UNIT-II : SYNTAX ANALYSIS

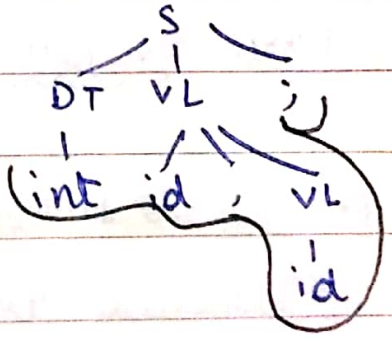
Syntax Analysis

- Programming language constructs can be represented in CFG form or BNF form.
- CFG

$G = (V, T, P, S)$; $V = S, DT, VL$; $T = int, float, ,, ;, id$; $S = S$

$S \rightarrow DT\ VL;$
$DT \rightarrow int / float;$ } P
$VL \rightarrow id / id, VL;$

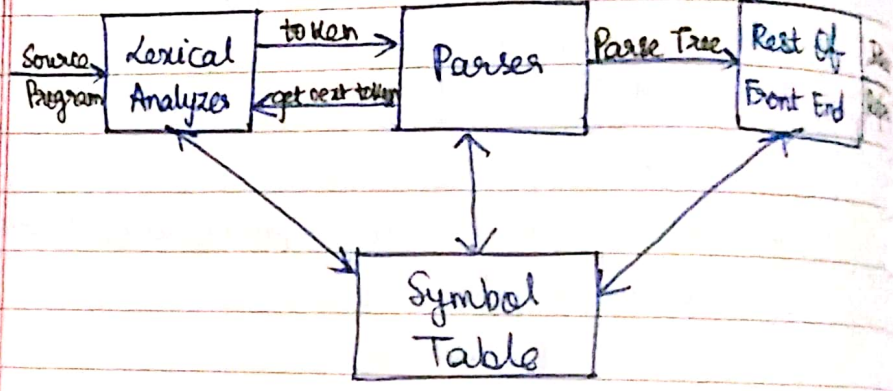
- Syntax Analysis has two types of algorithms:
  i) Top-Down Parsing
  ii) Bottom-Up Parsing
- Eg. int id, id; is valid or not?
  → Top-Down Parsing:
  Start with Start symbol and check if we can reach/derive string/sentence



  → Bottom-Up Parsing:
  Start with string and check if we can reach start symbol.
- Two algorithms can be used in top-down parsing:
  i) Predictive Parsing
  ii) Recursive Present Parser

- Parser Generator
  - It is used to generate parsers automatically
  - Generated by using YAAC tool.

The Role Of Parser



Error Detection And Handling

- The different types of errors are.
  i) Lexical Errors
    - Identified in lexical analysis phase.
    (Eg Not able to identify exact token, eg id)
  ii) Syntax Errors
    - An error in the syntax.
    (Eg Missing ; and if ( ), etc.)
  iii) Semantic Errors
    - Eg Type mismatch, type declaration
  iv) Logical Errors
    - Very difficult to find, must analyze syntax errors but expected output is
    (Eg =] for ==)

Context Free Grammar

- Left Recursive Grammar
  - First symbol in LHS and RHS is same, then this grammar is called left recursive grammar.
- Precondition for top down approach is that the grammar must not be left recursive.

Error Recovery Strategies

- Panic Mode Recovery
  - Discard input symbol one at a time until one of designated set of tokens is found.
  (Eg id ++ id ++ id (OR) id+/id+/id (or) id // id)
- Phrase Level Recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue.
- Error Productions
  - Augment the grammar with productions that generate the erroneous constructs.
- Global Correction
  - Choosing minimal sequence of chages to obtain a globally least cost corrections.
  - Only theoretical concept, it is not used practically.

→

Derivations

- Productions are treated as rewriting rules to generate a string.
- Rightmost and leftmost generations.

$E \to E+E$, $E \to E*E$, $E \to -E$, $E \to (E)$, $E \to id$

- The string is $-(id+id)$
- Leftmost: $E \to -E \to -(E)+E \to -(E+E)$
  $\to -(id+E) \to -(id+id)$
- Rightmost: $E \to -E \to -(E) \to -(E+\overset{E}{id})$
  $\to -(E+id) \to -(id+id)$

Elimination of Left Recursion - Immediate Left Recursion

- A grammar is left recursive if it has a non-terminal and such that there is a derivation: $A \overset{+}{\Rightarrow} A\alpha$
- Top down parsing methods can't handle left recursive grammars.
- A simple rule for direct left recursion elimination:
  - For a rule like $A \to A\alpha \mid \beta$
  - we may replace it with $A \to \beta A'$, $A' \to \alpha A' \mid \epsilon$
- Note: $A^+ \Rightarrow A\alpha \Rightarrow A \to A\alpha \Rightarrow A \to AA\alpha \Rightarrow A \to AAA\alpha \Rightarrow \dots$
- Eg: $E \longrightarrow E+T \mid T$
  $A \longrightarrow A\alpha \mid \beta$

Here: $A = E$, $\alpha = +T$, $\beta = T$

$A \to \beta A' \Rightarrow E \to TE'$

$A' \to \alpha A' / \epsilon \Rightarrow E' \to +TE' / \epsilon$

↳ Eliminates Left Recursion

- eg: $T \rightarrow T * F \mid E$
  $A \rightarrow A\alpha \mid \beta$
  Here $A = T$, $\alpha = *F$, $\beta = EF$
  $A \rightarrow \beta A' \Rightarrow T \rightarrow FT'$
  $A' \rightarrow \alpha A' / \epsilon \Rightarrow T' \rightarrow *FT' / \epsilon$
  ↳ Elimination of Left Recursion

Left Factoring

- Left Factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing
- Consider the following grammar:
  - stmt → if expr then stmt else stmt | if expr then stmt
- On seeing input if it is not clear for the parser which one to use.
- We can easily perform left factoring:
  - If we have $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ then we replace it with:
    - → $A \rightarrow \alpha A'$
    - → $A' \rightarrow \beta_1 \mid \beta_2$
- For the above example:
  $S \rightarrow$ if expr then stmt S'
  $S' \rightarrow$ else stmt $/ \epsilon$

Elimination of Left Recursion - Continuation

$S \rightarrow (L) / a$
$L \rightarrow L,S / S$
$\Rightarrow A \rightarrow A\alpha / \beta$
$A = L, \alpha = ,S, \beta = S$

$L \rightarrow SL'$
$L' \rightarrow ,SL' / \epsilon$

Grammar:
$S \rightarrow (L) / a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' / \epsilon$

Eg: $E \to E+E/T$, $T \to TF/F$ $F \to F*/a$

$\to E \to E+E/T \to T \to TF/F \to F \to F*/a$

$E \to TE'$ $T \to FT'$ $F \to aF'$

$E' \to +TE'/\epsilon$ $T' \to FT'/\epsilon$ $F' \to *F'/\epsilon$

Grammar:

$E \to TE'$ $T \to FT'$ $F \to aF'$

$E' \to +TE'/\epsilon$ $T' \to FT'/\epsilon$ $F' \to *F'/\epsilon$

28-32-28

→ For any number of A-productions - Immediate left recursion

$A \to A\alpha_1/A\alpha_2/\ldots\ldots/A\alpha_m/\beta_1/\beta_2/\ldots/\beta_n$

where $\alpha \in (V \cup T)^*$

Then:

$A \to \beta_1 A'/\beta_2 A'/\ldots\ldots/\beta_n A'$

$A' \to \alpha_1 A'/\alpha_2 A'/\ldots\ldots/\alpha_m A'/\epsilon$

→ Indirect Left Recursion

$S \to Ab/d$ → Indirect Left Recursion

$A \to Aa/Sac/\epsilon$

$\Rightarrow S \Rightarrow Ab \Rightarrow Sacb$

NOTE:

The above formulae for any number of A productions and 2 number of A productions are useful only for immediate left recursion. Indirect left Recursion cannot be eliminated using the above formulae.

Eg $A \longrightarrow ABd / Aa / a / b \quad B \longrightarrow Be / b$

$\rightarrow A \longrightarrow ABd / Aa / a / b \qquad B \longrightarrow Be / b$

$A \longrightarrow A\alpha_1 / A\alpha_2 / \beta_1 / \beta_2 \qquad A \longrightarrow A\alpha / \beta$

$\Rightarrow A \longrightarrow aA' / bA' \qquad \Rightarrow B \longrightarrow eB'$

$A' \longrightarrow BdA' / aA' / \epsilon \qquad B' \longrightarrow eB' / \epsilon$

$\therefore$ Grammar: $A \longrightarrow aA' / bA' \qquad B \longrightarrow eB'$

$A' \longrightarrow BdA' / aA' / \epsilon \qquad B' \longrightarrow eB' / \epsilon$

Eg $F \longrightarrow F* / a / b$

$A \longrightarrow A\alpha / \beta_1 / \beta_2$

$F \longrightarrow aF' / bF' \qquad F' \longrightarrow *F' / \epsilon$

$\therefore$ Grammar: $F \longrightarrow aF' / bF' \qquad F' \longrightarrow *F' / \epsilon$

Generalized Algorithm for Eliminating Left Recursion (both immediate and indirect)

Assume that the grammar has no cycles $A \overset{+}{\Longrightarrow} A$

Algorithm:

i/p: Grammar G with no cycles.

o/p: Equivalent grammar G with no left recursion.

