# 1.Define algorithm for LL1 parser, algorithm for first and follow (definition with rules), algorithm for left recursion, left factoring (definition, algorithm, rules).

## Algorithm to construct LL (1) Parsing Table:

Step 1:  First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First () and Follow() for all non-terminals.

1. First (): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow (): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.  
   To construct the parsing table, we have two functions:

In the table, rows will contain the non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

## Algorithm for first and follow

α → t β

Algorithm for calculating First set

Look at the definition of FIRST(α) set:

* if α is a terminal, then FIRST(α) = { α }.
* if α is a non-terminal and α → ℇ is a production, then FIRST(α) = { ℇ }.
* if α is a non-terminal and α → 𝜸1 𝜸2 𝜸3 … 𝜸n and any FIRST(𝜸) contains t then t is in FIRST(α).

Algorithm for calculating Follow set:

* if α is a start symbol, then FOLLOW() = $
* if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except ℇ.
* if α is a non-terminal and has a production α → AB, where B ℇ, then FOLLOW(A) is in FOLLOW(α).

### Definition and rules of first and follow

First(A) contains all terminals in first place of every string derived by A

Follow(A) contains set of all terminals present immediately in right of “A”.

Text, letter

Description automatically generated

## Algorithm for left recursion

### Definition

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

A → βA’

A’ → αA’ | ε

without changing the set of strings derivable from A.

### Algorithm

1. Arrange the non-terminals in some order A1, A2 . . . An.

2. **for** i := 1 **to** n **do** **begin**

**for** j := 1 **to** i-1 **do begin**

replace each production of the form Ai → Aj γ

by the productions Ai → δ1 γ | δ2γ | . . . | δk γ

where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;

**end**

eliminate the immediate left recursion among the Ai-productions

**end**

## Algorithm for left factoring

### Definition

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice

If there is any production A → αβ1 | αβ2 , it can be rewritten as

**A → αA’**

**A’ → β1 | β**

## Example 1:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> \*FT'|ε

F -> (E)|id

from the given grammar parse the given input string:

id+id\*id

### Nullable/First/Follow Table, Transition Table, Parsing table and Tree

E -> T E'

E' -> + T E'

E' -> ε

T -> F T'

T' -> \* F T'

T' -> ε

F -> ( E )

F -> id

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbol** | **FIRST** | | **FOLLOW** | |
| E | {(, id } | | { $ , ) } | |
| E' | {+, ε} | | { $ , ) } | |
| T | {id, (} | | { $ , ) ,+ } | |
| T' | {\* , ε } | | { $ , ) , + } | |
| F | {id, ( } | | { $ , ) , + , \* } | |
| **Stack** | **Input** | | **Rule** | |
| $ E | id + id \* id $ | |  | |
| $ E' T | id + id \* id $ | | E -> T E' | |
| $ E' T' F | id + id \* id $ | | T -> F T' | |
| $ E' T' id | id + id \* id $ | | F -> id | |
| $ E' T' | + id \* id $ | |  | |
| $ E' | + id \* id $ | | T' -> ε | |
| $ E' T + | + id \* id $ | | E' -> + T E' | |
| $ E' T | id \* id $ | |  | |
| $ E' T' F | id \* id $ | | T -> F T' | |
| $ E' T' id | id \* id $ | | F -> id | |
| $ E' T' | \* id $ | |  | |
| $ E' T' F \* | \* id $ | | T' -> \* F T' | |
| $ E' T' F | id $ | |  | |
| $ E' T' id | id $ | | F -> id | |
| $ E' T' | $ | |  | |
| $ E' | $ | | T' -> ε | |
| $ | $ | | E' -> ε | |
| Non-Terminal | INPUT SYMBOLS | | | | | |
|  | id | + | \* | ( | ) | $ |
| E | E -> TE’ |  |  | E -> TE’ |  |  |
| E’ |  | E’-> +TE’ |  |  | E’ -> ε | E’ -> ε |
| T | T -> FT’ |  |  | T -> FT’ |  |  |
| T’ |  | T’ -> ε | T’ -> \*FT’ |  | T’ -> ε | T’ -> ε |
| F | F -> id |  |  | F -> (E) |  |  |

