COMPILER DESIGN.

1 to 22: Into & Lexical Analysis
23 to 51 : Parsing

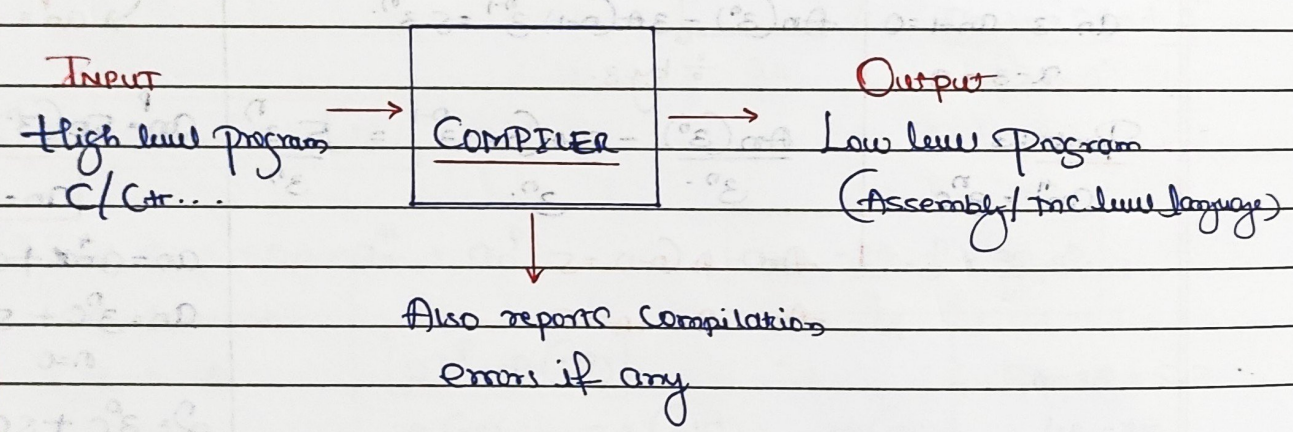
1. Phases of a Compiler
2. Lexical Analysis
3. Syntax Analysis
4. Syntax Directed Translations
5. Intermediate Code Generation
6. Code optimizations.
7. Runtime Environment.

Introduction

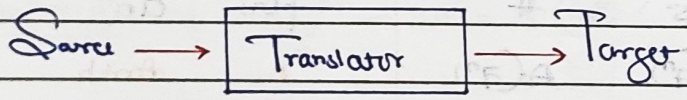
What is Compiler? → Translator

What is use or where we use Compiler? → language translation

How Compiler is designed? → 7 phases

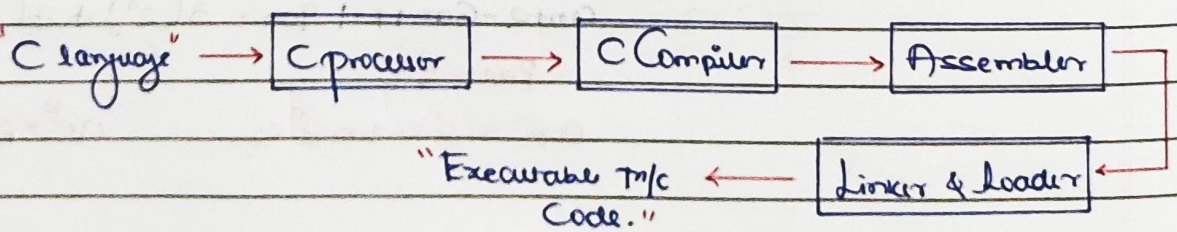


Translator:



Examples: Compiler, Interpreter, Editor, Pre-processor, word, linker..etc

Language Translation : (C language to executable code)



LEXICAL ANALYSIS & SYNTAX ANALYSIS

Pre Processor

constant expressions

```
#define MAX 10;
int x = MAX;
void main()
{ int y = MAX; }
```

→ [Preprocessor] →

```
int x=10;
void main()
{ int y=10; }
```

#define Constant Exp.
→ Replace Constant with expression.

Assembler:

```
MUL R1,R2
R1←R1*R2
```
Assembly Code → [Assembler] → Machine Code. 0010110 0110100 011100

Linker	Loader
It resolves all external refrances.	It performancis relocations.

```
extern int x;
void main()
{
printf("%d",x);}
```

xxx (address of code block) | x=x+10; (address of variable) } Codeblock.

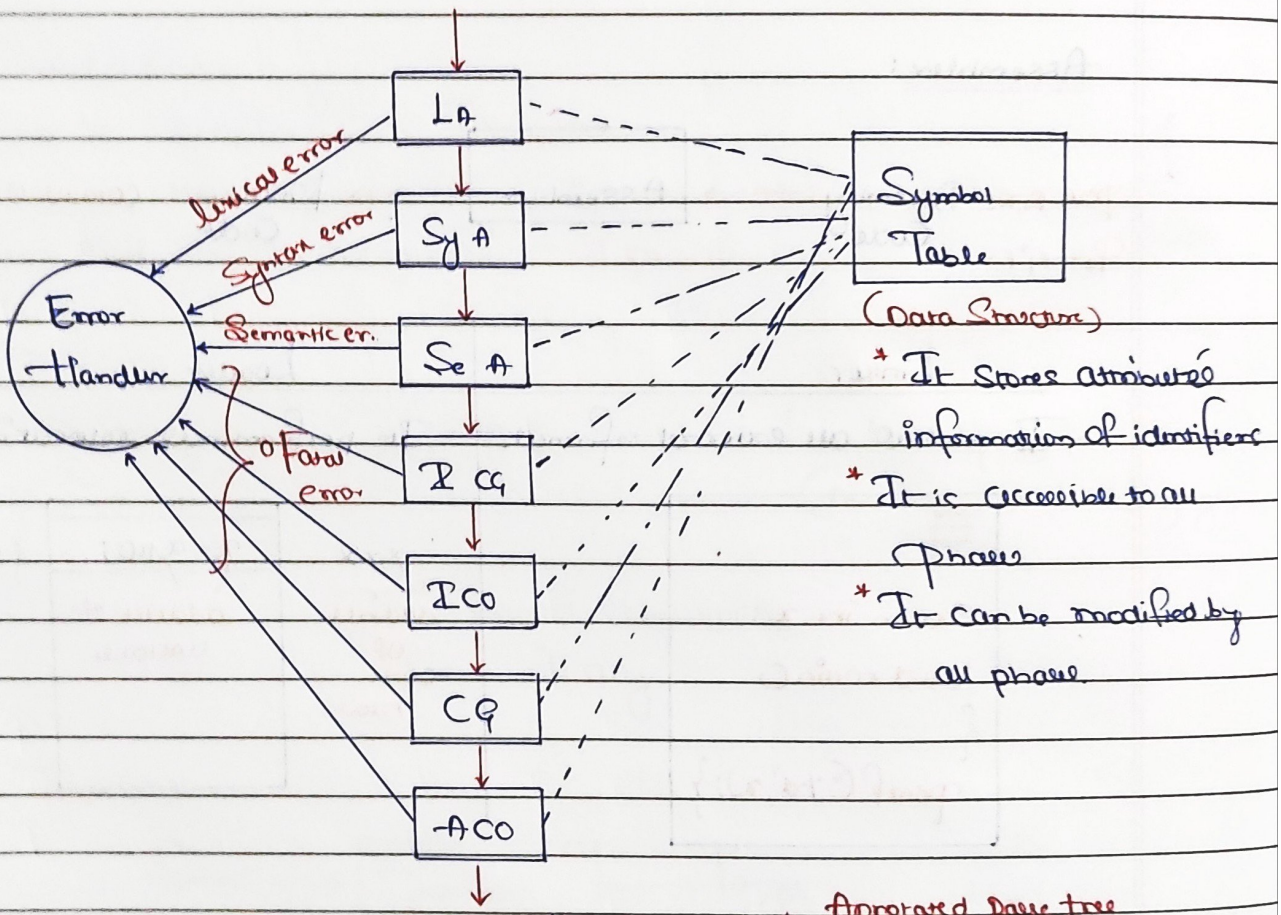
Loaders:

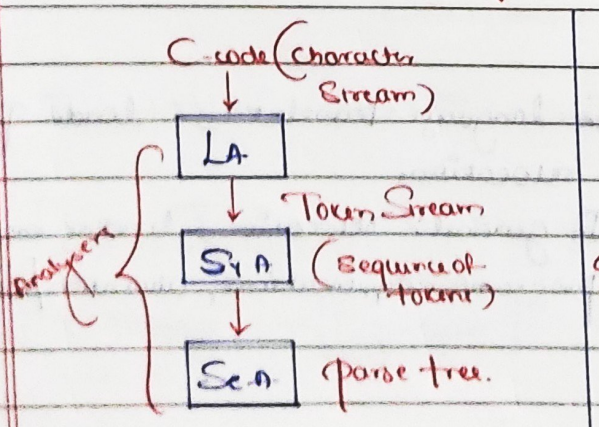
- Bootstrap loader
- Swapper
- Other loaders

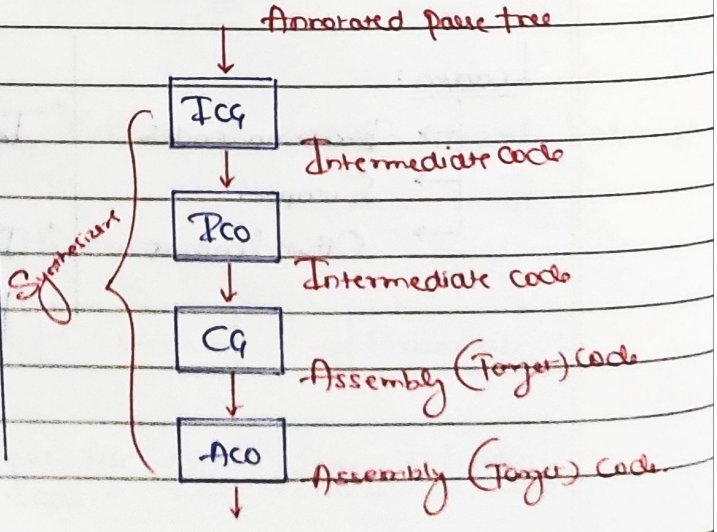
* In language translation: loader performs relocation.
* In general: relocation, loading modules, programs, applications, allocation/dellocation.

Phases of Compiler:

- 7 phases.
  1. Lexical analysis
  2. Syntax analysis
  3. Semantic analysis
  4. Intermediate Code generator
  5. Intermediate Code Optimization
  6. Code generation.
  7. Assembly (Target) code Optimization.







classmate
Date 29/8/23
Page

M/C Independent
- C Code } Language dependent
- I Code

M/C Independent
- A Code
- M Code
- E Code

} Language Independent

Code → Handle your data [I/p or O/p]
Data → Requirement

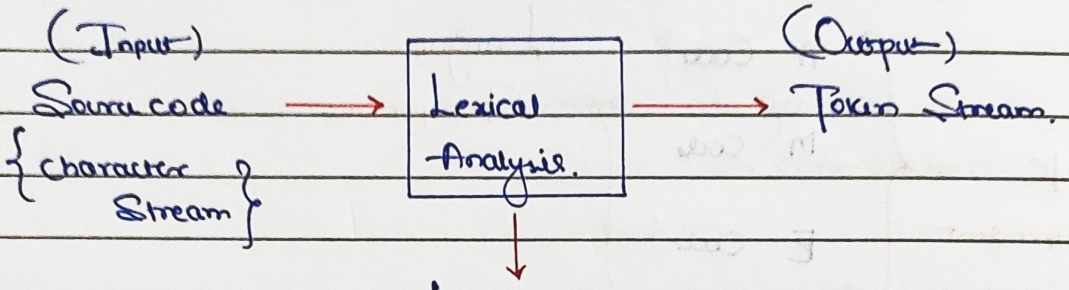
	C Language	C Compiler
1.	High level code	Machine code (Binary, Executable, low-level)
2	Machine independent code.	Machine dependent code
3.	Portable. → code can run on any machine	Not Portable. → code cannot run on many machine → runs only on specific m/c.

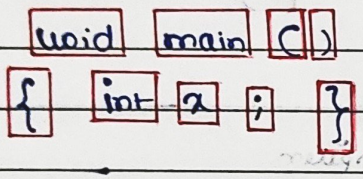
	Phase	Input	Output	Functionality.	
1.	LA [A, ii, a]	A Character Stream	i. Character Stream	a.	Token Recognizer
2.	SyA [B, iii, b]	B Token Stream	ii. Token Stream	b.	Syntax verifier
3.	SeA [C, iv, c]	C Parse tree	iii. Parse tree	c.	Type Checker
4.	ICG [D, v, d]	D Annotated P.T	iv. Annotated P.T.	d.	Three address code Generator
5.	ICO [E, v, e]	E Intermediate code	v. I.C	e.	Assembly code generator
6.	CG [F, vi, e]	F Assembly code	vi. A.C	f.	Machine code generator
7.	ACO [F, vii, f]	G Machine Code	vii. M.C	g.	English Code generator
		H English	viii. English	h.	Hindi Code generator.
				i.	Intermediate code optimizer
				f.	Assembly code optimization.

Classmate
Date 29/8/23
Page

LEXICAL ANALYSIS & SYNTAX ANALYSIS

Lexical Analysis





Lexical Analysis :

- Lexical phase
- Lexical Analyzer
- First phase of Compiler
- Scanner
- Linker phase
- Token recogniser
- Token Generator

Note:

* Compilation begins from the first character
* Execution begins from main ( ) function

Lexical Analysis Process.

1. It Scans whole Program Character by character
2. It groups Character into tokens
3. It ignores White Spaces and Comments
4. It uses "Largest/longest Prefix Rule" [Maximal Munch]
5. It also produces lexical errors if any
6. It creates a Symbol table and Stores some attribute information of identifiers (variable name, function name... etc)

Symbol Table:

- It is a data structure
- It stores attribute information of identifiers
- It is accessible to all phases of compiler.

eg: void fun (int x)
{ int a; }

Identifiers:

remaining phases

lexeme	Token	line	type	diff..	...
fun	Id_1				
x	Id_2				
a	Id_3				

Internal representation of lexeme.

Lexeme: Smallest unit of (logic) Program.

Token: It uniquely represents lexeme.

Types of tokens:

- Keywords
- Literals/Constants
- Operators
- Special tokens
- Identifiers

Q1.) Is "main" keyword?

→ No, it is not a keyword
→ It is a user defined function
→ It is an identifier

I **Keyword:** ① int, float, double, char, long, short, void, signed, unsigned, volatile, const, auto, static, extern, register.
② if, else, switch, case, default, while, do, for, break, continue, return, goto.
③ Struct, Union, Enum, typedef.
④ Sizeof().

II **Operators:** Unary: !, ~, +, −, ++, --, *, &
Arithmetic: +, −, *, /, %
Relational: <, >, ≤, ≥, ==, !=
Bitwise: &, |, ^, <<, >>
Logical: &&, ||
Assignment: =, +=, −=, /=, %=

classmate
Date 31/8/23
Page

III. Identifiers:
* Name given to variable/functions.
* Comprised of letters, digits, underscore.
* Should not start with digit.
* Should not be a keyword

IV. Constants:
- Integers → Decimal, Octal, Hexadecimal
- Real → float, double
- Character → 'a'
- String → "a"

V. Special Tokens
;, {, }, (,), [,], :

Find the tokens in the following codes

1. `int x;` (int: keyword, x: identifier, ; : spl) ⇒ 3.
2. `int float;` (int: key, float: key, ; : spl) ⇒ 3. No lexical error but Syntax error.
3. `int gate123;` ⇒ 3
4. `int 123gate;` ✓ int, ✗ 123gate: Invalid Sequence. Lexical error.
5. `int x = "gate";` ⇒ 5 ("gate": String constant)
6. `int x = 025;` ⇒ 5 (025: octal)
7. `int x = 029;` Lexical error. (029: octal, 9 ✗)
8. `int x = 0xabc;` ⇒ 5 tokens (0xabc: hexadecimal)
9. `int 23 = 45;` ⇒ 5 tokens. No lexical error, but syntax error.
10. `int /*comment*/ x;` ⇒ 3 (/*: start, */: end, comment: ignore)
11. `int/*comment*/t x;` ⇒ 4 (int: keyword, t: seperated by comment)
12. `x = +y;` ⇒ 5
13. `x = +y;` ⇒ 5
14. `x += y;` ⇒ 4

15. `x = x + ++y;` ⇒ 7

16. `x = **y;` ⇒ 6.

17. `x = y /*comment;` ⇒ Lexical error.
begin → search for end.

18. `x = y /*comment /*comment*/;` ⇒ 4.
ignore
begin. end.

19. `x = y comment */;` ⇒ 7.

Note:

1. `++` → 1
2. `**` → 2
3. `***` → 3
4. `+=` → 1
5. `=+` → 2
6. `<=` → 1
7. `|=` → 1

No. of tokens.

20. `x = A[20];` ⇒ 7

21. `x = fun(2,3);` ⇒ 9

22. `x = Printf("gate = %d", rank);` ⇒ 9

23. `x = Scanf("%d + %d", x, y);` ⇒ 11

No lexical er.

24. `x = y:` ⇒ 4 but Syntax error

25. `x = 'a';` ⇒ 4

26. `x = '\n';` ⇒ 4

27. `x = 'abc';` Lexical error.

→ T_1 = abc
T_2 = ad
T_3 = abe

Input : adabcabeadadabc
Output : $T_2T_1T_3T_2T_2T_1$.

	T_2 ad	T_1 abc	T_3 abe	T_2 ad	T_2 ad	T_1 abc
T_1	✓ ✗	✓ ✓ ✓				
T_2	✓ ✓	✓ ✗				
T_3	✓ ✗	✓ ✓ ✗				

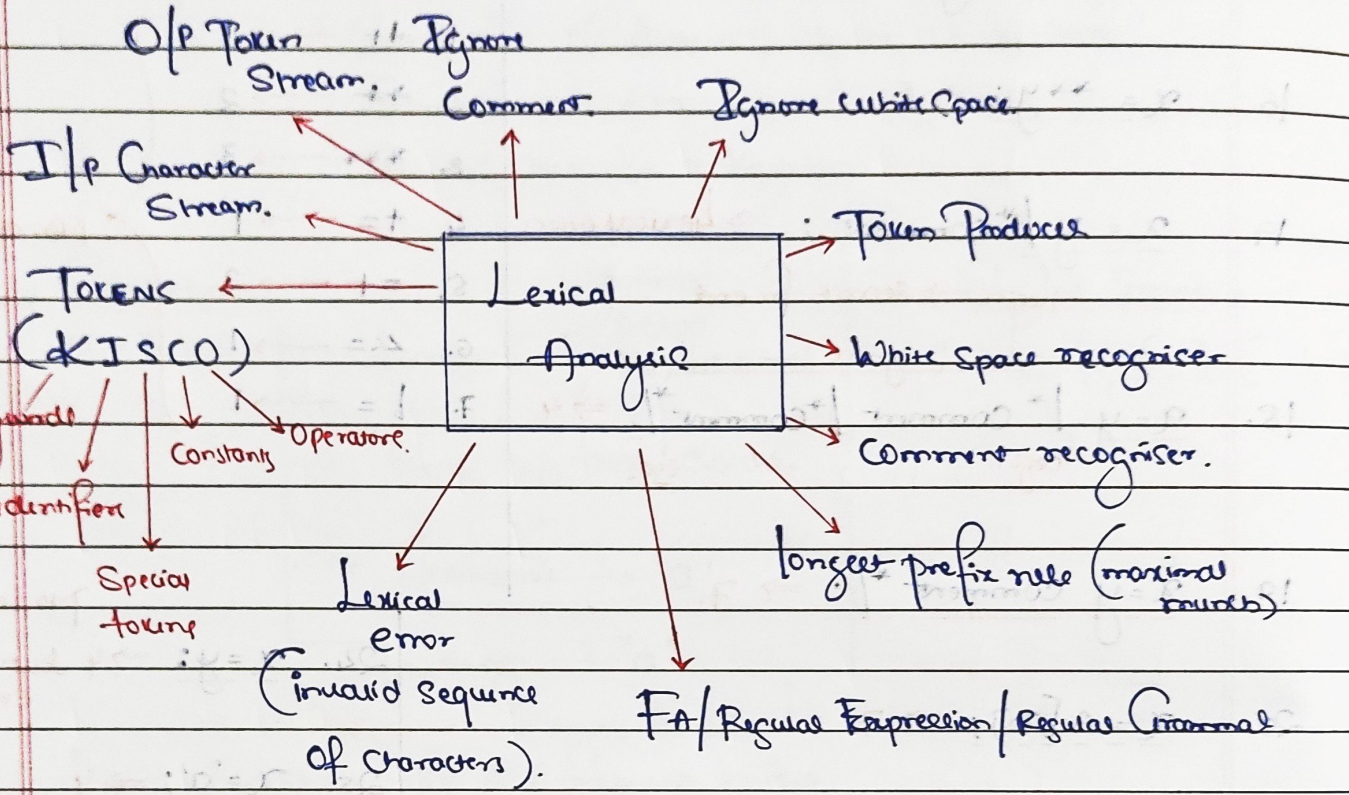
→ $T_1 = a^*b$
$T_2 = a^*bc$

Input = aabcaaabbbc
Output = $T_2T_1T_1T_2$

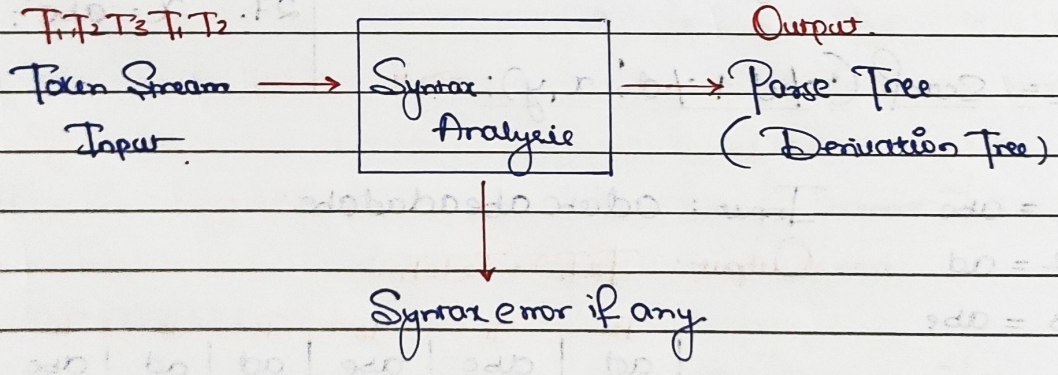
	$T_2 = a^*bc$ aabc	$T_1 = a^*b$ aaab	$T_1 = a^*b$ b	$T_2 = a^*bc$ bc
T_1	✓ ✓ ✓ ✗	✓ ✓ ✓ ✓	✓	✓ ✗
T_2	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	✓	✓ ✓

Incomplete

a^* → zero or more 'a's



SYNTAX ANALYSIS



Syntax Analysis.