Method: (i) Arrange the non-terminals in some order $A_1, A_2, A_3, \ldots\ldots, A_n$

(ii) for (each i from 1 to n) {

for (each j from 1 to i-1) {

replace each production of the form $A_i \longrightarrow A_j\gamma$ by the productions $A_i \longrightarrow \delta_1\gamma / \delta_2\gamma / \ldots\ldots / \delta_k\gamma$ where, $A_j \longrightarrow \delta_1 / \delta_2 / \ldots\ldots / \delta_k$ are all current A_j productions }

$A_1 \rightarrow A_2 a / b$

$A_2 \rightarrow A_2 c / A_1 d / \epsilon$

* Loop:

$i = 1$ $\quad$ if $A_1 \rightarrow A_2 a / b$ (No immediate left recur

$i = 2 \quad j = 1$

$A_1 \rightarrow A_2 a / b \rightarrow$ NO ILR

$A_2 \rightarrow A_2 c / A_1 d / \epsilon$

$A_2 \rightarrow A_2 c / A_2 a d / b d / \epsilon$

$\Downarrow$

$A_2 \rightarrow b d A_2' / A_2'$

$A_2' \rightarrow c A_2' / a d A_2' / \epsilon$

Grammar:

$A_1 \rightarrow A_2 a / b$

$A_2 \rightarrow b d A_2' / A_2'$

$A_2' \rightarrow c A_2' / a d A_2' / \epsilon$

By substituting the values, we get:

$S \rightarrow A a / b$

Left Factoring

- If the prefix of the productions are same, then there arises an ambiguity as to which production to choose.
- Left factoring for two options:

$A \longrightarrow \alpha\beta_1 / \alpha\beta_2$

$\Rightarrow$ $A \rightarrow \alpha A'$

$A' \longrightarrow \beta_1 / \beta_2$

- General Algorithm for Left Factoring

Algorithm: Left Factoring a Grammar

I/P : Grammar G

O/P : Grammar with left factoring eliminated

Method : (i) For each non-terminal A, find the largest prefix α common to two or more of its alternatives.

(ii) If $\alpha \neq \epsilon$, i.e., there is a non-trivial common prefix, replace all the A-productions.

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \ldots\ldots\ldots / \alpha\beta_n / \gamma$

where γ represents all productions that don't begin with 'α'

is replaced as:

$A \longrightarrow \alpha A' / \gamma$

$A' \rightarrow \beta_1 / \beta_2 / \beta_3 / \ldots\ldots\ldots / \beta_n$

where A' is a new non-terminal

(iii) Repeatedly apply this transformation until no two alternatives for a non terminal have a common prefix.

Eg. $S \to iEtS/iEtSeS/a$

$E \to b$

$\Rightarrow S \to iEtSS'/a$

$S' \to eS/\epsilon$

$E \to b$

i = if
E = Expn
t = then
S = stmt
e = else

Eg. $A \to aAB/aA/a$

$B \to bB/b$

$\Rightarrow A \to aA'$

$A' \to AB/A/\epsilon$

$\Downarrow$

$A \to aA'$

$A' \to AA''/\epsilon$

$A'' \to B/\epsilon$

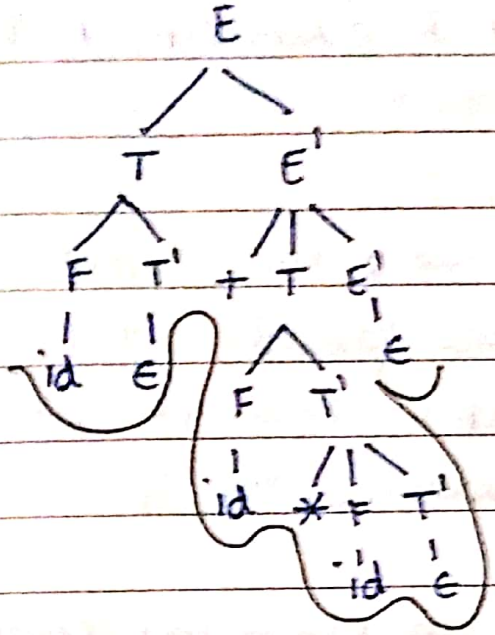
$B \to bB'$

$B' \to B/\epsilon$

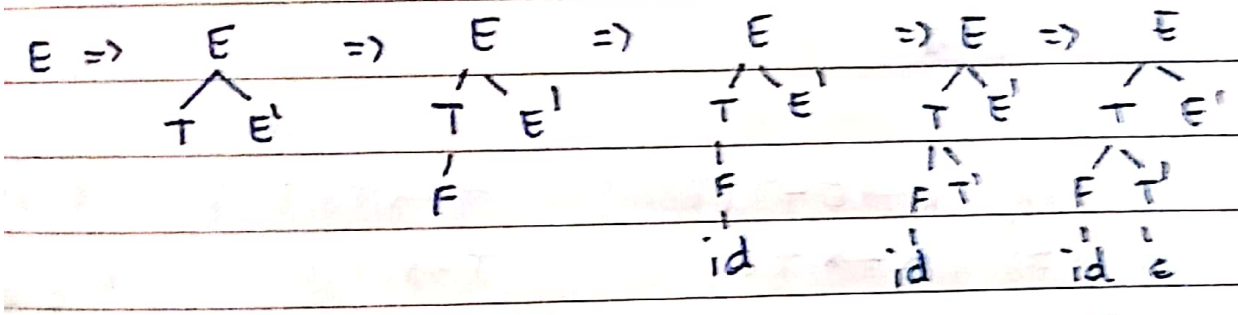
Grammar:

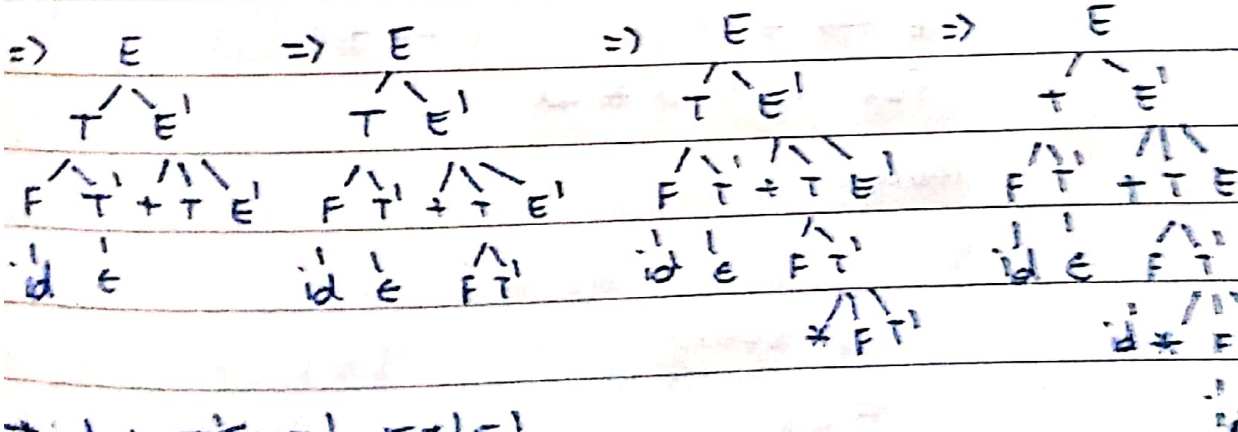
$A \to aA'$ A' ...

id + id * id $



Derivation:





→ id + ~~T * E'~~ ϵ T' E'

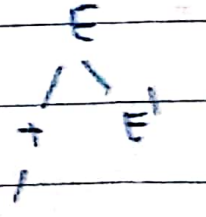
→ id + id T' E'

→ id + id * ~~id~~ F T' E'

→ id + id * id ~~F~~ T' E'

→ id + id * id ϵ E'

→ id + id * id



→ Leftmost Derivation

- Recursive Descent Parsing (RDP)

```
void A()
{
  Choose an A production A → x1 x2 ... xk
  for (i = 1 to k)
  {
    if (xi is a non-terminal)
      call procedure xi();
    else if (xi equals the current i/p symbol a)
      advance the i/p to the next symbol
    else
      /* an error has occured */
  }
}
```

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

$\Rightarrow$ $E \to TE'$ $\quad T \to FT'$ $\quad F \to (E) \mid id$

$E' \to +TE' \mid \epsilon$ $\quad T' \to *FT' \mid \epsilon$

For: id + id * id

```
main()
{
  i/p string
  E()
}
```

```
E()
{
  T()
  E'()
}
```

→

```
E'()
{
  if cp = '+'
     advance();
     T();
     E'();
}
```

```
T()
{
   F();
   T'();
}
```

```
F()
{
 if cp = '('
    advance();
    E();
    if cp = ')'
       advance;
    else
       printf (')' expected);
else if cp = id
    advance();
else
   printf ('identifier expected');
```

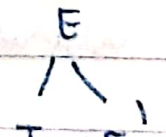
```
T'()
{
 if (cp = '*')
    advance();
    F();
    T'();
 else
    return;
}
```

