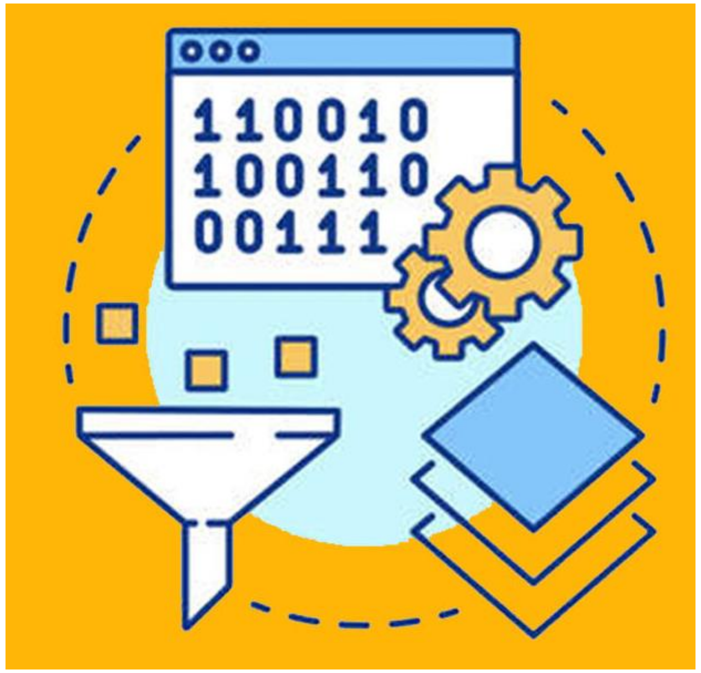
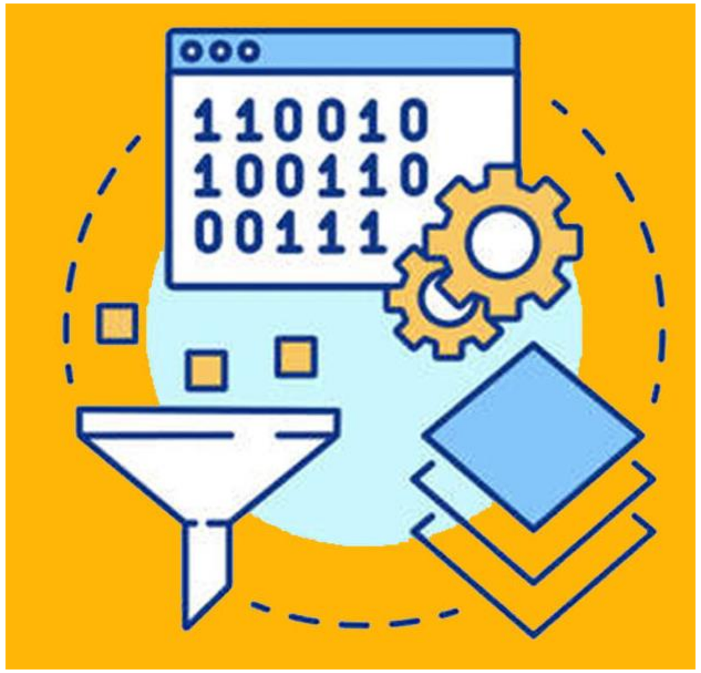
Compiler Design





Syntax Analysis

Bottom-up Parsing

Prof. S. D. PADIYA| SSGMCE, Shegaon
padiyasagar@gmail.com

2021-2022 (Spring)

Syllabus: Unit 01

Bottom-up Parsing:

Handle Pruning,

Stack implementation of Shift Reduce Parsing,

Conflicts during shift-reduce parsing.

LR parsers:

LR parsing algorithm,

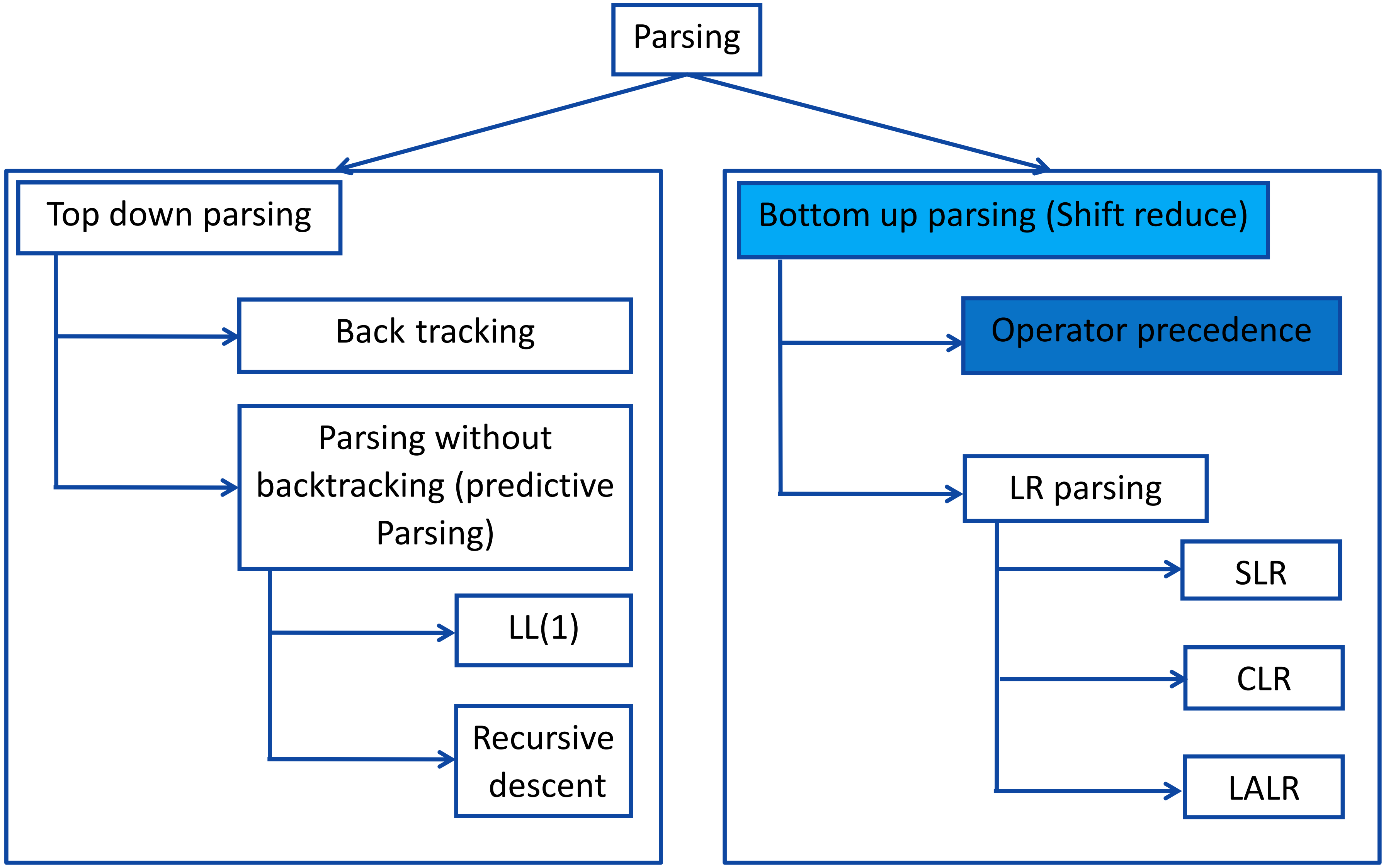
Construction of SLR parsing table,

Canonical LR parsing tables and canonical LALR parsing tables.

Error recovery in LR parsing,

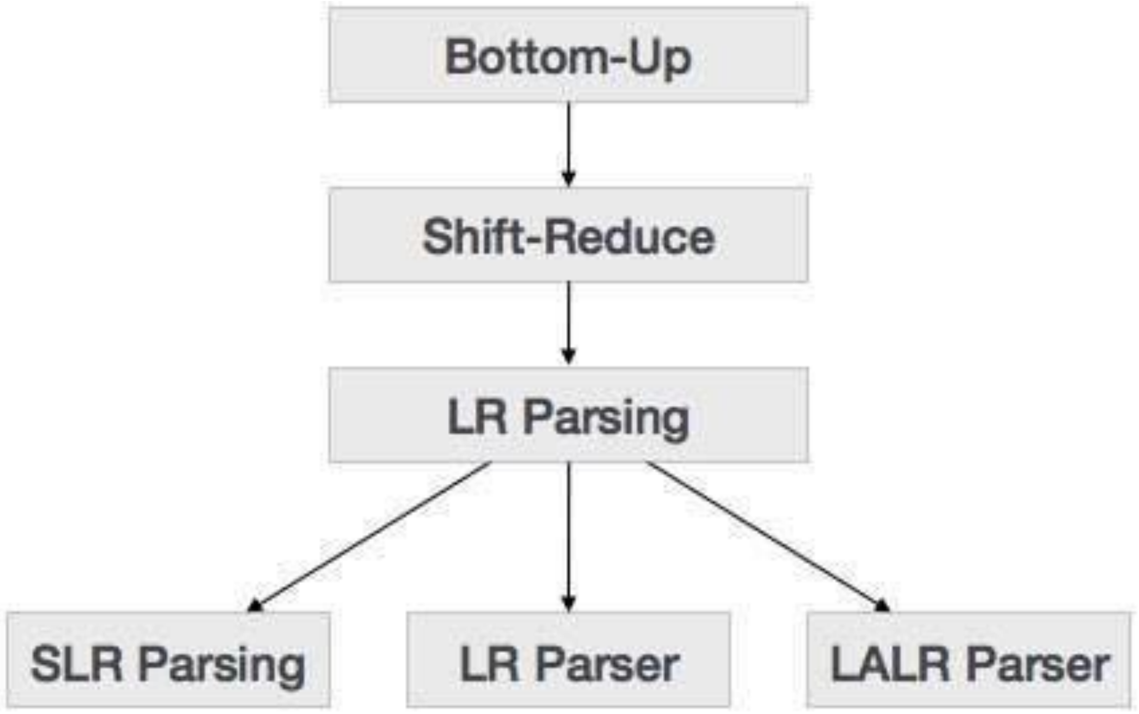
The parser generator YACC.

