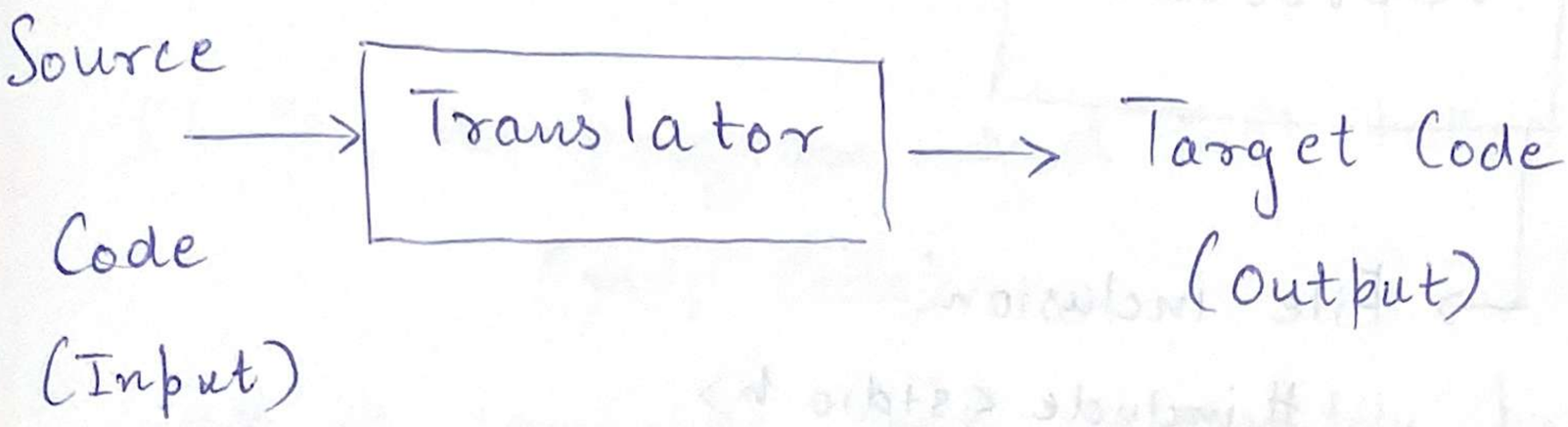
Compiler Design

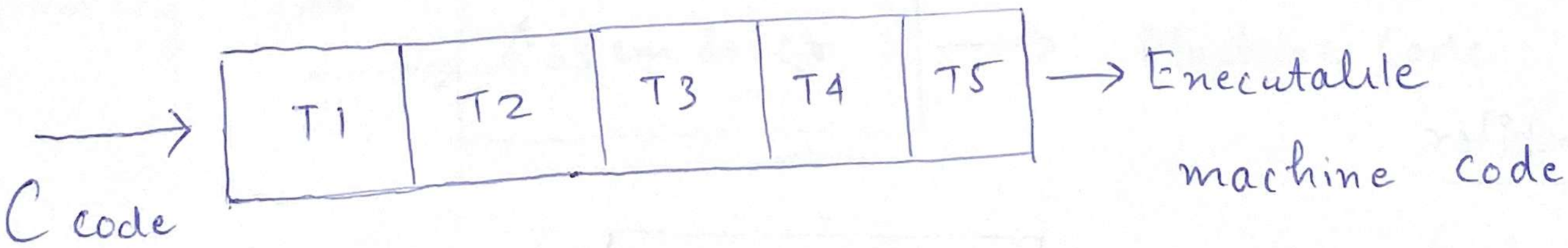
Translators

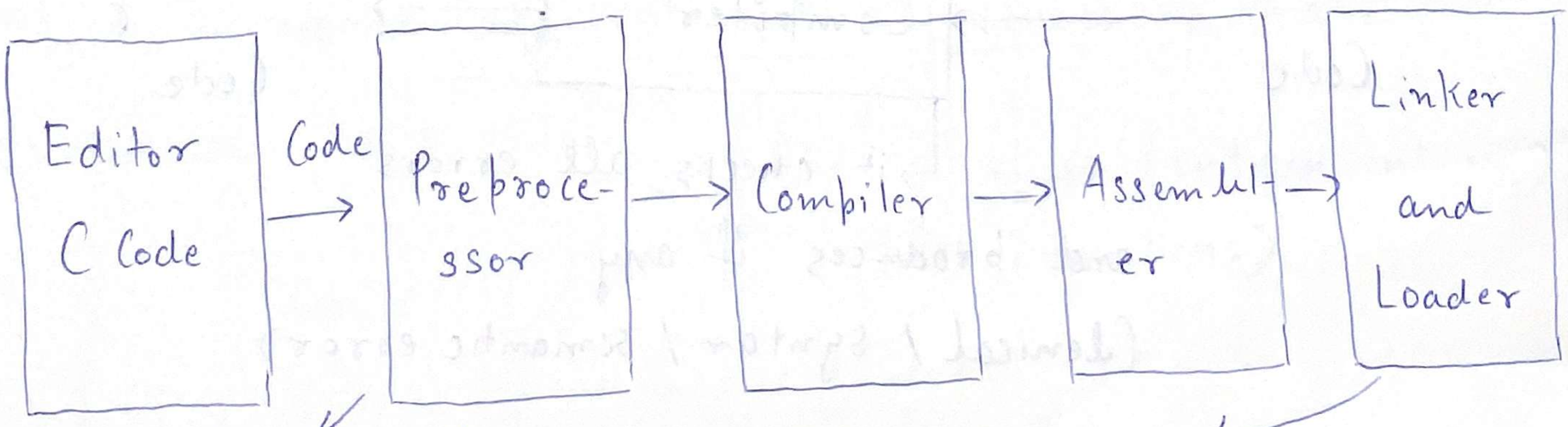


Eg :-

1) Compiler
2) Interpreter
3) Preprocessor
4) Assembler
5) Editor
6) Linker / Loader, etc.

C Language Processing System





Preproccesor:

error if any command is not understood

(# includ <stdio.h>
define MAX 100;)

P1.C

```
# ...
# ...
# ...
```

→ Pre processor

→ File inclusion

include <stdio.h>

include < P2.C>

→ Marcro substitution

define MAX 100

→ Conditional macros

if, # endif, # else

```
#define MAX 100
int A[MAX];
void main() {
    int n = MAX;
}
```

→

```
int A[100];
void main() {
    int n = 100;
    :
}
```

Compiler

Absolute C Code → Compiler → Assembly Code

↳ it checks all errors and produces if any (lenical / syntax / semantic error)

What is compiler?

C program → Compiler program → Assembly Program

It is a program (m/c Code) and that takes i/p as C program and it converts into assembly program, and produces error if any.

(M/C Independent)

High Level Code → Compiler (Low level Code) (M/C dependent) → Assembly Code (M/C dependent)

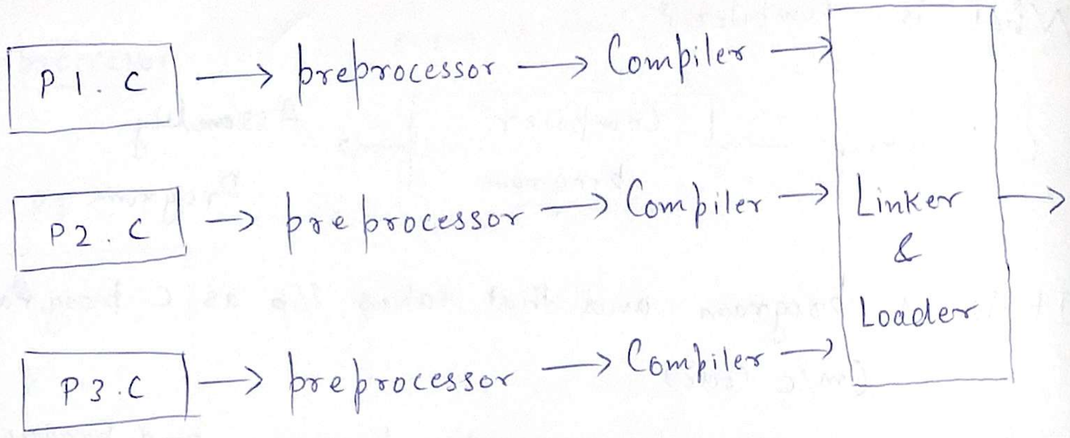
Assembler:

Assembly Code → Assembler → (Relocatable) Machine Code.

Linker: To resolve all external references to make executable code

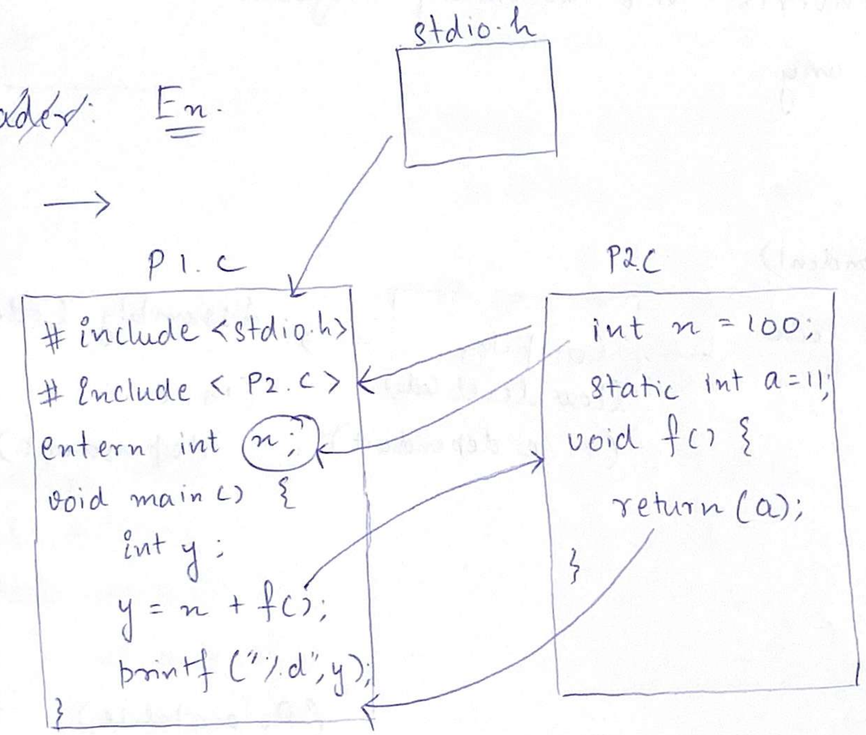
↳ variables (extern int n;)

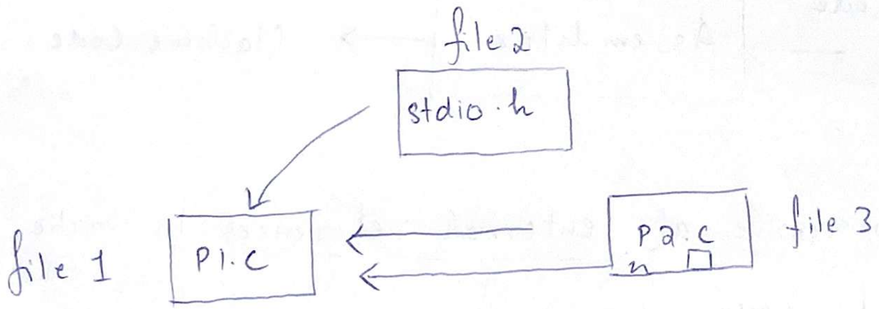
↳ files (functions)



~~Loader~~: Ex.

→



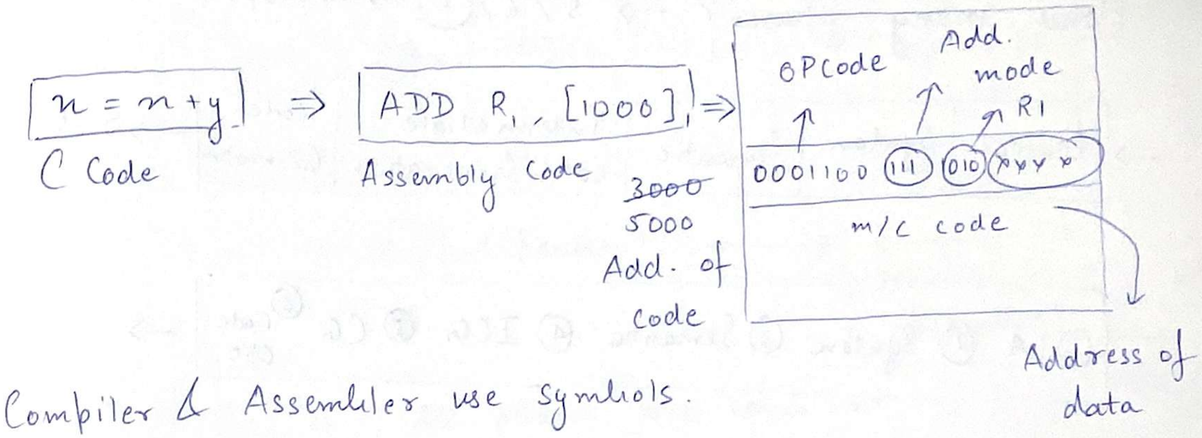


external references

For P1.C : P2.C, Stdio.h } external files
n } external variables

Loader:- It relocates the data / code

It alters the address of data / address of code.



Compiler & Assembler use symbols.

Preprocessor
Compiler
Assembler
Linker } Static / dynamic
Loader

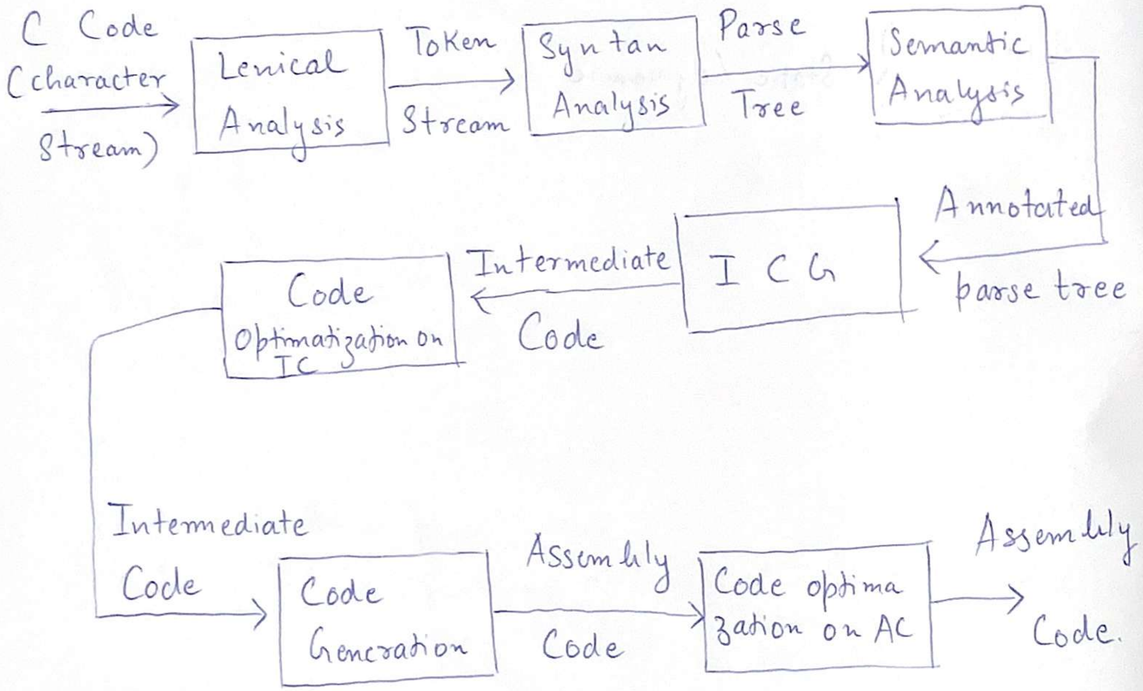
Phases of Compiler

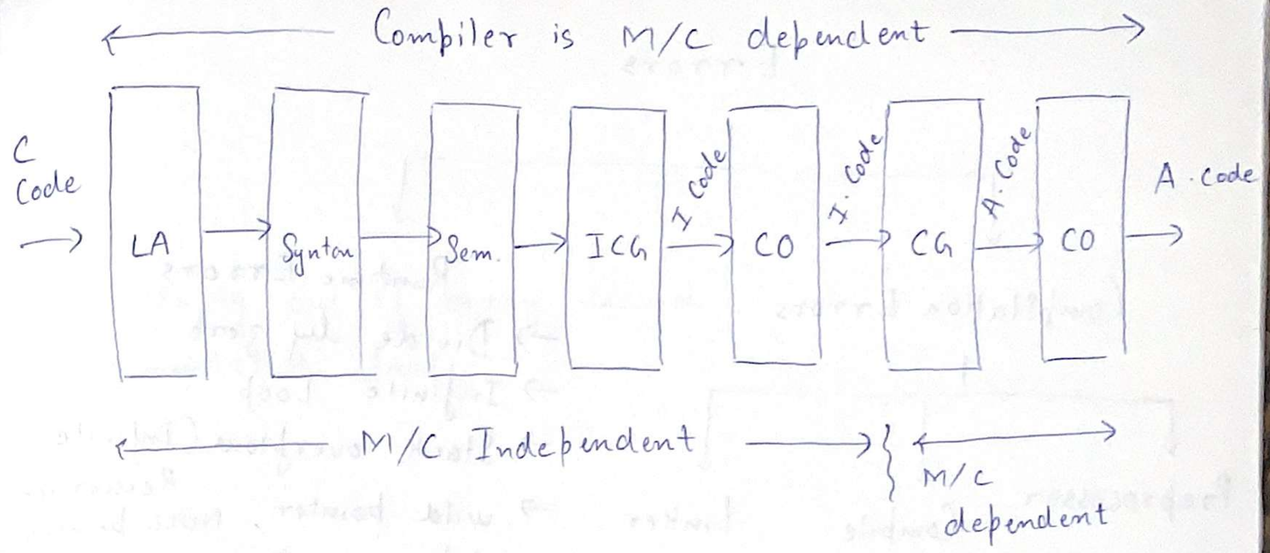
How many phases? → 5/6/7 → appropriate

→ | Lenical | Syntan | Semantic | Intermediate code Generation | Code Generation | → 5 phases