- Syntax Analyser
- Parser
- Syntax Verifier
- Parse Tree Generator
- Syntax Analyser.

Syntax Analysis (Process)

- Syntax error finding
- Context free grammar { Most suitable grammar to represent syntax of programming language }
  - definition
  - derivation of string
- Types of CFGs [ambiguous & unambiguous]
- Parsers.

Find the Syntax error in the following codes:

1. void main()
{ int x; } → No error.

2. void main()
{ int x, y; } → No error.

3. int x=y; → No Syntax error
y not declared but Semantic error.

4. int x, int y;
Syntax error.

5. int x+y;
Syntax error

6. int y=2; → No error
int x=y;

7. int x, y=2; → No error.
x=y=2;

8. int x, y;
x=2, y=2; → No error

9. if(2); → No error.
(2) True

10. if(); expression missing.
Syntax error.

11. x, y; No Syntax error
But Semantic error.

12. while(2); No Compilation Error.

13. while(); expression missing, Syntax error.

14. for();
not correct Syntax, error.

15. for(;;); considered true
No compilation error.

16. typedef int x; } x = int.
x y; new name for int (alias)
int y; valid, No error.

17. for(1,2,3): → Syntax error.

18. for(1;2;3); → No Compilation error.

19. for(2,2,2); → No Syntax error.
(either Semantic / linker error)

20. for(2;2;2):
→ Syntax error.

Q1) Consider the following ANSI C Program:

```
int main()
{ Integer e;
  return 0; }
```

Which one of the following phases of a Compiler will throw an error?
→ Semantic Analyser

Q2. Consider the following Statements

I. Symbol table is accessed only during lexical analysis and Syntax analysis.

II. Compilers for programming languages that support recursion necessarily need heap storage for memory allocation in the runtime environment. × (Stack)

III. Errors violating the conditions any variable must be declared before its use are detected during Syntax analysis

Which of them are true:?

Ans None of them are true!

Q3. A lexical analyser uses the following patterns to recognise three tokens T_1, T_2 and T_3 over alphabet $\{a,b,c\}$

$T_1 : a?(b|c)^*a = (\varepsilon+a)(b+c)^*a$

$T_2 : b?(a|c)^*b = (\varepsilon+b)(a+c)^*b$

$T_3 : c?.(b|a)^*c = (\varepsilon+a)(b+a)^*c.$

Note that 'x'? means 0 or 1 occurance of the symbol x. Note also that the analyser outputs the token that matches the longest possible prefix. If the string "bbaacabc" is processed by the analyser, which one of the following is the sequence of tokens it outputs?

Ans $T_3 T_3$.

	b	b	a	a	c	a	b	c
					T_3			T_3.
T_1	✓	✓	✓	×				
T_2	✓	✓	×					
T_3	✓	✓	✓	✓	✓	✓	✓	✓

Q4. Match the following:

P. Parse tree
Q. Character Stream
R. Intermediate representation
S. Token Stream

(i) Code generator
(ii) Syntax analyser
(iii) Semantic analyser
(iv) Lexical analyser

Qs. Match the following:

P. Lexical analysis	(i) leftmost derivation
Q. Top down parsing	(ii) type checking
R. Semantic analysis	(iii) regular expressions
S. Runtime environment	(iv) activation records

Q6. In a compiler, keywords of a language are recognized during

Ans The lexical analysis of a program.

Q7. Which data structure in a compiler is used for managing information about variables and their attributes?

Ans Symbol Table.

Context Free	Context Sensitive
{Structure, Syntax}	{Semantic} {Meaning}
S+V+O	
✓ I am Student	I am Student (Semantically Correct)
✓ I are Student	I are Student (Semantically Wrong)
Integer x; (Syntax Correct)	Integer x; (Semantically Wrong)

Context Free Grammar [CFG]

It can be used to represent Syntax/Structure of programming languages.

$$CFG = (V, T, P, S)$$

- S → Start Symbol $S \in V$
- P → Set of rules (productions)
- T → Set of terminals
- V → Set of Variables (non terminals)

For rule in P:

LHS → RHS

V → Any sequence

V → $(V \cup T)^*$

Identify CFGs.

1	$S \to \varepsilon$ ✓	4	$S \to aSb \mid SS \mid a$ ✓
2	$S \to Sab \mid a$ ✓	5	$S \to S \mid a \mid \varepsilon$ ✓
3	$S \to ABS \mid a$ ✗	6	$S \to aSb$ ✗
	$AB \to a$		$aS \to bb$

S: program.

8 Rules:
- program ⟹ R I ( ) { D; }   (R → void, I → main, D → T I, T → int, I → x)
- R → void / int
- I → identifier
- D → TI
- T → int / float / Char

```
void main()
{
   int x;
}.
```

V = { program, R, I, D, T }

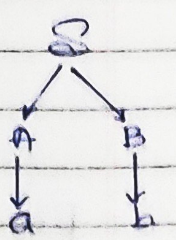
T = { (,), {, ;, }, void, int, Identifier, float, char }

Derivation of a String:

Linear { I Leftmost derivation (LMD)
Linear { II Rightmost derivation (RMD)

Non linear { III. Parse Tree derivation. (PT)

CFG	String
S → AB A → a B → b	w = ab

LMD	RMD	PT
LMD: S → AB → aB → ab	RMD: S → AB → Ab → ab	
In every step left most non-terminal is substituted.	Rightmost non-term is substituted.	Leaf: terminal or 'ε' Nonleaf: non-terminal

classmate
Date
Page

S → AB → aB → ab

Length of derivation = Derivation length = No. of steps = No. of substitutions = 3.

Lmd

S → AB → Ab → ab

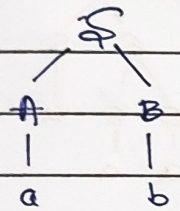
Length of derivation = 3

Rmd

Rmd Order: S, B, A

Lmd Order ⇒ SAB.

Note: Lmd and Rmd need not be same.



Derivation of length for "ab" = No. of leaf nodes = 3.

eg: S → a | Aa
A → ε | b.

⇒

S → a
S → Aa
A → ε, A → b

w = a ⇒ How many derivations for w?

S → a = a

or

S → Aa → εa = a.

= 2 derivations
= 2 parse trees
= 2 Lmds
= 2 Rmds.

"For a given string all of these will be same."

eg: S → AB
A → a
B → b

To this example:
Lmd and Rmd are different.

No. of Lmds for ab = No of Rmds for ab = 1.

No. of steps in Lmd = No. of steps in Rmd = 3

Lmd: S → AB → aB → ab

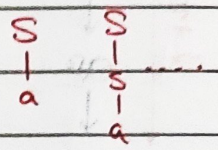
Rmd: S → AB → Ab → ab

1. $S \to SS \mid \varepsilon$

$w = \varepsilon$ No. of derivations = Infinite parse trees.

2. $S \to S \mid a \mid ab$

How many parse trees for "a" : Infinite

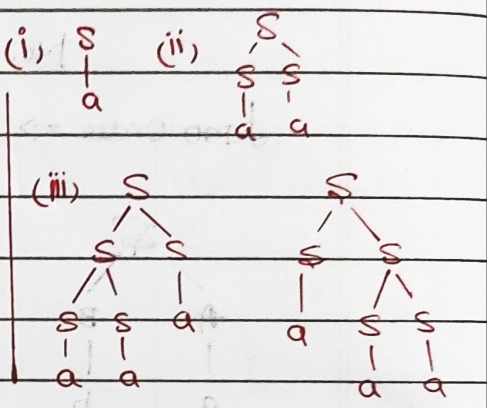


3. $S \to ss \mid a$

(i) How many PTs for "a"? $\Rightarrow 1$

(ii) How many PTs for "aa"? $\Rightarrow 1$

(iii) How many PTs for "aaa"? $\Rightarrow 2$

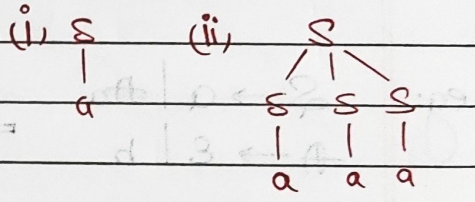


4. $S \to SSS \mid a$

(i) How many PTs for "a"? $\Rightarrow 1$

(ii) How many PTs for "aa"? $\to 0$

(iii) How many PTs for "aaa"? $\Rightarrow 1$



Types of CFGs based on derivations

① Ambiguous CFG	② Unambiguous CFG
$S \to \varepsilon \mid a \mid b \mid A$	$S \to \varepsilon \mid a \mid b$
$A \to a$	ε = 1pt
There is some string, which has more than one derivation.	a = 1pt b = 1pt.
Some string ≥ 1 PT	Every string has only 1 PT
$\exists w \in L(CFG) \Rightarrow > 1$ PT.	Every string only 1 PT
	$\forall w \in L(CFG) \Rightarrow 1$ PT.
* Easy to write	* Better

Identify Ambiguous and Unambiguous CFGs

1. $S \to a$: Unambiguous ✓ $a \Rightarrow 1$ PT

2. $S \to a \mid \varepsilon$: Unambiguous ✓ $\varepsilon = 1$, $a = 1$

3. $S \to AB \mid \varepsilon$: Unambiguous ✓ $\varepsilon = 1$ PT
   $A \to \varepsilon$, $B \to a$ $a = 1$ PT.

4. $S \to AB$
   $A \to \varepsilon$, $B \to \varepsilon$.

: Unambiguous ✓

5. $S \to AB$ $\varepsilon = 1$pt
   $A \to a \mid \varepsilon$ $a = b = 1$pt
   $B \to b \mid \varepsilon$ $ab = 1$pt.
   : Unambiguous ✓

6. $S \to AB$ $\varepsilon = 1$pt
   $A \to aA \mid \varepsilon$ $a = b = 1$pt
   $B \to bB \mid \varepsilon$. : Unambiguous ✓

7. $S \to AB$ $\varepsilon = 1$pt
   $A \to aA \mid \varepsilon$ $a = 2$pts
   $B \to aB \mid \varepsilon$.
   : Ambiguous ✓

8. $S \to SS \mid \varepsilon$.
   : Ambiguous ✓

9. $S \to Ss \mid a$
   $a = 1$pt
   $aa = 1$pt : Ambiguous ✓
   $aaa = 2$pts.

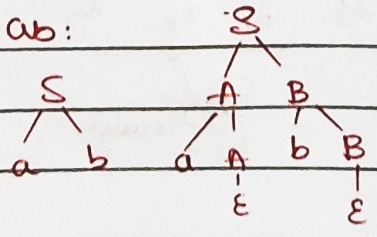
10. $S \to SaS \mid b$: left and right rec with same S.
    Ambiguous ✓

11. $S \to Sa \mid bS \mid c$: Ambiguous ✓

12. $E \to E + E \mid a$: Ambiguous ✓

13. $S \to AB \mid ab$
    $A \to aA \mid \varepsilon$
    $B \to bB \mid \varepsilon$

ab:



: Ambiguous ✓

Elimination of Left Recursion

(Conversion from left recursion to right recursion)

* **Top-down parser follows LMD**

$S \to Sa \mid b$

(first symbol)

b | .....

(first symbol)

1. $S \to Sa \mid b$

(left recursion) (non-left recursion)

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

$L = \{ b, ba, baa, baaa, \dots \}$

$= \{ ba^0, ba^1, ba^2, \dots \}$

$= ba^*$.

$S \to bS'$

$S' \to aS' \mid \varepsilon$

$S \to bX$

$X \to aX \mid \varepsilon$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$.

Formula:

(Direct left recursion Elimination)

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_k$

(left recursion) (non-left recursion)

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_k A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$

(Right Recursion)

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

$\beta \in (V \cup T)$

'Should not start with A'

2. $S \to Sab \mid c \mid d$

(α | B_1 | B_2)

$S \to Sx \mid B_1 \mid B_2$

$\Rightarrow$

$S \to cS' \mid dS'$

$S' \to abS' \mid \varepsilon$

or

$S \to cS' \mid dS' \mid c \mid d$

$S' \to abS' \mid ab$

3. $S \to Sa \mid Sb \mid cd \mid e \mid fs \mid sg$

$S \to Sa \mid Sb \mid Sg \mid cd \mid e \mid fs$

(α_1 α_2 α_3 | B_1 B_2 B_3)

(left Recursion) (Right recursion)

$S \to cdx \mid ex \mid fsx$

$X \to ax \mid bx \mid gx \mid \varepsilon$.

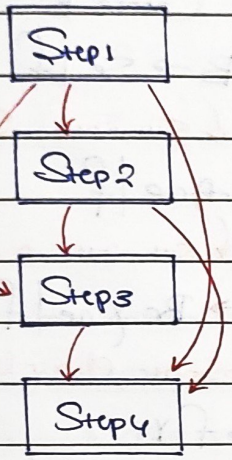
Direct left recursion Elimination

Left Recursion

$S \to Sa \mid b$ — Direct left recursion

$S \to Aa \mid d$
$A \to Sb \mid c$ — Indirect Left recursion

We can convert left recursion into direct left recursion using "Substitution".



Step 1: S (Direct left recursion in S)

Step 2: A (i) Substitute S in A item. (ii) Eliminate direct left-recursion in A.

Step 3: B (i) Substitute S and A productions recursively in B then (ii) Apply direct left recursion in B.

Step 4: C (i) Substitute S, A, B in C then (ii) Eliminate direct left recursion in C.

1. $S \to Aa \mid Sb \mid c$
$A \to Sb \mid Ae \mid f$

Step 1: Choose S productions:

$S \to Sb \mid Aa \mid c$

(Sb: left rec; Aa, c: Non left recursion.)

Direct left recursion

$S \to Aax \mid cx$ → (1)
$X \to bX \mid \varepsilon$

Step 2: Choose A productions.

$A \to Sb \mid Ae \mid f$

↳ Substitute S rules if appears in 1st position.

$S \to Aax \mid cx$

$A \to Aaxb \mid cxb \mid Ae \mid f$

(α_1 = axb, β_1 = cxb, α_2 = Ae, β_2 = f)

Direct left recursion

$A \to cxbY \mid fY$ → (2)
$Y \to axbY \mid eY \mid \varepsilon$

2. $E \to E+T \mid F$
$T \to F*T \mid F$ $\quad F \to id$

classmate
Date
Page

Step 1: $E \rightarrow E+T \mid T$ (α, β)

$$E \rightarrow T X$$
$$X \rightarrow +TX \mid \epsilon$$
— ①

Step 2: $T \rightarrow F*T \mid F$ — ②

(as E is in 1st position, so no substituting, no direct left recursion)

Step 3: $F \rightarrow id$ — ③

{Answer 1, 2, 3 combined}

2.
$S \rightarrow Aa$
$A \rightarrow Sb \mid Bc$
$B \rightarrow Se \mid f$.

Step 1: Choose S

$S \rightarrow Aa$ → ①

Step 2: Choose A

$A \rightarrow Sb \mid Bc$

(Substitute 'S' in 1st place: $S \rightarrow Aa$)

$A \rightarrow Aab \mid Bc$

(Eliminate direct left recursion in A.)

$$A \rightarrow BcX$$
$$X \rightarrow abX \mid \epsilon$$
→ ②

Step 3: Choose B

$B \rightarrow Se \mid f$

(Substitute S & A in B)

$B \rightarrow Aae \mid f$

(Substitute A in B.)

$B \rightarrow BcXae \mid f$

(eliminate direct left recursion)

$$B \rightarrow fY$$
$$Y \rightarrow cXaeY \mid \epsilon$$
→ ③

$X \rightarrow X\alpha_1 \mid X\alpha_2 \mid \beta_1 \mid \beta_2 \mid \beta_3$

↓

$X \rightarrow \beta_1 X' \mid \beta_2 X' \mid \beta_3 X'$
$X' \rightarrow \alpha_1 X' \mid \alpha_2 X' \mid \epsilon$

$S \rightarrow aX$
$X \rightarrow b \mid cd$

Left Refactoring:

CFG → [Elimination of common Prefix (left factoring)] → Left factored CFG (CFG without common Prefixes).

not left factored

$S \rightarrow ab \mid acd$

a is begin common Prefix

↓

issue is backtracking

1. $S \to a$: left factored CFG

2. $S \to a \mid \varepsilon \mid bc$: left factored CFG

3. $S \to Aa$, $A \to b$ } left factored CFG

4. $S \to ab \mid a \mid cd$

$S \to ax \mid cd$ left
$X \to b \mid \varepsilon$ factored

5. $S \to a \mid ab \mid abc \mid ef$.

$S \to ax \mid ef$ → (1)

$X \to \varepsilon \mid b \mid bc$

$X \to \varepsilon \mid bY$ → (2)
$Y \to \varepsilon \mid c$

6. $S \to Ab \mid aCA \mid d$

$S \to aeb \mid gb \mid aCA \mid d$

Ans: $S \to ax \mid gb \mid d$
$X \to Cb \mid CA$
$A \to ae \mid g$

FIRST SET

$S \to abc \mid de \mid \varepsilon$.

FIRST(S) = {a, d, ε}.
= {x | x is either ε or terminal where terminal is derived as 1st symbol from S}.

FIRST(X) = It is set of terminals (may include ε) where each terminal is derived as 1st symbol from X.

First(abc) = {a}
First(de) = {d}

We need X rules for computing First(X).

FOLLOW SET

$X \to aXb \mid Yb \mid b$
$Y \to bXef \mid Xm \mid \varepsilon$.

Follow(X) = {b, e, m}.
= {t | t is a terminal derived as 1st symbol after X.}.

Follow(X) = It is set of terminals where every terminal is derived as 1st symbol after X.

We need whole CFG where productions contain X helps to compute Follow(X).

Note: If S is start symbol then Follow(S) should contain $.
"$" is special end terminal

eg: S → Sa | Sb | c
Follow(S) = {$, a, b}.
(S = start, | = pipe)

(1) S → a | ε | bc Follow(S) = {$}
First(S) = {a, ε, b}.

(2) S → Sa | ε. Follow(S) = {$, a}
First(S) = {ε, a}.

(3) S → Ab | cde Follow(S) = {$} First(S) = {c, a, b, f}
A → ae | ε | fg. Follow(A) = {b}. First(A) = {a, ε, f}.

(4) S → A First(S) = {ε} Follow(S) = {$}
A → ε First(A) = {ε}. Follow(A) = {$}.

Note: X → αY
Follow(Y) = Follow(X)
{LHS.}

Xa
αYa (Same as Follow(X))

..... Xa .....
..... αYa .....
If you want to find after of Y, it is same as whatever after X.

First(X): X → O | O | O
(1st symbol (terminal), 1st, 1st.

Follow(X): Whole CFG take rule which contains X in RHS
LHS → X

(5) $S \to AB$, $A \to a$, $B \to b$.

	S	A	B
First	{a}	{a}	{b}
Follow	{$}	{b}	{$}

(6) $S \to AaBb$, $A \to f$, $B \to f$.

	S	A	B
First	{f}	{f}	{f}
Follow	{$}	{a}	{b}

(7) $S \to AB$, $A \to ab$, $B \to \varepsilon$.

	S	A	B
First	{f}	{f}	{f}
Follow	{$}	{a}	{b}

(8) $S \to AB$, $A \to \varepsilon$, $B \to ef$.

	S	A	B
First	{e}	{e}	{e}
Follow	{$}	{e}	{$}

(9) $S \to Ss \mid a$

	S
First	{a}
Follow	{$a}

(10) $S \to SS \mid (S) \mid \varepsilon$.

	S
First	{ε, c}
Follow	{$, c,)}

(11) $E \to E+E \mid E*E \mid (E) \mid a$

$First(E) = |\{c, a\}| = 2$

$Follow(E) = |\{+, *,), \$\}| = 4$

(12) $S \to AB$, $A \to ab \mid \varepsilon$, $B \to cd \mid \varepsilon$.

	S	A	B
First	a, c, ε	a, ε	c, ε
Follow	$	c, $	$

$S \to AB$

$S \to AcD$ — {c}

$S \to A$ — Follow(S)

(13) $S \to ABC$, $A \to ab \mid \varepsilon$, $B \to cd \mid \varepsilon$, $C \to fg \mid \varepsilon$.

