PPL

There are many Languages (over 8000), instead of learning each, it is better to learn basic principles of any Language.

Why study PPL?
- increased ability to learn new Language
- To allow better choice of programming Languages

Tiobe statistics

1) Python
2) C ← old but gold
3) C++
4) Java
5) C#
6) JavaScript

Syntax - structure of Language, word order, sentence composition.

Semantics - meaning of words.

Q. If a code uses same algorithm on same device, but same ~~different~~ programming Language. Can performance be different?

ans. Yes, depends on data structures used and internal implementation of data structures

Models of programming & Language

1) Imperative
2) Functional
3) Logic

1) Imperative.

Imperative Languages mimic Von Neumann Architecture

variables ↔ memory cells.

Assignment statements ↔ data piping between memory and CPU.

Operations and expressions ↔ CPU executions

Explicit control of execution flow ↔ program counter.

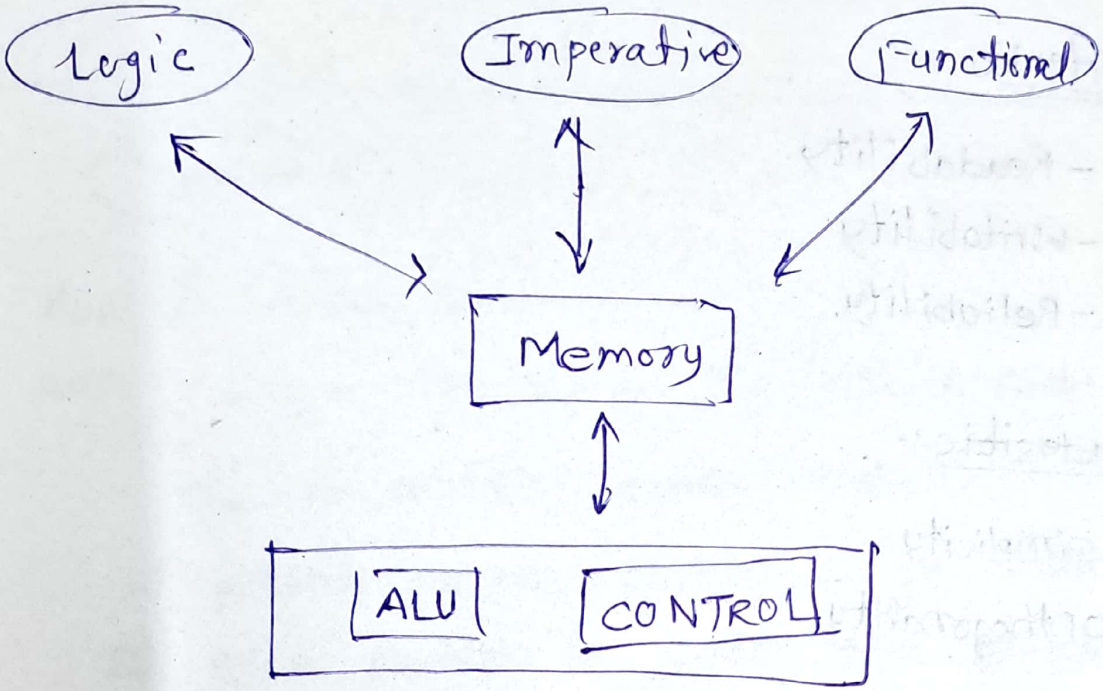
2) Functional

programming is like solving mathematical functions

A program, its subprograms are just implementation of mathematical functions

3) Logic.

- program expressed as rule in formal logic
- Execution by rule resolution
- has facts, rules & goals

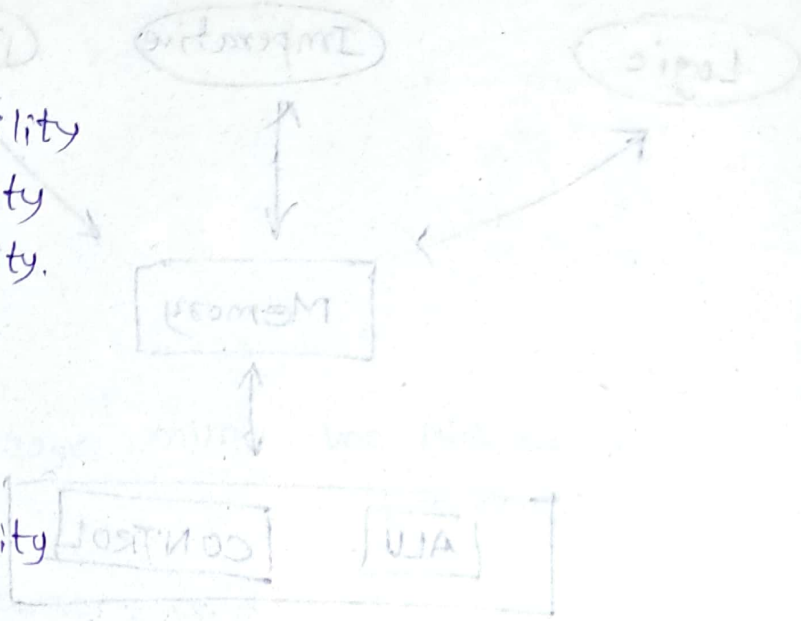


Von Neumann Architecture.

Evaluation Criteria & Characteristic

- Criteria
  - Readability
  - Writability
  - Reliability.

- Characteristic :-
  - simplicity
  - orthogonality



---

download putty.

Q] int i = 22;
float p = 22.3;
printf("%d", i+p);

ans:- garbage value.

i+p ⇒ 'i' will be promoted to float type.

but "%d" leads to demotion, thus Loss of data, also we get garbage value

To prevent this, we need to do explicit conversion.

but printf("%f", i+p);
will not give any error

Indentation

How to debug code?

ans:- $\log_2 n$ complexity, n = lines in code.