## Example 2:

S-> AA

S->aA

S->b

from the given grammar parse the given input string:

abab

### Nullable/First/Follow Table, Transition Table, Parsing table and Tree

## Example 3:

S ->aBa

B ->bB | e

from the given grammar parse the given input string Input :

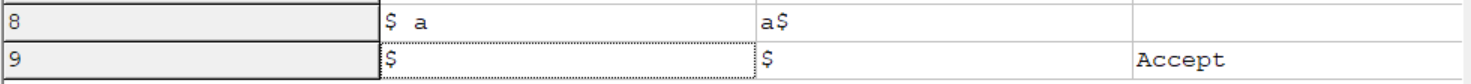
abba

### Nullable/First/Follow Table, Transition Table, Parsing table, and Tree

First(S): a Follow(S): $

First(B): be Follow(B): a

Graphical user interface, table

Description automatically generated with medium confidence 

Table

Description automatically generated

## Example 4:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> \*FT'|ε

F -> (E)|id

Input string to parse: id+id

### Nullable/First/Follow Table, Transition Table, Parsing table and Tree

|  |  |
| --- | --- |
| **FIRST** | **FOLLOW** |
|  |  |
| {(,id} | {$,)} |
|  |  |
| {+, ε } | {$,)} |
|  |  |
| {(,id} | {+,$,)} |
|  |  |
| {\*, ε } | {+,$,)} |
|  |  |
| {(,id} | {\*,+,$,)} |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Nonterminal | + | \* | ( | ) | id | $ |
| E |  |  | E -> T E' |  | E -> T E' |  |
|  |  |  |  |  |  |  |
| E' | E' -> + T E' |  |  | E' -> ε |  | E' -> ε |
|  |  |  |  |  |  |  |
| T |  |  | T -> F T' |  | T -> F T' |  |
|  |  |  |  |  |  |  |
| T' | T' -> ε | T' -> \* F T' |  | T' -> '' |  | T' -> ε |
|  |  |  |  |  |  |  |
| F |  |  | F -> ( E ) |  | F -> id |  |

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input** | **Rule** |
| $ E | id + id $ |  |
| $ E' T | id + id $ | E -> T E' |
| $ E' T' F | id + id $ | T -> F T' |
| $ E' T' id | id + id $ | F -> id |
| $ E' T' | + id $ |  |
| $ E' | + id $ | T' -> ε |
| $ E' T + | + id $ | E' -> + T E' |
| $ E' T | id $ |  |
| $ E' T' F | id $ | T -> F T' |
| $ E' T' id | id $ | F -> id |
| $ E' T' | $ |  |
| $ E' | $ | T' -> ε |
| $ | $ | E' -> ε |

# 2. What is Operator Precedence Parser-

## -Algorithm

**Input**: An input string w and a table of precedence relations.

**Output**: If w is well formed, a skeletal parse tree ,with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.

**Method**: Initially the stack contains $ and the input buffer the string w $. To parse, we execute the following program:

(1) Set ip to point to the first symbol of w$;

(2) repeat forever

(3) if $ is on top of the stack and ip points to $ then

(4) return

else begin

(5) let a be the topmost terminal symbol on the stack

and let b be the symbol pointed to by ip;

(6) if a < . b or a = b then begin

(7) push b onto the stack;

(8) advance ip to the next input symbol; end;

(9) else if a . > b then /\*reduce\*/

(10) repeat

(11) pop the stack

(12) until the top stack terminal is related by <.

to the terminal most recently popped

(13) else error( )

end

## -Rules to check whether given grammar is operator precedence or not

**Rule-01:**

If precedence of b is higher than precedence of a, then we define a < b

If precedence of b is same as precedence of a, then we define a = b

If precedence of b is lower than precedence of a, then we define a > b

**Rule-02:**

An identifier is always given the higher precedence than any other symbol.