id * id + id

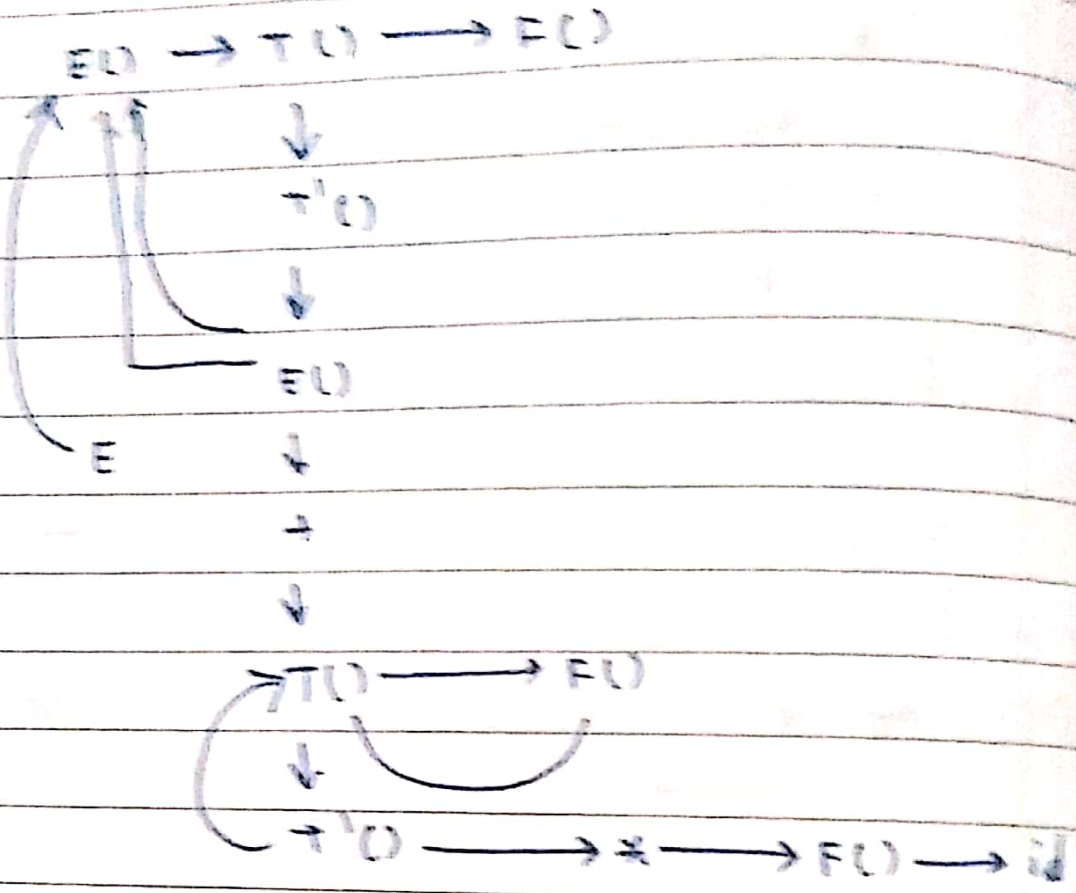
E → E + T | T

T → T * F | F

F → (E) | id



Eg: id + id * id



- Predictive Parsing

- This method is also called as table filling algorithm / table driven algorithm
- Input is scanned from left to right
- The prequisites of this algorithm are that we must eliminate left recursion and left factoring before performing this algorithm
- We make use of two functions called the FIRST() and FOLLOW() functions in this algorithm

FTO

PTO →

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

1) If X is a terminal, then FIRST(X) = {X}

2) If X is a non terminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1) ..... FIRST(Y_{i-1})

that is $Y_1 \cdots Y_{j-1} \Rightarrow \epsilon$

If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ then add ϵ to FIRST(X)

For eg. everything in FIRST(Y_i) is surely in FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on

3) If $X \to \epsilon$ is a production, then add ϵ to FIRST(X)

★ The terminal symbol we get by selecting a partic non-terminal is called FIRST set.

★ The terminals that follow for a particular non-term are placed in the FOLLOW set.

Eg. FIRST(a) = {a}

$S \to aBCA \Rightarrow$ FIRST(S) = {a}

$S \to ABCD \Rightarrow$ Let FIRST(A) = {x, ϵ}, FIRST(B) = {

FIRST(S) = FIRST(ABCD)

= FIRST(A) = {x, ϵ} $\cup$ FIRST(BC

= {x, y}

S → ABCD ⇒ Let FIRST(A) = {x, ∈}, FIRST(B) = {y, ∈}, FIRST(C) = {z, ∈}

FIRST(S) = FIRST(ABCD) = FIRST(A) ∪ FIRST(BCD)

= {x, ∈} ∪ FIRST(B) ∪ FIRST(CD)

= {x} ∪ {y, ∈} ∪ FIRST(C) ∪ FIRST(D)

= {x, y} ∪ {z, ∈} ∪ FIRST(D)

= {x, y, z} ∪ {a, ∈} = {x, y, z, a, ∈}

Eg. Find out the FIRST set for the grammar:

E → E + | T T → T * F | F F → (E) | id

→ After eliminating left recursion and left factoring, we get the productions as:

E → TE' T → FT' F → (E) | id

E' → +TE' | ∈ T' → *FT' | ∈

FIRST(E) = FIRST(TE') = FIRST(T) = FIRST(FT')

= FIRST(F) = {c, id}

∴ FIRST(E) = {c, id} [left parenthesis, id F → (E) | id]

FIRST(T) = FIRST(F) = {c, id}

FIRST(E') = ~~FIRST~~ {+, ∈}

FIRST(T') = {*, ∈}

FIRST(F) = {c, id}

[c, id, +, * = Terminals}

Eg. Find out the FOLLOW set for the grammar:

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

→ After eliminating left recursion and left factoring, we get the productions as:

$E \to TE' \qquad T \to FT' \qquad F \to (E) \mid id$

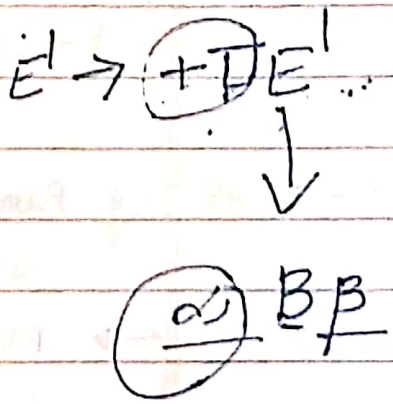
$E' \to +TE' \mid \epsilon \qquad T' \to *FT' \mid \epsilon$

FOLLOW(E):

$F \to (\mid E \mid)$

$A \to \alpha \mid B \mid \beta$

$\therefore$ FOLLOW(E) = {), $}



FOLLOW(E'):

$E' \to +TE' \qquad E \to TE'$

$A \to \alpha B \qquad A \to \alpha B$

$\therefore$ FOLLOW(E') $\therefore$ FOLLOW(E')

= FOLLOW(E) = {), $} = FOLLOW(E) = {), $}

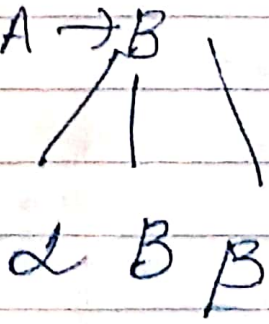
FOLLOW(T):

$E \to TE' \qquad E' \to +TE'$

$A \to \alpha B\beta' \qquad A \to \alpha B\beta'$

$\therefore$ FOLLOW(T)

= FOLLOW(E) = {), $}



FOLLOW(T):

$E' \to +TE' \qquad E \to TE'$

$\alpha B\beta \qquad B\beta$

FOLLOW(T) = FIRST(E') = {+, ε} ∪ FOLLOW(E') = {+,), $}

FOLLOW(T'):

$T \rightarrow FT'$ $\quad T' \rightarrow *FT'$

αB $\quad \alpha B$

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

FOLLOW(F):

$T \rightarrow FT'$ $\quad T' \rightarrow *FT'$

$\alpha B \beta$

FOLLOW(F) = FIRST(T') ∴

= {*, ∈} = {*} ∪ FOLLOW(T) [∵ FIRST(T') contains ∈]

= {*, +,), $}

04-01-'18 Eg. Find out FOLLOW(S), FOLLOW(A), FOLLOW(B)

$S \rightarrow AaAb / BbBa$ ∴ $A \rightarrow \epsilon$ $B \rightarrow \epsilon$

→ FOLLOW(S) = {$}

FOLLOW(A) => $S \rightarrow AaAb$

FOLLOW(A) = FIRST(aAb) ∪ FIRST(b)

= {a} ∪ {b} = {a, b}

FOLLOW(B) => $S \rightarrow BbBa$

FOLLOW(B) = FIRST(bBa) ∪ FIRST(a)

= {b} ∪ {a} = {a, b}

Eg. Find out the FIRST and FOLLOW sets for the grammar:

$S \rightarrow aAB / bA / \epsilon$

$A \rightarrow aAb / \epsilon$

$B \rightarrow bB / \epsilon$

→ S → aAB/bA/ϵ

FIRST(S) = FIRST(aAB) ∪ FIRST(bA) = {a} ∪ {b} = {a, b}

FIRST(A) = FIRST(aAb) = {a, ϵ}

FIRST(B) = FIRST(bB) = {b, ϵ}

FOLLOW(S) = {$}

FOLLOW(A)

=> S → aAB / bA A → aAb

αBβ αB αBβ

- S → aAB ⇒ FIRST(B) = {b, ϵ} ∪ FOLLOW(S) = {b} ∪ {$} = {b, $}
- S → bA ⇒ FOLLOW(S) = {$}
- S → aAb ⇒ FIRST(b) = {b}

∴ FOLLOW(A) = {b, $}

FOLLOW(B)

⇒ S → aAB B → bB S → aAB

αB αB

- S → aAB ⇒ FOLLOW(S) = {$}
- B → bB ⇒ FOLLOW(B) = {$}

∴ FOLLOW(B) = {$}

Eg. Find the FIRST and FOLLOW functions for the grammar:

S → ictSA/a A → eS/ϵ C → b

→ No left recursion and left factoring in the grammar

FIRST(S) = FIRST(ictSA) ∪ FIRST(a) = {i, a}

FIRST(A) = FIRST(eS) = {e, ϵ}

FIRST(C) = FIRST(b) = {b}

FOLLOW(S) = {$} [For start symbol]

⇒ S → ictSA A → eS

α Bβ αB

- S → ictSA ⇒ FIRST(A) = {e, ϵ} ∪ FOLLOW(S) = {e, $}
- A → eS ⇒ FOLLOW(A)

∴ FOLLOW(S) = {e, $} ∪ FOLLOW(A)

FOLLOW(A)

⇒ S → i c t S A
α B

∴ FOLLOW(A) = FOLLOW(S) = {e, $} ∪ FOLLOW(A)

FOLLOW(C)

⇒ S → i E t S A,
α B β

FOLLOW(C) = FIRST(tSA) = {t}

∴ FIRST(S) = {i, a} FOLLOW(S) = {e, $}

FIRST(A) = {e, ∈} FOLLOW(A) = {e, $}

FIRST(C) = {b} FOLLOW(C) = {t}

Eg. Find FIRST and FOLLOW functions for the gramm

S → (L) / a L → L, S / S

→ L → L, S / S

L → SL' } eliminating left recursion S → (L) / a

L' → ,SL' / ∈

FIRST(S) = {(, a}

FIRST(L) = FIRST(S) = {(, a}

FIRST(L') = {,, ∈}

FOLLOW(S) = {$}

⇒ L → SL' L' → ,SL'
αBβ αBβ

• L → SL' ⇒ FOLLOW(S) = FIRST(L') = {), $} ∪ FOLLOW(L)
= {)} ∪ FOLLOW(L)

