Compilar Design (6.10.2025)

High level ~~error~~ $\boxed{\text{compiler}}$ → low level language

↑

Error

Source code → Analysis → synthesis → Machin code

$\boxed{\text{Frontend}}$

Analysis → lexical analyzer → make token (stream of token)

→ syntex analyzer → make parse tree from token

→ semantic analyzer → make sematically parse tree.

→ Intermediate code generation → make 3-address code

$\boxed{\text{Backend}}$

Synthesis → code optimizer → modified 3-address code

→ target code generation → low level language

HLL

↓

Pre-processor

↓ pure HLL

compile

↓ assembly language

assembler

↓ Relocatable machin code

loader/Linker

↓ Absolute machin code

Executable code

Linker: is a computer program that combine and link many object files to create one executable file.

Loader: is a operating system component that load and execute files into main memory

Phases of compiler:

Symbol table: During execution store different value.

int x, a, b, c;
x = a + b * c

Symbol table

Num	Variable	Type
1	x	int
2	a	int
3	b	int
4	c	int

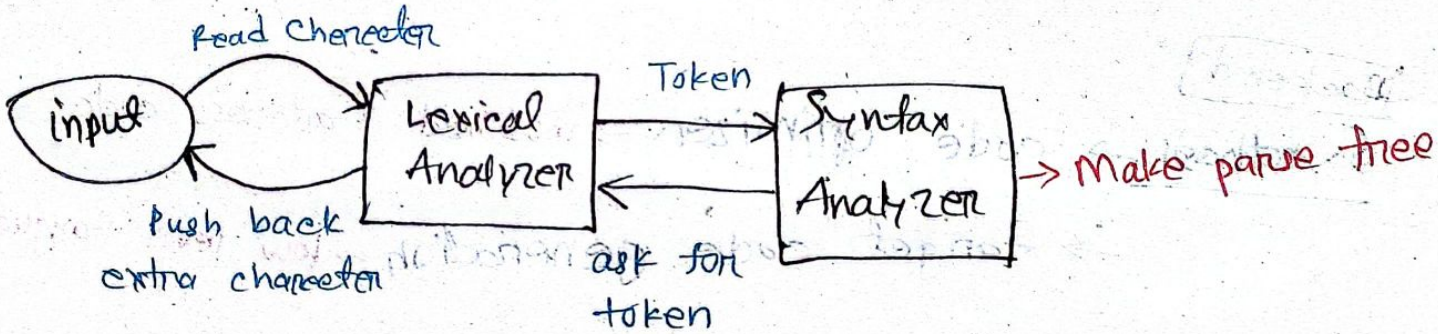
Lexical analyzer:

<id,1> = <id,2> + <id,3> * <id,4>

id → identifier

Lexical Analyzer: Like scanner.

→ It can be implement using DFA



Token: is a sequence of character that can be treated as a unit in the grammer of the programming langua

→ keyword, identifier, operator, constant, punctuation.

pre define in language

define by user
Use variable

lexemes	Token
x	identifier
=	operator
a	indetifier

Regex for identifier = l(l+d)*/_(l+d)*

1) Input pre processing → remove space, comment etc. (also add macro, header file)

2) Tokenization → make sequence of token.

3) Token classification → classify different type token.

4) Token validation → Validate token with proper type.

5) Error detection → Ditect syntax error.

* "%d" → here all things in quote is a single token.
* token never think about error.

printf ("the value %d", i); a == 20 int 'f' char *a

/* Farhan */ t A == B c+-++---; &a

→ printf (" value is); This is toke error. Single & Double quote must be end.

Grammer: It refers to the formal rules that define the syntax of a programming language.

G(V, T, P, S)

V → Variable/ Non-terminal

T → Terminal

P → Production rule

S → Start symbol

Language: refers to the set of all possible strings that can be generated by that grammer.

Finite → $S \to AB$, $A \to a$, $B \to b$

Infinite → $S \to aS / \epsilon$

Left Most Derivation: is a way of deriving string from a context free grammer by always expanding the leftmost non-terminal first.

Ex: $E \to E+T / T$, $T \to T*F / F$, $F \to (E) / id$.

Derive $id+id*id$ using left most derivation.

Sol: left most derivation

$$
\begin{aligned}
E &\to E+T \\
&\to T+T \\
&\to F+T*F \\
&\to id+F*F \\
&\to id+id*id
\end{aligned}
$$

Right Most Derivation:

$$
\begin{aligned}
E &\to E+T \\
&\to E+T*F \\
&\to E+T*id \\
&\to E+F*id \\
&\to E+id*id \\
&\to F+id*id \\
&\to id+id*id
\end{aligned}
$$

Parse tree:

$E \to E+E / E*E / id$

String: id+id*id

L.M.D: $E \to E+E$

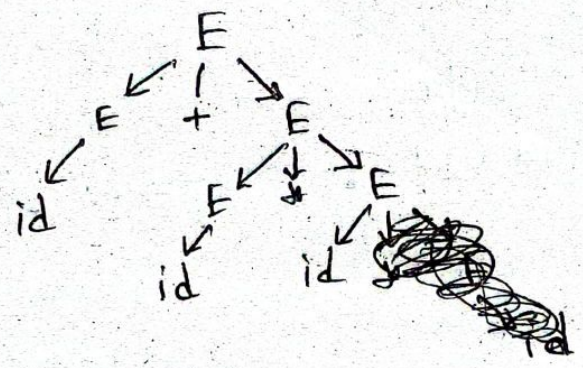
~~→ id+id*id~~

$\to id+E$

$\to id+E*E$

$\to id+id*E$

$\to id+id*id$



Ambiguous Grammer: when there exist more than one left most or right most or parse tree for the input string. (Only left most derivation or right most derivation or any multiple derivation)

→ more than one distinct parse tree

First & Follow:

First: The first set of non-terminal symbols represents the set of terminal symbols that can appear as the first token in any string derived from the non terminal.

$S \to abc | def$

$F(S) = \{a, d\}$

(i) $S \to Bbc$

$B \to aB$

$C \to c/d$

Start from bottom.

$\to \{a\}$ as B has terminal a

$\to \{a\}$

$\to \{c, d\}$

(ii) $S \to Bbc / Cd$

$B \to aB / \epsilon$

$C \to cC / \epsilon$

$\to \{a, b, c, d\}$

$\to \{a, \epsilon\}$

$\to \{c, \epsilon\}$

$S \to \cancel{B}^{\epsilon}bc \to \{a, b\}$

$S \to \cancel{C}^{\epsilon}d \to \{c, d\}$

(iii) $S \to AB$

$A \to aA / \epsilon$

$B \to bB / c$

if $B \to bB / \epsilon$ then

$\to \{a, b, c\}$ $\{a, b, \epsilon\}$

$\to \{a, \epsilon\}$

$\to \{b, c\}$ $\to \{b, \epsilon\}$

$S \to AB$

$\to a, b, B$

$\to a, b, c$

(iv) $E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to id / (E)$

$\to \{id, \epsilon\}$

$\{+, \epsilon\}$

$\to \{id, (\}$

$\to \{*, \epsilon\}$

$\to \{id, (\}$

(v) $S \to ACB / CbB / Ba$

$A \to dd / BC$

$B \to g / \epsilon$

$C \to h / \epsilon$

$\to \{d, gh, \epsilon, b, a\}$

$\to \{d, g, h, \epsilon\}$

$\to \{g, \epsilon\}$

$\to \{h, \epsilon\}$

Follow:

Start symbol doller $

→ right side এ কোথায় আছে সেখানে যাবে

১. যার follow বের করতে হবে, তার immediate right side এ যে থাকবে তাকে পিক করতে হবে।

→ Right side এ যদি Terminal symbol থাকে তাহলে সেটি বসবে।

→ যদি Non-terminal থাকে তাহলে তার First বের করে, First এর value বসে যাবে।

→ Right side এ যদি কিছু না থাকে তাহলে

follow of L.H.S = Follow of R.H.S

[Start symbol সবসময় $ দিয়ে শুরু হবে]

(1) $S \to ABd$
$B \to b$

$F(S) \to \{\$\}$ [ডানে আর কোথাও নেই] তাই $F(S) \Rightarrow$ শুধু $ হবে
$F(B) \to \{d\}$

(2) $S \to ABD$
$B \to b$
$D \to d$

$F(S) \to \{\$\}$
$F(B) \to \{d\}$ [First বের করে বসাইছি]
$F(D) \Rightarrow F(S) = \{\$\}$ [Last rule]

(3) $S \to ABDe$
$B \to b / \epsilon$
$D \to d / \epsilon$

$F(S) \to \{\$\}$
$F(B) \to \{d, e\}$ [First বসাইছি]
$F(D) \to \{e\}$ [ϵ হিসাব হবে না]

