JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

SUBJECT- COMPILER DESIGN

SEMESTER- 5TH SEM

VISSION AND MISSION OF INSTITUTE

To become a renowned center of outcome based learning and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities

M1: Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2: Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.

M3: Offer opportunities for interaction between academia and industry.

M4: Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION OF THE DEPARTMENT

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISION OF THE DEPARTMENT

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies.

M4: To develop aptitude of fulfilling social responsibilities

PROGRAM OUTCOMES

Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOME

CO1: Compare different phases of compiler and design lexical analyzer. CO2: Examine syntax and semantic analyzer by understanding grammars.

CO3: Illustrate storage allocation and its organization & analyze symbol table organization.

CO4: Analyze code optimization, code generation & compare various compilers.

CO-PO Mapping

Semester	Subject	Code	L/T/P	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
V	COMPILER DESIGN	5CS4 - 02	L	1. Compare different phases of compiler and design lexical analyzer.	3	3	3	3	2	1	1	1	1	2	1	3
			L	2. Examine syntax and semantic analyzer and illustrate storage allocation and its organization	3	3	3	3	1	1	1	1	1	2	2	3
			L	3. Analyze symbol table organization, code optimization and code generator	3	3	3	3	2	1	1	1	1	2	2	3
			L	4.Compare and evaluate various compilers and analyzers	3	3	3	3	2	1	1	1	1	2	1	3

PROGRAM EDUCATIONAL OBJECTIVES:

1. To provide students with the fundamentals of Engineering Sciences with more emphasis in **Computer Science &Engineering** by way of analyzing and exploiting engineering challenges.
2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.
3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.
4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self motivated life-long learning needed for a successful professional career.
5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge

PSO

PSO1. Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

PSO2. Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

SYLLABUS

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CS4-02: Compiler Design

Credit: 3 **Max. Marks: 150(IA:30, ETE:120)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:**Objective, scope and outcome of the course.	01
2	**Introduction:** Objective, scope and outcome of the course. Compiler, Translator, Interpreter definition, Phase of compiler, Bootstrapping, Review of Finite automata lexical analyzer, Input, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	06
3	**Review of CFG Ambiguity of grammars:** Introduction to parsing. Top down parsing, LL grammars & passers error handling of LL parser, Recursive descent parsing predictive parsers, Bottom up parsing, Shift reduce parsing, LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Operator precedence parsing, Introduction of automatic parser generator: YACC error handling in LR parsers.	10
4	**Syntax directed definitions;** Construction of syntax trees, S-Attributed Definition, L-attributed definitions, Top down translation. Intermediate code forms using postfix notation, DAG, Three address code, TAC for various control structures, Representing TAC using triples and quadruples, Boolean expression and control structures.	10
5	**Storage organization;** Storage allocation, Strategies, Activation records, Accessing local and non-local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.	08
6	**Definition of basic block control flow graphs;** DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG.	07

LECTURE PLAN:

Subject: Compiler Design (5CS4 – 02) **Year/Sem: III/V**

Unit No./ Total lec. Req.	**Topics**	**Lect. Req.**
Unit-1 (6)	Compiler, Translator, Interpreter definition, Phase of compiler	1
	Introduction to one pass & Multipass compilers, Bootstrapping	1
	Review of Finite automata lexical analyzer, Input, buffering,	2
	Recognition of tokens, Idea about LEX:, GATE Questions	1
	A lexical analyzer generator, Error Handling, Unit Test	1
Unit-2 (17)	Review of CFG Ambiguity of grammars, Introduction to parsing	2
	Bottom up parsing Top down Parsing Technique	5
	Shift reduce parsing, Operator Precedence Parsing	2
	Recursive descent parsing predictive parsers	1
	LL grammars & passers error handling of LL parser	1
	Conical LR & LALR parsing tables	3
	parsing with ambiguous grammar, GATE Questions	2
	Introduction of automatic parser generator: YACC error handling in LR parsers, Unit Test	1
Unit 3- (7)	Syntax directed definitions; Construction of syntax trees	1
	L-attributed definitions, Top down translation	1
	Specification of a type checker, GATE Questions	1
	Intermediate code forms using postfix notation and three address code,	2
	Representing TAC using triples and quadruples, Translation of assignment statement.	1
	Boolean expression and control structures, Unit Test	1
Unit 4- (4)	Storage organization, Storage allocation, Strategies, Activation records,	1
	Accessing local and non local names in a block structured language	1
	Parameters passing, Symbol table organization, GATE Questions	1
	Data structures used in symbol tables, Unit Test	1
Unit 5- (6)	Definition of basic block control flow graphs,	1
	DAG representation of basic block, Advantages of DAG,	1
	Sources of optimization, Loop optimization Idea about global data flow analysis, Loop invariant computation, Loop invariant computation, Tutorial	2
	Peephole optimization, GATE Questions, Tutorial	1
	Issues in design of code generator, A simple code generator, Code generation from DAG., UNIT TEST, Revision	1

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd/ 5th sem
Subject – Compiler Design
Unit – 2

Context-Free Grammar Introduction

Grammar: Grammar is a set of rules which check whether a string belong to a particular language or not.

Context-Free Grammar:

- It is a notation used to specify the syntax of language.
- Context free grammar are used to design parser.

CONTEXT FREE GRAMMAR

What are Context Free Grammars?

- In Formal Language Theory , a Context free Grammar(CFG) is a formal grammar in which every production rule is of the form

$$V \longrightarrow w$$

Where V is a single nonterminal symbol and w is a string of terminals and/or nonterminals (w can be empty)

- The languages generated by context free grammars are knows as the context free languages

Formal Definition of CFG

A context-free grammar G is a 4-tuple (V, ∑, R, S), where:

- V is a finite set; each element $v \in V$ is called *a non-terminal character* or a *variable*.
- ∑ is a finite set of *terminals*, disjoint from , which make up the actual content of the sentence.
- R is a finite relation from V to $(V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation.

If $(\alpha,\beta) \in R$, we write production $\alpha \rightarrow \beta$

β is called a **sentential form**

- S, the **start symbol**, used to represent the whole sentence (or program). It must be an element of V.

Production rule notation

- A production rule in R is formalized mathematically as a pair (α,β) , where α is a non-terminal and β is a string of variables and nonterminals; rather than using ordered pair notation, production rules are usually written using an arrow operator with α as its left hand side and β as its right hand side: $\alpha \rightarrow \beta$.
- It is allowed for β to be the empty string, and in this case it is customary to denote it by ε. The form $\alpha \rightarrow \varepsilon$ is called an ε-production.

Definition: A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where

- N is a set of non-terminal symbols.
- T is a set of terminals where $N \cap T = NULL$.
- P is a set of rules, $P: N \rightarrow (N \cup T)^*$, i.e., the left-hand side of the production rule P does have any right context or left context.
- S is the start symbol.

Example

- The grammar ({A}, {a, b, c}, P, A), P : $A \rightarrow aA$, $A \rightarrow abc$.
- The grammar ({S, a, b}, {a, b}, P, S), P: $S \rightarrow aSa$, $S \rightarrow bSb$, $S \rightarrow \varepsilon$
- The grammar ({S, F}, {0, 1}, P, S), P: $S \rightarrow 00S \mid 11F$, $F \rightarrow 00F \mid \varepsilon$

Derivation Tree/Parse Tree

Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique:

1. Root vertex: Must be labeled by the start symbol.
2. Vertex: Labeled by a non-terminal symbol.
3. Leaves: Labeled by a terminal symbol or ε.

There are two different approaches to draw a derivation tree:

1. Top-down Approach:

(a) Starts with the starting symbol S

(b) Goes down to tree leaves using productions

2. Bottom-up Approach:

(a) Starts from tree leaves

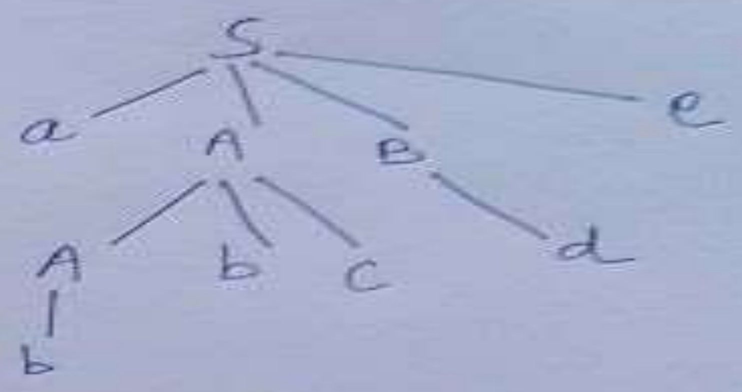
(b) Proceeds upward to the root which is the starting symbol S

Top down Approach

what to use

S → aABe
A → Abc / b
B → d

⇒

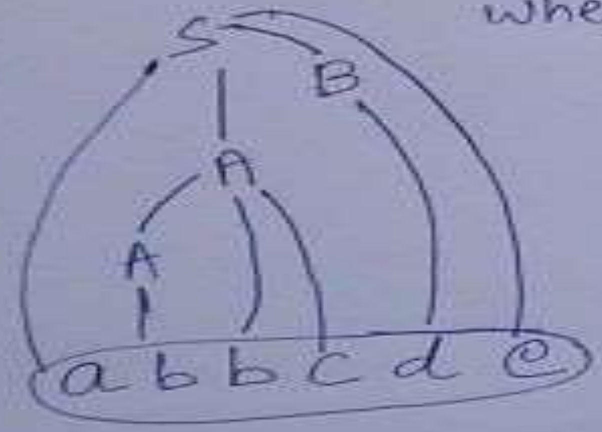


w → abbcde

S ⇒ aⒶBe
⇒ aⒶbcBe
⇒ abbcⒷe
⇒ abbcde

Bottom-Up Approach

when to Reduce



S ⇒ aABe
⇒ aAde
⇒ aAbcde
⇒ abbcde

Types of Derivation Tree

Leftmost and Rightmost Derivation of a String

1. **Leftmost derivation** - A leftmost derivation is obtained by applying production to the leftmost variable in each step.

2. **Rightmost derivation** - A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

over an alphabet {a}.

Find the leftmost derivation for the string "a+a*a".

Answer:

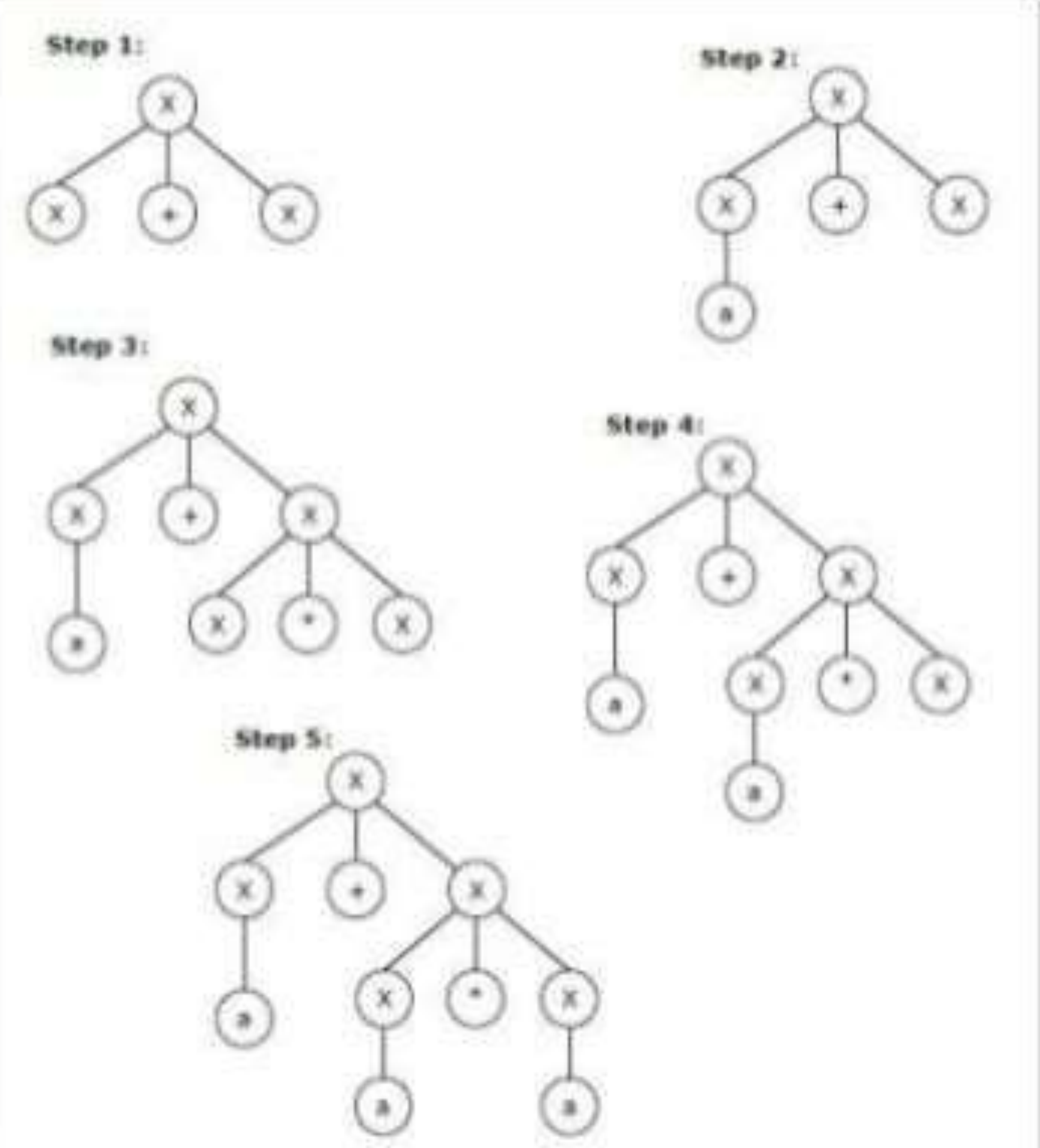
X → X+X

X→ a+X

X→ a+ X*X

X→a+a*X

X→ a+a*a



Example

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

over an alphabet {a}.

Find the Rightmost derivation for the string "a+a*a".

Answer:

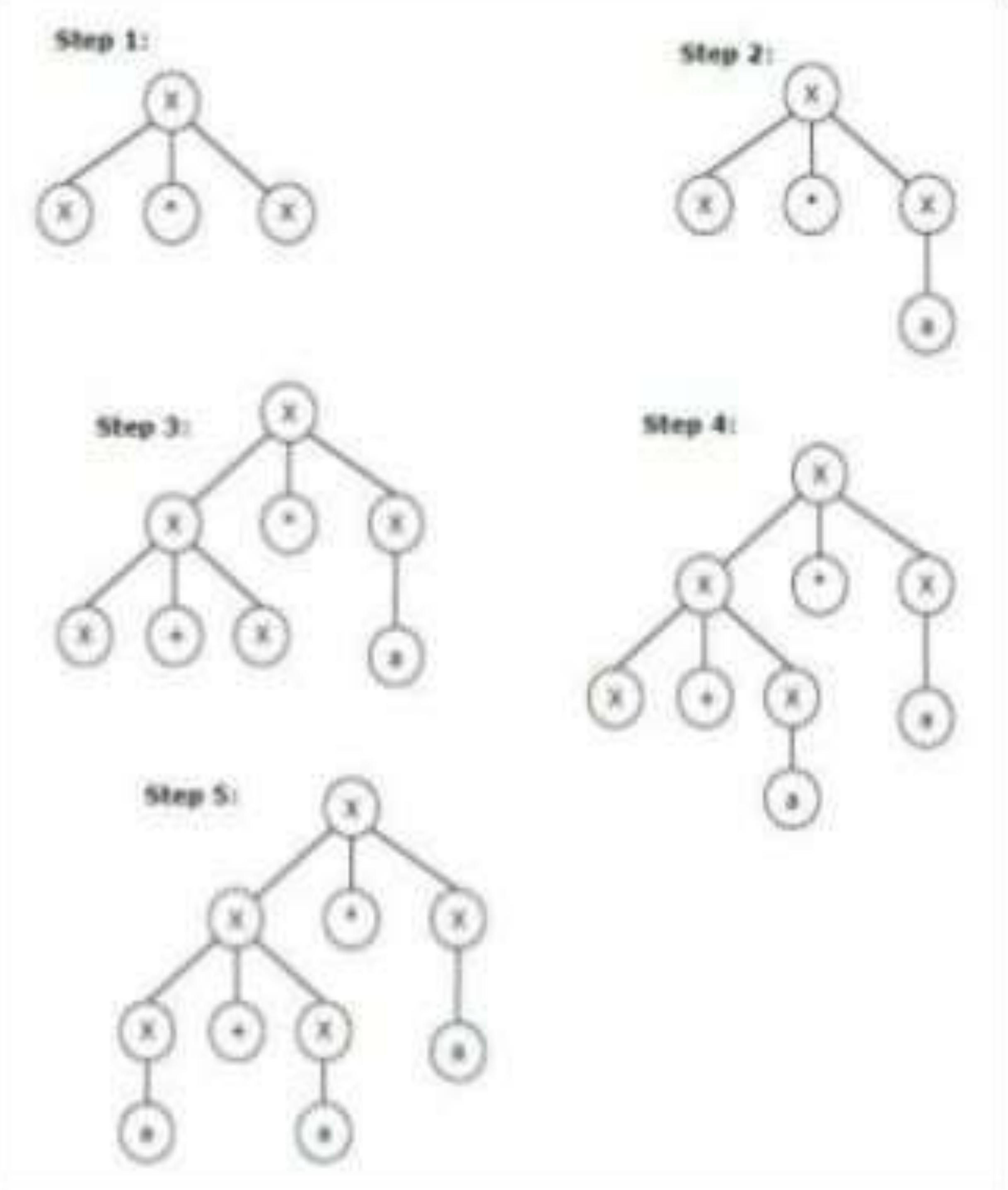
X → X*X

X→ X*a

X → X+X*a

X →X+a*a

X→ a+a*a



Ambiguity in CFG

If a context free grammar **G** has more than one derivation tree for some string **w ∈ L(G)**, it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

Check whether the grammar G with production rules −

X → X+X | X*X |X| a   is ambiguous or not.

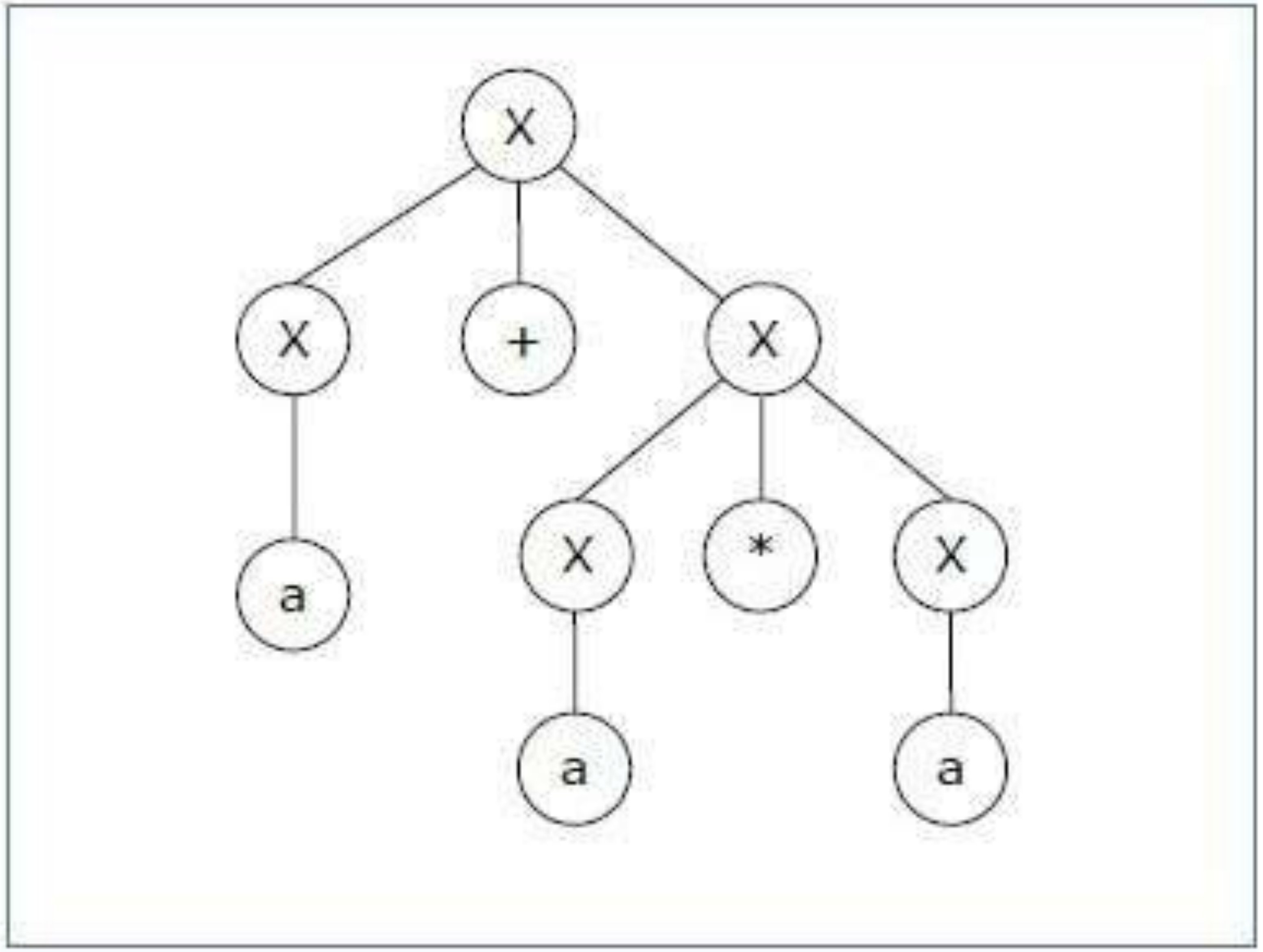
Solution

Let’s find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1 −

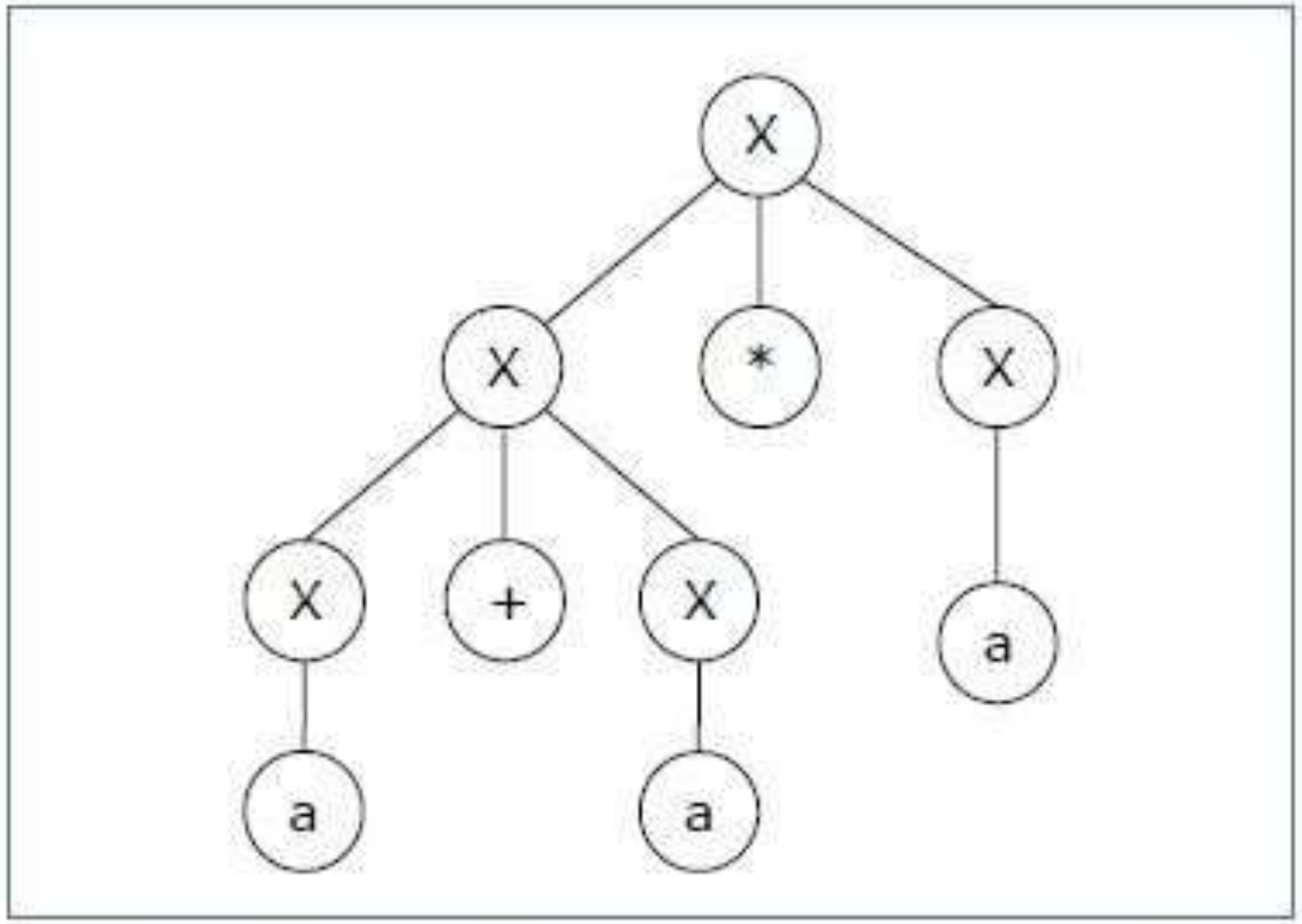
X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 –



Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 −



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

INTRODUCTION TO PARSING

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Parsing is used to derive a string using the production rules of a grammar. It is **used** to check the acceptability of a string. Compiler is **used** to check whether or not a string is syntactically correct. A **parser** takes the inputs and builds a **parse** tree.

