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++

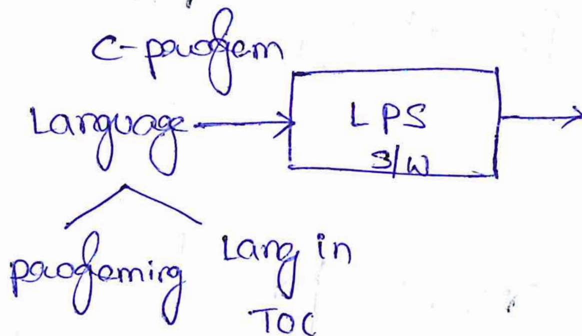
Java Compiler

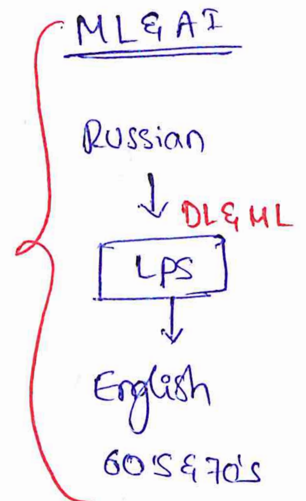
1.c → gcc → object code

?

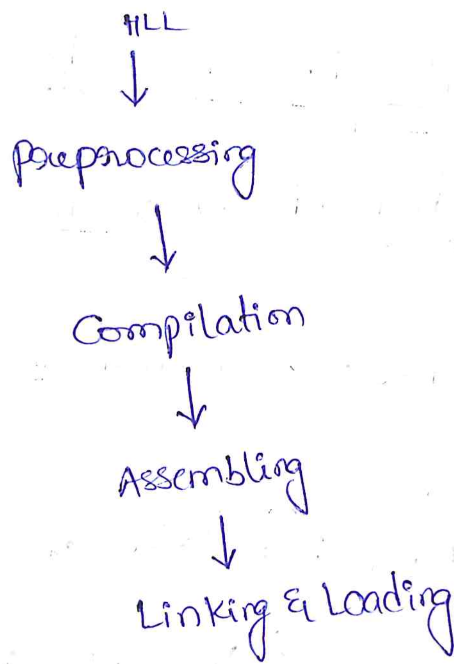
$ gcc 1.c
$ ./a.out

Language processing System: (LPS)





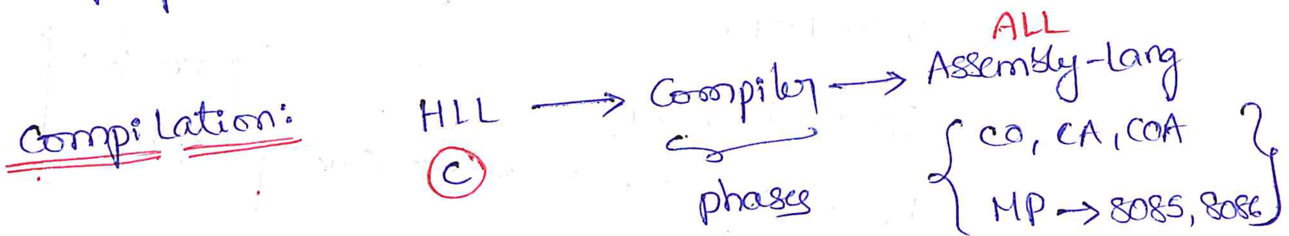
②

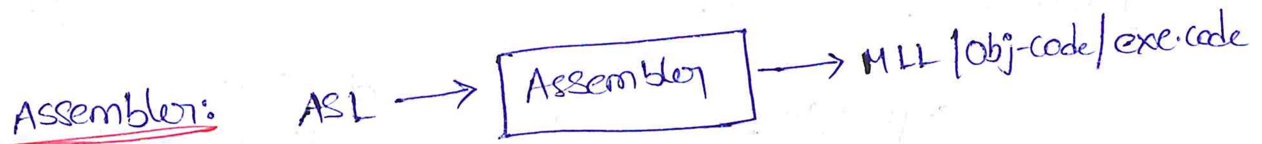


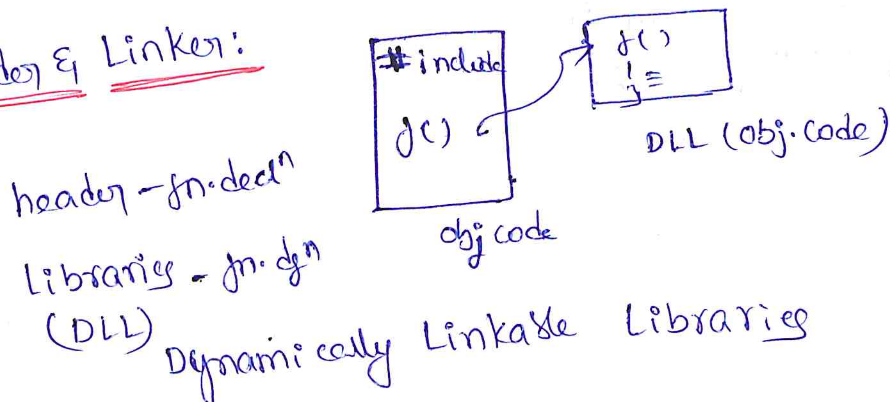
Preprocessing:

include <stdio.h> } Header file
include <stdlib.h>
define PI 3.1415 ← Macros

Preprocessor Handles all header files and macros.

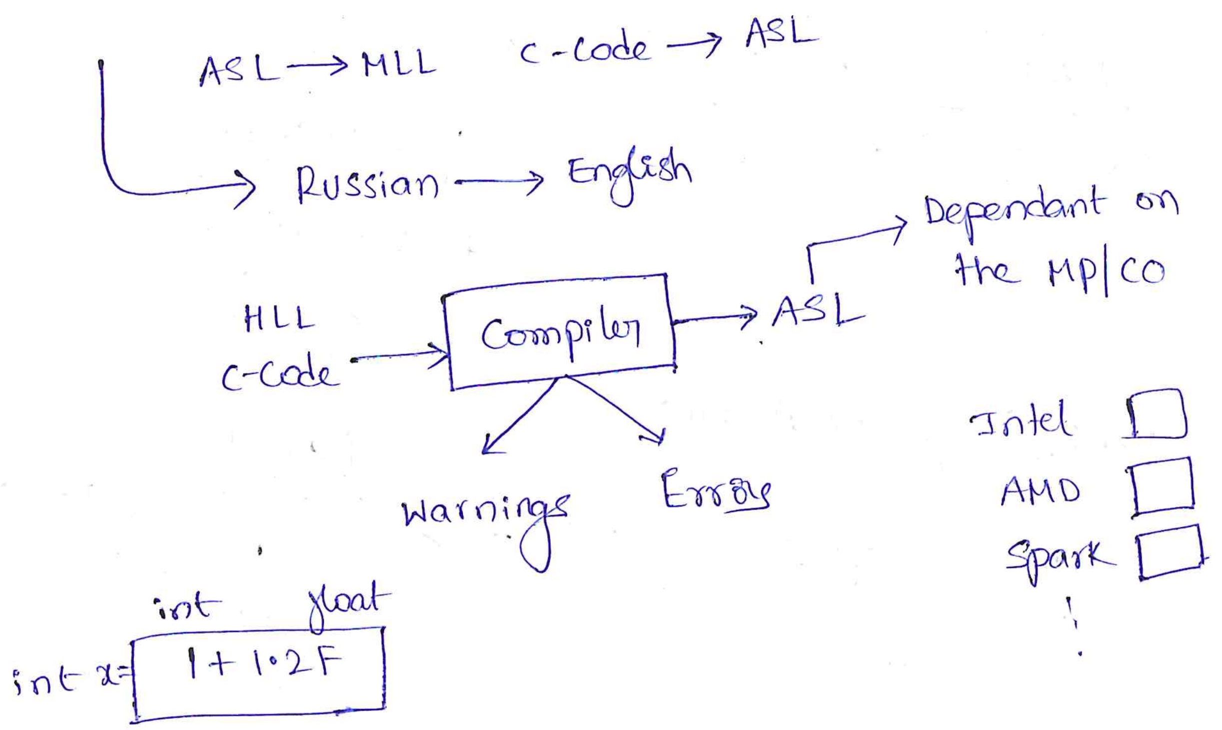




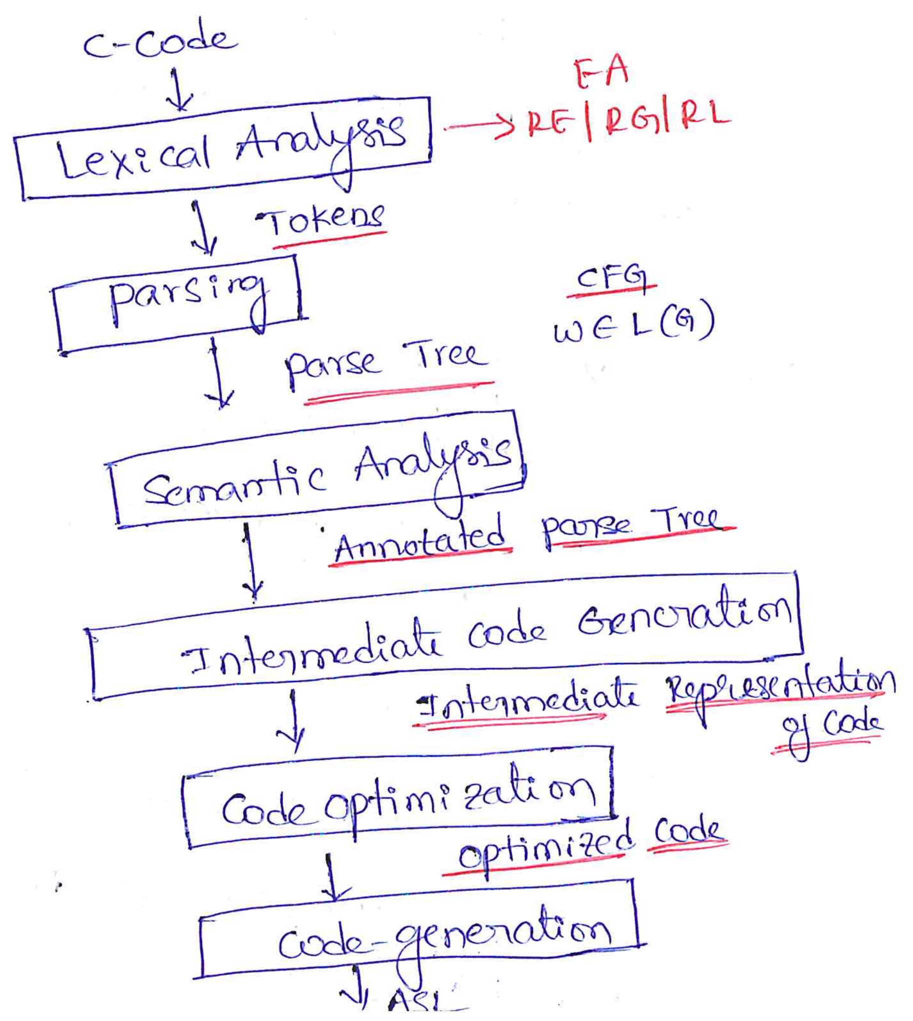


Loads all necessary files into memory.

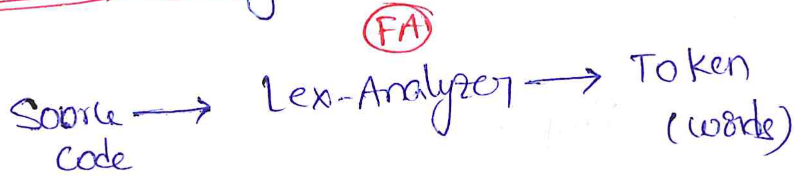
Translators : Assemblers, Compilers



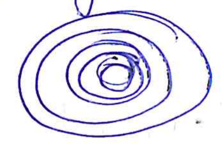
Phases of a Compiler:



Lexical-Analysis:



peeling an Onion



{ int x=10; → EOS
KW id op constant

KW: Keyword

Symbol | char

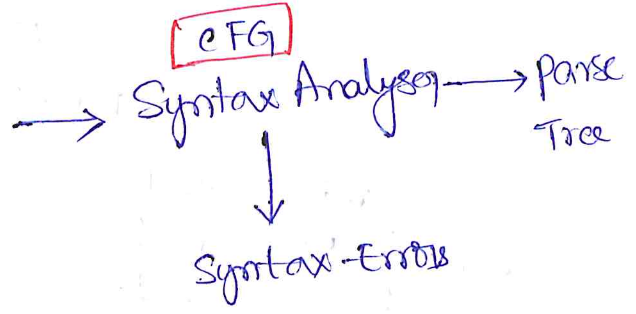
id: identifier

op: operator

Parsing | Syntax Analysis:

KW id op Constant EOS

Tokens



c-prog: CFG

int 10=x;

KW const op id EOS

Java: CFG

$w \in L(G)$? → decidable

⇓

CYK

Eg: x = a+b;

↓ lex-Analyser

id eqop id op id; (Tokens)

w

$w \in L(G)$?