Go to middle, cout<< "Hello";

if printed correctly, problem is in code below "Hello".

else problem in code above "Hello"

Repeat this for problematic part also

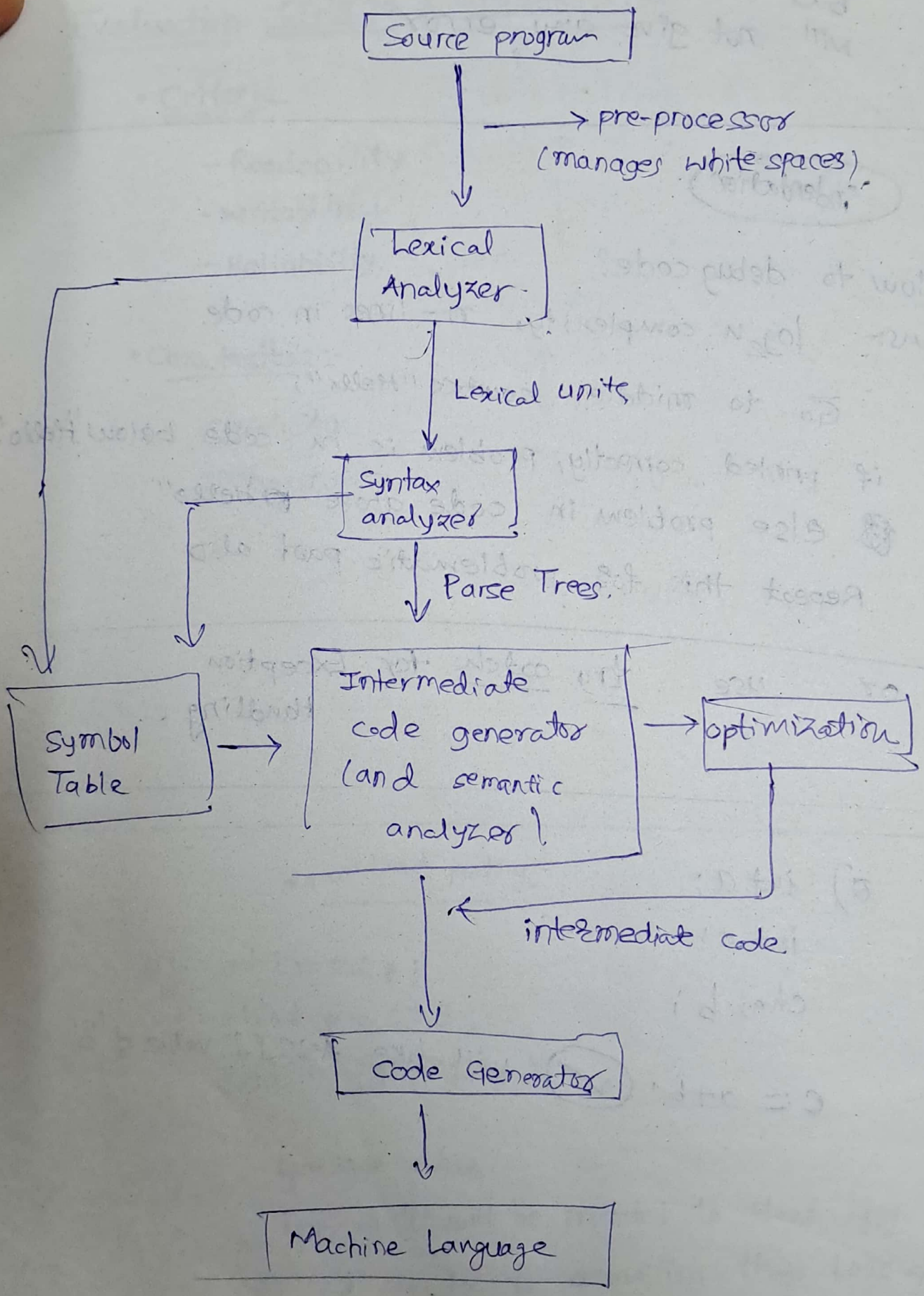
---

or use try catch for Exception Handling.

---

8) int a;
int c;
char b;

c = a+b; ✓ will take ASCII value of 'b'.



compiler does not stop at 1 error

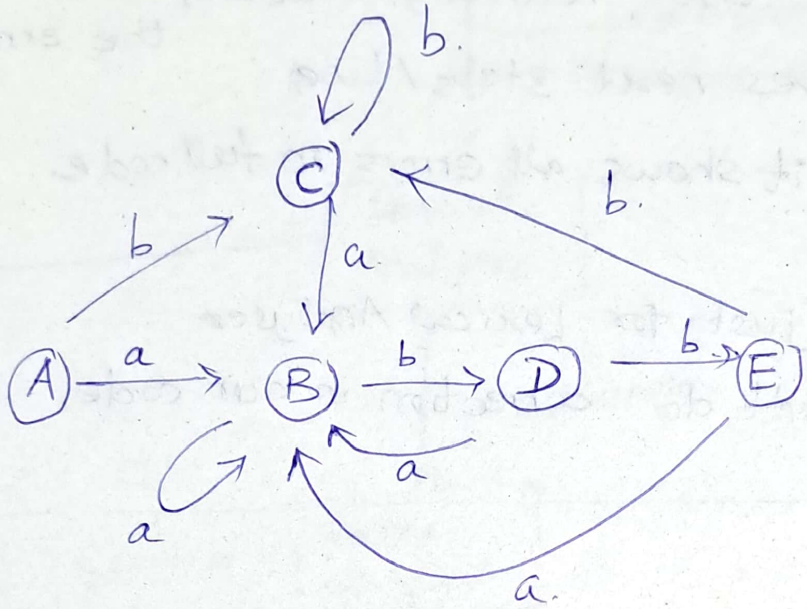
It does error recovery. (Just skip, ignore & note the error)
and moves next steps / Line.