→ ① LA ② Synton ③ Semantic ④ ICG ⑤ CG ⑥ Code Opt. →

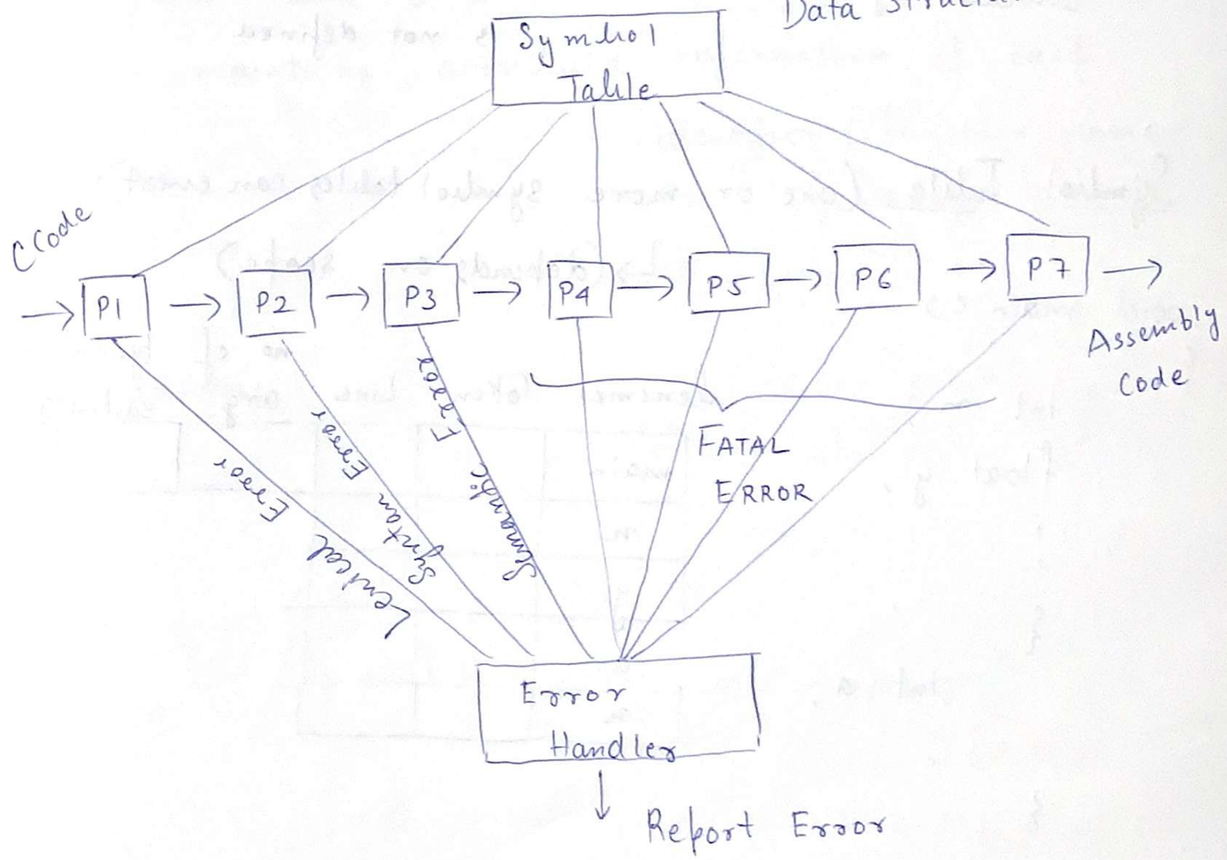
→ | LA | Synton | Semantic | ICG | Code Opt. | Code Generation | Code Optimazation | → 7 phases





Pass - 1 Compiler : All phases are combined

Two Pass Compiler : All phases are devided into two modules.



Errors

- Compilation Errors
  - Preprocessor error
    - # inlude
  - Compile Error
    - Lexical
    - Synton
    - Semantic
  - Linker Error
    - ↳ If any external file not exist, external variable is not defined.
- Runtime Errors
  - → Divide by zero
  - → Infinite Loop
  - → Stack overflow (Infinite Recursion)
  - → wild pointer, NULL pointer deference, Dangling pointer,
  - → accessing out of array bound
  - → segmentation fault.

<u>Symbol Table</u> (one or more symbol tables can exist)

↳ (depends on scope)

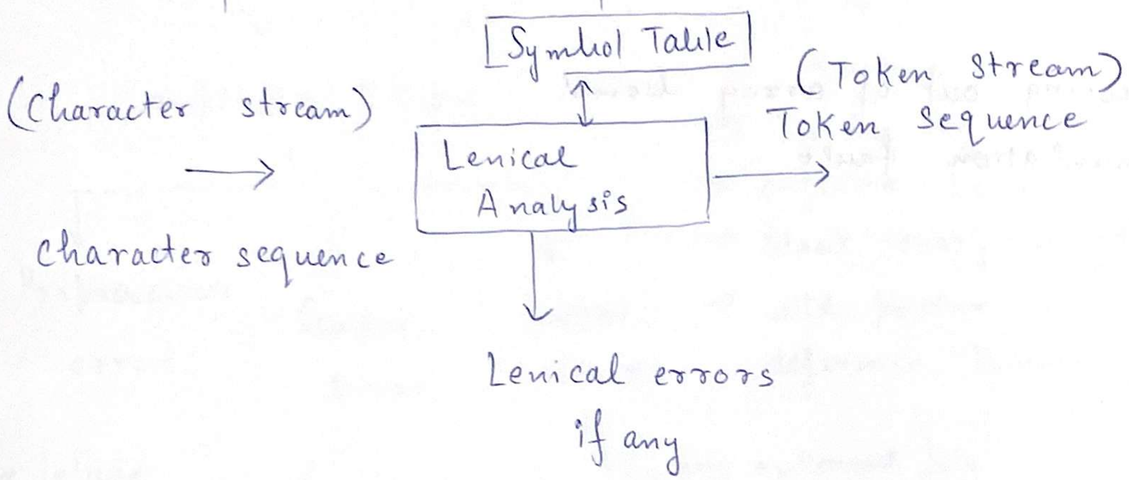
```
void main ()
{
    int x;
    float y;
    :
    {
        int a;
        :
    }
}
```

lexemes	Token	Line	no. of arg.	type of each arg.	return type	address
main						
x						
y						
a						
x						

→ Symbol Table is a data structure.

→ It maintains attribute information of each identifier (function name / variable name)

Lenical Analysis (Scanner / Token Recognizer / Token producer / Linear phase)



Functionality of Lenical Analysis

→ It scans whole program (each and every character).
→ It groups characters into tokens
→ It makes an entry of identifier into a symbol table
→ It recognizes tokens
→ It ~~recog~~ identifies comments and spaces
→ It produces tokens.
→ It also identifies lenical error if any
→ It uses maximal munch rule (longest prefix match)
→ It ignores / delets comments and spaces.

① Identifies tokens, comments & white spaces
② It produces tokens
③ It reports lenical error if any.

Leneme & Tokens

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

Lenemes		Tokens
int	→	Key_1
a		id_1
;		ss_1
main		id_2
(		ss_2
)		ss_3
{		ss_4
int		Key_1
x		id_3
;		ss_1
float		Key_2
y		id_4
=		op
10		C
;		ss_1
}		ss_5

Types of Tokens

① Identifiers
② Keywords
③ Operators : (++) 2 char, 1 token
④ Constants
⑤ Special Symbol (punctuations)

$z = x ++ y$

↳ maximal munch rule

(longest prefix match)

Finding number of Tokens

1.

```
void main ()
{
    x = x + y ;
}
```

12 tokens

no lexical error

semantic error

2. $x = ++y;$ $x, =, ++, y, ;$

5 tokens

3. $x += y -= z;$ 6 tokens

4. ~~x =~~

char * x = "gate"; 6 tokens

" ---- "

1 constant

5. printf ("gate = %d", x); 7 tokens

6. int = float + char; 6 tokens

(Syntax error)

7. int x = 10; /* comment */ 5 tokens

T_1 T_2 T_3 T_4 T_5 ignored

int x = 10; → [Lexical Analyzer] → T1 T2 T3 T4 T5

8.

```
void main()
{
    int x;
    float y;
    scanf ("%d%f", &x, &y);
    printf ("%d%f", x, y);
}
```

32 tokens

9. x << = y; 4 tokens

↳ (x = x << y) meaning

Maximal munch rule

<<= } 1 token

longest prefix match.

~~10. x? y:z;~~

12. int * * p; → ss 5 tokens

Keyword Opr identifier

11. x[2];

(x, [, 2, ;) = 4 tokens

10. x ? y : z ;

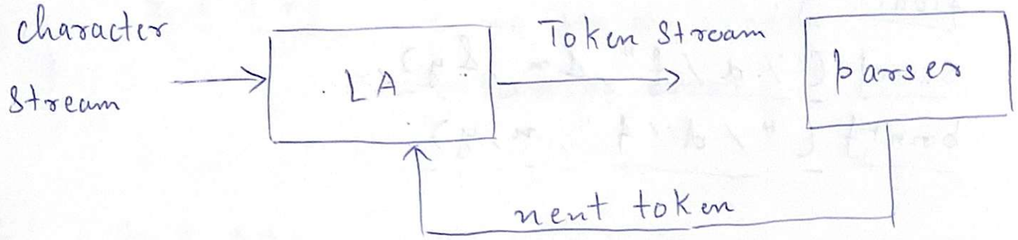
Lenical Analysis alone can't identify

Lenical analysis & parser all combined

⇒ (x, ?:, y, z, ;) = 5 tokens

Parser help is required to understand this code.

~~Lenical Error~~



Note

* → 1 token	+ → 1 token
* * → 2 token	++ → 1 token
* * p → 3 token	+++ → 2 token

13. +++a; ⇒ 4 tokens

Lenical error

① int 1abc; (Lenical error)

↳ Invalid sequence

2. char ch = 'a; (Lenical error)

↳ end quote is missing

3. /* ... ... (Lenical error)

/* Begin

*/ end pattern is missing

4. int @$# (Lenical error)

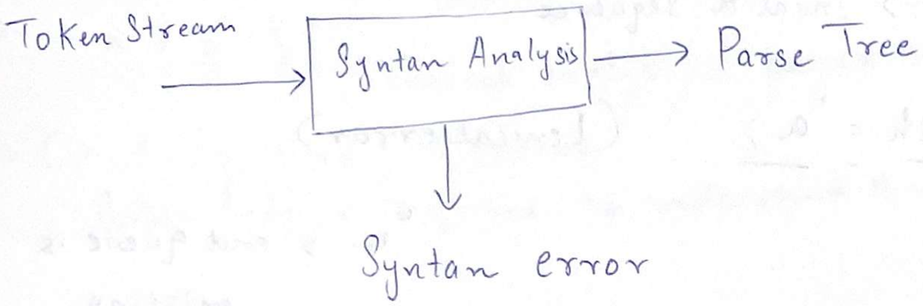
↳ These are not tokens

Invalid sequences

5. char *p = "gate; end is missing

(lenical error)

Syntan Analysis / Parsing / Syntan Verifier / Parse Tree Generator



Char stream

```
main ()
{
    int n;
    n = 10;
}
```

Id → T1
(→ T2
) → T2
{ → T2
} → T2
int → T3
10 → T4
= → T5

T1 T2 T2 T2 T3 T1 T2
T1 T5 T4 T2 T2

Token stream

Syntan Error

1. main (No lenical error, Syntan Error } compilation error
   {
   }

2. main () Syntan Error
   {
   int n = ;
   }

3. if ( ) ; expression is missing Syntan error

4. while ( ) ; Syntan Error

5. for (, ,) ; Syntan Error

6. main ()
   {
   { }
   { }
   {
   } } missing closed one of main() Syntan Error

7.

```
void main()
{
    int x, y;
    x = 2   ---> Syntan Error
    y = 3;
}
```

Compile error
- → lenical error
- → syntan error
- → semantic error

Syntan error → Structure is not proper

Lenical error → Invalid sequence of character

Semantic error → Type related error (meaning)

Content Free Grammer

1. What is CFG

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

- V → set of variables
- T → set of terminals (tokens)
- P → set of roles (productions)
- S → start symbol $S \in V$

$P: V \rightarrow (V \cup T)^*$

Eg :-

1.
$$\left.\begin{array}{l} S \rightarrow AB \\ A \rightarrow aA \mid b \\ B \rightarrow b \end{array}\right\} \text{4 productions}$$

$X \rightarrow \alpha \mid B \Leftrightarrow X \rightarrow \alpha$
$X \rightarrow B$

$V \rightarrow$ Any → any sequence of terminals & non terminals.

V → Exactly (only) one non-terminal

2. $S \rightarrow SS \mid a$

2. Derivation of a string

→ Linear Derivation
→ LMD (left most derivation)
→ RMD (right most der.)

→ Non - linear derivation
(Parse Tree) (Derivation Tree)

Eg

→ start symbol

$S \to AB$
$A \to a$
$B \to b$ } CFG

$w = ab$ } string

LMD

lest most non terminal

$S \Rightarrow \underline{A}B \Rightarrow a\underline{B} \Rightarrow \underline{ab}$

$\boxed{S \to AB}$ $\boxed{A \to a}$ $\boxed{B \to b}$

↳ string (sentence)

Derivation

Left Sentential forms

S AB aB ab

No. of steps in this derivation = 3 = no. of substitution. = length of derivation

RMD

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

$S \to AB$: $S \Downarrow \underline{AB}$ → rightmost non-terminal

$B \to b$: $\Downarrow \underline{Ab}$ → rightmost " "

$A \to a$: $\Downarrow ab$

Derivation

Right sentiantial form

S, AB, Ab, ab

length of RMD = no. of substitutions in RMD = no. of steps = 3

$S \to AB$

$A \to a$

$B \to b$

LMD: $S \to AB \to aB \to ab$

RMD: $S \to AB \to Ab \to ab$

Note:- For given string LMD RMD need not be same

$S \to a$ LMD: $S \to a$, RMD $S \to a$ } In the eg., both are same.

Parse Tree / Derivation Tree

$S \to AB$

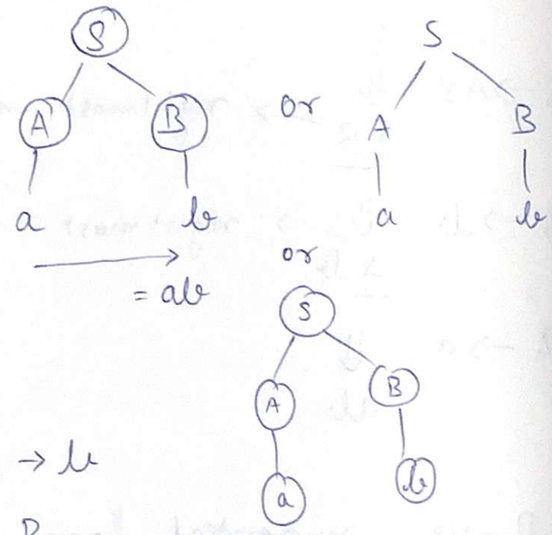
$A \to a$

$B \to b$

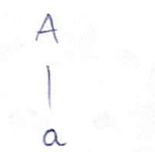
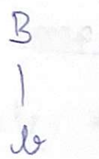
5 nodes

2 leaf nodes

3 non-leaf nodes



Production Tree

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

Length of derivation = No. of non leaf nodes

= no. of substitution

= 3

$S \to a \mid AB$

$A \to Ab \mid \epsilon$

$B \to a$

} Given CFG

$w = a$ } Given string.

How many derivations for $w = a$?

How many LMDs for $w = a$?

How many RMDs for $w = a$?

How many parse tree for $w = a$?

} same

No. of derivations = no. of LMDs = no. of RMDs = no. of parse trees.

LMDs

1. $S \to a$

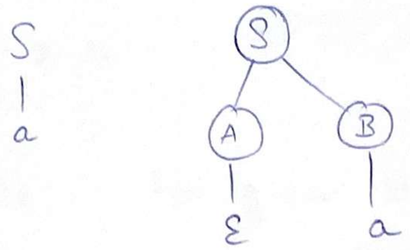
2. $S \to AB \to B \to a$

RMDs

1. $S \to a$

2. $S \to AB \to Aa \to a$

Parse Trees :-



Ambiguous CFG

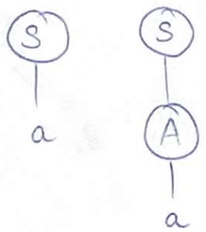
G is ambiguous iff for some string $\in L(G)$, more than one parse tree. (more than 1 LMD)
(more than 1 RMD)
(more than 1 derivation)

Ex.

$S \rightarrow a \mid A$

$A \rightarrow a \mid b$

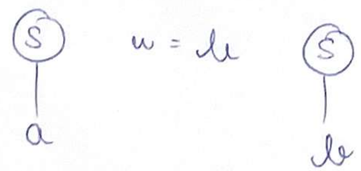
$w = a$



Unambiguous CFG

CFG G is unambiguous iff every string $\in L(G)$ has exactly 1 parse tree.

$S \rightarrow a \mid b$ $\quad w = a$



Eg

1) $S \rightarrow (L) \mid a$ $\quad$ Unambiguous

$L \rightarrow LS \mid S$

a , (a) , - - -

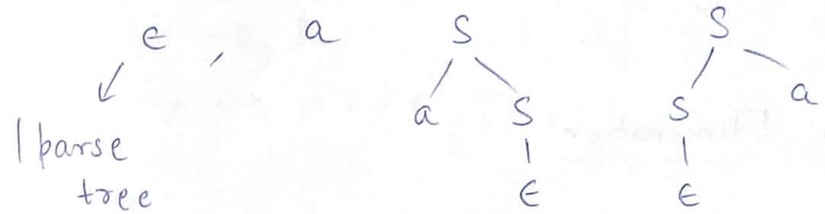
↓ 1 parse tree $\quad$ ↓ 1 parse tree

2) $S \rightarrow AA$ $\quad$ Unambiguous

$A \rightarrow aA \mid b$

bb, abb, bab, - - -

3) $S \rightarrow aS \mid Sa \mid \epsilon$ $\quad$ Ambiguous



$a\epsilon = a$

4) $S \rightarrow aS \mid Sa \mid ab$ $\quad$ Ambiguous

Same non-terminal participate in both left and right recurrsion then grammer is ambiguous.