(4) $S \to ABD$
$B \to b / \epsilon$
$D \to d / \epsilon$

$F(S) \to \{\$\}$
$F(B) \to \{d, \$\}$ → $First(D) = \{d, \epsilon\}$ then rule last
$F(D) = F(S) \to \{\$\}$

ϵ যেখানে D vanish হয়ে যাবে last বা right এ আর কিছু থাকেনা

Ex:

$E \to TE'$ —— $F(E) \to \{\$,)\}$

$E' \to +TE' / \epsilon$ —— $F(E') = F(E) \to \{\$,)\}$ ← আবারও E' থাকলেও set হওয়ায় repeat হবে না।

$T \to FT'$ —— $F(T) \to \{+, \$,)\}$

$T' \to *FT' / \epsilon$ —— $F(T') = F(T) \to \{+, \$,)\}$

$F \to id \mid (E)$ —— $F(F) = \{*, +, \$,)\}$

↳ $First(T') \to \{*, \epsilon\}$

ϵ যাবার পরে, $F(F) = F(T) \to \{+, \$, (\}$ $\Big\} \{*, +, \$, (\}$

* যার follow বের করব সে যত জায়গায় আছে সবখানেই দেখতে হবে।

First & Follow : 6/24/25

	First	Follow
$S \to ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \to a \mid \epsilon$	$\{a, \epsilon\}$	$\{b, c\}$
$B \to b \mid \epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \to c$	$\{c\}$	$\{d, e, \$\}$
$D \to d \mid \epsilon$	$\{d, \epsilon\}$	$\{e, \$\}$
$E \to e \mid \epsilon$	$\{e, \epsilon\}$	$\{\$\}$

উপর থেকে করতে।

নিচে থেকে করবে

	First	Follow
$S \to ACB \mid CbB \mid Ba$	$\{d, g, h, \epsilon, b, a\}$	$\{\$\}$
$A \to da \mid BC$	$\{d, g, c, \epsilon\}$	$\{h, g, \$\}$
$B \to g \mid \epsilon$	$\{g, \epsilon\}$	$\{\$, a, h, g\}$
$C \to h \mid \epsilon$	$\{h, \epsilon\}$	$\{g, \$, b, h\}$

* First এ ϵ থাকতে পারে কিন্তু follow তে থাকতে পারে না।

LL(1) Parsing Table: CFG দেয়া থাকবে।

First & Follow বের করি

	First	Follow
$S \to AaB$	$\{a\}$	$\{\$\}$
$A \to a$	$\{a\}$	$\{a\}$
$B \to \epsilon$	$\{\epsilon\}$	$\{\$\}$

Parsing Table বানাই-

Row তে non-terminal

Non-terminal	a	$
S	$S \to AaB$	
A	$A \to a$	
B		$B \to \epsilon$

column এ terminal সব এবং $ নিবে। তবে ε হবে না।

* cell এ যে production rule থেকে terminal এর first বের করছি সেটা।
* যদি ε এ যায় তবে Non-terminal এর follow column এ বসাবে।

→ First এর সকল terminal এ production rule বসাবে।

→ ε হলে ফলোর সকল terminal এ বসাবে।

Ex: 2

E $

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to +TE' \mid \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to *FT' \mid \epsilon$	{*, ε}	{+, $,)}
$F \to id \mid (E)$	{id, (}	{*, +, $,)}

Non-Ter	id	(	)	*	+	$
E	$E \to TE'$	$E \to TE'$				
E'			$E' \to \epsilon$		$E' \to +TE'$	$E' \to \epsilon$
T	$T \to FT'$	$T \to FT'$				
T'			$T' \to \epsilon$	$T' \to *FT'$	$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	$F \to (E)$				

Is this CFG is LL(1) Parser or not.

	First	Follow
$S \to A \mid B$	{a}	{$}
$A \to aC$	{a}	{$}
$B \to aD$	{a}	{$}
$C \to c$	{c}	{$}
$D \to d$	{d}	{$}

Non-Ter	a	c	d	$
S	$S \to A / S \to B$			
A	$A \to aC$			
B	$B \to aD$			
C		$C \to c$		
D			$D \to d$	

* একটি ঘরে একাধিক production rule থাকলে LL(1) না।

CJ : 1 28.06.25

Contex Free Grammer:

$G = \{V, \Sigma, S, P\}$

V = Set of variable or Non terminal symbol

Σ = set of terminal symbol.

S = Start symbol

P = Production Rule

$A \rightarrow a$

$a = \{V \cup \Sigma\}^*$ and $A \in V$

Classifications:

Grammer Type	Grammer	Language	Machin
Type: 0	Unrestricted	Recursively Enu	Turing Machin
Type: 1	Contex Sensitive	CSL	Linear Bounded Autom
Type: 2	Contex free	CFL	Push down automat
Type: 3	Regular Gramer	Regular language	Finite Automat

Chomsky Normal Form: (Special way to write CFG invented by)

i) No ϵ symbol. (without $S \rightarrow \epsilon$)

ii) In right side max 2 Non-terminal or 1-terminal symbol is allowed.

Simplication of CFG:

i) Reduction of CFG.

ii) Remove Null production. যে কোনো ভাবে Null থাকলে remove. → Direct or Indirect

iii) Remove Unite production. $(A \to B)$

iv) Only allow max 2-variable

v) Remove a terminal & non-terminal together.

Conversion CFG to CNF:

P: $S \to ASA \mid aB,\ A \to B \mid S,\ B \to b \mid \epsilon$

P: $S' \to S,\ S \to ASA \mid aB,\ A \to B \mid S,\ B \to b \mid \epsilon$ (i) Since S appear RHS

P: $S' \to S,\ S \to ASA \mid aB \mid a,\ A \to B \mid S \mid \epsilon,\ B \to b$ (ii) Remove Null production

P: $S' \to S,\ S \to ASA \mid aB \mid a \mid AS \mid SA \mid S,\ A \to B \mid S,\ B \to b$ → again remove null production

(iii) Remove Unite Production $S' \to S,\ A \to B,\ A \to S,\ S \to S$

$A \to B \mid S,\ B \to b$

$S' \to S,\ S' \to ASA \mid aB \mid a \mid AS \mid SA$ [After removing $S \to S$ unite production

~~$S' \to ASA \mid aB \mid a \mid AS \mid SA$~~

$S \to ASA \mid aB \mid a \mid AS \mid SA$

$S' \to S$

$A \to B \mid S,$

$B \to b$

$S \to S$ is a loop. So no need to add after remove.

$S' \to ASA \mid aB \mid a \mid AS \mid SA$

$S \to ASA \mid aB \mid a \mid AS \mid SA$

$A \to b \mid S$

$B \to b$

$A \to B$

$S' \to S$

$S \to \ldots$

Here first calculates the write above line

* after remove Unite Production must remove unreachable variable.

$S' \to ASA \mid aBa \mid AS \mid SA$

$S \to ASA \mid aB \mid a \mid AS \mid SA$

$A \to b \mid ASA \mid aB \mid a \mid AS \mid SA$

$B \to b$

put whole S value to A.

$A \to S$

(iv) Remove more than two variable in RHS

$S' \to AX \mid aB \mid a \mid AS \mid SA$

$S \to AX \mid aB \mid a \mid AS \mid SA$

$A \to b \mid AX \mid aB \mid a \mid AS \mid SA$

$B \to b$

$X \to SA$

(v) Remove production rules like aB, Ba

P:

$S' \to AX \mid YB \mid a \mid AS \mid SA$

$S \to AX \mid YB \mid a \mid AS \mid SA$

$A \to b \mid AX \mid YB \mid a \mid AS \mid SA$

$B \to b$

$X \to SA$

$Y \to a$

violate rules as it has one terminal and non-terminal together.

$S' \to aB$

$S \to aB$

$A \to aB$

CNF rules:

$G = (V, \Sigma, R, S)$

V → Non-terminal

Σ → Terminal

$A \to a$ where $A \in V$ and $a \in \Sigma$

"\" symbol → without → except

$A \to BC$ where $A, B, C \in V \setminus \{S\}$ (B, C can variable but S cannot.)

S cannot appear in RHS.

$S \to \epsilon$ only start symbol can do this if ϵ is allowed in this L.

If $\alpha \to \beta$

then $\alpha \in V$ (left side is non-terminal)

$\beta \in (V \cup \Sigma)^*$ (right side is made of variable and terminal)

But must follow CNF rules.

Bad Rules: (The rules are forbidden in CNF)

1. $A \to wSw$; $A \in V$ and $w \in (V \cup \Sigma)^*$ Start symbol viabction
2. $A \to \epsilon$; $A \in V \setminus \{S\}$ or $A \in V$ and $A \neq S$ Iphsilon rule
3. $A \to B$; $A, B \in V$ Unite Rule → Non-terminin produce Non-term
4. $A \to w$; $A \in V$ and $w \in (V \cup \Sigma)^*$ Mixed Rules → both V and T
5. $A \to w$; $w \in V^*$ and $|w| > 2$ long rule

   ↳ All non terminal

   → In RHS the non-terminal cannot be longer than length two.