Finally it shows all errors in full code.

Recovery is just for Lexical Analyser

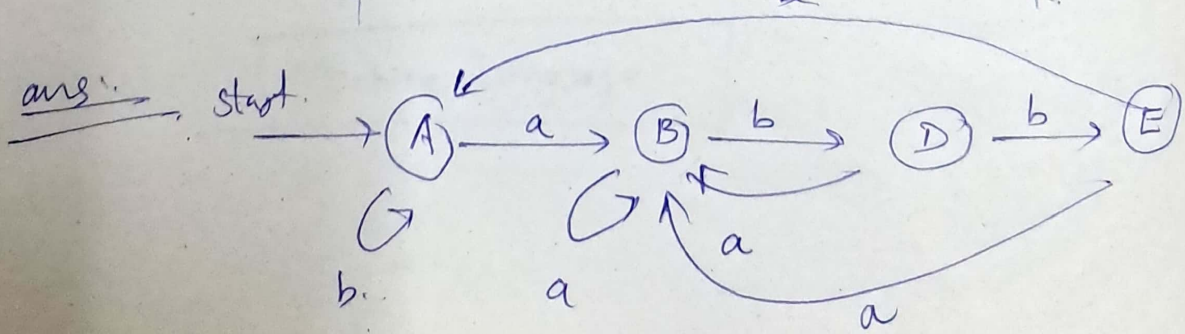
It doesn't do correction in our code.

•) Minimize the following DFA:-



ans:-

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



$\Pi_0 = \{A\}, \{B, C, D, E\}$

$\Pi_1 = \{A\}, \{B, C, D, E\}$

$\Pi_0 = \{A, B, C, D\} \{E\}$

$\Pi_1 = \{A, B, C\} \{D\} \{E\}$

$\Pi_2 = \{A\} \{B, C\} \{D\} \{E\}$

.....

error

Lex] c language Lexical Analyser
Flex
fast

JLex] Java lexical Analyser

Q] Try to find 5 errors that can arise in Lex phase?

ans:-

Find the FIRST and FOLLOW

$S \rightarrow ABCD \mid \epsilon$

$A \rightarrow a \mid \epsilon$

$B \rightarrow bA$

$C \rightarrow a \mid \epsilon$

$D \rightarrow d$

- it is for both terminals & non-terminals

ex:- Q] $E \rightarrow E+T \;/\; T$

$T \rightarrow F$

$F \rightarrow id$

- For finding First, just see Left Hand side

$E \rightarrow E+T$ recursion — drop.

$E \rightarrow T$ — non terminal - drop.

$T \rightarrow F$ — non terminal - drop.

$F \rightarrow \boxed{id}$ — terminal.

Hence First ('E') = {id.}

Similarly. First ('+') = {+}

First ('id') = {id}

First ('T') = {id}

First ('F') = {id}

Note:- For any terminal,

First ('+') = {+} ;

First ('terminal') = {terminal}

Q] Find First for following

$E \rightarrow E+T \;/\; E-T \;/\; T$

$T \rightarrow T*F \;/\; T/F \;/\; F$

$F \rightarrow id$

ans:-

$E \rightarrow E+T$ — recursion_drop

$E \rightarrow E-T$ — recursion-drop

$E \rightarrow T$ nonterminal

$T \rightarrow F$ nonterminal

$F \rightarrow id.$ terminal ✓

∴ First ('E') = {id}

First ('T') = ?

$T \longrightarrow T*F$ — recursion_drop

$T \longrightarrow T/F$ — recursion-drop

$T \longrightarrow F$

$F \longrightarrow id.$ terminal ✓

First ('T') = {id}

similarly First (T') = {id}.

for terminals,
- First (+) = {+}
- First (-) = {-}
- First (/) = {/}
- First (*) = {*}

Q] $S \longrightarrow aA \mid aB$

$S \longrightarrow aA$ → nonterminal.
$S \longrightarrow aB$ → nonterminal.

$S \rightarrow aA$
- a → terminal
- A → nonterminal.

First (S) = {a, a}. Ambiguous grammar. will ignore this.

Note:- Stop whenever you get terminal.

$S \rightarrow aA$
↳ stop. this must be first.

Q] $E \rightarrow E+T \;/\; T \;/\; \epsilon$

$T \rightarrow T*F \;/\; F \;/\; \epsilon$

$F \rightarrow id$

ans:-

$E \rightarrow E+T$ — recursion-drop

$E \rightarrow T$ — nonterminal.

$T \rightarrow \boxed{\epsilon}$ ✓ $\quad T \rightarrow F$

$F \rightarrow \boxed{id}$ ✓

$E \rightarrow \boxed{\epsilon}$ ✓

First ('E') = { E, ~~ϵ~~, id }

First ('T') = { id, ϵ }

First ('F') = {id}

~~First ('*') = {*}~~

First ('+') = {+}

First ('*') = {*}

Q] $S \rightarrow ABCD \mid \epsilon$

$A \rightarrow a \mid \epsilon$

$B \rightarrow bA \mid aA$

$C \rightarrow a \mid \epsilon$

$D \rightarrow d$

ans :-

$S \rightarrow ABCD$

$S \rightarrow \epsilon$

$A \rightarrow a \mid \epsilon$

$B \rightarrow bA \mid aA$

$C \rightarrow a \mid \epsilon$

$D \rightarrow d$

First ('S') = {ϵ, a, b}

First ('A') = {a, ϵ}

First ('B') = {b, a}

First ('C') = {a, ϵ}

First ('D') = {d}

explanation

$S \rightarrow ABCD, S \rightarrow \epsilon$

$A \rightarrow a$

$S \rightarrow aBCD$

$A \rightarrow \epsilon$

$S \rightarrow BCD$

$b \rightarrow bA \mid aA$

$S \rightarrow bACD$

$S \rightarrow aACD$

BCD

bACD

aACD

$S \rightarrow \{a, b, \epsilon\}$

FOLLOW

- Look at Right Hand side.
- 1 default symbol (augmented symbol) to add (ie $).
- because in PDA, bottom of stack has $
- Look at terminals after symbol for which you are calculating follow.
- stop after terminal reached.