• L' → ,SL' ⇒ FOLLOW(S) = FIRST(L') = {,,)} ∪ FOLLOW(L')
= {)} ∪ FOLLOW(L')

FOLLOW(S) = {,, $} ∪ FOLLOW(L) ∪ FOLLOW(L')
= {,, $} ∪ {)} ∪ {,,)} = {,, $,)}

FOLLOW(L)

=> S → (L)

αBβ

FOLLOW(L) = FIRST( ) = {)}

FOLLOW(L')

=> L → SL'

αB

L → ,SL'

αB

- L → SL' => FOLLOW(L') = FOLLOW(L) = {)}
- L' → ,SL' => FOLLOW(L') = FOLLOW(L') = {)}

∴ FIRST(S) = {c, a} FOLLOW(S) = {$,),)}

FIRST(L) = {(, a} FOLLOW(L) = {)}

FIRST(L') = {,, ϵ} FOLLOW(L') = {)}

Eg. Find the FIRST and FOLLOW functions for the grammar:

E → E+T | T T → TF | F F → F* | a | b

→ E → E+T | T T → TF | F F → F* | a | b

E → TE' T → FT' F → *F'

E' → +TE' | ϵ T' → FT' | ϵ F' → aF' | bF' | ϵ

↳ We eliminate left recursion from the grammar

- FIRST(E) = FIRST(T) = FIRST(F) = {*}
- FIRST(T) = FIRST(F) = {*}
- FIRST(F) = {*}
- FIRST(E') = {+ , ϵ}
- FIRST(T') = FIRST(F) ∪ {ϵ} = {*, ϵ}
- FIRST(F') = {a, b, ϵ}

FOLLOW(E)

=> E → E + T

αB β

FOLLOW(E) = FIRST(β)

= FIRST(+T) = {+}

08-01-'19

LL(1) Parser

- It is a table driven parser.
- The grammar must not be left recursive grammar or ambiguous grammar.
- Disadvantage of Recursive Descent Parser
  * May enter infinite loop if recursive depth is more.
  * In syntax analysis phase, if we are not able parse the string, it must give error. But recursive descent parser is not good for error messaging
  * Lookahead signals are long.
- To overcome the disadvantages of recursive descent parser, we use LL(1) parser.
- Recursive descent parser uses Brute Force method
- LL(1) parser is a table driven parser which uses dynamic programming.
- LL1 parser:
  1st L: i/p is scanned from left to right
  2nd L: derives left most derivation.
  '1' : no. of look ahead symbols is 1
- Predictive Parsing Table:
  - To construct a table and predict whether the string is
- Before performing predictive parsing, for the given CFGs, we must eliminate left recursion and left factoring, make the grammar unambiguous and find out the FIRST and FOLLOW sets of the grammar

- Algorithm to Construct Predictive Parsing Table:

→ I/P : The Context Free Grammar 'G'

→ O/P : Predictive Parsing Table 'M'

→ Algorithm: For each production $A \to \alpha$ of the grammar, do the following:

i) For each terminal 'a' in FIRST(α), add $A \to \alpha$ to $M(A, \alpha)$

ii) If ϵ is in FIRST(α), then for each terminal 'b' in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$

If ϵ is in FIRST(A) and \$ is in FOLLOW(A), add $M(A, \$)$ as well.

→ If after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to error.

Eg: $E \to E+T/T$ $\quad T \to T*F/F$ $\quad F \to (E)/id$

→ $E \to TE'$ $\quad T \to FT'$ $\quad F \to (E)/id$

$E' \to +TE'/\epsilon$ $\quad T' \to *FT'/\epsilon$

	FIRST()	FOLLOW()
E	{(, id}	{), \$}
E'	{+, ε}	{), \$}
T	{(, id}	{), +, \$}
T'	{+, ε}	{), +, \$}
F	{(, id}	{), +, *, \$}

→ Now, take variables as rows and terminals as columns in the next table

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

- E → TE' FIRST(TE') = FIRST(T) = FIRST(F) = {(, id}
  M[E, (] = E → TE' M[E, id] = E → TE'
- E' → +TE' FIRST(+TE') = {+} M[E', +] = E' → +TE'
- E' → ϵ FOLLOW(E') = {), $}
  M[E',)] = E' → ϵ M[E', $] = E' → ϵ
- T → FT' FIRST(FT') = FIRST(F) = {(, id}
  M[T, (] = T → FT' M[T, id] = T → FT'
- T' → *FT' FIRST(*FT') = {*} M[T, *] = T' → *FT'
- T' → ϵ A → α α ⇒ ϵ ⇒ FOLLOW(T') = {), +, $}
  M[T',)] = T' → ϵ FIRST(A) = FIRST(T') = {*, ϵ}
  M[T', +] = T' → ϵ FOLLOW(A) = FOLLOW(T') = {), +, $}
  M[T', $] = T' → ϵ ⇒ M[T', $] = T' → ϵ
- F → (E) FIRST(F) = {(} M[F, (] = F → (E)
- F → id FIRST(F) = {id} M[F, id] = F → id

All empty blocks/cells are error entries
If for any cell, we get two productions, the grammar is not LL1 grammar and is ambiguous.

Eg $E \to E+T \mid T$ $T \to TF \mid F$ $F \to F^* \mid a \mid b$

→ $E \to TE'$ $T \to FT'$ $F \to *E'aF' \mid bF'$

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

	FIRST	FOLLOW
E	{a, b}	{$}
E'	{+, ε}	{$}
T	{a, b}	{+, $}
T'	{a, b, ε}	{+, $}
F	{a, b}	{+, a, b, $}
F'	{*, ε}	{+, a, b, $}

Chain
a+b*a
a*b+b

	a	b	+	*	$
E	$E \to TE'$	$E \to TE'$			
E'			$E' \to +TE'$		$E' \to \epsilon$
T	$T \to FT'$	$T \to FT'$			
T'	$T' \to FT'$	$T' \to FT'$	$T' \to \epsilon$		$T' \to \epsilon$
F	$F \to aF'$	$F \to bF'$			
F'	$F' \to \epsilon$	$F' \to \epsilon$	$F' \to \epsilon$	$F' \to *F'$	$F' \to \epsilon$

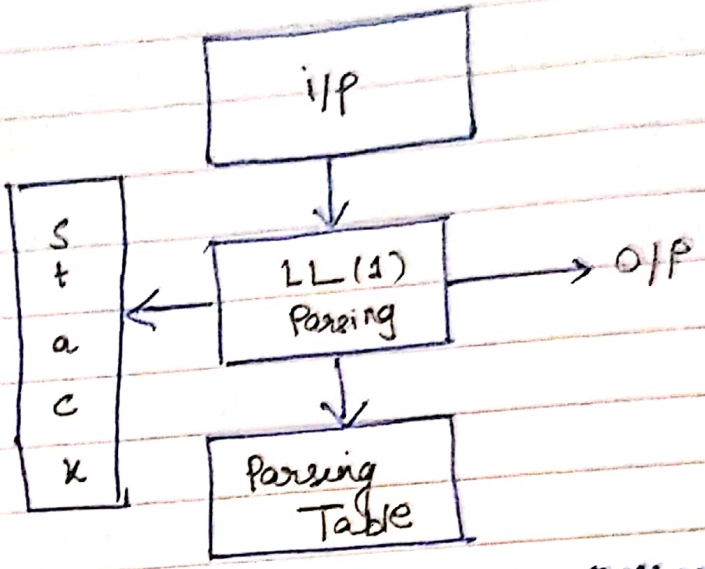
→ The grammar is unambiguous and parsed by LL1

- $E \to TE'$ FIRST(TE') = FIRST(T) = FIRST(F) = {a,b}
  M[E,a] = $E \to TE'$ M[E,b] = $E \to TE'$
- $E' \to +TE'$ FIRST($+TE'$) = {+} M[E', +] = $E' \to +TE$
- $E' \to \epsilon$ $\alpha \Rightarrow \epsilon$ ⇒ FOLLOW(E') = {$,)} M[E', $] = E'
- $T \to FT'$ FIRST(FT') = FIRST(F) = {a, b}
  M[T,a] = $T \to FT'$ M[T, b] = $T \to FT'$
- $T' \to FT'$ FIRST(FT') = FIRST(F) = {a,b}
  M[T',a] = $T' \to FT'$ M[T',b] = $T' \to FT'$
- $T' \to \epsilon$ $\alpha \Rightarrow \epsilon$ ⇒ FOLLOW(T') = {+, $}
  M[T', +] = $T' \to \epsilon$ M[T', $] = $T' \to \epsilon$
- $F \to aF'$ FIRST(aF') = a M[F,a] = $F \to aF'$
- $F \to bF'$ FIRST(bF') = b M[F,b] = $F \to bF'$
- $F' \to *F'$ FIRST($*F'$) = * M[F', *] = $F \to *F'$
- $F' \to \epsilon$ $\alpha \to \epsilon$ ⇒ FOLLOW(F') = {+, a, b, $}
  M[F', a] = $F' \to \epsilon$ M[F', b] = $F' \to \epsilon$ M[F', +] = $F' \to \epsilon$ M[F', $] = $F' \to \epsilon$