S → id eqop A

→ id eqop id op id ;

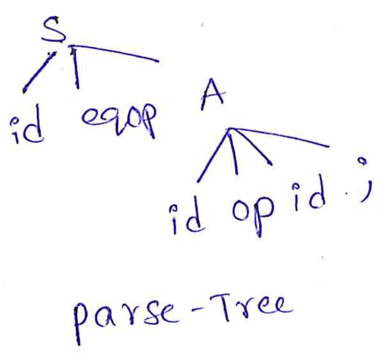
→ w

CFG:

S → id eqop A

A → id op id;

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



parse-Tree

Semantic Analysis:

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

```
int x=10;
bool b= TRUE;
char c= 'a';
int x= b+c;
       ↓  ↓
       X  ASCII ('a')
```

bool → int X

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.

6

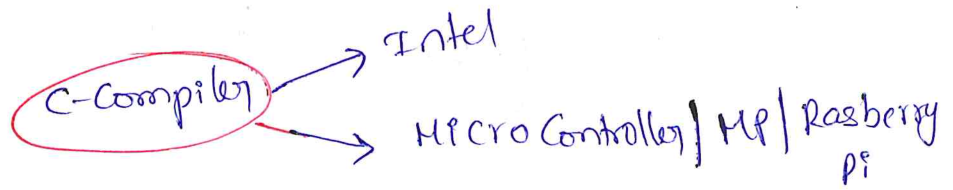
int x = y || 1;
⇓
int x = y | 1;

} Short-Circuiting

Micro processor

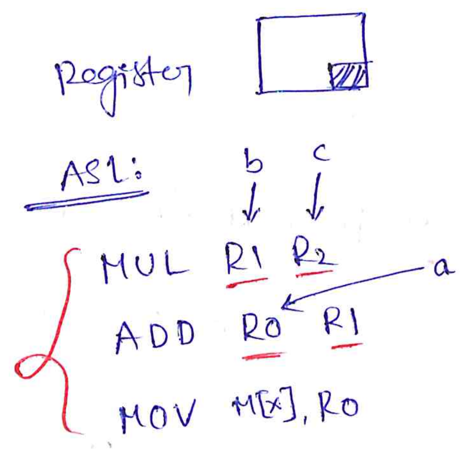
Front-End :: Lexical Analysis ⟶ Machine Independant optimization

Back-End: Machine dep optimization to code-gen



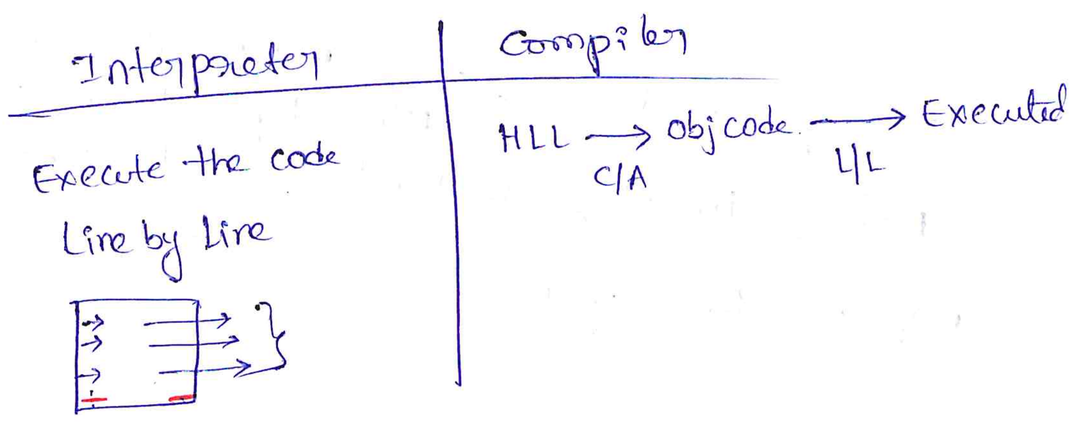
Code Generation:

x = a + b * c;
⇓
$t_1 = b * c;$
$t_2 = a + t_1;$
$x = t_2$



R1 : b*c
R0 : a+b*c

code-optimization is optional



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

Symbol Table & Error-Handler

var, funs, Constants

Stores information related to various symbols

Source-code
↓
Lexical Analysis
↓
Syntax Analysis
↓
Semantic Analysis
↓
Code-generation (ICG, CO, CG)
↓
ASL

Symbol Table ↔ Lexical Analysis, Syntax Analysis, Semantic Analysis, Code-generation

Error Handler ← Lexical Analysis, Syntax Analysis, Semantic Analysis, Code-generation

Example:

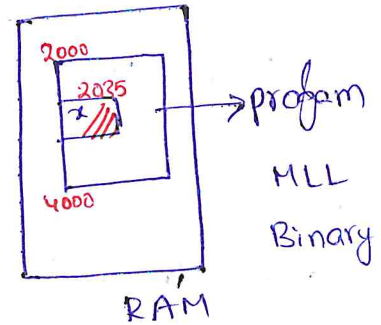
```
extern double bar (double x);

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

K	Value	
Symbol Name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	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 Errors:** nti x=10; int x=10;

   nti → Not KW

2. **Syntax-Errors:** int x=10 int x=10;

   ↓ No semi-colon

   if (x>y) { } , (x>y) if { } ✗

3. **Semantic-Errors:**

   int x = "abc"; → "abc" Array of char

   ↓ KW ↓ id ↓ eqop ↓ const ;

   int x=10;

   int y=20;

   z=x+y;

   z → Not declared

4. **Fatal Errors:** → 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

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

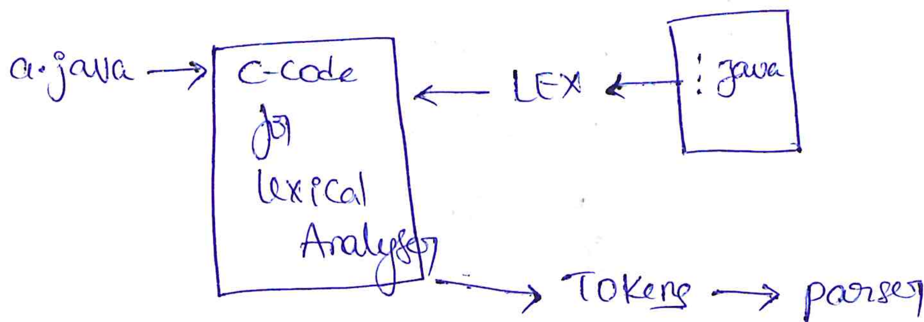
Table / Arrays: KW, op ...

Rules: Identifiers, 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/5Mbolqa HAjk

Syntax-Analysis | Parsing:

↳ Most of this course

CFG, parse-tree, derivation

↑

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

Lex-Analyser 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: $E \rightarrow E=E \mid E+E \mid E*E \mid id$

2 + 3 * b;

↓ Lex Analysis

$id_1 + id_2 * id_3$

↓ Syntax Analysis

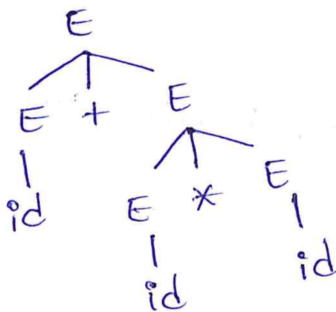
W (String) = $id_1 + id_2 * id_3$

$w \in L(G)$?

(Start-symbol)

Derivation
$E \rightarrow E+E$
$\rightarrow E+E*E$
$\rightarrow id+id*id = w$

Derivation Tree / parse Tree:



Pictorial Representation of parsetree (or) derivation

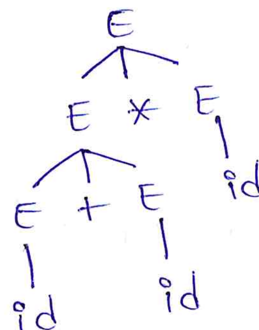
Types of derivation
- Left most derivation
- Right most derivation

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

$E \rightarrow E+E$

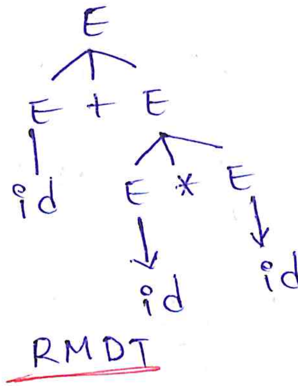
$\rightarrow E+E*E$

$\rightarrow E+E*id$

$\rightarrow E+id*id$

$\rightarrow id+id*id = w$

RMD



Not REC-Lang (TM)
↳ TTM

Classification of CFG:

Recursive Grammar:

① $S \rightarrow Sa|b$

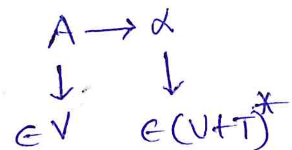
∃ atleast one production having same variable on LHS & RHS

② $S \rightarrow aS|b$

③ $S \rightarrow aSb|\epsilon$

L(RCFG) = Infinite

Non-Recursive Grammar:



① $S \rightarrow AaB$

$A \rightarrow b|c$

$B \rightarrow c|d$

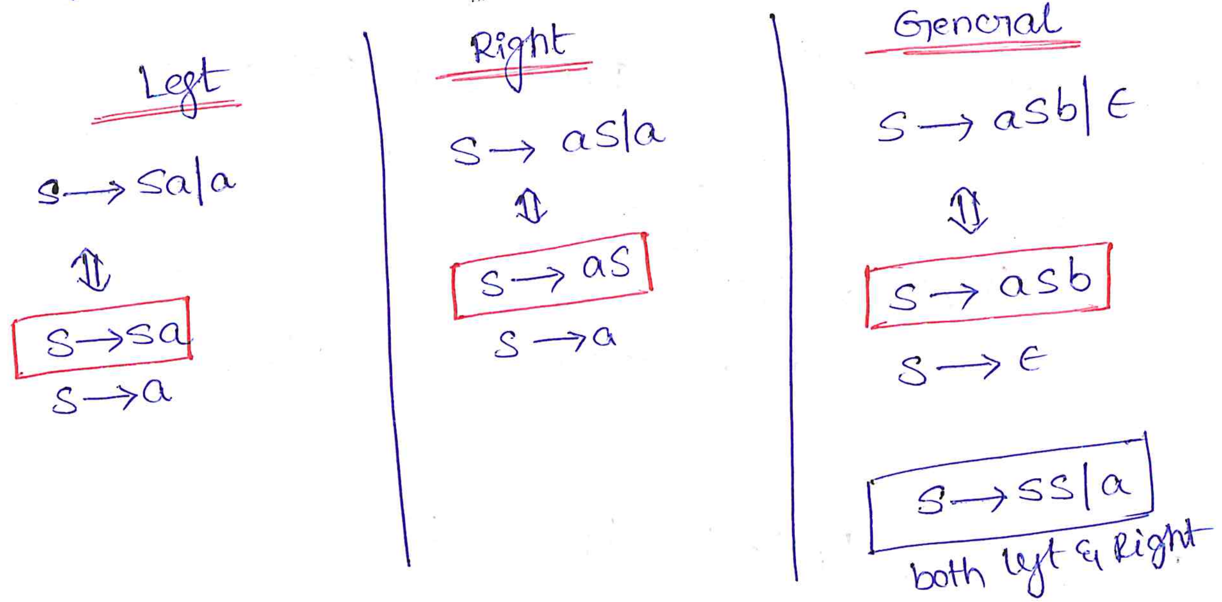
② $S \rightarrow AB|BA$

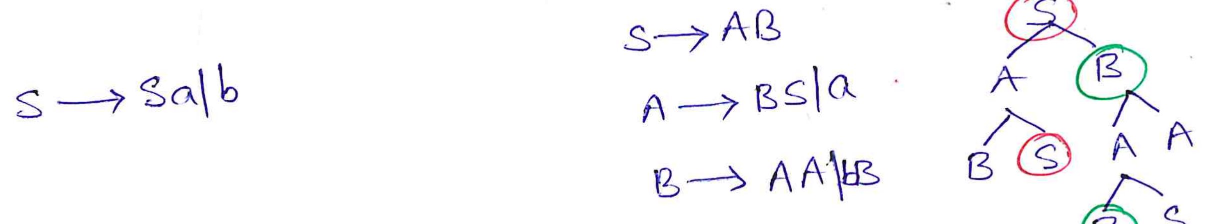
$A \rightarrow aB|b$

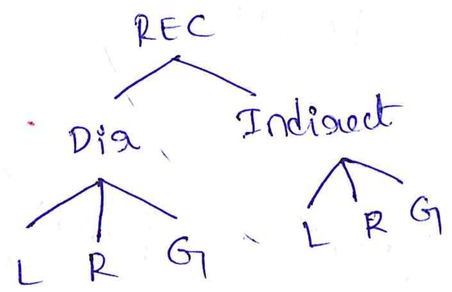
$B \rightarrow b|c$

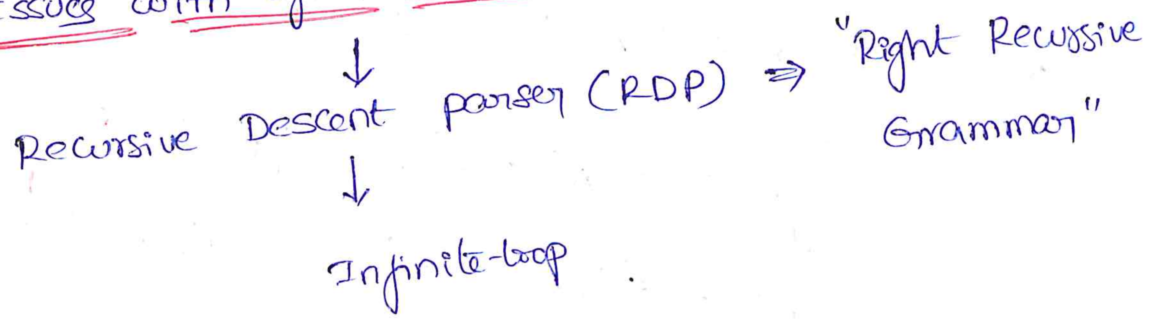
L(NRCFG) = Finite

Types of Recursion:









Conversion from left-Recursion to Right Recursion

① $A \rightarrow A\alpha \mid \beta$ $\qquad$ $\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha \ldots$

$$A \rightarrow \beta\alpha^*$$

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

Right Recursive Grammar

② $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ $\qquad$ LRG

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

③ $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$

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

④ $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

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

Eg ①:

$A \rightarrow \underset{\alpha}{Aab} \mid \underset{\beta}{c}$

$A \rightarrow cA'$
$A' \rightarrow abA' \mid \epsilon$

RRG

②

$A \rightarrow AaB \mid b$

$B \rightarrow Ba \mid c$

$\Rightarrow$
$A \rightarrow bA'$
$A' \rightarrow \epsilon \mid aBA'$
$B \rightarrow cB'$
$B' \rightarrow \epsilon \mid aB'$

RRG

③

$A \rightarrow \underset{\alpha}{A(A)} \mid \underset{\beta}{0}$

$\Sigma = \{ (,), 0 \}$

$\Rightarrow$
$A \rightarrow 0A'$
$A' \rightarrow (A)A' \mid \epsilon$

④

$A \rightarrow AA \mid 0$

$\Rightarrow$
$A \rightarrow 0A'$
$A' \rightarrow AA' \mid \epsilon$

⑤

$A \rightarrow \underset{\alpha}{A(A)A} \mid \underset{\beta_1}{0} \mid \underset{\beta_2}{1}$

$A \rightarrow 0A' \mid 1A'$
$A' \rightarrow (A)AA' \mid \epsilon$

} RRG

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

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

Eg 7: $S \rightarrow \underset{\alpha}{S0S1} \mid \underset{\beta}{\epsilon}$

$S \rightarrow S'$
$S' \rightarrow 0S1S' \mid \epsilon$ } 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: $S \rightarrow S0S \mid 1$

$S \rightarrow 1S'$
$S' \rightarrow 0SS' \mid \epsilon$ } RRG

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

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

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

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

Eg 12:

$E \to E+T \mid T$

$T \to id$

$\Rightarrow$

$E \to TE'$

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

$T \to id$

} RRG

Eg 13:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

} RRG

Eg 14:

$S \to (L) \mid a$

$L \to L,S \mid S$

$S \to (L) \mid a$

$L \to SL'$

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

} RRG

Eg 15:

$R \to \underset{\alpha_1}{RR} \mid \underset{\alpha_2}{Ra} \mid aR \mid \underset{\beta}{b}$

RULE 3

$R \to aR$

$R \to bR'$

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

} RRG

Eg 16:

$S \rightarrow AaB$
$A \rightarrow aA \mid Ba$
$B \rightarrow AB \mid b$

Indirect Recursion

$\Longleftrightarrow$

$S \rightarrow AaB$
$A \rightarrow aA \mid ba \mid ABa$
$B \rightarrow AB \mid b$

(or)

$S \rightarrow AaB$
$A \rightarrow aA \mid Ba$
$B \rightarrow aAB \mid BaB \mid b$

