About Lex/Flex

And little bit on Yacc/Bison

Vikrant Gajria[1]

[1]BE Computer Engineering, DJSCE

24 March 2021

Lexical Analysis

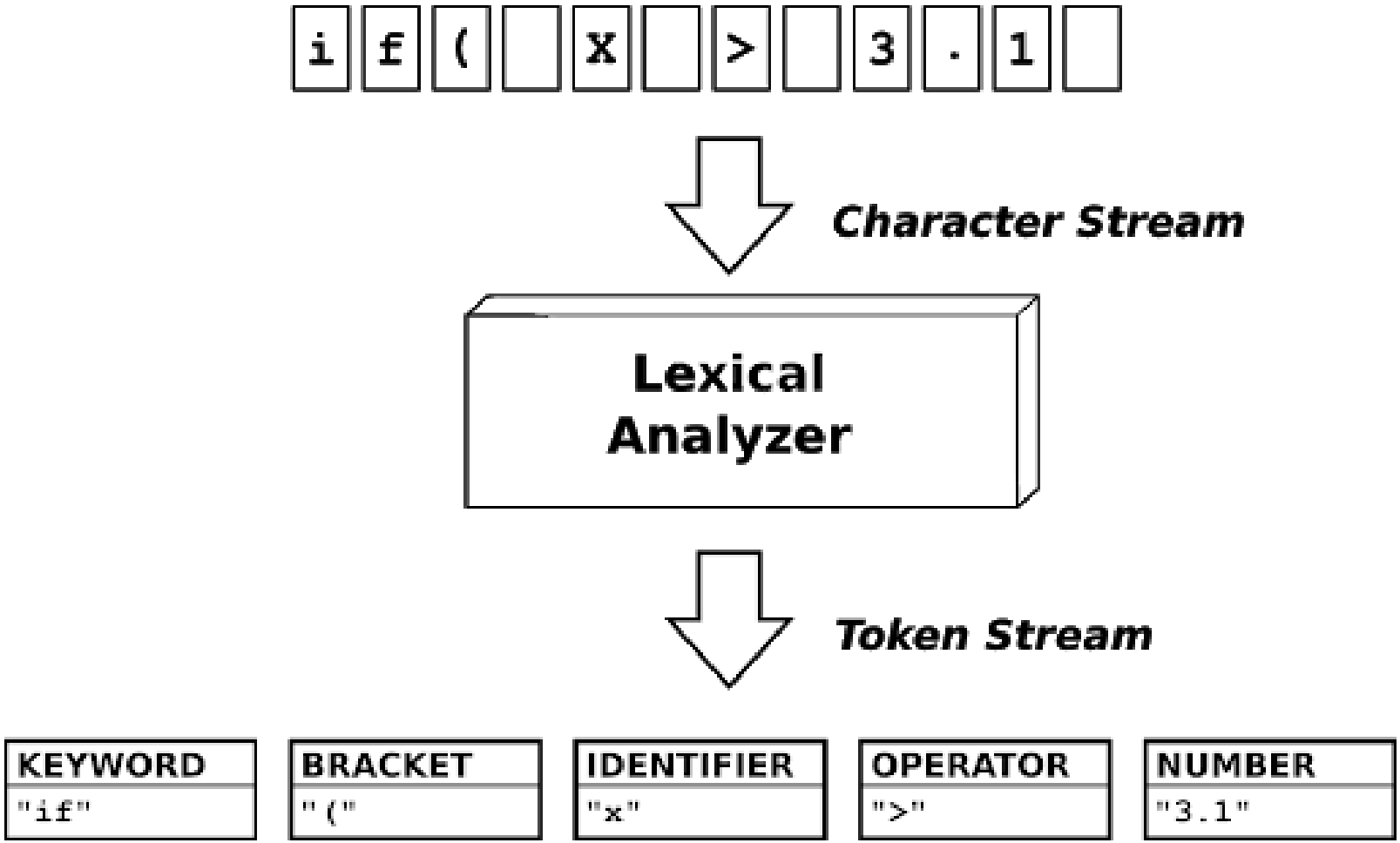

Figure: Lexer operation

Lexical Analysis

$$< token, value >$$

- token = unique integer value, e.g. ID = 1, INT = 2
- value (optional) = data related to token, e.g. "x", 5

