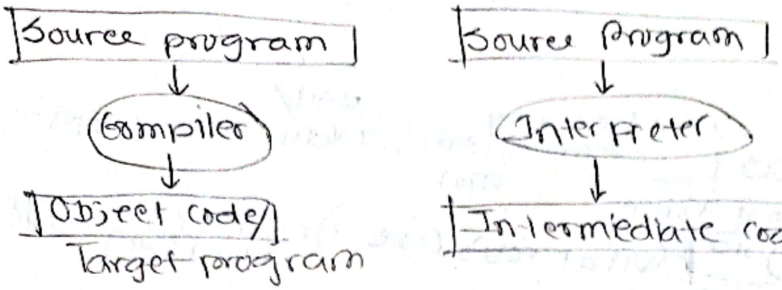
Date: 13.04.25

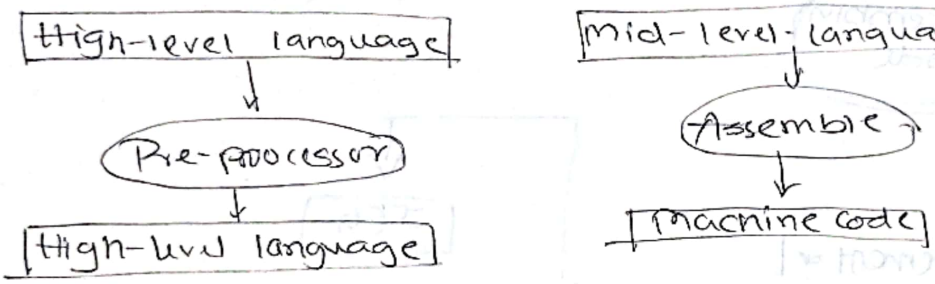


* Compiler vs Interpreter

- whole program
- Fast
- Platform dependant
- large and compux

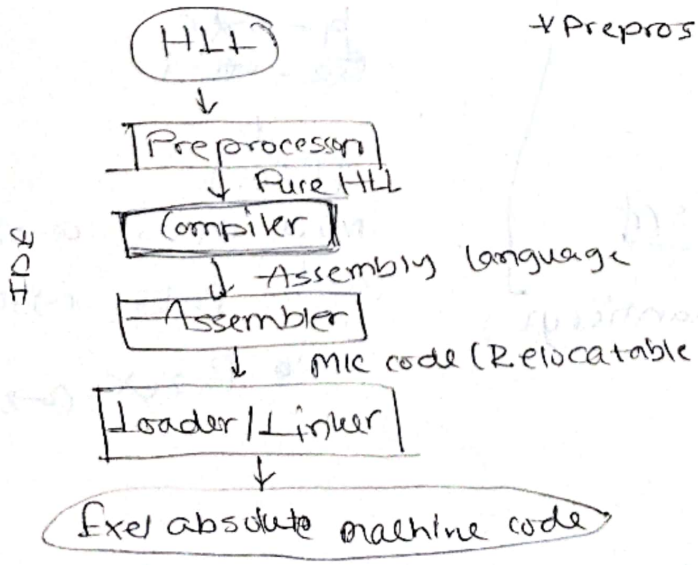
- one line at a time
- slow
- Independent
- Simple, short programs
- High/mid level language

* other translators:



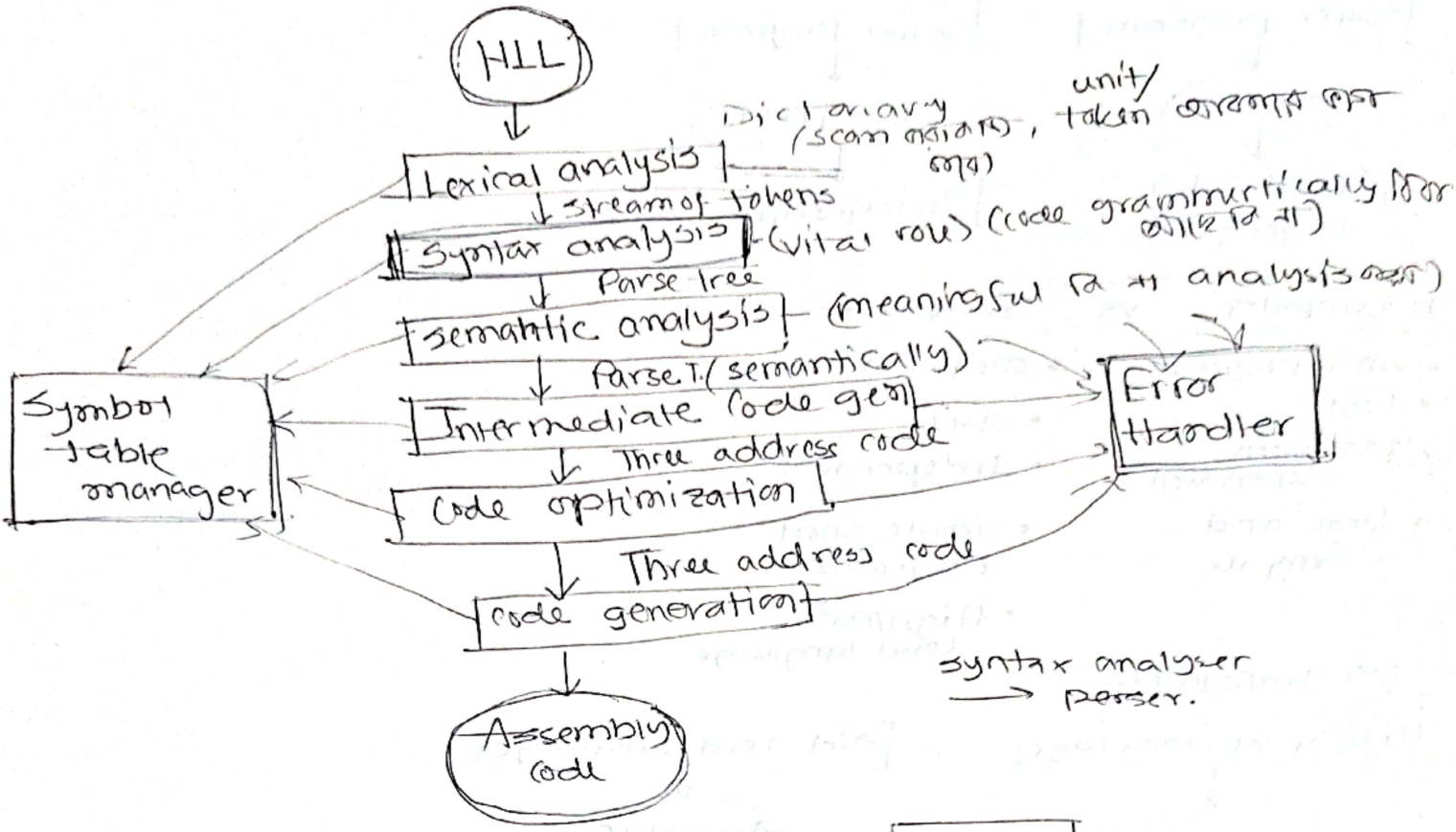
* Source code, object code, Byte code, machine code
  - Source code: input to compiler
  - Byte code: code that can be executed by virtual machine (platform independent)
  - machine code: Directly executable by computer's physical processor without other translators.