$ symbol is always given the lowest precedence.

**Rule-03:**

If two operators have the same precedence, then we go by checking their associativity.

## -Implementation of Operator Precedence Parser from Parsing a Given String-

Consider the grammar E → E+E | E-E | E\*E | E/E | E↑E | (E) | id. Input string is id+id\*id .The implementation is as follows:

Table

Description automatically generated

## -To find handle

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Consider the grammar:

E → E+E

E → E\*E

E → (E)

E → id

And the input string id1+id2\*id3

The rightmost derivation is:

E → E+E

→ E+E\*E

→ E+E\*id3

→ E+id2\*id3

→ id1+id2\*id3

In the above derivation the underlined substrings are called handles.

## -Operator-Precedence Relations table

## -How to Create Operator-Precedence Relations

There are three disjoint precedence relations namely

< . - less than = - equal to

. > - greater than

The relations give the following meaning:

a < . b – a yields precedence to b

a = b – a has the same precedence as b

a . > b – a takes precedence over b

1. If operator θ1 has higher precedence than operator θ2, then make

θ1 . > θ2 and θ2 < . θ1

2. If operators θ1 and θ2, are of equal precedence, then make

θ1 . > θ2 and θ2 . > θ1 if operators are left associative

θ1 < . θ2 and θ2 < . θ1 if right associative

3. Make the following for all operators θ:

θ < . id , id . > θ

θ < . ( , ( < . θ

) . > θ , θ . > )

θ . > $ , $ < . θ

Also make

( = ) , ( < . ( , )  .> ) , ( < . id, id .> ) , $ <. id , id .> $ , $ <. ( , )

## Example 1:

E → E+E | E-E | E\*E | E/E | E^E | (E) | -E | id

input string

id+id\*id

### Table

Shape

Description automatically generated with medium confidence

## Example 2:

E → EAE | id

A → + | x

input string

id + id x id

### **Step-01:**

We convert the given grammar into operator precedence grammar.

The equivalent operator precedence grammar is-

E → E + E | E x E | id

### Step**-02:**

 The terminal symbols in the grammar are { id, + , x , $ }

We construct the operator precedence table as-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **id** | **+** | **x** | **$** |
| **id** |  | > | > | > |
| **+** | < | > | < | > |
| **x** | < | > | > | > |
| **$** | < | < | < |  |

Operator Precedence Table

### **Parsing Given String-**

Given string to be parsed is **id + id x id**.

We follow the following steps to parse the given string-

#### **Step-01:**

We insert $ symbol at both ends of the string as-

**$ id + id x id $**

We insert precedence operators between the string symbols as-

**$ < id > + < id > x < id > $**

#### **Step-02:**

We scan and parse the string as-

$ **< id >** + < id > x < id > $

$ E + **< id >** x < id > $

$ E + E x **< id >** $

$ E + E x E $

$ + x $

$ < + **< x >** $

$ **< + >** $

$ $

## Example 3:

Consider the following grammar-

S → ( L ) | a

L → L , S | S

Construct the operator precedence parser and parse the string

( a , ( a , a ) )

The terminal symbols in the grammar are { ( , ) , a , , }

### We construct the operator precedence table as-

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **a** | **(** | **)** | **,** | **$** |
| **a** |  | > | > | > | > |
| **(** | < | > | > | > | > |
| **)** | < | > | > | > | > |
| **,** | < | < | > | > | > |
| **$** | < | < | < | < |  |

Operator Precedence Table

### **Parsing Given String-**

Given string to be parsed is **( a , ( a , a ) )**.

