

Introduction to Bottom Up Parser

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

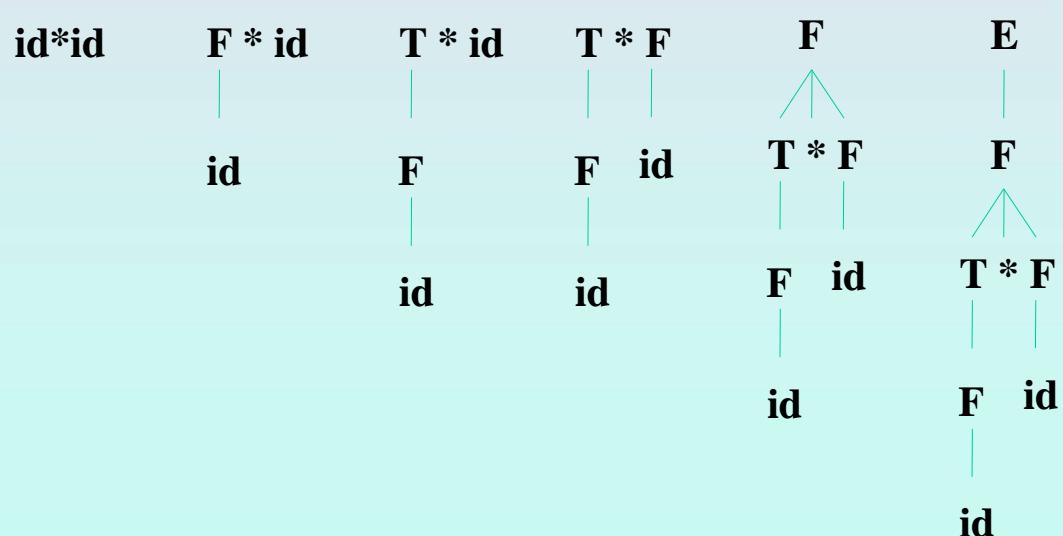
$S \Rightarrow \dots \Rightarrow \omega$ (the right-most derivation of ω)

\leftarrow (the bottom-up parser finds the right-most derivation in the reverse order)

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
 - At each shift action, the current symbol in the input string is pushed to a stack.
 - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will be replaced by the non-terminal at the left side of that production.
 - There are also two more actions: accept and error.

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id^*id

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$


Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
 - $E \Rightarrow T \Rightarrow T^*F \Rightarrow T^*id \Rightarrow F^*id \Rightarrow id^*id$

- Bottom-Up Parsing

- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
- Shift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.
- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.

A String $\xrightarrow{\text{reduce to}}$ the starting symbol

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

a string \rightarrow the starting symbol
reduced to
- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation:

$$S \xrightarrow[\text{rm}]{*} \omega$$

Shift-Reduce Parser finds:

$$\omega \xleftarrow[\text{rm}]{} \dots \xleftarrow[\text{rm}]{} S$$

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$ input string: aa~~a~~bb

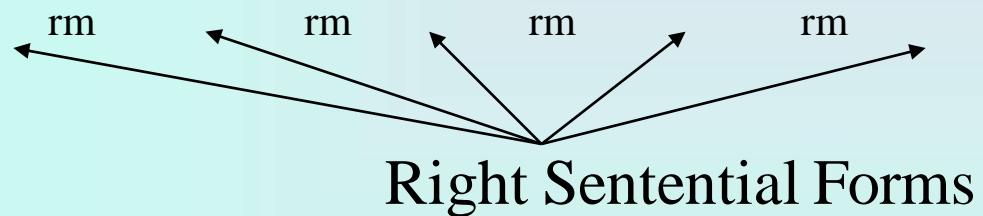
$A \rightarrow aA \mid a$ a~~a~~~~A~~bb

$B \rightarrow bB \mid b$ aAb~~b~~ \Downarrow reduction

aABb

S

$S \Rightarrow aABb \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb$



- How do we know which substring to be replaced at each reduction step?

$S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

Input: $abbcde$

$abbcde$

$aabbcd$
A

$aabbcd$
A
A

$aabbcd$
A
A
B

$aabbcd$
S
A
A
B

$abbcde$

$aAbcd$

aAd

aAe

S

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
 - But not every substring matches the right side of a production rule is handle
- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ .

$$S \xrightarrow[\text{rm}]{*} \alpha A \omega \xRightarrow[\text{rm}]{*} \alpha \beta \omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
$\text{id}^* \text{id}$	id	$F \rightarrow \text{id}$
$F^* \text{id}$	F	$T \rightarrow F$
$T^* \text{id}$	id	$F \rightarrow \text{id}$
$T^* F$	$T^* F$	$E \rightarrow T^* F$

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
\$	w\$

- Acceptance configuration

Stack	Input
\$S	\$

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S = \gamma_0 \xrightarrow{rm} \gamma_1 \xrightarrow{rm} \gamma_2 \xrightarrow{rm} \cdots \xrightarrow{rm} \gamma_{n-1} \xrightarrow{rm} \gamma_n = \omega$$

← input string

- Start from γ_n , find a handle $A_n \rightarrow \beta_n$ in γ_n , and replace β_n in by A_n to get γ_{n-1} .
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1} , and replace β_{n-1} in by A_{n-1} to get γ_{n-2} .
- Repeat this, until we reach S .

A Shift-Reduce Parser

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T^*F \mid F$$
$$F \rightarrow (E) \mid id$$

Right-Most Derivation of $id + id^* id$

$$E \Rightarrow E+T \Rightarrow E+T^*F \Rightarrow E+T^*id \Rightarrow E+F^*id$$
$$\Rightarrow E+id^*id \Rightarrow T+id^*id \Rightarrow F+id^*id \Rightarrow id + id^* id$$

Right-Most Sentential Form

id + id^*id

F + id^*id

T + id^*id

$E + \underline{id}^*id$

$E + \underline{F}^*id$

$E + T^*\underline{id}$

$E + \underline{T^*F}$

$E+T$

E

Reducing Production

$F \rightarrow id$

$T \rightarrow F$

$E \rightarrow T$

$F \rightarrow id$

$T \rightarrow F$

$F \rightarrow id$

$T \rightarrow T^*F$

$E \rightarrow E+T$

Handles are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

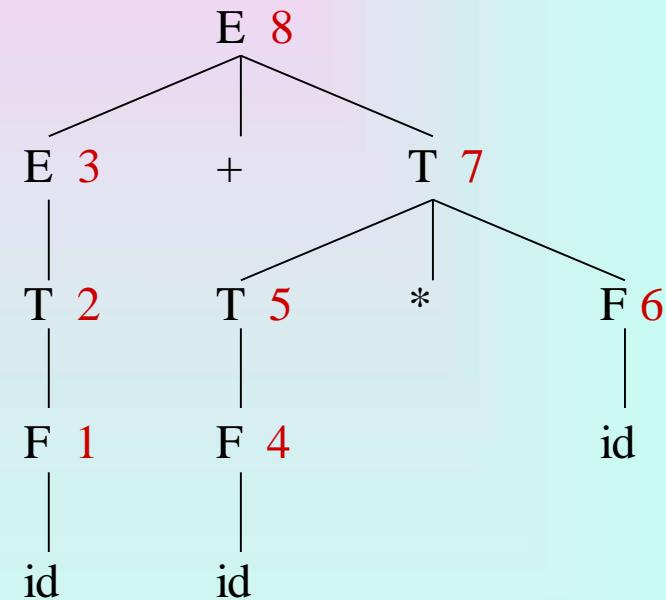
- There are four possible actions of a shift-parser action:
 1. **Shift** : The next input symbol is shifted onto the top of the stack.
 2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
 3. **Accept**: Successful completion of parsing.
 4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.
- Initial stack just contains only the end-marker \$.
- The end of the input string is marked by the end-marker \$.

A Stack Implementation of A Shift-Reduce Parser

Stack

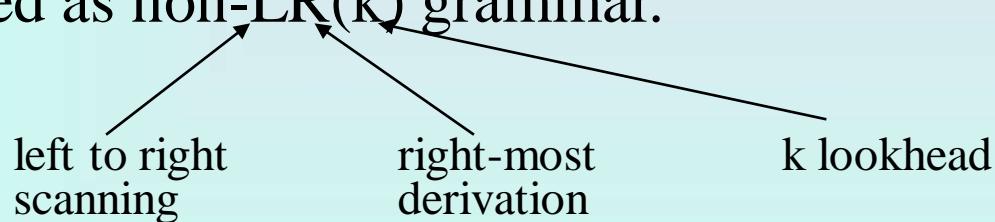
<u>Stack</u>	<u>Input</u>	<u>Action</u>
\$	id+id*id\$	shift
\$id	+id*id\$	reduce by F → id
\$F	+id*id\$	reduce by T → F
\$T	+id*id\$	reduce by E → T
\$E	+id*id\$	shift
\$E+	id*id\$	shift
\$E+id	*id\$	reduce by F → id
\$E+F	*id\$	reduce by T → F
\$E+T	*id\$	shift
\$E+T*	id\$	shift
\$E+T*id	\$	reduce by F → id
\$E+T*F	\$	reduce by T → T*F
\$E+T	\$	reduce by E → E+T
\$E	\$	accept

Parse Tree



Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
 - **shift/reduce conflict**: Whether make a shift operation or a reduction.
 - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as **non-LR(k) grammar**.



- An ambiguous grammar can never be a LR grammar.

Shift-Reduce Parsers

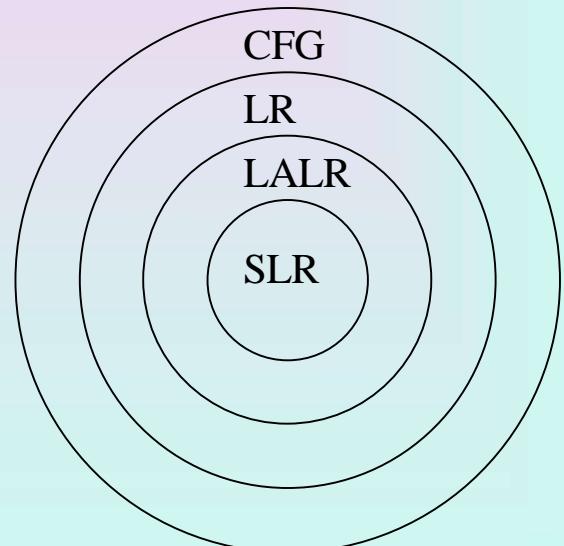
- There are two main categories of shift-reduce parsers

1. Operator-Precedence Parser

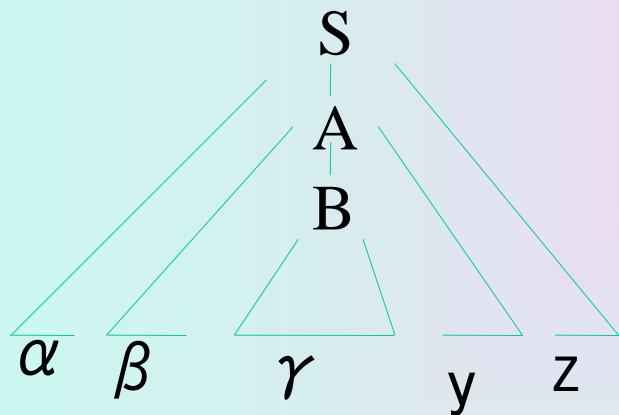
- simple, but only a small class of grammars.

2. LR-Parsers

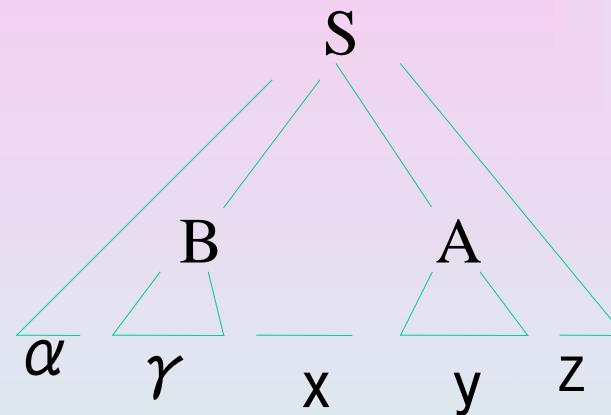
- covers wide range of grammars.
 - SLR – simple LR parser
 - LR – most general LR parser
 - LALR – intermediate LR parser (lookhead LR parser)
- SLR, LR and LALR work same, only their parsing tables are different.



Handle will appear on top of the stack



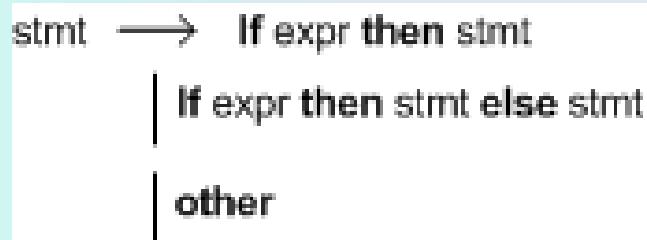
Stack	Input
$\$ \alpha \beta \gamma$	$yz\$$
$\$ \alpha \beta B$	$yz\$$
$\$ \alpha \beta B y$	$z\$$



Stack	Input
$\$ \alpha \gamma$	$xyz\$$
$\$ \alpha B x y$	$z\$$

Conflicts during shift reduce parsing

- Two kind of conflicts
 - Shift/reduce conflict
 - Reduce/reduce conflict
- Example:



Reduce/reduce conflict

stmt -> id(parameter_list)

stmt -> expr:=expr

parameter_list->parameter_list, parameter

parameter_list->parameter

parameter->id

expr->id(expr_list)

expr->id

expr_list->expr_list, expr

expr_list->expr

Stack

... id(id

Input
,id) ...\$

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.

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-

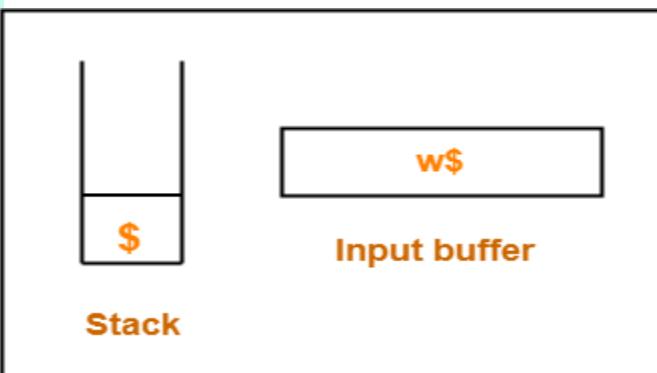
- An error is detected.

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

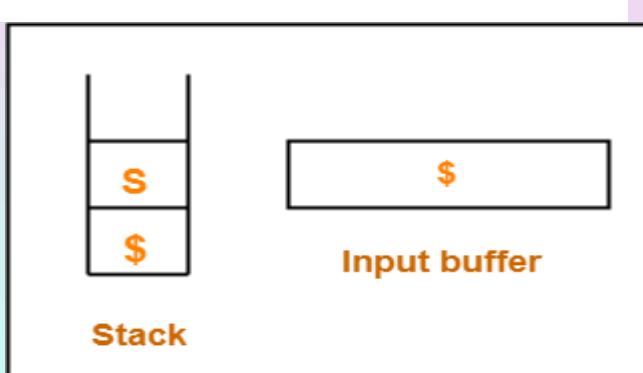
After achieving this configuration,

- The parser stops / halts.

- It reports the successful completion of parsing.



Initial Configuration



Final Configuration

Possible Actions-

A shift-reduce parser can possibly make the following four actions-

1. Shift-

In a shift action,

- The next symbol is shifted onto the top of the stack.

2. Reduce-

In a reduce action,

- The handle appearing on the stack top is replaced with the appropriate non-terminal symbol.

3. Accept-

In an accept action,

- The parser reports the successful completion of parsing.

4. Error-

In this state,

- The parser becomes confused and is not able to make any decision.

- It can neither perform shift action nor reduce action nor accept action.

Stack	Input Buffer	Parsing Action
\$	id – id x id \$	Shift
\$ id	– id x id \$	Reduce E → id
\$ E	– id x id \$	Shift
\$ E –	id x id \$	Shift
\$ E – id	x id \$	Reduce E → id
\$ E – E	x id \$	Shift
\$ E – E x	id \$	Shift
\$ E – E x id	\$	Reduce E → id
\$ E – E x E	\$	Reduce E → E x E
\$ E – E	\$	Reduce E → E – E
\$ E	\$	Accept

Problem-01:

Consider the following grammar-

$$E \rightarrow E - E$$

$$E \rightarrow E \times E$$

$$E \rightarrow id$$

.

Parse the input string
id – id x id

using a shift-reduce parser

Solution-

The priority order is:

$$id > x > -$$

- **Problem-02:**
- Consider the following grammar-
 - $S \rightarrow (L) \mid a$
 - $L \rightarrow L, S \mid S$
- Parse the input string
- $(a, (a, a))$
- using a shift-reduce parser.

Stack	Input Buffer	Parsing Action
\$	(a , (a , a)) \$	Shift
\$ (a , (a , a)) \$	Shift
\$ (a	, (a , a)) \$	Reduce $S \rightarrow a$
\$ (S	, (a , a)) \$	Reduce $L \rightarrow S$
\$ (L	, (a , a)) \$	Shift
\$ (L ,	(a , a)) \$	Shift
\$ (L , (a , a)) \$	Shift
\$ (L , (a	, a)) \$	Reduce $S \rightarrow a$
\$ (L , (S	, a)) \$	Reduce $L \rightarrow S$
\$ (L , (L	, a)) \$	Shift
\$ (L , (L ,	a)) \$	Shift
\$ (L , (L , a)) \$	Reduce $S \rightarrow a$
\$ (L , (L , S)) \$	Reduce $L \rightarrow L, S$
\$ (L , (L)) \$	Shift
\$ (L , (L) \$	Reduce $S \rightarrow (L)$
\$ (L , S) \$	Reduce $L \rightarrow L, S$
\$ (L) \$	Shift
\$ (L	\$	Reduce $S \rightarrow (L)$
\$ S	\$	Accept

Problem-03:

- Consider the following grammar-
 - $S \rightarrow T L$
 - $T \rightarrow \text{int} \mid \text{float}$
 - $L \rightarrow L, \text{id} \mid \text{id}$
- Parse the input string
- int id , id ;
- using a shift-reduce parser.

Stack	Input Buffer	Parsing Action
\$	int id , id ; \$	Shift
\$ int	id , id ; \$	Reduce $T \rightarrow \text{int}$
\$ T	id , id ; \$	Shift
\$ T id	, id ; \$	Reduce $L \rightarrow \text{id}$
\$ T L	, id ; \$	Shift
\$ T L ,	id ; \$	Shift
\$ T L , id	; \$	Reduce $L \rightarrow L, \text{id}$
\$ T L	; \$	Shift
\$ T L ;	\$	Reduce $S \rightarrow T L$
\$ S	\$	Accept

Problem-04:

Considering the string

“10201”,

design a shift-reduce parser for the following grammar-

$S \rightarrow 0S0 \mid 1S1 \mid 2$

Stack	Input Buffer	Parsing Action
\$	1 0 2 0 1 \$	Shift
\$ 1	0 2 0 1 \$	Shift
\$ 1 0	2 0 1 \$	Shift
\$ 1 0 2	0 1 \$	Reduce $S \rightarrow 2$
\$ 1 0 S	0 1 \$	Shift
\$ 1 0 S 0	1 \$	Reduce $S \rightarrow 0 S 0$
\$ 1 S	1 \$	Shift
\$ 1 S 1	\$	Reduce $S \rightarrow 1 S 1$
\$ S	\$	Accept

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:

$E \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

not operator grammar

$E \rightarrow EOE$

$E \rightarrow id$

$O \rightarrow +|*|/$

not operator grammar

$E \rightarrow E+E \mid$

$E^*E \mid$

$E/E \mid id$

operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

$a < \cdot b$ b has higher precedence than a

$a = \cdot b$ b has same precedence as a

$a \cdot > b$ b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,
 - <· with marking the left end,
 - =· appearing in the interior of the handle, and
 - > marking the right hand.
- In our input string $\$a_1a_2\dots a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

$$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E^\wedge E \mid (E) \mid -E \mid id$$

The partial operator-precedence table for this grammar

	id	+	*	\$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
\$	<·	<·	<·	

- Then the input string $id+id^*id$ with the precedence relations inserted will be:

$\$ <· id ·> + <· id ·> * <· id ·> \$$

To Find The Handles

1. Scan the string from left end until the first $\cdot >$ is encountered.
2. Then scan backwards (to the left) over any $= \cdot$ until a $< \cdot$ is encountered.
3. The handle contains everything to left of the first $\cdot >$ and to the right of the $< \cdot$ is encountered.

$\$ < \cdot \text{id} \cdot > + < \cdot \text{id} \cdot > * < \cdot \text{id} \cdot > \$$	$E \rightarrow id$	$\$ \text{id} + id * id \$$
$\$ < \cdot + < \cdot \text{id} \cdot > * < \cdot \text{id} \cdot > \$$	$E \rightarrow id$	$\$ E + \text{id} * id \$$
$\$ < \cdot + < \cdot * < \cdot \text{id} \cdot > \$$	$E \rightarrow id$	$\$ E + E * \text{id} \$$
$\$ < \cdot + < \cdot * \cdot > \$$	$E \rightarrow E^*E$	$\$ E + E * E \$$
$\$ < \cdot + \cdot > \$$	$E \rightarrow E+E$	$\$ E + E \$$
$\$ \$$		$\$ E \$$

Operator-Precedence Parsing Algorithm

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

Algorithm:

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 $a = \cdot b$) **then** { /* SHIFT */

push b onto the stack;

advance p to the next input symbol;

}

else if ($a \cdot b$) **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

<u>stack</u>	<u>input</u>	<u>action</u>	
\$	id+id*id\$	\$ < id	shift
\$id	+id*id\$	id > +	reduce E → id
\$	+id*id\$	shift	
\$+	id*id\$	shift	
\$+id	*id\$	id > *	reduce E → id
\$+	*id\$	shift	
\$+*	id\$	shift	
\$+*id	\$	id > \$	reduce E → id
\$+*	\$	* > \$	reduce E → E*E
\$+	\$	+ > \$	reduce E → E+E
\$	\$	accept	

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.
1. If operator O_1 has higher precedence than operator O_2 ,
 $\rightarrow O_1 \cdot > O_2$ and $O_2 \cdot < O_1$
 2. If operator O_1 and operator O_2 have equal precedence,
they are left-associative $\rightarrow O_1 \cdot > O_2$ and $O_2 \cdot > O_1$
they are right-associative $\rightarrow O_1 \cdot < O_2$ and $O_2 \cdot < O_1$
 3. For all operators O ,
 $O \cdot < id$, $id \cdot > O$, $O \cdot < ($, $(\cdot < O$, $O \cdot >$), $) \cdot > O$, $O \cdot > \$$, and $\$ \cdot < O$
 4. Also, let
 - $(= \cdot)$ $\$ \cdot < ($ $id \cdot >)$ $) \cdot > \$$
 - $(\cdot < ($ $\$ \cdot < id$ $id \cdot > \$$ $) \cdot >)$
 - $(\cdot < id$

Operator-Precedence Relations

	+	-	*	/	\wedge	id	()	\$
+	$\dot{>}$	$\dot{>}$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{>}$
-	$\dot{>}$	$\dot{>}$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{>}$	$\dot{>}$
*	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{>}$	$\dot{>}$
/	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{>}$	$\dot{>}$
\wedge	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{<}.$	$\dot{<}.$	$\dot{<}.$	$\dot{>}$	$\dot{>}$
id	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$			$\dot{>}$	$\dot{>}$
($\dot{<}.$	$=.$							
)	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$	$\dot{>}$			$\dot{>}$	$\dot{>}$
\$	$\dot{<}.$								

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
 - The lexical analyzer will return two different operators for the unary minus and the binary minus.
 - The lexical analyzer will need a lookahead to distinguish the binary minus from the unary minus.
- Then, we make

$O < \cdot$ unary-minus

for any operator

unary-minus $\cdot >$ O

if unary-minus has higher precedence than O

unary-minus $< \cdot$ O

if unary-minus has lower (or equal) precedence than O

Operator-Precedance Grammars

Let G be an ϵ -free operator grammar (No ϵ -Production). For each terminal symbols a and b , the following conditions are satisfies.

1. $a \doteq b$, if \exists a production in RHS of the form $aa\beta by\gamma$, where β is either ϵ or a single non Terminal. Ex $S \rightarrow iCtSeS$ implies $i \doteq t$ and $t \doteq e$.
2. $a < \cdot b$ if for some non-terminal A \exists a production in RHS of the form $A \rightarrow aaA\beta$, and $A \Rightarrow^+ \gamma b\delta$ where γ is either ϵ or a single non-terminal. Ex $S \rightarrow iCtS$ and $C \Rightarrow^+ b$ implies $i < \cdot b$.
3. $a \cdot > b$ if for some non-terminal A \exists a production in RHS of the form $A \rightarrow aAb\beta$, and $A \Rightarrow^+ \gamma a\delta$ where δ is either ϵ or a single non-terminal. Ex $S \rightarrow iCtS$ and $C \Rightarrow^+ b$ implies $b \cdot > t$.

Example: $E \rightarrow E+E \mid E^*E \mid (E) \mid id$ is not a Operator precedence Grammar

By Rule no. 3 we have $+ < \cdot + \& + \cdot > +$. Where as we can modify the Grammar is as follow

$E \rightarrow E+T \mid T, T \rightarrow T^*F \mid F, F \rightarrow (E) \mid id$

Operator Precedence Relations.

To find the Table we have to find the last & first terminal for each non-terminal as follows:

<u>Non terminal</u>	<u>First terminal</u>	<u>Last terminal</u>
E	$*, +, (, \text{id}$	$*, +,), \text{id}$
T	$*, (, \text{id}$	$*,), \text{id}$
F	$(, \text{id}$	$), \text{id}$

By Applying the Rule of Operator Precedence Grammar

	$+$	$*$	$($	$)$	id	$\$$
$+$	$>$	$<\cdot$	$<\cdot$	$>$	$<\cdot$	$>$
$*$	$>$	$>$	$<\cdot$	$>$	$<\cdot$	$>$
$($	$<\cdot$	$<\cdot$	$<\cdot$	\doteq	$<\cdot$	
$)$	$>$	$>$		$>$		$>$
id	$>$	$>$		$>$		$>$
$\$$	$<\cdot$	$<\cdot$	$<\cdot$		$<\cdot$	

Operator Precedence Relations, Continue....

To produce the Table we have to follow the procedure as:

$\text{LEADING}(A) = \{ a \mid A \Rightarrow^+ \gamma a \delta, \text{ where } \gamma \text{ is } \in \text{ or a single non-terminal.} \}$

$\text{TRAILING}(A) = \{ a \mid A \Rightarrow^+ \gamma a \delta, \text{ where } \delta \text{ is } \in \text{ or a single non-terminal.} \}$

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b .

$f(a) < g(b)$ whenever $a < \cdot b$

$f(a) = g(b)$ whenever $a = \cdot b$

$f(a) > g(b)$ whenever $a \cdot > b$

Disadvantages of Operator Precedence Parsing

- **Disadvantages:**
 - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
 - Small class of grammars.
 - Difficult to decide which language is recognized by the grammar.
- **Advantages:**
 - simple
 - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

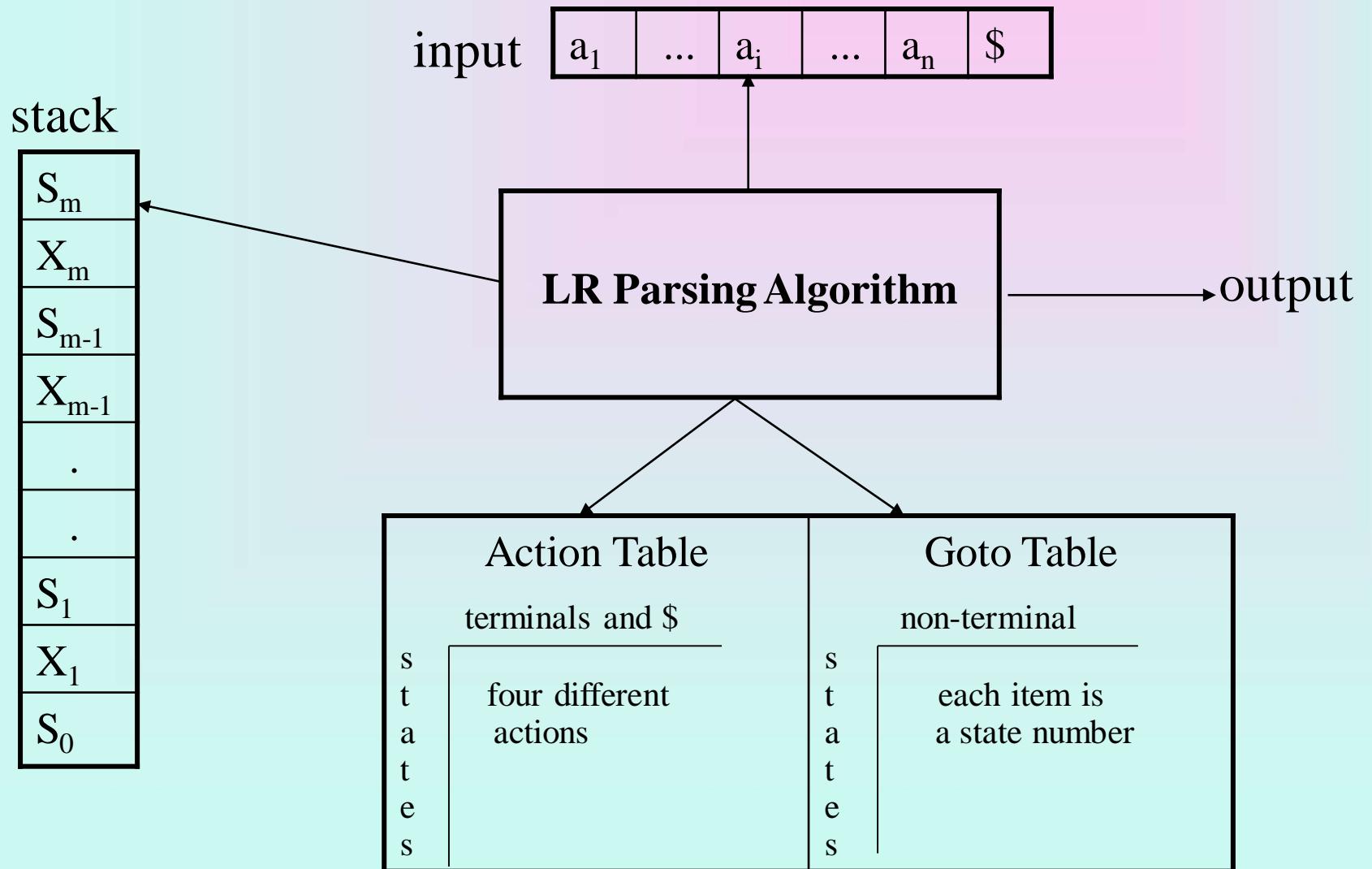


- LR parsing is attractive because:
 - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
 - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
 $LL(1)\text{-Grammars} \subset LR(1)\text{-Grammars}$
 - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

- **LR-Parsers**
 - covers wide range of grammars.
 - SLR – simple LR parser
 - LR – most general LR parser
 - LALR – intermediate LR parser (look-head LR parser)
 - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(\underbrace{S_o X_1 S_1 \dots X_m S_m}_{\text{Stack}}, \underbrace{a_i a_{i+1} \dots a_n \$}_{\text{Rest of Input}})$$

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)
- A configuration of a LR parsing represents the right sentential form:

$$X_1 \dots X_m a_i a_{i+1} \dots a_n \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack
 $(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_m S_m a_i s, a_{i+1} \dots a_n \$)$
2. **reduce A→β** (or **rn** where n is a production number)
 - pop $2|\beta| (=r)$ items from the stack;
 - then push **A** and **s** where **s=goto[s_{m-r},A]**
 $(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_{m-r} S_{m-r} A s, a_i \dots a_n \$)$
 - Output is the reducing production reduce $A \rightarrow \beta$
3. **Accept** – Parsing successfully completed
4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop $2|\beta|$ ($=r$) items from the stack; let us assume that $\beta = Y_1 Y_2 \dots Y_r$
- then push A and s where $s=\text{goto}[s_{m-r}, A]$

($S_o X_1 S_1 \dots X_{m-r} S_{m-r} Y_1 S_{m-r} \dots Y_r S_m, a_i a_{i+1} \dots a_n \$$)
→ ($S_o X_1 S_1 \dots X_{m-r} S_{m-r} A s, a_i \dots a_n \$$)

- In fact, $Y_1 Y_2 \dots Y_r$ is a handle.

