Compiler Design

1. Phases of Compiler
2. Lexical Analysis
3. Syntax Analysis ***
4. Syntax Directed Translation
5. Intermediate Code Generation **
6. Code Generation **
7. Runtime Environments

5M – 7M weightage

15 days
≤ 30 hours

Telegram Link:-
DEVA SIR PW

How to prepare Compiler Design?

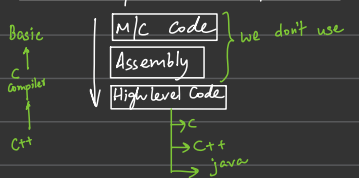
⇒ 1. Regular to Class
2. Notes ⇒ practice
3. GATE PYQs

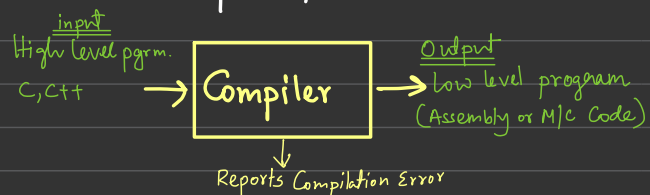
Introduction :-

What is Compiler?
Where we use Compiler?
How Compiler is Designed?

What is Compiler?

How compiler was Implemented?



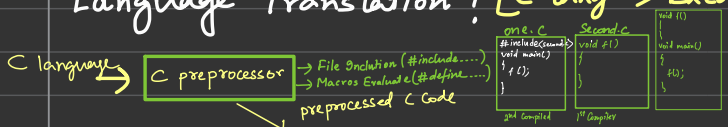


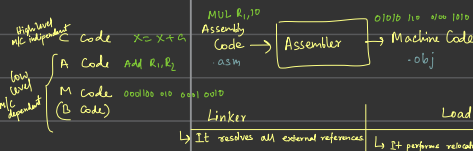
1) Compiler — our focus
2) interpreter
3) editors (IDE)
4) preprocessor
5) Assembler
6) Linker
7) loader
8) Spell checker
9) word/excel / powerpoint /...
10) others...

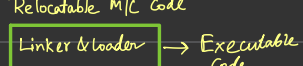
Where we use compiler?

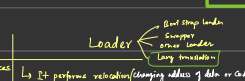
⇒ used in language translation

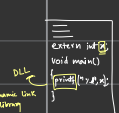
Language Translation ! [C lang → Executable Code]







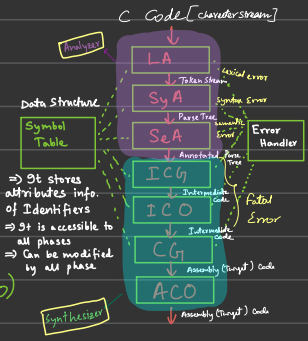




Phases of a Compiler:

7 phases

1) Lexical Analysis (LA)
2) Syntax Analysis (Sy A)
3) Semantic Analysis (SeA)
4) Intermediate Code Generation (ICG)
5) Intermediate Code Optimization (ICO)
6) Code Generation (CG)
7) Assembly (Target) Code optimization (ACO)



M/C dependent
- C Code } language dependent
- I Code } language independent

M/C independent
- A Code
- M Code
- E Code

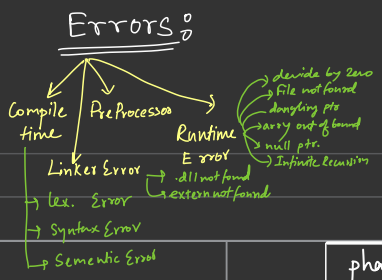
Data – unorganised Collection of values
v/s
Code – Instruction that handels data
v/s
Statement – Simple Sequence of Tokens
Compound (block, Function)
v/s
program – Set of statements
v/s
language – Set of program

Code

```
void main()
{
   int x=10;
   printf("%d",x);
}
```

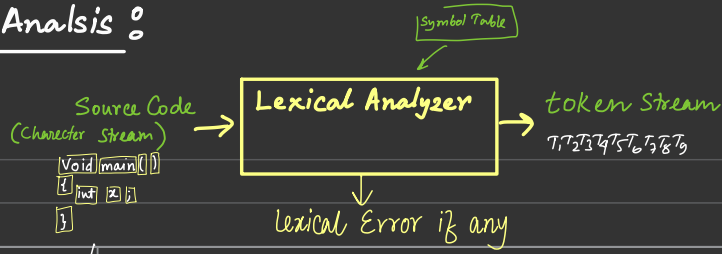
data = 10
Variable = x

C Language :-	C Compilers :-
⊘ High level Code	⊘ Machine Code [Low level executable code]
⊘ Machine independent	⊘ Machine Dependent
⊘ portable Code	⊘ Non-Portable (specific to machine)



phase	i/p	o/p	Functionality
LA	Char. Strm	Token strm	Token Recognition
SyA	Token strm	Parse Tree	Syntax Verifier
SeA	Parse Tree	Anotated Parse tree	Type checker
ICG	Anotated Parse tree	Intermidiate Code	Three Address Code generator
ICO	Intermidiate Code	Optimized Intermidiate Code	Three Address code optimization
ACG	Optimized Intermidiate Code	Assembly Code	Assembly Code (Target) genration
ACG	Assembly Code	optimized Assembly Code	Assembly Code (Target) optimization

Lexical Analsis :



Others name : Tokenizer / Lexical Analysis / Lexical Analyzer / Lexical Phase / 1st Phase of Compiler / Scanner / Linear Phase / Token Recogniser / Token Generator

Lex Tool

typedef int INT;	#define INT int
it is analyzed in Semantic analysis	analyzed/evaluated at preprocessing stage
INT x; (sent to int)	int x; (INT @preprocessing)

Functionality :

I) It scans whole program char. by char from start to end. [Top to Bottom, Left to Right]
II) It groups characters into tokens
III) It ignores white spaces and Comments
IV) It uses longest Prefix rule "Maximal Munch"
V) It also produces lexical errors if any.
VI) It creates symbol table and stores some attribute information of Identifiers. [Variable name, Function name]
VII) To design Lexical Analyzer. Reg. Exp / FA / Regular Grammer can be used.

void ✓

Symbol Table :→

I) It is a data Structure
II) It stores attribute information of identifiers
III) It is accessible to all phase of Compiler

```
void fun(int x)
{ int a;
}
```

lexeme	Token	remaining phases: Line	Type	offset
fun	ID1			
x	ID2			
a	ID3			

Lexeme : Smallest unit of (logic) program
Token : It uniquely represents Lexeme (Internal representation of lexeme)

I) Is "main" a keyword ?
⇒ no
↳ It is not a keyword
↳ It is a user defined function
↳ It is an identifier

I) Is "exit()" a keyword ?
⇒ no
it is a built-in function to terminate the program.

I) Is "sizeof()" a keyword ?
⇒ Yes
↳ it is a keyword, also a operator

Lexical Analysis

- lexeme & Token
- Types of Token
  - Keywords
  - operators
  - Identifiers
  - Literals/Constants
  - special tokens [punctuation]
- How to Find Token ?
- What are lexical errors ?

Keywords :
① int, float, double, char, long, short, void, signed, unsigned, volatile, const, static, register, extern, auto
② if, else, switch, case, default, break, continue, do, while, for, goto, return
③ struct, union, enum, typedef
④ sizeof

operators : (+, -, *, /, %, <<, >>, <=, <, >=, >, &&, ||, &, |, =, ^, ~, +=, -=, *=, /, ==, !=, ?, :)

Identifiers : Name given to variable or Function to represent address. (_, 0...9, A..Z, a...z)
» Should not be a keyword » should not start with number.

Constants/Literals : Int ← Dec, Hex, Octal
Real
Char
String

Special Tokens : ; ( ) { } :

Find the token in the following codes:

① `int x;`

② `int float;` → no lexical error by Syntax error

③ `int gate123;`

④ `int 123gate;` → Lexical error

⑤ `int x = "gate";` (String Constant)

⑥ `int x = 025;` (octal Constant)

⑦ `int x = 029;` → invalid sequence → lexical error

⑧ `int x = 0xabc;` (Hexadecimal Constant) — x abc ✓ | xABC ✗ | XABC ✓ | Xabc ✗

⑨ `int 23 = 45;` → syntax error

① If Comment is written between token then token will be separated

no of Token

`++`	= 1
`**`	= 2
`***`	= 3
`+=`	= 1
`=+`	= 2
`<<=`	= 1
`!=`	= 1

Note: `<<=` (11) `!=` (10)

⑩ `int /* Comment */ x;` @3 (ignore)

⑪ `in /* Comment */ t x;` @4

⑫ `in t x;` @4

⑬ `x = ++y;` @5

⑭ `x = +y;` @5

⑮ `x += y;` @4

⑯ `x = x+++y;` @7

⑰ `x = **y;` @6

⑱ `x = y /* Comment` → end rule is missing → lexical error

⑲ `x = y /* Comt /* Comment */;` (search for end) @4

⑳ `x = y Comment */;` @7 Token

㉑ `x = A[20];` @7

㉒ `x = fun(2,3);` @9

㉓ `x += printf("gate = %od", rank);` @9

㉔ `x = scanf("%d%d", &x, &y);` @11

㉕ `x = 5` @3

㉖ `x = 'a';` @4

㉗ `x = '\n';` @4

㉘ `x = 'abc';` ← lexical error (invalid sequence)

㉙ `x = "abc";`

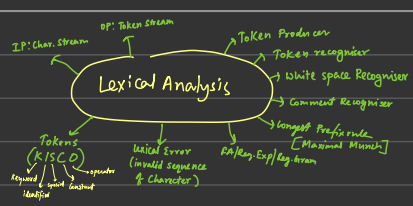
㉚ `x = y+++++z;`

㉛

T_1 = abc	input: adabcabeadadabc
T_2 = ad	o/p : T_2 T_1 T_3 T_2 T_2 T_1
T_3 = abe	

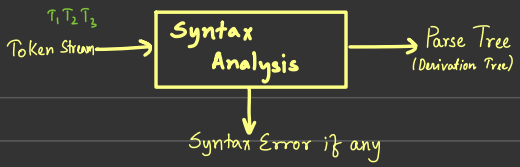
㉜ $T_1 = a^*b$ $T_2 = a^*bc$

Input: aabcaaabbbc $T_2 T_1 T_1 T_2$



Syntax Analysis :

Also known as:
- Syntax Analyzer
- Parser
- Syntax Verifier
- Parse Tree Generator
- Structure Analyzer



We will do
- Syntax Error finding
- CFGs [most suitable grammar in the world]
  - to Definition → to represent syntax of prog. lang.
  - Derivation & parsing
- *** Types of CFGs [Ambiguous and unambiguous]
  - ..... Removal of Left Recursion

Find the syntax error:

① `void main() { int x; }` — no error

② `void main() { int x,y; }` — no error

③ `void main() { int x = y; }` — no syntax error but semantic error [y is not declared]

④ `void main() { int x, int y; }` — syntax error

⑤ `void main() { int x+y; }` — syntax error

⑥ `void main() { int y = 2; int x = y; }` — no error

⑦ `void main() { int x,y; x=y=2; }` — no error

⑧ `void main() { int x, y; (x=2, y=2); }` — no error

⑨ `void main() { int x; int y; }` — no error

⑩ `void main() { if(2); }` — no error

⑪ `void main() { if(); }` — Syntax Error, Expression missing

⑫ `void main() { x y ; }` — no syntax error but semantic error

⑬ `void main() { while(2); }` — no compilation error

⑭ `void main() { while(); }` — Syntax Error, Exp required

⑮ `void main() { for(;); }` — syntax error

⑯ `void main() { for( ; ; ); }` — by default True; No Compilation Error

⑰ `void main() { typedef int x; x y; }` — no error

⑱ `void main() { for(1,2,3); }` — Syntax Error

⑲ `void main() { fro(1,2,3); }` — no syntax error [but either semantic error or linker error depending on Program]

⑳ `void main() { for(2;2;2); }` — no compilation error

㉑ `void main() { fro(2;2;2); }` — Syntax error

```
#include<stdio.h>
void main()
{
  f();
}
```
Linker error — Compiler assumes f() definition may exist in other file

```
void main()
{
  f();
}
```
Semantic error

Q. MCQ Consider the following ANSI C program

```
int main()
{ INTEGER x;
  return 0;
}
```

GATE 21 : 1 Marks

Which of the following phase in a seven phase compiler will throw an error?
a) Lexical analysis
b) syntax analysis
c) Symentic analysis
d) Machine dependent optimizer.

Q. Consider the following Statements: [G20: CS1M]

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

II. Compilers for programming language that support recursion necessarily need heap (stack) storage for memory allocation in the run-time environment.

III. Errors violating the condition 'any variable must be declared before its use' are detected during syntax analysis (semantic analysis).

Which of the following statements is/are TRUE?

No option is true.

Q. A lexical analyzer uses the following pattern to recognize three tokens T_1, T_2, T_3 over the alphabet {a,b,c}.

T_1: a?(b|c)*a = ε(a)(b+c)*a

T_2: b?(a|c)*b = ε(b)(a+c)*b

T_3: c?(b|a)*c = ε(c)(b+a)*c

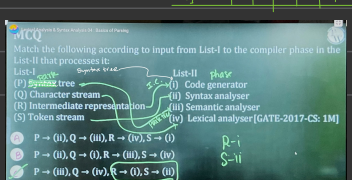
note that x? means 0 or 1 occurance of the symbol x. Note also that the lexical analyzer outputs the token that matches the longest possible prefix. If the string bbaacabc is processed by the analyzer which one of the following is the sequence of tokens is output?