Q] $E \longrightarrow E+T \;/\; E-T \;/\; T \;/\; \epsilon$

$T \longrightarrow T*F \;/\; T-F \;/\; F \;/\; \epsilon$

$F \longrightarrow id.$

ans:-

Follow (E) =

{ $, + , - }

default.

T occurs at 5 positions

Follow (T) = { * , - }

Note:- Q] Ea , Ta

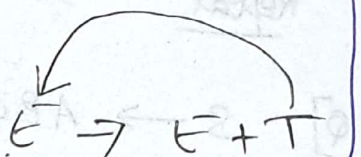
Follow(E) = {a}

Follow(T) = {a}

ans $E \rightarrow T$.

Rule:- $E \rightarrow E+T$

follow(E) = follow(E).



E a

E+T a

Repeat

Q] $E \rightarrow E+T \;/\; E-T \;/\; T \;/\; \varepsilon$

$T \rightarrow T*F \;/\; T/F \;/\; F \;/\; \varepsilon$

$F \rightarrow id$

ans:-

Follow (E) = { $, +, − }

Follow (T) = { $, +, −, *, / }

Follow (F) = { $, +, −, *, / }

Rule 2 $S \rightarrow A\ B\ C\ D$.

Follow (A) = first (B)
Follow (B) = first (c)
Follow (c) = first (D)

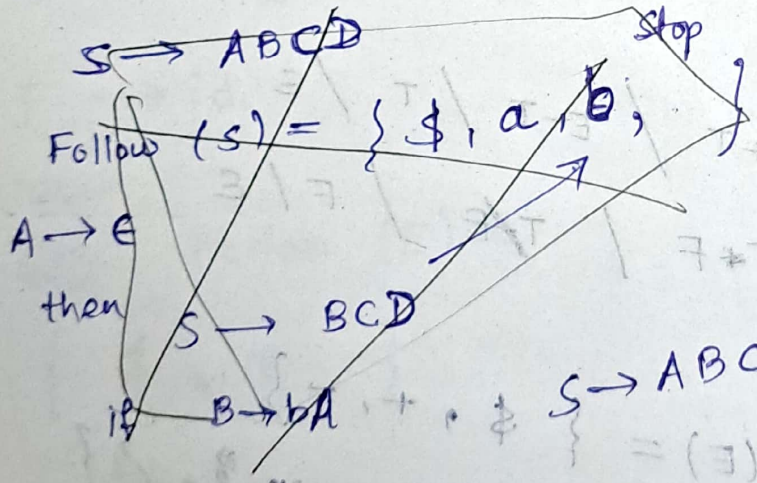
Rule 3
'ϵ' not come in follow

Repeat

Q] $S \rightarrow ABCD \mid \epsilon$
$A \rightarrow a \mid \epsilon$
$B \rightarrow bA$
$c \rightarrow a \mid \epsilon$
$D \rightarrow d$

Follow (S) = { } nothing hence $.

ans :-



Follow (A) = { first (B) } = { b, a, $, d }
ie A

$B \rightarrow bA$
A will include B elements

Follow (B) = { first c } = { a, $, d }

~~Follow (D) = Nothing~~

Follow (c) = (first D) = { d }

Follow D = follow(S) = { } nothing
= { $ }

Q] $S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \epsilon$

$D \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

ans:

Follow(S) = { $ }

Follow(F) = Follow(D) = { h }

Follow(B) = (first D) = { g, f, h }

Follow(C) = (Follow B, first C) = { g, f, h }

Follow(D) = h.

Follow(E) = (first F) = f

Rule 4:- Remove useless / unreachable symbols

Rule 5:- If Grammar has ambiguity, then backtrack & ignore it.

$S \rightarrow aB \mid aC$ ⇒ will be avoided.

$C \rightarrow e$

$B \rightarrow d$

Website:- ANSI C grammar

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

additive_expression

2024-even-venkat-iiita

Methods of parsing

- Top-down
- Bottom-up

(•) If we go left → right execution.
will need to remove Left Recursion.

(•) Elimination of Left-Recursion :-

$A \longrightarrow A\alpha$

ans:- $A \longrightarrow \alpha A \mid \epsilon$

$A \longrightarrow A\alpha \mid \beta$

ans:- $A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' \mid \epsilon$

Q

$E \longrightarrow E + T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow (E) \mid id$

ans:-

$E \longrightarrow E + T$

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow (E) \mid id$

Q2) $E \rightarrow E+T \mid T \mid F$

$E \rightarrow TE' \mid FE'$

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

Q3) $E \rightarrow E+T \mid E+F \mid T \mid F$

$E \rightarrow TE' \mid FE'$

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

- We will be seeing predictive parsing.
- We will not consider ambiguity and non-determinism

ex:-

$A \longrightarrow a\beta \mid a\gamma$ ✗ Left factoring present

$A \longrightarrow aA'$

$A' \longrightarrow \beta \mid \gamma$ ✓ Left Factoring Removed.

Q] Grammar.

Note:- ε, $\in$, ë are same

Will use ë in this course.

TABLE MAKING

$E \rightarrow T\ E'$

$E' \rightarrow +TE' \mid \ddot{e}$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \ddot{e}$

$F \rightarrow (E) \mid id.$

⟹ elaborate

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \ddot{e}$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow \ddot{e}$

$F \rightarrow (E)$

$F \rightarrow id.$

Symbol	First	Follow
E	(, id	$,)
E'	+, ë	$,)
T	(, id	+, $,)
T'	*, ë	+, $,)
F	(, id	*, +, $,)

Based on first, follow we will create a table that parser will follow.

