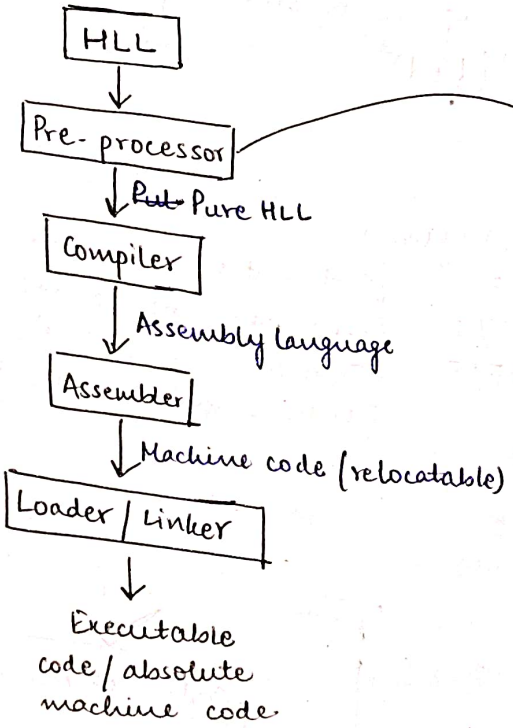
COMPILER DESIGN

⊙ Assembler is dependent on the platform.

Introduction to various phases of compiler.

Main aim of compiler = convert HLL to LLL.



File inclusion [replacing #include with relevant file]

Macro expansion [#define constants.. small functions]

Linker –

- ⊙ combines various pieces of codes & source codes to obtain executable code.
- ⊙ combines object modules

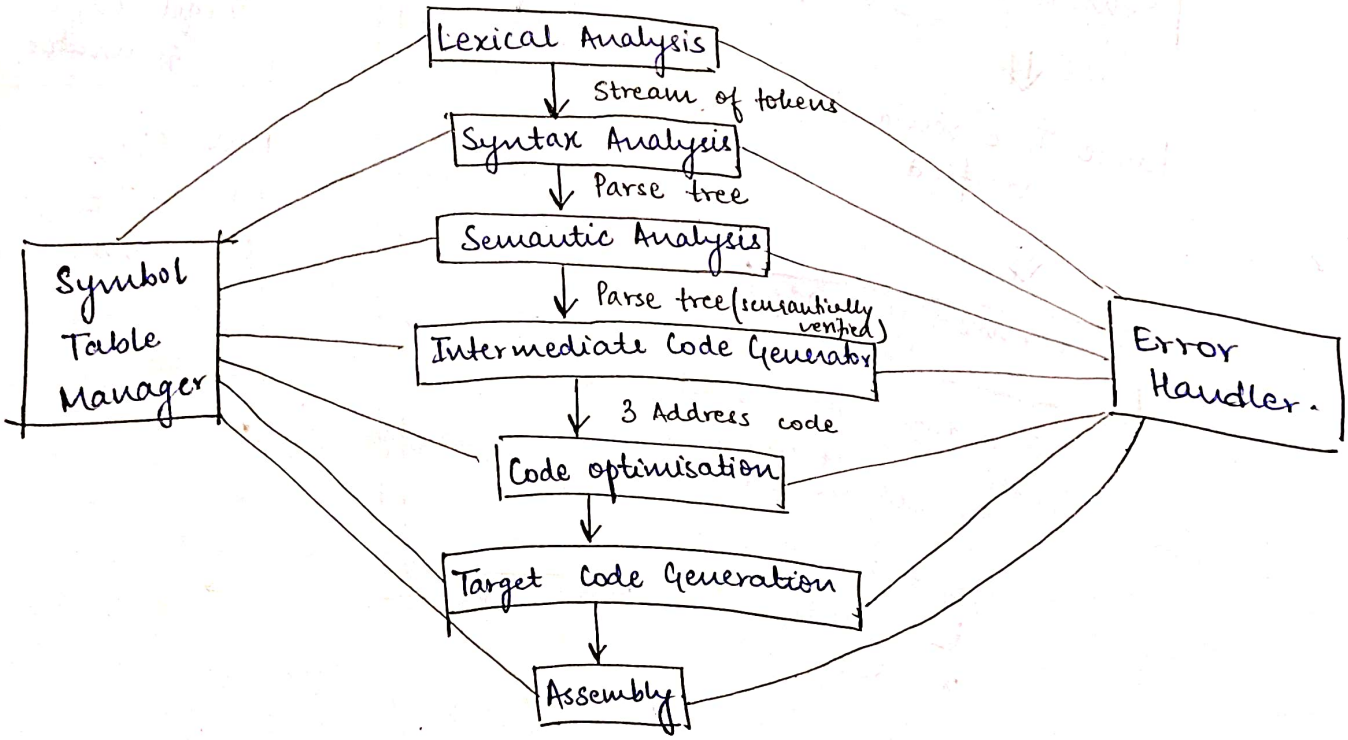
I/P → Machine code.

Loader –

part of OS responsible for loading programs to memory.

I/P → executable code generated by linker.

Compiler – phases



Example

$x = a + b * c$

⇓

Lexical analyzer → identifies the 'identifiers', 'tokens' by using some regular expressions called patterns.

$L(L+d)^*$

$id = id + id * id$

⇓

Syntax analyser ← Context Free grammar

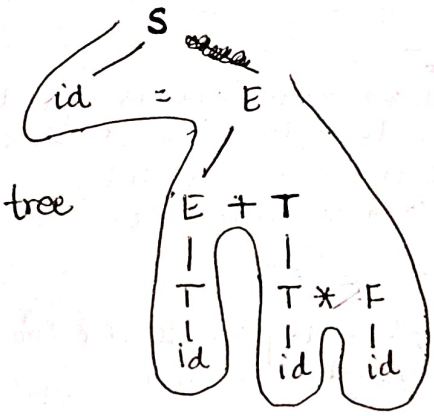
$S \rightarrow id = E;$ [Statement can be identifier = expression;]

$E \rightarrow E + T / T$ [Expression can be expression + term or term]

$T \rightarrow T * F / F$ [Term can be term * factor / factor]

$F \rightarrow id$ [Factor is an identifier]

Parse tree



⇓

Semantic analyser — Whether it is meaningful or not

⇓

Parse Tree semantically verified.

⇓

Intermediate Code generator

3 address code is most common.
In every statement only 3 addresses are present

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

⇓

Code optimisation

$t_1 = b * c$
$x = a + t_1$

⇓

Target Code generator

Mul	$R_1 R_2$	$a \rightarrow R_0$
Add	$R_0 R_2$	$B \rightarrow R_1$
Mov	$R_2 X$	$C \rightarrow R_2$

Introduction to symbol table [only theory req.]

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurances of various entities such as variable names, function names, objects, classes, interfaces etc.

The information is collected in the analysis phase of compiler and used by the synthesis phase.

Usage of symbol table in various phases —

PHASE	USAGE
Lexical analysis	Creates new entries for each new identifier.
Syntax analysis	Adds info regarding attributes like type, scope, dimension, line of reference, line of use.
Semantic analysis	Uses the available information to check for semantics and is updated.
Intermediate code generation	Information in symbol table helps to add temporary variable info (like type etc)
Code optimisation	Info in symbol table is used in optimisation by considering address and aliased variables information.
Target Code Generation	Generates the code by using the address info of identifiers.

Symbol Table Entries —

1. Name
2. Type
3. Size
4. Dimension
5. Line of declaration
6. Line of usage (linked list)
7. Address.

- All the attributes are not of same size.
- Size of symbol table should be dynamic to allow the increase in size in compilation phase.

Operations on the symbol table.

1. Non-block structured language —
   * Contains only one instance of variable declaration and its scope is throughout the program.
   * For non-block structured languages, the operations are —
     → Insert → Lookup.

2. Block structured language (C++, Java)
   * The variables may be redeclared & its scope is only in that particular ~~scope~~ block.
   * The operations are —
     → Insert → Lookup
     → Set → Reset [changing scope of a variable]

Gate 2012

Access time of a table is logarithmic if it is implemented by —

a) Linear list $O(n)$
b) Search Tree $O(\log_k n)$
c) Hash Table $O(1)$
d) NOTA

LEXICAL ANALYSER

This is the only phase the reads the input character by character.
- Ignores comments
- Eliminates white spaces.

```
int max(x,y)
int x,y;
/* find max of x & y */
{
return (x>=y?x:y);
}
```

25 tokens are present.

printf("%d Hai", &x);
① ② ③ ④ ⑤ ⑥ ⑦ ⑧

8 tokens

→ Input preprocessing (removes comments --)
→ Tokenization (generates tokens)
→ Token classification (classifies tokens as identifier, keyword, operator)
→ Token validation (checks if token are acc to rules or not var. name)
→ Output generation

Syntax Analyser is also called parser.

Grammar

- If a grammar has more than one derivation trees (parse trees) for the same string, then, the grammar is ambiguous.
- Ambiguity problems are undecidable.
- Ambiguous grammars need to be converted to unambiguous to be used by parsers.

Ambiguous grammars and making them unambiguous —

- Associativity [In order to overcome associativity, grammar should be left/right recursive]
- Precedence [Highest precedence operator should be as far from start symbol as possible]

- Grammars should not be left recursive because TDP parsers can't process left recursive grammar.

∴ Left Recursive grammars is converted into right recursive.

$A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$
$A' \rightarrow \epsilon / \alpha A'$

- Grammars should not be non-deterministic.

Non Deterministic grammars are converted into deterministic by using common prefix.

left factoring

Example-

$S \rightarrow bSSaaS /$
$bSSaSb /$
$bSb /$
a

Non deterministic because common prefix is there for production of same variable

$\Rightarrow$

$S \rightarrow bSS' / a$
$S' \rightarrow SaaS / SaSb / b / \epsilon$

$\Downarrow$

$S \rightarrow bSS' / a$
$S' \rightarrow SaS'' / b$
$S'' \rightarrow aS / Sb.$

Elimination of non-determinism does not eliminate unambiguity.

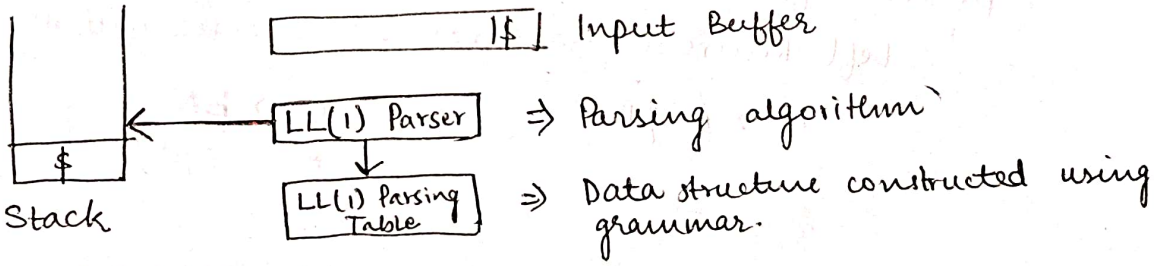
Parsers

Ambiguity X

- Top Down Parsers (TDP) — what to use
  - TDP with backtracking
    - Brute force method.
  - TDP without backtracking — Left recursion X, Non Determinism X
    - Recursive Descent
    - Non Recursive Descent (LL(1))
- Bottom up Parsers (BUP) — what to reduce — SR parser (shift reduce)
  - Operator Precedence Parser (Ambiguous)
  - LR Parser
    - LR(0)
    - SLR(1)
    - LALR(1)
    - CLR(1)

LL(1) parser / Non Recursive Descent

L — Input is scanned from Left to Right
L — Left most Derivation is used.
(1) — 1 → No. of look aheads (characters we can see when making a decision)



For LL(1) parsing table — FIRST and FOLLOW functions are used.
First & Follow sets are needed so that the parser can apply needed production rule at the correct position

FIRST( ) –

First(α) is a set of terminal symbols that begin in strings derived from α.

Example–

A → abc / def / ghi

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

Rules for calculating first function —

Rule 01 For the production rule $X \to \epsilon$,
$First(X) = \{\epsilon\}$.

Rule 02 For any terminal symbol a,
$First(a) = \{a\}$.

