# Compiler Design



## Compilers : A high Level overview

Compilers: → TOC → Grammars (CFG, RG), RE

pre req : → TOC

→ Basic working Knowledge of C / Java

{ Variables, Keywords, Identifiers }

GNU C-Compiler

gcc

Turbo C (IDE)

Borland C

Visual C++

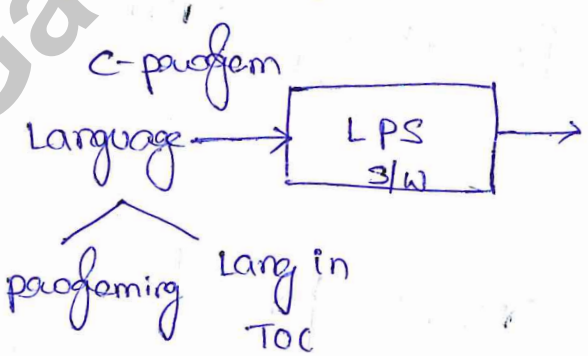
Java Compiler

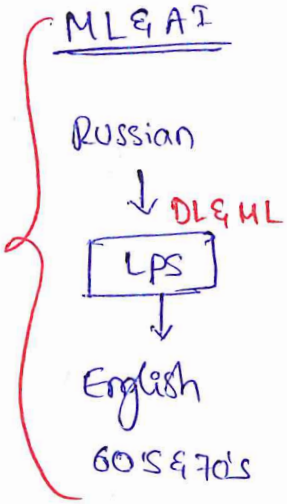
```
$gcc 1.c
$./a.out
```

1.c → gcc → object code

?

## Language processing System : (LPS)



HLL
↓
Preprocessing
↓
Compilation
↓
Assembling
↓
Linking & Loading

**Preprocessing:** `#include<stdio.h>` } Header file
`#include<stdlib.h>` }
`#define PI 3.1415` ← Macros

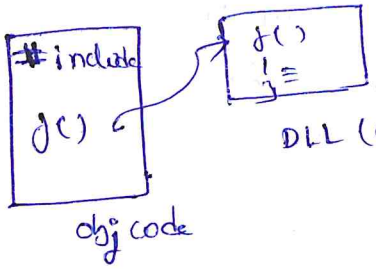
preprocessor Handly all header files and macros.

**Compilation:** HLL → Compiler → ALL Assembly-Lang
(C)
phases
{ CO, CA, COA
MP → 8085, 8086 }

**Assembler:** ASL → [Assembler] → MLL / obj-code / exe.code

**Loader & Linker:**

header - fn.decl^n
Libraries - fn.dfn^n
(DLL)




obj code
DLL (obj. code)

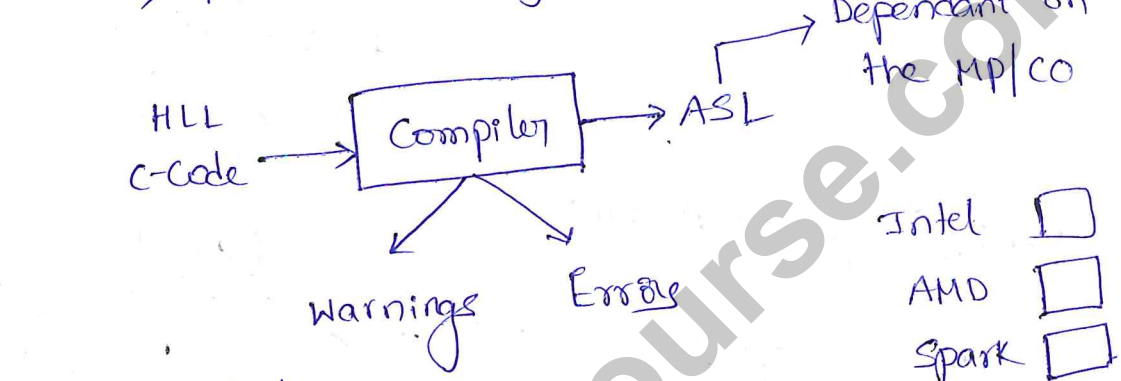
Dynamically Linkable Libraries

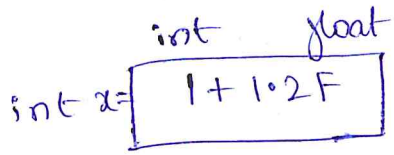
Loade all necessary jilg into memoge.

Translates : Assemblers, Compilers

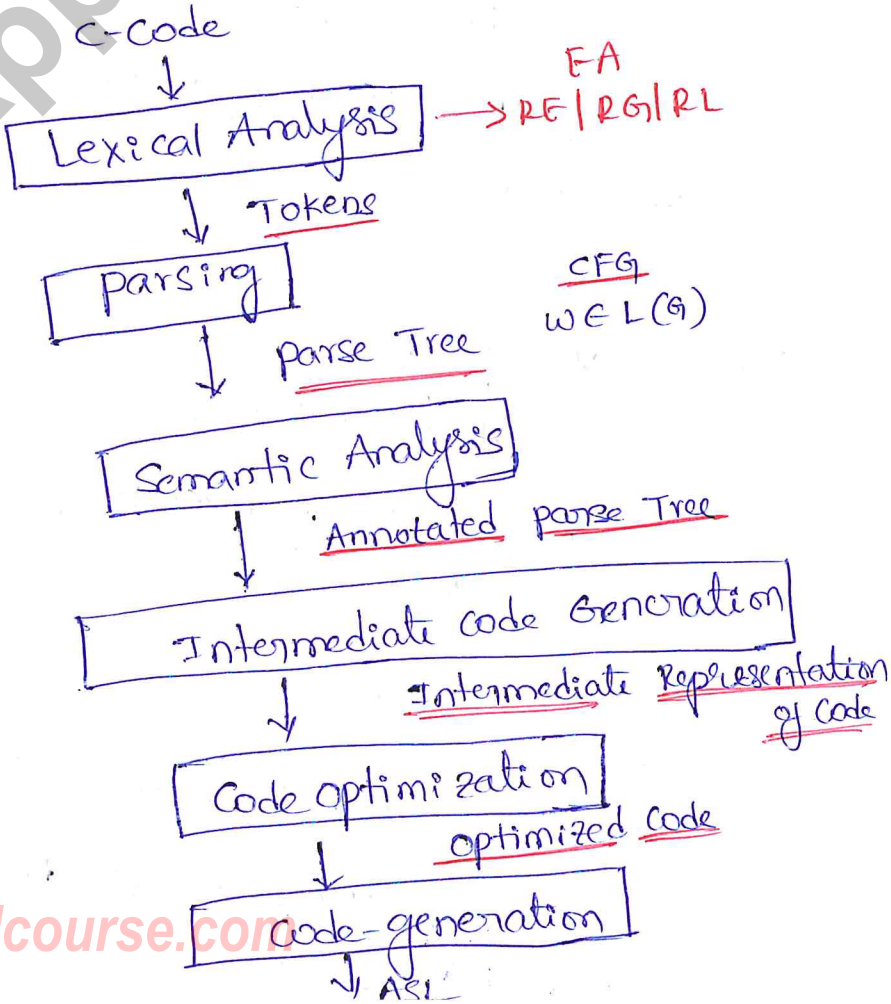
ASL → MLL  C-Code → ASL

→ Russian → English




## Phases of a Compiler:



## Lexical Analysis:

FA

Source code → Lex-Analyzer → Token (words)

peeling an Onion

{ int x=10; → EOS
  KW id op constant

KW: Keyword
Symbol | char
id: identifier
Op: operator

## Parsing | Syntax Analysis:

CFG

KW id op constant EOS → Syntax Analyser → Parse Tree

Tokens

Syntax-Errors

c-prog: CFG

int 10=x;

KW const op id EOS

$w \in L(G)$? → decidable

CYK

java: CFG

Eg: x = a+b;

↓ Lex-Analyser

id eqop id op id; (Tokens)

w

CFG:

S → id eqop A

A → id op id;

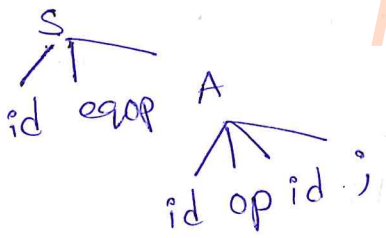
$\Sigma = \{id, eq, op, ;\}$

$w \in L(G)$?

S → id eqop A

→ id eqop id op id;

→ w




parse-Tree

## Semantic Analysis:

- Type-checking
- Type Conversion
- variable-declared?

```
int x=10;
bool b= TRUE;
char c = 'a';
int x = b+c;
```

b → ✗ ASCII ('a')

bool → int ✗

## Intermediate Code-Gen:

Generate IC to make the next-phase easier

$x = a+b*c;$

IC $t_1 = b*c;$

$t_2 = a+t_1;$

$x = t_2;$

## Code-optimization:

- M/c indep optimization
- M/c dep optimization

Reduce the # of instructions in the code without impacting the output.

```
int x = y || 1;
```

Short-Circuiting

```
int x = y | 1;
```

Microprocessor

Front-End : Lexical Analysis → Machine Independant optimization

Back-End : Machine dep optimization to code-gen

C-Compiler → Intel

C-Compiler → Micro Controller | MP | Rasberry pi

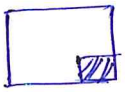
## Code Generation:

$x = a + b * c;$

$t_1 = b * c;$

$t_2 = a + t_1;$

$x = t_2$

Register 

ASL: b c

```
MUL R1 R2
ADD R0 R1   (a → R0)
MOV M[x], R0
```

R1 : b*c

R0 : a+b*c

Code-optimization is optional

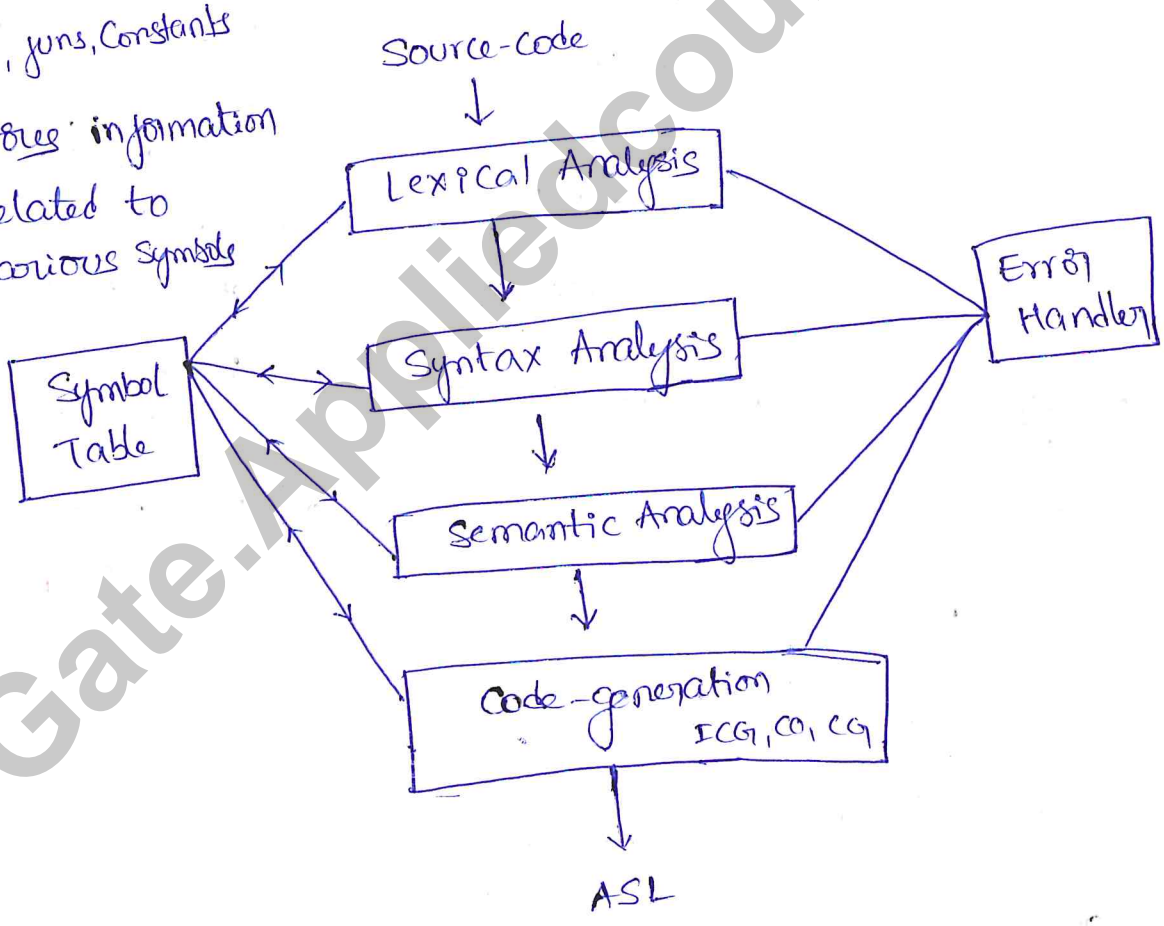
| Interpreter | Compiler |
|---|---|
| Execute the code Line by Line  | HLL → (C/A) obj code → (L/L) Executed |

| Interpreter | Compiler |
|---|---|
| SQL, LISP, Shell-scripting → bash, Python | FORTRAN, C, C++, Java, Python<br>Both Compiler & Interpreter<br>a.java —Compiler (javac)→ Byte code a.class → JVM Interpreter<br>Python |

## Symbol Table & Error-Handler

var, funs, Constants

Stores information related to various symbols




**Example:**

```
extern double bar (double x);

double foo (int count)
{
    double sum = 0.0;
    for (int i=1; i<=count; i++)
        sum += bar ((double) i);
    return sum;
}
```