E gets id, (

because of $E \to TE'$ production.

don't go in depth, ie. ~~$E \to TE'$~~, ~~$T \to F \to id$~~.

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

Note: If a cell has more than 1 entry, that means Grammar has Non-determinism.

Whenever $A \to \epsilon$

write ($A \to \epsilon$) rule in follow of A.

ex: $T' \to \epsilon$, Follow(T') = {+, $,)}

- Empty spaces are located as Error Telling us that our syntax is wrong.

Conflicts :- cell with more than 1 entry :-

- First | First.
- First | Follow

- First | first

ex :- $A \longrightarrow ad \mid a\beta$

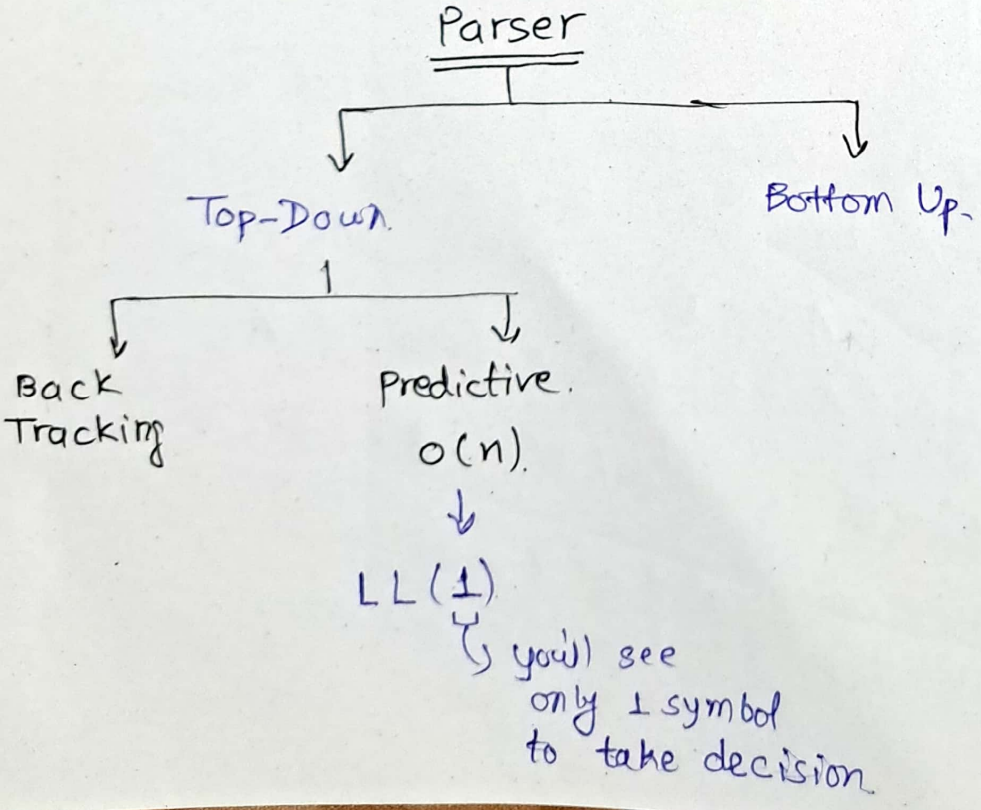
$First(A) = \{a\}$

- First | Follow

$First(T') = \{*, \epsilon\}$ $\quad Follow(T') = \{*, +, \$,)\}$

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

- Parser will only use table :
ie. only Table passed in gcc.



* Grammar is LL1, when every cell has no multiple entry.

---

* Parser uses PDA (push-down-automata).
  uses stack

Question :- id + id * id $

- when input == Top of stack ⇒ (pop)
- '$' is bottom of stack.

if E → AB
put in stack as $E
↳ $BA

Load in stack in Reverse Format.

Stack	Input	Output
$E ↑	id + id * id $ ↑	E → TE'
$E'T ↑	id + id * id $ ↑	T → FT'
$E'T'F ↑	id + id * id $ ↑	F → id
$E'T'id ↑	id + id * id $ ↑	matching, pop stack move input pointers.
$E'T' ↑	+ id * id $ ↑	T' → ∈
$E' ↑	+ id * id $ ↑	E' → +TE'
$E'T+ ↑	+ id * id $ ↑	match, pop.

$ E' T	id * id $	T → FT'
$ E' T' F	id * id $	F → id
$ E' T' id	id * id $	match, pop.
$ E' T'	* id $	T' → *FT'
$ E' T' F *	* id $	match, pop.
$ E' T' F	id $	F → id
$ E' T' id	id $	match, pop.
$ E' T'	$	T' → ∈ .
$ E'	$	E' → ∈ .
$	$	Accepted.

Q. If the input was) id + id * id $

ans:- seems invalid.

) id + id * id $

error valid. (prev-question)

Recover and report at the end.

ie [E,)] ⇒ error, skip.

No compiler uses this parser.

All modern compilers use Bottom-up parser.

In c language, there are many productions with Left-Recursion, Hence using Bottom-up parser is useful,

LEX] → C, C++ Homework Do practical.
FLEX
JLEX] → java. TA will take viva.

If dot comes after a Letter
Means the production is recognized.

$E \to E+.T$
↳ E+ has been recognized.