Rule 03 For the production $X \to Y_1 Y_2 Y_3$

$First(X) = First(Y_1)$ if $\epsilon \notin First(Y_1)$

$First(X) = \{First(Y_1) - \epsilon\} \cup First(Y_2 Y_3)$ if $\epsilon \in First(Y_1)$

$S \to aABCD$
$A \to b$
$B \to c$
$C \to d$
$D \to e$

$First(S) = a$
$First(A) = b$
$First(B) = c$
$First(C) = d$
$First(D) = e$

FOLLOW( ) -

Follow(a) is a set of terminal symbols that appear immediately to the right of a.

Rules for calculating follow function —

Rule 01 For the start symbol S, place $ in Follow(S)

Rule 02 for any production $A \to aB$
$Follow(B) = Follow(A)$

Rule 03 for any production $A \to \alpha B \beta$,
$Follow(B) = First(\beta)$ if $\epsilon \notin First(\beta)$
$Follow(B) = \{First(\beta) - \epsilon\} \cup Follow(A)$ if $\epsilon \in First(\beta)$

Follow function cannot contain ϵ

Question — Calculate First & follow functions for the given grammar —
$S \to aBDh$ $B \to cC$ $C \to bC/\epsilon$ $D \to EF$ $E \to g/\epsilon$ $F \to f/\epsilon$

$First(S) = a$
$First(B) = c$
$First(C) = \{b, \epsilon\}$
$First(D) = \{g, f, \epsilon\}$
$First(E) = \{g, \epsilon\}$
$First(F) = \{f, \epsilon\}$

$follow(S) = \{\$\}$
$follow(B) = \{g, f, h\}$
$follow(C) = \{g, f, h\} = B$
$follow(D) = \{h\}$
$follow(E) = \{f, h\}$
$follow(F) = \{h\}$.

Productions	Firsts	Follows
$S \to A$ $A \to aB/Ad$ (b: left Recursive) $B \to b$ $C \to g$	First(S) = {a} First(A) = {a} First(A') = {d, ϵ} first(B) = {b} first(c) = {g}	Follow(S) = {$} Follow (A) = {$} Follow (A') = {$} Follow (B) = {d, $} Follow (C) = {N A}

After elimination:

$S \to A$
$A \to aBA'$
$A' \to dA'/\epsilon$
$B \to b$
$C \to g$

$ \cup \{First(L') - \epsilon\} \cup Follow(L) \cup Follow(S)$
$\{\$, ,,)\}$

$S \to (L)/a$ $L \to SL'$ $L' \to ,SL'/\epsilon$	First(S) = {(, a} First(L) = {(, a} First(L') = {,, ϵ}	Follow(S) = {} Follow (L) = {) } Follow(L') = {) }

$S \to AaAb/BbBa$ $A \to \epsilon$ $B \to \epsilon$	First(S) = {a, b} First(A) = {ϵ} First(B) = {ϵ}	Follow(S) = {$} Follow (A) = {a, b} Follow (B) = {a, b}

$S \to Bb/Cd$ $B \to aB/\epsilon$ $C \to cC/\epsilon$	First(S) = {a, b, c, d} First(B) = {a, ϵ} First(C) = {c, ϵ}	Follow (S) = {$} Follow (B) = {b} Follow (C) = {d}

$E \to TE'$ $E' \to +TE'/\epsilon$ $T \to FT'$ $T' \to *FT'/\epsilon$ $F \to id/(E)$	First(E) = { id, (} First(E') = {+, ϵ, ϵ} First(T) = {id, (} First(T') = {*, ϵ, ϵ} First(F) = {id, (}	Follow(E) = {$,)} Follow(E') = {$,)} Follow (T) = {+, $,)} Follow(T') = {+, $} Follow(F) = {*, +, $,)}

$S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

First(S) = {d, g, h, ε, b, a}
First(A) = {d, g, h, ε}
First(B) = {g, ε}
First(C) = {h, ε}

Follow(S) = {$}
Follow(A) = {h, g, $}
Follow(B) = {$, a, h, g}
Follow(C) = {g, $, b}

Construction of LL(1) Parsing Table

Variables are in ~~columns~~ rows (vertical)
Terminals are in ~~rows~~ column (horizontal)

Each production should be in
Row of left hand side ~~of~~ variable
column in first of right hand side. [For ε production, follow of L.H.S.]

	First	Follow
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \epsilon$	{+, ε}	{$,)}
$T \rightarrow FT'$	{id, (}	{+,), $}
$T' \rightarrow *FT' \mid \epsilon$	{*, ε}	{+,), $}
$F \rightarrow id \mid (E)$	{id, (}	{*, +,), $}

LL(1) Parsing Table-

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

Simple example

$S \to (S) / \epsilon$

LL(1) Parsing Table

	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

Every cell has only one entry ∴ LL(1) parser can be constructed.

$S \to AaAb / BbBa$
$A \to \epsilon$
$B \to \epsilon$

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

Grammars which are Left Recursive & Non Deterministic cannot be used for LL(1) Parsing.

Question:- Check whether the grammars are LL(1) or not —

1) $S \to aSbS / bSaS / \epsilon$
First(S) = {a, b, ε} Follow(S) = {a, b, $}
Production $S \to aSbS$ should be put in S row & a column
Production $S \to \epsilon$ should also be put in S row & a column
∴ Not LL(1) grammar.

2)

$S \to aABb$	{a}	{$}
$A \to c / \epsilon$	{c, ε}	{a, b}
$B \to d / \epsilon$	{d, ε}	{b}

No 2 entries in the same cell
∴ LL(1) grammar

3)

$S \to A / a$	{a}	{$}
$A \to a$	{a}	{$}

$S \to A$ in S row a column
$S \to a$ in S row a column

Not LL(1) grammar.

5)

Recursive Descent Parser

- Top Down Parser
- Every variable has a function.

Grammar: $E \rightarrow iE'$
$E' \rightarrow +iE' / \epsilon$

```
E( ){
    if(l=='i'){
        match('i');
        E'();
    }
}
```

```
E'( ){
    if(l=='+'){
        match('+');
        match('i');
        E'();
    }
    else
        return
}
```

```
match(char t){
    if(l==t){
        l=getchar();   // move to next character
    }
    else{
        printf("Error");
    }
}
```

```
int main( ){
    E();
    if(l=='$')
        printf("Parsing successful");
}
```

l → look ahead

Operator Precedance Parser

- Bottom up parser
- Parser that interprets an operator grammar.
- Ambiguous grammars are allowed

→ Creates operator Relation Table.

Operator grammar

The grammar that is used to define mathematical operators is called operator grammar or operator precedence parser.

→ No null productions
→ No 2 adjacent non-terminals on R.H.S.

Example –
$E \rightarrow E+E / E*E / id$

Grammar

$E \to E+E / E*E / id$

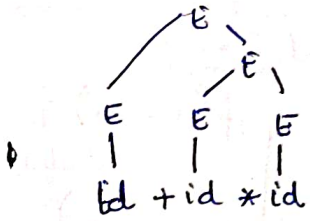
→ id has higher precedence than other operators
→ $ has least precedence than other operators.

Operator Relational Table

	id	+	*	$
id	–	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	–

$W = id + id * id$

$	id	+	id	*	id



(Bottom up parsing)

Algorithm

→ Whenever the top of the stack is ⋖ than the look ahead, then, push it.

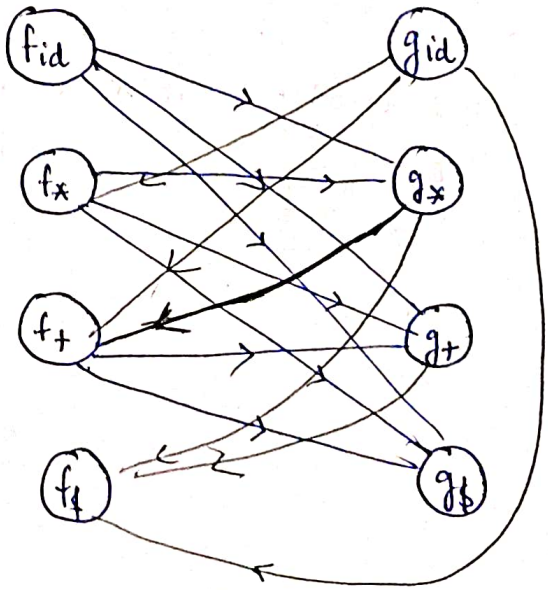
→ Whenever the top of the stack is ⋗ than the look ahead, then, pop it & then push it.

!! Disadvantage of Operator Relation Table is that if there are n operators, $O(n^2)$ space is required.

To solve the problem, Operator Function Table is used.

	id	*	+	$
id	–	⋗	⋗	⋗
*	⋖	⋗	⋗	⋗
+	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	–

f ↓



⋗ : $F_{id} \to g_{id}$ (F → G)

⋖ : $g_{id} \to f_{id}$ (G → F)

The longest path from f_{id} is $f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

The longest path from g_{id} is $g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

The longest path from each node is calculated and Operator functions Table is created.

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

Now, if we need to compare (+, +),

f_+ g_+

$2 > 1$ $\therefore$ (+ :> +)

The size of the table for n operators is of the order O(2n)

In Operator Function Table, there cannot be any blank entries. So, the error detecting capability of Operator Function Table is less than that of Operator Relation Table.

EDC [Operator Functional Table] < EDC [Operator Relation Table]

Ques :- Create Operator Relation Table –

P → SR / S
R → bSR / bS
S → wbS / w
W → L * W / L
L → id

⇒

P → SbP / SbS / S
S → WbS / W
W → L * W / L
L → id.

	id	*	b	$
id	–	:>	:>	:>
*	<·	<·	:>	:>
b	<·	<·	<·	:>
$	<·	<·	<·	–

W → L * W
$\therefore$ * is right associative

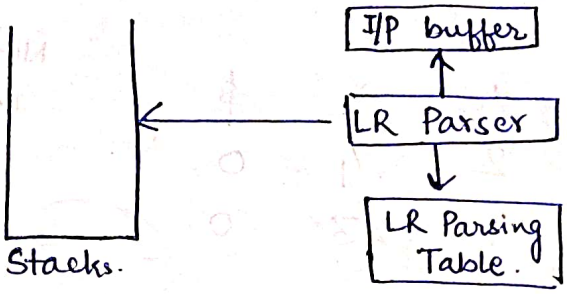
S → WbS , P → SbP
$\therefore$ b is right associative

LR Parsers

- LR(0)
- SLR(1) — Simple LR
- LALR(1) — Look Ahead LR
- CLR(1) — Canonical LR.

LR(0), SLR(1) } use LR(0) items

LALR(1), CLR(1) } use LR(1) items.



Only difference is LR Parsing Table.

Grammar:- $S \to AA$, $A \to aA/b$ } In LR Parsers, we have CLOSURE and GOTO operations

Augmented grammar is $S' \to .S$

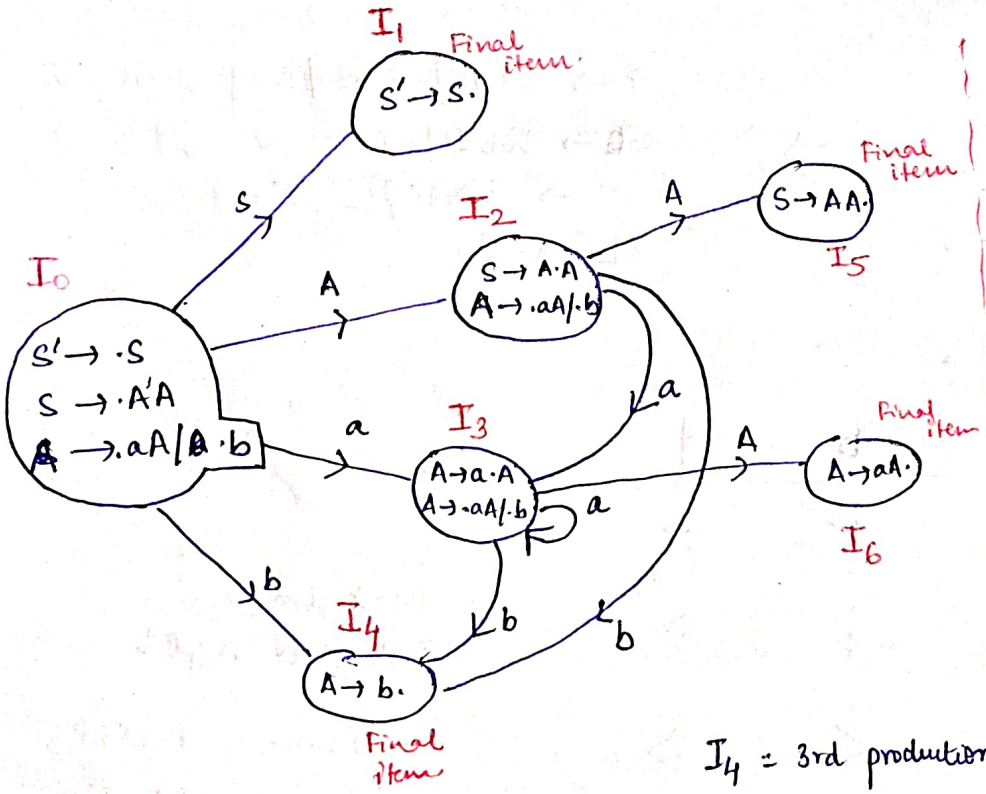
① $S \to AA$

$A \to aA/b$ ② ③

Any production with a dot(·) in the R.H.S is called an item.