BNF (Backus Naur Form) ALGOL 60

→ It is a notation techinique for CFG of language.

<symbol> := Expression1 | Expression2 | Ex3

Non-terminal — sequence of symbol

E → E+E / E*E

→ optional symbol are written in [ ]

→ Repet Zero or more time a*

→ Repet One or more time a+

→ Alternative choice a|b

→ Grouping ( )

<sentence> ::= <subject> <verb> <object> ::= is define as

SVO → subject verb object I eat rice

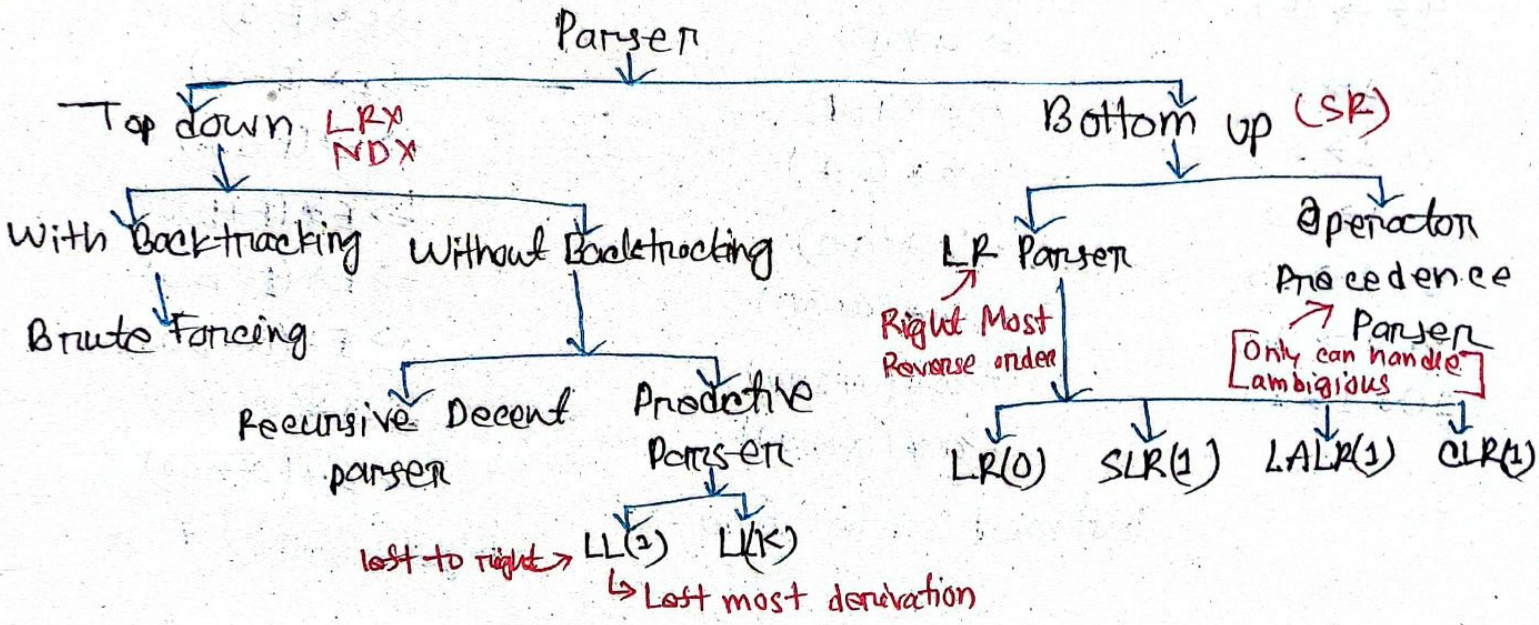
SOV → subject object Verb আমি ভাত খাই

→ BNF defines rules for writting structured rule-based language like code query protocal.

→ By changing the order of non-terminal it can produce natural language.

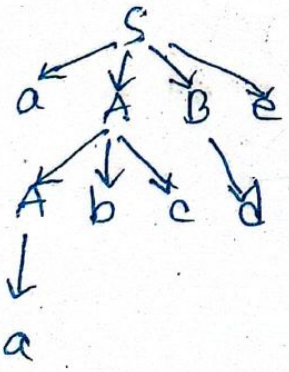
Parser/ Syntax Analyzer:

A parser check if a code is correct according to the rules of grammer. If a parser can make a parse tree starting from the start symbol and ends with the input token using all grammer rules, then the code is syntactically correct even though it is semantically incorrect.



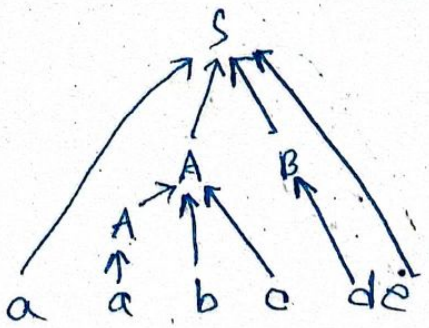
Topdown → From start to end → Decide which production rule use.

Bottom Up → start from end to start → Decide when to reduce.



S = aabcde

Left Most Derivation



S = aabcde

Right Most Derivation

Parse tree Derivation (Sential Form) Parser

→ Each step is called sential form that replace one non-terminal using production rule..

Given:

1. $E \to I$
2. $E \to E+E_1$
3. $E \to E*E$
4. $E \to (E)$
5. $I \to a$
6. $I \to s$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Now, produce

LEFT MOST

$E \Rightarrow E*E$

$\Rightarrow I*E$

$\Rightarrow a*E$

$\Rightarrow a*(E)$

$\Rightarrow a*(E+E)$

$\Rightarrow a*(I+E)$

$\Rightarrow a*(a+I)$

$\Rightarrow a*(a+I0)$

$\Rightarrow a*(a+I00)$

$\Rightarrow a*(a+b00)$

left most sential form

Right Most

$E \Rightarrow E*E$

$\Rightarrow E*(E)$

$\Rightarrow E*(E+E)$

$\Rightarrow E*(E+I)$

$\Rightarrow E*(E+I0)$

$\Rightarrow E*(E+I00)$

$\Rightarrow E*(E+b00)$

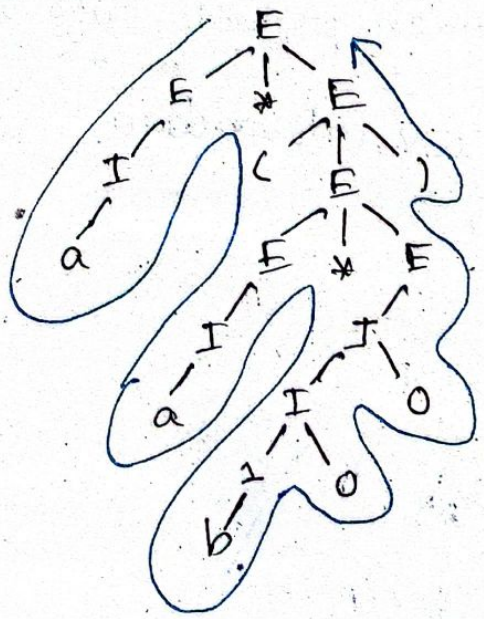
$\Rightarrow E*(I+b00)$

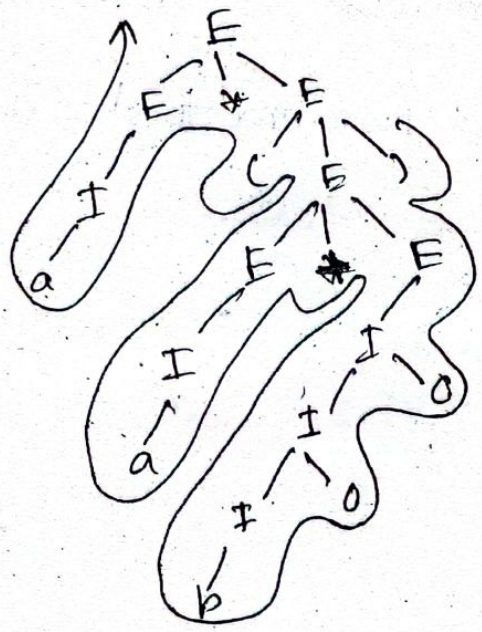
$\Rightarrow E*(a+b00)$

$\Rightarrow I*(a+b00)$

$\Rightarrow a*(a+b00)$

Right Sential Form





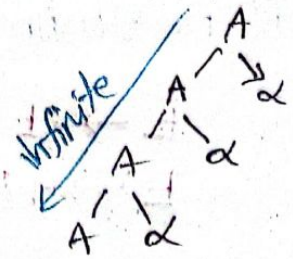
Elemination of Left Recursion: (Same symbol come at first)

