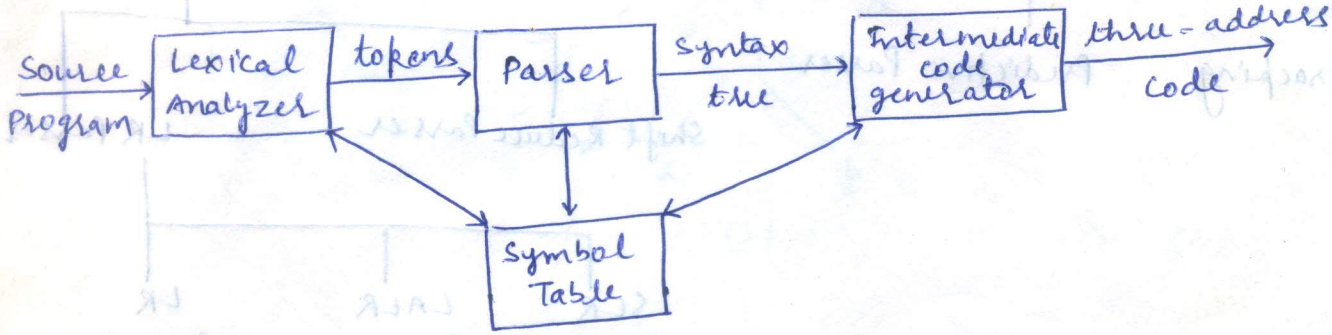
# UNIT-2

## Parser :→

- The component that performs the syntax analysis is called syntax analyzer or parser.




A model of a compiler front end

The main functions and tasks performed by a parser are –

(i) A parser obtains a sequence of tokens from the lexical analyzer.

(ii) checks that the tokens appearing in its input occur in patterns permitted by the specification for the source language.

(iii) A parser verifies that the string can be generated by the grammar for the source language.

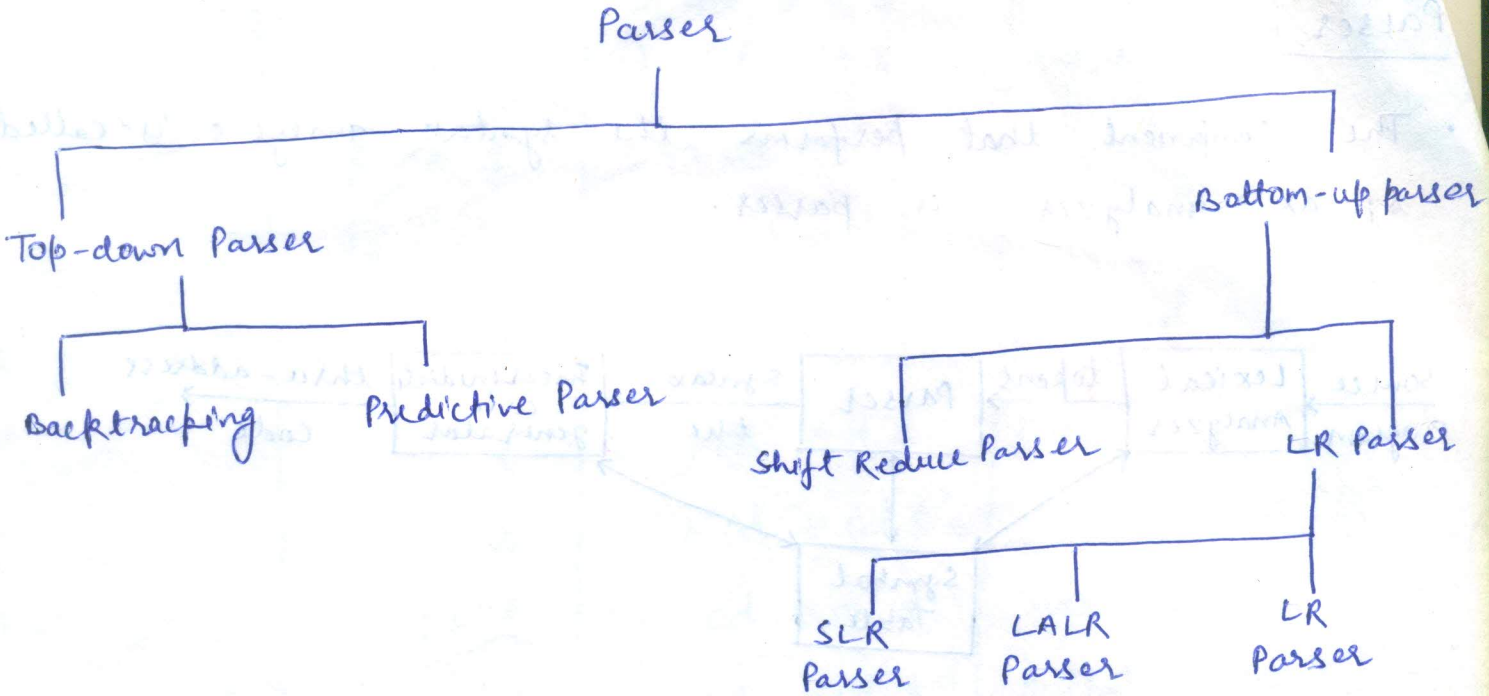
(iv) A parser imposes a hierarchical tree like structure (Parse tree) on the incoming token stream from which the intermediate code can be generated.

(v) A parser detects and reports any syntax error.

(vi) Collects all the information in the symbol table.

(vii) Passes its output to the intermediate code generator for further processing.

# Types of parser




## Parsing Techniques

## Top - Down Parsing

- Top down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.
- Equivalently, top down parsing can be viewed as finding a leftmost derivation for an input string.

Ex:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

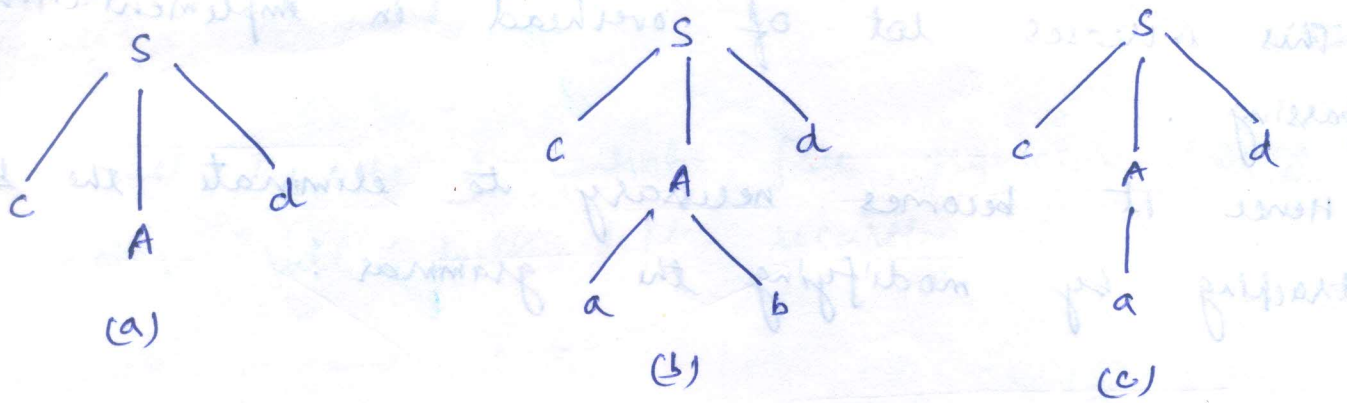
input $\Rightarrow$ id + id * id

Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string $w = cad$.




## Problems with Top-down Parsing –

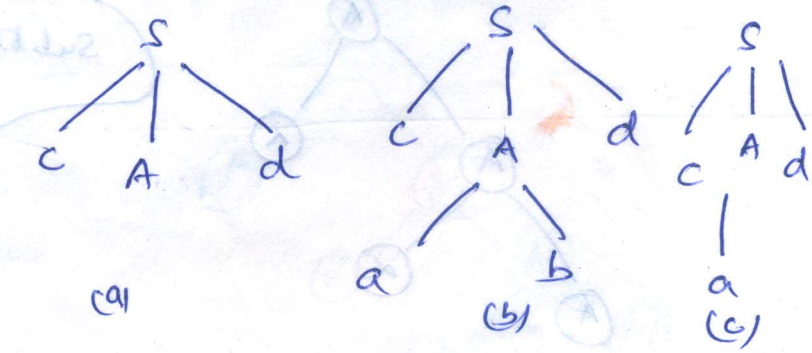
- In order to implement the parsing we need to eliminate following problems.