LR(0) Parsing Tree is —

canonical collection of LR(0) items.



I_4 = 3rd production

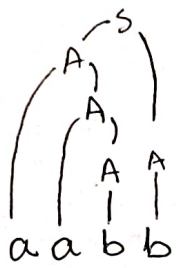
Parsing Table

	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

$S' \to S$

① $S \to AA'$

$A \to aA/b$ (② ③)



Let the given string be aabb$

I/P pointer ↑↑↑↑↑

0	a	3	a	3	b	4	A	6	A	6	A	2	b	4	A	5	$	1

Accept

- Always the top of stack contains state (and the first state is 0).
- Initially, I_0 on 'a' is S_3 [means shift the i/p you are looking at & as well as the state no. on the stack. I/P = a state no. = 3 & increment i/p pointer]
- If R action is ~~to~~ to take place, pop 2*(No. of terminals/non-terminals from RHS-$ of that production) elements.

  If length of RHS = x, pop 2x symbols from stack & push the LHS symbol on the stack & see the last no. of the state

When we see reduce move, we do not increment the input pointer

SLR(1)

The main difference b/w LR(0) and SLR(1) parsing tables —

"Reduce moves are placed only if the next symbol is the follow of current b symbol."

Parsing Table

	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	R_3	R_3	R_3		
5			R_1		
6	R_2	R_2	R_2		

$A \to b$ R_3 is placed only in Follow(A) = {a, b, $}

$S \to AA$ R_1 is placed only in Follow(S) = {$}

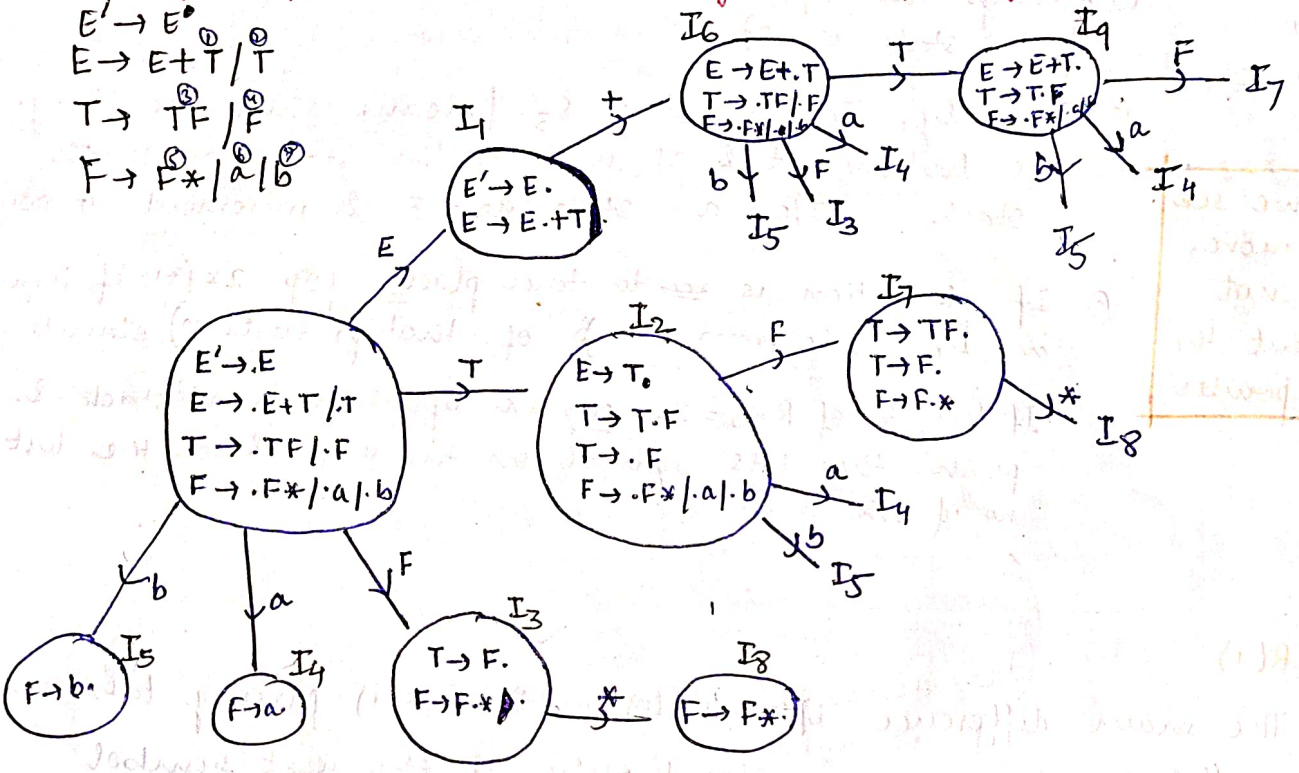
All grammars are not suitable for LR parsing due to shift-reduce conflict.

	a	b
5	S_6/r_1	r_1

SR-conflict.

Examples of LR(0) and SLR(1) ~~grammar~~ parsing.

$E' \to E$ ①
$E \to E+T$ ② / T ③
$T \to TF$ ④ / F ⑤
$F \to F*$ ⑥ / a ⑦ / b ⑧



$I_2 \to$ SR conflict ∴ the grammar is ~~CP~~ not LR(0)

Reduce move – $E \to T.$

shift move → $F \to .a/.b$

To check if ~~SLR(1)~~ grammar is SLR(1) or not, a part of parsing table is constructed (~~which~~ for those states which have SR conflict)

	ACTION				
	a	b	*	+	$
2	S_4	S_5		R_2	R_2
3	R_4	R_4	S_8	R_4	R_4
9	S_4	S_5		R_1	R_1
7	R_3	R_3	S_8	R_3	R_3

Grammar is ∴ SLR(1)

Grammar is not LR(0)

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

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

Follow(E) = {+, $}

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

Question:- Check whether the grammar is LL(1), LR(0) or SLR(1)

$S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

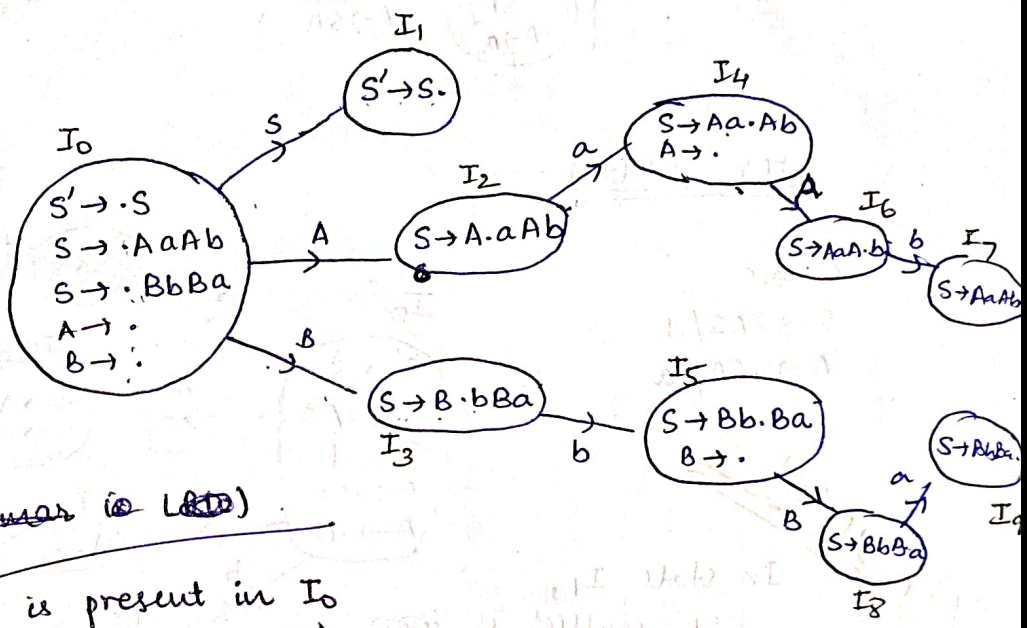
LL(1) Parsing Table

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

∴ The grammar is LL(1)

For LR(0)

① $S' \rightarrow .S$
② $S \rightarrow .AaAb$
③ $S \rightarrow .BbBa$
④ $A \rightarrow .$
⑤ $B \rightarrow .$



No SR conflict

∴ ~~The grammar is LR(0)~~

But RR conflict is present in I_0
(2 reduce moves)

∴ The grammar is not LR(0)

For SLR(1)

	a	b	$
0	r_4/r_5	r_4/r_5	

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

∴ The grammar is not SLR(1) (RR conflict)

Grammar– $S \to AS / b$, $A \to SA / a$ — LL(1)? LR(0)? SLR(1)?

For LL(1) —

Parsing Table –

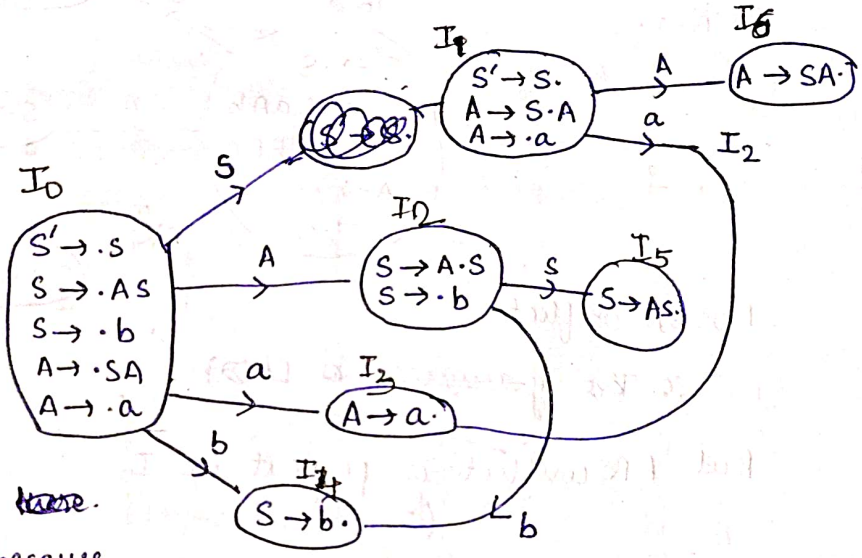
	a	b	$
S	S → AS	S → AS / S → b	
A	A → SA / A → a	A → SA	

conflict

∴ The given grammar is not LL(1)

For LR(0) —

$S' \to .S$
$S \to .AS / .b$
$A \to .SA / .a$



WRONG!!

In state I_1,
SR conflict is ~~there~~ not there [because $S' \to S.$ is not reduce action]

∴ The given grammar is LR(0)

For SLR(1) —

Since there are no SR/RR conflict, the grammar is SLR(1) as well

Grammar

LR(0) ? LL(1)? SLR(1) ?

$S \to AS / b$
$A \to SA / a$

For LL(1) — Parsing Table.

