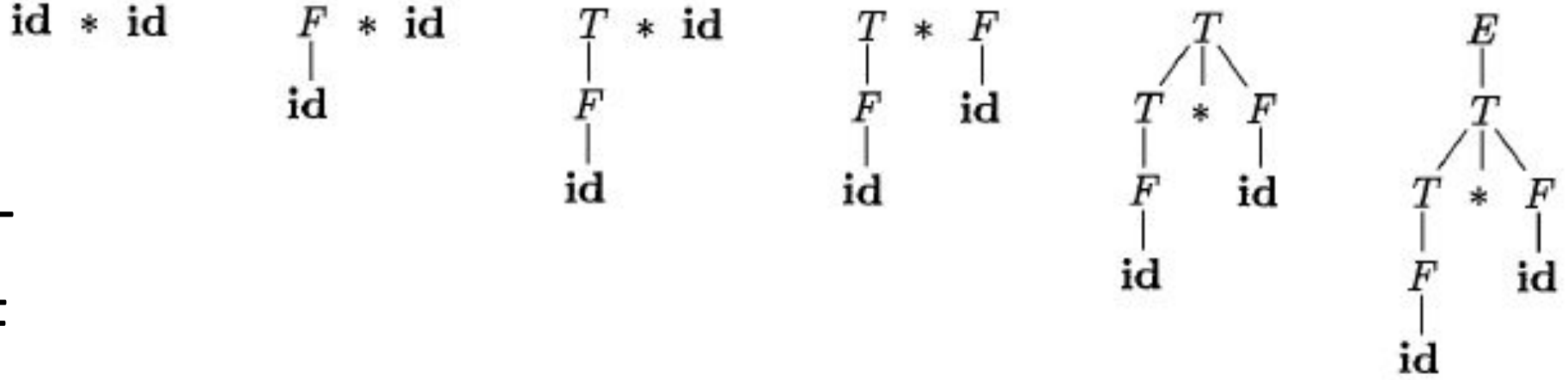
Bottom Up Parsing

Bottom Up Parsing

- Construction of a parse tree for an input string beginning at the leaves (the bottom) and working towards the root (top).



E -> E + T | T

T -> T * F | F

F -> (E) | id

Figure 4.25: A bottom-up parse for **id * id**

Reductions

- Process of ‘reducing’ a string **w** to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string $\mathbf{id} * \mathbf{id}$. The first reduction produces $F * \mathbf{id}$ by reducing the leftmost **id** to F, using the production $F \rightarrow \mathbf{id}$. The second reduction produces $T * \mathbf{id}$ by reducing F to T.

Now, we have a choice between reducing the string T, which is the body of $E \rightarrow T$, and the string consisting of the second **id**, which is the body of $F \rightarrow \mathbf{id}$. Rather than reduce T to E, the second **id** is reduced to T, resulting in the string $T * F$. This string then reduces to T. The parse completes with the reduction of T to the start symbol E.

By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

This derivation is in fact a rightmost derivation.

Handle Pruning

- Bottom – up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse.
- 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.

- For ex, adding subscripts to the tokens id for clarity, the handles during the parse of id1*id2 according to the grammar. Although T is the body of the production E -> T, the symbol I is not a handle in the sentenial form T * id2.
- If T were indeed replaced by E, we would get the string E * id2, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Ex 2:

S -> aABe
A -> Abc | b
B -> d
Handles during parsing of abbcde

Right Sentential form	Handle	Reducing Productions
abbcde	b	A -> b
aAbcde	Abc	A -> Abc
aAde	d	B -> d
aABe	aABe	S -> aABe
S		

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we refer to the body β rather than $A \to \beta$ as a handle. Note we say "a handle" rather than "the handle," because the grammar could be ambiguous, with more than one rightmost derivation of $\alpha\beta w$. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals w to be parsed. If w is a sentence of the grammar at hand, then let $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \to \beta_n$ to obtain the previous right-sentential form γ_{n-1}. Note that we do not yet know how handles are to be found, but we shall see methods of doing so shortly.

We then repeat this process. That is, we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2}. If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

Shift Reduce Parsing

- Shift reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.
- We use $ to mark the bottom of the stack and also the right end of the input.

STACK	INPUT
$	w $

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
\$ S	\$

Upon entering this configuration, the parser halts and announces successful completion of parsing. Figure 4.28 steps through the actions a shift-reduce parser might take in parsing the input string $\mathbf{id}_1 * \mathbf{id}_2$ according to the expression grammar.