(a) $T_1 T_2 T_3$
(b) $T_1 T_1 T_3$
(c) $T_2 T_1 T$
(d) $T_3 T_3$

	b	b	a	a	c	a	b	c
T_1	✓	✓	✓	✗		✓	✓	✓
T_2	✓	✓	✗	✓	✓	✓	✓	✗
T_3	✓	✓	✓	✓	✓	✓	✓	✓

(T_3 longest; T_2 next longest)

Lexical Analysis — Leftmost Derivation
Top Down Parsing — Type Checking
Semantic analysis — Regular Expression
Runtime Environment — Activation Records



(a) In a compiler, keywords of a language are recognized during
(i) Parsing of program (Syntax)
(ii) the code generation (Assembly Code generation)
(iii) the lexical analysis of program
(iv) dataflow analysis (Code optimization)

(b) Which data structure is used for managing information about variables and their attributes?
(i) Abstract syntax tree
(ii) Symbol table
(iii) Semantic stack
(iv) Parse Table

Context Free Grammer

	Context Free (Structure) (Syntax)	Context Sensitive (meaning) (Semantic)
✓ S+V+O → I am student		I am student ✓
✗ S+V+O → I are student		I are student ✗

Integer x ;
syntax Correct

Integer x ;
Symentic wrong

Index	Type
x	int
Id_2	int

Context Free Grammer

→ It can be used to represent syntax/structure of Prog. lang.

→ CFG = (V, T, P, S)
- V: Set of Variable or non-Terminal
- T: Set of Terminals
- P: set of production rules
- S: start symbol ∈ V

Each rule in P follows a rule $V \to (V \cup T)^*$ (LHS → RHS)

example: Identify CFG

① $S \to \varepsilon$ CFG
② $S \to Sab \mid a$ CFG
③ $S \to ABS \mid a$, $AB \to aV$ Not CFG
④ $S \to aSb \mid SS \mid a$ CFG
⑤ $S \to S \mid a \mid \varepsilon$ CFG
⑥ $S \to aSb$, $aS \to bb$ not CFG

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

S → AB
A → a
B → b

program → RI(){D;}
R → void | int
I → Identifier
D → TI
T → int | float | char

V = {Program, R, I, D, T}
T = {void, (,), {, }, ;, int, float, char, Identifier}

Derivation of a String:

LMD	RMD	Parse Tree
In every step leftmost non-terminal is substituted	In every step right most non-terminal is substituted	
S ⇒ AB ⇒ aB ⇒ ab	S ⇒ AB ⇒ Ab ⇒ ab	S → A B; A → a; B → b; leaf :- Terminal; Inner :- non-terminal

Note: LMD and RMD need not be same.

Derivation Length = no of non leaf nodes (5)

Length of Derivation = Derivation length = No of Steps = No of Substitutions = 3

LMD order = SAB
RMD order = SDA

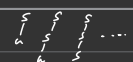
$S \to a \mid Aa$
$A \to \varepsilon \mid b$

2 derivation Tree
W = a
W = ba
only 2 derivation tree

No of Derivation Tree: 2

Note: For a particular String,
no of derivations
= no of distinct Parse Trees.
= no of LMDs
= no of RMDs

① $S \to SS \mid \varepsilon$ # no of derivations = ∞
W = ε
① $S \to \varepsilon$ ② $S \to SS \to \varepsilon$ ③

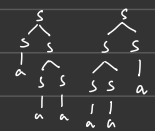
(2) $S \to S|a|ab$ $w = "a"$ → infinite 

(3) $S \to SS|a$

① $w = "a"$ → ⊥ $S \to a$

② $w = "aa"$ → 1 $S \to SS \to as \to aa$

③ $w = "aaa"$ → 2



(9) $S \to sss|a$

① $w = 'a'$ → ⊥

⑪ $w = "aa"$ → 0

⑩ $w = "aaa"$ → ⊥

Types of CFG based on derivation:

① Ambiguous CFG

$S \to \varepsilon|a|b|A$

$A \to a$

there is some string ("a") which has more than 1 derivation.

Some string has more than 1 PTs

$\exists w \in L(G) \Rightarrow > 1$ pt ⇒ easy to write

② unambiguous CFG

$S \to \varepsilon|a|b$

every string generated by the grammer has only 1 parse tree

$\forall w \in L(G) \Rightarrow 1$ PT ⇒ Better

Identify Unambiguous & ambiguous Grammer:

① $S \to a$ Unambiguous

② $S \to a|\varepsilon$ Unambiguous

③ $S \to AB|\varepsilon$, $A \to \varepsilon$, $B \to a$ Unambiguous

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

⑤ $S \to AB$, $A \to a|\varepsilon$, $B \to b|\varepsilon$ unamb

⑥ $S \to AB$, $A \to aA|\varepsilon$, $B \to bB|\varepsilon$ Unamb

⑦ $S \to AB$, $A \to aA|\varepsilon$, $B \to AB|\varepsilon$ ambiguous (a)

⑧ $S \to SS|\varepsilon$ ambiguous (ε)

⑨ $S \to SS|a$ ambiguous (aaa)

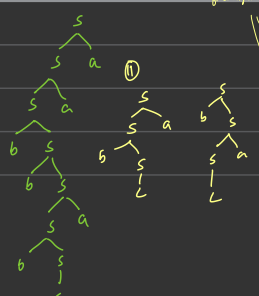
Left & Right Recursion with same symbol

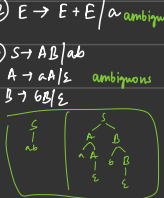
⑩ $S \to SaS|b$ ambiguous

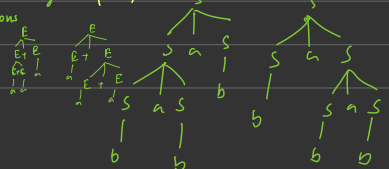
⑪ $S \to Sa|bS|c$ b^*ca^* → $w = baba$ ambiguous (bca)

⑫ $E \to E+E|a$ ambiguous

⑬ $S \to AB|ab$, $A \to aA|\varepsilon$, $B \to bB|\varepsilon$ ambiguous







Elimination of Left Recursion ⇒ Conversion from Left to Right Recursion.

↳ Top Down parser follows LMD

LL(1) $S \to Sa \mid b$ — In some cases it may go to infinite recursion

Left Recursive — Non Left Recursive

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

$\Leftrightarrow A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_k A'$

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

#use Simplified grammar

Left Recursion

Direct — $S \to Sa \mid b$

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

Indirect left Recursion: How to convert into Direct Left Recursion?

⇒ Substitution approach

e.g. $S \to Aa \mid bA$, $A \to Sb \mid c$ ⇒ $S \to Sba \mid bAa \mid ca$, $A \to Aab \mid cb \mid bAb$

Algo 1: Step 1: S, Step 2: A

Algo 2: step 1: A, step 2: S

Step 1: S : Direct Left Rec in S

A : substitute S and apply Direct Left Rec

B : Substitute SA then apply Direct Left Rec

C : Substitute S, AB then apply Direct Left Rec

① $S \to Sa \mid b$ (Left Rec. — non left Rec.) $L = b\,a^*$

$S \to bS'$, $S' \to aS' \mid \varepsilon$ — $S \to bX$, $X \to aX \mid \varepsilon$

② $S \to Sab \mid c \mid d$

$S \to cS' \mid dS'$, $S' \to abS' \mid \varepsilon$ OR $S \to cS' \mid dS' \mid c \mid d$, $S' \to abS' \mid ab$ (w/o null prod.)

③ $S \to Sa \mid Sb \mid cd \mid ef \mid fS \mid g$

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

$S \to cdS' \mid eS' \mid fSS'$, $S' \to aS' \mid bS' \mid gS' \mid \varepsilon$

Indirect Left Recursion — Direct Left Recursion

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

Step 1: Choose S productions

$S \to cX \mid AaX$, $X \to bX \mid \varepsilon$ ①

Step 2: Choose A productions

A → Ae | Sb | f

Substitute S and apply to 1st Prod.

A → Ae | AaXb | cXb | f

Direct Left Rec.

$A \to fY \mid cXbY$, $Y \to eY \mid aXbY \mid \varepsilon$ ②

Final Ans: $S \to cX \mid AaX$, $X \to bX \mid \varepsilon$, $A \to fY \mid cXbY$, $Y \to eY \mid aXbY \mid \varepsilon$

② $E \to E + T \mid f$ (α_1, β) $T \to F * T \mid f$ $F \to id$

Using Algo2

Step 1: $E \to FX$, $X \to +TX \mid \varepsilon$

Step 2: no E in 1st prod. & no direct left Rec.

Step 3:

Final: $E \to FX$, $X \to +TX \mid \varepsilon$, $T \to F * T \mid F$, $F \to id$

③ $S \to Aa$, $A \to Sb \mid Bc$, $B \to Se \mid f$

choose S: $S \to Aa$ — ①

choose A: $A \to Sb \mid Bc$

Subs. S: $A \to Aab \mid Bc$

$A \to BcX$, $X \to abX \mid \varepsilon$ — ②

choose B: $B \to Se \mid f$

Subs. S: $B \to Aae \mid f$

Subs. A: $B \to BcXae \mid f$

$B \to fY$, $Y \to cXaeY \mid \varepsilon$ — ③

Ans: $S \to Ae$, $A \to BcX$, $X \to abX \mid \varepsilon$, $B \to fY$, $Y \to cXaeY \mid \varepsilon$

Short Note:

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

$$\Downarrow$$

$$X \to \beta_1 Y \mid \beta_2 Y \mid \beta_3 Y$$

$$Y \to \alpha_1 Y \mid \alpha_2 Y \mid \varepsilon$$

BackTracking Issue

More Time Consumption

Left Factoring :

$S \to ab \mid acd$

one length Common prefix

CFG (w/ common prefixes) → Elimination of Common Prefixes (Left Factoring) → CFG (w/o common prefixes)

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

#no Common prefix

① $S \to a$ Left factored

② $S \to a \mid \varepsilon \mid bc$ "

③ $S \to Aa$
$A \to b$ "

④ $S \to ab \mid ad \mid cd$

final
$S \to aX \mid cd$
$X \to b \mid \varepsilon$

⑤ $S \to a \mid ab \mid abc \mid ef$

$S \to ef \mid aX$
$X \to \varepsilon \mid bY$
$Y \to \varepsilon \mid c$

⑥ $S \to Ab \mid acA \mid d$
$A \to ae \mid g$

$A \to ae \mid g$ ①

Substitute A
$S \to aeb \mid gb \mid d \mid acA$

$S \to gb \mid d \mid aX$
$X \to eb \mid cA$ ②

final
$S \to gb \mid a \mid aX$
$A \to ae \mid g$
$X \to eb \mid cA$

First Set	Follow Set
$S \to a \mid abc \mid d \mid c \mid \varepsilon$	$X \to aXb \mid Yb$ $Y \to bXef \mid Xm \mid \varepsilon$
First(S) = {a, d, c, ε}	Follow(X) = {b, e, m, $}
= {x | x is either ε or terminal where terminal is derived as 1st symbol from S}	= {t | t is a terminal derived as 1st symbol after X}
First(X): It is set of terminal (may include ε) where each terminal is derived as 1st symbol from X.	Follow(X): It is a set of terminals where every symbol is derived as 1st symbol after X.
#We need X rules for Computing First(X):	#We need whole occurrence where production contain X helps to calculate Follow(X):
① $S \to a \mid \varepsilon \mid bc$ First(S) = {a, ε, b}	Note: If S is first symbol then follow(S) must contain $ ($ is special end terminal which always end to rope)
② $S \to Sa \mid \varepsilon$ First(S) = {a, ε}	① $S \to a \mid \varepsilon \mid bc$ Follow(S) = {$}
③ $S \to Ab \mid cde$ First(S) = {a, b, c, ε} $A \to ae \mid \varepsilon \mid fg$ First(A) = {a, f, ε}	② $S \to Sa \mid \varepsilon$ Follow(S) = {$, a}
	③ $S \to Ab \mid cde$ Follow(S) = {$} $A \to ae \mid \varepsilon \mid fg$ Follow(A) = {b}
	④ $S \to A$ Follow(S) = {$} $A \to \varepsilon$ Follow(A) = {$}

Note: $X \to \alpha Y$ ⇒ Follow(Y) = { } ↳ Follow(X)

⑤ $S \to AB$, $A \to a$, $B \to b$

⑥ $S \to AaBb$, $A \to f$, $B \to f$

⑦ $S \to AB$, $A \to ab$, $B \to \varepsilon$

⑧ $S \to AB$, $A \to \varepsilon$, $B \to ef$

⑨ $S \to Ss \mid a$

⑩ $S \to ss(S) \mid \varepsilon$

NonTer.	First	Follow
S	{a}	{$}
A	{a}	{b}
B	{b}	{$}
S	{f}	{$}
A	{f}	{a}
B	{f}	{b}
S	{a}	{$}
A	{a}	{$}
B	{ε}	{$}
S	{ε}	{$}
A	{ε}	{e}
B	{e}	{$}
S	{a}	{a, $}
S	{(, ε}	{(,), $}

① Left Factoring
② First & Follow

L6 Lexical Analysis & Syntax analysis

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

First(E) = {a, (}

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

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

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

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

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

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

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

2024

Recursive Descent Parser
predictive Parser

Parsers [Syntax Analyzer]

- Top Down
  - LL(0)
  - LL(1)
- BoHom Up
  - LR parser
    - LR(0)
    - SLR(1)
    - LALR(1)
    - CLR(1) / LR(1)
  - operator precedence Parser

Top Down Parser

(i) Follows LMD

"L"

LL(1)

Left To Right

(ii) LL(1) parser is default

BoHom Up:

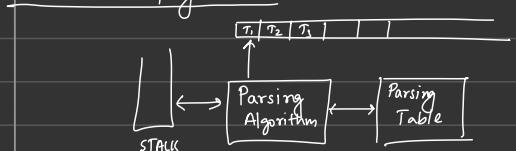
(i) follows "RMD in Reverse"

"R"

LR

(ii) LR(1) parser is powerful

Parser Configuration:



(i) LL(1) CFG

(ii) LL (1) Table

(iii) LL(1) algo

LL(1) Parser:

① It is top down parser
② Predictive Parser
③ It can be implemented using 2 ways
- i) Recursive-Decent Parser [Procedure Driven Approach]
- ii) Non-Recursive Decent Parser [Table Driven Approach]

every rule have a procedure

S → aA

S()
{ if (next Token = a)
A();

every production will be stored in Table

	a
S	S → aA

Q. LL(1) parser is _____.
- (a) Top Down Parser.
- (b) Bottom Up Parser.
- (c) Recursive Decent predictive Parser.
- (d) non Recursive Decent Predictive Parser.