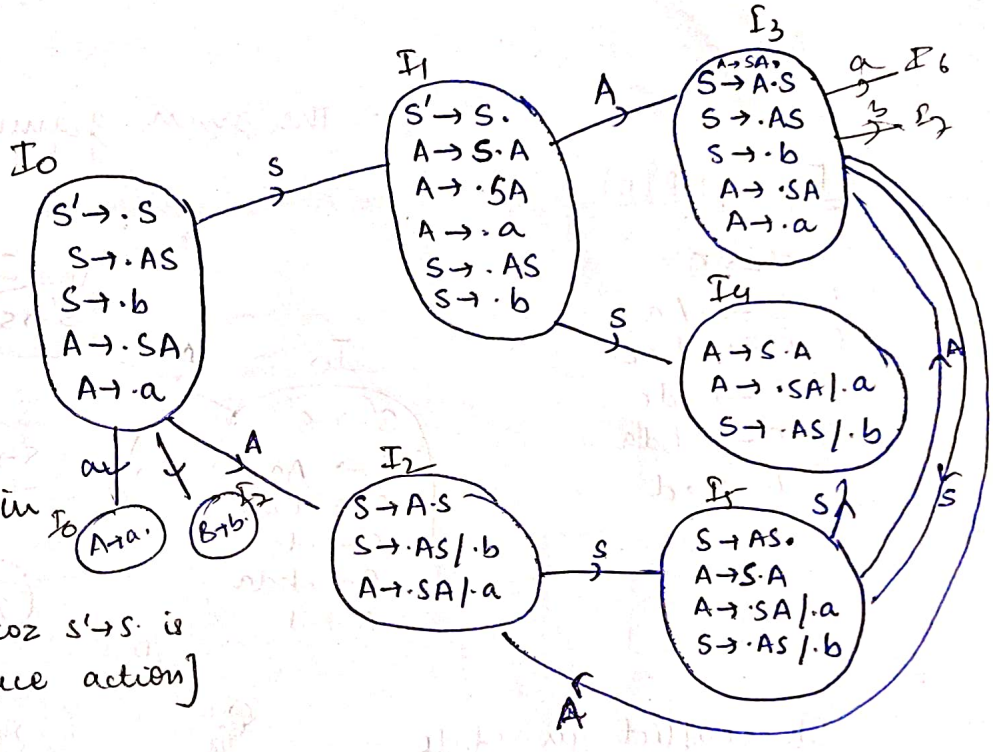
	a	b	$
S, A	$S \to AS$	$S \to AS / S \to b$	

→ conflict.

∴ Not LL(1)

For LR(0) —

$S' \to .S$
$S \to .AS$ ① / $.b$ ②
$A \to .SA$ ③ / $.a$ ④



SR conflict in state I_3.

[Not in I_1 bcoz $S' \to S$. is not reduce action]

∴ Not LR(0)

For SLR(1)

	a	b	$
3	R_3/S_6	R_3/S_7	

Follow(A) = {a, b}

SR conflict

∴ Grammar is not SLR(1).

Note :- The grammar is ambiguous
Therefore, it can't be parsed by any parser.

Grammar

LL(1)? LR(0)? SLR(1)?

$S \to Aa \mid bac \mid dc \mid bda$

$A \to d$

For LL(1) Parsing Table.

	a	b	c	d	$
S		S→bac / S→bda		S→Aa / S→dc	
A					

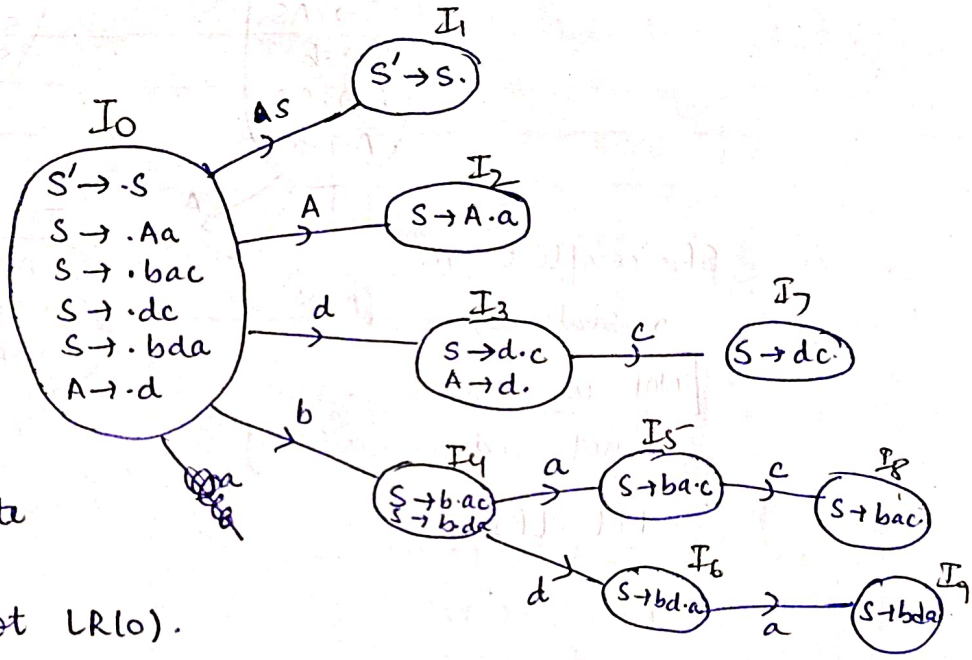
conflict.

∴ The given grammar is not LL(1).

For LR(0)

$S' \to .S$

① $S \to .Aa$
② $S \to .bac$
③ $S \to .dc$
④ $S \to .bda$
⑤ $A \to .d$



SR conflict in state I_3

∴ grammar is not LR(0).

For SLR(1)

	a	b	c	d	$
3	R_5		S_7		

Follow(A) = {a}

∴ No conflict

∴ grammar is SLR(1)

LALR(1)
CLR(1)
} Canonical collection of LR(1) items is used

LR(1) item = LR(0) item + look ahead

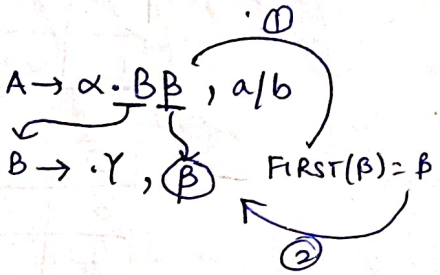
LR(0) item = item ending with .

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

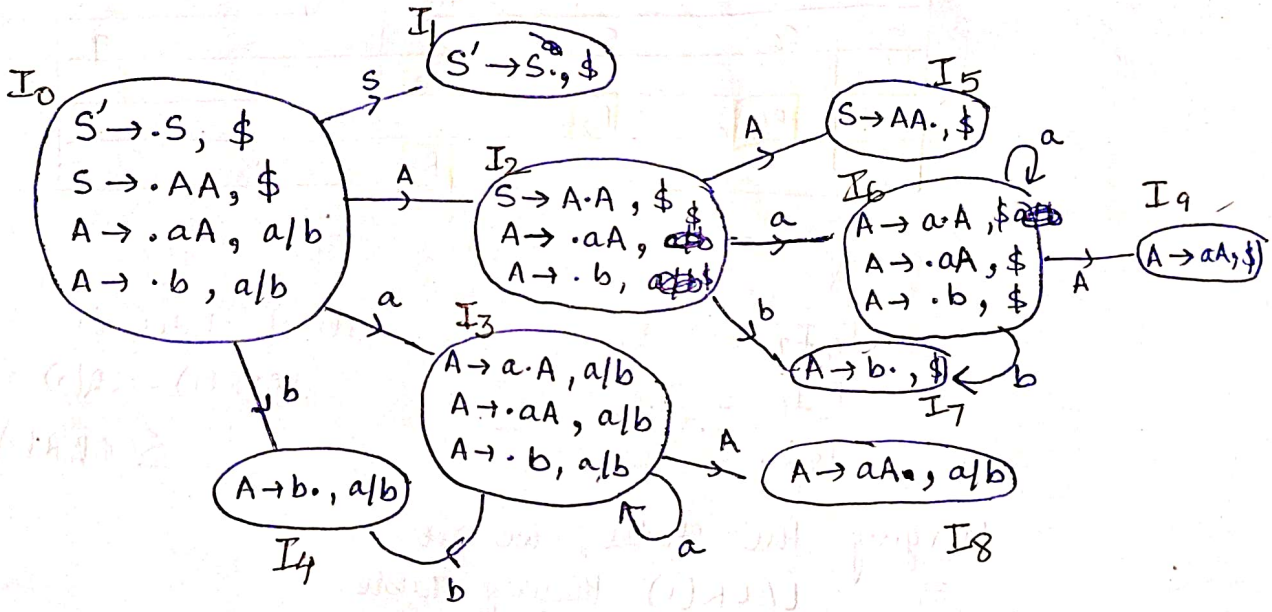
Augmented grammar is

$S' \to .S$ (1)
$S \to .AA$ (0) (1)
$A \to .aA$ (2)
$\quad .b$ (3)

In case we are doing closure of $A \to \alpha . B\beta$, a/b



The canonical collection of LR(1) items for the given grammar is —



Important points —

I_4 , I_7 → same in LR(0) but different states in LR(1)
I_3 , I_6 → same in LR(0) but ✗ different in LR(1)

The GOTO part and shift operation are same as LR(0), SLR(1) parsing tables. The main difference lies in the placement of reduce moves.

Reduce moves in LR(0) = entire row
" " " SLR(1) = follow of LHS
" " " CLR(1) & LALR(1) = only in look ahead symbols.

CLR(1) Parsing Table

	ACTION			GOTO	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

$I_4 I_7 = I_{47}$
$I_3 I_6 = I_{36}$
$I_8 I_9 = I_{89}$.

No. of states in
LALR(1) = LR(0) = SLR(1)
$\leq$ CLR(1)

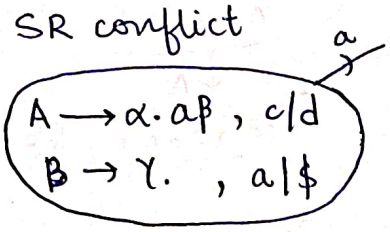
Merging the states, we get.
LALR(1) Parsing Table.

	a	b	$
0	S_{36}	S_{47}	
1			Accept
2	S_{36}	S_{47}	
36	S_{36}	S_{47}	
47	R_3	R_3	R_3
5			R_1
89	R_2	R_2	R_2

Conflicts in LR(1)

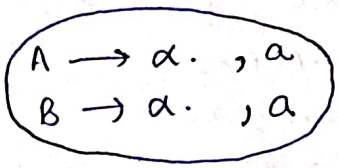
Conflicts are ~~not~~ very less frequent ~~than~~ than LR(0) items.

SR conflict



Shift move in a column
Reduce move in a column
∴ SR conflict.

RR conflict.



2 Reduce moves in the same state for same look aheads
∴ RR conflict.

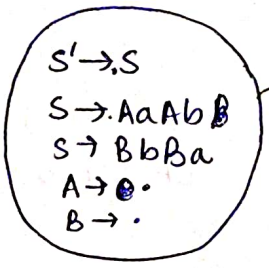
Grammar

S' → S
S → AaAb / ①
BbBa ②
A → ϵ ③
B → ϵ ④

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

The grammar is LL(1) because
S → AaAb is placed in 5th row & 'a' column
S → BbBa is placed in 5th row & 'b' column
A → ϵ is placed in 4th row in {a, b} columns
B → ϵ is placed in 8th row in {a, b} columns.