We follow the following steps to parse the given string-

#### **Step-01:**

We insert $ symbol at both ends of the string as-

**$ ( a , ( a , a ) ) $**

We insert precedence operators between the string symbols as-

**$ < ( < a > , < ( < a > , < a > ) > ) > $**

#### **Step-02:**

We scan and parse the string as-

$ < ( **< a >** , < ( < a > , < a > ) > ) > $

$ < ( S , < ( **< a >** , < a > ) > ) > $

$ < ( S , < ( S , **< a >** ) > ) > $

$ < ( S , **< ( S , S ) >** ) > $

$ < ( S , **< ( L , S ) >** ) > $

$ < ( S , **< ( L ) >** ) > $

$ **< ( S , S ) >** $

$ **< ( L , S ) >** $

$ **< ( L ) >** $

$ **< S >** $

$ $

# 3. Find out left recursion from given grammar

## Example 1:

* A → ABd / Aa / a
* B → Be / b

### The grammar after eliminating left recursion is-

A → aA’

A’ → BdA’ / aA’ / ∈

B → bB’

B’ → eB’ / ∈

## Example 2:

* E → E + E / E x E / a

### The grammar after eliminating left recursion is-

E → aA

A → +EA / xEA / ∈

## Example 3:

* E → E + T / T
* T → T x F / F
* F → id

### The grammar after eliminating left recursion is-

E → TE’

E’ → +TE’ / ∈

T → FT’

T’ → xFT’ / ∈

F → id

## Example 4:

* S → (L) / a
* L → L , S / S

### The grammar after eliminating left recursion is-

S → (L) / a

L → SL’

L’ → ,SL’ / ∈

## Example 5:

* S → S0S1S / 01

### The grammar after eliminating left recursion is-

S → 01A

A → 0S1SA / ∈

## Example 6:

* A → Ba / Aa / c
* B → Bb / Ab / d

### The grammar after eliminating left recursion is-

S → A

A → aBA’ / acA’

A’ → dA’ / eA’ / ∈

B → bBc / f

## Example 7:

* X → XSb / Sa / b
* S → Sb / Xa / a

This is a case of indirect left recursion.

### Step-01:

First let us eliminate left recursion from A → Ba / Aa / c

Eliminating left recursion from here, we get-

A → BaA’ / cA’

A’ → aA’ / ∈

 Now, given grammar becomes-

A → BaA’ / cA’

A’ → aA’ / ∈

B → Bb / Ab / d

### **Step-02:**

Substituting the productions of A in B → Ab, we get the following grammar-

A → BaA’ / cA’

A’ → aA’ / ∈

B → Bb / BaA’b / cA’b / d

### **Step-03:**

 Now, eliminating left recursion from the productions of B, we get the following grammar-

A → BaA’ / cA’

A’ → aA’ / ∈

B → cA’bB’ / dB’

B’ → bB’ / aA’bB’ / ∈

 This is the final grammar after eliminating left recursion.

## Example 8:

* S → Aa / b
* A → Ac / Sd / ∈

This is a case of indirect left recursion.

### **Step-01:**

 First let us eliminate left recursion from S → Aa / b

This is already free from left recursion.

### **Step-02:**

Substituting the productions of S in A → Sd, we get the following grammar-

S → Aa / b

A → Ac / Aad / bd / ∈

### **Step-03:**

Now, eliminating left recursion from the productions of A, we get the following grammar-

S → Aa / b

A → bdA’ / A’

A’ → cA’ / adA’ / ∈

 This is the final grammar after eliminating left recursion.

## Example 9:

* S → A
* A → Ad / Ae / aB / ac
* B → bBc / f

### The grammar after eliminating left recursion is-

S → A

A → aBA’ / acA’

A’ → dA’ / eA’ / ∈

B → bBc / f

# 4.Find out left factor from given grammar

## Example 1:

* S → a / ab / abc / abcd

### **Step-01:**

S → aS’

S’ → b / bc / bcd / ∈

Again, this is a grammar with common prefixes.

### **Step-02:**

S → aS’

S’ → bA / ∈

A → c / cd / ∈

Again, this is a grammar with common prefixes.