	S	A	B	C
First	{a, c, f, ε}	{a, ε}	{c, ε}	{f, ε}
Follow	{$}	{c, f, $}	{f, $}	{$}

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

	E	T	F
First	{(, id}	{(, id}	{(, id}
Follow	{$, +,)}	{$, +,)}	{*, $, +,)}

$Fo(E) = \{*\} \cup Fo(T)$

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

$Follow(F) = \{*\}$, $Follow(F) = Follow(T)$

$= \{*\} \cup Follow(T)$

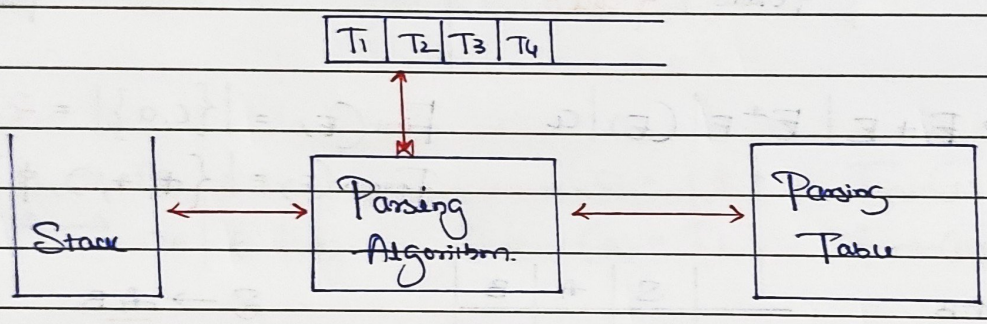
↳ is always present in Follow(start symbol)

Classmate
Date
Page

Parsers

- Top Down Parser
  - LL(0) parser
  - LL(1) parser.
- Bottom-up Parser (SR parser)
  - LR parser
    - LR(0)
    - SLR(1)
    - LALR(1)
    - ⋮
  - Operator precedence parser

Parser Configuration



TDP	BUP
I) Followed "Lmd" (L)	I) Followed "Rmd" in reverse (R)
LL(1)	LR
↓ left to right scan.	↓ left to right scan
II) LL(1) parser def - ault.	II) LR(1) Parser powerful

LL(1) Parser:

* Top down Parser
* Predictive Parser
* It can be implemented in 2 ways:
  - (i) Recursive-descendent parser [procedure driven approach]
    Every Production equivalent procedure.

$S \rightarrow aA$ S()
{ if (nextToken == a) A() }

(ii) Non Recursive - descent parser [Table driven approach]

	a	q
S	$S \to aA$	$S \to q$

↳ every production will be stored in a table.

Q1. LL(1) parser is _____

A. Top down parser
B. Bottom up parser
C. Recursive - descent Predi
D. Non-recursive descent predictive parser

How to write LL(1) CFG?

Step1: Take Unambiguous CFG
↓ Eliminate left recursion
Step2: Unambiguous and non left recursive CFG
↓ eliminate common prefixes of 1 keys, (left factoring)
Step3: Unambiguous, non left recursive, left factored CFG
↓
Step4: Build LL(1) Table
If every cell has atmost 1 Rule then given CFG is LL(1)

How to identify LL(1) CFG?

** I. Theory
II. Table Concept
*** III. Shortcut.

Q2. Which of the following is sufficient to be LL(1) CFG?

A. Ambiguous, left recursion, left factored CFG
B. Unambiguous, non left recursive, non left factored CFG
C. Unambiguous, non left recursive, left factored CFG
D. None of these.

If CFG is unambiguous, non left recursive, and left factored then need not be LL(1)

Q2 Which of the following is necessary to be LL(1) CFG?

A. Ambiguous, left recursion, left factored CFG

B. Unambiguous, non left recursive, non left factored CFG

C. ✓ Unambiguous, non left recursive, left factored CFG

D. None of these

If CFG is LL(1) then it is unambiguous, non left recursive and left factored

LL(1) Table Construction

Step 1: Compute first set for every non terminal in CFG.

Step 2: If any first set contain ε then compute follow set for the same non terminal

Step 3: Fill the table with First and Follow sets

First M[S, a]:

	a
S	Fill with all productions of S which derive 'a' as 1st symbol

Follow M[S, b]:

	b
S	Fill all productions of S which derive ε

① $S \to aSb \mid \varepsilon$

$First(S) = \{a, \varepsilon\}$

$Follow(S) = \{\$, b\}$

Every entry of table have atmost 1 Rule

↳ Given CFG is LL(1)

M	a	b	$
S	$S \to aSb$	$S \to \varepsilon$	$S \to \varepsilon$

② $S \to aSa \mid \varepsilon$ $First(S) = \{a, \varepsilon\}$

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

This CFG is unambiguous, non left recur, left factored.

classmate

Date 07/9/23

Page

M[S, a] has more than 1 rule

↓ Given CFG is not LL(1)

	a	$
S	S → aSa S → ε.	S → ε.

③ S → SS | (S) | ε

First(S) = { (, ε }

Follow(S) = {), (, $ }

S → SS

ε

S → ε.

Only focus on deriving ε

	(	)	$
S	S → (S) S → SS S → ε	S → SS S → ε	S → SS S → ε

④ S → AB.

A → Aa | ε

B → Bb | ε.

Fi(S) = {a, b, ε}

Fi(A) = {a, ε}

Fi(B) = {b, ε}.

Fo(S) = {$}

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

Fo(B) = {b, $}.

	a	b	$
S	AB	AB	AB
A	Aa ε	ε	ε
B		Bb ε	ε

⑤ S → aAb

A → Bb | ε

B → ab

Fi(S) = {a}

Fi(A) = {a, ε}

Fi(B) = {a}.

Fo(A) = {b}.

	a	b	$
S	S → aAb		
A	A → Bb	A → ε.	
B	B → ab		

Shortcut for identifying LL(1) CFG

Rule 1: X → α₁ | α₂

If first(α₁) ∩ First(α₂) ≠ 0 then given CFG is not LL(1)

S → a | ab | c → It is not LL(1)

Rule 1   Rule 1   Rule 1

= ϕ.

Rule 2: X → α | ε

If First(α) ∩ Follow(X) ≠ 0, then given CFG is not LL(1)

LHS

~~It is not~~

eg. $S \to aSb \mid \varepsilon$

$First(a|b) \cap Follow(S) = \phi$ } It is LL(1)

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

① $S \to a \mid b \mid cd \mid \varepsilon$. ⇒ It is LL(1)

② $S \to Sa \mid bS \mid c \mid d \mid \varepsilon$. → Not LL(1)

③ $S \to a \mid bc \mid aef$ ⇒ Not LL(1).

④ $S \to AB \mid ab$
$A \to a \mid \varepsilon$ → Not LL(1)
$B \to b \mid \varepsilon$.

⑤ $S \to aAbB \mid bAaB \mid \varepsilon$
$A \to S$
$B \to S$
Not LL(1)

Q1. If CFG is not LL(1) then which is Possible?

~~A.~~ Ambiguous
~~B.~~ Left recursive
~~C.~~ Non left factored
~~D.~~ Some table entry may contain more than one Productions.

Q2. If CFG is not LL(1) then which is True?

A. Ambiguous
B. Left recursive } Necessary.
C. Non left factored
guranteed ~~D.~~ Some table entry may contain more than one Productions

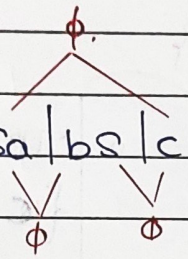
Q3. If CFG is LL(1) then which is True?

~~A.~~ Unambiguous
~~B.~~ Non left recursive
~~C.~~ Left factored
~~D.~~ A, B, & C
~~E.~~ Every entry of table has almost 1 production

Q4. CFG is LL(1) if CFG is _____

A. Unambiguous
B. Non left recursive
C. Left factored
D. ~~A, B & C~~
E. Every entry of table has almost 1 production
F. ~~A, B, C, & E~~
G. B & C
H. C & E
I. B & E
J. ~~A, B, E~~

(1) $S \to aSa \mid bS \mid c$ is LL(1)



(2) $S \to \overset{I}{aAbB} \mid \overset{II}{bAaB} \mid \overset{III}{\varepsilon}$

$A \to S^{II}$

$B \to S^{I}$

	a	b	$
S	I, E_1 III	II, E_2 III	III
A	.	.	.
B	.	.	.

What is E_1 and E_2?

First(S) = {a, b, ε}

Follow(S) = {$, a, b}.

E_1 = I, III.
= {S → aAbB, S → ε}

E_2 = II, III = {S → bAaB, S → ε}

(3) $P \to xQRS$

$Q \to yz \mid z$ Follow(Q) = {w, y}.

$R \to w \mid \varepsilon$

$S \to y$.

(4) $E \to E-T \mid T$

$T \to T+F \mid F$

$F \to (E) \mid id$

Equivalent non-left recursive grammar is:

$E \to Tx$

$x \to -Tx \mid \varepsilon$

$T \to Fy$

$y \to +Fy \mid \varepsilon$

$F \to (E) \mid id$

(5.) $S \to daT \mid Rf$
$T \to aS \mid baT \mid \varepsilon$
$R \to caTR \mid \varepsilon$

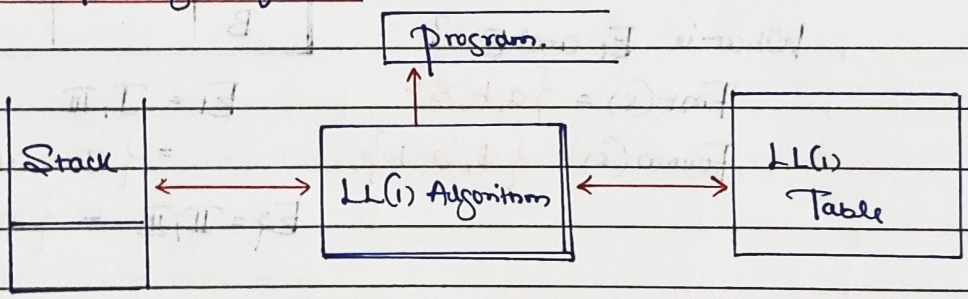
	a	b	c	d	f	$
S			Rf ①		Rf ②	
T			$T \to \varepsilon$ ③			④ $T \to \varepsilon$
R						

$Fo(S) = Fo(T) \cup \{\$\}$
$Fo(R) = \{f\} \cup Fo(R)$

$First(S) = \{a, c, f\}$
$First(T) = \{a, b, \varepsilon\}$
$Follow(T) = \{\$, c, f\}$

$S \to daT$	$T \to baT$	$R \to caTR$	$R \to ca$	TR
$Fo(T) = Fo(S)$ $= Fo(R) \cup \{\$\}$	X	$\{c\} \cup Fo(R)$ $\{c\} \cup \{f\}$		$\{\varepsilon\} \cup Fo(R)$

LL(1) Parsing Algorithm



I/P = ab$

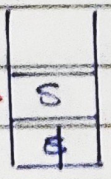
Given Table:

M	a	b	$
S	$S \to aSb$	$S \to \varepsilon$	$S \to \varepsilon$

I Prediction
If T.o.S = Non-terminal then
M[T.o.S, input] : LHS → RHS
(i) Pop T.o.S (LHS)
(ii) Push RHS in reverse order

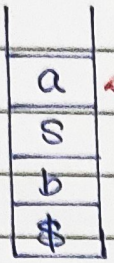
II Match T.o.S == I/P
(i) Pop T.o.S
(ii) Increment Pointer.

Step 1:



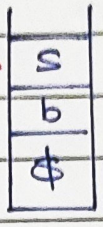
ab$ (Prediction)
M[S, a], $S \to aSb$ → : (i) Pop S
(ii) Push b, S, a

Step 2:



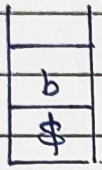
ab$
↑ T.o.s == i/p.
∴ match
(i) pop off S
(ii) increment pointer.

Step 3: b$
↑ M[S, b] S → ε



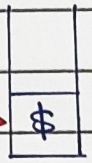
(i) pop S
(ii) Push nothing.

Step 4: b$
↑ T.o.s == i/p



(i) pop b
(ii) increment pointer

Step 5: $
↑ T.o.s == $ == i/p
ACCEPT
(No syntax error)



eg: S → aSb | ε.

ab	S								
	↓		Prediction →	a	match →		Prediction →		match ↓
Lmo	aSb	S	S → aSb	S		S		b	
	↓	b		b		b		$	
↓	ab	$		$		$			

What is max size of stack by assuming initially $ and S are already on stack?
⇒ 4.

Conflicts During LL(1) Parsing.

Syntax Error

I) N-Conflict: If ToS is non-terminal and table has blank entry, then N-conflict.

II) T-Conflict: If T.oS is terminal and T.oS ≠ ~~terminal~~ I/P then T-conflict.

M	t
X	X → α

If T.o.S == X
Prediction (Substitution in Lmo)
→ Pop X
→ push α in reverse order

classmate
Date 08/9/23.
Page

① $S \to a$ LL(0) CFG ✓ [LL(1), LL(2)...]

② $S \to aA$, $A \to bB$, $B \to c$ LL(0) CFG ✓ [LL(1), LL(2)....]

③ $S \to a|b$ LL(0) ×
LL(1) ✓ and LL(2)....

④ $S \to a|ab$ LL(0) × LL(1) × LL(2) ✓

⑤ $S \to a|f|abc|abcde$
LL(4) ✓ (and above).

Note: * Every LL(k) is LL(k+1) CFG
* Every LL(2) CFG is convertable to LL(1) CFG
* Every LL(3) CFG is convertable to LL(2) CFG
* Set of L.L. languages generated by LL(1) CFGs
=
Set of languages generated by LL(2) CFGs

TDP	BUP
1. Predictive parser (LL(1) parser)	2. SR parser
2. Predictions (Lmd Substitutions).	2. Shift and reduced actions
3. Usee Lmd to verify Syntax (to produce Parse tree) or (to derive input)	3. Usee "Reverse of Rmd".

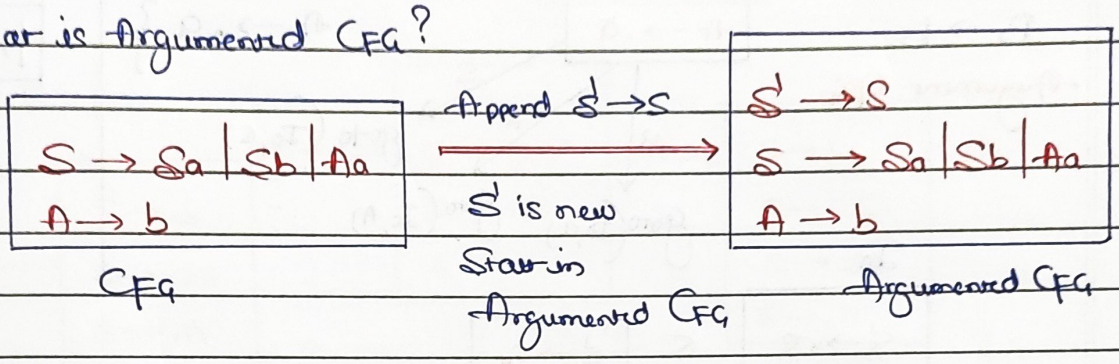
LR(0) Parser:
- no look ahead
- Rmd in reverse
- Left to right Scanning.

1>. LR(0) Parsing diagram [LR(0) DFA]
2>. Conflicts checking for LR(0) CFG
   - → SR Conflict
   - → RR Conflict
3>. LR(0) Table Construction.

LR(0) DFA:

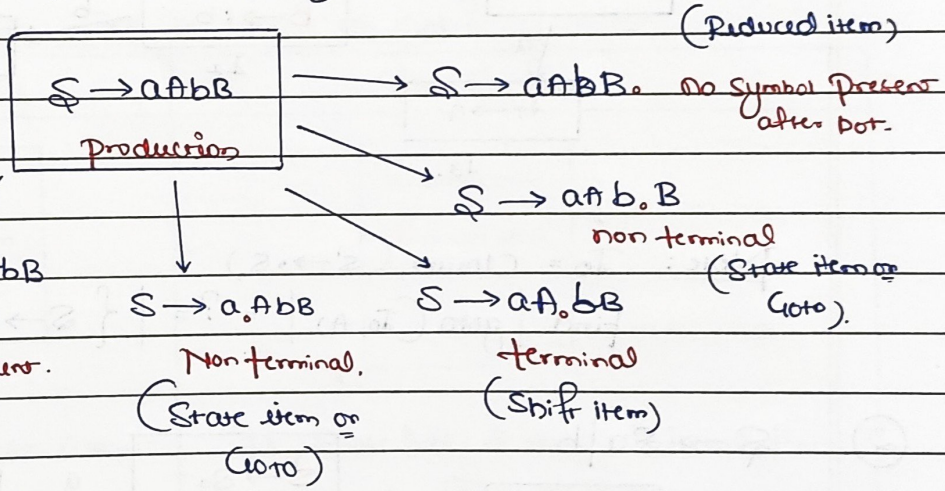
1.) Argumented CFG
2.) Item and types of items?
3.) Closure() and goto() functions. ***
4.) How to construct LR(0) DFA?
5.) How to check SR and RR conflicts in LR(0)?

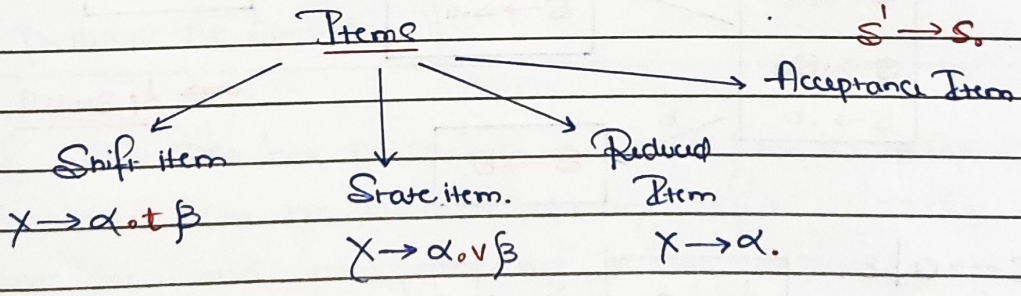
What is Argumented CFG?



LR(0) Item?

→ It is a production along with a Dot.





Note:

Dot is used to track progress in derivation (to perform Shift and Reduced actions).

Note: * We must focus on "Dot" and next symbol after "Dot".

* If non-terminal present just after "Dot" then we must add all productions in the same State.

① $S' \to S$
$S \to AB$
$A \to a$
$B \to b$
Augmented CFG

I_0
$S' \to .S$
$S \to .AB$
$A \to .a$

Closure $(S' \to S)$ { $S' \to .S$, $S \to .AB$, $A \to .a$ }

let X = Closure $(S \to A.B)$
then $|X| = ?$
$X = \{ S \to A.B,\ B \to .b \}$
$|X| = 2$

goto (I_0, S), goto (I_0, a), goto (I_0, A)

I_0
$S' \to .S$ —S→ $S' \to S.$ (I_1)
$S \to .AB$ —A→ $S \to A.B$, $B \to .b$ (I_2)
$A \to .a$ —a→ $A \to a.$ (I_3)
$S \to A.B$ —B→ $S \to AB.$ (I_4)
$B \to .b$ —b→ $B \to b.$ (I_5)

⟹ 6 States

Note: I_0 = Closure $(S' \to .S)$
Find $|\text{goto}(I_0, A)| = ? = |\{ S \to A.B,\ B \to .b \}| = 2.$

② $S \to Sa \mid b$

$S' \to .S$
$S \to .Sa$
$S \to .b$

—S→ $S' \to S.$, $S \to S.a$
—a→ $S \to Sa$
—b→ $S \to b.$

⟹ 4 States

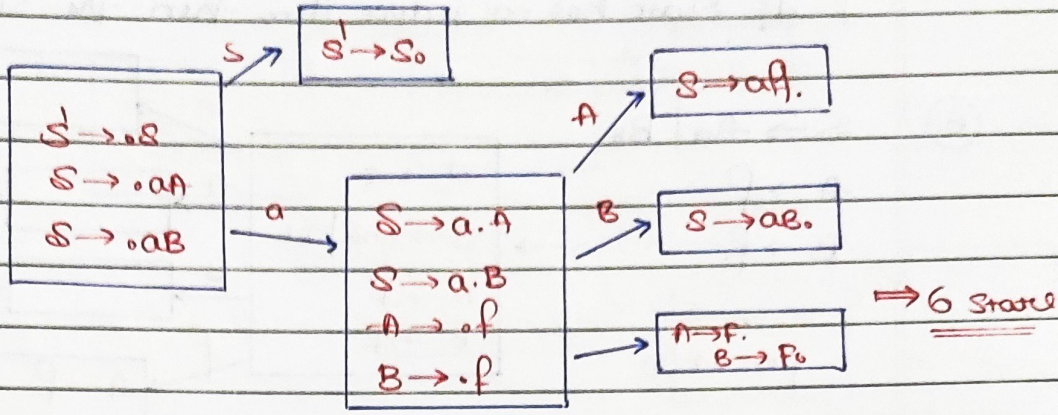
⑤ eg: $S \to a \mid \varepsilon$

$S' \to .S$
$S \to .S$
$S \to .$

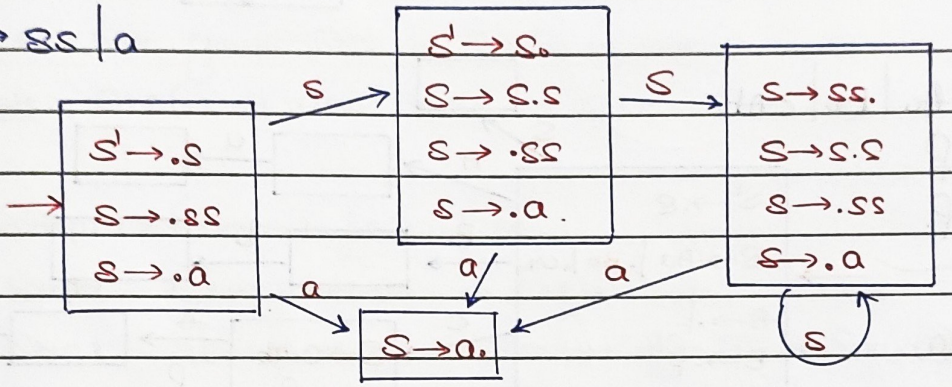
—S→ $S' \to S.$
—a→ $S \to a.$

classmate
Date
Page

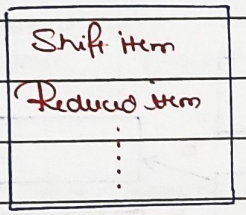
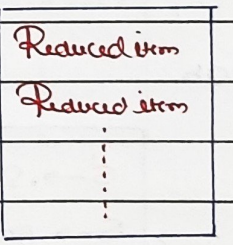
(3) $S \to aA \mid aB$
$A \to f$
$B \to f$



(4) $S \to SS \mid a$



How to Check given CFG is LR(0) or not?