How to write LL(1) CFG?

Step 1: Take unambiguous CFG

Step 2: eliminate left Recursive (unambiguous, non left Recursive Gram)

Step 3: eliminate common prefixes of one length (left Factored unambiguous, non left Recursive Gram)

Step 4: Build LL(1) Table if every entry has atmost 1 rule then given CFG is LL(1);

How to find LL(1) CFG?

Note: If CFG is unambiguous, non left factored, non left Recursive then CFG need not be LL(1)

① ② ③ ⇏ LL(1)

Note: if CFG is LL(1) then it is unambiguous, non left Recursive, left factored

LL(1) → ① ② ③

Which of the following is sufficient to be LL(1) CFG?
- (a) Ambiguous, left Recursive, left Factored CFG
- (b) Unambiguous, Non left Recursive, Non left Factored
- (c) Unambiguous, Non left Recursive, left factored
- (d) none of these

Which of the following is necessary to be LL(1) CFG?
- (a) Ambiguous, left Recursive, left Factored CFG
- (b) Unambiguous, Non left Recursive, Non left Factored
- (c) Unambiguous, Non left Recursive, left factored
- (d) none of these

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 using First and Follow Sets.

every entry of table has atmost 1 rule, so given CFG is LL(1) — 0/1

eg① $S \to a\overset{1}{S}b \mid \overset{2}{\varepsilon}$

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

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

M	a	b	$
S	1	2	2

LL(1) table size ⇓ (no of non. Terminal) × (no of Terminal + 1)

Fill with all productions of S which derive 'a' as 1st Symbol. — $S \Rightarrow aB$, $S \Rightarrow aBB$

Fill all the production of S which derive nothing (ε). [directly or Indirectly] — $|V| \times (|T|+1)$

M[S,a] = ? (depends on first)

M[S,$] = ?

M[S,b] = ? (depends on follow)

eg1: $S \to A\overset{1}{B}\overset{2}{\varepsilon}$, $A \to \overset{3}{a}\overset{4}{B}$, $B \to \overset{5}{\varepsilon}$

	a	b	$
S	3	1,2,5	1,2,5
A	3	4	
B		5	5

Entry with null rule.

eg2: $S \to AB \mid \varepsilon$, $A \to a \mid \varepsilon$, $B \to b \mid \varepsilon$

	a	$
S	S→AB, S→A	S→ε

Follow(S) = {a, $}

② $S \to a\overset{1}{S}a \mid \overset{2}{\varepsilon}$

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

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

M	a	$
S	⊥, 2	2

as ⊥ entry of Table has more than 1 rule so given grammer is not LL(1)

This grammer is ① unambiguous ② left Factored ③ Non left Recursive but it is not LL(1)

③ $S \to ss \mid (s) \mid \varepsilon$

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

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

only Rules on ε

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

④ $S \to \overset{1}{A}\overset{2}{B}$, $A \to \overset{3}{a}\overset{4}{A} \mid \overset{5}{\varepsilon}$, $B \to \overset{6}{b}\overset{7}{B} \mid \overset{8}{\varepsilon}$

First(S) = {a, b, ε} → Follow(S) = {$}

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

First(B) = {b, ε} → Follow(B) = {$, b}

M	a	b	$
S	⊥	⊥	⊥
A	2 3	3	3
B		4 5	5

CFG is not LL(1)

⑤ $S \to aAb$, $A \to Bb \mid \varepsilon$, $B \to ab$

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

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

CFG is LL(1)

2. $S \to aAbB \mid bAaB \mid \varepsilon$, $A \to S$, $B \to S$

	a	b	$
S	E_1	E_2	.
A	.	.	.
B	.	.	.

what is E_1 & E_2?

⇒ First(S) = {a, b, ε}, follow(S) = {$, b, a}

$E_1 = \{S \to aAbB, S \to \varepsilon\}$

$E_2 = \{S \to bAaB, S \to \varepsilon\}$

How to identify LL(1) CFG?

I) Theory → Sometime best

II) Short Cut → best

III) Table → Lengthy

Short Cut:

Rule 1: $X \to \alpha_1 \mid \alpha_2$: $\alpha_1, \alpha_2 \neq \varepsilon$

If $First(\alpha_1) \cap First(\alpha_2) \neq \phi$ then Given CFG is not LL(1)

$\alpha_1, \alpha_2 A = (T \cup V)^+$

$S \to a \mid Ab \mid C$, $C \to \varepsilon \mid \phi$ — R_1: {a}, R_2: {ε, φ}

Rule 2: $X \to \alpha \mid \varepsilon$

$First(\alpha) \cap Follow(X) \neq \phi$ then given CFG is not LL(1)

$S \to aSb \mid \varepsilon$

$First(aSb) \cap Follow(S) = \{a\} \cap \{b\} = \phi$ so it is LL(1)

① $S \to a \mid b \mid cd \mid \varepsilon$ — R_1, R_2, Follow(S) = {$} — It is LL(1)

② $S \to Sa \mid bS \mid c \mid d \mid \varepsilon$ — It is not LL(1) as Left Recursion is there.

③ $S \to a \mid bc \mid aef$ — not left Factored

④ $S \to AB \mid ab$, $A \to a \mid \varepsilon$, $B \to b \mid \varepsilon$ — ambiguous grammer.

⑤ $S \to aAbB \mid bAaB \mid \varepsilon$, $A \to S$, $B \to S$

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

Fo(S) ∩ First(aAbB) ≠ φ

Fo(S) ∩ First(bAaB) ≠ φ

so it is not LL(1)

(1) If CFG is not LL(1) then Which is possible :-

A) Ambiguous
B) Left Recursive
C) not left factored
D) Some table entry may contain more than one production rule.

(2) If CFG is not LL(1) then Which is True :-

A) Ambiguous
B) Left Recursive
C) not left factored
D) Some table entry may contain more than one production rule.

(3) If CFG is LL(1) then Which is True?

A) Unambiguous
B) Non Left Recursive
C) Left Factored
D) Every entry contains atmost 1 production

(4) CFG is LL(1) iff CFG is (D) every entry contains atmost 1 production

(1) $S \to aSa \mid bS \mid c$ is (A) LL(1) (B) not LL(1)

(2) 2. $S \to aAbB \mid bAaB \mid \varepsilon$
$A \to S$
$B \to S$

	a	b	$
S	E_1	E_2	.
A	.	.	.
B	.	.	.

What is E_1 & E_2?

First(S) = {a, b, ε} follow(S) = {$, b, a}

$E_1 = \{S \to aAbB, S \to \varepsilon\}$

$E_2 = \{S \to bAaB, S \to \varepsilon\}$

(3) $P \to xQRS$
$Q \to y z \mid z$
$R \to w \mid \varepsilon$
$S \to y$

follow(Q) = ?

A) {z} {w, y}
B) {w}
C) {w, y}
D) {w, $}

(4) $E \to E-T \mid T$
$T \to T+F \mid F$
$F \to (E) \mid id$

Equivalent non-Left Recursive Grammer is

(A) $E \to E-T \mid T$
$T \to T+F \mid F$
$F \to (E) \mid id$
Left Rec.

(B) $E \to TE'$
$E' \to -TE \mid \varepsilon$
$T \to T+F \mid F$
$F \to (E) \mid id$
Left Rec.

(C) $E \to TX$
$X \to -TX \mid \varepsilon$
$T \to FY$
$Y \to +FY \mid \varepsilon$
$F \to (E) \mid id$

(D) $E \to TX \mid (TX)$
$X \to -TX \mid +TX \mid \varepsilon$
$T \to id$

$X \to X\alpha_1 \mid \beta_1$

$X \to \beta_1 X'$
$X' \to \alpha_1 X \mid \varepsilon$

(5)

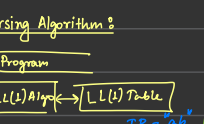
S → daT | Rf
T → as | baT | ε
R → Ca | TR | ε

	a	b	c	d	f	$
S			①		②	
T		③				④
R						

First(R) = {c, ε}
First(S) = {d, f, c}
First(T) = {a, b, ε}
Follow(T) = {$, c, f}
Follow(S) = {$}
Follow(R) = {f}

① [S, c] = {S → Rf}
② [S, f] = {S → Rf}
③ [T, c] = {T → ε}
④ [T, $] = {T → ε}

LL(1) parsing Algorithm :



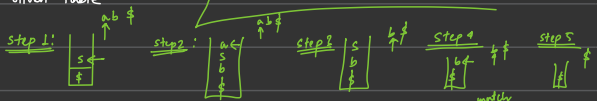
If TOS = Non Terminal
then M[tos, input]
i) Pop TOS
ii) push RHS in reverse → prediction (Derivation in Reverse order)

If TOS = Terminal
then Match ip & StackTop and increment ip pointer
pop StackTop

If TOS = InputPtr = $ then Accept

S	a	b	$
	S → aSb	S → ε	S → ε

assume Given Table



Conflict

max Size of stack assuming $ and S already in stack at start = 4

i) M Conflict :- If TOS is non Terminal and Table has blank entry.

ii) T Conflict :- If TOS is Terminal and TOS is not matched with input.

LL(0) ⇒ only one production for each non-Terminal.

LL(0) ✓
↳ LL(1), LL(2), ...

① $S \to a$ — LL(0)

② $S \to aA$, $A \to bB$, $B \to c$ — LL(0)

③ $S \to a|b$ — not LL(0) one two production for S; LL(1) ✓

④ $S \to a|ab$ — not LL(0), not LL(1), but LL(2) ✓✓

⑤ $S \to a|t|abc|abcde$ — LL(0) ✗, LL(1) ✗, LL(2) ✓, LL(3) ✗, LL(4) ✗, LL(5) ✓, LL(6) ✓, LL(9) ✓

Notes:
- (I) Every LL(k) CFG is LL(K+1) CFG
- (II) Every LL(2) " " Convertable to LL(1) CFG
  " LL(3) " " " to LL(1) "
  " LL(9) " " " " LL(1) "
- (III) Set of all language by LL(1) CFG ≡ Set of all language by LL(2) CFG

Top Down Parser	Bo Hom Up parser
1) predictive Parser	1) SR parser
2) Prediction (LMD substitution)	2) Shift and Reduce Action
3) Uses LMD "to verify Syntax" → To produce Parse Tree, → To derive input	3) uses "Reverse of RMD"

LR(0) parser:

I) LR(0) parsing diagram [LR(0) DFA]
II) Conflicts checking for LR(0) CFG
  → RR conflict
  → SR conflict
III) LR(0) Table Construction

L R (0)
L: Left To Right Scan
R: RMD in Reverse
0: No look ahead

LR(0) DFA:

1. Augmented CFG
2. Items and types of items
3. ** Closure() and goto() function
4. How to Construct LR(0) DFA?
5. How to check conflicts in LR(0)?

What is Augmented CFG?

$S \to Sa/Sb/Aa$
$A \to b$

Append $S' \to S$ → $S' \to S$, $S \to Sa/Sb/Aa$, $A \to b$

S' is new start symbol

Why we add $S' \to S$ in CFG?

How to accept given input?

$S' \to S \Rightarrow S' \to S.$ Accept

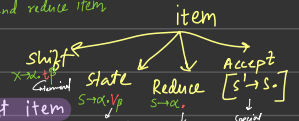
LR(0) items?

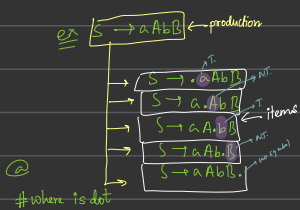
(•) dot is used to track progress in derivation

(•) dot is used to perform shift and reduce item

↳ It is a production along with a • (dot)

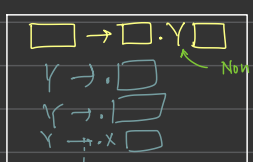
1. if after dot (•) terminal present it is called shift item
2. if after dot (•) nonterminal present it is called goto or state item
3. if dot (•) present at last it is called reduced or Completed item





Note: We must focus on dot (•) and next symbol after dot (•)

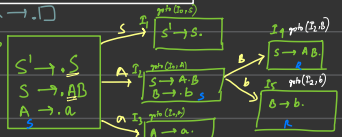
Note: If non terminal present just after dot (•) then we must add all productions in the same state



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

Let X = Closure $(S \to A.B)$

Then $|X| = 2$ = { $S \to A.B$, $B \to .b$ }



Total 6 state

It is LR(0) CFG

So SLR(1) CFG

$x \varepsilon = \varepsilon x = x$

$\varepsilon \cdot \varepsilon = \varepsilon$

Augmented CFG

$S' \to S$
$S \to AB$
$A \to a$
$B \to b$

Note:

I. $S' \to S.$ is acceptance item (It not involves in Conflicts)

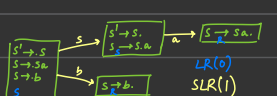
II. State Item also do not participate in any Conflict

III. State with only one item never produce any Conflict

IV. If state has reduced item then only thair is a chance to produce either SR/RR Conflict

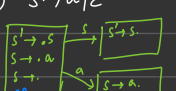
V. If no reduced item then no Conflict

② $S' \to S$, $S \to Sa/b$



4 state min

⑤ $S \to a/\varepsilon$



not LR(0)

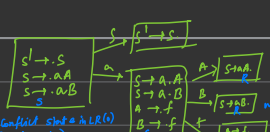
Follow(S) = {$} $a \notin$ Follow(S)

So it is SLR(1) CFG

③ $S' \to S$, $S \to aA/aB$, $A \to f$, $B \to f$

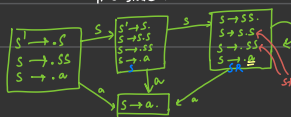
Follow(A) = {!}

Follow(B) = {$}



6 state min

④ $S' \to S$, $S \to SS/a$



Follow(S) = {$, a}

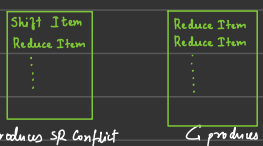
as $a \in$ Follow(S) it is SR conflict

not LR(0)

So CFG is not SLR(1)

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

if LR(0) DFA has no Conflict then given CFG is LR(0)



G produces SR Conflict — G produces RR Conflict (s).