Execution of a Program:



* Preprossing

* Phases or structure of compiler:



$x = a + b * c$; /* statement */

↓

Logical analysis

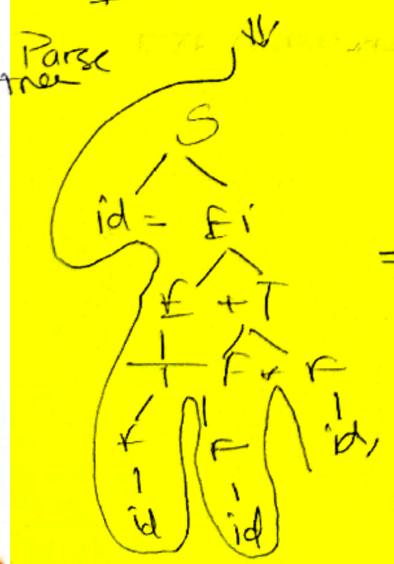
↓

id = id + id * id

↓

Systax analysis ⇐ S → id = E ; E → E+T/T ; T → T*F/F ; F → id.

Parse tree



⇒ Semantic analysis (Parse tree semantically)

CFG

((i+d)*

↓

ICG

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

↓

CO

$t_1 = b * c$

$t_2 = a + i$

↓

TCG

mul R1, R2 a → R0

add R0, R2 c → R0

mov R2, x R0 → x

* compiler must
  - Preserve the meaning
  - improve source code
  - Speed • Space
  - Feedback.

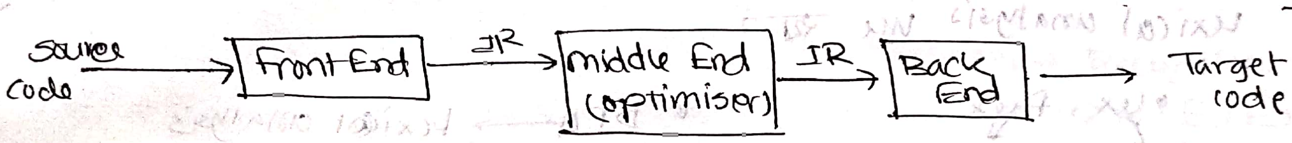
Date: 20.04.2025

* Lexical Analysis (Scanning)

  output → token and is pair of the form <type, lexemes> or <token-class, atribute>

* Intermediate code generation
  - 3-address code → 3টার বেশি variable use করা যাবে না

* code optimisation: to improve intermediate code.
  - IR → Intermediate Representation



* code generation phase এর সাথে processor + related register বা mechinary যুক্ত।

- যে phase গুলো analysis করে → Front end of compiler
- যে phase গুলো মূল implementation/ Synthesis করে → Back end of compiler.

* mxn compilers with m+n components:
* Qualities of a Good compiler:
  - generates correct code (first and foremost)
  - " fast code
  - conforms to the specification of the input language.

* principles of compiler:
* Historical Notes:

Scanned with OKEN Scanner

Introduction to Lexical Analysis:

Language → set of scripts → letters ● Alphabet
ASCII character
numerical, other logical terms

Sentence / Statement → Always infinite.

Grammer (T)
words: 1 user defined identifiers, keywords,
can be finite / infinite

- Token → sequence of characters
- Lexem → sequence of characters, matched by a pattern
- Pattern → Rule of description

* Tokenization:
→ * Attribute in token
* Why all this? (Lexical) → Why study lexical analysis?

* manually by hand কিভাবে এ ignore করা difficulty lexical analysis use করা

- Lex, Flex (unit based)

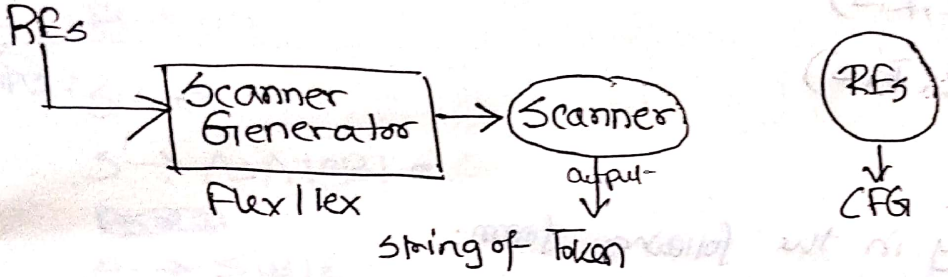
* DFA → Lexical analysis. (Deterministic)

* Scanner generator (Full story)!
- Thomson's construction.
- Hopcroft's algorithm.

Date: 24.05.2025

Lexical structures of tokens:

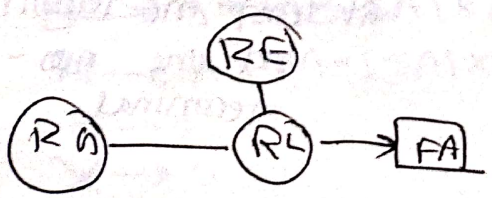
RE
↓
Lexical Analyser (FSA) / (FSM)
↓ (Finite state Automata / Machine).
FSAs ⟺ RE, RG ⟺ FSA (Finite state Acceptor)



* Syntax analysis:

* Context free-Grammer (CFG): → Natural + Recursive
① * Backus Naur form (BNF): E → E+E / E*E.
↓ not only CFG, all other grammers. * CFG ⊆ subset RG.
Symbol ← Terminals (Σ) / Non Terminals (V) + Rules of production (S)
Touples: G = (V, Σ, S, S) ← CFG grammer → Easy to understand + Recursive
② * Chomskey Normal form (CNF):

* CYK Algorithm → (ε notation) use হয় → CNF.
Cocke Younger Kassami



Scanned with OKEN Scanner

27.09.25

+ CFG BNF

A special way to write CFG introduced by Noam Chomsky is called Chomsky Normal Form (CNF)

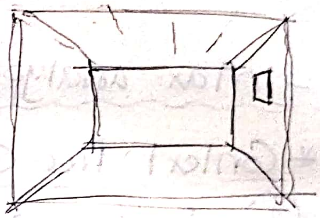
A CFG say (G) is in CNF say G'.

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

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

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

R contains only in the following form:



i) $A \to a$; $A \in V$

ii) $A \to BC$; $A, B, C \in V \setminus \{S\}$

iii) $S \to \varepsilon$;

"Bad rules": ← violations of Chomsky inside CFG.

(i) $A \to a$
(ii) $A \to BC$
(iii) $S \to \varepsilon$.

} এই তিনটি ছাড়া বাকি violation.

1. $A \to wSw$; $A \in w$; $w \in (V \cup \Sigma)^*$
"Start symbol rule".

2. $A \to \varepsilon$; $A \in V \setminus \{S\}$.
"epsilon rule"

3. $A \to B$; $A, B \in V$.
"unit rule"

4. $A \to w$; $A \in V$ and $w \in (V \cup \Sigma)^*$; w must contain at least one terminal and one non-terminal.
"mixed rule"

5. $A \to w$; $w \in V^*$ and $|w| > 2$.
"long rule"

Scanned with OKEN Scanner

→ Transformation of CFG to CNF: R → set of production rules.

R:
$S \to ASA \mid aB$
$A \to B \mid S$
$B \to b \mid \varepsilon$

R_1: (Fixing "start symbol rule")
$S_1 \to S$
$S \to ASA \mid aB$
$A \to B \mid S$
$B \to b \mid \varepsilon$

R_2:
$S_1 \to S$
$S \to ASA \mid aB \mid a$
$A \to S \mid B \mid \varepsilon$
$B \to b$.