8)



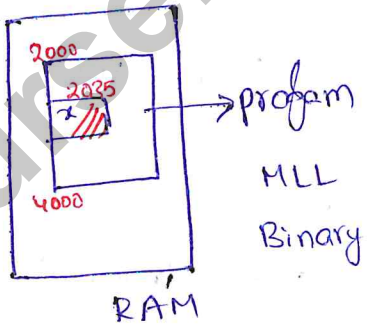
K — Value

| Symbol Name | Type | Scope |
|---|---|---|
| bar | function, double | extern |
| x | double | function parameter |
| foo | function, double | global |
| coont | int | function parameter |
| sum | double | block local |
| i | int | for-loop statement |

} C-prog

## Symbol Table

1. Name
2. Type
3. Scope
4. Size
5. offset
6. Misc




program
start : 2000
End : 4000
x : 2035, 2036

offset(x) = 2035 - 2000
= 35

## Implementation of Symbol Table:

1. Linear-Table
2. List
3. Tree
4. Hash Table

## operations:

1. Insert / ~~lookup~~
2. search / Lookup
3. Modify
4. Delete

```
{ int i;
  {
    float i;
  }
```

## Error-Handler: Warnings, Errors



① Lexical Error: nti x=10; int x=10;

nti → Not KW

② Syntax-Error: int x=10 → No Semi-colon

int x=10;

if (x>y) { } (x>y) if { } ✗

③ Semantic-Error:

int x="abc"; — "abc" → Array of char

int → KW; x → id; = → eqop; "abc" → const;

int x=10;
int y=20;
z=x+y; → Not declared

④ Fatal Error: → Memory insufficiency → Runtime

# Lexical Analysis: A deep dive

int x;

Grouping of characters into words

| Lexeme | Token |
|---|---|
| int | Keyword |
| x | id |
| ; | EOS |

int x ;

KW id op

↓ parser / Syntax Analyser

double

PI = 3.1415;

| lexeme | Token |
|---|---|
| double | KW |
| PI | id |
| = | op |
| 3.1415 | Const |
| ; | op |

## Secondary / Additional Task of lexical Analyser

① Remove comments // ----

/* ----

---- */

② Remove white space char

blanks, Tabs

③ Correlate the error with the line number

1: int x
↓
LA
↓
1: kw id
↓ parser
Error

Error in Line number 1

## Building a Lexical Analyser:

↓

Simple - String pattern-matching System — Regular Expression

email-addr
—@—.—

⇓

Regular Languages

⇓

FA

Moore / Mealy Machine

① Write your own code

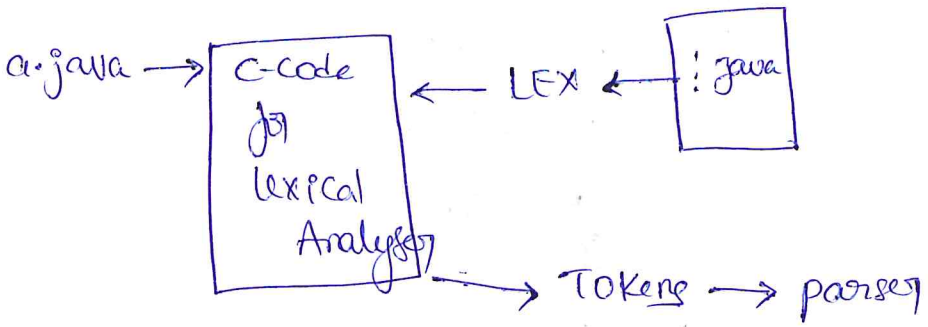
Table / Arrays: KW, OP ...

Rule : Identifier, Constant

② LEX (Unix, Linux, Mac)

↳ Special purpose Tool

→ Give the Rules in a Specific format



# YACC

↓

Syntax Analysis

LEX-tool Tutorial: https://youtu.be/5MbolqaHAfk

**Syntax-Analysis / Parsing:**

↳ Most of this course

CFG, parse-tree, derivation

↑

TOC, CYK, $w \in L(G)$?

Lex-Analysis LEX

Semantic Analysis

ICG

CO

CG

Grammar is a 4 Tuple

$G = \langle V, T, P, S \rangle$

V: set of Variables / Non-terminals

T: set of Terminals

P: production rules set

S: start-Symbol $\in V$

$P = \{ E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow E=E$

$E \rightarrow id$

$\}$

$V = \{E\}$

$T = \{id, +, *, =\}$

$S = \{E\}$

## Derivation:

G: $E \rightarrow E=E \mid E+E \mid E*E \mid id$

w (String) = $id_1 + id_2 * id_3$

$w \in L(G)$?

(Start-symbol)

Derivation:

$E \rightarrow E+E$

$\rightarrow E+E*E$

$\rightarrow id+id*id = w$

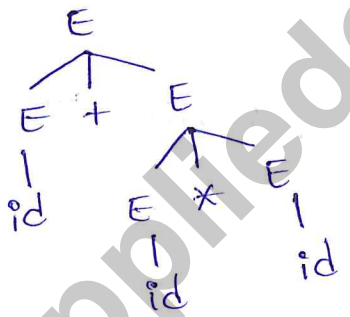
2+3*b;

↓ Lex Analysis

$id_1 + id_2 * id_3$

↓ Syntax Analysis

## Derivation Tree / parse Tree:




Pictorial Representation of parsetree (or) derivation

Types of derivation → Left most derivation
→ Right most derivation

G: $E \rightarrow E*E \mid E+E \mid E=E \mid id$

$w = id+id*id \quad w \in L(G)$

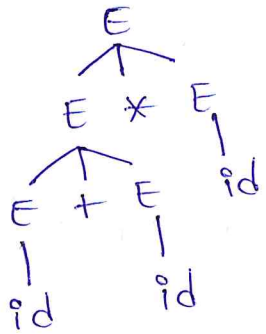
LMD

$E \rightarrow E*E$

$\rightarrow E+E*E$

$\rightarrow id+id*E$

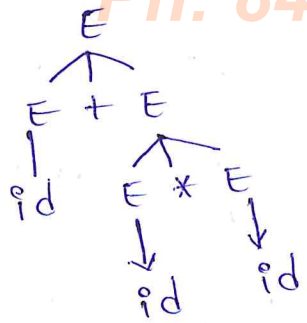
$\rightarrow id+id*id$




LMDT

$E \to E+E$
$\to E+E*E$
$\to E+E*id$
$\to E+id*id$
$\to id+id*id = w$

RMD




RMDT

## Classification of CFG:

- Recursive CFG
  - recursion in programming
- Non-Recursive CFG

Not REC-Lang (TM)
↳ TTM

### Recursive Grammar:

① $S \to Sa|b$

∃ atleast one production having same variable on LHS & RHS

② $S \to as|b$

③ $S \to asb|\epsilon$

L(RCFG) = Infinite

### Non-Recursive Grammar:

① $S \to AaB$
$A \to b|c$
$B \to c|d$

② $S \to AB|BA$
$A \to aB|b$
$B \to b|c$

$A \to \alpha$
$\alpha \in V$ $\quad \alpha \in (V+T)^*$

L(NRCFG) = Finite

## Types of Recursion:

### Left

$S \rightarrow Sa|a$

$\Updownarrow$

$S \rightarrow Sa$

$S \rightarrow a$

### Right

$S \rightarrow as|a$

$\Updownarrow$

$S \rightarrow aS$

$S \rightarrow a$

### General

$S \rightarrow asb|\epsilon$

$\Updownarrow$

$S \rightarrow asb$

$S \rightarrow \epsilon$

$S \rightarrow SS|a$

both Left & Right

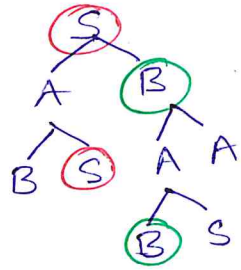
## Direct-Recursion vs Indirect-Recursion:

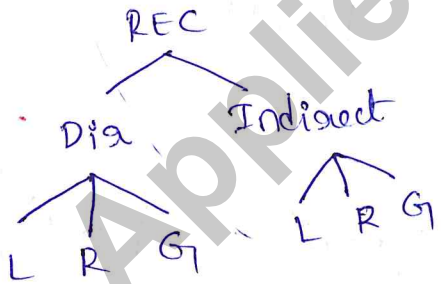
$S \rightarrow Sa|b$

$S \rightarrow AB$

$A \rightarrow BS|a$

$B \rightarrow AA|bB$




## Issues with Left-Recursive Grammars:

↓

Recursive Descent parser (RDP) ⇒ "Right Recursive Grammar"

↓

Infinite-loop

# Conversion from left-Recursion to Right Recursion

① $A \longrightarrow A\alpha \mid \beta$ $\quad \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha \cdots$

$A \longrightarrow \beta\alpha^*$

$$\Rightarrow \left\{ \begin{array}{l} A \longrightarrow \beta A' \\ A' \longrightarrow \epsilon \mid \alpha A' \end{array} \right\}$$

Right Recursive Grammar

② $A \longrightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ LRG

$$\Rightarrow \left\{ \begin{array}{l} A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \longrightarrow \epsilon \mid \alpha A' \end{array} \right\} \text{RRG}$$

③ $A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta$

$$\left\{ \begin{array}{l} A \longrightarrow \beta A' \\ A' \longrightarrow \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \end{array} \right\} \text{RRG}$$

④ $A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

$$\left\{ \begin{array}{l} A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \longrightarrow \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \end{array} \right\} \text{RRG}$$

Eg ①: $A \rightarrow \underbrace{Aab}_{\alpha} | \underbrace{c}_{\beta}$

$A \rightarrow cA'$

$A' \rightarrow abA' | \epsilon$

RRG

② $A \rightarrow AaB | b$

$B \rightarrow Ba | c$

$\Rightarrow$

$A \rightarrow bA'$

$A' \rightarrow \epsilon | aBA'$

$B \rightarrow cB'$

$B' \rightarrow \epsilon | aB'$

RRG

③ $A \rightarrow \underbrace{A(A)}_{\alpha} | \underbrace{0}_{\beta}$ $\quad \Sigma = \{(, ), 0\}$

$\Rightarrow$

$A \rightarrow 0A'$

$A' \rightarrow (A)A' | \epsilon$

④ $A \rightarrow AA | 0$

$\Rightarrow$

$A \rightarrow 0A'$

$A' \rightarrow AA' | \epsilon$

⑤ $A \rightarrow \underbrace{A(A)A}_{\alpha} | \underbrace{0}_{\beta_1} | \underbrace{1}_{\beta_2}$

$A \rightarrow 0A' | 1A'$

$A' \rightarrow (A)AA' | \epsilon$

} RRG

**Eg 6:** $S \rightarrow \underset{\alpha}{SSS} \mid \underset{\beta}{a}$

$S \rightarrow aS'$
$S' \rightarrow SSS' \mid \epsilon$ } RRG

**Eg 7:** $S \rightarrow \underset{\alpha}{SOS1} \mid \underset{\beta}{\epsilon}$

$S \rightarrow S'$
$S' \rightarrow OS1S' \mid \epsilon$ } RRG

**Eg: 8** $S \rightarrow \underset{\alpha_1}{SaSb} \mid \underset{\alpha_2}{SbSa} \mid \underset{\beta}{\epsilon}$

$S \rightarrow S'$
$S' \rightarrow aSbS' \mid bSaS' \mid \epsilon$ } RRG

**Eg: 9** $S \rightarrow SOS \mid 1$

$S \rightarrow 1S'$
$S' \rightarrow OSS' \mid \epsilon$ } RRG

**Eg: 10** $S \rightarrow \underset{\alpha}{Sa} \mid \underset{\beta_1}{aS} \mid \underset{\beta_2}{b}$

$\Rightarrow S \rightarrow Sa \mid b \Rightarrow$ $S \rightarrow bS'$, $S' \rightarrow aS' \mid \epsilon$, $S \rightarrow aS$ } RRG
$S \rightarrow aS$

**Eg 11:** $E \rightarrow \underset{\alpha_1}{E+E} \mid \underset{\alpha_2}{E*E} \mid \underset{\beta}{id}$

$\Rightarrow E \rightarrow id\,E'$
$E' \rightarrow +EE' \mid *EE' \mid \epsilon$

Eg 12: $E \rightarrow E+T \mid T$

$T \rightarrow id$

$\Rightarrow$ $E \rightarrow TE'$

$E' \rightarrow +TT' \mid \epsilon$ } RRG

$T \rightarrow id$

Eg 13: $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$ } RRG

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

$F \rightarrow (E) \mid id$

Eg 14: $S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$ } RRG

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

Eg 15: $R \rightarrow \underbrace{Rb}_{\alpha_1} \mid \underbrace{Ra}_{\alpha_2} \mid \underbrace{aR}_{\beta} \mid \underbrace{b}_{\beta}$

$R \rightarrow aR$

$R \rightarrow bR'$ } RULE 3 RRG

$R' \rightarrow \epsilon \mid RR' \mid aR'$

Eg 16:

$S \to AaB$

$A \to aA \mid Ba$

$B \to AB \mid b$

Indirect Recursion

$\Leftrightarrow$

$S \to AaB$

$A \to aA \mid ba \mid ABa$

$B \to AB \mid b$

(81)