$X_1 \dots X_{m-r} A a_i \dots a_n \$ \Rightarrow X_1 \dots X_m Y_1 \dots Y_r a_i a_{i+1} \dots a_n \$$

(SLR) Parsing Tables for Expression Grammar

- 1) $E \rightarrow E + T$
- 2) $E \rightarrow T$
- 3) $T \rightarrow T^* F$
- 4) $T \rightarrow F$
- 5) $F \rightarrow (E)$
- 6) $F \rightarrow id$

state	id	Action Table						Goto Table		
		+	*	()	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4			8	2	3	
5		r6	r6		r6	r6				
6	s5			s4				9	3	
7	s5			s4					10	
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Actions of A (S)LR-Parser -- Example

<u>stack</u>	<u>input</u>	<u>action</u>	<u>output</u>
0	id*id+id\$	shift 5	
0id5	*id+id\$	reduce by F→id	F→id
0F3	*id+id\$	reduce by T→F	T→F
0T2	*id+id\$	shift 7	
0T2*7	id+id\$	shift 5	
0T2*7id5	+id\$	reduce by F→id	F→id
0T2*7F10	+id\$	reduce by T→T*F	T→T*F
0T2	+id\$	reduce by E→T	E→T
0E1	+id\$	shift 6	
0E1+6	id\$	shift 5	
0E1+6id5	\$	reduce by F→id	F→id
0E1+6F3	\$	reduce by T→F	T→F
0E1+6T9	\$	reduce by E→E+T	E→E+T
0E1	\$	accept	

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0)** item of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: $A \rightarrow \bullet aBb$
(four different possibility) $A \rightarrow a \bullet Bb$
 $A \rightarrow aB \bullet b$
 $A \rightarrow aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar:*
 G' is G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol.

The Closure Operation

- If I is a set of LR(0) items for a grammar G , then $\text{closure}(I)$ is the set of LR(0) items constructed from I by the two rules:
 1. Initially, every LR(0) item in I is added to $\text{closure}(I)$.
 2. If $A \rightarrow \alpha \bullet B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production rule of G ; then $B \rightarrow \bullet \gamma$ will be in the $\text{closure}(I)$.We will apply this rule until no more new LR(0) items can be added to $\text{closure}(I)$.

The Closure Operation -- Example

$E' \rightarrow E$

$\text{closure}(\{E' \rightarrow \bullet E\}) =$

$E \rightarrow E + T$

{ $E' \rightarrow \bullet E$ ← kernel items

$E \rightarrow T$

$E \rightarrow \bullet E + T$

$T \rightarrow T^* F$

$E \rightarrow \bullet T$

$T \rightarrow F$

$T \rightarrow \bullet T^* F$

$F \rightarrow (E)$

$T \rightarrow \bullet F$

$F \rightarrow \text{id}$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet \text{id}$ }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then $\text{goto}(I, X)$ is defined as follows:
 - If $A \rightarrow \alpha \bullet X \beta$ in I
then every item in $\text{closure}(\{A \rightarrow \alpha X \bullet \beta\})$ will be in $\text{goto}(I, X)$.

Example:

$$I = \{ E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, \\ T \rightarrow \bullet T^* F, T \rightarrow \bullet F, \\ F \rightarrow \bullet (E), F \rightarrow \bullet \text{id} \}$$

$$\text{goto}(I, E) = \{ E' \rightarrow E \bullet, E \rightarrow E \bullet + T \}$$

$$\text{goto}(I, T) = \{ E \rightarrow T \bullet, T \rightarrow T \bullet * F \}$$

$$\text{goto}(I, F) = \{ T \rightarrow F \bullet \}$$

$$\text{goto}(I, \bullet) = \{ F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T^* F, T \rightarrow \bullet F, \\ F \rightarrow \bullet (E), F \rightarrow \bullet \text{id} \}$$

$$\text{goto}(I, \text{id}) = \{ F \rightarrow \text{id} \bullet \}$$

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- *Algorithm:*

C is $\{ \text{closure}(\{S' \rightarrow \bullet S\}) \}$
repeat the followings until no more set of LR(0) items can be added to C .
for each I in C and each grammar symbol X
if $\text{goto}(I, X)$ is not empty and not in C
 add $\text{goto}(I, X)$ to C
- goto function is a DFA on the sets in C .

The Canonical LR(0) Collection -- Example

$I_0: E' \rightarrow .E$ $I_1: E' \rightarrow E.$ $I_6: E \rightarrow E+.T$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T^*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$E \rightarrow E.+T$

$I_9: E \rightarrow E+T.$

$T \rightarrow .T^*F$

$T \rightarrow T.^*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$I_{10}: T \rightarrow T^*F.$

$F \rightarrow .id$

$I_2: E \rightarrow T.$

$T \rightarrow T.^*F$

$I_3: T \rightarrow F.$

$I_4: F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T^*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_7: T \rightarrow T^*.F$

$F \rightarrow .(E)$

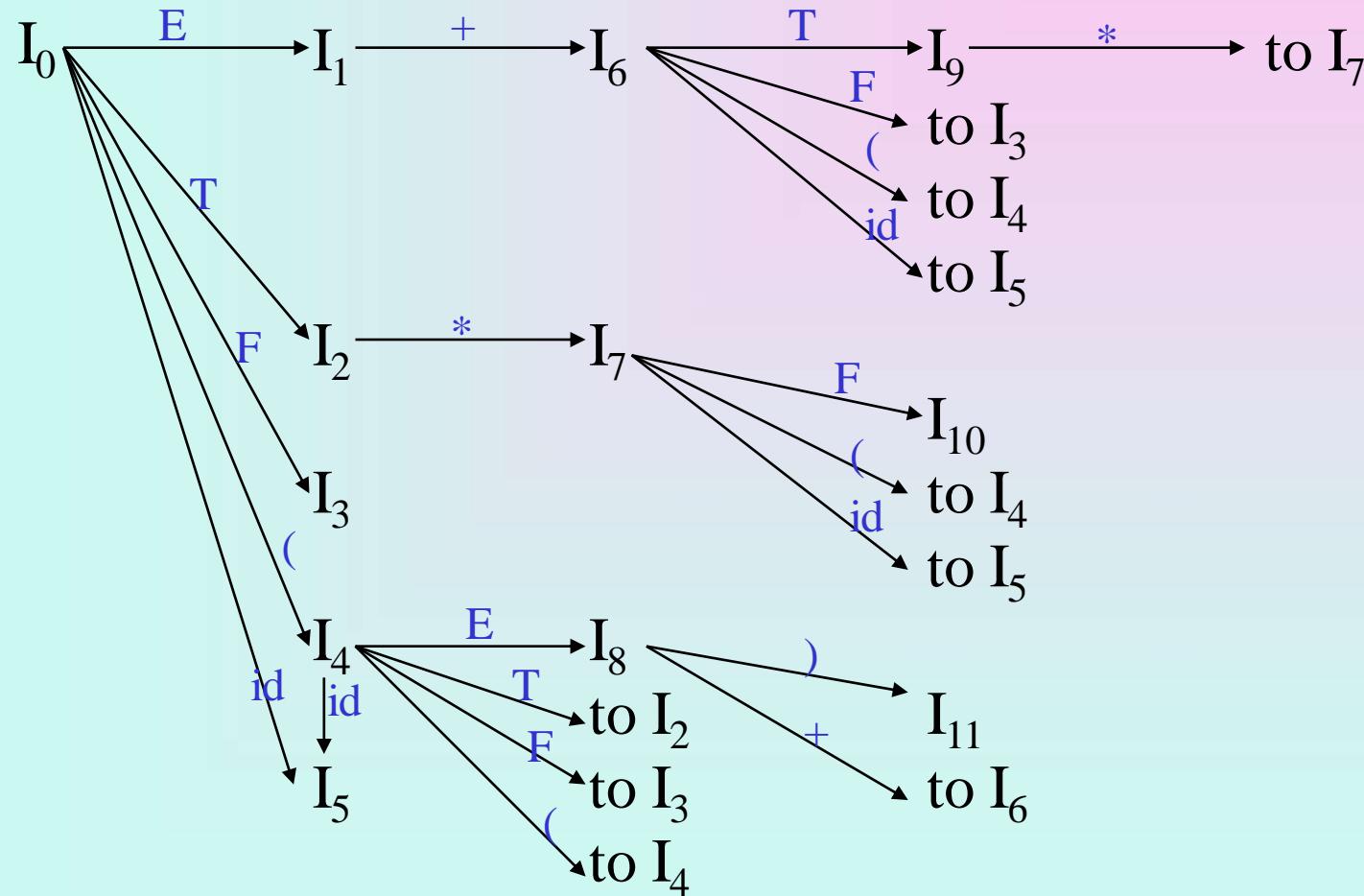
$F \rightarrow .id$

$I_8: F \rightarrow (E.)$

$E \rightarrow E.+T$

$I_5: F \rightarrow id.$

Transition Diagram (DFA) of Goto Function



LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with $k \leq 1$
- Why LR parsers?
 - Table driven
 - Can be constructed to recognize all programming language constructs
 - Most general non-backtracking shift-reduce parsing method
 - Can detect a syntactic error as soon as it is possible to do so
 - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
 - For $A \rightarrow XYZ$ we have following items
 - $A \rightarrow .XYZ$
 - $A \rightarrow X.YZ$
 - $A \rightarrow XY.Z$
 - $A \rightarrow XYZ.$
 - In a state having $A \rightarrow .XYZ$ we hope to see a string derivable from XYZ next on the input.
 - What about $A \rightarrow X.YZ$?

Constructing canonical LR(0) item sets

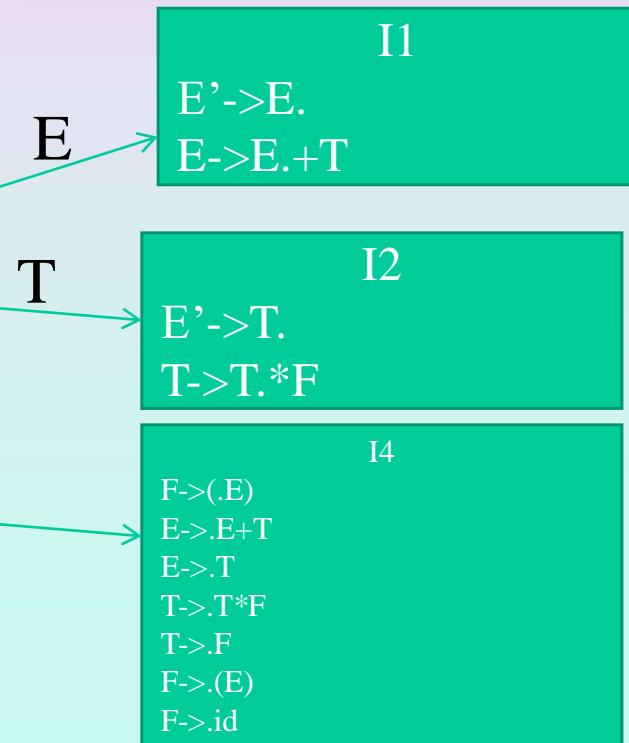
- Augmented grammar:
 - G with addition of a production: $S' \rightarrow S$
- Closure of item sets:
 - If I is a set of items, $\text{closure}(I)$ is a set of items constructed from I by the following rules:
 - Add every item in I to $\text{closure}(I)$
 - If $A \rightarrow \alpha.B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production then add the item $B \rightarrow .\gamma$ to $\text{closure}(I)$.
- Example:
 - $E' \rightarrow E$
 - $E \rightarrow E + T \mid T$
 - $T \rightarrow T * F \mid F$
 - $F \rightarrow (E) \mid \text{id}$

```
I0=closure({[E'->.E]})  
E'->.E  
E->.E+T  
E->.T  
T->.T*F  
T->.F  
F->.(E)  
F->.id
```

Constructing canonical LR(0) item sets (cont.)

- Goto (I, X) where I is an item set and X is a grammar symbol is closure of set of all items $[A \rightarrow \alpha X. \beta]$ where $[A \rightarrow \alpha X \beta]$ is in I
- Example

```
I0=closure({[E'->.E]})  
E'->.E  
E->.E+T  
E->.T  
T->.T*T  
T->.F  
F->.(E)  
F->.id
```



Closure algorithm

```
SetOfItems CLOSURE(I) {
```

```
    J=I;
```

```
    repeat
```

```
        for (each item A-> α.Bβ in J)
```

```
            for (each production B->γ of G)
```

```
                if (B->.γ is not in J)
```

```
                    add B->.γ to J;
```

```
    until no more items are added to J on one round;
```

```
    return J;
```

GOTO algorithm

```
SetOfItems GOTO(I,X) {
```

```
    J=empty;
```

```
    if (A->  $\alpha$ .X  $\beta$  is in I)
```

```
        add CLOSURE(A->  $\alpha$ X.  $\beta$  ) to J;
```

```
    return J;
```

```
}
```

Canonical LR(0) items

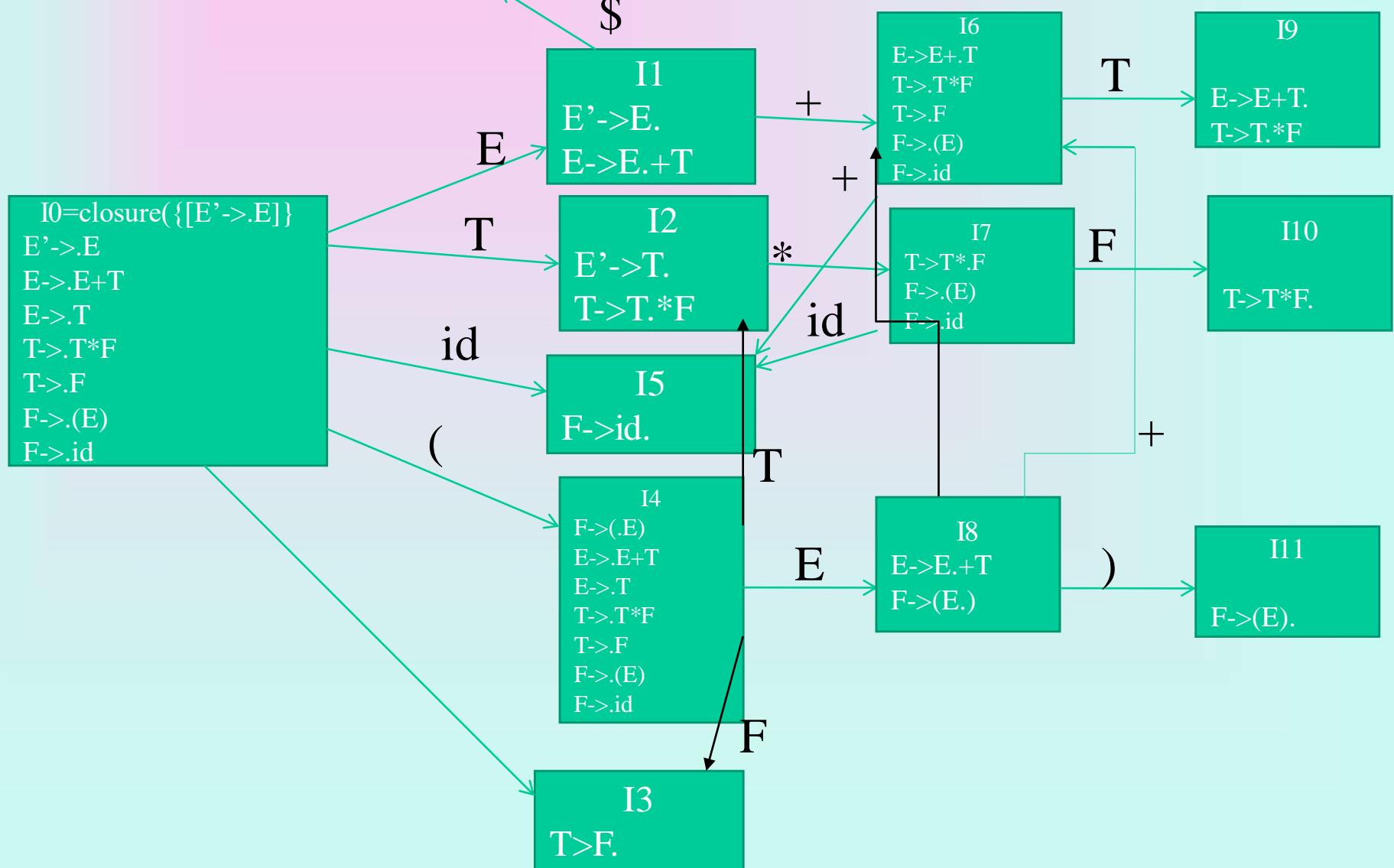
```
Void items(G') {  
    C= CLOSURE({[S'->.S]});  
    repeat  
        for (each set of items I in C)  
            for (each grammar symbol X)  
                if (GOTO(I,X) is not empty and not in C)  
                    add GOTO(I,X) to C;  
    until no new set of items are added to C on a round;  
}
```

Steps to Solve $LR(0)$ Sums

1. Augment the given grammar
2. Draw Conical Collection of $LR(0)$ item / DFD
3. Number the Production
4. Create the parsing table
5. Stack implementation
6. Draw parse tree

Example

$E' \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid \text{id}$



Example

STATE	ACTION							GOTO		
	id	+	*	()	\$	E	T	F	
0	S ₅			S ₄			1	2	3	
1		S ₆				Acc				
2		R ₂	S ₇		R ₂	R ₂				
3		R ₄			R ₄	R ₄				
4	S ₅			S ₄			8	2	3	
5	R ₆	R ₆								
6	S ₅			S ₄			9		3	
7	S ₅			S ₄					10	
8		S ₆			S ₁₁					
9	R ₁	R ₁	S ₇	R ₁	R ₁	R ₁				
10	R ₃	R ₃								
11	R ₅	R ₅								

- (0) E'->E
- (1) E -> E + T
- (2) E-> T
- (3) T -> T * F
- (4) T-> F
- (5) F -> (E)
- (6) F->**id**

id*id+id?

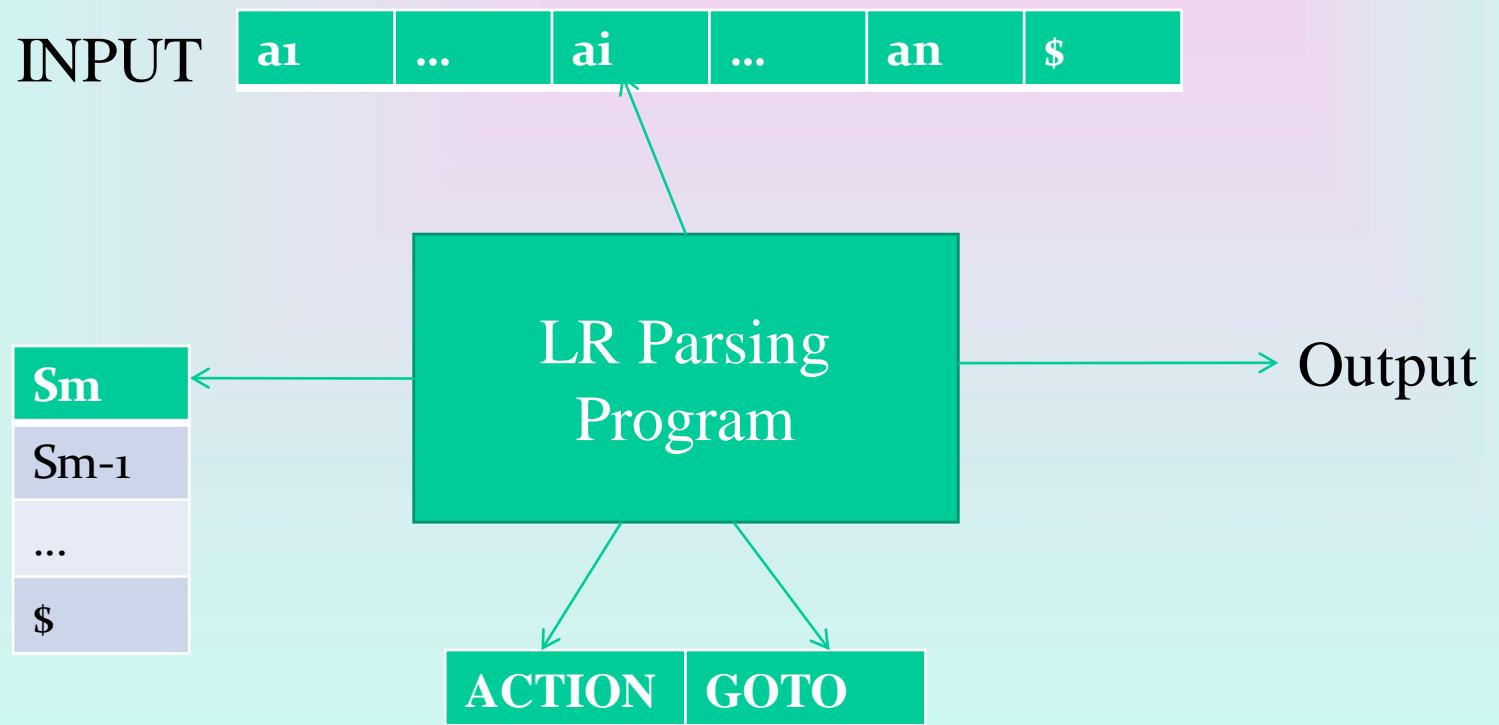
Line	Stack	Symbol	Input	Action
(1)	o		id*id+id\$	Shift to 5
(2)	o5	id	*id+id\$	Reduce by F->id
(3)	o3	F	*id+id\$	Reduce by T->F
(4)	o2	T	*id+id\$	Shift to 7
(5)	o27	T*	id+id\$	Shift to 5
(6)	o275	T*id	+id\$	Reduce by F->id
(7)	o2710	T*F	+id\$	Reduce by T->T*F
(8)	o2	T	+id\$	Reduce by E->T
(9)	o1	E	+id\$	Shift
(10)	o16	E+	id\$	Shift
(11)	o165	E+id	\$	Reduce by F->id
(12)	o163	E+F	\$	Reduce by T->F
(13)	o169	E+T`	\$	Reduce by E->E+T
(14)	o1	E	\$	accept

Use of LR(0) automaton

- Example: $\text{id}^* \text{id}$

Line	Stack	Symbols	Input	Action
(1)	o	\$	$\text{id}^* \text{id} \$$	Shift to 5
(2)	o5	\$id	$*\text{id} \$$	Reduce by $F \rightarrow \text{id}$
(3)	o3	\$F	$*\text{id} \$$	Reduce by $T \rightarrow F$
(4)	o2	\$T	$*\text{id} \$$	Shift to 7
(5)	o27	\$T*	$\text{id} \$$	Shift to 5
(6)	o275	\$T*id	\$	Reduce by $F \rightarrow \text{id}$
(7)	o2710	\$T*T	\$	Reduce by $T \rightarrow T^*F$
(8)	o2	\$T	\$	Reduce by $E \rightarrow T$
(9)	o1	\$E	\$	accept

LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;  
while(1) { /*repeat forever */  
    let s be the state on top of the stack;  
    if (ACTION[s,a] = shift t) {  
        push t onto the stack;  
        let a be the next input symbol;  
    } else if (ACTION[s,a] = reduce A->β) {  
        pop |β| symbols of the stack;  
        let state t now be on top of the stack;  
        push GOTO[t,A] onto the stack;  
        output the production A->β;  
    } else if (ACTION[s,a]=accept) break; /* parsing is done */  
    else call error-recovery routine;  
}
```

Example

STATE	ACTION							GOTO		
	id	+	*	()	\$	E	T	F	
0	S ₅			S ₄			1	2	3	
1		S ₆				Acc				
2		R ₂	S ₇		R ₂	R ₂				
3		R ₄	R ₇		R ₄	R ₄				
4	S ₅			S ₄			8	2	3	
5		R ₆	R ₆		R ₆	R ₆				
6	S ₅			S ₄			9		3	
7	S ₅			S ₄					10	
8		S ₆			S ₁₁					
9		R ₁	S ₇		R ₁	R ₁				
10		R ₃	R ₃		R ₃	R ₃				
11		R ₅	R ₅		R ₅	R ₅				

- (0) E'->E
- (1) E -> E + T
- (2) E-> T
- (3) T -> T * F
- (4) T-> F
- (5) F -> (E)
- (6) F->**id**

id*id+id?

Line	Stack	Symbol	Input	Action
(1)	o		id*id+id\$	Shift to 5
(2)	o5	id	*id+id\$	Reduce by F->id
(3)	o3	F	*id+id\$	Reduce by T->F
(4)	o2	T	*id+id\$	Shift to 7
(5)	o27	T*	id+id\$	Shift to 5
(6)	o275	T*id	+id\$	Reduce by F->id
(7)	o2710	T*F	+id\$	Reduce by T->T*F
(8)	o2	T	+id\$	Reduce by E->T
(9)	o1	E	+id\$	Shift
(10)	o16	E+	id\$	Shift
(11)	o165	E+id	\$	Reduce by F->id
(12)	o163	E+F	\$	Reduce by T->F
(13)	o169	E+T`	\$	Reduce by E->E+T
(14)	o1	E	\$	accept

$$S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA/b$$

$$S' \rightarrow \cdot S$$

$$S \rightarrow \cdot AA$$

$$A \rightarrow \cdot aA/\cdot b$$

$$S' \rightarrow S \cdot$$

$$S \rightarrow A \cdot A$$

$$A \rightarrow \cdot aA/\cdot b$$

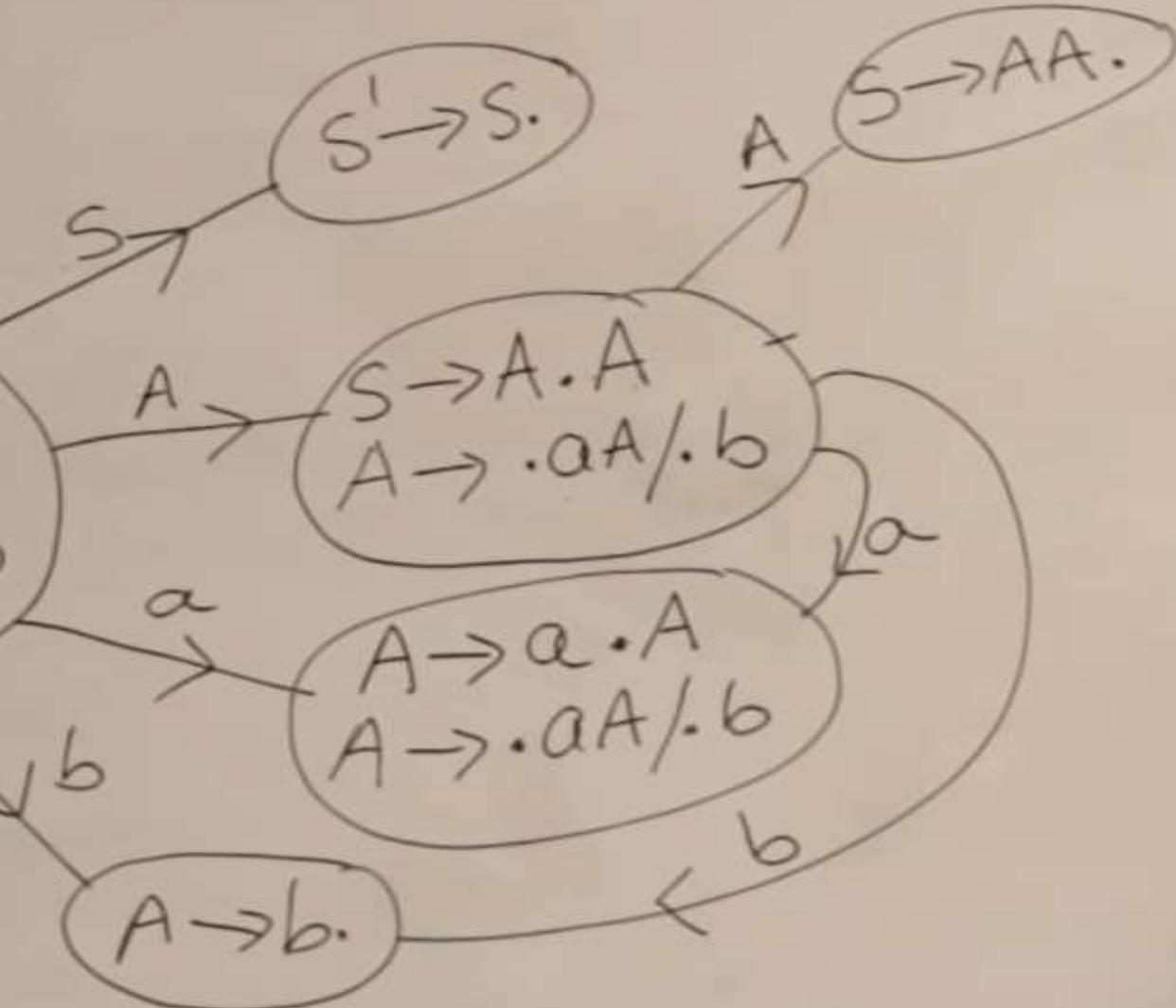
$$A \rightarrow a \cdot A$$

$$A \rightarrow \cdot aA/\cdot b$$

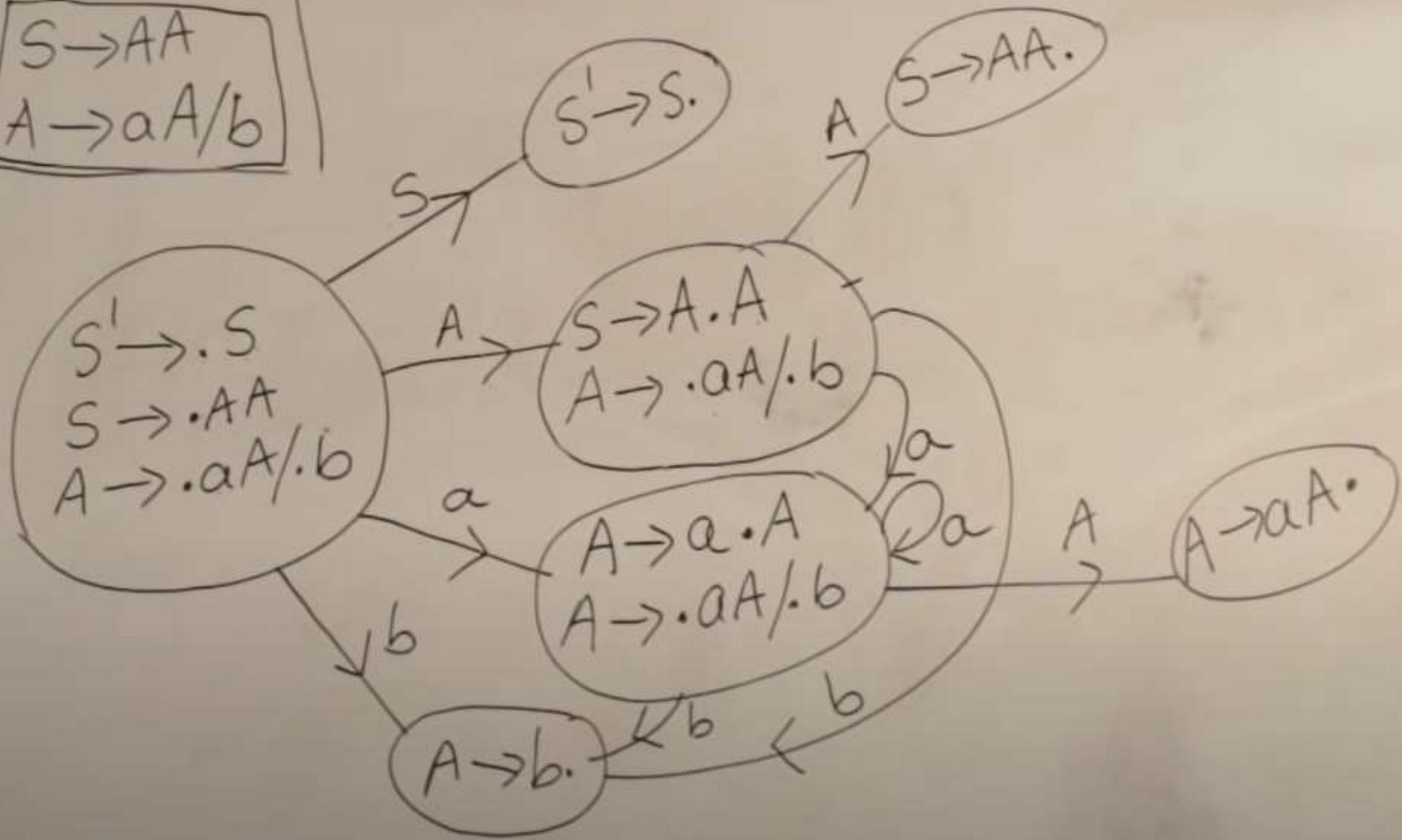
$$A \rightarrow b \cdot$$

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

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



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



$$S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA/b$$

I_0

$$S' \rightarrow .S$$

$$S \rightarrow .AA$$

$$A \rightarrow .aA/b$$

$S \rightarrow$

I_1

$$S' \rightarrow S.$$

I_5

$$S \rightarrow AA.$$

$A \rightarrow$

I_2

$$S \rightarrow A.A$$

$$A \rightarrow .aA/b$$

$a \rightarrow$

I_3

$$A \rightarrow a.A$$

$$A \rightarrow .aA/b$$

$a \rightarrow$

$A \rightarrow$

I_6

$$A \rightarrow aA.$$

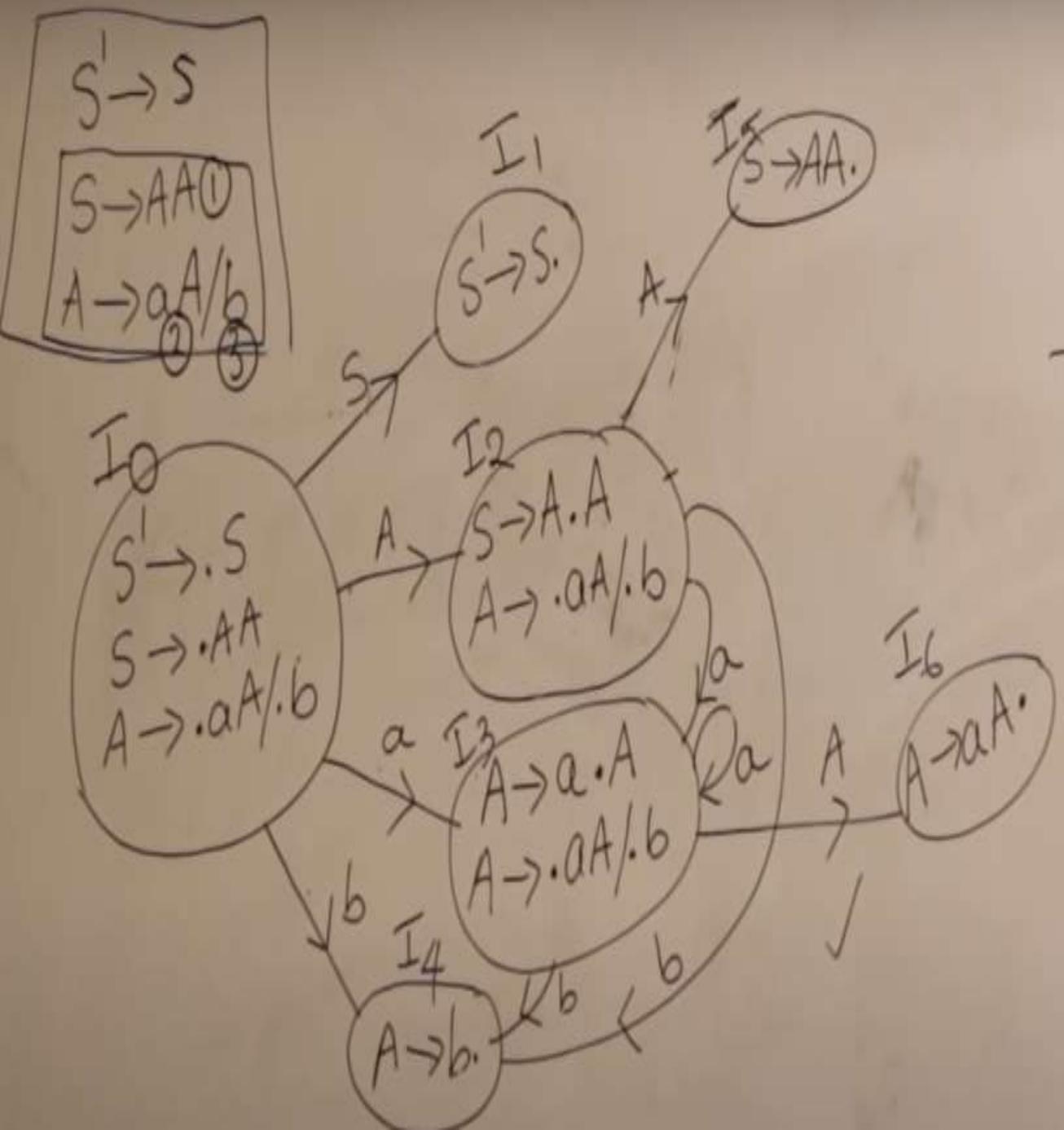
$b \rightarrow$

I_4

$$A \rightarrow b.$$

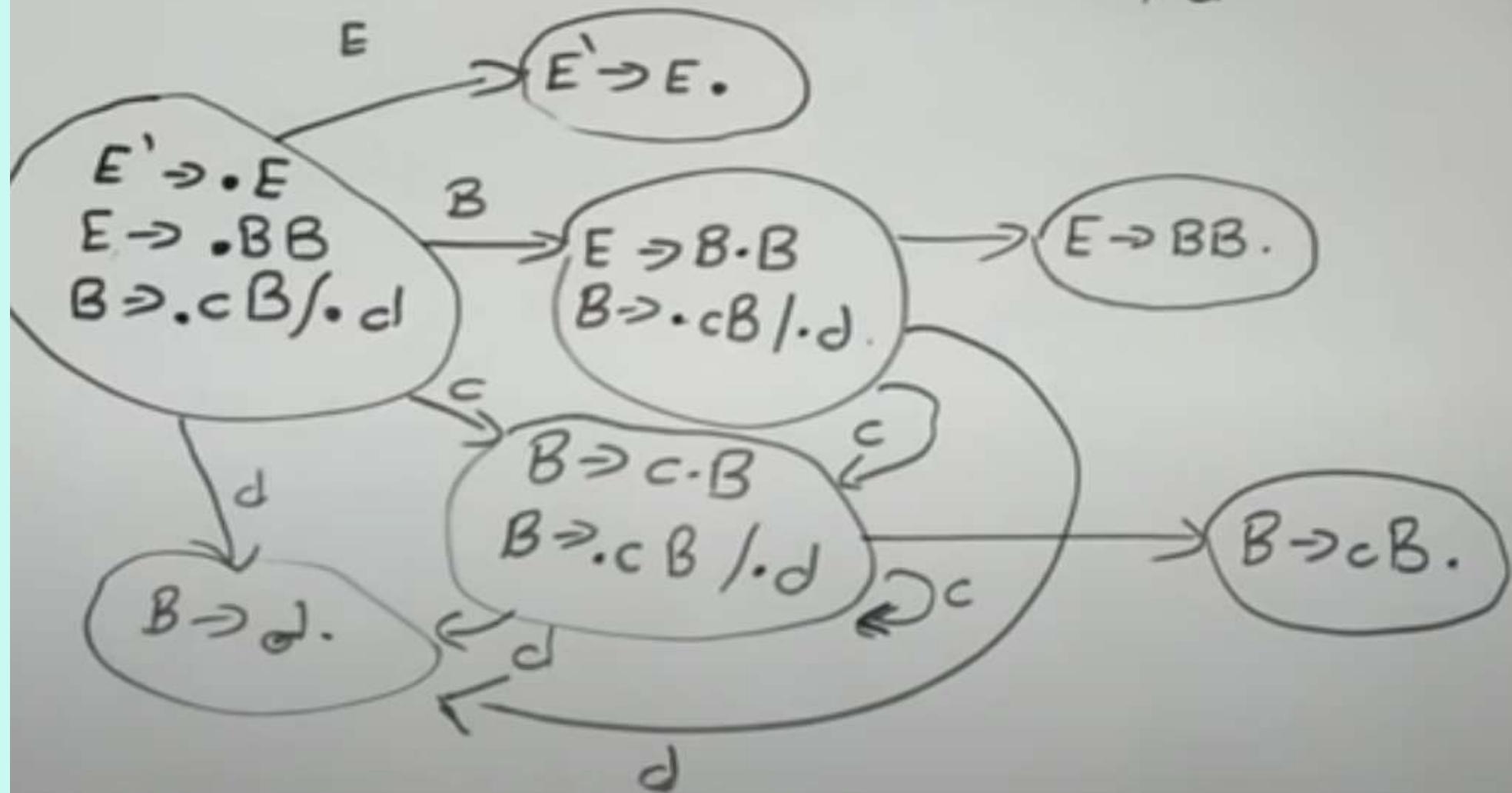
$b \rightarrow$

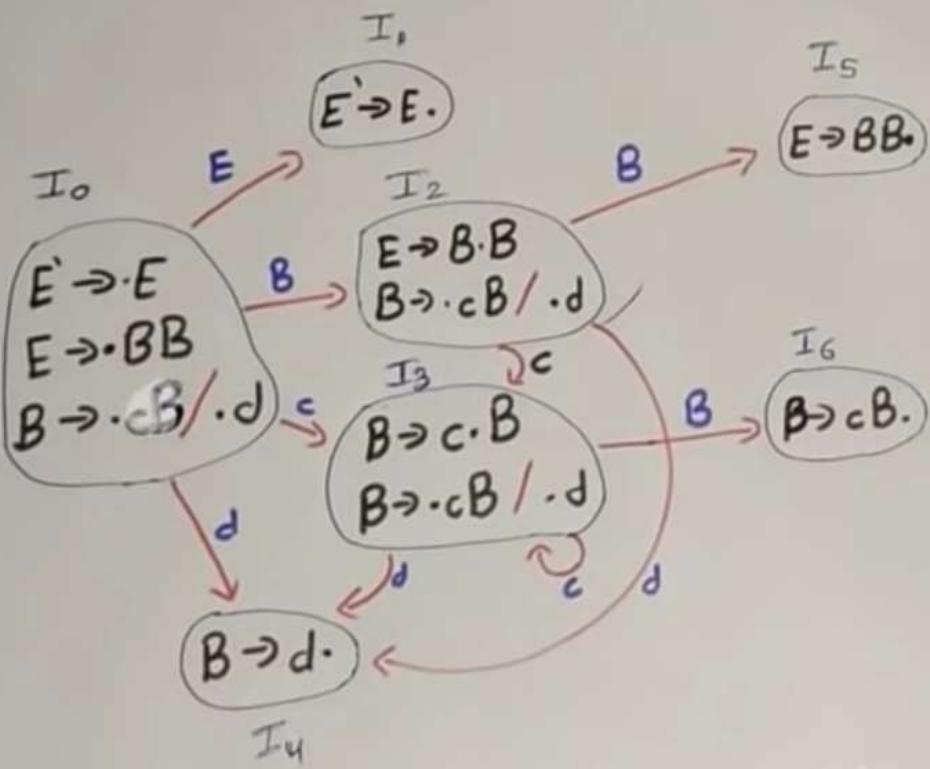
I_4



action Go to

	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	γ_3	γ_3	γ_3		
5	γ_1	γ_1	γ_1		
6	γ_2	γ_2	γ_2		

$E' \rightarrow E$ $E \rightarrow BB$ $B \rightarrow cB / d$ $B \Rightarrow c \cdot B$ $B \Rightarrow \cdot cB / \cdot d$ 



Step 3) Number the Production

$$\begin{aligned}
 &E' \rightarrow E \\
 &E \rightarrow BB \quad ① \\
 &B \rightarrow cB \quad ② / d \quad ③
 \end{aligned}$$

4) Parsing table

State	Action	Goto			
	c	d	\$	E	B
I ₀	s ₃	s ₄		1	2
I ₁					Accpt
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 ₃		

Step 5) Stack implementation

$$\begin{aligned}
 &E' \rightarrow E \\
 &E \rightarrow BB \\
 &B \rightarrow cB \quad / \quad d
 \end{aligned}$$

4) Parsing Table

State	Action	Go to	Stack	Input	Action	State 3
I ₀	s ₃ s ₄	1 2	\$0	ccdd\$	shift c in the stack and Go to	
I ₁	Accept		\$0c3	ddd\$	shift C → s ₃	
I ₂	s ₃ s ₄	5	\$0c3c3	dd\$	shift d → s ₄	
I ₃	s ₃ s ₄	6	\$0c3c3d4	d\$	reduce cC B → d	
I ₄	γ ₃ γ ₃ γ ₃		\$0c3c3B6	d\$	reduce B → cB	
I ₅	γ ₁ γ ₁ γ ₁		\$0c3B6	d\$	reduce B → cB	
I ₆	γ ₂ γ ₂ γ ₃		\$0B2	d\$	shift d → s ₄	
			\$0B2d4	\$	reduce B → d	
			\$0B2B5	\$	reduce E → BB	
			\$0E1	\$	Accept	

Step 5) Stack implementation

$E \rightarrow E$

ccdd\$

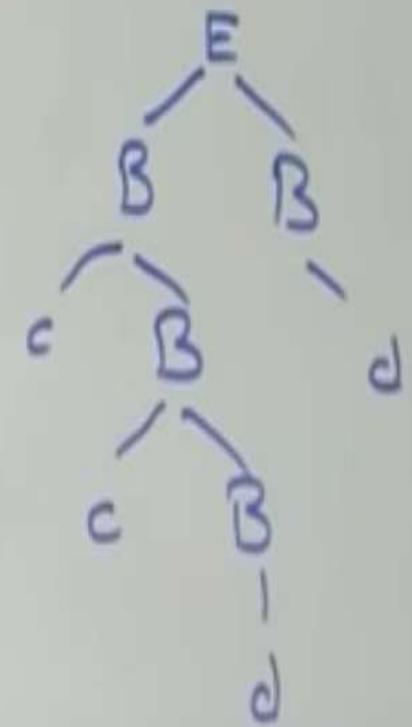
$E \rightarrow BB \textcircled{1}$

$B \rightarrow cB / d$

③

Stack	Input	Action	state 3.
\$0	ccdd \$	Shift c in the stack and Go to state 3.	
\$0c3	ccdd \$	shift c → s3	
\$0c3c3	ccdd \$	shift d → s4	
\$0c3c3d4	d \$	reduce γ_3 $B \rightarrow d$	
\$0c3c3B6	d \$	reduce $B \rightarrow cB$	
\$0c3B6	d \$	reduce $B \rightarrow cB$	
\$0B2	d \$	shift d → s4	
\$0B2d4	\$	reduce $B \rightarrow d$	
\$0B2B5	\$	reduce $E \rightarrow BB$	
\$0E1	\$	Accept	

Step 6) Draw the Parse tree



Constructing SLR Parsing Table