R_3:
$S_1 \to S$
$S \to ASA \mid AS \mid SA \mid aB \mid a$.
$A \to S \mid B$
$B \to b$

R_4:
$S_1 \to ASA \mid AS \mid SA \mid aB \mid a$.
$S \to ASA \mid AS \mid SA \mid aB \mid a$.
$A \to ASA \mid AS \mid SA \mid aB \mid a \mid b$.
$B \to b$

R_5:
$S_1 \to ASA \mid AS \mid SA \mid XB \mid a$.
$S \to ASA \mid AS \mid SA \mid XB \mid a$.
$A \to ASA \mid AS \mid SA \mid XB \mid a$.
$B \to b$
$X \to a$

R_6:
$S_1 \to AY_1 \mid AS \mid SA \mid XB \mid a$
$S \to AY_1 \mid AS \mid SA \mid XB \mid a$
$A \to AY_1 \mid AS \mid SA \mid XB \mid a$
$B \to b$.
$X \to a$
$Y \to SA$

$A \to BCDE$
$A \to BY_1$
$Y_1 \to CY_2$
$Y_2 \to DE$

Scanned with OKEN Scanner

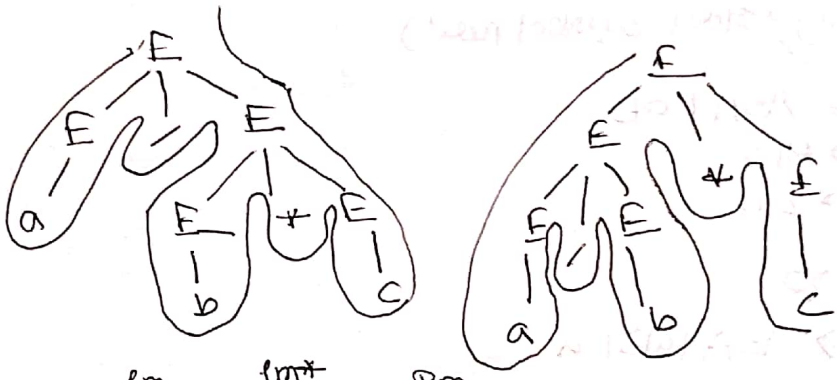
Date: 04.05.2025

Syntax Analyzer / Parser:

(a) If checks the stream of tokens is syntactically correct or not.

a+/b;
id+/id;

(b) a/b*c;
i)



Derivation: Denoted by: $\overset{lm}{\Rightarrow}$ বা $\overset{lm*}{\Rightarrow}$ বা $\overset{Rm}{\Rightarrow}$
is a process to confirm if a string is from a language.

1. E → I
2. E → E+E
3. E → E*E
4. E → (E)
5. I → a
6. I → 5
7. I → Ia
8. I → Ib
9. I → I0
10. I → I1.

language.

"a*(a+b00)" → এই string, language এর মধ্যে কি না?

(lm → left most) (Rm → Right most)

$E \overset{lm}{\Rightarrow} E*E$
$\overset{lm}{\Rightarrow} I*E$
$\overset{lm}{\Rightarrow} a*E$
$\overset{lm}{\Rightarrow} a*(E)$
$\Rightarrow a*(E+E)$
$\Rightarrow a*(E+I)$
$\Rightarrow a*(a+I)$
$\Rightarrow a*(a+I_0)$
$\Rightarrow a*(a+I_{00})$
$\Rightarrow a*(a+b00)$

left most sentential form

$E \overset{Rm}{\Rightarrow} E*E$
$\Rightarrow E*(E)$
$\Rightarrow E*(E+E)$
$\Rightarrow E*(E+I)$
$\Rightarrow E*(E+I_0)$
$\Rightarrow E*(E+I_{00})$
$\Rightarrow E*(E+b00)$
$\Rightarrow E*(I+b00)$
$\Rightarrow E*(a+b00)$
$\Rightarrow I*(a+b00)$
$\Rightarrow a*(a+b00)$

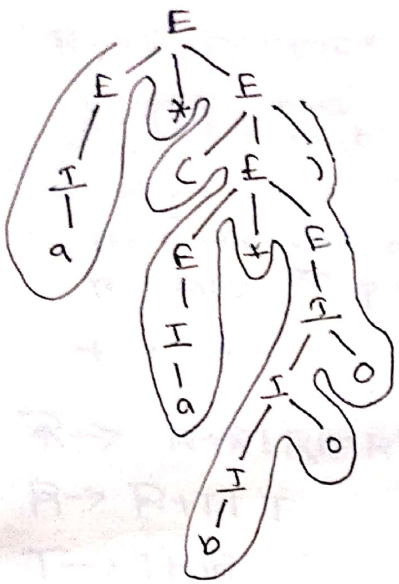
right sentetial form

same

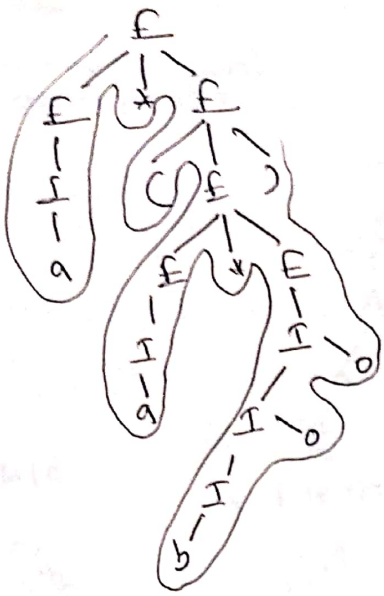
Non terminals and/or terminals

Scanned with OKEN Scanner

LM



RM



sparse tree দুটি same-ই হচ্ছে because bracket দিয়ে আলাদা করে রাখছে।

Scanned with OKEN Scanner

Date: ২৬/০৫/২৫

$R \rightarrow R+R \mid RR \mid R^*$ (Regular expression) generate করা

$\Sigma = \{\rho\}$ $r = a$, $r_2 = b$

ε, ρ

$r = r_1 + r_2 = a+b$ a/b

$r = r_1 r_2 = ab$ φ

+ · *

$R \rightarrow R+R \mid RR \mid R^* \mid a \mid b \mid c$

$R \rightarrow R+T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow R^* \mid a \mid b \mid c$

infinite loop এ যাবে

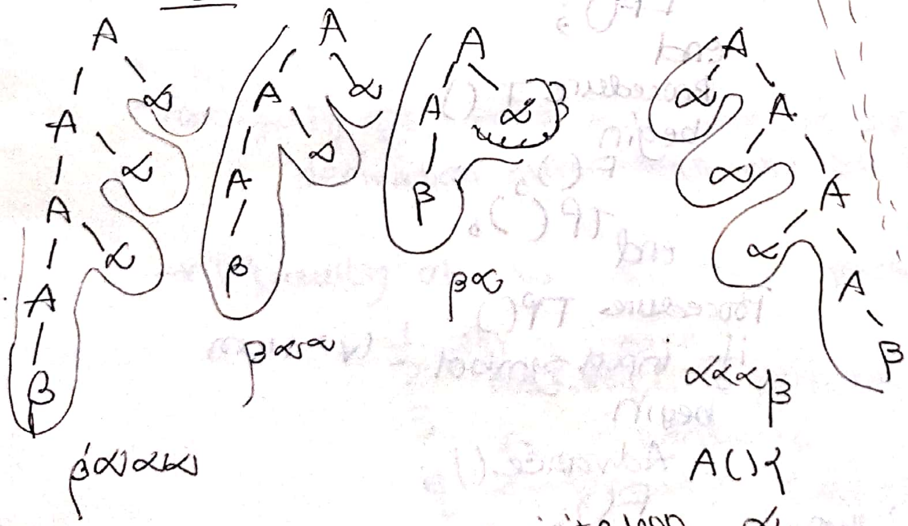
Recursive grammer (RG)

Left RG (LR) — Right RG (RR)

LR: $A \rightarrow A\alpha \mid \beta$

RR: $A \rightarrow \alpha A \mid \beta$

LR — RR



$\beta\alpha\alpha\alpha\alpha$

$L = \{\beta\alpha^n \mid n \geq 0\}$

A(){ α; A(); }

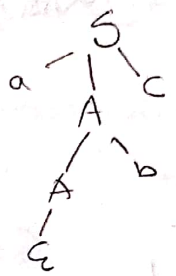
infinite loop হবে না।

A(){ A(); ← infinite loop এ চলে যাবে
 β; }

$S \rightarrow aAc$

$A \rightarrow Ab \mid \varepsilon$

$L = \{ab^nc \mid n \geq 0\}$.