### **Step-03:**

S → aS’

S’ → bA / ∈

A → cB / ∈

B → d / ∈

This is a left factored grammar.

## Example 2:

* S → bSSaaS / bSSaSb / bSb / a

### **Step-01:**

S → bSS’ / a

S’ → SaaS / SaSb / b

Again, this is a grammar with common prefixes.

### **Step-02:**

S → bSS’ / a

S’ → SaA / b

A → aS / Sb

This is a left factored grammar.

## Example 3:

* A → aAB / aBc / aAc

### **Step-01:**

A → aA’

A’ → AB / Bc / Ac

Again, this is a grammar with common prefixes.

### **Step-02:**

A → aA’

A’ → AD / Bc

D → B / c

This is a left factored grammar.

## Example 4:

* S→ aSSbS / aSaSb / abb / b

### **Step-01:**

S → aS’ / b

S’ → SSbS / SaSb / bb

Again, this is a grammar with common prefixes.

### **Step-02:**

S → aS’ / b

S’ → SA / bb

A → SbS / aSb

This is a left factored grammar.

## Example 5:

* S → aAd / aB
* A → a / ab
* B → ccd / ddc

### The left factored grammar is-

S → aS’

S’ → Ad / B

A → aA’

A’ → b / ∈

## Example 6:

* S → iEtS / iEtSeS / a
* E → b

### The left factored grammar is-

S → iEtSS’ / a

S’ → eS / ∈

E → b

# 5.Explian different phases of Compiler with example

### *Main Para*

A Compiler operates in phases, each of which transforms the source program from one representation into another.

The following are the phases of the compiler:

Main phases:

1) Lexical analysis

2) Syntax analysis

3) Semantic analysis

4) Intermediate code generation

5) Code optimization

6) Code generation

Sub-Phases:

1) Symbol table management

2) Error handling

Diagram

Description automatically generated

## LEXICAL ANALYSIS:

* + It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
  + It reads the characters one by one, starting from left to right and forms the tokens.
  + **Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: a + b = 20

Here, a,b,+,=,20 are all separate tokens.

Group of characters forming a token is called the **Lexeme**.

* + The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

## SYNTAX ANALYSIS:

* + It is the second phase of the compiler. It is also known as parser.
  + It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.
  + Syntax tree:

It is a tree in which interior nodes are operators and exterior nodes are operands.

* + Example: For a=b+c\*2, syntax tree is

=

a +

b \*

c 2

## SEMANTIC [ANALYSIS:](http://csetube.weebly.com/)

* + It is the third phase of the compiler.
  + It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
  + It performs type conversion of all the data types into real data types.

## INTERMEDI[ATE CODE GENERATION:](http://csetube.weebly.com/)

* + It is the fourth phase of the compiler.
  + It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
  + The three-address code consists of a sequence of instructions, each of which has atmost three operands.

Example: t1=t2+t3

## CODE OPTI[MIZATION:](http://csetube.weebly.com/)

* + It is the fifth phase of the compiler.
  + It gets the intermediate code as input and produces optimized intermediate code as output.
  + This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
  + During the code optimization, the result of the program is not affected.
  + To improve the code generation, the optimization involves
    - deduction and removal of dead code (unreachable code).
    - calculation of constants in expressions and terms.
    - collapsing of repeated expression into temporary string.
    - loop unrolling.
    - moving code outside the loop.
    - removal of unwanted temporary variables.