### (a) Backtracking :–

- Backtracking is a technique in which for expansion of non terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any –

ex. $S \rightarrow cAd$

$A \rightarrow ab \mid a$




- If for a non-terminal there are multiple production rules beginning with the same input symbol then

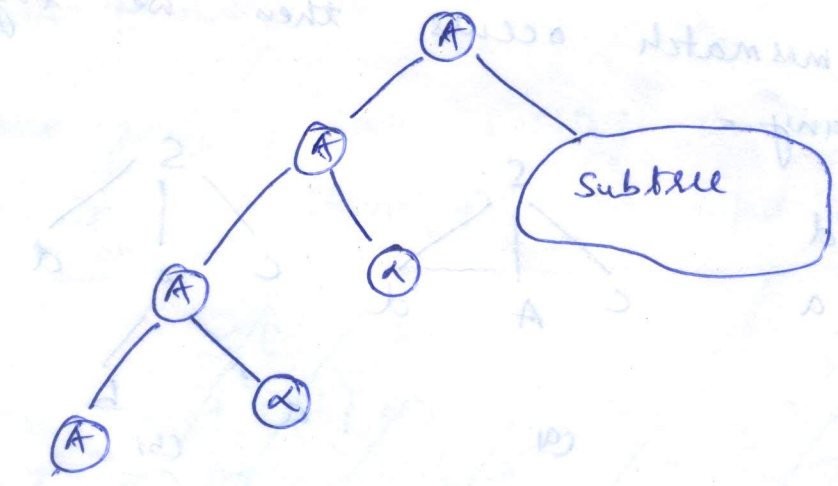
to get the correct derivation we need to try all these alternatives.

- secondly, in backtracking we need to move some levels upward in order to check the possibilities.
- This increases lot of overhead in implementation of passing.
- Hence it becomes necessary to eliminate the backtracking by modifying the grammar.

(b) Left Recursion :-

$$A \overset{+}{\Rightarrow} A\alpha$$

- Here $\overset{+}{\Rightarrow}$ means deriving the input in one or more steps.
- The A can be non terminal and $\alpha$ denotes some input string.
- Because of left recursion the top down parser can enter on infinite loop.




left recursion

Thus expansion of A causes further expansion of A only and due to generation of A, Aα, Aαα, ... the input pointer will not be advanced.

- To eliminate left recursion we need to modify the grammar.

→ Let G be a context free grammar having a production rule with left recursion.

$$A \rightarrow A\alpha \quad \text{---------- (1)}$$

$$A \rightarrow \beta$$

Then we eliminate left recursion by re-writing the production rule as -

$$A \rightarrow \beta A'$$

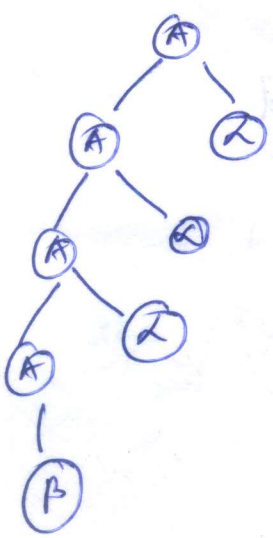
$$A' \rightarrow \alpha A' \quad \text{---------- (2)}$$

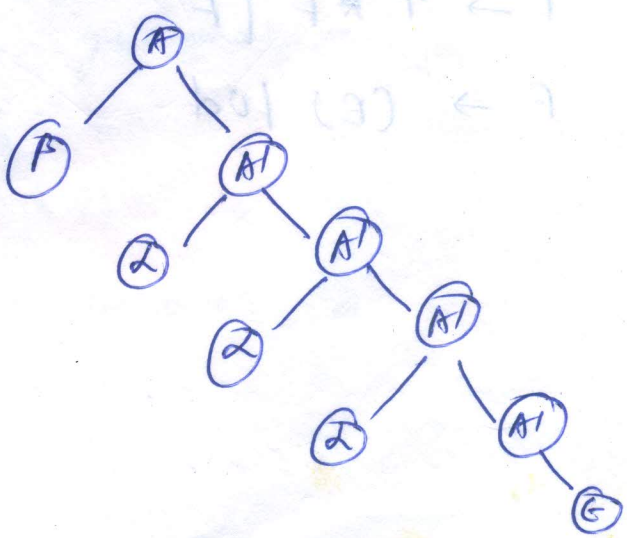
$$A' \rightarrow \in$$

Thus a new symbol A' is introduced. We can also verify, whether the modified grammar is equivalent to original or not.

| β | α | α | α |
|---|---|---|---|

input string



For ex:- Consider the grammar

$$E \rightarrow E+T \mid T$$

we can map this grammar with the rule $A \rightarrow A\alpha \mid \beta$.

Then using equation (2) we can say,

$A = E$

$\alpha = +T$

$\beta = T$

then the rule becomes,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

similarly for the rule,

$T \rightarrow T*F \mid F$

we can eliminate left recursion as

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

The grammar for arithmetic expression can be equivalently written as -

$E \rightarrow E+T \mid T$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T \rightarrow T*F \mid F \Rightarrow T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id \Rightarrow F \rightarrow (E) \mid id$

## Left factoring :→

- Left factoring is used when it is not clear that which of the two alternatives is used to expand the non terminal.
- By left factoring we may be able to re-write the production in which the decision can be deferred until enough of the input is seen to make the right choice.

In general if

$A \to \alpha\beta_1 \mid \alpha\beta_2$ is a production then it is not possible for us to take a decision whether to choose first rule or second.

- In such situation the above grammar can be left factored as

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2 \mid \epsilon$$

for ex. consider the following grammar

$$S \to iEtS \mid iEtSeS \mid a$$
$$E \to b$$

the left factored grammar becomes,

$$S \to iEtSS' \mid a$$
$$S' \to eS \mid \epsilon$$
$$E \to b$$

Ex- Do left factoring in the following grammar-

$A \rightarrow aAB \mid aA \mid a$

$B \rightarrow bB \mid b$

If the rule is $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots$ is a production then the grammar needs to be left factored.

$A \rightarrow aAB \mid aA \mid a$

$\alpha \; \beta_1 \quad \alpha \; \beta_2 \quad \alpha \; \beta_3$

We have to convert it to

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

$\Rightarrow$

$A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \epsilon$

Similarly

$B \rightarrow bB \mid b$

$A \quad \alpha \; \beta_1 \quad \alpha \; \beta_2$

$\Rightarrow$

$B \rightarrow bB'$

$B' \rightarrow B \mid \epsilon$

To summarize, the grammar with left factor operation will be -

$A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \epsilon$

$B \rightarrow bB'$

$B' \rightarrow B \mid \epsilon$

## Ambiguity -

Same as in unit-1.

A backtracking parser will try different production rules to find the match for the input string by backtracking each time.

- more powerful than predictive parsing but slower than predictive parsing and is not preferred for practical compilers.

Predictive Parsers

- Recursive Descent Parsing
- LL(1) Parsing

The predictive parser tries to predict the next construction using one or more lookahead symbols from input string.

### (1) Recursive Descent Parsing :→

- A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent (RD) Parser.

- The R.H.S. of the production rule is directly converted to a program.
- For each non-terminal a separate procedure is written and body of the procedure (code) is R.H.S. of the corresponding non terminal.

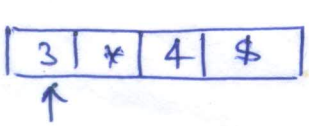
## Basic steps for construction of RD parser :-

The R.H.S. of the rule is directly converted into program code symbol by symbol -

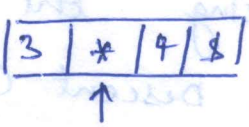
1. If the input symbol is non-terminal then a call to the procedure corresponding to the non-terminal is made.
2. If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
3. If the production rule has many alternatives then all these alternates has to be combined into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol.