* Elemination of left-recursive:

$A \rightarrow A\alpha \mid \beta$

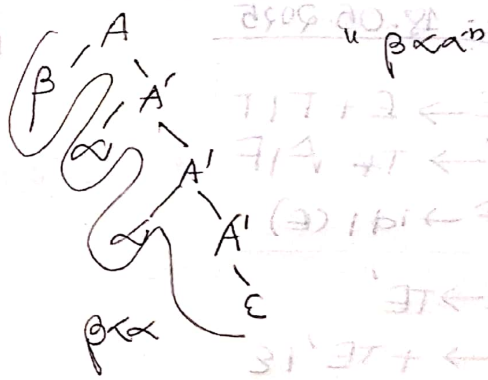
$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$A \rightarrow \alpha A \mid \beta$.

$L = \beta\alpha^*$

$L = \alpha^*\beta$.



$E \rightarrow E+T \mid T$

$T \rightarrow id$

$E \rightarrow TE'$

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

$T \rightarrow id$.

same

$R \rightarrow TR'$

$R' \rightarrow +TR' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow FT' \mid \varepsilon$.

$F \rightarrow F^* \mid a \mid b \mid c$

loop এ যাবে না

$\alpha, \beta \in (V \cup T)^*$

Scanned with OKEN Scanner

$A \to A\alpha_1 | \beta_1 | A\alpha_2 | \beta_2$

e.g. $A \to A\alpha_1 | A\alpha_2 | \ldots A\alpha_m | \beta_1 | \beta_2 \ldots \beta_n$

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

$A' \to \alpha_1 A' | \alpha_2 A' | \ldots \alpha_m A' | \varepsilon$

$A \to B\alpha | c$

$B \to A\beta | D$

$A \to (A\beta | D)\alpha | c$.

$A \to A\beta\alpha | D\alpha | c$

$A \to D\alpha A' | cA'$

$A' \to \beta\alpha A' | \varepsilon$

Date: 18.05.2025

$E \to E+T | T$

$T \to T*F | F$

$F \to id | (E)$

$E \to TE'$

$E' \to +TE' | \varepsilon$

$T \to FT'$

$T' \to *FT' | \varepsilon$

$F \to id | (E)$

~~Procedure~~

$W =$ "id+id*id"

* Recursive descent parsing
  - ⟶ LL parser.

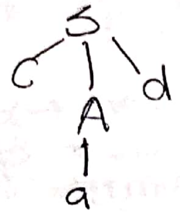
```
Procedure E();
begin
   T();
   EP();
end
Procedure EP();
 if input-symbol == '+' then
 begin
    Advance();
    T();
    EP();
 end
 Procedure T()
 begin
    F();
    TP();
 end
Procedure TP()
 if input symbol = '*' then
 begin
    Advance();
    F();
    TP();
 end
Procedure F();
 if input-symbol = 'id';
   Advance();
```

```
else if input-symbol = '(' then
 begin Advance();
  F();
  if . .
```

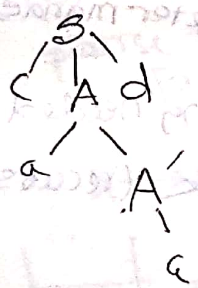
Scanned with OKEN Scanner

$S \rightarrow cAd$
$A \rightarrow ab | a$
$W = $ 'cad'

$S \Rightarrow cAd$
$\Rightarrow cad$.



$S \rightarrow cAd$
$A \rightarrow a | ab$
$S \rightarrow cAd$.
$A \rightarrow aA'$
$A' \rightarrow b | \epsilon$



$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$
'$\alpha\beta_3$'

Non-deterministic → না থামলে fail করার সম্ভাবনা থাকে।
left-factoring

$A \rightarrow \alpha(\beta_1 | \beta_2 | \beta_3)$
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 | \beta_2 | \beta_3$

$S \rightarrow iEtS | iEtSeS | a$
$S \rightarrow iEtSS' | a$
$S' \rightarrow \epsilon | eS$.
$S \rightarrow bS(SaaS | SaSb | b) | b$

$S \rightarrow bSS' | b$
$S' \rightarrow Sa\ S'' | b$
$S'' \rightarrow aS | Sb$

LL grammer

* Top-down.
* Ambiguity কর্ যাইলে more than one spase-tree Generate হবে, derivation more than one হলে হবে না।

* Difficulty of the top-down parsing:
  1. left recursive
  2. Backtracking problem
  3. Error-handling
  4. syntaxically correct or wrong.

Date: 22.05.2025

• Top down parsing style

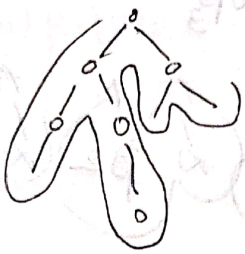
CFG
- AG
- UG
  - LR
  - RR

Eliminate left recursion ↓ LL —left factoring→ LL(1) . . . LL(K)

LL: Leading left to right

LL(1) ↳ Non-recursive decent parser.

LLParser with back tracking



$S \to aB | aC$

$B \to b$

$C \to c$

first(S) = first(aB) ∪ first(aC) , aB ∩ ac ≠ φ (common prefix আছে)

(~~aB~~)

aB ∩ ac = φ → non deter mánaeom

⇓

Backtracking needed

↳ Stack (recursive)

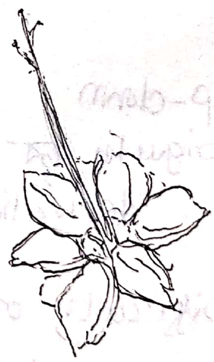
Properties of LL(1) grammer:

* Un-ambiguous.
* Non-left-recursive.
* Determinístic.

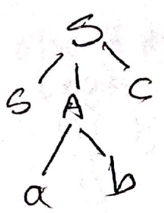
Predictive perser: LL(1) ছিল এ perser.

Buttomup perser / Table driven perser /

↳ Look ahead



* Predictive perser: Predictive is a non-cursive table. driven ~~per~~ top-down perser, that maintain stack activation explecitly by parser it-self.

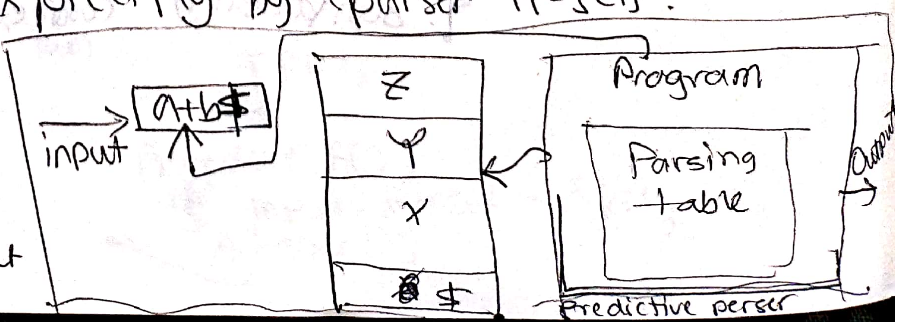


bac

↑

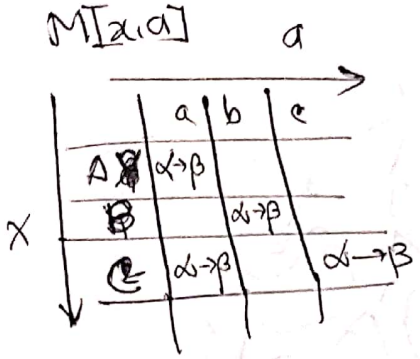
ab|b

$ → end input



Date:

M[x,a]



parse table

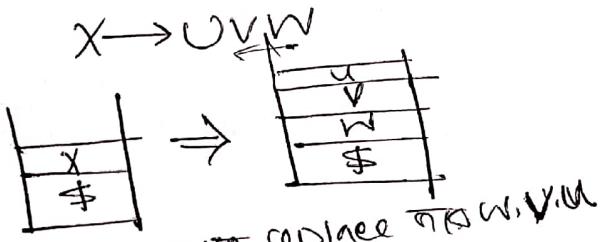
α → non terminal
β → non terminal, terminal mixed.