Hence, we have $< ID,$ "x" $>, < INT, 5 >$

Lexical Analysis

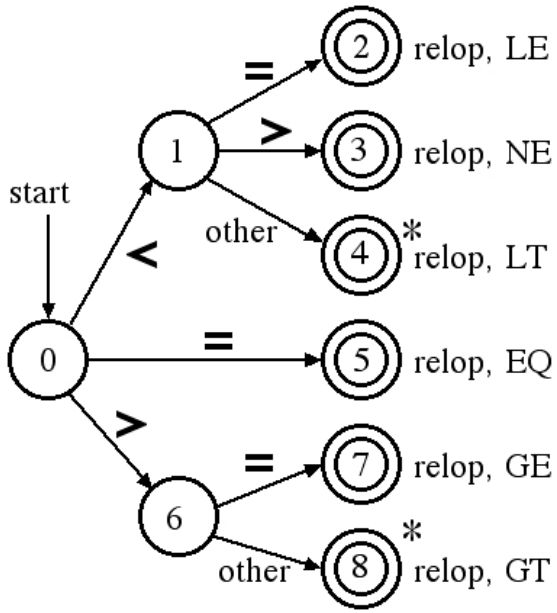

Figure: Lexical Analysis as a DFA

Problem

Writing so much logic is very tedious and unmaintainable in code!

Lex/ Flex

- Computer program that generates lexical analyzers (also known as "scanners"" or "lexers")
- Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex

It is not a framework or a library, it writes the code for you

Why Lex

- Write rules, not code
- Regular Expression "Patterns" for DFA building
- Streaming lexer, does not load entire files all at once
- Hence, it is extremely fast
- Written for C orignally but can be used for C++
- Reimplemented for other languages like Rust, Go, Python...

Generating code


```
%%

[0-9]           printf("Digit: %s \n", yytext);
\n              printf("New line \n");
.               printf("Any: %s \n", yytext);
```

These 4 lines generate 1734 lines of C code!

%% Sections

The %% is important because it marks the start of Rules sections

Example 1

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

Figure: Regex pattern primitives

yytext

yytext is a global variable that stores the text of the lexeme matched by regex.

Using flex

Generate code using:

```
flex <filename>
gcc -lfl lex.yy.c -o <outputname>
```

Run interactive mode:

```
./<outputname>
```

Pass in a file's text:

```
cat <file> | ./<outputname>
```

Example 2


```
DIGIT     [0-9]
ID        [a-zA-Z][a-zA-Z0-9]*

%%

[\t ]              { /* Perform no action */ }
{DIGIT}+           printf("Digit: %s \n", yytext);
{ID}               printf("Identifier: %s \n", yytext);
\n                 printf("New line \n");
```

%% Sections

The section above %% is Declarations section with C code, declarations (DIGIT, ID), and other configurations.

Example 3


```
... declarations

%%
... rules

%%

int main(argc, argv)
int argc;
char **argv;
{
    ++argv, --argc;
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
}
```

%% Sections

An optional user code section can be added after the rules. This section is copied verbatim, i.e. directly into the generated code.

Example 3

yyin

File to be processed. Set it to a readable file pointer. Default is stdin i.e. command line input.

yylex

Continuously lexically analyse a file. If you return something, it will pause execution and start again if you call it again. You can change the return type by modifying YYDECL macro (advanced).