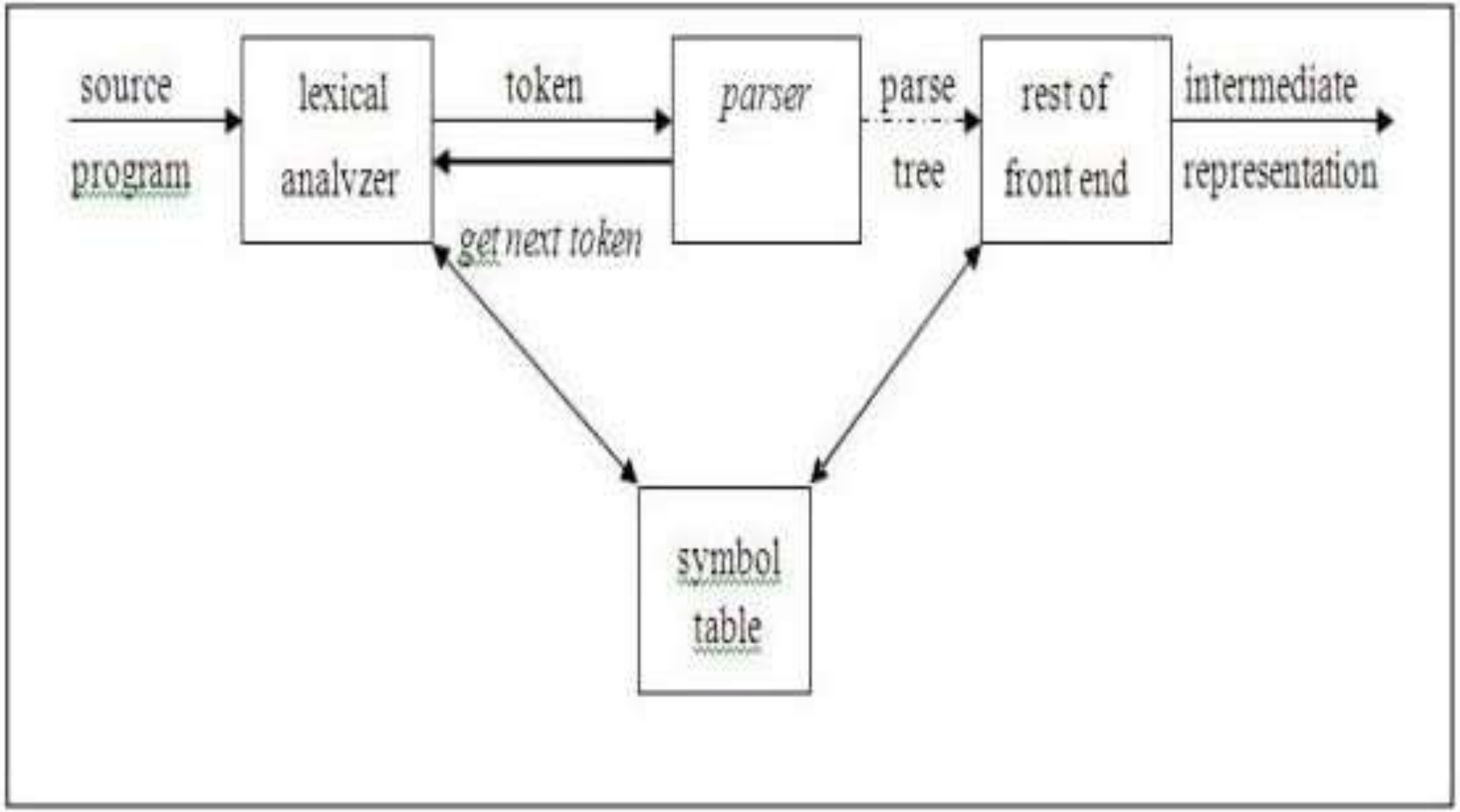
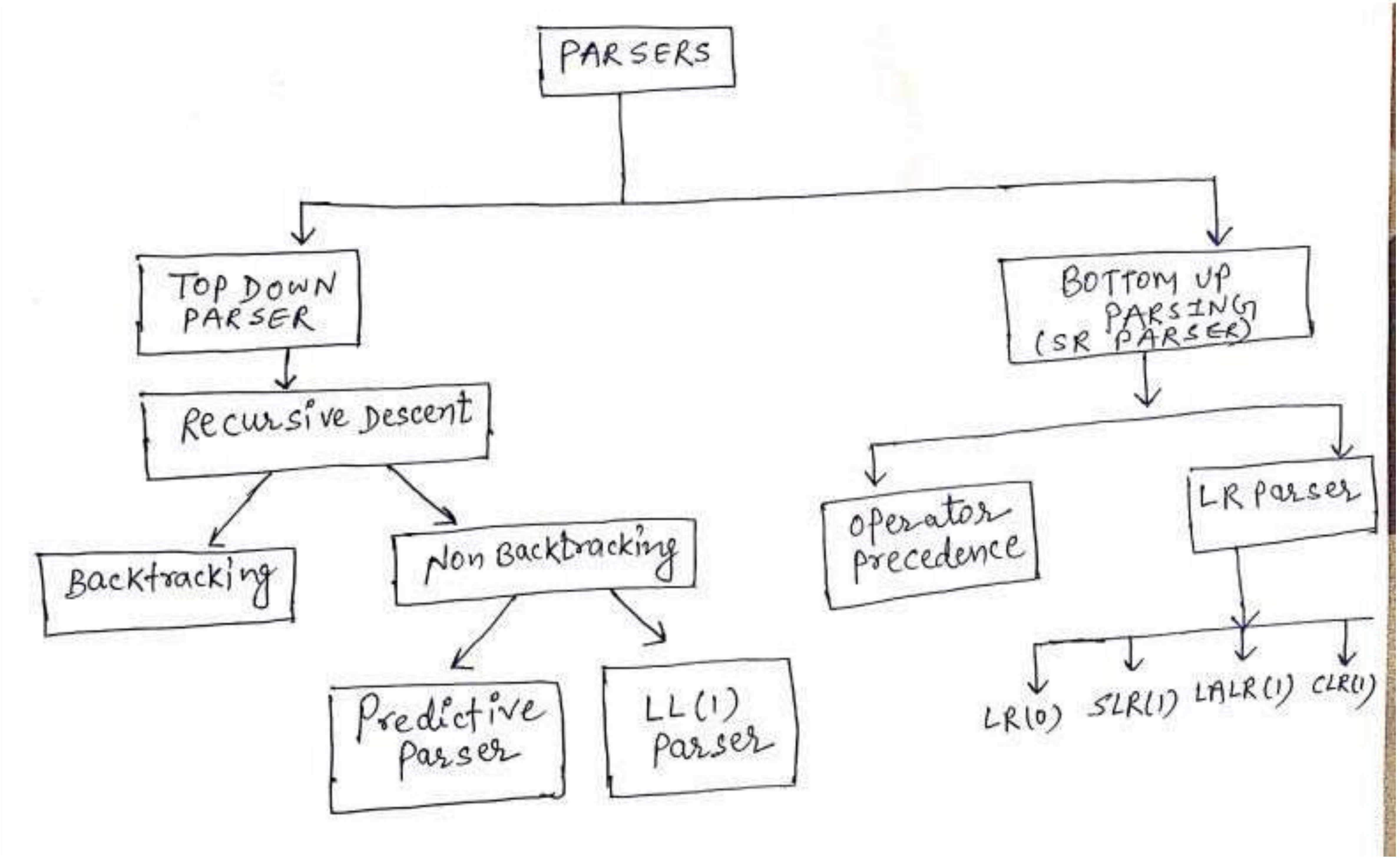


Fig. 2.1 Position of parser in compiler model

TYPES OF PARSERS



Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).

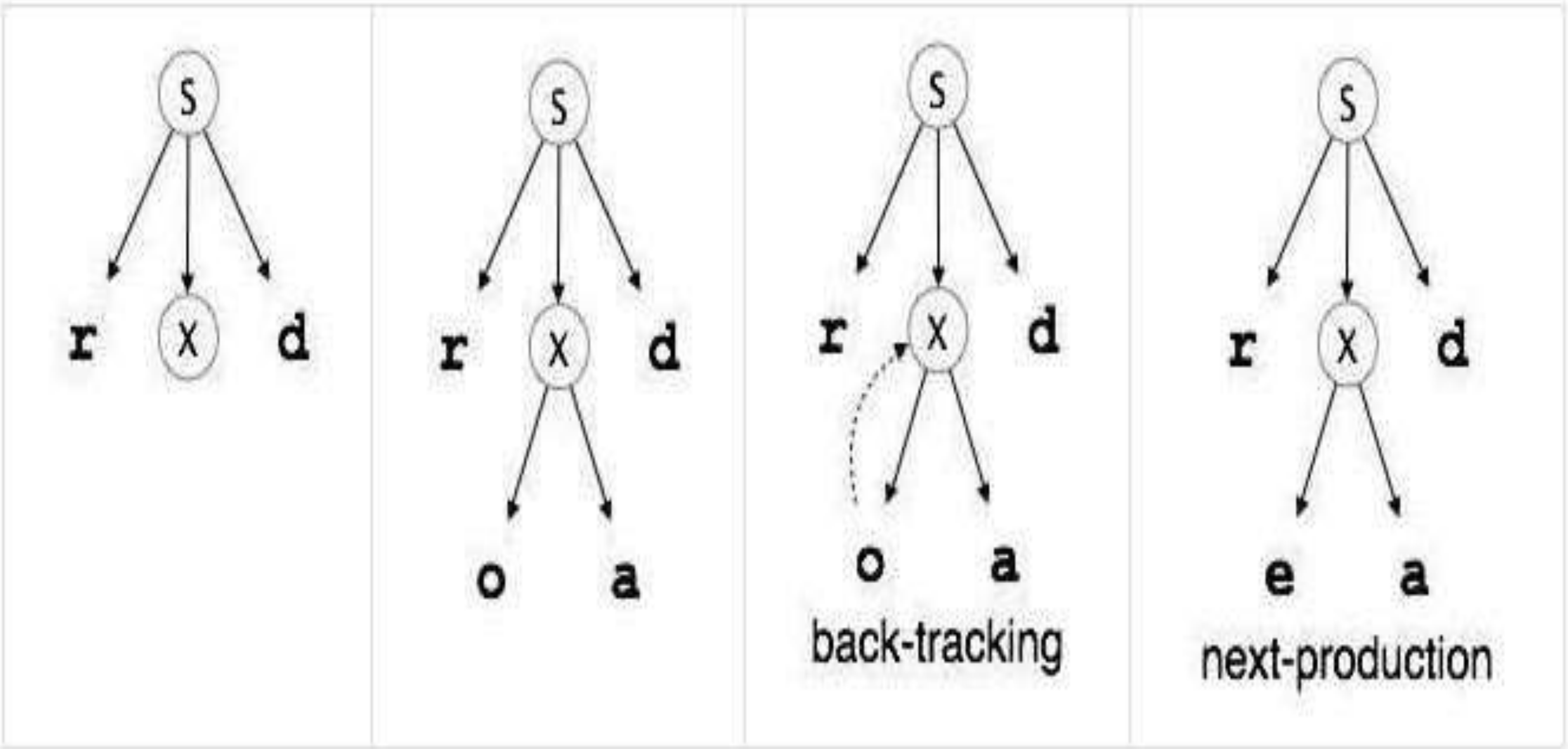
To understand this, take the following example of CFG:

S → rXd | rZd X → oa | ea Z → ai

String →read

For an input string: **read**, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).


S
r
X
d
o
a
back-tracking
e
next-production

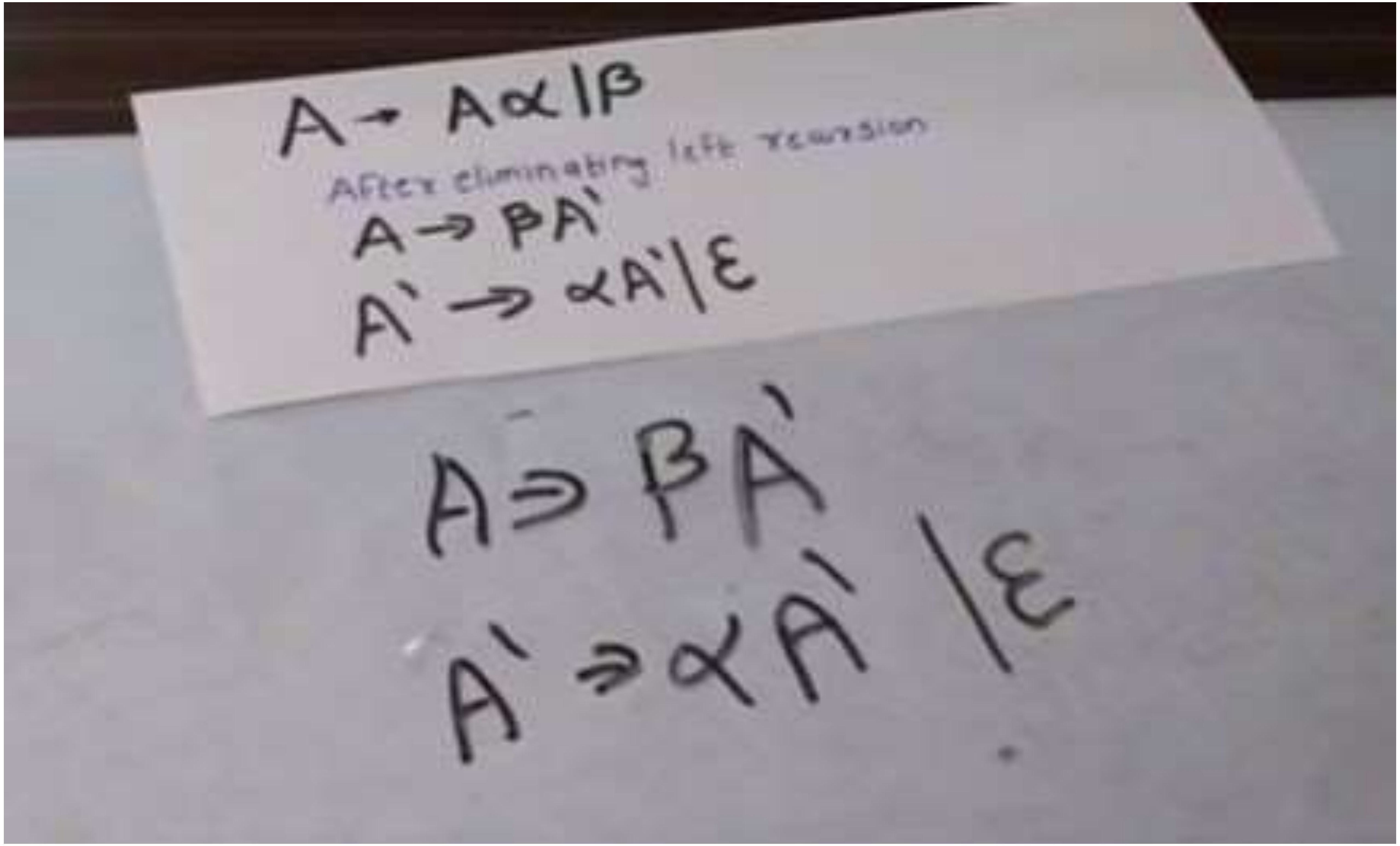
LEFT RECURSION

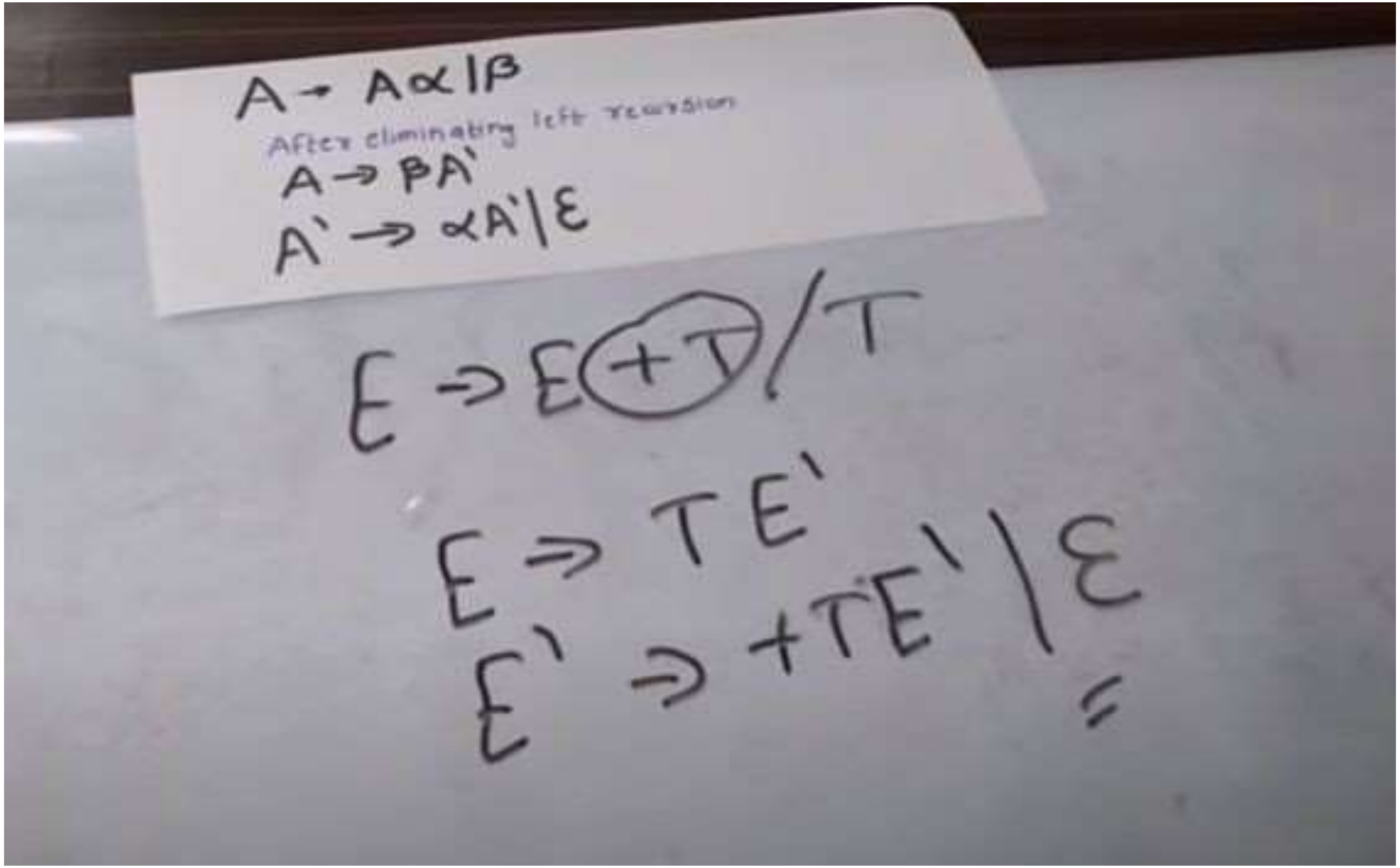
- Left recursion is a case when the left-most non-terminal in a production of a non-terminal is the non-terminal itself(direct left recursion) or through some other non-terminal definitions, rewrites to the non-terminal again(indirect left recursion). Consider these examples -
- (1) A -> Aq (direct)
- (2) A -> Bq

  B -> Ar (indirect)
- Left recursion has to be removed if the parser performs top-down parsing

LEFT RECURSION

- We have to eliminate left recursion because top down parsing methods can not handle left recursive grammars.


A→ Aα|β
After eliminating left recursion
A→βA`
A`→αA`|ε
A⇒βA`
A`⇒αA`|ε


A → Aα|β
After eliminating left recursion
A → βA`
A` → αA`|ε
E → E+T/T
E → TE`
E` → +TE`|ε

LEFT FACTORING

- In left factoring it is not clear which of two alternative productions to use to expand a nonterminal A.

  i.e. if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

- W e don't know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$
- To remove left factoring for this grammar replace all A productions containing α as prefix by $A \rightarrow \alpha A'$ then $A' \rightarrow \beta_1 \mid \beta_2$

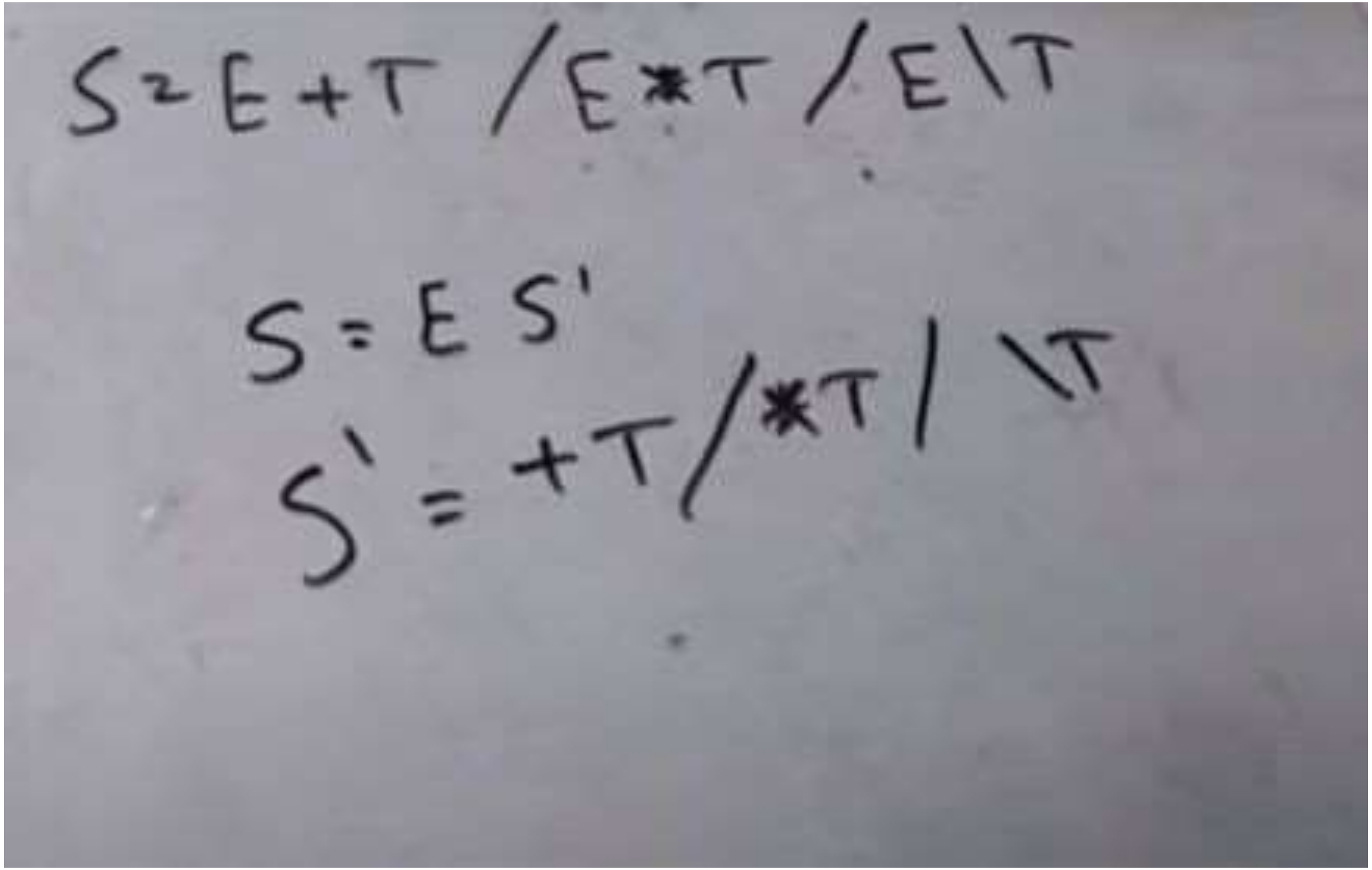
Left factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

After eliminating left factoring

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$


S = E+T /E*T /E\T
S = E S'
S' = +T /*T / \T

Recursive Descent Parsing

- Recursive descent parsing is a top-down method of syntax analysis in which a set recursive procedures to process the input is executed.
- A procedure is associated with each nonterminal of a grammar.
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- Equivalently, it attempts to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

EXAMPLE

Grammar:
E --> i E'
E' --> + i E' | ε

-Here non terminals are E and E'
- String we have to parse is: i+i$

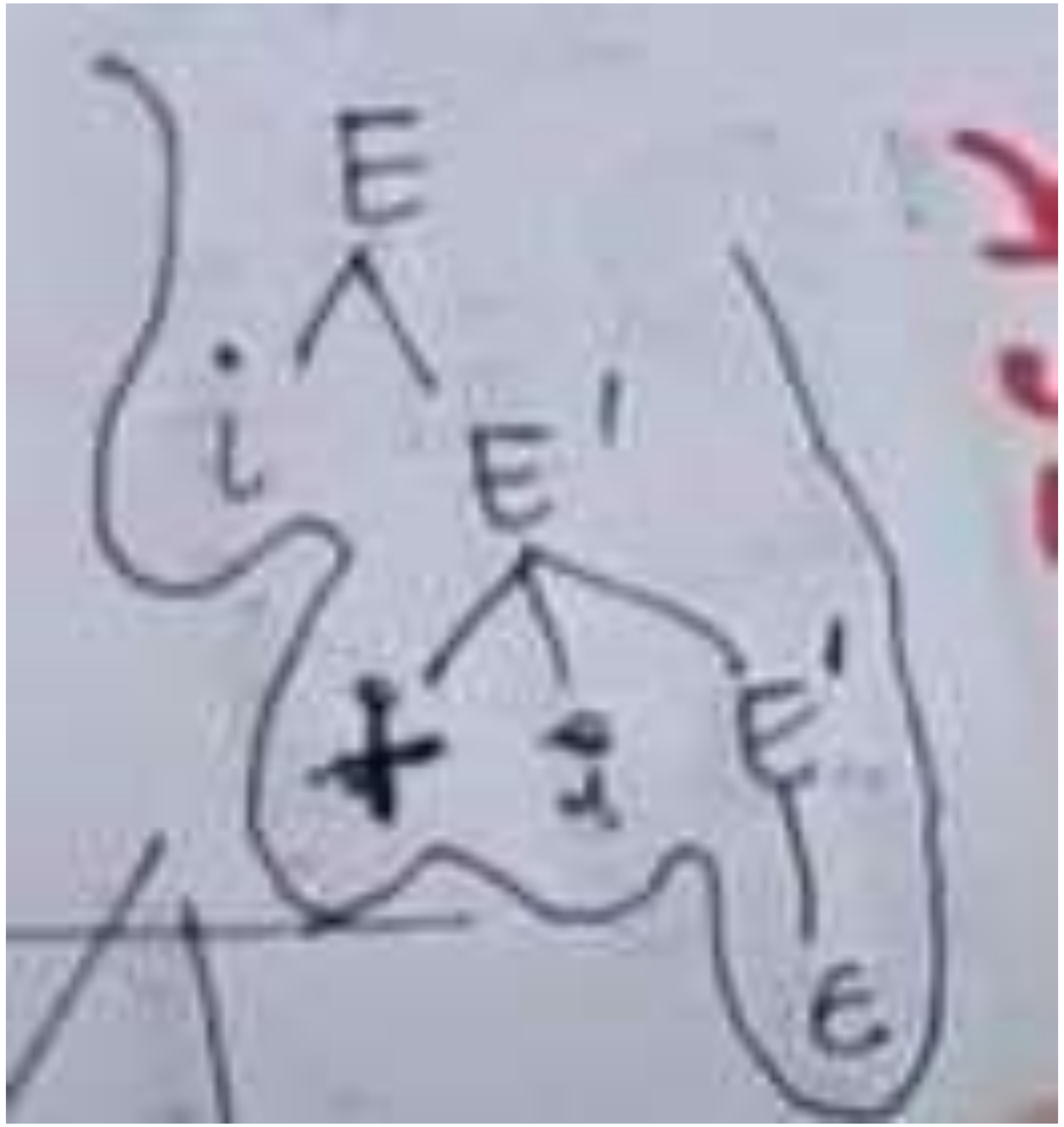
```
int main()
{
    // E is a start symbol.
    E();

    // if lookahead = $, it represents the end of the string
    // Here l is lookahead.
    if (l == '$')
        printf("Parsing Successful");
}
```

```
// Definition of E, as per the given production

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

// Definition of E' as per the given production

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