SR Conflict	RR Conflict
	
If state has both S item and R item then it Produces SR Conflict	If State has 2 reduced items then it produces RR conflict

→ If LR(0) DFA has no conflicts then given CFG is LR(0)
→ Acceptance item not participate in any conflict $S' \to S.$
→ State item also not participate in any conflict
→ State with only one item never produces any conflict
→ If state has reduced item then only there is a possibility to produce either RR or SR conflict.

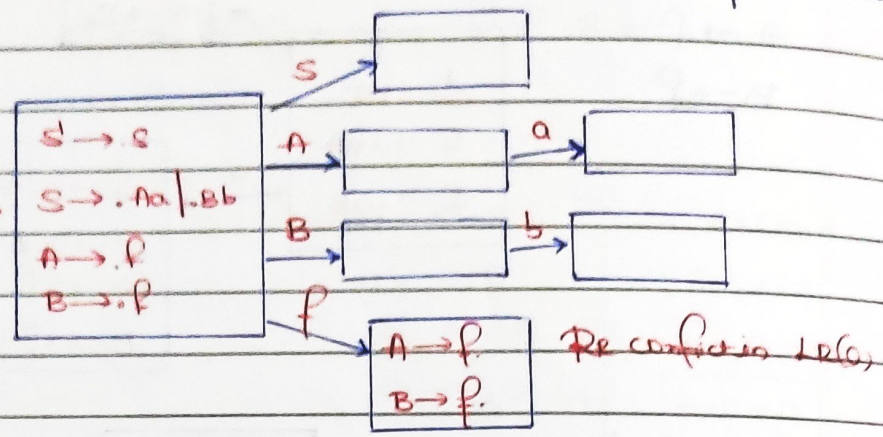
→ If state has no reduce item then the state never produces conflict

⑥ $S \to Aa \mid Bb$

$A \to f$

$B \to f$

Not LR(0)

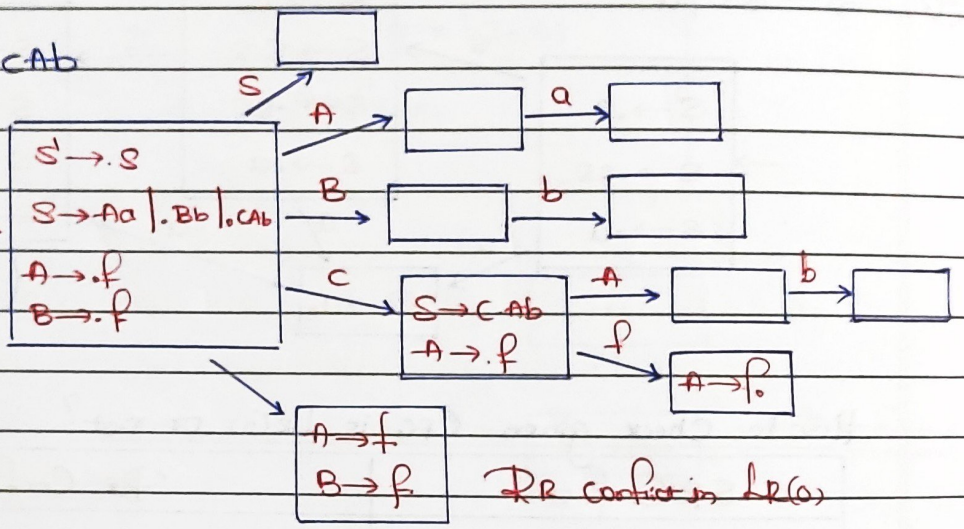


⑦ $S \to Aa \mid Bb \mid cAb$

$A \to f$

$B \to f$

Not LR(0)

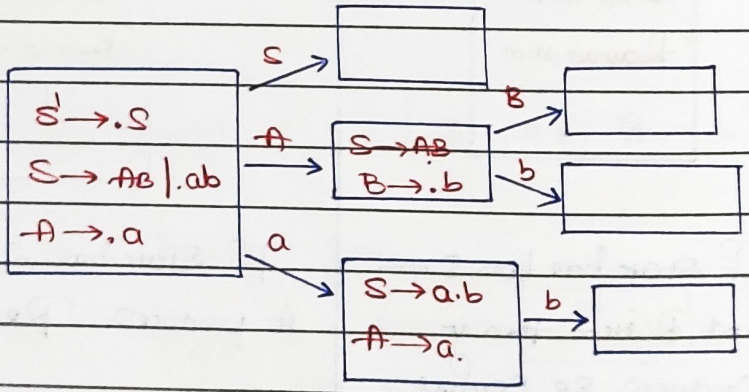


⑧ $S \to AB \mid ab$

$A \to a$

$B \to b$

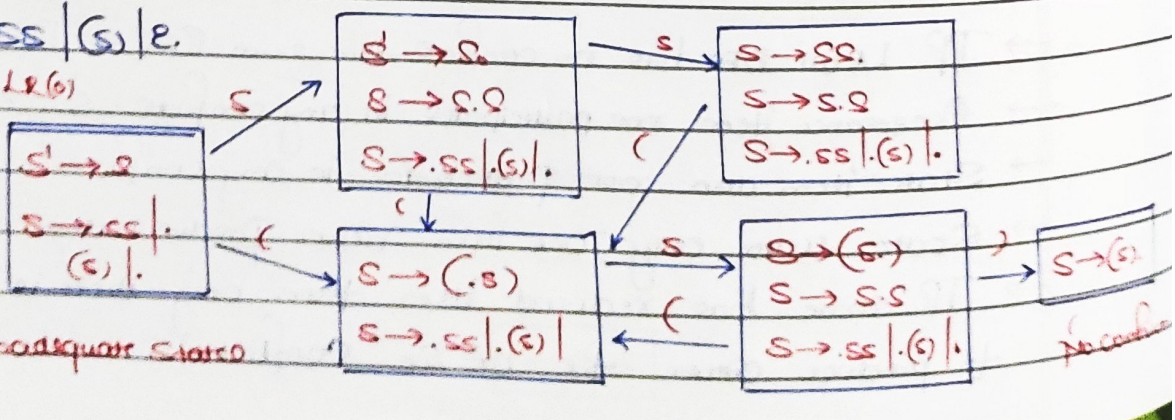
Not LR(0)



SR Conflict in LR(0)

⑨ $S \to SS \mid (S) \mid \varepsilon$

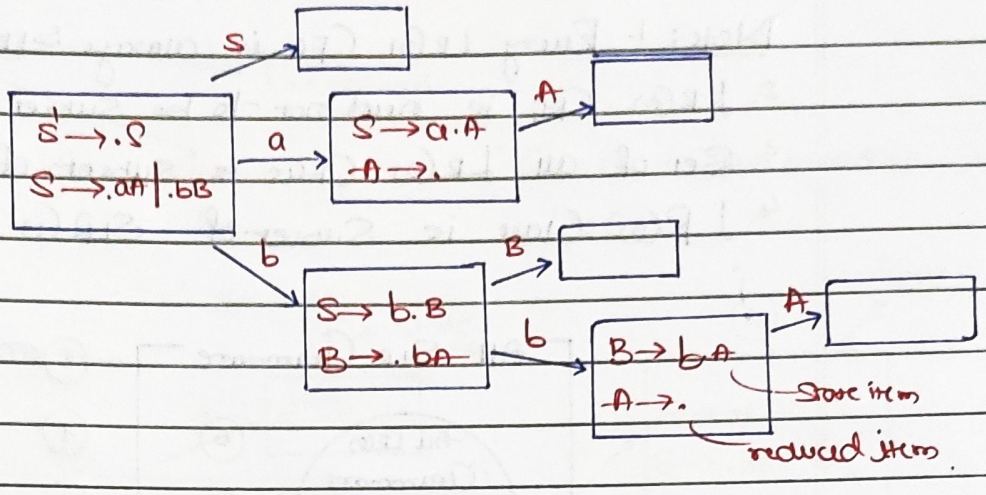
(Not LR(0))



(10) $S \to aA \mid bB$

$A \to \varepsilon$

$B \to bA$



How to check given CFG is SLR(1) or not?

Note: SLR(1) is also called as SLR.

Step1: Construct LR(0) DFA

Step2: Check conflicts in SLR(1)

Step3: If there is no conflict then CFG is SLR(1).

S → Simple

L → Left to right scan

R → Rmost of Rmd

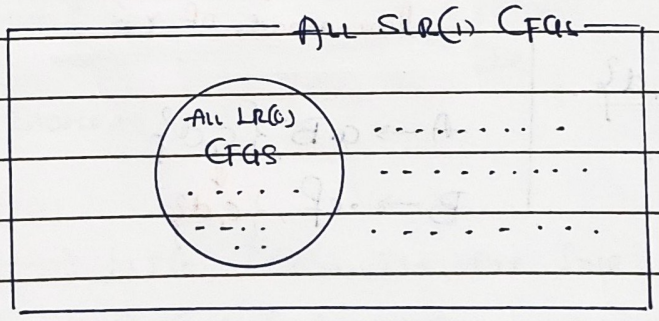
1 → One look ahead (Computed using Cubole CFG)

SR Conflict	RR Conflict
Shift item. / Reduced item. / ⋮ : $X \to \alpha . t \beta$. ; $Y \to \alpha . t$	Reduced item1 : $X \to a.$; Reduced item 2 : $Y \to a.$; ⋮ — If Follow(X) ∩ Follow(Y) ≠ ϕ.
* If $t \in$ Follow(Y) then SR conflict in SLR(1)	

→ If LR(0) DFA has no SLR Conflicts then given CFG is SLR.

→ "Terminal of shift item" present in "follow of reduced item".

$t \in$ Follow(Reduced Item).

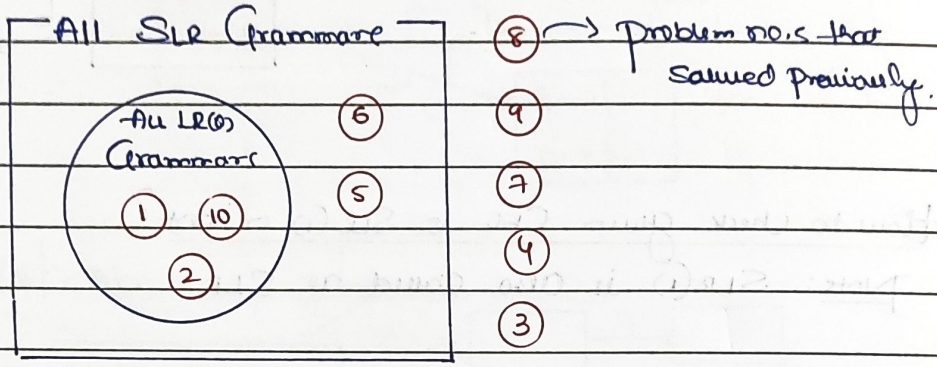


Note: 1. Every LR(0) CFG is always SLR(1) CFG

2. LR(0) CFG is need not to be subset of SLR(1) CFG

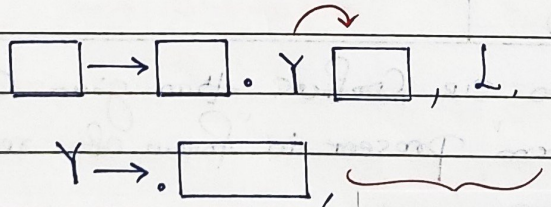
3. Set of all LR(0) CFGs is subset of set of all SLR(1) CFGs

4. LR(0) Class is subset of SLR(1) Class.



LR(0) Parser, SLR(1) Parser } Depends upon LR(0) Items	CLR Parser, LALR parser } Depends upon LR(1) items
$X \to \alpha . \beta$ LR(0) Item.	LR(1) Item: LR(0) Item + look-ahead set $X \to \alpha . \beta, L$

How to compute look-ahead set in LR(1) item?



$X \to \alpha . Y \beta, L$

$Y \to . \gamma, First(\beta L)$

$A \to a . B b, \{c,d\}$ $B \to . ef, \{b\}$	$A \to a . B, \{c,d\}$ $B \to . ef, \{c,d\}$

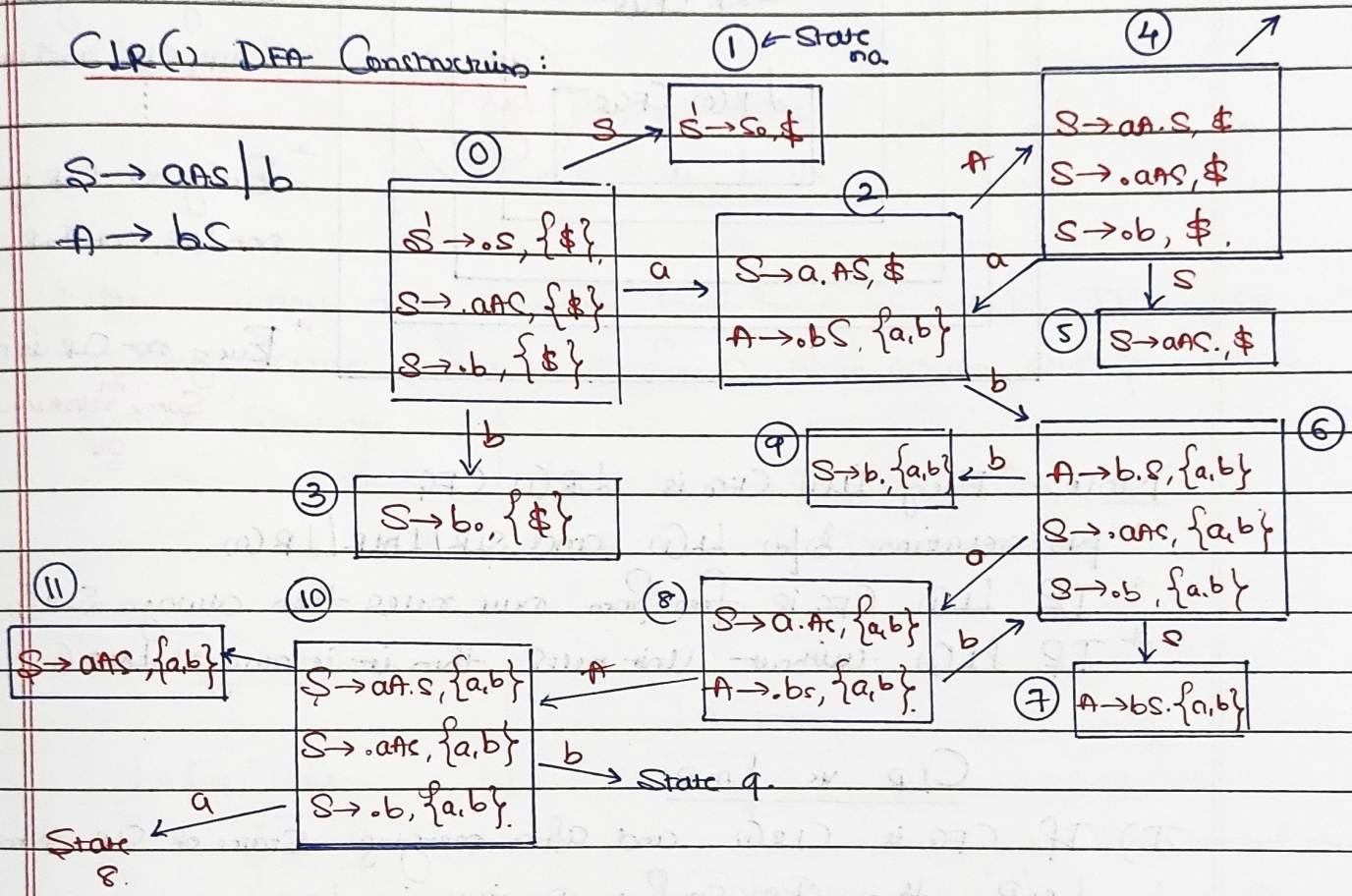
CLR(1) → look-ahead

- Canonical (exact)
- left to right Scan
- Rmost of Rmo

* CLR(1) is also called as LR(1)

CLR(1) DFA Construction:

1. S → aAS | b
   A → bS



LALR(1) DFA: Step 1: Construct CLR DFA

Step 2: Merge state of CLR if they have same items but look-aheads may be different.

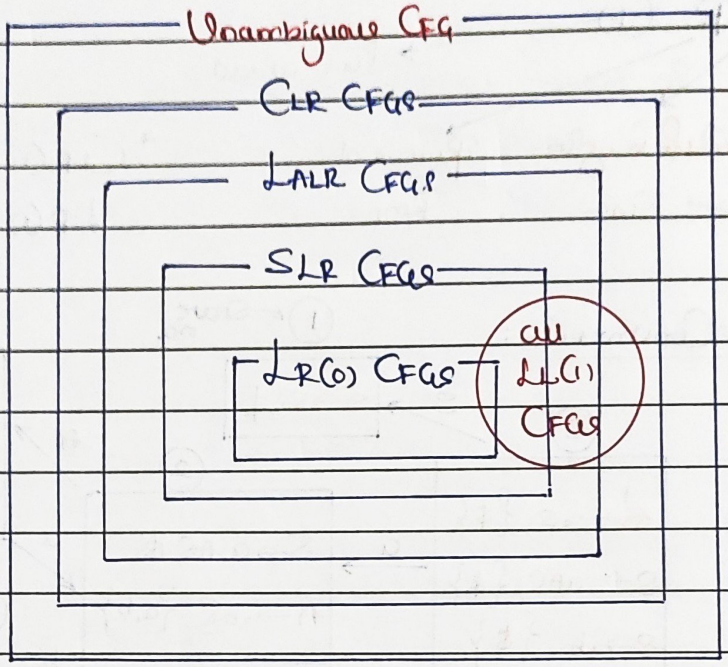
No of States in LR(0)
=
No of States in SLR(1)
=
No of States in LALR

≤ no. of states in CLR.

2. S → a → LR(0), LR(1), SLR, CLR, LALR.
3. S → a | ab → SLR, CLR, LALR
4. S → SS | a → Ambiguous

5. S → a | ε → SLR, CLR, LALR

6. S → aSa | ε → ×



* Every LR(0) grammar is SLR, LALR, CLR and Unambiguous.
* Every SLR grammar is LALR, CLR, and Unambiguous
⋮
* Every non CLR grammar is not LALR, not SLR, not LR(0)
* Every non SLR is non LR(0)

(Same type relations for all)

Note: * Every LL(1) CFG is LR(1) CFG
* No relation b/w LL(1) and SLR/LALR/LR(0)
* If LL(1) CFG is free from null rules then always SLR(1)
* If LL(1) without Unit rules then it is always LALR(1)

CLR vs LALR

I) If CFG is CLR(1) and after merging States of CLR to make LALR then RR conflict possible in LALR.

$A \to a., \{t_1\}$ $B \to a., \{t_2\}$	&	$A \to a., \{t_2\}$ $B \to a., \{t_4\}$
S_1 No RR conflict.		S_2 No RR conflict

$A \to a., \{t_1, t_2\}$ $B \to a., \{t_2, t_4\}$
S_{12}

RR conflict possible in LALR.

II) If CFG is CLR(1) and after merging States of CLR to make LALR then SR conflict never possible in LALR.

$A \to a.b, L_1$ $B \to a., L_2$	&	$A \to a.b, L_3$ $B \to a., L_4$

No SR conflict in CLR

$b \notin L_2$

No SR conflict in CLR

$b \notin L_4$

$b \notin L_2$ and $b \notin L_4$

$\Downarrow$

$b \notin L_2 \cup L_4$

$A \to a.b, L_1 \cup L_3$ $B \to a., L_2 \cup L_4$

No SR conflict in LALR

Note: Today many compilers built based on LALR. If any SR conflict occurs, then it performs shift action and proceeds for further passing.

- LALR is more popular parser
- CLR is more powerful parser.