Ex- $E \rightarrow num\ T$

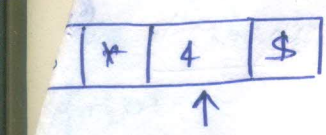
$T \rightarrow * num\ T \mid \epsilon$



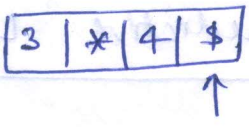

$E \rightarrow num\ T$ ;




$T \rightarrow * num\ T$




$T \rightarrow *numT$ ;




Declare success

## Pseudo code for recursive descent parser of above example -

```
Procedure E
{
  if lookahead = num then
  {
     match(num);
     T'; /* call to procedure T
  }
else error;
if lookahead = $
{
  declare success; /* Return on success
  }
else
    error;
} /* end of procedure E
```

```
Procedure T
{
  if lookahead = '*'
{
   match('*');
If lookahead = 'num'
{  match(num);
    T';
} else error }
```

```
else NULL /* Indicates ε
Here the other alternate is combined into same
                procedure T
  }

procedure match(token t)
{ if lookahead = t
lookahead = next_token;
  else
  error
}
```

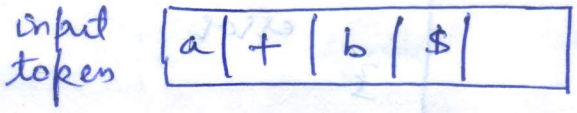
```
procedure error
{
 print("Error");
 }
```

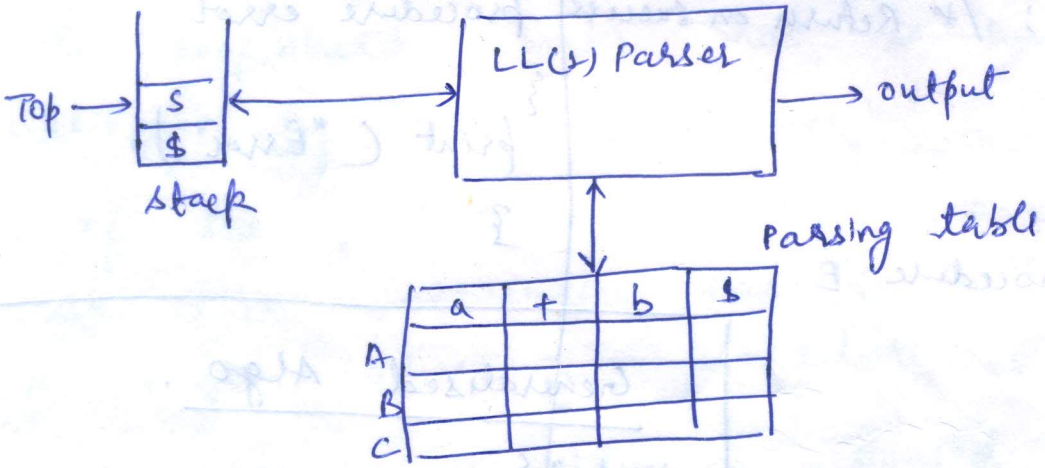
## Generalised Algo.

```
void A{
choose an A production, A → X1, X2 .. Xk;
 for (i=1 to k) {
 if (Xi is a nonterminal)
     call procedure Xi();
elseif (Xi equals the current
  input symbol a)
     advance the i/p to the next symbol;
else error
```

# Predictive LL(1) Parser

- This top-down parsing algorithm is of non-recursive type and in this a table is built.
- The first L means the input is scanned from left to right.
- The second L means it uses leftmost derivation for input string.
- The number 1 in the input symbol means it uses only one input symbol (lookahead) to predict the parsing process.




Model for LL(1) parser

- The data structures used by LL(1) are i) input buffer ii) stack iii) passing table.
- LL(1) parser uses i/p buffer to store the input tokens.

The stack is used to hold the left sentential form.

- The symbols on R.H.S. of rule are pushed into the stack in reverse order i.e from right to left.
- Thus use of stack makes this algorithm non-recursive.
- The table is basically a two dimensional array.
- The table has rows for non terminal and column for terminals.
- The table can be represented as M[A,a] where A is a non terminal and a is current input symbol.
- The parser consults the table M[A,a] each time while taking the parsing actions.
- The configuration of LL(1) parser can be defined by top of the stack and a lookahead token.
- one by one configuration is performed and the input is successfully parsed if the parser reaches the halting configuration.
- When the stack is empty and next token is $ then it corresponds to successful parse.

# FIRST and FOLLOW Function

The construction of both top down and bottom up parser is aided by two functions, FIRST and FOLLOW.

## FIRST function -

- FIRST($\alpha$) is a set of terminal symbols that are first symbols appearing at R.H.S. in derivation of $\alpha$.

  If $\alpha \Rightarrow \epsilon$ then $\epsilon$ is also in FIRST($\alpha$).

Following are the rules used to compute the FIRST functions -

1. If the terminal symbol a the FIRST(a) = {a}.
2. If there is a rule $X \to \epsilon$ then FIRST(X) = {$\epsilon$}
3. For the rule $A \to X_1 X_2 \cdots X_K$ FIRST(A) = {FIRST($X_1$) $\cup$ FIRST($X_2$) . FIRST($X_K$).

   where K $X_j \leq n$ such that $1 \leq j \leq K-1$.

## FOLLOW function :-

FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A.

Alternatively FOLLOW(A) = {a | $S \overset{*}{\Rightarrow} \alpha A a \beta$ where $\alpha$ and $\beta$ are some grammar symbols may be terminal or non terminal.

Rules - 1. For the start symbol S place $ in FOLLOW(S).

If there is a production $A \to \alpha B\beta$ then everything in FIRST($\beta$) without $\epsilon$ is to be placed in FOLLOW(B).

3. If there is a production $A \to \alpha B\beta$ or $A \to \alpha B$ and FIRST($\beta$) = {$\epsilon$} then FOLLOW(A) = FOLLOW(B) or FOLLOW(B) = FOLLOW(A).

That means everything in FOLLOW(A) is in FOLLOW(B).

Example -

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

1. FIRST(F) = FIRST(T) = FIRST(E) = {(, id}

- Because the two productions for F have bodies that start with these two terminal symbols, id and the left parenthesis.
- T has only one production, and its body starts with F.
- Since F does not derive $\epsilon$, FIRST(T) must be the same as FIRST(F).

2. FIRST(E') = {+, ∈}

Because one of the two productions for E' has a body that begins with terminal +, and the other body is ∈.

- Whenever a nonterminal derives ∈, we place ∈ in FIRST for that nonterminal.

3. FIRST(T') = {*, ∈} because analogous to that for FIRST(E').

4. FOLLOW(E) = FOLLOW(E') = {), $}.

- Since E is the start symbol, FOLLOW(E) must contain $.
- The production body (E) explains why the right parenthesis is in FOLLOW(E).
- For E', this nonterminal appears only at the ends of bodies of E - productions. Thus FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW(T') = {+, ), $}

Because T appears in bodies only followed by E'.

- Thus everything except ∈ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol +.
- However, since FIRST(E') contains ∈ (i.e. $E' \Rightarrow^* \epsilon$) and E' is the entire string following T in the bodies of the E - productions, everything in FOLLOW(E) must also be in FOLLOW(T).

hat explains the symbols $ and the right parentheses.

As for T', since it appears only at the ends of the T- productions, it must be that FOLLOW(T') = FOLLOW(T).

6. FOLLOW(F) = {+, *, ), $} because analogous to that for T in point (5).

| Symbols | FIRST | FOLLOW |
|---|---|---|
| E | {(, id} | {), $} |
| E' | {+, ∈} | {), $} |
| T | {(, id} | {+, ), $} |
| T' | {*, ∈} | {+, ), $} |
| F | {(, id} | {+, *, ), $} |

## Construction of a predictive parsing table

For each production A → α of the grammar, do the following –

1. For each terminal a in FIRST(α), add A → α to M[A, a].
2. If ∈ is in FIRST(α), then for each terminal b in FOLLOW(A), add A → α to M[A, b].
   - If ∈ is in FIRST(α) and $ is in FOLLOW(A) add A → α to M[A, $] as well.

If after performing the above, there is no production at all in M[A, a], then set M[A, a] to error (empty entry in the table).