~~কাকা~~ ~~ফাঁকা~~ → error

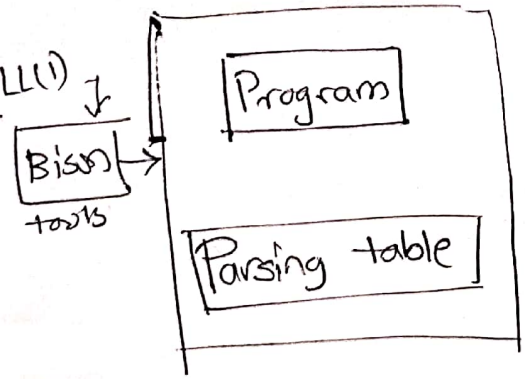
1. $X = a = \$$
declare successfully complete.

2. $X = a \neq \$$

(a)
(b) X — non terminal হলে,
~~parse~~ tree ~~go~~ যাবে result করবে,
M[x,a]



X এর replace এ U, V, W
push করবে



M[x,a] ← Parse table from LL(1).

x \ a	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)		

X, a

1. $X = a = \$$
2. $X = a \neq \$$
3. X is non-terminal

Left recursive

E → ① TE'
E' → ② +TE' / ③ ε
T → ④ FT'
T' → ⑤ *FT' / ⑥ ε
F → ⑦ id / ⑧ (E)

w = "id+id*id"

stack	input	Productions
$E	id+id*id$	
$E'T	id+id*id$	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ε
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	T' → *FT'
$E'T'F*	id$	
$E'T'F	id$	F → id

Scanned with OKEN Scanner

Stack	input	Productions
$E	id+id*id$	
$E'T	id+id*id$	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ε
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	
$E'T'F*	*id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Scanned with OKEN Scanner

$E \Rightarrow TE'$ [using ①]

$\Rightarrow FT'E'$ [using ④]

$\Rightarrow idT'E'$ [using ⑧]

$\Rightarrow id\varepsilon E'$ [u ⑥]

$\Rightarrow id+TE'$ [u ②]

$\Rightarrow id+FT'E'$ [u ⑨]

$\Rightarrow id+idT'E'$ [u ⑦]

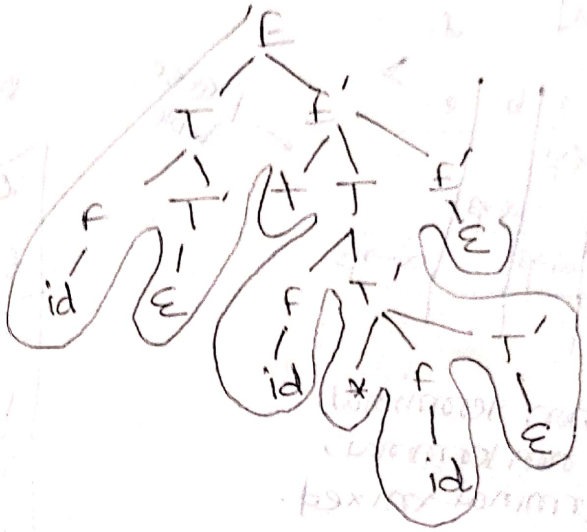
$\Rightarrow id+id*FT'E'$ [u ⑤]

$\Rightarrow id+id*idT'E'$ [u ⑧]

$\Rightarrow id+id*id\varepsilon E'$ [u ⑥]

$\Rightarrow id+id*id$ [u ③]

Pre order top down parsing

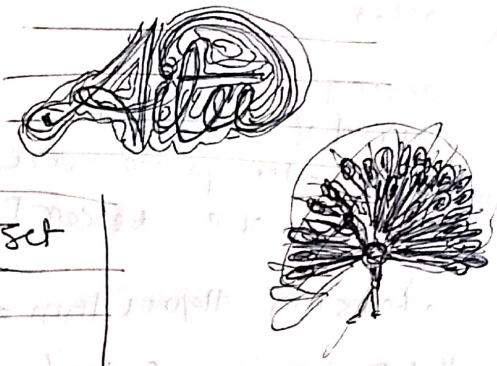


id + id * id.

+ FIRST, FOLLOW দিয়ে parsing table তৈরি করা হয়।

Date: 19.06.2025

Parsing table of LL(1) grammer:



* First set, follow set

Rules	First set	follow set
S → AcB|CbB|Ba	{d, g, h, ε}	{$}
A → da|BC	{d, g, h, ε}	{h, g, $}
B → g|ε	{g, ε}	{a, h, g, $}
C → h|ε	{h, ε}	{g, b, h, $}

fro(S) = ?
fro(C) = {h, ε}
Fr(A) = Fro(da) ∪ fro(BC) ⇒ Fro(B) ∪ Fr(C)
= {d} ∪ {g, ε} = {d, g, h, ε}
∪{h, ε}
fro(S) = Fro(ACB) ∪ Fro(CbB) ∪ Fro(Ba)
= fro(A) ∪ fro(C) ∪ Fro(B)
= {d, g, h, ε} ∪ fro(c) ∪ Fr(B)
= {d, g, h, ε} ∪ {b} ∪ {a}
= {d, g, h, b, a, ε}

Designing Parser

1. Make the given grammar LL(1)
2. Define Fr and follow
3. construct Parsing table
4. Write parsing algorithm
5. Test with example string

Rules	first set	follow set
E → TE'	{id, (}	{), $}
E' → +TE' | ε	{+, ε}	{), $}
T → FT'	{id, (}	{+,), $}
T' → *FT' | ε	{*, ε}	{+,), $}
F → id | (E)	{id, (}	{*, +,), $}

parsing table.

variables	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)		

* S → A
A → aB\|Ad
B → bBC\|f
C → g

grammar given. parser table বানাতে একটা

← LL1 grammar না, conflicting

Rules
S → aSbS | bSaS | ε

First	follow
{a, b, ε}	{b, a, $}

	a	b	$
S	S→aSbS	S→bSaS	

Rules	First	Follow
S → A		
A → a|B|Ad		
B → bBC|f		
C → g		

Scanned with OKEN Scanner

22/06/25

Parsing

2 kind of parsers:

1. Operator precedence
2. Recursive descent → Top down parsing.

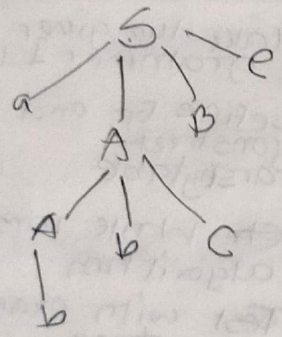
- Parsing Algorithms.