$E \to E+T.$
↳ means the production is over & now

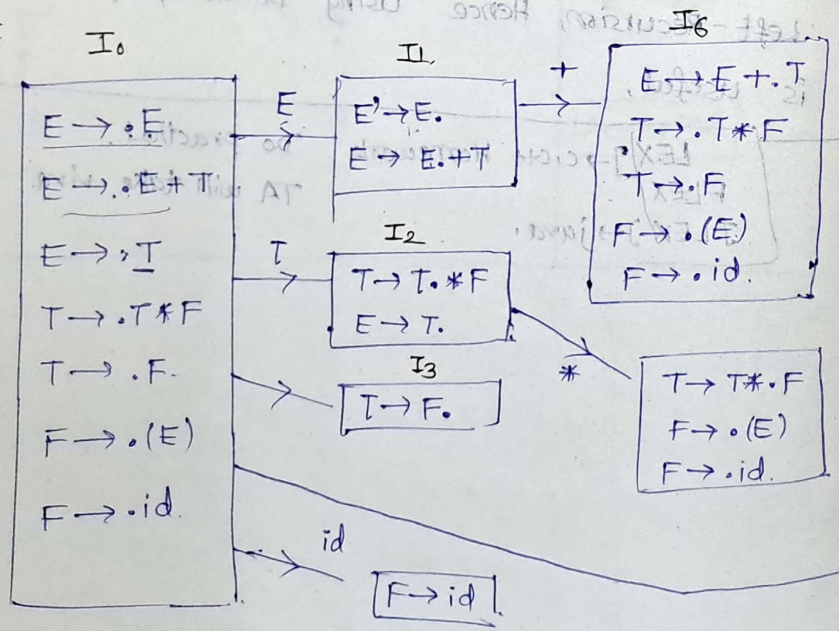
•) We can introduce new productions also. E' is new production for Below example

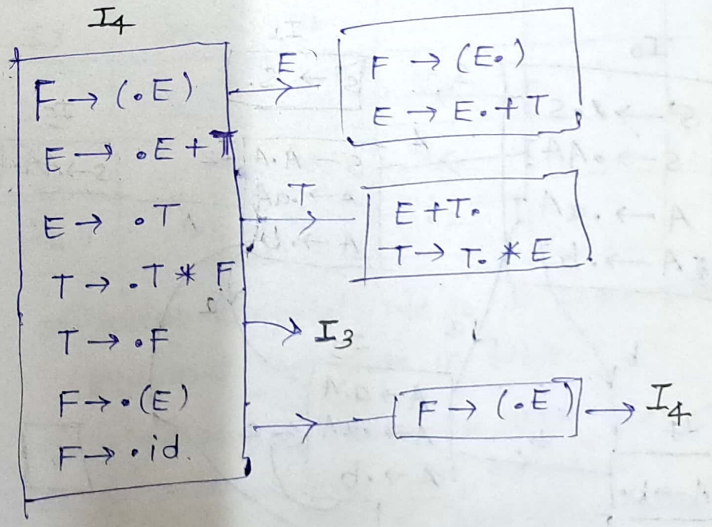
a] $E \to E+T \mid T$

$T \to T*F \mid F$

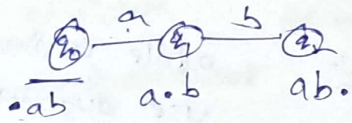
$F \to (E) \mid id$

ans:





ab



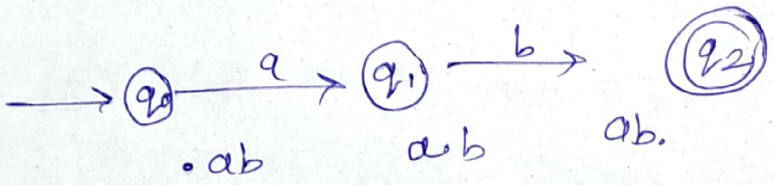
Q] $S \to AA$

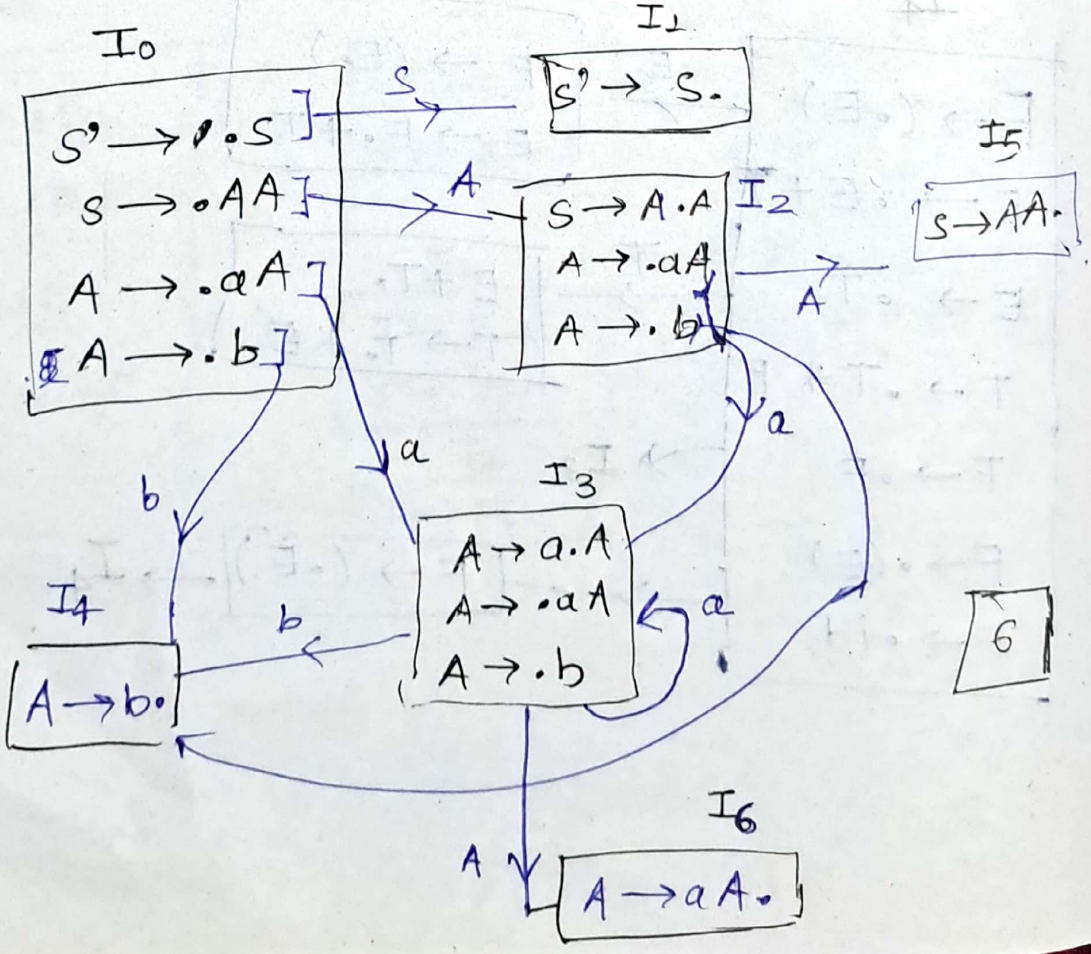
$A \to aA \mid b$

ans:- while constructing item set, use augmented symbol (S').

(•) is before non-terminal, then take closure of non-terminals until terminals occur / dot move forward.

(ab)





Shift and Reduce

Instead of transition, we say it as shift as reduce.

- shift → Moving to new state
- reduce → Taking dot forward.

- Also include $, non-terminals & terminals.

LR(0) Table

	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1	γ_0	γ_0	γ_0 Accepted		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	γ_3	γ_3	γ_3		
I_5	γ_1	γ_1	γ_1		
I_6	γ_2	γ_2	γ_2		

on reading a symbol, we go to some state.

- Fill no. of that state in table.

γ — reduce, will come for terminals, never for non-terminal.

S_i — state denotion for terminals.

1, 2, 3 ... — state denotion for non-terminals.

we will write Accept only for augmented production.

For reduce, write the reduction rules and number them according to your wish

(0) $S' \longrightarrow S$

(1) $S \longrightarrow AA$

(2) $A \longrightarrow aA$

(3) $A \longrightarrow b$.

Another Table for same parser SLR Table

	a	b	$	S	A	
I_0	S_3	S_4		1		
I_1			Accept.			
I_2	S_5	S_4			5	
I_3	S_3	S_4			6	
I_4	γ_3	γ_3	γ_3			
I_5			γ_1.			
I_6	γ_2	γ_2	γ_2			

- We will only for follows of particular symbol
- shift is same, for reduction difference comes.

- If any cell has more than 1 entry, The grammar is ambiguous.

We have already created LR(0) itemset.

Now, Lets create LR(1)

LR(1) = LR(0) + Lookahead.

LR(1): Item set

LR(0): Item set

Lookahead: ↓ means we will also add follow of respective symbols in LR(0) conversion.

Q] $E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

LR(0)

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

for above example

LR(1) = LR(0) + Lookahead

LR(1): more items than LR(0)

LR(0): 7 (as seen from LR(0) table)

Lookahead: 9

LR(1)

$E \longrightarrow .E, \$$

$E \longrightarrow .E+T, \$$

$E \longrightarrow .T, \$$

$T \longrightarrow .T * F, \$$

$T \longrightarrow .F, \$$

$F \longrightarrow .(E), \$$

$F \longrightarrow .id, \$$

We can merge reductions

	+	$
I_2		r_1
I_7	r_1	

$\Rightarrow$

	+	$
I_{27}	r_1	r_1

Hence we will reduce no. of states so that space complexity reduces. (SLR → LALR)

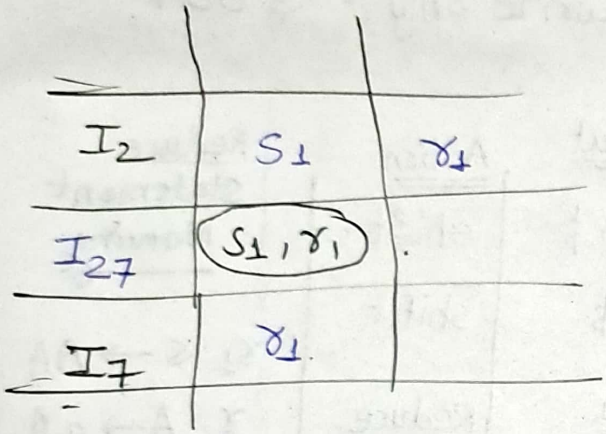
Idea:-

items → 7 LR(0) → 7 SLR → LR(0) itemset

9 CLR(1) → 7 LALR → LR(1) itemset

Note :- LALR is used by modern compiler.

Sometimes grammar accepted by CLR is rejected by LALR.



S_1, γ_1

⇓

shift & reduce conflict.

What is Allowed?

ans:- Reduce & Reduce

What creates conflict?

ans:- shift - shift

shift - reduce

In exam, you should point out such conflicts directly to show that grammar will not accept it. Instead of writing all grammar derivations.

If input is wrong, Apply Error Recovery Mechanism.

Input :- abb $

(Look in LR(0) Table.