Eg. $S \to iCtSA/a$ $A \to eS/\epsilon$ $C \to b$

09-01-'19

- Text Parsing



- LL(1) parser gives better error message compared to recursive descent parser.
- Table Driven Parsing : Algorithm
  - → i/p : A string 'w' and a parsing table 'M' for grammar 'G'
  - → o/p : 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$' - is the input buffer.
    Start symbol S of G is on top of the stack above $.
    Let 'a' be the first symbol of 'w'.
    Let 'X' be the top stack symbol.
    While (X ≠ $) stack is not empty
    {
    if (X == a)
    pop the stack and '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, Y3, ..., Yk)
{   pop the stack;
    push Yk, Yk-1, Yk-2, ..., Y1 onto the stack
    with Y1 on top;
}
let 'X' be the top stack symbol. }

Eg E → E+T | T T → T*F | F F → (E) | id

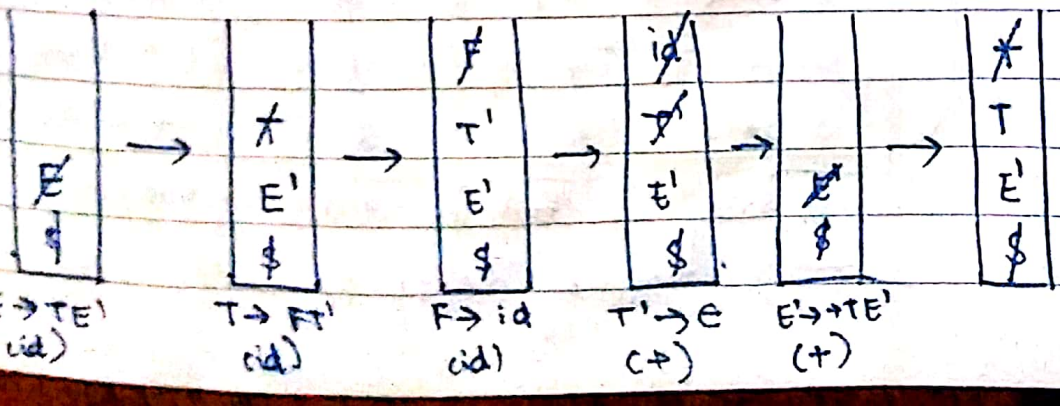
→ Table:

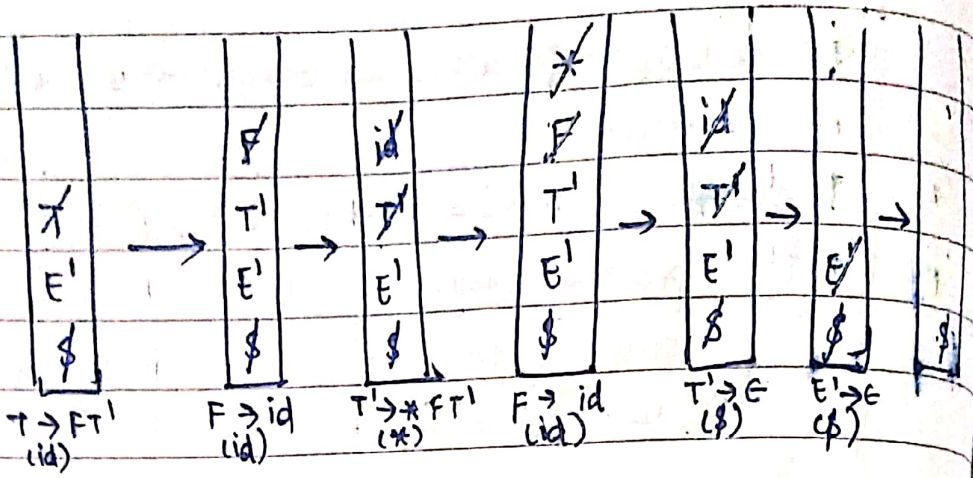
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ϵ	E' → ϵ
T	T → FT'			T → FT'		
T'		T' → ϵ	T' → *FT'		T' → ϵ	T' → ϵ
F	F → id			F → (E)		

* If TOS is a terminal, the current i/p symbol and TOS must match. If TOS is a non terminal, then there must be an entry with non-terminal and i/p symbol for table.

→ id + id * id $

* When current i/p symbol matches TOS, pop TOS





→ Table :

Method	Stack	Input	Action
	E$	id+id*id$	
	TE'$	id+id*id$	O/P: E→TE'
	FT'E'$	id+id*id$	O/P: T→FT'
	id T'E'$	id+id*id$	O/P: F→id
id	T'E'$	+id*id$	Matched id
id	E'$	+id*id$	O/P: T'→∈
id	+TE'$	+id*id$	O/P: E'→+TE'
id+	TE'$	id*id$	Matched +
id+	FT'E'$	id*id$	O/P: T→FT'
id+	id T' E'$	id*id$	O/P: F→id
id+id	T'E'$	*id$	Matched: id
id+id	*FT'E'$	*id$	O/P: T'→*FT'
id+id*	FT'E'$	id$	Matched: *
id+id*	id T'E'$	id$	O/P: F→id
id+id*id	T'E'$	$	Matched: id
id+id*id	E'$	$	O/P: T'→∈
id+id*id	$	$	O/P: E'→∈

Leftmost Derivation

i/p string — Both are equal

⇒ i/p string accepted

Eg: id + id id $ id * id id $

→ Table

Method	Stack	Input	Action
	E$	id+id$	
	TE'$	id+id$	O/P: E → TE'
	FT'E'$	id+id$	O/P: T → FT'
	id T'E'$	id+id$	O/P: F → id
id	T'E'$	+id$	Matched: id
id	E'$	+id$	O/P: T' → ∈
id	+TE'$	+id$	O/P: E' → +TE'
id +	TE'$	id$	Matched: +
id +	FT'E'$	id$	O/P: T → FT'
id +	id T'E'$	id$	O/P: F → id
id + id	T'E'$	$	Matched: id
id + id	E'$	$	O/P: T' → ∈
id + id	$	$	O/P: E' → ∈

I/P string — Both are equal

⇒ I/P string is accepted.

Eg: id * id id *

Method	Stack	Input	Action
	E$	id*id id$	
	TE'$	id*id id$	O/P: E → TE'
	FT'E'$	id*id id$	O/P: T → FT'E'
	id T'E'$	id*id id$	O/P: F → id
id	T'E'$	*id id$	Matched: id
id	* FT'E'$	*id id$	O/P: T' → *FT'
id *	FT'E'$	id id$	Matched: *

id *	FT'E'$	id id $	Matched: *
id *	idT'E'$	id id $	O/P: F→id
id * id	T'E'$	id $	Matched: id
id * id	*T'E'$	id $	O/P: T'→*FT'

not i/p string — not equal

⟹ i/p string is not accepted

31-01-19

- Verifying if the grammar is LL(1) or not

→ A Grammar 'G' is LL1 if and only if whenever $A \to \alpha / \beta$ are two distinct productions of 'G' the following conditions hold:

(i) For no terminal 'a' do both 'α' and 'β' derive strings beginning with 'a'.

(ii) At most one of 'α' and 'β' can derive the empty string.

(iii) If 'β' derives 'ϵ' then 'α' does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if 'α' derives 'ϵ', i.e., $\alpha \overset{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

This implies that:

(i) FIRST(α) and FIRST(β) are disjoint sets.

(ii) If 'ϵ' is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets and likewise if 'ϵ' is in FIRST(α), then FIRST(α) and FOLLOW(B) are disjoint sets.

Eg Verify if the given grammar is LL(1) or not:
$S \to AaAb \mid BbBa \quad A \to \epsilon \quad B \to \epsilon$

→ $FIRST(S) = FIRST(AaAb) \cap FIRST(BbBa)$
$= \{a\} \cap \{b\} = \phi$ → Condition 1 is satisfied

∵ FIRST(AaAb) or FIRST(BbBa) do not contain 'ϵ', we do not check second condition.

∴ The given grammar is LL(1) grammar.

Eg Verify if the given grammar is LL(1) or not:
$S \to ictSS' \mid a \quad S' \to eS \mid \epsilon \quad C \to b$

→ $S \to ictSS' \mid a$

$FIRST(ictSS') \cap FIRST(A) = \{i\} \cap \{a\} = \phi$

↳ = FIRST(S)

↳ Don't check 2nd condition for S ∵ ϵ is not derived

$FIRST(S') = FIRST(eS) \cap FIRST(\epsilon) = \{es\} \cap \{\epsilon\} = \phi$

↳ 'β' derives 'ϵ', so check 2nd condition

∴ $FIRST(eS) \cap FOLLOW(S') = FIRST(eS) \cap FOLLOW(S)$

$= \{e\} \cap \{e, \$\} = \{e\} \neq 0$

↳ Set is not disjoint.

∴ ↳ Hence, the grammar is not LL(1) grammar.

Eg Verify if the grammar is LL(1) or not:
$S \to 1AB \mid \epsilon \quad A \to 1AC \mid 0C \quad B \to 0S \quad C \to 1$

→ ~~FIRST FIRST~~ $FIRST(1AB) \cap FIRST(\epsilon) = \{1\} \cap \{\epsilon\} = \phi$

↳ Satisfies 1st condition

↳ 2nd condition: $FIRST(1AB) \cap FOLLOW(S) = \{1\} \cap \{\$\} = \phi$

↳ 2nd condition is also satisfied

$FIRST(1AC) \cap FIRST(0C) = \{1\} \cap \{0\} = \phi$

∴ The above grammar is LL(1) grammar.

- Error Recovery Strategies in LL(1) Parsing

1) Panic Mode Recovery
   - Discard the input until we get synchronizing tokens/sets
   - Synchronizing sets include the sets of FOLLOW(A) or FIRST(A) and by introducing a new production $A \to \epsilon$ or some production rule
   - We must use different synchronizing sets to continue the parser

Eg. $E \to E+T/T$ $T \to T*F/F$ $F \to (E)/id$

→ $E \to TE'$ $T \to FT'$ $F \to (E)/id$

$E' \to +TE'/\epsilon$ $T' \to *FT'/\epsilon$

FOLLOW(E) = {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = {), +, $}
FOLLOW(T') = {), +, $}
FOLLOW(F) = {), +, *, $}

For every 'a' in Follows in table M, replace M[A, a] as synch if no entry is already present in that column

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

Corrections / Error Recovery:

(i) If the parser looks up a free M[A, a], & find that it is blank, then, the input symbol 'a' is skipped.