6. $S \to Aa \mid Bb$, $A \to \epsilon$, $B \to \epsilon$

not LR(0) CFG is SLR

Follow(A) = {a}, Follow(B) = {b}, Follow(A) ∩ Follow(B) = ϕ

RR Conflict in LR(0)

7. $S \to Aa \mid Bb \mid cAb$, $A \to \epsilon$, $B \to \epsilon$

not LR(0) not SLR

Follow(A) = {a, b}, Follow(B) = {b}, Follow(A) ∩ Follow B = {b}

RR Conflict in LR(0)

8. $S \to AB \mid ab$, $A \to a$, $B \to b$

not LR(0) not SLR

Follow(A) = {b}, b ∈ Follow(A)

SR Conflict in LR(0) grammer

Note If CFG is Ambiguous then CFG is not LR(0) ϕ LR(1) ϕ CLR ϕ SLR

9. $S \to SS \mid (S) \mid \epsilon$

not LR(0) not SLR

SR Conflict in SLR(1)

RR in LR(0)

SR in LR(0)

SR in SLR(1)

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

10. $S \to aA \mid bB$, $A \to \epsilon$, $B \to bA$

no Conflict

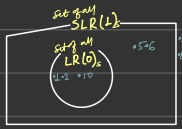
So LR(0) grammer

no Conflict so SLR(1) CFG

S → Simple
L → left to right scan
R → reverse of RMD
(1) → one look-a-head (Computed using whole CFG)

also known as SLR.

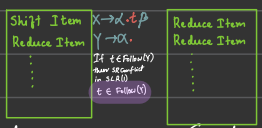
How to check given CFG is SLR(1) or not?



Note: i) Every LR(0) CFG is SLR(1) CFG
ii) LR(0) CFG need not be subset of SLR(1) CFG
iii) Set of all LR(0) is subset of all SLR(1)

Step 1: Construct LR(0) DFA

Step 2: Check Conflict in SLR(1)
if there is no conflict then CFG is SLR(1)



G produces SR Conflict
① If terminal of shift item present in follow of reduce item
t ∈ Follow(LHS of Reduce item)

G produces RR Conflict (s).
① If follow of the Reduce items are same
Follow(X) ∩ Follow(Y) ≠ ϕ

LR(0) parser
SLR(1) parser
} depends on LR(0) items

$X \to \alpha . \beta$

LR(0) items

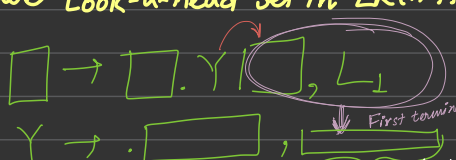
CLR Parser
LALR Parser
} Depends on LR(1) items

LR(1) item: LR(0) item + look-a-head

$X \to \alpha . \beta , \gamma$ ← look-a-head

LR(1) items

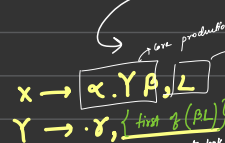
How to Compute Look-a-head set in LR(1) item?



How to Compute look-a-head of Y

ex 1
A → a.Bb, {c,d}
B → .ef, {b}

ex 2
A → a.B, {c,d}
B → .ef, {c,d}



C — Canonical

L — Left To Right Scanning

R — Reverse of RMD

(1) — Look ahead (based on exact path)

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

OR

$A \to a.b, c/d/e$

OR

$A \to a.b, L_1$ where $L_1 = \{c,d,e\}$

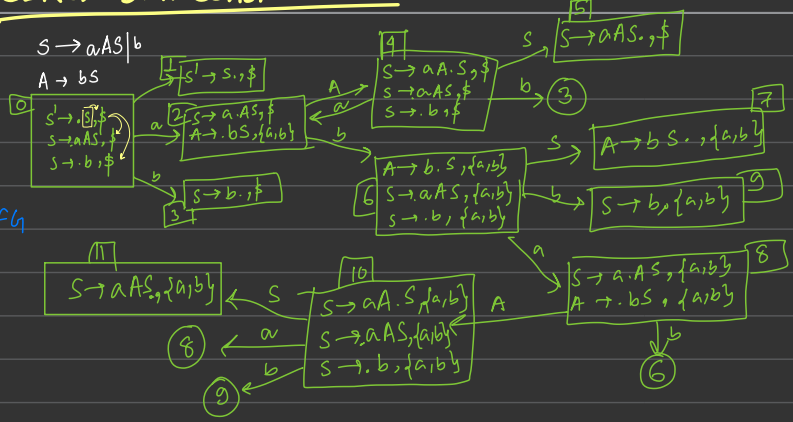
OR

$A \to a.b, L_1$ where L_1 is lookahead of A $L_1 = \{c,d,e\}$

CLR(1) DFA Construction:

$S \to aAS \mid b$

$A \to bS$



It is CLR CFG

In LALR we can Combine

① 4 10

② 3 9

③ 5 11

④ 2 8

Conflict Checking in both CLR & LALR:

SR Conflict	RR Conflict
Shift: $X \to \alpha.t\beta, L_1$ Reduce: $Y \to \alpha., L_2$ ⋮	Reduce: $X \to \alpha., L_1$ Reduce: $Y \to \alpha., L_2$ ⋮
if $t \in L_2$ then SR Conflict	if $L_1 \cap L_2 \neq \phi$ then RR Conflict

LALR DFA :

Step 1: Construct CLR DFA

Step 2: Merge state of CLR if they have same items but look-a-head may differ.

Merge(4,10) = 410

Merge(3,9) = 39

Merge(5,11) = 115

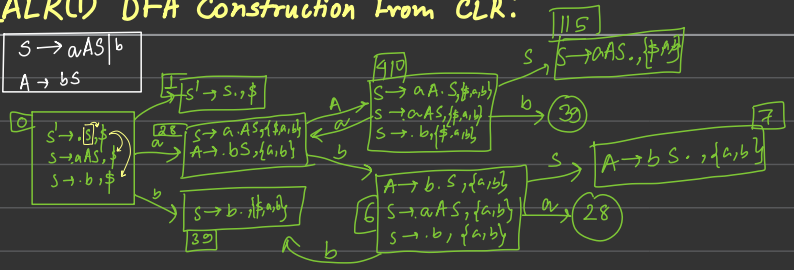
Merge(2,8) = 28

Note : If we remove look-a-head of LALR DFA it will be same as LR(0)

LALR(1) DFA Construction From CLR:

$S \to aAS \mid b$

$A \to bS$



It is LALR CFG

It is also LR(0)

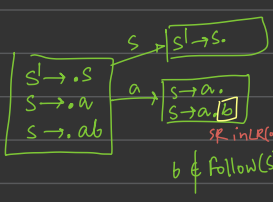
No of State in LR(0) = No of state in SLR(1) = No of State in LALR $\leq$ no of state in CLR

② $S \to a$

- (A) LL(1)
- (B) LR(0)
- (C) SLR
- (D) CLR
- (E) LALR
- (F) AMBIGUOUS

③ $S \to a \mid ab$

- (A) LL(1)
- (B) LR(0) → SR
- (C) SLR
- (D) CLR
- (E) LALR
- (F) AMBIGUOUS



④ $S \to SS \mid a$

- (A) LL(1)
- (B) LR(0)
- (C) SLR
- (D) CLR
- (E) LALR
- (F) AMBIGUOUS

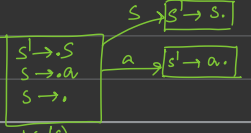
LL1
① unambiguous
② left factored
③ no left Recursion

⑤ $S \to a \mid \varepsilon$

first(a) ∩ follow(S) = ϕ

follow(S) = {$}

- (A) LL(1)
- (B) LR(0) SR
- (C) SLR
- (D) CLR
- (E) LALR
- (F) AMBIGUOUS



M	a	$
S	S → a	S → ε

first(S) = {ε, a}

follow(S) = {$}

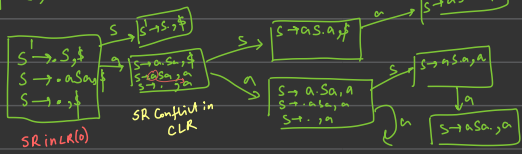
⑥ $S \to aSa \mid \varepsilon$

follow(S) ∩ first(aSa) = {a}

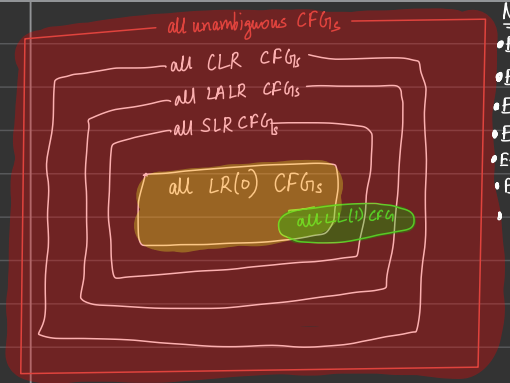
ϕ

unambiguous

- (A) LL(1)
- (B) LR(0)
- (C) SLR ← a ∈ fo(S)
- (D) CLR
- (E) LALR
- (F) AMBIGUOUS



follow(S) = {a, $}



Note:

- Every LR(0) CFG is SLR, LALR, CLR & Unambiguous
- Every SLR CFG is LALR, CLR & Unambiguous
- Every LALR CFG is CLR & Unambiguous
- Every CLR CFG is Unambiguous
- Every not CLR CFG is not LALR, not SLR not LR(0)
- Every not LALR CFG is not SLR, not LR(0)
- Every not SLR CFG is not LR(0)

LL(1) Vs LR(1)

LR(1) ≡ CLR

I) every LL(1) CFG is LR(1) CFG.

II) No relation b/w LL(1) and SLR/LALR/LR(0).

III) If LL(1) CFG is free from null rules then always SLR(1).

IV) If LL(1) without unit rules then it is always LALR(1).

Ex: $S \to a \mid b$

a) LL(1)
b) LR(1)
c) not LL(1)
d) not LR(1)

CLR V/s LALR

Note: (1) If CFG is CLR(1) and after merging states of CLR to make LALR then RR Conflict possible in LALR.

(I) $A \to a.,\{L_1\}$, $B \to a.,\{L_2\}$ — no RR Conflict in CLR

&

(II) $A \to a.,\{L_2\}$, $B \to a.,\{L_3\}$ — two RR Conflict in CLR

after merging (I) and (II): $A \to a.,\{L_1,L_2\}$, $B \to a.,\{L_2,L_3\}$

$L_2 \cap L_3 \neq \emptyset$ so RR Conflict in SLR

Note: (2) If CFG is CLR(1) and after merging states of CLR to make LALR then SR Conflict never possible (impossible)

(I) $A \to a.b, L_1$; $B \to a., L_2$ — $b \notin L_2$

(II) $A \to a.b, L_3$; $B \to a., L_4$ — $b \notin L_4$

after Combining (I) and (II): $A \to a.b, L_1 \cup L_3$; $B \to a., L_2 \cup L_4$ — $b \notin L_2 \cup L_4$

So SR Conflict never possible

Note 3: Today many compilers built based on LALR. If any SR Conflict occurs then it performs shift action and proceeds for further parsing.

YACC → Yet Another Compiler Compiler

LALR is more popular

CLR is more powerful but consume more space.

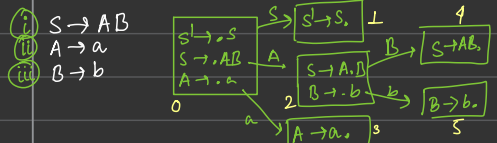
Table Construction:

LL(1) Table	LR Table
entries:- (a) production/rule (b) blank	entries:- (a) shift (b) reduce (c) goto (state) (d) accept (e) blank Action ↳ Shift, Reduce, Accept GOTO ↳ goto
Size: $(NT) \times (T+1)$ = No of Non Terminal * (No of Terminal +1) = $\lvert V \rvert \times \lvert \{T\} \cup \{\$\} \rvert$	Size: Action Table: (No of State) X (No of Terminal +1) GoTo Table: (No of State) X (No of Non Terminal)

Exam me Table Nahi banana hai

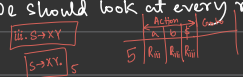
LR(0) Table Construction:

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

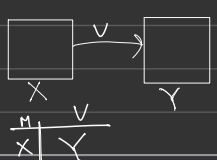


Action shift:

Reduce → We should look at every reduce item.