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

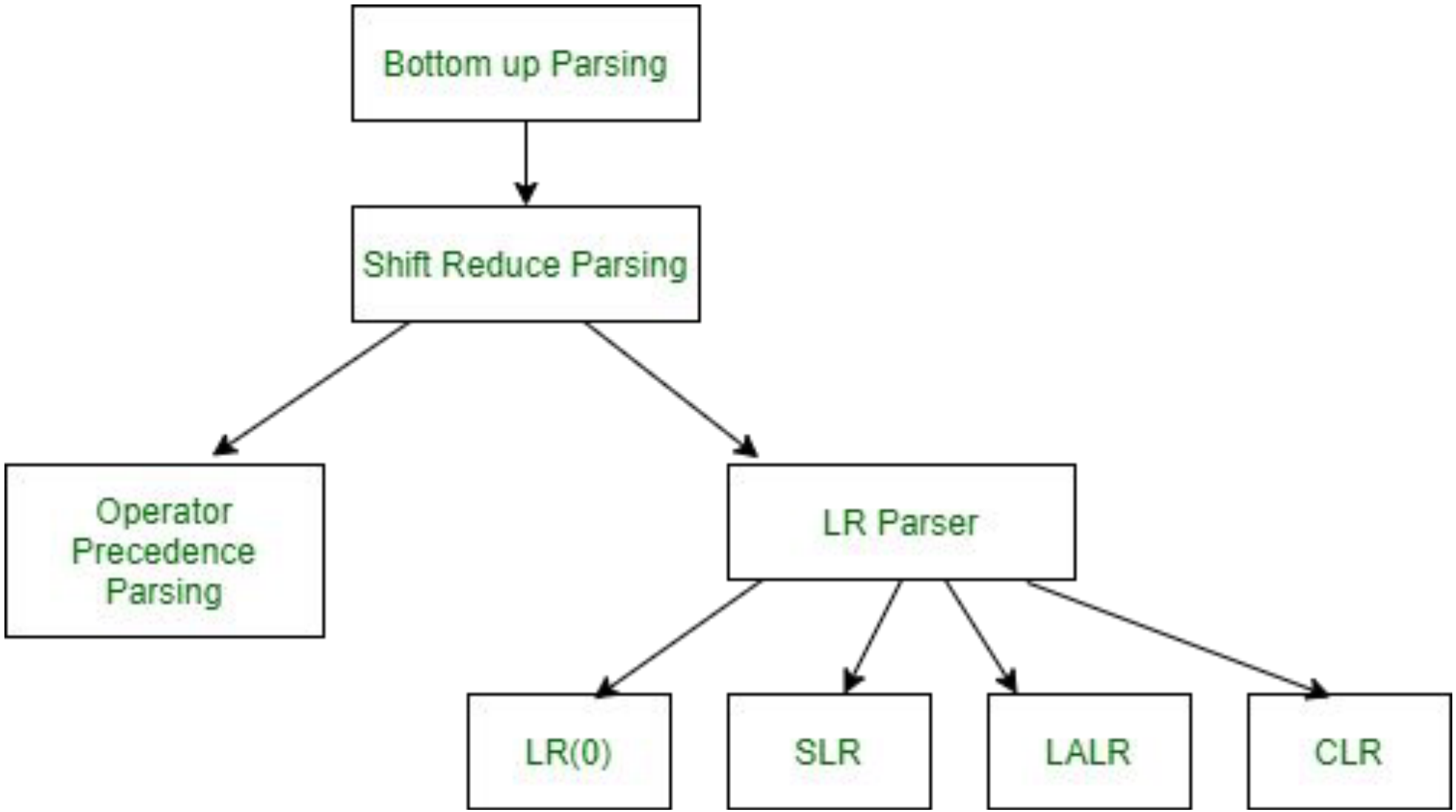
1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside. This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation.
In case (1), A is replaced by βBy, and then the rightmost nonterminal B in the body βBy is replaced by γ. In case (2), A is again expanded first, but this time the body is a string y of terminals only. The next rightmost nonterminal B will be somewhere to the left of y.

Grammar:
S -> S + S
S -> S – S
S -> (S)
S -> a
Input string: a1 – (a2 + a3)

Stack	Input String	Action
$	a1 – (a2 + a3) $	Shift a1
$a1	- (a2+ a3) $	Reduce by S -> a
$ S	- (a2+ a3) $	Shift -
$ S -	(a2+ a3) $	Shift (
$ S – (	a2+ a3) $	Shift a2
$ S – (a2	+ a3) $	Reduce by S -> a
$ S – (S	+ a3) $	Shift +
$ S – (S +	a3) $	Shift a3
$ S – (S + a3	)$	Reduce S -> a
$ S – (S + S	)$	Shift)
$ S – (S + S)	$	Reduce S -> S + S
$ S – (S)	$	Reduce by S -> (S)
$ S - S	$	Reduce by S -> S - S
$ S	S	Accept

LR Parsing


Bottom up Parsing
Shift Reduce Parsing
Operator Precedence Parsing
LR Parser
LR(0)
SLR
LALR
CLR

LR parser

- The term parser LR(k) parser, here the L refers to the left-to-right scanning, R refers to the rightmost derivation in reverse and k refers to the number of unconsumed “look ahead” input symbols that are used in making parser decisions. Typically, k is 1 and is often omitted.