$S \rightarrow S \ldots$ (left recurrsion) $\quad$ $S \rightarrow \ldots S$ (right)

5.) $S \rightarrow \underline{S}\,\underline{S} \mid (S) \mid a$ Amligous

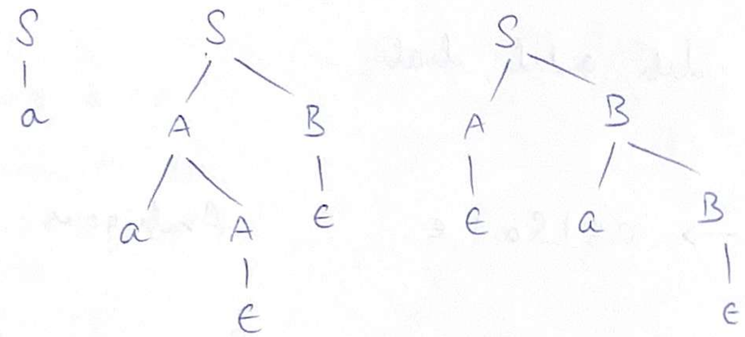
6.) $E \rightarrow \underline{E} + \underline{E} \mid E * E \mid id$ Amligous

7.) $S \rightarrow AB \mid a$

$A \rightarrow aA \mid \epsilon$ Amligous

$B \rightarrow aB \mid \epsilon$

a : > 1 Parse Tree



Left Recursion Elimination

Q Why do you eliminate left recursion?

→ Predictive (top down) parser do not support left recursive grammer

→ It helps in conversion from arlitrary CFG to GNF CFG.

Q How ~~may~~ many types of left recursions?

→ Direct left recursion

$S \rightarrow Sa \mid b$

→ Indirect left recursion

$S \rightarrow Ab$

$A \rightarrow Sa \mid b$

How to elimate direct left recursion (converting left recursion to right recursion)

1. $\underline{A} \rightarrow \underline{A}a \mid b$

$\Rightarrow L = \{ b, ba, baa, \ldots \}$

$= ba^*$

$X \rightarrow bY$

$Y \rightarrow aY \mid \epsilon$

$A \rightarrow Aa \mid b$ (left rec A)

$\Downarrow$

$A \rightarrow bA'$

$A' \rightarrow aA' \mid \epsilon$ (Right recc. using A')

2. $A \rightarrow Aab \mid cde$

$\Downarrow$

$A \rightarrow cdeA'$

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

(with null rule)

or

$A \rightarrow cdeA' \mid cde$

$A' \rightarrow abA' \mid ab$

(without ϵ)

3. $A \rightarrow Aab \mid Acde \mid f$

$\Downarrow$

$A \rightarrow fA'$

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

4. $A \rightarrow \overbrace{Aa \mid Ab}^{\text{left recc. terms}} \mid \underbrace{cd \mid e \mid fg}_{\text{non left recc terms}}$

$\Downarrow$

$A \rightarrow cdA' \mid eA' \mid fgA'$

$A' \rightarrow aA' \mid bA' \mid \epsilon$

General rule for direct left recursion

(no ϵ in given grammer)

$$A \rightarrow \underbrace{A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n}_{\text{Left recc. terms}} \mid \underbrace{B_1 \mid B_2 \mid \cdots \mid B_K}_{\text{non left recc. terms}}$$

$A \rightarrow B_1A' \mid B_2A' \mid \cdots \mid B_KA'$

$A' \rightarrow \alpha_1A' \mid \alpha_2A' \mid \cdots \mid \alpha_nA' \mid \epsilon$

How to eliminate indirect left recursion

$S \rightarrow A\alpha$

$A \rightarrow S\beta \mid \gamma$

$S \rightarrow$ S productions

$A \rightarrow$ A productions

$B \rightarrow$ B productions

$C \rightarrow$ C productions

Step 1. Choose order of productions

S, A, B, C

Step 2: Choose S

Apply direct left recc. elimination

Step 3: Choose A

i) Substitute S in A production

ii) Apply direct left recc. elimination

Step 4. Chose B

i) Substitute S, A [recc.]

ii) Apply direct left recc. elimination.

Step 5. Choose C

i) Substitute S, A, B [recc]

ii) Apply direct left recc. elimination.

Eg $S \to Sa \mid Ab$

$A \to Sa \mid b$

Step 1 : S, A (choose any order)

Step 2. choose S productions

$S \to Sa \mid Ab$

$\Downarrow$

$S \to Ab\,S'$ — ①

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

Step 3. Choose A productions

$A \to Sa \mid b$ [substitute only if 'S' appears in first position]

Substitute S $\Downarrow$

$A \to Ab\,S'a \mid b$

$\Downarrow$ Eliminate direct left recc.

$A \to bA'$ — ②

$A' \to bS'aA' \mid \epsilon$

① + ②

$S \to Ab\,S'$

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

$A \to bA'$

$A' \to bS'aA' \mid \epsilon$

eg $S \to ab \mid Ab$

$A \to Sa \mid b$

Step 1 $\xrightarrow{S,\ A}$

Step 2 choose 'S'

$S \to ab \mid Ab$ — ①

no left recc.

Step 3. Choose A

$A \rightarrow \underline{Sa} \mid b$

Substitute 'S' productions

$A \rightarrow aba \mid Aba \mid b$

Eliminate left recc.

$A \rightarrow abaA' \mid bA'$ — ②

$A' \rightarrow baA' \mid \epsilon$

① + ②

$S \rightarrow ab \mid Ab$

$A \rightarrow abaA' \mid bA'$

$A' \rightarrow baA' \mid \epsilon$

Eg $S \rightarrow Sa \mid Ab$

$A \rightarrow aa \mid b$

Step 2:- $\underline{S} \rightarrow \underline{Sa} \mid Ab$

⇓ Eliminate left recc.

$S \rightarrow Abs'$

$S' \rightarrow as' \mid \epsilon$ — ①

Step 3 $A \rightarrow aa \mid b$ — ②

① + ②

$S \rightarrow Abs'$

$S' \rightarrow as' \mid \epsilon$

$A \rightarrow aa \mid b$

<u>Left Factoring</u> / Eliminate common prefixes of 1 length.

1. $S \rightarrow a \mid b$ → left factored CFG

Unambiguous CFG → $\boxed{\text{left factoring}}$ → left factored CFG

$S \rightarrow a \mid ab$

Non left factored CFG

$S \rightarrow ax$

$x \rightarrow \epsilon \mid b$

Left factored

2. $S \rightarrow c \mid a \mid ab \xrightarrow[\text{factoring}]{\text{left}}$ $S \rightarrow c \mid ax$, $x \rightarrow \epsilon \mid b$

(Non left factored) (left factored CFG)

3. $S \to a \mid ab \mid abc$ $\xrightarrow[\text{factoring}]{\text{left}}$ $S \to aX$, $X \to \epsilon \mid b \mid bc$

(Non left factored)

$\xrightarrow[\text{factoring}]{\text{left}}$

$S \to aX$
$X \to \epsilon \mid bY$
$Y \to \epsilon \mid c$

4. $S \to A\,a \mid bcA$
$A \to bd \mid e$

(non-left factored CFG)

Substitute A

$S \to bda \mid ea \mid bcA$

$S \to bX \mid ea$
$X \to da \mid cA$

$\Rightarrow$ $S \to bX \mid ea$
$X \to da \mid cA$ (Ans)
$A \to bd \mid e$

(left factored CFG)

First Set Computation

First Set (X) : It is set of terminals including ϵ where each terminal is derived as first symbol from X.

$X \Rightarrow a\ldots$
$X \Rightarrow b\ldots$
$X \Rightarrow d\ldots$
$X \Rightarrow \epsilon$

First (X) = $\{a, b, d, \epsilon\}$

1. $S \to ab \mid cde$

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

2. $S \to Ab$
$A \to cde$

First (S) = First (A) = $\{c\}$ if A doesn't derive ϵ

First (A) = $\{c\}$

I.) First $(abc) = \{a\}$

II.) First (Abc) = First (A) if A doesn't derive ϵ ($\epsilon \notin$ First (A))

III.) First $(Abc) = (\text{First}(A) - \{\epsilon\}) \cup \{b\}$ if A derives ϵ i.e. $\epsilon \in$ First (A)

IV.) $S \to AB$, then First (S) = First (A) if A doesn't derive ϵ

First (A) = $\{F_i(A) - \{\epsilon\}\} \cup F_i(B)$

If $A \to \epsilon$, $B \not\to \epsilon$

~~iii)~~

IV) If $S \to ABC$

If $A \to \epsilon$, $B \to \epsilon$, $C \not\to \epsilon$

$$F_i(S) = (F_i(A) - \{\epsilon\}) \cup (F_i(B) - \{\epsilon\}) \cup F_i(C)$$

Follow Set Computation

Follow(X): It is a set of terminals where each terminal is derived as first symbol after X.

1. $S \to aSb \mid c$

If S is start symbol, then $\$ \in$ Follow(S)

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

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

2. $S \to Sb \mid Aa$

$A \to aSc \mid de$

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

Fo(A) = $\{a\}$

I) If S is start symbol then $\$ \in$ Follow(S)

II) If $X \to \alpha Y B$, then i) Follow(Y) = First(B) If B doesn't derive ϵ

ii) Follow(Y) = First(B) $- \{\epsilon\} \cup$ Follow(X)

III) If $X \to \alpha Y$, then

Follow(Y) = Follow(X)

Eg

1.) $S \to AB$

$A \to a$

$B \to b$

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

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

First(B) = $\{b\}$ Follow(B) = Follow(S) = $\{\$\}$

2.

$S \to AB$

$A \to \epsilon$

$B \to b$

First(S) = $(\cancel{\text{First}}(A) - \cancel{\{\epsilon\}}) \cup F_i(B) = \{b\}$

First(A) = $\{\epsilon\}$

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

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

$Follow(A) = First(B) = \{b\}$

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

3.

$S \rightarrow AB$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

$F_i(S) = \{\epsilon\}$ $\quad$ $F_o(S) = \{\$\}$

$F_i(A) = \{\epsilon\}$ $\quad$ $F_o(A) = F_o(S) = \{\$\}$

$F_i(B) = \{\epsilon\}$ $\quad$ $F_o(B) = F_o(S) = \{\$\}$

4.

$S \rightarrow Aa \mid Bb$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

$F_i(S) = \{a\} \cup \{b\} = \{a, b\}$

$F_i(A) = \{\epsilon\}$

$F_i(B) = \{\epsilon\}$

$F_o(S) = \{\$\}$

$F_o(A) = \{a\}$

$F_o(B) = \{b\}$

5.

$S \rightarrow Aa \mid Bb \mid AB$

$A \rightarrow \epsilon \mid c$

$B \rightarrow \epsilon$

$F_i(S) = \{c, a, b, \epsilon\}$

$F_i(A) = \{\epsilon, c\}$

$F_i(B) = \{\epsilon\}$

$F_o(S) = \{\$\}$

$F_o(A) = \{a\} \cup F_o(S)$

$= \{a, \$\}$

$F_o(B) = \{b\} \cup F_o(S)$

$= \{b, \$\}$

6.

$S \rightarrow \underline{S}\,\underline{S} \mid (\underline{S}) \mid a$

$F_i(S) = \{a, (\}$ $\quad$ $F_o(S) = \{\$,), (, a\}$

$S \rightarrow S\,S$ $\quad$ $F_i(S)$

$S \rightarrow S\,S$ $\quad$ $F_o(S)$

$S \rightarrow (S)$ $\quad$ $)$

7. $S \to aS \mid Sa \mid ab$

$Fi(S) = \{a\} \cup Fi(S)$

$= \{a\}$

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

8.

$S \to Aa \mid Bb \mid AB$

$A \to \epsilon \mid c$

$B \to \epsilon \mid d$

$Fi(S) = \{c, a, b, d, \epsilon\}$

$Fi(A) = \{\epsilon, c\}$

$Fi(B) = \{\epsilon, d\}$

$Fo(S) = \{\$\}$

$Fo(A) = \{a\} \cup \{d\} \cup Fo(S) = \{a, d, \$\}$

$Fo(B) = \{b\} \cup Fo(S) = \{b, \$\}$

9

$E \to TR$

$R \to +TR \mid \#$

$T \to FY$

$Y \to *FY \mid \#$

$F \to (E) \mid i$

$Fi(E) = Fi(T) = Fi(F) = \{(, i\}$

$Fi(R) = \{+, \#\}$

$Fi(T) = Fi(F) = \{(, i\}$

$Fi(Y) = \{*, \#\}$

$Fi(F) = \{(, i\}$

$Fo(E) = \{\$,)\}$

$Fo(R) = Fo(E) \cup Fo(R) = \{\$,)\}$

$Fo(T) = Fi(R) \cup Fi(R) = \{+, \#\}$

$Fo(Y) = Fo(T) \cup Fo(Y) = \{+, \#\}$

$Fo(F) = Fi(Y) = \{*, \#\}$

10.

$S \to Bc \mid DB$

$B \to ab \mid cS$

$D \to d \mid \epsilon$

$Fi(S) = Fi(B) \cup \{d\} \cup Fi(B) = \{a, c, d\}$

$Fi(B) = \{a, c\}$

$Fi(D) = \{d, \epsilon\}$

$Fo(S) = Fo(B) \cup \{\$\} = \{c, \$\}$

$Fo(B) = \{c\} \cup Fo(S) = \{c, \$\}$

$Fo(D) = \{a, c\}$

Top Down Parser

LL(0) LL(1) LL(2) -- LL(K)

L: Left to Right Scanning

L: Left Most Derivation

Top Down Parser

- Recursive descent
- Non-recursive decent

For each nonterminal we write procedure

Table Driven Approach

LL(1) Parser

→ Predictive parser

→ Non-recursive descent (By default) Table to impl.

→ Recursive descent. Procedure oriented.

→ Follows LMD

→ It uses LL(1) Grammer.

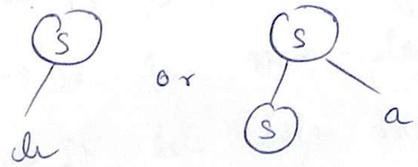
→ It stores LL(1) grammer inside LL(1) table.

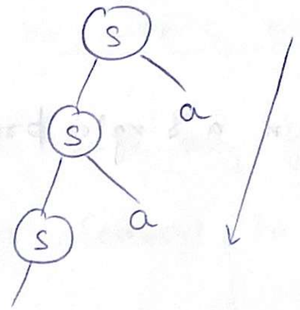
→ It uses stack, table to understand program with the help of LMD.

$S \rightarrow Sa \mid b$ (Left Recursion) LL(1)

↳ 1 look-ahead

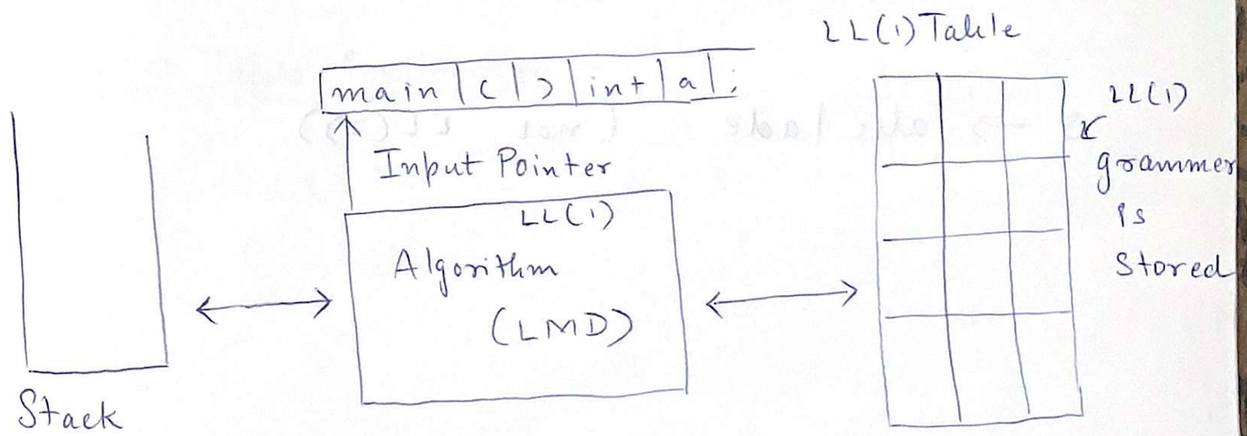
b	





There is a chance to enter infinite loop.

$S \rightarrow aS \mid b$



Top-down parser [LL(1)]

LL(0) Grammer:

$S \rightarrow a$

$S \rightarrow Aa$

$A \rightarrow bc$

$S \rightarrow a \mid b$

(not LL(0))

Every not terminal must contain a single production

SUCH CFG is LL(0)

LL(1) Grammer:

1 look-a-head

$S \rightarrow abc \mid def$ (LL(1))

$S \rightarrow abc \mid ade$ (not LL(1))

LL(2) Grammer:

2 look-a-head

$S \rightarrow abc \mid ade$ (LL(2))

$S \rightarrow abc \mid abe$ (not LL(2))

LL(1) Grammer

→ It is CFG

→ It is unambigous CFG

→ It is non-left recursive CFG

→ It is left factored.

How to write LL(1) Grammer?

Step 1: Take any grammer CFG.

No algorithm exist.

Step 2: Convert to unambigous CFG

↓ Elimination of left recc. algo exists

Step 3: Convert to non-left recursive CFG

↓ Left-factoring algo. exist

Step 4: Left-factored grammer

↓

LL(1) grammer

LL(1) Table Construction

1. $S \rightarrow (L) \mid a$

$L \rightarrow LS \mid S$

Step 1: Compute first set for every non-terminal.

$F_i(S) = \{c, a\}$

$F_i(L) = \{c, a\}$

Step 2: If $\varepsilon \in$ First (any non terminal x), then compute Follow (x).

Step 3. LL(1) Table

	(	)	a	$
S	S→(L)		S→a	
L	L→LS L→S		L→LS L→S	

S row means all S productions. that derives 'a' as first symbol.

↳ Given grammer is not LL(1)

Note

1. | main | c | > | { | } | $ |

↳ end terminal

2. If any entry (cell) of table has more than one production [multiple entries] then given CFG is not LL(1).

2. $S \to AB \mid c$

$A \to aA \mid \epsilon$

$B \to bB \mid \varepsilon$

$F_i(S) = \{a, b, \epsilon, c\}$ $\quad$ $F_o(S) = \{\$\}$

$F_i(A) = \{a, \varepsilon\}$ $\quad$ $F_o(A) = \{b, \$\}$

$F_i(B) = \{b, \epsilon\}$ $\quad$ $F_o(B) = \{\$\}$

	a	b	c	$
S	AB	AB	c	AB
A	aA	ε		ε
B		bB		ε

when we are looking at follow set make null productions as entry into table.

Every entry has atmost 1 production so given grammar is LL(1).

