Compiler 1

Lecture 6: Lexical Analyzer and Syntax analysis

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Lexical Analyzer: How It Works

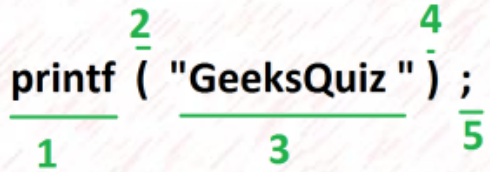
1. Tokenization i.e. Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4.provides help in generating error messages by providing row numbers and column numbers.

For example, consider the program

```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
 }
```

All the valid tokens are: 'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';' 'a' '=' '10' ';' 'return' '0' ';' '}'

As another example, consider below printf statement.



There are 5 valid token in this printf statement.

- Count number of tokens :

```
int main()
{
  int a = 10, b = 20;
  printf("sum is :%d",a+b);
  return 0;
}
```

Answer: Total number of token: 27.

The Role of the Lexical Analyzer

1- The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.

2- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

3- There are suggested interactions between the lexical analyzer and the symbol table. When, the parser call the lexical analyzer to suggest by the get Next Token command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

4- One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

5-Another task is correlating error messages generated by the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

When the token pattern does not match the prefix of the remaining input, the lexical analyzer gets stuck and has to recover from this state to analyze the remaining input. In simple words, a lexical error occurs when a sequence of characters does not match the pattern of any token. It typically happens during the execution of a program.

Types of Lexical Error: Types of lexical error that can occur in a lexical analyzer are as follows:

1. Exceeding length of identifier or numeric constants.

Example:

```
#include <iostream>
using namespace std;
int main() {
        int a=2147483647 +1;
        return 0;
}
```

This is a lexical error since signed integer lies between −2,147,483,648 and 2,147,483,647

2. Appearance of illegal characters

```
#include <iostream>
using namespace std;
int main() {
        printf("Geeksforgeeks");$
        return 0;
}
```

This is a lexical error since an illegal character $ appears at the end of the statement.

3. Unmatched string

```
#include <iostream>
using namespace std;
int main() {
/* comment
        cout<<"GFG!";       return 0; }
```

This is a lexical error since the ending of comment “*/” is not present but the beginning is present.

4. Spelling Error

```
#include <iostream>
using namespace std;
int main() {            int 3num= 1234;     return 0; }
```