$S \rightarrow AaB$
$A \rightarrow aA \mid \underbrace{ba}_{\beta} \mid A\underbrace{Ba}_{\alpha}$
$B \rightarrow AB \mid b$

$\Downarrow$

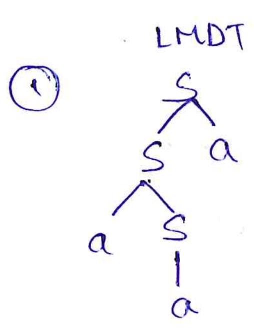
$S \rightarrow AaB$
$A \rightarrow aA$
$A \rightarrow baA'$
$A' \rightarrow \epsilon \mid BaA'$
$B \rightarrow AB \mid b$

Ambiguous and Unambiguous Grammar:

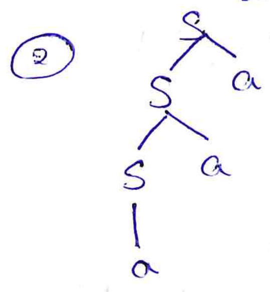
G: $S \rightarrow Sa \mid aS \mid a$

$w = aaa$

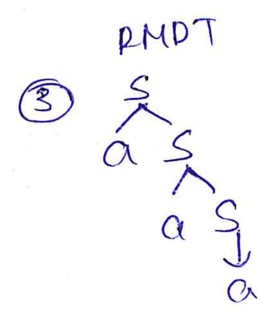
① LMDT



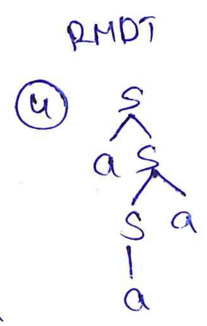
② LMDT



③ RMDT



④ RMDT



Ambiguous:
∃ more than one derivation for any word ∈ L(G)

Unambiguous Grammar:
∃ only one derivation for all words ∀ w ∈ L(G)

Properties:

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

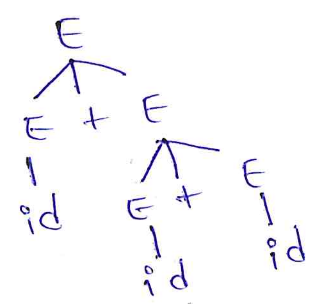
Eliminating Ambiguity:

G_1 { $E \rightarrow E+E \mid id$; w = id + id + id }

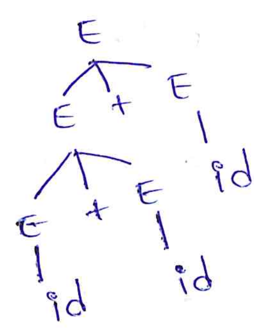
⇒ unambiguous

$E \rightarrow E+T \mid T$
$T \rightarrow id$ } G_2

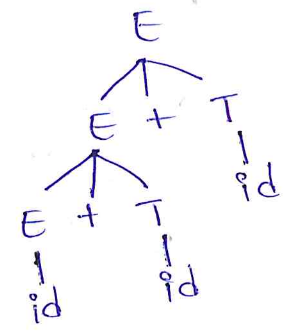
$L(G_1) = L(G_2)$



≠



w = id + id + id



Ambiguous or not :

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

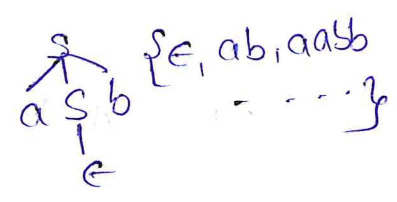
2. $S \rightarrow SaS | SbS | \epsilon$ — Ambiguous

3. $S \rightarrow SSS | 0$ — Ambiguous

4. $S \rightarrow SOS1S | 1$ — Ambiguous

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

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

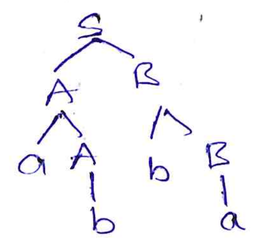


7. $S \rightarrow SAB$
   $A \rightarrow aAB | b$
   $B \rightarrow BS | a$

   $S \rightarrow SABS | SAa$ — 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

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

Same prefix

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

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

$\Rightarrow$

$A \rightarrow \alpha B$

$B \rightarrow \beta_1 | \beta_2 | \beta_3$

① $A \rightarrow aAb | aAd | e$

⇓ LF

$A \rightarrow aAB | e$

$B \rightarrow b | d$

② $A \rightarrow (A)A \mid (A)Ab \mid (A)Aba \mid e \mid f$

⇓

$A \rightarrow (A)A \mid (A)AbB \mid e \mid f$

$B \rightarrow \epsilon \mid a$

⇓

$A \rightarrow (A)AC \mid e \mid f$

$B \rightarrow \epsilon \mid a$

$C \rightarrow \epsilon \mid bB$

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

LF ⇓

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

$B \rightarrow \epsilon \mid e$

⇓

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

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

$B \rightarrow \epsilon \mid e$

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

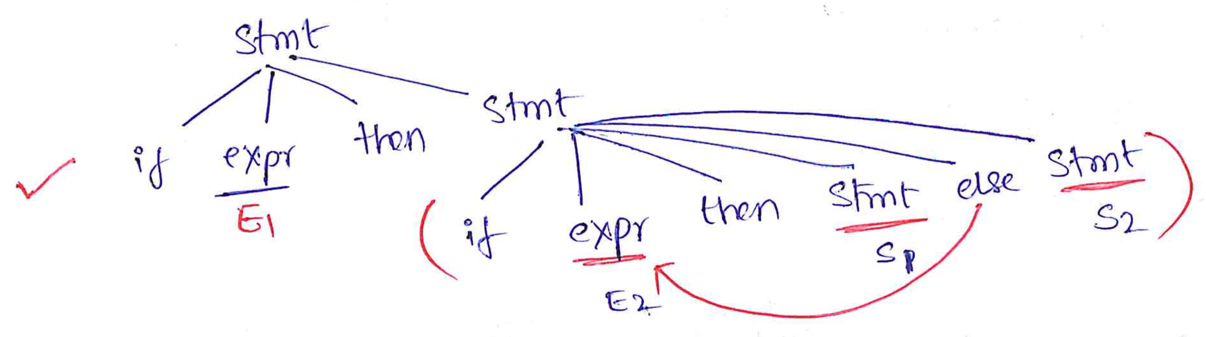
⇓

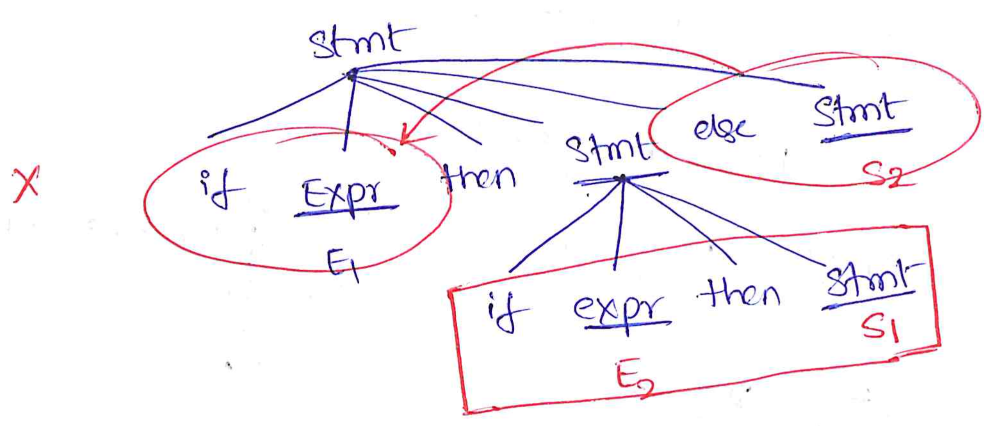
2-parse trees

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 is unambiguous.

Grammar for programming Languages:

① if ... else ... production rules

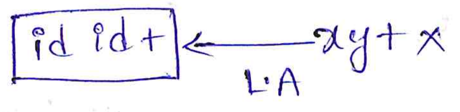
② $E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid id$

id+id x+y x+y+z

x+y*z ↓ lex-Analyser

id+id*id

↓

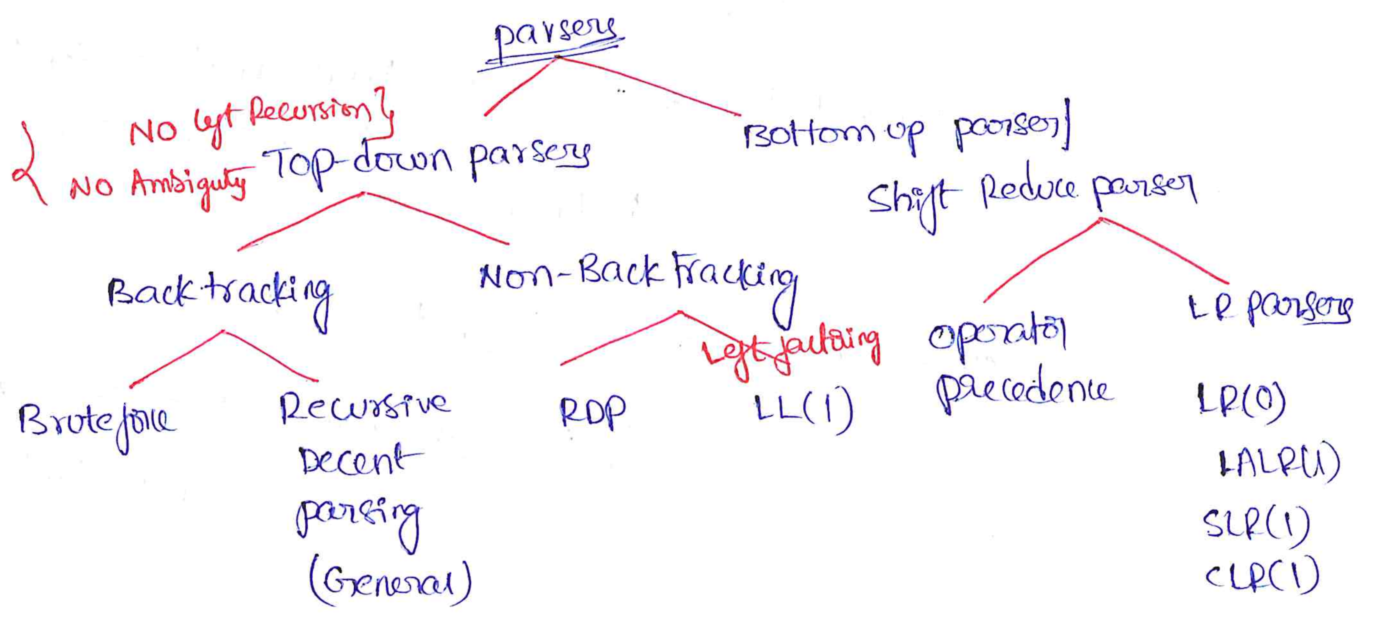


③ C-programming Language

⇓

parser CFG

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



Top-down parser: Free from Left-Recursion

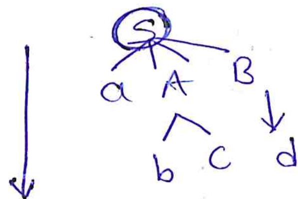
left-Recursive

① $S \rightarrow aAB$

$A \rightarrow bc \mid b$

$B \rightarrow d$

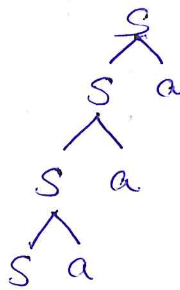
let $w = abcd$



parse-Tree

Left-most derivation Tree

② $S \rightarrow Sa \mid a$



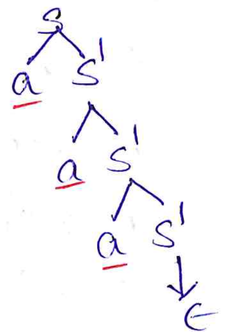
$W = aaa$

Unambiguous

③ $S \rightarrow aS'$

$S' \rightarrow \epsilon \mid aS'$

$w = aaa$



Bottom-up parsing:

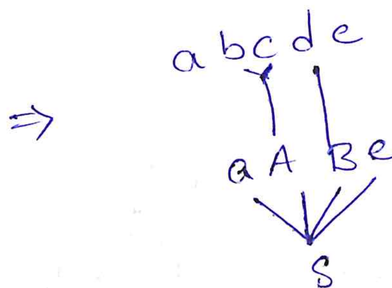
$S \rightarrow aABe$

$A \rightarrow bc$

$B \rightarrow d$

$w = abcde$

⇒



Take the input String and derive the start Symbol of the grammar.

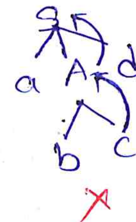
Brute Force: (TDP)

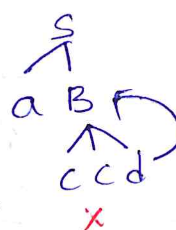
$S \rightarrow aAd \mid aB$

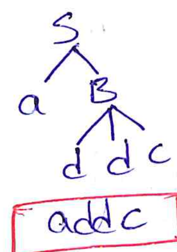
$A \rightarrow b \mid bc$

$B \rightarrow ccd \mid ddc$

$w = addc$







Brute Force ⇒ Try all possibilities

Recursive Descent parser: (General)

⇓

May need back track

Slightly Smarter than Brute Force

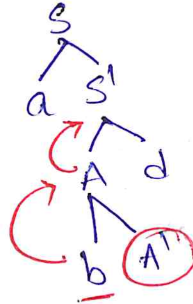
$S \rightarrow aS'$

$S' \rightarrow Ad \mid B$

$A \rightarrow bA'$

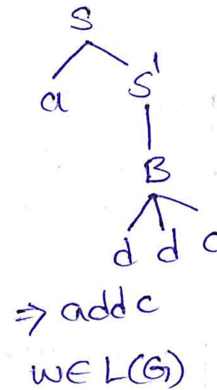
$A' \rightarrow c \mid \epsilon$

$B \rightarrow ccd \mid ddc$



$w = addc$

No Left-Recursion, unambiguous



⇒ addc

$w \in L(G)$

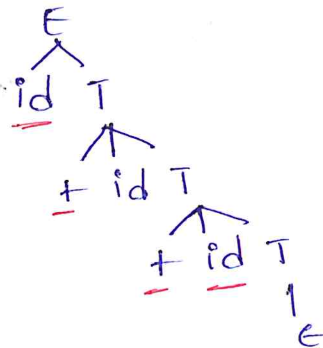
Recursive Descent parser (program)

$E \rightarrow id\,T$

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

$w = id+id+id$

(No backtracking here)



Left-factored Grammar + RDP:

$S \rightarrow aAd \mid aB$

$A \rightarrow b \mid bc$

$B \rightarrow ccd \mid ddc$

$w = addc$

$S \rightarrow aS'$

$S' \rightarrow Ad \mid B$

$A \rightarrow bA'$

$A' \rightarrow \epsilon \mid c$

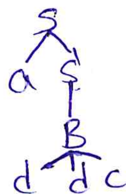
$B \rightarrow ccd \mid ddc$

⇓

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 (α), s.t

$X \to \alpha\beta$ → string of terminals & Variables
(α ↓ terminal)

① $S \to a|b|\epsilon$

First(S) = {a, b, ϵ}

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

③ $S \to aA|bB$ First(S) = {a, b}
$A \to AB|c$ First(A) = {c}
$B \to cB|d$ First(B) = {c, d}