$S \to AaB$

$A \to aA \mid Ba$

$B \to aAB \mid BaB \mid b$

$S \to AaB$

$A \to aA \mid \underbrace{ba}_{\beta} \mid \underbrace{ABa}_{\alpha}$

$B \to AB \mid b$

$\Downarrow$

$S \to AaB$

$A \to aA$

$A \to baA'$

$A' \to \in \mid BaA'$

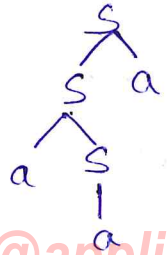
$B \to AB \mid b$

## Ambiguous and Unambiguous Grammar:

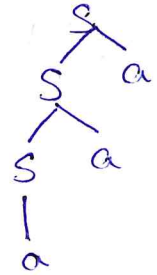
G: $S \to Sa \mid aS \mid a$

$w = aaa$

(1) LMDT



(2) LMDT



(3) RMDT

(4) RMDT

**Ambiguous:**
∃ more than one derivation for any word $\in L(G)$

**Unambiguous Grammar:**
∃ only one derivation for all words $\forall w \in L(G)$

**Properties:**

1. Unambiguous Grammars ⇒ (LMDT = RMDT)
2. Grammar: Left rec & Right rec ⇒ Ambiguous
3. Ambiguity of CFG is undecidable.
   ⇒ ∄ an algorithm to determine ambiguity of CFG

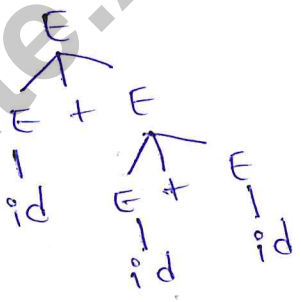
**Eliminating Ambiguity:**

$G_1$: $E \to E+E \mid id$, $w = id+id+id$

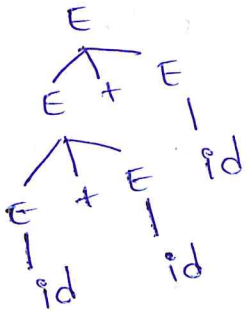
⇒ Unambiguous

$E \to E+T \mid T$
$T \to id$
$\Big\}\ G_2$

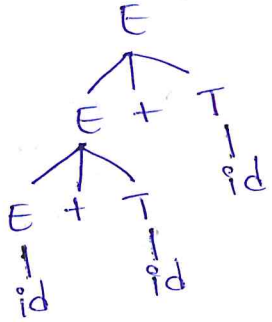
$L(G_1) = L(G_2)$




≠




$w = id+id+id$



## Ambiguous & not :

① $S \rightarrow SOS|\epsilon$ — Ambiguous

② $S \rightarrow SaS|SbS|\epsilon$ — Ambiguous

③ $S \rightarrow SSS|0$ — Ambiguous

④ $S \rightarrow SOSIS|1$ — Ambiguous

⑤ $S \rightarrow E+E|E*E|id$ — Ambiguous

⑥ $S \rightarrow aSb|\epsilon$ — unambiguous

$\{\epsilon, ab, aaSb \ldots\}$

⑦
$S \rightarrow SAB$
$A \rightarrow aAB|b$
$B \rightarrow BS|a$

⑤ $S \rightarrow SABS|SAa$ — Ambiguous

$A \rightarrow aAB|b$

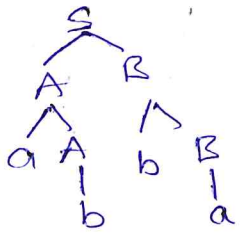
$B \rightarrow BS|a$

⑧ $A \rightarrow (A)|AA|0$ — Ambiguous

⑨ $S \rightarrow AB$ — unambiguous

$A \rightarrow aA|b$

$B \rightarrow bB|a$



⑩ Regular Grammar : unambiguous

CFL / CFG : Ambiguous or Unambiguous

## Left Factoring: Transformation → LRG to RRG



→ Ambiguous to unambiguous

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$ Same prefix

$\alpha \in (V+T)^*$

$\beta_i \in (V+T)^*$

$\Rightarrow$

$A \to \alpha B$

$B \to \beta_1 | \beta_2 | \beta_3$

① $A \to aAb | aAd | e$

⇓ LF

$A \to aAB | e$

$B \to b | d$

② $A \to (A)A | (A)Ab | (A)Aba | e | f$

⇓

$A \to (A)A | (A)AbB | e | f$

$B \to e | a$

⇓

$A \to (A)AC | e | f$

$B \to e | a$

$C \to e | bB$

(3) $A \rightarrow abd \mid abde \mid abdA \mid abdAe \mid f \mid g$

LF ⇓

$A \rightarrow abd \mid abde \mid A\ abdAB \mid \epsilon \mid f \mid g$

$B \rightarrow \epsilon \mid e$

⇓

$A \rightarrow abdC \mid \epsilon \mid f \mid g$

$C \rightarrow \epsilon \mid e \mid AB$

$B \rightarrow \epsilon \mid e$

## Non-deterministic & Deterministic Grammar:

TOC { DCFG → DCFL → DPDA
NCFG → CFL (NCFL) → NPDA/PDA

**Note:** If G is DCFG then G is unambiguous

## Dangling-Else Ambiguity:

Stmt → if expr then Stmt |
if expr then Stmt else Stmt |
other

w = if $E_1$ then if $E_2$ then $S_1$ else $S_2$

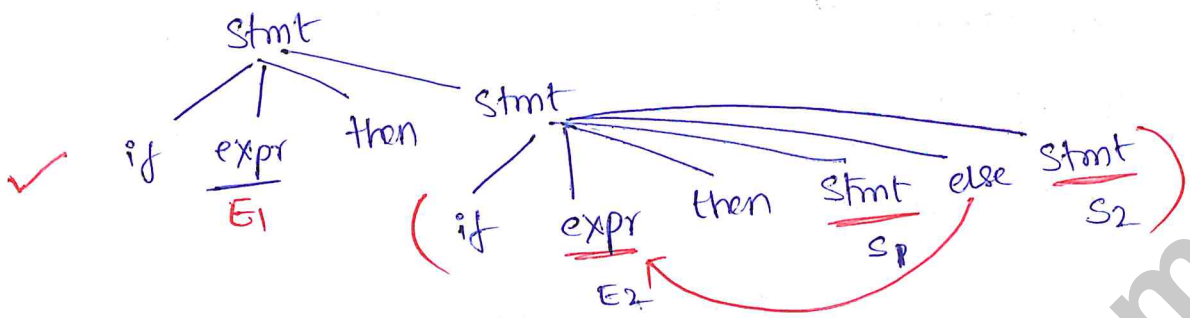
⇓

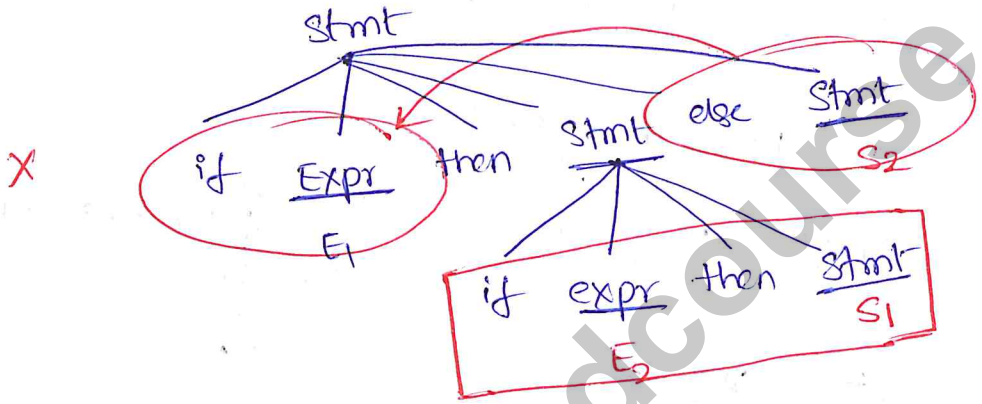
2-parse trees

Source Code
↓ lex analyser
Tokens
↓ parser

Syntactically correct

## Ambiguity:




Most programming Languages support else will belongs to the nearest/closest unmatched if Statement.

## Eliminating Ambiguity:

Stmt → matched-stmt
| open-Stmt

matched-stmt → if expr then matched-stmt
else matched-stmt
| other

open-stmt → if expr then Stmt
| if expr then matched-stmt else
open-stmt

$w$ = if $E_1$ then if $E_2$ then $S_1$ else $S_2$

The grammar will generate unique parse tree for the given string.

$G_1$ is unambiguous.

## Grammars for programming Languages:

① if ... else .... production-rules

② $E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid id$

id+id  x+y  x+y+z  x+y*z → lex-Analyser → id+id*id

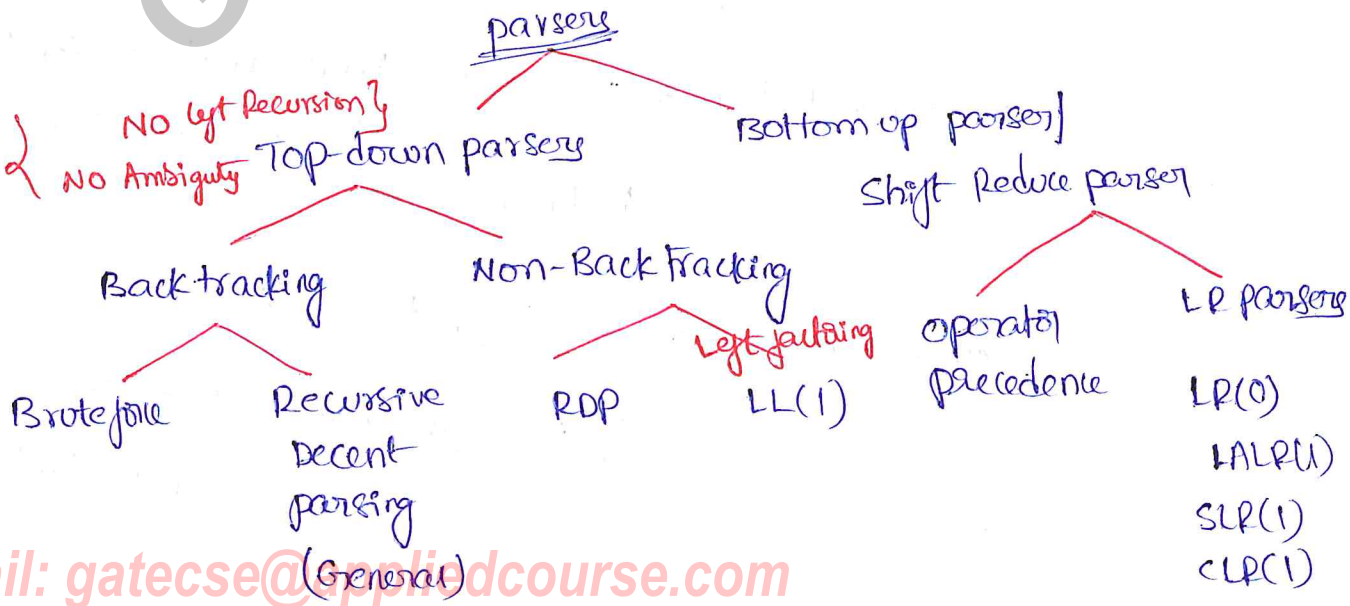
id id+ ← x y + x
L.A

③ C-programming Language

⇓

parser CFG

URL: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html



## Top-down parser: Free from Left-Recursion

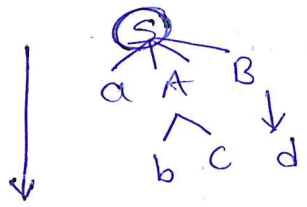
Left-Recursive

① $S \to aAB$

$A \to bc \mid b$

$B \to d$

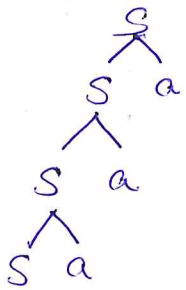
Let $w = abcd$




Parse-Tree

Left-most derivation Tree

② $S \to Sa \mid a$




$w = aaa$

Unambiguous

③ $S \to aS'$

$S' \to \epsilon \mid aS'$

$w = aaa$


## Bottom-up parsing:

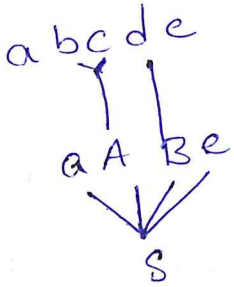
$S \to aABc$

$A \to bc$

$B \to d$

$w = abcde$

$\Rightarrow$




Take the input string and derive the start symbol of the grammar.

## Brute Force: (TDP)

$S \to aAd \mid aB$

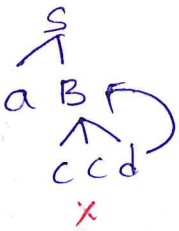
$A \to b \mid bc$

$B \to ccd \mid ddc$

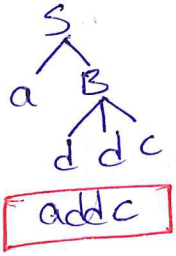
$w = addc$


✗


✗




✗




addc

Baute-Face ⇒ Try all possibilities

## Recursive Descent parser: (General)

⇓

May need back track

Slightly Smarter than Baute Face

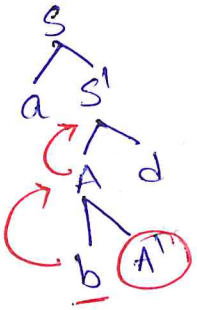
$S \rightarrow aS'$

$S' \rightarrow Ad \mid B$

$A \rightarrow bA'$

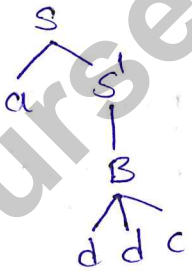
$A' \rightarrow c \mid \epsilon$

$B \rightarrow ccd \mid ddc$




$w = addc$

No Left-Recursion, unambiguous




⇒ addc

$w \in L(G)$

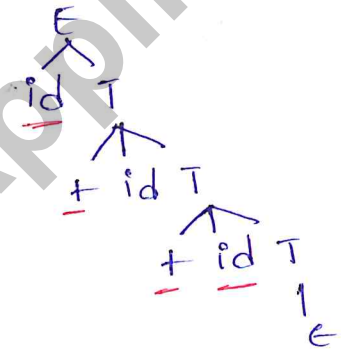
## Recursive Descent parser (program)

$E \rightarrow id\,T$

$T \rightarrow +id\,T \mid \epsilon$

$w = id+id+id$

(No backtracking here)



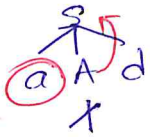

## Left-factored Grammar + RDP:

$S \rightarrow aAd \mid aB$

$A \rightarrow b \mid bc$

$B \rightarrow ccd \mid ddc$

$w = addc$




$S \rightarrow aS'$

$S' \rightarrow Ad \mid B$

$A \rightarrow bA'$

$A' \rightarrow \epsilon \mid c$

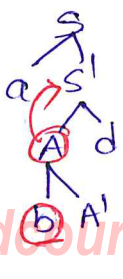
$B \rightarrow ccd \mid ddc$

⇓

Unambiguous + No Left Recursion

No Same prefixes

addc

Left factoring simplifies in choosing the productions.