```
// Match function

match(char t)
{
    if (l == t) {
        l = getchar();
    }
    else
        printf("Error");
}
```


E
i
E'
+
E'
ε

Why FIRST and FOLLOW in Compiler Design?

Why FIRST?

We saw the need of backtrack in Introduction to Syntax Analysis, which is really a complex process to implement. There can be easier way to sort out this problem:

If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

Example

S -> cAd

A -> bc|a

And the input string is **“cad”.**

Thus, in the example above, if it knew that after reading character ‘c’ in the input string and applying S->cAd, next character in the input string is ‘a’, then it would have ignored the production rule A->bc (because ‘b’ is the first character of the string produced by this production rule, not ‘a’)

And directly use the production rule A->a (because ‘a’ is the first character of the string produced by this production rule, and is same as the current character of the input string which is also ‘a’). Hence it is validated that if the compiler/parser knows about first character of the string that can be obtained by applying a production rule, then it can wisely apply the correct production rule to get the correct syntax tree for the given input string.

Calculation of FIRST Set in Syntax Analysis

Rules to compute FIRST set:

1. If X is a terminal, then FIRST(X) = { ‘X’ }
2. (i) If X is a non terminal and X-> aα then add a to FIRST{ X }.
   (ii) If X-> Є, is a production rule, then add Є to FIRST(X).
3. If X->Y1 Y2 Y3….Yn is a production(Y1,Y2.. Are non terminal),
   a)FIRST(X) = FIRST(Y1 Except Є)
   b)If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y2) Except Є }
   c)If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Example 1:

Production Rules of Grammar

E -> TE'

E' -> +T E'|Є

T -> F T'

T' -> *F T' | Є

F -> (E) | id

FIRST sets FIRST(E) = FIRST(T) = { (, id }

FIRST(E') = { +, Є }

FIRST(T) = FIRST(F) = { (, id }

FIRST(T') = { *, Є }

FIRST(F) = { (, id }

Example 2:

Production Rules of Grammar

S -> ACB | Cbb | Ba

A -> da | BC

B -> g | Є

C -> h | Є

FIRST sets

FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) = { d, g, h, Є, b, a}

FIRST(A) = { d } U FIRST(B) = { d, g , h, Є }

FIRST(B) = { g , Є }

FIRST(C) = { h , Є }

Why FOLLOW?

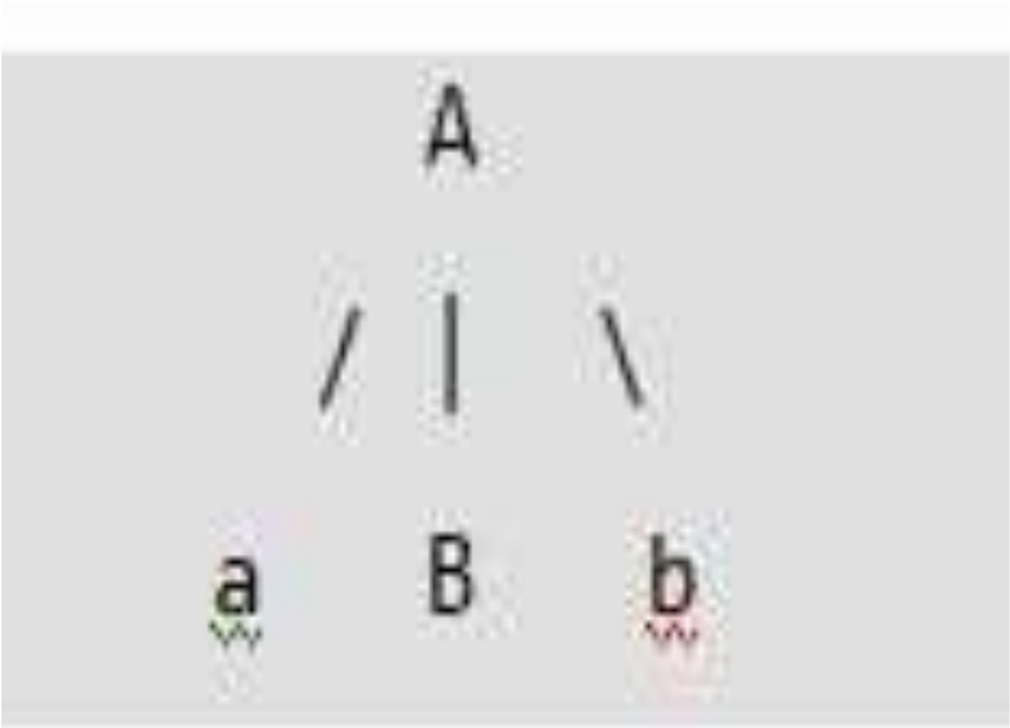
The parser faces one more problem. Let us consider below grammar to understand this problem.

A -> aBb

B -> c | ε

And suppose the input string is “ab” to parse.

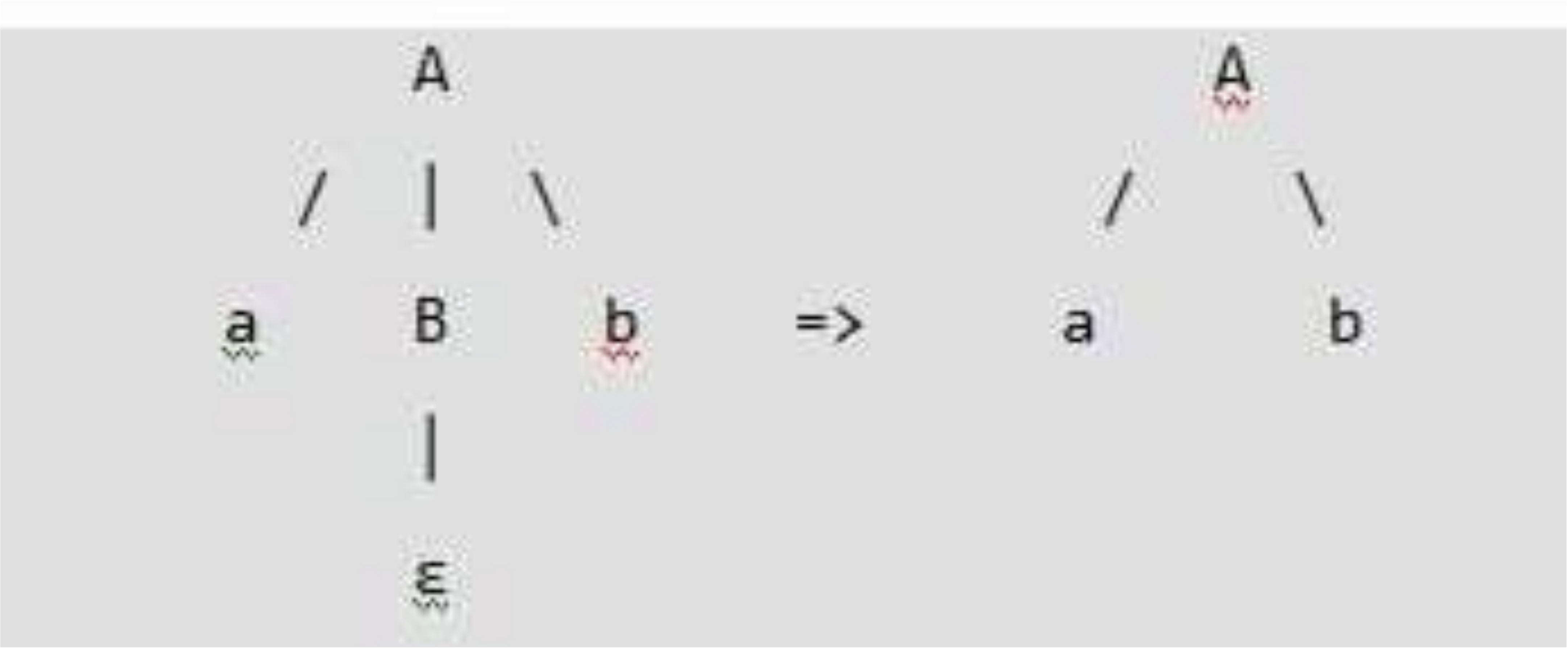
As the first character in the input is a, the parser applies the rule A->aBb.



Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can’t get any string derivable from B that contains b as first character.

But the Grammar does contain a production rule B -> ε, if that is applied then B will vanish, and the parser gets the input “ab” , as shown below. But the parser can apply it only when it knows that the character that follows B in the production rule is same as the current character in the input.

In RHS of A -> aBb, b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string “ab” from the given grammar.



So FOLLOW can make a Non-terminal to vanish out if needed to generate the string from the parse tree.

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

Rules to compute FOLLOW set:

1) FOLLOW(S) = { $ }   // where S is the starting Non-Terminal
2) 2) If A -> αBβ is a production, where α,B, and β are any grammar symbols, where β ≠ Є then everything in FIRST(β) except Є is in FOLLOW(B).
3) If A->αB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
4) If A->αBβ is a production and FIRST(β) contains Є,   then Everything in FOLLOW(A) is in FOLLOW(B).

Production Rules:

E -> TE'

E' -> +T E' |Є

T -> F T'

T' -> *F T' | Є

F -> (E) | id

FIRST set

FIRST(E) = FIRST(T) = { (, id }

FIRST(E') = { +, Є }

FIRST(T) = FIRST(F) = { (, id }

FIRST(T') = { *, Є }

FIRST(F) = { (, id }

FOLLOW Set

FOLLOW(E) = { $,) }

FOLLOW(E') = FOLLOW(E) = { $,) }

FOLLOW(T) = { FIRST(E') – Є } U FOLLOW(E') U FOLLOW(E) = { + , $,) }

FOLLOW(T') = FOLLOW(T) = { + , $,) }

FOLLOW(F) = { FIRST(T') – Є } U FOLLOW(T') U FOLLOW(T) = { *, +, $,) }

Production Rules:

S -> aBDh

B -> cC

C -> bC | Є

D -> EF

E -> g | Є

F -> f | Є

FIRST set

FIRST(S) = { a }

FIRST(B) = { c }

FIRST(C) = { b , Є }

FIRST(D) = FIRST(E) U FIRST(F) = { g, f, Є }

FIRST(E) = { g , Є }

FIRST(F) = { f , Є }

FOLLOW Set

FOLLOW(S) = { $ }

FOLLOW(B) = { FIRST(D) – Є } U FIRST(h) = { g , f , h }

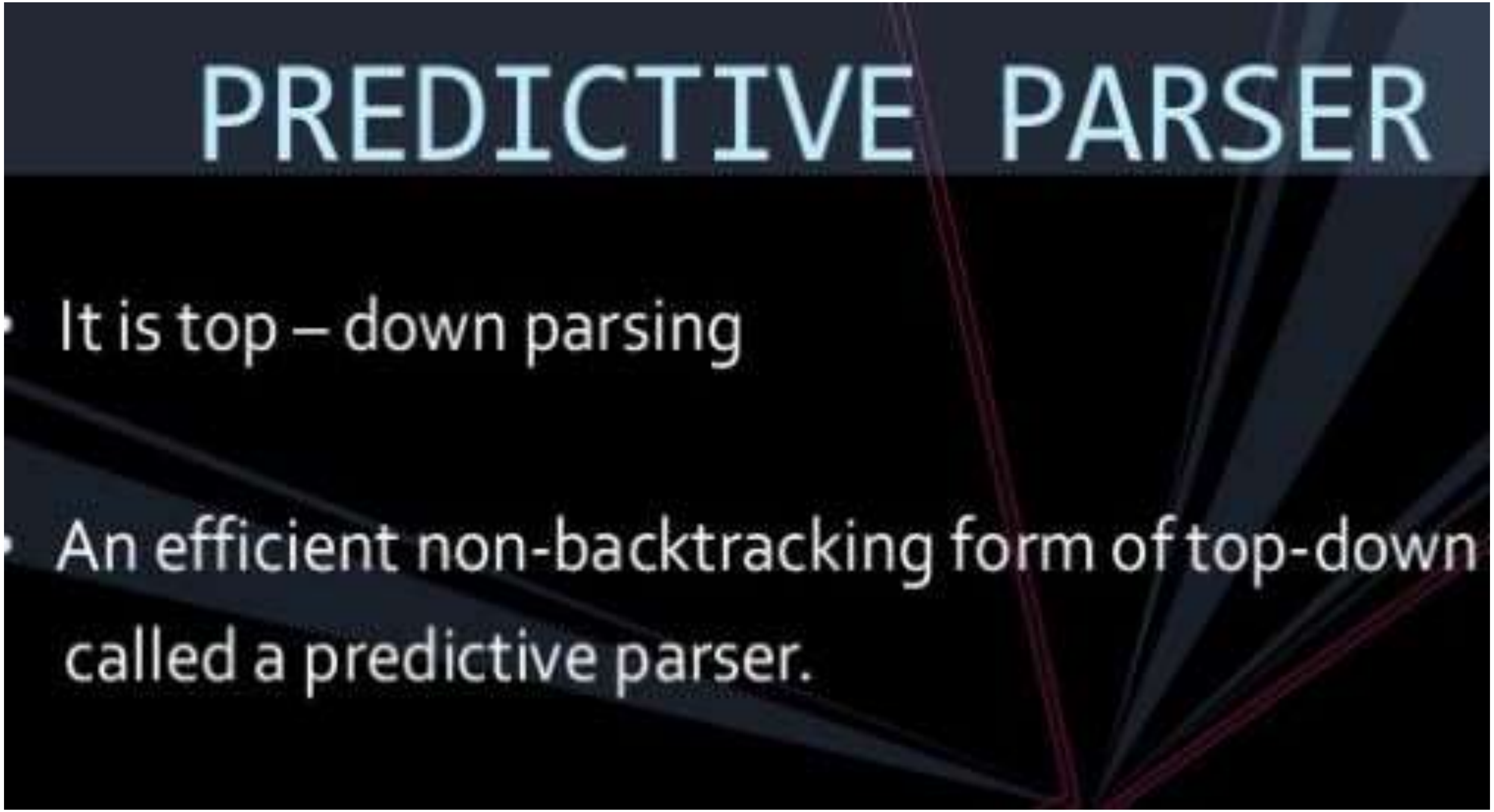
FOLLOW(C) = FOLLOW(B) = { g , f , h }

FOLLOW(D) = FIRST(h) = { h }

FOLLOW(E) = { FIRST(F) – Є } U FOLLOW(D) = { f , h }

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

PREDICTIVE PARSING



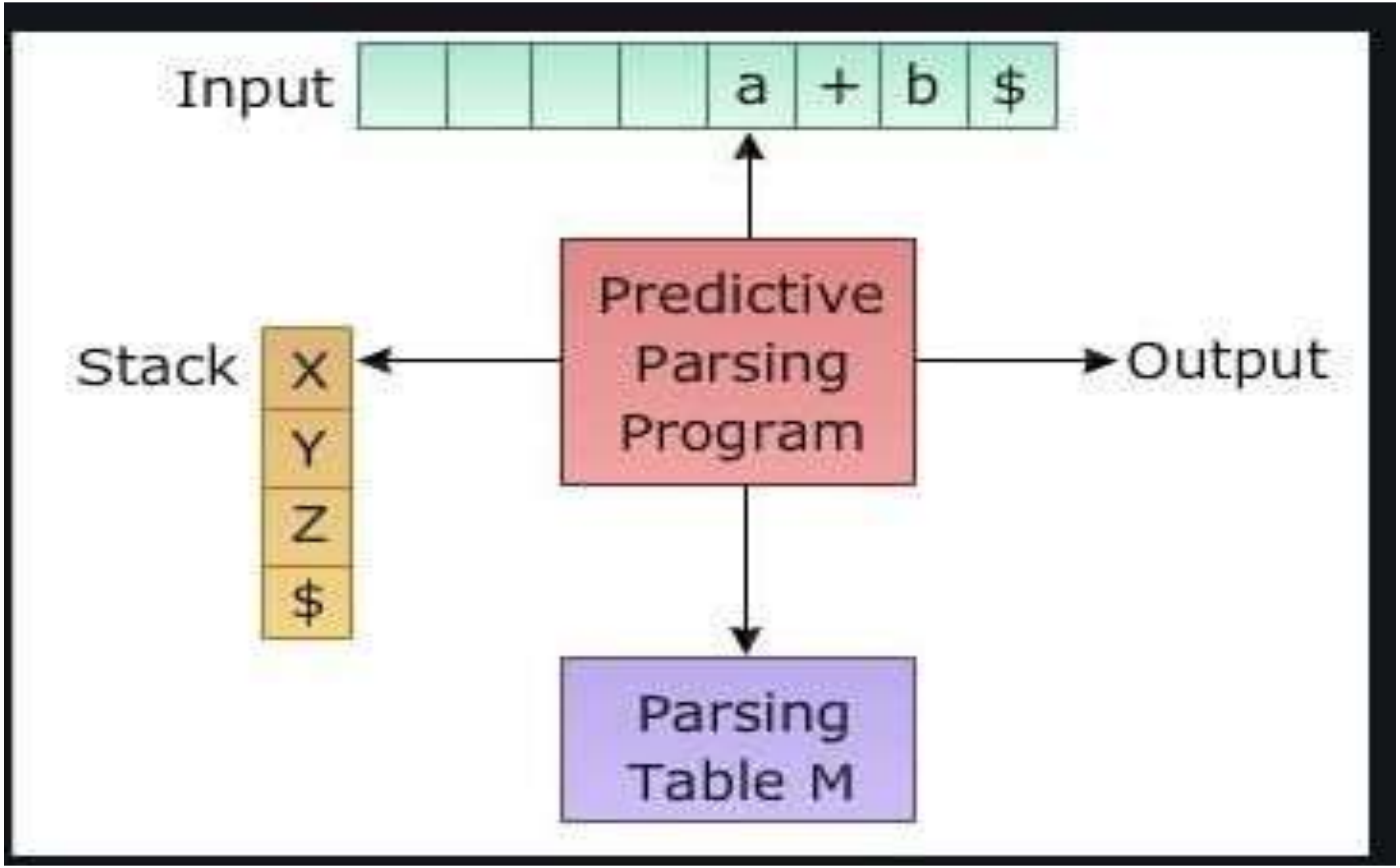
- To construct a predictive parser we must know
  - Input symbol a
  - Non terminal A to be expanded
  - Alternatives of production A-->a1|a2|.....|an
  - That derives a string beginning with a
  - Proper alternative must be detectable by looking at only first symbol it derives.

predictive parser has :-

- input - contain string to parser followed by $
- stack - sequence of grammar symbols with $
- parsing table - 2 dimension array M[A,a]
- output stream - gives output

Parsing Program- Used to apply correct production from parsing table

MODEL OF PREDICTIVE PARSING



Construction of Parsing Table

INPUT- GRAMMAR G
OUTPUT: Parsing Table M
METHOD: Following steps are used for the production rule A -> α

1. For each a in FIRST(α) create an entry A -> α for M[A,a] where a is terminal symbol.
2. If FIRST(α) ={ε} create an entry M[A,b]= A -> α where b is the symbol in FOLLOW(A).
3. If FIRST(α) ={ε} and FOLLOW(A)=$ create an entry M[A,$]= A -> α
4. Leftover entries are marked as error entry.

Example:

Production Rules:

E -> TE'

E' -> +T E' |Є

T -> F T'

T' -> *F T' | Є

F -> (E) | id

FIRST set

FIRST(E) = FIRST(T) = { (, id }

FIRST(E') = { +, Є }

FIRST(T) = FIRST(F) = { (, id }

FIRST(T') = { *, Є }

FIRST(F) = { (, id }

FOLLOW Set

FOLLOW(E) = { $,) }

FOLLOW(E') = FOLLOW(E) = { $,) }

FOLLOW(T) = { FIRST(E') – Є } U FOLLOW(E') U FOLLOW(E) = { + , $,) }

FOLLOW(T') = FOLLOW(T) = { + , $,) }

FOLLOW(F) = { FIRST(T') – Є } U FOLLOW(T') U FOLLOW(T) = { *, +, $,) }

PARSING TABLE FOR GRAMMAR G

Nonter-minal	Input Symbol					
	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->ε	E'->ε
T	T->FT'			T->FT'		
T'		T'->ε	T'->*FT'		T'->ε	T'->ε
F	F->id			F->(E)		

Grammar:

E → TE'

E' → +TE' | Є

T → FT'

T' → *FT' | Є

F → (E) | id

PREDICTIVE PARSING ALGORITHM