1. The stack is empty, and we are looking to reduce the rule by S’→S$.

2. Using a “.” in the rule represents how many of the rules are already on the stack.

3. A dotted item, or simply, the item is a production rule with a dot indicating how much RHS has so far been recognized. Closing an item is used to see what production rules can be used to expand the current structure. It is calculated as follows:

Rules for LR parser :
The rules of LR parser as follows.

- The first item from the given grammar rules adds itself as the first closed set.
- If an object is present in the closure of the form A→ α. β. γ, where the next symbol after the symbol is non-terminal, add the symbol's production rules where the dot precedes the first item.
- Repeat steps (B) and (C) for new items added under (B).

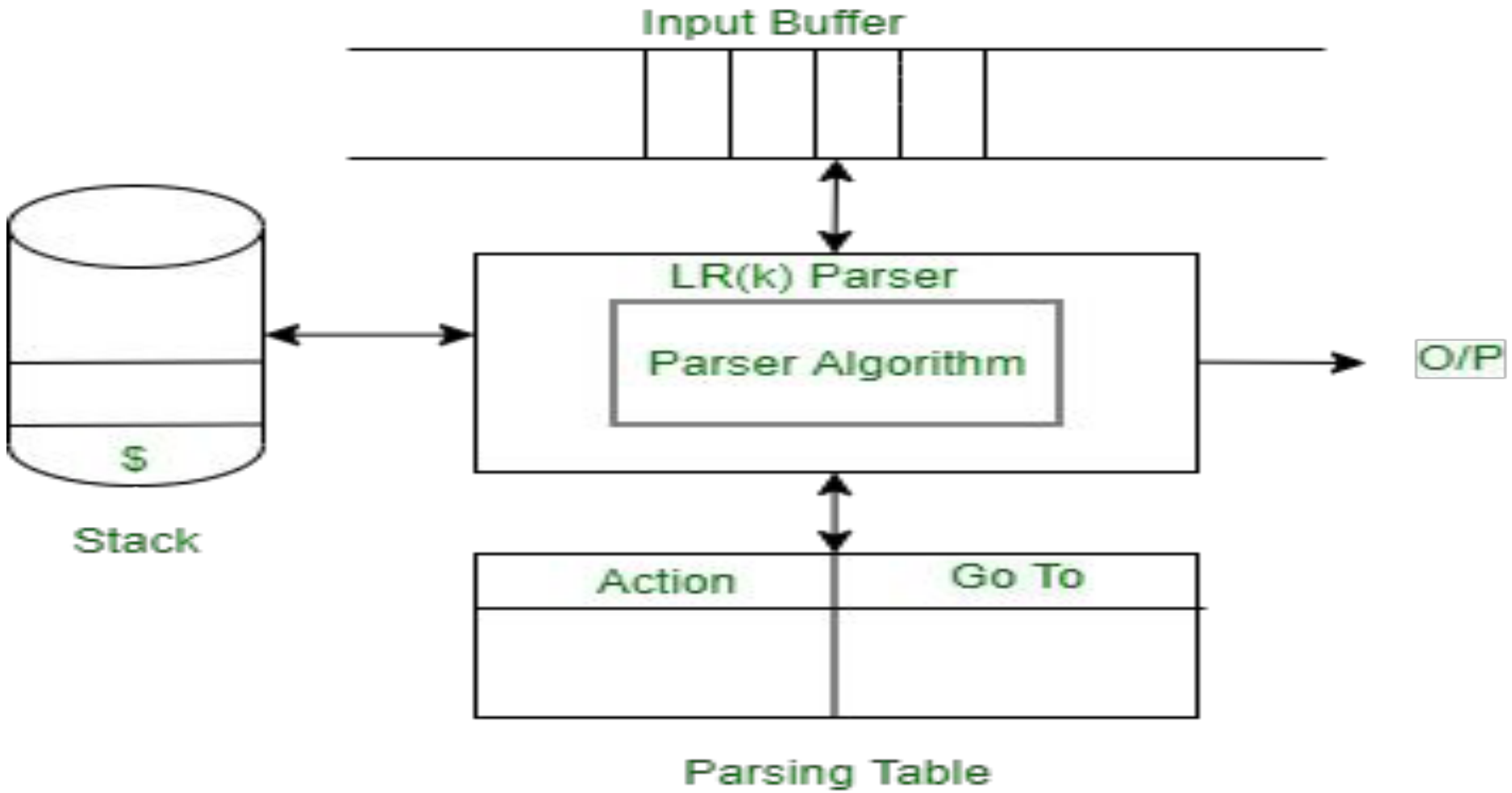
LR parser algorithm :
LR Parsing algorithm is the same for all the parser, but the parsing table is different for each parser. It consists following components as follows.