## LL(1) Parser: FIRST(X), Follow(A)

FIRST(X) = Set of all terminals ($\alpha$), s.t

$X \longrightarrow \alpha\beta \longrightarrow$ String of terminals & Variables

$\alpha$ ↓ terminal

① $S \longrightarrow a|b|\epsilon$

First(S) = {a, b, ϵ}

② $S \longrightarrow aA|\epsilon$ First(S) = {a, ϵ}

$A \longrightarrow b$ First(A) = {b}

③ $S \longrightarrow aA|bB$ First(S) = {a, b}

$A \longrightarrow AB|c$ First(A) = {c}

$B \longrightarrow cB|d$ First(B) = {c, d}

④ $S \longrightarrow AB$

$A \longrightarrow a|b$

$B \longrightarrow d$

First(S) = First(A) = {a, b}

First(B) = {d}

$S \xrightarrow{*} aB$

$S \longrightarrow AB$

aB bB

$S \xrightarrow{*} bB$

⑤ $S \longrightarrow AB$

$A \longrightarrow a|\epsilon$

$B \longrightarrow c|d$

First(S) = {a, c, d}

First(A) = {a, ϵ}

First(B) = {c, d}

$S \longrightarrow AB$

$\xrightarrow{*} aB$

$\xrightarrow{*} B$

{c, d}

(6) $S \to AB$, $A \to aA/\epsilon$, $B \to bB/\epsilon$

First(S) = {a, b, ε}
First(A) = {a, ε}
First(B) = {b, ε}

(7) $S \to AaB$, $A \to bA/\epsilon$, $B \to cB/d$

First(S) = {b, a}
First(A) = {b, ε}
First(B) = {c, d}

(8) $S \to ABb$, $A \to Ba/\epsilon$, $B \to b/d$

First(S) = F(A) = {b, d}
First(A) = {ε, b, d}
First(B) = {b, d}

(9) $S \to AaB/Ba$, $A \to Ba/b$, $B \to aB/\epsilon$

First(S) = {a, b, ε}
First(A) = {a, b}
First(B) = {a, ε}

(10) $S \to AaB/\epsilon$, $A \to BbA/a$, $B \to Db/d$, $D \to e/f$

| Sym | First |
|---|---|
| S | {e, f, d, a, ε} |
| A | {e, f, d, a} |
| B | {e, d, d} |
| D | {e, f} |

(11) $E \to TE'$, $E' \to +TE'/\epsilon$, $T \to FT'$, $T' \to *FT'/\epsilon$, $F \to (E)/id$

| Sym | First |
|---|---|
| E | {(, id} |
| E' | {+, ε} |
| T | {(, id} |
| T' | {*, ε} |
| F | {(, id} |

(12) $S \to (L) \mid a$

$L \to SL'$

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

| | First |
|---|---|
| S | {(, a} |
| L | {(, a} |
| L' | {, , ε} |

(13) $S \to ABCDE$

$A \to a \mid \epsilon$

$B \to b \mid \epsilon$

$C \to c \mid \epsilon$

$D \to d \mid \epsilon$

$E \to \epsilon$

| | First |
|---|---|
| S | {a, b, c, d, ε} |
| A | {a, ε} |
| B | {b, ε} |
| C | {c, ε} |
| D | {d, ε} |
| E | {ε} |

(14) $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

| | First |
|---|---|
| S | {a, b} |
| A | {ε} |
| B | {ε} |

**Follow(A):** Set of terminals ($\alpha$) s.t

↓ Variable

$S \xrightarrow{*} \alpha A \alpha \gamma \to (V+T)^*$

$(V+T)^*$ var Terminal

$\alpha \in$ Follow(A)

$w = abab\$$ → To represent end of the string

(2) $S \to \alpha A$

Then Follow(A) = Follow(S)

(3) If S is the Start Symbol of the Grammar

then add '$' to the follow(S).



① $S \rightarrow a|b|\epsilon \Rightarrow Follow(S) = \{\$\}$ let $w = a\$$

② $S \rightarrow aAb \Rightarrow Follow(S) = \{\$\}$

$A \rightarrow Aa|c \quad Follow(A) = \{a, b\}$

left-recursive Grammar

③ $S \rightarrow asb|\epsilon \Rightarrow Follow(S) = \{\$, b\}$

④ $S \rightarrow as|sa|\epsilon \Rightarrow Follow(S) = \{\$, a\}$

⑤ $S \rightarrow aA \Rightarrow Follow(S) = \{\$\}$

$A \rightarrow b \quad Follow(A) = \{\$\}$

Special case

⑥ $S \rightarrow aA \Rightarrow Follow(S) = \{\$\}$

$A \rightarrow aAb|c \quad Follow(A) = \{\$, b\}$

⑦ $A \rightarrow (A)|a \Rightarrow Follow(A) = \{\$, )\}$

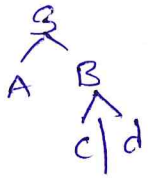
⑧ $S \rightarrow AB \Rightarrow Follow(S) = \{\$\}$

$A \rightarrow a|b \quad Follow(A) = \{c, d\}$

$B \rightarrow c|d \quad Follow(B) = \{\$\}$

$S \rightarrow AB$

$First(B) = \{c, d\}$



(9) $S \to AB$

$A \to a|b|\epsilon$

$B \to c|d$

$\Rightarrow$

| | Follow |
|---|---|
| S | {$} |
| A | {c, d} |
| B | {$} |

(10) $S \to AB$

$A \to a|b$

$B \to c|d|\epsilon$

$S \to A\epsilon \to A$

$\Rightarrow$

| | Follow |
|---|---|
| S | {$} |
| A | {c, d, $} |
| B | {$} |

(11) $S \to SOS|S2|\epsilon$ $\Rightarrow$ Follow(S) = {$, 0, 1, 2}

(12) $S \to ABC$

$A \to a|\epsilon$

$B \to b|\epsilon$

$C \to d|\epsilon$

$\Rightarrow$

| | Follow |
|---|---|
| S | {$} |
| A | {$, b, d} |
| B | {$, d} |
| C | {$} |

ABC

(13) $S \to AA$

$A \to aA|b$

| | Follow |
|---|---|
| S | {$} |
| A | {$, a, b} |

$S \to AA$

First(A) = {a, b}

(14) $E \to TE'$

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

$T \to FT'$

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

$F \to (E)|id$

| | Follow |
|---|---|
| E | {$, )} |
| E' | {$, )} |
| T | {+, $, )} |
| T' | {+, $, )} |
| F | {*, +, $, )} |

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

$\Downarrow$

$E' \to +TE$

$E' \to +T$

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

$\Rightarrow T' \to *FE$

$T \to *F$

(15) $S \to (L) \mid a$

$L \to SL'$

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

| | Follow |
|---|---|
| S | {$, ,, )} |
| L | {)} |
| L' | {)} |

$L \to SL'$

First(L') = {,, ε}

$L \to S\epsilon$

$L \to S$

(16) $S \to ABCDE$

$A \to a \mid \epsilon$

$B \to b \mid \epsilon$

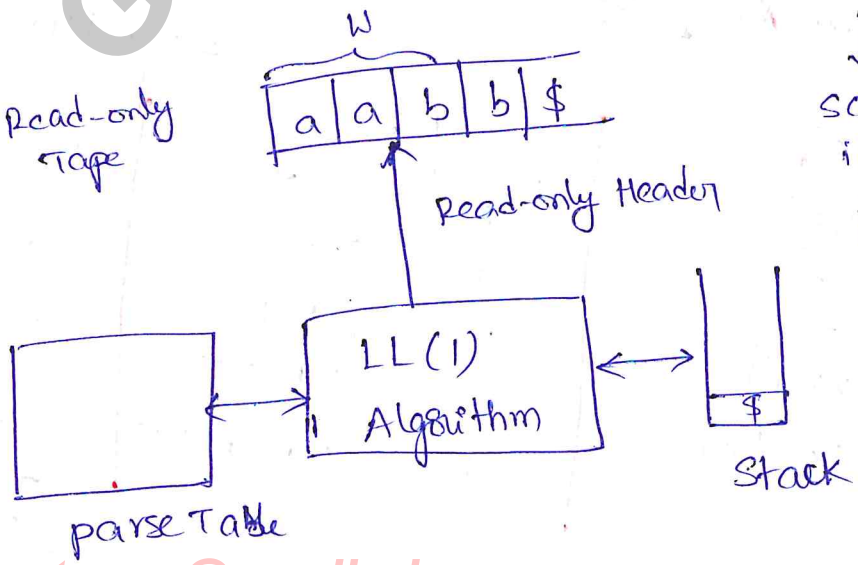
$C \to c \mid \epsilon$

$D \to d \mid \epsilon$

$E \to \epsilon$

| | Follow |
|---|---|
| S | {$} |
| A | {$, b, c, d} |
| B | {$, c, d} |
| C | {$, d} |
| D | {$} |
| E | {$} |

## LL(1) parser:




LL(1) → Scan the input from L to R; LMDT; # of Lookahead Symbols

LL(k)

① Free From Ambiguity

② Free From left-Recursion

③ Free From Common-prefixes.

RDP: Recursion → Internal Stack

## parse-Table:

① $A \rightarrow (A) \mid a$

$A \rightarrow (A)$ ①

$A \rightarrow a$ ②

| Vars | Terminals: ( | ) | a | $ |
|---|---|---|---|---|
| A | ① | | ② | |

Empty Cells represent error

① place $A \rightarrow \alpha$ in $T[A, a]$ $\forall a \in First(\alpha)$

② If First($\alpha$) Contains $\epsilon$, then add $A \rightarrow \alpha$ $\forall b \in Follow(A)$

② $S \rightarrow AA$ ①

$A \rightarrow aA \mid b$ ② ③

First(S) = First(A) = {a, b}

| | a | b | $ |
|---|---|---|---|
| S | ① | ① | |
| A | ② | ③ | |

③

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE'$
3. $E' \rightarrow \epsilon$
4. $T \rightarrow FT'$
5. $T' \rightarrow *FT'$
6. $T' \rightarrow \epsilon$
7. $F \rightarrow (E)$
8. $F \rightarrow id$

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | (1) | | | (1) | | |
| E' | | (2) | | | (3) | (3) |
| T | (4) | | | (4) | | |
| T' | | (6) | (5) | | (6) | (6) |
| F | (8) | | | (7) | | |

First(E) = {(, id}
First(E') = {+, ∈}
First(T) = {(, id}
First(T') = {*, ∈}
First(F) = {(, id}

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

## Parsing Algorithm:

① ②

G: A → (A) | a

| | ( | ) | a | $ |
|---|---|---|---|---|
| A | ① | | ② | |

let w = ((a))

| Stack X | input a | Action |
|---|---|---|
| $ | ((a))$ | PUSH(S) |
| $A | ((a))$ | POP)A( ① |
| $)A( | ((a))$ | pop |
| $)A | ((a))$ | PUSH)A( ① |
| $))A( | (a))$ | pop |
| $))A | (a))$ | PUSH)A( ① |
| $)))A( | a))$ | pop |
| $)))A | a))$ | PUSH a ② |

| Stack | input | Action |
|---|---|---|
| $ ) ) ) a | a ) ) ) $ | pop |
| $ ) ) ) | ) ) ) $ | pop |
| $ ) ) | ) ) $ | pop |
| $ ) | ) $ | pop |
| $ | $ | Accept |

Lets Assume X: Stack top a: current input Symbol

① X = a ≠ $ pop & increment the input pointer

② X = a = $ Accept

③ T(X, a) X → α pop X from the Stack and push 'α' into the Stack in reverse order.

w = ) a ( ⇒ w ∉ L(G)

④ T(X, a) = Empty Syntax Error



Example 2:

① S → AA

② A → aA

③ A → b

| | a | b | $ |
|---|---|---|---|
| S | 1 | 1 | |
| A | 2 | 3 | |
| a | | | |

Let w = abab

| Stack | input | Action |
|---|---|---|
| $ | abab$ | PUSH (S) |
| $S | abab$ | PUSH AA |
| $AA | abab$ | PUSH Aa |
| $AAa | bab$ | pop & increment |
| $Aa | bab$ | PUSH b |
| $Ab | ab$ | pop & increment |
| $A | ab$ | PUSH Aa |
| $Aa | b$ | ~~PUSH a~~ pop & Increment |
| $A | b$ | PUSH b |
| $b | b$ | pop & Increment |
| $ | $ | Accept |

Example 3:

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

| Stack | Input | Action |
|---|---|---|
| $E | id+id*id$ | E → TE' |
| $E'T | id+id*id$ | T → FT' |
| $E'T'F | id+id*id$ | F → id |
| $E'T'id | id+id*id$ | pop |
| $E'T' | +id*id$ | T' → ∈ |
| $E' | +id*id$ | E' → +TE' |
| $E'T+ | +id*id$ | pop |

| Stack | input | Action |
|---|---|---|
| $ E' T | id * id $ | T → FT' |
| $ E' T' F | id * id $ | F → id |
| $ E' T' ~~id~~ | ~~id~~ * id $ | pop |
| $ E' T' | * id $ | T' → * FT' |
| $ E' T' F ~~*~~ | ~~*~~ id $ | F → id |
| $ E' T' ~~id~~ | ~~id~~ $ | T' → ∈ |
| $ E' | $ | E' → ∈ |
| $ | $ | Accept |

LL(1) Grammary : Unambiguous + No Left Recursion + left-factored

① G is a gramar free from ∈-productions, then G is LL(1) if ∀ $A \to \alpha_1 | \alpha_2 | \cdots$ (aB, aC)

$First(\alpha_1) \cap First(\alpha_2) \cdots \cap First(\alpha_n) = \phi$

② G has ∈-productions ∀ $A \to \alpha | \in$

$First(\alpha) \cap Follow(A) = \phi$

then G is LL(1)

③ If G has only one alternative on RHS ∀ variable then G is LL(1)

$A \to \alpha$
$B \to \gamma$
$C \to \beta$
⋮

Eg

① $S \to a|b|c$ LL(1)

② $S \to \underset{\alpha_1}{aA} | \underset{\alpha_2}{bB}$ ✓
$A \to aA|b$ ✓
$B \to bB|c$ ✓
LL(1)

③ $S \to \underset{\alpha_1}{aAb} | \underset{\alpha_2}{Ba}$
$A \to aB|bB$
$B \to aB|b$
First($\alpha_1$) = {a}
First(B) = {a,b}
{a} ∩ {a,b} ≠ ϕ
Not LL(1)

④ $S \to aAb$
$A \to aA|\epsilon$
LL(1)
First(aA) = {a} Follow(A) = {b}
{a} ∩ {b} = ϕ

⑤ $S \to aAb|\epsilon$
$A \to aAa|\epsilon$
First(S) = {a, ϵ} Follow(S) = {$}
First(A) = {a} Follow(A) = {a,b}
{a} ∩ {a,b} ≠ ϕ
NOT LL(1)

⑥ $S \to aAb|bB$ ✓
$A \to aAb|\epsilon$ ✓
$B \to bBb|\epsilon$
NOT LL(1)

⑦ $S \to AS|dBA$
$A \to aAb|bB|\epsilon$
$B \to bBa|aA|\epsilon$
First(B) = {a,b} Follow(B) = {a}
NOT LL(1)

(8) $S \to aA \mid bB \mid db$

$A \to bA \mid Ba \mid \epsilon$

$B \to aB \mid dA \mid \epsilon$

First(A) = {a, b, d}

Follow(A) = {$, a}

NOT LL(1)

(9) $S \to AB$

$A \to aB$ LL(1)

$B \to b$

**Bottom-up parsers:** → widely used

↳ Ambiguous & Unambiguous Grammars

↳ Faster

**Eg:**

$G_1$ { $S \to aABe$, $A \to bc$, $B \to d$

TDP

BUP

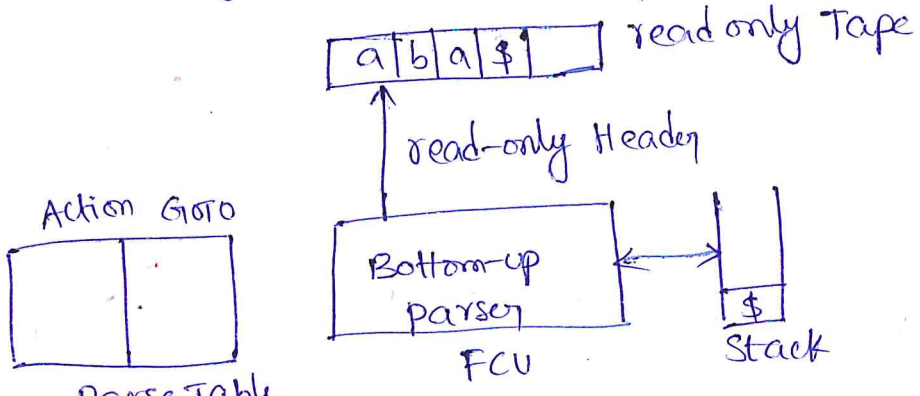
$w = abcde$

abcde → (w)

a A B e

S

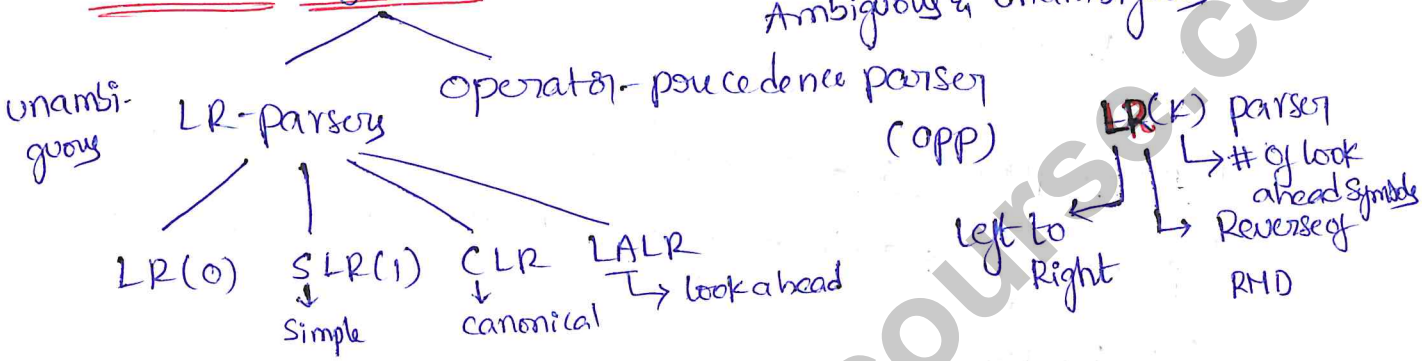
↳ Shift-Reduce parsers



operations:

- Shift ✓
- Reduce ✓
- ✓ Accept ⇒ $w \in L(G)$
- ✓ Error ⇒ $w \notin L(G)$

Overview of BUP:




Handle & Handle-pruning:

↳ part of the input string that matches the RHS of a production rule.

handles: b, c, d

$S \to aABe$

$A \to bc$

$B \to d$

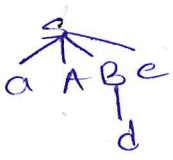
Replacing the Handle, with its LHS variable is known as Handle pruning.

$S \to aABe$

$A \to bc$

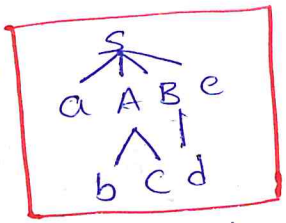
$B \to d$

$w = abcde$




⇒

Reverse of RMD




Top down parsers generate LMD

⇒ abcde
abcBe
aABe
S

## Example of Shift & Reduce operations:

$S \to AA$
$A \to aA$
$A \to b$

| Stack | input | Action |
|---|---|---|
| $ | abab$ | Shift |
| $a | bab$ | Shift |
| $ab | ab$ | Reduce $A \to b$ |
| $aA | ab$ | Reduce $A \to aA$ |
| $A | ab$ | Shift |
| $Aa | b$ | Shift |
| $Aab | $ | Reduce $A \to b$ |
| $AaA | $ | Reduce $A \to aA$ |
| $AA | $ | Reduce $S \to AA$ |
| $S | $ | Accept |

Algorithm determining the action

## LR(0): Construct the parse table:

① Construct the Augmented Grammar

$G$: $A \to aA$
$A \to b$

$G'$: $A' \to A$
$A \to aA$
$A \to b$

② Construct canonical collection of LR-items

LR-item: Item of the Compiler

$A \to \cdot abc$

$A \to a \cdot bc$
$A \to \cdot aA$
$A \to a \cdot A$

**Final-items:**

$A \to aA \cdot$

$A \to ab \cdot$

$A \to b \cdot$

**Canonical Collection:**

$C = \{ I_0, I_1, I_2 \dots I_n \}$

↑ Set of items

**closure (I):**

Closure $(A' \to \cdot A)$

$G'$: $A' \to \cdot A$
$A \to \cdot aA$
$A \to \cdot b$
} items

$\{ A' \to \cdot A$
$A \to \cdot aA$
$A \to \cdot b \}$ $I_0$

closure $(A \to \alpha \cdot B\beta)$

$B \to \cdot \gamma$

$A \to \alpha \cdot B\beta$
$B \to \cdot \gamma$

Eg 1: closure $(A \to \cdot aA) = \{ A \to \cdot aA \}$

② **GOTO(I, X):** Goto $(A \to \alpha \cdot X\beta, X) =$ Closure $(A \to \alpha X \cdot \beta)$

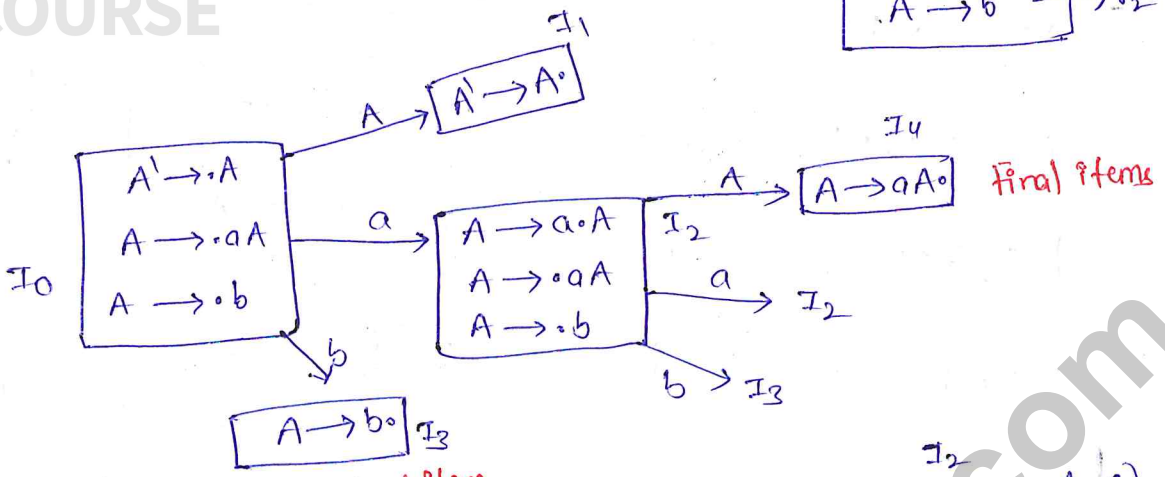
item, Symbol, item

③ Construct the graph / DFA:

G

$A \to aA$ → $r_1$

$A \to b$ → $r_2$

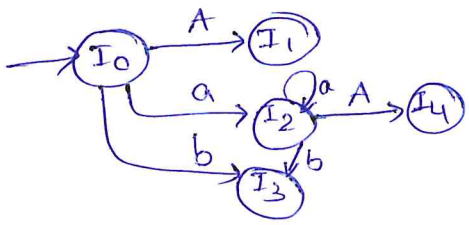



$I_2 = GOTO(A \to .aA, a)$

$= closure(A \to a.A)$

$I_3 = GOTO(A \to .b, b)$

$= closure(A \to b.)$




④ Construct parse-table:

| Terminals → | Action | | | Goto | ← Variable |
|---|---|---|---|---|---|
| | a | b | $ | A | |
| $I_0$ | $s_2$ | $s_3$ | | 1 | |
| $I_1$ | | | Accept | | |
| $I_2$ | $s_2$ | $s_3$ | | 4 | |
| $I_3$ | $r_2$ | $r_2$ | $r_2$ | | |
| $I_4$ | $r_1$ | $r_1$ | $r_1$ | | |

s: shift

r: Reduce

LR(0) parse table

⑤ Using the table to parse the input aab

| Stack | input | Action |
|---|---|---|
| $0 | aab$ | $s_2$ |
| $0 a 2 | ab$ | $s_2$ |
| $0 a 2 a 2 | b$ | $s_3$ |

| Stack | input | Action |
|---|---|---|
| $0a2a2b3 | $ | $r_2$ $A \to b$ |
| $0a2a2A4 | $ | $r_1$ $A \to aA$ |
| $0a2A4 | $ | $r_1$ $A \to aA$ |
| $0A1 | $ | Accept |