(ii) If the entry is synchronized, then non-terminal on the top of the stack is popped in an attempt to resume parsing.

(iii) If a token on top of the stack does not match the output symbol, then we pop the token from the stack.

Eg. String :) id * + id $

→	Stack	i/p	Remark
	E$	) id * + id $	Skip right parentheses [∵ if we pop E, stack is empty, we cannot parse further, so put E] Expected : left parenthesis / letter
	E$	id * + id $	
	TE'$	id * + id $	
	FT'E'$	id * + id $	
	idT'E'$	* + id $	
	T'E'$	* + id $	
	*FT'E'$	* + id $	
	FT'E'$	+ id $	M[F, +] Synch, pop F Error message: expected identifier skip +
	T'E'$	+ id $	
	E'$	+ id $	
	+TE'$	+ id $	
	TE'$	id $	
	FT'E'$	id $	
	idT'E'$	id $	
	TE'$	$	
	E'$	$	
	$	$	

Eg. string: (id+id)$

→ Stack	Input	Remark
E$	(id+id)$	
TE'$	(id+id)$	
FT'E'$	(id+id)$	
(E)T'E'$	(id+id)$	
E)T'E'$	id+id)$	
TE')T'E'$	id+id)$	
FT'E')TE'$	id+id)$	
idT'E')TE'$	id+id)$	
T'E')T'E'$	+id)$	
E')T'E'$	+id)$	
+TE')T'E'$	+id)$	
TE')T'E'$	id)$	
FT'E')T'E'$	id)$	
idT'E')T'E'$	id)$	
T'E')T'E'$	)$	
E')T'E'$	)$	
)T'E'$	)$	
T'E'$	)$	
E'$	)$	
$	$	

- Phase Level Recovery

Bottom Up Parsing

- It derives rightmost derivation in reverse.
- We start with string and get root node or start symbol.
- We need not eliminate left recursion and left parsing to perform bottom up parsing.
- Scan the input string from left to right and check for which production the terminal matches.
- Eg. $E \to E+T \mid T$ $T \to T*F \mid F$ $F \to (E) \mid id$

  $w = id*id\$$

→ id * id → handle

F * id	(F → id)
T * id	(T → F)
T * F	(F → id)
T	(T → T * F)
E	(E → T)

Rightmost Derivation for given string id * id.

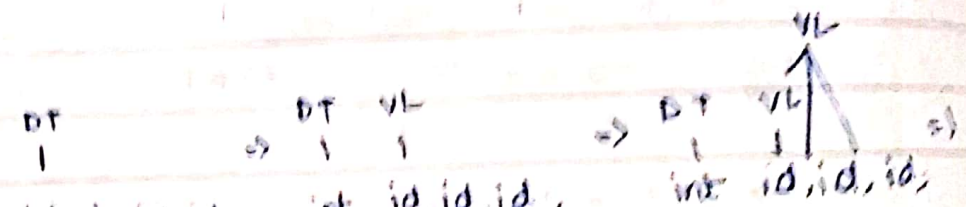
$T \to T \to T*F \to T*id \to F*id \to id*id$

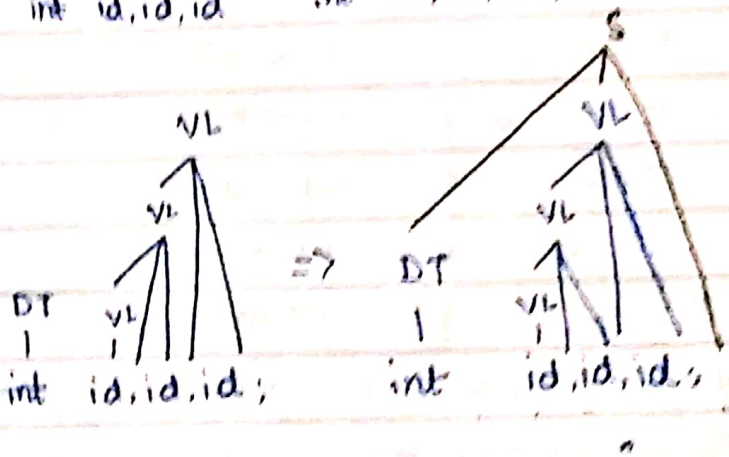
* Thus, bottom up production derives rightmost derivation in reverse.
- We replace / derive start symbol from the string.
- Body of production is replaced with head of production.
- Body of production is called as handle.
- We must replace the handle by a head symbol.
- This entire process is called handle pruning or reduction.

Eg: S → DT VL; DT → int / float VL → VL, id | id

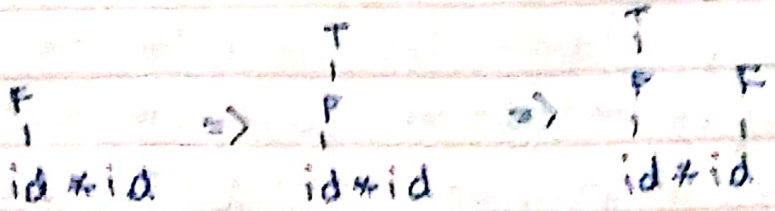
i/p: int id, id, id;

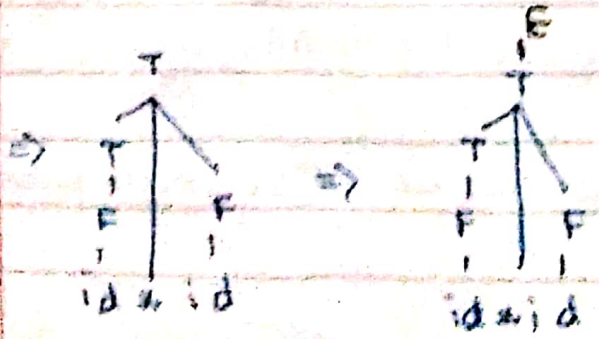
→ int id, id, id;
DT id, id, id; (DT → int)
DT VL, id, id; (VL → id)
DT VL, id; (VL → VL, id)
DT VL; (VL → VL, id)
S (S → DT, VL)





For Previous Problem:

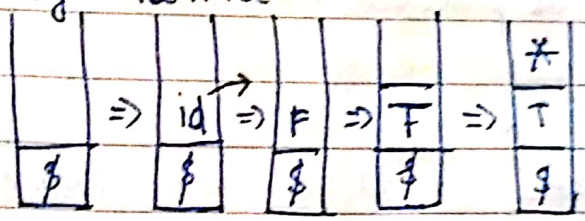


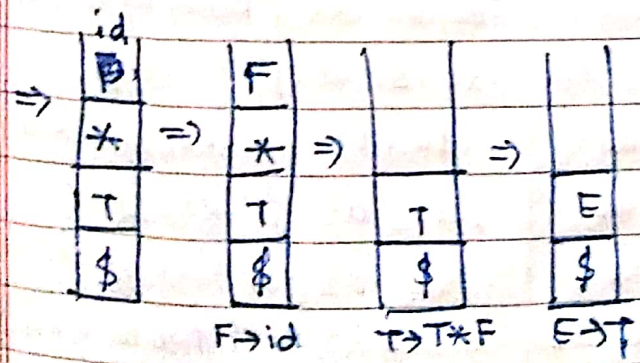


- Shift Reduce Parser
  - It is a simple parser.
  - We write right sentential form, handle and reduction
  - Eg: id * id

Right Sentential form	Handle	Reduction
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T
E		

  - In shift reduce parser, there are four types of actions, which are shift, reduce, accept and error.
  - We make use of stack here
  - Eg: id * id





NOTE: id is matching with any body of production whether when pop id it matching

• Eg: id1 * id2

Pushing into stack = shift action

Stack	Input	Action
$	id1 * id2 $	Shift
$ id1	* id2 $	Reduce by F→id
$ F	* id2 $	Reduce by T→F
$ T	* id2 $	Shift
$ T*	id2 $	Shift
$ T*id2	$	Reduce by F→id
$ T* F	$	Reduce by T→T*F
$ T	$	Reduce by E→T
$ E	$	Accept

→ If the top of string is stack symbol and entire string is processed, then the string is accepted.

• Eg: int id1,id2,id3;

Stack	Input	Action
$	int id1,id2,id3;$	Shift
$ int	id1,id2,id3;$	Reduce by DT→int
$ DT	id1,id2,id3;$	Shift
$ DT id1	,id2,id3;$	Reduce by VL→id
$ DT VL	,id2,id3;$	Shift
$ DT VL,	id2,id3;$	Shift
$ DT VL,id2	,id3;$	Reduce by VL→VL,id
$ DT VL	,id3;$	Shift
$ DT VL,	id3;$	Shift
$ DT VL,id3	;$	Reduce by VL→VL,id

$DT VL	; $	Shift
$DT VL;	$	Reduce by S → DT VL;
$S	$	Accept

⇒ Accepted

Conflicts During Shift Reduce Parsing

→ Two types of conflicts:

i) Shift - Reduce Conflicts

ii) Reduce - Reduce Conflicts

Eg Shift - Reduce Conflicts → One cell contains both shift and reduce actions

S → ictS/ictSes/id

W → ictSes

Stack

$ict S $ictSe

↓ Reduce Action ↓ Shift Action

Eg: CFG for Reduce - Reduce Conflict

Stmt → id (parameter-list)

Stmt → expr := expr

parameter_list → parameter_list, parameter

parameter_list → parameter

parameter → id

expr → id (expr_list)

expr → id

expr_list → expr_list, expr

expr_list → expr

Consider : id (id, id)

stack	i/p
id(id	,id)

[∵ parameter → id
expn → id]

↳ One cell has two reduce actions = Reduce-Reduce conflict

- LR Parsers

i) Simple LR Parser
ii) Canonical LR Parser
iii) LALR Parser

* Compared to LL grammars, most compilers use LR parsers that too CLR and LALR parsers.
* Advantages of LR parsers:
  - CFGs written for programming constructs, 99% it accepts all the constructs by LR parsers
  - Class of grammars of LR grammars is supersets of LL grammars.
  - It uses non-backtracking methods.
  - Efficient error messaging compared to LL parsers
* Disadvantages of LR parsers:
  - Difficult or Robust to Construct.
* Notation of LR parsers

LR(k)
- L → i/p is scanned from left to right
- R → rightmost derivation in reverse
- (k) → number of lookahead symbols

1) Simple LR (SLR) Parser

- We must construct a table (similar to predictive parsing table) from an automata having a set of states $S_0, S_1, \ldots, S_n$ and their transition symbols are the terminals of the CFG.