3. $S \to AB \mid a$

$A \to aA \mid \epsilon$

$B \to bB \mid \varepsilon$

$F_i(S) = \{a, b, \epsilon\}$ $\quad$ $F_o(S) = \{\$\}$

$F_i(A) = \{a, \epsilon\}$ $\quad$ $F_o(A) = \{\$, , b\}$

$F_i(B) = \{b, \varepsilon\}$ $\quad$ $F_o(B) = \{\$\}$

	a	b	$
S	S → AB S → a	S → AB	S → AB
A	A → aA	A → ε	A → ε
B		B → bB	B → ε

Given CFG is not LL(1).

Note

$S \rightarrow \overset{\text{NULL}}{AB} \mid \overset{\text{NULL}}{\varepsilon}$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$\underbrace{Fo(S) = \{\$\}}_{\text{NULL Productions}}$

	$
S	S → AB S → ε

Representation of LL(1) Parsing Table.

rule number

1. $S \rightarrow a$
2. $S \rightarrow (L)$
3. $L \rightarrow (S)$
4. $L \rightarrow \varepsilon$

I

	a	(	)	$
S	S → a	S → (L)		
L		L → (S)	L → ε	

II

	a	(	)	$
S	a	(L)		
L		(S)	ε	

III

	a	(	)	$
S	1	2		
L		3	4	

IV

Non terminal	Terminal	Rule
S	a	S → a
S	(	S → (L)
L	(	L → (S)
L	)	L → ε

V

Non terminal	Terminal	Rule
S	a	1
S	(	2
L	(	3
L	)	4

How to parse given input using LL(1) parser?

↓

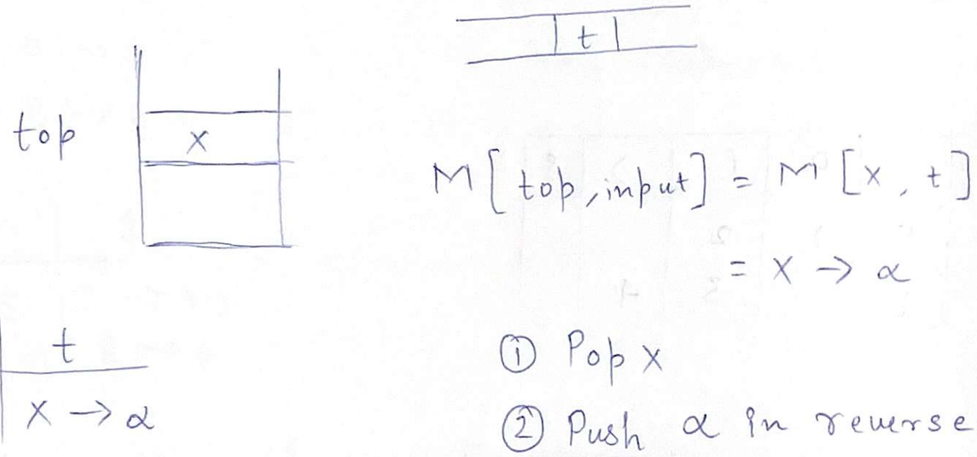
derive

↓

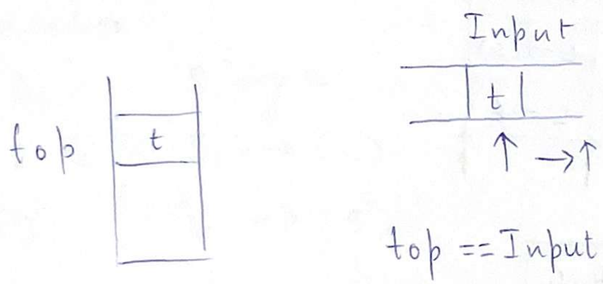
verify

Prediction & Match

Prediction:
(Substitution)



Match:



LL(1) Parsing Algorithm

(Accept)
1) If top == input == $ then accept

2) If top == input then
(Match) pop top of the stack; advance to next input symbol; goto 1;

3) If top is non terminal
(Prediction) if M[top, input] is a production then replace top with the production; goto 1;

4) Otherwise error.

eg Input: ((a))

M	a	(	)	$
S	S → a	S → (L)		
L		L → (S)	L → ε	

Stack	Input	Action
$ ~~S~~	((a)) $	Prediction: S → (L)
$) L ~~(~~	→ ((a)) $	Match
$) ~~L~~	(a)) $	Prediction: L → (S)
$)) S ~~(~~	(a)) $	Match
$)) ~~S~~	a)) $	Prediction: S → a
$)) a	a)) $	Match top == i/p
$))	)) $	Match top == i/p

$)	)$	Match TOS == i/p
		Accept
$	$	

S
↓
(L)
↓
((S))
↓
((a))

T- Conflict & N- conflits

T- Conflict :

input

top is terminal

top ≠ input

$S \longrightarrow ab$

a	c	$

↑

	a
S	ab

$c \neq b$

T-conflit

N - conflit :

$M[\underbrace{top}_{\text{non terminal}}, input] = \text{Blank}$
$= \text{error}$

How to identify LL(1) Grammar ?

1. Apply Theory (or)
2. Contruct Table (or)
3. Use shortcut .

Note :- If any entry of table is multiply defined then G is not LL(1).

- If G is ambigous
- If G is left recursive
- If G is not - left factored.
- And in other cases as well.

Shortcut :

Rule 1 :

$A \rightarrow \alpha_1 | \alpha_2 \Rightarrow F_i(\alpha_1) \cap F_i(\alpha_2) \neq \phi$

then given grammer is not LL(1)

$A \rightarrow a | b | bc$ It is not LL(1)

For LL(1), $\forall i, j$ $F_i(\alpha_i) \cap F_i(\alpha_j) = \phi$

Rule 2:

$A \to \alpha \mid \epsilon$

If $F_i(\alpha) \cap F_o(A) \neq \phi$, then not LL(1)

Eg

1. $S \to (L) \mid a$

$L \to LS \mid S$

$F_i((L)) \cap F_i(a) = \phi$

$F_i(LS) \cap F_i(S) \neq \phi$

$\{(, a\} \cap \{(, a\}$

Not LL(1)

2. $S \to AA$

$A \to aA \mid b$

LL(1)

$F_i(aA) \cap F_i(b) = \phi$

3. $S \to aS \mid Sa \mid \epsilon$

$F_i(aS) \cap F_i(Sa)$

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

Not LL(1)

4. $S \to aS \mid Sa \mid ab$

$F_i(aS) \cap F_i(Sa) \neq \phi$

Not LL(1)

5. $S \to SS \mid (S) \mid a$

$F_i(SS) \cap F_i((S)) = \{(, a\} \cap \{(\} \neq \phi$

Not LL(1)

6. $E \to E + E \mid E * E \mid id$

Not LL(1)

7. $S \to AB \mid a$

$A \to aA \mid \varepsilon$

$B \to aB \mid \varepsilon$

$F_i(AB) \cap F_i(a)$

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

Not LL(1)

8. $S \to iEtSeS \mid iEtS \mid a$

$E \to b$

not LL(1)

9. $S \to AB \mid ab$

$A \to a$

$B \to b$

Ambigous.
Not LL(1)

Non left factored CFG (not LL(1))

10. $S \rightarrow Sa \mid b$ — Left Recursion

Not LL(1)

11. $S \rightarrow a \mid ab$ — Common Prefix

Not LL(1)

Non left factored CFGs [not LL(1)]

12. $S \rightarrow aSa \mid \epsilon$ — Special Grammer

Not LL(1)

It is unambiguous
It is non-left recursive
It is left factored.

Note:

If no ϵ production

Left factored CFG $\Rightarrow$ LL(1)

Non left factored CFG $\Rightarrow$ not LL(1).

Top Down Parser v/s Bottom Up Parser

Follows LMD	Follows RMD in reverse.
Prediction	Shift and reduce. (Performs reduction)

LMD v/s Reverse of RMD

$S \rightarrow AB \mid c$
$A \rightarrow a$
$B \rightarrow b$

String = ab

LMD:

$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$

$S \rightarrow AB$ $\quad A \rightarrow a$ $\quad B \rightarrow b$

LMD order : $\xrightarrow{S, A, B}$

RMD:

$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$

$S \rightarrow AB$ $\quad B \rightarrow b$ $\quad A \rightarrow a$

RMD order : $\xrightarrow{S, B, A}$

Reverse of RMD

$S \Leftarrow AB \Leftarrow Ab \Leftarrow ab$

$S \rightarrow \underline{AB}$ $\quad B \rightarrow \underline{b}$ $\quad A \rightarrow \underline{a}$

Handle $\quad$ Handle $\quad$ Handle

RHS of a production

Reverse of RMD order : $\xrightarrow{A, B, S}$

TDP

S

↓ substitutions

String (sentence)

LMD

BUP

S

↑ reductions

String

RMD in reverse

Types of Bottom-Up Parsers

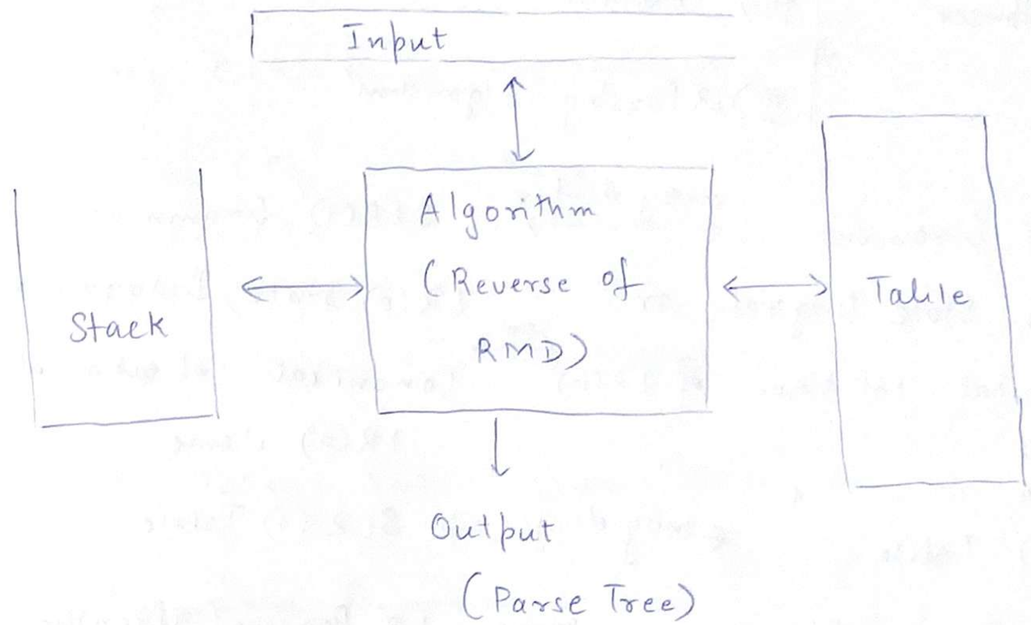
SR parsers [Shift Reduce]

- Left to right scan
- reverse of RMD
- LR parser
  - LR(0) parser → no look-a-head
  - SLR(1) parser → Simple LR
  - LALR(1) parser → lookahead LR(1)
  - CLR(1) parser [LR(1)] → Canonical LR(1)
  - LR(2), LR(3) --, LR(k)
- operator precedence parser

LL(1) parser ⇒ $O(n)$

BUP parser ⇒ $O(n^3)$

Configration of LR Parser



LR(0) Parser & SLR(1) Parser.

LR(0) parser ⇐ LR(0) Grammer

SLR(1) parser ⇐ SLR(1) Grammer

LALR(1) parser ⇐ LALR(1) Grammer

LR(1) parser ⇐ LR(1) Grammer

Grammer
↓
Table
↓
Parser

Build an LR(0) Parser:

I) Write an LR(0) Grammer

II) Contruct DFA / set of LR(0) items

III) Construt a Table (LR(0) Table)

IV) LR Parsing Algorithm.

LR(0) Grammer $\xleftrightarrow{\text{may differ}}$ SLR(1) Grammar

LR(0) State Diagram or Camonical collection of LR(0) items $\xleftrightarrow{\text{Same}}$ LR(0) State Diagram or Cononical collection of LR(0) items.

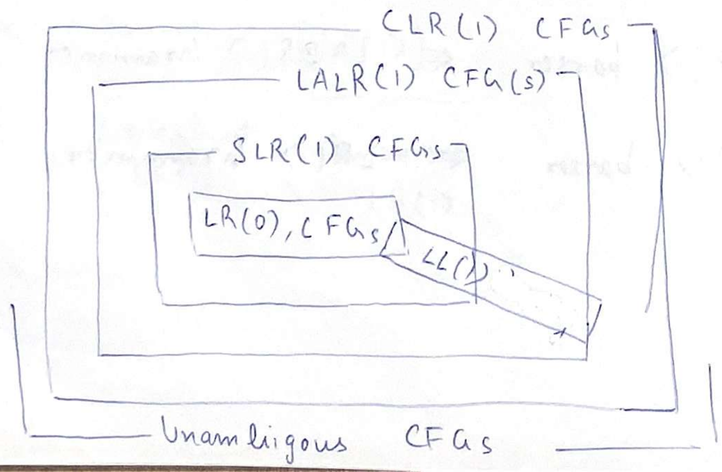
LR(0) Table $\xleftrightarrow{\text{may differ}}$ SLR(1) Table

LR Parsing Algorithm $\xleftrightarrow{\text{Same}}$ LR Parsing algorithm

→ Every LR(0) grammer is SLR(1) grammar but converse may not be true

→ LR(0) parser is less powerful that SLR(1) parser

→



Every LR(0) is unambiguous CFG.

Every SLR(1) is unambiguous CFG

Every LALR(1) is unambiguous CFG

Every CLR(1) is unambiguous CFG

LR(0) CFG → SLR(1)

SLR(1) CFG → LALR(1)

LALR(1) CFG → LR(1)

LR(1) CFG → Unambiguous.

LR(0) Parsing State Diagram / DFA

1. Augmented Grammar
2. Types of Item: Shift item, Reduced item, Acceptance item, and State item
3. Closure Set Computation (closure function)
4. Goto Set Computation (Goto function)
5. Identify SR and RR conflicts

Note:

- If every state has no conflict then grammer is LR(0)
- If ~~every~~ any state has some conflict, then grammar is not LR(0)

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

(Given CFG)

$\longrightarrow$

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

(Augmented CFG)

Add $S' \to S$: It helps to accept input.

$S' \to S.$ [Acceptance Item]

LR(0) items :

$X \to YaZ$ → (production)

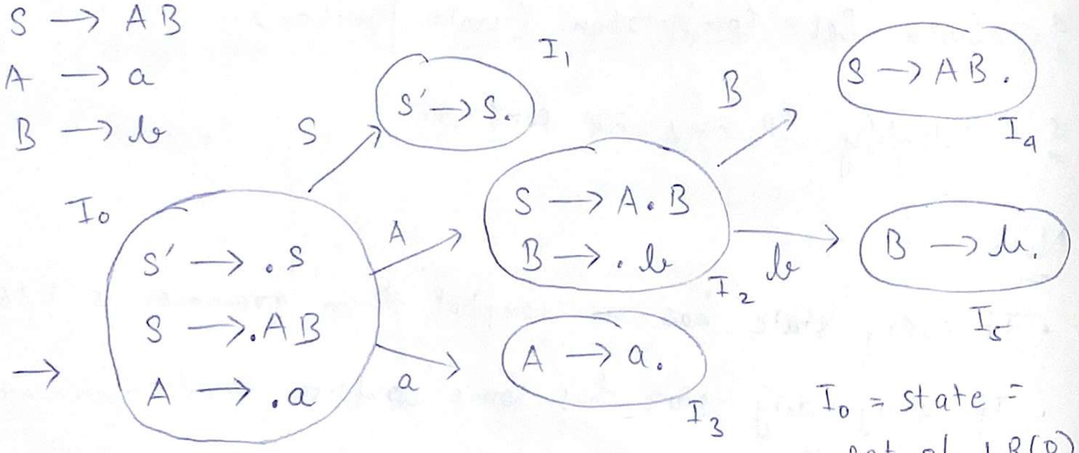
$X \to .YaZ$ [state Item] (nonterminal)
$X \to Y.aZ$ [Shift Item]
$X \to Ya.Z$ [State Item]
$X \to YaZ.$ [Reduced Item / Completed item]

Incompleted Items

LR(0) DFA : {Canonical collection of set of LR(0) set of items}.

← state →

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



I_0 = state = set of LR(0) items.

I_0 = Closure ($S' \to .S$)
= {$S' \to .S$, $S \to .AB$, $A \to .a$}

I_1 = Goto (I_0, S)
I_2 = Goto (I_0, A)
I_3 = Goto (I_0, a)
I_4 = Goto (I_2, B)
I_5 = Goto (I_5, b)

How to identify LR(0) Grammer?

1. Shift Reduce Conflict (SR Conflit)

If any state has both shift item and reduced item

2. Reduce-reduce conflit

If same state has atleast 2 reduced items.

→ Single item never produce any conflict.

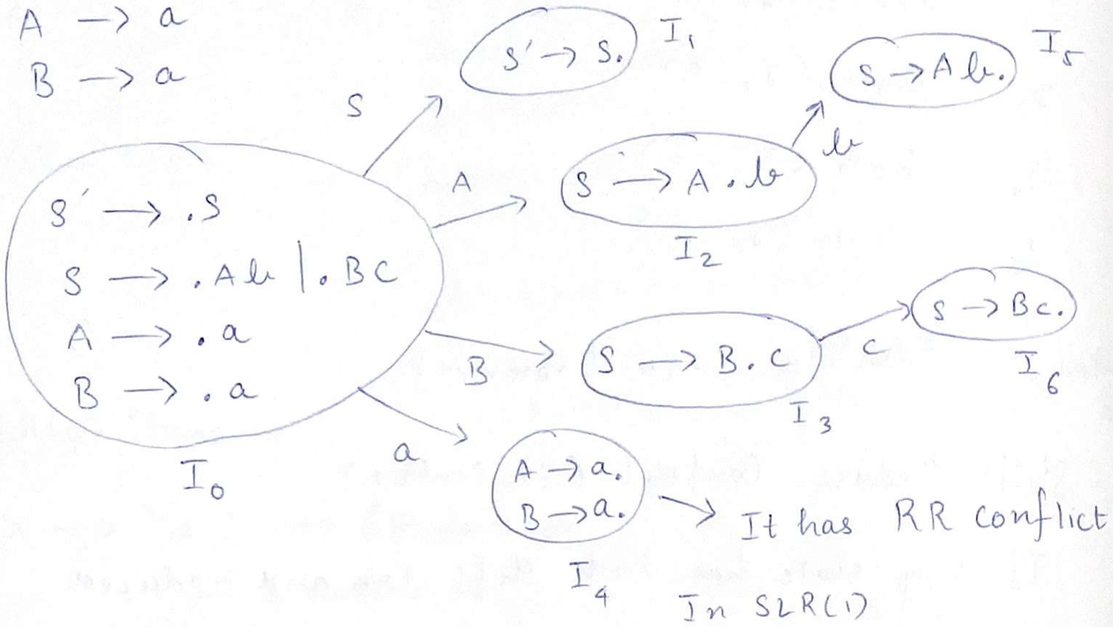
$X \to \alpha.B$ no conflict.