## CODE GENERATION:

* + It is the final phase of the compiler.
  + It gets input from code optimization phase and produces the target code or object code as result.
  + Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
  + The code generation involves
    - allocation of register and memory
    - generation of correct references
    - generation of correct data types
    - generation of missing code

## SYMBOL TABLE MANAGEMENT:

* + Symbol table is used to store all the information about identifiers used in the program.
  + It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
  + It allows to find the record for each identifier quickly and to store or retrieve data from that record.
  + Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

## ERROR HANDL[ING:](http://csetube.weebly.com/)

* + Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
  + In lexical analysis, errors occur in separation of tokens.
  + In syntax analysis, errors occur during construction of syntax tree.
  + In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
  + In code optimization, errors occur when the result is affected by the optimization.
  + In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c\*2. The figure shows the representation of this statement after each phase:

Diagram

Description automatically generated

# 6.Explain role for Lexical Analysis.

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

## THE ROLE OF THE LEXICAL ANALYZER

* + The lexical analyzer is the first phase of a compiler.
  + Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
  + Upon r[eceiving a “get next token” command from the parser, the lexical an](http://csetube.weebly.com/)alyzer reads input c[haracters until it can identify the next token.](http://csetube.weebly.com/)

Diagram

Description automatically generated

# 7.what are tokens, lexemes, pattern

## TOKENS

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language: sum=3+2;

|  |  |
| --- | --- |
| **Lexeme** | **Token type** |
| sum | Identifier |
| = | Assignment operator |
| 3 | Number |
| + | Addition operator |
| 2 | Number |
| ; | End of statement |

## LEXEME:

Collection or group of characters forming tokens is called Lexeme.

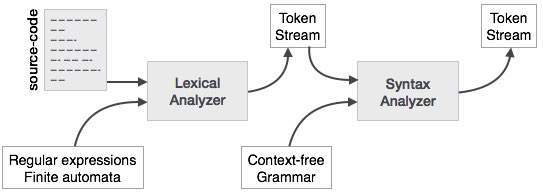
## PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

# 8.Explain role of Syntax Analyser(parser)

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

# 9.Need of Parser

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

A parser usually checks all data provided to ensure it is sufficient to build a data structure in the form of a parse tree or an abstract syntax tree.

For the code written in human-readable form to be understood by a machine, it must be converted into machine language. This task is usually performed by a translator (interpreter or compiler). The parser is commonly used as a component of the translator that organizes linear text in a structure that can be easily manipulated (parse tree). To do so, it follows a set of defined rules called “grammar”.

# 10.Different types of Parsers (Top down, bottom-up parser. Their types, difference between top down and bottom up)

## Top–down parsing

A parser can start with the start symbol and try to transform it to the input string. It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves

### 1. RECURSIVE DESCENT PARSING

* + Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
  + This parsing method may involve **backtracking**, that is, making repeated scans of the input.

### 2. PREDICTIVE PARSING

* + Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
  + The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

## Bottom-up parsing

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**

### SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

## Difference between top down and bottom up

|  |  |
| --- | --- |
| **Top-Down Approach** | **Bottom-Up Approach** |
| Top-Down Approach is Theory-driven. | Bottom-Up Approach is Data-Driven. |
| Emphasis is on doing things (algorithms). | Emphasis is on data rather than procedure. |
| Large programs are divided into smaller programs which is known as decomposition. | Programs are divided into what are known as objects is called Composition. |
| Communication is less among the modules. | Communication is a key among the modules. |
| Widely used in debugging, module documentation, etc. | Widely used in testing. |
| The top-down approach is mainly used by Structured programming languages like C, Fortran, etc. | The bottom-up approach is used by Object-Oriented programming languages like C++, C#, Java, etc. |
| May contains redundancy as we break up the problem into smaller fragments, then build that section separately. | This approach contains less redundancy if the data encapsulation and data hiding are being used. |

# 11. Algorithm for Recursive descent and predictive parser.

## Recursive descent

A picture containing graphical user interface

Description automatically generated

## Predictive parser

**Input:** Grammar *G* **Output** : Parsing table *M* **Method** :

1. For each production *A* → α of the grammar, do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add *A* → α to *M*[*A*, *a*].
3. If ε is in FIRST(α), add A → α to *M*[*A*, *b*] for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and $ is in FOLLOW(*A*) , add *A* → α to *M*[*A*, $].
4. Make each undefined entry of *M* be **error**.