Eg -

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

| Non Terminal | Input Symbol | | | | | |
|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ |
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| E' | | $E' \rightarrow +TE'$ | | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ | | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow id$ | | | $F \rightarrow (E)$ | | |

consider production $E \rightarrow TE'$

since $FIRST(TE') = FIRST(T) = \{(, id\}$

this production is added to $M[E, (]$ and $M[E, id]$

Production $E' \rightarrow +TE'$ is added to $M[E', +]$ since

$FIRST(+TE') = \{+\}$

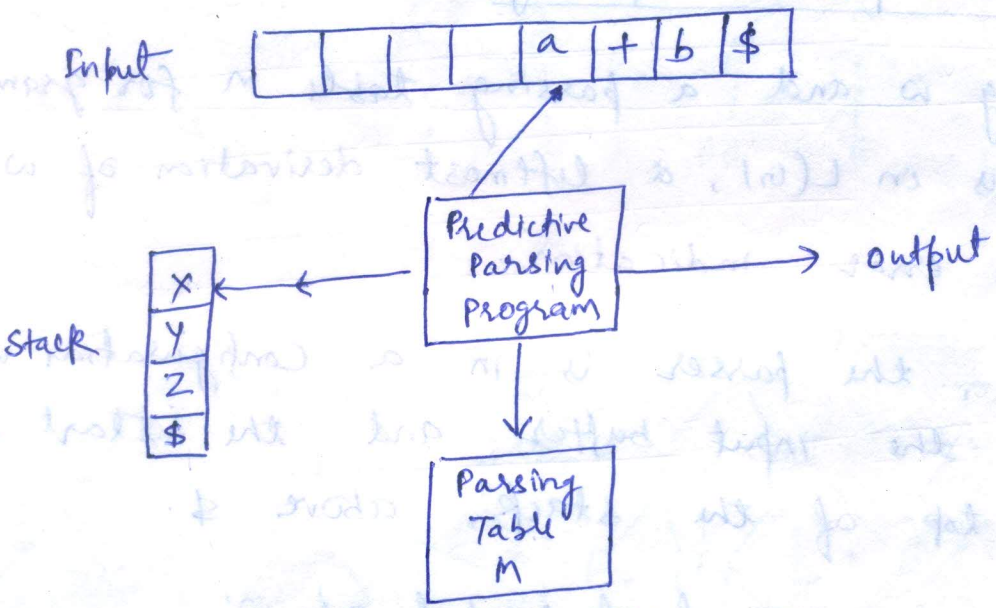
since $FOLLOW(E') = \{), \$\}$, production $E' \rightarrow \epsilon$ is added to $M[E', )]$ and $M[E', \$]$.

# Non recursive Predictive Parsing

A non recursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.

- If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols $\alpha$ such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$




Model of a table-driven predictive parser

- Parsing table is constructed by previous algo. and according to the above diagram, the input buffer contains the string to be parsed, followed by the endmarker $.
- We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $.

- The parser is controlled by a program that considers X, the symbol on top of the stack and a, the current input symbol.
- If X is a nonterminal, the parser chooses an X-production by consulting entry M[X, a] of the parsing table M.
- Otherwise, it checks for a match between the terminal X and current input symbol a.

Algo. (Table-driven predictive parsing)

Input: A string w and a parsing table M for grammar G.

output: If w is in L(G), a leftmost derivation of w; Otherwise, an error indication.

Method: Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above $.

```
let a be the first symbol of w;
let X be the top stack symbol;
while (X ≠ $) { if (X=a) pop the stack and
                    let a be the next symbol of w;
else if (X is a terminal) error();
else if (M[X,a] is an error entry) error();
else if (M[X, a] = X → Y1, Y2 ... Yk {
        output the production X → Y1, Y2 ... Yk;
        pop the stack;
        push Yk, Yk-1 ... Y1 onto the stack with Y1 on top;
    } let X be the top stack symbol; }
```

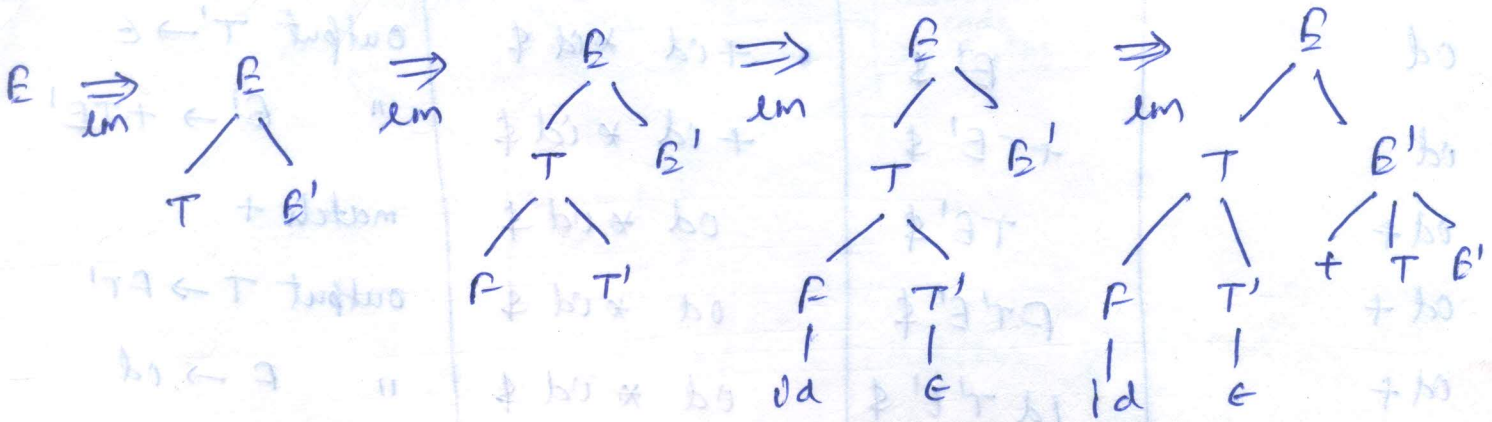
Predictive Parsing Algo.