Syntax


```
[
%{
... user code for header files and other config
%}
]

... declarations, if any

%%
... rules in tabular form, regex and action

[
%%
... user code for anything after the rules, like main
]
```

Where [*block*] means optional

Standard structure

This structure is used in all Lex implementations. Yacc and its implementations also uses same structure!

yyVariables

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

Figure: Predefined global variables

Example 4

Assembly-like calculator

```
ADD 5;
PRINT;

* 4;
PRINT;
/ -2;
ADD 7;
PRINT;

sub 13;
print;

clear;
print;
EXIT;
```

Yacc - Parser Generator

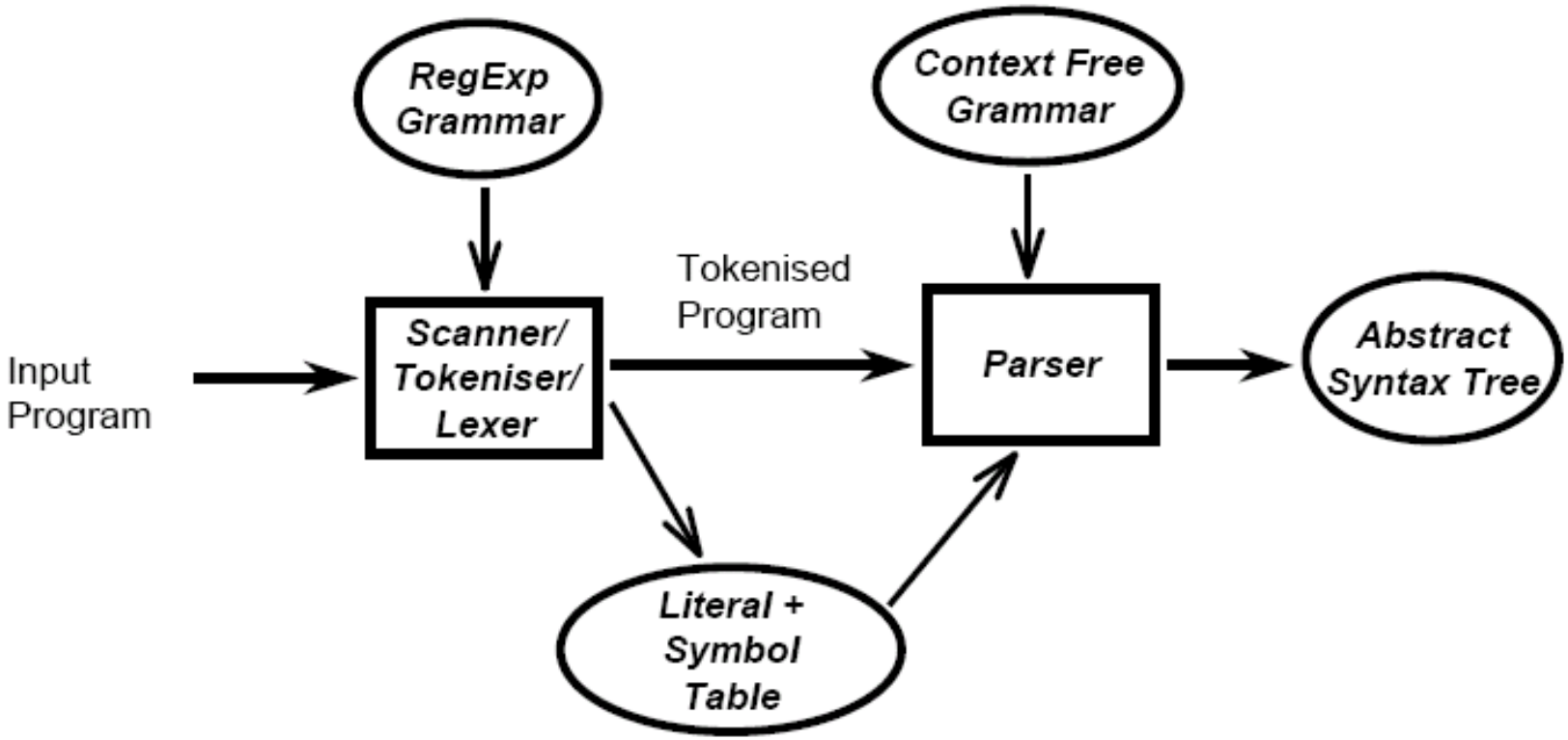

Figure: Lexer to Parser pipeline

Integrating Yacc/Bison with Lex/Flex

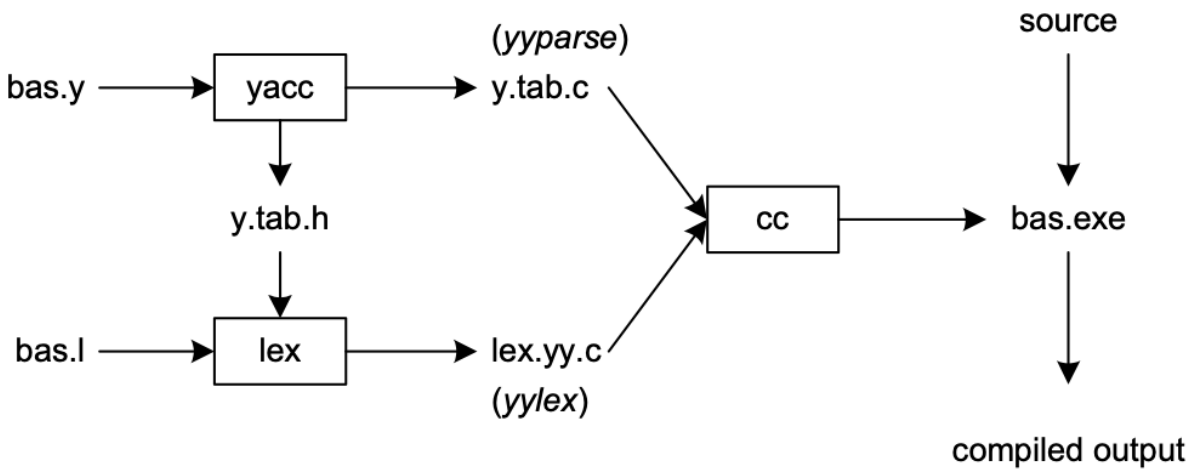

Figure: Steps to compile Lex and Yacc/ Flex and Bison

yylval

yylval, defined in *.tab.c*, is used to store data related to tokens (value from $< token, value >$). It's datatype is defined using YYSTYPE macro and is usually defined using Yacc/ Bison.

Example 5

Calculator with infix syntax

```
1 + 2
3.0 / 2
2 * (3.1427 / 3)
1.1 + 2 - 3 * 4 / 5.
```

Is based on

```
E -> E + E | E - E | T
T -> T * F | T / F | F
F -> ( E ) | num | id
```

Code is generated using Yacc/ Bison

Experiment yourself

- CSV parser using Flex
- HTML/ XML parser using Flex
- JSON/ YAML/ TOML/ CFG/ INI parser using Flex or Flex+Bison
- Assembly-like stack machine using Flex or Flex+Bison
- Pascal/ TCL/ C language parser using Flex+Bison
- Use C++ and multiple files, explore the option flags

Tom Niemann. Lex yacc tutorial. URL https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf.

John Millaway Vern Paxson, Will Estes. Lexical analysis with flex. 2012. URL https://www.iith.ac.in/~ramakrishna/Compilers-Aug14/doc/flex.pdf.

Keith Schwarz Julie Zelenski. flex in a nutshell. 2012. URL https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf.

Lan Gao. Flex tutorial. URL http://alumni.cs.ucr.edu/~lgao/teaching/flex.html.

Flex. The flex manual page. 2021. URL http://dinosaur.compilertools.net/flex/manpage.html.