```
Repeat
Begin
    Let X be the top stack symbol and a the next input symbol
if  X is a terminal then
    if X=a then
POP X from the stack and remove a from input and also advance the input pointer.
    else ERROR()
else                  / *  X is a non terminal  *\
    if M[X, a]= X->Y1 Y2......Yk then
        Begin
            POP X from the stack
            PUSH Yk .......Y2 Y1 on the stack
        end
    else
        ERROR()
End
Until X=$             / *  Stack has Emptied *\
```

Example of Parsing a string using Predictive Parser

With input id+id*id the predictive parser makes the sequence of moves as shown below

STACK	INPUT	OUTPUT
$E	id+id*id$	
$E'T	id+id*id$	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	Match id
$E'	+id*id$	T' → ε
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	Match +
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	Match id
$E'T'F*	*id$	T' → *FT'
$E'T'F	Id$	Match *
$E'T'id	id$	F → id
$E'T'	$	Match id
$E'	$	T' → ε
$	$	E' → ε

20

Nonter-minal	Input Symbol					
	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->ε	E'->ε
T	T->FT'			T->FT'		
T'		T'->ε	T'->*FT'		T'->ε	T'->ε
F	F->id			F->(E)		

Grammar:

E → TE'

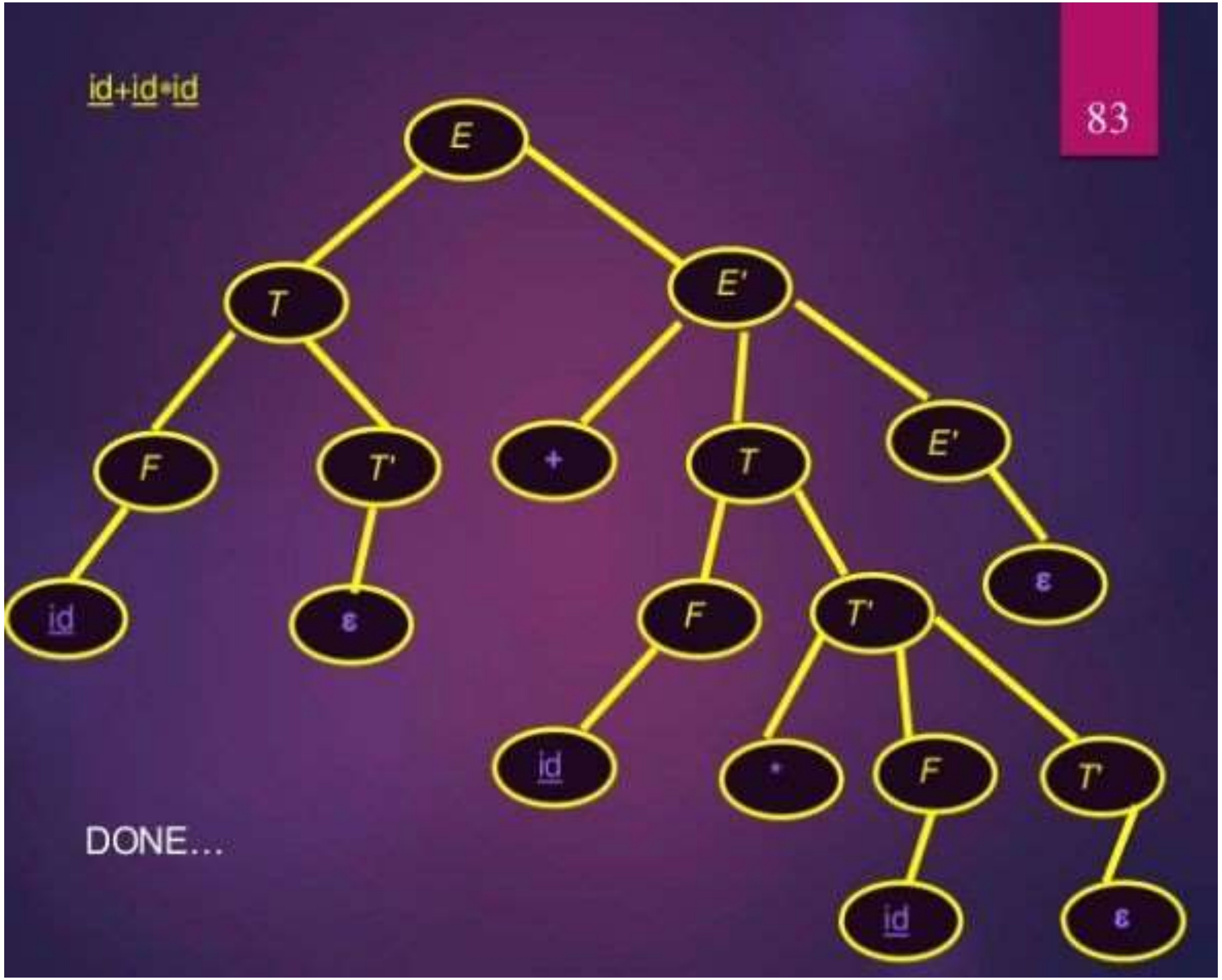
E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Parse Tree



Error recovery in predictive parsing

- An error is detected during predictive parsing when
  - ✓ The terminal on the top of the stack does not match the next input symbol.
  - ✓ When nonterminal A is on top of the stack ,*a* is the next input symbol and the parsing table entry M[A,a] is empty.

SYNTAX ANALYSIS

When the stream of tokens coming from Lexical Analyzer disobeys the syntactic (grammatical) rules of a language, syntactic errors occur.

Handling Syntactic Errors

- Report errors
- Recover from discovered errors
- Aim at not slowing down the processing of the remaining pgm.

Error Recovery Strategies

1. PANIC MODE RECOVERY : The Parser discards the input symbols one at a time untill one of the designated set of synchronizing tokens is found.

- Simple and it does not go into ∞ loop.
- Adequate when the presence of multiple errors in same stmt is rare.

2. PHRASE LEVEL RECOVERY : The Parser performs local correction on remaining input when the error is discovered

- The parser replaces the prefix of the remaining input by some string that allows the parser to carry on its execution.

drawback : Error correction is difficult when actual error occured before the point of detection

3. ERROR PRODUCTIONS

If we know the common errors that can be encountered, we can augment the grammar for the language with productions that generate erroneous constructs.

- Use the new grammar (with the augmented productions) for the parser.
- In case an error production is used by parser, generate appropriate diagnostics to indicate the errors/erroneous constructs.

4. GLOBAL CORRECTION

The aim is to make as few changes as possible while converting an incorrect input string to a valid string.

- Given an incorrect input x, find a parse tree for a related string w (using the given grammar) such that the no. of changes (insertions/deletions) required to transform x to w is minimum
- Too Costly To Implement.

Panic mode recovery

- Panic mode recovery:

➢ Skip symbols on the input until a token in a selected set of synchronizing tokens appear

✓ Use FOLLOW symbols as synchronizing tokens

✓ Use *synch* in predictive parsing table to indicate synchronizing tokens obtains from the follow set of the non-terminal.

Panic mode recovery

Rules:

1.If parser looks up entry *M[A,a]* and finds it blank then the input symbol a is skipped.

2.If the entry is *synch* then the nonterminal on top of the stack is popped in an attempt to resume parsing.

3.If a token on the top of the stack does not match the input symbol, then we pop the token from the stack.

“synch” indicating synchronizing tokens obtained from FOLLOW set of the nonterminal.

If the parser looks up entry M[A,a] and finds that it is blank, the input symbol a is skipped.

If the entry is synch, the nonterminal on top of the stack is popped.

If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

Example

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E'	{+,ε}	{), $}
T'	{*,ε}	{+,), $}

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> **id**			F -> (E)		

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'	synch	synch
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'	synch		T -> FT'	synch	synch
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> **id**	synch	synch	F -> (E)	synch	synch

Example

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'	synch	synch
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'	synch		T -> FT'	synch	synch
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> id	synch	synch	F -> (E)	synch	synch

STACK	INPUT	REMARK
$\$E$	) **id** ∗ + **id** $	error, skip)
$\$E$	**id** ∗ + **id** $	**id** is in FIRST(E)
$\$E'T$	**id** ∗ + **id** $	
$\$E'T'F$	**id** ∗ + **id** $	
$\$E'T'$**id**	**id** ∗ + **id** $	
$\$E'T'$	∗ + **id** $	
$\$E'T'F*$	∗ + **id** $	
$\$E'T'F$	+ **id** $	error, $M[F, +]$ = synch
$\$E'T'$	+ **id** $	F has been popped
$\$E'$	+ **id** $	
$\$E'T+$	+ **id** $	
$\$E'T$	**id** $	
$\$E'T'F$	**id** $	
$\$E'T'$**id**	**id** $	
$\$E'T'$	$	
$\$E'$	$	
$	$	

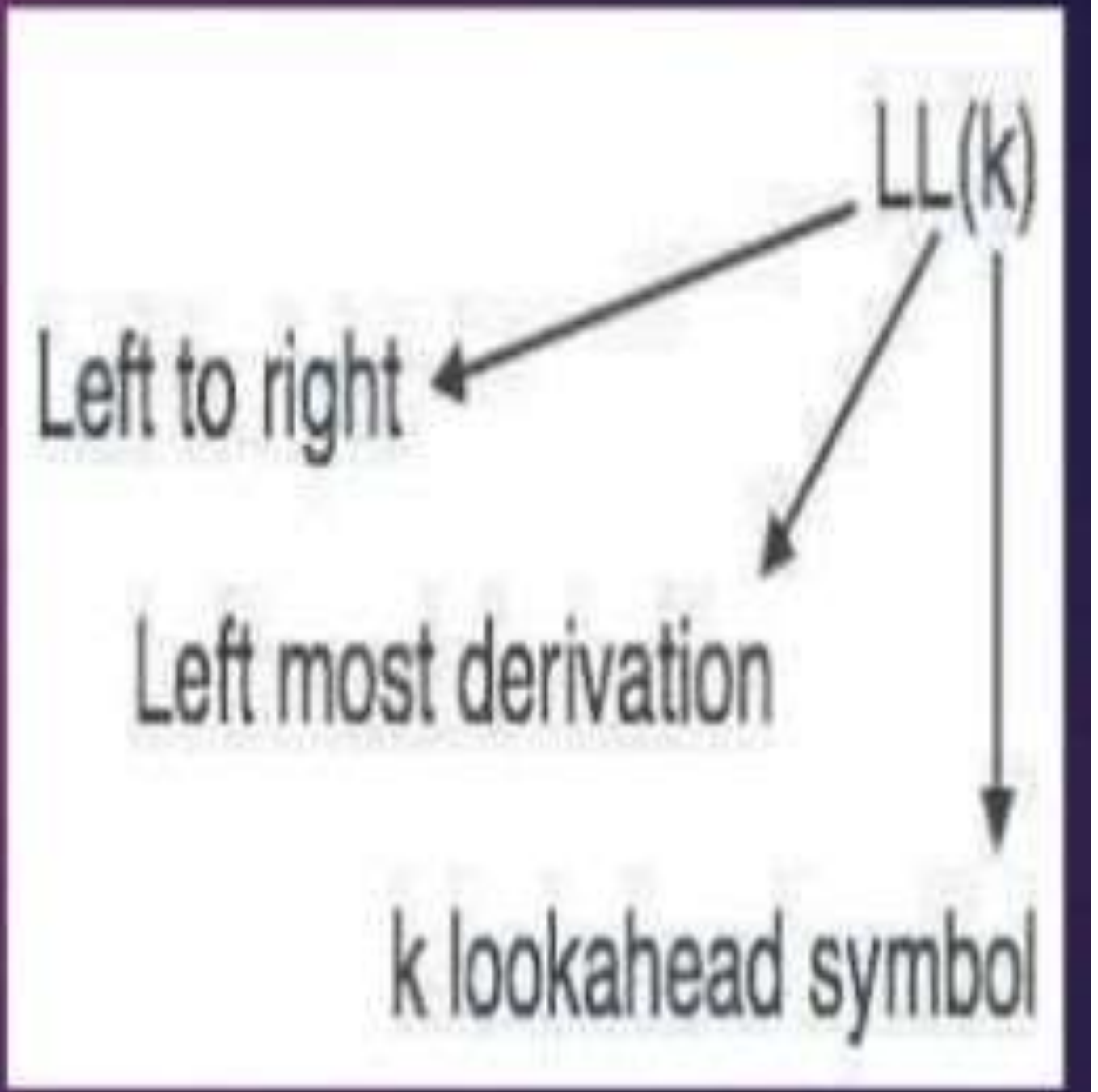
Fig. 4.19. Parsing and error recovery moves made by predictive parser.

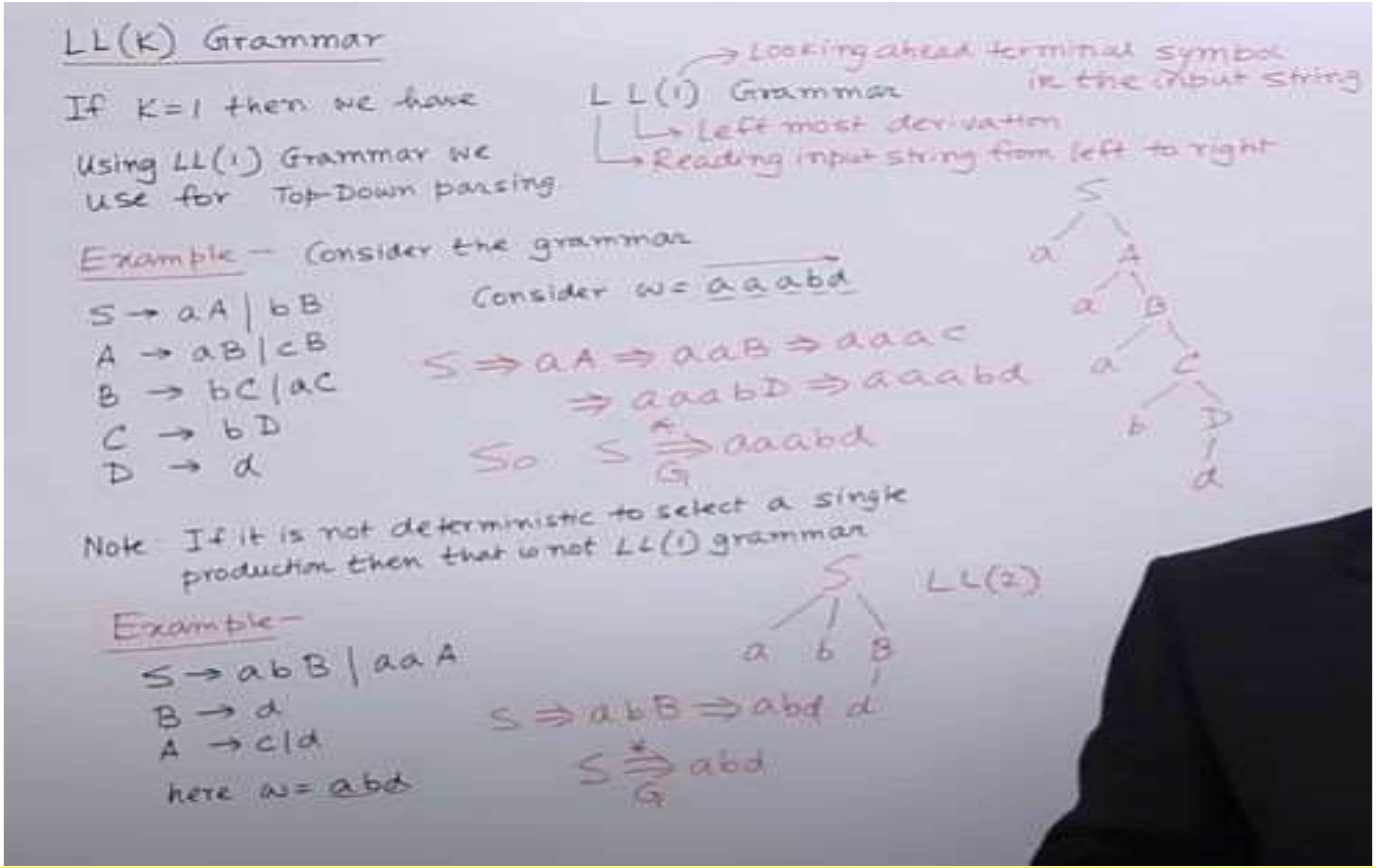
LL(k) Parser:

- top-down parser - starts with start symbol on stack, and repeatedly replace nonterminals until string is generated.
- predictive parser - predict next rewrite rule
- first L of LL means - read input string left to right
- second L of LL means - produces leftmost derivation
- k - number of lookahead symbols

LL(1)

- An LL parser is called an LL(*k*) parser if it uses *k* tokens of look ahead when parsing a sentence.
- LL grammars, particularly LL(1) grammars, as parsers are easy to construct, and many computer languages are designed to be LL(1) for this reason.
- The 1 stands for using **one** input symbol of look ahead at each step to make parsing action decision.




LL(K) Grammar
If K=1 then we have
Using LL(1) Grammar we use for Top-Down parsing
LL(1) Grammar
Looking ahead terminal symbol in the input string
Left most derivation
Reading input string from left to right
Example – Consider the grammar
S → aA | bB
A → aB | cB
B → bC | aC
C → bD
D → d
Consider w = aaabd
S ⇒ aA ⇒ aaB ⇒ aaaC
⇒ aaabD ⇒ aaabd
So S ⇒* aaabd
G
S
a A
a B
a C
b D
d