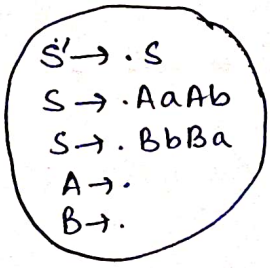
Canonical collection of LR(0) items



→ 2 reduce moves in the state I_0 (RR conflict)

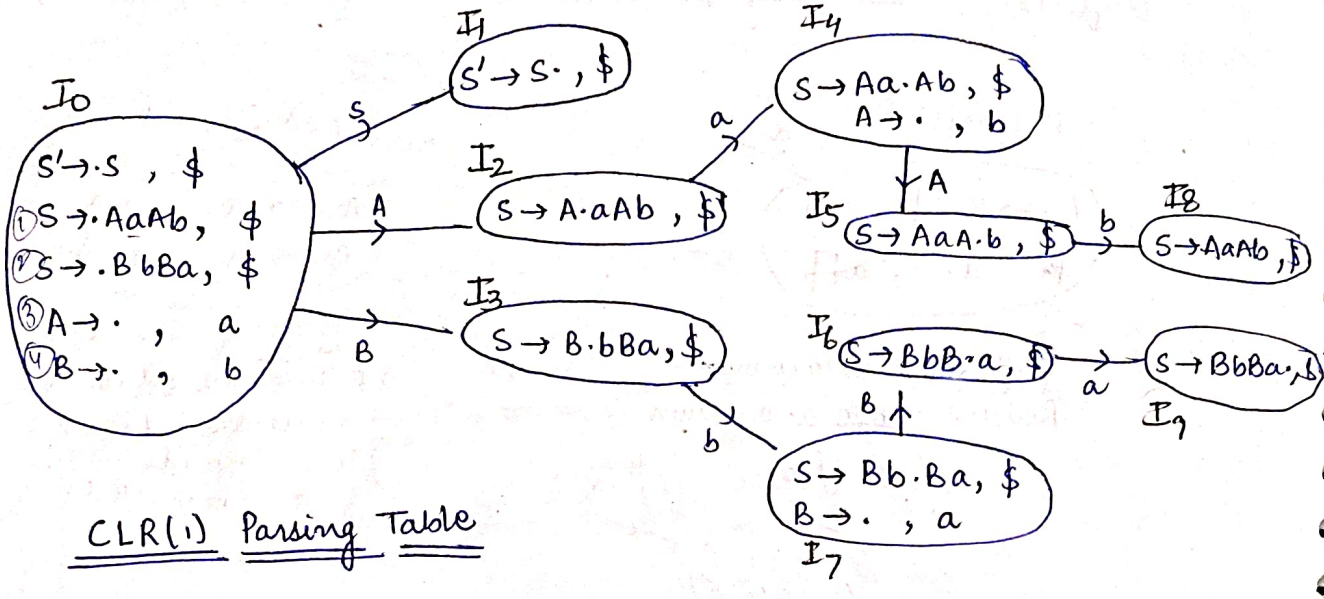
∴ The given grammar is not LR(0)

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



∴ The given grammar is not SLR(1)
bcoz Follow (A) ∩ Follow (B) ≠ ϕ.

Canonical collection of LR(1) items



CLR(1) Parsing Table

	a	b	$
0			
1			Accept
2	S_4		
3		S_7	
4		R_3	
5		S_8	
6	S_9		
7	R_4		
8			R_1
9			R_2

∴ The given grammar is CLR(1)

Also, since the no. of LR(0) item = LR(1) items, no. of states is same for all parser.
∴ LALR(1) grammar

Grammar

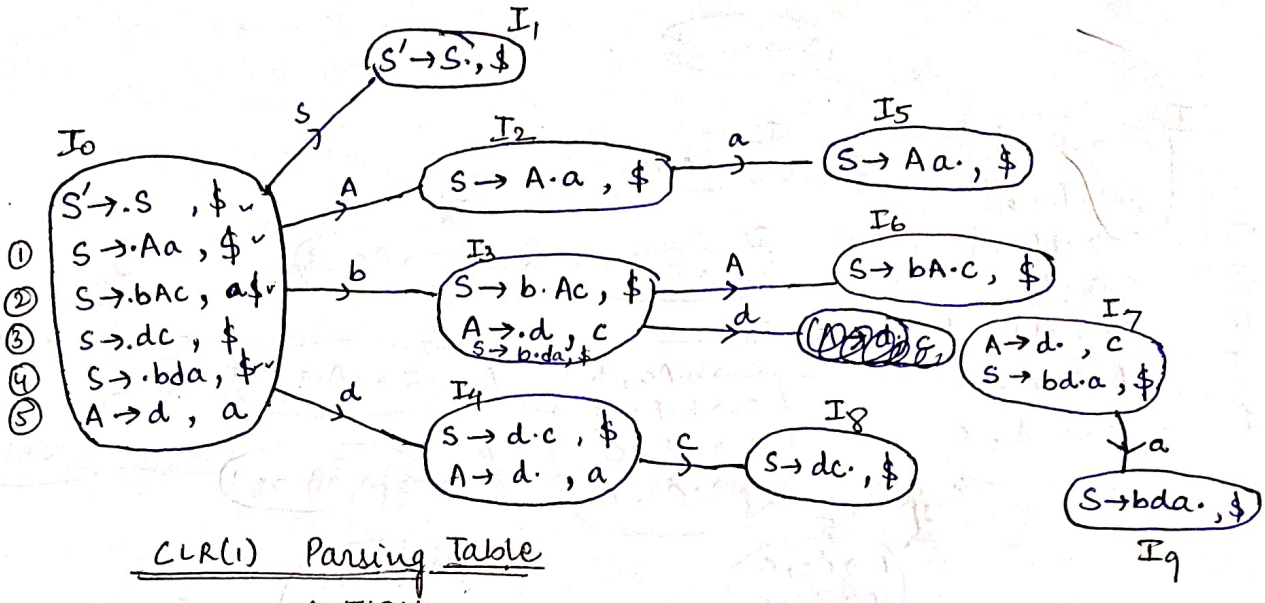
LL(1) ? LR(0) ? SLR(1) ? CLR(1), LALR(1) ?

S → Aa/
bAc/
dc/
bda
A → d.

Not LL(1) because S → bAc & S → bda in same cell.

Already proved and it is not SLR(1) & LR(0)

Canonical collection of LR(1) items –



CLR(1) Parsing Table

	ACTION				
	a	b	c	d	$
I4	R_5		S_8		
I7	S_9		R_5		

∴ The given grammar is CLR(1)

Also, there are no mergings,

∴ the grammar is LALR(1) as well

LL(1)	X
LR(0)	X
SLR(1)	X
CLR(1)	✓
LALR(1)	✓

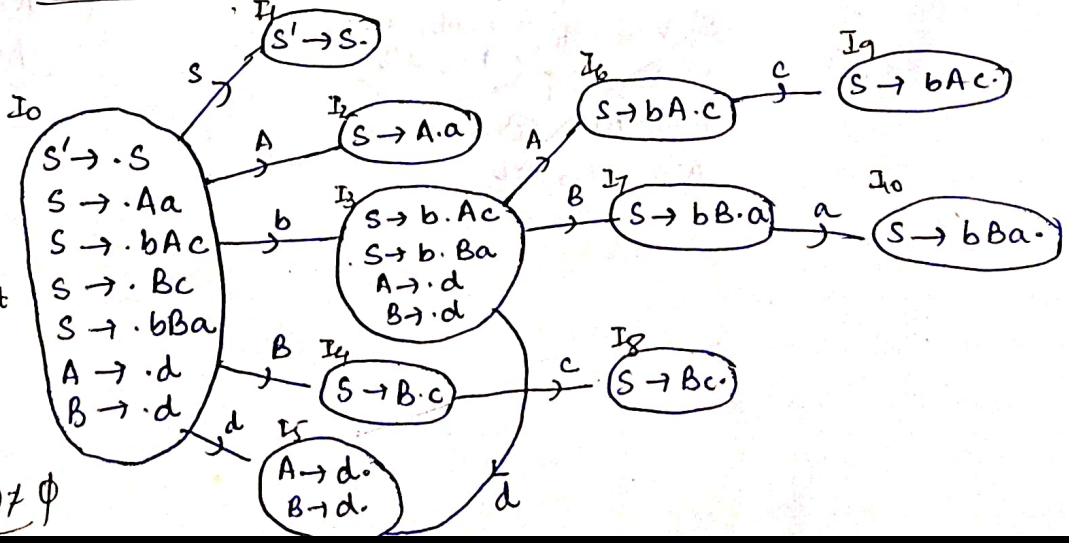
Grammar

S' → S
S → Aa / bAc / Bc / bBa
A → d
B → d

X X X
LL(1)? LR(0)? SLR(1)? CLR(1) LALR(1)

Not LL(1) because S → bAc & S → bBa will be in same cell.

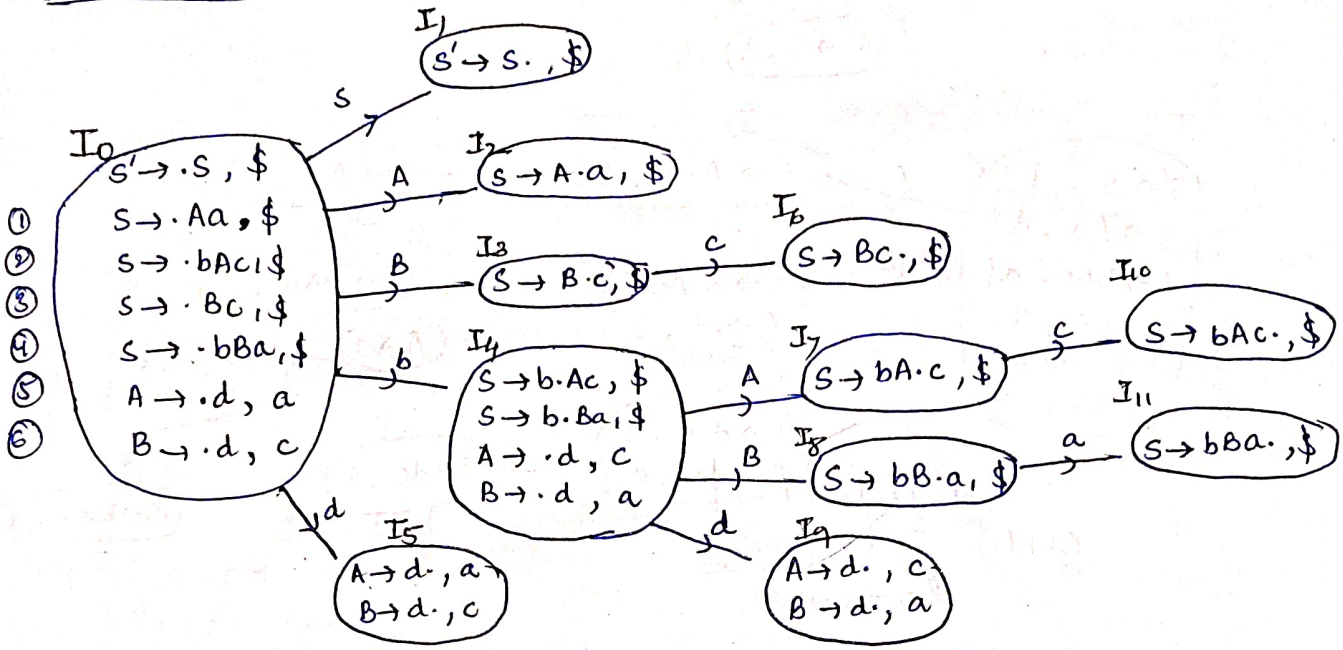
Canonical collection of LR(0) items –



The grammar is not LR(0) because RR conflict in I_5.