④ $S \to AB$ First(S) = First(A)
$A \to a|b$ = {a, b}
$B \to d$ First(B) = {d}

$S \xrightarrow{*} aB$
$S \to AB$
aB bB
$S \xrightarrow{*} bB$

⑤ $S \to AB$ First(S) = {a, c, d}
$A \to a|\epsilon$ First(A) = {a, ϵ}
$B \to c|d$ First(B) = {c, d}

$S \to AB$
$\xrightarrow{*} aB$
$\xrightarrow{*} B$
{c, d}

(6)

$S \rightarrow AB$ First(S) = {a, b, ε}

$A \rightarrow aA \mid \epsilon$ First(A) = {a, ε}

$B \rightarrow bB \mid \epsilon$ First(B) = {b, ε}

(7)

$S \rightarrow AaB$ First(S) = {b, a}

$A \rightarrow bA \mid \epsilon$ First(A) = {b, ε}

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

(8)

$S \rightarrow ABb$ First(S) = F(A) = {b, d}

$A \rightarrow Ba \mid \epsilon$ First(A) = {ε, b, d}

$B \rightarrow b \mid d$ First(B) = {b, d}

(9)

$S \rightarrow AaB \mid Ba$ First(S) = {a, b, [illegible]}

$A \rightarrow Ba \mid b$ First(A) = {a, b}

$B \rightarrow aB \mid \epsilon$ First(B) = {a, ε}

(10)

$S \rightarrow AaB \mid \epsilon$

$A \rightarrow BbA \mid a$

$B \rightarrow Db \mid d$

$D \rightarrow e \mid f$

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

(11)

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

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

$L \rightarrow SL'$

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

	First
S	{ (, a }
L	{ (, a }
L'	{ , , ∈ }

(13) $S \rightarrow ABCDE$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c \mid \epsilon$

$D \rightarrow d \mid \epsilon$

$E \rightarrow \epsilon$

	First
S	{ a, b, c, d, ∈ }
A	{ a, ∈ }
B	{ b, ∈ }
C	{ c, ∈ }
D	{ d, ∈ }
E	{ ∈ }

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

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

	First
S	{ a, b }
A	{ ∈ }
B	{ ∈ }

Follow(A): Set of terminals (α) s.t
↓
Variable

$S \xrightarrow{*} x A \alpha y \longrightarrow (V+T)^*$

x → $(V+T)^*$, A → Var, α → Terminal

$\alpha \in Follow(A)$

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

(2) $S \rightarrow x A$
↓
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 $S \rightarrow aAb$ $\Rightarrow Follow(S) = \{\$\}$
$A \rightarrow Aa|c$ $Follow(A) = \{a, b\}$
left-recursive Grammar

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

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

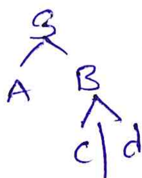
⑤ $S \rightarrow aA \Rightarrow Follow(S) = \{\$\}$
$A \rightarrow b$ $Follow(A) = \{\$\}$

⑥ $S \rightarrow aA$ $\Rightarrow Follow(S) = \{\$\}$
$A \rightarrow aAb|c$ $Follow(A) = \{\$, b\}$

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

⑧ $S \rightarrow AB$ $\Rightarrow Follow(S) = \{\$\}$
$A \rightarrow a|b$ $Follow(A) = \{c, d\}$
$B \rightarrow c|d$ $Follow(B) = \{\$\}$

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



⑨ $S \to AB$

$A \to a|b|\epsilon$

$B \to c|d$

$\Rightarrow$

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

⑩ $S \to AB$

$A \to a|b$

$B \to c|d|\epsilon$

$\Rightarrow$

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

$S \to A\epsilon \to A$

⑪ $S \to S0S1S2|\epsilon$ $\Rightarrow$ 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	{*,+,$,)}

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

⇓

$E' \to +T\epsilon$

$E' \to +T$

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

$\Rightarrow T' \to *F\epsilon$

$T' \to *F$

(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) $S \to ABCDE$

$A \to a \mid \epsilon$

$B \to b \mid \epsilon$

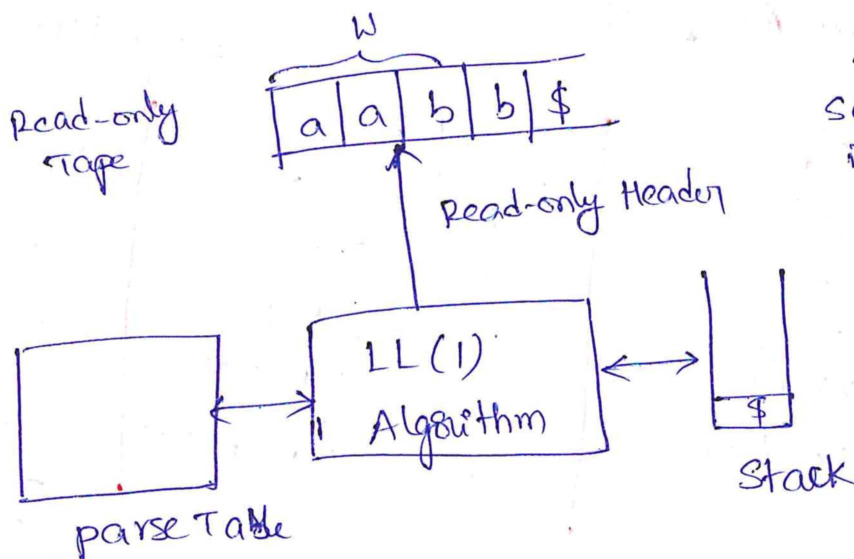
$C \to c \mid \epsilon$

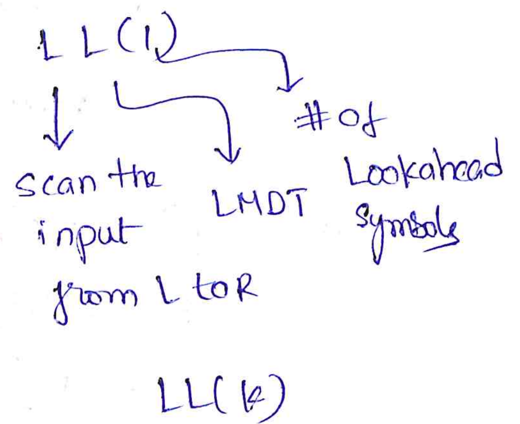
$D \to d \mid \epsilon$

$E \to \epsilon$

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

LL(1) parser:





① Free From Ambiguity

② Free From left-Recursion

③ Free From Common-prefixes.

{RDP: Recursion ⇓ internal Stack}

parse-Table:

① $A \to (A) \mid a$

$A \to \underset{\alpha}{(A)}$ ①

$A \to \underset{\alpha}{a}$ ②

Terminals

T	(	)	a	$
A	①		②	

Var (rows)

Empty cells represent error

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

② If First (α) Contains ϵ, then add $A \to \alpha$ $\forall b \in Follow(A)$

② $S \to AA$ ①

$A \to aA \mid b$ ② ③

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

	a	b	$
S	①	①	
A	②	③	

③ 1. $E \to TE'$

2. $E' \to +TE'$

3. $E' \to \epsilon$

4. $T \to FT'$

5. $T' \to *FT'$

6. $T' \to \epsilon$

7. $F \to (E)$

8. $F \to id$

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

First(E) = { (, id }

First(E') = { +, ε }

First(T) = { (, id }

First(T') = { *, ε }

First(F) = { (, id }

Follow(E) = { $,) }

Follow(E') = { $,) }

Follow(T) = { +, $,) }

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

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

Parsing Algorithm:

G: A → (A) | a
(1) (2)

T	(	)	a	$
A	1		2	

let w = (((a)))

Stack X	input a	Action
$	(((a)))$	PUSH (S)
$A	(((a)))$	PUSH)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
$))) α	α))) $	pop
$)))	))) $	pop
$))	)) $	pop
$)	) $	Accept

Lets Assume X: Stack top a: current input Symbol

① $X = a \neq \$$ pop & increment the input pointer

② $X = a = \$$ Accept

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

$w =)a(\Rightarrow w \notin L(G)$

④ $T(X, a)$ = Empty Syntax Error

Example 2:

① $S \rightarrow AA$

② $A \rightarrow aA$

③ $A \rightarrow b$