Goto:



Accept:

State	Action a	Action b	Action $	Goto S	Goto A	Goto B
0	S_3			1	2	
1			Accept			
2		S_5				4
3	Rii	Rii	Rii			
4	Ri	Ri	Ri			
5	Riii	Riii	Riii			

No of shift entry
= No of Terminal Transition

No of Goto/State entry
= No of Non Terminal Transition

No of Reduce entry

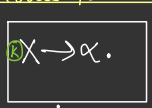
No of Accept Entry = 1

SLR(1) Table & LR(0) Table

→ Shift Entry are same (=) equal
→ goto Entry are same (=) equal
→ only reduce entry may differ

reduce Entry in LR(0) ≥ # reduce Entry in SLR(1)

Rules for Reduce Entry in SLR :



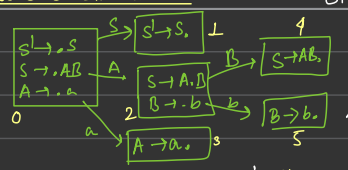
	t_1	t_2	t_3
i	R_K	R_K	

Rule: Reduce entry only under follow set of X

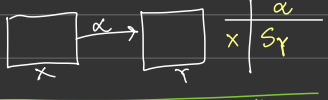
SLR(1) Table Construction :

(i) S → AB
(ii) A → a
(iii) B → b

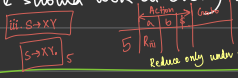
Follow(S) = {$}
Follow(A) = {b}
Follow(B) = {$}



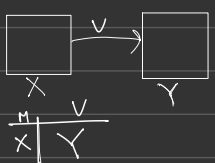
Action
shift :



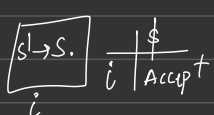
Reduce → we should look at every reduce item.



Goto :



Accept :



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

No of Shift entry
= No of Terminal Transition

No of Goto/State entry
= No of Non Terminal Transition

No of Reduce entry

No of Accept Entry = 1

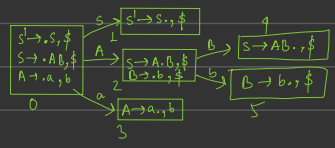
CLR Reduced Entries
→ look at every reduced item
$[X \to \alpha .\ ,\ t_1, t_3]$

i	t_1	t_3
i	R_K	R_K

CLR & LALR table is same for the given CFG

CLR & LALR Table Construction :

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



	Action			Goto		
	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) Shift Entries in LR(0), SLR and LALR are always same
ii) GoTo " " " " "
iii) Reduce entry may differ in all.

i) Shift entries of LR(0) = Shift entries of SLR = Shift entries of LALR $\leq$ Shift entry (CLR)
ii) goto entries of LR(0) = goto entries of SLR = goto entries of LALR $\leq$ goto entry (CLR)
iii) Reduced entries of LR(0) $\geq$ Reduced entries of SLR $\geq$ Reduced entries of LALR ? Reduced entry (CLR)

(all terminal) (follow using whole grammer) (Combined path) (exact path)

Relation :

Parser :

More popular — More powerful

$$LR(0) < SLR < LALR < CLR$$ (CLR = LR(1))

no relation

$$LL(1) < CLR$$

LL(1) CFGs ⊄ X

Grammer :

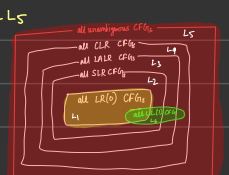
$$LR(0)\ CFGs \xrightarrow{is} SLR(1)\ CFGs \to LALR\ CFGs \to CLR\ CFGs \to Unambiguous$$

(⊄ at each step; ⇐ "need not be true")

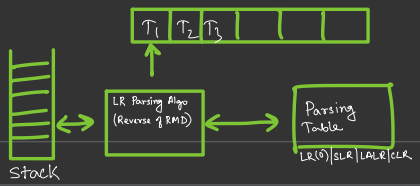
Classes :

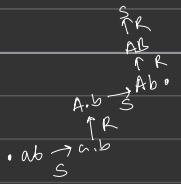
L_1 = set of all LR(0) CFGs
L_2 = " " " SLR "
L_3 = " " " LALR "
L_4 = " " " CLR "
L_5 = " " " Unambiguous "
L_6 = " " " LL(1) CFGs

① $L_1 \subset L_2 \subset L_3 \subset L_4 \subset L_5$
& ② $L_6 \subset L_4 \subset L_5$



LR Parsers :

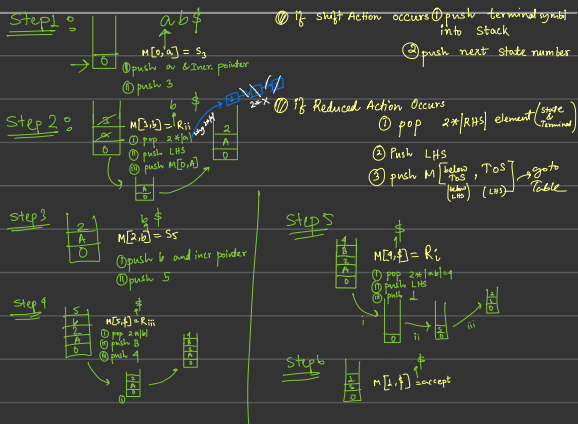




i) $S \rightarrow AB$
ii) $A \rightarrow a$
iii) $B \rightarrow b$

Parsing Table

	Action			Goto		
	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}			



$S \rightarrow AB$
$A \rightarrow Ab$
$B \rightarrow b$

W = abb

Reverse

LMD : SAABB	RMD : SBABA	Rev. of RMD : ABABS
S	S	S
AB	AB	AB
ABB	Ab	Ab
aBB	ABb	ABb
abB	Abb	Abb
abb	abb	abb

operator precedence parsing

- → with the help of Parse Tree
- → " " " " Grammer
- → " " " " Table
- → " " " " Theory

Highest	*, /, %	Left to right
Lowest	+, −	Left to right

equal

What is precedence?

I) highest
II) lowest
III) equal

a + b * c

a * b + C

S → A + B
A → a
B → b * C

It is impossible to derive using grammer

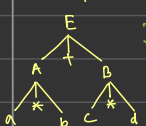
	+	*
+	—	<
*	—	—

(left, right)

a + b + C

right associative, left Associative

Q1. Find precedence relation using Parse Tree:

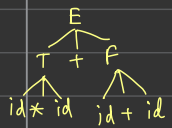


A. + is lowest
B. * is highest
C. + is highest
D. None

Q. Find Correct operator precedence table

left \ Right	+	*
+	—	<
*	>	—

Q2. Find precedence relation using Parse Tree:

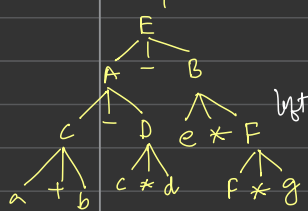


Q1: Chose Correct statement
A. + is lowest
B. * is highest
C. + is left Associative
D. + is right Associative

Q2 fill all entries

	+	*
+	<	—
*	>	—

Q3. Find precedence relation using Parse Tree:



left \ Right	+	*	—
+	—	—	>
*	—	<	>
—	—	<	>

OPARATOR GRAMMER :

↳ It is a CFG which do not contain Consequtive two non-Terminal & also do not contain null production.

ex ~~S → AB~~ ~~A → a~~ ~~A → b~~ | S → A+B, A → a, B → b | ~~S → A+B~~ ~~A → ε~~ ~~B → a~~ | S → AB, ~~S → ε~~, ~~B → b~~

Oparator Grammer

↳ It may be ambiguous or unambiguous

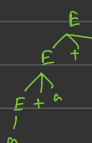
left Recursion create left associative.

(4) Find precedence relation using CFG:

E → E+a | a

w = a+a+a

'+' is left associative



	+
+	>

(5) Find precedence relation using CFG:

E → A+B '+' is left associative

A → a+a

B → a

(6) Find precedence relation using CFG:

E → a+E | a '+' is right associative

(7) Find precedence relation using CFG:

E → a+A '+' is right associative

A → a+a

(8) Find precedence relation using CFG:

E → E+T | a

T → F*T | b

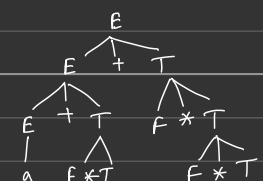
F → c

i) * highest

ii) + lowest

iii) + is left associative

iv) * is right associative



	+	*
+	>	<
*	>	<

(9) E → E+E | E-E | id

and

+ is highest

- is lowest

+ is left to right associative

- is right to left associative

(*) Find output for:-

2+3-1-5+6-1

⇒ 5 - 1 - 11 - 1

⇒ 5 - 1 - 10

⇒ 5 - (-9)

⇒ 14

	+	-
+	>	>
-	<	<

2+3 ——— 4+5

↖ independent ↗

2 + 3 + 4

left to right

Q1. Which of the following statements is TRUE?

(A) The LALR parser for a grammar G cannot have reduce-reduce conflict if the LR(1) parser for G does not have reduce-reduce conflict.

(B) Symbol table is accessed only during the lexical analysis

(C) Data flow analysis is necessary for runtime memory management

(D) LR(1) parsing is sufficient for deterministic context free languages

DCFL $\cong$ LR(1) CFG $\cong$ LR(1) languages

(I) For every DCFL, LR(1) CFG exist.

(II) Every LR(1) generates DCFL

Set of all DCFL = Set of all languages generated by LR(K) CFGs, $K \geq 1$

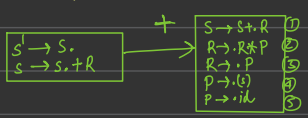
Q2. Consider the augmented grammer with {+, *, (,), id} as the set of terminals.

$S' \rightarrow S$

$S \rightarrow S+R \mid R$

$R \rightarrow R*P \mid P$

$P \rightarrow (S) \mid id$



if I_0 is the set of two LR(0) items $\{[S' \rightarrow .S], [S \rightarrow .S+R]\}$ then goto(Closure(I_0), +) contains exactly 5 items.

CYK Algorithm is
- Bottom Up parsing
- Dynamic Programming
- $O(n^3)$ algo
- Membership Algo for CFG
- It verifies given string generated by given CFG or not

Q3 Consider the following statements

SI) every SLR(1) grammer is unambiguous but there are certain unambiguous grammers that are not SLR(1).

SII) For any context free grammer there is a parser that takes at most $O(n^3)$ time to parse a string of length n.

Which one of the following option is correct?

a) S_I is true S_{II} is false

b) S_I is false S_{II} is True

c) S_I is True S_{II} is True

d) S_I is False S_{II} is False

Q4. Consider the following Augmented grammer with {#, @, <, >, a, b, c} as the set of terminals

$S' \rightarrow S$

$S \rightarrow S\#CS$

$S \rightarrow SS$

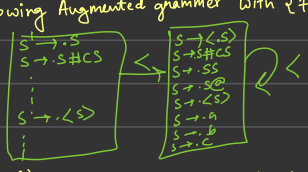
$S \rightarrow S@$

$S \rightarrow \langle S \rangle$

$S \rightarrow a$

$S \rightarrow b$

$S \rightarrow c$



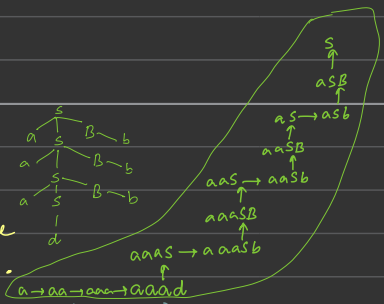
Let I_0 = Closure({S' → .S}). The no of items in the set GoTo(GoTo(I_0, <), <) is 8.

Q Consider the following grammer:

$S \to aSB/d$

$B \to b$

The no of Reduction Steps taken by a bottom up parser while accepting the String aaadbbb is _____.



No of Reduction steps in Bottom Up Parser (Rev. of RMD)

= No of Steps in RMD

= No of Non Terminal in ParseTree

Q Which of the following is used by LR parser?

GATE IM

(A) Left Most

(B) Left Most in Reverse

(C) Right Most

(D) Right Most in Reverse ✓

Q Consider the operator precedence and associativity rules for the integer arithmatic operators given in the table below:

operator	precedence	Associativity
+	Highest	Left
−	High	Right
*	Medium	Right
/	low	Right

The value of the expression $3+1+5*2/7+2-9-7-6/2$ as per the above rule is _____.

$3+1+5*2/7+2-9-7-6/2$

$= 9*2/9-9-7-6/2$

$= 9*2/9-9-1/2$

$= 9*2/9-3/2$

$= 9*2/6/2$

$= 18/6/2$

$= 18/3$

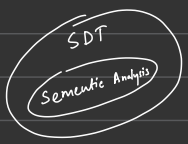
$= 6$

Topics to be Covered:
- What is SDT
- Use of SDT
- Lexical v/s Syntax v/s Semantic v/s SDT
- Concept of SDT
  - Attributes of SDT
  - Defination of SDT
  - Evaluation of SDT

CH3 Syntax Directed Translation

Application:
(I) Semantic Analysis
(II) Syntax tree generation
(III) Intermediate Code generation
(IV) Generating Perse tree
(V) any meaningful activity

SDT = Syntax + Translation
= CFG + Translation
Translation → any thing / syntax / Semantic Actions / Program



(I) Can be used to translate Expression:
(Infix / Prefix / Postfix ⇒ Infix / prefix / Postfix)

(II) Translate Numbers [Binary, Octal, HexaDecimal, Decimal]

(III) It can evaluate expression.

$E \rightarrow E+E$ {Translation}
$E \rightarrow a$ {Translation}

SDT = CFG + Translation

Lexical Analysis	Syntax Analysis	Semantic Analysis	SDT
Tokens	Structure → Declaration → If-else → Loop → Fun^n → Expression	Type checking → in expression → in function → declaration before use ⋮	Anything, You want, has some logical information in Syntax It is more powerful than Compiler.

Syntax Directed Translation: (SDT)

- Attributes
  - Inheritated Attributes
  - Synthesized Attributes
- Definition
  - L-attributed SDT
  - S-attributed SDT
- Evaluation

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

What is CFG?
What is Translation?
What is attributes? {x, val}
What is Defination?
What is evaluation?

Attribute:

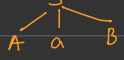
(I) **Inherited Attribute:**
↳ Computation depends on parent/siblings.

$S \rightarrow AB$ {A.x = B.y} → A depends on B, x is inherited
Ax = By + Sa
x, y attributed

(II) **Synthesized Attribute:**
↳ Computation depends on children.