Follow(A) = {a, c}
Follow(B) = {c, a}
Follow(A) ∩ Follow(B) ≠ ∅
∴ Not SLR(1)

Canonical collection of LR(1) items



CLR(1) Parsing Table

	ACTION				
	a	b	c	d	$
5	R_5		R_6		
9	R_6		R_5		

∴ The grammar is CLR(1)

LALR(1) Parsing Table

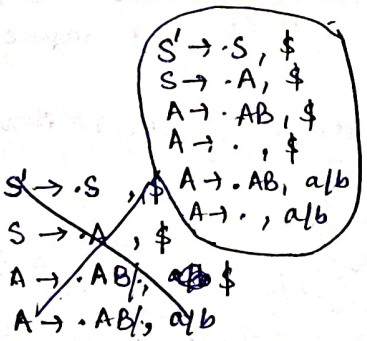
	ACTION				
	a	b	c	d	$
59	R_5/R_6		R_6/R_5		

RR conflict

∴ The grammar is not LALR(1)

Grammar

S → A
A → AB | ϵ
B → aB | b



≃

S' → .S , $
S → .A , $
A → .AB , $/a/b
A → . , $/a/b

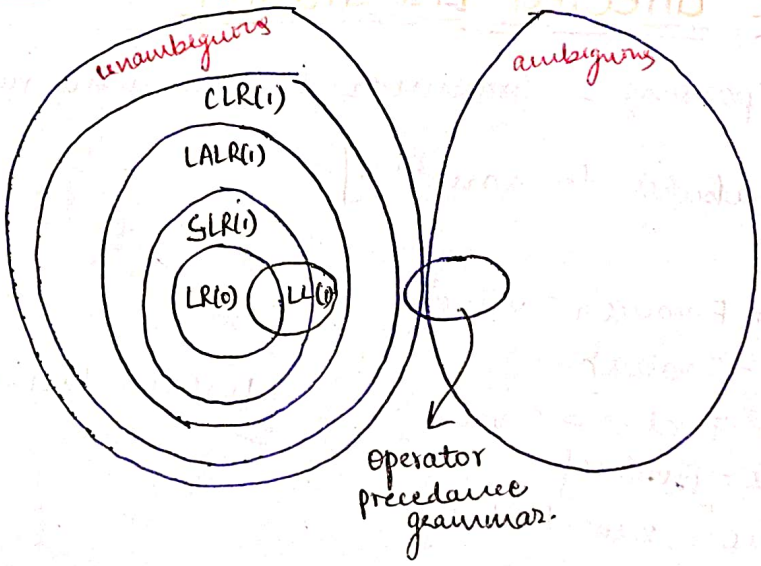
$E' \to .E$
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to i$

$E' \to .E, \$$
$E \to .E+T, \$$
$E \to .T, \$$
$E \to .E+T, +$
$E \to .T, +$
$T \to .T*F, \$$
$T \to .F, \$$
$T \to .T*F, +$
$T \to .F, +$
$T \to .T*F, *$
$T \to .F, *$
$F \to i, \$$
$F \to i, +$
$F \to i, *$

$\simeq$

$E' \to .E, \$$
$E \to .E+T, \$/+$
$E \to .T, \$/+$
$T \to .T*F, \$/+/*$
$T \to F, \$/+/*$
$F \to i, \$, +, *$

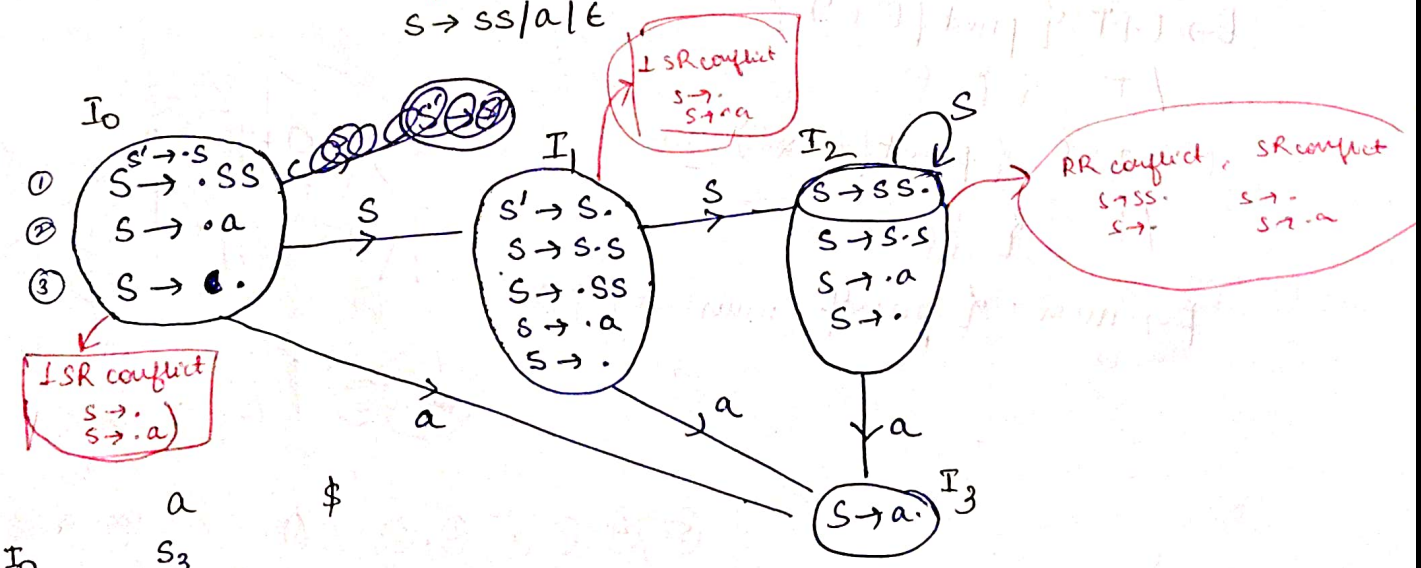
Relation between grammars



- All LL(1) grammars are LALR(1)
- Some operator precedence grammars are ambiguous & some can be unambiguous

Question:- Find the number of SR and RR conflicts in the DFA with LR(0) items (canonical collection of LR(0) items)

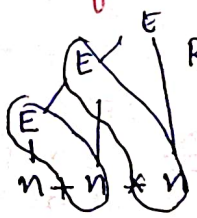
$S \to SS \mid a \mid \epsilon$



	a	$
I_0	S_3	
I_1		
I_2		
I_3		

$E \to E+n$ / $E*n$ / n

For the string $n+n*n$, the handles in right sequential form of reduction are –



Rigat sequential form = Bottom up parsing

Handles → The tokens selected to be reduced.

∴ Handles are n, $E+n$, $E*n$.

Syntax directed translation

Syntax directed parsing = Grammar + Semantic rules.

[Adding attributes to variables]

$E \to E+T$ { E.value = E.value + T.value }
/ T { E.value = T.value }

$T \to T*F$ { T.value = T.value * F.value }
/ F { T.value = F.value }

$F \to num$ { F.value = num.Lvalue }

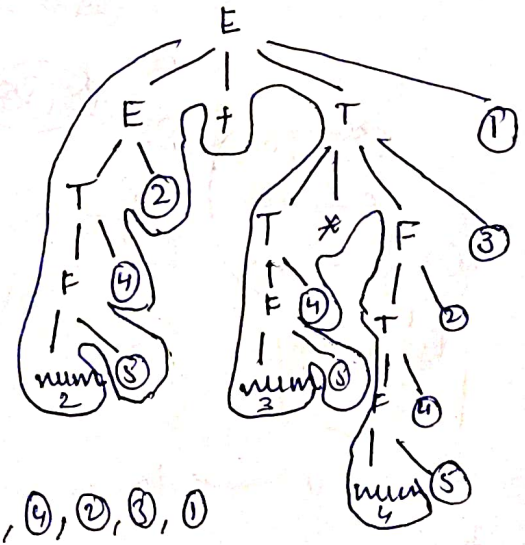
[Lvalue = Lexim value]

$E \to E+T$ { printf("+"); } ①
/ T { } ②

$E \to T*F$ { printf("*"); } ③
/ F { } ④

$F \to num$ { printf(num.lval); } ⑤

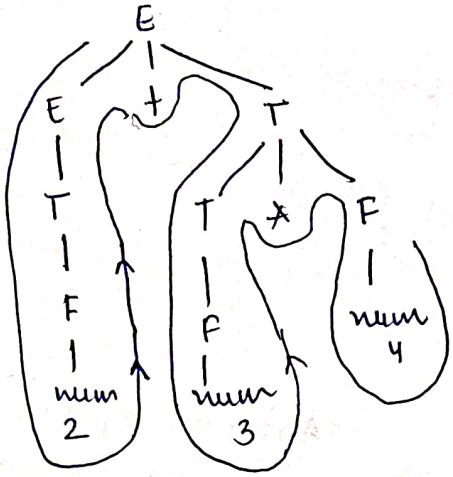
2+3*4



⑤, ④, ②, ⑤, ④, ②, ③, ①

postfix 2 3 4 * +

Bottom up parser for same grammar

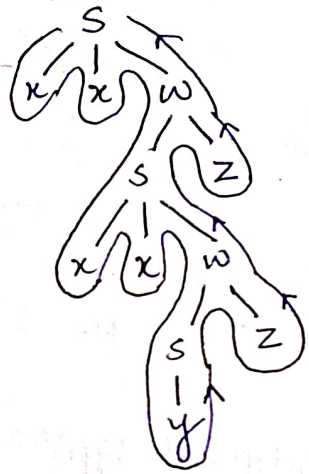


2 3 4 * +

S → xxw { printf(1); } ①
/y { printf(2); } ②
w → sz { printf(3); } ③

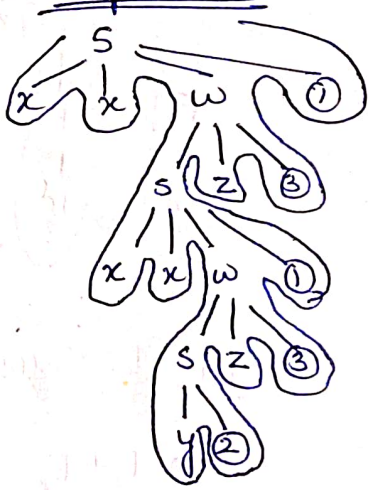
String to be generated = xxxxyzz

Bottom up



o/p: 23131

Top down

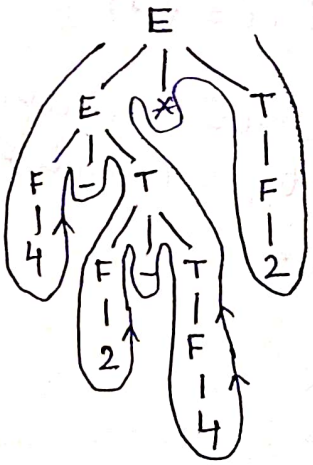


o/p → 23131

E → E*T { E.val = E.val * T.val; }
/T { E.val = T.val; }
T → F-T { T.val = F.val - T.val }
/F { T.val = F.val; }
F → 2 { F.val = 2 }
/4 { F.val = 4 }

'-' is right associative.

String to be generated = 4-2-4*2



No. of reductions = No. of non leaves = 10

Inferences —

* has lower precedence than -

- is right associative

∴ Ans

= (4 - (2-4)) * 2

= (4 - (-2)) * 2

= 6 * 2 = 12

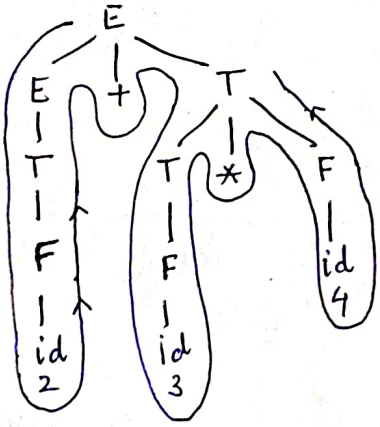
SDT to build syntax tree

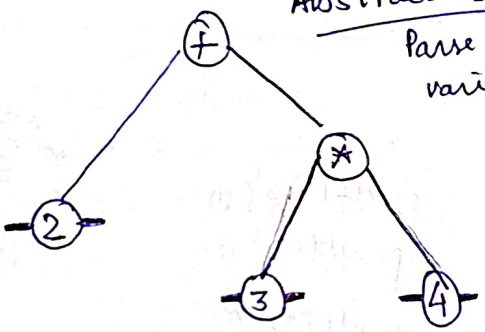
$E \to E_1 + T$ { E.nptr = mknode (E_1.nptr, '+', T.nptr) }
/ T { E.nptr = T.nptr }

$T \to T_1 * F$ { T.nptr = mknode (T_1.nptr, '*', F.nptr) }
/ F { T.nptr = F.nptr }

F → id { F.nptr = mknode (null, id.name, null); }

2+3*4





Abstract syntax tree
Parse tree in which variables are not present.