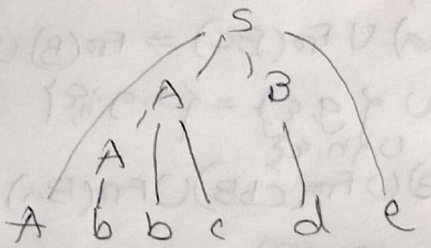
↓ LR Parsing (shift reduce parser)

① S → aABe
② A → Abc | b ③
④ B → d

S ⇒ aABe [①]
⇒ aAbcBe [②]
⇒ abbcBe [③]
⇒ abbcde [④].

S ⇒ aABe [①]
⇒ aAde [④]
⇒ aAbcde [②]
⇒ Abbcde [③]





22/06/95

Parsing

2 kind of parsers:

1. Operator precedence
2. Recursive descent → Top down parsing.

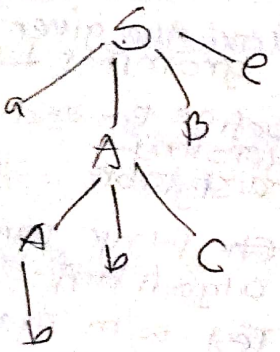
- Parsing Algorithms.

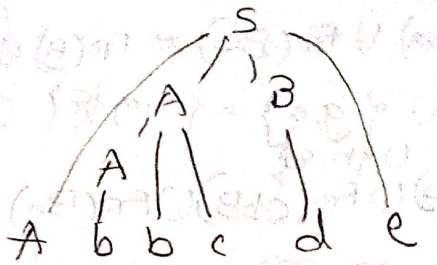
↓ LR Parsing (shift reduce parser)

① S → aABe
② A → Abc | b ③
④ B → d

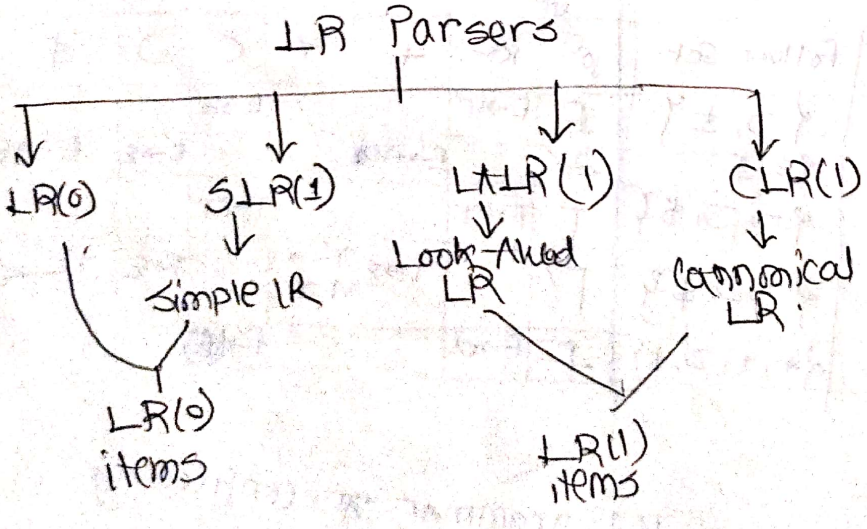
S ⇒ aABe [①]
⇒ aABBe [②]
⇒ aAbcBe [③]
⇒ abbcde [④].

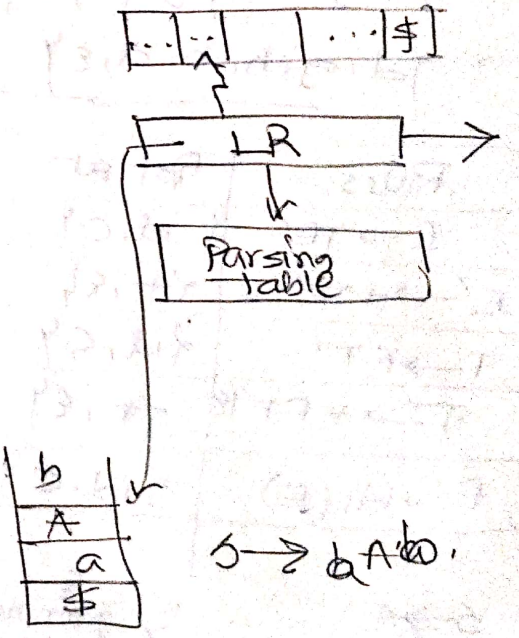
S ⇒ aABe [①]
⇒ aAde [④]
⇒ aAbcde [②]
⇒ Abbcde [③]





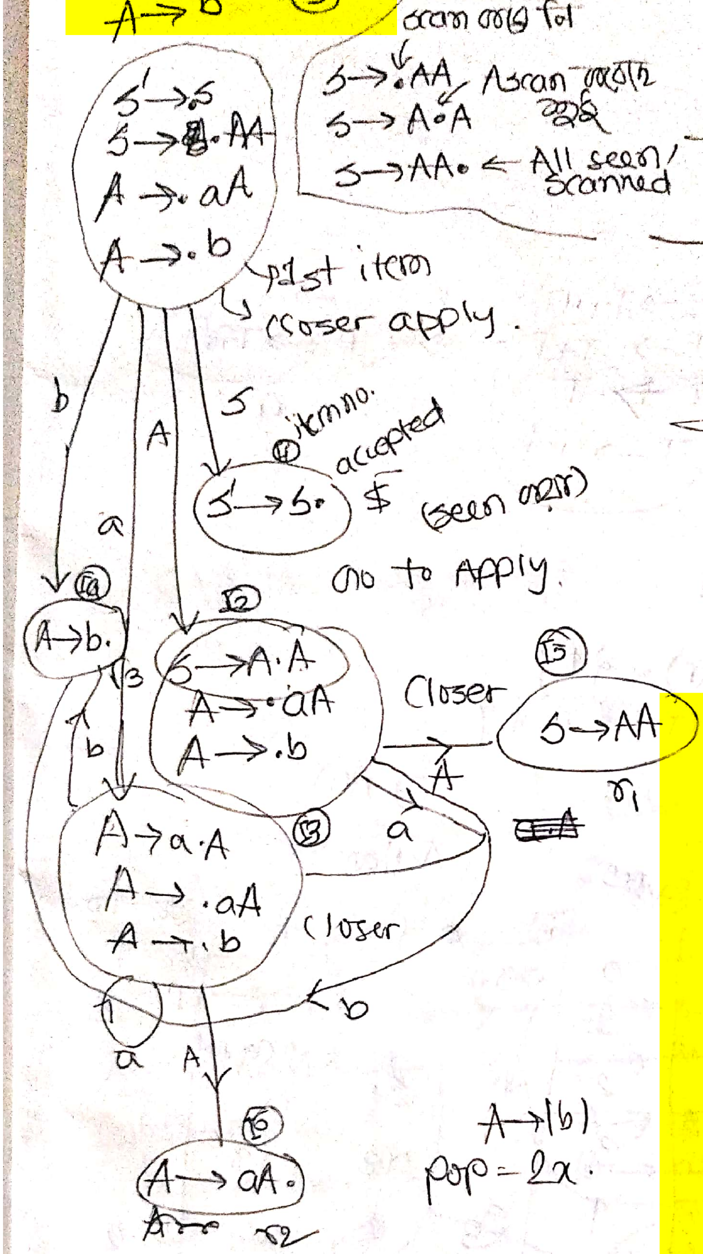
Date: 29.06.95





Scanned with OKEN Scanner

- S → AA ①
- A → aA | b ②
- A → b ③

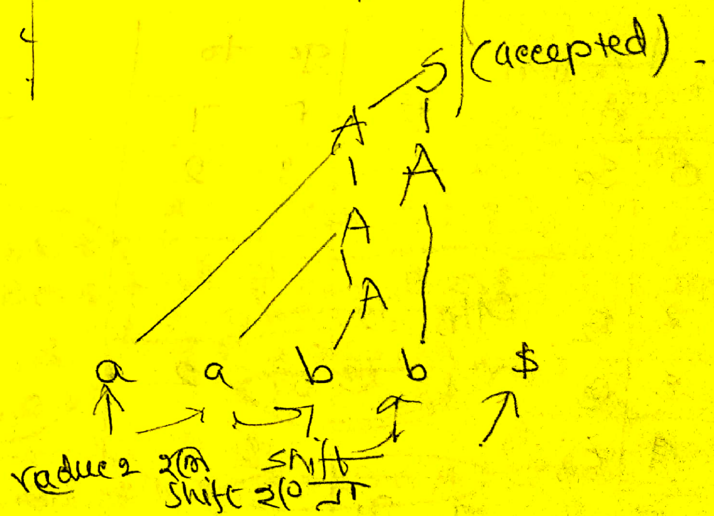


LR(0) Parsing table!