- **Input Buffer –**
  It contains the given string, and it ends with a $ symbol.
- **Stack –**
  The combination of state symbol and current input symbol is used to refer to the parsing table in order to take the parsing decisions.

Parsing Table :
Parsing table is divided into two parts- Action table and Go-To table. The **action table** gives a grammar rule to implement the given current state and current terminal in the input stream. There are four cases used in action table as follows.

- **Shift Action-** In shift action the present terminal is removed from the input stream and the state ***n*** is pushed onto the stack, and it becomes the new present state.
- **Reduce Action-** The number ***m*** is written to the output stream.
- The symbol ***m*** mentioned in the left-hand side of rule ***m*** says that state is removed from the stack.
- The symbol ***m*** mentioned in the left-hand side of rule **m** says that a new state is looked up in the goto table and made the new current state by pushing it onto the stack.

LR parser diagram :



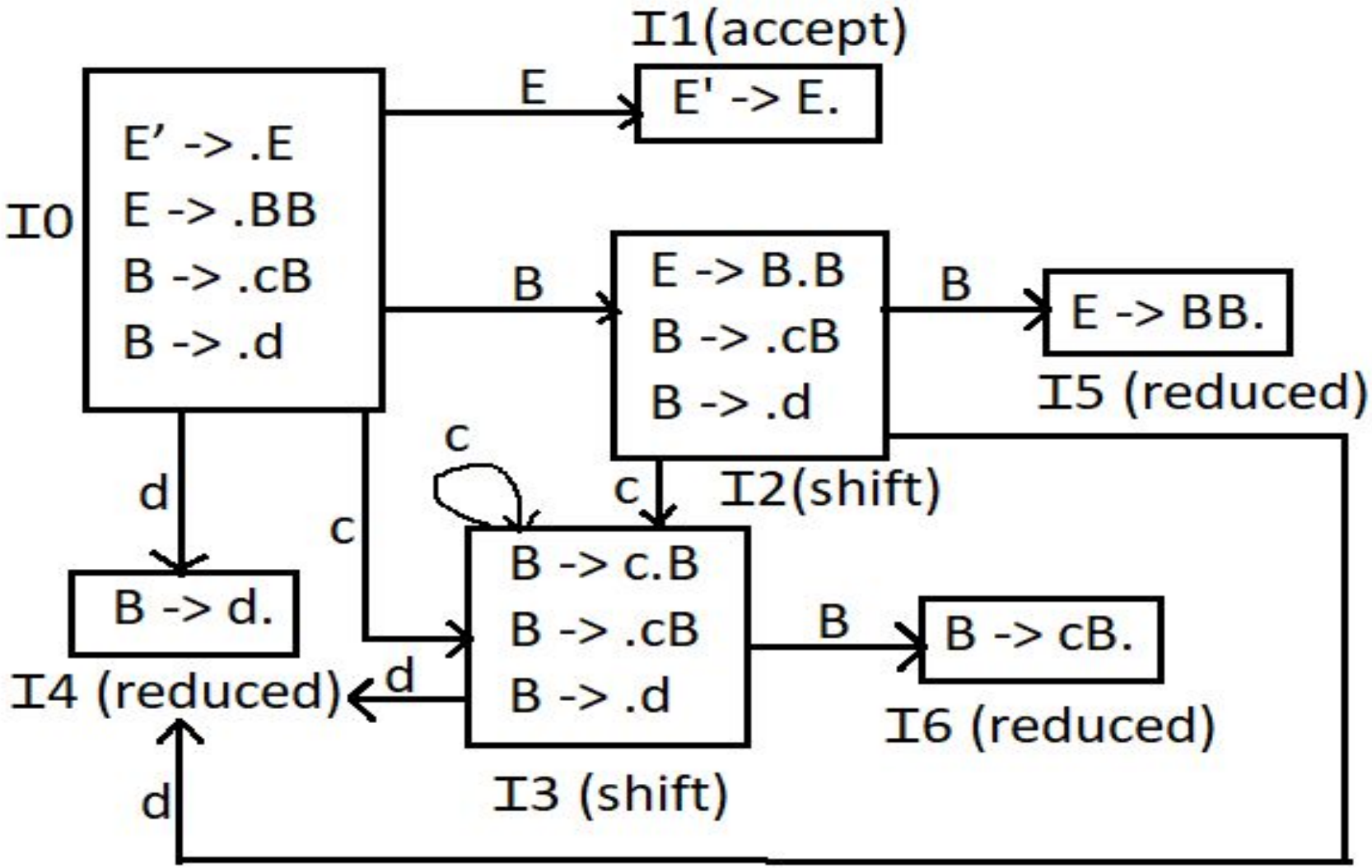
Steps to solve LR(0) Parser:

Step 1: Augment the given grammar
Step 2: Draw conical collection of LR(0) items / DFD
Step 3: Number the production
Step 4: Create the parsing table
Step 5: Stack implementation
Step 6: Draw parse tree

Example:
E -> BB
B -> cB | d Draw parse tree for ccdd.