(of an augmented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.

$$C \leftarrow \{I_0, \dots, I_n\}$$

2. Create the parsing action table as follows

- If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $\text{goto}(I_i, a) = I_j$ then $\text{action}[i, a]$ is *shift j*.
- If $A \rightarrow \alpha.$ is in I_i , then $\text{action}[i, a]$ is *reduce A $\rightarrow \alpha$* for all a in $\text{FOLLOW}(A)$ where $A \neq S'$.
- If $S' \rightarrow S.$ is in I_i , then $\text{action}[i, \$]$ is *accept*.
- If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table

- for all non-terminals A , if $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$

Parsing Tables of Expression Grammar

state	id	Action Table						Goto Table		
		+	*	()	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4				9	3	
7	s5			s4					10	
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$I_0: S' \rightarrow .S$

$S \rightarrow .L=R$

$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .id$

$R \rightarrow .L$

$I_1: S' \rightarrow S.$

$I_2: S \rightarrow L.=R$
 $R \rightarrow L.$

$I_3: S \rightarrow R.$

$I_6: S \rightarrow L=.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .id$

$I_9: S \rightarrow L=R.$

$I_4: L \rightarrow *.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .id$

$I_7: L \rightarrow *R.$

$I_8: R \rightarrow L.$

Problem

$\text{FOLLOW}(R) = \{=, \$\}$

= ↗ shift 6

↘ reduce by $R \rightarrow L$

shift/reduce conflict

$I_5: L \rightarrow id.$

Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

$I_0: S' \rightarrow .S$
 $S \rightarrow .AaAb$
 $S \rightarrow .BbBa$
 $A \rightarrow .$
 $B \rightarrow .$

Problem

$\text{FOLLOW}(A) = \{a, b\}$

$\text{FOLLOW}(B) = \{a, b\}$

a
 └──▶ reduce by $A \rightarrow \epsilon$
 └──▶ reduce by $B \rightarrow \epsilon$

reduce/reduce conflict

b
 └──▶ reduce by $A \rightarrow \epsilon$
 └──▶ reduce by $B \rightarrow \epsilon$

reduce/reduce conflict

$$E \rightarrow T+E / T$$
$$T \rightarrow i$$

- i) check Grammar is $LR(0)$ or not
- ii) check Grammar is $SLR(1)$ or not

Step 1) Augment the given grammar

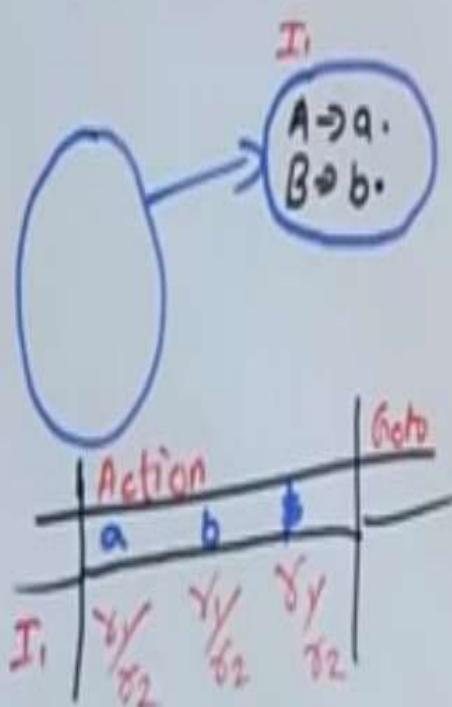
$$E' \rightarrow E$$
$$E \rightarrow T+E / T$$
$$T \rightarrow i$$

Step 2) Draw conical collection of $LR(0)$ item

Conflict

RR

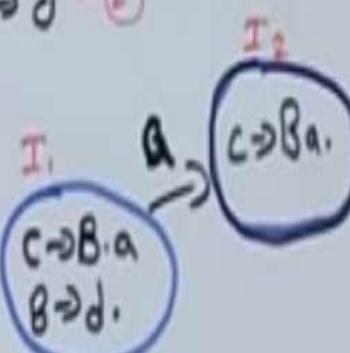
$$\begin{aligned} A &\rightarrow a \quad -① \\ B &\rightarrow b \quad -② \\ C &\rightarrow aB \quad -③ \end{aligned}$$



SR

$$C \rightarrow \beta a \quad -①$$

$$B \rightarrow d \quad -②$$



Action		Goto
a	d	\$
I ₁		
I ₂		

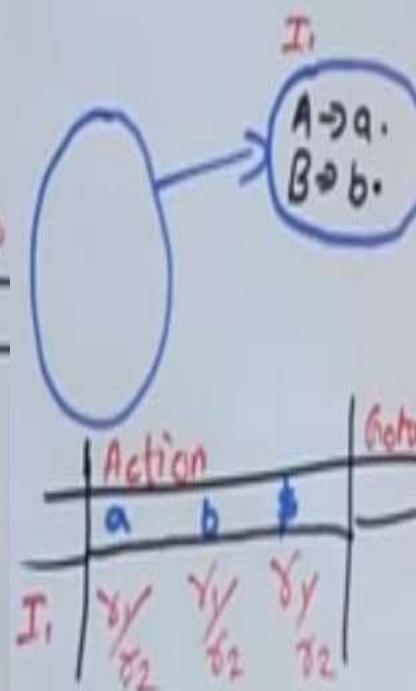
Conflict

RR

$$A \rightarrow a \quad -①$$

$$B \rightarrow b \quad -②$$

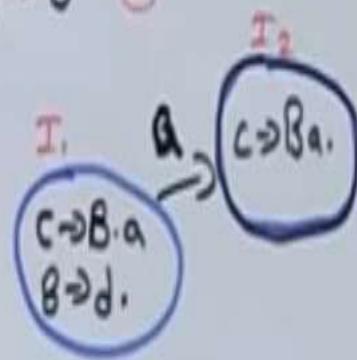
$$C \rightarrow aB \quad -③$$



SR

$$C \rightarrow \beta a \quad -①$$

$$B \rightarrow d \quad -②$$

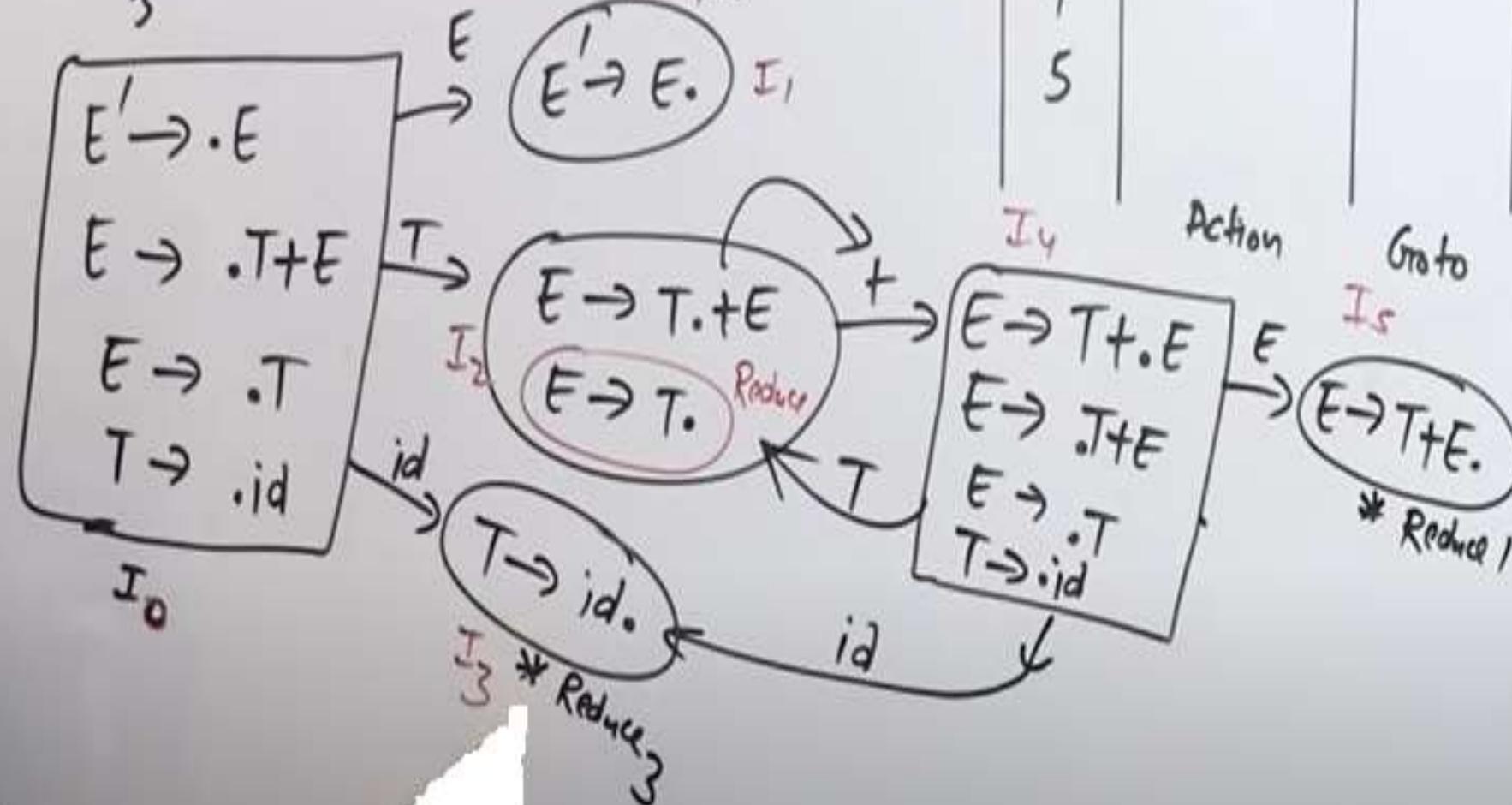


Action		Goto
a	d	\$
I ₁	γ_1/γ_2	γ_1/γ_2
I ₂		

LR(0) Parsing table

$$E \rightarrow T+E \quad / \quad T$$

$$T \rightarrow id$$



State	id	+	\$	E	T
0	S_3			1	2
1				-	
2	g_{12}	S_4/g_{12}	g_{12}		
3					
4					
5					

LR(0) Parsing table

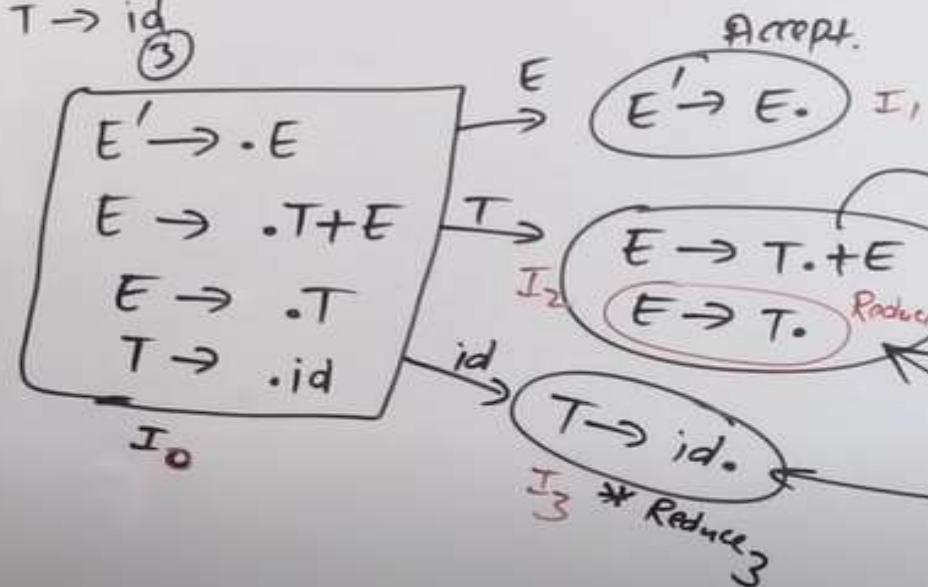
$$E \rightarrow T+E / T$$

$$T \rightarrow id$$

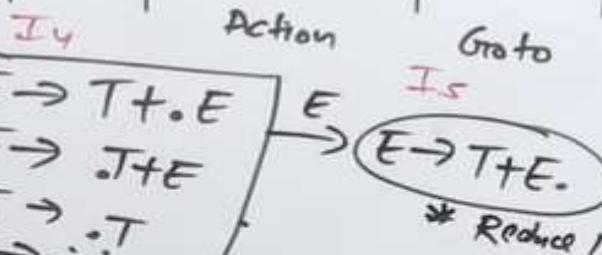
$E' \rightarrow \cdot E$	E
$E \rightarrow \cdot T+E$	T
$E \rightarrow \cdot T$	$+ E$
$T \rightarrow \cdot id$	

I₀

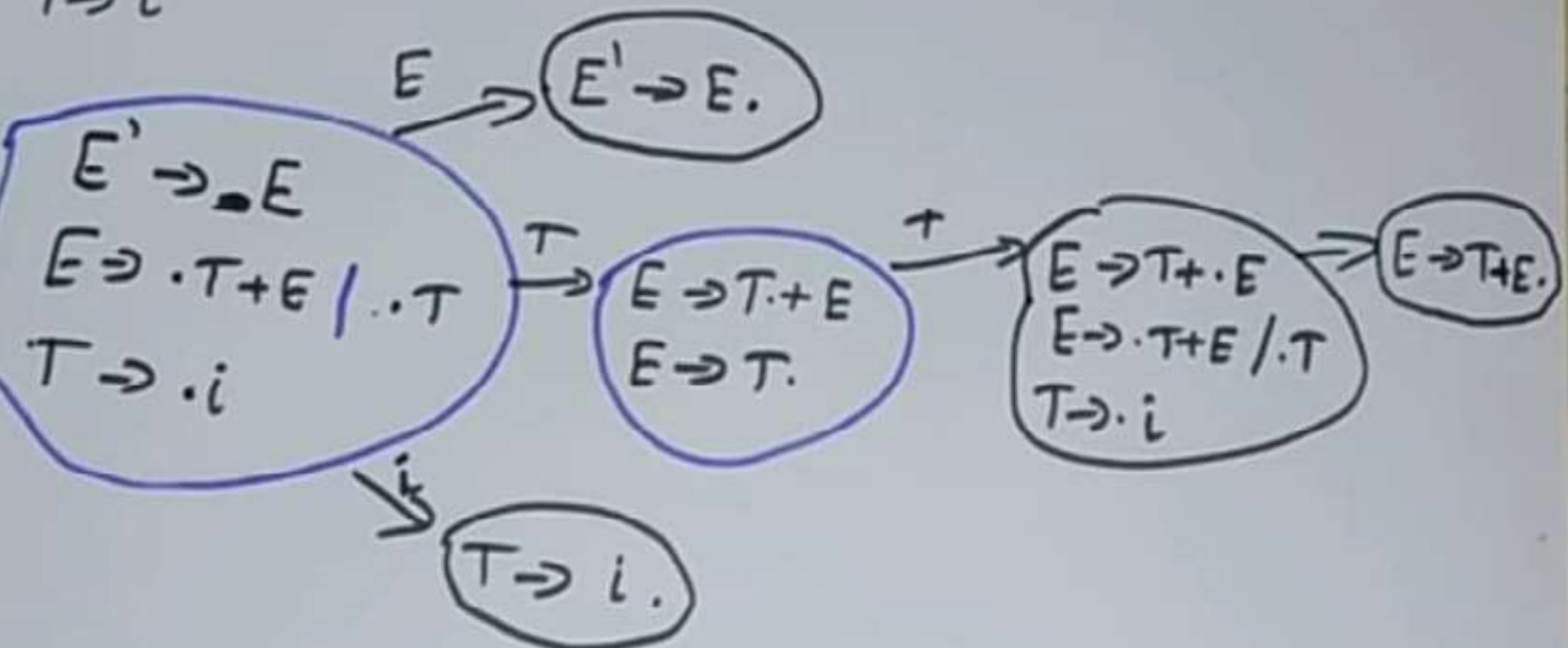
LR(0) CI



State	id	+	\$	E	T
0	S_3			1	2
1		S_3		-	
2	g_{12}		S_4/g_{12}	g_{12}	
3	g_{13}		g_{13}	g_{13}	
4					
5	g_{11}	g_{11}	g_{11}	5	2



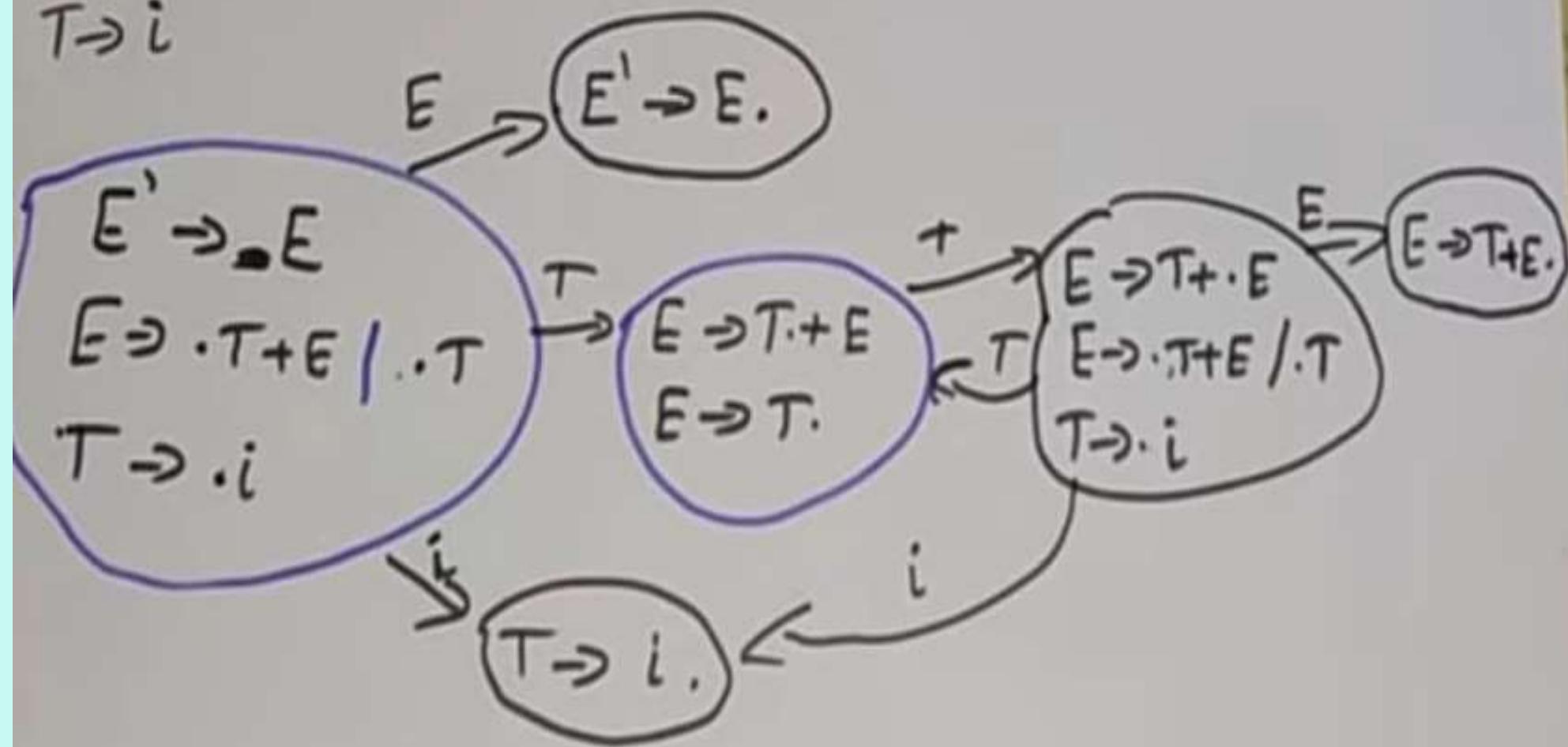
State	id	+	\$	E	T
0	S_3			1	2
1		S_3		-	
2	g_{12}		S_4/g_{12}	g_{12}	
3	g_{13}		g_{13}	g_{13}	
4	S_3				
5	g_{11}	g_{11}	g_{11}	5	2

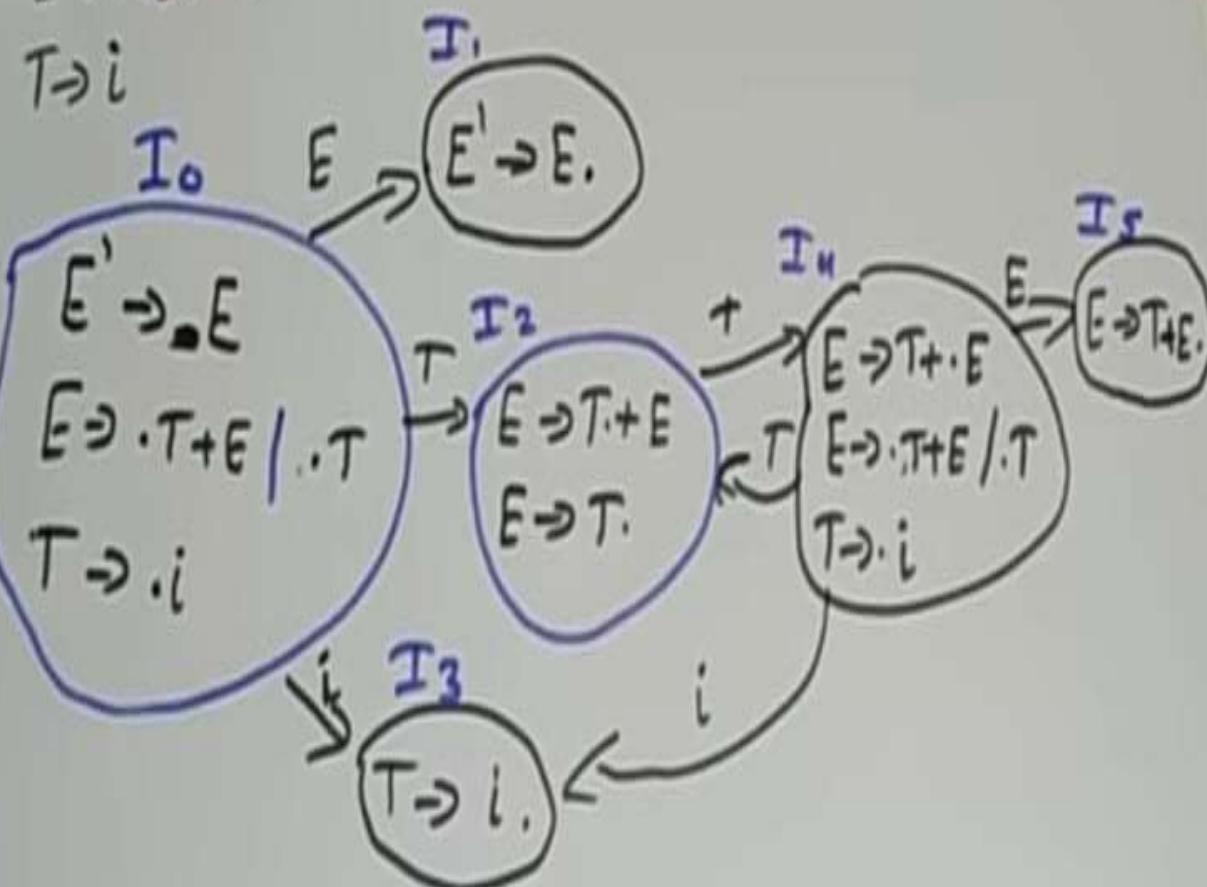
$E' \rightarrow E$ $E \rightarrow T+E / .T$ $T \rightarrow i$ 

$E' \rightarrow E$

$E \rightarrow T+E / .T$

$T \rightarrow i$



$E' \rightarrow E$ $E \rightarrow T+E / T$ $T \rightarrow i$ 

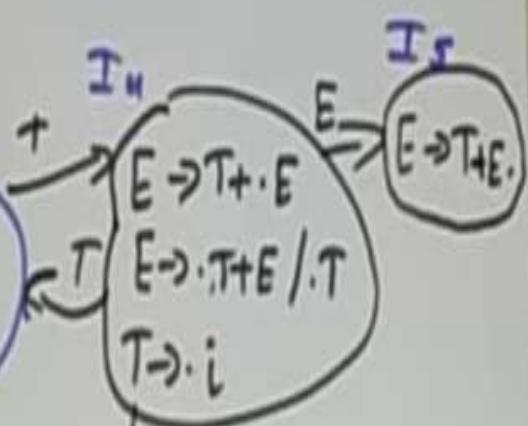
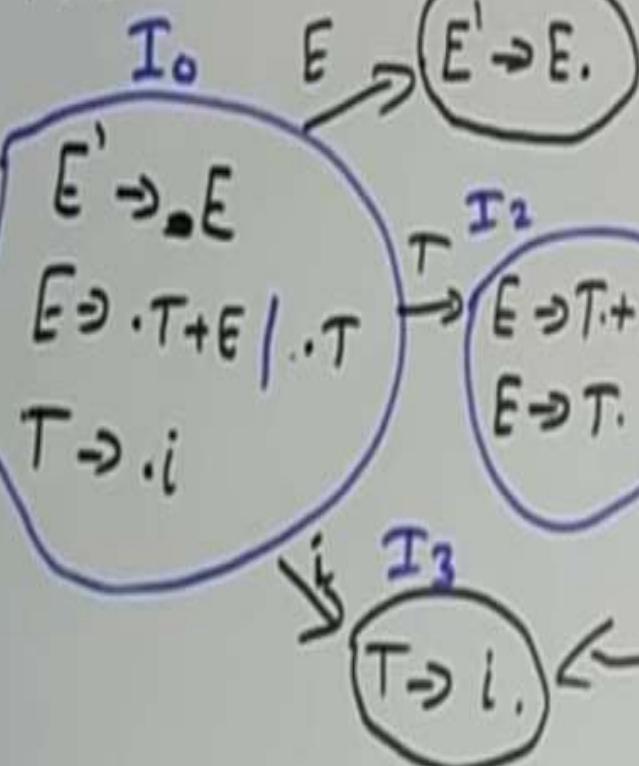
State	Actions			Goto	
.	+	i	\$	E	T
I_0			s_3	1	2
I_1					Accept
I_2	s_4 / γ_1	γ_2	γ_2		
I_3	γ_3	γ_3	γ_3		
I_4			s_3	5	2
I_5	γ_1	γ_1	γ_1		

 $E' \rightarrow E$ $E \rightarrow T+E^{\textcircled{1}} / T^{\textcircled{2}}$ $T \rightarrow i$

③

$E' \rightarrow E$ ①
 $E \rightarrow T+E / T-T$ ②

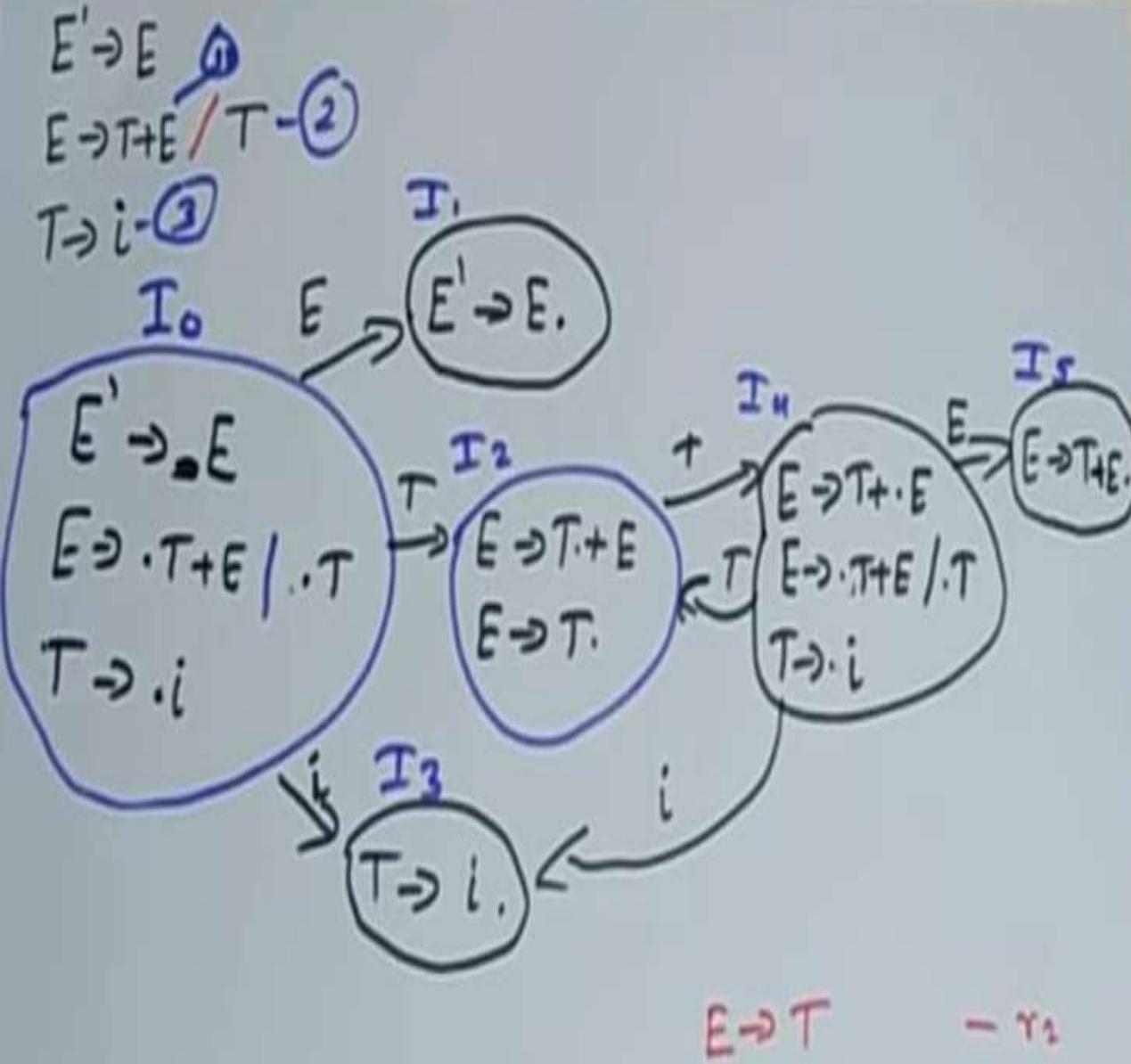
$T \rightarrow i -$ ③



State	Actions	Info
I_0	$+$ i $\$$	E T
I_1		1 2
I_2		S_4
I_3		
I_4	S_3	5 2
I_5		