⇐ This grammar is LR(0).

$S \to Ab \mid Bc$

$A \to a$

$B \to a$



It has RR conflict

In SLR(1)

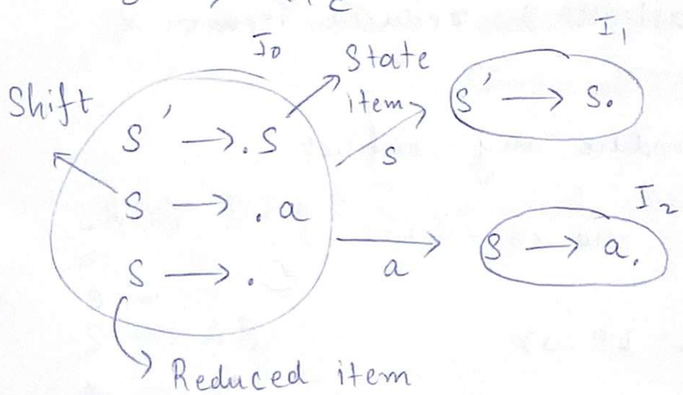
It is not LR(0) Grammar.

$Fo(A) \cap Fo(B)$

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

No RR conflict in SLR(1).

$S \to a \mid \varepsilon$



Both shift and reduced item are present.

$\Rightarrow$ SR - conflit in LR(0)

Not LR(0) grammar.

$\left.\begin{array}{l} .\varepsilon \\ \varepsilon. \\ . \end{array}\right\}$ Same

How to identify SLR(1) Grammar?

↳ Follow Set

1. Shift - Reduce Conflit

$X \to \alpha . t\beta$ (Shift item)

$Y \to \gamma.$ (Reduced item)

If $t \in Follow(Y)$ then SR Conflit

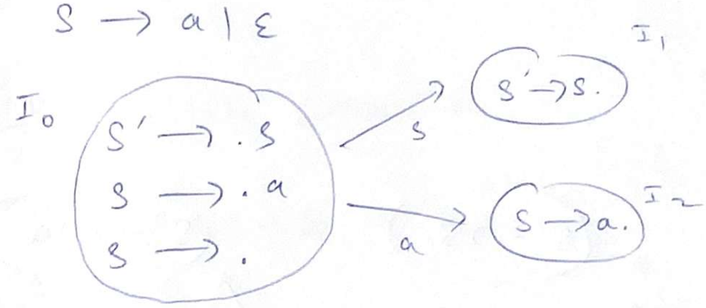
2. Reduce - reduce conflict.

$X \to \alpha.$

$Y \to \beta.$

If $Fo(X) \cap Fo(Y) \neq \phi$ then RR conflit.

$S \to a \mid \varepsilon$



Does $a \in Fo(S)$

$F_o(S) = \{\$\}$

$a \notin Fo(S)$

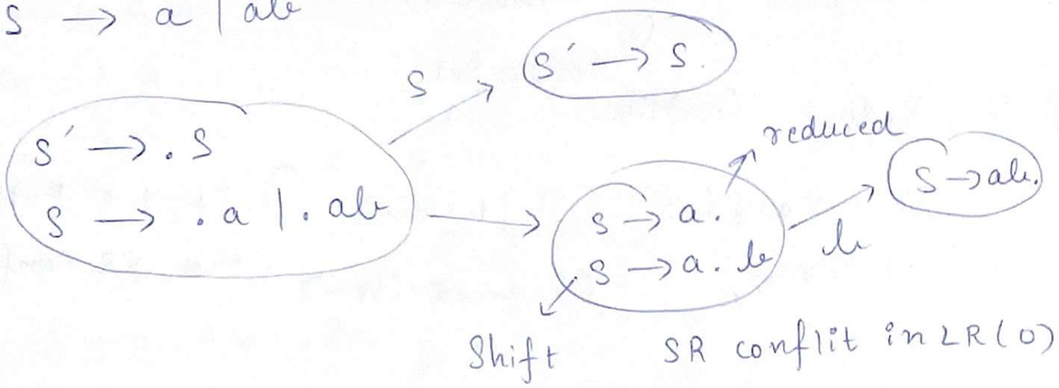
No conflict in SLR(1).

It is SLR(1) Grammar

Relation between LR(0) Grammar & SLR(1) Grammar.

Same set of LR(0) items used for both. SR & RR conflicts checking is different.

$S \to a \mid ab$



SR conflit in LR(0)

$Fo(S) = \{\$\}$

$b \notin Fo(S)$

No SR conflit SLR(1)

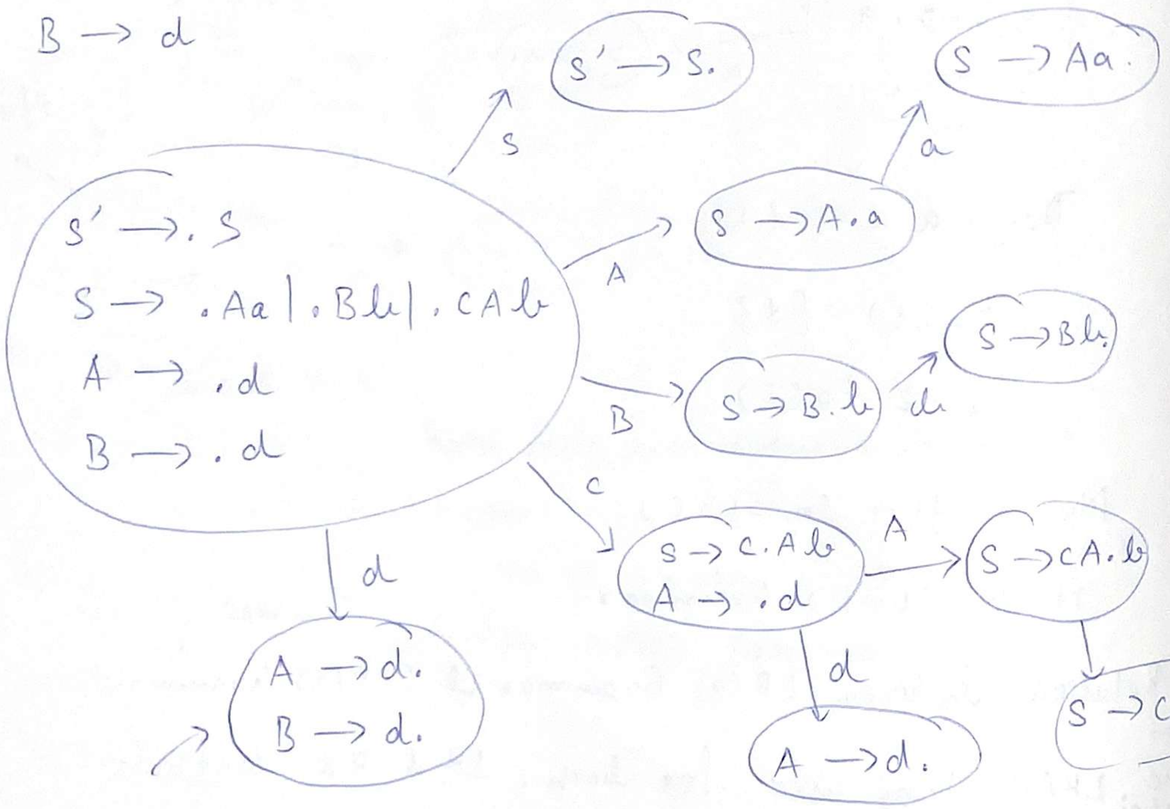
SLR(1) but not LR(0)

Why SLR(1) is not powerful than LALR(1)?

$S \to Aa \mid Bb \mid cAb$

$A \to d$

$B \to d$



RR conflict in LR(0)

$Fo(A) \cap Fo(B)$

$= \{a, b\} \cap \{b\} = \{b\} \neq \emptyset$

RR conflict in SLR(1)

LR(0) item v/s LR(1) item

LR(0) item:

$X \to \alpha . \beta$

(production with a dot)

[LR(0) & SLR(1)]

LR(1) item:

$X \to \alpha . \beta , v$

(production with a dot) (look-a-head set)

[LALR(1) & LR(1) parser]

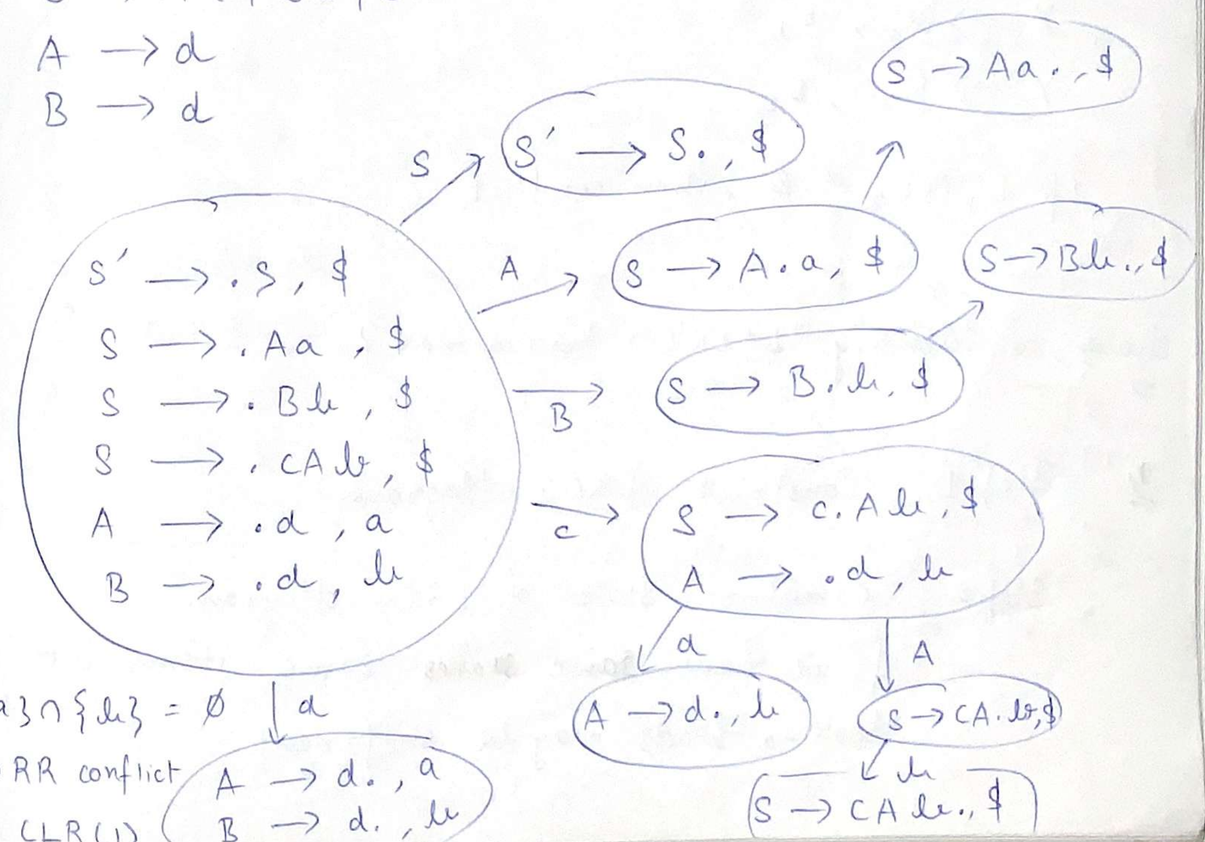
LR(1) State Diagram

It is CLR(1)

$S \to Aa \mid Bb \mid cAb$

$A \to d$

$B \to d$



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

No RR conflict in CLR(1)

$X \rightarrow \alpha . Y \underbrace{\beta , \gamma}$

$Y \rightarrow .$, $\underbrace{\text{First}(\beta\ \gamma)}_{\text{Look aheads of } Y \text{ productions.}}$

How to identify LR(1) Grammar?

1. Shift - Reduce Conflict.

$X \rightarrow \alpha . t B , L_1$

$Y \rightarrow \gamma . , L_2$

If $t \in L_2$, then conflict

2. Reduce - Reduce Conflict.

$X \rightarrow \alpha . , L_1$

$Y \rightarrow B . , L_2$

If $L_1 \cap L_2 \neq \emptyset$, then conflict.

How to identify LALR(1) Grammar?

Step 1: Construct LR(1) diagram

Step 2: Combine state of LR(1) diagram
If we have states with same items but look-a-heads may be different.

Step 3: Check SR & RR conflicts

$\underbrace{X \rightarrow \alpha . B}_{\text{same}} , L_1$ $\qquad$ $\underbrace{X \rightarrow \alpha . B}_{\text{same}} , L_2$

In LALR(1),

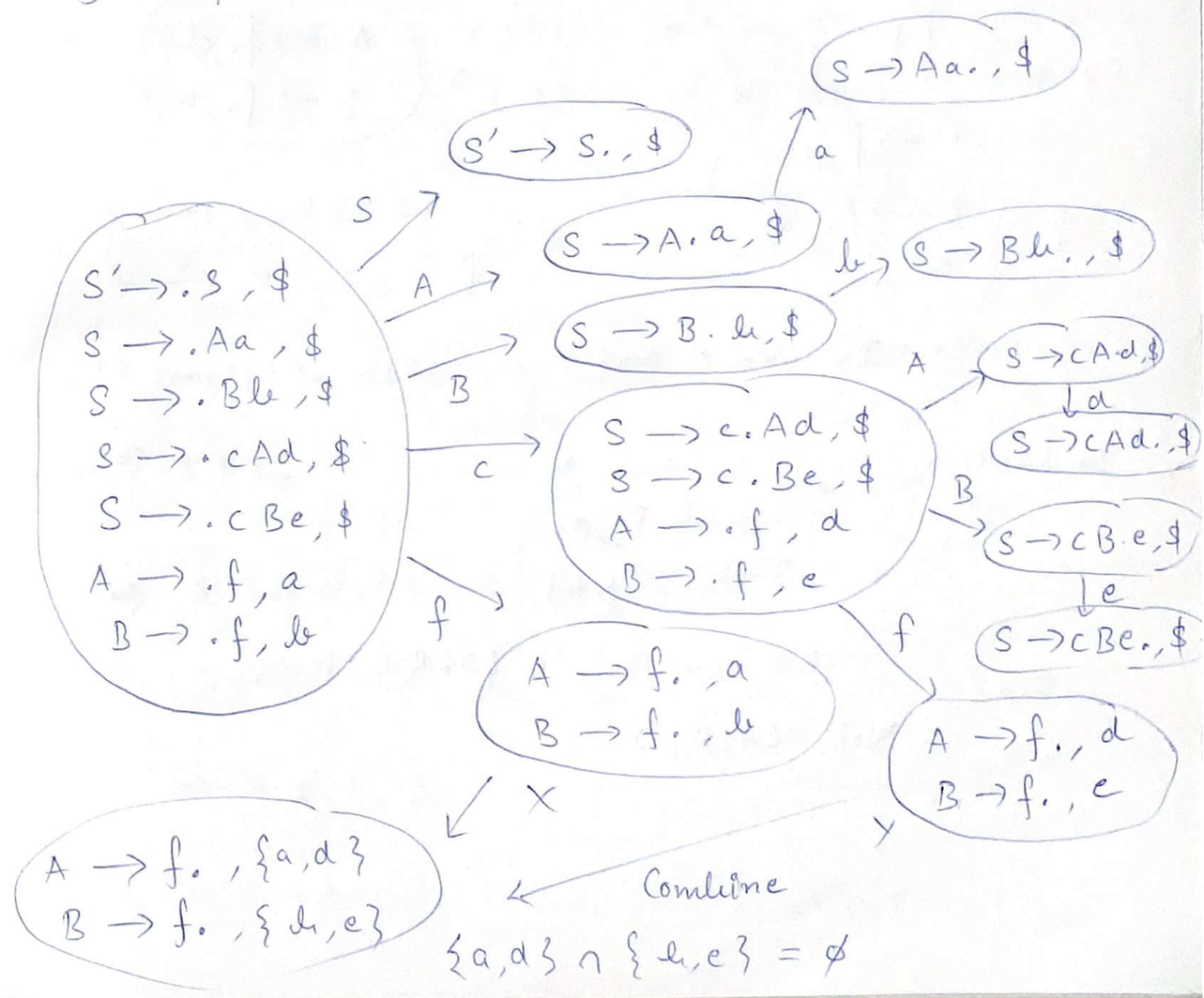
$X \rightarrow \alpha . B , L_1 \cup L_2$

$S \rightarrow Aa \mid Bb \mid cAd \mid cBe$

$A \rightarrow f$

$B \rightarrow f$

It is LALR(1) and SLR(1) also.