- We place a dot '.' symbol before the string that has to be parsed yet and the dot is after the string that is parsed.
  Dot at end of production means ready for production.

$S \to ABC \Rightarrow S \to \cdot ABC \Rightarrow S \to A \cdot BC \Rightarrow S \to AB \cdot C \Rightarrow S \to ABC \cdot$

- First step in LR Parsing: Augmentation of Grammar.
  Introduce a new non-terminal $S' \to \cdot S$
  The grammar is accepted if $S' \to S \cdot$
  Accepted if S is reduced to S'

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

→ Augmentation of Grammar
$E' \to \cdot E$
Find out LR(0) items using 'CLOSURE' function and 'GOTO' function.
GOTO: Move from one transition to another transition

Eg: $S \to A \cdot BC$ CLOSURE: $B \to \cdot a$ $B \to \cdot B$
GOTO: GOTO(I0, B) : (I0, I1)

1. $E' \longrightarrow \cdot E$
2. $E \longrightarrow \cdot E + T$
3. $E \longrightarrow \cdot T$
4. $T \longrightarrow \cdot T * F$
5. $T \longrightarrow \cdot F$
6. $F \longrightarrow \cdot (E)$
7. $F \longrightarrow \cdot id$

1st Augment Grammar

Finding Closure

One Canonical LR Initial State I_0

Now, we find GOTO

GOTO (I_0, E) = $E \to E \cdot + T$, $E' \to E \cdot$

GOTO (I_0, T) = $E \to T \cdot$, $T \to T \cdot * F$

GOTO (I_0, F) = $T \to F \cdot$

GOTO $(I_0, ($ = $F \to (\cdot E)$

GOTO (I_0, id) = $F \to id \cdot$

→ CLOSURE of item sets

If 'I' is a set of items for a Grammar 'G', then CLOSURE(I) is the items constructed from I by the two rules:

1.) Initially add every item in I to CLOSURE(I)

2.) If $A \to \alpha B \beta$ is in CLOSURE(I) and $B \to \gamma$ is a problem, than add the item $B \to .\gamma$ to CLOSURE(I). If it is not already there apply this rule until we no more new items can be added to CLOSURE(I).

* COMPUTATION OF CLOSURE

Set of items CLOSURE(I)

{

J = I';
repeat
    for (each production A → α·Bβ in J)
        for (each production B → γ in G')
            if (B → ·γ is not in J)
                add B → ·γ to J';
    until no more items are added
        to J on one round;
    return J';
}

→ GOTO of item sets

GOTO function is used to define the transitions in the LR(0) automaton for a grammar.
GOTO(I, X) → specifies the transitions from the state for I, under i/p X
State for I, under i/p X
where I is a set of items
X is a grammar symbol (either terminal or non terminal

Eg: A → α · Bβ
then
goto (A → α·Bβ, B) = A → αB·β
i.e., moving dot to the right side

→

18-02-'19

LR(0) Itemsets

- Before constructing the table, find LR(0) itemset. If itp is $S \to t$ then it is $S \to .$
- After constructing auto LR(0) itemset, then construct automata, from automata construct table then parse
- Here, there are two itemsets, kernal and non kernal
  If '.' is at the starting, then it is kernal
  If '.' is not at the starting, it is non kernal
- Eg Consider the CFG:

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

I_0 = Initial state

I_0:

$E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

* After . if non-terminal is there, include its production (CLOSURE)

I_1:

GOTO(I_0, E)
$E' \to E.$
$E \to E.+T$
* After we got terminal so stop this state

I_2:

GOTO(I_0, T)
$E \to T.$
$E \to T.*F$
*. is at the end indicates T is ready for reduction

I_3:

GOTO(I_0, F)
$T \to F.$

I_4:

GOTO(I_0, ()
$F \to (.E)$
* after E, non terminal is there, so write production of F
$E' \to .E+T$
$E \to .T$ - non terminal
$T \to .T*F$
$T \to .F$ - non terminal
$F \to .(E)$
$F \to .id$

I_5:

GOTO(I_0, id)
$F \to id.$

classmate
Date
Page

I6
GOTO(I1, +)
+ is from I1.
after •, + is there
E → E+ • T - non terminal
T → • T * F
T → • F - non terminal
F → • (E)
F → • id

I7:
GOTO(I2, *)
* is from I2,
after •, * is there
T → T* • F - non terminal
F → • (E)
F → • id

I8:
GOTO(I4, E)
F → (E •)
E → E • + T

X • GOTO(I4, T)
E → T •
T → T • * F
≅ I2
→ So not needed

X • GOTO(I4, F)
T → F •
≅ I3
→ So not needed

X • GOTO(I4, ()
F → (• E)
E → • E+T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id
≅ I4
→ So not needed

X • GOTO(I4, id)
F → id •
≅ I5
→ So not needed

X • I5 state is over because F → id • is ready for reduction

I9:
GOTO(I6, T)
E → E + T •
T → T • * F

X • GOTO(I6, F)
T → F •
≅ I3
→ So not needed

X • GOTO(I6, ()
F → (• E)
E → • E+T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id
≅ I4
→ So not needed

X • GOTO(I6, id)
F → id •
≅ I5
→ So not needed

I10:
GOTO(I7, F)
T → T * F •

X • GOTO(I7, id)
F → id •
≅ I5
→ So not needed

= R2

R4

X · GOTO(I7, C)

F → (·E)

E → ·E+T

E → ·T

T → ·T*F

T → ·F

F → ·(E)

F → ·id

≅ I4

→ So not needed

X · GOTO(I8, +)

E → E+·T

T → ·T*F

T → ·F

F → ·(E)

F → ·id

≅ I6

→ So not needed

• I10 state

because E+T

is ready for

reduction

• I11 state is

because F → (E)

is ready for

reduction

✓ • I11:

GOTO(I8,))

F → (E)·

X · GOTO(I9, *)

T → T*·F

F → ·(E)

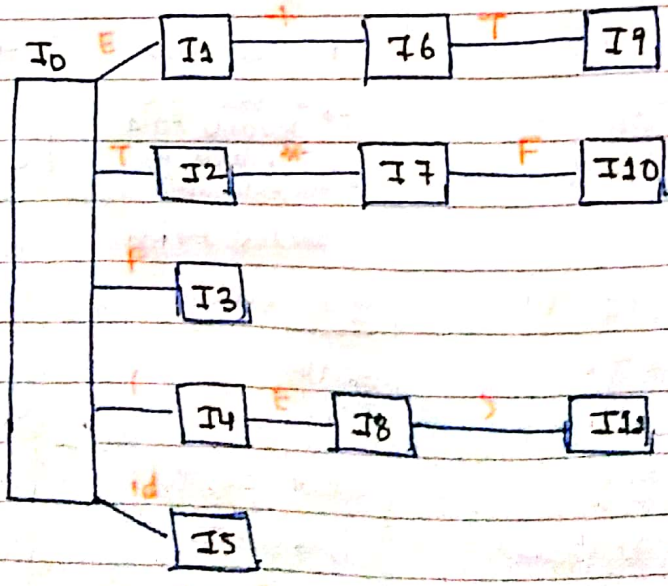
F → ·id

≅ I7

→ So not needed

→ Now, since all the states are over, stop.

No repetition is there, so construct finite automata



Algorithm for Constructing the Parsing Table

- Augmented Grammar G'
- SLR parsing table function ACTION and GOTO for G'

- Method

i) Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of sets of LR(0) items for G'

ii) State i is constructed from I_i, the parsing actions for state i are determined as follows:

a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $GOTO(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j". Here, a must be a terminal.

b) If $[A \to \alpha \cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A) Here, A may not be S'.

c) If $[S' \to S \cdot]$ is in I_i, then set ACTION$[i, \$]$ to "ACCEPT".