$$\text{follow}(T) = \{ +, \$ \}$$

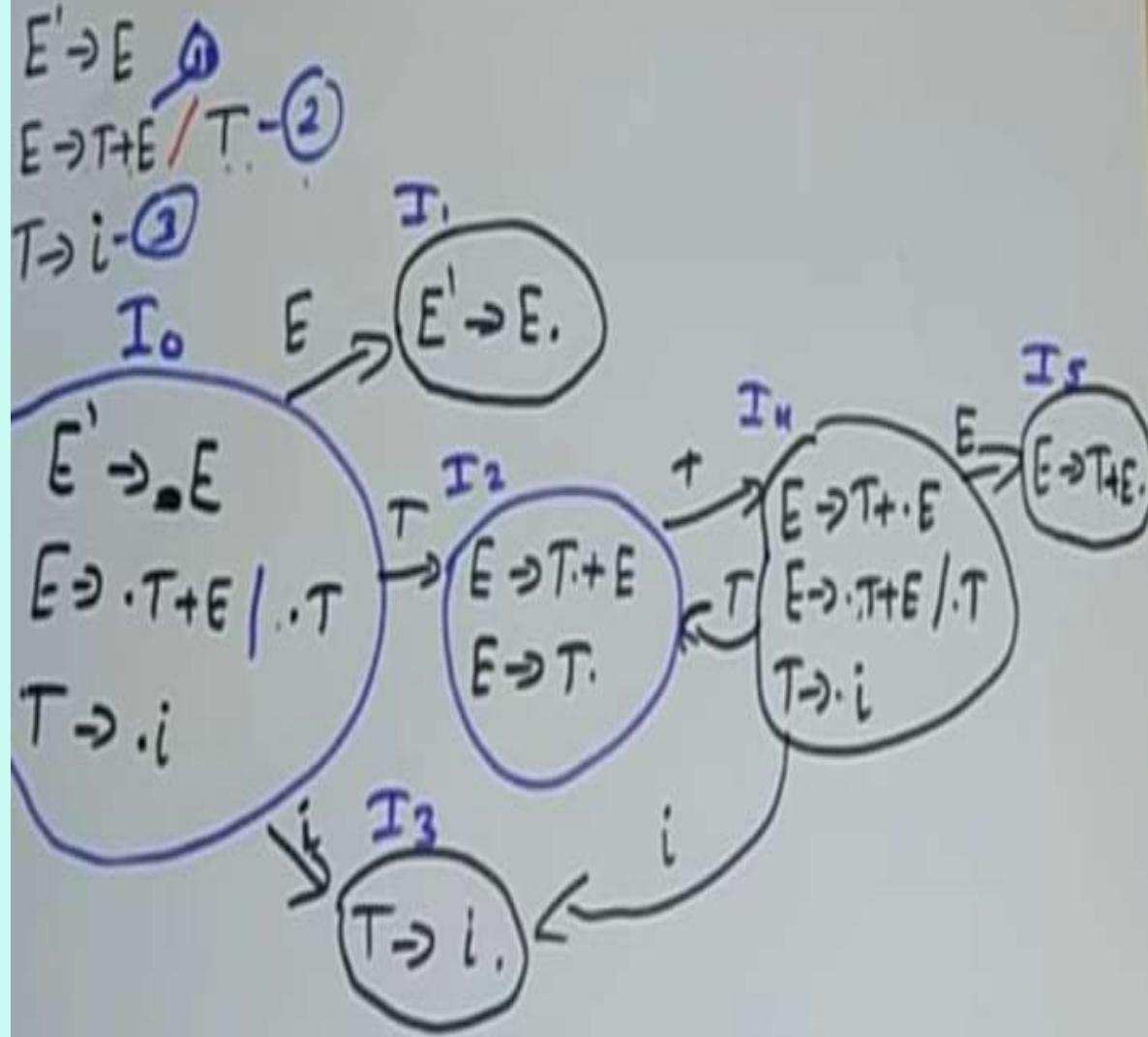
$$\text{follow}(E) = \{ \$ \}$$



State	Actions			Goto	
	+	i	\$	E	T
I_0					
I_1					Accept
I_2					
I_3					
I_4					
I_5					

$\text{follow}(T) = \{ +, \$ \}$

$\text{follow}(E) = \{ \$ \}$

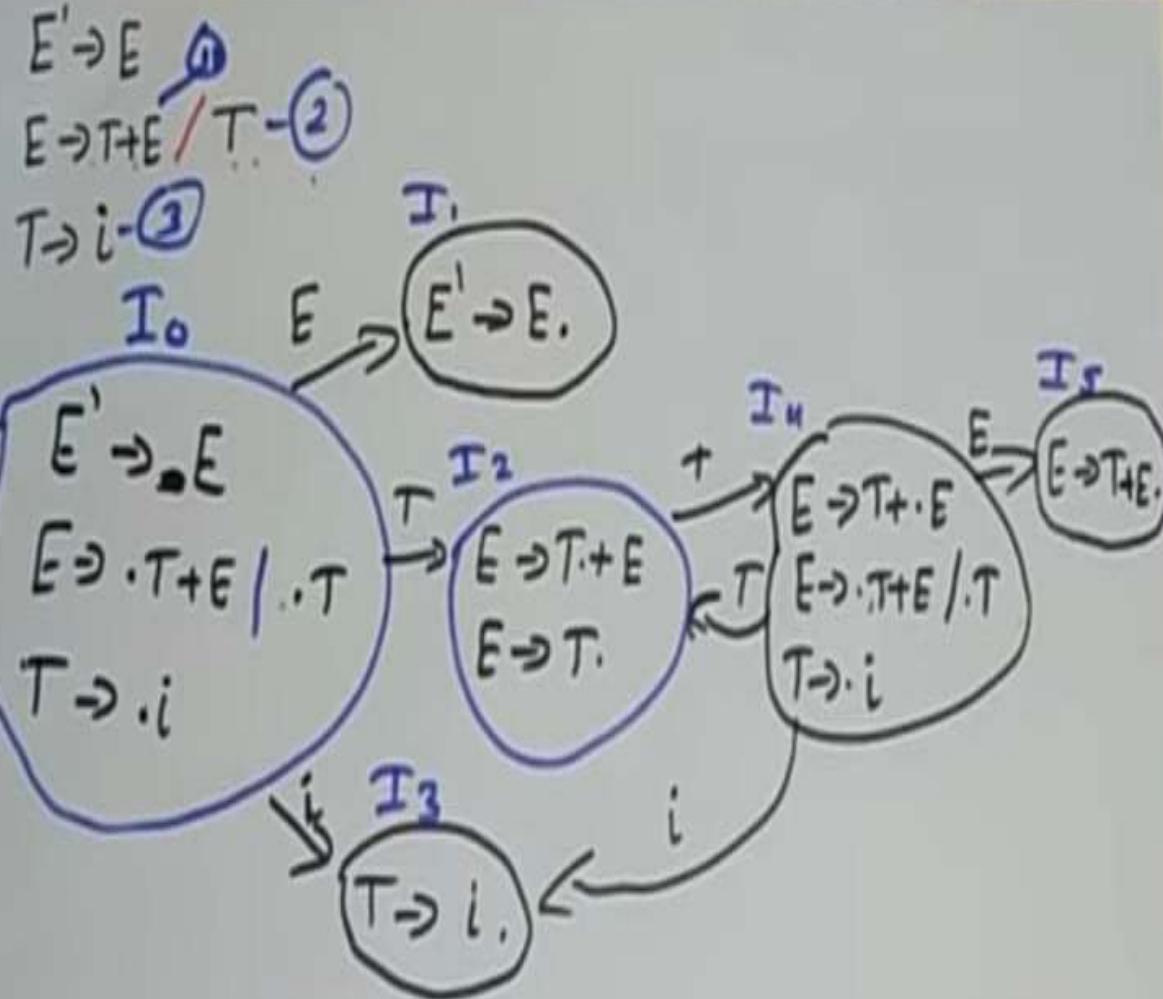


$E \rightarrow T$ - γ_2
 $T \rightarrow i$

State	Actions			Info	
	+	i	\$	E	T
I_0	s_3			1	2
I_1				Accept	
I_2	s_4			γ_2	
I_3					
I_4	s_3			5	2
I_5					

$$\text{follow}(T) = \{ +, \$ \}$$

$$\text{follow}(E) = \{ \$ \}$$



$E \rightarrow T$ $- \gamma_1$
 $T \rightarrow i \cdot$
 $E \rightarrow T+E$

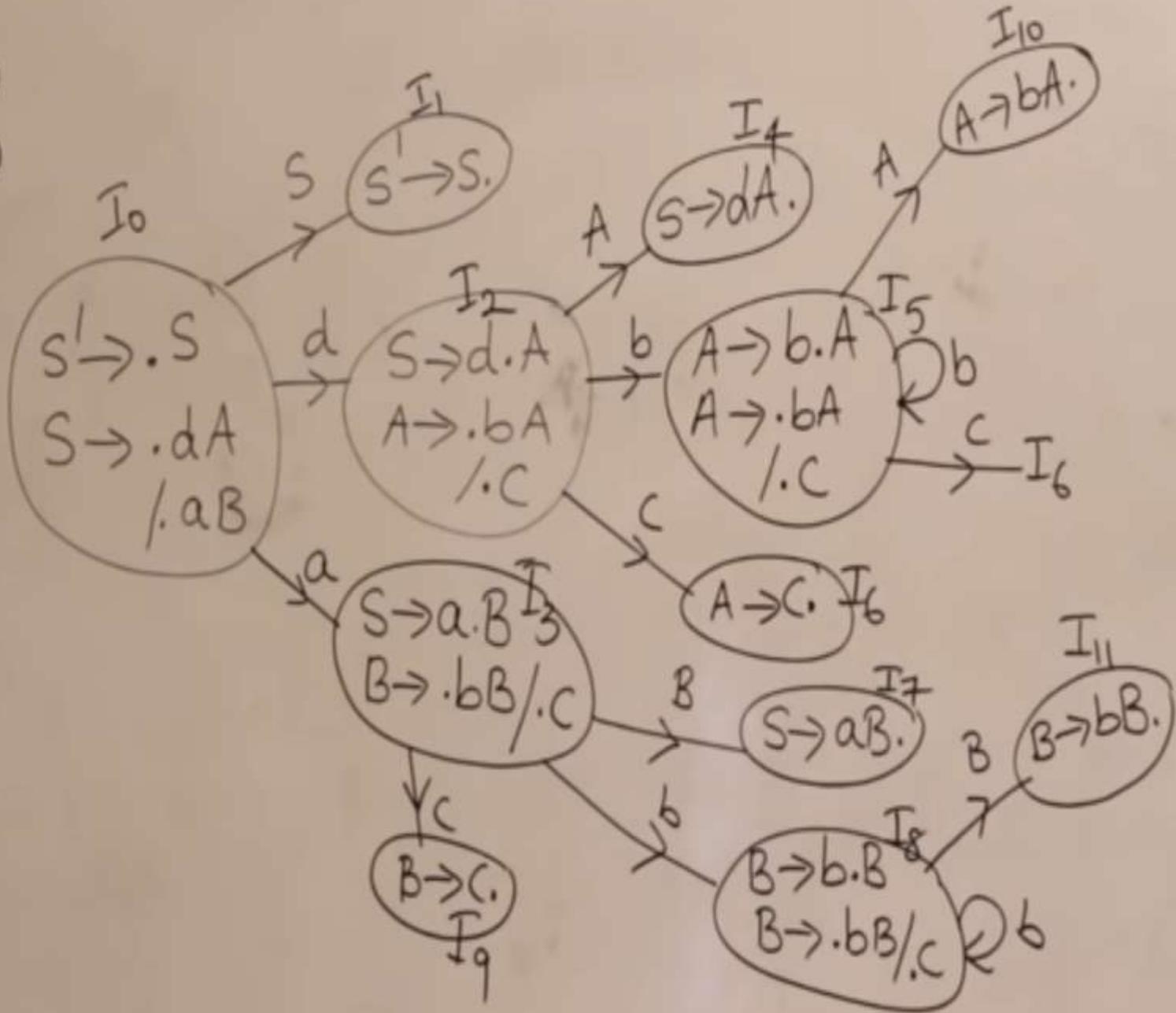
State	Actions			Final
	+	i	\$	E T
I_0	s_3			1 2
I_1				Accept
I_2	s_4			γ_2
I_3		γ_3		γ_3
I_4	s_3			5 2
I_5			γ_1	

$$\text{follow}(T) = \{ +, \$ \}$$

$$\text{follow}(E) = \{ \$ \}$$

- (i) LL(1)
- (ii) LR(0)
- (iii) SLR(1)

$S \rightarrow dA$
 $/aB$
 $A \rightarrow bA/c$
 $B \rightarrow bB/c$



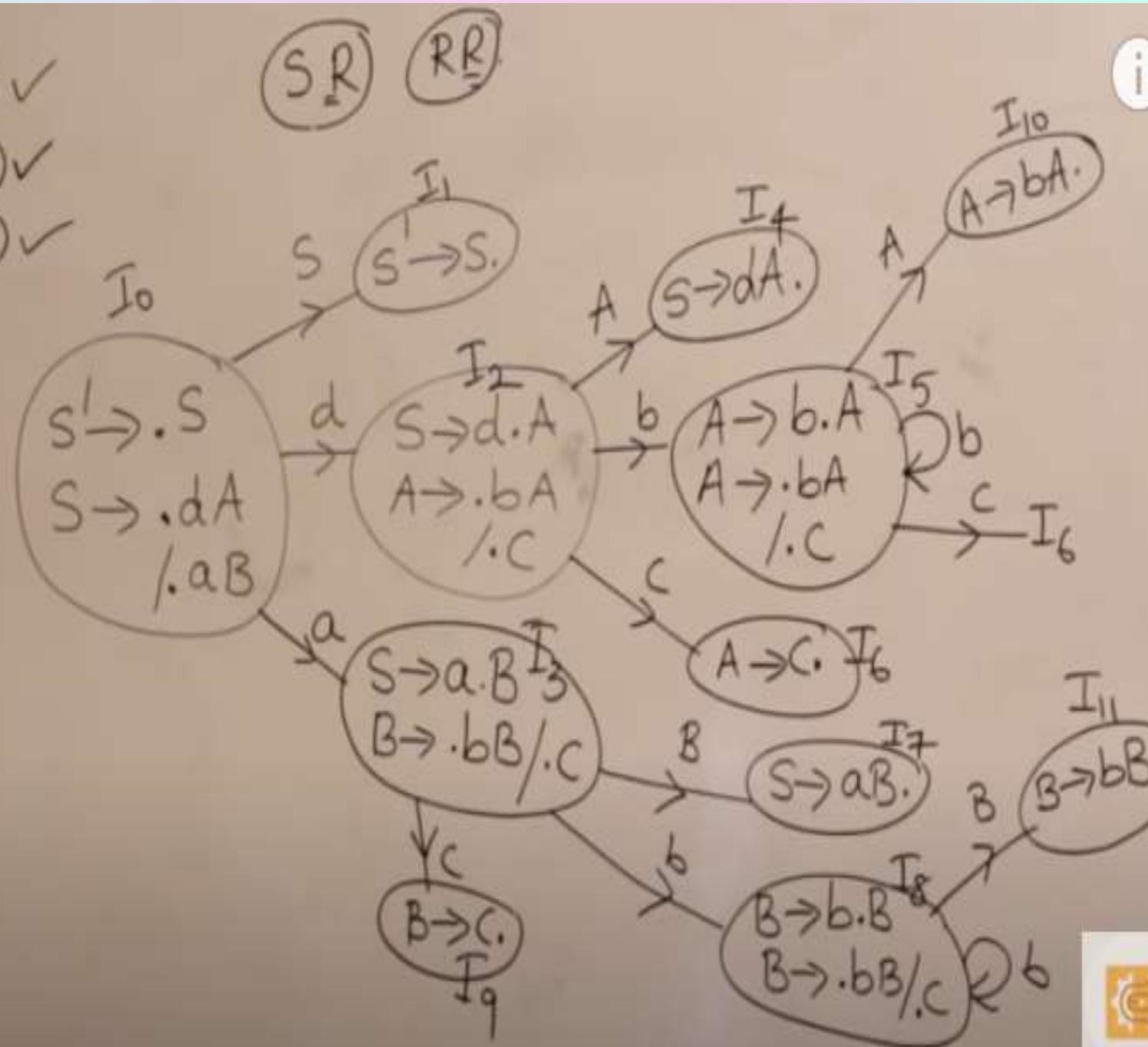
(i) LL(1) ✓

(ii) LR(0) ✓

(iii) SLR(1) ✓

S.R
R.R.

$S \rightarrow dA$
 $/aB$
 $A \rightarrow bA/c$
 $B \rightarrow bB/c$



$\{a\}$ $\{a\}$
 $S \rightarrow A/a$ $LR(0)^*$
 $A \rightarrow a$

$LL(1)$

$LR(0)^*$

$SLR(1)X$

S

$S' \rightarrow .S$

$S \rightarrow .A/a$

$A \rightarrow .a$

$S \rightarrow S.$

a

$S \rightarrow a.$

A

$S \rightarrow A.$

\$

\$

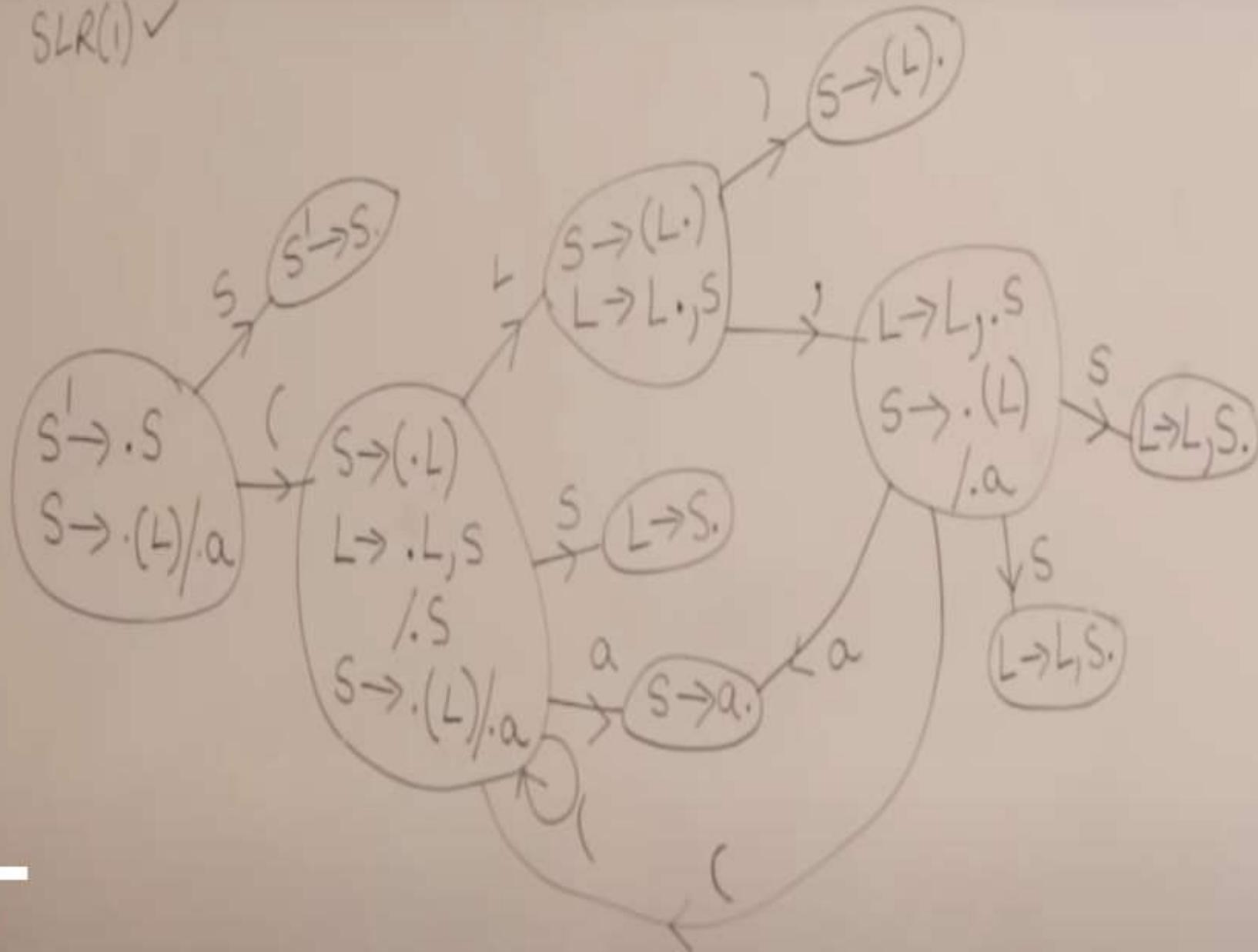
LL(1) ✓

LR(0) ✓

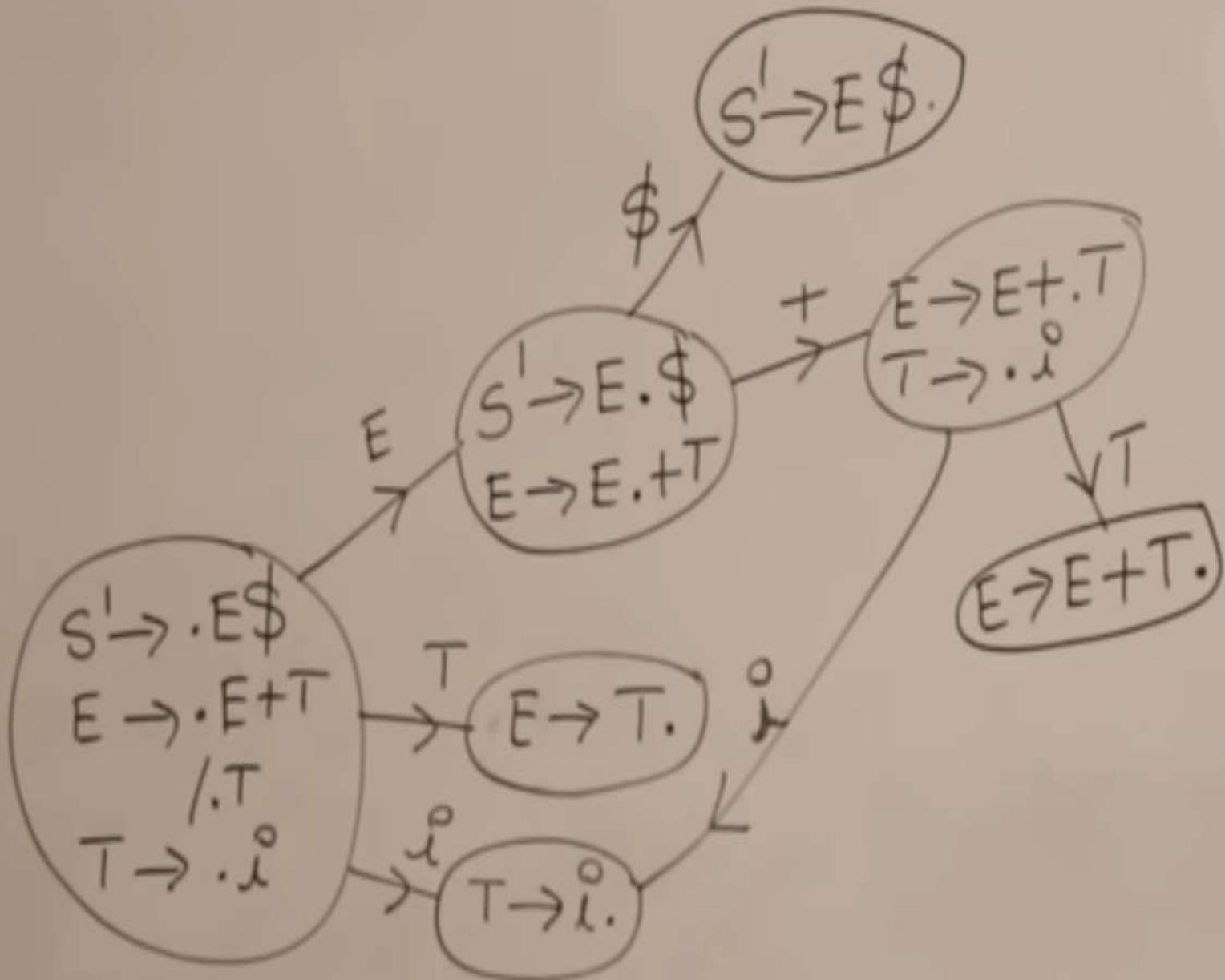
SLR(1) ✓

$S \rightarrow (L)/a$

$L \rightarrow L, S$
/s



$$\begin{array}{l} E \rightarrow E + T \\ /T \\ T \rightarrow i^o \end{array}$$

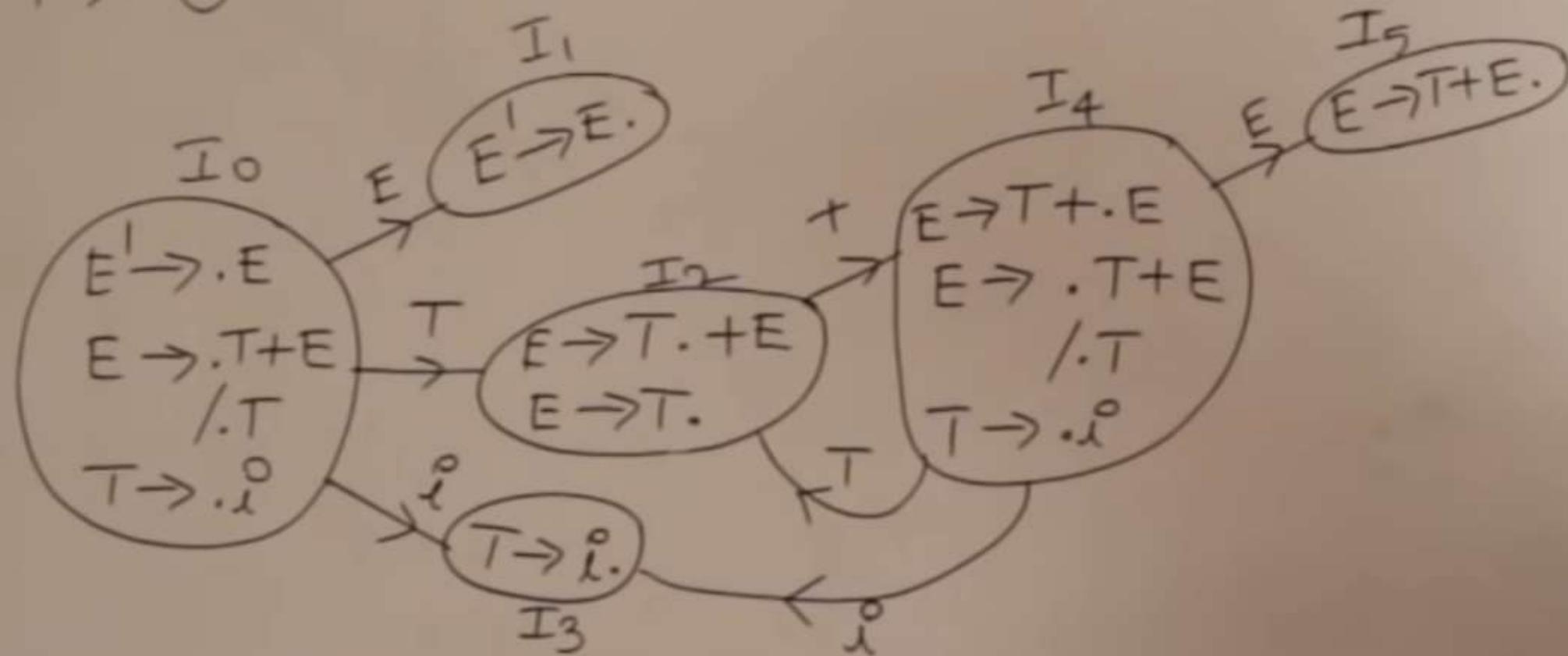


LL(1) X

$$E \rightarrow T + E \quad (1)$$

$$T \rightarrow i^o \quad (2)$$

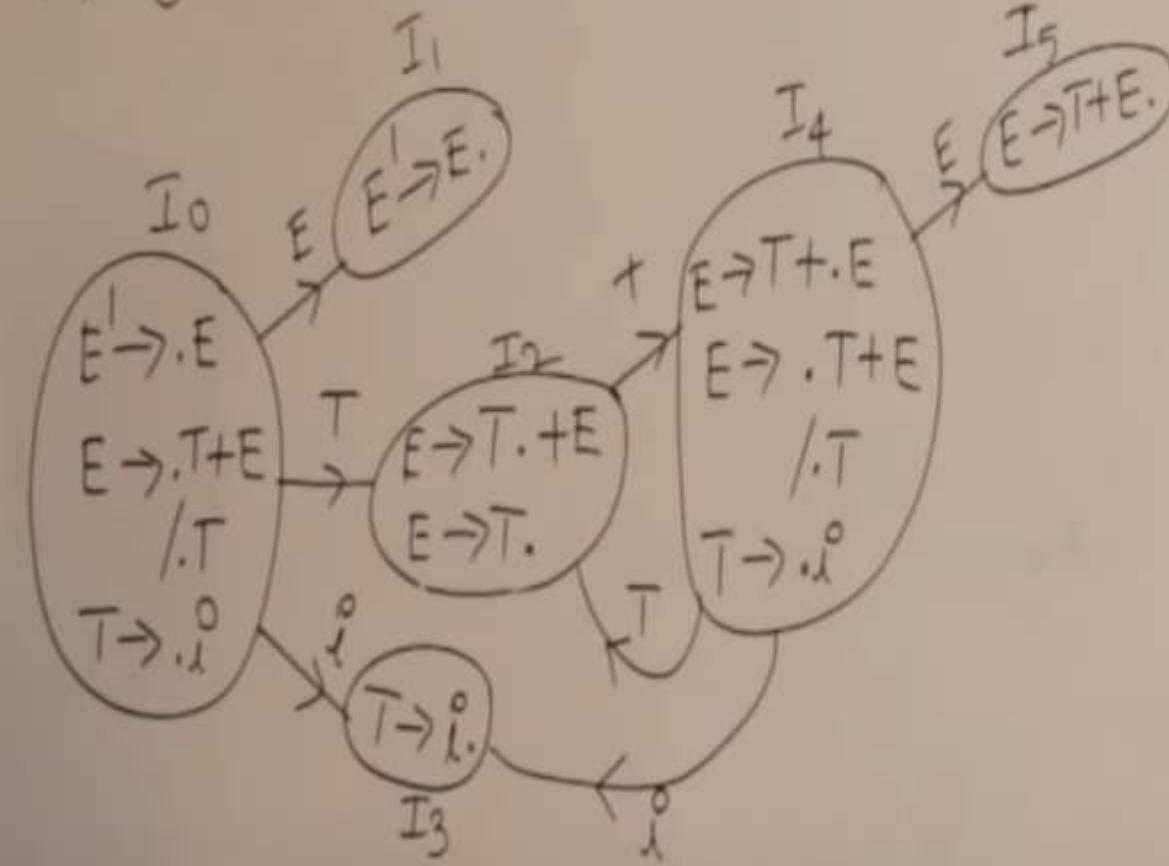
$$T \rightarrow \cdot i^o \quad (3)$$



$$E \rightarrow T + E \circ$$

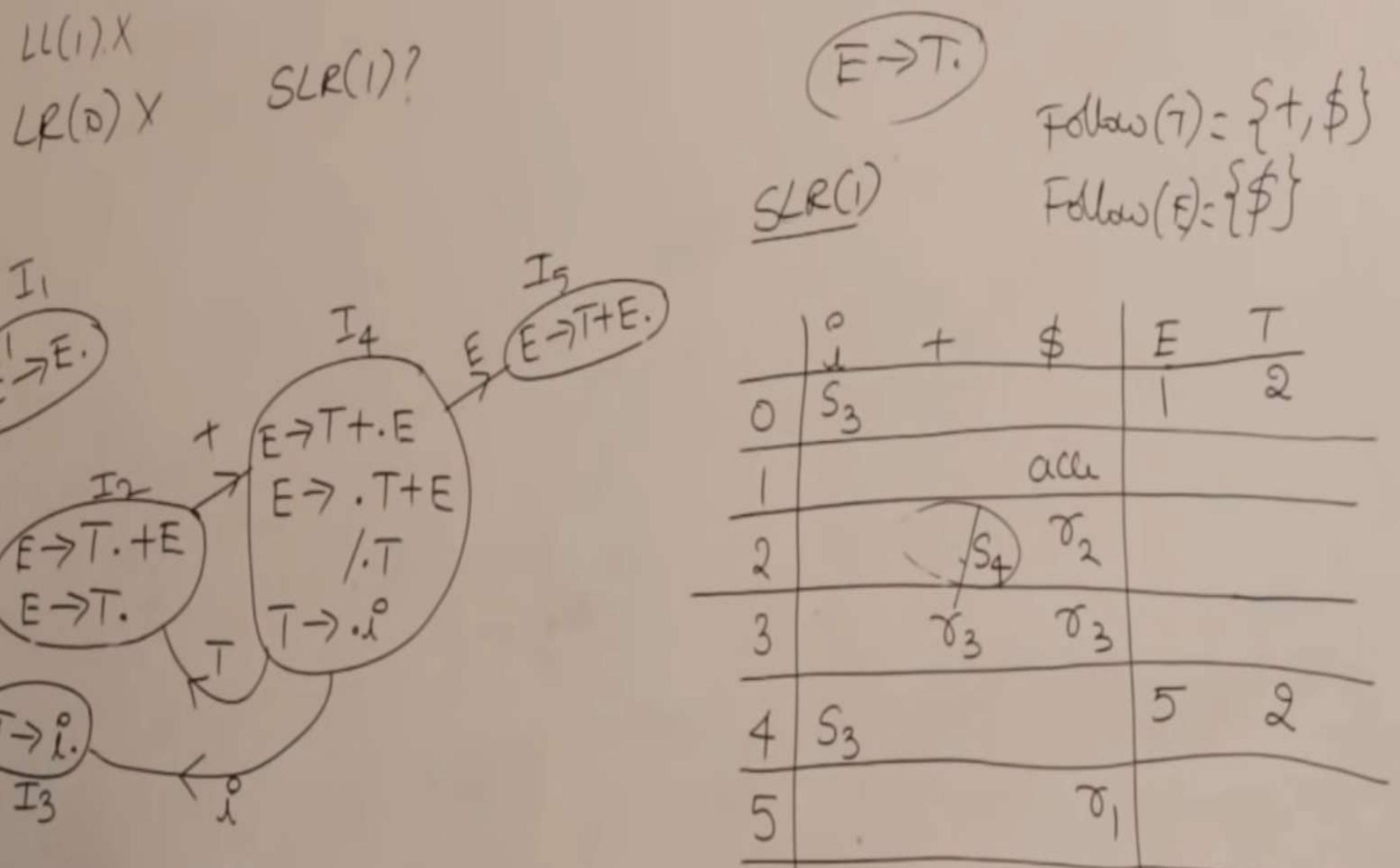
$$T \rightarrow \lambda^o \circ$$

$$T \rightarrow \lambda^o \circ$$



LL(0) pair

	λ^o	+	\$	E	T
0	S_3			1	2
1					acc
2	τ_2	δ_2	S_4	τ_2	
3	τ_3	τ_3	τ_3	τ_3	
4	S_3			5	2
5	τ_1	τ_1	τ_1		



$$E \rightarrow E + T \quad (1)$$

/ T

$$T \rightarrow T F \quad (2)$$

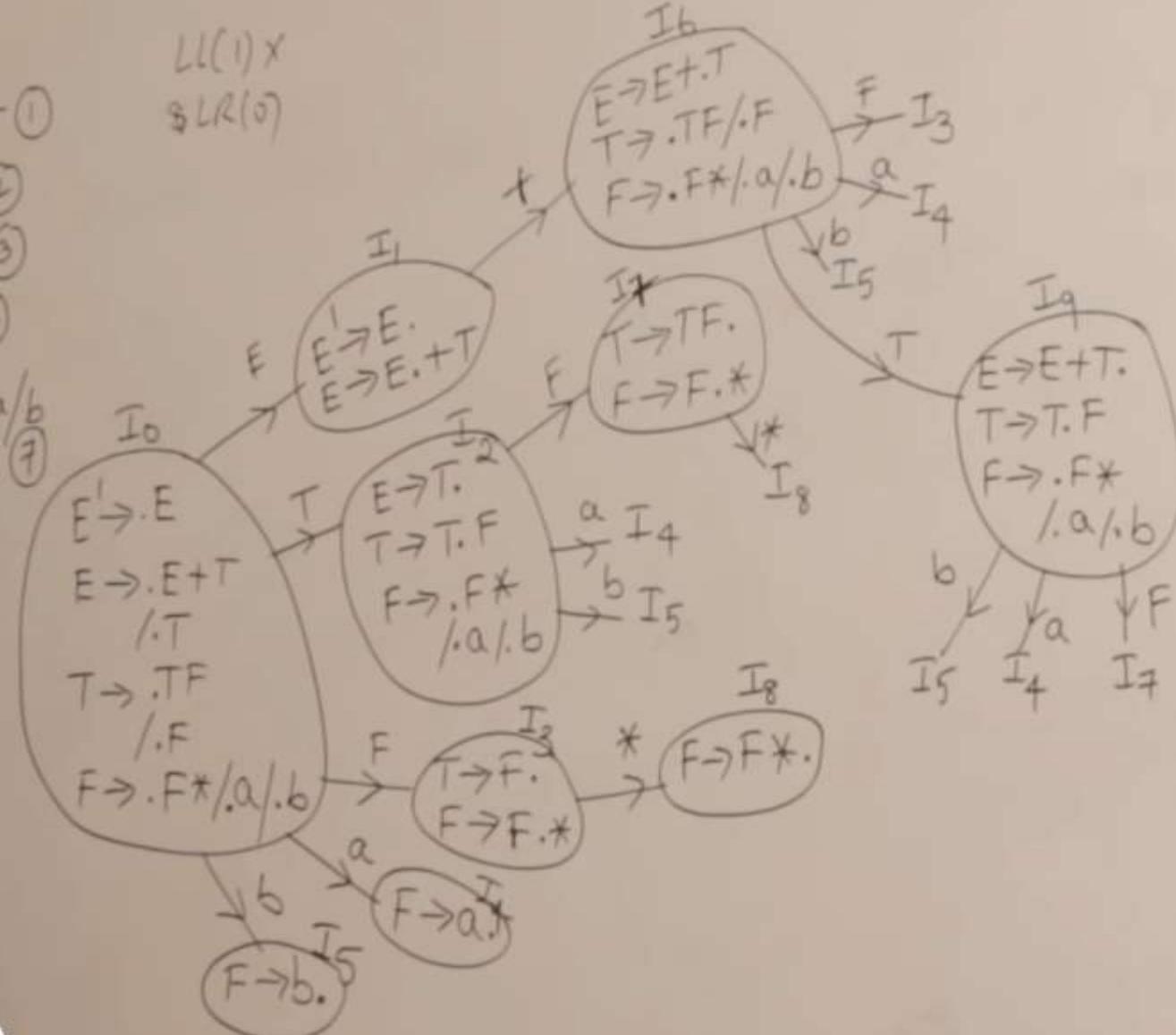
/ F

$$F \rightarrow F * / a / b$$

(3) (4) (5)

LL(1) X

SLR(0)



$\text{Follow}(E) = \{+, \$\}$

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

$\text{Follow}(F) = \{+, *, \$, a, b\}$

$$E \rightarrow E + T \quad ①$$

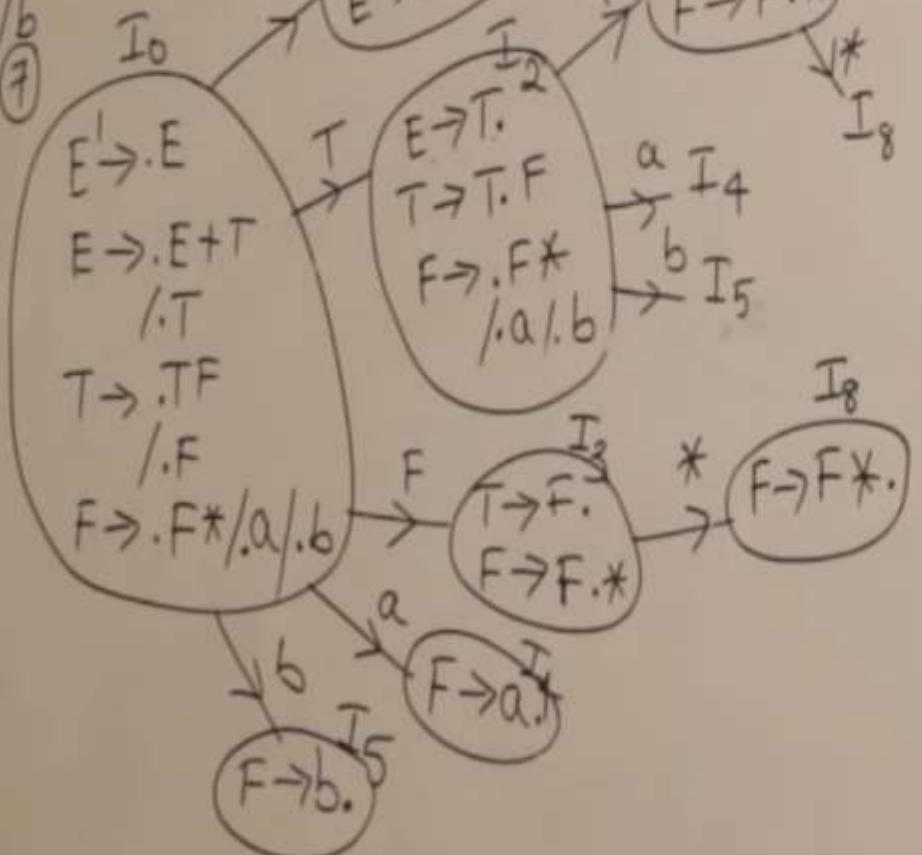
1/T ②

$$T \rightarrow TF \quad ③$$

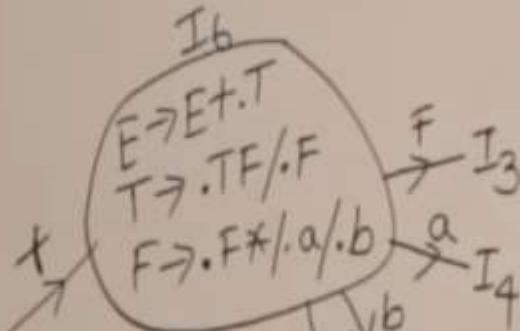
/F ④

$$F \rightarrow F^*/a/b \quad ⑤$$

⑥ ⑦



LL(1) X
SLR(0) X
SLR(1)



Follow(E) = {+, \$}

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

Follow(F) = {+, *, \$, a, b}

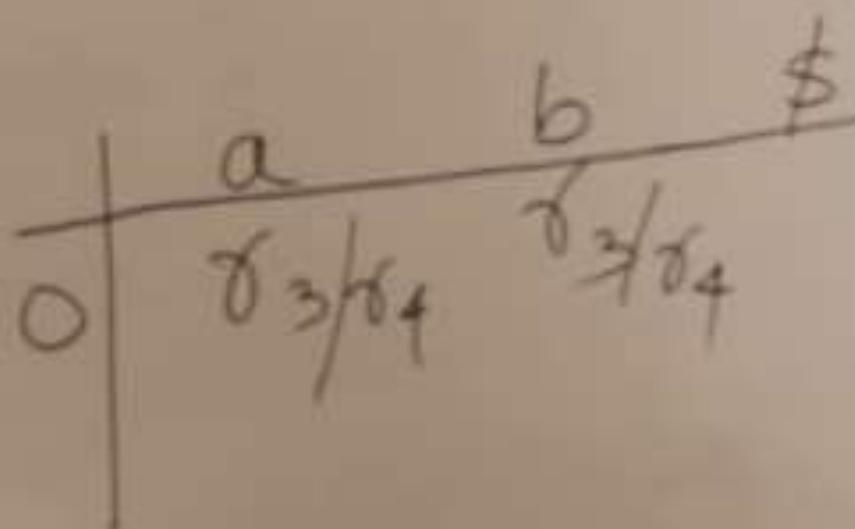
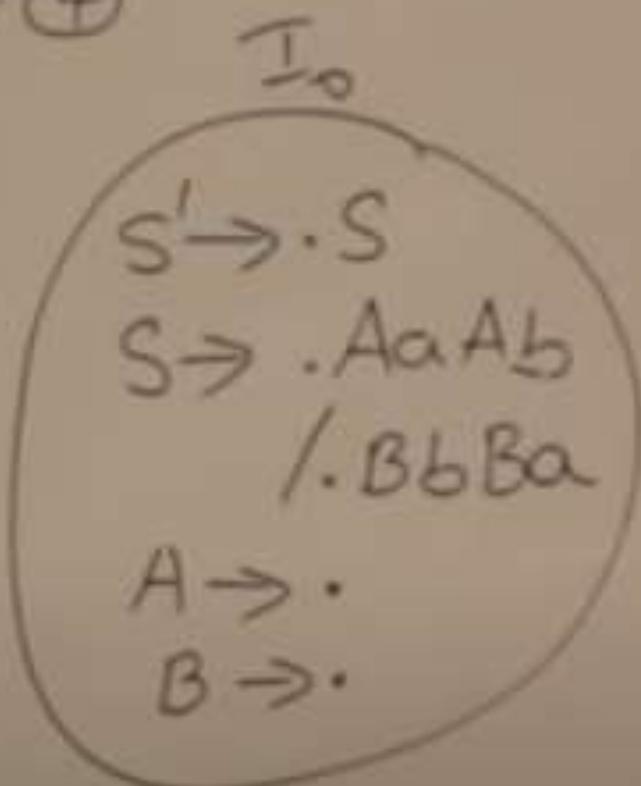
	a	b	+	*	\$
2	S4	S5	T2		T2
3	T4	T4	T4	S8	T4
7	T3	T3	T3	S8	T3
9	S4	S5	T1		T1

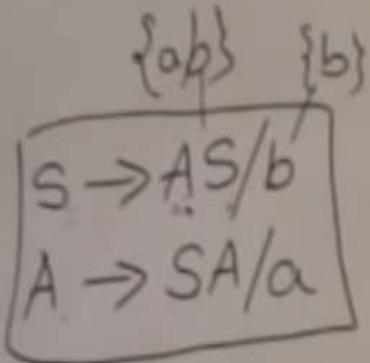
$$A \rightarrow \cdot C \quad A \rightarrow \cdot$$

$$A \rightarrow C \cdot$$

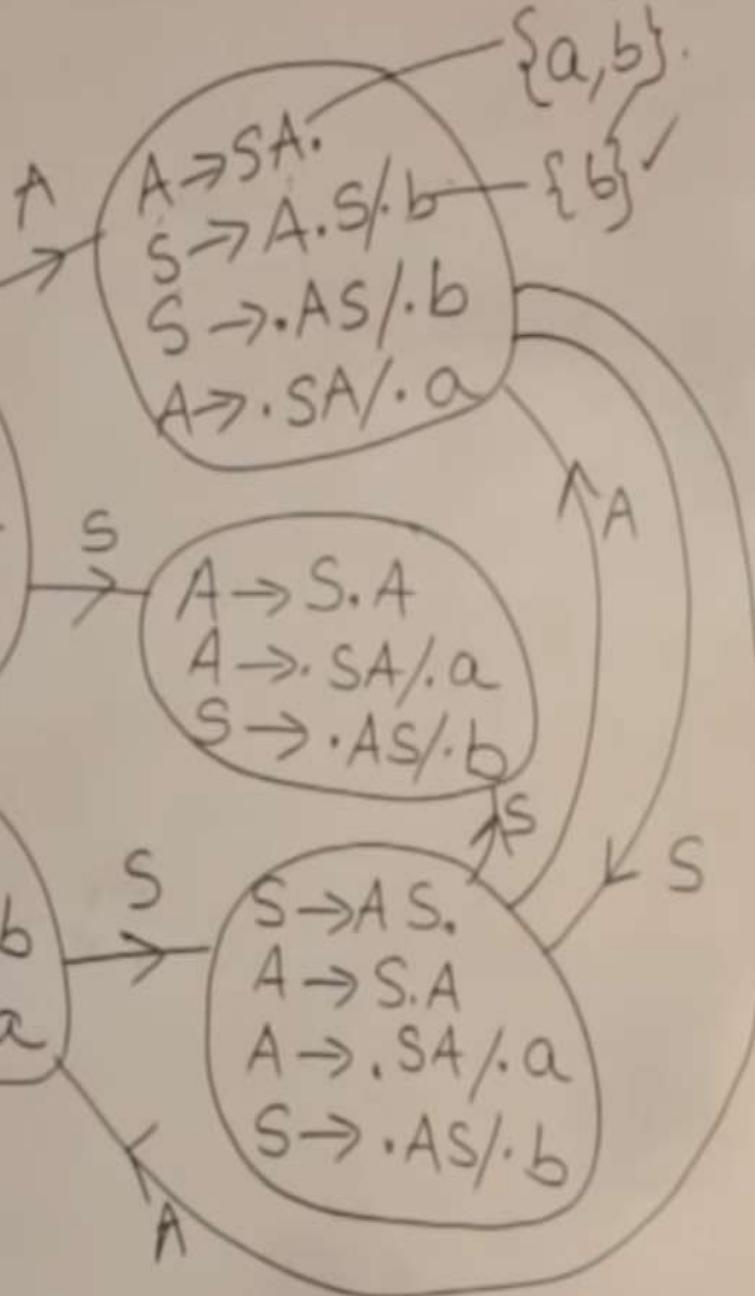
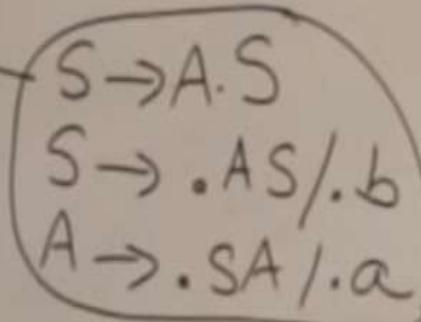
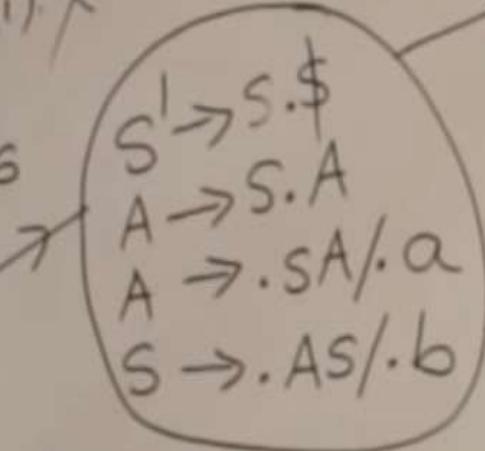
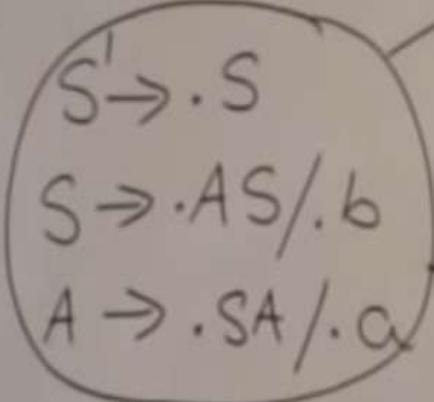
$LL(1)$
 $LR(0) \times$
 $SLR(1) \times$

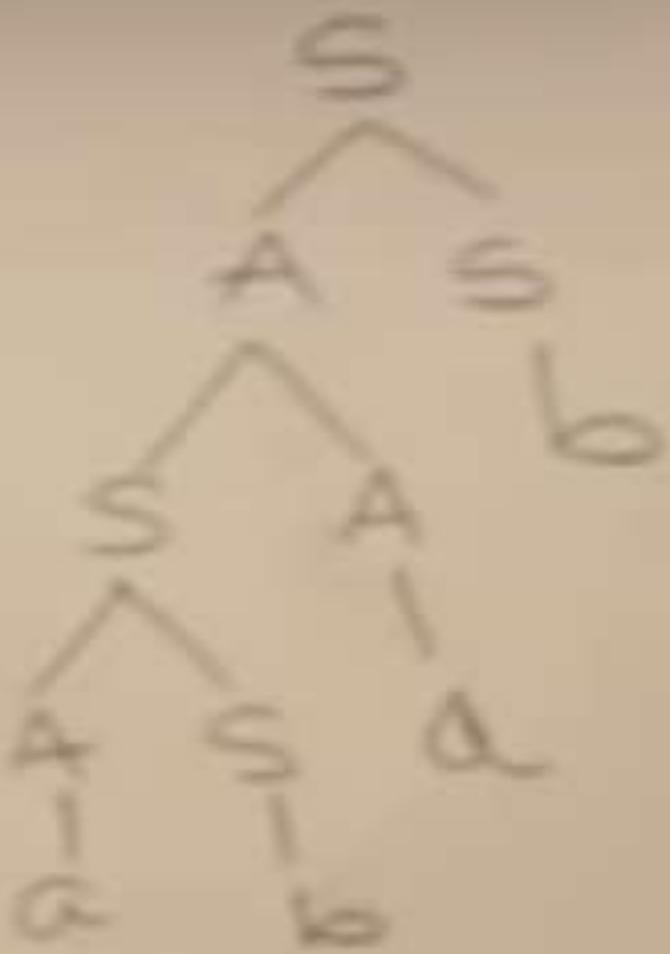
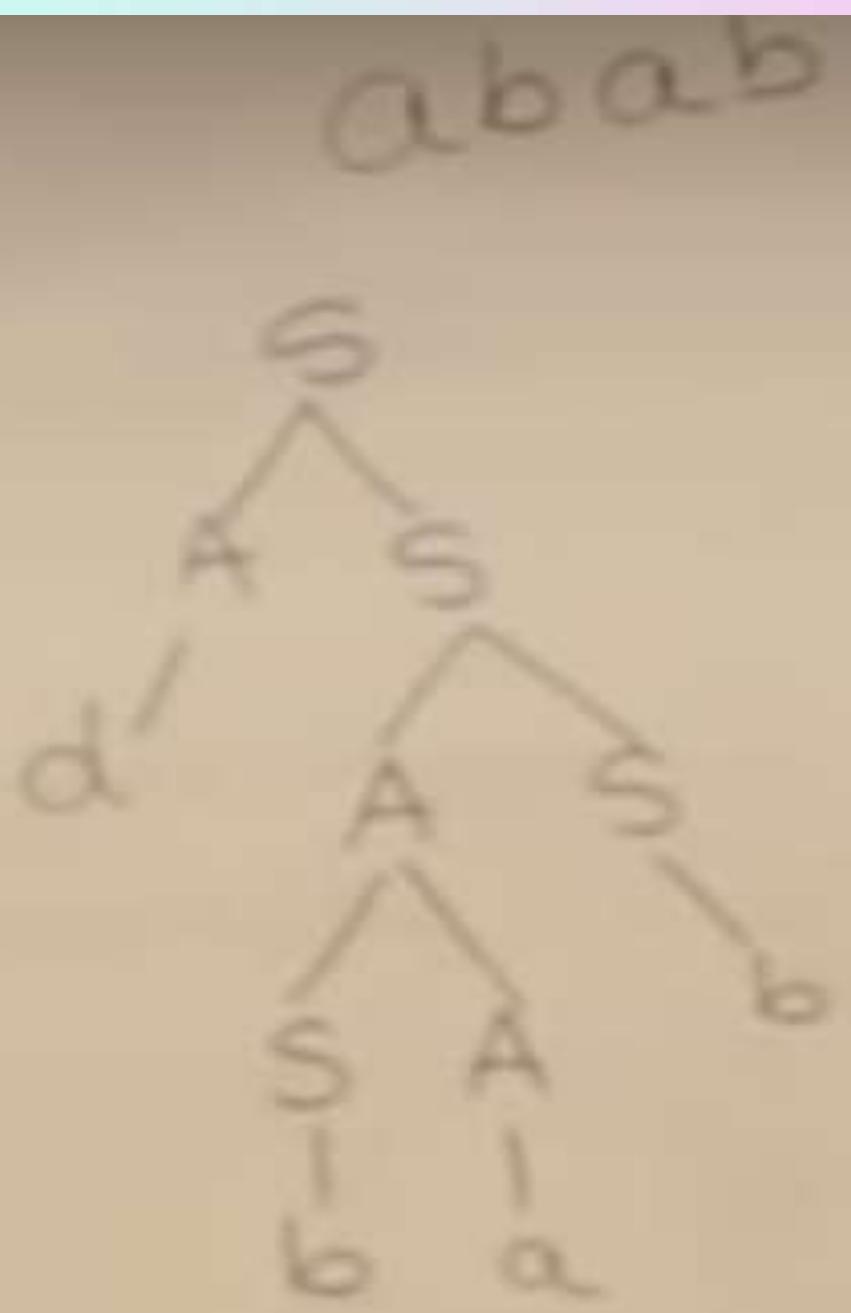
$S \rightarrow AaAb \textcircled{1}$
 $/ BbBa \textcircled{2}$
 $A \rightarrow \epsilon \textcircled{3}$
 $B \rightarrow \epsilon \textcircled{4}$





$LL(1) \times$
 ~~$LR(0) \times$~~
 ~~$SLR(1) \times$~~





$LL(1) \times$

$\{d\}$

$LR(0) \times$

$SLR(1)$

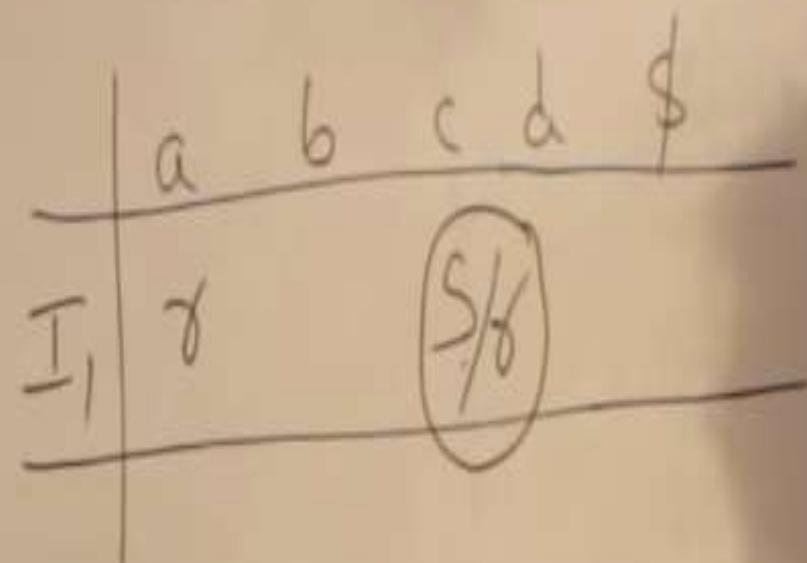
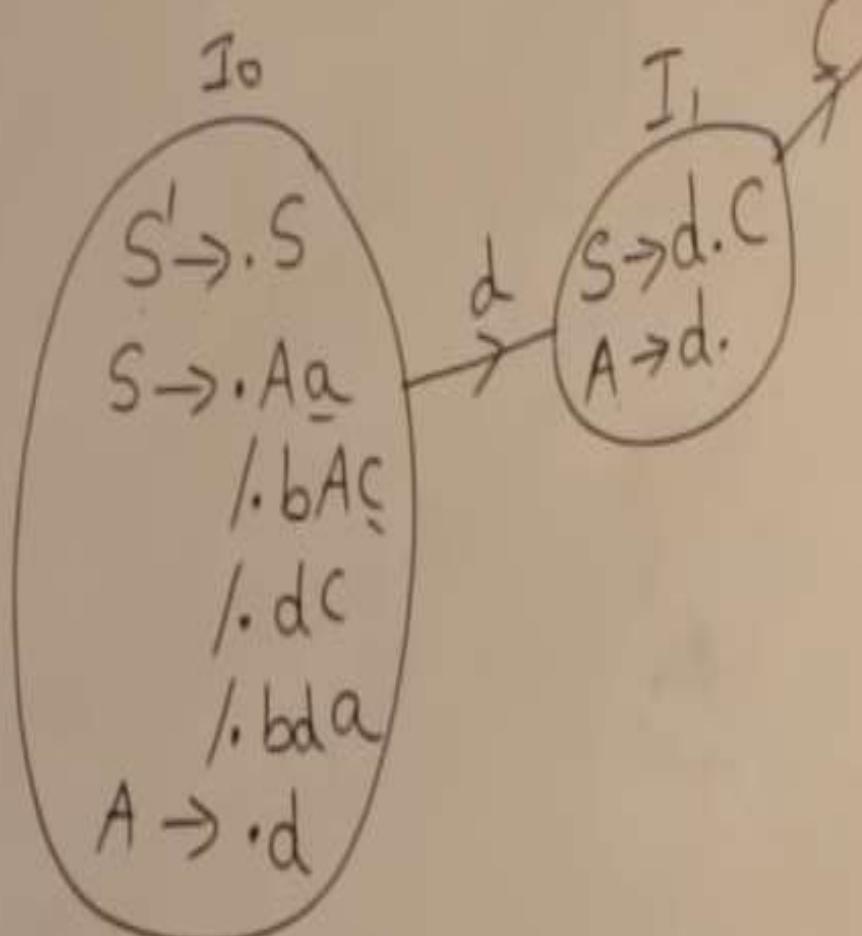
$\rightarrow Aa$

$/bAC \{b\}$

$/dc \{d\}$

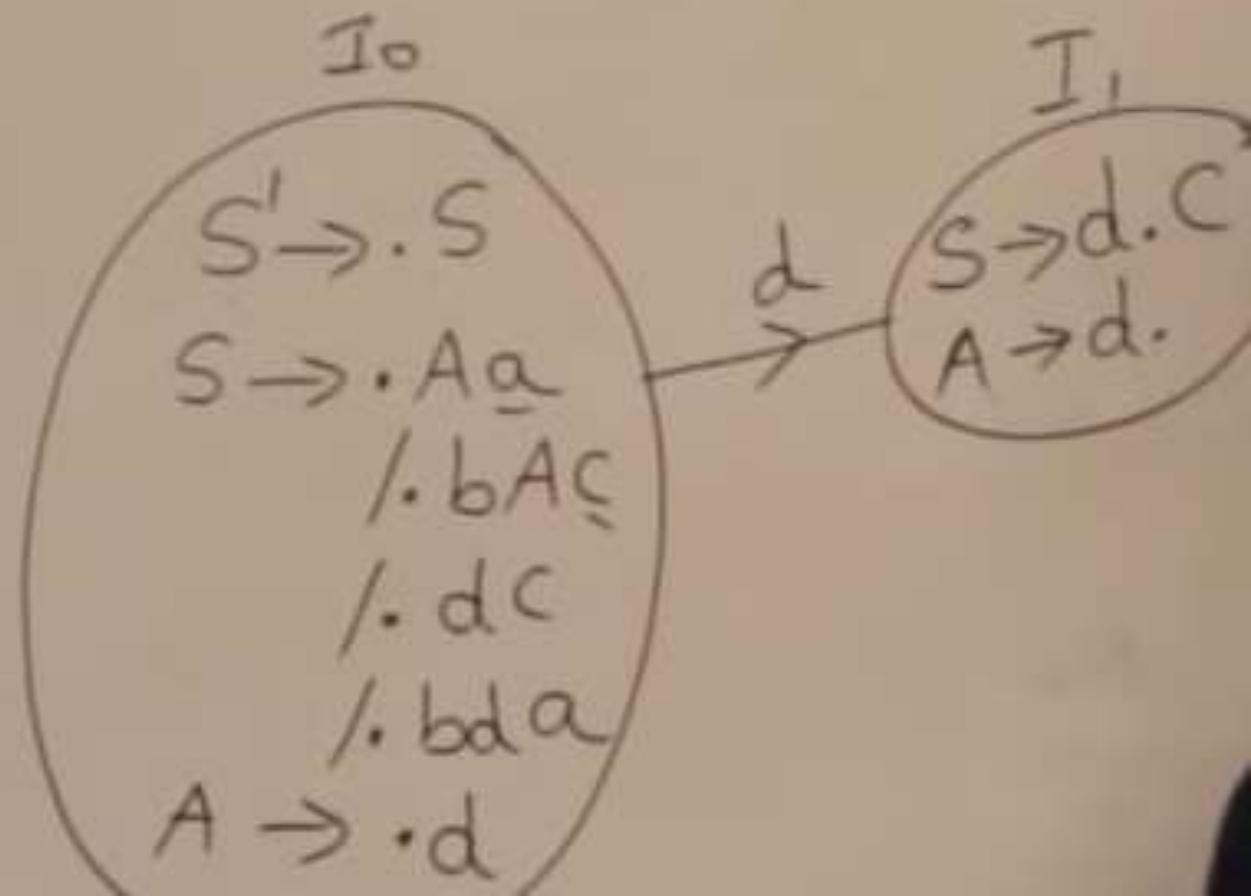
$/bda$

$\rightarrow d$



$S \rightarrow Aa \quad \{d\}$
 $/bAC \quad \{b\}$
 $/dc \quad \{d\}$
 $/bda$
 $A \rightarrow d$

$LL(1) X$
 $LR(0) X$
 $SLR(1) X$



Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a :
 - if the $A \rightarrow \alpha.$ in the I_i and a is $\text{FOLLOW}(A)$
- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$

$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$

$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$Aab \Rightarrow \varepsilon ab$

$Bba \Rightarrow \varepsilon ba$

$B \rightarrow \varepsilon$

$AaAb \Rightarrow Aa \varepsilon b$

$BbBa \Rightarrow Bb \varepsilon a$

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

$$A \rightarrow \alpha \bullet \beta, a$$