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

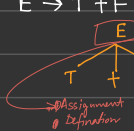
$S \rightarrow A\ a\ B$



production tree

S: parent of A, a, B
A: Child of S, left Sibling of a and B
a: Child of S, Left Sibling of B, right sibling of A
B: Child of S, right Sibling of a, A

$E \rightarrow T+F$ {Ex = Ty * F.z}



Where is Computation happening?
(A) E
(B) T
(C) F
(D) E, T and F

E is computed by depending its Child T and F.

in Whole grammer if a attribute is synthesized it is sythesized
" " " " " " inherited " " inherited
" " " " " both synthesized & inherited ⇒ than it is neither inherited & not synthesized for SDT

Identify Type of attribute:

(1) (i) $E \to E_1 + E_2$ { E.x = E_1.x + E_2.x }
(ii) $E \to id$ { E.x = id.val }

↳ subscript to distinguish Terminals. actually no subscript present

Q1. in Rule (i), x is synthesized.
Q2. in Rule (ii), x is synthesized.
Q3. in SDT, x is synthesized

(2) i) $S \to D\ L$; { L.type = D.type }
ii) $D \to int$ { D.type = int }
iii) $L \to L_1$, id { L_1.type = L.type }
iv) $L \to id$ { }

Q in rule I, type is inheritated
Q in rule II, type is synthesized
Q in rule III, type is inheritated
Q in whole SDT, type is neither inheritated nor synthesized
⇒ some time some times

(3) $S \to S_1\ S_2$ { S.count = S_1.count + S_2.count }
$S \to (S_1)$ { S.count = S_1.count + 1 }
$S \to \varepsilon$ { }

In above SDT, Count is synthesized

(4) $S \to Aa$ { S.x = A.x ; A.y = S.x }
$A \to b$ { A.x = 100 ; A.y = 1000 }

(i) x is synthesized attribute
(ii) y is inheritated attribute.
in SDT

(5) $E \to T + f$ { E.x = T.y ; F.y = E.x + 2 ; E.y = F.y - 1 }
$T \to id$ { T.x = 10 ; T.y = id.val }
$f \to id$ { f.x = id.val ; f.y = 20 }

(i) X is synthesized.
(ii) y is neither synthesized nor inheritated.
Some times inheritated, some times synthesized.

SDT Defination:

TDP & BUP evaluations depends on Non terminal order if Translation order is changed o/p will remain same.

L-Attributed SDT

I) Computed depends on parent/left Sibling/children

II) Translation can be placed anywhere in production. ex $X \to \alpha\ \beta$

III) evaluation depends on translations order from left to right
if Translation position changes then o/p may change

S-Attributed SDT

I) Computation depends on only children

II) Translation should be at the end of production

III) Evaluation depends on bottom up parsing.
Non Terminal order ← (reverse of RMD)

Note:
I) Every S-attributed Grammer is L attributed
II) L-attributed SDT may or may not be S-attributed.

(1) $E \to E_1 + E_2$ { E.x = E_1.x + E_2.x }
$E \to id$ { E.x = id.val }

MCQ
Q. The given SDT is
A) L-attributed only
B) S-attributed only
C) both S-attributed & L-attributed
D) None

MSQ
a) L-attributed
b) S-attributed
c) both A and B
d) None

(2) i) $S \to D\ L$; { L.type = D.type } (right Sibling)
ii) $D \to int$ { D.type = int } (leaf)
iii) $L \to L_1$, id { L_1.type = L.type } (parent)
iv) $L \to id$ { }

MSQ
a) L-attributed (but not S-attributed)
b) S-attributed
c) both A and B
d) None

(3) $S \to S_1\ S_2$ { S.count = S_1.count + S_2.count } (child)
$S \to (S_1)$ { S.count = S_1.count + 1 }
$S \to \varepsilon$ { }

Above SDT is both S-attributed & L-attributed

MCQ
A) L-Attributed
B) S-Attributed
C) L-Attributed but not S-attributed
D) none

(4) $S \to Aa$ { S.x = A.x ; A.y = S.x }
$A \to b$ { A.x = 100 ; A.y = 1000 }

⇒ It is not S-attributed. but L-attributed.

(5) ~~$E \to T + F$ { E.x = T.y ; F.y = E.x + 2 ; F.y = f.y - 1 }~~
$T \to id$ { T.x = 10 ; T.y = id.val }
$F \to id$ { f.x = id.val ; f.y = 20 }

⇒ It is not S-attributed. but L-attributed only

(6) $S \to S_1 S_2$ { S_1.x = S_2.x + S.x }
$S \to a$ { S.x = a.val }
depending on right Sibling

Above SDT is
A) L-attributed only
B) S-attributed only
C) both
D) none [neither L-attributed nor S-attributed]

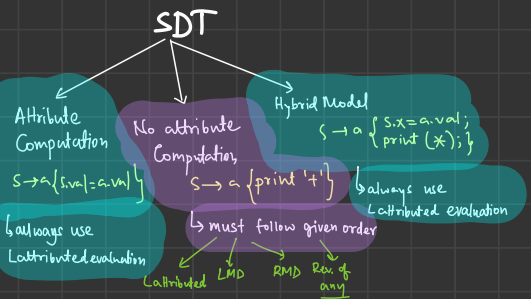
A) L-Attributed only
B) L-Attributed
C) L-Attributed but not S-attributed
D) none

(7) $S \to b$ { S.x = a.val } a
$S \to b$ { S.x = b.val }
It is only L attributed
Translation in the begining of production

Evaluation of SDT :

I) In S-attributed SDT, attributes are evaluated using Bottom up approach. (Reverse of RMD)

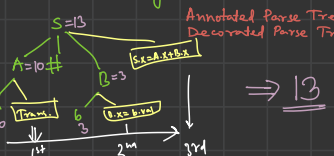
II) In L attributed SDT, Inherited attributes are evaluated using Top Down approach and synthesized attributes are evaluated using bottom up approach



L-Attributed SDT	S-Attributed SDT
Evaluation (Translation)	Evaluation (Translation)
↳ All translations are evaluated from left to right.	↳ Evaluation of non-terminal depends on "Reverse of RMD"
(Topological Order)	(Bottom up Parsing)
(In order)	(SR parsing)
(DFS, Left to Right)	(LR parsing)

① S → A#B {S.x = A.x + B.x}
A → a {A.x = a.val}
B → b {B.x = b.val}
(Input: 10#3). Find the attribute value Computed at the root for the given Input?
It is S attributed (& L attributed)

⇒ Method 1: Using L-attributed evaluation

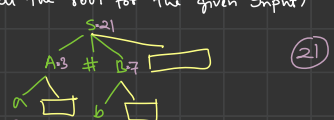


⇒ 13

Method 2: Use S attributed Method (Reverse of RMD)
RMD = S B A
Rev. RMD = A B S

13

② S → A#B {S.x = A.x * B.x}
A → a {A.x = a.val+1}
B → b {B.x = b.val+2}
Input 2#5. Find the attribute value Computed at the root for the given Input?



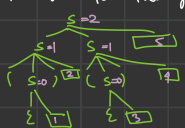
21

③ S → S1 S2 {S.x = S1.x − S2.x}
S → (S1) {S.x = 3 + S1.x}
S → ε {S.x = 0}

Ans = 0/∞

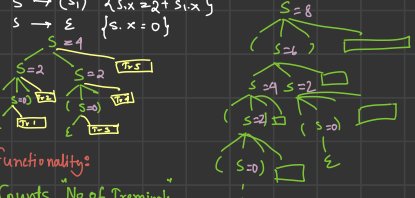
Input:)(() . Find the attribute value Computed at the root for the given Input? Ambiguity (Multiple Answer)

④ S → S1 S2 {S.x = S1.x + S2.x}
S → (S1) {S.x = 1 + S1.x}
S → ε {S.x = 0}
Input: (()() Find the attribute value Computed at the root for the given Input?



⑤ Functionality: Counts no of balanced Parenthesis at root
Input ((())())
O/P: 4

⑥ S → S1 S2 {S.x = S1.x + S2.x}
S → (S1) {S.x = 2 + S1.x}
S → ε {S.x = 0}



Functionality:
Counts "No of Terminals"
Counts "Length no of String"

⑦ E → E1 + E2 {E.x = E1.x + E2.x}
E → id {E.x = id.val}
Input: 2+4+7
What is attribute value Computed at root? (13)

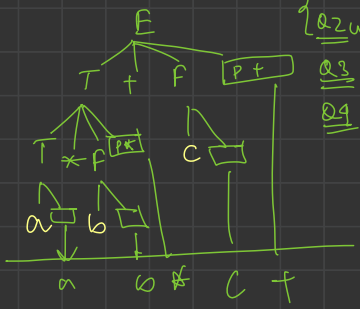
⑧ E → E1 + E2 {E.x = E1.x + E2.x + 1}
E → id {E.x = 0}
Functionality: no of operator

⑨ E → E1 + E2 {E.x = E1.x + E2.x}
E → id {E.x = 1}
Functionality: No of operands

⑩ E → E1 + E2 {E.x = E1.x + E2.x + 1}
E → id {E.x = 1}
Functionality: no of (operator + operand) (Terminals) (inputs)

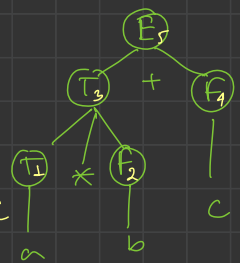
11) E → T+f {print +}
T → id {print id.val}
T → T*F {print *}
T → id {print id.val}
F → id {print id.val}

Input: a*b+C
output: ab*C+

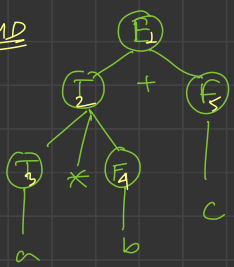


Q1 output : ab*C+ (postfix)
Q2 what is output using L attributed? ab*C+
Q3 what is output using Bottomup parsing? ab*C+
Q4 what is output using Top Down Parsing? +*abC (prefix)

RMD

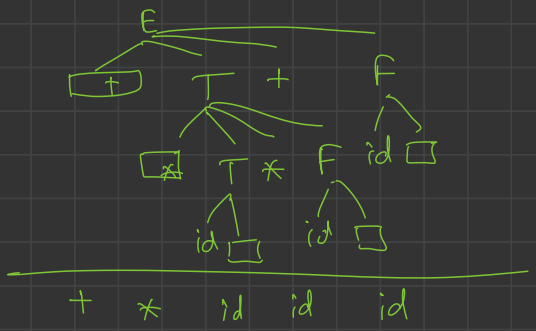


LMD



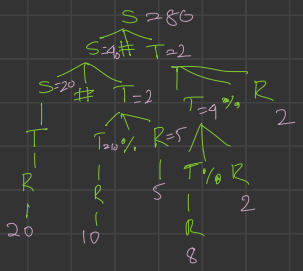
12) E → {print +}T+f
T → {print id.val}id
T → T*F {print *}
T → id {print id.val}
F → id {print id.val}

Q1 output prefix +*abC
Q2 what is output using L attributed? prefix +*abC
Q3 what is output using Bottomup parsing? ab*C+ (postfix)
Q4 what is output using Top Down Parsing? +*abC (prefix)

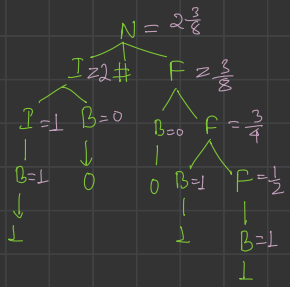


① $S \to S_1 \# T \quad \{S.val = S_1.val * T.val\}$
$S \to T \quad \{S.val = T.val\}$
$T \to T_1 \% R \quad \{T.val = T_1.val / R.val\}$
$T \to R \quad \{T.val = R.val\}$
$R \to id \quad \{R.val = id.val\}$

Input: 20#10%5#8%2%2
Compute value at Root.



