# 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 and parsing table

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

## Example 3:

S ->aBa

B ->bB | e

from the given grammar parse the given input string Input :

abba

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| FIRST | FOLLOW | | Nonterminal | aBa | bB | e | $ |
|  |  | |  |  |  |  |  |
| {aBa} | {$} | | S | S -> aBa |  |  |  |
|  |  | |  |  |  |  |  |
| {bB,e} | {undefined} | | B |  | B -> bB | B -> e |  |
| Trace | |
| |  |  |  | | --- | --- | --- | | Stack | Input | Rule | | $ S | abba $ |  | | |

## Example 4:

E -> TE'

E'-> +TE'|ε

T -> FT'

T' -> \*FT'|ε

F -> (E)|id

Input string to parse: id+id

|  |  |
| --- | --- |
| **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

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

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

## -To find handle

## -Operator-Precedence Relations table

## -How to Create Operator-Precedence Relations

## Example 1:

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

input string

id+id\*id

## Example 2:

E → EAE | id

A → + | x

input string

id + id x id

## 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 ) )

# 3. Find out left recursion from given grammar

## Example 1:

* A → ABd / Aa / a
* B → Be / b

## Example 2:

* E → E + E / E x E / a

## Example 3:

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

## Example 4:

S → (L) / a

L → L , S / S

## Example 5:

* S → S0S1S / 01

## Example 6:

1. A → Ba / Aa / c

B → Bb / Ab / d

## Example 7:

1. X → XSb / Sa / b

S → Sb / Xa / a

## Example 8:

S → Aa / b

A → Ac / Sd / ∈

## Example 9:

* S → A

A → Ad / Ae / aB / ac

B → bBc / f

# 4.Find out left factor from given grammar

## Example 1:

S → a / ab / abc / abcd

## Example 2:

* S → bSSaaS / bSSaSb / bSb / a

## Example 3:

* A → aAB / aBc / aAc

## Example 4:

* S→ aSSbS / aSaSb / abb / b

## Example 5:

S → aAd / aB

A → a / ab

B → ccd / ddc

## Example 6:

* S → iEtS / iEtSeS / a
* E → b

# 5.Explian different phases of Compiler with example

# 6.Explain role for Lexical Analysis.

# 7.what are tokens,lexmes,pattern

# 8.Explain role of Syntax Analyser(parser)

# 9.Need of Parser

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

# 11Algorithm for Recursive descent and predictive parser.