SDT for Type Check [Data type of expression]

$E \to E_1 + E_2$ { if(((E_1.type == E_2.type) && (E_1.type = int)) then E.type = int else error }
/ $E_1 == E_2$ { if(((E_1.type == E_2.type) && (E_1.type = int/boolean)) then E.type = boolean else error }

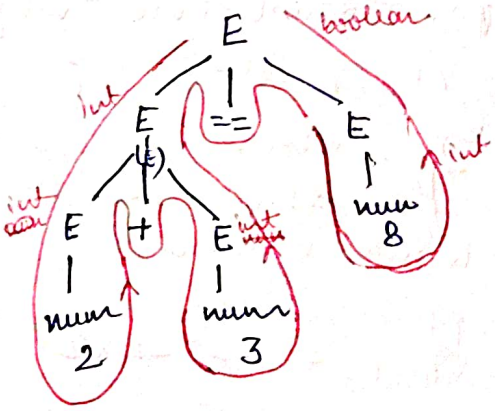
/ (E_1) { E.type = E_1.type }

/ num { E.type = int }

/ true { E.type = boolean }
/ false { E.type = boolean }

Expression → (2+3)==8

Parse tree —



SDTs for binary numbers —

	Count all 1's	Count all 0's	No. of bits
N → L	{N.count = L.count}	⇒	⇒
L → L1 B	{L.count = L1.count + B.count}	⇒	⇒
/ B	{L.count = B.count}	⇒	⇒
B → 0	{B.count = 0}	{B.count = 1}	{B.count = 1}
/ 1	{B.count = 1}	{B.count = 0}	{B.count = 0}

Decimal value

N → L	{N.dval = L.dval}
L → L1 B	{L.dval = L1.dval * 2 + B.dval}
/ B	{L.dval = B.dval}
B → 0	{B.dval = 0}
/ 1	{B.dval = 1}

S-Attributed and L-Attributed definitions

11.01 = 3.25

3 ↵ ↳ $\frac{1}{2^2} = \frac{1}{4} = 0.25$

∴ Value of fractional part = $\frac{\text{Decimal value of fraction}}{2^{\text{No. of bits in fraction}}}$

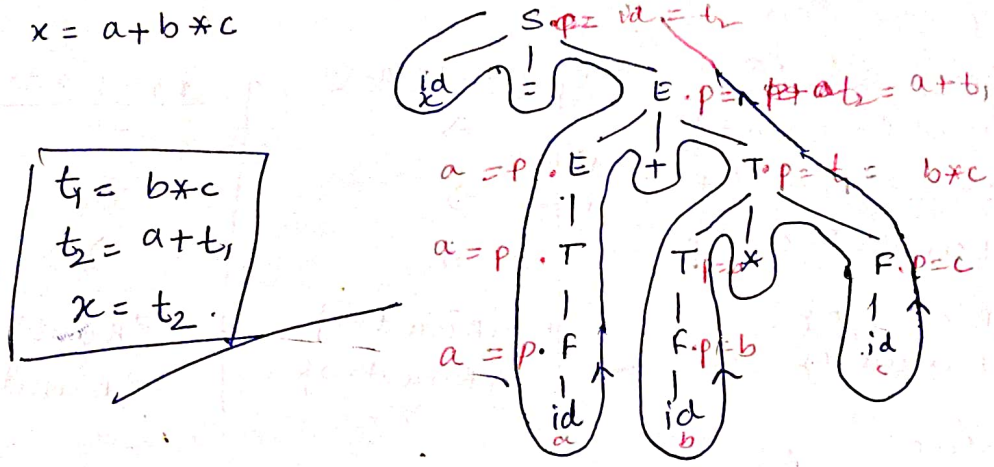
SDT to find decimal value of a fractional binary number

N → L1 . L2	{N.dval = L.dval + $\frac{L_2.dval}{2^{L_2.count}}$}
L → L B	{L.count = L1.count + B.count; L.dval = L1.dval * 2 + B.dval}
/ B	{L.count = B.count; L.dval = B.dval}
B → 0	{B.count = 1; B.dval = 0}
/ 1	{B.count = 1; B.dval = 1}

SDT to generate 3 address code

S → id = E	{gen(id.name = E.place)}
E → E1 + T	{E.place = newtemp(); gen(E.place = E1.place + T.place)}
/ T	{E.place = T.place}
T → T * F	{T.place = newtemp(); gen(T.place = T1.place * F.place)}
/ F	{T.place = F.place}
F → id	{F.place = id.name}

$x = a + b * c$



Attributes in SDT are of 2 types

- **Synthesized attributes**
  The value of attribute is derived from its children.
- **Inherited attributes**
  The value of the attribute is derived from other parent or siblings.

S attributed SDT

- uses only synthesized attributes.
- Semantic actions are placed at the right end of the production.
  $A \rightarrow BCC\{\ldots\}$
- Attributes are evaluated using bottom up parsing.

L attributed SDT

- uses both synthesized & inherited attribute.
  Each inherited attribute is restricted to inherit from parent or left sibling only.
- Semantic actions can be placed anywhere in the production
  $A \rightarrow \{\ldots\}BC \mid D\{\ldots\}E \mid FG\{\ldots\}$
- Attributes are evaluated using top-to-down, left to right parsing

Check whether the SDTs are left ⊗ S-attributed or L-attributed

① $A \to LM \quad \{ L.i = f(A.i);\ M.i = f(L.S);\ A.S = f(M.S); \}$ L-attributed
(inherited)

$A \to QR \quad \{ R.i = f(A.i);\ Q.i = f(R.i);\ A.S = f(Q.S); \}$
↳ not L-attributed

a) S-attributed b) L-attributed c) Both d) None

② $A \to BC \quad \{ B.S = A.S; \}$
(inherited ∴ not S)

a) S-attri b) L-attri c) Both d) None

SDT to store type information in symbol table.

$D \to TL \quad \{ L.in = T.type \}$
$T \to int \quad \{ T.type = int; \}$
$\quad | char \quad \{ T.type = char; \}$
$L \to L_1 id \quad \{ L_1.in = L.in,\ addtype(id.name, L_1.in); \}$
$\quad | id \quad \{ addtype(id.name, L.in); \}$

L-inherited SDT

S-attributed SDT

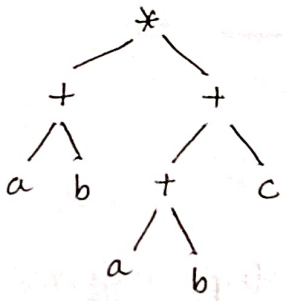
$D \to D_1, id \quad \{ add\_type(id.name, D_1.type); \}$
$\quad | T\ id \quad \{ add\_type(id.name, T.type),\ D.type = T.type; \}$
$T \to int \quad \{ T.type = int \}$
$\quad | char \quad \{ T.type = char \}$

Intermediate Code Generation

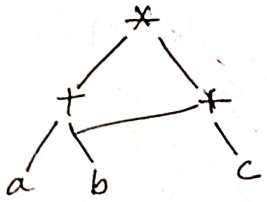
Ex→ (a+b)* (a+b+c)

Intermediate code

- Linear form
  - Postfix: ab+ab+c+*
  - Three Address Code:
    $t_1 = a+b$
    $t_2 = a+b$
    $t_3 = t_1+c$
    $t_4 = t_1 * t_3$
- Tree form
  - Syntax Tree



  - DAG (Directed acyclic graph)



Types of 3 address code

1. $x = y \langle op \rangle z$ [Binary operation]
2. $x = \langle op \rangle y$ [Unary operation]
3. $x = y$ [Assignment]
4. if $x \langle relop \rangle y$ goto L [conditional]
5. goto L [Jump]
6. $A[i] = x$
   $y = A[i]$ [Arrays]
7. $x = *p$
   $y = \&x$ [Pointers]

Various Representations of 3 address code

- Quadruple
- Triple
- Indirect Triple.

Example→ (a+b)*(c+d)+(a+b+c)

1) $t_1 = a+b$
2) $t_2 = c+d$
3) $t_3 = t_1 * t_2$
4) $t_4 = a+b$
5) $t_5 = t_4 + c$
6) $t_6 = t_3 + t_5$.

Quadruple

	opr	opt1	opt2	Result
1)	+	a	b	t_1
2)	+	c	d	t_2
3)	*	t_1	t_2	t_3
4)	+	a	b	t_4
5)	+	t_4	c	t_5
6)	+	t_3	t_5	t_6

Adv :- Statements can be moved around.
Dis :- More space waited.

Triple

	opr	opt1	opt2
1)	+	a	b
2)	+	c	d
3)	*	(1)	(2)
4)	+	a	b
5)	+	(4)	c
6)	+	(3)	(5)

Adv :- Space not wasted
Dis :- Statments can't be moved around.

Indirect Triple

i) (1)
ii) (2)
iii) (3)
iv) (4)
v) (5)
vi) (6)

Adv :- Statements can be moved.
Dis :- Two access of memory.

Back Patching

```
if (a<b) then t = 1
else t=0
```

(i): if a < b goto i+3
(i+1): t = 0
(i+2): goto i+4
(i+3): t = 1
i+4

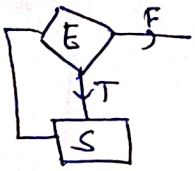
Leaving the table as empty and filling them back is called backpatching.

```
while (a<b) do
    x = y+z
```

L: if a<b goto L1
   goto L2
L1: t = y+z
    x = t
    goto L
L2:

While loop in 3 address code

```
while E do S
```

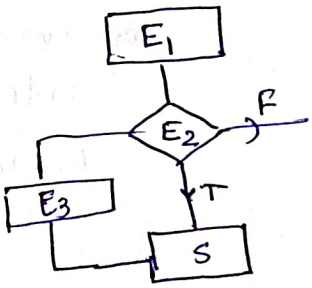


L: if E == false goto L1
   S
   Goto L
L1:

L: if (E) goto L1
   goto L2
L1: S
    goto L
L2:

For loop in 3 address code

for (E_1 ; E_2 ; E_3)
    S




```
for (i=0; i<10 ; i++)
    a = b+c
```

i = 0
L1: if i<10 goto L2
    goto L3
L2: t_1 = b+c
    a = t_1
    t_2 = i+1
    i = t_2
    goto L1
L3:

switch using 3 address code.

```
switch(i+j) {
    case 1: a = b+c;
            break;
    case 2: P = Q+R;
            break;
    default: x = y+z
             break;
}
```

```
t = i+j
goto L4

L1: t1 = b+c
    a = t1
    goto last

L2: t2 = Q+R
    P = t2
    goto last

L3: t3 = y+z
    x = t3
    goto ___

L4: if t==1
        goto L1
    if t==2
        goto L2
    goto L3

last:
```

2 dimensional array to 3 address code

$x = A[y, z]$

Row major ordering

A: 10 × 20 (rows × column)

$t_1 = y * 20$

$t_2 = t_1 + z$

$t_3 = t_2 * 4$ → size of each element

$t_4 = $ base address of A

$x = t_4[t_3]$

Gate 2007

In a simplified computer, the instructions are:

OP Rj Ri — Performs Rj OP Ri & stores result in Ri

OP m Ri — Performs val OP Ri & stores in Ri
val = value in memory location m.

MOV m Ri — Moves content of the memory location m to register Ri

MOV Ri m — Moves content of register Ri to memory location m.

The computer has only 2 registers and OP is either ADD or SUB

Consider the basic block

$t_1 = a+b$
$t_2 = c+d$
$t_3 = e-t_2$
$t_4 = t_1-t_3$

Assume that all operands are initially in memory. The final value of computation should be in memory. What is the minimum no. of MOV instructions for this basic block

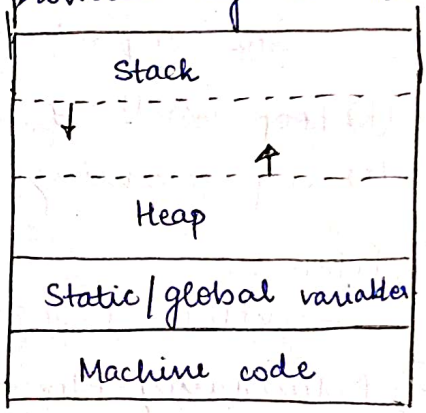
```
MOV a R1
ADD b R1
MOV c R2
ADD d R2
```

```
SUB e , R2
SUB R1 R2
MOV R2 , m
```

⇒ 3 MOV operations

Runtime environments

Runtime environment refers to the support provided by OS to run a program.