Step1: Augment the grammar
E’ -> E
E -> BB
B -> cB | d

Step 2: Draw conical collection of LR(0) items



Step 3: Number the production

- E -> BB - 1
- B -> cB - 2
- B -> d - 3

Step 4:

Create the parsing table

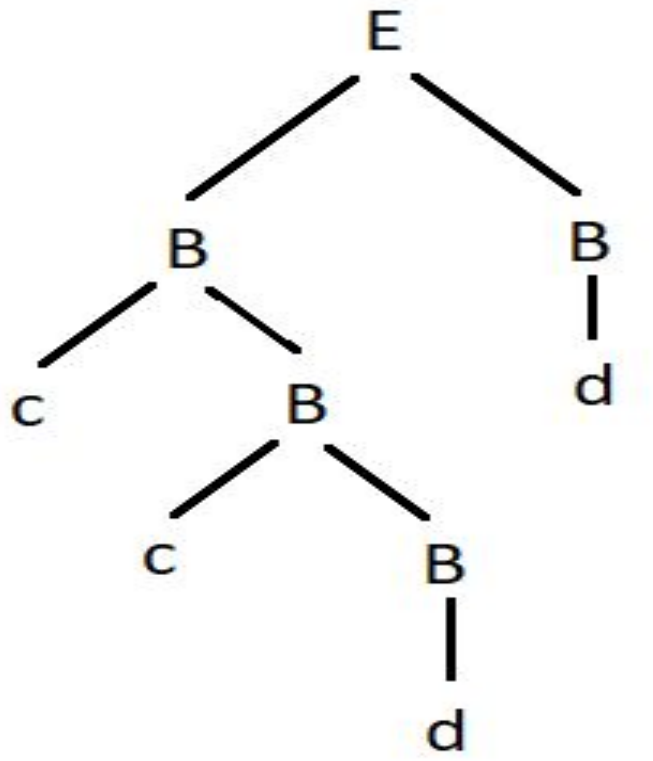
State	Action			Goto	
	c	d	$	E	B
0	s3	s4		1	2
1			accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

Step 5: Stack implementation

Stack	**Input String**	**Action**
$ 0	ccdd$	Shift c into stack (s3 state)
$ 0 c3	cdd$	Shift c (s3)
$ 0 c3 c3	dd$	Shift d (s4)
$ 0 c3 c3 d4	d$	Reduce by B -> d
$ 0 c3 c3 B6	d$	Reduce by B -> cB
$ 0 c3 B6	d$	Reduce by B -> cB
$ 0 B2	d$	Shift d (s4)
$ 0 B2 d4	$	Reduce by B -> d
$ 0 B2 B5	$	Reduce by E -> BB
$ 0E1	$	Accept

Step 6: Draw parse tree

Start from bottom of the stack implementation. Include reduce statements in rightmost derivation



Example 2:

E -> T + E | T

T -> id

Step 1: Augment the grammar

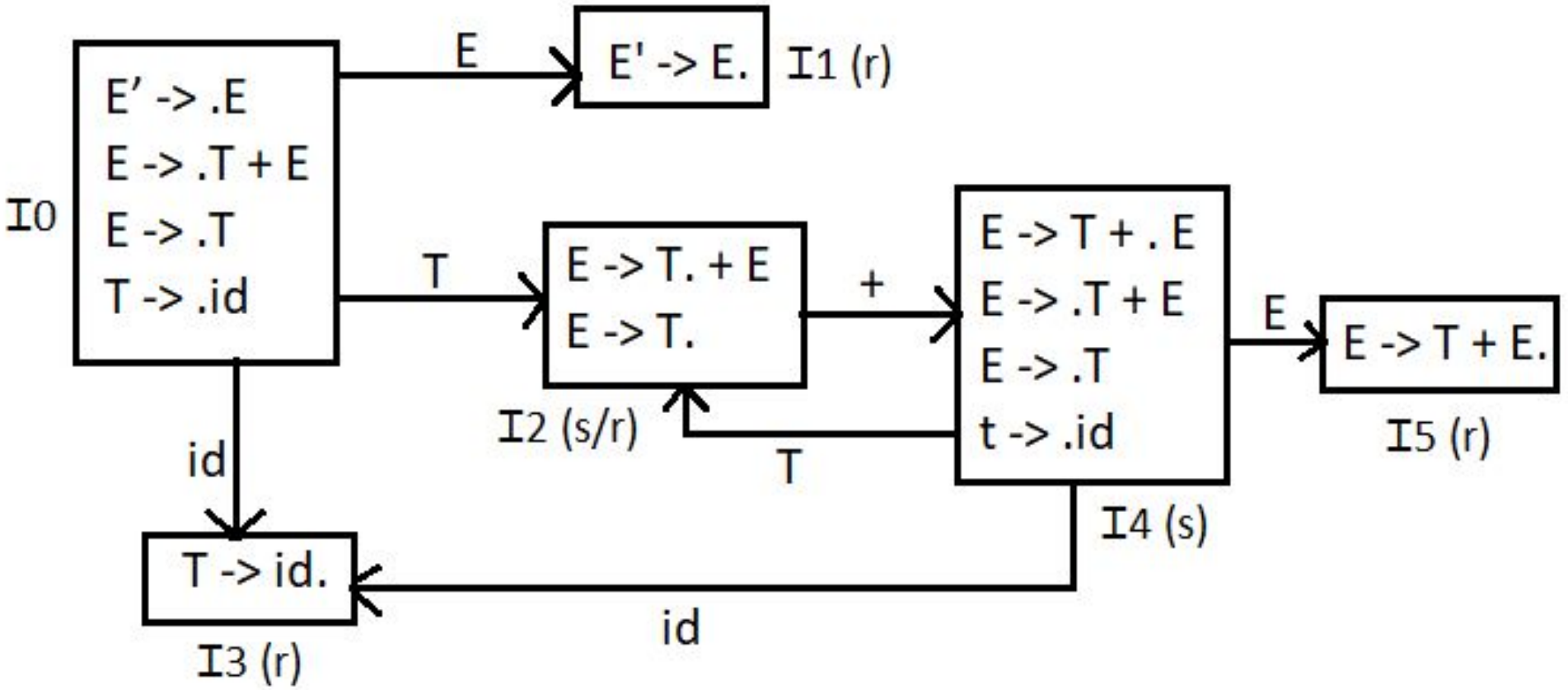
E’ -> E

E -> T + E

E -> T

T -> id

Step 2: Draw conical collection of LR(0) items



Step 3: Number the production

E -> T + E -1

E -> T -2

T -> id -3

Step 4:

Create the parsing table

There are 2 entries in one box in (I2, +)(shift-reduce conflict) that is not allowed so LR(0) parser for this grammar is not possible.

State	Action			Goto	
	+	id	$	E	T
0		s3		1	2
1			accept		
2	r2/s4	r2	r2		
3	r3	r3	r3		
4		s3		5	2
5	r1	r1	r1		

Solution: SLR parser

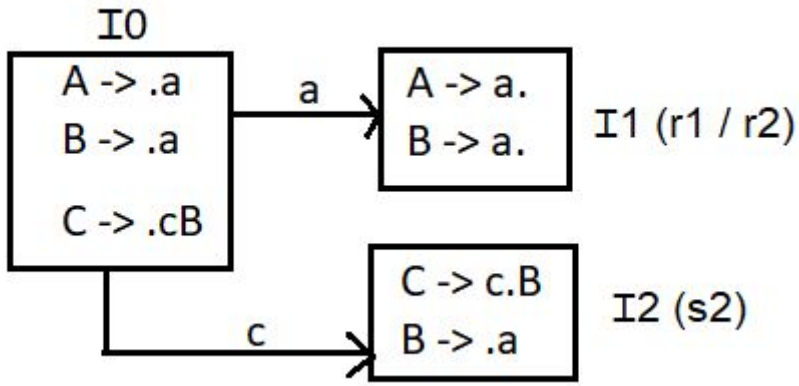
Conflicts in LR(0) parser

1. RR (reduce-reduce) conflict:

A -> .a -1

B -> .a -2

C -> .cB -3

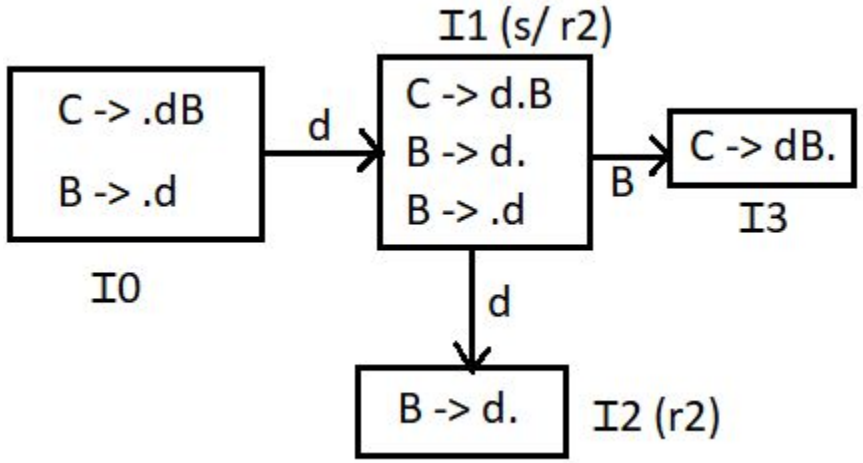


State	Goto		Action		
	a	c	A	B	C
0	s1	s2			
1	r1/r2	r1/r2			
2					

2. SR (shift-reduce) conflict:

C -> .dB -1

B -> .d -2



State	Goto	Action	
	d	B	C
0	s1		
1	r2/s2	3	
2	r2		

SLR Parser

SLR Parser

- SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing.
- The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states.
- We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items