Table Construction:

LL(1) Table:	LR Table
$\Downarrow$ entries	$\Downarrow$ entries
I. Production (Rule) II. blank	I. Shift entry II. Goto entry III. Reduced entry IV. Acceptance entry V. Blank entry.

Shift, reduced, accept → Action Table

Goto → Goto Table.

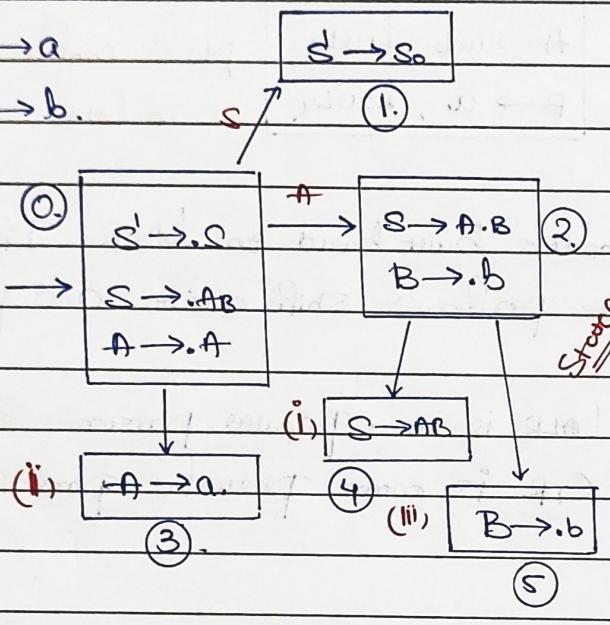
Goto entries.

Size of Table:

LL(1) Table	LR Table
→ $\|N\| * (\|T\|+1)$	→ No. of rows x no. of columns
→ No. of non terminals x (No. of terminals + 1).	Action Table: No. of States * (no. of terminals +1) Goto Table: No. of States * (no. of non terminals).

LR(0) Table Construction : I Shift entry
II Goto entry
III Reduced entry.
IV. Accept entry.

i $S \to AB$
ii $A \to a$
iii $B \to b.$



	Action			Goto		
LR(0)	a	b	$	S	A	B
0	S_3			1	2	
1			Accept			
2		S_5				4.
3	R_{ii}	R_{ii}	R_{ii}			
4	R_i	R_i	R_i			
5	R_{iii}	R_{iii}	R_{iii}.			

Shade

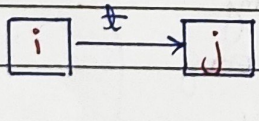
Reduced Entry for LR(0):

→ We should look at every reduced item.

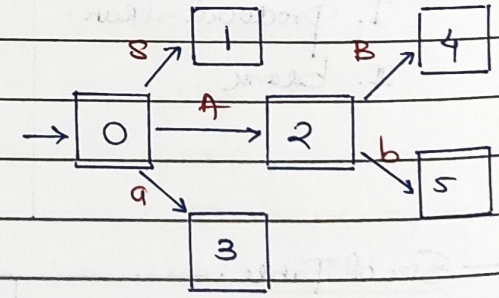
(ii) $A \to a$

	← Action →			← Goto →
3	R_{ii}	R_{ii}	R_{ii}	

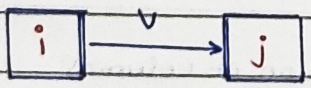
Shift Entry:



	t
i	Sj.

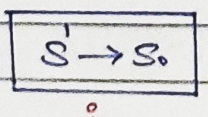


State (Goto) Entry:



	V
i	j.

Acceptance Entry:



	$
i	Accept.

Note:

I. No. of shift entries = No. of terminal transitions in DFA

II. No. of state entries = No. of non-terminal transitions in DFA.

LR(0) Table & SLR(1) Table

→ Shift entries are same

→ Goto entries are same

→ Only reduced entries may differ

Note:

I. No. of shift entries in LR(0) = No. of shift entries in SLR(1)

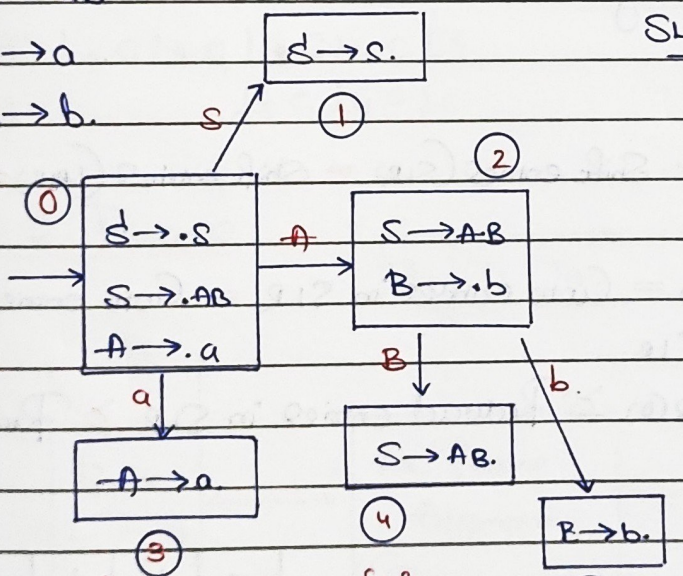
II. No. of Goto entries in LR(0) = No. of goto entries in SLR(1)

III. No of reduced entries in LR(0) ≥ No. of reduced entries in SLR(1)

i. $S \to AB$

ii. $A \to a$

iii. $B \to b.$

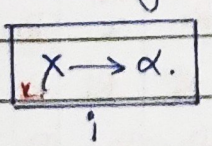


SLR(1)	Actions			Gotos		
	a	b	$	S	A	B
0	S_3			1	2	
1			Accept			
2		S_5				4
3		R_{ii}				
4			R_i			
5			R_{iii}			

Follow(A) = {b}. Follow(S) = {$}.

Follow(B) = {$}.

Reduced entry in SLR(1)



	t_1	t_2
i	R_k	R_k

Follow(X) = {t_1, t_2}

CLR and LALR Reduced Entries

→ Look at every reduced item

(k) $X \to \alpha.$, {t_1, t_3}

State i

	t_1	t_3
i	R_k	R_k

CLR and LALR Table

(for previous data).

LALR(1) / CLR(1)	a	b	$	S	A	B
0	S_3			1	2	
1			Accept			
2		S_5				4
3		R_{ii}				
4			R_i			
5			R_{iii}			

(Action: a, b, $ — Goto: S, A, B; States: 0–5)

LR(0) Table | SLR(1) | LALR | CLR

I. Shift entries in LR(0), SLR, and LALR are always same
II. Goto entries in LR(0), SLR, and LALR are always same
III. Reduced entries may differ.

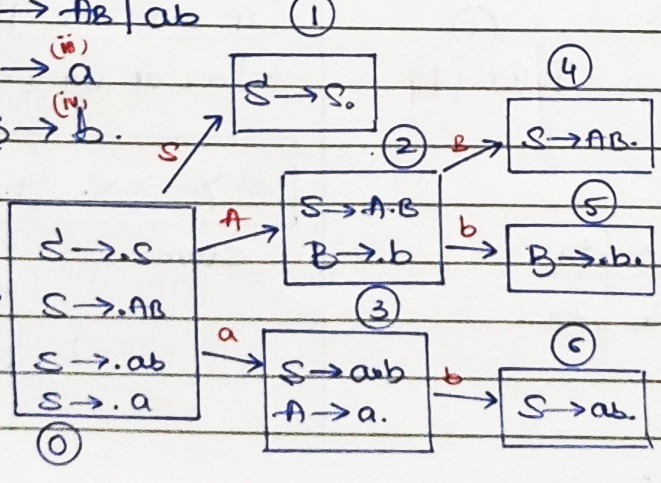
Note:

* Shift entries (LR(0)) = Shift entries (SLR) = Shift entries (LALR) ≤ Shift entries in CLR(1)
* Goto entries in LR(0) = Goto entries in SLR = Goto entries in LALR ≤ Goto entries in CLR.
* Reduced entries in LR(0) ≥ Reduced entries in SLR ≥ Reduced entries in LALR.

eg. $S \to \overset{(I)}{A}B \mid \overset{(II)}{a}b$

$A \to \overset{(III)}{a}$

$B \to \overset{(IV)}{b}$.



	a	b	$	S	A	B
0	S_3			1	2	
1			Accept			
2		S_5				4
3	R_{iii}	R_{ii} / S_6	R_{iii}			
4	R_i	R_i	R_i			
5	R_{iv}	R_{iv}	R_{iv}			
6	R_{ii}	R_{ii}	R_{ii}			

classmate
Date
Page

I. Relations: Less powerful ... More powerful

Parsers: LR(0) < SLR < LALR < CLR
LR(1)

LL(1) < LR(1).

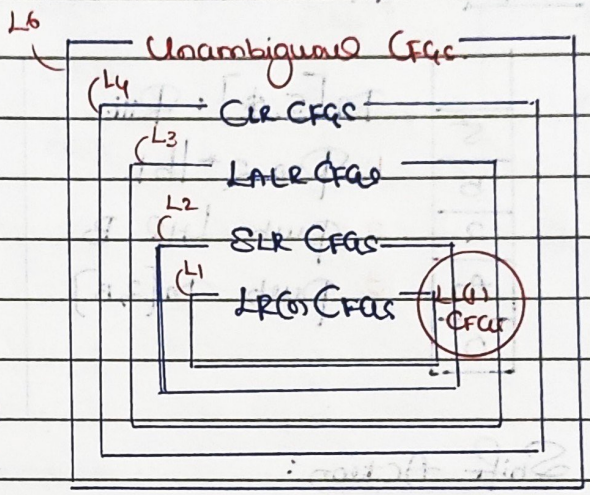
II. Grammars:

LL(1) CFG ⇓ is.

LR(0) CFG ⇒ is SLR(1) CFG ⇒ is LALR CFG ⇒ is CLR CFG ⇒ is Unambiguous

III. Classes

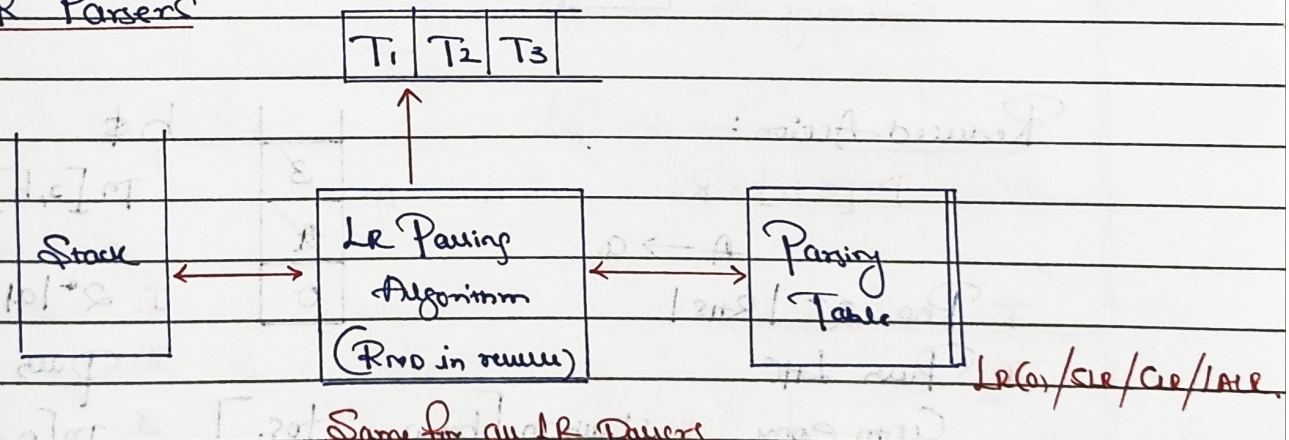
L_1 = Set of LR(0) CFGs
L_2 = " SLR "
L_3 = " LALR "
L_4 = " CLR "
L_5 = " LL(1) "
L_6 = " Unambiguous CFGs



I $L_1 \subset L_2 \subset L_3 \subset L_4 \subset L_6$

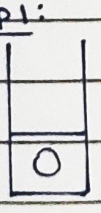
II $L_5 \subset L_4 \subset L_6$

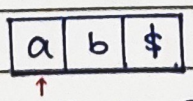
LR Parsers



LR Parsing Algorithm. I/p: ab

Step1:

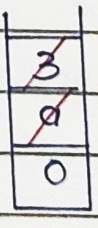




Tr[0,a] = S3.

1. Push a and increment Pointer.
2. push 3.

Step2: b $



Tr[3,b] : Rii

1. 2*1 al pop
2. push LHS
3. push Tr[0,A]

M	a	b	$	S	A	B
0	S_3			1	2	
1			Accept			
2		S_5				4
3		R_{ii}				
4			R_i			
5			R_{iii}			

(i) $S \to AB$ (ii) $A \to a$
(iii) $B \to b$.

Step 2: b$

2
A
0

M[2,b] : S_5
1. Push b
2. Push 5

Step 4: $

~~5~~
~~b~~
2
A
0

M[5,$] : R_{iii}.
1. Pop $2*|b|$.
2. Push LHS B
3. Push M[2,B]

Step 6: $

1
5
0

M[1,$] : Accept.

Shift Action:

	a
0	S3.

Shift entry.

3.
a
0

a
↑

I Push input a onto stack & increment pointer.
II. Push next state number

Reduced Action:

M[3,b] : R_{ii}
$A \to a$.

I Pop $2*|RHS|$
II. Push LHS
Goto entry III. Push to [below tos. / tos, (LHS)]
(below LHS)

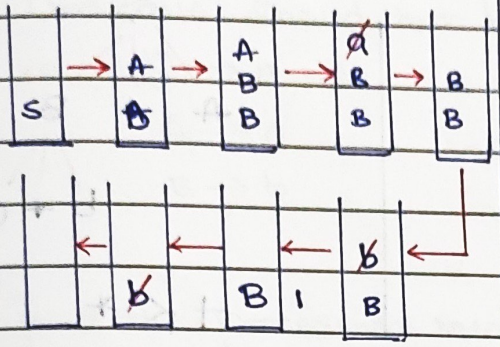
A
~~3~~
~~a~~
0

b$
↑
M[2,b] : $A \to a$.
I. $2*|a|$
II. Push A
III. M[0,A] push
Goto entry

S → AB w = abb

A → AB | a

B → b



LMD : SAABB	RMD : SBABA	Reverse of RMD.
S	S	S
↓ AB	↓ AB	↑ AB
↓ ABB	↓ Ab	↑ Ab
↓ aBB	↓ ABb	↑ ABb
↓ abB	↓ Abb	↑ Abb
↓ abb	↓ abb	↑ abb

Precedence Relations :

I. Higher

II. Lower

III. Equal → L to R associativity.
R to L associativity.

Higher ↓ Lower.

*, /, % (Same precedence)	→ left to right
+, −	→ left to right

① S → A+B

A → a

B → b*c

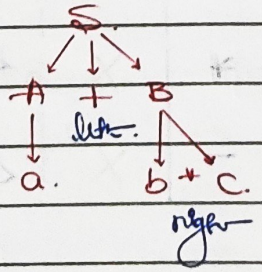
I. a+b+c

a + b * c

left right

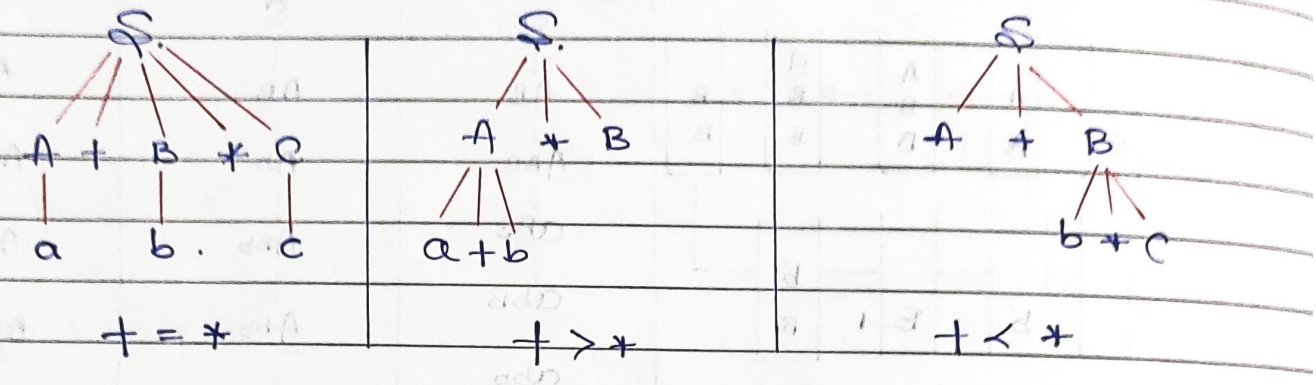
Right (input)

left (Stack)	+	*
+	−	<.
*	−	−

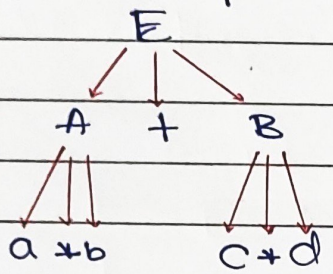


a + b * c

Left Right



Q1. Find the precedence using (relating) Parse tree.

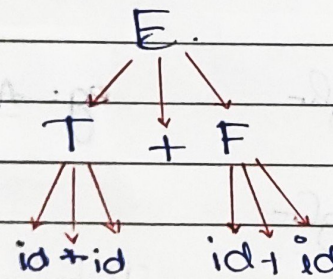


→ + is lower
→ * is higher.

	+	*
+	N.R	<
*	>	N.R

(No relation)

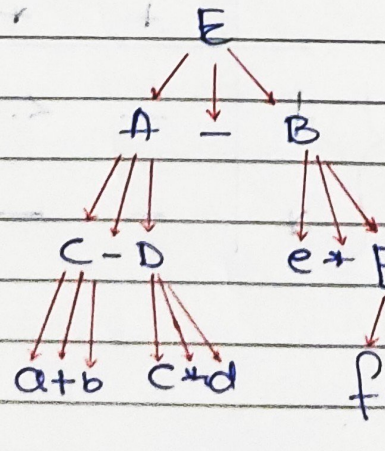
Q2.



→ + is lower than *
→ * is higher than +
→ + is right associative.

	+	*
+	<	No Relation
*	>	No Relation

Q3.



	+	*	−
+	X	X	>
*	X	<	>
−	X	<	>

Operator Grammar:

→ It may be ambiguous or unambiguous

→ It is CFG which do not contain consecutive non-terminals and also should not have null value.

eg: $S \rightarrow A+B$

$A \rightarrow a$

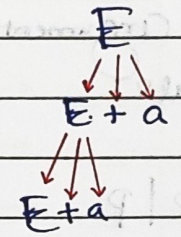
$B \rightarrow b$

Q4. Find the precedence relations using CFG.

$E \rightarrow E+a \mid a$

Left recursion.

'+' is left associative.

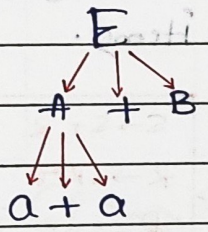


	Right
+	>

Q5. $E \rightarrow A+B$

$A \rightarrow a+a$

$B \rightarrow a$



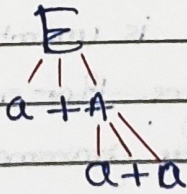
'+' is left associative.

Q6 $E \rightarrow a+E \mid a$

'+' is Right associative.

Q7. $E \rightarrow a+A$

$A \rightarrow a+a$



'+' is Right associative.

Q8. $E \rightarrow E+T \mid a$

$T \rightarrow F*T \mid b$

$F \rightarrow c$

→ * is higher

→ = is lower

→ + is left associative

→ * is Right associative.

Q9. $E \to E+E \mid E-E \mid id$

→ + is highest

→ − is lowest

→ + is left-to-right associative.

→ − is right-to-left associative.

PYQs

Q1 Which of the following statements is true?

Ans. LR(1) parsing is sufficient for deterministic context-free languages.

Q2. Consider the argumented grammar $\{+, *, (,), id\}$ as the set of terminals?

$S' \to S$

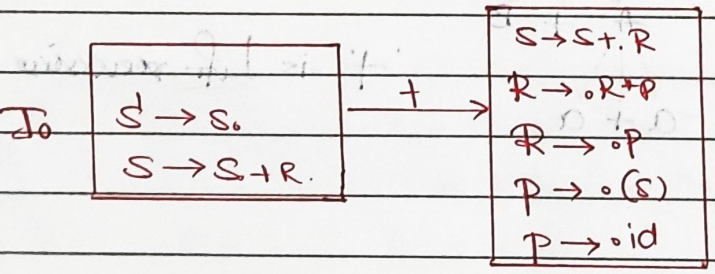
$S \to S+R \mid P$

$R \to R*P \mid P$

$P \to (S) \mid id$

If I_0 is set of two LR(0) items $\{[S' \to .S], [S \to S.+R]\}$, then goto (closure($I_0$), +) contains exactly 5 items.

Ans.



Q3. Consider the following statements:

S_1: Every SLR(1) grammar is unambiguous but there are certain unambiguous grammars that are not SLR(1).

S_2: For any context-free grammar, there is a parser that takes atmost $O(n^3)$ time to parse a string of length n.

Which of the following options is correct?

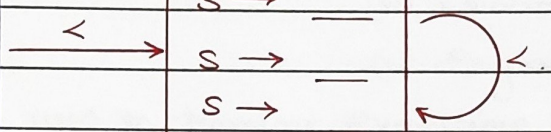
Ans. S_1 and S_2 both are true.