Note If it is not deterministic to select a single production then that is not LL(1) grammar
Example –
S → abB | aaA
B → d
A → c | d
here w = abd
S
a b B
d
LL(2)
S ⇒ abB ⇒ abd
S ⇒* abd
G

LL(1) Grammar

- A grammar whose parsing table has no multiply-defined entries.

This grammar is LL(1).

E→TE'
E'→+TE'|ε
T→FT'
T'→*FT'|ε
F→(E)|id

FIRST(E)=FIRST(T)=FIRST(F)={(,id}
FIRST(E')={+,ε}
FIRST(T')={*,ε}

FOLLOW(E)=FOLLOW(E')={), $}
FOLLOW(T)=FOLLOW(T')={+,), $}
FOLLOW(F)={+, *,), $}

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

EXAMPLE2:

This grammar is not LL(1).

S → iEtSS' | a
S' → eS | ε
E → b

FIRST(S)={i, a}	FOLLOW(S)={e, $}
FIRST(S')={e, ε}	FOLLOW(S')={e, $}
FIRST(E)={b}	FOLLOW(E)={t}

Multiply-defined entries

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → ε S' → eS			S' → ε
E		E → b				

- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - ✓ **Condition 1**: For no terminal a, do both α and β derive strings beginning with a.

    (FIRST(α) ∩ FIRST(β) = ϕ)
  - ✓ **Condition 2**: At most one of α and β can derive empty string.
  - ✓ **Condition 3**: If $\beta \xrightarrow{*} \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A).

    (FIRST(α) ∩ FOLLOW(A) = ϕ)

This grammar is LL(1).

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

FIRST(S)={a, b}

FIRST(A)={ε}

FIRST(B)={ε}

FOLLOW(S)={$}

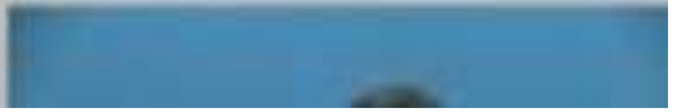
FOLLOW(A)={a, b}

FOLLOW(B)={a, b}

$FIRST(AaAb) \cap FIRST(BbBa) = \phi$

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

Non-terminal	Input Symbol		
	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	



This grammar is not LL(1).

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

FIRST(S)={i, a}	FOLLOW(S)={e, $}
FIRST(S')={e, ε}	FOLLOW(S')={e, $}
FIRST(E)={b}	FOLLOW(E)={t}

1) FIRST(iEtSS') ∩ FIRST(a) = ϕ
{i} ∩ {a} = ϕ

2) FIRST(eS) ∩ FIRST(ε) = ϕ
{e} ∩ {ε} = ϕ

3) FIRST(eS) ∩ FOLLOW(S') = ϕ
{e} ∩ {e, $} ≠ ϕ

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S→ a			S→ iEtSS'		
S'			S' → ε S' →eS			S' →ε
E		E → b				

Check following grammar for LL(1):

$S \rightarrow CC$

$C \rightarrow cC \mid d$

$FIRST(cC) \cap FIRST(d) = \phi$

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

This grammar is LL(1).

$S \rightarrow CC$

$C \rightarrow cC \mid d$

FIRST(S)={c, d}	FOLLOW(S)={\$}
FIRST(C)={c, d}	FOLLOW(C)={c, d, \$}

Non-terminal	Input Symbol		
	c	d	\$
S	$S \rightarrow CC$	$S \rightarrow CC$	
C	$C \rightarrow cC$	$C \rightarrow d$	

SR PARSER

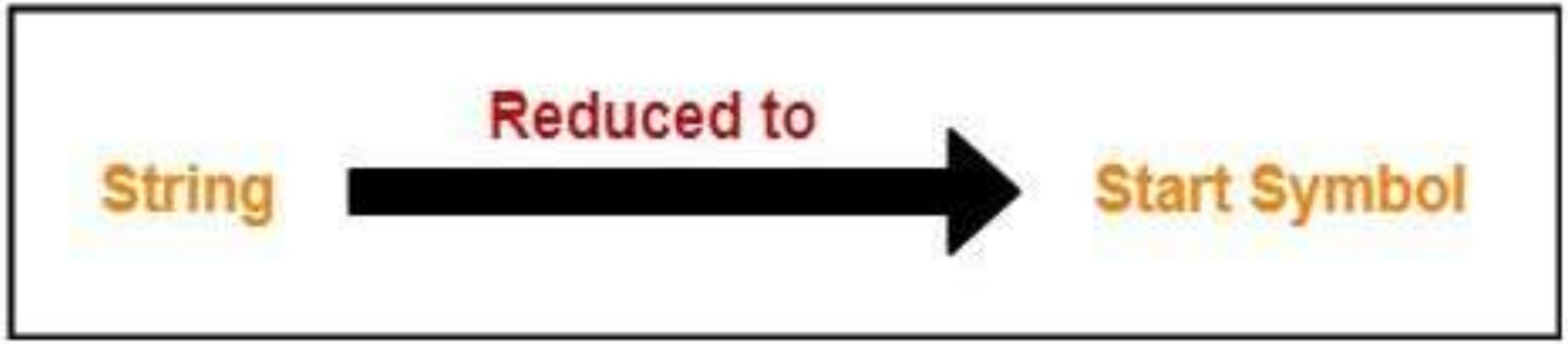
Bottom up parsing

- Bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- It uses rightmost derivation to construct the parse tree.
- The rightmost derivation is the one in which we always expand the rightmost non-terminal.

STACK IMPLEMENTATION OF SR PARSER

It takes the given input string and builds a parse tree-

- Starting from the bottom at the leaves.
- And growing the tree towards the top to the root.



Data Structures-

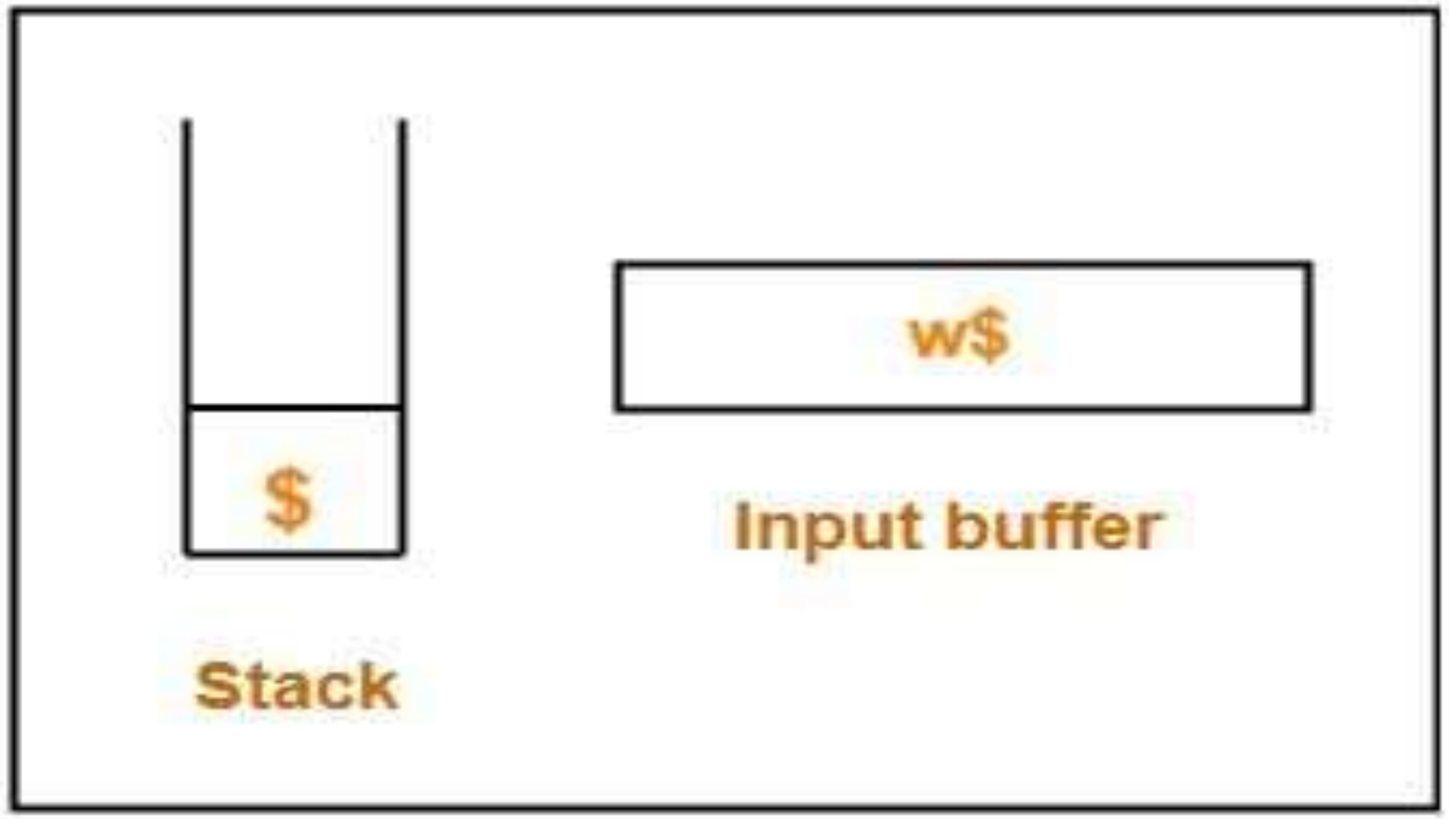
Two data structures are required to implement a shift-reduce parser-

- A **Stack** is required to hold the grammar symbols.
- An **Input buffer** is required to hold the string to be parsed.

Working-

Initially, shift-reduce parser is present in the following configuration where-

- Stack contains only the $ symbol.
- Input buffer contains the input string with $ at its end.



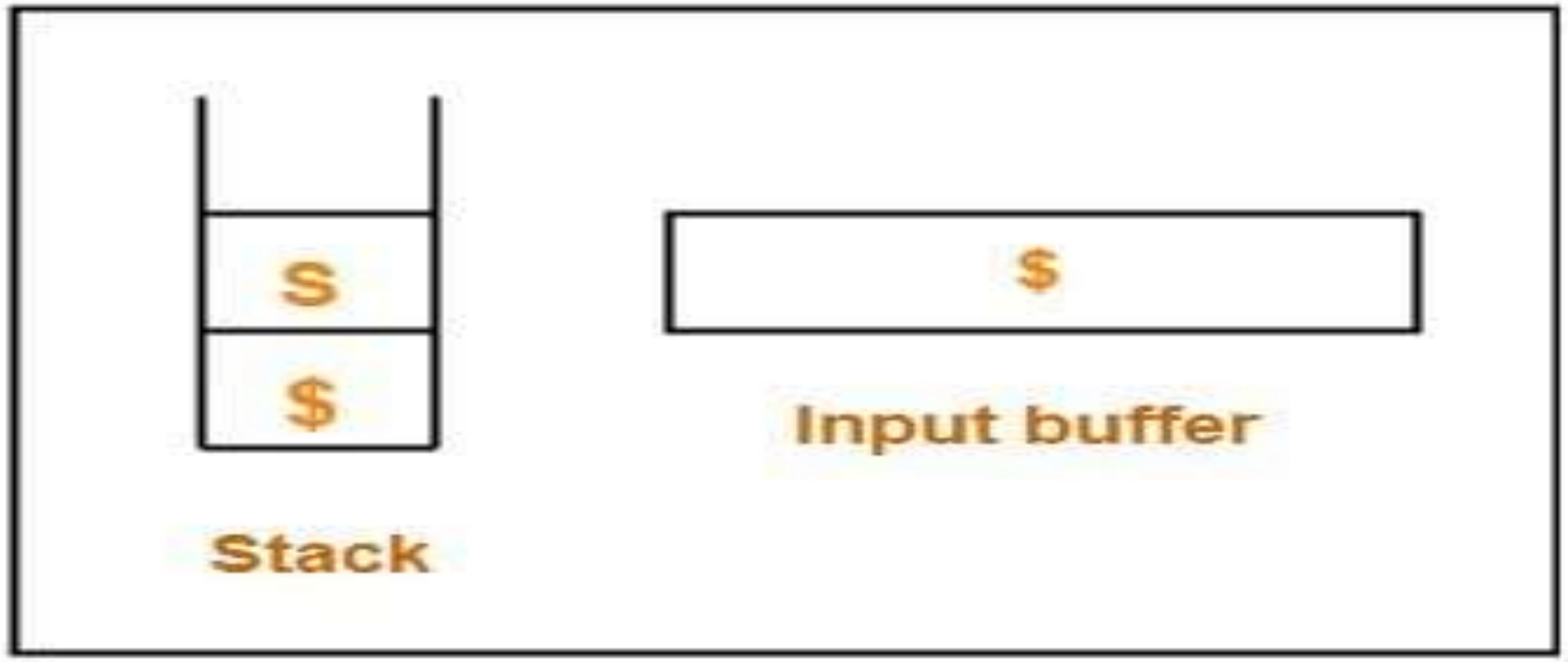
Initial Configuration

The parser works by-

- Moving the input symbols on the top of the stack.
- Until a handle β appears on the top of the stack.

The parser keeps on repeating this cycle until-

- An error is detected.
- Or stack is left with only the start symbol and the input buffer becomes empty.



Final Configuration

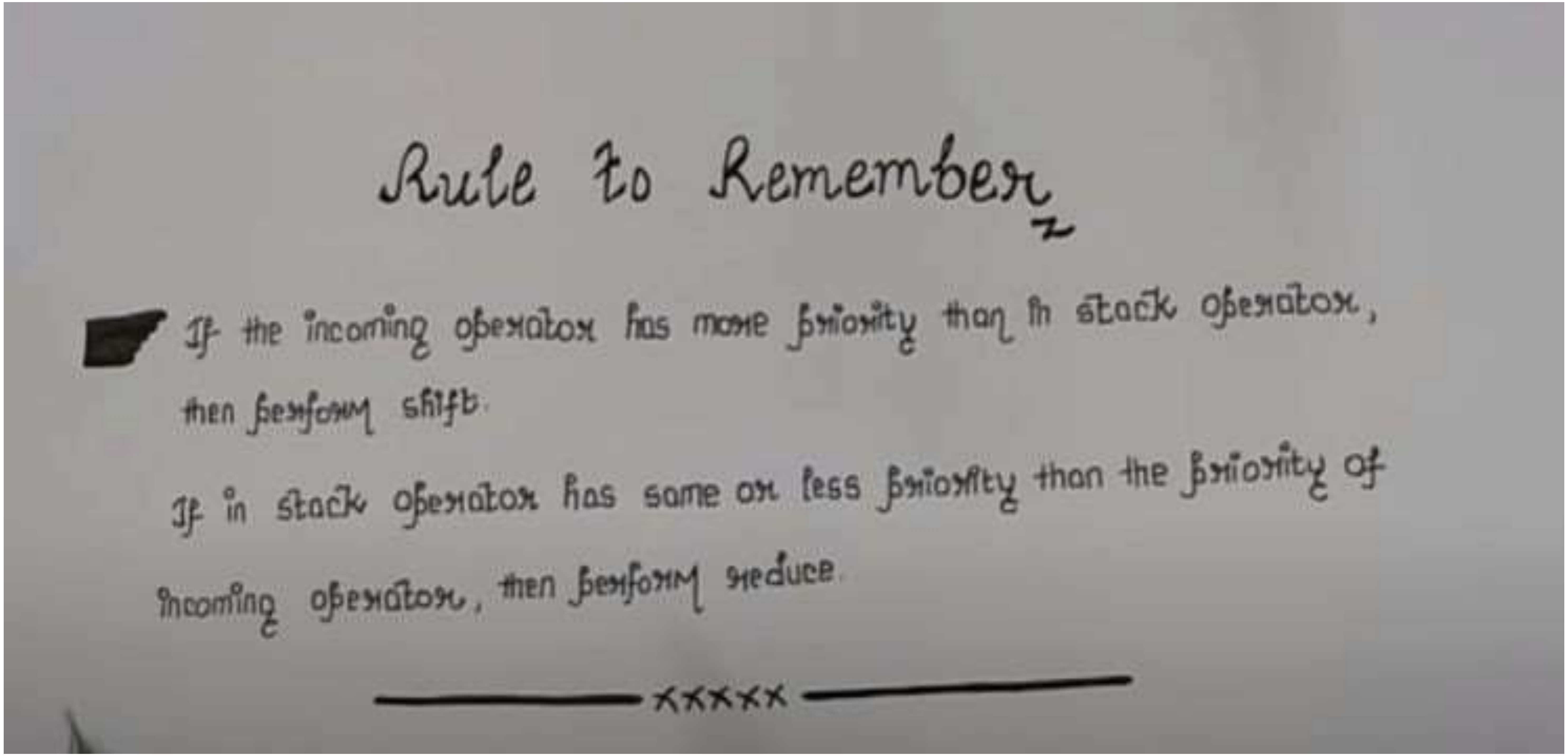
After achieving this configuration,

- The parser stops / halts.
- It reports the successful completion of parsing.

ACTION PERFORMED BY THE SR PARSER

1. In a shift action, the next input symbol is shifted onto the top of the stack.
2. In a reduce action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.
3. In an *accept* action, the parser announces successful completion of parsing.
4. In an *error* action, the parser discovers that syntax error has occurred and calls an error recovery routine.

RULE FOR SHIFT/REDUCE



Rule to Remember

- If the incoming operator has more priority than in stack operator, then perform shift.

If in stack operator has same or less priority than the priority of incoming operator, then perform reduce.

xxxxx

- Example:

E → E+T | T

T → T*F | F

F → id

Input string is id * id

Rightmost derivation

E => T

=> T * F

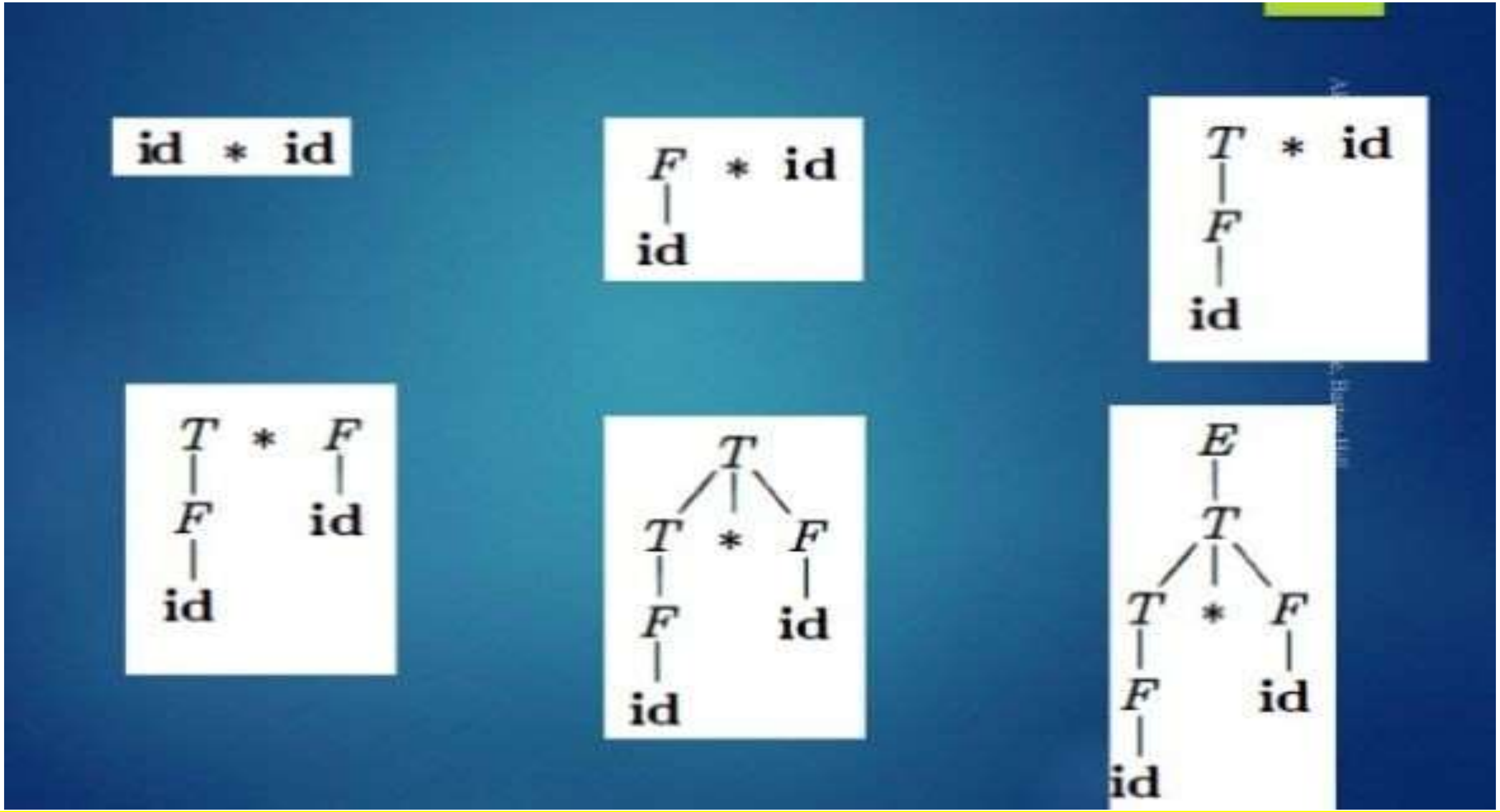
=> T * **id**

=> F * **id**

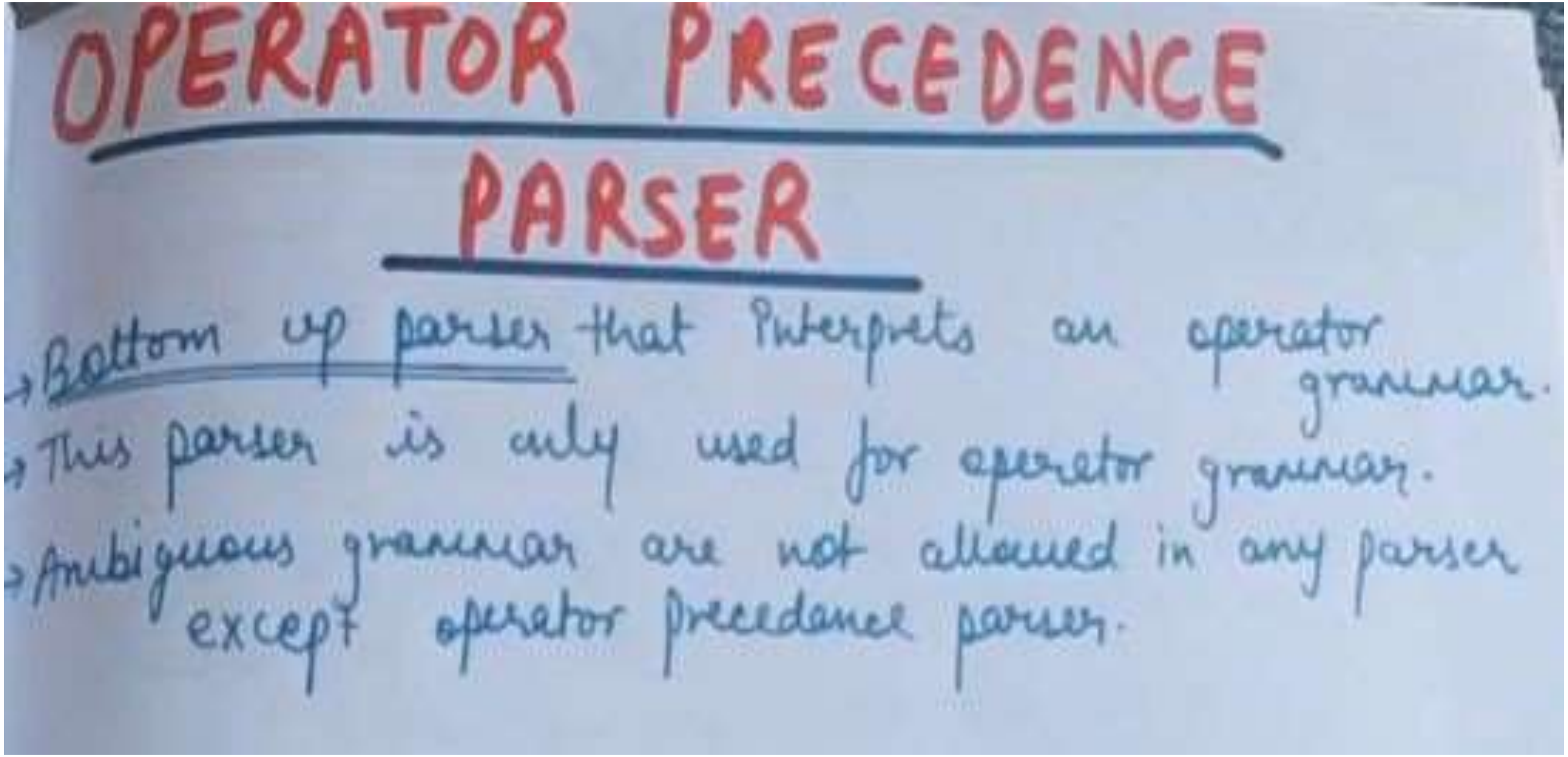
=> **id** * **id**

STACK	INPUT	ACTION
$	id * id $	Shift
$ id	* id $	Reduce by F → id
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce by F → id
$ T * F	$	Reduce by T → T * F
$ T	$	Reduce by E → T
$ E	$	Accept

PARSE TREE GENERATED BY SR PARSER



OPERATOR PRECEDENCE PARSER



OPERATOR PRECEDENCE PARSER

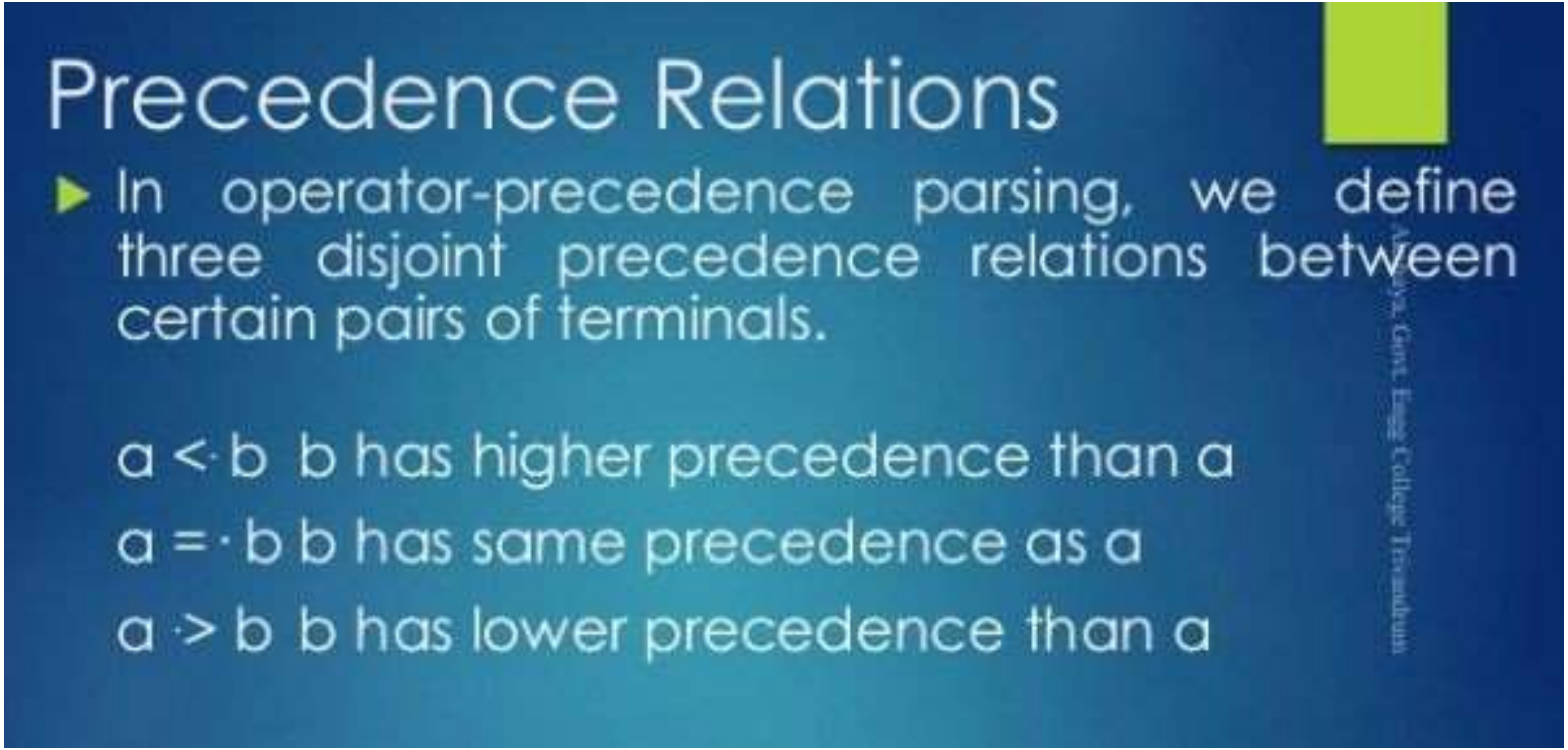
- Bottom up parser that interprets an operator grammar.
- This parser is only used for operator grammar.
- Ambiguous grammar are not allowed in any parser except operator precedence parser.

OPERATOR PRECEDENCE PARSER

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.
- Ex:

E→AB	E→EOE	E→E+E |
A→a	E→id	E*E |
B→b	O→+ | * | /	E/E | id
not operator grammar	not operator grammar	operator grammar

OPERATOR PRECEDENCE PARSER



How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator θ_1 has higher precedence than operator θ_2,
   → $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$
2. If operator θ_1 and operator θ_2 have equal precedence,
   they are left-associative → $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$
   they are right-associative → $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$
3. For all operators θ, $\theta \lessdot id$, $id \gtrdot \theta$, $\theta \lessdot ($, $(\lessdot \theta$, $\theta \gtrdot)$, $) \gtrdot \theta$, $\theta \gtrdot \$$, and $\$ \lessdot \theta$
4. Also, let

$(\doteq)$	$\$ \lessdot ($	$id \gtrdot)$	$) \gtrdot \$$
$(\lessdot ($	$\$ \lessdot id$	$id \gtrdot \$$	$) \gtrdot)$

Operator Precedence Parsing

- ✓ Consider the Grammar

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-

Operator-precedence relations

Precedence:
id ·> * ·> + ·> $

+ ·> + (Left Associative)
Example: id + id + id

Operator-Precedence Parsing Algorithm

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