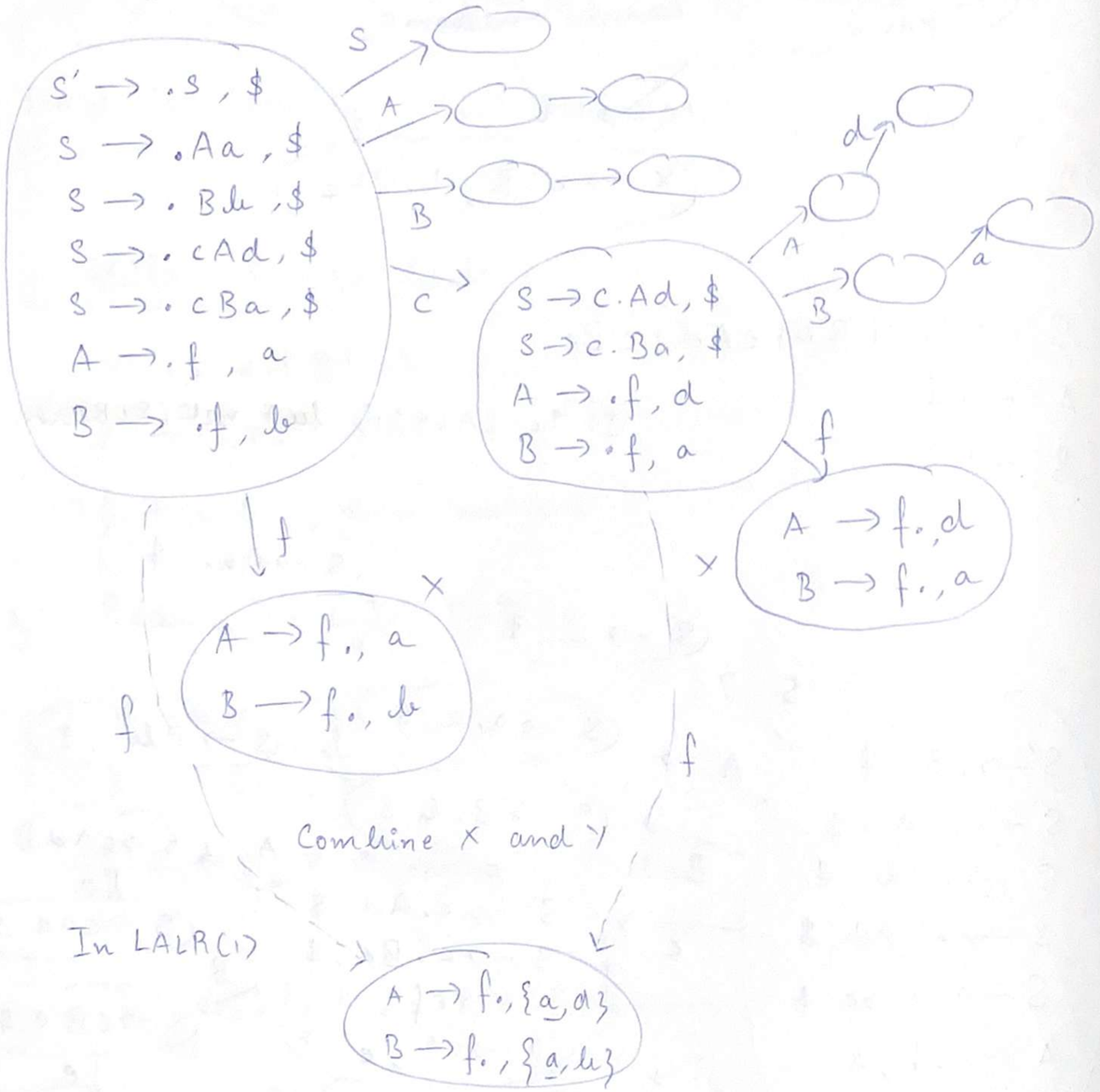


$S \to Aa \mid Bb \mid cAd \mid cBa$

It is CLR(1)

$A \to f$

$B \to f$



RR conflict in LALR(1)

Not LALR(1)

CLR(1) Grammar V/s LALR(1) Grammar

Q If the grammar is already CLR(1), then what kind of conflicts is possible in LALR(1)?

Given: Grammar is CLR(1). So no SR and no RR conflict in CLR(1).

If we combine states of CLR(1) to obtain LALR(1), then what kind of conflict possible?

Is SR conflict possible in LALR(1) after combining states of CLR(1)

No SR conflict in CLR(1). Will we get SR conflict in LALR(1) after combining states?

$X \to \alpha . t B, L_1$ & $X \to \alpha . t B, L_3$

$Y \to \gamma ., L_2$ $\quad$ $Y \to \gamma ., L_4$

(no SR conflict in CLR(1)) $\Rightarrow t \notin L_2$

(no SR conflict in CLR(1)) $\Rightarrow t \notin L_4$

$\Rightarrow X \to \alpha . t B, L_1 \cup L_3$

$Y \to \gamma ., L_2 \cup L_4$

$\Rightarrow t \notin L_2 \cup L_4$

$\therefore$ It is impossible to have SR conflict in LALR(1).

Is RR conflict possible in LALR(1) after combining states of CLR(1)?

No RR conflict in CLR(1)

$X \to \alpha.,\ L_1 = a$ $\qquad$ $X \to \alpha.,\ L_3 = b$

$Y \to \beta.,\ L_2 = b$ $\qquad$ $Y \to \beta.,\ L_4 = c$

$L_1 \cap L_2 = \emptyset$ $\qquad$ $L_3 \cap L_4 = \emptyset$

$X \to \alpha.,\ L_1 \cup L_3 = \{a, b\}$ $\qquad$ $\{a, b\} \cap \{b, c\} \neq \emptyset$

$Y \to \beta.,\ L_2 \cup L_4 = \{b, c\}$ $\qquad$ (Possible)

RR conflict is possible

Note

(more powerful) CLR(1) Takes more space

↓

(popular) LALR(1) Takes less space

Entries of LR Tables

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

I) Rules : How to give an entry?

II) Entries :

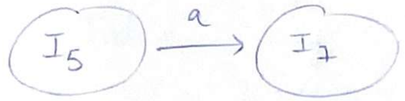
1) Shift Entries (Shift item)
2) State Entries (State item)
3) Accept Entry ($S' \to S.$)
4) Reduced Entries ($X \to \alpha.$)
5) Blank Entries (error)

Shift Entry for LR Table

Shift Entry Rule is same for all LR Tables

1. Every terminal transition in the diagram will give one shift entry in table.

2. No. of shift entries in the table = No of terminal transitions in DFA.



	a (col)
5 (row)	S_7 (entry)

This rule is applicable for LR(0), SLR(1), LALR(1), CLR(1).

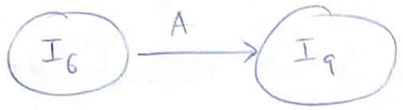
3. # Shift entries of LR(0) = # Shift entries in SLR(1) = # Shift entries in LALR(1) ≤ # Shift entries of CLR(1)

State Entry (goto entry) for LR entry

1. Every non-terminal transition in the diagram will give one state entry.

2. Number of state entries the table = Number of non-terminal transitions

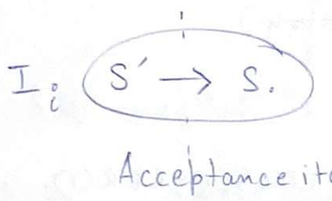
3. # State entries for LR(0), SLR(1), & LALR(1) is same



	A (col)
6 (row)	9 (entry)

Acceptance Entry for LR Table

Find the state X has $(S' \to S)$ item. Make $M[n, \$]$ entry as accept.

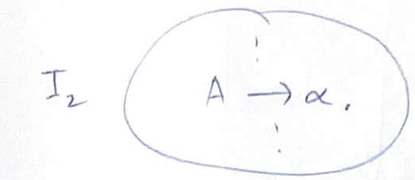


	$
i	Accept

Reduced Entry for LR(0)

LR(0)	SLR(1)	LALR(1)	CLR(1)
We don't look at followers	We look at followers of whole grammar	We look at followers based on combine path	We look at followers based on exact path.
$A \to \alpha.$ Follow (A) based on grammar		$A \to \alpha., t_1 \mid t_2$ Based on combined path	$A \to \alpha., t_1$ follow on exact path

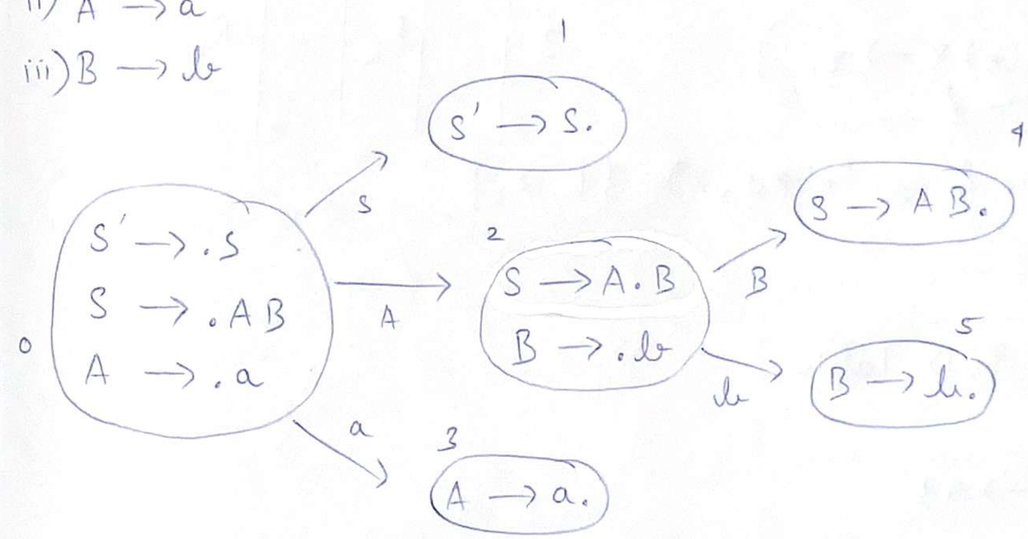
Every reduced item in diagram will have reduced entries in the table. If state has more than one reduced item, consider each reduced item.



	Action part	Goto Part
2	$A \to \alpha$ R_{iii} | $A \to \alpha$ R_{iii} | $A \to \alpha$ R_{iii}	

LR(0) Table

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



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

← Action → ← Goto →

Reduced Entry for SLR (1)

I_3 (k) $X \to \alpha.$

	t_1	t_3
3	R_k	R_k

Compute follow Follow (X) = $\{t_1, t_3\}$

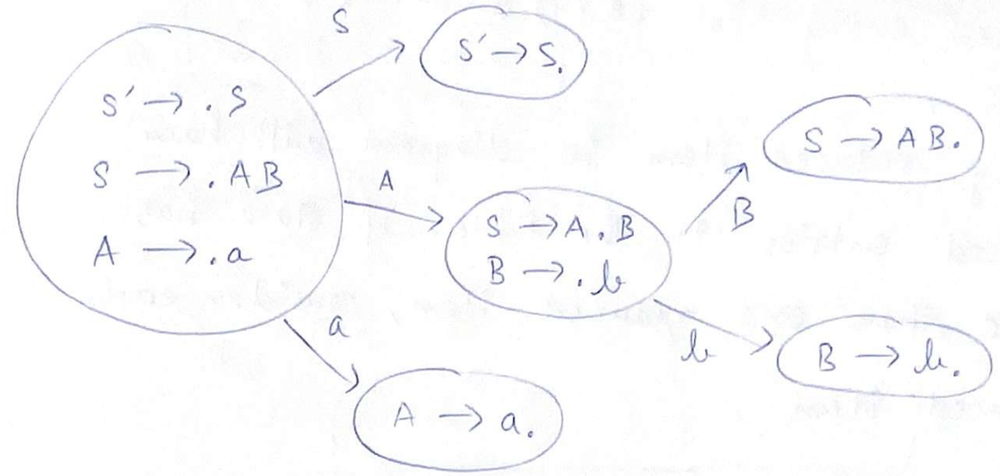
(X)

SLR(1) Table

i) $S \to AB$

ii) $A \to a$

iii) $B \to b$



Fo(S) = {$}

Fo(A) = {b}

Fo(B) = {$}

	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}			

Note

No. of reduced enteries of LR(0) table ≥

No. of reduced enteries of SLR(1)

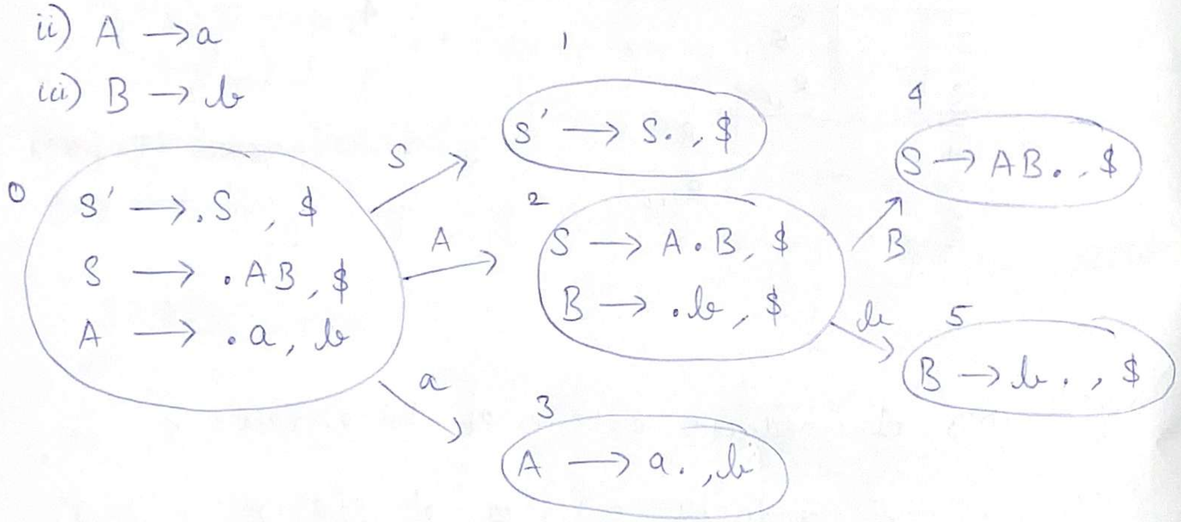
Reduced entry for LR(1) & LALR(1)

Every reduced item in diagram will have reduced entries in the table. If state has more than one reduced item, consider each reduced item.

i (K) $X \rightarrow \alpha.,\ \{t_1, t_3\}$

	t_1	t_3
i	R_K	R_K

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



This is both CLR(1) & LALR(1).

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

1. Shift Entries
2. State Entries
3. Accept Entries

→ Rule is same for all. No. of entries of LR(0), SLR(1) & LALR(1) will be same.

4. Reduced Entries → Rule is different for all; Entries may differ
5. Blank entries

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

≥ No. of reduced entries in LR(1)

Comparisons of LR Parsers / Tables / Grammars

LR(0) Parser < SLR(1) Parser < LALR(1) Parser

< CLR(1) Parser (Expressive Power)

Set of LR(0) CFGs ⊂ Set of SLR(1) CFGs ⊂

Set of LALR(1) CFGs ⊂ Set of CLR(1) CFGs

For a particular grammar

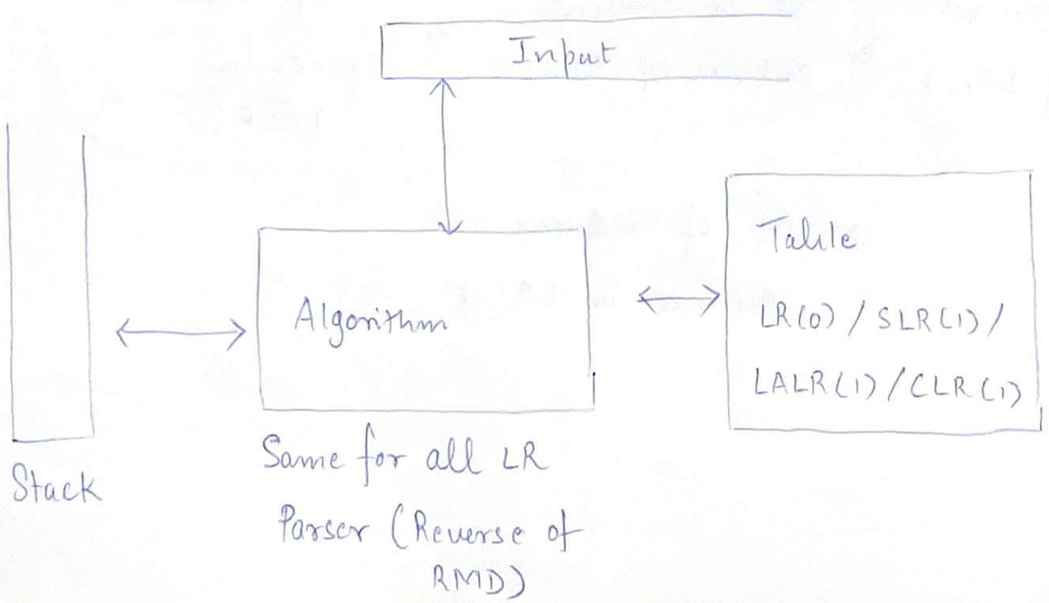
Size of LR(0) Table = Size of SLR(1) Table =

Size of LALR(1) Table ≤ Size of CLR(1) table

No. of states (LR(0)) = No. of states (SLR(1)

= No. of states (LALR(1)) ≤ No. of states (CLR(1))

LR Parsing



How to verify syntax of given program / input string ?

Shift Action : $M[top, input] = S_n$

1. Push "Input" & Inc. pointer
2. Push n

Reduced Action : $M[top, input] = R_k : X \rightarrow \alpha$

1. Pop $2 * |RHS| = 2 * |\alpha|$
2. Push LHS(X)
3. Push M[Below X, X]

Operator Grammar

What is Operator Grammar?

→ Content Free Grammar

→ Every production in operator Grammar

1. No null production
2. No two consecutive non-terminals.

Q Find Operator Grammar

1. $E \rightarrow \underline{AB}$
   $A \rightarrow a$ not O.G
   $B \rightarrow b$

2. $E \rightarrow E \& E$
   $E \rightarrow \sim E$ It is OG.
   $E \rightarrow (E) \mid id$

3. $E \rightarrow E + E$
   $E * E$
   $E / E \mid id$

4. $E \rightarrow \underline{EAE} \mid (E) \mid - E \mid id$ Not O.G
   $A \rightarrow + \mid - \mid * \mid / \mid \wedge$

5. $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E \mid (E) \mid -E \mid$ id It is O.G

Operator Precedence

Relation	Meaning
$a <. b$	a yields precedence to b
$a \doteq b$	a has the same precedence as b
$a .> b$	a takes precedence over b.

Associativity of Operator

If two operators have equal precedence then

1. Left Associative: Left to Right associative
2. Right Associative: Right to Left associative.

Higher Precedence:

$S_1 \dot{>} S_2$; iff there exists a production $U \rightarrow \alpha X S_2 \beta$

such that $X \overset{+}{\Rightarrow} \ldots S_1$ or $X \overset{+}{\Rightarrow} \ldots S_1 Y$, where

$S_1 ; S_2 \in V_T$, and $X, Y \in V_N$

Lesser Precedence:

$S_1 \lessdot S_2$, iff there exists a production $U \rightarrow \alpha S_1 X \beta$

such that $X \overset{+}{\Rightarrow} S_2 \ldots$ or $X \overset{+}{\Rightarrow} Y S_2 \ldots$, where

$S_1 ; S_2 \in V_T$ and $X, Y \in V_N$

Equal Precedence:

In same production

$S_1 \doteq S_2$ iff there exists a production $U \rightarrow \alpha S_1 S_2 \beta$

or $U \rightarrow \alpha S_1 X S_2 \beta$, where $S_1, S_2 \in V_T$ and

$X \in V_N$.