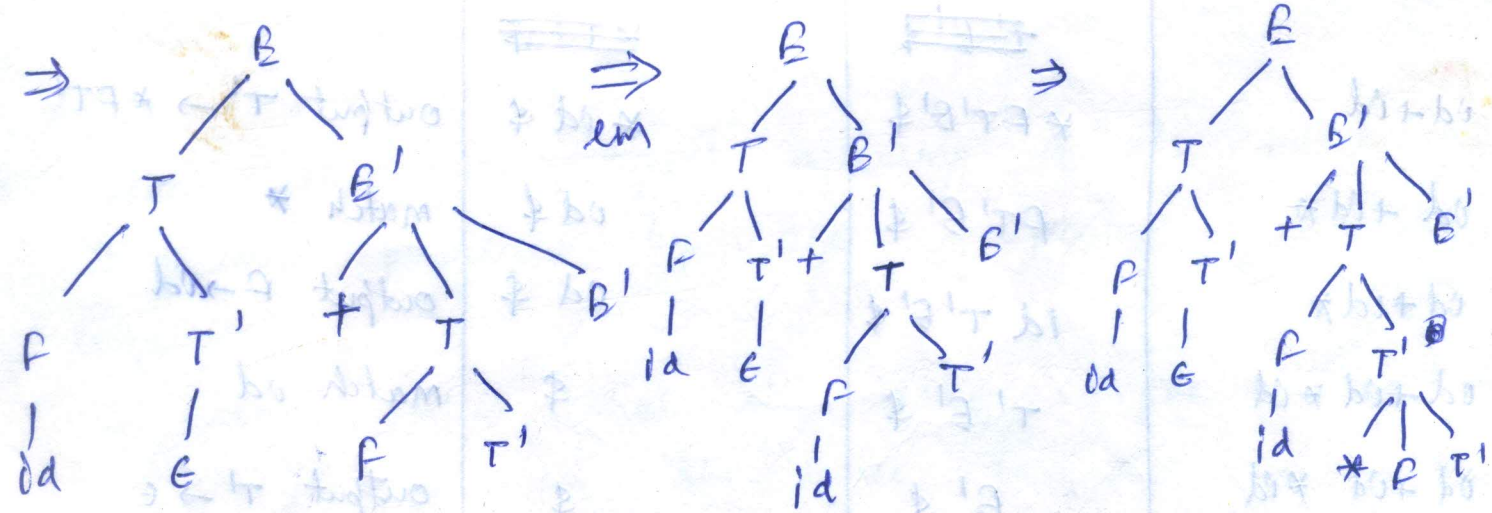
Example -

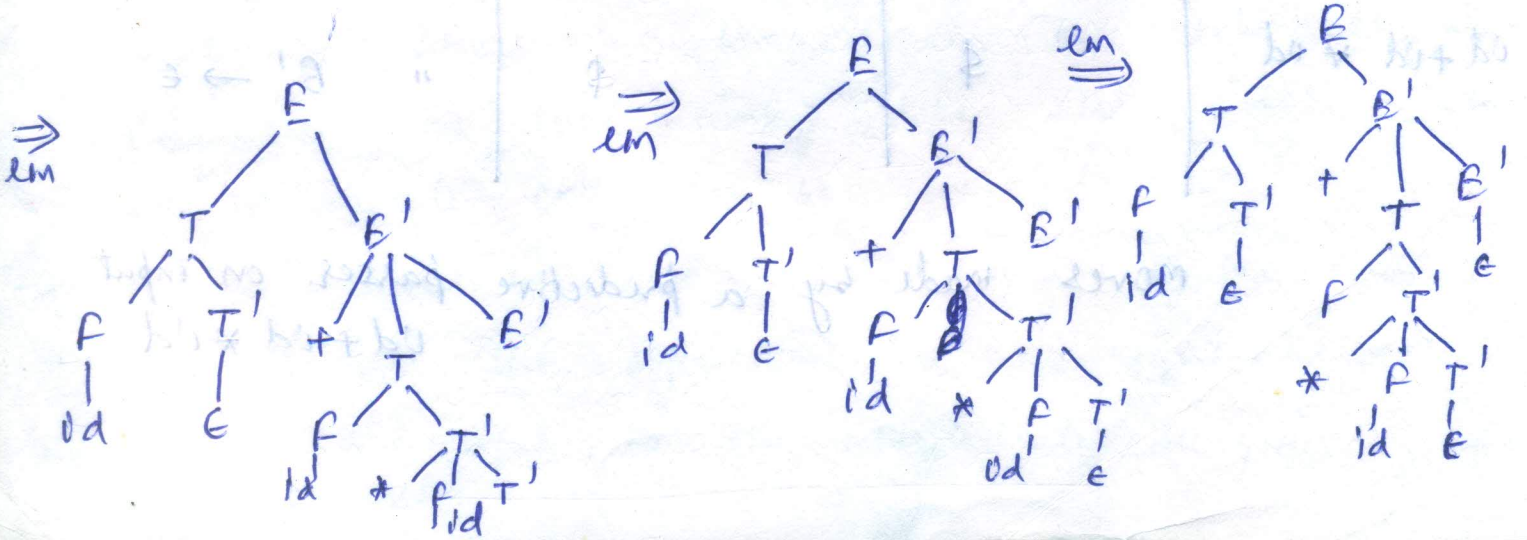
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

Apply non recursive predictive algo. on input id+id*id

Derivation.







| matched | Stack | Input | Action |
|---|---|---|---|
| | $E\$$ | $id + id * id\$$ | |
| | $TE'\$$ | $id + id * id\$$ | output $E \to TE'$ |
| | $FT'E'\$$ | $id + id * id\$$ | " $T \to FT'$ |
| | $id\,T'E'\$$ | $id + id * id\$$ | " $F \to id$ |
| $id$ | $T'E'\$$ | $+ id * id\$$ | matched $id$ |
| $id$ | $E'\$$ | $+ id * id\$$ | output $T' \to \epsilon$ |
| $id$ | $+TE'\$$ | $+ id * id\$$ | " $E' \to +TE'$ |
| $id +$ | $TE'\$$ | $id * id\$$ | match $+$ |
| $id +$ | $FT'E'\$$ | $id * id\$$ | output $T \to FT'$ |
| $id +$ | $id\,T'E'\$$ | $id * id\$$ | " $F \to id$ |
| $id + id$ | $T'E'\$$ | $* id\$$ | match $id$ |
| | ~~$+TE'\$$~~ | ~~$* id\$$~~ | |
| $id + id$ | $*FT'E'\$$ | $* id\$$ | output $T' \to *FT'$ |
| $id + id *$ | $FT'E'\$$ | $id\$$ | match $*$ |
| $id + id *$ | $id\,T'E'\$$ | $id\$$ | output $F \to id$ |
| $id + id * id$ | $T'E'\$$ | $\$$ | match $id$ |
| $id + id * id$ | $E'\$$ | $\$$ | output $T' \to \epsilon$ |
| $id + id * id$ | $\$$ | $\$$ | " $E' \to \epsilon$ |

moves made by a predictive parser on input $id + id * id$

# Bottom-Up Parsing -

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).

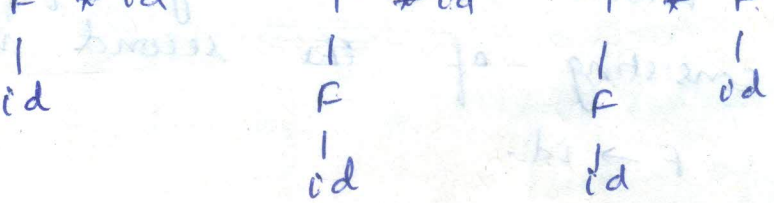
Ex- $E \rightarrow E+T \mid T$

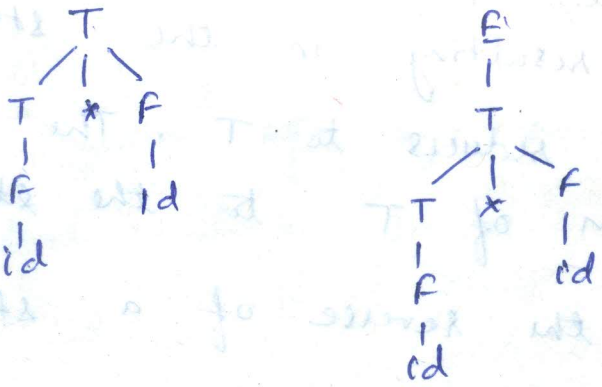
$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Derive parse tree for the token stream id * id.

id * id




A bottom-up parse for id * id

## Reductions :-

- Bottom-up parsing is a process of reducing a string w to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.

According to the previous example reduction is performed as below -

id * id , F * id , T * id , T * F , T, E

leaves (bottom) root (top)

- The sequence starts with the input string id * id.
- The first reduction produces F * id by reducing the leftmost id to F, using the production F → id.
- The second reduction produces T * id by reducing F to T.
- Now there is choice between reducing the string T, which is the body of E → T, and the string consisting of the second id, which is the body of F → id.
- Rather than reduce T to E, the second id is reduced to F, resulting in the string T * F.
- This string then reduces to T. The parse completes with the reduction of T to the start symbol E.
- A reduction is the reverse of a step in derivation.

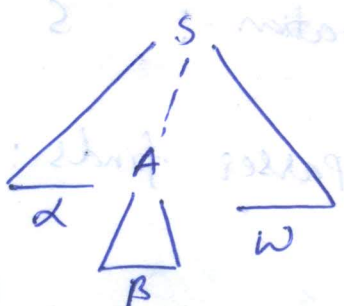
## Handle Pruning →

- Bottom-up parsing during a left to right scan of the input constructs a rightmost derivation in reverse.
- Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation.

| Right sentential Form | Handle | Reducing Production |
|---|---|---|
| $id_1 * id_2$ | $id_1$ | $F \to id$ |
| $F * id_2$ | $F$ | $T \to F$ |
| $T * id_2$ | $id_2$ | $F \to id$ |
| $T * F$ | $T * F$ | $T \to T * F$ |
| $T$ | $T$ | $E \to T$ |

Handles during a parse of $id_1 * id_2$

- Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha \beta w$, as in next figure then production $A \to \beta$ in the position following $\alpha$ is a handle of $\alpha\beta w$.
- Alternatively, a handle of a right sentential form $\gamma$ is a production $A \to \beta$ and a position of $\gamma$ where the string $\beta$ may be found such that replacing $\beta$ at that position by $A$ produces the previous right sentential form in a rightmost derivation of $\gamma$.
- The string $w$ to the right of the handle must contain only terminal symbols.
- For convenience, we refer to the body $\beta$ rather than $A \to \beta$ as a handle.




A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

- A rightmost derivation in reverse can be obtained by "handle pruning".
- That is, we start with a string of terminals w to be parsed.
- If w is a sentence of the grammar at hand, then let $w = \gamma_n$, where $\gamma_n$ is the nth right sentential form of some as yet unknown rightmost derivation.

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

## Shift - Reduce Parsing

- A shift reduce parser tries to reduce the given input string into the starting symbol.
- At each reduction step, a substring of the input matching to right side of a production rule is replaced by the non terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation : $S \underset{rm}{\overset{*}{\Rightarrow}} w$

shift - Reduce Parser finds : $w \underset{rm}{\Leftarrow} \cdots \Leftarrow S$

- We use $ to mark the bottom of the stack and also the right end of the input.

| stack | Input |
|---|---|
| $ | w $ |

- During a left to right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
- It then reduces β to the head of the appropriate production.
- Parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

Ex. $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Shift Reduce Parser on input $id_1 * id_2$

| stack | Input | Action |
|---|---|---|
| $ | $id_1 * id_2$ $ | shift |
| $ $id_1$ | $* id_2$ $ | Reduce by $F \to id$ |
| $ F | $* id_2$ $ | Reduce by $T \to F$ |
| $ T | $* id_2$ $ | shift |
| $ T * | $id_2$ $ | shift |
| $ T * $id_2$ | $ | Reduce by $F \to id$ |
| $ T * F | $ | Reduce by $T \to T*F$ |
| $ T | $ | Reduce by $E \to T$ |
| $ E | $ | Accept |

There are four possible actions

1. Shift - Shift the next input symbol onto the top of the stack.
2. Reduce - The right end of the string to be reduced must be at the top of the stack.
   - Locate the left end of the string with on the stack and decide with what nonterminal to replace the string.
3. Accept - Announce successful completion of parsing
4. Error - Discover a syntax error and call on error recovery routine.