	a	b	$
S	1	1	
A	2	3	
$			

Let w = abab

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

Example 3:

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

Stack	Input	Action
$E	id+id*id$	$E \rightarrow TE'$
$E'T	id+id*id$	$T \rightarrow FT'$
$E'T'F	id+id*id$	$F \rightarrow id$
$E'T'id	id+id*id$	pop
$E'T'	+id*id$	$T' \rightarrow \epsilon$
$E'	+id*id$	$E' \rightarrow +TE'$
$E'T+	+id*id$	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' → ϵ
$ E'	$	E' → ϵ
$	$	Accept

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

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

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

② G has ϵ-productions ∀ $A \to \alpha | \epsilon$

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

then G is LL(1)

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

$A \rightarrow \alpha$

$B \rightarrow \gamma$

$C \rightarrow \beta$

⋮

Eg

① $S \rightarrow a|b|c$ LL(1)

② $S \rightarrow aA \mid bB$ ✓ (α_1, α_2)

$A \rightarrow aA \mid b$ ✓

$B \rightarrow bB \mid c$ ✓

LL(1)

③ $S \rightarrow aAb \mid Ba$ (α_1, α_2)

$A \rightarrow aB \mid bB$

$B \rightarrow aB \mid b$

$First(\alpha_1) = \{a\}$

$First(B) = \{a, b\}$

$\{a\} \cap \{a, b\} \neq \phi$

Not LL(1)

④ $S \rightarrow aAb$

$A \rightarrow aA \mid \epsilon$

LL(1)

$First(aA) = \{a\}$ $Follow(A) = \{b\}$

$\{a\} \cap \{b\} = \phi$

⑤ $S \rightarrow aAb \mid \epsilon$

$A \rightarrow aAa \mid \epsilon$

$First(S) = \{a, \epsilon\}$ $Follow(S) = \{\$\}$

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

$\{a\} \cap \{a, b\} \neq \phi$

NOT LL(1)

⑥ $S \rightarrow aAb \mid bB$ ✓

$A \rightarrow aAb \mid \epsilon$ ✓

$B \rightarrow bBb \mid \epsilon$

NOT LL(1)

⑦ $S \rightarrow AS \mid dBA$

$A \rightarrow aAb \mid bB \mid \epsilon$

$B \rightarrow bBa \mid aA \mid \epsilon$

$First(B) = \{a, b\}$ $Follow(B) = \{a\}$

NOT LL(1)

(8)

$S \rightarrow aA \mid bB \mid db$

$A \rightarrow bA \mid Ba \mid \epsilon$

$B \rightarrow aB \mid dA \mid \epsilon$

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

Follow(A) = {$, a}

NOT LL(1)

(9)

$S \rightarrow AB$

$A \rightarrow aB$

$B \rightarrow b$

LL(1)

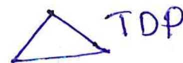
Bottom-up parsers: → widely used

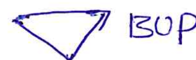
↳ Ambiguous & Unambiguous Grammars

↳ Faster

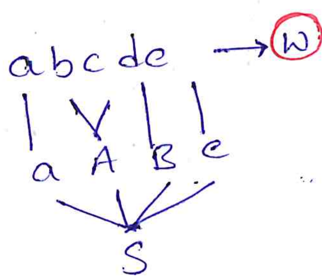
Eg:

G { $S \rightarrow aABe$, $A \rightarrow bc$, $B \rightarrow d$ }

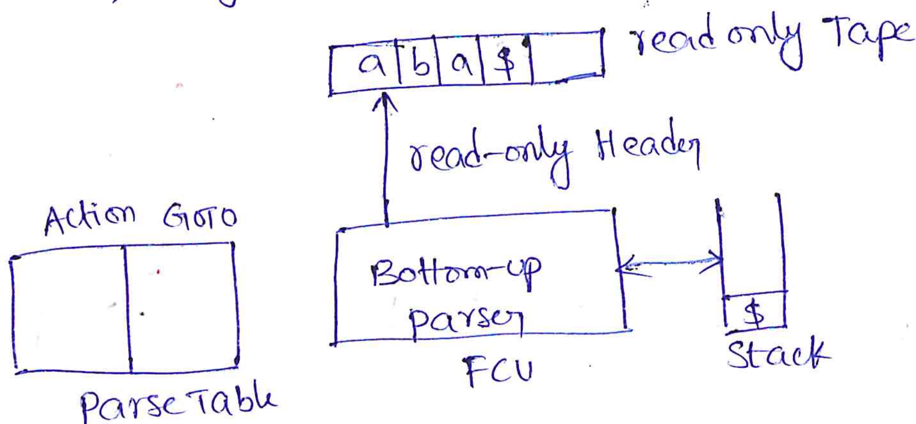




w = abcde



↳ Shift-Reduce parsers

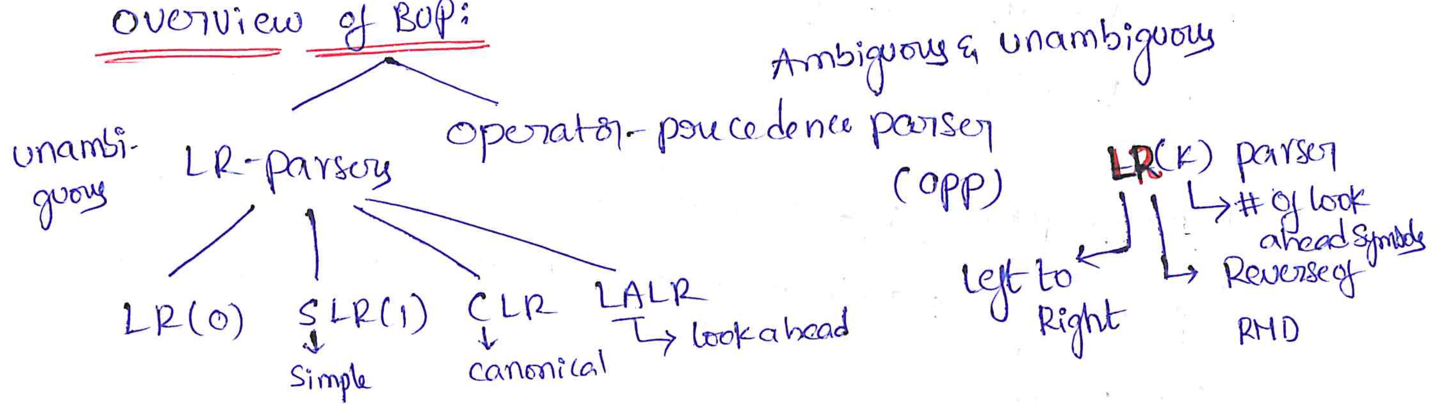


(42)

operations:

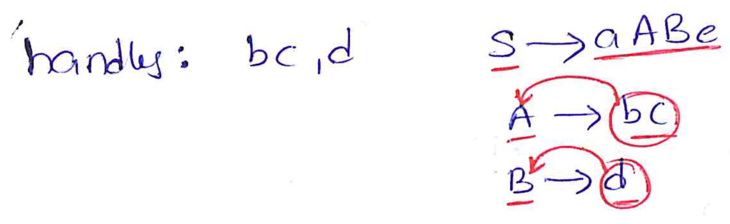
- Shift ✓
- Reduce ✓
- ✓ Accept ⇒ $w \in L(G)$
- ✓ Error ⇒ $w \notin L(G)$

Overview of BUP:

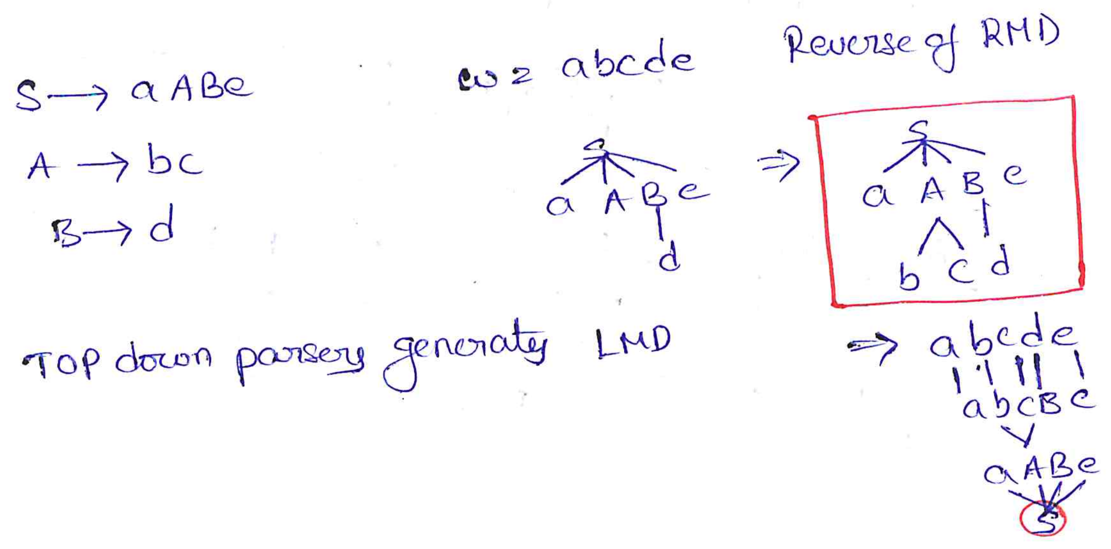


Handle & Handle-pruning:

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



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



Example of Shift & Reduce operations:

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

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

Algorithm determining the action

LR(0): Construct the parse table:

① Construct the Augmented Grammar

G: $A \rightarrow aA$
$A \rightarrow b$

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

② Construct canonical collection of LR-items

LR-item: Item of the Compiler

$A \rightarrow \cdot abc$

$A \rightarrow a \cdot bc$

$A \rightarrow \cdot aA$

$A \rightarrow a \cdot A$

Final-items:

$A \rightarrow aA\cdot$

$A \rightarrow ab\cdot$

$A \rightarrow b\cdot$

Canonical Collection:

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

↑ Set of items

closure (I):

G': $A' \rightarrow \cdot A$, $A \rightarrow \cdot aA$, $A \rightarrow \cdot b$ } items

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

$\{ A' \rightarrow \cdot A$, $A \rightarrow \cdot aA$, $A \rightarrow \cdot b \}$ — I_0

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

$B \rightarrow \cdot \gamma$

⇓

$A \rightarrow \alpha \cdot B\beta$

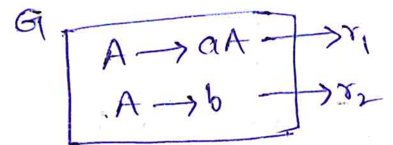
$B \rightarrow \cdot \gamma$

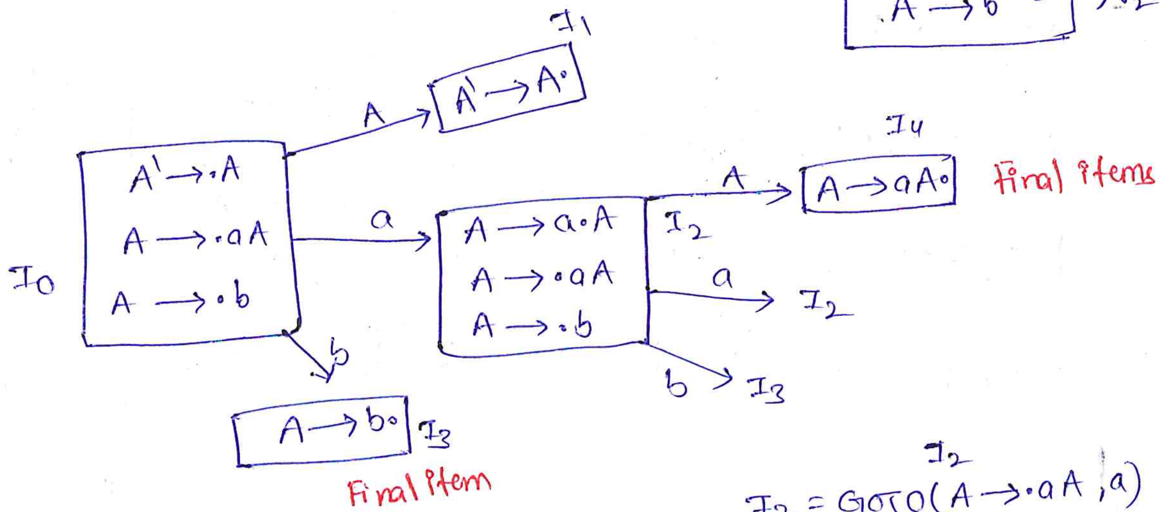
Eg 1: Closure $(A \rightarrow \cdot aA) = \{ A \rightarrow \cdot aA \}$

② **GOTO(I, X):** $\text{Goto}(A \rightarrow \alpha \cdot X\beta, X) = \text{Closure}(A \rightarrow \alpha X \cdot \beta)$

(item, symbol, item)

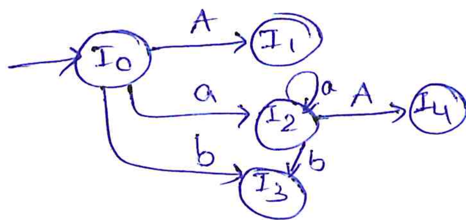
③ Construct the graph / DFA:





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

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



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

$= closure(A \to b\cdot)$

④ Construct parse-table:

Terminals → ; Variable ←

	Action a	Action b	Action $	GOTO A
I_0	S2	S3		1
I_1			Accept	
I_2	S2	S3		4
I_3	r2	r2	r2	
I_4	r1	r1	r1	

S : shift

r : Reduce

LR(0) parse table

⑤ Using the table to parse the input aab

Stack	input	Action
$0	aab$	S2
$0 a 2	ab$	S2
$ 0 a 2 a 2	b$	S3

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

w∈L(G)

LR(0) parser: More Examples:

w = abb.

G: S→AA r_1

A→aA r_2

A→b r_3

① Augmented Grammar

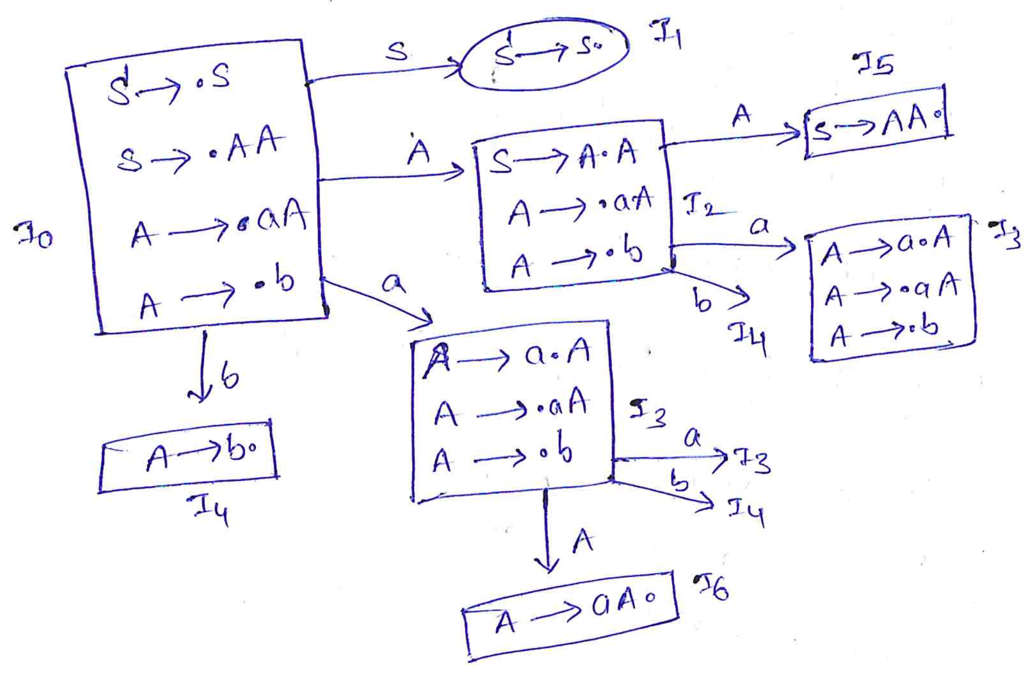
S'→S

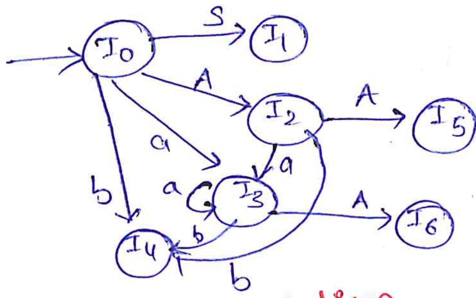
S→AA

A→aA

A→b

② Find canonical collection of items





	Action a	b	$	Goto S	A
I0	S3	S4		1	2
I1			Accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