where **a** is the look-head of the LR(1) item
(**a** is a terminal or end-marker.)

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha \cdot \beta, a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha \cdot, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in $\text{FOLLOW}(A)$).
- A state will contain $A \rightarrow \alpha \cdot, a_1 \dots a_n$ where $\{a_1, \dots, a_n\} \subseteq \text{FOLLOW}(A)$

...

$A \rightarrow \alpha \cdot, a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha.B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then $\text{goto}(I,X)$ is defined as follows:
 - If $A \rightarrow \alpha.X\beta, a$ in I
then every item in $\text{closure}(\{A \rightarrow \alpha X.\beta, a\})$ will be in $\text{goto}(I,X)$.

Construction of The Canonical LR(1) Collection

- *Algorithm:*

C is $\{ \text{closure}(\{S' \rightarrow .S, \$\}) \}$

repeat the followings until no more set of LR(1) items can be added to C .

for each I in C and each grammar symbol X

if $\text{goto}(I, X)$ is not empty and not in C

 add $\text{goto}(I, X)$ to C

- goto function is a DFA on the sets in C .

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \cdot \beta, a_1$$

...

$$A \rightarrow \alpha \cdot \beta, a_n$$

can be written as

$$A \rightarrow \alpha \cdot \beta, a_1/a_2/\dots/a_n$$

LALRC(1) CLR(1)

LR(1) item = LR(0) item + lookahead

$S \rightarrow .aA, a/b$

$(S \rightarrow aA.), c/d$

LALRC(1) CLR(1)

LR(1) item = LR(0) item + lookahead

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

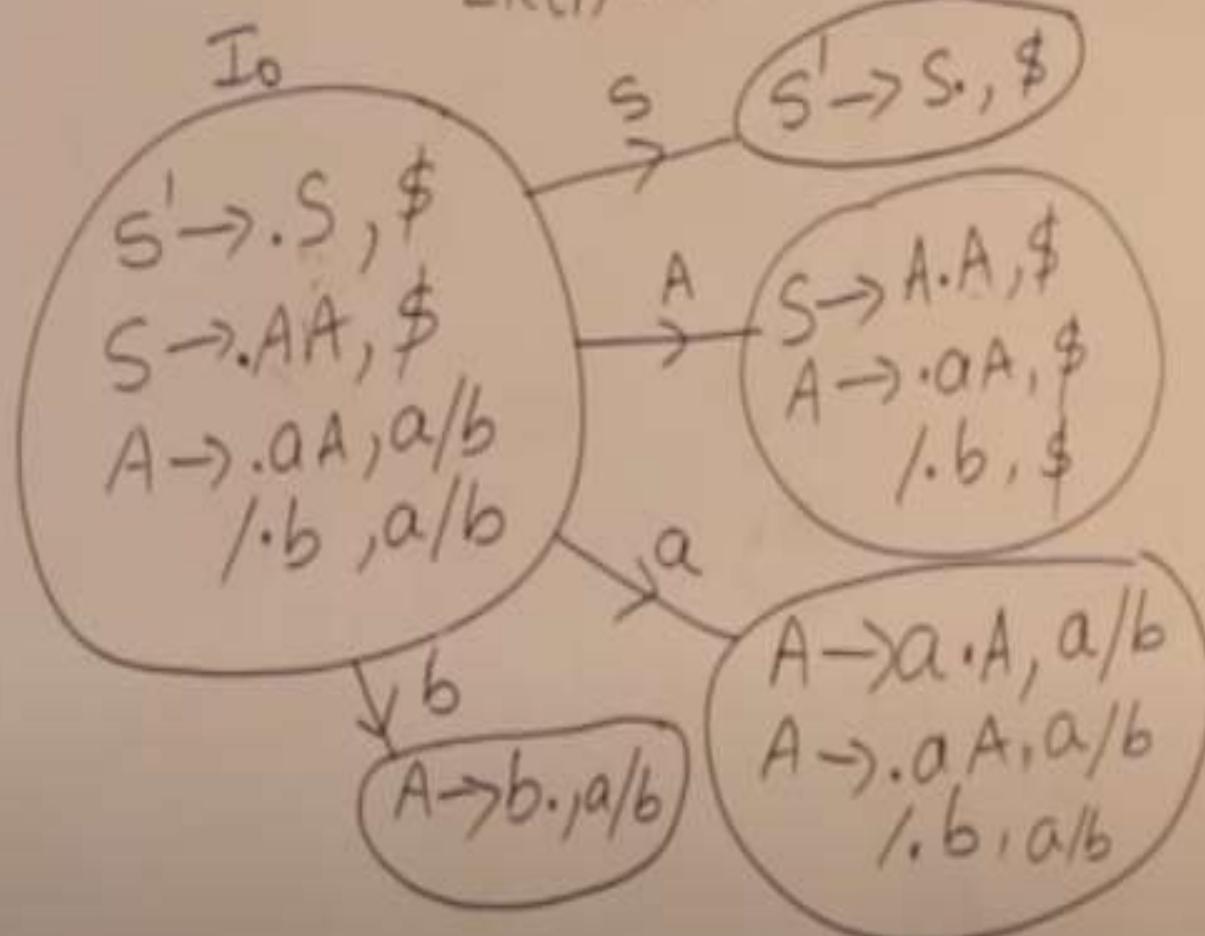
$S' \rightarrow .S, \$$
 $S \rightarrow .AA,$

$A \rightarrow d.B^*, a/b$
 $B \rightarrow .r, a/b$

LALR(1) CLR(1)

LR(1) item = LR(0) item + lookahead

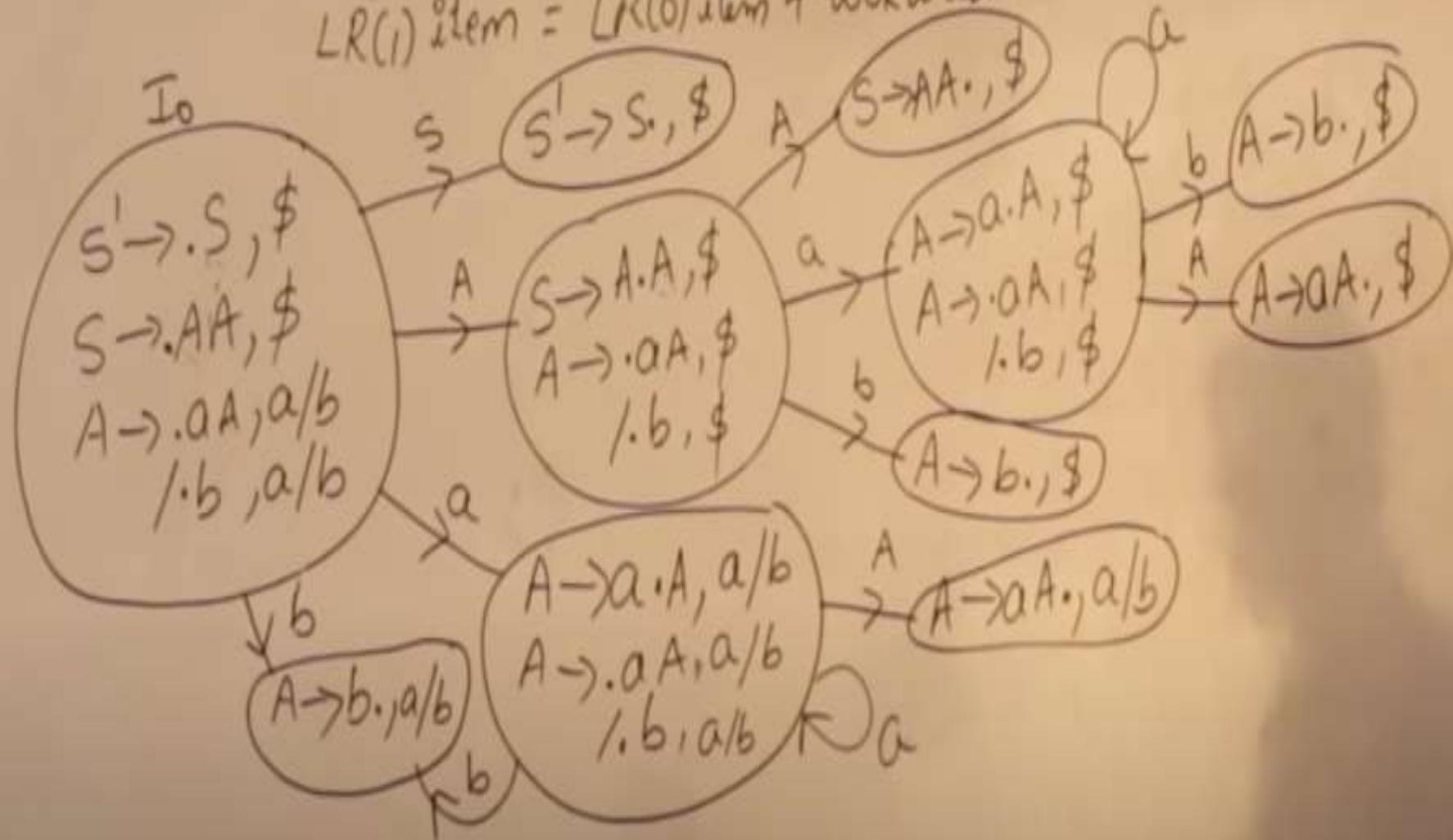
$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow aA/b \end{aligned}$$



LALR(1) CLR(1)

LR(1) item = LR(0) item + lookahead

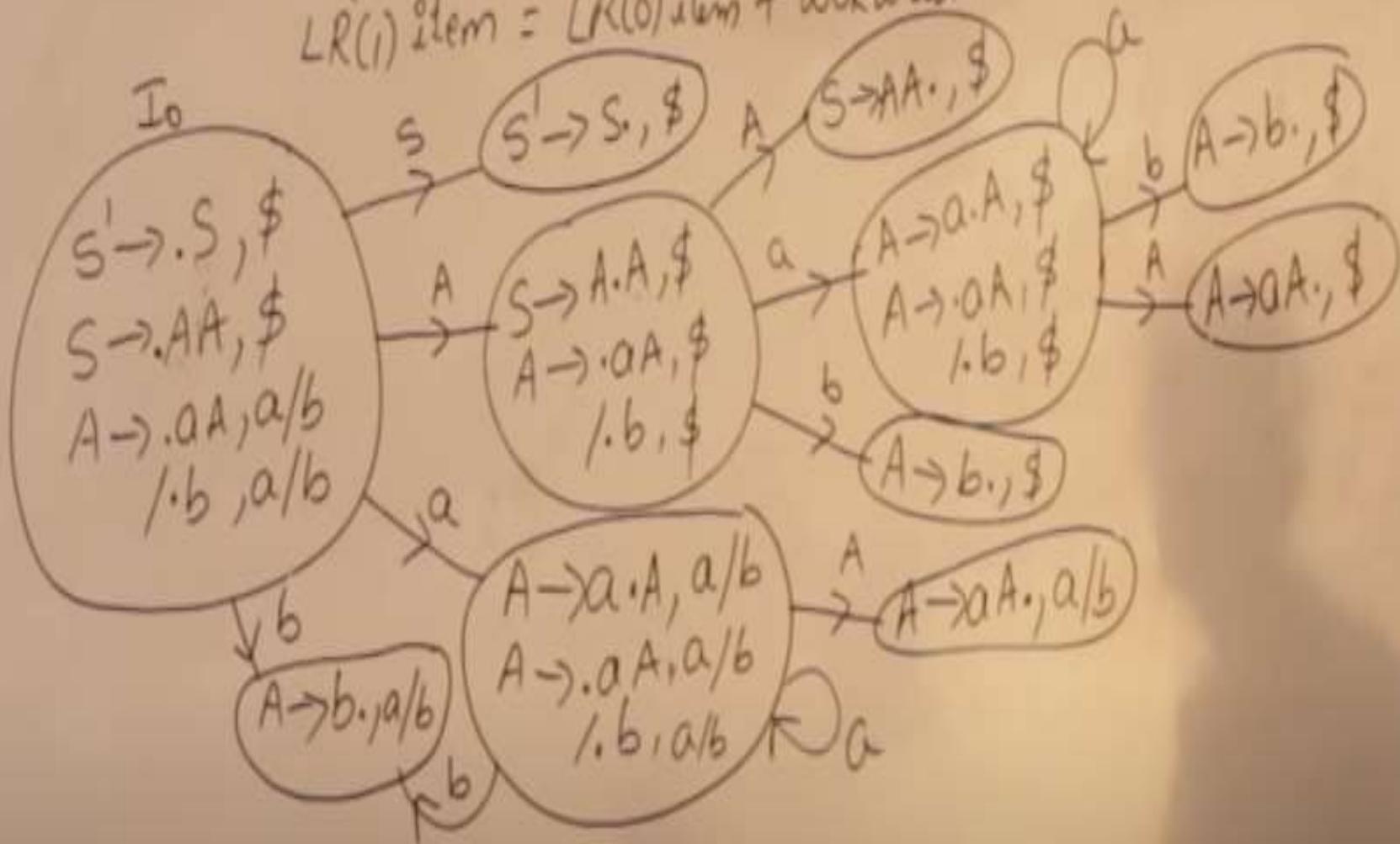
$\rightarrow AA$
 $\rightarrow aA/b$



LALR(1) CLR(1)

LR(1) item = LR(0) item + lookahead

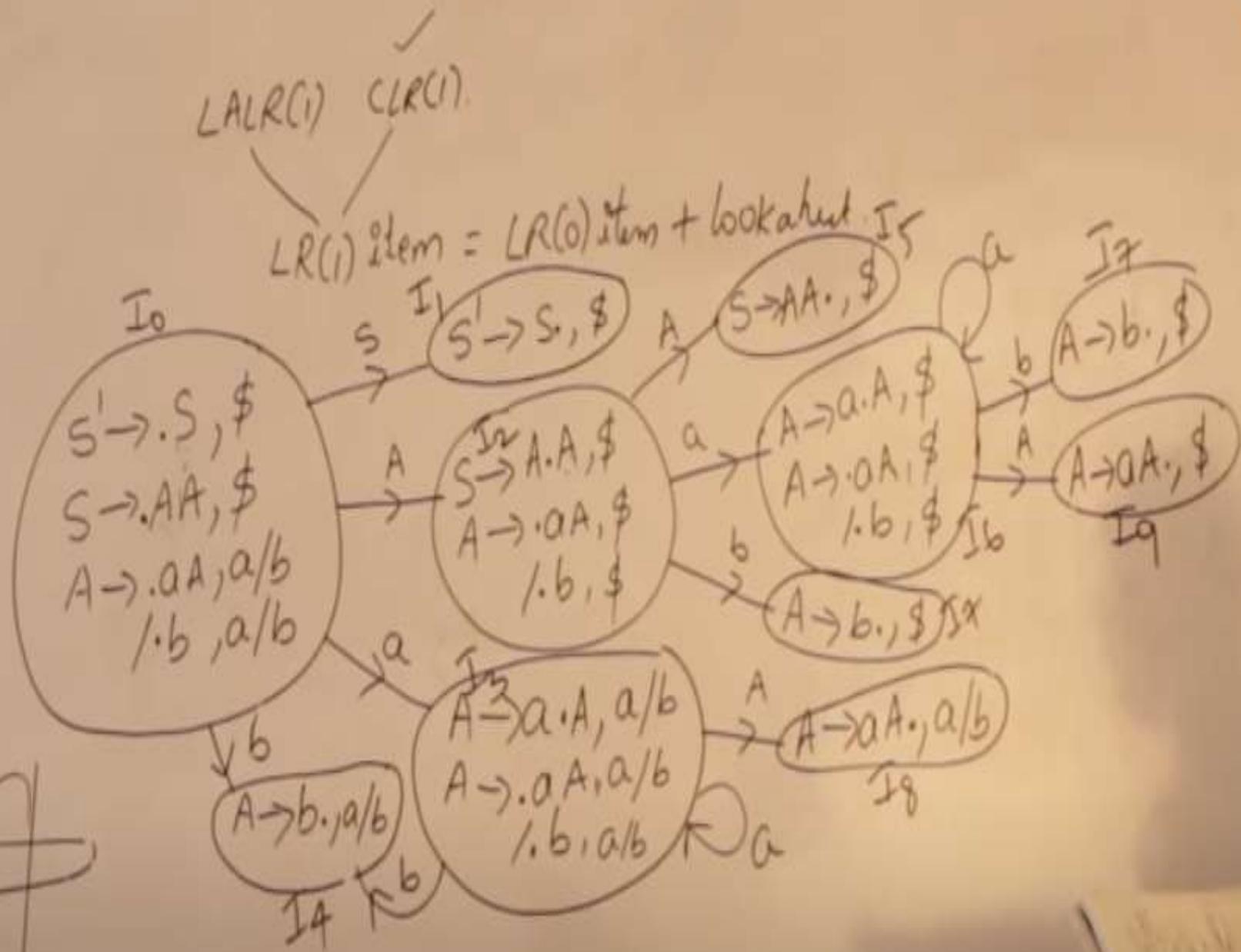
$$S \rightarrow AA$$
$$A \rightarrow aA/b$$



$$S \rightarrow AA$$

$$A \rightarrow aA/b$$

~~mark~~



CLR(1) parsing table

	a	b	\$	s	A
0	s_3	s_4			2
1					
2	s_6	s_7			5
3	s_3	s_4			8
4	τ_3	τ_3			
5			τ_1		
6	s_6	s_7			9
7			τ_3		
8	τ_2	τ_2			
9			τ_2		

I_3, I_6

I_4, I_7

I_8, I_9

CLR(1) parsing table

	a	b	\$	S	A
0	s_3	s_4			2
1					
2	s_6	s_7			5
3	s_3	s_4			8
4	τ_3	τ_3			
5			τ_1		
6	s_6	s_7			9
7			τ_3		
8	τ_2	τ_2			
9			τ_2		

I_3, I_6

I_4, I_7

I_8, I_9

$S \rightarrow AA. \textcircled{1}$

$A \rightarrow aA \textcircled{2}$

$A \rightarrow b. \textcircled{3}, a/b$

CLR(1) parsing table

	a	b	\$	S	A	7
0	s_3	s_4				
1						
2	s_6	s_7				5
3	s_3	s_4				8
4	τ_3	τ_3				
5			τ_1			
6	s_6	s_7				9
7			τ_3			
8	τ_2	τ_2				
9			τ_2			

CLR(1) parsing table

	a	b	\$	S	A	7
0	s_3	s_4				
1						
2	s_6	s_7				5
3	s_3	s_4				8
4	τ_3	τ_3				
5			τ_1			
6	s_6	s_7				9
7			τ_3			
8	τ_2	τ_2				
9			τ_2			

$$[I_3, I_6] \Rightarrow I_{36}$$

$$[I_4, I_7] \Rightarrow I_{47}$$

$$[I_8, I_9] \Rightarrow I_{89}$$

$$S \rightarrow AA. \textcircled{1}, \$$$

$$A \rightarrow aA \textcircled{2}$$

$$A \rightarrow b. \textcircled{3}, a/b$$

$$[I_3, I_6] \Rightarrow I_{36}$$

$$[I_4, I_7] \Rightarrow I_{47}$$

$$[I_8, I_9] \Rightarrow I_{89}$$

$$S \rightarrow AA. \textcircled{1}, \$$$

$$A \rightarrow aA \textcircled{2}$$

$$A \rightarrow b. \textcircled{3}, a/b$$

CLR(1) parsing table

7

$CLR(1) \supset LR(0)$: $SLR(1)$

	a	b	\$	S	A
0	s_{36}	s_{47}		s	$\frac{A}{2}$
1					
2	s_3	s_7		s	s
36	s_{36}	s_{47}			89
47	τ_3	τ_3			
5				τ_1	
36	s_3	s_{47}			89
47			τ_3		
89	τ_2	τ_2			
89			τ_2		

$$[I_3, I_6] \Rightarrow I_{36}$$

$$[I_4, I_7] \Rightarrow I_{47}$$

$$[I_8, I_9] \Rightarrow I_{89}$$

$$S \rightarrow AA. \textcircled{1}, \$$$

$$A \rightarrow 0A \textcircled{2}$$

$$A \rightarrow b. \textcircled{3}, a/b$$

LALR(1) parsing table

7

	a	b	\$	S	A
0	s_{36}	s_{47}		s	λ
1					
2	s_{36}	s_7			5
36	s_{36}	s_{47}			89
47	τ_3	τ_3	τ_3		
5				τ_1	

$CLR(1) \supset L_R(0)$ $\vdash L_R(0) = SLR(1)$
 $LALR(1) \supset CLR(1)$

89	τ_2	τ_2	τ_2	
----	----------	----------	----------	--

$$[I_3, I_6] \Rightarrow I_{36}$$

$$[I_4, I_7] \Rightarrow I_{47}$$

$$[I_8, I_9] \Rightarrow I_{89}$$

$$S \rightarrow AA. \textcircled{1}, \$$$

$$A \rightarrow aA \textcircled{2}$$

$$A \rightarrow b. \textcircled{3}, a/b$$

CLR Parsing Table (LR1 Canonical Items)