CYK Algorithm

- Bottom up parsing
- Dynamic Programming
- $O(n^3)$ algorithm
- Membership algorithm for CFG
- It verifies given string generated given CFG or not.

Q4. Consider the following augmented grammar with {*, @, <, >, a, b, c,} as the set of terminals.

$S' \rightarrow S$	$S \rightarrow .<S>$
$S \rightarrow S*cS$	$S \rightarrow$ ___
$S \rightarrow SS$	$S \rightarrow$ ___
$S \rightarrow S@$	$S \rightarrow$ ___
$S \rightarrow <S>$	$S \rightarrow$ ___
$S \rightarrow a$	$S \rightarrow <.S>$
$S \rightarrow b$	$S \rightarrow$ ___
$S \rightarrow c$	$S \rightarrow$ ___



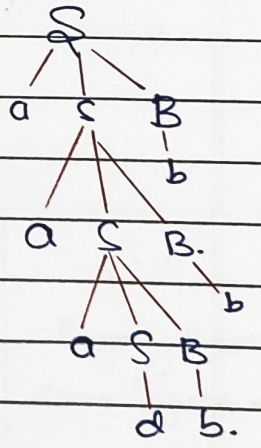
Let I_0 = CLOSURE ($\{S' \rightarrow .S\}$). The number of items in the set GOTO (GOTO (I_0, <), <) is 8.

Q5. Consider the following grammar:

$S \rightarrow aSB \mid d$

$B \rightarrow b$

The number of reduction steps taken by a bottom-up parser while accepting the string is 7.



No. of reductions steps in BUP (reverse of RMD)

=

No of steps in RMD

Q6. Which of the following kinds of derivations is used by LR parsers?

Ans Rightmost in reverse.

Q7. Which of the following statements about parser is/are correct?

I. Canonical LR is more powerful than SLR.

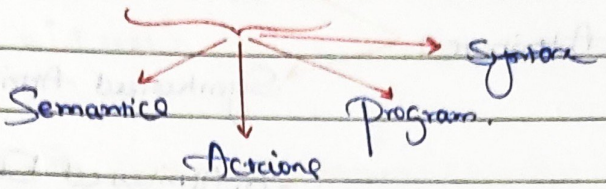
II. SLR is more powerful than LALR

III. SLR is more powerful than Canonical LR.

Ans. Only I is True.

Syntax Directed Translations.

SDT = Syntax + Translation
= CFG + Translation.



SDT Applications:

→ It can be used to perform: (i) Semantic analysis
(ii) Syntax tree generate
(iii) Intermediate code generation
(iv) Generating Parse tree
(v) Any meaningful activity.

→ It can be used to translate expressions:
(Infix/prefix/postfix → infix/prefix/postfix).

→ It can be used to translate numbers: (Binary/decimal/other → to any other/decimal/binary).

→ It can evaluate expressions.

$E \rightarrow E+E$ {Translations}

$E \rightarrow a.$ {Translations}.

CFG + Translations = SDT.

Lexical	Syntax.	Semantic	SDT
Tokens	Structures: → declarative Syntax → if, if else Syntax → loop Syntax → function Syntax → expression Syntax	Type checking → in expression → in function → declare before use of variable	Anything that has some logical information in syntax. More powerful than Compiler.

Syntax Directed Translations :

1. Attributes → Inherited Attribute
   → Synthesized Attribute
2. Definations → L-Attributed SDT
   → S-attributed SDT
3. Evaluations

eg: $S \rightarrow a$ { $S.x = a.val$ }

CFG, Translation, Attribute (x, val)

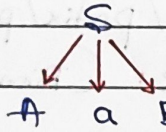
eg: $S \rightarrow AaB$

S: Parent of A, a, B

A: Child of S, but left sibling of a and B

a: Child of S, right sibling of A, left sibling of B

B: Child of S, right sibling of A and a.

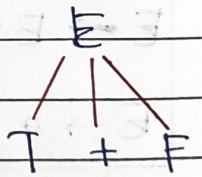


Productions tree

Q1. $E \rightarrow T + F$ { $E.x = T.y * F.z$ }.

Where is the computations?

Ans "E", it is computed depending on its children, T and F.



Attributes :

1. Inherited Attribute : Computation depends on Parent/Siblings

eg: $S \rightarrow AB$ { $A.x = B.y$ }.

x is inherited attribute

2. Synthesized Attribute : Computation depends upon Children

eg: $E \rightarrow a$ { $E.x = a.val$ }.

Identify the type of attribute:

① (i) $E \rightarrow E_1 + E_2$ { $E.x = E_1.x + E_2.x$ }.

(ii) $E \rightarrow id$ { $E.x = id.val$ }.

depends on child

→ In Rule (i), x is Synthesized attribute

→ In Rule (ii), x is Synthesized attribute

→ In SDT, x is Synthesized attribute.

② I. $S \rightarrow D\ L$; { $L.type = D.type$ }

II. $D \rightarrow int$ { $D.type = int$ }

III. $L \rightarrow L_1, id$ { $L_1.type = L.type$ }

IV. $L \rightarrow id$ { }.

→ In Rule I, type is inherited attribute

→ In Rule II, type is Synthesized attribute

→ In Rule III, type is inherited attribute

→ In whole SDT, type is neither inherited nor Synthesized.

③ $S \rightarrow S_1 S_2$ { $S.count = S_1.count + S_2.count$ }

$S \rightarrow (S_1)$ { $S.count = S_1.count + 1$ }

$S \rightarrow \varepsilon$ { }

→ Count is Synthesized attribute in above SDT

④ $S \rightarrow Aa$ { $S.x = A.x$; $A.y = Sx$ }

$A \rightarrow b$ { $A.x = 100$; $A.y = 1000$ }.

→ x is Synthesized attribute

→ y is inherited attribute

⑤ $E \rightarrow T + F$ { $E.x = T.y$; $F.y = E.x + 2$; $E.y = F.y - 1$ }

$T \rightarrow id$ { $T.x = 10$; $T.y = id.val$ }

$F \rightarrow id$ { $F.x = id.val$; $F.y = 20$ }.

→ x is Synthesized attribute

→ y is neither inherited nor Synthesized attribute

L-Attributed SDT	S-Attributed SDT
* Computation depends on parent / left sibling / children.	* Computation depends on only children.
* Translation can be placed anywhere in production	* Translation should be at the end of production.
* Evaluation depends upon translations order from left to right.	* Evaluation depends on bottom up parsing. (Reverse of RMD) (Non-terminal Order)

Note: * Every S-attributed grammar is L-attributed.

* L-attributed SDT may or may not be S-attributed.

Identify the type of attribute:

① (i) $E \rightarrow E_1 + E_2$ { $E.x = E_1.x + E_2.x$ }

(ii) $E \rightarrow id$ { $E.x = id.val$ }.

→ The above SDT is both L and S Attributed.

② $S \rightarrow D\,L$; { $L.type = D.type$ } (depends on left sibling)

$D \rightarrow int$ { $D.type = int$ }

$L \rightarrow L_1$, id { $L_1.type = L.type$ }

$L \rightarrow$ id { } (depends on parent.)

→ The given SDT is L-attributed but not S-attributed

③ $S \rightarrow S_1 S_2$ { $S.count = S_1.count + S_2.count$ }

$S \rightarrow (S_1)$ { $S.count = S_1.count + 1$ }

$S \rightarrow \varepsilon$ { }

Above SDT is both 'L' and 'S' Attributed grammar.

MCQ.

A. L attributed

B. S attributed

C. L attributed but not S

D. NONE.

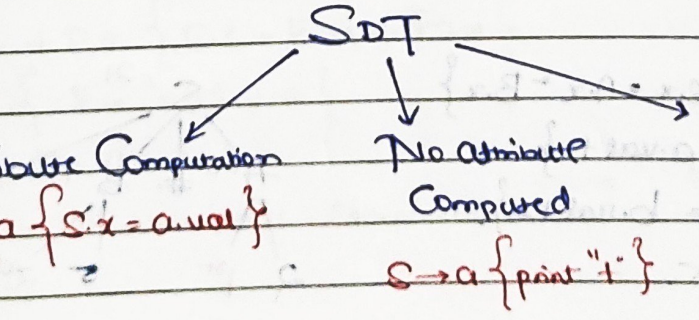
(4.) $S \rightarrow Aa \{ S.x = A.x ; A.y = S.x \}$
$S \rightarrow b \{ A.x = 100 ; A.y = 1000 \}$.
→ SDT is not S-attributed
→ SDT is L-attributed.

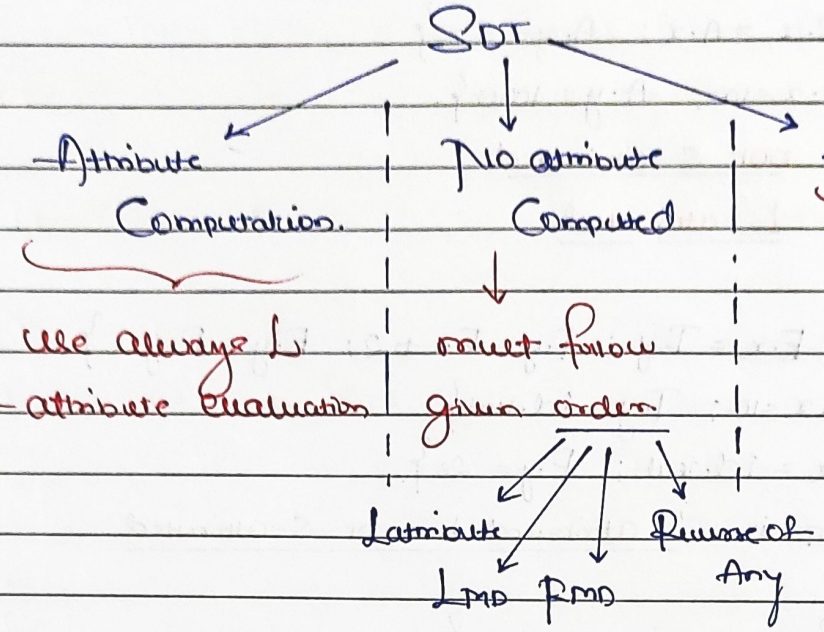
(5.) $E \rightarrow T+F \{ E.x = T.y ; F.y = E.x + 2 ; E.y = F.y - 1 \}$
$T \rightarrow id \{ T.x = 10 ; T.y = id.val \}$
$F \rightarrow id \{ F.x = id.val ; F.y = 20 \}$.
→ Above SDT is L attributed but not S-attributed.

(6.) $S \rightarrow S_1 S_2 \{ S_1.x = S_2.x + S.x \}$
$S \rightarrow a \{ S.x = a.value \}$.
→ Above SDT is neither S-attributed nor L-attributed.

(7.) $S \rightarrow \{ S.x = a.val \}$
$S \rightarrow b \{ S.x = b.val \}$
→ Above SDT is L-attributed but not S-attributed.

L-attributed SDT	S-attributed SDT
Evaluation (Translation) : All translations evaluated from left-to-right. * (Topological order) * (In order) * (DFS, left to right)	Evaluation (Translation) : Evaluation on non-terminals depends on source of Prod. * (Bottom up parsing) * (LR parsing) * (SR parsing).





① $S \rightarrow A \# B \{ S.x = A.x + B.x \}$

$A \rightarrow a \{ A.x = a.val \}$

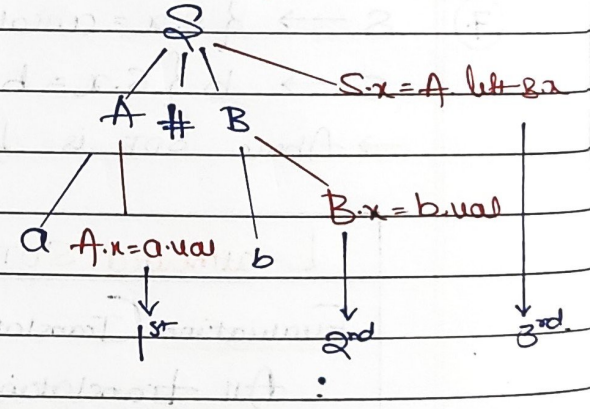
$B \rightarrow b \{ B.x = b.val \}$.

Input : 10 # 3.

Method I : Using L attributed evaluation.

$\Rightarrow 13$

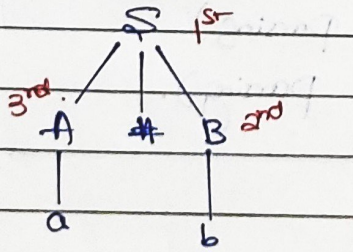
(Decorated PT)
Annotated parse tree.



Method II : Using S-attributed evaluation (Reverse of RMD)

$\Rightarrow 13$

RMD.



RMD numbering : 1st 2nd 3rd : S B A

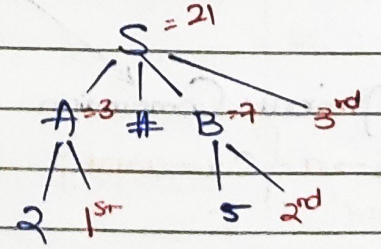
Reverse of RMD : 3rd 2nd 1st : A B S

② $S \rightarrow A \# B \{ S.x = A.x * B.x \}$

$A \rightarrow \{ A.x = a.val + 1 \}$

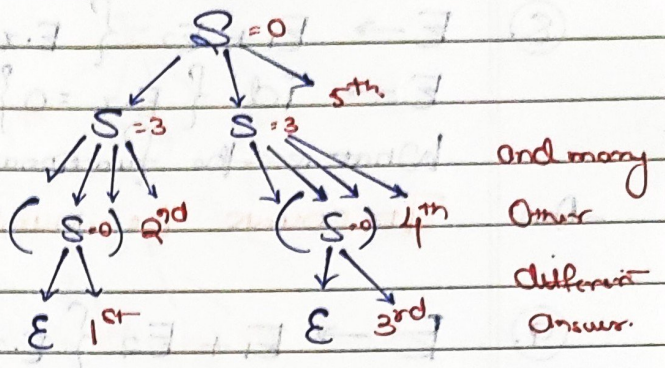
$B \rightarrow \{ B.x = b.val + 2 \}$.

Input : 2 # 5.

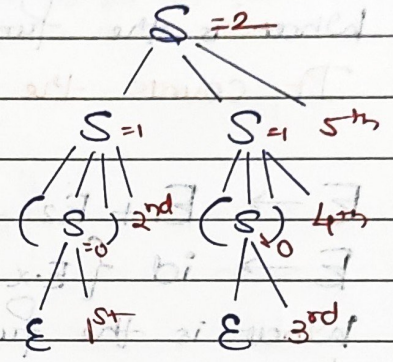


classmate
Date
Page

(3) $S \rightarrow S_1 S_2 \{ S.x = S_1.x - S_2.x \}$
$S \rightarrow (S_1) \{ S.x = 3 + S_1.x \}$
$S \rightarrow \varepsilon \{ S.x = 0 \}$.
Input = ()()
Multiple Answers
∴ Ambiguity.



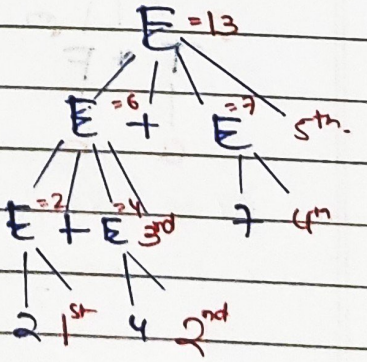
(4) $S \rightarrow S_1 S_2 \{ S.x = S_1.x + S_2.x \}$
$S \rightarrow (S_1) \{ S.x = 1 + S_1.x \}$
$S \rightarrow \varepsilon \{ S.x = 0 \}$.
Input: ()()
⇒ 2



(5) $S \rightarrow S_1 S_2 \{ S.x = S_1.x + S_2.x \}$
$S \rightarrow (S_1) \{ S.x = 1 + S_1.x \}$
$S \rightarrow \varepsilon \{ S.x = 0 \}$
What is the functionality of given SDT?
Ans It counts no. of balanced parenthesis at root.

(6) $S \rightarrow S_1 S_2 \{ S.x = S_1.x + S_2.x \}$
$S \rightarrow (S_1) \{ S.x = 2 + S_1.x \}$
$S \rightarrow \varepsilon \{ S.x = 0 \}$.
What is the functionality of the given SDT?
Ans It counts number of (parenthesis) terminals in the input (length of input).

(7) $E \rightarrow E_1 + E_2 \{ E.x = E_1.x + E_2.x \}$
$E \rightarrow id \{ E.x = id.val \}$
Input: 2+4+7
What is attribute value computed at root?
Ans 13



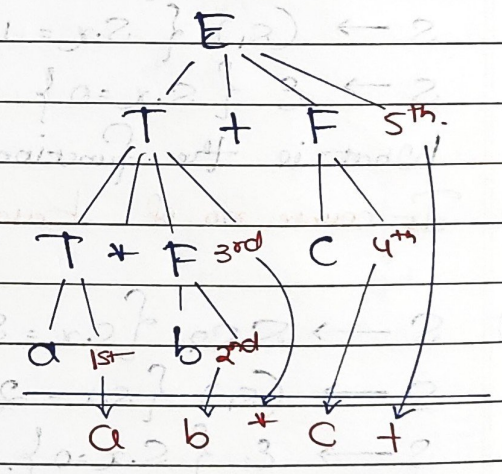
classmate
Date
Page

(8) $E \rightarrow E_1 + E_2 \{ E.x = E_1.x + E_2.x + 1 \}$
$E \rightarrow id \{ E.x = 0 \}$.
What is the functionality of the given SDT?
Ans It counts the number of Operators.

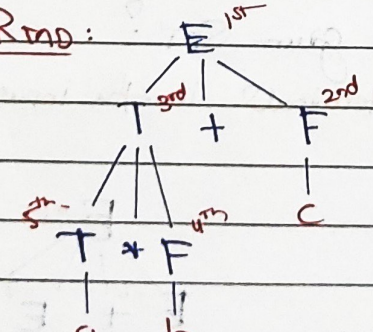
(9) $E \rightarrow E_1 + E_2 \{ E.x = E_1.x + E_2.x \}$
$E \rightarrow id \{ E.x = 1 \}$
What is the functionality of the given SDT?
Ans It counts the number of Operands.

(10) $E \rightarrow E_1 + E_2 \{ E.x = E_1.x + E_2.x + 1 \}$
$E \rightarrow id \{ E.x = 1 \}$
What is the functionality?
Ans Length of input (no. of terminals, Operands & Operators).

(11) $E \rightarrow T + F \{ print + \}$
$E \rightarrow id \{ print\ id.val \}$
$T \rightarrow T * F \{ print * \}$
$T \rightarrow id \{ print\ id.val \}$
$F \rightarrow id \{ print\ id.val \}$.
Input: a*b+c.
Output: ab*c+

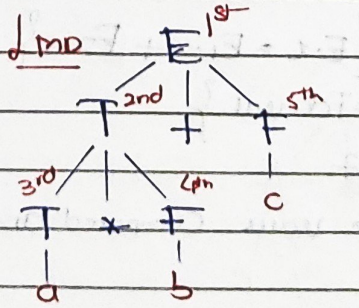


Rmd:



Reverse of Rmd: ab*c+ (5th 4th 3rd 2nd 1st)

Lmd:



Lmd: +*abc (1st 2nd 3rd 4th 5th)

(12) $E \rightarrow \{print +\}\ T+F$

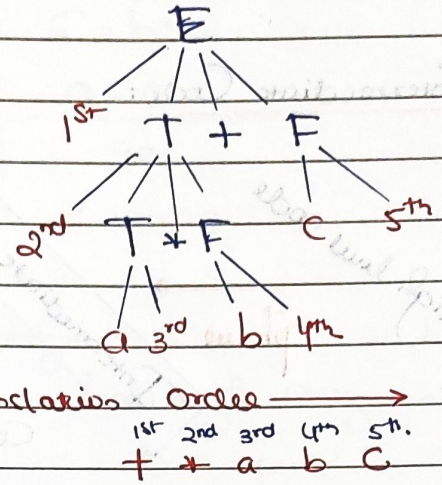
$E \rightarrow id\ \{print\ id.val\}$

$T \rightarrow \{print *\}\ T*F$

$T \rightarrow id\ \{print\ id.val\}$

$F \rightarrow id\ \{print\ id.val\}$.

Input: a+b+c.



Translation Order →

1st 2nd 3rd 4th 5th.

\+ * a b c

(i) What is the output?

Ans +*abc

(ii) What is output using bottom up parsing?

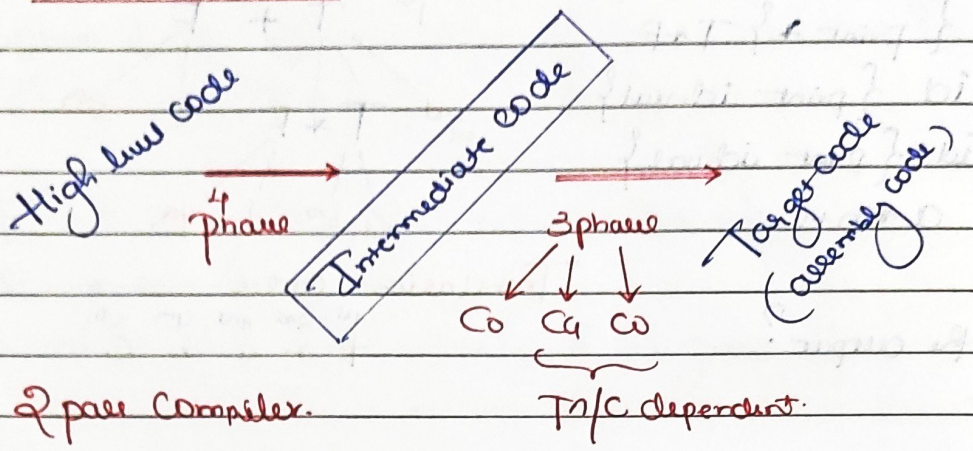
Same as Q11. bottom up parsing.