Q Find Precedence of Operators

1: $E \rightarrow E + T \mid T$

$T \rightarrow T \times F \mid F$

$F \rightarrow (E) \mid a$

① $+ \dot{>} +$

Left Right

left to right associative

$E + T$ left → + is not possible

$E + T$ right

$E + T$ left

② $* \dot{>} *$

left to right associate.

$T * F$ left → not possible immediately or just after one symbol

$T * F$ right

$T * F$

3. $+ \lessdot *$

Left

4. $* \dot{>} +$

Left

5. $(\lessdot ($

6. $(\doteq)$

7. $) \quad ($ No relation

8. $) \dot{>})$

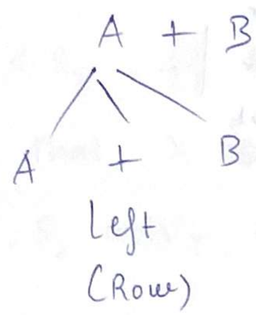
2. Find precedence relations using following CFG

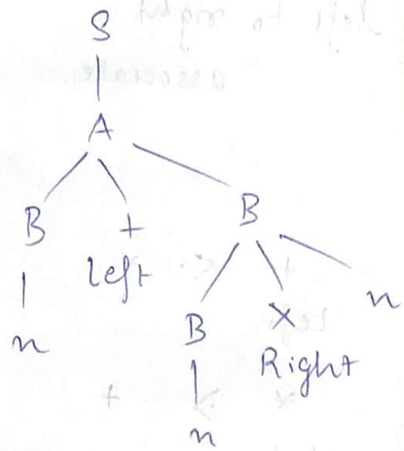
$S \rightarrow A \mid B$

$A \rightarrow A + B \mid B + B$

$B \rightarrow B \times n \mid n$

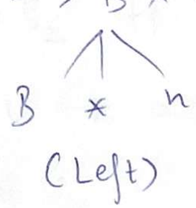
(Column) Right





(Right)

$B \rightarrow B * n$



Right (input)

left (top of stack)	+	×
+	>	<
×		>

Note: Left Recursion ⇒ Left Associative.

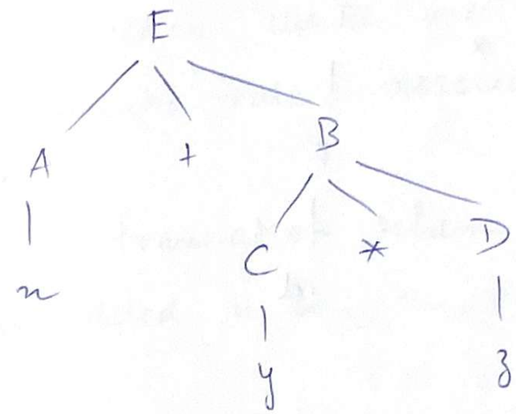
$E \rightarrow E + id \mid id$

+ is left associative

$E \rightarrow id * E \mid id$

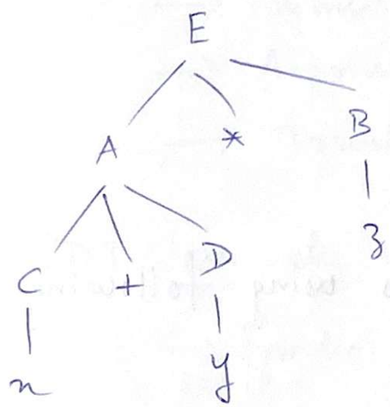
* is right associative

3. Find precedence relations using following parse tree.



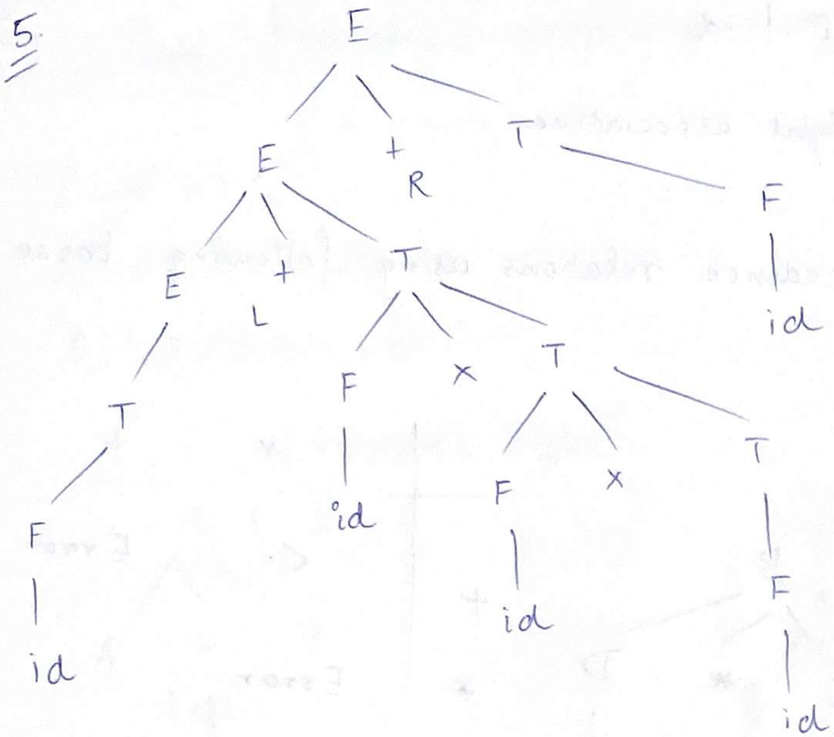
	*	+
+	<	Error
*	Error	

4.



	*
+	>

5.



		Right	
		+	*
Left	+	·>	<·
	*	·>	<·

6. Find the percentage relations using following table.

	id	(	)	$
id	e3	e3	>	>
(	<·	<·	=	e4
)	e3	e3	>	>
$	<·	<·	e2	e1

(<· (⇒ Right Associative

Left Right

Syntan Directed Translation

→ In attribute grammar, attributes are associated with grammar symbols of each production which are evaluated by the semantic rules associated with production rules.

→ In translation scheme, the semantic rules are embedded within the right side of the productions.

Semantic Rule

↳ Symbols of grammar

→ Attributes

→ Translation.

SDT can be used for:

→ Symbol table updates (adding types)

→ Performing type checking

→ Produced error message

→ Generating intermediate code

→ Generating parse tree

→ Any purpose.

Attribute may contain any information.

→ Type

→ String

→ Number

→ Address

→ Any data

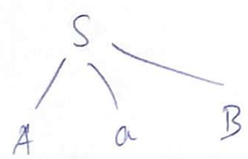
Attribute

1. Inherited Attribute: The computation at any node depends on parent or sibling.

Production

$S \rightarrow AaB$

Production tree



S is parent A, a, B

A is child of S

is sibling to A & B

a is right sibling to A

is left sibling to B

is child of S.

depends on parent Computation

$S \rightarrow \overbrace{A}^{\text{depends on parent}} a B \quad \{ \overbrace{A.n = S.n + 2}^{\text{Computation}} \}$

Computation is at A. Depends on S.

Synthesized Attribute: Computation depends on children

Eg1

Production	Semantic Rules
1. $L \rightarrow E_n$	$L.val = E.val$
2 $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3 $E \rightarrow T$	$E.val = T.val$
4. $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5. $T \rightarrow F$	$T.val = F.val$
6 $F \rightarrow (E)$	$F.val = E.val$
7 $F \rightarrow digit$	$F.val = digit.lexval$

Val is synthesized attribute.

Eg2

Production	Semantic Rules
$D \rightarrow T\ id\ ;\ D$	addType (id, entry, T.t)
$T \rightarrow BC$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow int$	$B.t = integer$
$B \rightarrow float$	$B.t = float$
$C \rightarrow [num]\ C_1$	$C.t = array(num.val, C_1.b\ C_1.t)$
$C \rightarrow \varepsilon$	$C.t = C.b$

t → Synthesized attribute

le → Inherited attribute

Eg 3.

Production Rule	Semantic action
$D \rightarrow TL$	$T_1.type = D_2.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L_1, id$	$L_1.type = L.type$
$L \rightarrow id$	addType (id.entry, L.type)
	addType (id.entry, L.type)

type is neither inherited nor synthesized.

Attribute Grammar

An attribute grammar is a syntax-directed defination where the semantic rules cannot have side effects.

1. L - Attributed Grammar
2. S - Attributed Grammar

L - Attributed Grammar

1. Computation depends on parent / left sibling / children
2. Semantic rule can be placed anywhere in RHS of production

S - Attributed Grammar

1. Attribute computed based on children.
2. Semantic rules must be at end of children.

Eg 1:

Production	Semantic Rules
$L \rightarrow En$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

val is synthesized

SDT is S - attributed CFG

Every S - attributed is L - attributed grammar.

Eg 2:

Production	Semantic Rules
$D \rightarrow T\ id;\ D$	add Type (id-entry, T.t)
$T \rightarrow BC$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow int$	$B.t = integer$
$B \rightarrow float$	$B.t = float$
$C \rightarrow [num]C_1$	$C.t = array(num.val, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \varepsilon$	$C.t = C.b$

t → synthesited attribute

b → Inherilut attribute

L - attributed but not S - attributed.

Eg 3

Production	Semantic action
$D \rightarrow TL$	$T.type = D.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L_1, id$	$L_1.type = L.type$ add Type (id.entry, L.type)
$L \rightarrow id$	add Type (id.entry, L.type)

L - attributed grammar.

SDT - Evaluation Techniques

→ SDT with attributes: L - attributed (Depth first traversal (from left to right)) & S - attributed (Bottom up parsing)

→ SDT without attributes: Any order can be followed.

We can also depend on L - attribute if SDT is having attributes in every rule.

Evaluation of L - attributed grammar (Dependency Graph)

Depth-first-traversal

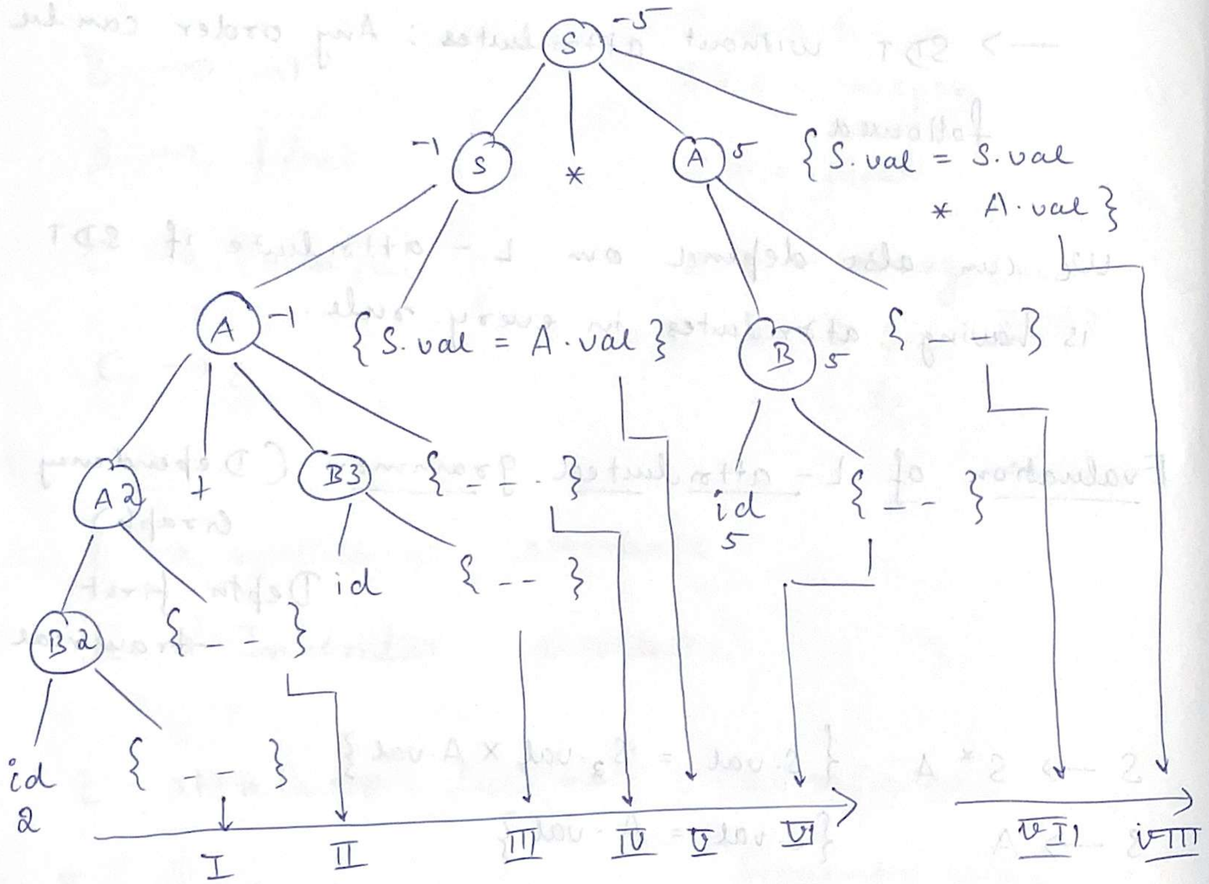
$S \rightarrow S * A$	$\{S_1.val = S_2.val \times A.val\}$
$S \rightarrow A$	$\{S.val = A.val\}$
$A \rightarrow A + B$	$\{A.val = A_1.val - B.val\}$
$B \rightarrow (S)$	$\{B.val = 2\}$
$A \rightarrow B$	$\{A.val = B.val\}$
$B \rightarrow id$	$\{B.val = id.val\}$

Input id + id * id 2 + 3 * 5

Find the attribute value computed at the root.

Annotated parse Tree / Decorated Parse Tree /

Semantic Tree → parse tree + semantic rules.



Traverse : Depth - first

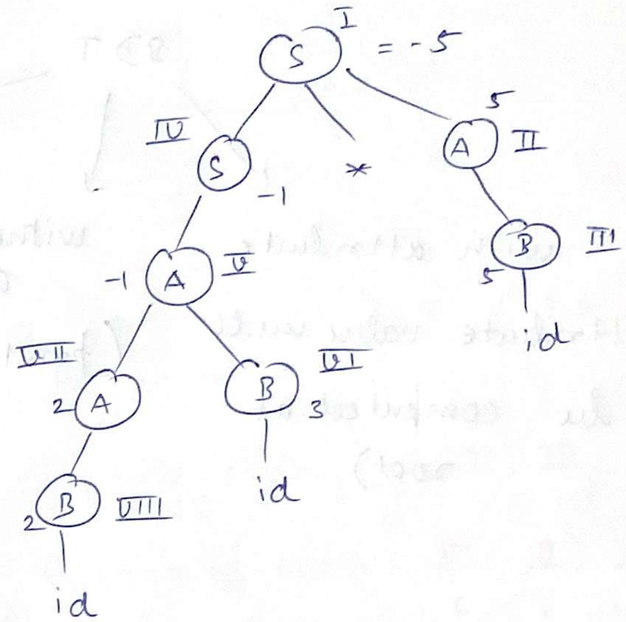
Evaluation of translations from left to right:

Evaluation of S- Attributed Grammar.

RMD in reverse (non - terminals order)

RMD

S
$\Downarrow$
$S * A$
$\Downarrow$
$S * B$
$\Downarrow$
$S * id$
$\Downarrow$
$A * id$
$\Downarrow$
$A + B * id$
$\Downarrow$
$A + id * id$
$\Downarrow$
$B + id * id$
$\Downarrow$
$id + id * id$



RMD Numbering

I, II, III, IV, V, VI, VII, VIII

Reverse of RMD

VIII, VII, VI, V, VI, VII II, I

Note: If attributed grammar is given and it is both L-attributed and S-attributed, we can always use depth first traversal to find attribute value.

SDT

- with attribute (attribute value will be computed at root)
- without attribute (print something)
- with attribute & print something (compute attribute value at root & produce some output)

Translation Scheme without Attributes

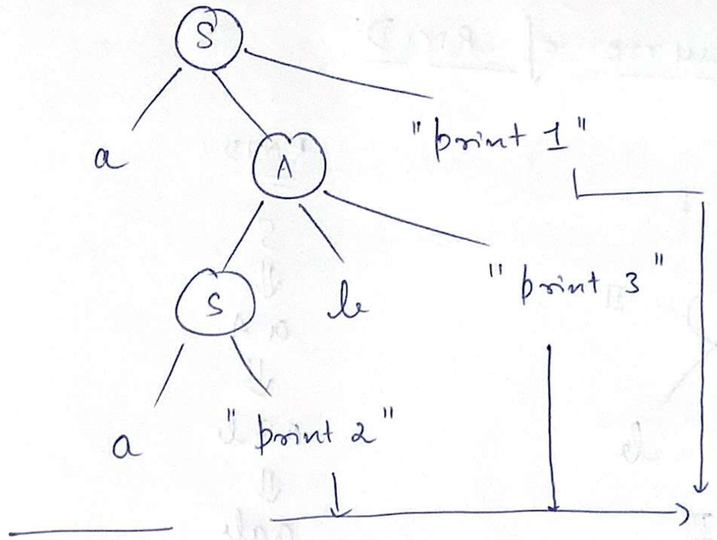
$S \rightarrow aA$ {print 1}

$S \rightarrow a$ {print 2}

$A \rightarrow Sb$ {print 3}

Depth First Traversal (from left to right)

Input = aab



O/p = 231

LMD & Reverse of LMD

LMD:

$S \rightarrow I$

$\Downarrow$

$aA \rightarrow II$

$\Downarrow$

$asb \rightarrow III$

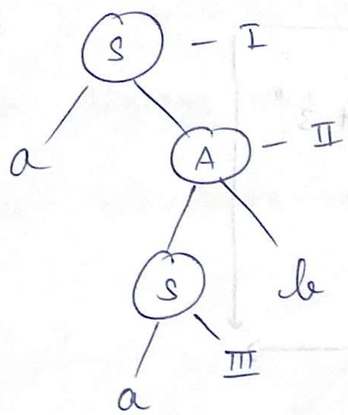
$\Downarrow$

$aab \rightarrow IV$

LMD order: I (1), II (3), III (2)

Reverse of LMD: III (2), II (3), I (1)

RMD & reverse of RMD



RMD:

$S \Downarrow aA \Downarrow aSb \Downarrow aab$

RMD order: I, II, III
1 3 2

Reverse of RMD: 2 3 1

Note: If Semantic rule position changed, then o/p of SDT also may change in L-attributed concept.