- Initial stack just contains only the end-marker $ and also end of the input string is marked by the end-marker $.

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Right-most Derivation of id + id * id

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$
$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id$
$\Rightarrow id+id*id$

Right most sentential form

<u>id</u> + id * id
<u>F</u> + id * id
<u>T</u> + id * id
E + <u>id</u> * id
E + <u>F</u> * id
E + T * <u>id</u>
E + <u>T * F</u>
<u>E + T</u>
E

Reducing Production

$F \to id$
$T \to F$
$E \to T$
$F \to id$
$T \to F$
$F \to id$
$T \to T*F$
$E \to E+T$

* Handles are underlined

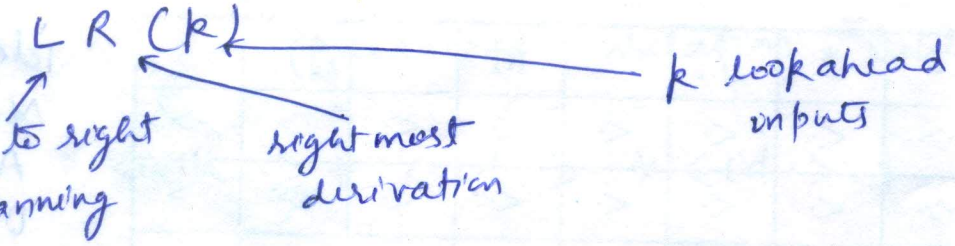
# Conflicts During Shift-Reduce Parsing

- There are context free grammars for which shift-reduce parsers can not be used.
- Stack contents and the next input symbol may not decide action.

– Shift/Reduce conflict – Whether make a shift operation or a reduction.

– Reduce/reduce conflict – The parser can not decide which of several reductions to make

- If a shift reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar



# Operator Precedence Parser

A grammar $G_1$ is said to be operator precedence if it posses following properties -

1. No production on the right side is $\epsilon$.
2. There should not be any production rule possessing two adjacent non-terminals at the right hand side.

Ex- $E \rightarrow EAE \mid (E) \mid -E \mid id$

This grammar is not an operator precedent grammar as in the production rule -

$E \rightarrow EAE$ ; it contains two consecutive non terminals.

## Operator-Precedence Relations

| | + | - | * | / | ^ | id | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|---|
| + | $\cdot>$ | $\cdot>$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $\cdot>$ | $\cdot>$ |
| - | $\cdot>$ | $\cdot>$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $\cdot>$ | $\cdot>$ |
| * | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $\cdot>$ | $\cdot>$ |
| / | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $\cdot>$ | $\cdot>$ |
| ^ | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $\cdot>$ | $\cdot>$ |
| id | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | | | $\cdot>$ | $\cdot>$ |
| ( | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $=$ | |
| ) | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | $\cdot>$ | | | $\cdot>$ | $\cdot>$ |
| $ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | $<\cdot$ | | |

identifier.
NOT ~
AND ∧
OR ∨
( Open Bracket
* ↑ / ↑
+ −
) Close Bracket
$

In operator precedence parsing we will first define precedence relation $<\cdot$, $=$, and $\cdot>$ between pair of terminals.

$p <\cdot q$ → p gives more precedence than q.

$p = q$ → p has same precedence as q.

$p \cdot> q$ → p takes precedence over q.

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E$$

$$E \rightarrow (E) \mid -E \mid id$$

Now consider the string -

id + id * id

- we will insert $ symbols at the start and end of the input string.
- we will also insert precedence operator by referring the precedence relation table.

$\$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot> \$$

We will follow following steps to parse the given string -

1) scan the input from left to right until first $\cdot>$ is encountered.
2) scan backwards over $=$ until $<\cdot$ is encountered
3) The handle is a string between $<\cdot$ and $\cdot>$.

The parsing can be done as follows.

$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$

Handle id is obtained between $\lessdot \cdot \gtrdot$ Reduce this by $E \to id$

$E + \lessdot id \gtrdot * \lessdot id \gtrdot \$$

Handle id is obtained b/w $\lessdot \cdot \gtrdot$ Reduce this by $E \to id$

$E + E * \lessdot id \gtrdot \$$

Handle id is obtained b/w $\lessdot \cdot \gtrdot$ Reduce this by $E \to id$

$E + E * E$

Remove all the non-terminals

$+ *$

Insert $ at the beginning at the end also insert the precedence operator

$\$ \lessdot + * \lessdot * \cdot \gtrdot \$$

The * operator is surrounded by $\lessdot \cdot \gtrdot$ This indicates that * becomes handle. That means we have to reduce $E * E$ operation first

$\$ \lessdot + \cdot \gtrdot \$$

Now + becomes handle. Hence we evaluate $E + E$

$\$ \quad \$$

Parsing is done.

# How to create operator Precedence Relations

Step 1. check whether grammar is operator grammar or not

Step 2. Calculate the leading and trailing for every non terminal involved in given grammar.

1. For any production like $X \rightarrow YaZ$

   Leading (X) = {a} if Y is $\epsilon$ or a single non terminal

2. For any production like $X \rightarrow B\alpha$

   Leading (X) = Leading (B)

3. For any production like $X \rightarrow YaZ$

   Trailing (X) = {a} if Z is $\epsilon$ or a single non terminal

4. For any production like $X \rightarrow \alpha B$

   Trailing (X) = Trailing (B)

5. For any production like $A \rightarrow B$

   Leading (A) = Leading (B), Trailing (A) = Trailing (B)

6. For any production like $A \rightarrow a$

   Leading (A) = {a}, Trailing (A) = {a}

$E \to E+T \Rightarrow L(E) = +$ (Rule-1)
$T(E) = +$ (Rule-3)

$E \to T \Rightarrow L(E) = L(T)$ (Rule-5)
$T(E) = T(T)$

$T \to T*F \Rightarrow L(T) = *$ (Rule-1)
$T(T) = *$ (Rule-3)

$T \to F \Rightarrow L(T) = L(F)$ (Rule-5)
$T(T) = T(F)$

$F \to (E) \Rightarrow L(F) = ($ (Rule-1)
$T(F) = )$ (Rule-3)

$F \to id \Rightarrow L(F) = T(F) = id$ (Rule-6)

| Non-Terminal | E | T | F |
|---|---|---|---|
| Leading (L) | +, *, (, id | *, (, id | (, id |
| Trailing (T) | +, *, ), id | *, ), id | ), id |

Step 3 - Now establish the precedence relation between to terminals as follows -

(i) $a \doteq b \Rightarrow$ if on RHS of any production $X \to \alpha a \beta b \delta$ where $\beta$ is nothing or single non terminal

(ii) $a \lessdot b$ : if b is on Lead (B) where B is immediate

right non terminal of a in any production,

$A \to \alpha a B \beta$ a(row) $\lessdot$ { all lead of B } (column)

3. a $\gtrdot$ b : if a is in Last(B) where B is Immediate left non terminal of b in any production,

$A \to \alpha B b \beta$ { all trailing of B } $\gtrdot$ b

4. \$ $\lessdot$ { all lead of S }

5. { all trailing of S } $\gtrdot$ \$

| | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|
| + | $\gtrdot$ | $\lessdot$ | $\lessdot$ | $\gtrdot$ | $\lessdot$ | $\gtrdot$ |
| * | $\gtrdot$ | $\gtrdot$ | $\lessdot$ | $\gtrdot$ | $\lessdot$ | $\gtrdot$ |
| ( | $\lessdot$ | $\lessdot$ | $\lessdot$ | $\doteq$ | $\lessdot$ | |
| ) | $\gtrdot$ | $\gtrdot$ | | $\gtrdot$ | | $\gtrdot$ |
| id | $\gtrdot$ | $\gtrdot$ | | $\gtrdot$ | | $\gtrdot$ |
| $ | $\lessdot$ | $\lessdot$ | $\lessdot$ | | $\lessdot$ | |

$E \to E+T$ rule-2

$E \to T$ rule-3

$T \to T*F$ rule-2

$T \to F$ rule-3

$F \to (E)$ rule-1 & 2

$F \to id$ rule-3

rule-4

rule-5

| Non-Terminal | E | T | F |
|---|---|---|---|
| Leading (L) | +, *, (, id | *, (, id | (, id |
| Trailing (T) | +, *, ), id | *, ), id | ), id |

# Disadvantages of Operator Precedence Parsing

- **Disadvantages:**
  - It is difficult to handle operators (like - unary minus) which have different precedence (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.
- **Advantages:**
  - Simple and Easy to implement
  - powerful enough for expressions in programming languages
  - Can be constructed by hand after understanding the grammar.
  - Simple to debug

---

## LR Parser

- An LR parser is a parser for context free grammars that reads input from left to right and produces a right most derivation.
- The term LR(k) parser is also used where
  - 'L' means scanning the input from left to right.
  - 'R' means conducting a right most derivation in reverse.
  - 'k' refers to the number of unconsumed 'lookahead' input symbols that are used in making parsing decisions.