Note:

1. L-attribute evaluation depends on translation order. (If translation position changes then O/p may change.)
2. Top and Bup evaluations depends on non-terminal Order. (If translation position changes then O/p will change remain same as previous).

Intermediate Code & Code Optimization

Intermediate Code:



(Lexically, Syntactically and Semantically) Analysed High level Code → ICG → Intermediate Code

Intermediate Code Representations (Re):

1. Linear Form: → (i) postfix form
   (ii) 3AC / TAC / Three adder code
   (iii) SSA Code (Static Single assignment)

2. Non Linear Form → (i) Syntax tree
   (ii) Directed Acyclic graph
   (iii) Control flow graph

eg: $x = a + b * c$

Postfix code: $x = a\ b\ c\ *\ +$

$\Rightarrow x\ \boxed{a\ \boxed{b\ c\ *}\ +}$

Three-Address Code:

→ Every Statement contains almost 3 addresses (variables)

$t_1 = b*c$	$b = b*c$	$c = b*c$
$x = a + t_1$	$b = a+b$	$a = a+c$
3AC	3AC	3AC
5 variable	3 variable	3 variable.

SSA Code : (Static Single-Assignment)

↳ It is 3AC but every variable has single assignment.

$t_1 = b*c$ (first time defined)

$x = a + t_1$ (reading only)

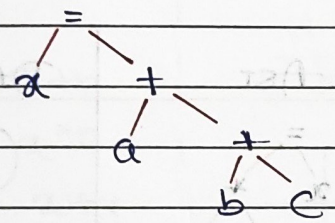
3AC — also SSA code

a, x, b, t_1, c, } Single definition

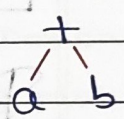
Syntax tree : (Abstract Syntax Tree)

Leaf : Operand

Non leaf : Operator



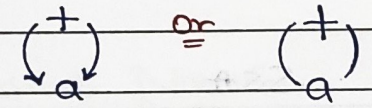
→ a+b



Directed Acyclic Graph (DAG)

↳ It eliminates common sub expressions.

a+b



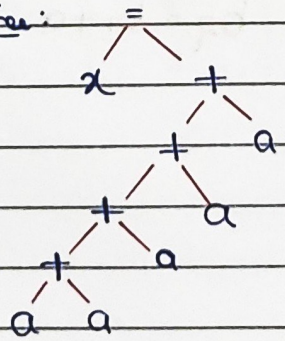
DAG DAG

2 nodes

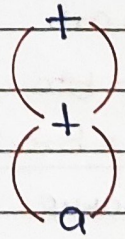
2 edges

eg: $x = a+a+a+a$

Syntax Tree:



DAG



3 nodes

4 edges

classmate
Date
Page

1. $x = a+b-b+c$

Postfix

$x = ((a+b)-b)+c$

$ab+$

$ab+b-$

$ab+b-c+$

$xab+b-c+=$

3AC

With min no. of variables

$x = a+b-b+c$

$x = a+c$

$\Downarrow$

$a = a+c$

2 variable

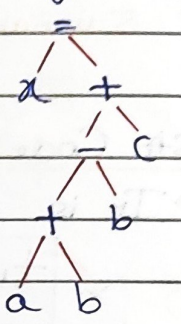
SSA code

min. no. of variables

$a_1 = a+c$

3 variables

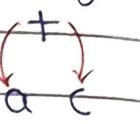
Syntax Tree



DAG

min nodes

min edges



2. $x = (a+b)*(a+b)$

Postfix

$xab+ab+*=$

3AC

$a = a+b$

$a = a*a$

2 variable

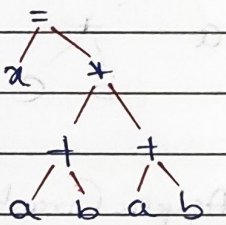
SSA

$a_1 = a+b$

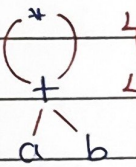
$a_2 = a_1 * a_1$

4 variable

AST



DAG



4 nodes

4 edges

3. $x = a*b+a+b*c;$

3AC

Find min no of variables

$x = (a*b)+a+(b*c)$

$= (b*a)+(b*c)+a$

$= b*(a+c)+a$

$c = a+b$

$b = b*c$

$b = b+a$

3 variable

SSA

Find min no of variables

$c_1 = a+c$

$b_1 = b*c_1$

$b_2 = b_1+a$

} 6 variables

4. $x = a + b * c - b$

$c_1 = b * c$	$c_1 = b * c$
$c_2 = a + c_1$	$c_2 = a + c_1$
$c_3 = c_2 - b$	$c_3 = c_2 - b$
3AC	SSA
3 variables	6 variables.

Three Address Code Notations:

* Triple notation
* Quadruple notation
* Indirect triple notation.

Advantage: less space
Disadvantage: computation takes much time

eg: $x = a + b$
$y = x * c$
$z = x + y$
$w = z$.

Triple Notation.

	operator	operand 1	operand 2
1000	+	a	b
1010	*	1000	c
1020	+	1000	1010
1030	=	1020	

1000 : (+, a, b)
1010 : (*, 1000, c)
1020 : (+, 1000, 1010)
1030 : (=, 1020)

Quadruple Notation:

	Operator	Operand1	Operand2	result
1000	+	a	b	x
1015	*	x	c	y
1030	+	x	y	z
1045	=	z		w.

Advantage: Takes less time to compute
Disadvantage: more space.

eg:

$x = a + b$
$y = x * c$
$z = x + y$
$w = z$

Indirect Triple Notation

	operator	operand1	operand2
1000	+	a	b
1010	*	6000	c
1020	+	6000	7000
1030	=	8000	

Indirect	Actual Address
6000	1000
7000	1010
8000	1020
9000	1030

Three Address Code for 'if else' Statement:-

```
if (x<y):
    z = x
else:
    z = y;
z = z * z;
```

```
_t0 = x<y;
If z _t0 Goto _L0;
z = x;
Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

Three Address Code for 'while' Statement:

```
while (x<y){
    x = x*2;
}
y = x;
```

```
_L0:
    _t0 = x<y;
    If z _t0 Goto _L1;
    x = x*2;
    Goto _L0;
_L1:
    y = x;
```

Three Address Code for Array

Assume declaration : A[n1, n2]

3AC for A[i,j] is :

$t_1 = n_2 * i$

$t_2 = t_1 \times j$

$t_3 = t_2 * w$

$t_4 =$ Base address

$t_5 = t_4[t_3]$

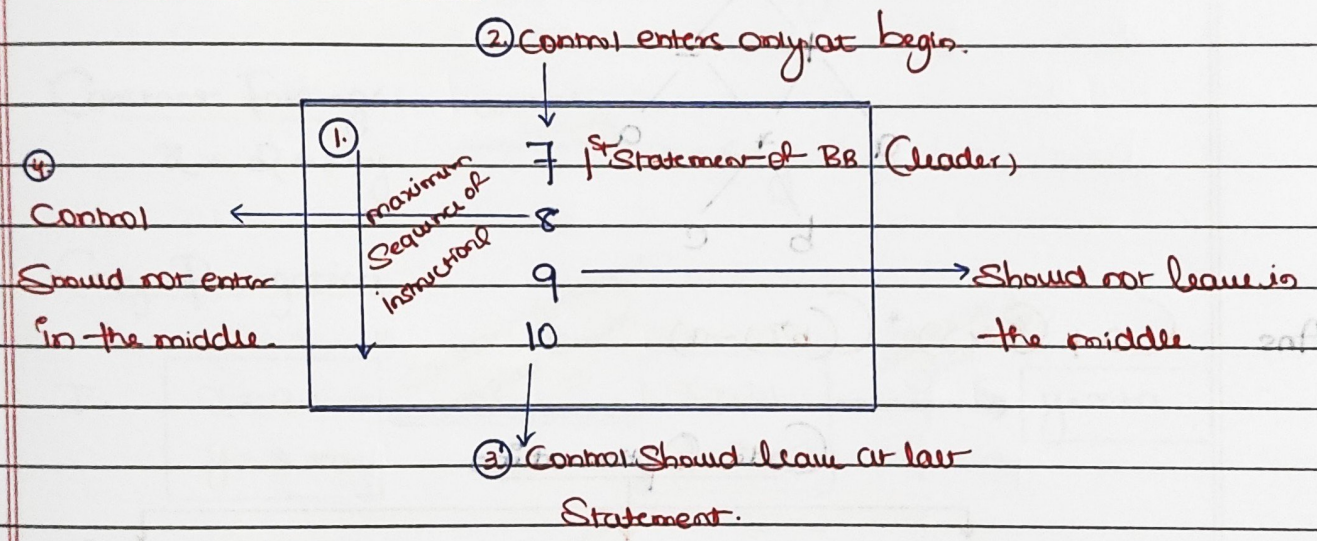
$\longrightarrow A[i,j] =$ Base address $+ (n_2 * i + j) * w$

Here, n_2 is size of each row and w is size of each element.

Control Flow Graph:

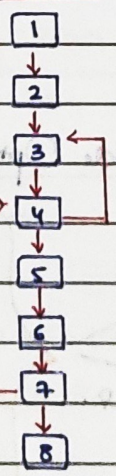
* It comprises of nodes and edges
* It is collection of basic blocks and control.
* It represents flow of program execution using basic blocks.

What is basic block (BB)?

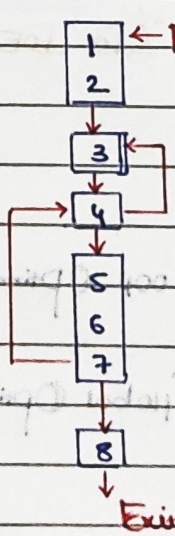



```
1. a = a+b
2. y = a+y
3. z = x+y
4. if (z>d) goto 3
5. z = x-y
6. z = z+a
7. if (z>1000) goto 4
8. print z.
```

Flow



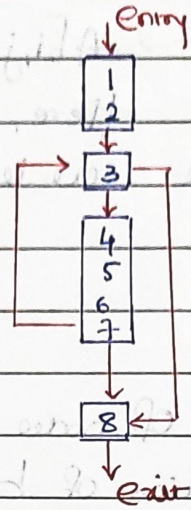
Control Flow Graph



5 BBs
7 nodes
8 edges.

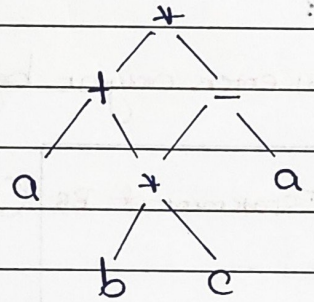
Control Flow Graph

```
s=0
i=1
L1: if i > n goto L2
    t = i*i
    s = s+t
    i = i+1
    goto L1
L2: return s
```



No of B.Bs = 4
No of nodes = 6
No of edges = 6

Q1 Find equivalent expression for the following Syntax tree.



Ans $(a+(b*c))*((b*c)-a)$

Code Optimizations

- High Level — In IDE
- Intermediate Level — 5th phase
- Target Level (Assembly) — 7th phase

To compiler

Code Optimizations

- Loop Optimizations
- Global Optimizations
- Statement level
- Block level (Basic block, peephole)
- Loop level
- Function level
- Program Level

Code Optimization Techniques:

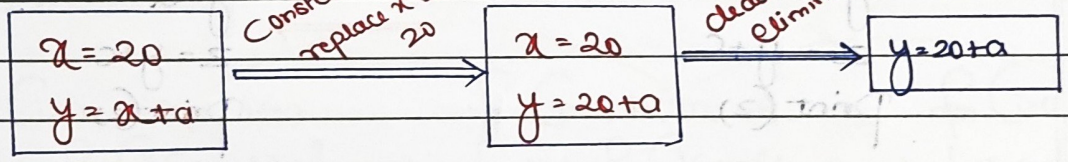
1. Constant folding
2. Copy Propogation → Constant propogation
   → Variable propogation
3. Common Sub Expression elimination
4. Strength reduction
5. Algebraic Simplification
6. Dead code elimination
7. Loop Optimizations → Code motion
   → induction variables elimination
   → loop merging
   → loop unrolling

① Constant Folding:

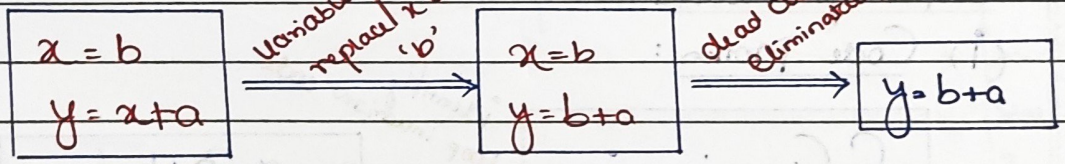
$x = 2 * 3 + y$ ⟹ (Constant folding) $x = 6 + y$

② Copy Propogation

I.



II.



③ Common Sub-Expression Elimination.

→ DAG can be used

$x = (a+b) * (a+b) \Longrightarrow t_1 = a + b$

$x = t_1 * t_1$

④ Strength Reduction:

↳ It replaces costlier code with cheaper.

$x = a * 2 \longrightarrow x = a + a$

$x = a * 2 \longrightarrow x = a << 1$

multiplication is costlier

Cheaper

⑤ Algebraic Simplifications:

Cancellation law (i) $x = a + b - b + c \longrightarrow x = a + c$

Identity law (ii) $x = a + b * 1 \longrightarrow x = a + b$

Identity law (iii) $x = a + b + 0 \longrightarrow x = a + b$

Domination (iv) $x = a + b + c * 0 \longrightarrow x = a + b$

⑥ Dead Code Eliminations:

$x = a + b$ — dead code (useless)

```
y = a * b
z = y + c
print(z)
```

⟹

```
y = a * b
z = y + c
print(z)
```

⑦ Loop Optimizations:

(i) Code Motion:

```
for (i=0; i<n; i++)
{  x = a+b;
   y = x*i; }
```

identify loop invariant code

⟹

```
x = a+b
for (i=0; i<n; i++)
{  y = x*i; }
```

and move outside loop.

(ii) Induction Variable elimination:

```
k=0;
for(i=0, j=0; i<n; i++)
{  x=a+i;
   y=b*j;
   z=c+k;
   p=x*y+z;
   j=j+1;
   k=k+1; }
```

⟹

```
for(i=0; i<n; ++i)
{  x=a*i;
   y=b*i;
   z=c*i;
   p=x*y+z; }
```

(iii) Loop Merge

```
for(i=0; i<n; ++i)
{ A[i]=i+a; }
for(j=0; j<n; j++)
{ B[j]=j*b; }
```

⟹

```
for(i=0; i<n; ++i)
{ A[i]=i+a;
  B[i]=i*b; }
```

(iv) Loop Unrolling:

```
for(i=0; i<4n; i++)
{ printf("GATE"); }
```

⟹ (half)

```
for(i=0; i<2n; ++i)
{ printf("GATE");
  printf("GATE"); }
```

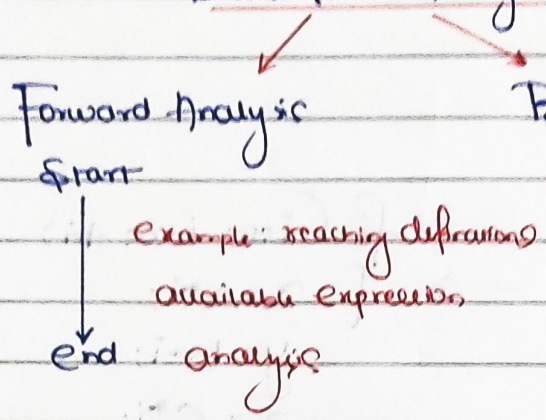
⟹ (double)