O/P will not effect for LMD, RMD, Reverse of LMD, Reverse of RMD.

Intermediate Code Representation

→ Linear Representation (Postfix, 3AC, SSA)

→ Non-Linear Representation. (Syntax Tree & DAG)

Postfix Code

$x = a * b + a * b$

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

$x\ ab*\ ab*\ +\ =$

Three Address Code: Every statement has atmost 3 add.

$x = a*b + a*b$	$t_1 = a*b$	$a = a*b$
$t_1 = a*b$	$x = t_1 + t_1$	$a = a+a$
$t_2 = a*b$		
$t_3 = t_1 + t_3$	4 variables	2 variables
$x = t_3$		

6 variables.

High Level Code $\xrightarrow{\text{Easy}}$ 3AC $\xrightarrow{\text{Easy}}$ Assembly Code

Machine Independent — Machine Dependent

→ Triple Notation

→ Quadruple Notation

→ Indirect Triple Notation

3 - address Code

1 t1 = b*c

2 t2 = a + t1

3 t3 = b*c

4 t4 = d / t3

5 t5 = t2 - t4

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)

Quadruples

op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Indirect Triple Notation

	op	arg_1	arg_2
0	*	b	c
1	+	a	100
2	*	b	c
3	/	d	102
4	-	101	103

100	0
101	1
102	2
103	3
104	4

SSA Code

→ It is 3 Address Code

→ Every variable has single assignment. (Defination/meaning)

a = b*c

b = a+b

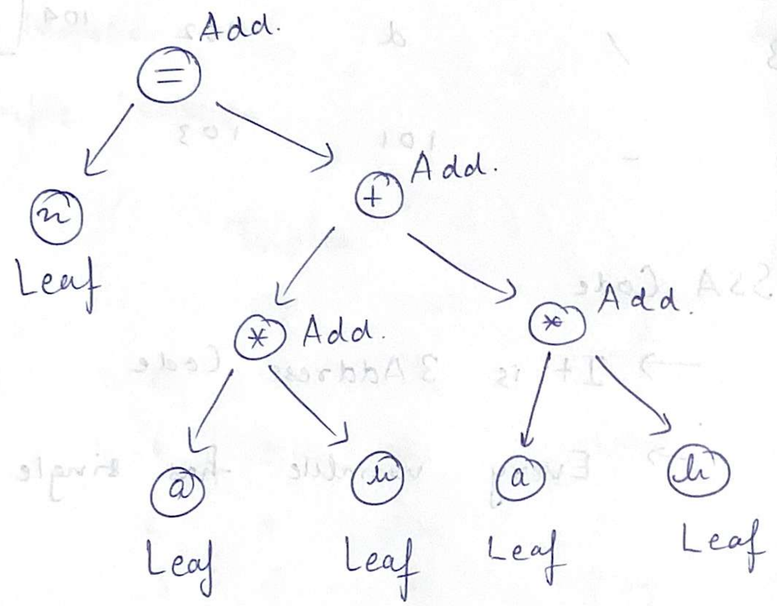
↳ not allowed in SSA code.

Syntax Tree (for expressions)

operators → non leaf

operands → leaf

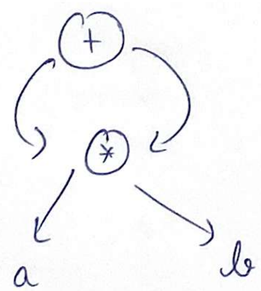
$x = a*b + a*b$



Directed Acyclic Graph (To eliminate common sub expression)

$x = a*b + a*b$

(4 nodes 4 edges)



Eg :

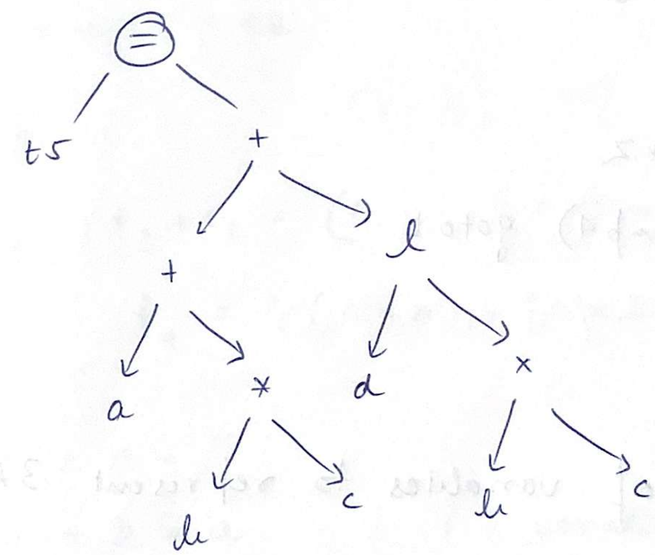
1. $t1 = b*c$
2. $t2 = a + t1$
3. $t3 = b*c$
4. $t4 = d / t3$
5. $t5 = t2 - t4$

Expression:

$t_5 = t_2 - t_4 = (a + t_1) + (d / t_3)$

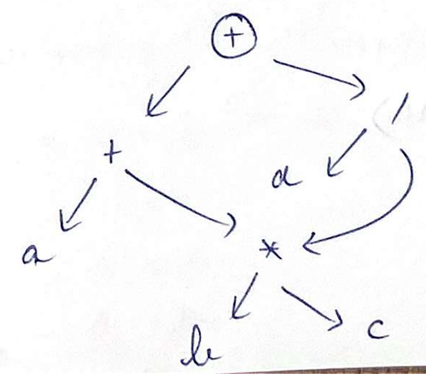
$t_5 = [a + (b \times c)] + [d / (b \times c)]$

Syntax Tree :



Syntax tree

Dag



Eg : Three Address Code.

Source :

if $(x + y * z > x * y + z)$

$a = 0$;

Three Address Code:

$tmp\ 1 = y * z$

$tmp\ 2 = x + tmp\ 1$

$tmp\ 3 = x * y$

$tmp\ 4 = tmp\ 3 + z$

if $(tmp\ 2 <= tmp\ 4)$ goto L

$a = 0$

L :

Eg Find number of variables to represent 3AC & SSA code.

$x = a * b + a * b$

$a = a * b$

$a = a + a$ (Best 3AC)

2 variables

$a_1 = a * b$

$a_2 = a_1 + a_1$

a, b, a_1, a_2 (4 Variables)

Q How many min. no. of variables in equivalent 3AC?

$t1 = a * a$

$t2 = a * b$

$t3 = t1 * t2$

$t4 = t3 + t2$

Exp.

$t_4 = t_3 + t_2 = (t_1 * t_2) + (a * b)$

$t_4 = ((a * a) * (a * b)) + (a * b)$

$b = a * b$

$a = a * a$ (2 variables)

$a = a * b$

$a = a + b$

How many min. variables in SSA?

$b_1 = a * b$ a, b, a_1, a_2, a_3 (5 variables)

$a_1 = a * a$

$a_2 = a_1 * b_1$

$a_3 = a_2 + b_1$

Q Write best SSA code.

$t1 = b * c$

$t2 = a + t1$

$t3 = b * c$

$t4 = d * t3$

$t5 = t2 + t4$

$t5 = t2 + t4$

$= (a + t1) + (d * t3)$

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

$b = b * c$ $\quad$ a, b, c, d

$a = a + b$ $\quad$ (4 variables)

$b = d * b$

$a = a + b$

$b_1 = b * c$

$a_1 = a * b_1$ $\quad$ a, b, c, d

$b_2 = d * b_1$ $\quad$ b_1, a_1, b_2, a_2

$a_2 = a_1 + b_2$ $\quad$ (8 variables)

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;
```

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;
z = x;
Goto _L1;

_L0:
    z = y;

_L1:
    z = z * z;
```

Three Address Code for Array

Assume Declaration: A [n1, n2]

3AC for A [i, j] is:

A[i, j] = Base Address + ((n2 * i + j) * w)

Here, n2 is size of each row and w is size of the element.

```
t1 = n2 * i
t2 = t1 + j
t3 = t2 * w
t4 = Base Address
t5 = t4 [t3]
```

Control Flow Graph

→ It is group of basic blocks.

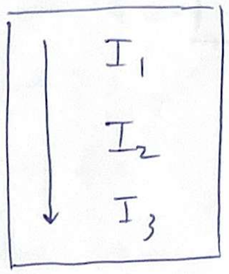
→ CFG has nodes and edges to define basic blocks and controls.

Basic Block

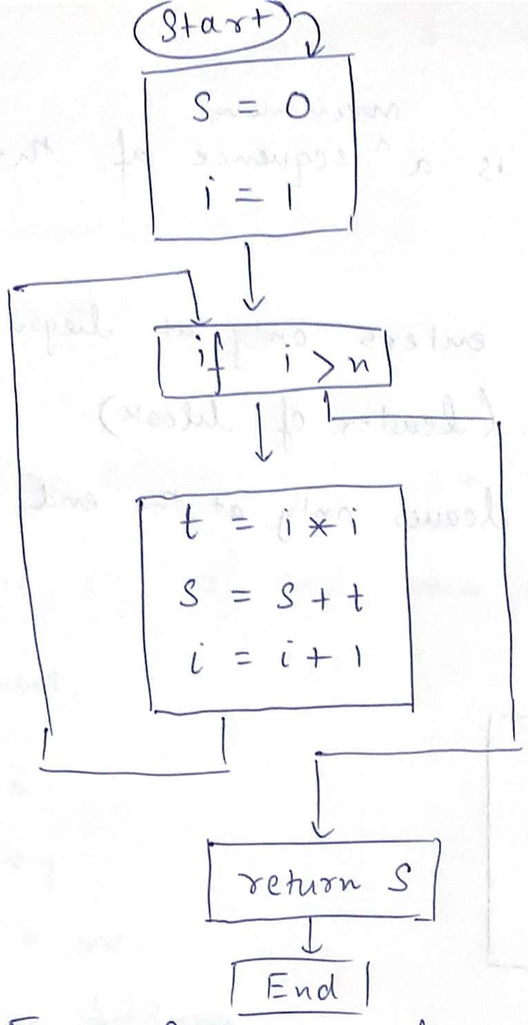
A basic block is a maximum sequence of three address codes.

→ Control enters only at begin of the sequence. (leader of block)

→ Control leaves only at the end of the sequence




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



6 Nodes

4 Basic Blocks

4 Controls

6 edges

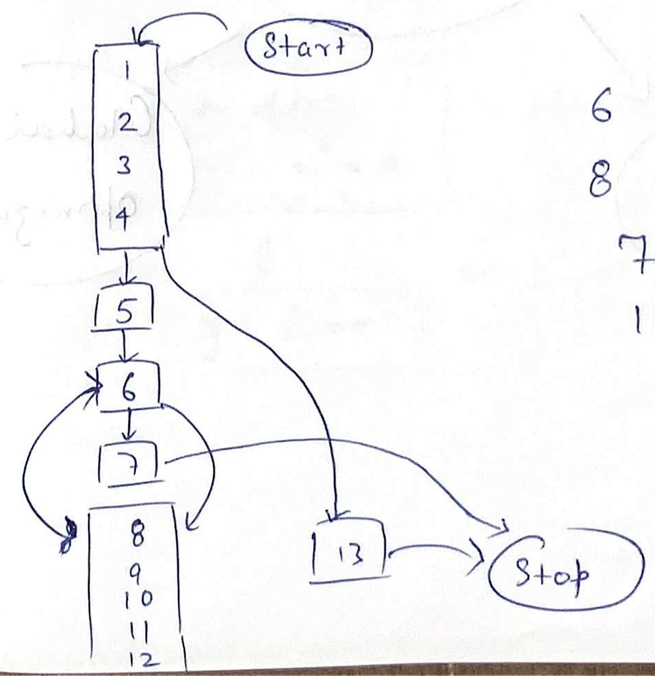
Leader : First Statement of every basic block is called as leader.

Every control enters only at leader of basic block.

Any control that leaves from basic block must goto leader of some block.

Eg

```
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
```



6 Basic Blocks

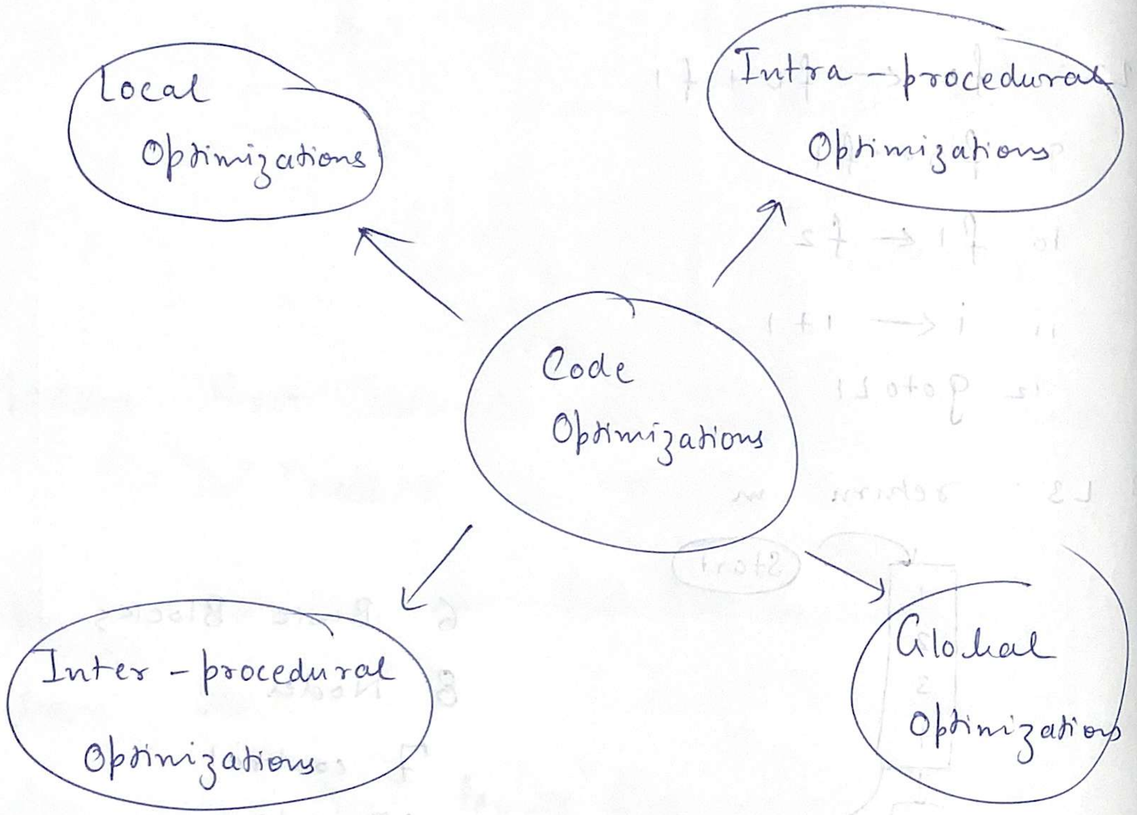
8 Nodes

7 control

10 edges

Code Optimization

- → Statement Level
- → Basic Block Level
- → Loop Level
- → Function Level
- → Program Level



Code Optimization Techniques

1. Constant Folding

$$x = 2 * 3 + y \; ; \quad \Rightarrow \quad x = 6 + y$$

2. Copy Propogation

a.) Constant Propogation

$$\boxed{\begin{array}{l} x = 2 \\ \downarrow \\ y = x * a \; ; \end{array}}$$

$\Downarrow$ Replace x with 2

$$\boxed{y = 2 * a \; ;}$$

b.) Variable Propagation.

$$\boxed{\begin{array}{l} x = b \; ; \\ y = x * a \end{array}}$$

$$\Downarrow$$

$$\boxed{y = b * a}$$

Eg

constant propogation

$x = 3;$ $\Rightarrow$ $y = 3 * 2 + a$

$y = x * 2 + a;$ $\quad y = 6 + a$ constant folding

3. Algebraic Simplifications

$y = x + a$

$z = y - a$

$w = z * b$

$\Rightarrow$

$w = z * b$

$= (y - a) * b$

$= (x + \cancel{a} - \cancel{a}) * b$

$\boxed{w = x * b}$

4. Strength Reduction (It replaces costlier instruction with cheaper)

$x = y * 2$ (costlier)

$\Downarrow$

$x = y + y$ (cheaper)

or

$x = y << 1$ (cheaper)

$x = y/4$

$\Downarrow$

$x = y >> 2$

5. Dead Code Elimination

$x = a * b$ (dead code)

↳ (not used in future)

$y = a + c$

$z = y * d$

print(z)

$\Downarrow$

$y = a + c$

$z = y * d$

print z

if (2) always true

printf ("gate");

Else

printf ("exam"); } (never be executed at run time) unreachable code

$\Downarrow$ After elimination of dead code & simplification

printf ("gate")

6. Common Sub-expression Elimination
(DAG can be used)

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

(a & b not modified)
(common)

This is not required

↓ Common Sub expression $a * b$ is computed only once

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

7. Loop Optimizations

1. Code Motion
2. Induction Variables Elimination
3. Loop merge / fusion / combine
4. Loop unrolling.

1. Code Motion

```
for (int i=0; i<n; i++) {
    n = 10 * a;
    y = y * i;
}
```

→ Loop invariant

(n times multi)

⇓ code motion.

$n = 10 * a;$ ⇒ 1 multiplication

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

2. Induction Variable Elimination

```
j = 0;
for (i=0; i<n; i++, j++) {
    n = n + i;
    y = y * j;
}
```

Induction Variable i, j, n, y

j is ⇒ eliminated

```
for (int i=0; i<n; i++)
{
    n = n + i;
    y = y + i;
}
```

3. Loop Merge

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

$\Longrightarrow$

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

4. Loop Unrolling

```
for (i=1; i<=4n; i++)
{
    printf ("gate");
}
```

(slower)

$\Longrightarrow$

Loop Unrolling

```
for (int i=1; i<=2n; i++)
{
    printf ("gate");
    printf ("gate");
}
```

(faster)

Loop unrolling $\Longrightarrow$

```
for (i=1; i<=n; i++) {
    printf ("gate");
    printf ("gate");
    printf ("gate");
    printf ("gate");
}
```