spelling error as identifier cannot start with a number

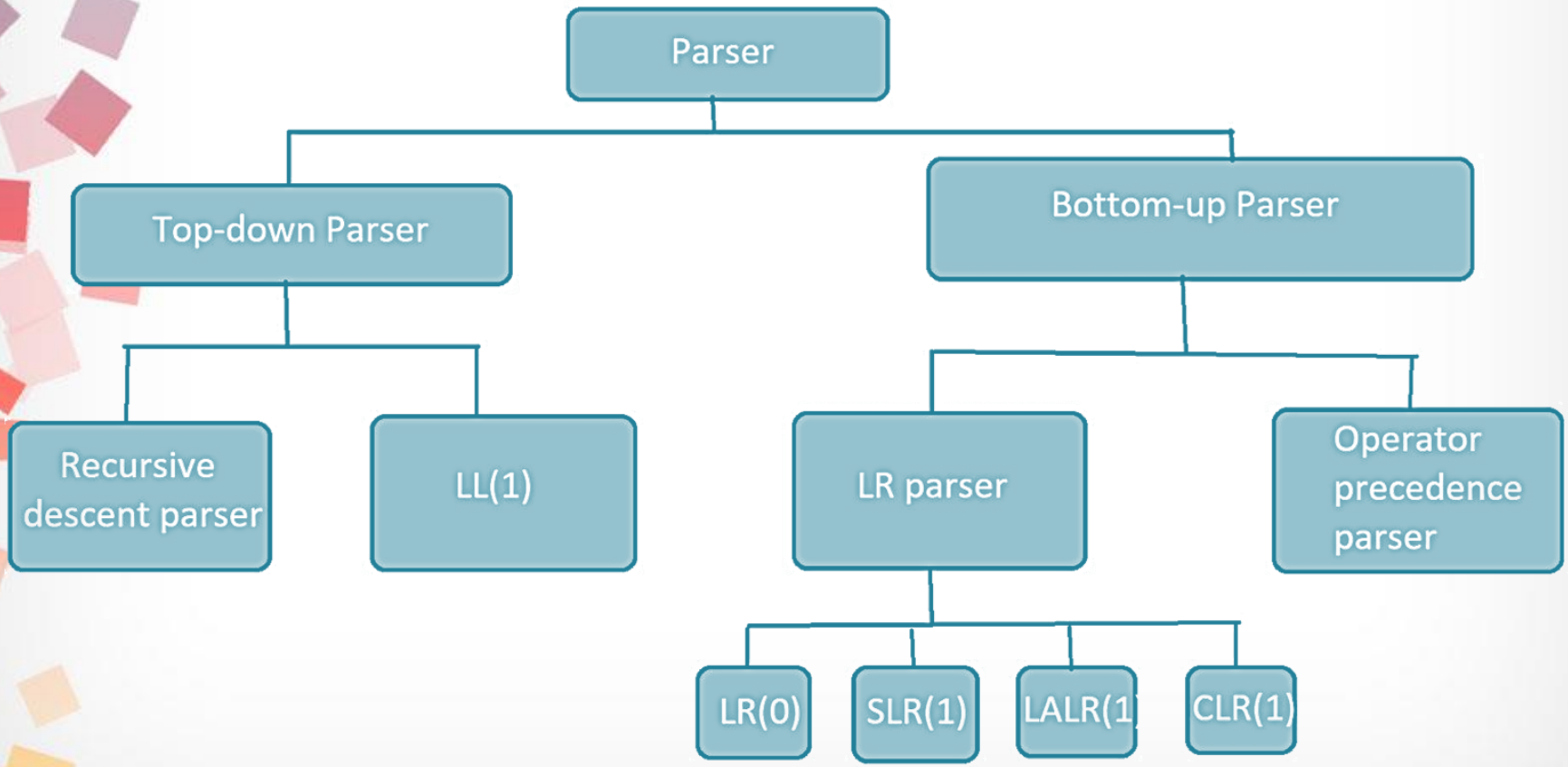
5. Replacing a character with an incorrect character.

```
#include <iostream>
using namespace std;
int main() {int x = 12$34       return 0; }
```

lexical error as '$' doesn't belong within 0-9 range

Syntax analysis

The ***parser*** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation (IR). The parser is also known as ***Syntax Analyzer***.



Classification of Parser

The parser has two functions:

1. It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language.
2. It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

Parse Tree and Derivations

1. Parse tree is the hierarchical representation of terminals or non-terminals.
2. These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings.
3. In parsing, the string springs using the beginning symbol.
4. The starting symbol of the grammar must be used as the root of the Parse Tree.
5. Leaves of parse tree represent terminals.
6. Each interior node represents productions of a grammar.

Top-Down Parser:

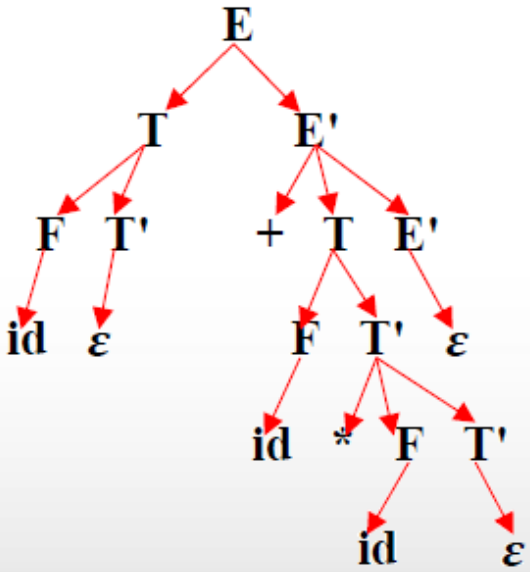
The ***top-down parser*** is the parser that generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses ***left most derivation***.

Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

1. ***Recursive descent parser:*** is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.
2. ***Non-recursive descent parser*** is also known as ***LL(1)*** parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

Example: The sequence of parse trees for the input **id+id×id** is a top-down parse according to the following grammar:

E → TE'
E' → +TE' | ε
T → FT'
T' → ×FT' | ε
F → (E) | id



Recursive-Descent Parsing

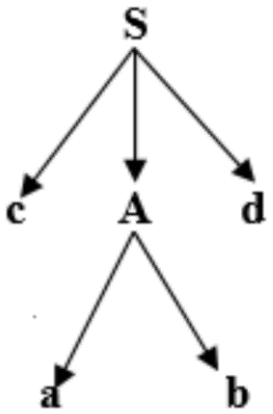
A typical procedure for a nonterminal in a top-down parser:

```
void A( ) {
1. Choose an A-production A → X1, X2,… Xk;
2. for (i = 1 to k) {
3. if (Xi is a nonterminal)
4. call procedure Xi ( );
5. else if (Xi equals the current input symbol a)
6. advance the input to the next symbol;
7. else /* an error has occurred */;}
}
```

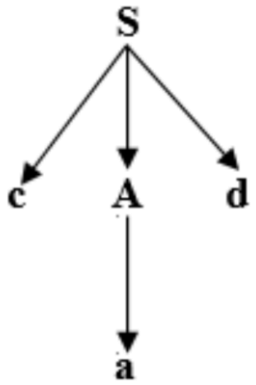
Example:-

S→ cAd

A→ ab|a



w= “cabd”



w=“cad”

A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand a nonterminal A, we may eventually find ourselves again trying to expand A without having consumed any input.

FIRST and FOLLOW

The construction, of both top-down and bottom-up parsers is aided by two functions. FIRST and FOLLOW, associated with a grammar G. During top-down parsing FIRST and FOLLOW allow us to choose which production to apply based on the next input symbol.

During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin string derived from α, if α →ε, then ε is also in FIRST (α).

Define FOLLOW(A), for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form S →αAαβ, for some α and β,

In addition, if A can be the rightmost symbol in some sentential form, then **$** is in FOLLOW(A); recall that **$** is a special "end marker" symbol that is assumed not to be a symbol of any grammar.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ***ε*** can be added to any FIRST set.

Example: Consider again the non-left-recursive grammar:

E → TE‘
E' → +TE' | ε
T → FT‘
T' → ×FT' | ε
F → (E) | id

Solution:

First (E) = {(, id}
First (E') = {+, ε}
First (T) = {(, id}
First (T') = {×, ε}
First (F) = {(, id}

Follow (E) = {$,)}
Follow (E') = {$,)}
Follow (T) = {+, $,)}
Follow (T') = {+, $,)}
Follow (F) = {×, +, $,)}

Example2:

S → ABCDE
A → abc | ε
B → b | ε
C → c
D → d | ε
E → e | ε

FIRST(S)={a,b,c}
FIRST(A)={a,ϵ}
FIRST(B)={b,ϵ}
FIRST(C)={c}
FIRST(D)={d,ϵ}
FIRST(E)={e,ϵ}

FOLLOW(S)={$}
FOLLOW(A)={b,c}
FOLLOW(B)={c}
FOLLOW(C)={d,e,$}
FOLLOW(D)={e,$}
FOLLOW(E)={$}

H.W.: Find First and Follow functions for the following grammar

S → Bb | Cd
B → aB | ε
C → cC | ε