NOTE: If any conflicting actions result from the above rules, we say the grammar is not SLR(1).

iii) The GOTO transitions for state i are constructed for all non-terminals A using the rule

If GOTO$[I_i, A] = I_j$ then GOTO$[i, A] = j$

iv) All entries not defined by rule (ii) and rule (iii) are made error.

v) The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$

Table:

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCEPT			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Productions:

1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

I_0: $GOTO(I_0, E) = I_1$ } $E \rightarrow \cdot E+T$

$GOTO(I_0, T) = I_2$ } $E \rightarrow \cdot T$, $T \rightarrow \cdot T*F$

$GOTO(I_0, F) = I_3$ } $T \rightarrow \cdot F$

$GOTO(I_0, C) = I_4$ } $F \rightarrow \cdot (E)$

↳ ACTION(I_0, C) = Shift 4 = S4

$GOTO(I_0, id) = I_5$ } $F \rightarrow \cdot id$

↳ ACTION(I_0, id) = Shift 5 = S5

I_1: $S' \rightarrow S\cdot \Rightarrow E' \rightarrow E\cdot$ Accept

ACTION$(1, \$)$ = ACCEPT

$GOTO(I_1, +)$ } $E \rightarrow E\cdot +T$

$= I_6$ = Shift 6

= S6

I2: E → T•

ACTION(2, □) = Reduce by E → T = R2

□ ∈ FOLLOW(E)

→ E → E+T E → T T → T*F T → F F → (E) F → id

FOLLOW(E) = {+,), $}

FOLLOW(T) = {+, *,), $}

FOLLOW(F) = {+, *,), $}

→ ACTION(2,)) = ACTION(2, +) = ACTION(2, $) = R2

GOTO(I2, *) = I7 ∵ T → T•*F

ACTION(I2, *) = Shift 7 = S7

I3: T → F•

ACTION(I3, □) = Reduce by T → F = R4

□ = FOLLOW(T) = {+, *,), $}

ACTION(I3, +) = ACTION(I3, *) = ACTION(I3,)) = ACTION(I3, $) = R4

I4:

GOTO(I4, E) = I8 ∵ F → (E), E → •E+T

GOTO(I4, T) = I2 ∵ E → •T, T → •T*F

GOTO(I4, F) = I3 ∵ T → •F

F → •(E) ⇒ ACTION GOTO(I4, () = I4 = Shift 4 = S4

F → •id ⇒ ACTION GOTO(I4, id) = I5 = Shift 5 = S5

I5: F → id•

ACTION(I5, □) = Reduce by F → id = R6

□ = FOLLOW(F) = {+, *,), $}

ACTION(I5, +) = ACTION(I5, *) = ACTION(I5,)) = ACTION(I5, $) = R6

I6: $E \to E + \cdot T$, $T \to \cdot T * F$

GOTO (I6, T) = I9

$T \to \cdot F$

GOTO (I6, F) = I3

$F \to \cdot (E)$ => ACTION GOTO (I6, () = Shift 4 = S4

$F \to \cdot id$ => ACTION GOTO (I6, id) = Shift 5 = S5

I7: $E \to T * \cdot F$ => GOTO (I7, F) = I10

$F \to \cdot (E)$ => ACTION (I7, () = Shift 4 = S4

$F \to \cdot id$ => ACTION (I7, id) = Shift 5 = S5

I8: $F \to (E \cdot)$ => ACTION GOTO (I8,)) = Shift 11 = S11

$F \to E \cdot + T$ => ACTION (I8, +) = Shift 6 = S6

I9: $E \to E + T \cdot$

ACTION (I9, □) = Reduce by $E \to E + T$ = R1

□ = FOLLOW (E) = {+,), $}

ACTION (I9, +) = ACTION (I9,)) = ACTION (I9, $) = R1

$T \to T \cdot * F$ => ACTION (I9, *) = Shift 7 = S7

I10: $T \to T * F \cdot$

ACTION (I10, □) = Reduce by $T \to T * F$ = R3

□ = FOLLOW (T) = {+, *,), $}

ACTION (I10, +) = ACTION (I10, *) = ACTION (I10,)) = ACTION (I10, $) = R3

I11: $F \to (E) \cdot$

ACTION (I11, □) = Reduce by $F \to (E)$ = R5

□ = FOLLOW (F) = {+, *,), $}

ACTION (I11, +) = ACTION (I11, *) = ACTION (I11,))

= ACTION (I11, $) = R5

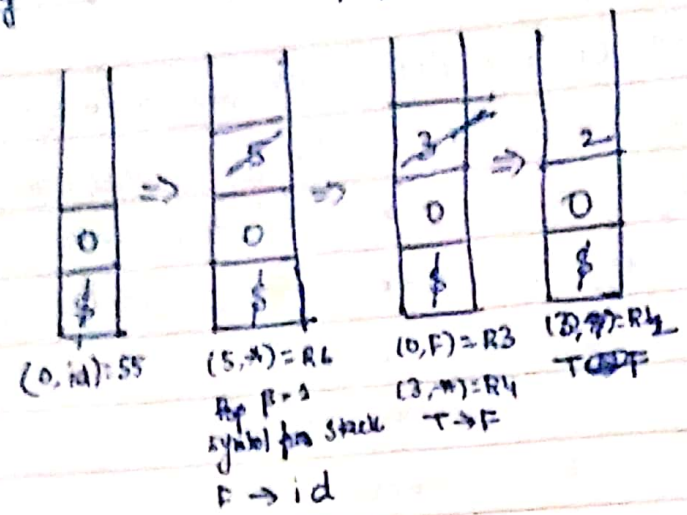
LR Parsing Algorithm:

→ i/p : i/p string w
LR parsing table with ~~act~~ functions ACTION and GOTO for a grammar G

→ O/P : If w is in L(G) the reduction steps of a bottom-up parser for w, otherwise an error

→ Method : Initially
So - on the stack
So - initial state
w$ in the i/p buffer
Let a be first symbol of w$
while (1)
{ Let S be the state on the top of the stack
if (ACTION [S, a] = Shift t)
{ push t onto the stack
let a be the next i/p symbol }
else if (ACTION [S, a] = Reduce A→β)
{ pop (β) symbols off the stack,
let state t now be on the top of the stack;
push GOTO(t, A) onto the stack
O/P the production A→β; }
else if (ACTION [S, a] = accept)
break;
else
call error recovery routine;
}

Eg. i/p string id*id+id



Stack	Symbols	i/p	Action
0		id*id+id$	Shift
05	id	*id+id$	Reduce F→id
03	F	*id+id$	Reduce by T→F
02	T	*id+id$	Shift
027	T*	id+id$	Shift
0275	T*id	+id$	Reduce by F→id
02710	T*F	+id$	Reduce by T→T*F
02	T	+id$	Reduce by E→T
01	E	+id$	Shift
016	E+	id$	Shift
0165	E+id	$	Reduce by F→id
0163	E+F	$	Reduce T→F
0169	E+T	$	Reduce by E→E+T
01	E	$	Accept

ii) Canonical LR Parsers (CLR)

- Normal LR Parsers are CLR parsers.
- Used in compiles construction.
- There are `1' no. of lookahead symbols.
- We must consider LR(1) itemsets.
- Algorithm for constructing LR(1) itemsets:

```
set of Items CLOSURE(I)
{
  repeat
    for (each item [A → α·Bβ, a] in I)
      for (each production B → γ in G')
        for (each terminal b in FIRST(βa))
          add [B → ·γ, b] to set I,
          until no more items are added to I;
          return I;
}
set of Items GOTO(I, X)
{
  Initialize J to be the empty set;
  for (each item [A → α·Xβ, a] in I)
    add item [A → αX·β, a] to set J;
    return CLOSURE [J];
}
void items(G')
{
  Initialize C to {CLOSURE ({[S' → S, $]})}
```

repeat
for (each set of Items I in C)
for (each grammar symbol X)
if (GOTO (I, X) is not empty and not in C)
add GOTO(I, X) to C;
until no new sets of items are added to C;
}

Eg:
$S' \rightarrow \cdot S \Rightarrow S' \rightarrow \cdot S, \$$
If $S \rightarrow \alpha, X \Rightarrow S \rightarrow \cdot \alpha, \$ \quad S \rightarrow \cdot X, \$$
$S \rightarrow \alpha \cdot B\beta, a \quad FIRST(\beta a)$

- Construct LR(1) itemsets for the following grammar:
$S \rightarrow CC \quad C \rightarrow aC/d$

→ $S' \rightarrow \overset{\alpha \cdot B\beta}{\cdot S}, \$ \quad \alpha \cdot B\beta, a$
$S \rightarrow \overset{\alpha : B\beta}{\cdot CC}, \$ \quad b \in FIRST(\beta a) = FIRST(\epsilon\$) = \$$
$C \rightarrow \cdot aC, \$/a/d \quad b \in FIRST(\beta a) = FIRST(C\$) = \{a/d\}$
$C \rightarrow \cdot d, \$/a/d$

✓I0:
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot aC, a/d$
$C \rightarrow \cdot d, a/d$

✓I1:
GOTO(I0, S) = $S' \rightarrow S \cdot, \$$

✓I2:
GOTO(I0, C) = $S \rightarrow \overset{\alpha \cdot B\beta}{C \cdot C}, \$$
$C \rightarrow \cdot aC, \$$ } $b = FIRST(\beta a) = FIRST(\epsilon\$) = \$$
$C \rightarrow \cdot d, \$$ }

✓I3: GOTO (I0, a) = $C \rightarrow \overset{a \cdot B\beta}{a \cdot C}, a/d$
$C \rightarrow \cdot aC, a/d$ } $b = FIRST(\beta a) = FIRST(Ca) \cup FIRST(\epsilon) = a/d$
$C \rightarrow \cdot d, a/d$ }

✓ I4: GOTO (I0, d) = C → d·, a/d

✓ I5: GOTO (I2, C) = S → CC·, $

✓ I6: GOTO (I2, a) = C → a·C, $
C → ·aC, $
C → ·d, $
} FIRST(βa) = FIRST(C$) = $

✓ I7: GOTO (I2, d) = C → d·, $

✓ I8: GOTO (I3, C) = ~~C → aC·, a/d~~ C → aC·, a/d

✗ I9: GOTO (I3, a) = a·C, a/d
C → a·aC, a/d C → ·d, a/d
≈ I3

✗ I9: GOTO (I3, d) = C → d·, a/d ≈ I4

✗ I4, I5 → already reduced actions

✓ I9: GOTO (I6, C) = C → aC·, $

✗ I10: GOTO (I6, a) = C → a·C, $
C → ·aC, $ C → ·d, $ ≈ I6

✗ I10: GOTO (I6, d) = C → d·, $ ≈ I7

✗ I7, I8, I9 → already reduced actions

Table

	TERMINALS			VARIABLES	
	a	d	$	S	C
0	S3	S4		1	2
1			ACCEPT		
2	S6	S7			
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

Reductions:

1. S → CC
2. C → aC
3. C → d

I1: S' → S•, $

(1, $) = ACCEPT

8 → CC C → d

S → CC

I4: C → d•, a/d

(4, a) = (4, d) = R3

I5: S → CC•, $

(5, $) = R1

C → dd

I7: C → d•, $

(7, $) = R3

I8: C → aC, a/d

I9: C → aC•, a/d

(8, a) = (8, d) = R2

C → aC

I9: C → aC•, $

(9, $) = R2