→ Heap grows upwards
→ stack grows downward.

STORAGE ALLOCATION STRATEGIES

1) STATIC
- Allocation is done at compile time.
- Bindings do not change at runtime.
- One activation record per procedure.

DIS – → Recursion not supported.
→ size of data objects must be known beforehand.
→ Data structures cannot be created dynamically.

2) STACK

Whenever a new activation begins, activation record is pushed onto the stack and whenever activation ends, activation record pops off.

Local variables are bound to fresh storage.

DIS → Local variables cannot be retained once activation ends

3) HEAPS

Allocation and deallocation can be in any order.

DIS → Heap management is overhead.

SUMMARY

Activations can have –
→ Permanent lifetime in case of static allocation.
→ Nested lifetime in case of stack allocation
→ Arbitrary lifetime incase of heap allocation.

CODE OPTIMISATION

reducing the number of lines in a program (3 address code)

Optimisation

- Machine independent
  1) Loop optimisations
     (a) Code motion (or) frequency reduction
     (b) Loop unrolling
     (c) Loop jamming
  2) Folding
     - Constant Propagation
  3) Redundancy Elimination
  4) Strength Reduction.
- Machine Dependent
  1) Register allocation
  2) Use of addressing modes
  3) Peephole optimisation
     (a) Redundant load/store
     (b) Flow of control options
     (c) Strength Reduction.
     (d) use of machine idioms.

Loop optimisations

- To apply optimisations, we must first detect loops.
- For detecting loops, we can use Control Flow Analysis (CFA) using Program Flow Graph (PFG)
- To find PFG, we need basic blocks.

> Basic block is a sequence of 3 address statements where control enters in the beginning & leaves at the end without any jumps or halts.

Finding basic blocks

In order to find basic blocks, we need to find leaders in program. Then a basic block will start from one leader to the next leader but not including next leader.

Optimisation
↓
Loops
↓
CFA (PFG)
↓
Basic Block
↓
Leaders.

Identifying leaders in basic block

- First statement is a leader.
- Statement that is target of conditional or unconditional statements is a leader.

  Ex→ if ( ) goto 300 , goto 400
  (300 → leader; 400 → leader.)

- Statement that follows immediately a conditional or unconditional goto statement is a leader.

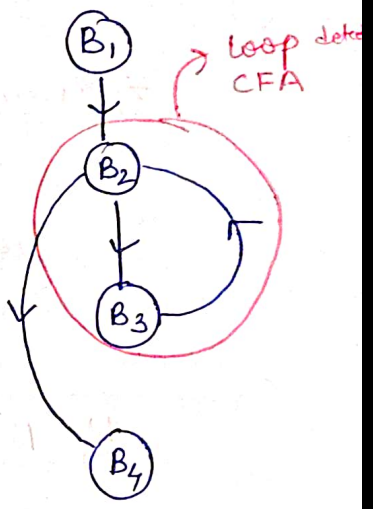
Example

```
fact(x) {
    int f=1;
    for(i=2;i<=x;i++){
        f=f*i;
    }
    return f;
}
```

3 address code

```
* (1) f=1                    B1
  (2) i=2
* (3) if i>x goto 9          B2
* (4) t1 = f*i               B3
  (5) f = t1
  (6) t2 = i+1
  (7) i = t2
  (8) goto 3
* (9) goto calling program   B4
```

PFG



Types of loop optimisations

FREQUENCY REDUCTION

Moving the code from higher frequency region to a low frequency region is called frequency reduction.

```
while (i<5000){
    A = sinx/cosx * i;
    i++;
}
```

⇓

```
t = sinx/cosx;
while (i<5000){
    A = t*i;
}
```

LOOP UNROLLING

```
while(i<10){
    x[i]=0
    i++
}
```

⇓

```
while (i<10){
    x[i]=0
    i++
    x[i]=0
    i++
}
```

LOOP JAMMING

Combining bodies of 2 loops.

```
for(i=0; i<10; i++)
    for(j=0; j<10; j++)
        x[i,j]=0;
for(i=0; i<10; i++)
    x[i,i] = 0
```

⇓

```
for(i=0; i<10; i++){
    for(j=0; j<10; j++){
        x[i,j]=0
    }
    x[i,i]=0;
}
```

other Machine Independent Optimisations

1) Folding - Replacing an expression that can be computed at compile time by its values.

Ex→ 2+3+C+B ⇒ 5+C+B

2) Redundancy Elimination (DAG) → expression that is already evaluated is used again and again.

A = B+C
D = 2+B+3+C

D = 2+3+A

3) Strength Reduction - Replacing a costly operation by a cheaper one.

<Multiplication & division are costly, bit manipulation is cheaper>

B = A*2
B = A<<1

4) Algebraic Simplification

A = A+0
B = B*1
} eliminate these trivial statements.

Machine Dependent Optimisations

- Register allocation (max. utilization of registers)
  - → local allocation
  - → global allocation
- use addressing modes
- Peephole optimisation
  - (a) Redundant load and store

x = y+z

```
MOV y R0
ADD z R0
MOV R0 x
```

a = b+c
d = a+e

```
MOV b R0
ADD c R0
MOV R0 a
MOV a R0
ADD e R0
```

→ Redundan

(b) Flow of control optimisation

Avoid jumps on jumps

```
L1: jump L2      L4
    :
L2: jump L3
    :
L3: jump L4
```

Eliminate dead code.

```
#define x 0
if(x)
{
    //dead code.
}
```

(c) use of machine idioms

i = i+1

```
MOV  R0 i
ADD  R0,1
MOV  i,R0
```

inc i → machine code for increment.

Data Flow Analysis

Structured graph based analysis of the program

1) Constant Propagation

```
int x=14;
int y = 7 - x/2;
return y * (28/x + 2)
```

Propagate x →

```
int x=14
int y = 7-14/2;
return y * (28/14 +2);
```

↓ Propagate y

```
int x=14
int y=0
return 0;
```

← dead code elimination

```
return 0;
```

Common Subexpression Elimination (CSE)

Redundancy elimination is a form of CSE.

Ex → a = b*c + g —①
d = b*c + e —②

b*c is the common subexpression.

temp = b*c;
a = temp + g
d = temp + e

} space required increased due to the additional temporary variables.

Trade off —

if cost(b*c) > cost(temp creation & fetching), then, it is useful.

- Local Common Subexpression Elimination → within single block.
- Global Common Subexpression Elimination → Entire function / procedure.

Live Variable Analysis

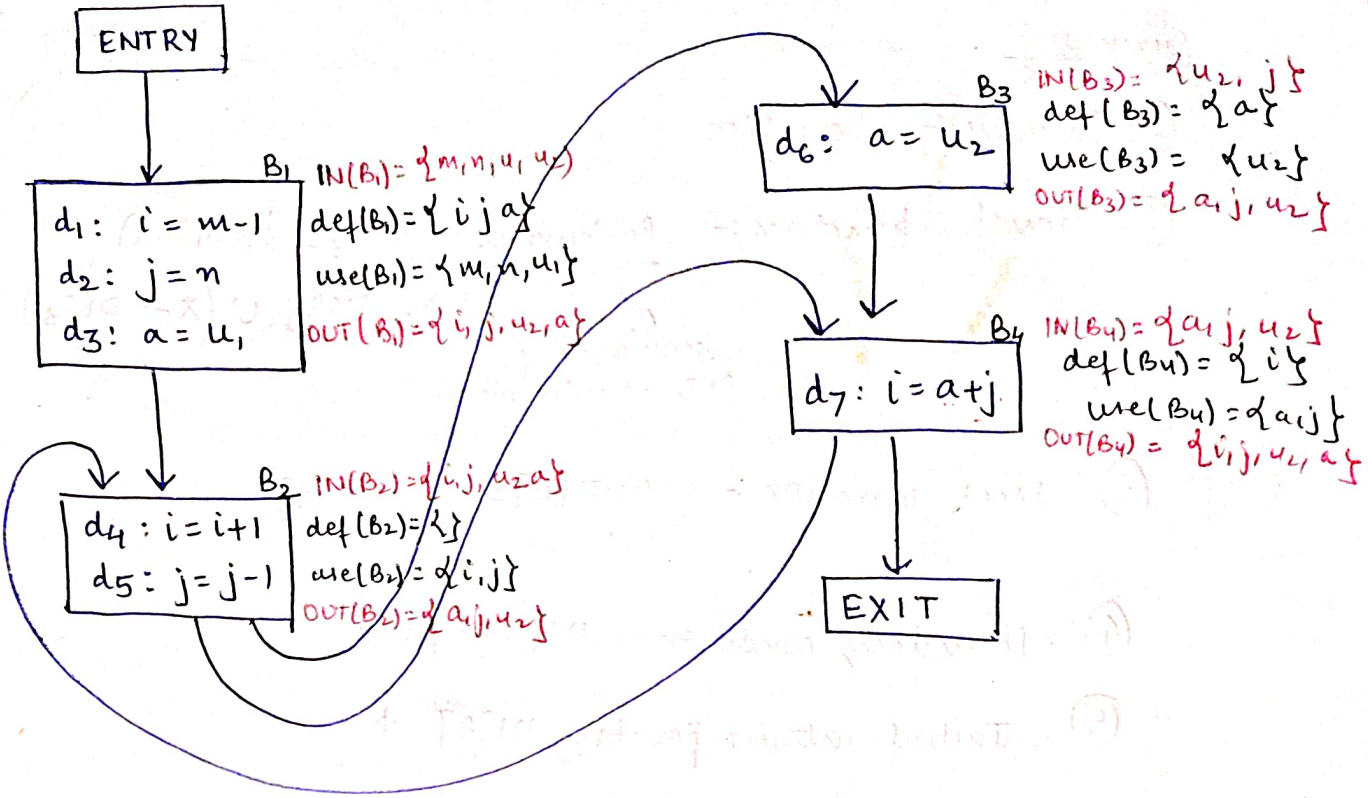
Variable v is live at point P if value of v is used in some path in flow graph starting at P.

else it is dead.

Application of Live Variable Analysis —

Register allocation
Live variables are stored in registers (instead of dead variables)

set of variables ←
- DEF(B) = all the variables defined in B before being used
- USE(B) = variables that are used in the block before defining them



OUT[B] → Set of variables alive after block B = $\bigcup_{S:\text{Successor}}$ IN[S]

IN[B] → Set of variables alive before block B = USE[B] ∪ (OUT[B] - DEF[B])

Algorithm: Live variable Analysis

INPUT: Flow graph ~~with~~ with DEF and USE calculated for each block.

OUTPUT: IN[B] and OUT[B], the set of variables live at entry and exit of each block.

```
IN[EXIT] = ϕ
for(each basic block B other than EXIT)  IN[B] = ϕ
while(changes to any IN occur){
    for(each basic block B other than exit){
        OUT[B] = ∪ (S = successor of B) . IN[S]
        IN[B] = USE[B] ∪ (OUT[B] - DEF[B]);
    }
}
```

Summary

① Domain :- Variables

② Transfer function :- Backward $IN[B] = f(OUT[B])$

$f_B(x) = USE_B \cup (X - DEF_B)$

because using OUT, we calculate IN

③ Meet operation :- $OUT[B] = \bigcup_S IN[S]$

④ Boundary condition :- $IN[EXIT] = \phi$

⑤ Initial interior points :- $IN[B] = \phi$

Static Single Assignment (SSA)

very similar to 3 address code but different

optimization becomes easy.

> Basic concept → each variable can have atmost one assignment/ definition.

```
i ← 0
:
:
:
if(i<0)
:
:
```

it is sure i has not been modified since.

optimization can be done always here

In 3 address code i might have been redefined ∴ optimization cant be done

Optimizations must be safe
execution of transformed code must yeild same results as the original code ~~safe~~ for all possible executions.

Optimization Techniques
- ↦ common subexpression elimination
- ↦ Dead code elimination
- ↦ Copy propagation
- ↦ Constant propagation.

Other optimizations
- ↦ Arithematic simplification
  $x = u*a \equiv x = a<<2$
- ↦ Constant folding
  $x = u*5 \rightarrow x = 20$

Live liness analysis is used for dead code elimination

Available expression analysis is used for common subexpression elimination