② $N \to I \# F \quad N.val = I.val + F.val$
$I \to I_1 B \quad I.val = 2 I_1.val + B.val$
$I \to B \quad I.val = B.val$
$F \to B F_1 \quad F.val = \frac{1}{2}(B.val + F_1.val)$
$F \to B \quad F.val = \frac{1}{2} B.val$
$B \to 0 \quad B.val = 0$
$B \to 1 \quad B.val = 1$



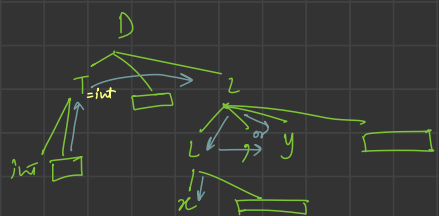
Input 10 #011 : Ans $2\frac{3}{8}$

③ $D \to TL \quad \{X_1.type = X_2.type\}$
$T \to int \quad \{T.type = int\}$
$T \to float \quad \{T.type = float\}$
$L \to L_1, id \quad \{X_3.type = X_4.type ; Addtype(id.entry, X_5.type)\}$
$L \to id \quad \{Addtype(id.entry, X_6.type)\}$

Ans: $X_1 = L \quad X_5 = either(L \text{ or } L_1)$
$X_2 = T \quad X_6 = L$
$X_3 = L_1$
$X_4 = L$

Input: int x,y;

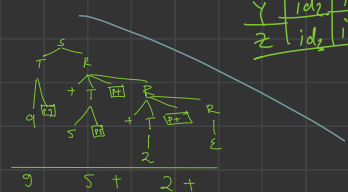
x	id_1	int
y	id_2	int
z	id_3	int



int x.type
y.type
z.type

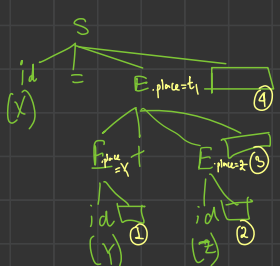
all same

④ $S \to TR$
$R \to +T\{print '+'\}R$
$R \to \varepsilon$
$T \to num \{print(num.val)\}$
Input: 9+5+2
Ans Output: 95+2+ (postfix)



⑤ $S \to id = E \{ gen(id.place = E.place) \}$ (print())
$E \to E_1 + E_2 \{ t = newtemp(); E.place = t; gen(t = E_1.place + E_2.place) \}$ (create new temp variable $t_1, t_2, t_3 \ldots$)
$E \to id \{ E.place = id.place \}$

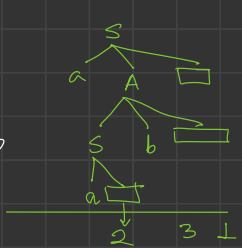
Input: X = Y+Z



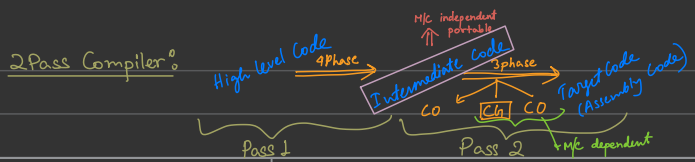
output: $t_1 = Y + Z$
$X = t_1$

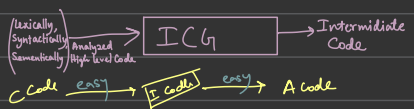
⑥ $S \to aA \{print 1\}$
$S \to a \{print 2\}$
$A \to Sb \{print 3\}$

Input: aab
output: 231



Intermediate Code : → It helps portability



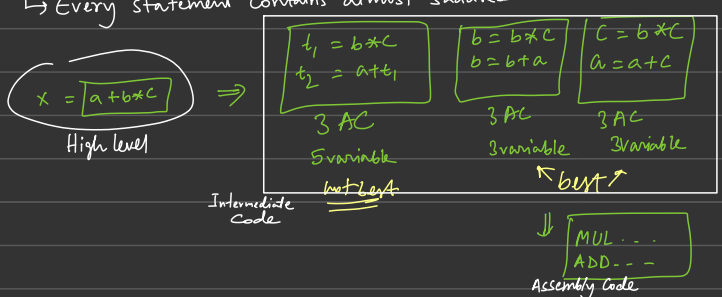


Intermediate Code Representation :

① Linear Form :
i) Postfix Form
ii) 3AC / TAC / Three Address Code
iii) SSA Code (Static Single Assignment)

② Non Linear Form :
i) Syntax Tree
ii) DAG (Directed Acyclic Graph)
iii) Control Flow Graph

Three Address Code :
↳ Every statement contains atmost 3 address (variable)



postfix Code : (reverse polish Notation)

a+b*c ⇒ a b c * +

X = a+b*c ⇒ x a b c * + =

SSA Code (Static Single Assignment Code):

(Defination)

↳ It is a 3AC but every variable have single assignment (defination).

```
b = b * c      → 3 defination for a single varinable(b)
b = b + a      ⊥ defination for (a).
               ⊥ defination for (c)
```

Not SSA Code

```
t1 = b * c
t2 = a + t1
```

SSA Code

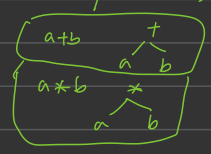
Q How many assignment for b ; `b = b + c;`

Ans = 2 (b = ② + ③, b = ⑤)

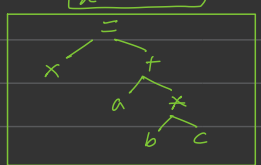
Syntax Tree: (Abstract Syntax Tree) (AST)

leaf : oparand

non-leaf: oparator

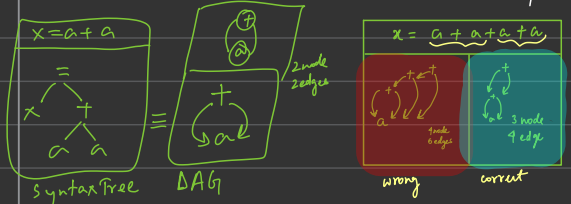


$x = a + b * c$



Directed Acyclic Graph: (DAG)

↳ It eliminates common sub-expression.



In DAG no assignment required.

Q1. $X = a + b - b + c$

postfix : $X\ a\ b + b - c + =$

3AC with min Varinables (optimization)

$a = a + c$ (2 variable)

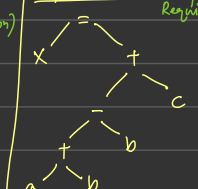
$X = a + b - b + c$

$X = a + c$

SSA min variable : (optimization)

$X = a + c$ (3 variable)

Syntax Tree : (No optimization Required)



min node min edges DAG

$X = a + b - b + c$

optimization

$X = a + c$

Q2. $X = (a + b) * (a + b)$

postfix : $X\ a\ b + a\ b + * =$

3AC with min Varinables (optimization)

$a = a + b$

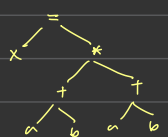
$a = a * a$ (2 variable)

SSA min variable : (optimization)

$t_1 = a + b$

$X = t_1 * t_1$ (4 variable)

Syntax Tree :



DAG : Min Edge Min Node

$P+Q+r = r+P+Q$

Reordering

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

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

Q3 $X = a*b + a + b*c$

3AC (min var)

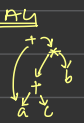
$t_1 = a*b$
$t_1 = t_1 + a$
$a = b*c$
$t_1 = t_1 + a$
(3 variable) Wrong

$C = a+c$
$C = C*b$
$C = C+a$
3 variable Correct

SSA (#min var)

$t_1 = a+c$
$t_2 = b*t_1$
$t_3 = a+t_2$
(6 variable)

DAG



Q4 $X = a+b*C-b$

3AC

$C = b*C$
$C = C-b$
$C = a+C$
(3 variable)

SSA

$C_1 = b*C$
$C_2 = C_1 - b$
$C_3 = a+C_2$
(6 var)

Three Address Code notation:

- ☐ Triplet Notation
- ☐ Quadruple Notation
- ☐ Indirect Triple Notation

(1) Triple Notation

if we need any value of address we have to compute it it can't store data

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

	operator	operand1	operand2	
	+	a	b	1000
	*	1000	C	1010
	+	1000	1010	1020
	=	1020		1030

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

Advantage: – takes less space

Disadvantage: – Computation takes much time

(2) Quadrapole Notation:

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

	operator	operand1	operand2	Result	
	+	a	b	x	1000
	*	x	C	y	1015
	+	x	y	z	1020
	=	z		w	1035

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

Advantage: – takes less time to compute

Disadvantage: – require more space

Code	Triple Not^{n}				Quadruple Not^{n}				
	op	arg1	arg2		op	arg1	arg2	result	
$t_1 = b*c$	*	b	c	0	*	b	c	t_1	
$t_2 = a+t_1$	+	a	(0)	1	+	a	t_1	t_2	
$t_3 = b*c$	*	b	c	2	*	b	c	t_3	
$t_4 = d/t_3$	/	d	(2)	3	/	d	t_3	t_4	
$t_5 = t_2 - t_4$	−	(1)	(3)	4	−	t_2	t_4	t_5	

Application: pointer (C, C++)

Advantage: Code movable relocation possible

Indirect Triple Notation:

(I)

	operator	operand1	operand2	
X=a+b	+	a	b	1000
y=X*C	*	6000	C	1010
z=x+y	+	6000	7000	1020
W=Z	=	8000		1030

(II)

Indirect Addr.	Actual Address
6000	1000
7000	1010
8000	1020
9000	1030

↳ if we have to move any line to another address we have to change table (II) only

Ex Three Address Code for If-else Statement

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

z = z * z;
```

≡

```
    _t0 = x < y;
    Ifz t0 Goto _L0;      → false
    z = x;                → True
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

IFNZ → if non zero

Ex Three Address Code for while Statement

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

≡

```
_L0:
    _t0 = x < y;
    Ifz t0 Goto _L1;
    x = x * 2;
    Goto _L0;
_L1:
    y = x;
```

Ex Three Address Code for Array

Assume declaration: A[n1, n2]

3AC for A[i,j] is:

$t_1 = n_2 * i$

$t_2 = t_1 + j$

$t_3 = t_2 * W$

t_4 = Base Address

$t_5 = t_4[t_3]$

↳ $t_4 + t_3$ Same

$$A[i,j] = \text{Base Address} + (n_2 * i + j) * W$$

n2 is the size of each row and
w is the size of each element

Base[ ]

***

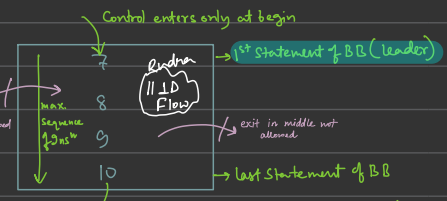
Control Flow Graph:

- It comprises of nodes & edges
- It is a collection of Basic Blocks and Controls
- It is represent flow of program execution using Basic Blocks

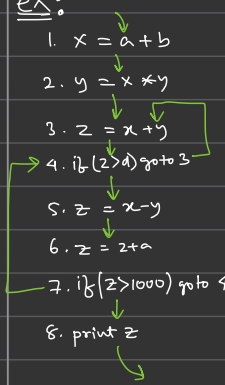
Basic Block (BB):

4 important points:

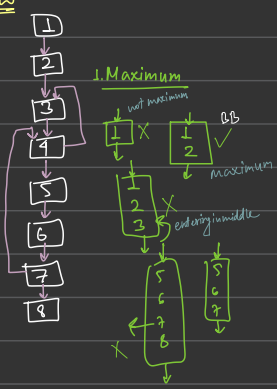
1. Max. Seq of Stmts
2. entry only at begin
3. exit only at end
4. no entry/exit at middle



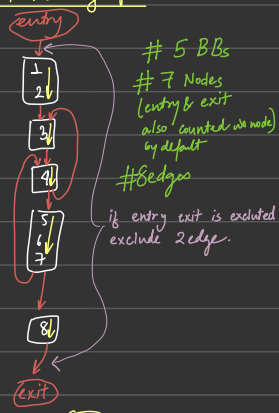
ex:



Flow



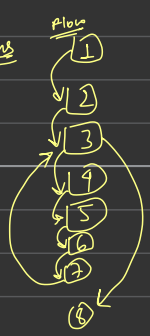
Control flow graph



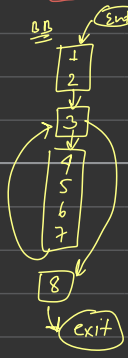
Q:

1. S := 0
2. i := 1
3. L1: if i > n goto L2
4. t := i * i
5. S := S + t
6. i := i + 1
7. goto L1
8. L2: return S

Ans



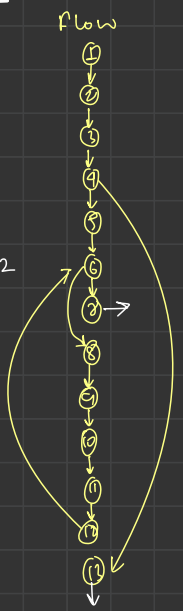
BB

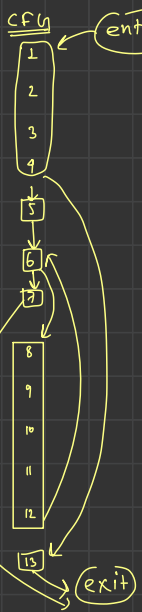


4 BBs
6 nodes
6 edges

Control Flow Graph:

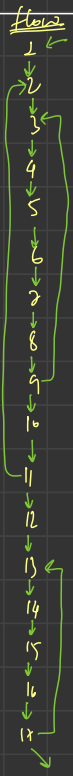
```
1     receive m (val)
2       f0 ← 0
3       f1 ← 1
4       if m <= 1 goto L3
5       i ← 2
6     L1: if i <= m goto L2
7       return f2
8     L2: f2 ← f0 + f1
9       f0 ← f1
10      f1 ← f2
11      i ← i + 1
12      goto L1
13    L3: return m
```

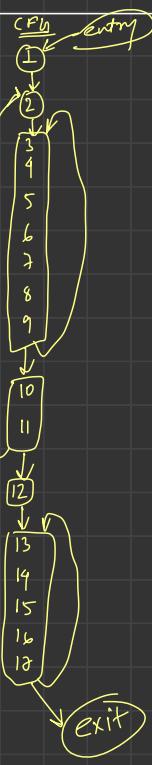




BBs = 6
nodes = 8
edges = 9

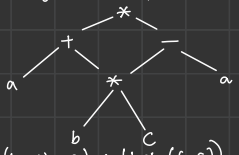
```
1   j = 1
2   j = 1
3   t1 = 10 * i
4   t2 = t1 + j
5   t3 = 8 * t2
6   t4 = t3 - 88
7   a[t4] = 0.0
8   j = j + 1
9   if j <= 10 goto (3)
10  i = i + 1
11  if i <= 10 goto (2)
12  i = 1
13  t5 = i - 1
14  t6 = 88 * t5
15  a[t6] = 1.0
16  i = i + 1
17  if i <= 10 goto (13)
```





BBs = 6
nodes = 8
edges = 10

Q. Find equivalent expression for the following syntax tree:

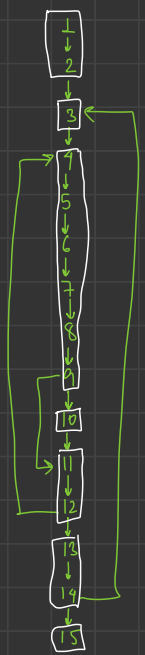


It is DAG

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

Q. Which one of the following correctly specifies the number of basic blocks and no of instruction in the largest basic block respectively?