	Action (terminal)			Go to variable	
	a	b	$	S	A
0	s3	s4	–	1	2
1	–	–	accepted		
2	s3	s4	–		6
3	s3	s4	–		6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- S → AA ①
- A → aA ②
- A → b ③

w = aabb$.



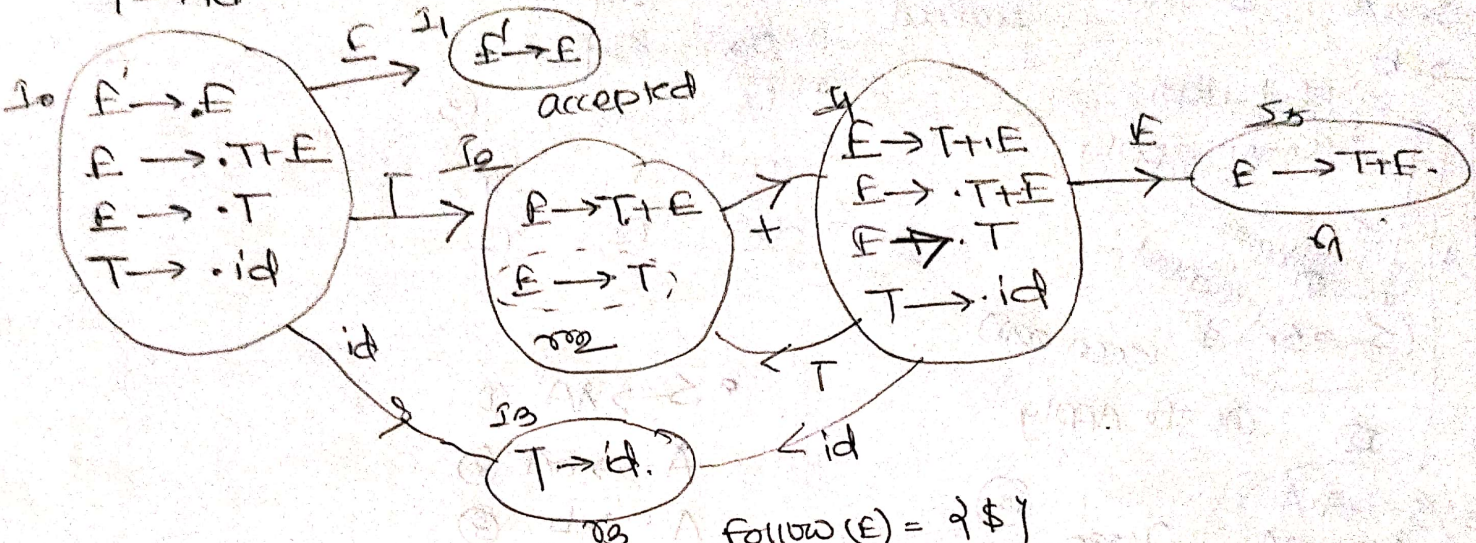
0	a	3	a	3	b	4	A	6	A	6	A	2	b	4	A	5

Date:

LR(0) SLR(1)

$E \to T + E$ (1) | T (2)

$T \to id$ (3)



follow(E) = {$}

follow(T) = {+, $}

LR(0)

States	Action			go to	
	id	+	$	E	T
0	S3			1	2
1					
2	r2	S4/r2	r2		
3	r3	r3	r3		
4	S3			5	2
5	r1	r1	r1		

SLR(1)

States	Action			go to	
	id	+	$	E	T
0	S3			1	2
1			accepted		
2		S4	r2		
3		r3	r3		
4	S3			5	2
5			r1		

LR(0) + follow

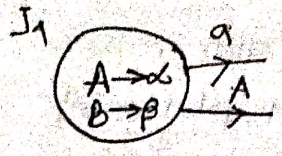
= LR(1) = SLR(1)

FA + Stack = PDA.

Scanned with OKEN Scanner

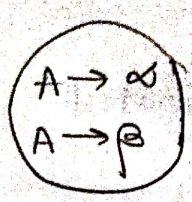
Date:

SLR(1) LR(0) + Follow = LR(1) - SR SLR(1)



Shift Reduce conflict!

In

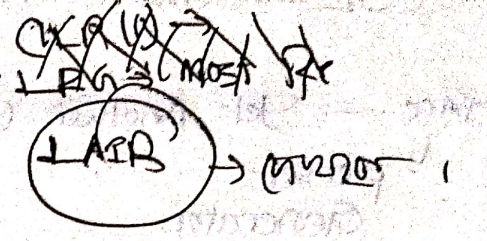


Reduce-Reduce conflict.

CLR(1) Parsing table.
look ahead

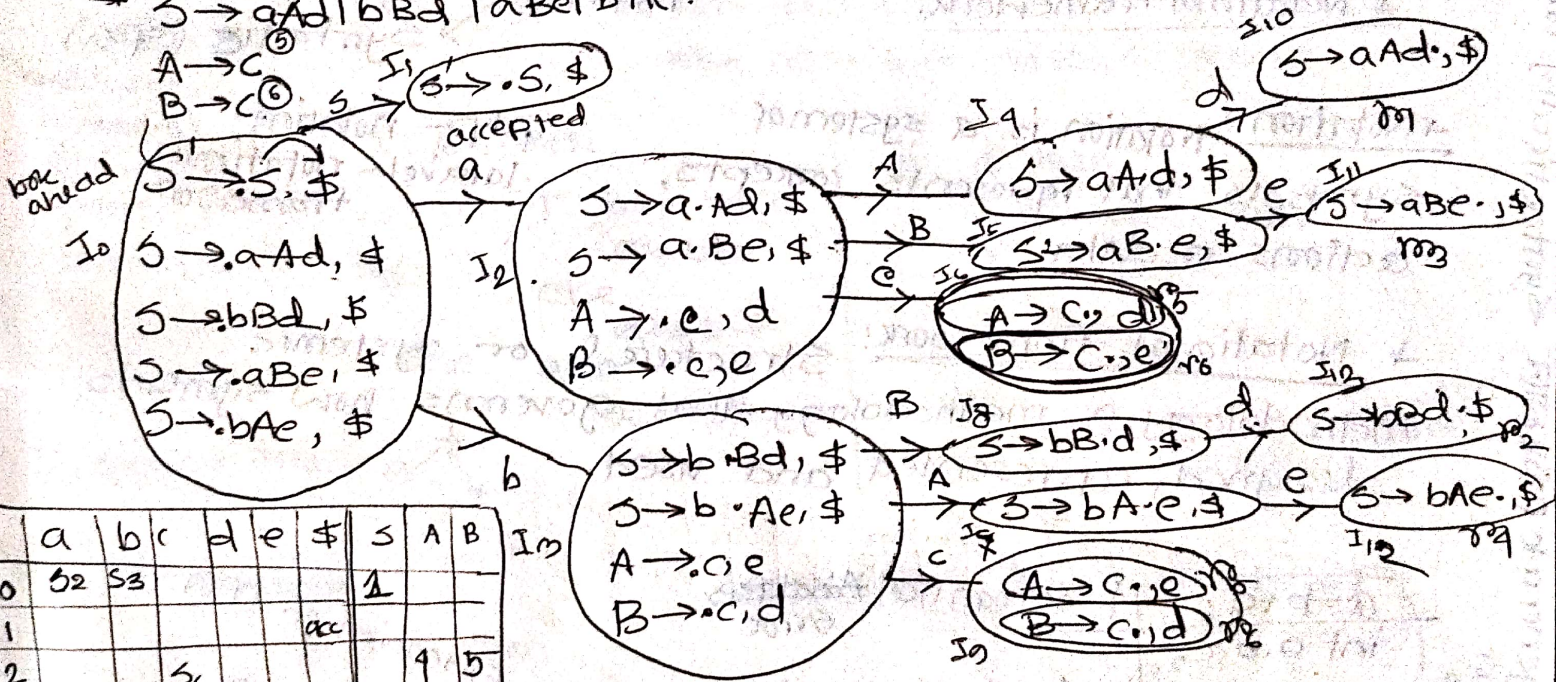
SLR(1)