Like $A \to A\alpha \mid \beta$

RE: $\beta\alpha^*$

$A \to \beta A'$

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



$P \to P + Q \mid Q$

$A \quad A \quad \alpha \quad \beta$

$A \to \beta A'$

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

To remove

$A \to \beta A'$

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

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

$A \quad \alpha \quad \beta$

Sol: $E \to TE'$

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

Ex: $E \to E+E \mid E \times E \mid a$

$A \quad \alpha \quad A \quad \alpha \quad \beta$

Sol: $E \to aE'$

$E' \to +EE' \mid \times EE' \mid \epsilon$

Ex: $A \to aBd \mid Aa \mid a$

$a \quad \alpha \quad a \quad \alpha \quad \beta$

Sol: $A \to aA'$

$A' \to BdA' \mid aA' \mid \epsilon$

Ex: $A \to (B) \mid b$

$B \to B \times A \mid A$

$A \quad \alpha \quad \beta$

Sol: $A \to (B) \mid b$

$B \to AB'$

$B' \to \times AB' \mid \epsilon$

Finall, we can say-

If $A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$ → without left recursion

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \epsilon$

But right recursion is safe because input ends and infinite is stoped.

→ A recursive decent parser cannot handle left recursion

→ With left recursion, the parser can go into infinite recursion

Compilar Design

$(a-b)-c$ left associative → + - * / % Normal, From left to right

$a-(b-c)$ Right associative → ^ ↑ = 2^3^2 → 2^(3^2)

Ex: a=b=c=5; a=(b=(c=5));

Ex: $E \to E+E \mid E-E \mid E \uparrow E \mid E*E \mid id$ make unambigious

Assume procedence →

1. ↑
2. *
3. +, -

Associativity →

1. ↑ right-associative
2. *, +, - left-associative

Now, $E \to E+T \mid E-T \mid T$ → initial left associative +, -

$T \to T*F \mid F$ → Enforce * than higher precedence than +-

$F \to G \uparrow F \mid G$ → Higher than * and right associative

$G \to id$ → Basic operand.

Must Know → Each level should contain

i) One or more recursive case

ii) One base case.

Ex: $E \to E * E \mid E + E \mid E - E \mid num$

$E \to E + T \mid E - T \mid T$

$T \to T * F \mid F$

$F \to num$

Ex: $E \to E \uparrow E \mid E / E \mid id$

$E \to E / T \mid T$

$T \to F \uparrow T \mid F$ Right Associative

$F \to id$

Ex: $E \to E = E \mid E + E \mid E * E \mid id$

$E \to id = E \mid A$ Right associative

$A \to A + B \mid B$

$B \to B * F \mid F$

$F \to id$

Q1: Convert $E \to E \wedge E \mid E \vee E \mid \neg E \mid f \mid t$ to LL(1)

Left-Recursive: $E \to E \wedge E \mid E \vee E$

Right-Recursive: $E \to \neg E \mid f \mid t$

$E \to \neg E E' \mid f E' \mid t E'$

$E' \to \wedge E E' \mid \vee E E' \mid \varepsilon$

Condition for LL(1) Grammer

→ No left recursion

→ Left factored (No common Prefix)

→ No Ambiguity

Q2:

$S \to aABb$

$A \to c \mid \varepsilon$

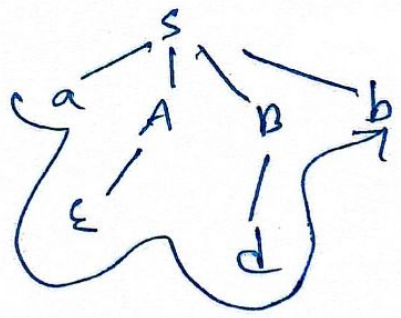
$B \to d \mid \varepsilon$

for stin adb make parse-tree

$S \to aABb$

$\to aBb$

$\to adb$



Recursive Decend Parsing: (Noso)

For Top-Down

→ CFG should not have left recursion or nonditerminism

$E \to iE'$
$E' \to +iE' \mid \varepsilon$

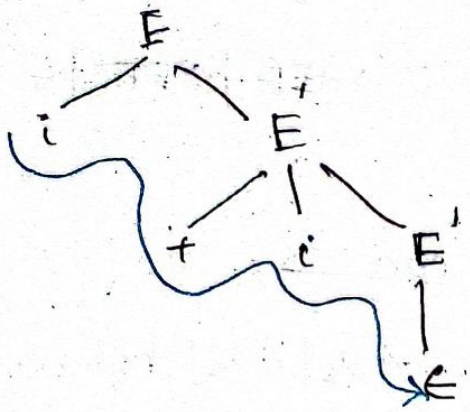
```
main()
{
    E();
    if(look_ahead = '$')
        printf("Parsing Successfull");
}

E()
{
    if(look ahead = 'i')
    {
        match('i');
        ~~match~~
        E'();
    }
}
E'()
{
    if(look ahead = '+')
    {
        match('+');
        match('i');
        E'();
    }else{
        return;
    }
}
match(char c){
    if(look_ahead = c)
        look_ahead = getchar();
    else
        printf("ERROR");
}
```

input = i+i$

look ahead → input indicator

→ যখন কোনো function E(), E'() কল হবে সেটাও parse tree তৈরি করবে।

→ যখনই match() যাবে তখনই parse tree তৈরি হতে থাকবে।



→ take next input character

Compilar Design (Neso-Academy - 28.06.25)

Syntax Analyser:

$S \rightarrow id = E;$ Expression can be assigned with id and by ;

$E \rightarrow E+T \mid T$ T → Term. Expression also can be a term.

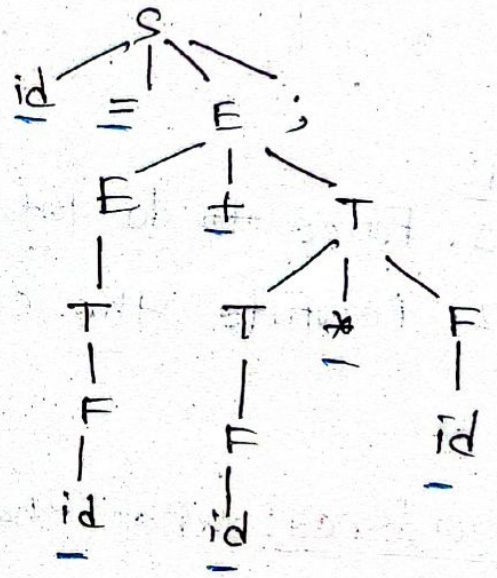
$T \rightarrow T*F \mid F$ F → Factor. Term can be a factor

$F \rightarrow id$ Factor may an identifier.

Terminal: id, =, ;, +, *

Non-terminal: S, E, T, F

id = id + id * id;



~~Semantic Analyser:~~

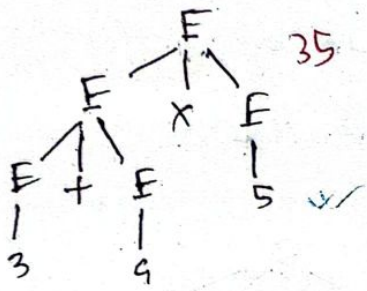
Errors:

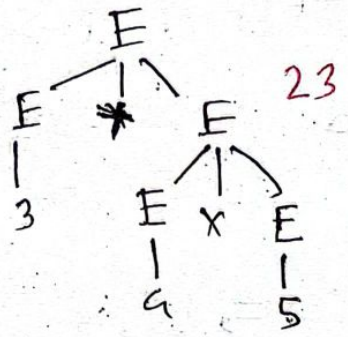
compile time

- lexical Errors
  - i) Too long identifier
  - ii) Too long constant
  - iii) ill-formed declar
  - iv) Illegal character
- Syntax Errors
  - i) Undeclair variable
  - ii) Type mismatch
  - iii) Return type mismatch
  - iv) Incompitable type expression
- Semantic Errors
  - i) Missing semicolon
  - ii) Mismatch bracket
  - iii) improper declaration
  - iv) Invalid Expression

Ambiguiti Problem in CFG:

Associativity: It define how operator of same precedence associat, groping are grouped when there are no parentheses to force grouping.