## Properties of LR parser –

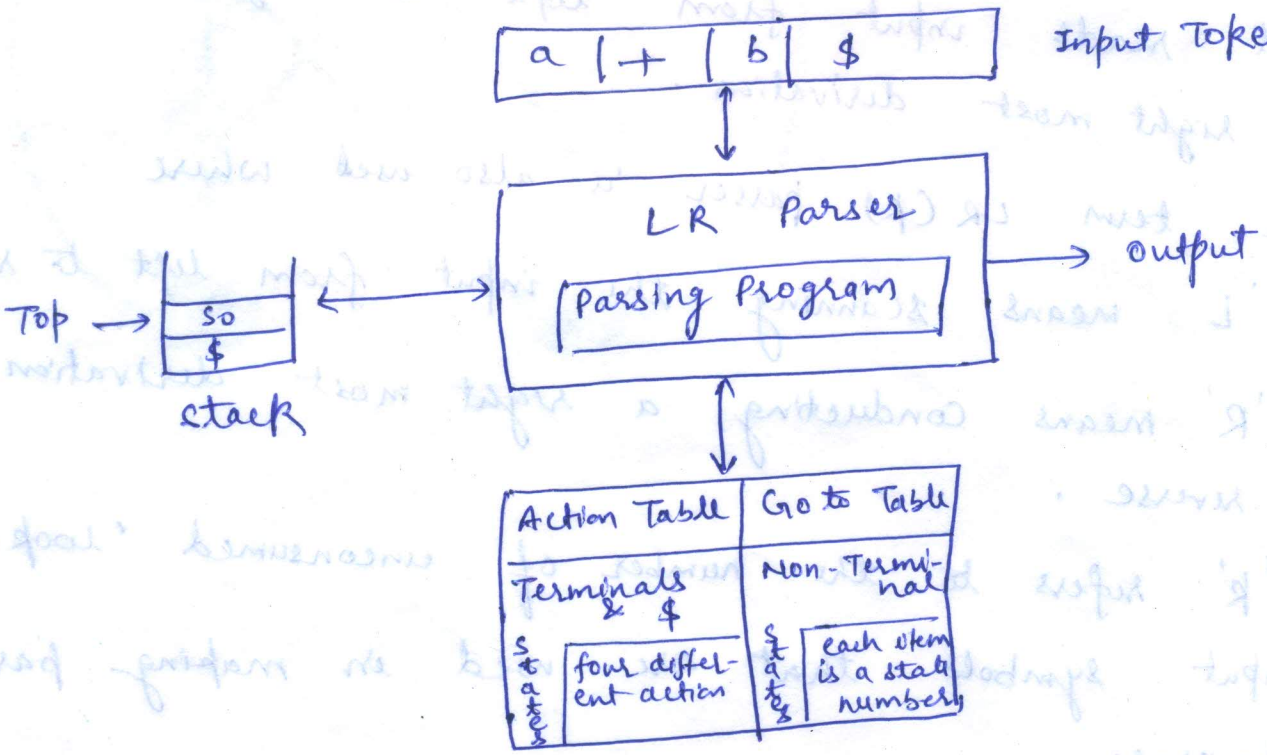
(i) LR parsers can be constructed to recognize most of the programming languages for which context free grammar can be written.

(ii) The class of grammar that can be passed by LR parser is a superset of class of grammars that can be passed using predictive parsers.

(iii) LR parser works using non backtracking shift reduce technique yet it is efficient one.

- LR parsers detect syntactical errors very efficiently.

## Structure of LR parser →

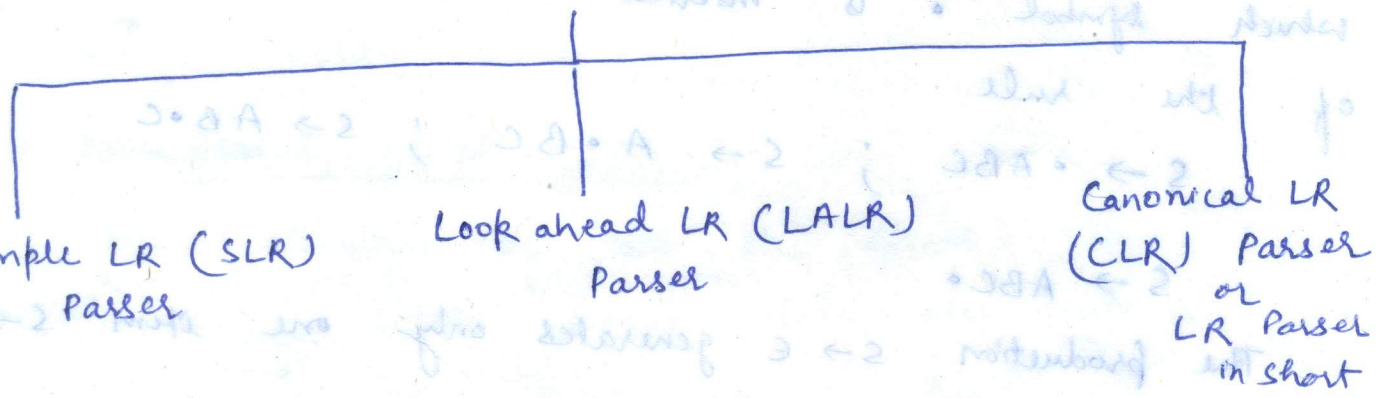



Structure of LR parser

The driving program works on following line –

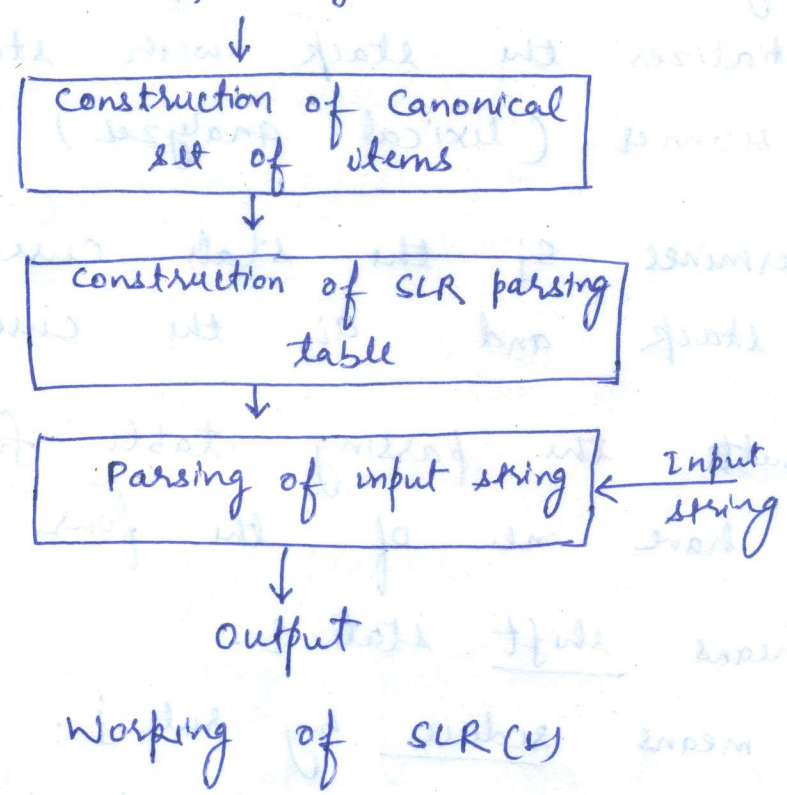
1. It initializes the stack with start symbol and invokes scanner (lexical analyzer) to get next token.
2. It determines $s_j$ the state currently on the top of the stack and $a_i$ the current input symbol.
3. It consults the parsing table for the action $[s_j, a_i]$ which can have one of the four values.
   (i) $S_i$ means shift state $i$.
   (ii) $r_j$ means reduce by rule $j$.
   (iii) Accept means successful parsing is done
   (iv) Error indicates syntactical error.