$w \in L(G)$

## LR(0) parser: More Examples:

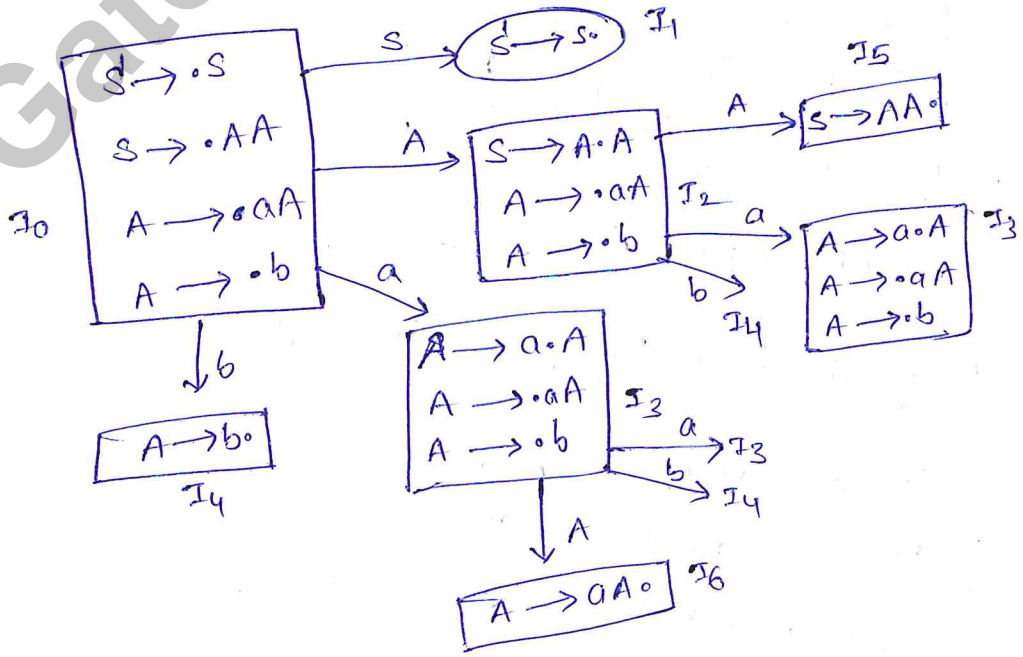
$w = abb$

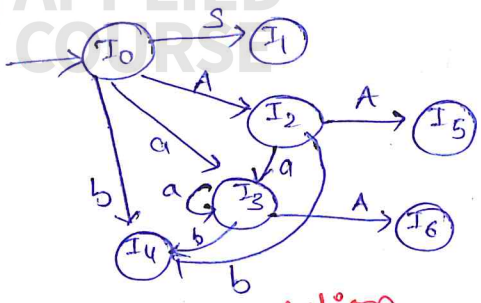
$G_1$: $S \to AA$ $r_1$
$A \to aA$ $r_2$
$A \to b$ $r_3$

① Augmented Grammar

$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

② Find canonical collection of items






| | Action | | | GOTO | |
|---|---|---|---|---|---|
| | a | b | $ | S | A |
| $I_0$ | $S_3$ | $S_4$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_3$ | $S_4$ | | | 5 |
| $I_3$ | $S_3$ | $S_4$ | | | 6 |
| $I_4$ | $r_3$ | $r_3$ | $r_3$ | | |
| $I_5$ | $r_1$ | $r_1$ | $r_1$ | | |
| $I_6$ | $r_2$ | $r_2$ | $r_2$ | | |

| Stack | input | Action |
|---|---|---|
| $0 | abb$ | $S_3$ |
| $0a3 | bb$ | $S_4$ |
| $0a3b4 | b$ | $r_3$ A→b |
| $0a3A | b$ | 6 |
| $0a3A6 | b$ | $r_2$ A→aA |
| $0A | b$ | 2 |
| $0A2 | b$ | $S_4$ |
| $0A2b4 | $ | $r_3$ A→b |
| $0A2A5 | $ | $r_1$ S→AA |
| $0S | $ | 1 |
| $0S1 | $ | Accept |

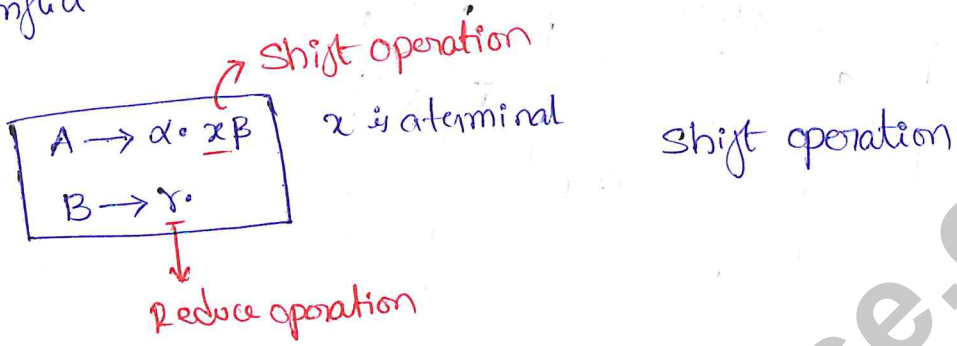
abb
w∈L(G)

## Conflicts: LR(0)

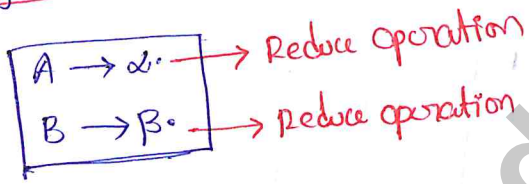
↳ if ∃ multiple entries in a cell in the parse-table.

① SR Conflict

② RR Conflict




### RR Conflict:

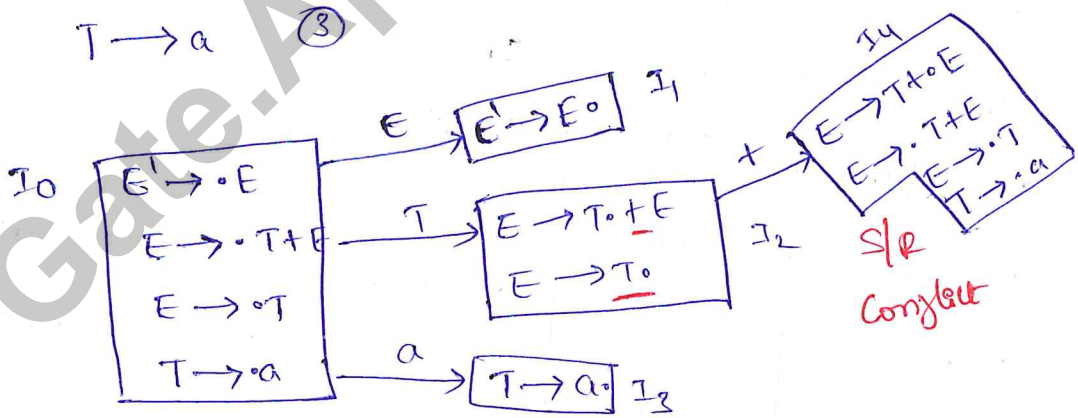



### Eg 1:

$E \to T+E$ ①

$E \to T$ ②

$T \to a$ ③




GOTO($I_4$, E) → $E \to T+E \cdot$ $I_5$

GOTO ($I_4$, T) → $I_2$

GOTO ($I_4$, a) → $I_3$

| | + | a | $ | E | T |
|---|---|---|---|---|---|
| $I_0$ | | $S_3$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_4/r_2$ | $r_2$ | $r_2$ | | |
| $I_3$ | $r_3$ | $r_3$ | $r_3$ | | |
| $I_4$ | | $S_3$ | | 5 | 2 |
| $I_5$ | $r_1$ | $r_1$ | $r_1$ | | |

S/R Conflict

NOT LR(0) Grammar

NOT LL(1) Grammar

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

LR(0) Grammar

↳ if LR(0) parse table has no Conflicts.

SLR(1) parser:

↳ Simple Extension to LR(0) parser.

⇓

parse-table (Min 8)

G

$E \rightarrow T+E$ ①

$E \rightarrow T$ ②

$T \rightarrow a$ ③

Instead of placing the reduce operation under all the terminals, place them under follow (LHS) variables.

SLR(1) parse table for the Grammar

| | + | a | $ | E | T |
|---|---|---|---|---|---|
| $I_0$ | | $S_3$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_4$ | | $r_2$ | | |
| $I_3$ | $r_3$ | | $r_3$ | | |
| $I_4$ | | $S_3$ | | 5 | 2 |
| $I_5$ | | | $r_1$ | | |

$E \rightarrow T.$

Follow(E) = {$}

$T \rightarrow a.$

Follow(T) = {$, +}

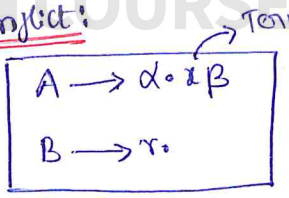
$E \rightarrow T+E.$ $r_1$

$E \rightarrow T+E.$

Follow(E) = {$}

## SR & RR Conflicts in SLR(1) parse-table:

**SR Conflict:**




SR-Conflict in LR(0)

SLR(1)

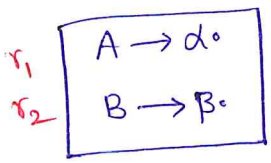
First(x) ∩ Follow(B)

⇓

x ∩ Follow(B) ≠ ϕ

S/R Conflict in SLR(1)

**RR Conflict:**




Follow(A) ∩ Follow(B) ≠ ϕ

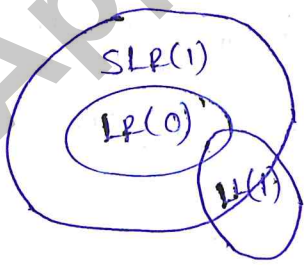
RR Conflict

## SLR(1) Grammar:

↳ if SLR(1) parse table has no Conflicts.

↓

Fewer entries than LR(0)

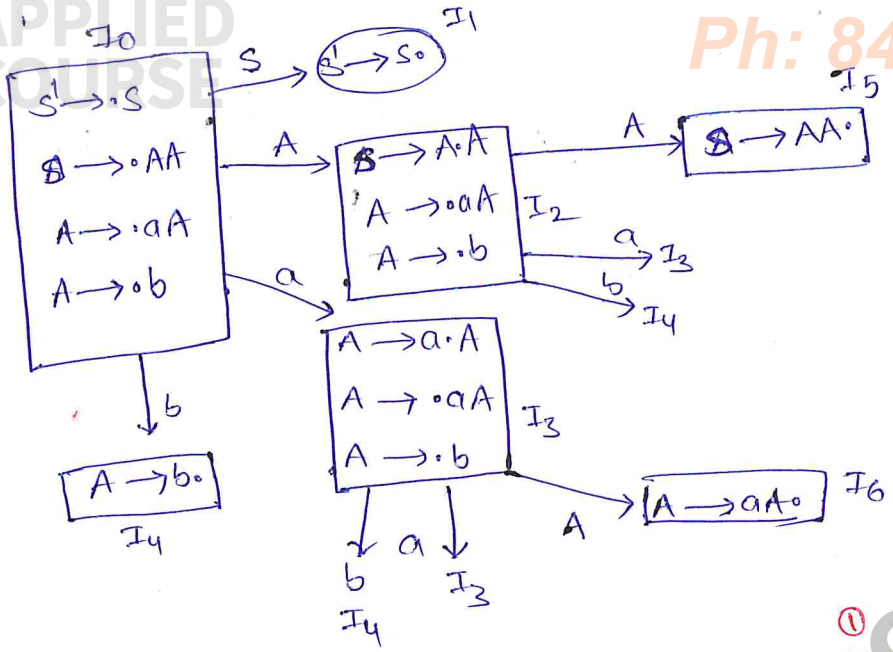



SLR(1) is more powerful than LR(0) parser.

LL(1) is not a proper subset of LR(0)

G1: S' → •S

1. S → •AA
2. A → •aA
3. A → •b




① S → AA
② A → aA
③ A → b

| | a | b | $ | S | A |
|---|---|---|---|---|---|
| $I_0$ | $S_3$ | $S_4$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_3$ | $S_4$ | | | 5 |
| $I_3$ | $S_3$ | $S_4$ | | | 6 |
| $I_4$ | $r_3$ | $r_3$ | $r_3$ | | |
| $I_5$ | | | $r_1$ | | |
| $I_6$ | $r_2$ | $r_2$ | $r_2$ | | |

Follow(A) = {a, b, $}

Follow(S) = {$}

Both LR(0) SLR(1)

No S/R & R/R conflicts.

## CLR(1) & LALR(1):

**LR(1) items:** LR(0) items + Lookahead

**Eg:** S → AA
A → aA
A → b

⇒ Augmented Grammar

S' → S
S → AA
A → aA
A → b

$I_0$:
$S' \to .S, \$$
$S \to .AA, \$$
$A \to .aA, a/b$
$A \to .b, a/b$

$A \to \alpha . B\beta, \$$ First$(B\$)$
$B \to \gamma ., $ First$(B\$)$

1. $S \to AA$
2. $A \to aA$
3. $A \to b$

GOTO $(I_0, S)$

$I_1$: $S' \to S., \$$

GOTO $(I_0, A)$

$I_2$:
$S \to A.A, \$$
$A \to .aA, \$$
$A \to .b, \$$

GOTO $(I_2, A)$ $\Rightarrow$ $I_5$: $S \to AA., \$$

GOTO $(I_2, a)$

$I_6$:
$A \to a.A, \$$
$A \to .aA, \$$
$A \to .b, \$$

GOTO $(I_6, A)$ $I_9$: $A \to aA., \$$

GOTO $(I_6, a)$ $I_6$

GOTO $(I_6, b)$ $I_7$

GOTO $(I_0, a)$

$I_3$:
$A \to a.A, a/b$
$A \to .aA, a/b$
$A \to .b, a/b$

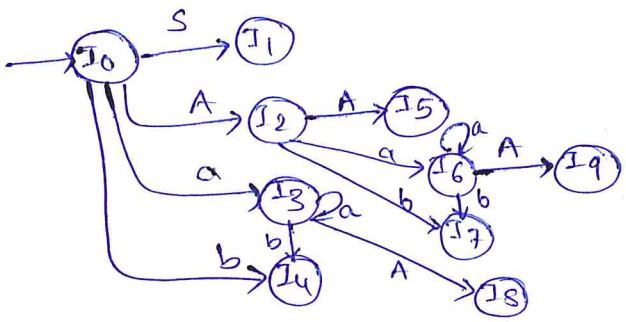
GOTO $(I_2, b)$ $I_7$: $A \to b., \$$

GOTO $(I_3, A)$ $I_8$: $A \to aA., a/b$

GOTO $(I_0, b)$ $I_4$: $A \to b., a/b$

GOTO $(I_3, b)$ $I_4$

GOTO $(I_3, a)$ $I_3$:
$A \to a.A, a/b$
$A \to .aA, a/b$
$A \to .b, a/b$



| | a | b | $ | S | A |
|---|---|---|---|---|---|
| $I_0$ | $S_3$ | $S_4$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_6$ | $S_7$ | | | 5 |
| $I_3$ | $S_3$ | $S_4$ | | | 8 |
| $I_4$ | $r_3$ | $r_3$ | | | |
| $I_5$ | | | $r_1$ | | |
| $I_6$ | $S_6$ | $S_7$ | | | 9 |
| $I_7$ | | | $r_3$ | | |
| $I_8$ | $r_2$ | $r_2$ | | | |
| $I_9$ | | | $r_2$ | | |