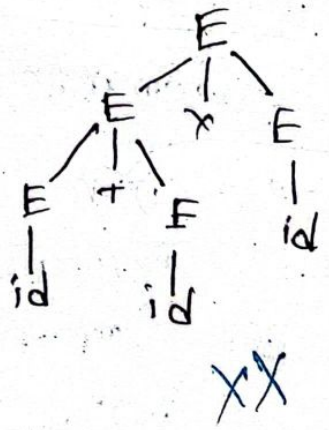




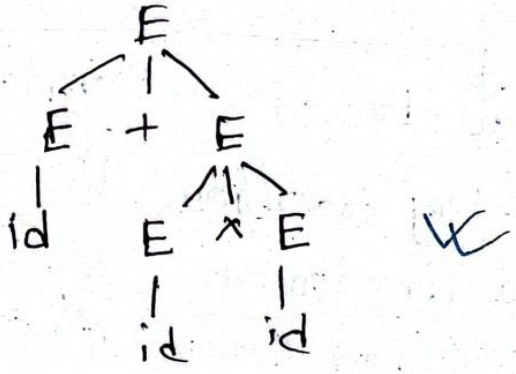
Sol:

1. Force to do left or righ associativity as nedded.
2. Rewrite the CFG to enforce associativity.

Precedence: when the grammer does not enforce operator precedence, can do lower precedence before higher precedence lead multiple parse tree that is ambiguous



Here plus is calculated before multiplication.



At first, in low level multiplicat is done first.

Elimination of Ambiguity:

1. Rewrite grammer to enforce precedence and associativy.

Before: $E \to E+E \mid E*E \mid (E) \mid id$

Rewrite: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

2. Elimination Left Recursion.

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

Left: Recursion

Now, $A \to \beta$

$A \to A\alpha$

$\to \beta\alpha$

$A \to A\alpha$

$\to A\alpha\alpha$

$\to \beta\alpha\alpha$

Produce: $\beta, \beta\alpha, \beta\alpha^2, \beta\alpha^3 \ldots$

RE: $\beta\alpha^*$

Right: Recursion

Now, $A \to \alpha A \mid \beta$

$A \to \beta$

$A \to \alpha A$

$\to \alpha\beta$

$A \to \alpha A$

$\to \alpha\alpha A$

$\to \alpha\alpha\beta$

Produce: $\beta, \alpha\beta, \alpha^2\beta, \alpha^3\beta \ldots$

RE: $\alpha^*\beta$

Why Top-Down Fails here?

Top-down parser use leftmost derivation it alway try to check from left to right so like $E \to E+T \mid T$, $T \to id$ this gramer $id+id$ input cannot start consume because top-down parser alway try left most derivation and stack in loop because there is no $E \to id$ direct rule to consume input.

Ex: $A \to Ab \mid Ac \mid Bd \mid Cd \mid Fc$

Sol: $A \to BdA' \mid CdA' \mid FcA'$

$A' \to bA' \mid cA' \mid \varepsilon$

Ex: $A \to \underbrace{Ab^*c}_{a\ \alpha} \mid \underbrace{A^*c}_{a\ \alpha} \mid \underbrace{ABc^*d}_{a\ \alpha} \mid \underbrace{cd}_{\beta} \mid \underbrace{Fc}_{\beta}$

Sol: $A \to CdA' \mid FcA'$

$A' \to b^*CA' \mid {}^*cA' \mid Bc^*dA' \mid \varepsilon$

Ex: $S \to \underbrace{S01S0S}_{a\ \alpha} \mid \underbrace{01}_{\beta}$

Sol: $S \to 01S'$

$S' \to 01S0SS' \mid \varepsilon$

Ex: $L \to \underbrace{L, S}_{a\ \alpha} \mid \underbrace{S}_{\beta}$

Sol: $L \to SL'$

$L' \to ,SL' \mid \varepsilon$

Left Factoring: Left factoring is the process to remove ambiguity caused by common prefix and make grammer suitable for top down parser. Make non-deterministic grammer to deterministic grammer. Now no need to backtrack

If $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \alpha\beta_4$, how to recognize $\alpha\beta_4$. it face common prefix problem

Sol: $A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3$

Ex: $S \to \underline{a}ssbs \mid \underline{a}sasb \mid \underline{a}bb \mid b$

Sol: (i) $S \to aS' \mid b$

$S' \to \underline{s}sbs \mid \underline{s}asb \mid bb$

(ii) $S' \to sS'' \mid bb$

(iii) $S'' \to sbs \mid asb$

Ex: $S \to \underline{a} \mid \underline{ab} \mid \underline{a}bc \mid \underline{a}bcd$

Sol:

(i) $S \to aS'$

$S' \to \epsilon \mid \underline{b} \mid \underline{bc} \mid \underline{bcd}$

(ii) $S' \to \epsilon \mid bS''$

$S'' \to \epsilon \mid \underline{c} \mid \underline{cd}$

(iii) $S'' \to \epsilon \mid cS'''$

(iv) $S''' \to \epsilon \mid d$

First Follow Example:

	First	Follow
$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$\{g, f, h\}$
$C \to bC \mid \epsilon$	$\{b, \varepsilon\}$	$\{g, f, h\}$
$D \to EF$	$\{g, f, \varepsilon\}$	$\{h\}$
$E \to g \mid \varepsilon$	$\{g, \varepsilon\}$	$\{f, h\}$
$F \to f \mid \varepsilon$	$\{f, \varepsilon\}$	$\{h\}$

	First	Follow
$E \to TE'$	$\{id, (\}$	$\{\$,)\}$
$E' \to \varepsilon \mid +TE'$	$\{\varepsilon, +\}$	$\{\$,)\}$
$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
$T' \to \varepsilon \mid *FT'$	$\{\varepsilon, *\}$	$\{+, \$,)\}$
$F \to id \mid (E)$	$\{id, (\}$	$\{*, +, \$,)\}$

$S \to Bb \mid cd$	$\{a, b, c, d\}$	$\{\$\}$
$B \to aB \mid \varepsilon$	$\{a, \varepsilon\}$	$\{b\}$
$C \to cC \mid \varepsilon$	$\{c, \varepsilon\}$	$\{d\}$

$S \to AaAb \mid BbBa$	$\{a, b\}$	$\{\$\}$
$A \to \varepsilon$	$\{\varepsilon\}$	$\{a, b\}$
$B \to \varepsilon$	$\{\varepsilon\}$	$\{b, a\}$

Predictive Parser:

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to \varepsilon \mid +TE'$	{ε, +}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to \varepsilon \mid *FT'$	{ε, *}	{+, $,)}
$F \to id \mid (E)$	{id, (}	{*, +, $,)}

Size of table:

Non-terminal * (Terminal+1)

LL(1) Parsing Table:

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

rules:

1. Variable থেকে terminal এ যে production rule থাকে first আসে, যার first এর terminal এ production rule হবে।
2. যদি ε ∈ first হয় তবে variable এর follow সকলোতে production rule বসবে।

⊞ check id+id$ is accepted using stack.

Stack	input	Production rule
$E	id+id$	E→TE'
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id
$E'T'id	id+id$	pop
$E'T'	+id$	T'→ε
$E'	+id$	E'→+TE'
$E'T+	+id$	pop
$E'T	id$	T→FT'
$E'T'F	id$	F→id
$E'T'id	id$	pop
$E'T'	$	T'→ε
$E'	$	E'→ε
$	$	Accept

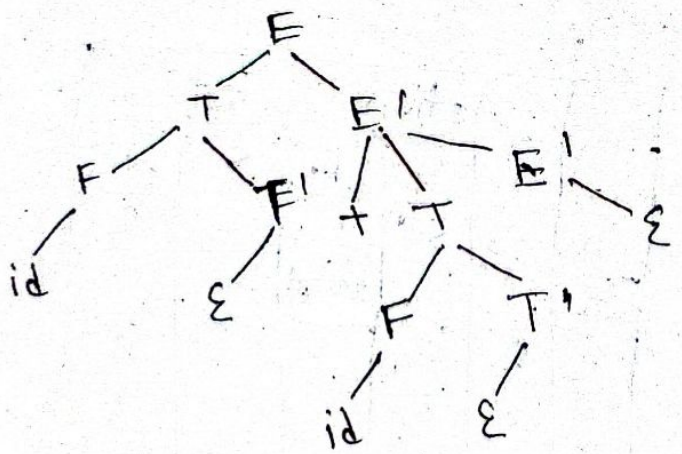
$→ mean bottom of stack
E→ start variable
[Replace in stack reverse]

id, id match

→ ε হলে শুধু rule remove ↳ stack

→ pop হলে stack, input উভয়ের remove

$, $ match → accept



Make parse tree using production rule from top to bottom

For input: id+id*id

Bottom up Parser:

1. Make augment grammer
2. Put scanner dot (.) before each rules.
3. Explore (closure) all child at same time from left to right after scanner.
4. Go to depth explore and then back-track to explore sibling.