Stack	input	Action
$0	abb$	S3
$0a3	bb$	S4
$0a3~~b4~~	b$	r3 A→b
$0a3A	b$	6
$0~~a3A6~~	b$	r2 A→aA
$0A	b$	2
$0A2	b$	S4
$0A2~~b4~~	$	r3 A→b
$0A2A5	$	r1 S→AA
$0S	$	1
$0S1	$	Accept

abb

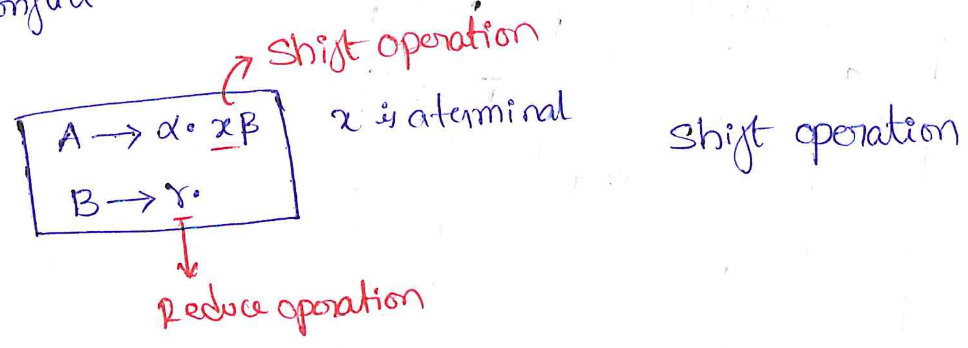
w∈L(G)

Conflicts: LR(0)

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

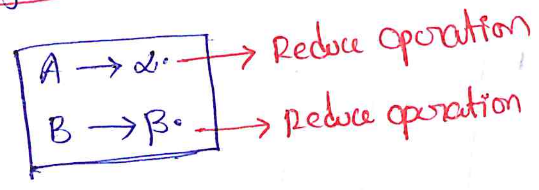
① SR Conflict

② RR Conflict



Shift operation

RR Conflict:

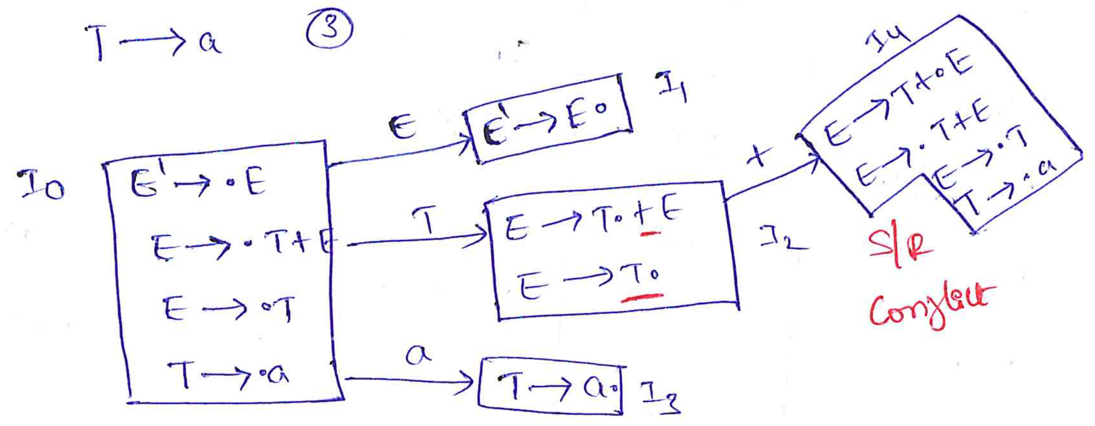


Eg 1:

E → T+E ①

E → T ②

T → a ③



GOTO(I4, E)

E → T+E· I5

GOTO (I4, T)

I2

GOTO (I4, a)

I3

	+	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 (Minor)

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) = {$}

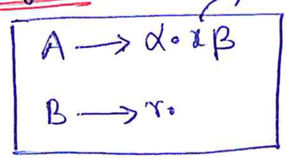
$T \rightarrow a.$
Follow(T) = {$, +}

$E \rightarrow T+E.$ 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)

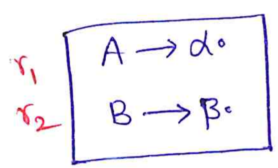
First(x) ∩ Follow(B)

⇓

x ∩ Follow(B) ≠ ϕ

S/R Conflict in SLR(1)

RR Conflict:



Follow(A) ∩ Follow(B) ≠ ϕ

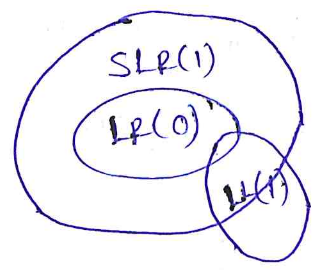
RR Conflict

SLR(1) Grammar:

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

↓

Fewer entries than LR(0)



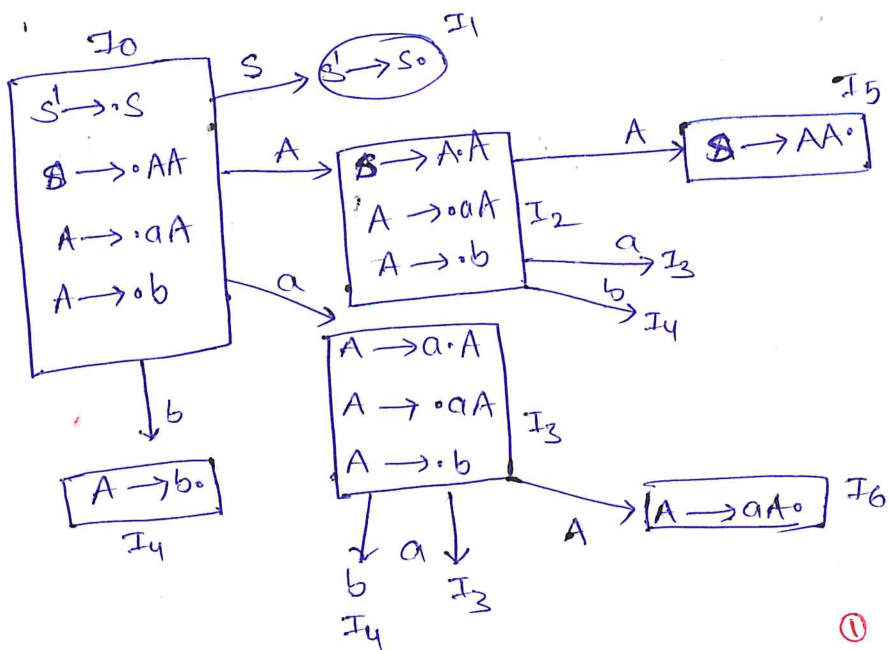
SLR(1) is more powerful than LR(0) parser.

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

G:

$S' \rightarrow \cdot S$

1. $S \rightarrow \cdot AA$
2. $A \rightarrow \cdot aA$
3. $A \rightarrow \cdot b$



	a	b	$	S	A
I0	s3	s4		1	2
I1			Accept		
I2	s3	s4			5
I3	s3	s4			6
I4	r3	r3	r3		
I5			r1		
I6	r2	r2	r2		

1. S → AA
2. A → aA
3. A → b

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 ⇒ Augmented Grammar

A → aA

A → b

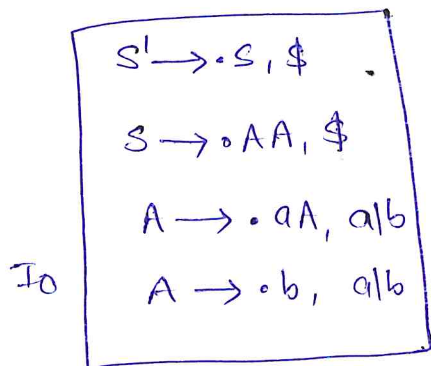
S' → S

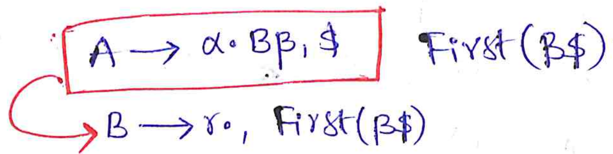
S → AA

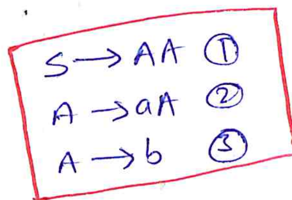
A → aA

A → b

I_0

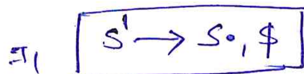






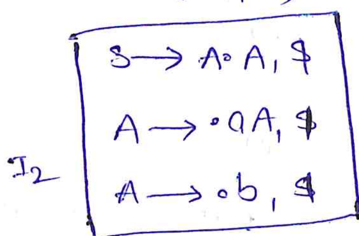
GOTO (I_0, S)

I_1



GOTO (I_0, A)

I_2

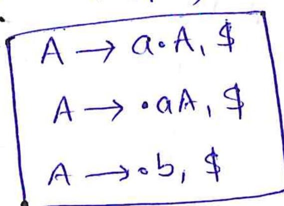


⇒

GOTO (I_2, A)

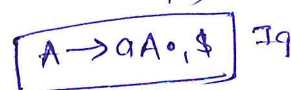
I_5

GOTO (I_2, a)



I_6

GOTO (I_6, A)



I_9

GOTO (I_6, a)

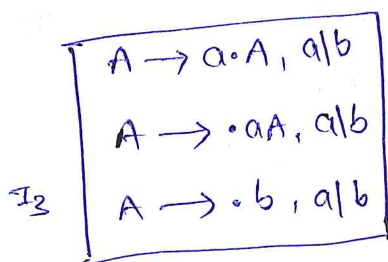
I_6

GOTO (I_6, b)

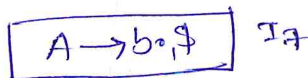
I_7

GOTO (I_0, a)

I_3



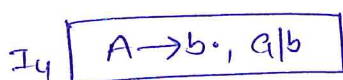
GOTO (I_2, b)



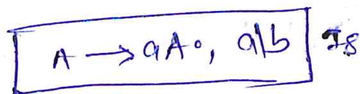
I_7

GOTO (I_0, b)

I_4



GOTO (I_3, A)

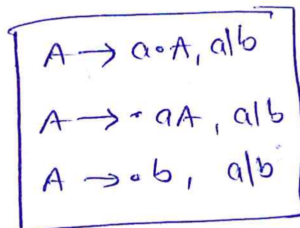


I_8

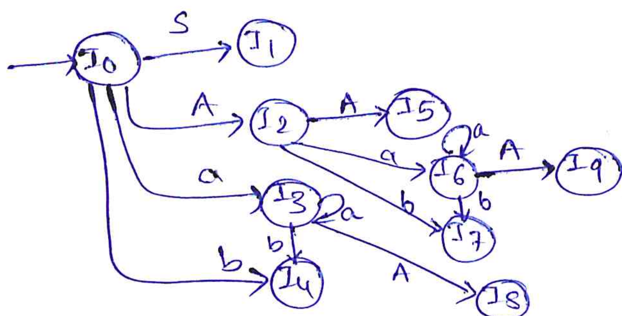
GOTO (I_3, b)

I_4

GOTO (I_3, a)



I_3.

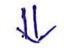


	a	b	$	S	A
I_0	S3	S4		1	2
I_1			Accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	r3	r3			
I_5			r1		
I_6	S6	S7			9
I_7			r3		
I_8	r2	r2			
I_9			r2		

Number of States 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:

3 & 6, 4 & 7, 8 & 9

Merge the States

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

	a	b	$	S	A
I_0	S36	S47		1	2
I_1			Accept		
I_2	S36	S47			5
I_{36}	S36	S47			89
I_{47}	r3	r3	r3		
I_5			r1		
I_{89}	r2	r2	r2		

No. of States in LR(0)

= SLR(1) = LALR(1)

≤ CLR(1)

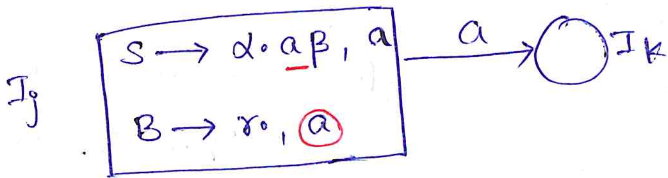
CLR(1) and LALR(1) Grammars and Examples:

CLR(1) → LR(1)

SLR(1), LR(0)

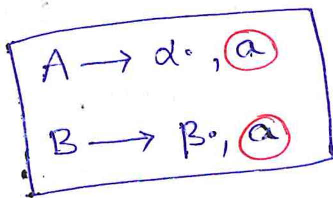
Conflicts : S|R & R|R

I_j



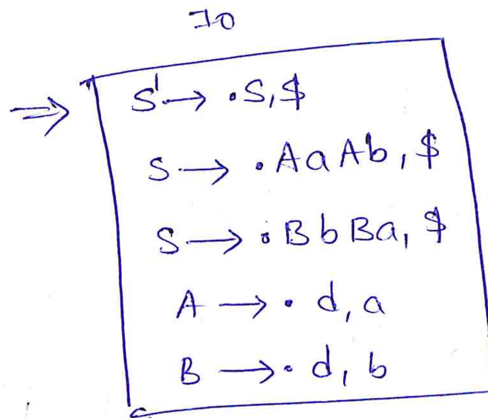
S|R Conflict in CLR(1)

	a
I_j	S_k/r_1
I_3	r_1/r_2

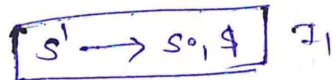


R|R Conflict in CLR(1)

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

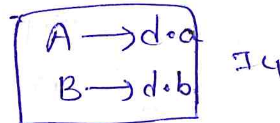


GOTO (I0, S)



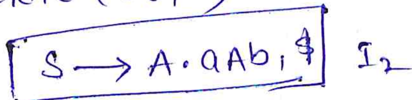
I_1

GOTO (I0, d)



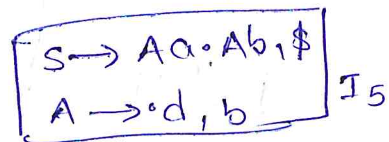
I_4

GOTO (I0, A)



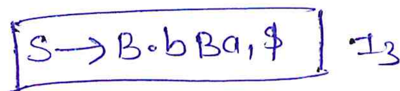
I_2

GOTO (I2, a)



I_5

GOTO (I0, B)



I_3

GOTO(I_3, b)

I_6
S → Bb·Ba, $
B → ·d, a

GOTO(I_5, A)

I_7
S → AaA·b, $

GOTO(I_7, b)

I_{11}
S → AaAb·, $

GOTO (I_5, d)

I_8
(3) A → d·, b

GOTO (I_9, a)

I_{12}
S → BbBa·, $

GOTO(I_6, B)

I_9
S → BbB·a, $

GOTO(I_6, d)

I_{10}
(4) B → d·, a

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

	a	b	d	$	S	A	B
I_0		S4			1	2	3
I_1				Accept			
I_2	S5						
I_3		S6					
I_4	r3	r4					
I_5			S8				7
I_6		~~S10~~	S10				9
I_7		S11					
I_8		r3					
I_9	S12						
I_{10}	r4						
I_{11}				r1			
I_{12}				r2			

No conflicts

CLR(1) Grammar

LALR(1) Grammar & Conflicts:

⇓

Combining states in CLR(1) parse table.

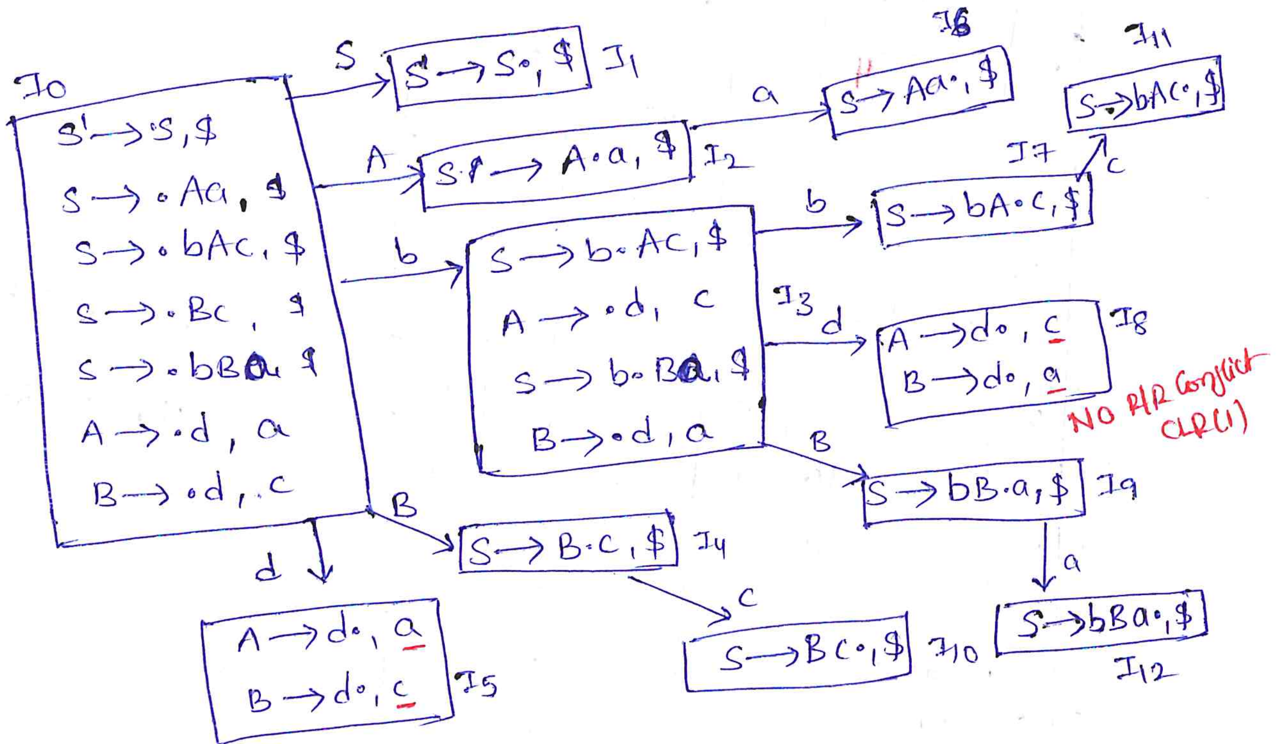
→ If CLR(1) parse table has no SR Conflicts then no SR conflicts in LALR(1).

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

Eg 1:

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

CLR(1) but not LALR(1)



No Conflicts in CLR(1)

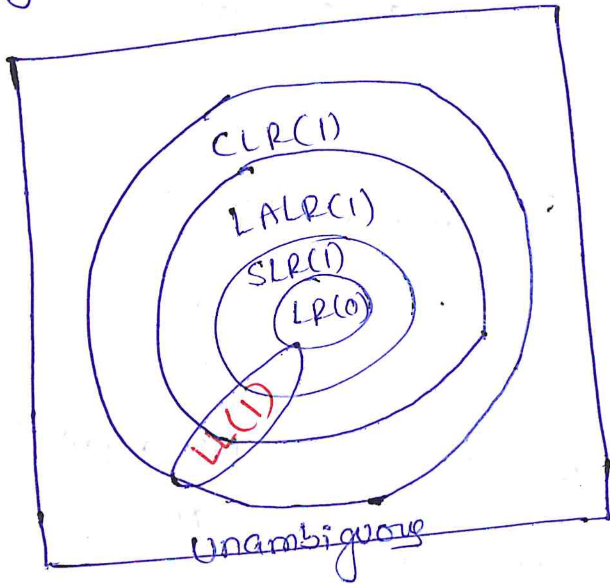
I_{58}

⑤ A → d· a/c
⑥ B → d· a/c

R/R Conflict in LALR(1)

	a	c
I_{58}	r_5/r_6	r_5/r_6

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:

↳ BUP, ambiguous & unambiguous

↳ cannot work for all CFG

→ +, −, *, /, %, - - - -

Operator Grammar:

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

AaB

all terminals
↓
operators

(58)

Eg: ① $S \to AB$
$A \to a$
$B \to b$

NOT OPG
operator Grammar

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

No ε-prod
No Adjacent variables

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

Not an operator Grammar

④ $E \to EAE | id$