## LR Parser




## Simple LR (SLR) Parser

- It is the weakest of the three methods but it is easiest to implement.
- A grammar for which SLR parser can be constructed is called SLR grammar.




Working of SLR(1)

## Definition of LR(0) items and related terms →

(i) The **LR(0) item** for grammar $G_1$ is production rule in which symbol • is inserted at some position in R.H.S. of the rule.

$S \rightarrow \bullet ABC$ ; $S \rightarrow A \bullet BC$ ; $S \rightarrow AB \bullet C$

$S \rightarrow ABC \bullet$

The production $S \rightarrow \epsilon$ generates only one item $S \rightarrow \bullet$.

(ii) **Augmented grammar :—**

• If a grammar $G_1$ is having start symbol S then augmented grammar is a new grammar $G_1'$ in which S' is the new start symbol such that $S' \rightarrow S$.

- The purpose of this grammar is to indicate the acceptance of input i.e. when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.

(iii) Non Kernel items :→

- It is collection of items in which • are at the left end of R.H.S. of the rule.

(iv) Functions closure and go to :→

- These are two important functions required to create collection of canonical set of items.

(v) Viable prefix :→

- It is the set of prefixes in the right sentential form of production $A \rightarrow \alpha$.
- This set can appear on the stack during shift/reduce action.

(vi) Kernel items :→

- The collection of the items $S' \rightarrow \cdot S$ and all the items whose dots are not at the left most end of R.H.S. of the rule.

Closure operation →

For a context free grammar G, If I is the set of items then the function closure (I) can be constructed using these rules -

1. Consider I is a set of canonical items and initially every item I is added to closure (I).

2. If rule $A \rightarrow \alpha \cdot B\beta$ is a rule in closure (I) and there is another rule for B such as $B \rightarrow \gamma$ then

closure (I): $A \rightarrow \alpha \cdot B\beta$

$B \rightarrow \cdot\gamma$

This rule has to be applied until no more new items can be added to closure (I).

- The meaning of rule $A \rightarrow \alpha \cdot B\beta$ is that during derivation of the input string [redacted] at some point we may require strings [redacted] derivable from $B\beta$ as input.
- A nonterminal immediately to the right of • indicates that it has to be expanded shortly.

## Goto operation -

- If there is a production $A \rightarrow \alpha \cdot B\beta$ then

  goto $(A \rightarrow \alpha \cdot B\beta, B) = A \rightarrow \alpha B \cdot \beta$.

- That means simply shifting of • one position ahead over the grammar symbol (may be terminal or non terminal)
- The rule $A \rightarrow \alpha \cdot B\beta$ is in I then the same goto function can be written as goto (I, B).

Ex. $X \rightarrow Xb \mid a$

compute closure (I) and goto (I)

Let $I: X \rightarrow \cdot Xb$

closure (I) $= X \rightarrow \cdot Xb$

$= X \rightarrow \cdot a$

goto function can be computed as

$X \rightarrow \cdot X b \Rightarrow X \rightarrow X \cdot b$

$X \rightarrow \cdot a \Rightarrow X \rightarrow a \cdot$

goto $(I, X) = X \rightarrow X \cdot b$

goto $(I, a) = X \rightarrow a \cdot$

Ex - Consider the following grammar -

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

(a) Write the augmented grammar corresponding to the given grammar

(b) If $I = \{[E' \rightarrow \cdot E]\}$ then find closure (I)

(c) If $I = \{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$ then find goto $(I, +)$.

(a) augmented grammar

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

(b) $I = \{[E' \rightarrow \cdot E]\}$

closure (I) :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E+T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

given that $I = \{[E' \to E.], [E \to E. + T]\}$

Goto $(I_1, +)$ = Closure $([E \to E + . T])$

therefore Goto $(I_1, +)$ consists of

$E' \to E + . T$

$T \to . T * f$

$T \to . f$

$f \to . (E)$

$f \to . id$

## Construction of Canonical Collection of LR(0) items :→

- For the grammar G initially add $S' \to . S$ in the set of item C(closure).
- For each set of items $I_i$ in C and for each grammar symbol X (may be terminal or non-terminal) add closure $(I_i, X)$.
- Above process should be repeated by applying goto $(I_i, X)$ for each X in $I_i$ such that goto $(I_i, X)$ is not empty and not in C.
- Set of items has to constructed until no more set of items can be added to C.

Ex.
$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

- In this grammar we will add the augmented grammar $E' \rightarrow \cdot E$ in the I then we have to apply closure(1).

| $I_0$: |
|---|
| $E' \rightarrow \cdot E$ |
| $E \rightarrow \cdot E+T$ |
| $E \rightarrow \cdot T$ |
| $T \rightarrow \cdot T*F$ |
| $T \rightarrow \cdot F$ |
| $F \rightarrow \cdot (E)$ |
| $F \rightarrow \cdot id$ |

→ The item $I_0$ is constructed starting from the grammar $E' \rightarrow \cdot E$.
→ Now immediately right to $\cdot$ is $E$. Hence we have applied closure ($I_0$) and thereby we add $E$-productions with $\cdot$ at the left end of the rule.
→ That means we have added $E \rightarrow \cdot E+T$ and $E \rightarrow \cdot T$ in $I_0$.

→ But again the rule $E \rightarrow \cdot T$ which we have added contains non terminal T immediately right to $\cdot$.
→ So we have added T- productions in $I_0$.

$T \rightarrow \cdot T*F$ and $T \rightarrow \cdot F$

→ In T- productions after $\cdot$ comes T and F respectively.
- But since we have already added T productions so we will not add those.
- But we will add add the F-productions having dots.

- The $F \to \cdot(E)$ and $F \to \cdot id$ will be added.
- Now after dot '(' and 'id' are coming in these two productions.
- The ( and id are terminal symbols and are not deriving any rule. Hence our closure function terminates over here.
- Since there is no rule further we will stop creating $I_0$.

Now apply goto $(I_0, E)$

$E' \to \cdot E$ — shift dot to right → $E' \to E\cdot$

$E \to \cdot E + T$ — shift dot to right → $E \to E\cdot + T$

Thus $I_1$ becomes,

goto $(I_0, E)$

$I_1$ : $E' \to E\cdot$

$E \to E\cdot + T$

Since in $I_1$, there is no nonterminal after dot we can not apply closure $(I_1)$.

By applying goto on T of $I_0$

goto $(I_0, T)$

$I_2$ : $E \to T\cdot$

$T \to T\cdot * F$

Since on $I_2$ there is no nonterminal after dot we can not apply closure $(I_2)$.

By applying goto on F of $I_0$

goto $(I_0, F)$

$I_3$ : $T \to F\cdot$

since after dot in $I_3$ there is nothing, hence we can not apply closure ($I_3$).

- By applying goto on C of $I_0$ but after dot E comes hence we will apply closure on E, then on T, then on F.

goto ($I_0$, C)

$I_4$: $T \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

$\Rightarrow$

By applying goto on id of $I_0$

goto ($I_0$, id)
$I_5$: $F \to id \cdot$

since in $I_5$ there is no non-terminal to the right of dot we can not apply closure here.

- Thus we have completed applying gotos on $I_0$.

→ We will consider $I_1$ for applying goto.

- In $I_2$ there are two productions $E' \to E \cdot$ and $E \to E \cdot + T$
- There is no point applying goto on $E' \to E \cdot$, hence we will consider $E' \to E \cdot + T$ for applying of goto

goto ($I_1$, +)
$I_6$: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

There is no other rule in $I_1$ for applying goto.

- So we will move to $I_2$. In $I_2$ there are two productions: $E \to T\bullet$ and $T \to T\bullet * F$.
- We will apply goto on *.

goto ($I_2$, *)

$I_7$: $T \to T * \bullet F$

$F \to \bullet (E)$

$F \to \bullet id$

- The goto cannot be applied on $I_3$. Apply goto on E in $I_4$. In $I_4$ there are two productions having E after dot ($F \to \bullet E$ and $E \to \bullet E + T$).
- Hence we will apply goto on both of these productions. The $I_8$ becomes -

goto ($I_4$, E)

$I_8$: $F \to (E\bullet)$

$E \to E\bullet + T$

- If we will apply goto on ($I_4$, T) but we get $E \to T\bullet$ and $T \to T\bullet * F$ which is $I_2$ only. Hence we will not apply goto on T.
- Similarly we will not apply goto on F, ( and id as we get the states $I_3$, $I_4$, $I_5$ again.