$S \to A|B|C$
$A \to D|E$
$D \to F$
$B \to C|D$
$O \to N$
$C \to a$

$\longrightarrow$

$S' \to .S$
$S \to .A$
$S \to .B$
$S \to .C$
$A \to .D$
$A \to .E$
$B \to .C$
$B \to .O$
$O \to .N$
$C \to .a$

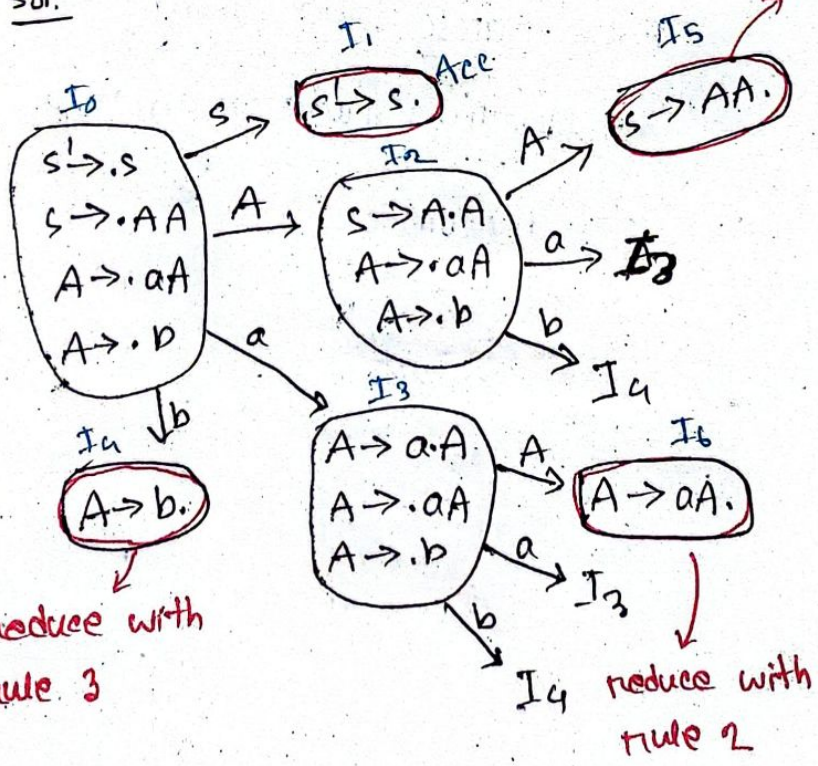
← It always try to go depth and explore more rather than explore sibling equally.

LR(0)

Ex:

$S \to A^{①}A$ → রুল নামকরন
$A \to a^{②}A|b^{③}$

Sol:



	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

→ ভরে যাবে

LR(0) → Table

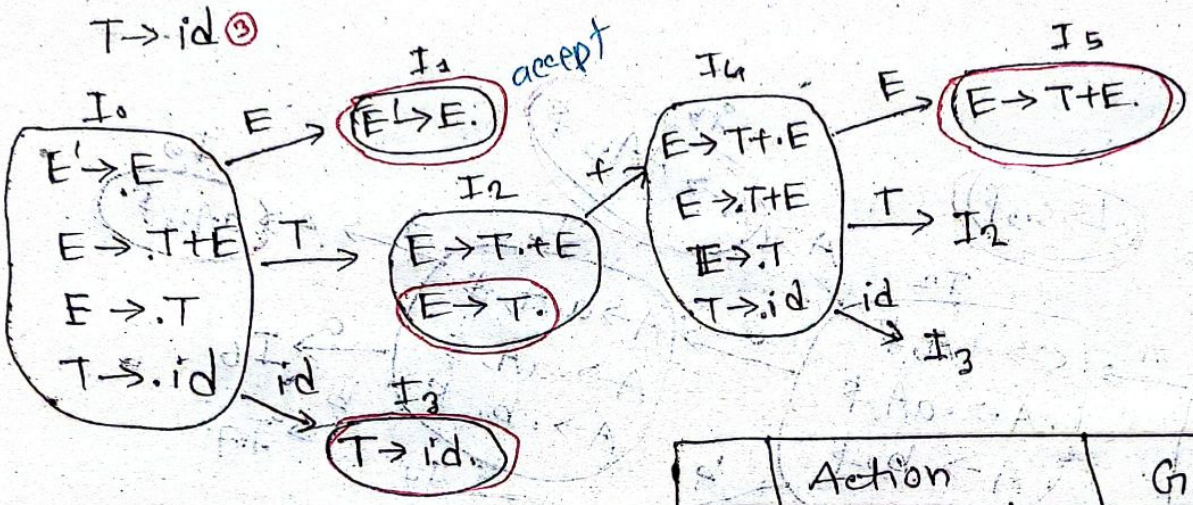
একই ঘরে shift এবং reduce হলে LR(0) হবে না।

SLR(1) LR(0) + follow

কিন্তু conflict reduce হওয়ার পরও আবার conflict থাকতে পারে, তখন সেটা SLR(1) হবে না।

$E \to T+E \mid T$ ①

$T \to id$ ③



	Follow
$E \to T+E \mid T$	$\{\$\}$
$T \to id$	$\{+, \$\}$

	Action			Goto	
	id	+	$	E	T
0	S_3			1	2
1			accept		
2		S_4	r_2		
3		r_3	r_3		
4	S_3			5	2
5			r_1		

1. shift স্বাভাবিক করবে।
2. reduce এর জন্য যে reduce হওয়া production rule এর বাম পাশের variable এর follow তে যে production rule, reduce

id+id$
~~aabb$~~ এরকম হবে কী না?

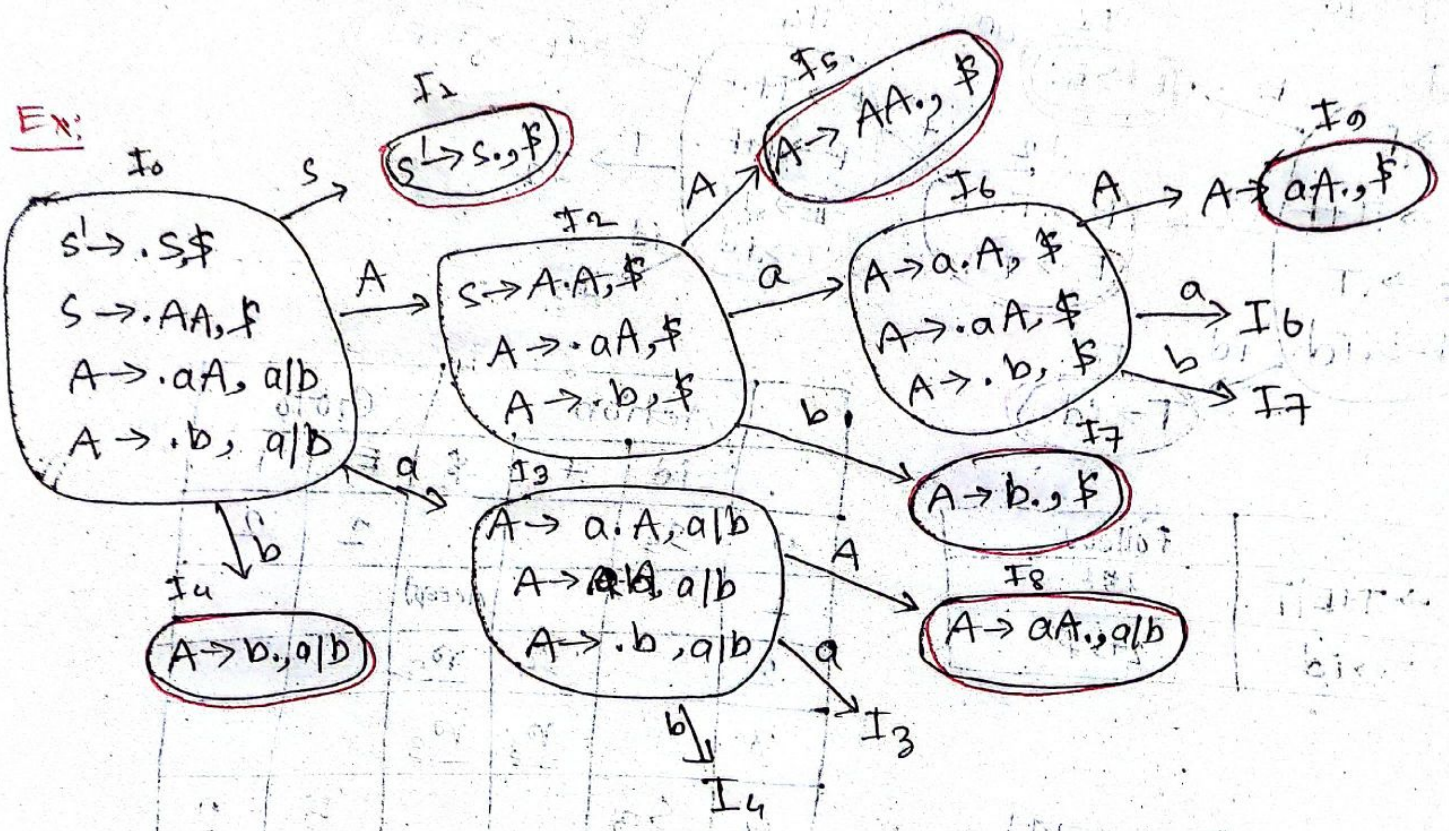
0	~~id~~	~~3~~	T	2	~~+~~	~~4~~	~~id~~	~~3~~	~~T~~	~~2~~	~~E~~	~~5~~	E	1

1. stack এর সাথে scanner মিলিয়ে, shift হলে scanner সরাবে, shift চালু করবে।
2. reduce হলে যে production rule | left side | × 2 time reduce হবে। (→ all)
   যে rule এর left variable stack a যাবে। যে left value আর তার previous number হিসাব করে stack এ বসাবে।

CLR(1): LR(0) + look ahead

→ যা explore করব তারপর যা থাকবে তার first.