```
set p to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and p points to $ ) then return
   else {
      let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
      if ( a <· b or a =· b ) then { /* SHIFT */
         push b onto the stack;
         advance p to the next input symbol;
      }
      else if ( a ·> b ) then           /* REDUCE */
         repeat pop stack
         until ( the top of stack terminal is related by <· to the terminal most recently popped );
      else error();
   }
```

Operator-Precedence Parsing Algorithm -- Example

stack	input	action
$	id+id*id$	$ < id shift
$id	+id*id$	id > + reduceE → id
$	+id*id$	shift
$+	id*id$	shift
$+id	*id$	id > * reduceE → id
$+	*id$	shift
$+*	id$	shift
$+*id	$id > $	reduceE → id
$+*	$* > $	reduceE → E*E
$+	$+ > $	reduceE → E+E
$	$	accept

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

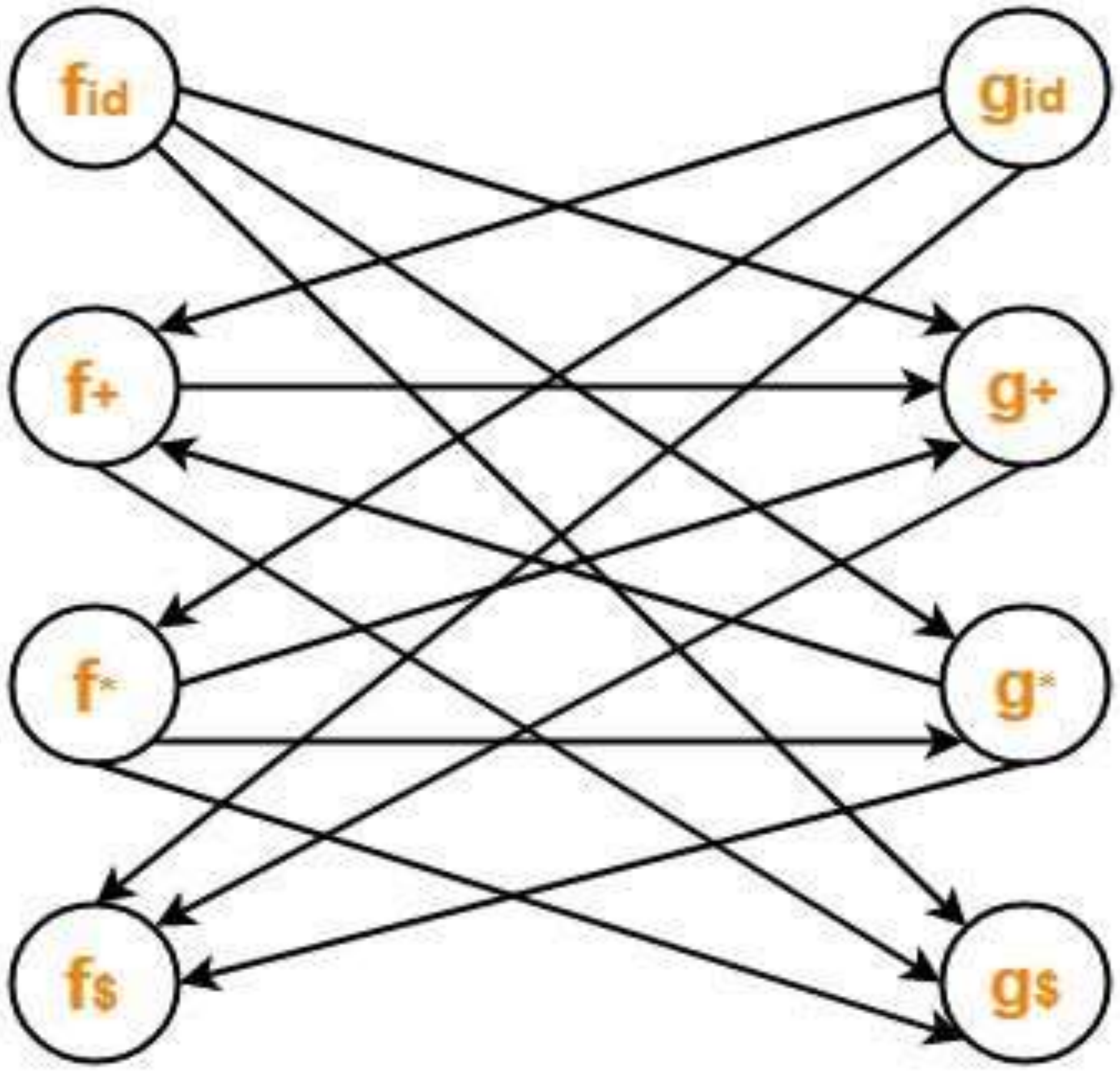
OPERATOR FUNCTION

We construct the operator precedence table as-

	g →				
f ↓		**id**	**+**	**x**	**$**
	id		>	>	>
	+	<	>	<	>
	x	<	>	>	>
	$	<	<	<	

Operator Precedence Table

The graph representing the precedence functions is-



Graph Representing Precedence Functions

Here, the longest paths are-

- $f_{id} \longrightarrow g_{\times} \longrightarrow f_{+} \longrightarrow g_{+} \longrightarrow f_{\$}$
- $g_{id} \longrightarrow f_{\times} \longrightarrow g_{\times} \longrightarrow f_{+} \longrightarrow g_{+} \longrightarrow f_{\$}$

The resulting precedence functions are-

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

LR PARSER

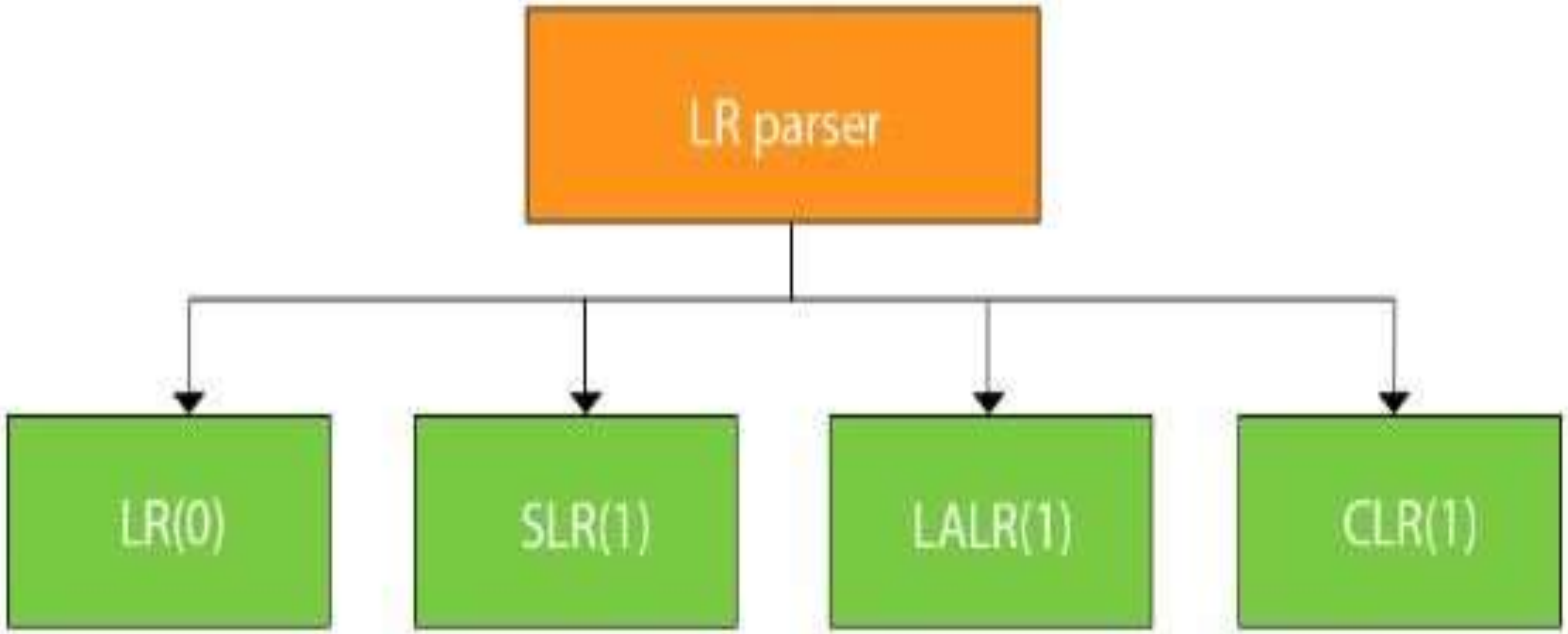


Fig: Types of LR parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.

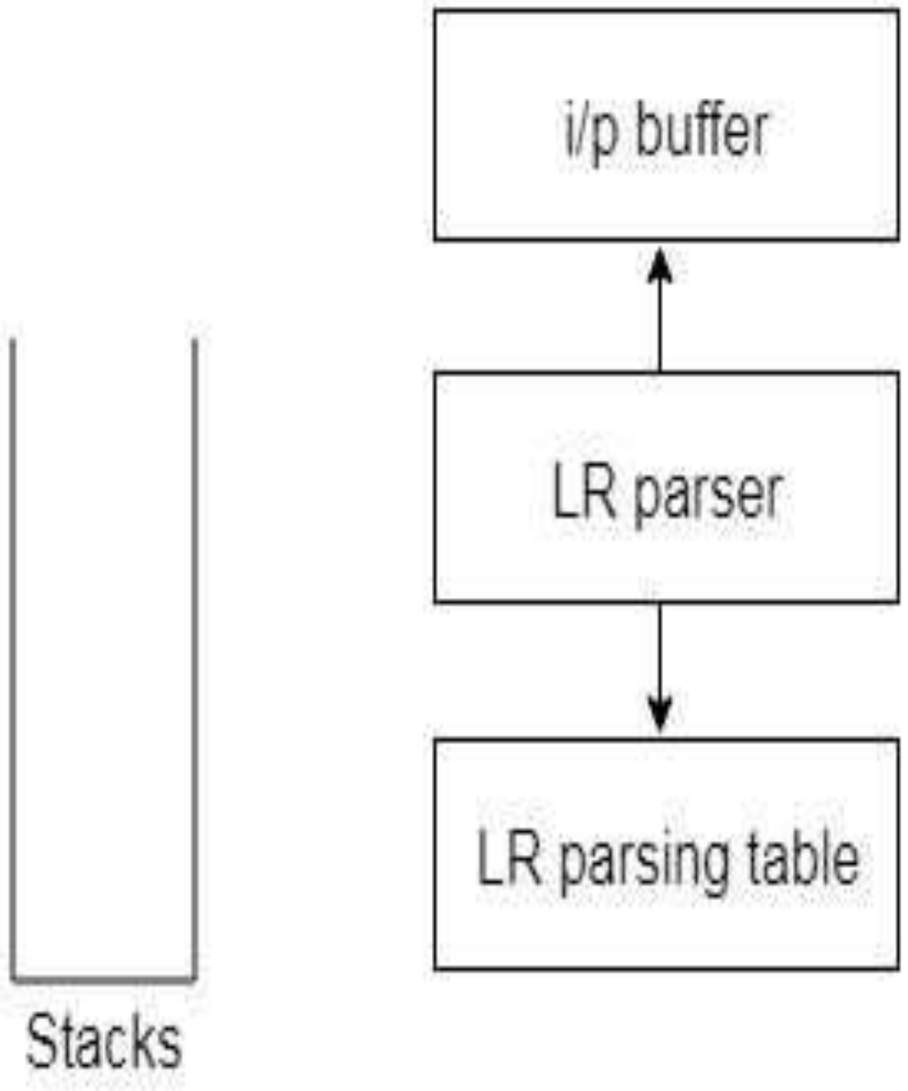
"R" stands for constructing a right most derivation in reverse.

"K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

LR algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

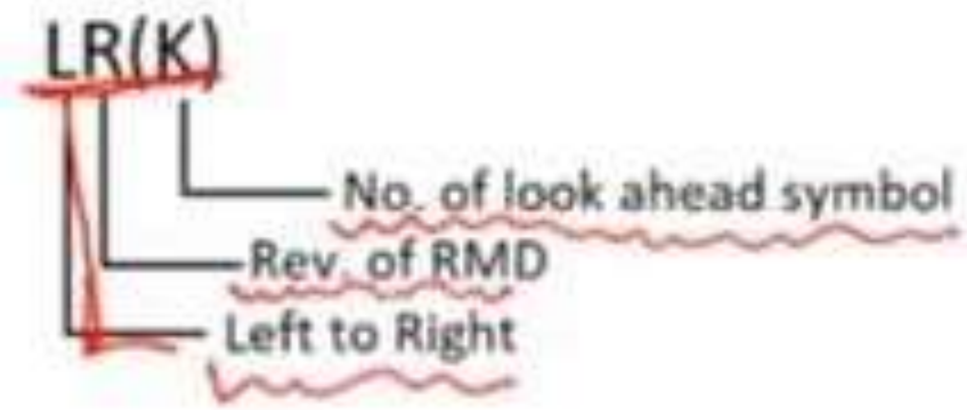


Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.

A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.

Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

LR Parser



Classification of LR Parser :- Based on the construction of the parse table LR parser are classified into 4 types

1. LR(0)
2. SLR(1)

} LR(0) item

3. CLR (1)
4. LALR (1)

} LR(1) item

Procedure to construct the LR Parse table

1. Obtain the Augmented grammar for then given Grammar.
2. Create the canonical collection of LR items.
3. Draw the DFA and prepare the parse table based on LR items.

Augmented Grammar: The grammar which is obtained by adding one more production that derives the start symbol of the grammar is called as Augmented Grammar.

ex

Grammar

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

Augmented Grammar

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

- Augmented grammar helps us in separating the final item from the non final items.
- The need of Augmented grammar is to decide for the final item when ever a multiple production are there from the start symbol.

LR (0) items: The production with (.) any where on R.H.S. is know as LR(0) items.

Ex – $A \Rightarrow abc$

LR(0) items:

- $A \to .abc$ } Non final
- $A \to a.bc$ } Non final
- $A \to ab.c$ } Non final
- $A \to abc.$ } final item

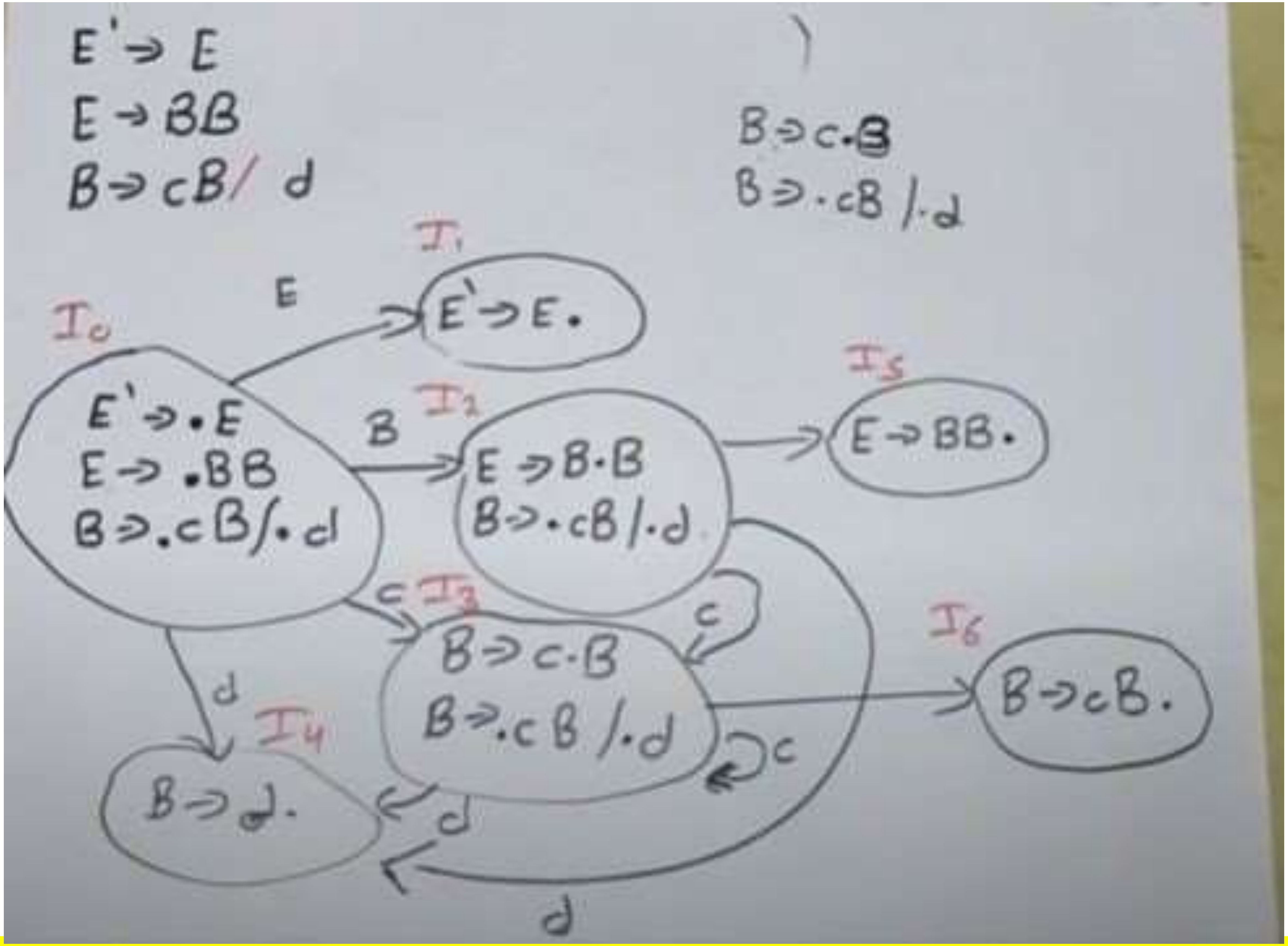
Stack	i/p	Action
\$	.abab\$	push
\$a	a.bab\$	
\$ab	ab.ab\$	

Canonical collection :- If I_0, I_1, I_2 ...... I_n be the LR(0) item then the set $C = \{I_0, I_1, I_2 .... I_n\}$ is called as canonical collection.

Function used to get LR(0) items

1. Closure (I)
2. Go to (I, X)

GATE


E' → E
E → BB
B → cB / d
B → c.B
B → .cB / .d
I1
E
E' → E.
I0
E' → .E
E → .BB
B → .cB / .d
B
I2
E → B.B
B → .cB / .d
I5
E → BB.
c
I3
B → c.B
B → .cB / .d
c
I6
B → cB.
d
I4
B → d.
c
d
d

Construction of LR(0) Parser table: Parse table consist of two parts

1. Action