Number of Stats in CLR(1) ≥ SLR(1)

CLR(1) parser

⇓

placing $r_i$ in fewer cells than SLR(1)

⇓

Lesser chance of Conflict

LR(0): Full row

SLR(1): Follow of LHS

CLR(1): Look ahead

## LALR(1) parser:

Merge the Stats

3&6, 4&7, 8&9

$I_{36}$ $I_{47}$ $I_{89}$

| | a | b | $ | S | A |
|---|---|---|---|---|---|
| $I_0$ | $S_{36}$ | $S_{47}$ | | 1 | 2 |
| $I_1$ | | | Accept | | |
| $I_2$ | $S_{36}$ | $S_{47}$ | | | 5 |
| $I_{36}$ | $S_{36}$ | $S_{47}$ | | | 89 |
| $I_{47}$ | $r_3$ | $r_3$ | $r_3$ | | |
| $I_5$ | | | $r_1$ | | |
| $I_{89}$ | $r_2$ | $r_2$ | $r_2$ | | |

# No. of Stats in LR(0) = SLR(1) = LALR(1) ≤ CLR(1)

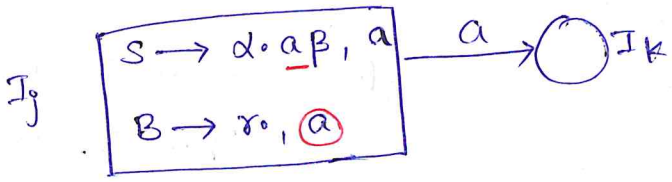
# CLR(1) and LALR(1) Grammars and Examples:



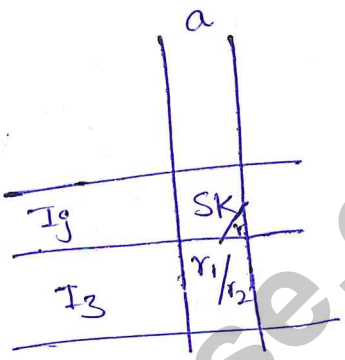
CLR(1) → LR(1)

SLR(1), LR(0)

**Conflicts** : S/R & R/R




S/R Conflict in CLR(1)




A → α · , a
B → β · , a

R/R Conflict in CLR(1)

G1: ① S → AaAb
② S → BbBa
③ A → d
④ B → d

⇒ I0

S' → · S, $
S → · AaAb, $
S → · BbBa, $
A → · d, a
B → · d, b

GOTO (I0, S)

S' → S · , $ — I1

GOTO (I0, A)

S → A · aAb, $ — I2

GOTO (I0, B)

S → B · bBa, $ — I3

GOTO (I0, d)

A → d · a
B → d · b — I4

GOTO (I2, a)

S → Aa · Ab, $
A → · d, b — I5

GOTO($I_3$, b)

| $S \to Bb \cdot Ba, \$$ <br> $B \to \cdot d, a$ | $I_6$ |
|---|---|

GOTO($I_5$, A)

| $S \to AaA \cdot b, \$$ | $I_7$ |
|---|---|

GOTO($I_7$, b)

| $S \to AaAb \cdot, \$$ | $I_{11}$ |
|---|---|

GOTO($I_5$, d)

| (3) $A \to d \cdot, b$ | $I_8$ |
|---|---|

GOTO($I_9$, a)

| $S \to BbBa \cdot, \$$ | $I_{12}$ |
|---|---|

GOTO($I_6$, B)

| $S \to BbB \cdot a, \$$ | $I_9$ |
|---|---|

GOTO($I_6$, d)

| (4) $B \to d \cdot, a$ | $I_{10}$ |
|---|---|

No S/R and R/R Conflicts. CLR(1) Grammar.

| | a | b | d | $ | S | A | B |
|---|---|---|---|---|---|---|---|
| $I_0$ | | S4 | | | 1 | 2 | 3 |
| $I_1$ | | | | Accpt | | | |
| $I_2$ | S5 | | | | | | |
| $I_3$ | | S6 | | | | | |
| $I_4$ | $r_3$ | $r_4$ | | | | | |
| $I_5$ | | | S8 | | | | 7 |
| $I_6$ | | S10 | S10 | | | | 9 |
| $I_7$ | | S11 | | | | | |
| $I_8$ | | $r_3$ | | | | | |
| $I_9$ | S12 | | | | | | |
| $I_{10}$ | $r_4$ | | | | | | |
| $I_{11}$ | | | | $r_1$ | | | |
| $I_{12}$ | | | | $r_2$ | | | |

No conflicts

CLR(1) Grammar

## LALR(1) Grammar & Conflicts:

⇓

Combining states in CLR(1) parse table.

→ If CLR(1) parse table, has no SR conflicts then no SR conflicts in LALR(1).

→ If CLR(1) parse table has no RR conflicts, then there may/may not exists RR conflicts in LALR(1).

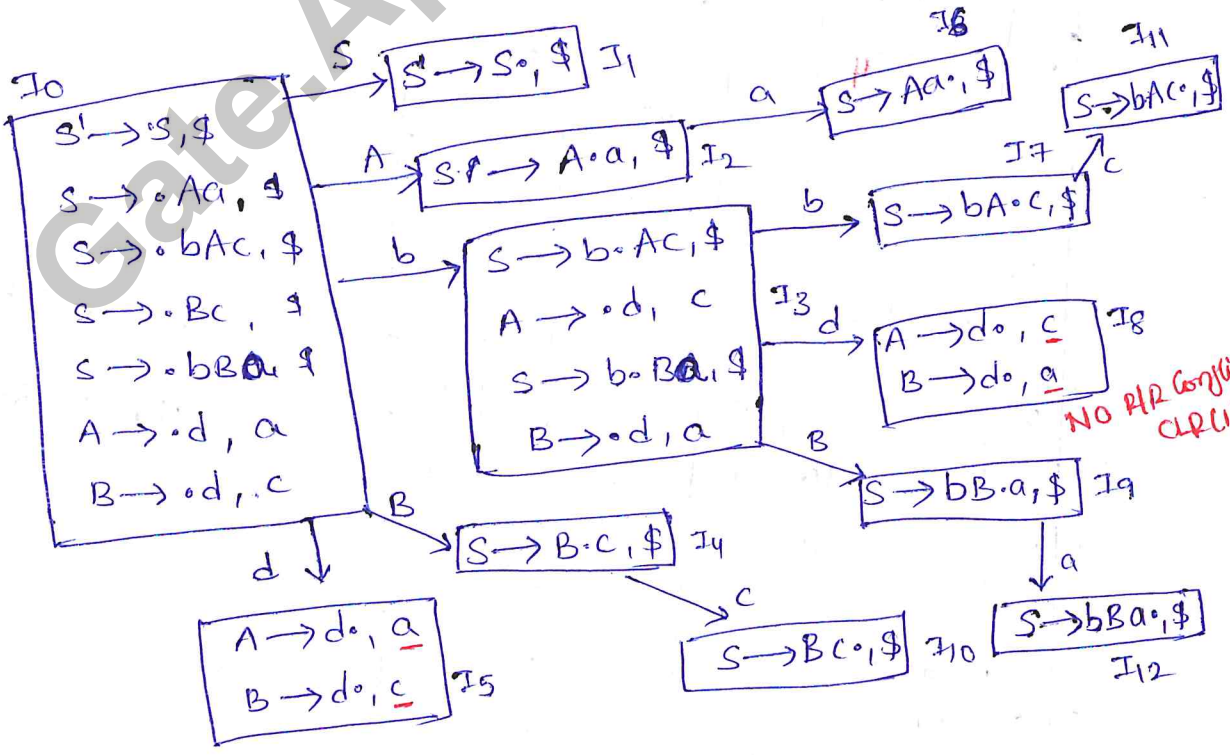
**Eg1:**

1 $S \to Aa$
2 $S \to bAc$
3 $S \to Bc$
4 $S \to bBa$
5 $A \to d$
6 $B \to d$

⇒

$S' \to S$
$S \to Aa$
$S \to bAc$
$S \to Bc$
$S \to bBa$
$A \to d$
$B \to d$

CLR(1) but not LALR(1)



No Conflicts in CLR(1)

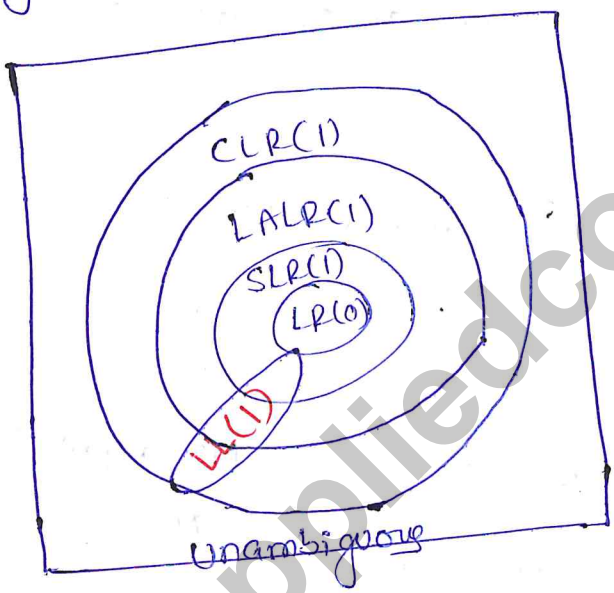
R/R Conflict in LALR(1)

$I_{58}$

| | |
|---|---|
| (5) | $A \to d\cdot$ a/c |
| (6) | $B \to d\cdot$ a/c |

| | a | c |
|---|---|---|
| $I_{58}$ | r5/r6 | r5/r6 |

Every LALR(1) is CLR(1), but not vice versa.




CLR(1) parser is more powerfull.

LALR(1) parser → practice

↑

fewer states

Operator precedence parser:

↳ BOP, ambiguous & unambiguous

↳ cannot work for all CFG

↳ +, −, *, /, %, ----

Operator Grammar:

↳ CFG { no ∈-transitions
No AB on RHS
~ Adjacent Variables

AaB

all terminals
↓
operators

Eg: ① $S \to AB$
$A \to a$
$B \to b$

NOT OPG
operator Grammar

② $S \to AaB$
$A \to aA \mid b$
$B \to bB \mid a$

No $\epsilon$-prod
No Adjacent Variables

③ $S \to AaB$
$A \to aA \mid b$
$B \to bB \mid \epsilon$

Not an operator Grammar

④ $E \to EAE \mid id$
$A \to + \mid - \mid *$

Adjacent variables
→ Not OPG

⇓

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

No adjacent variables
No $\epsilon$-productions
operator Grammar

↙
Ambiguous Grammar

⇒ G: $E \to E+E$
$E \to E*E$
$E \to id$

Ambiguous opr. Gr

w2 = id + id * id $

E E
| |

<· PUSH/Shift
·> Pop & Reduce
Empty Error

| Stack | input | Action |
|---|---|---|
| $ | id + id * id $ | PUSH |
| $ id | + id * id $ | pop + Reduce |
| $ E | + id * id $ | PUSH |
| $ E + | id * id $ | PUSH |
| $ E + id | * id $ | pop & Reduce |

| | id | + | * | $ |
|---|---|---|---|---|
| id | E | ·> | ·> | ·> |
| + | <· | ·> | <· | ·> |
| * | <· | ·> | ·> | ·> |
| $ | <· | <· | <· | E |

n×n

Precedence Table

2+3*5 = 2+15 = 17

2+3+4 ⇒ 5+4 ⇒ 9

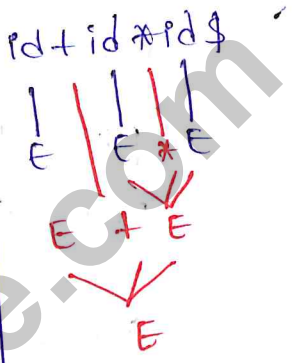
3+2*5

↓ LA

id + id * id $

Operator precedence Grammar

| Stack | input | Action |
|---|---|---|
| $ + | <· * id $ | PUSH |
| $ + * | <· id $ | PUSH |
| $ + * id | ·> $ | pop & Reduce |
| $ + * | ·> $ | pop & Reduce |
| $ + | ·> $ | pop & Reduce |
| $ | $ | Accept |

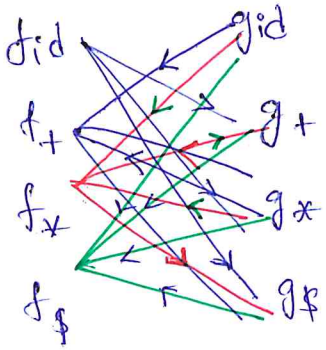



O(n²) to O(2n) table:

| | id | + | * | $ |
|---|---|---|---|---|
| id | | ·> | ·> | ·> |
| + | <· | ·> | <· | ·> |
| * | <· | ·> | ·> | ·> |
| $ | <· | <· | <· | |

Rel Table

① Functional Table No-cycles




② $f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

Compute Largest path

③

| | + | * | id | $ |
|---|---|---|---|---|
| f | 2 | 4 | 4 | 0 |
| g | 1 | 3 | 5 | 0 |

Functional Table

O(2n)

$f_{id}$ $g_+$

4 > 1

Compare (id, +)

Compare (+, *)

$f_+$ $g_*$

2 <= 3

Compare (id, id)

$f_{id}$ > $g_{id}$

Error

Functional table captures all the information from the precedence table, but they will not get the information in Empty cell in precedence table.