Ex:



	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

→ lookahead এর ঘরে reduce production rule বসাবো।

LALR(1) এসব I এর সবকিছু একই কিন্তু lookahead আলাদা তারা merge

	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S6	S7			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
36	S36	S47			89
47	r3	r3	r3		
89	r2	r2	r2		
89	r2	r2	r2		

→ একই রকম শুধু lookahead আলাদা গুলোকে merge করব। যব একই কিন্তু lookahead আলাদা।

	Action			Goto	
	a	b	$	S	A
36	S36	S47			89
47	r3	r3	r3		
89	r2	r2	r2		

→ এদের কোনো combination এ আগারও মার্জ করা হলে সেগুলো লিখব।

→ যাদের নিয়ে মার্জ করা হয়েছিল সেখানে replace করব

Operator precedence parse tree: Make parse tree deals with operator & parenthese

1. No epsilon in grammer. (No null production)
2. No two adjacent two terminal or non-terminal. in RHS.

Step:

i) Make operator precedence relation table.

ii) Make operator precedence table

iii) Function graph.

iv) Function table.

Ex: $E \to E+E \mid E*E \mid id$

Make table for terminal in row & column

F | G	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	–	>
$	<	<	<	A accept

Relation Table:

→ For same operator left is higher precedence ++

→ id, x, y, z or terminal are most high precedence

→ $ is always low power.

→ id id cannot compair

$E \to E+E \mid E*E \mid id$

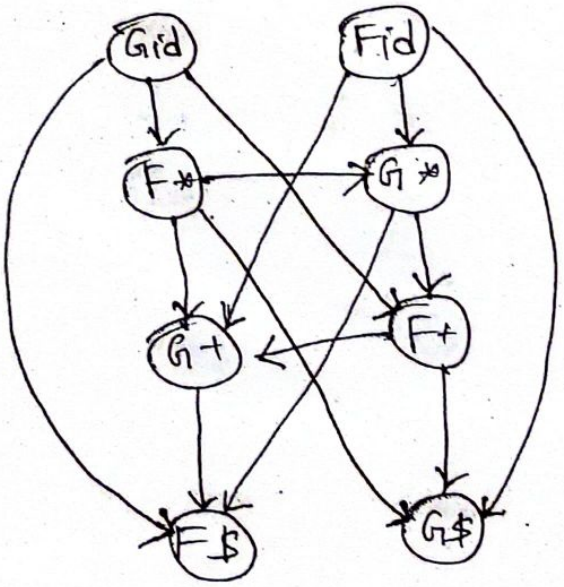
Stack	Relation	input	comment
$	<	id+id*id $	Shift → id
$id	>	+id*id $	Reduce → E
$ E	<	+id*id $	Shift → +
$ E+	<	id*id $	Shift → id
$ E+id	>	*id $	Reduce → E
$ E+E	<	id $	Shift → *
$E+E*	<	id $	Shift → id
$E+E*id	>	$	Reduce → E
$ E+E*E	>	$	Reduce → E*E
$ E+E	>	$	Reduce → E+E

→ input precedence বেশি হলে shift. কম হলে reduce

→ Shift বা reduce যাই থাকে সেটা stack এ push হবে।

→ Reduce এর ক্ষেত্রে Left side variable stack এ যাবে।

Relation table থেকে function graph এ function table



→ Relation table থেকে যার precedence বেশি তার থেকে যার কম সেদিকে যাবে।

	+	*	id	$
F	2	4	4	0
G	1	3	3	0

→ column এর variable থেকে row এর terminal এ highest longest path এর length বসাবে।

Notational Framework:

Notation: System of symbols that represents concepts, action or data.

Notational Framework: Structure or sytematic methodology or methodology that governs how symbol design, represented and used.

SDT: Syntax directed translation scheme.

Grammer + subroutine/action/ semantics

Intermediate code generation:

Classification:

1. Postfix notation.
2. Syntax tree
3. Three address code
   i) Quadraple
   ii) Triple

Question solve Theory:

A Finite state machin is a mathematical model of computer consist of Five tuples.

$M = (Q, \Sigma, \delta, q_0, F)$

$Q \rightarrow$ Finite Set of State

$\Sigma \rightarrow$ Finite Set of symbol

$\delta \rightarrow$ Transition Function

$q_0 \rightarrow$ Start Symbol

$F \rightarrow$ Final state.

Q: If any ambigious gramer given How make LL(1) table

Ex: $E \rightarrow (E) \mid E+E \mid E-E \mid E*E \mid id$

Step:

1. Eliminate ambugity $\rightarrow$ using operator precedence or associativity.
2. Eliminate Left Recursion.
3. If have common prefix, make left factoring.
4. Compute First() and Follow set
5. Make LL(1) table.

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

$E \to E+T \mid E-T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

	First( )	Follow( )
$E \to TE'$	$\{(, id\}$	$\{\$,)\}$
$E' \to +TE' \mid -TE' \mid \varepsilon$	$\{+, -, \varepsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{(, id\}$	$\{+, -, \$,)\}$
$T' \to *FT' \mid \varepsilon$	$\{*, \varepsilon\}$	$\{+, -, \$,)\}$
$F \to (E) \mid id$	$\{(, id\}$	$\{*, +, -, \$,)\}$

	id	+	−	*	(	)	$
E	$E \to TE'$				$E \to TE'$		
E'		$E' \to +TE'$	$E' \to -TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$				$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$				$F \to (E)$		

四 Using SDT construct syntax tree

$E \to E + T$ {E.npt := mkn (E.npt, "+", T.npt}

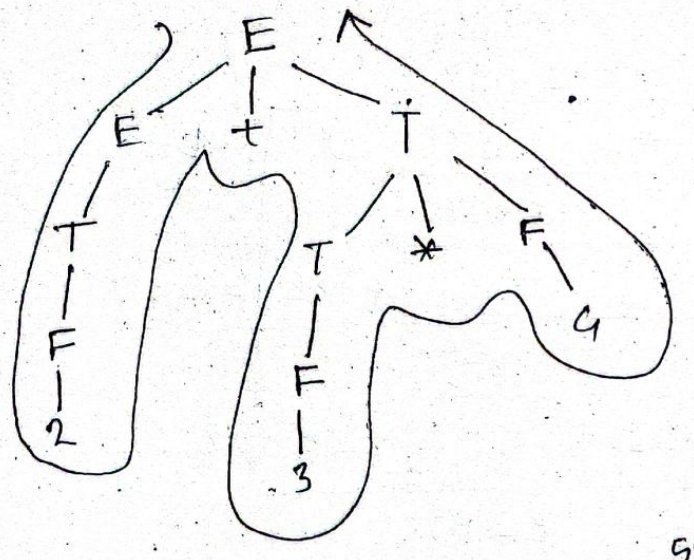
$\mid T$ {E.npt := T.npt}

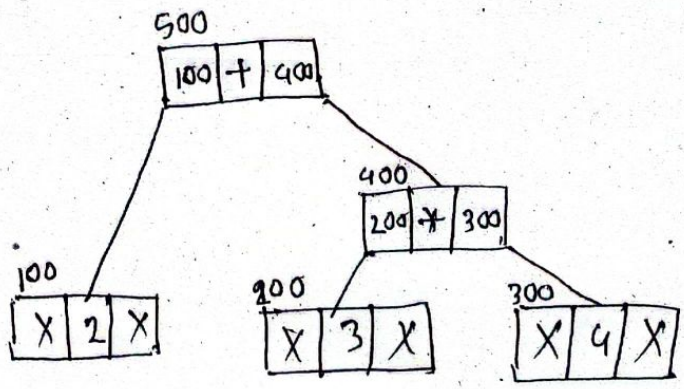
$T \to T * F$ {T.npt := mkn (T.npt, "*", F.npt}