LR(1) = LR(0) + Follow



* S → aAd | bBd | aBe | bAe.
① ② ③ ④

A → c ⑤
B → c ⑥



	a	b	c	d	e	$	S	A	B
0	S_2	S_3					1		
1						acc			
2			S_6					4	5
3			S_7					7	8
4				S_{10}					
5					S_{11}				
6				r_5	r_6				
7						S_{12}			
8				S_{13}					

	a	b	c	d	e	$	S	A	B
9				r_6	r_5				
10						r_1			
11						r_3			
12						r_4			
13						r_2			

Scanned with OKEN Scanner

Date: 17.07.2025 (very Important class)

* compiler design এ ৬টি phase module. ← Theoritically.
* কিন্তু practically ২টি phase এবং module - ২ ভাগে।
  * Front end: Analysis. → Independent (port, platform) OS + Machine
  * Back end - Implement. → dependent
    → order maintain

* Semantic Analysis:

RL → Lexical Analysis.
CFL → Syntax Analysis. → Semi - automated
CSL → Semantic Analysis.

YACC ← Yet Another Compiler Compiler → Input ▲ Grammar নিবে
↳ perser Generator. output ▲ parser দিবে।

* Notational Framework:

Notations → Lexical (RE)
Notations → Syntactic (CFG)

*Notation: Notation is a system of symbols that represents concepts, actions or data

php - notation, laravel - notational framework.

* Notational Framework: Structured or systemic methodology or methodology that governs how symbols designed, represented and used.

a=b+c; } semantical error
int a,b,c;
↳ Notation

* grammar + sub routine = Notational Framework. = SDT
Extension of CFG
scheme

* CFG + Action/subroutine = SDT.
E → E+E { action }

PDA = FA + stack.
S ⇒

Parse tree generate প্রোগ্রাম syntactically correct, sentence grammar এ থাকে
top down
bottom up

Subroutine, actions, semantics
↳ same-ই

Syntax Directed Translation Scheme.

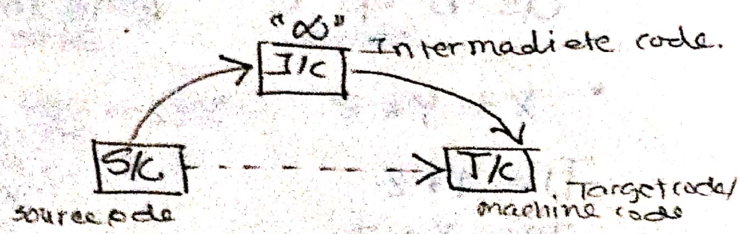
Scanned with OKEN Scanner

Date: 20.07.2025

Syntax Directed Translation (SDT): → 2য় phase ব্যবহার করা।

Grammar ~~Syntax~~ + semantic action = SDT

$A^* \Rightarrow xYz \overset{*}{\Rightarrow} w$



* Intermediate code (I/C) → Machine dependent / Machine independent

↓

Low Language এর জন্য বেশি understandable
High " " " কম "

সহজ লাগে আমার দুপুরে! কিন্তু তোমার দুপুরে!

Intermediate code's type:

1. Postfix notation ······ Infix, prefix
2. Syntax tree
3. Three-address code.

1. * if a then if b−c then a:= b+c else a:= b−c else a:= b*c.

conditional statement (infix) কে postfix এ conversion ⇒

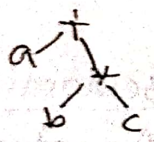
```
if a then
      if b-c then
            a:= b+c.
      else
            a:= b-c
else  ~~else~~
            a:= b*c.
```

```
if a then
    if bc- then
        bc+
    else
        bc-
else
    bc*
```

⇓ ternary operator (?) (3 operand)

abc−bc+bc−?bc*?

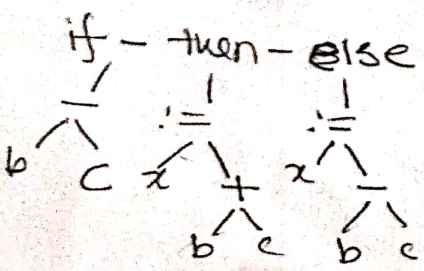
2. Syntax tree:



~~if a then~~

if b−c then x:= b+c

~~else~~ x:= b−c



Scanned with OKEN Scanner

3. Three address code: (TAC)

$a = b \text{ op } c$ (operator) (কোনো ক্ষেত্রেই ৩টার বেশি address ব্যবহার হবে না)

* $X + Y * Z$.   XYZ → User generated; T_1, T_2 → M/C generated.

$T1 := Y * Z$

$T2 := X + T_1$

* To implement TAC, two kinds of data structure can be used:
  1. Quadraple
  2. Triple.

* $a := -b + c * d$.

(0) $T_1 = -b$

(1) $T_2 = c * d$

(2) $T_3 = T_1 + T_2$

(3) $a := T_3$ → result. (assign)

1. Quadraple: ৪টা field এর মাধ্যমে represent.

(Temporary variable আছে) Argument

(index)	op	AR	AR2	Result
(0)	−	b		T_1
(1)	*	c	d	T_2
(2)	+	T_1	T_2	T_3
(3)	:=	T_3		a

2. Triple: ৩টা field দিয়ে represent:

- কোনো temporary variable নেই।

	op	ARG1	ARG
(0)	−	b	
(1)	*	c	d
(2)	+	(0)	(1)
(3)	:=	a	(2)

SDT:

Grammar + semantic Action.

$E \rightarrow E + E$ { $E.val := E^{(1)}.val + E^{(2)}.val$ }

$E \rightarrow$ digit {$E.val := $ digit}

* Semantic Actions:
  - main: (1) Evaluation
  - (2) I/C generation
  - (3) Error message generation (এইখানে error হলে)
  - (4) Rewriting/update symbol table.

* Semantic Errors:
  1. (1) Type miss matched
  2. (2) Multiple/duplicate variable declaration
  3. (3) Undeclaration of variable.
  4. (4) Reserved word/key word.
  5. (5) Scope parameter

Scanned with OKEN Scanner

"2+3*4"

$E \rightarrow E^{(1)} + T$ {E.val := E.val + T.val}

$\rightarrow T$ {E.val := T.val} ②

$T \rightarrow T^{(1)} * F$ {T.val := T.val * F.val} ②

$\rightarrow F$ {T.val := F.val} ③

$F \rightarrow$ digit {F.val := ~~num lexical~~ num.lval} ⑤

lval ⇒ lexical value.