it finds the longest prefix of the input that matches one of the *patterns* P_i . It then executes its associated action A_i , and this action usually returns to the parser again. In the case where it does not go back to the parser (when P_i describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of its actions causes a return to the parser. The lexical analyzer returns a single value, which is the **token name**, to the parser, but uses the shared variable `yyval` to pass additional information about the lexeme found.

2 Lex Analyzers in the Market

2.1 WinFlexBison

WinFlexBison is a Windows port of Flex (the fast lexical analyser) and GNU Bison (parser generator). Both `win_flex` and `win_bison` are based on upstream sources but depend on system libraries only.

- Operating System: Windows 10
- Author(s): Alex Zhondin
- License Type: Free (GNU)
- Creation: 2018
- Update: Jan 20, 2022
- Source: <https://github.com/lexxmark/winflexbison>

2.2 flex

`flex` is a tool for generating scanners: programs which recognize lexical patterns in text.

- Operating System: Linux Distributions
- Author(s): Will Estes, John Millaway, Aaron Stone, Vern Paxson, Jef Poskanzer and Kevin Gong.
- License Type: Free (GNU)
- Creation: 2001

- Update: Jan 3, 2020
- Source: <https://github.com/westes/flex>

2.3 Quex

The goal of this project is to provide a generator for lexical analyzers of maximum computational efficiency and maximum range of applications. This includes the support for Unicode (UTF8, UTF16, ...) and a large variety of other encodings directly and via nested converters such as ICU(tm) and IConv.

- Operating System: Windows, Mac and Solaris
- Author(s): Frank-Rene Schaefer
- License Type: Free (MIT)
- Creation: 2009
- Update: May 3rd, 2021
- Source: <https://github.com/westes/flex>

3 Our Option

We choose **flex** since it comes already installed on every UNIX or Linux Distribution, and the use of a terminal program is very user friendly according to our perception. We both have Linux systems on our computers, so we did not have to install it and this was an advantage.

3.1 Download flex

In theory, every Linux distribution has it, but if it doesn't appear, you can install it in the following way. The example is done using Ubuntu 20.04 or macOS.

```
sudo apt-get install flex
```

This command will download and install the following files and their corresponding space required:

- flex: 317 kB
- libfl-dev: 6,316 Bytes
- libsigsegv2: 13.9 kB
- m4: 199 kB

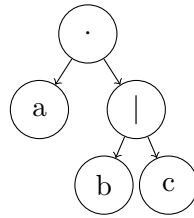
The official website is: <https://howtoinstall.co/es/flex> where you can check the installation guide.

4 Lexical Analyzer With Flex

4.1 Example 1

For this example we will create an analyzer for **all the strings for a followed by b or c**. Whenever it finds a string of this language, the program will print that it has found such string, for anything else, it will print besides the string that it is not a part of the language.

4.1.1 Syntax Tree



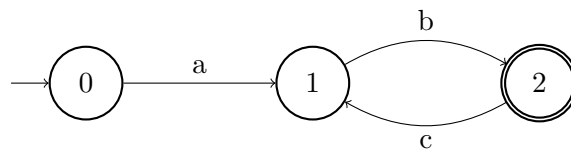
4.1.2 Automata (DFA)

$$S = \{0, 1, 2\}$$

$$\Sigma = \{a, b, c\}$$

$$S_0 = \{0\}$$

$$F = \{2\}$$



4.1.3 Transitions table

	a	b	c
0	{1}		
1		{2}	{2}
2			

4.1.4 Implementation in Lex

The following code in Lex implements the Example 1:

```
/** Definition Section has one variable
which can be accessed inside yylex()
and main() */
%{
int count = 0;
%}

%%
a(b|c) {printf("%s string that has a followed by b or c.\n", yytext);
count++;}
. {printf("%s not a string generated by this language.\n", yytext);}
\n {return 0;}
%%

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

```
printf("\nnumber of strings generated by the language "
      "in the given input - %d\n", count);

return 0;
}
```

And the testing and output looks like in Figure 1. The first step is to compile the Lex file, then its output in C must be also compiled by gcc or similar. Finally, run the object program and input a string, such as \$abacabb acc.

```
karenbocardo@macbook Desktop % ./a.out
abacabb acc
ab string that has a followed by b or c.
ac string that has a followed by b or c.
ab string that has a followed by b or c.
b not a string generated by this language.
not a string generated by this language.
ac string that has a followed by b or c.
c not a string generated by this language.