Error detection will be done efficiently in precedence Table.

## Syntax Directed Translation: (SDT)

Src. Code

↳ Lexical Analysis → Syntax Analysis

G

| | |
|---|---|
| S → AB | {print (*)} ① |
| A → a | {print (a)} ② |
| A → b | {print (b)} ③ |

↓ PT

Semantic Analysis CFG + Actions = SDT

↓

ICG

↓

code optimization

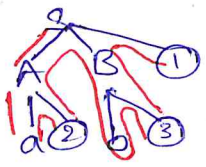
↓

Code Generation.

CFG + Semantic Actions / Rules

Type checking

let w = ab




parse tree

TDP

Top down

L → R

output: ab *

input = ab
output : ab* postfix Expression

## What can be done using SDT?

Dragon book

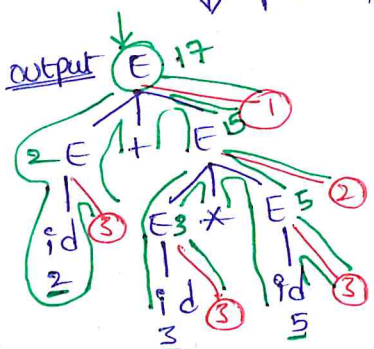
→ Type-checking
→ Algebric Expression Evaluation
→ Syntax-tree
→ Verify variable declaration

## Annotated Parse Tree: (Decorated parse Tree)

G: $E \rightarrow E+E$ { E.val = E.val + E.val } ①
$E \rightarrow E*E$ { E.val = E.val * E.val } ②
$E \rightarrow id$ { E.val = id.val } ③

CFG — action

w = 2+3*5
↓ lex. Analysis (Symbol Table)
↓
w = id + id * id
↓ parser (TDP)



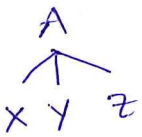

Top-down
L → R

## SDT:

E.val / id.val { Evaluate algebraic Expression }

## Types of attribute: Synthesized & Inherited

① $A \rightarrow XYZ$




S is an attribute corresponding to A

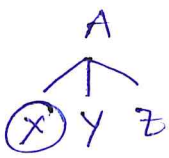
children only

$A.S = f(X.S, Y.S, Z.S)$

Attribute S is known as Synthesized attribute.

## Inherited Attribute:

$A \rightarrow XYZ$




A.S
X.S
Y.S
Z.S

$X.S = f(A.S, Y.S, Z.S)$

parent and/or Siblings.

Attribute S : Inherited Attribute

## Constructing SDT with examply:

① Construct SDT for expression Evaluation

$E \rightarrow E+T$ { E.val = E.val + T.val } ①

$E \rightarrow T$ { E.val = T.val } ②

$T \rightarrow T*F$ { T.val = T.val * F.val } ③

$T \rightarrow F$ { T.val = F.val } ④

$F \rightarrow id$ { F.val = id.val } ⑤

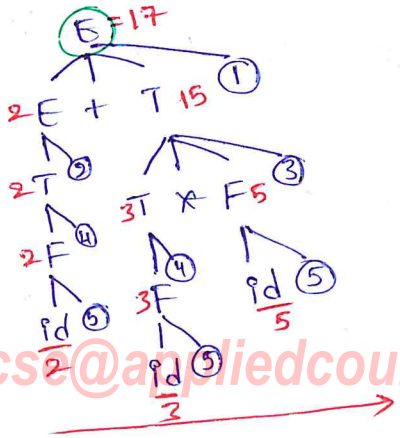
Actions

W = 2+3*5

↓ LA + S.T

id + id * id

↓




Top down

L → R

input = 2+3*5

output = 17

## ② SDT for infix to postfix Conversion:

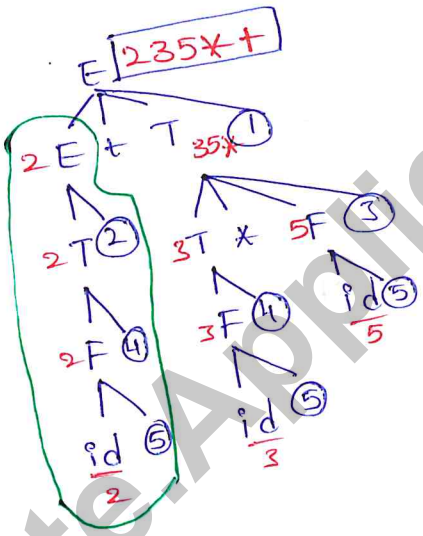
input: 2+3*5
output: 235*

| | |
|---|---|
| $E \rightarrow E+T$ | { E.val = E.val T.val + } |
| $E \rightarrow T$ | { E.val = T.val } |
| $T \rightarrow T*F$ | { T.val = T.val F.val * } |
| $T \rightarrow F$ | { T.val = F.val } |
| $F \rightarrow id$ | { F.val = id.val } |
| CFG | Actions |

2+3*5

↓ LA+ST

id+id*id




Top down

L → R

parsing: stack

## ③ SDT for Type-checking:

| | |
|---|---|
| $E \rightarrow E+T$ | { if (E.type == T.type) && (E.type == int) then E.type = int else Error } |
| $E \rightarrow T$ | { E.type = T.type } |
| ~~$T \rightarrow F$~~ | |
| $T \rightarrow T*F$ | { if (T.type == F.type) && (F.type == int) then T.type = int else Error } |
| $T \rightarrow F$ | { T.type = F.type } |
| $F \rightarrow id$ | { F.type = id.type } |

$W_1 = 2+3*5$ ✓

$W_2 = 2+3*a$ ✓

$W_3 = 2+3.2*a$ — ERROR

id id id

## (4) SDT to Count parenthesis:

$A \rightarrow (A)$ { A.Count = A.Count + 1 } (1)

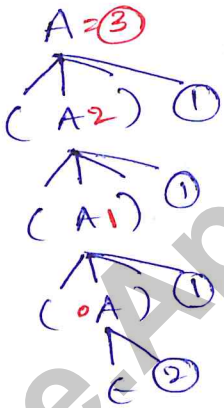
G1:

$A \rightarrow \epsilon$ { A.Count = 0 } (2)

CFG — Actions

W = ( ( ( ) ) ) input

3 output




Top down

L → R

## (5) SDT to Construct Syntax Trees


$E \rightarrow E+T$ { E.ptr = mknode(E.ptr, +, T.ptr) }

L R

$E \rightarrow T$ { E.ptr = T.ptr }

$T \rightarrow T*F$ { T.ptr = mknode(T.ptr, *, F.ptr) }

$T \rightarrow F$ { T.ptr = F.ptr }

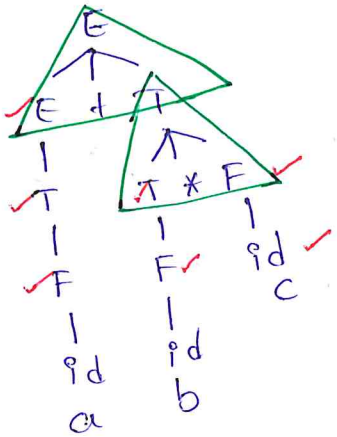
$F \rightarrow id$ { F.ptr = mknode(NULL, id, NULL) }

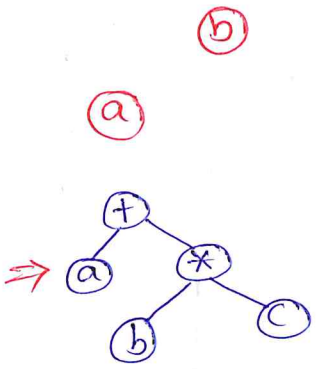
input: a+b*c

output:

Parse-tree: Terminals & non-terminals

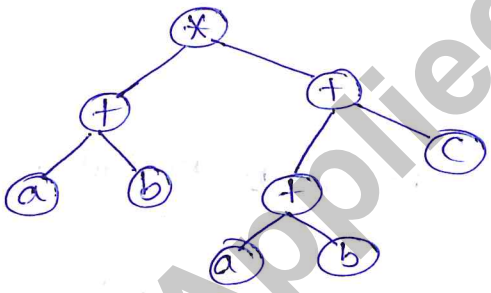
In a syntax-tree all ~~trees~~ are terminals nodes



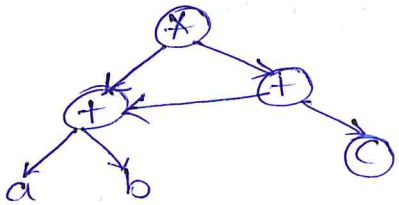




Syntax Tree

## (6) SDT for expression-eval DAGs:



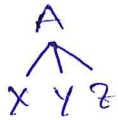

Syntax-tree




Avoids duplicate Computation

### Types of SDT:

| S-attributed SDT | L-attributed SDT |
|---|---|
| ① Only uses Synthesized attributes | ① uses both Synthesized & inherited attributes ⇓ are restricted to using attributes from parent & left sibling only. |




$X.S = A.S$

$Y.S = \{A.S, X.S\}$

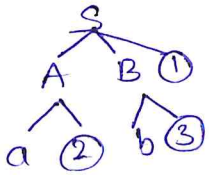
## S-attributed

② actions are placed @ the end of RHS

$S \rightarrow AB$ { print x } ①

$A \rightarrow a$ { print a } ②

$B \rightarrow b$ { print b } ③




③ Attributes can be evaluated using Topdown & bottomup methods

## L-attributed

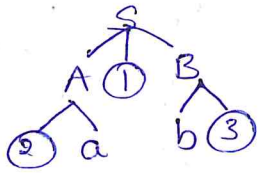
② Actions can be placed any where on RHS.

$S \rightarrow A$ ① $B$

$A \rightarrow$ ② $a$

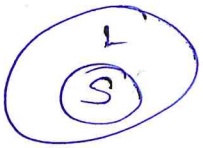
$B \rightarrow b$ ③

$w = ab$




③ Depth First L → R

**Note:** Every S-attributed SDT is an L-attributed SDT.




### L-attributed SDT:

$S \rightarrow A$ ① $B$

$A \rightarrow$ ② $a$

$B \rightarrow b$ ③

### S-attributed SDT

$S \rightarrow ARB$ { }

$R \rightarrow \epsilon$ ①

$A \rightarrow Xa$ { }

$X \rightarrow \epsilon$ ②

$B \rightarrow b$ ③

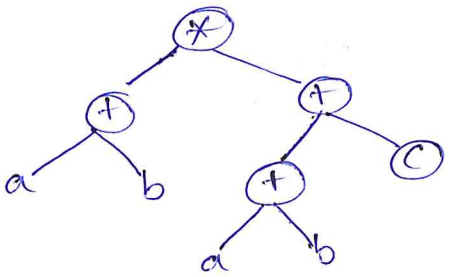
## ICG + 3- Address codes: Platform Independant



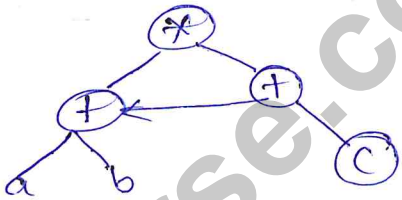
Improve code-generation efficiency.

(a+b) * (a+b+c)

### Syntax Trees (Abstract Syntax Trees)




① Syntax Tree




② DAG

③ postfix : ab+ ab+c+*

↳ one way of representing intermediate code.

④ Three-Address Code:

$t_1 = a+b$

$t_2 = a+b$

$t_3 = t_2+c$

$t_4 = t_1 * t_3$

3-address code.

In 3-address code atmost three addresses are there.



## Representation of 3-address code:

$x = a + b * c$

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

① Quadraples

| | | | op |
|---|---|---|---|
| $t_1$ | b | c | * |
| $t_2$ | a | $t_1$ | + |
| x | | $t_2$ | = |

② Triply

① bc *
② a ① +
③ x ② =

No Temp Var

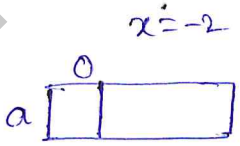
③ Indirect Triply

(101) bc *
(102) a (101) +
(103) x (102) =

pointers
reuse
Computation

## Types of 3-addr codes:

① $x = y$ op $z$ ✓
② $x =$ op $y$ ✓
③ $x = y$ ✓
④ $x = *y$ ✓
⑤ $x = \&y$ ✓
⑥ $x = a[i]$ ✓
⑦ $a[i] = x$ ✓
⑧ GOTO L ✓ unconditional Jump
⑨ if (expr) GOTO L ✓
⑩ $y = f(x_1, x_2 \cdots x_n)$ X
more than 3-addresses



① $x = a-b-c+d/e$

$t_1 = a-b$

$t_2 = t_1-c$

$t_3 = d/e$

$t_4 = t_2+t_3$

$x = t_4$

②
```
int x=1;
int y=2;
int z=x+y;
print(z);
```

$x = 1$

$y = 2$

$z = x+y$

goto L

L: print(z)

③
```
if (x>y)
    print("TOC")
else
    print("CD")
```

① If (x>y) GOTO L5 Cond-Jump

② ~~goto L2~~

③ L1: print("CD")

④ goto ⑥

⑤ L2: print("TOC")

⑥ END/EXIT

④
```
for(int i=1; i<=10; i++)
{
    int x = 2*i;
    print(x);
}
```