# Syntax Analysis

DFA, Flex, Bison, ANTLR

YEGOR BUGAYENKO

Lecture #2 out of 10 90 minutes

All videos are in this YouTube playlist.

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as websites. Copyright belongs to their respected authors.

Extended Backus-Naur Form

Lexical Analysis

Syntactic Analysis

Chapter #1:

Extended Backus-Naur Form

In 1959, John Backus proposed a metalanguage to describe the syntax of IAL, known today as ALGOL 58. Further development of ALGOL led to ALGOL 60. In the committee's 1963 report, Peter Naur called Backus's notation Backus normal form. Donald Knuth argued that BNF should rather be read as Backus–Naur form, as it is "not a normal form in the conventional sense."

EBNF is now the way to specify formal grammars. Proposed by Niklaus Wirth in 1977 as an alternative to BNF.

The International Organization for Standardization adopted an EBNF Standard, ISO/IEC 14977, in 1996. However, there are many notations of EBNF.

```
This first published version looked like: <number> ::= <digit> |<number> <digit> <digit> ::= 0 |1 |2 |3 |4 |5 |6 |7 |8 |9
```

#### Our language EBNF:

```
program = line { eol line };
line = number command arguments;
arguments = [ argument { "," argument } ];
number = digit { digit };
digit = "0" | "1" | "2" ... | "9";
eol = "\n";
```

Chapter #2:

Lexical Analysis

Lexical analyzer (<u>lexer</u> or <u>scanner</u>) takes input language and produces tokens (which then can be parsed into parse tree by a parser).

Lexemes are said to be a sequence of characters in a token.

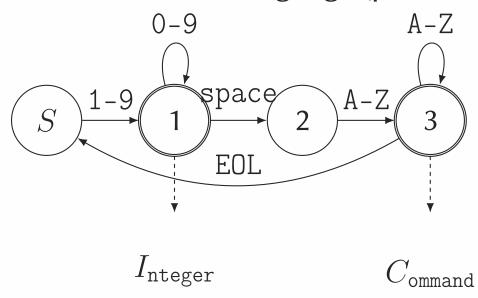
Lexers are implemented as DFAs, which use regular expressions. For example, this is the language:

```
1 10 PRINT
```

<sup>20</sup> RENDER

<sup>30</sup> EXIT

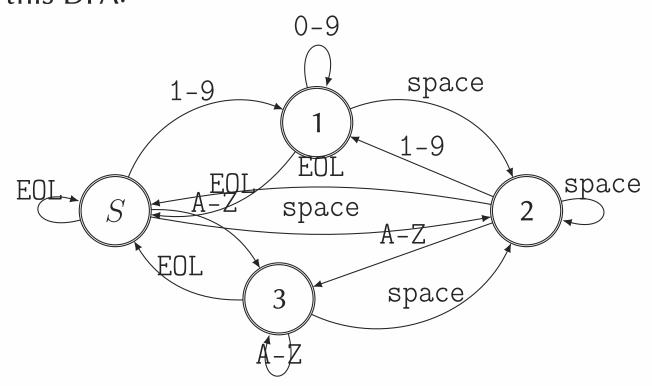
The DFA for this language (pattern matching rules on the edges):



The stream of tokens (with lexems inside them) produced:

```
Integer("10"), Command("PRINT"), Integer("20"), Command("RENDER"), Integer("30"), Command("EXIT").
```

Some lexems (like spaces or EOLs) are ignored and do not become tokens. They are <u>non-token</u> elements. However, they could become tokens, like in this DFA:



Q: What would be the stream of tokens for "10 LINPUT n LULL 20"?

Q: How many tokens in this C-language program?:

Chapter #3:

Syntactic Analysis

While <u>lemmatization</u> focuses purely on feature extraction and data cleaning, <u>syntactic analysis</u> analyzes the relationship between words and the grammatical structure of sentences.

## Top-Down and Bottom-Up Parsing

Top-down parsing builds the parse tree from the top (start symbol) down; most top-down methods are LL. Bottom-up parsing builds the parse tree from the leaves (terminal symbols) up; most methods are LR.

LL means Left-to-right + Leftmost derivation.

LR means Left-to-right + Rightmost derivation.

## Predictive Parsing

A <u>recursive descent parser</u> is the one the checks every rule before making a decision which one is right.

Predictive parsing is possible only for the class of LL(k) grammars, which are the CFGs for which there exists some positive integer k that allows a parser to decide which production rule to use by examining only the next k tokens of input.

### Flex and Bison

These tools are called <u>compiler-compilers</u> (originally lex and yacc) or parser generators.

Make this simple Flex program in foo.x:

```
%option noyywrap
DIGIT [0-9]

LETTER [a-z]

%%

{DIGIT}+ { printf( "int: %s\n",yytext ); }

{LETTER}+ { printf( "word: %s\n",yytext ); }

%%

int main(int argc,char** argv) { yylex(); }
```

[ LL/LR Predictive  $\underline{CC}$  ANTLR ]

Then, compile it with Flex and then with Gcc:

```
1 $ flex foo.x
2 $ gcc lex.yy.c
3 $ ./a.out
```

Each time the program needs a token, it calls yylex(), which reads a little input and returns the token. When it needs another token, it calls yylex() again. The scanner acts as a <u>coroutine</u>; that is, each time it returns, it remembers where it was, and on the next call it picks up where it left off.

The <u>action code</u> is what stays in the brackets after the <u>pattern</u>. If action code returns, scanning resumes on the next call to yylex(); if it doesn't return, scanning resumes immediately.

Lexical errors may be <u>handled</u> and the lexer may <u>recover</u> from some of them: we don't want the lexer to stop at the first error. See how Flex recovers.

#### This is Bison code in foo.y:

```
1 %token WORD
2 %token INT
 input: date | sentence;
 date:
    INT INT INT
   { printf("date!\n"); };;
 sentence:
   sentence WORD
   { printf("sentence!\n"); };
12 %%
int main(int argc, char** argv) {
```

```
14  yyparse();
15 }
16 void yyerror(char *s) {
17  fprintf(stderr, "error: %s\n",s);
18 }
```

We compile them together as such:

```
bison -d foo.y

lex foo.x

gcc foo.tab.c lex.yy.c
```

Bison generates foo.tab.h file, which we must #include into foo.x.



ANTLR breaks the stream into tokens (capitalized names) and non-terminals:

```
grammar basic;
program: line+;
line: order command tail;
order: INTEGER;
command: NAME;
tail: argument*;
INTEGER: [1-9][0-9]*;
NAME: [A-Z]+;
SPACE: (' ')+ { skip(); };
```