Steps to solve SLR Parser:

Step 1: Augment the given grammar
Step 2: Draw parsing diagram
Step 3: Number the production
Step 4: Calculate FOLLOW of variables
Step 4: Create the parsing table
Step 5: Stack implementation
Step 6: Draw parse tree

Example:

E -> T + E | T

T -> id

Step 1: Augment the grammar

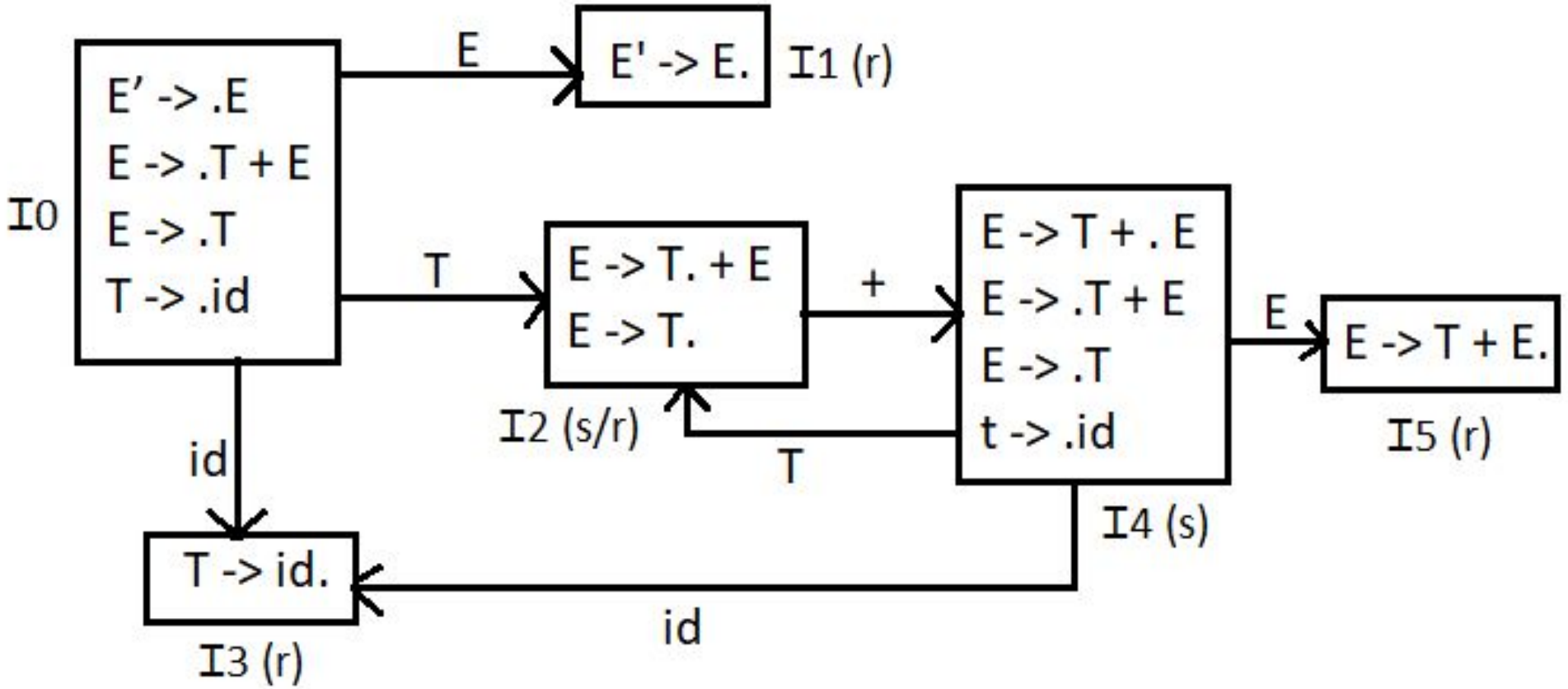
E’ -> E

E -> T + E

E -> T

T -> id

Step 2: Parsing diagram



Step 3: Number the production

E -> T + E -1

E -> T -2

T -> id -3

Step 4: Calculate FOLLOW of variables

FOLLOW(E) = {$}

FOLLOW(T) = {+, $}

Step 5: Create the parsing table

Entries of reduced state will be done in terminals that are in the FOLLOW of left hand side non terminal.