L1: $t_1 = -1$
L2: $t_2 = 0$
L3: $t_3 = 0$
L4: $t_4 = 4 * t_3$
L5: $t_5 = 4 * t_2$
L6: $t_6 = t_5 * M$
L7: $t_7 = t_4 + t_6$
L8: $t_8 = a[t_7]$
L9: if t_8 <= max goto L11
L10: $t_1 = t_8$
L11: $t_3 = t_3 + 1$
L12: if $t_3 < M$ goto L4
L13: $t_2 = t_2 + 1$
L14: if $t_2 < N$ goto L3
L15: max $= t_1$



A) 6 and 6
B) 6 and 7
C) 7 and 7
D) 7 and 6

Technique

Data flow Analysis

Code Optimization

To Save Space

To Save Time

Code Optimization

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

In Compiler

Code Optimization Type

- Local optimization
  - Statement level (how many statement in intermediate code)
  - Block level [Basic Block, Peephole] (how many statement in assembly code)
  - Loop level
- Function Level [Intra procedural]
- Program Level [Inter procedural]
- Global Optimization

Code Optimization Techniques

1. Constant Folding
2. Copy Propagation → Constant Propagation, Variable Propagation
3. Common Subexpression Elimination
4. Strength Reduction
5. Algebric Simplification
6. Dead Code elimination
7. Loop optimization → Code Motion, Induction variable elimination, loop Merging, loop unrolling

1. Constant Folding

$x = 2*3 + y \Rightarrow x = 6 + y$ (Const. Folding)

$x = y*3 + 2$ → not possible

2. Copy Propagation

$x = 20$, $y = x + a$ ⇒ $x = 20$, $y = 20 + a$ ⇒ $y = 20 + a$ (Dead Code elimination)

Constant propagation: replace x with 20

$x = b$, $y = x + a$ ⇒ $x = b$, $y = b + a$ ⇒ $y = b + a$ (Dead Code elimination)

Q. $x = 10$, $y = x * 2 - b$

⇒ $x = 10$, $y = 10 * 2 - b$ (Const. propagation)

⇒ $x = 10$, $y = 20 - b$ (Const. Folding)

⇒ $y = 20 - b$ (variable propagation: replace x with 10; dead code elimination)

Find possible optimization :-

a) Constant Folding
b) Copy propagation
c) Dead code elimination
d) all of These

③ Common Sub Expression Elimination:

↳ DAG can be used

↳ Available expression data flow analysis can be used

$X = (a+b) * (a+b)$ ⇒ $t_1 = a+b$, $X = t_1 * t_1$

2 add, 1 mull ⟶ 1 add, 1 mul

④ Strength Reduction:

↳ It replaces costlier Inst^n with cheaper one.

$x = a*2$ ⇒ $x = a+a$ (cheaper) or $x = a<<1$ (cheaper) — costly

$x = a*8$ ⇒ $x = a<<3$

$x = a/8$ ⇒ $x = a>>3$

⑤ Algebraic Simplification:

Cancellation i) $X = a+b-b+c$ ⇒ $X = a+c$

Identity ii) $X = a+b*1$ ⇒ $X = a+b$

Identity iii) $X = a+b+0$ ⇒ $X = a+b$

domination iv) $X = a+b+c*0$ ⇒ $X = a+b$ (Dom, Identity)

$x = 10$, $y = x*1+c$ → 1) Copy Prop 2) dead code → $y = 10*1+c$ → Const Folding → $y = 10+c$

1) Algebric Simp → $x = 10$, $y = x+c$ → 1) copy prop. 2) dead code → $y = 10+c$

⑥ Dead Code Elimination:

$x = a+b$ ← dead code

$y = a*b$

$z = y+c$

print (z)

if (2>3) print("you") else print("Me") → Unreachable → print("Me") (Strength Reduction)

Q. optimization Possible

✓ Constant Folding (2>3 ⇒ 0, print "you")

✓ Dead Code Elimination

✓ Strength Reduction (loop → no loop)

⑦ Loop Optimization:

i) Code Motion

Identify loop invariant code (does not depend on loop) and move outside loop;

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

⇓

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

(n-1 cycle saved)

ii) Induction Variables 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
}
```

(1) (j) (K) (4) (3) (2) (5)

Induction Variable:

↳ It depends on iteration (value changes in some iteration)

need not every

eg i, j, k

n, a, b, c } not induction variable

⇓ (j, K eliminated)

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

iii) Loop Merge / Fusion:

```
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<4*n;i++)
{ printf("GATE");
}
```

(4*n: half; printf("GATE"): double)

⇓ Unroll ①

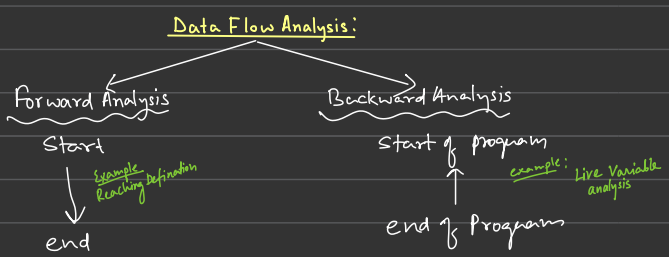
```
for(i=0; i<2*n;i++)
{ printf("GATE");
  printf("GATE");
}
```

⇓ Unroll ②

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

Data Flow Analysis:

- To analyze program
- To understand each variable
- Lattices can be used to analyze data
- Control Flow Graph can help to analyze data



- What is live variable?
- How to compute live variables at any statement?
- Back-ward analysis:
  - → Compute GEN(USE) & Kill(DEF) for each Basis Block
  - → Compute IN and OUT for each Basic Block

Live Variable:

Remember → blind
Understand → Mind
Apply → Mind + Heart
Analyze → time
evaluate → cross check
Create → Confidence

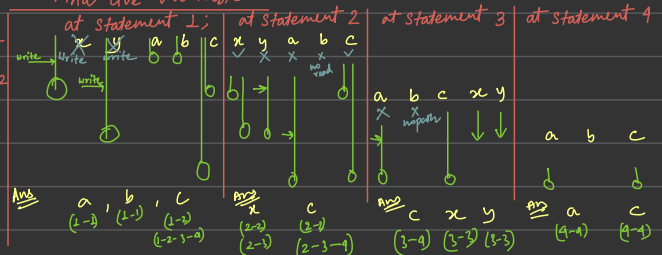
At Statement (Just before statement)

x is live variable at statem Si iff
I) There exist statatement Sj that reads x
II) There exist a path from Si to Sj
III) There is no assignment into x before Sj
no write
no defination

10. write x = (x+a); read
Si
Sj

Q. Find Live Variable

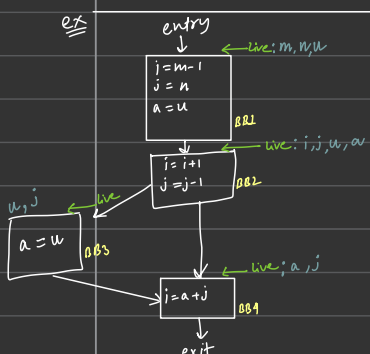
ex 1. x = a+b; ←1
2. y = x*c; ←2
3. a = x+y; ←3
4. b = a-c; ←4



ex

1	a = a+b	live at 1: a(1-1) b(1-1); (1-2)
2	b = a +b	live at 2: a(2-2) (2-3) b(2-2)
3	c = b + a	live at 3: a(3-3) b(3-3)

ex

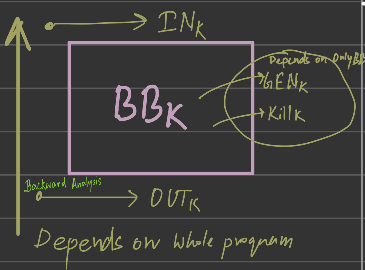


Live Variable Analysis:

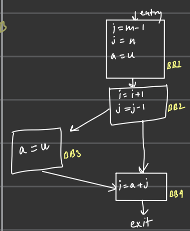
GEN SET & KILL SET:

Compute Gen & Kill Set For Each Basic Block

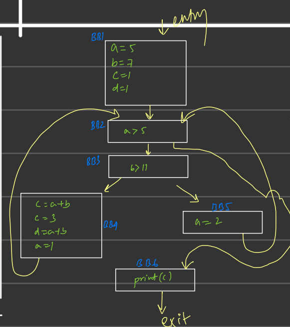
Computed for BasicBlock



Gen(USE) Set (→ read / use)	Kill(DEF) Set (→ redefination / assignment)
BB_K: $x = a + b$, $y = x * c$, $z = y - d$ $GEN_K = \{a, b, c, d\}$ $Gen(K) = \{ l \mid l$ is used (read) at some statement in BB_K and no defination before the statement $\}$	$KILL_K = \{x, y, z\}$ $= \{ v \mid v$ has a definition in $BB_K \}$



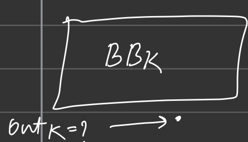
	GEN	KILL
BB_1	m, n, u	i, j, a
BB_2	i, j	i, j
BB_3	u	a
BB_4	a, j	i



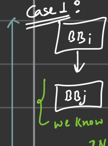
	GEN (use)	KILL (DEF)
BB_1	ϕ	a b c d
BB_2	a	ϕ
BB_3	b	ϕ
BB_4	a b	c d a
BB_5	ϕ	a
BB_6	c	ϕ

IN & OUT SET:

OUT_K for BB_K:

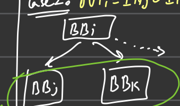


Case 1: $OUT_i = IN_j$



we know every thing already IN, OUT

Case 2: $OUT_i = IN_j \cup IN_K \dots \cup IN.$



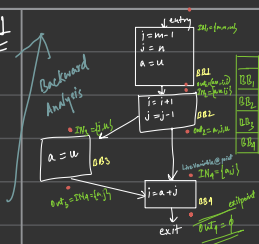
we know every thing already GEN, KILL, IN, OUT.

IN_K for BB_K:

$IN_K = (OUT_K - KILL_K) \cup GEN_K$

Compute IN & OUT Set for the given CFG:

Q1

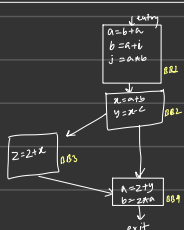


	GEN	KILL
BB_1	m,n,u	i,j,a
BB_2	i,j	i,j
BB_3	u	a
BB_4	a,j	i

	IN	OUT
BB_1	m,n,u	a,u,i,j
BB_2	a,u,i,j	a,u,j
BB_3	j,u	a,j
BB_4	a,j	ϕ

Q2

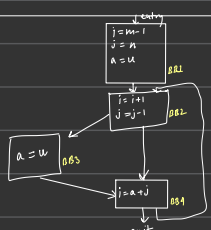
IN = OUT - KILL ∪ GEN



	GEN	KILL
BB_1	a,b,i	a,b,j
BB_2	a,b,c	x,y
BB_3	z,x	z
BB_4	z,y	a,b

	IN	OUT
BB_1	a,b,c,z,i	a,b,c,z
BB_2	a,b,c,z	x,y,z
BB_3	x,y,z	z,y
BB_4	z,y	ϕ

Q3

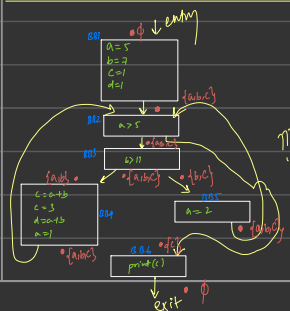


	GEN	KILL
BB_1	m,n,u	i,j,a
BB_2	i,j	i,j
BB_3	u	a
BB_4	a,j	i

	IN	OUT
BB_1	{m,n,u}	{a,u,i,j}
BB_2	{a,u,i,j}	{a,u,j}
BB_3	{a,j,u}	{a,j} {a,u,j}
BB_4	{a,j} {a,u,j}	ϕ {a,u,i,j}

Q4

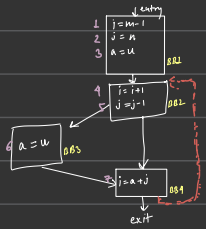
HW



??

	GEN	KILL	IN	OUT
BB_1	ϕ	a,b,c,d	ϕ	a,b,c
BB_2	a	ϕ	a,b,c	a,b,c
BB_3	c	ϕ	a,b	a,b,c
BB_4	a,b	c,d,a	a,b	ϕ a,b,c
BB_5	ϕ	a	ϕ b,c	ϕ a,b,c
BB_6	c	ϕ	c	ϕ

LIVE RANGE :



At statement 1 : Live Range(i) = ?

⇒ 1 → 2 → 3 → 4

At statement 4 : Live Range(i) = ?

⇒ Nil as (Not Used later)

At statement 4 : Live Range(i) = ?

after the loop live range

= Nil as Redefination in 7

At statement 7 : Live Range(i) = ?

after loop 7 → 4

Live Rang of J at Stmt 2 ?

2 → 3 → 4 → 5

Live Rang of J at Stmt 5?

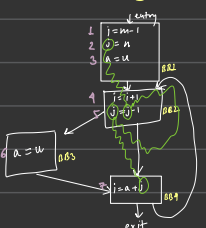
5 → 6 → 7 → 5

5 → 7 → 5

5 → 6 → 7

5 → 7

Reaching Defination: (only Link not Path)



I) USE-DEF CHAIN :

AT STMT 5 : for j → 5→2, 5→5 (6→6→7→4→5, 5→7→4→5)

AT STMT 7 : for i → 7→5

II) DEF-USE CHAIN :

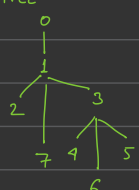
@1 for i : 1 → 4

@5 for j : 5→5, 5→7

Domination Tree : always start from begin

BB	Dominated B
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

Tree



Code Generation

- Dependent on m/c
- Register, addresses, Instruction, Addressing mode
- Register Allocation
  - Requires Program Analysis
  - Uses Graph Coloring

Runtime Environment

- Parameter Passing
  - Call by Value
  - Call by Reference
- Memory Allocation
  - Static
  - Stack } Dynamic
  - Heap }
- Activation Record