Some books write both number & symbol :- $ 0 a 3 b 4

while some books write only :- $ 034

Stack			Input	Action	Reduce Statement Naming
$ 0	:	$ 0	abb $	shift	
$ 03	(or)	$ 0 a 3	bb $	shift	
$ 034	:	$ 0 a 3 b 4	b $	Reduce	r_1 S → AA
$ 03	(or)	$ 0 a 3 ⌴⌴	b $		r_2 A → aA
pop / delete characters double of characters in front of $	:				r_3 A → b
$ 034	:	$ 0 a 3 b 4	b $		
$ 036	:	$ 0 a 3 A 6	b $		
$ 02	:	$ 0 A 2	b $		
$ 024	:	$ 0 A 2 b 4	$		
$ 025	:	$ 0 A 2 A 5	$		
$ 01	:	$ 0 S 1	$		
$	:	$	$		

LR (1)

$E' \longrightarrow .E, \$$

$E \longrightarrow .E+T, \$$

$E \longrightarrow .T, \$$

$E \longrightarrow .E+T, +$

$E \longrightarrow .T, +$

$T \longrightarrow .T*F, \$$

$T \longrightarrow .F, \$$

$T \longrightarrow .T*F, +$

$T \longrightarrow .F, +$

$T \longrightarrow .T*F, *$

$T \longrightarrow .F, *$

$F \longrightarrow .(E), \$$

$F \longrightarrow .id, \$$

$F \longrightarrow .(E), +$

$F \longrightarrow .id, +$

$F \longrightarrow .(E), *$

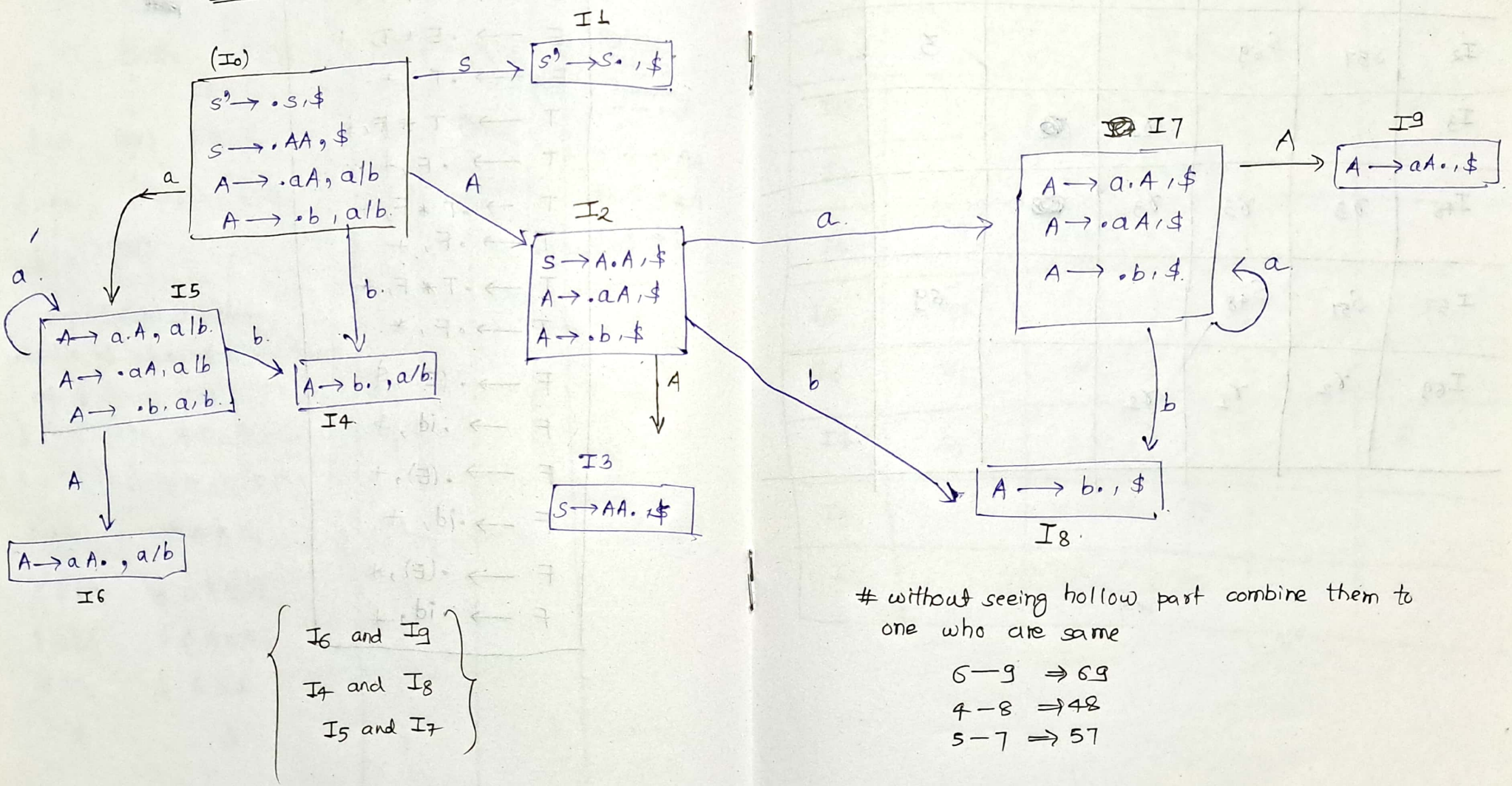
$F \longrightarrow .id, *$

Q. $S \to AA$
$A \to aA$
$A \to b$

0 → $S' \to .S$
1 → $S \to AA$
2 → $A \to aA$
3 → $A \to b$

CLR(1)

bottom up parsing reached starting symbol.



{ I_6 and I_9
I_4 and I_8
I_5 and I_7 }

without seeing hollow part combine them to one who are same

6 — 9 ⇒ 69
4 — 8 ⇒ 48
5 — 7 ⇒ 57

LALR parser

	a	b	$	S	A	
I_0	S_{57}	S_{48}		1	2	
I_1			accepted.			
I_2	S_{57}	S_{69}			3	
I_3			γ_1			
I_{48}	γ_3	γ_3	γ_3			
I_{57}	S_{57}	S_{48}			69	
I_{69}	γ_2	γ_2	γ_2			

CLR(1)

	a	b	$	S	A
I_0	S_5	S_4		1	2
I_1			accepted		
I_2	S_7	S_6			3
I_3			γ_1		
I_4	γ_3	γ_3			
I_5	S_5	S_4			6
I_6	γ_2	γ_2			
I_7	S_7	S_8			9
I_8			γ_3		
I_9			γ_2		