$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

LALR
CLR \rightarrow LR(1)

$A \rightarrow C$

$B \rightarrow C$

$S' \rightarrow .S, \$$

$S \rightarrow .aAd, \$$

$S \rightarrow .bBd, \$$

$S \rightarrow .aBe, \$$

$S \rightarrow .bAe, \$$

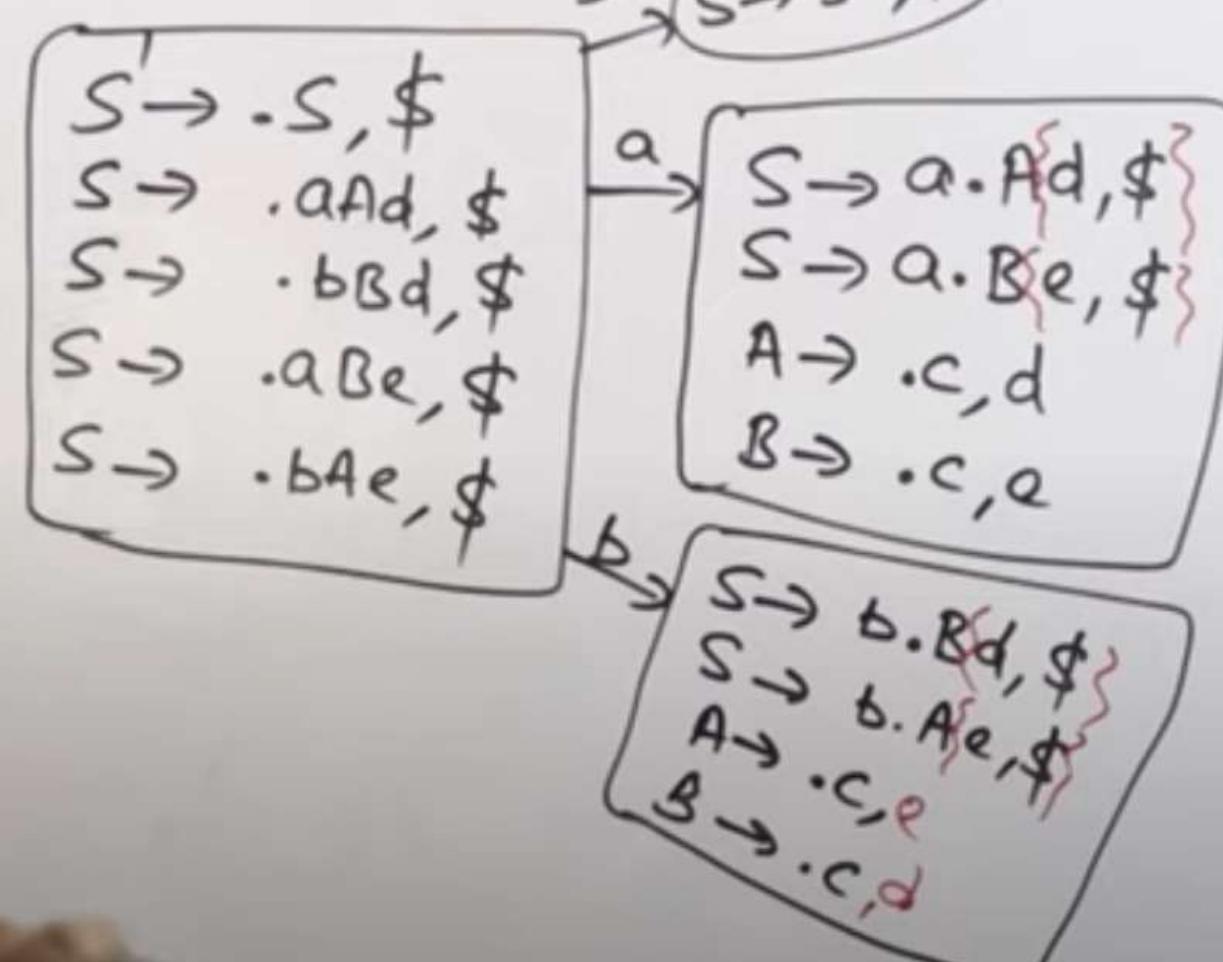
LLR₁ Parsing Table (LR(1) Canonical)

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

LALR
CLR \rightarrow LR(1)

$A \rightarrow C$

$B \rightarrow C$



CLR Parsing Table (LR1 Canonical)

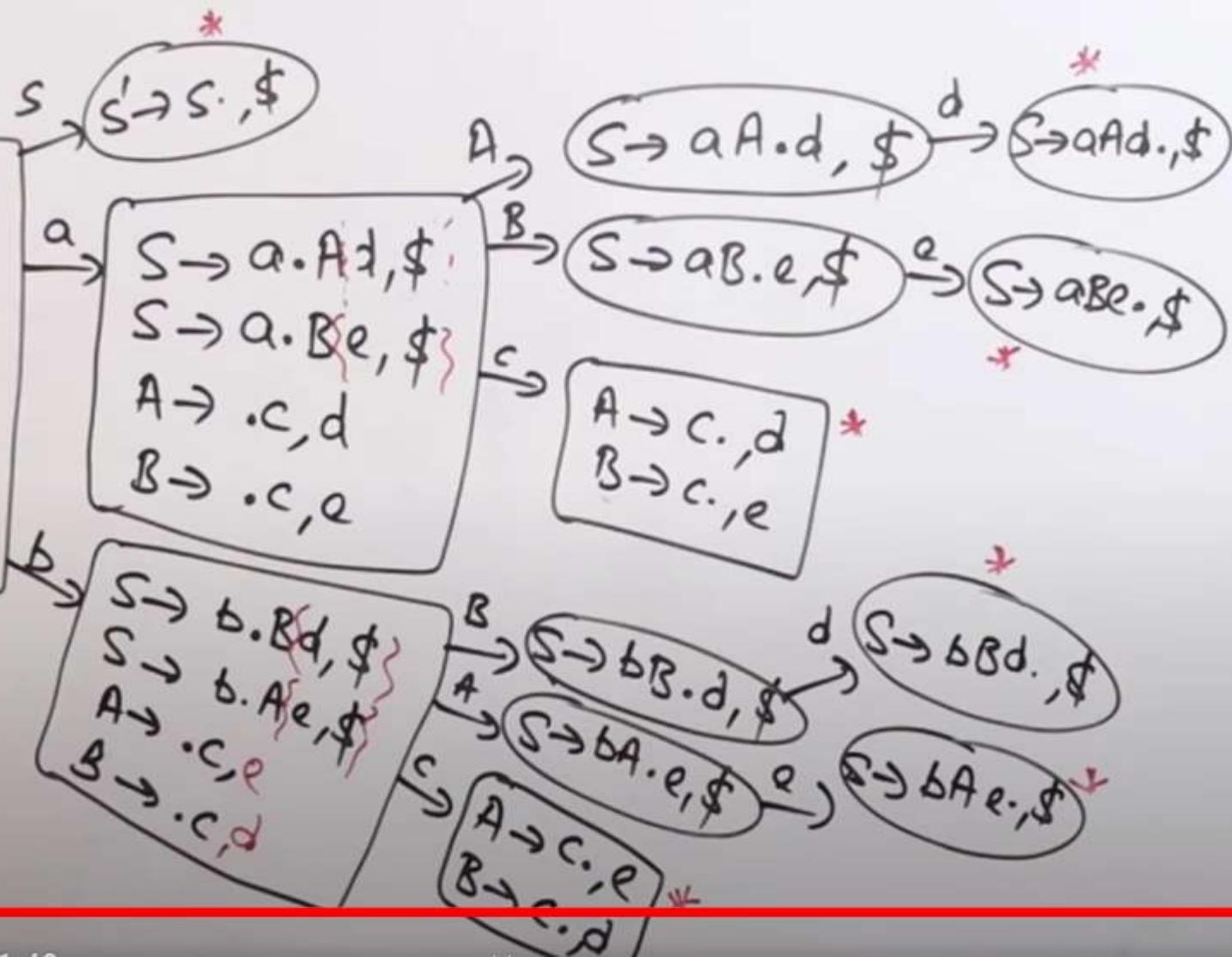
$S \rightarrow a^1 A^2 d^3 | b^2 B^3 d^1 | a^3 B^1 e^4 | b^1 A^4 e^2$

LALR
CLR \rightarrow LR(1)

$A \rightarrow C^5$

$B \rightarrow C^6$

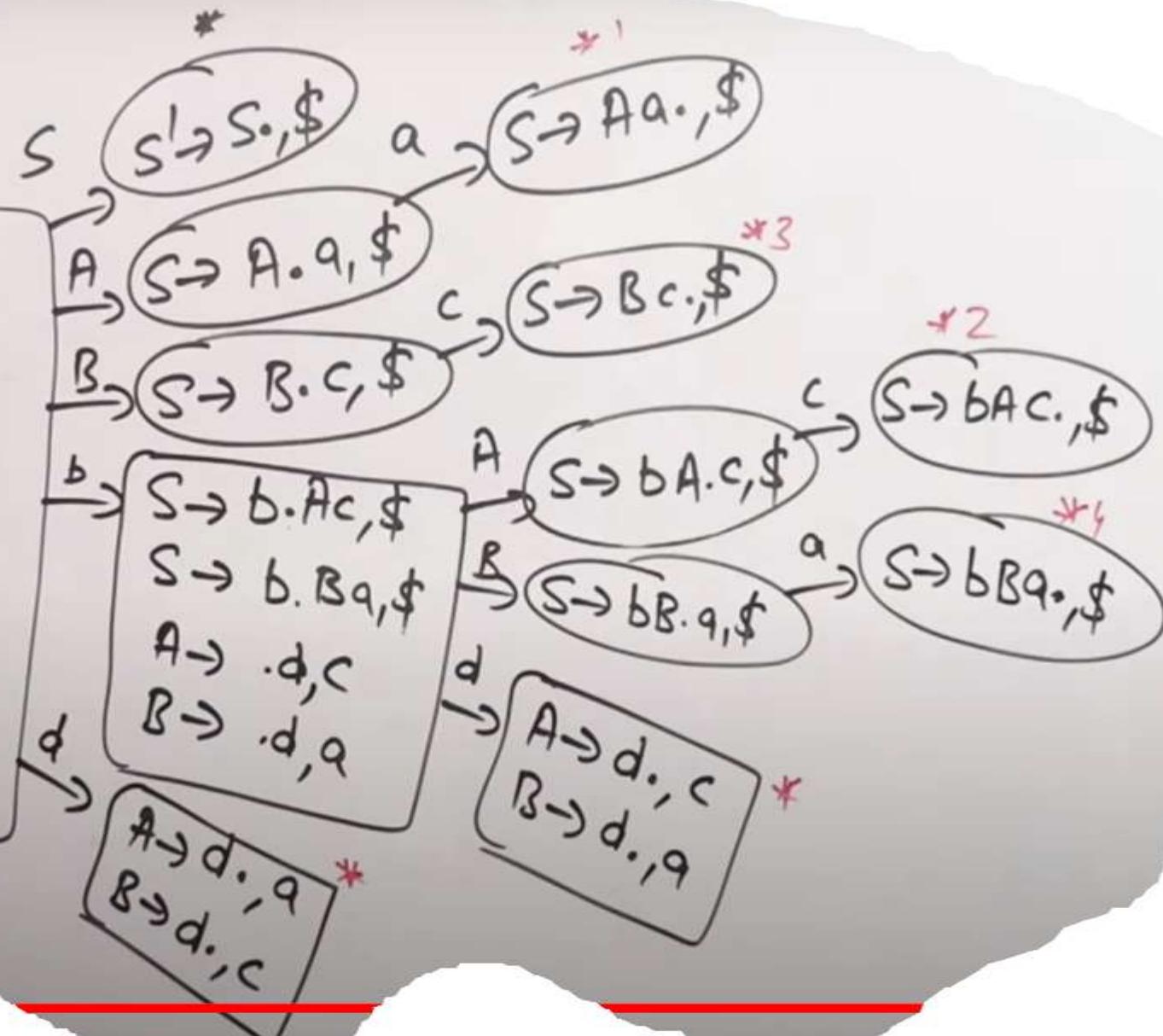
$S' \rightarrow S^1, \$$
 $S' \rightarrow .S^1, \$$
 $S' \rightarrow .aA^2d, \$$
 $S' \rightarrow .bB^3d, \$$
 $S' \rightarrow .aB^1e, \$$
 $S' \rightarrow .bA^4e, \$$



14. LR(0) Parsing

 $S \rightarrow Aa \mid bAc \mid Bc \mid bBa$
 $A \rightarrow d$ 5

 $B \rightarrow d$ 6

 $S' \rightarrow .S, \$$
 $S \rightarrow .Aa, \$$
 $S \rightarrow .bAc, \$$
 $S \rightarrow .Bc, \$$
 $S \rightarrow .bBa, \$$
 $A \rightarrow .d, a$
 $B \rightarrow .d, c$


1 a | bAc | Bc | bBa

5

6

$S' \rightarrow .S, \$$
 $S \rightarrow .Aa, \$$
 $S \rightarrow .bAc, \$$
 $S \rightarrow .Bc, \$$
 $S \rightarrow .bBa, \$$
 \vdots
 \vdots
 \vdots
 $\rightarrow .d, a$
 $\rightarrow .d, c$

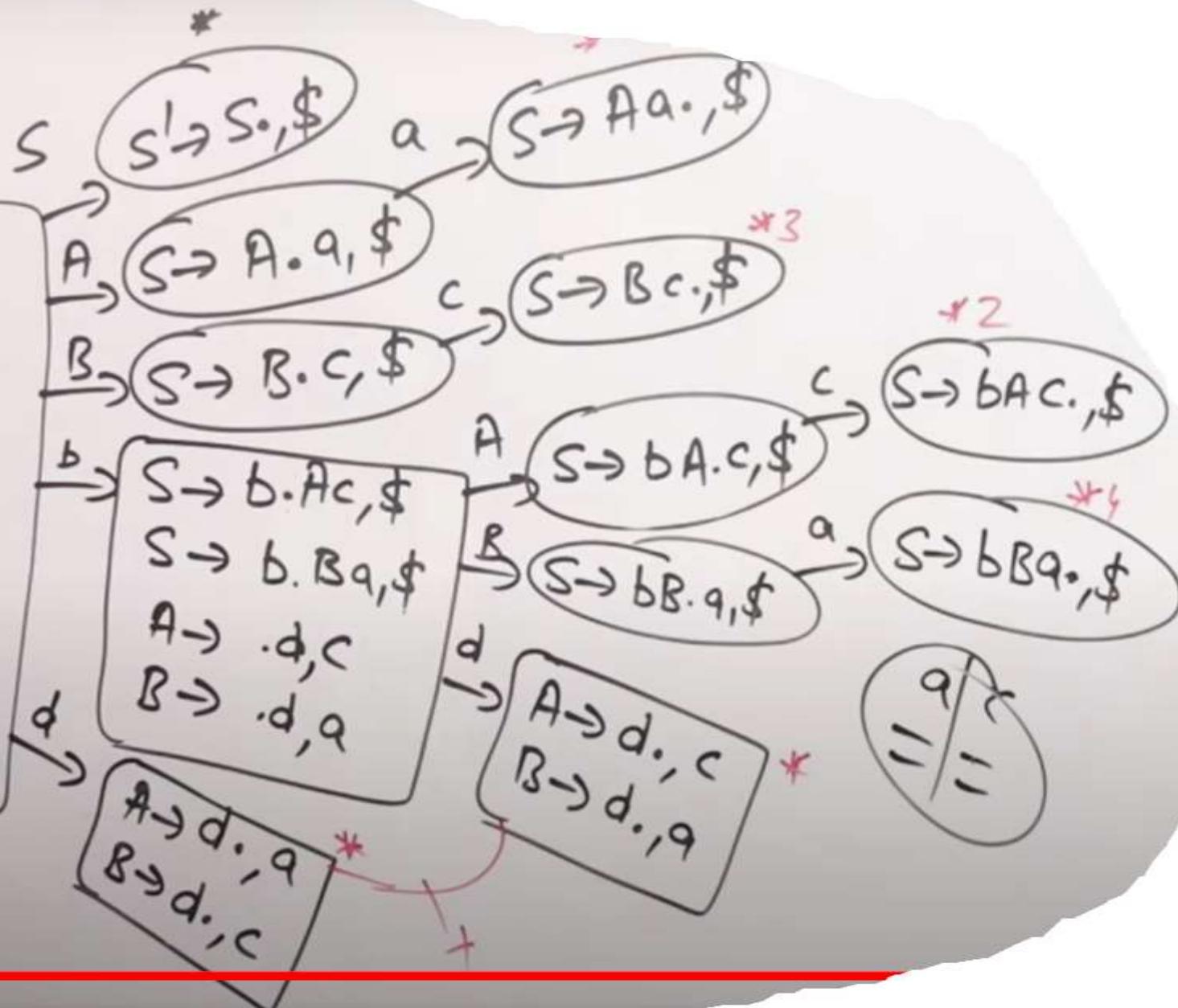
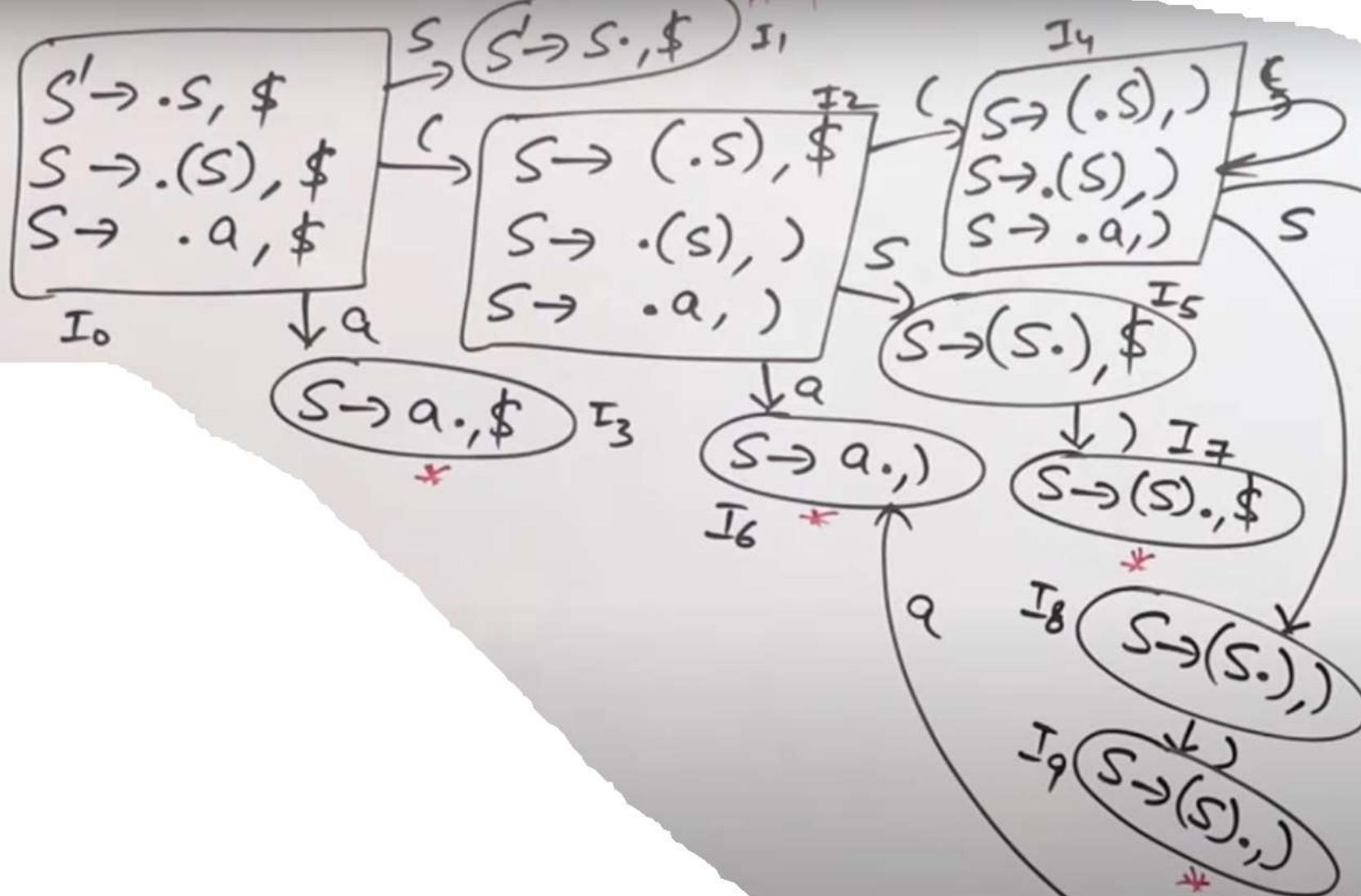
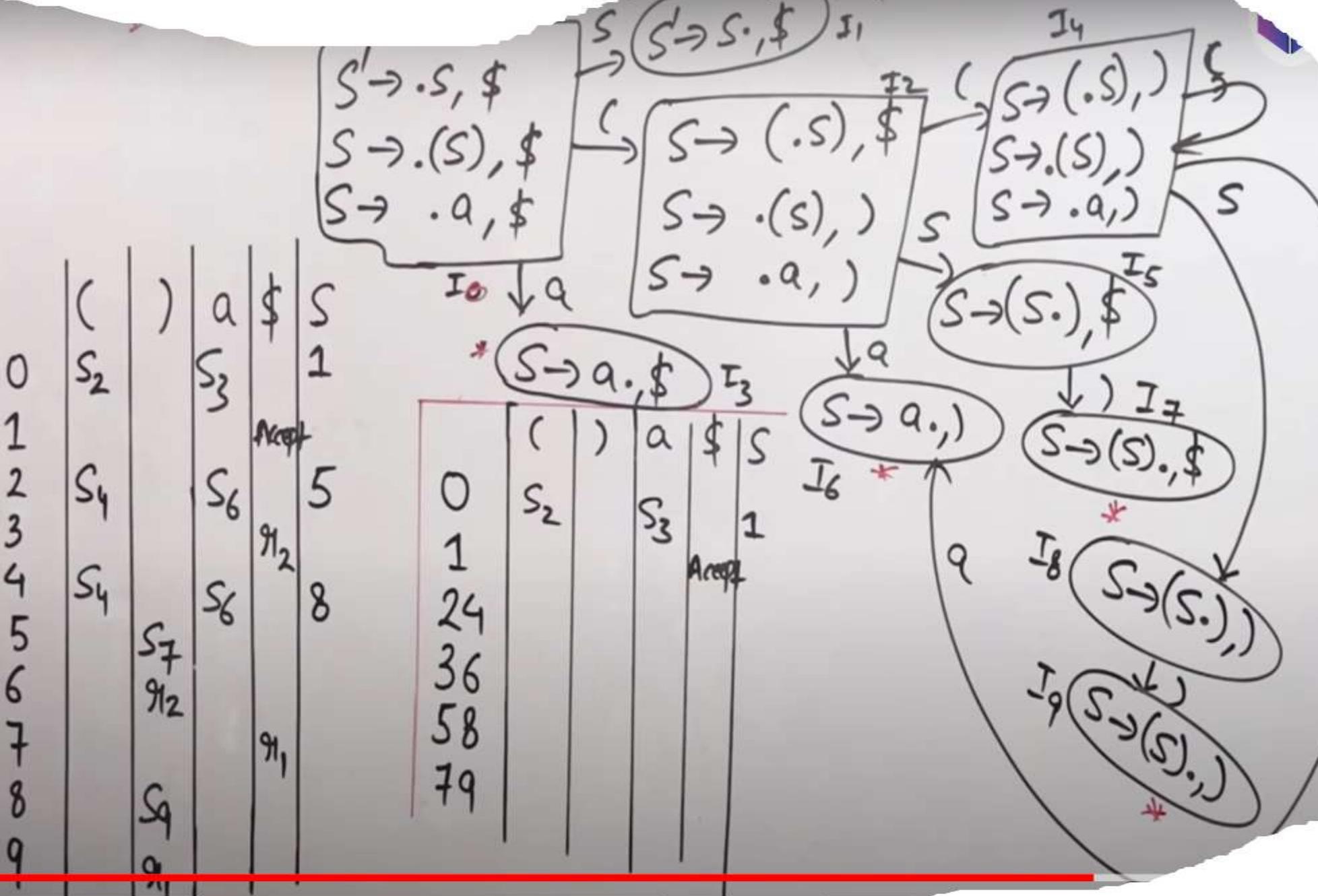
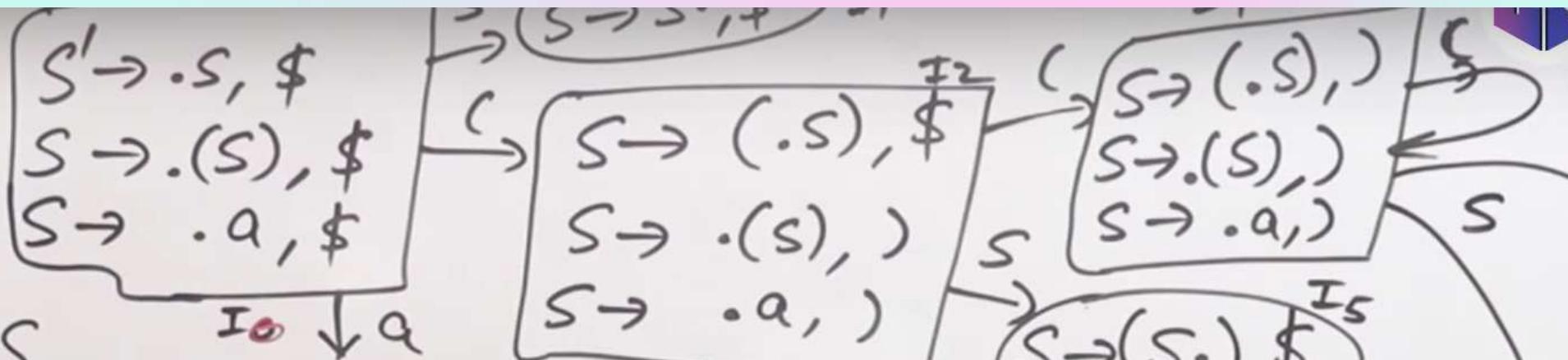


CHART & LR(0) parsing table | Imp. Examples

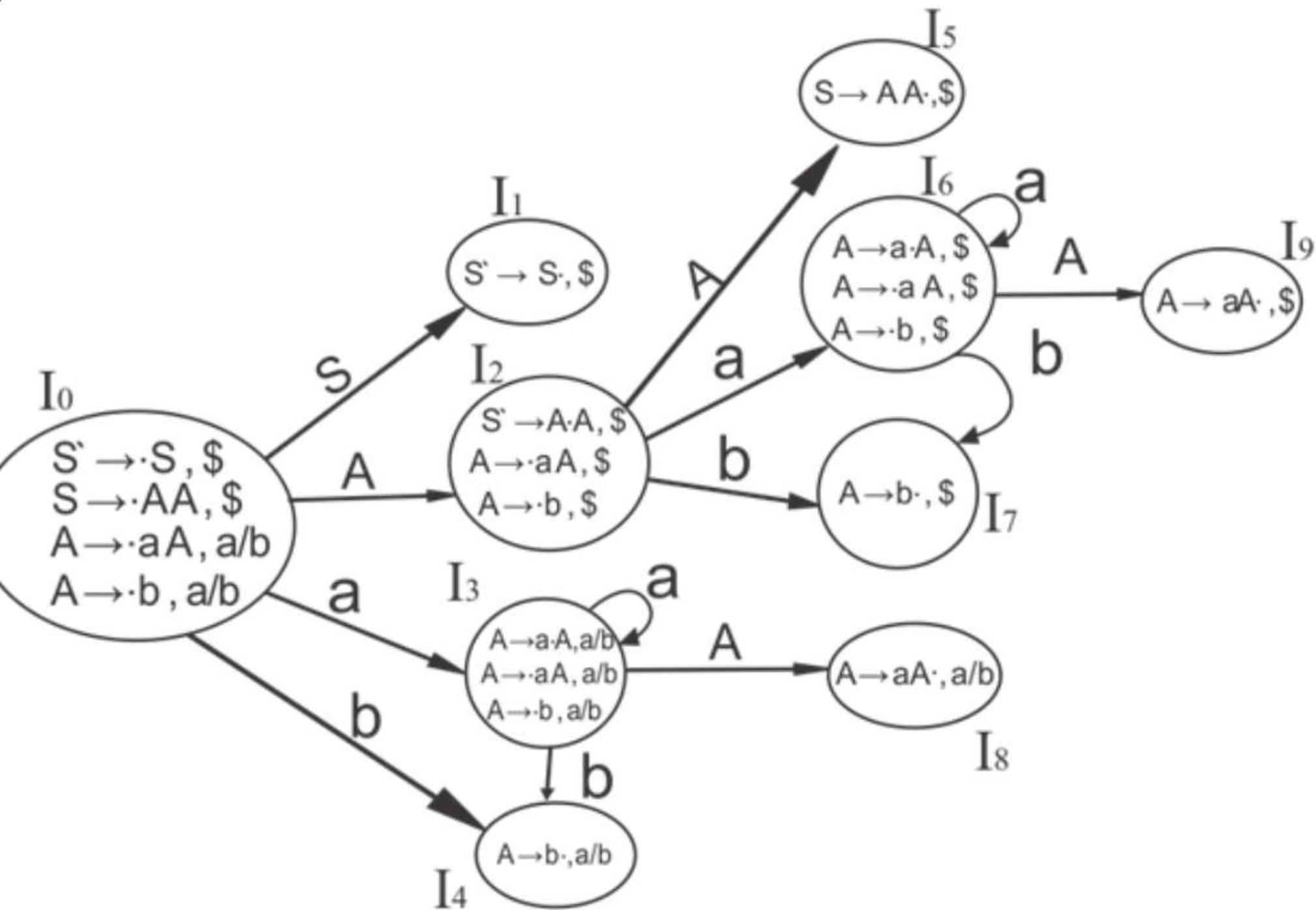






	()	a	\$	S	Accept
0						
1	s_2					
24			s_3			
36				s_4		
58					s_5	
79						

SR X
 CALRX
 RRX
 CAIRL



States	a	b	\$	S	A
I ₀	S ₃	S ₄			2
I ₁			Accept		
I ₂	S ₆	S ₇			5
I ₃	S ₃	S ₄			8
I ₄	R ₃	R ₃			
I ₅			R ₁		
I ₆	S ₆	S ₇			9
I ₇			R ₃		
I ₈	R ₂	R ₂			
I ₉			R ₂		

Canonical LR(1) Collection -- Example

$$S \rightarrow AaAb$$

$$S \rightarrow BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

$$I_0: S' \rightarrow .S , \$$$

$$S \rightarrow .AaAb , \$$$

$$S \rightarrow .BbBa , \$$$

$$A \rightarrow . , a$$

$$B \rightarrow . , b$$

$$I_1: S' \rightarrow S. , \$$$

$$I_2: S \rightarrow A.aAb , \$ \xrightarrow{a} \text{to } I_4$$

$$I_3: S \rightarrow B.bBa , \$ \xrightarrow{b} \text{to } I_5$$

$$I_4: S \rightarrow Aa.Ab , \$ \xrightarrow{A} I_6: S \rightarrow AaA.b , \$ \xrightarrow{a} I_8: S \rightarrow AaAb. , \$$$

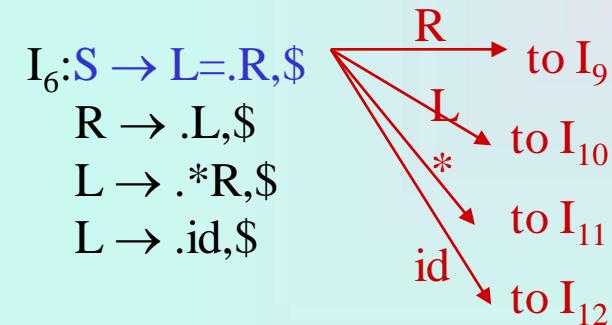
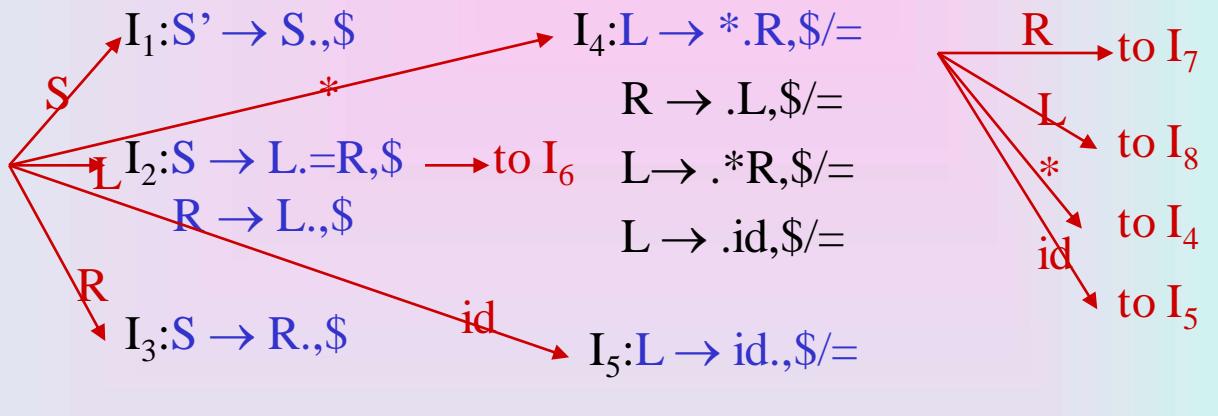
$$A \rightarrow . , b$$

$$I_5: S \rightarrow Bb.Ba , \$ \xrightarrow{B} I_7: S \rightarrow BbB.a , \$ \xrightarrow{b} I_9: S \rightarrow BbBa. , \$$$

$$B \rightarrow . , a$$

Canonical LR(1) Collection – Example2

$S' \rightarrow S$	$I_0: S' \rightarrow .S, \$$
1) $S \rightarrow L=R$	$S \rightarrow .L=R, \$$
2) $S \rightarrow R$	$S \rightarrow .R, \$$
3) $L \rightarrow *R$	$L \rightarrow .*R, \$/=$
4) $L \rightarrow id$	$L \rightarrow .id, \$/=$
5) $R \rightarrow L$	$R \rightarrow .L, \$$



$I_7: L \rightarrow *R., \$/=$

$I_8: R \rightarrow L., \$/=$

$I_9: S \rightarrow L=R., \$$

$I_{10}: R \rightarrow L., \$$

$I_{11}: L \rightarrow *.R, \$$

- $R \rightarrow .L, \$$
- $L \rightarrow .*R, \$$
- $L \rightarrow .id, \$$

$I_{12}: L \rightarrow id., \$$

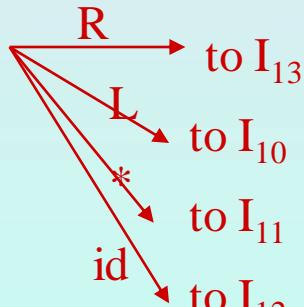
$I_{13}: L \rightarrow *R., \$$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}



Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G' .

$$C \leftarrow \{I_0, \dots, I_n\}$$

2. Create the parsing action table as follows

- If a is a terminal, $A \rightarrow \alpha \cdot a\beta, b$ in I_i and $\text{goto}(I_i, a) = I_j$ then $\text{action}[i, a]$ is *shift j*.
- If $A \rightarrow \alpha \cdot, a$ is in I_i , then $\text{action}[i, a]$ is *reduce A $\rightarrow \alpha$* where $A \neq S'$.
- If $S' \rightarrow S \cdot, \$$ is in I_i , then $\text{action}[i, \$]$ is *accept*.
- If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table

- for all non-terminals A , if $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow \cdot S, \$$

LR(1) Parsing Tables – (for Example2)

	id	*	=	\$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

no shift/reduce or
no reduce/reduce conflict



so, it is a LR(1) grammar

LALR Parsing Tables

- LALR stands for LookAhead LR.
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- *yacc* creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser



LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet = R, \$$ \rightarrow $S \rightarrow L \bullet = R$ Core
 $R \rightarrow L \bullet, \$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

$I_1: L \rightarrow id \bullet, =$ A new state: $I_{12}: L \rightarrow id \bullet, =$
 \rightarrow $L \rightarrow id \bullet, \$$

$I_2: L \rightarrow id \bullet, \$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.
$$C = \{I_0, \dots, I_n\} \rightarrow C' = \{J_1, \dots, J_m\} \quad \text{where } m \leq n$$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
 - Note that: If $J = I_1 \cup \dots \cup I_k$ since I_1, \dots, I_k have same cores
 \rightarrow cores of $\text{goto}(I_1, X), \dots, \text{goto}(I_k, X)$ must be same.
 - So, $\text{goto}(J, X) = K$ where K is the union of all sets of items having same cores as $\text{goto}(I_1, X)$.
- If no conflict is introduced, the grammar is LALR(1) grammar.
(We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \rightarrow \alpha \bullet , a \quad \text{and} \quad B \rightarrow \beta \bullet a\gamma, b$$

- This means that a state of the canonical LR(1) parser must have:

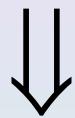
$$A \rightarrow \alpha \bullet , a \quad \text{and} \quad B \rightarrow \beta \bullet a\gamma, c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

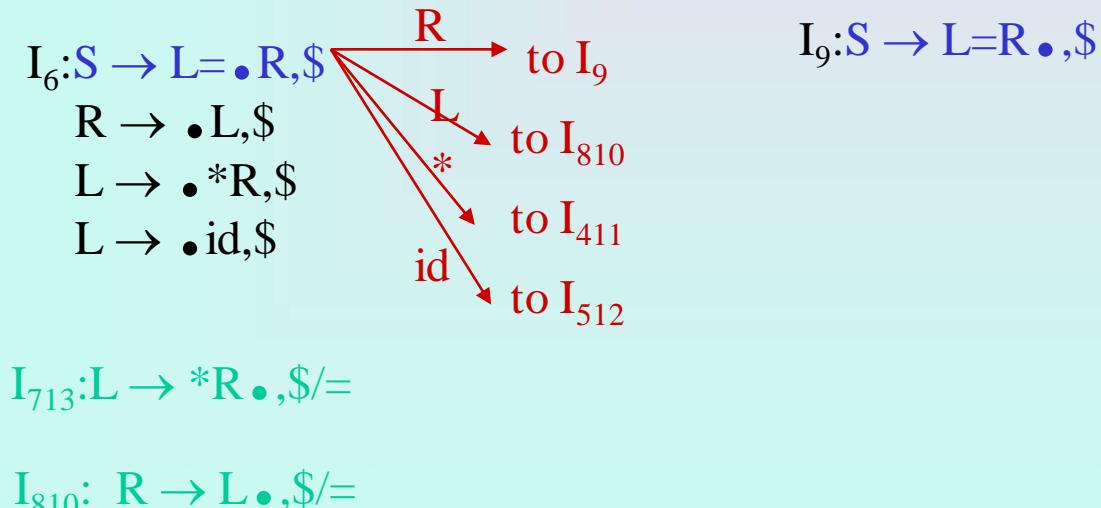
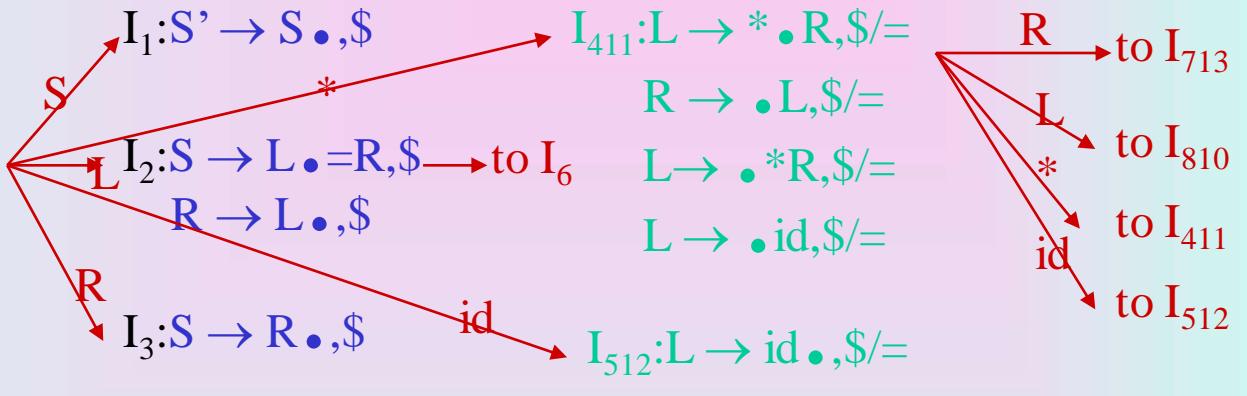
- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$$I_1 : A \rightarrow \alpha \bullet, a$$
$$B \rightarrow \beta \bullet, b$$
$$I_2 : A \rightarrow \alpha \bullet, b$$
$$B \rightarrow \beta \bullet, c$$

$$I_{12} : A \rightarrow \alpha \bullet, a/b$$
$$B \rightarrow \beta \bullet, b/c$$

→ reduce/reduce conflict

Canonical LALR(1) Collection – Example2

$S' \rightarrow S$	$I_0: S' \rightarrow \bullet S, \$$
1) $S \rightarrow L=R$	$S \rightarrow \bullet L=R, \$$
2) $S \rightarrow R$	$S \rightarrow \bullet R, \$$
3) $L \rightarrow *R$	$L \rightarrow \bullet *R, \$/=$
4) $L \rightarrow id$	$L \rightarrow \bullet id, \$/=$
5) $R \rightarrow L$	$R \rightarrow \bullet L, \$$



Same Cores
 I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	\$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict



so, it is a LALR(1) grammar

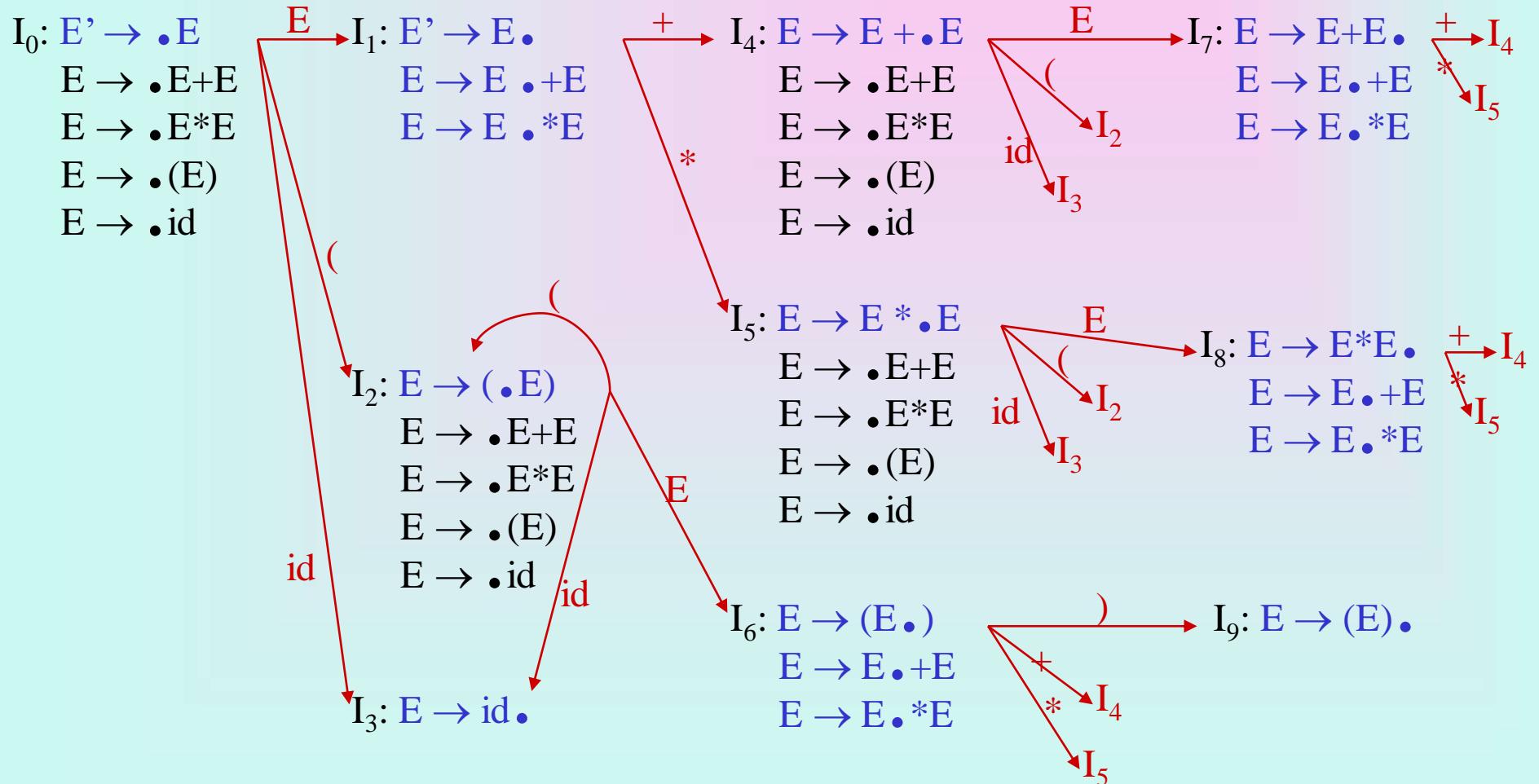
Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
 - Yes, but they will have conflicts.
 - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
 - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
 - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
 - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

$$E \rightarrow E+E \mid E^*E \mid (E) \mid id$$

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T^*F \mid F$$
$$F \rightarrow (E) \mid id$$

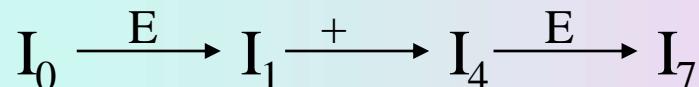
Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

$$\text{FOLLOW}(E) = \{ \$, +, *,) \ }$$

State I_7 has shift/reduce conflicts for symbols $+$ and $*$.



when current token is $+$

shift $\rightarrow +$ is right-associative

reduce $\rightarrow +$ is left-associative

when current token is $*$

shift $\rightarrow *$ has higher precedence than $+$

reduce $\rightarrow +$ has higher precedence than $*$

SLR-Parsing Tables for Ambiguous Grammar

$$\text{FOLLOW}(E) = \{ \$, +, *,) \ }$$

State I_8 has shift/reduce conflicts for symbols + and *.



when current token is *

shift \rightarrow * is right-associative

reduce \rightarrow * is left-associative

when current token is +

shift \rightarrow + has higher precedence than *

reduce \rightarrow * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	Action	Goto						
	id	+	*	()	\$	E	
0	s3			s2				1
1		s4	s5			acc		
2	s3			s2				6
3		r4	r4		r4	r4		
4	s3			s2				7
5	s3			s2				8
6		s4	s5		s9			
7		r1	s5		r1	r1		
8		r2	r2		r2	r2		
9		r3	r3		r3	r3		

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state **s**).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow **A**.
 - The symbol **a** is simply in FOLLOW(**A**), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal **A** is normally is a basic programming block (there can be more than one choice for **A**).
 - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
 - missing operand
 - unbalanced right parenthesis

- <https://www.asheeshkg.com/compare-slr-clr-and-lalr-parsers/>
- <https://www.techtud.com/short-notes/clr1-and-lalr1-parsers-notes>