2. GO-TO

	ACTION		GOTO
	Terminals	$	Non terminals
I_0			
I_1			
I_2			
I_3			
⋮			
I_n			

$m \times n$

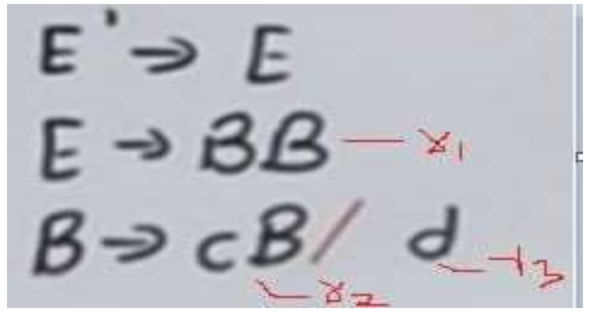
1. Goto (I, X) = $S_j(I_i)$

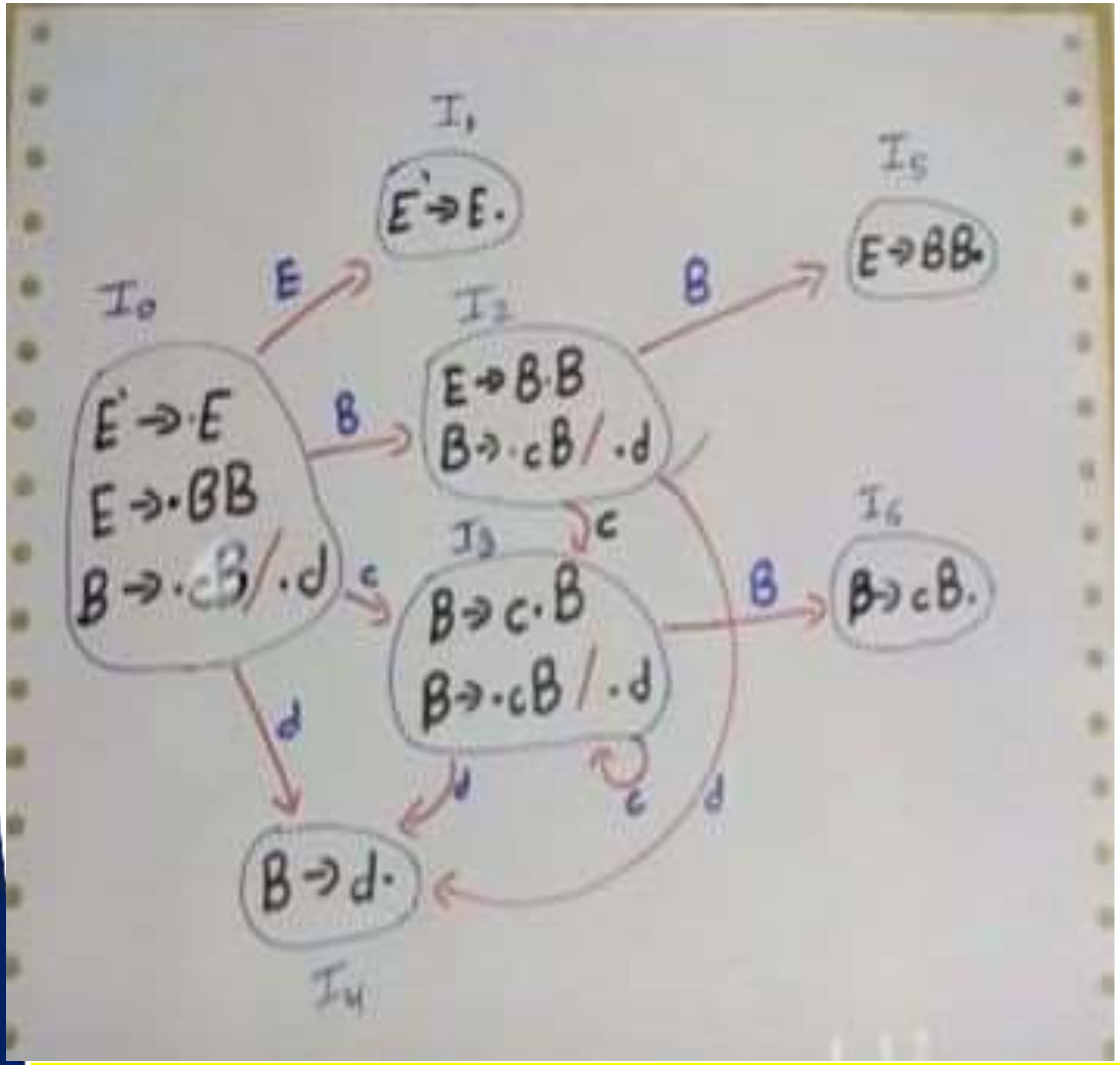
   X → terminal

2. Goto (I, X) = $J(I_j)$

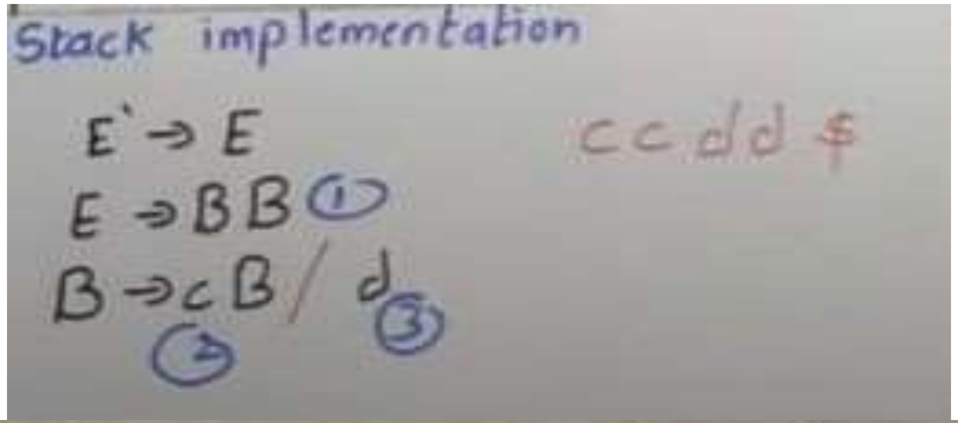
   X → Non terminal

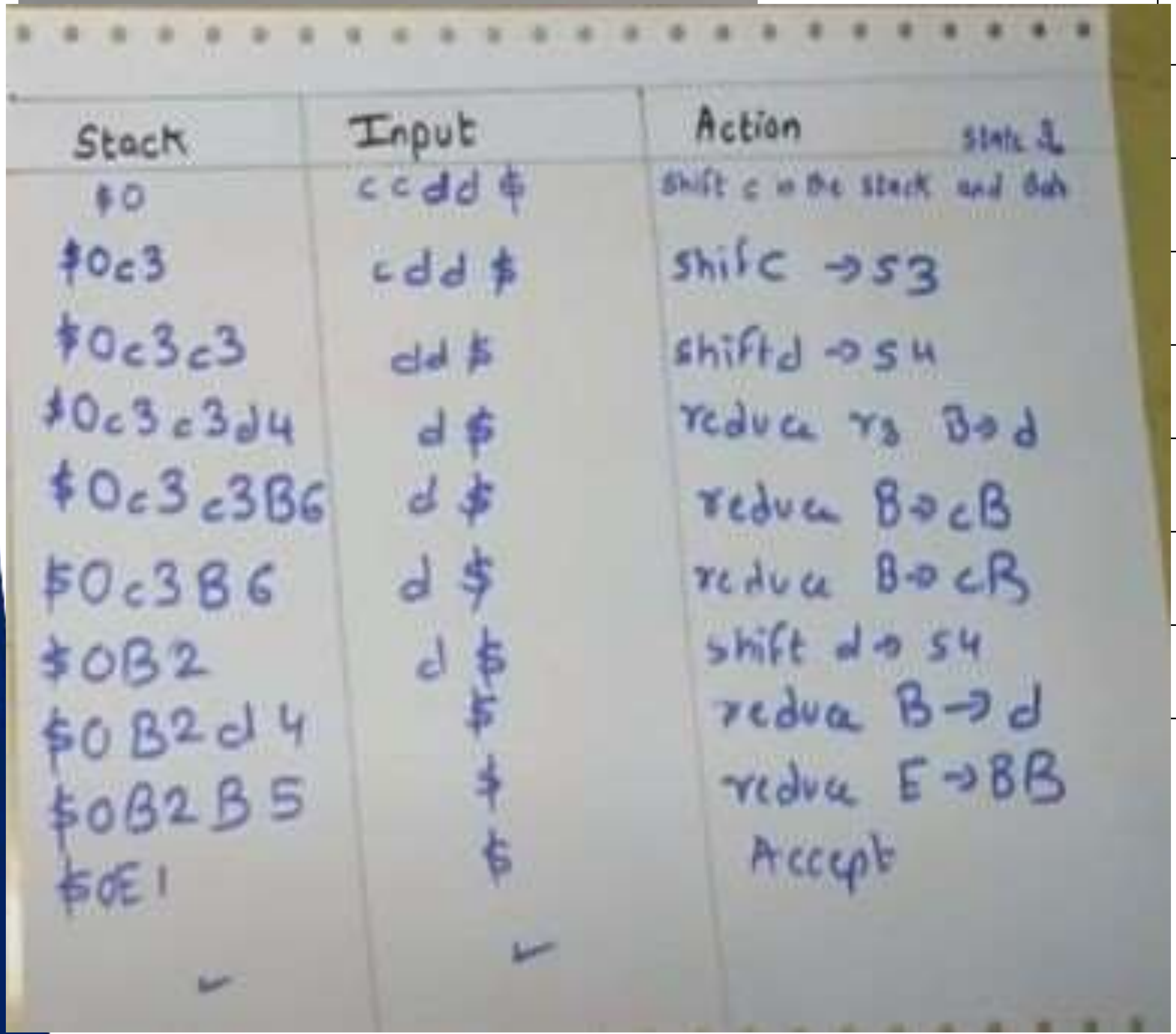
3. If I_i is the final item containing r_i rule of grammar then place r_i under all terminal.





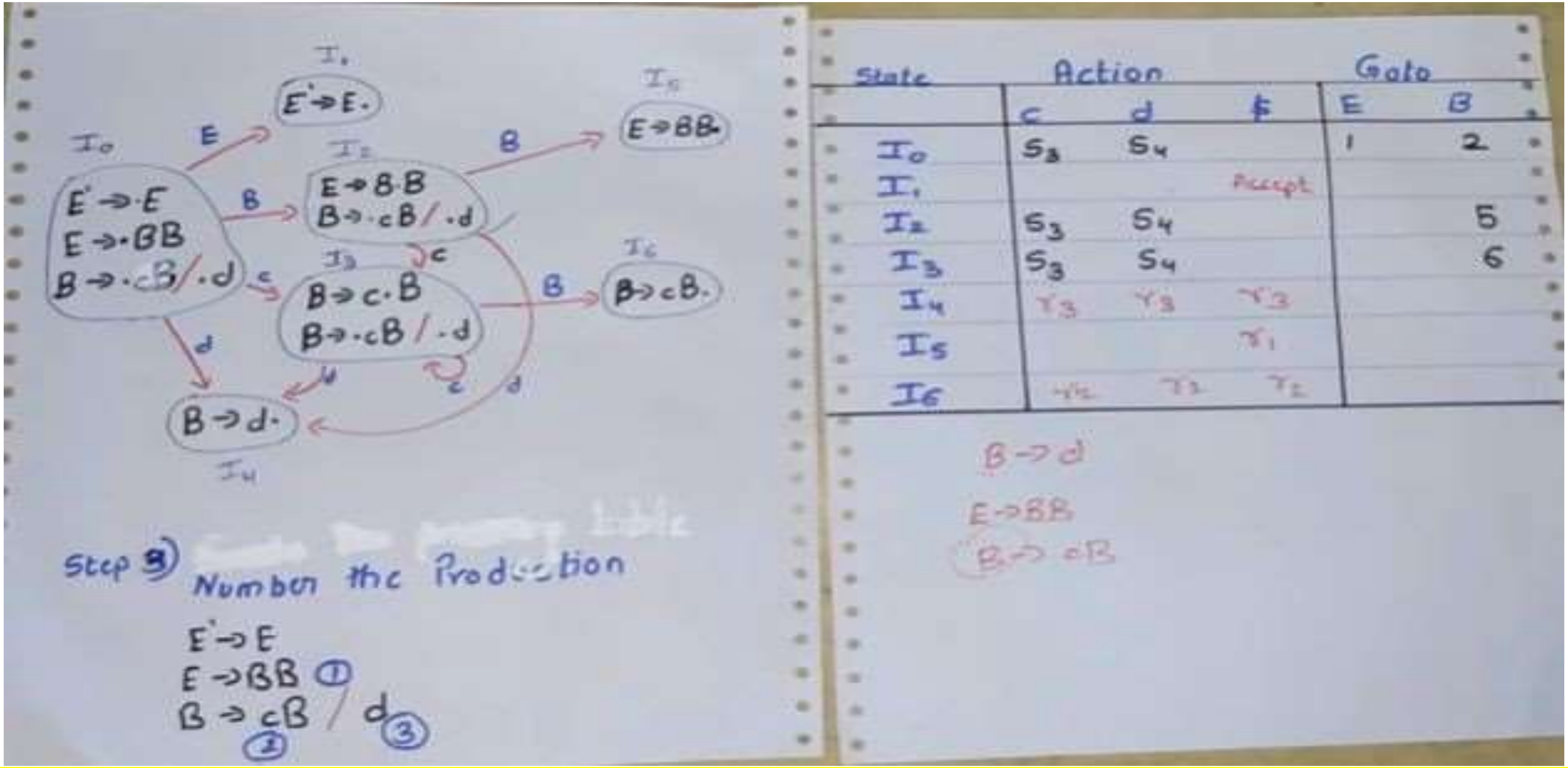
STATES	ACTION			GOTO	
	c	d	$	E	B
I0	S3	S4		1	2
I1			Accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		





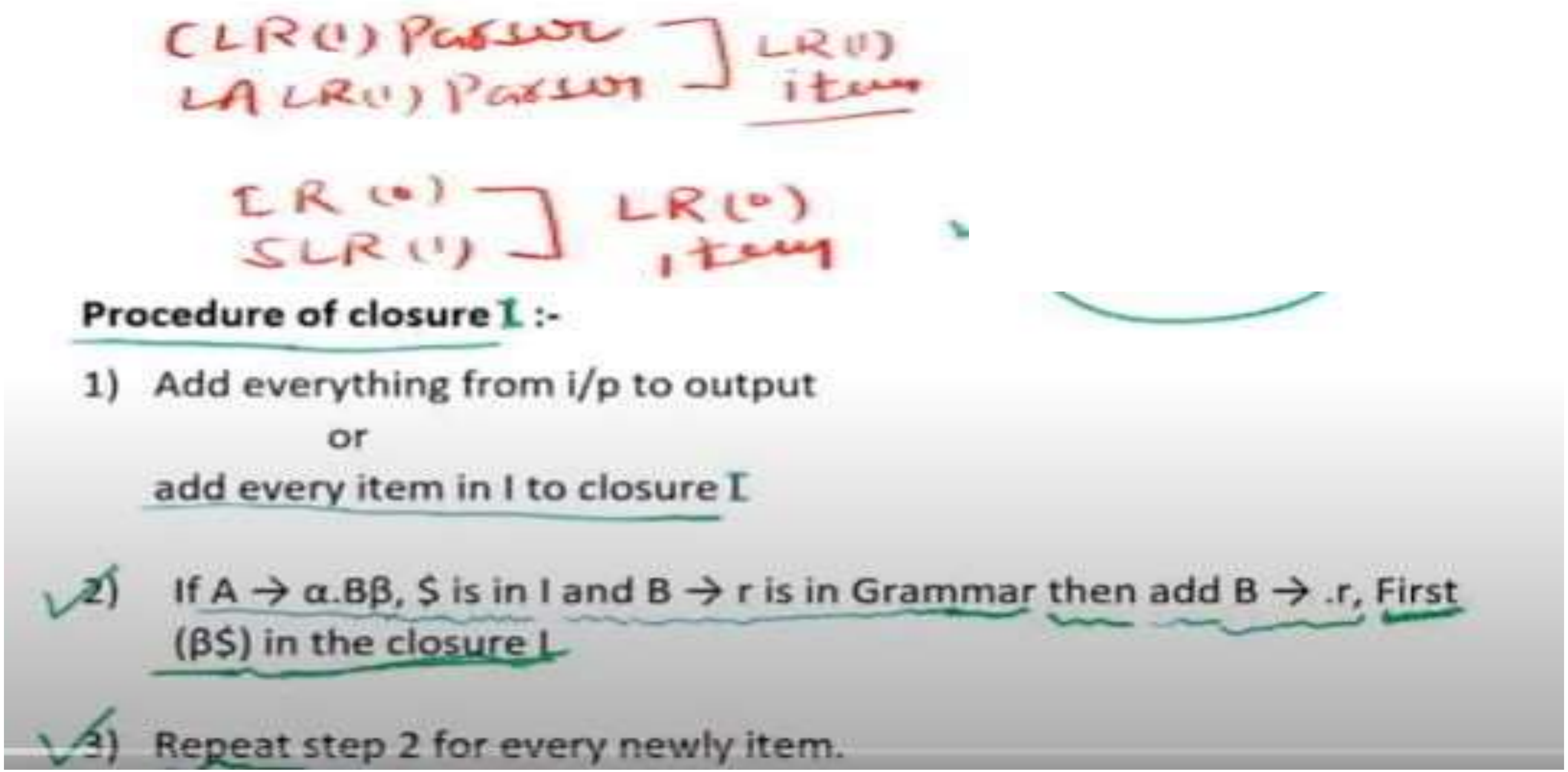
STATES	ACTION			GOTO	
	c	d	$	E	B
I0	S3	S4		1	2
I1			Accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

SLR(1)



State	Action c	Action d	Action $	Goto E	Goto B
I0	S3	S4		1	2
I1			Accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5			r1		
I6	r2	r2	r2		

CLR(1)



CLR(1) Parser] LR(1) item
LALR(1) Parser]

LR(0)] LR(0) item
SLR(1)]

Procedure of closure I :-

1) Add everything from i/p to output

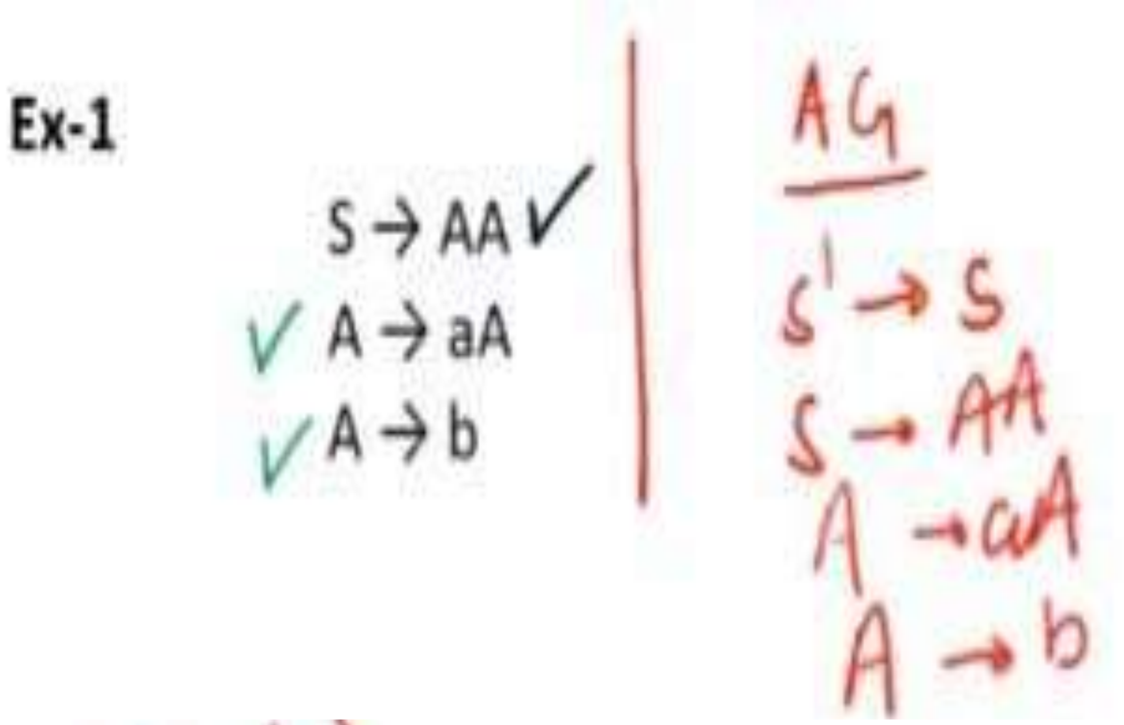
   or

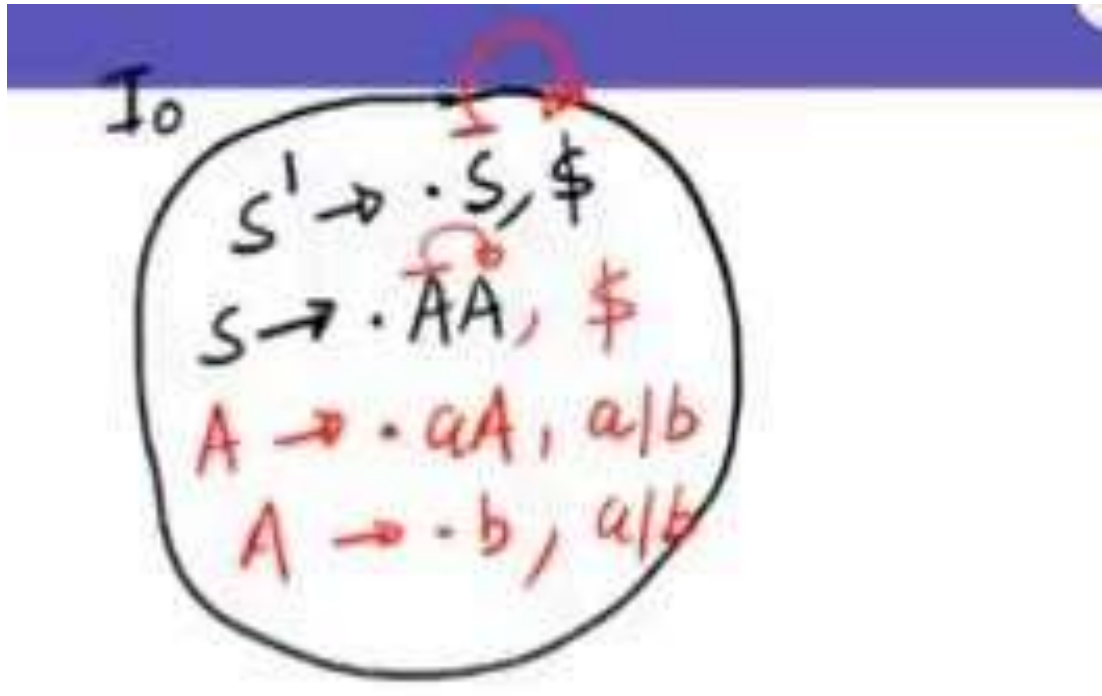
   add every item in I to closure I

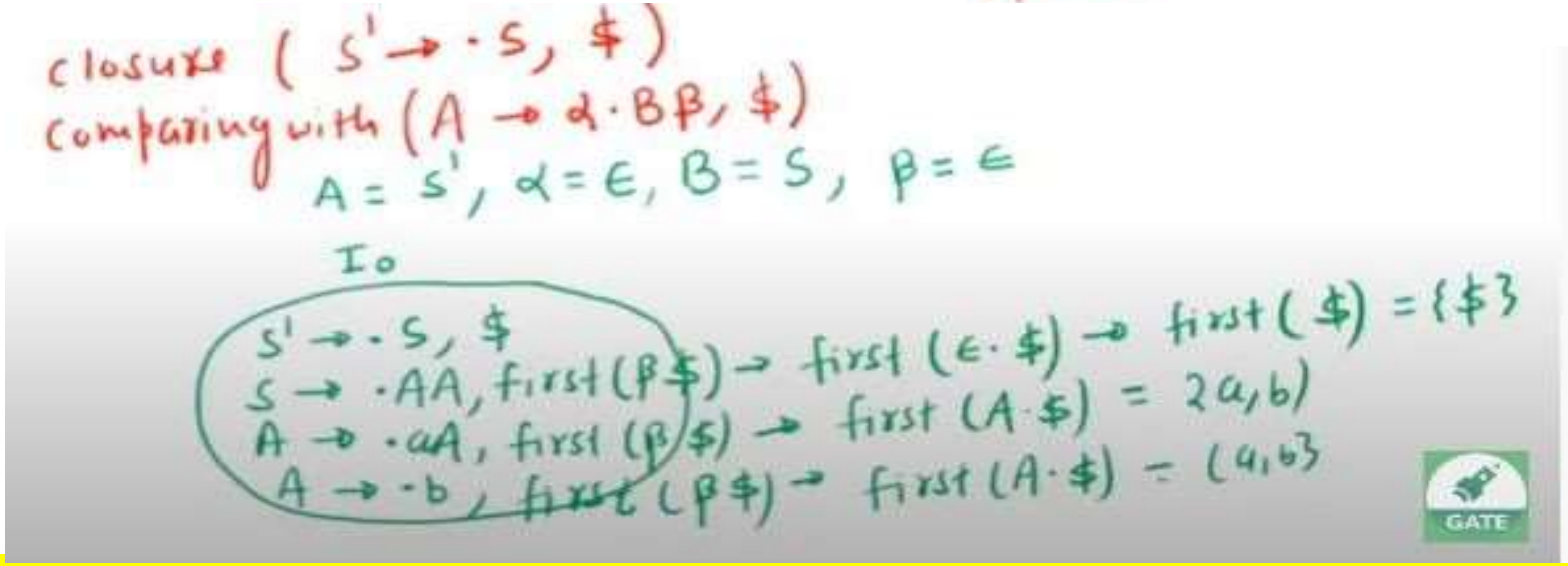
2) If A → α.Bβ, $ is in I and B → r is in Grammar then add B → .r, First (β$) in the closure I

3) Repeat step 2 for every newly item.

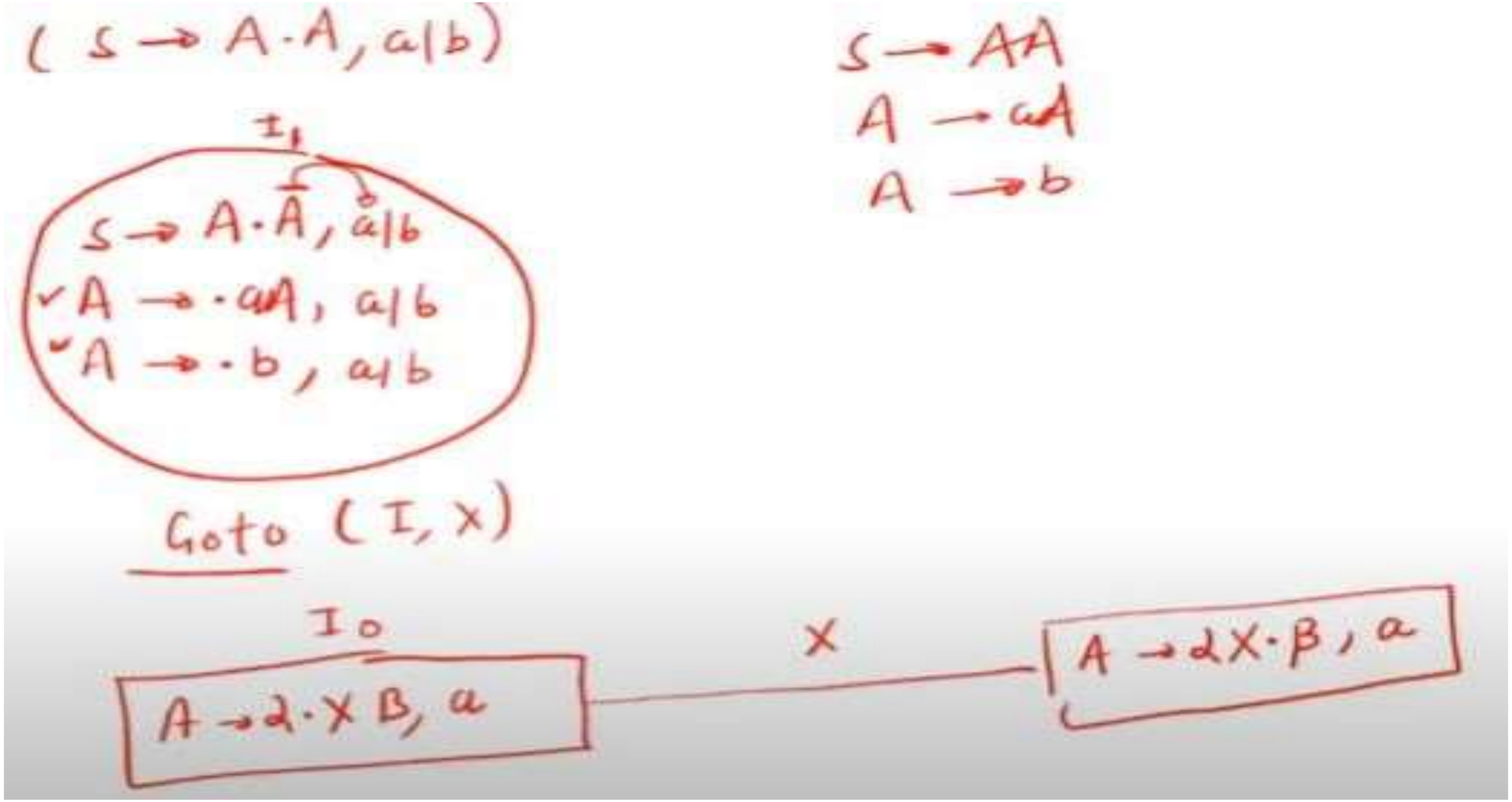
CLR(1)


Ex-1
S → AA ✓
✓ A → aA
✓ A → b
AG
S' → S
S → AA
A → aA
A → b

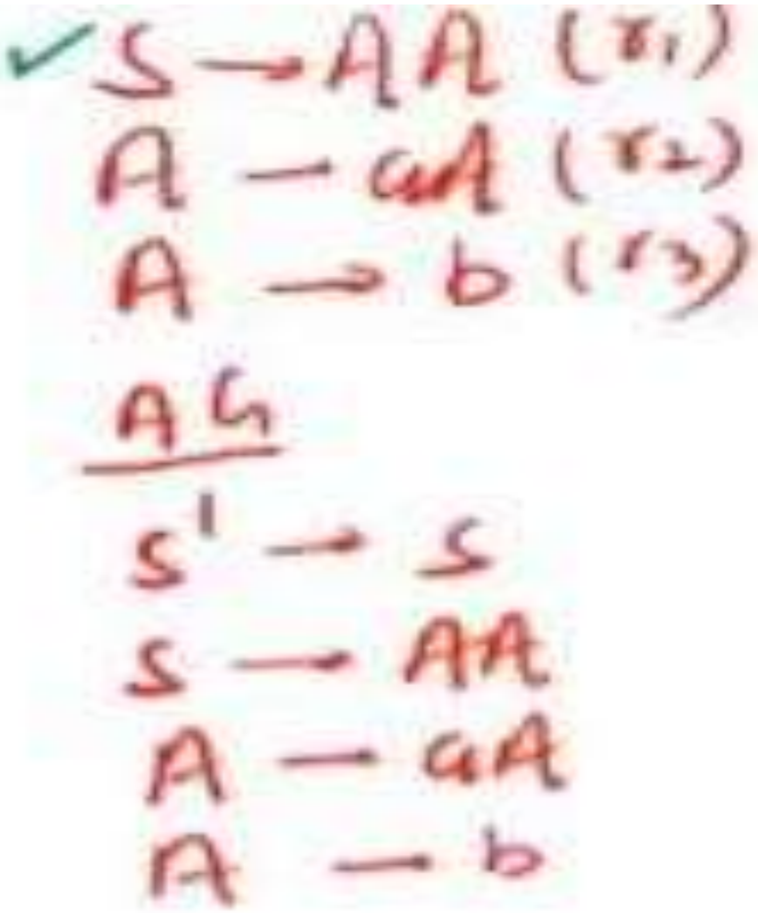

I0
S' → ·S, $
S → ·AA, $
A → ·aA, a/b
A → ·b, a/b


closure (S' → ·S, $)
comparing with (A → α·Bβ, $)
A = S', α = ε, B = S, β = ε
I0
S' → ·S, $
S → ·AA, first(β$) → first (ε·$) → first($) = {$}
A → ·aA, first (β$) → first (A·$) = {a,b}
A → ·b, first (β$) → first (A·$) = {a,b}
GATE

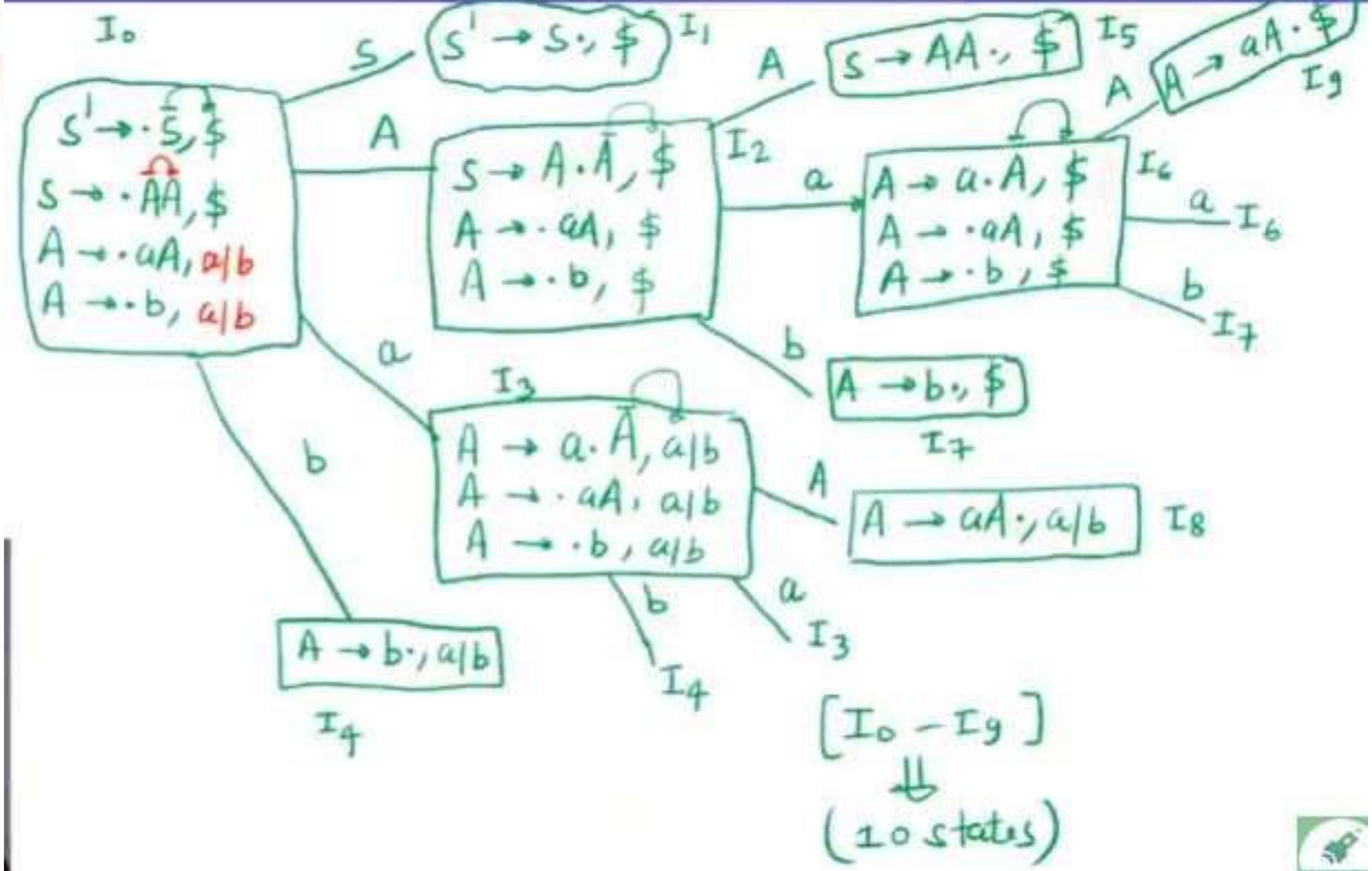
CLR(1)



CLR (1) Parser



S → AA (r1)
A → aA (r2)
A → b (r3)
AG
S' → S
S → AA
A → aA
A → b



I0
S' → ·S, $
S → ·AA, $
A → ·aA, a/b
A → ·b, a/b
I1
S' → S·, $
I2
S → A·A, $
A → ·aA, $
A → ·b, $
I3
A → a·A, a/b
A → ·aA, a/b
A → ·b, a/b
I4
A → b·, a/b
I5
S → AA·, $
I6
A → a·A, $
A → ·aA, $
A → ·b, $
I7
A → b·, $
I8
A → aA·, a/b
I9
A → aA·, $
[I0 – I9]
(10 states)

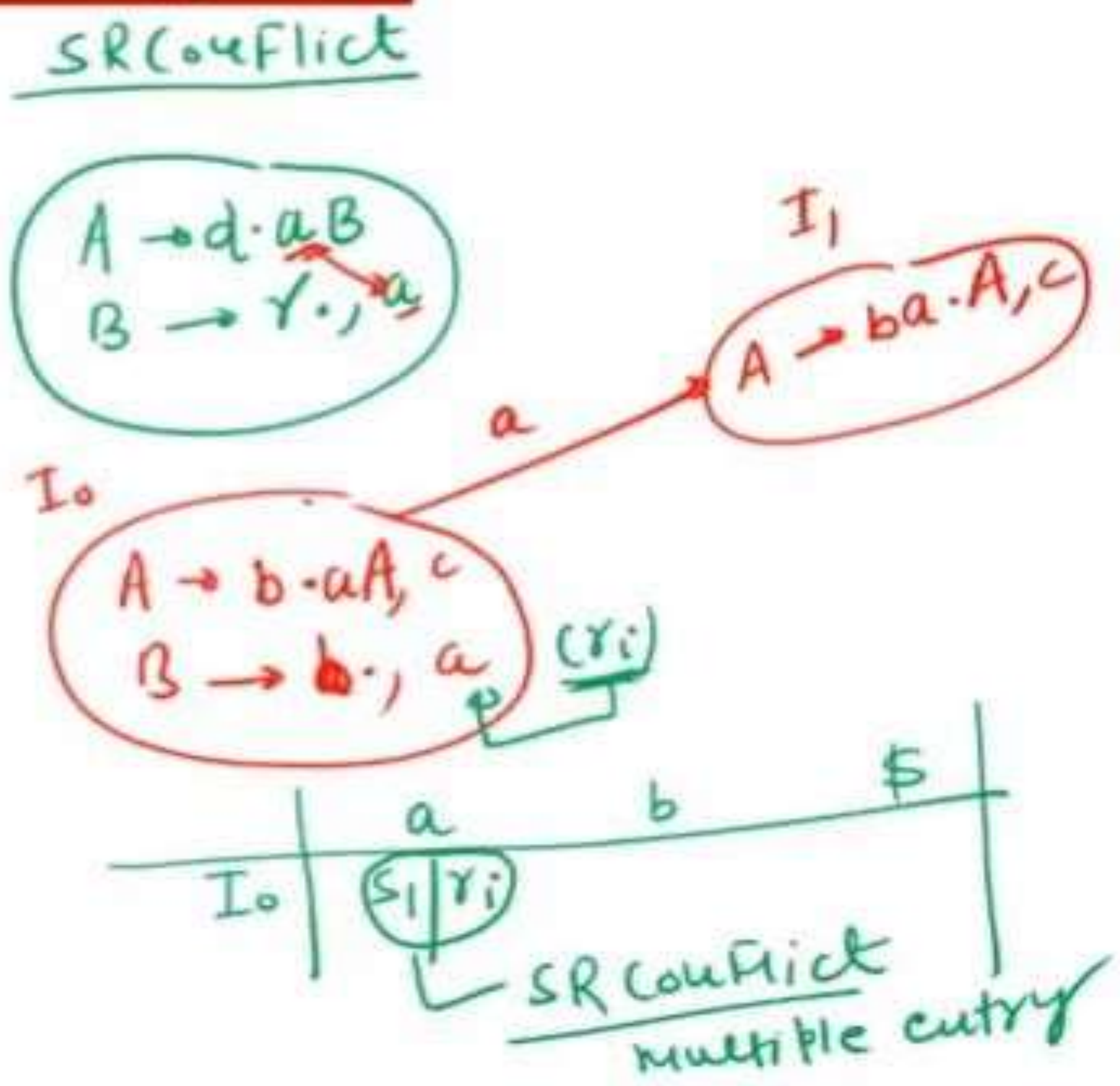
PARSE TABLE

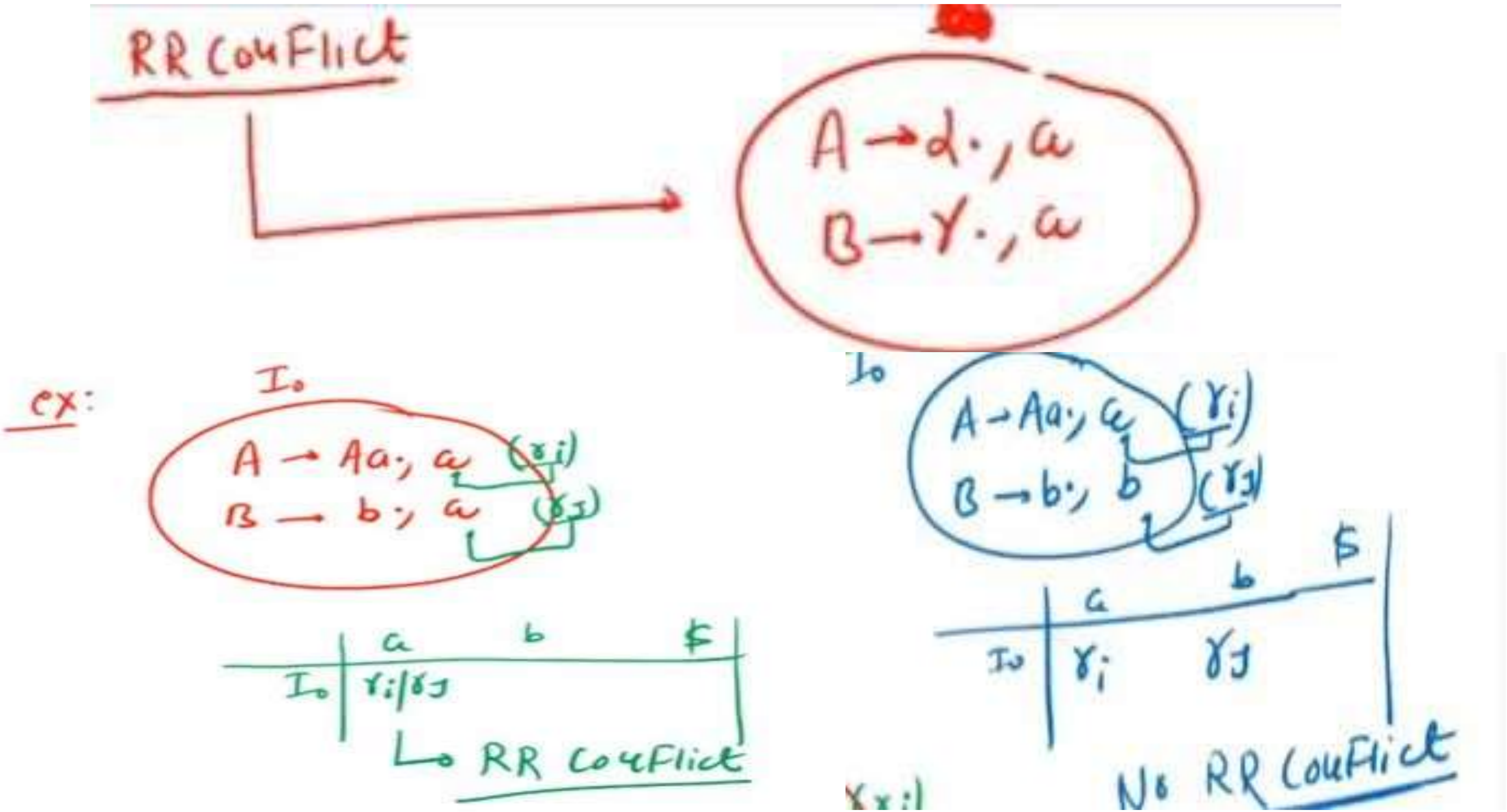
	a	b	$	S	A
I_0	s3	s4		1	2
I_1			Accept		
I_2	s6	s7			5
I_3	s3	s4			8
I_4	r3	r3			
I_5			r1		
I_6	s6	s7			9
I_7			r3		
I_8	r2	r2			
I_9			r2		

LALR (1) Parser

Conflict in CLR(1) Parser

1. SR conflict
2. SR conflict




RR conflict
A → α· , a
B → γ· , a
ex:
I₀
A → Aa· , a (γi)
B → b· , a (γj)
a b $
I₀ γi/γj
RR conflict
I₀
A → Aa· , a (γi)
B → b· , b (γj)
a b $
I₀ γi γj
No RR conflict

1. For every grammar if SLR(1) parser can be constructed then LR(1) Parser can also be constructed. But if LR(1) Parser can be constructed for a grammar then SLR(1) Parser may or may not be constructed.

   i.e. every SLR(1) grammar is CLR(1) but every CLR(1) grammar need not be SLR(1).

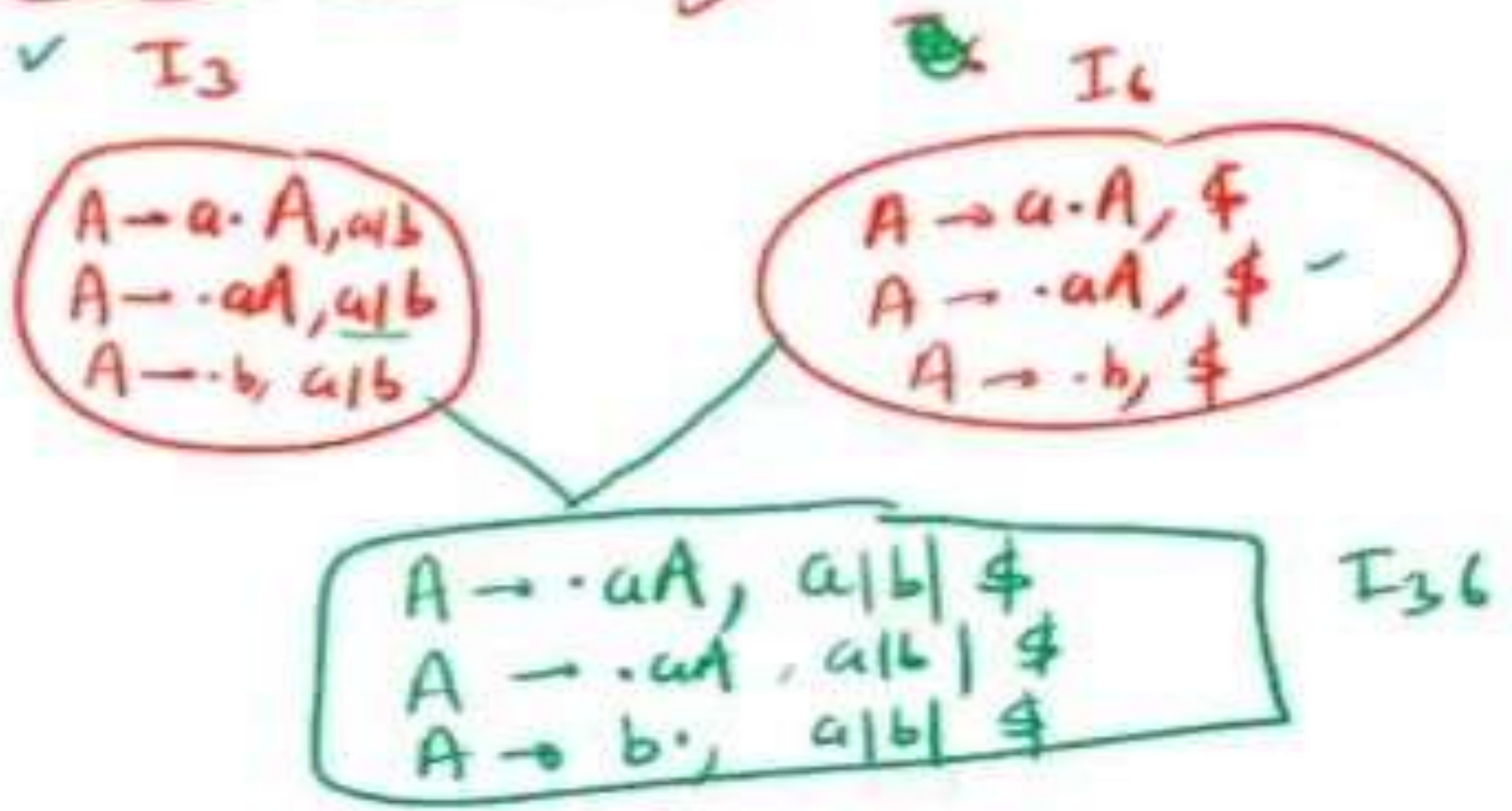
2. CLR(1) Parser is more powerful than any other parser and more costly also.

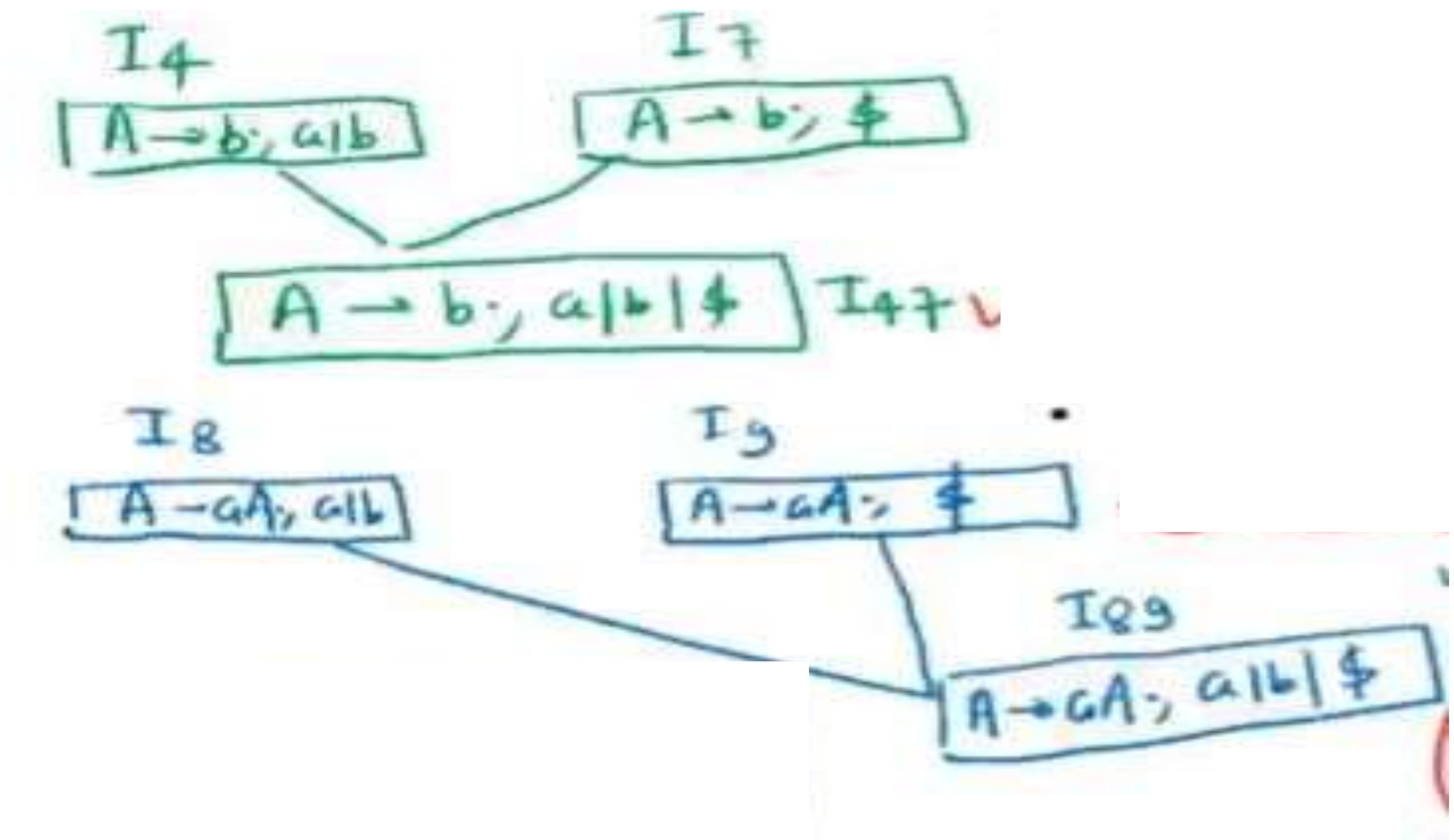
3. Medium level company will not prefer this kind of mechanism to construct the parser in compiler projects.

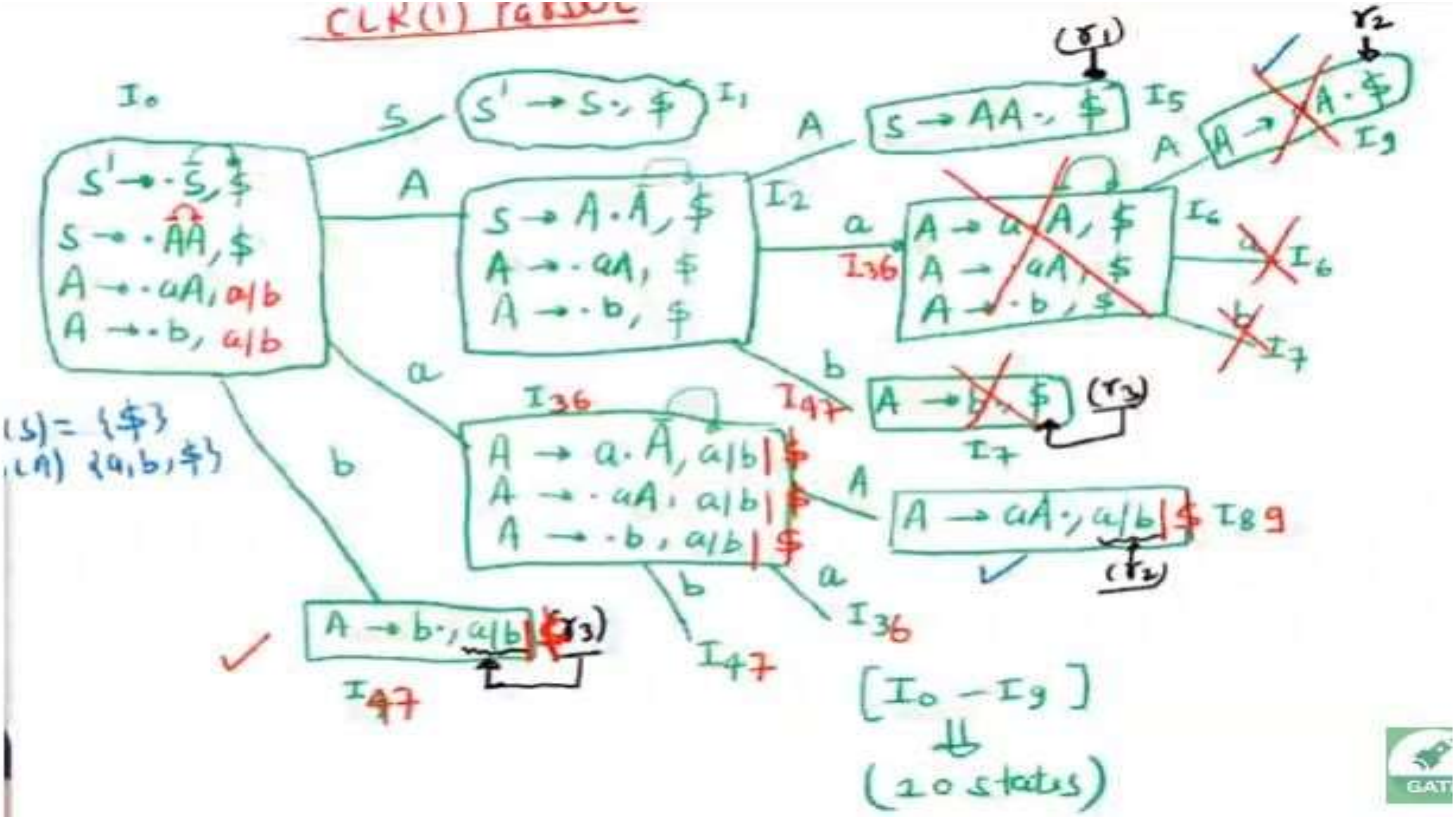
LALR(1) Parser:-

The DFA of CLR(1) contain some state which contain the item with same production part and different look ahead part.

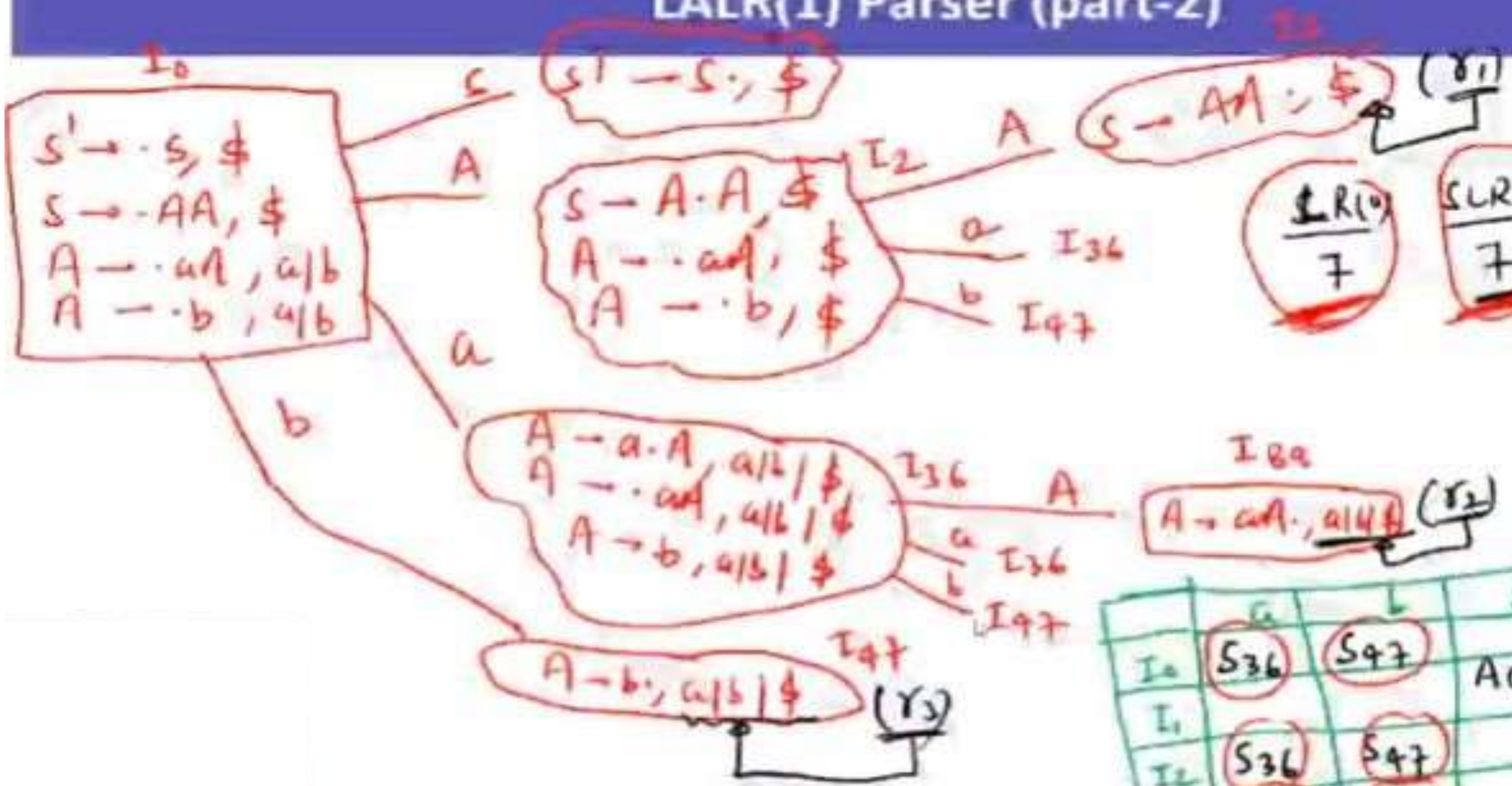
- Now combine the state with common production part and different look ahead part in a single state and construct the parse table.
- If parse table is free from multiple entries i.e. free from SR and RR conflict then grammar is LALR(1) grammar.




I4
A→b·, a|b
I7
A→b·, $
A→b·, a|b|$
I47
I8
A→aA·, a|b
I9
A→aA·, $
I89
A→aA·, a|b|$


CLR(1) parser
I0
S' → ·S, $
S → ·AA, $
A → ·aA, a|b