$A \to + | - | *$

} Adjacent variables
→ Not OPG

⇓

$E \to E+E | E-E | E*E | id$ — operator Grammar

No adjacent variables
No ε-productions

↙ Ambiguous Grammar

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

Ambiguous opr. Gr

w = id + id * id $

(E above each id)

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

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

n×n

precedence Table

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

2+3*5 = 2+15 = 17

2+3+4 ⇒ 5+4 ⇒ 9

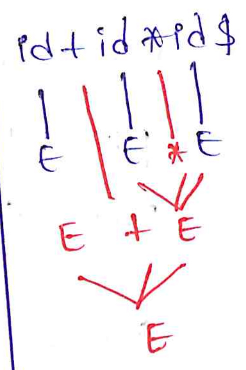
3+2*5

↓ LA

id + id * id $

operator precedence Grammar

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

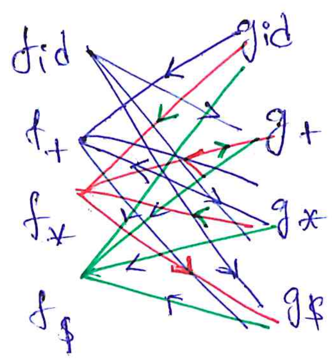


O(n²) to O(2n) table:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Rel Table

① Functional Table   No-cycle



② fid → g* → f+ → g+ → f$

gid → f* → g* → f+ → g+ → f$

Compute Largest path

③

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

Functional Table

$O(2n)$ Compare (+,*) Compare (id,id)

f_{id} g_+ f_+ g_* f_{id} > g_{id}

4 > 1 2 <· 3 Error

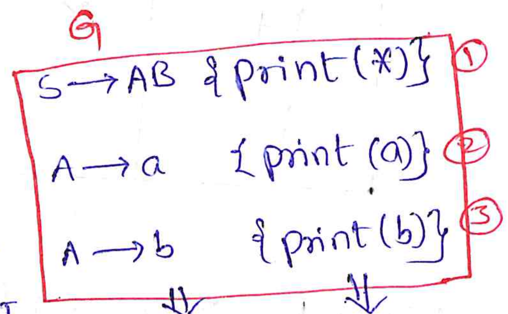
Compare (id,+)

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
↓ PT
Semantic Analysis
↓
ICG
↓
code optimization
↓
Code Generation.

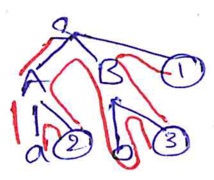


CFG + Actions = SDT

CFG + Semantic Actions | Rules

Type checking

let w = ab



parse tree

TDP Topdown
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 → E+E { E.val = E.val + E.val } (1)

E → E*E { E.val = E.val * E.val } (2)

E → id { E.val = id.val } (3)

CFG — action

w = 2+3*5

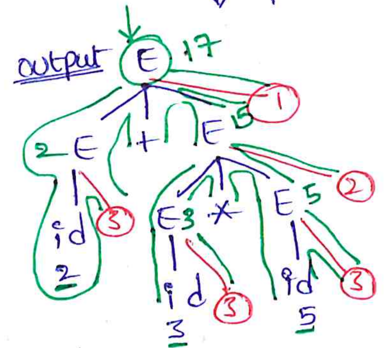
↓ (Symbol Table)

lex. Analysis

↓

w = id+id*id

↓ parser (TDP)



Top-down

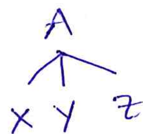
L → R

SDT:

E.val
id.val } Evaluate algebraic Expression

Types of attributes : Synthesized & Inherited

(1) A → XYZ



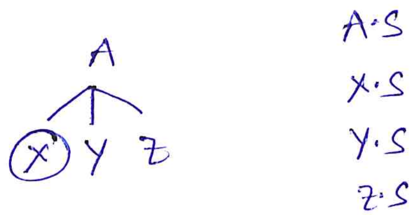
S is an attribute corresponding to A children only

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

Attribute S is known as Synthesized attribute.

Inherited Attribute:

$A \rightarrow XYZ$



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

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

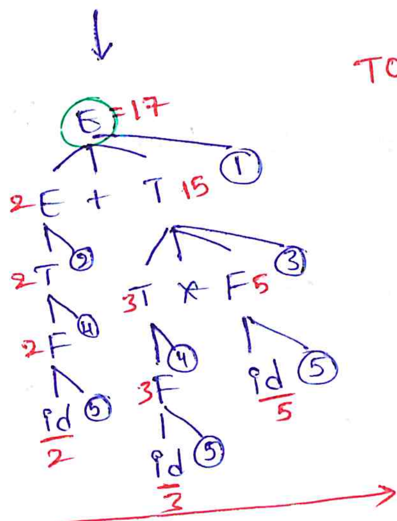
Actions

W = 2+3*5

↓ LA + S.T

id + id * id

↓



Top down

L → R

input = 2+3*5

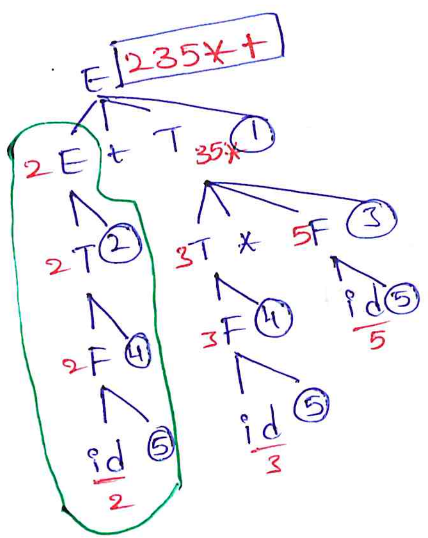
output = 17

② SDT for infix to postfix Conversion:

input : 2+3*5
output: 235*

E → E+T	{ E.val = E.val T.val + }
E → T	{ E.val = T.val }
T → T*F	{ T.val = T.val F.val * }
T → F	{ T.val = F.val }
F → 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 → E+T	{ if (E.type == T.type) && (E.type == int) then E.type = int else Error }
E → T	{ E.type = T.type }
~~T → F~~	
T → T*F	{ if (T.type == F.type) && (F.type == int) then T.type = int else Error }
T → F	{ T.type = F.type }
F → id	{ F.type = id.type }

$W_1 = 2+3*5$ ✓

$W_2 = 2+3*a$ ✓

$W_3 = 2+3.2*a$ — ERROR

(2, 3.2, a → id id id)

④ SDT to Count parenthesis:

G:

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

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

CFG — Actions

W = ((())) input

3 output

A = ③
(A2) ①
(A1) ①
(0A) ①
ε ②

Topdown
L→R

⑤ SDT for 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 }

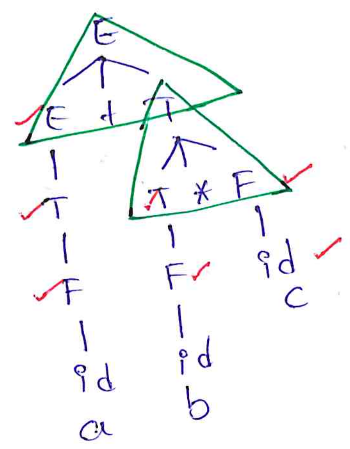
$F \rightarrow id$ { F.ptr = mknode(NULL, id, NULL) }

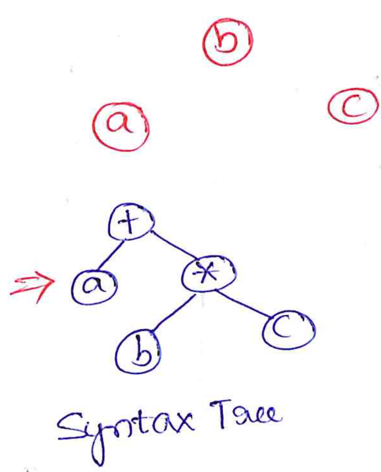
input: a+b*c

output: + (a, *(b, c))

parse-tree : Terminals & non-terminals

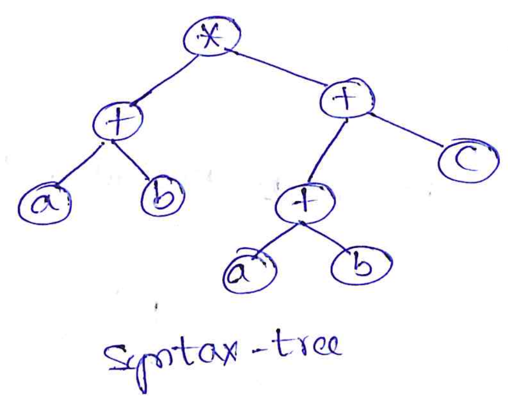
In a Syntax-tree all ~~trees~~ nodes are terminals.



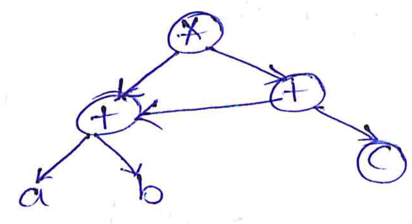


Syntax Tree

⑥ SDT for expression-eval DAGs:



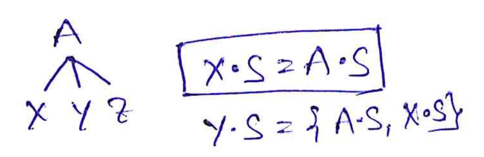
Syntax-tree



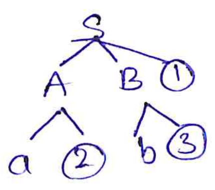
Avoids duplicate Computation

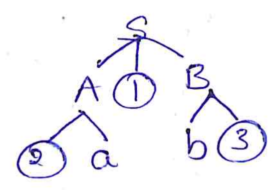
Types of SDT:

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.

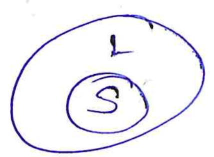


S-attributed	L-attributed
② actions are placed @ the end of RHS	② Actions can be placed any where on RHS.
S → AB {print x} ① A → a {print a} ② B → b {print b} ③	S → A ① B A → ② a B → b ③ w = ab
③ Attributes can be evaluated using Topdown or bottomup methods	③ Depth First L → R





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



L-attributed SDT:

S → A ① B

A → ② a

B → b ③.

S-attributed SDT

S → ARB { }
R → ∈ ①

A → X a { }
X → ∈ ②

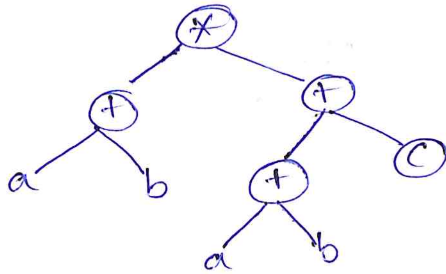
B → b ③

ICG + 3-Address codes: Platform Independant

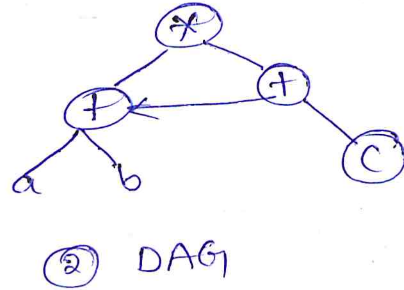
Improve code-generation efficiency.

(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 Triples

①	b	c	*
②	a	①	+
③	x	②	=

No Temp var

③ Indirect Triples

(101)	b	c	*
(102)	a	(101)	+
(103)	x	(102)	=

pointers

reuse Computation

Types of 3-addr codes:

① $x = y \text{ op } z$ ✓

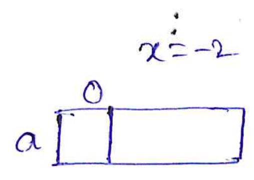
② $x = \text{op } y$ ✓

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 $y = f(x_1, x_2 \ldots x_n)$ ✗
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 (L6)

⑤ L2: print("TOC")

⑥ END/EXIT

④

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

1. i=1
2. if (i>10) goto 9 : Conditional Jump
3. x=2*i
4. Goto L : Unconditional
5. L: print(x)
6. t=i+1
7. i=t
8. goto (2)
9. END/EXIT