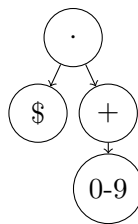
number of strings generated by the language in the given input - 4
karenbocardo@macbook Desktop %
```

Figure 1: Testing of Example 1.

4.2 Example 2

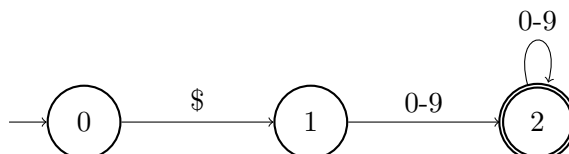
For this example we will create an analyzer for **all the strings of \$ followed by one digit**. Whenever it finds a string of this language, the program will print that it has found such string, for anything else, it will print besides the string that it is not a part of the language.

4.2.1 Syntax Tree



4.2.2 Automata (DFA)

$$\begin{aligned}
 S &= \{0, 1, 2\} \\
 \Sigma &= \{\$, 0-9\} \\
 S_0 &= \{0\} \\
 F &= \{2\}
 \end{aligned}$$



4.2.3 Transitions table

	\$	digit
0	{1}	
1		{2}
2		{2}

4.2.4 Implementation in Lex

The following code in Lex implements the Example 2:

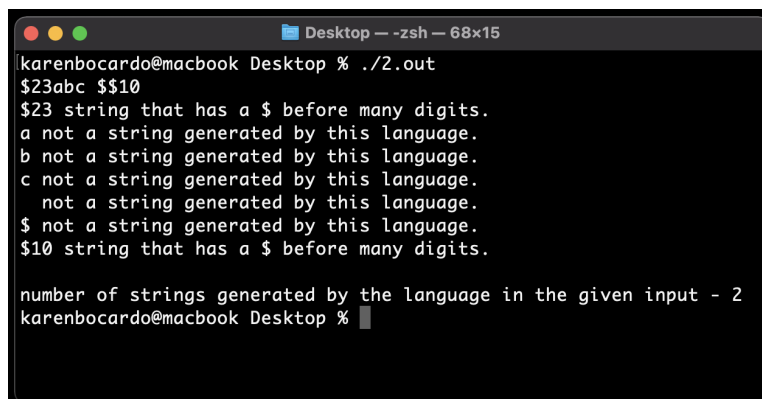
```
/** Definition Section has one variable
which can be accessed inside yylex()
and main() */
%{
int count = 0;
%}
%%
\[0-9]+\ {printf("%s string that has a $ before many digits.\n", yytext);
count++;}
. {printf("%s not a string generated by this language.\n", yytext);}
\n {return 0;}
%%

int yywrap(){
int main(){

yylex();
printf("\nnumber of strings generated by the language "
"in the given input - %d\n", count);

return 0;
}
```

And the testing and output looks like in Figure 2. The first step is to compile the Lex file, then its output in C must be also compiled by gcc or similar. Finally, run the object program and input a string, such as \$23abc \$\$10.



```
Desktop --zsh-- 68x15
karenbocardo@macbook Desktop % ./2.out
$23abc $$10
$23 string that has a $ before many digits.
a not a string generated by this language.
b not a string generated by this language.
c not a string generated by this language.
 not a string generated by this language.
$ not a string generated by this language.
$10 string that has a $ before many digits.

number of strings generated by the language in the given input - 2
karenbocardo@macbook Desktop %
```

Figure 2: Testing of Example 2.

5 Conclusions

Lex generator is then a very useful tool to make the coding of a lexical analyzer easier, since the rules and syntax of the Lex program requires a simple structure and relatively small amount of code, in comparison to making that logic from scratch. This could be used in a compiler, where the lexical analyzer can use this compiled module to recognize strings that belong to the language and strings that don't.

References

- [1] Firas Dib. *regex101: build, test, and debug regex*. URL: <https://regex101.com/r/c08lqs/9> (visited on 03/23/2022).
- [2] Jonny Fox. *Regex tutorial — A quick cheatsheet by examples - Factory Mind*. Dec. 21, 2021. URL: <https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285> (visited on 03/23/2022).
- [3] GeeksforGeeks. *Flex (Fast Lexical Analyzer Generator)*. Nov. 16, 2021. URL: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/> (visited on 03/23/2022).
- [4] M. Lesk. *Lex - A Lexical Analyzer Generator*. URL: <http://dinosaur.compilertools.net/lex/index.html>.
- [5] Lex software search. *Lex - A Lexical Analyzer Generator*. URL: <https://sourceforge.net/directory/os:windows/?q=lex+-+a+lexical+analyzer+generator>.
- [6] Skinner. *RegExr: Learn, Build, Test RegEx*. 2019. URL: <https://regexr.com> (visited on 03/23/2022).
- [7] Trzesniewski. *Create Syntax tree from given Regular Expressions (For RE to DFA)*. Nov. 15, 2014. URL: <https://stackoverflow.com/questions/26948724/create-syntax-tree-from-given-regular-expressions-for-re-to-dfa> (visited on 03/23/2022).