State	Action			Goto	
	+	id	$	E	T
0		s3		1	2
1			accept		
2	s4		r2		
3	r3		r3		
4		s3		5	2
5			r1		

Steps for constructing the SLR parsing table :

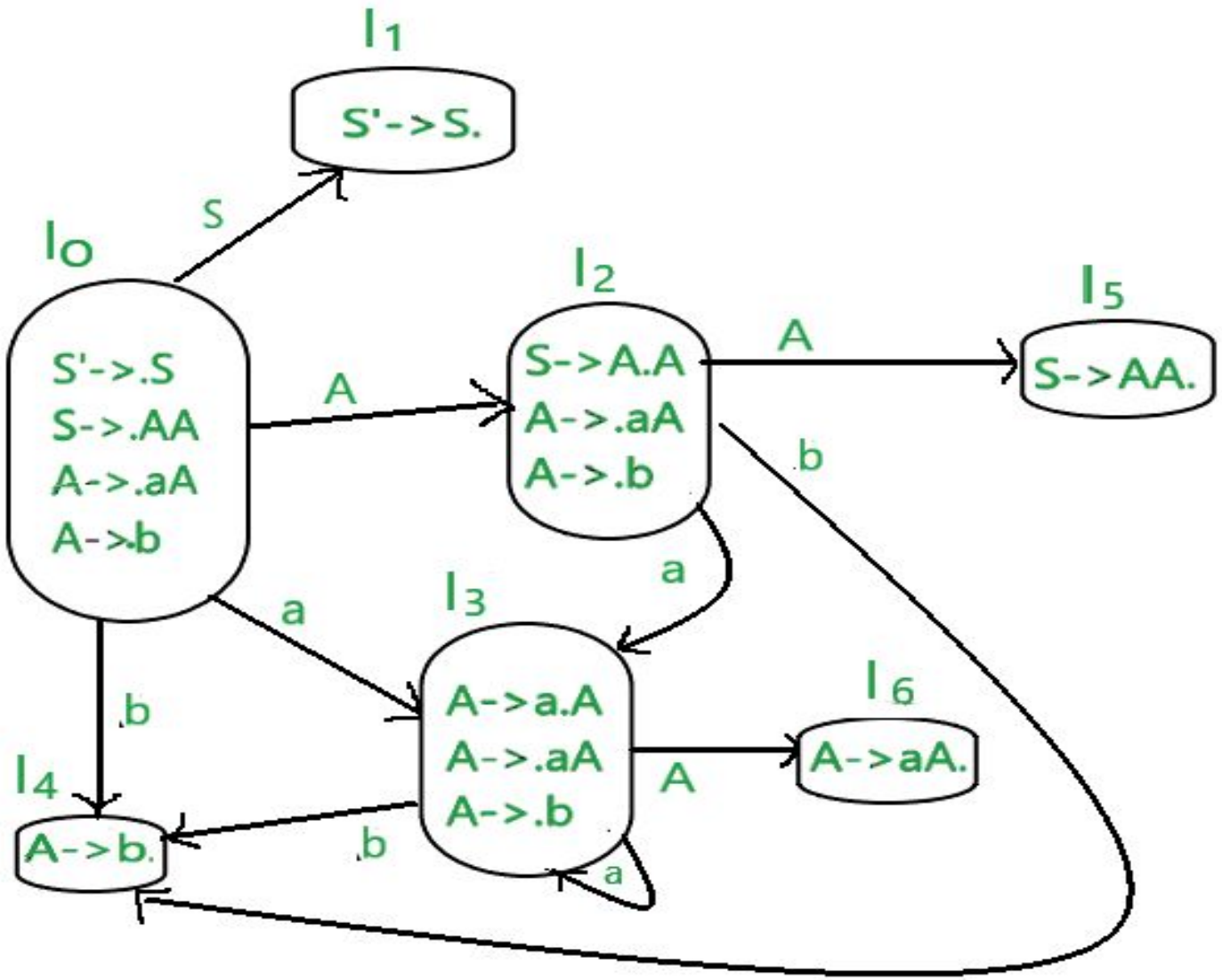
- Writing augmented grammar
- LR(0) collection of items to be found
- Find FOLLOW of LHS of production
- Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the parsing table.

Example: S–>AA
A–>aA |b

STEP1 – Find augmented grammar
The augmented grammar of the given grammar is:-

S'–>.S [0th production]
S–>.AA [1st production]
A–>.aA [2nd production]
A–>.b [3rd production]

Step 2: Find LR(0) collection of items



STEP3 –

Find FOLLOW of LHS of production

FOLLOW(S)=$

FOLLOW(A)=a, b, $

- **STEP 4-**

  Defining 2 functions: goto [list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		