$\mid F$ {T.npt := F.npt}

$F \to$ {digit F.npt = mkn (null, digit, null)}

w = 2+3*4





Using SDT produce three address code for x=a-b/c

$S \rightarrow id = E$ {id.npt = E.npt}

$E \rightarrow E - T$ {E.npt := mkn(E.npt, "-", T.npt}

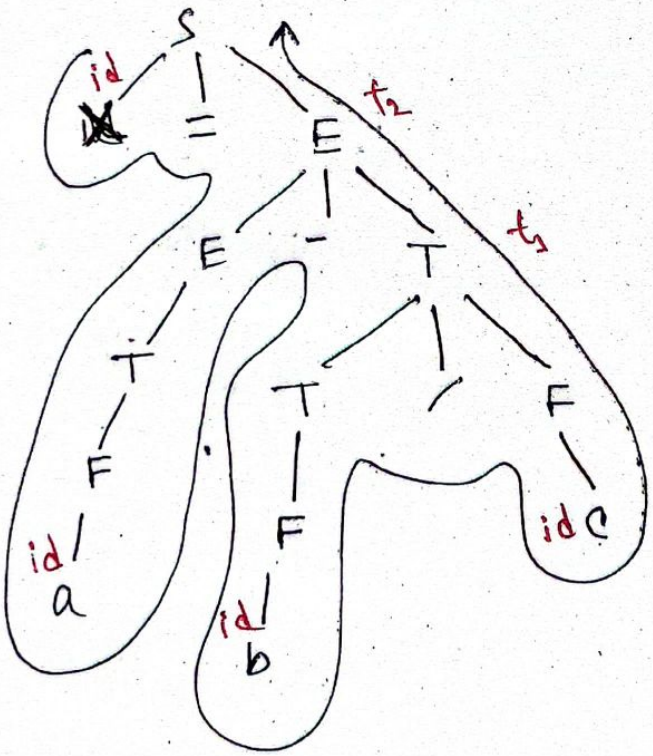
$| \; T$ {E.npt := T.npt}

$T \rightarrow T / F$ {T.npt := mkn(T.npt, "/", F.npt}

$| \; F$ {T.npt := F.npt}

$F \rightarrow id$ {F.npt := id.}

x = a-b/c



$t_1 = b/c$

$t_2 = a - t_1$

$x = t_2$

Q: Explain operator precedence parsing for an expression a+b*c with the following grammer.

$E \to EAE \mid a \mid b \mid c$, $A \to + \mid *$

Sol:

$E \to E+E \mid T$

$T \to T*F \mid F$

$F \to a \mid b \mid c$

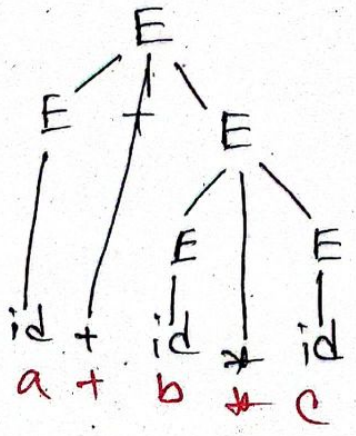
Operator Precidence table

	id	*	+	$
id	Error	>	>	>
*	<	>	>	>
+	<	<	>	>
$	<	<	<	Accept

a+b*c$

$	a	+	b	*	c		

stack	relation	input	rules



এভাবেও accept কিনা চেক করা যায়।

Q: Convert CFG to CNF

$S \to ASaBA$, $A \to B|S$, $B \to b|\varepsilon$

Sol:

$S_0 \to S$

$S \to ASaBA$

$A \to B|S$

$B \to b|\varepsilon$

(Remove $B \to \varepsilon$)

$S_0 \to S$

$S \to ASaBA|ASaA|$

$A \to B|S|\varepsilon$

$B \to b$

(Remove $A \to \varepsilon$) → ওপরের যেকোন A এর জায়গায় replace হবে।

$S_0 \to S$

$S \to ASaBA|ASaA|SaBA|ASaB|SaB|SaA|ASa|Sa$

$A \to B|S$

$B \to b$

(Remove $S_0 \to S$, $A \to S$, $A \to B$ Unite production)

$S_0 \to ASaBA|ASaA|SaBA|ASaB|SaB|SaA|ASa|Sa$

$A \to b|ASaBA|ASaA|SaBA$ ------

$B \to b$

Now make two non terminal who are long.

Q: Draw the syntax tree and parse tree for the expression

$(A+B/C)/(A-C/F)*F+(H*Y*Z)$

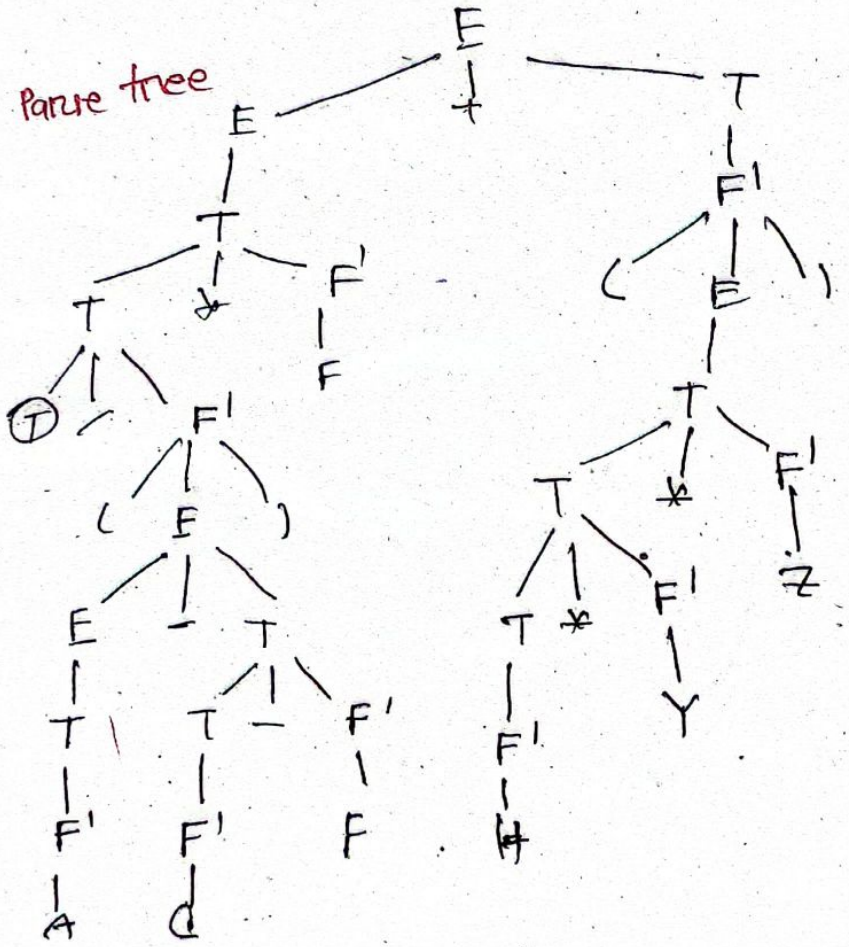
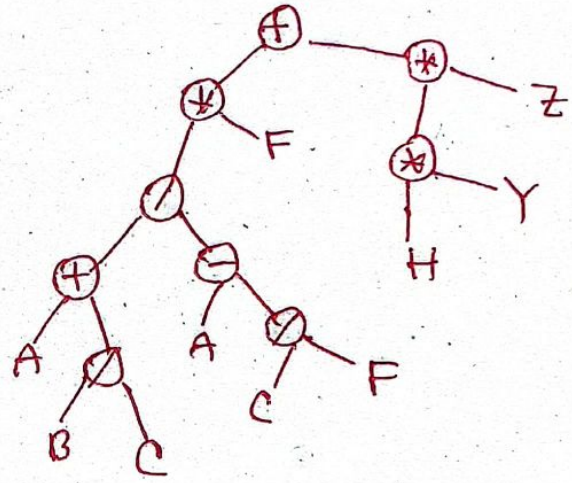
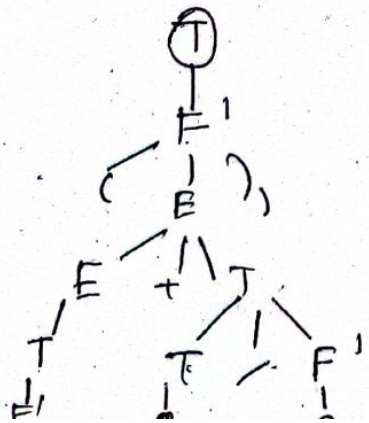
Sol: let the grammer be

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

$T \rightarrow T*F' \mid T/F' \mid F'$

$F' \rightarrow (E) \mid A \mid B \mid C \mid F \mid H \mid Y \mid Z$

Parse tree



Syntax tree