```
for(i=0; i<n; ++i)
{ printf("GATE");
  printf("GATE");
  printf("GATE");
  printf("GATE");
}
```

Data Flow Analysis

- To analyse Programs
- To understand each variable
- Lattices can be used to analyse data
- Control flow graph can help to analyse data

Data Flow Analysis



Live Variable Analysis [Liveness Analysis]

- What is Live variable?
- How to compute live variables at any Statement?
- Backward Analysis:
  - Compute GEN(USE) & KILL(DEF) for each
  - basic block
  - Compute IN and OUT for each BB.

Live Variable: x is live variable at Statement Si if:

I. There exists statement Sj that reads x
II. There exists path from Si to Sj
III. There is no assignment into x before Sj.

Find live variables at Statement 1, 2, 3, & 4

1. $x = a + b;$ ← x, y, a✓ 1→1, b✓ 1→1, c✓ 1→2 or 1→2→3→4

2. $y = x * c;$ ← x✓ 2→2 or 2→3, y, a, b, c✓ 2→2 or 2→3→4

3. $a = x + y;$ ← x✓ 3→3, y✓ 3→3, a, b, c✓ 3→4

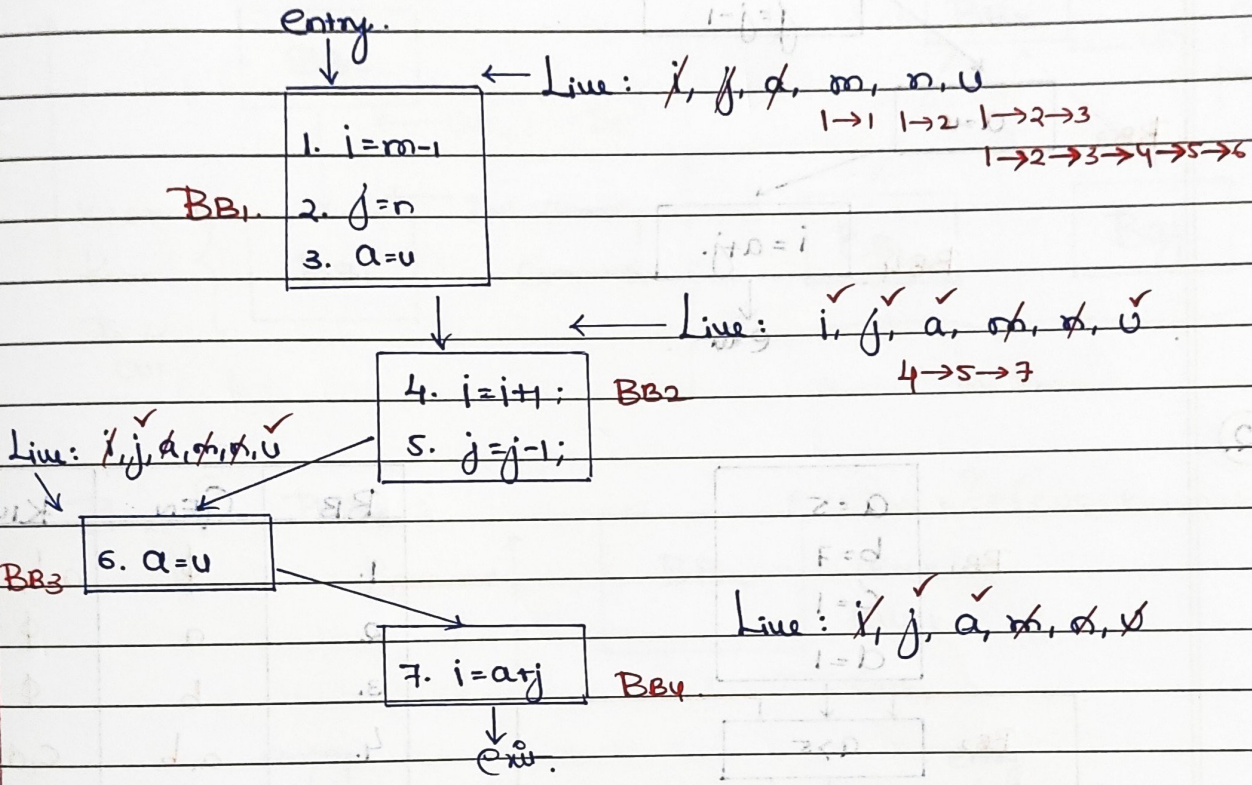
4. $b = a - c$ ← x, y, a✓, b, c✓

classmate
Date 29/9/23
Page

eg:

1. $a = a+b$ ← At 1: a ✓ (1→1), b ✓ (1→1, 1→2), i ✗
2. $b = a+b$ ← At 2: a ✓ (2→2, 2→3), b ✓ (2→2), i ✗
3. $c = b+a$ ← At 3: a ✓ (3→3), b ✓ (3→3), i ✗

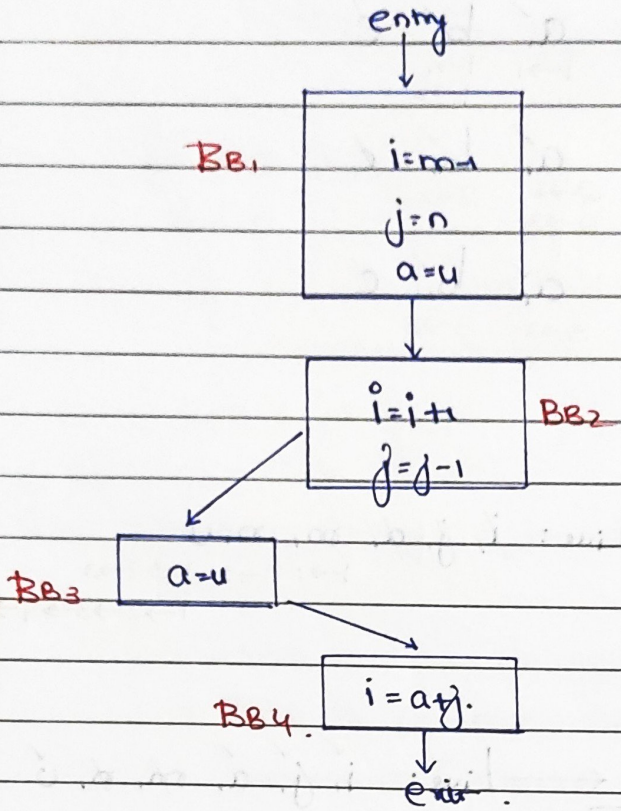
eg:



GEN (USE) Set	KILL (DEF) Set
BBK x = a+b y = x*c z = y-d GEN K = {a, b, c, d}. = {l | l is used at some statement in BBK and no definition before the statement}.	KILL K = {x, y, z}. = {v | v has definition in BBK}.

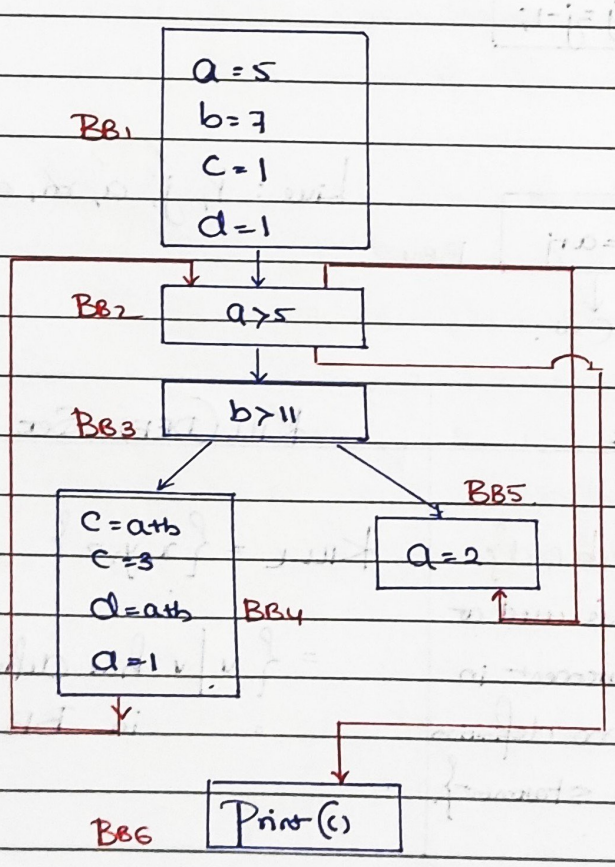
classmate
Date
Page

1. Compute GEN & KILL for each BB



	GEN Set	KILL Set
BB1	{m, n, u}	{i, j, a}
BB2	{i, j}	{i, j}
BB3	{u}	{a}
BB4	{a, j}	{i}

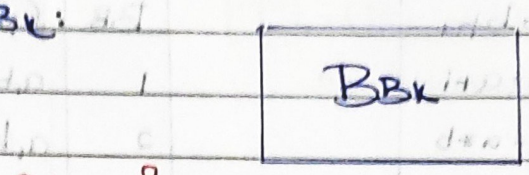
2.

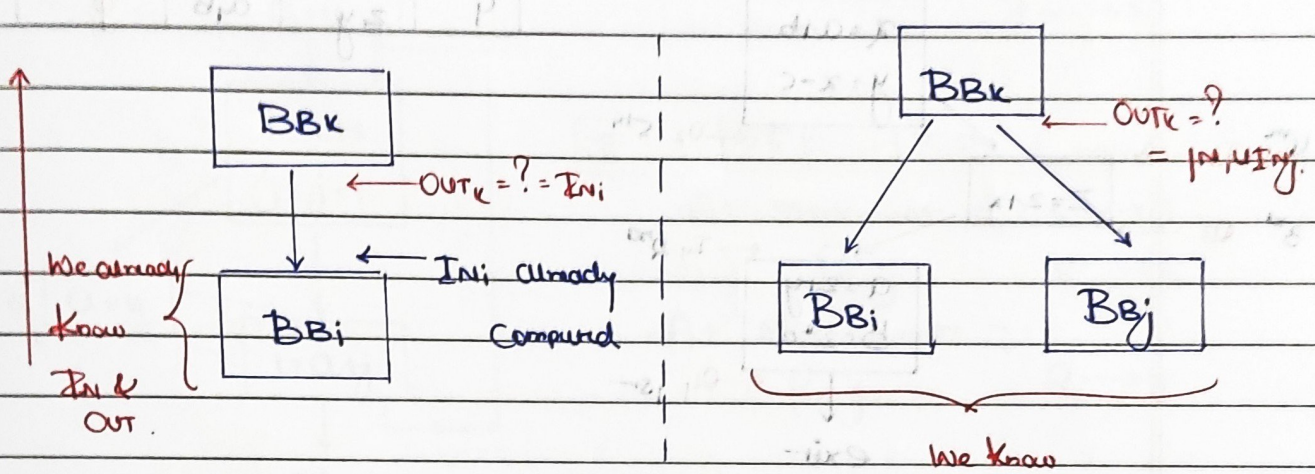


BB	GEN	KILL
1.	ϕ	a, b, c, d
2.	a	ϕ
3.	b	ϕ
4.	a, b	c, d, a
5.	ϕ	a
6.	c	ϕ

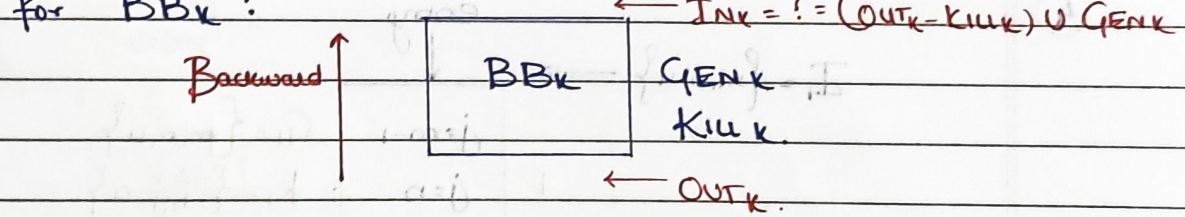
IN and OUT sets computation:

OUT$_K$ for BB$_K$:

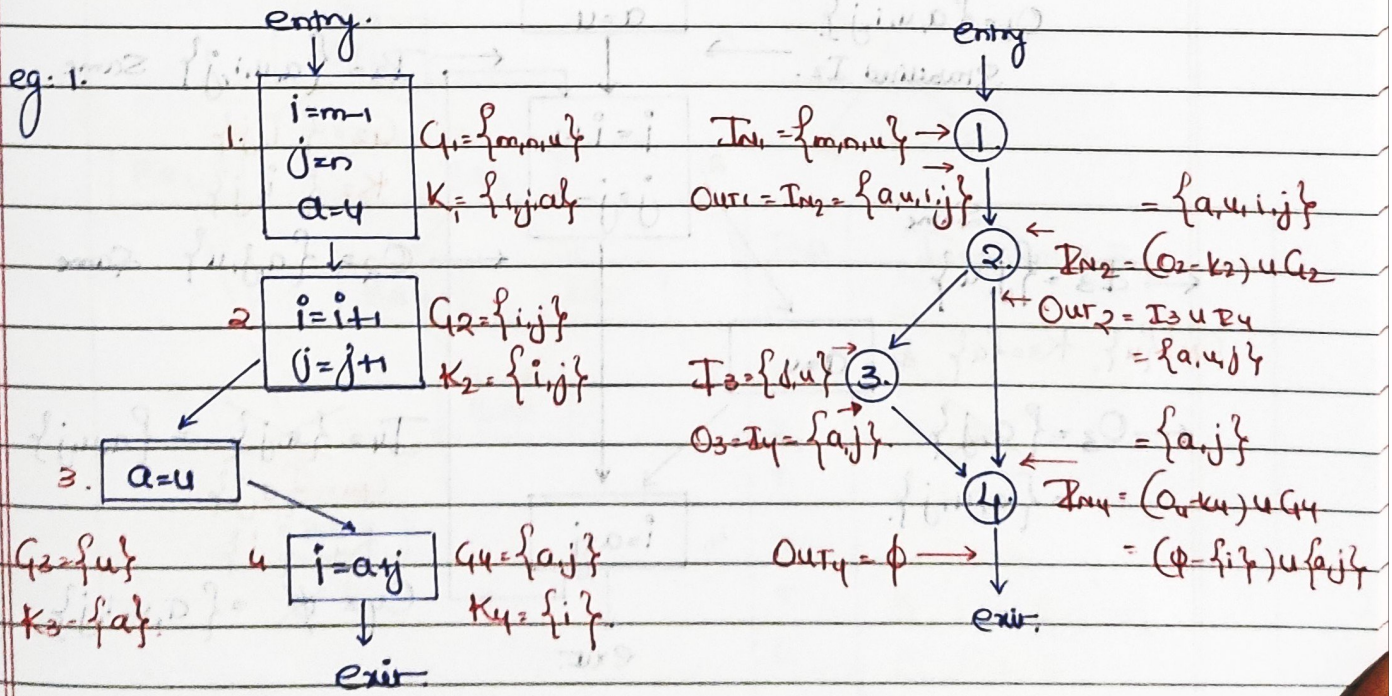




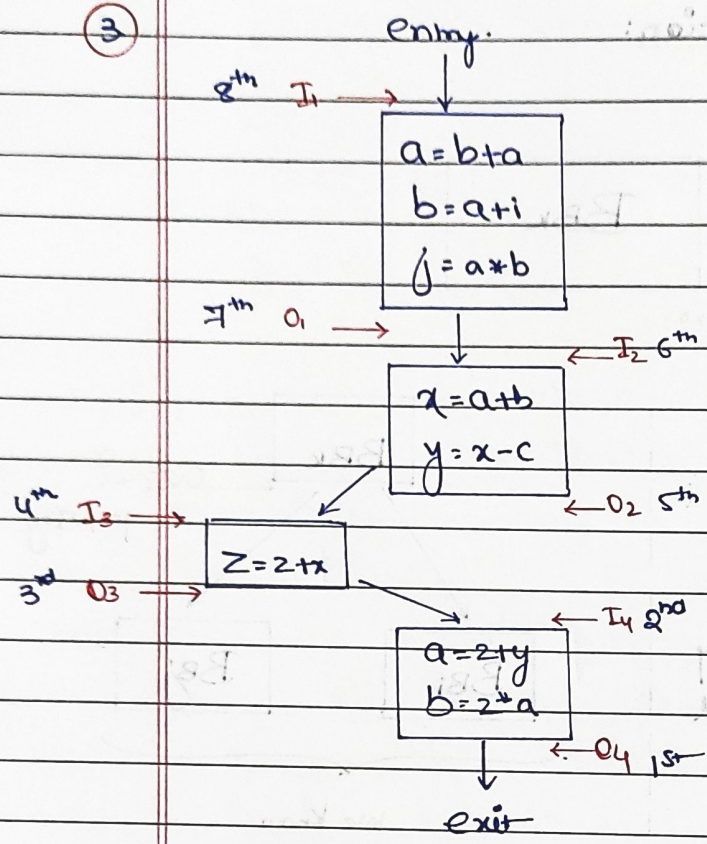
IN$_K$ for BB$_K$:



eg: 1.

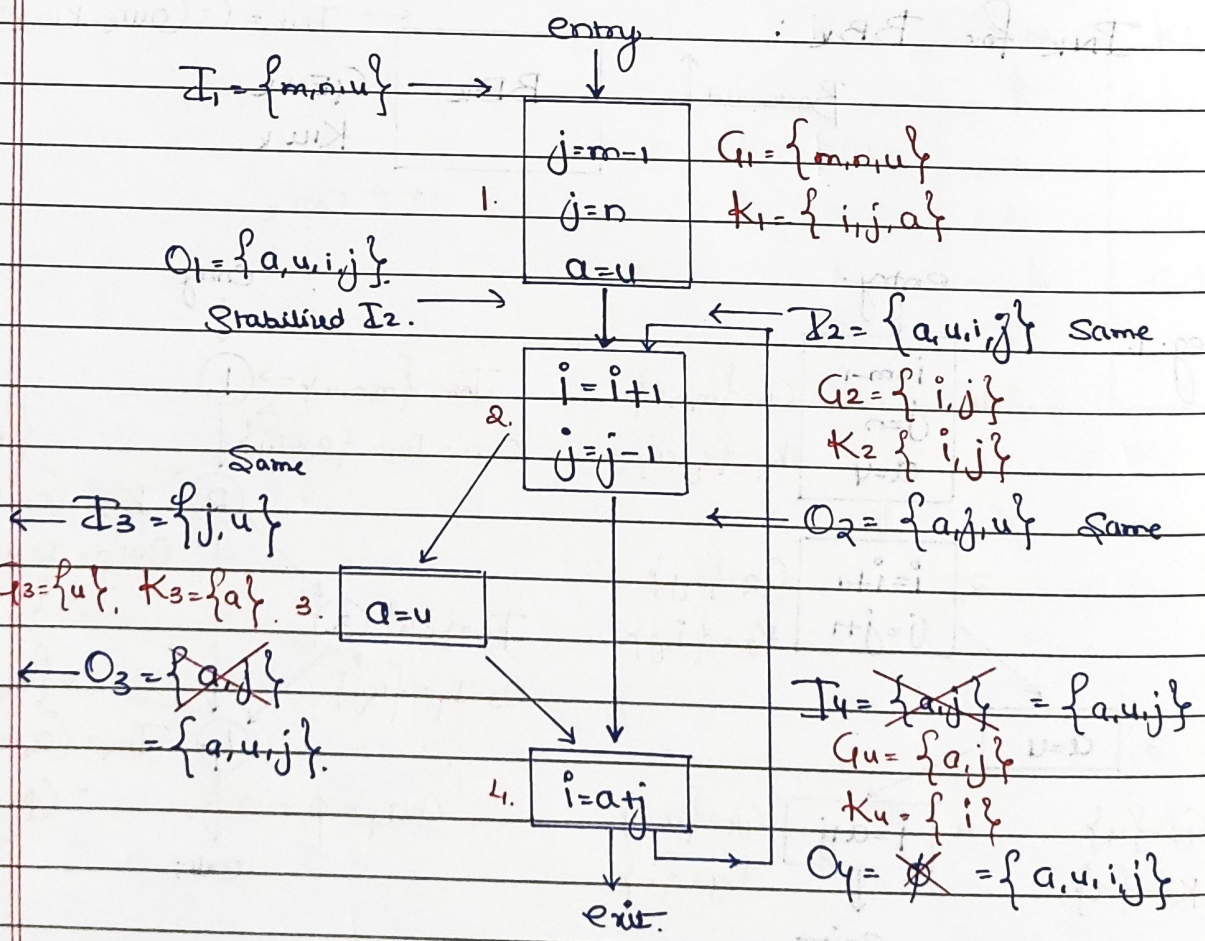


③

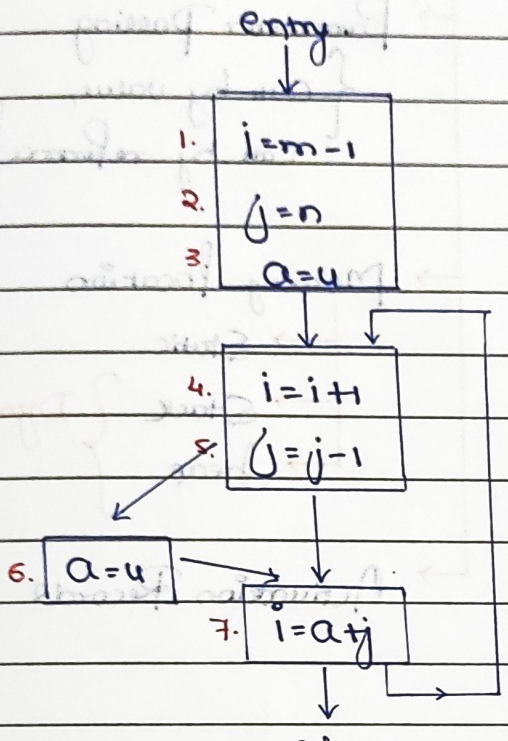


BB	GEN	KILL	OUT	IN
1	a,b,i	a,b,j	a,b,c,z	a,b,i,z
2	a,b,c	x,y	x,y,z	a,b,c,z
3	x,z	z	I_4=z,y	x,y,z
4	z,y	a,b	ϕ	z,y

④



Reaching Definitions



Source destination

I. USE - DEF Chains

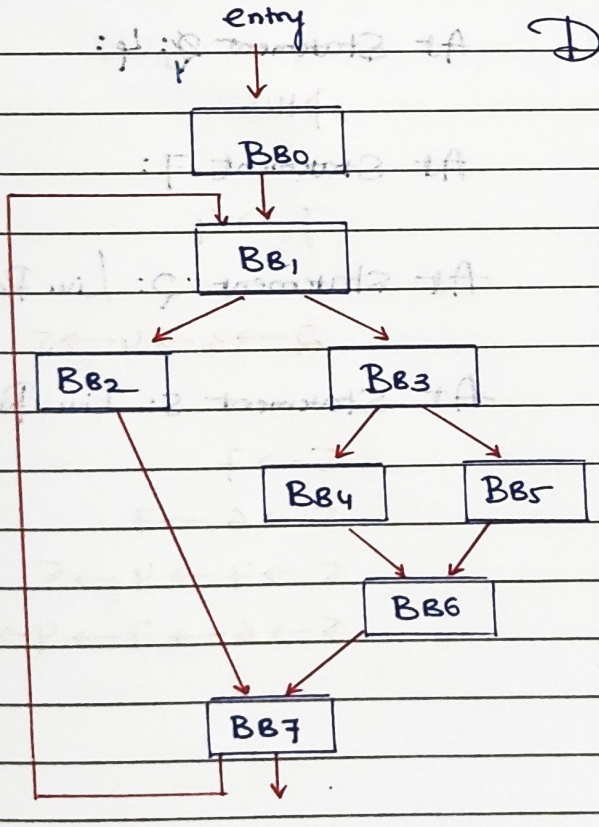
At statement 5: 5 → 2
5 → 5

At Statement 7:
7 → 5

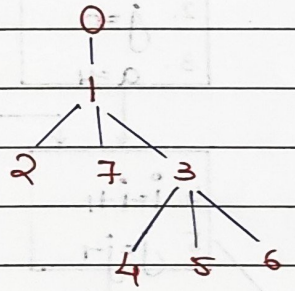
II. DEF - USE Chains

At Statement 5: 5 → 7
for j 5 → 5

At Statement 2:
for j. 2 → 5

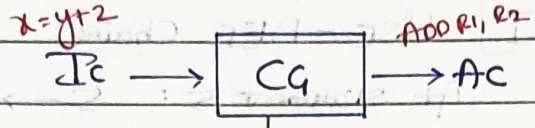


Dominator Tree



BB	Dominated By
0	0
1	0,1
2	0,1,2
3	0,1,3
4	0,1,3,4
5	0,1,3,5
6	0,1,3,6
7	0,1,7

Code Generation

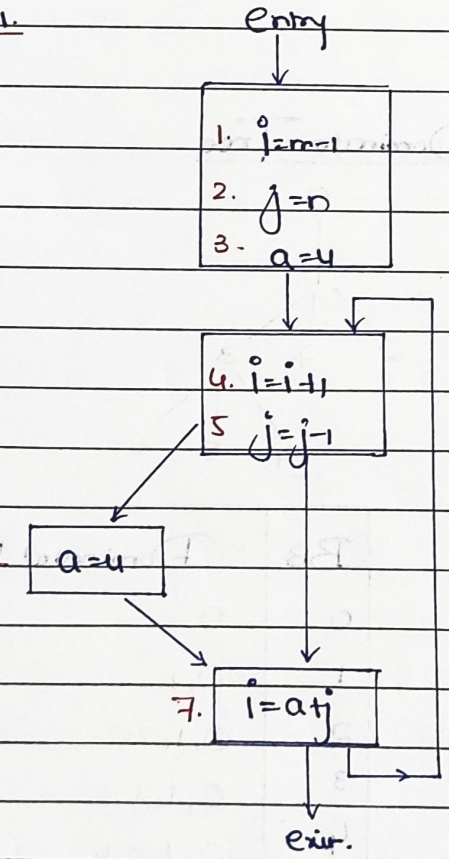


- Dependent on machine
- registers, addresses, instructions, addressing modes
- Register allocation
  - required program analysis
  - use graph coloring

Run Time Environment

- Parameter Passing { call by value, call by reference }
- Memory location
  - Static
  - Stack } Dynamic
  - Heap
- Activation Records

eg: 1.



At statement 1: Live Range (i) = ?

1 → 2 → 3 → 4

At Statement 4:

NIL

At Statement 7:

7 → 4

At Statement 2: Live Range (j) = ?

2 → 3 → 4 → 5

At Statement 5: Live Range (j) = ?

5 → 7

5 → 6 → 7

5 → 7 → 4 → 5

5 → 6 → 7 → 4 → 5

Note:

Static Memory Allocation → doesn't Support recursion

Stack memory allocation → Supports recursion