A → ·b, a|b
S' → S·, $
I1
S → A·A, $
A → ·aA, $
A → ·b, $
I2
S → AA·, $
I5
A → aA·, $
I9
A → a·A, $
A → ·aA, $
A → ·b, $
I36
I6
I7
A → b·, $
I47
A → a·A, a|b|$
A → ·aA, a|b|$
A → ·b, a|b|$
A → aA·, a|b|$
I89
A → b·, a|b|$
(S) = {$}
(A) {a, b, $}
[I0 – I9]
(10 states)

LALR(1) Parser (part-2)



LALR(1) DFA

LR(0)	SLR(1)	CLR(1)	LALR(1)
7	7	10	7

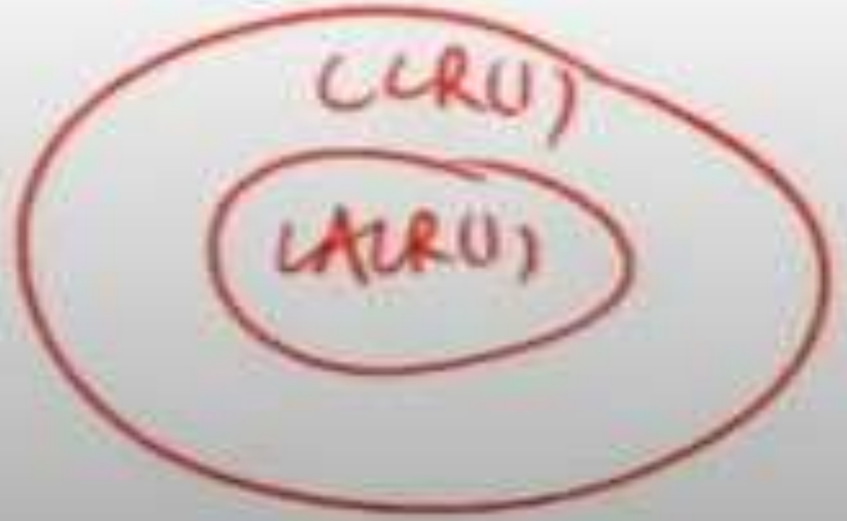
	a	b	$	S	A
I0	S36	S47		1	2
I1			Accept		
I2	S36	S47			5
I36	S36	S47			89
I47	r3	r3	r3		
I5			r1		
I89	r2	r2	r2		

Note:

- If the DFA of CLR(1) parser does not contain more than one state with common production part and diff. look ahead part then the grammar is CLR(1) and LALR(1).

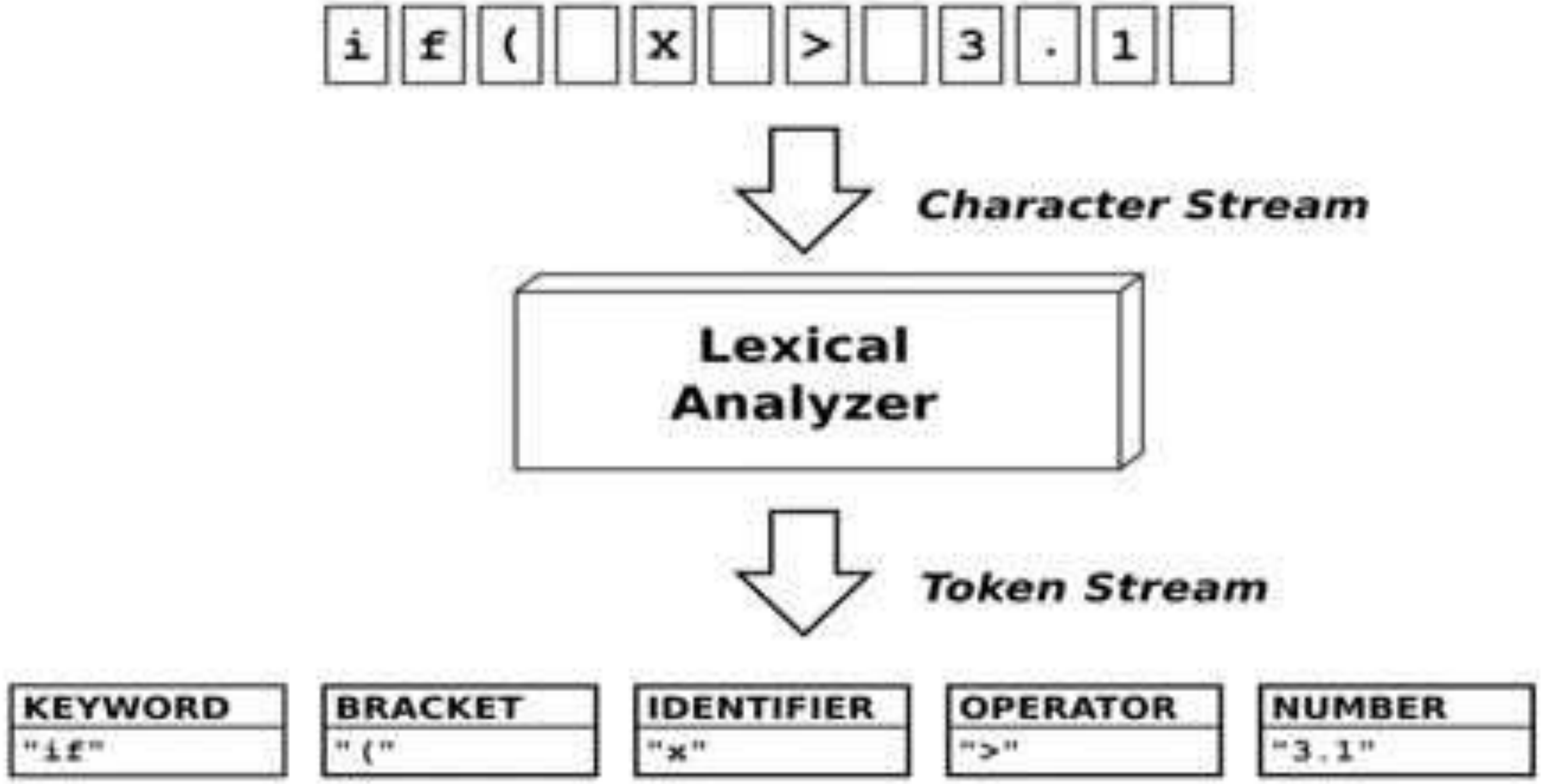
- If the DFA of CLR(1) Parser contain more than one state with common production part and diff. look ahead part then the grammar may or may not be LALR(1).

- every LALR(1) grammar is CLR(1) but every CLR(1) grammar need not be LALR(1).

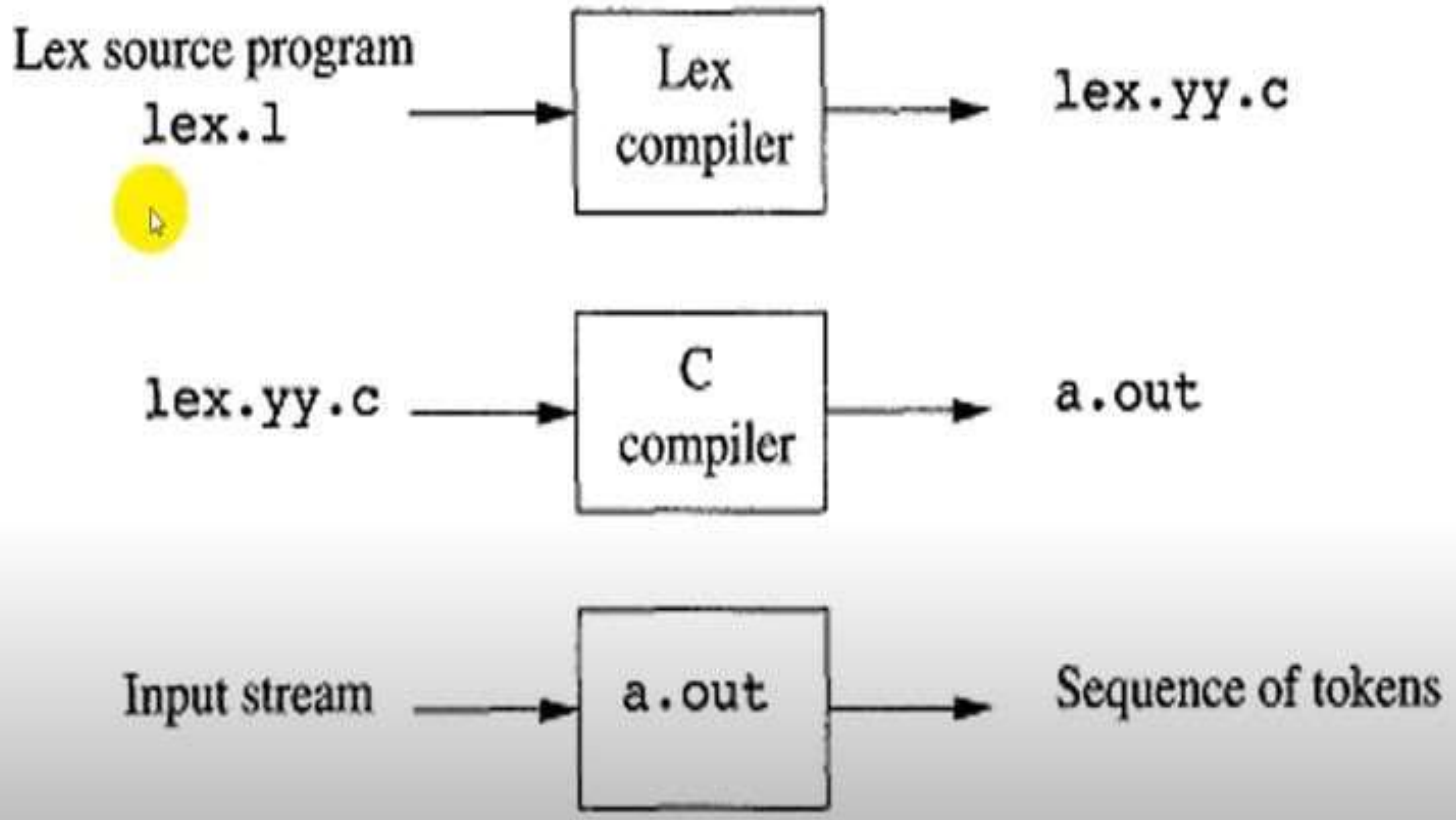


LEX

- It is tool which Generate Lexical analyser
- Lexical analyser is first phase of compiler which take input as source code and generate output as tokens



- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.
- The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c


Lex source program
lex.l
Lex compiler
lex.yy.c
lex.yy.c
C compiler
a.out
Input stream
a.out
Sequence of tokens

Working

- An input file, which we call lex.l, is written in the Lex language and describes the lexical analyser to be generated.
- The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c.
- The later file is compiled by the C compiler into a file called a.out, as always.
- The C-compiler output is a working lexical analyser that can take a stream of input characters and produce a stream of tokens

Structure of Lex Programs:

A Lex program has the following form:

{declarations}

%%

{translation rules}

%%

{auxiliary functions}

YACC

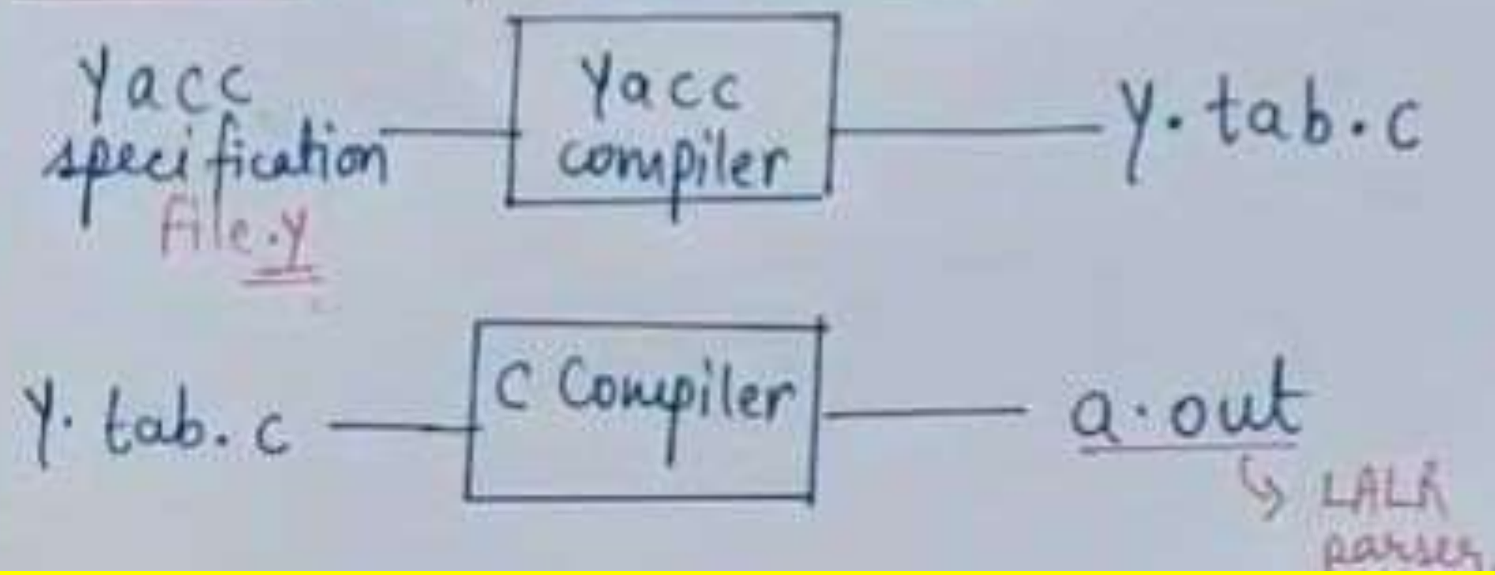
- YACC stands for Yet Another Compiler Compiler.
- It is a tool which Generate LALR Parser
- Syntax analyser (parser) is second phase of compiler which take input as token and generate syntax tree

YACC IN COMPILER DESIGN

It stands for Yet Another Compiler-Compiler (developed by Stephen C. Johnson)

- It is a tool for generating Look Ahead Left-to-Right (LALR) parser.
- It takes i/p from the Lexical Analyzer & generates parse Tree.
- Syntax Analyzer / Parser is the 2nd phase of the compiler which takes i/p as tokens and generates a parse tree.

WORKING (3 steps) / BLOCK DIAGRAM



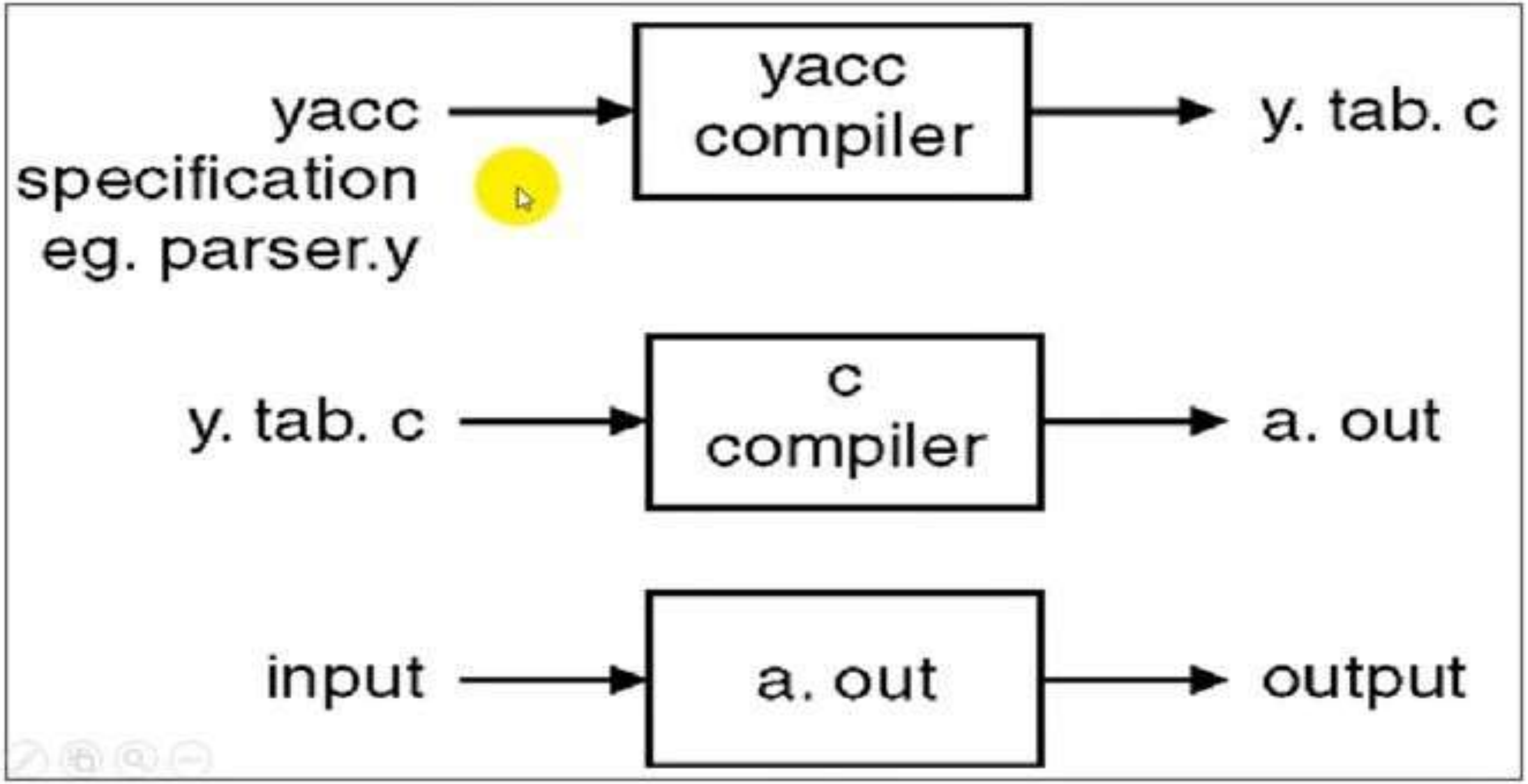
WORKING

1) I/P to the Yacc compiler will be a file with .y extension. It will contain desired grammar in Yacc format. YACC compiler will convert it into a C code in the form of y.tab.c file.

2) This y.tab.c file will be given as a input to the C Compiler and the output will be the LALR parser (ie. a.out)

3) Tokens generated by the lexical analyzer (using the lex tool) will be given as a input to a.out ie. our LALR parser and we will get the parse tree as output.

Working


yacc
specification
eg. parser.y
yacc
compiler
y. tab. c
y. tab. c
c
compiler
a. out
input
a. out
output

Syntax

Definitions

%%

Rules

%%

Supplementary Code

- **Definition Section**: All code between % and % is copied to the C file. The definitions section is where we configure various parser features such as defining token codes, establishing operator precedence and associativity and setting up variables used to communicate between the scanner and the parser.
- **Rules Section:** The required productions section is where we specify the grammar rule.
- **Supplementary Code Section:** It is used for ordinary C code that we want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines to the bottom.

JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Thank you!