PARSING


Parsing
Top down parsing
Back tracking
Parsing without backtracking (predictive Parsing)
LL(1)
Recursive descent
Bottom up parsing (Shift reduce)
Operator precedence
LR parsing
SLR
CLR
LALR

PARSING

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



PARSING

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

1) HANDLE AND HANDLE PRUNING

Handle:

- A handle is a concept used in the bottom-up parsing of compiler design. Bottom-up parsing is the method of syntax analysis in which sentences belonging to certain context-free grammars are analyzed.
- This analysis results in the **construction of parse trees**. This construction begins at the leaf nodes and proceeds to the root non-terminal hence the name bottom up which is sometimes called the shift-reduce parsing.
- Each symbol is moved onto the stack until there is a matching right-hand side nonterminal. This is followed with a reduction by the replacement of the production's right-hand side by its left-hand side. This process continues until eventually the string is reduced to the start non-terminal.
- The set of strings to be replaced at each reduction step is called a handle.

1) HANDLE AND HANDLE PRUNING

Handle Pruning:

- The process of discovering a handle and reducing it to appropriate left hand side non terminal is known as handle pruning.

- This describes the process of identifying handles and reducing them to the appropriate left most non-terminals. It is the basis of bottom-up parsing and is responsible for the accomplishment of syntax analysis using bottom-up parsing.

- Right most derivation is achieved using handle pruning in reverse order.

1) HANDLE AND HANDLE PRUNING

Example-1: Derive the following given grammar to obtain the handle and reduce it to appropriate left hand side non terminal for the input string id1+id2*id3.

$E \rightarrow E+E \mid E*E \mid id$

Rightmost Derivation

E

$E+\underline{E}$

$E+E*\underline{E}$

$E+\underline{E}*id3$

$\underline{E}+id2*id3$

$id1+id2*id3$

Right sentential form	Handle	Production
id1+id2*id3	id1	E→id
E+id2*id3	id2	E→id
E+E*id3	id3	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

SDPadiya
SSGMCE

1) HANDLE AND HANDLE PRUNING

Example-2: Derive the following given grammar to obtain the handle and reduce it to appropriate left hand side non terminal for the input string id*id.

E→ E+E

E→ E*E

E→ id

Example-3: Derive the following given grammar to obtain the handle and reduce it to appropriate left hand side non terminal for the input string abbcde.

S→ aABe

A→ Abc | b

B→ d

2) SHIFT REDUCE PARSER

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

2) SHIFT REDUCE PARSER

Shift Reduce Parser is a type of Bottom-Up Parser. It generates the Parse Tree from **Leaves to the Root**. In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.

Shift Reduce Parser requires two Data Structures

- Input Buffer
- Stack

There are the various steps of Shift Reduce Parsing which are as follows −

- It uses a stack and an input buffer.
- Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.

2) SHIFT REDUCE PARSER

The shift reduce parser performs following basic operations:

1. **Shift**: Moving of the symbols from input buffer onto the stack, this action is called shift.

2. **Reduce**: If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.

3. **Accept**: If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.

4. **Error**: A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.

2) SHIFT REDUCE PARSER

Example-1: Perform the Bottom-Up Parsing for the id+id*id as per the following grammar by shift reduce parser.

E→ E+T | T

T→ T*F | F

F→ id

Stack	Input Buffer	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce F→id
$F	+id*id$	Reduce T→F
$T	+id*id$	Reduce E→T
$E	+id*id$	Shift
$E+	id*id$	Shift
$E+id	*id$	Reduce F→id
$E+F	*id$	Reduce T→F
$E+T	*id$	Shift
$E+T*	id$	Shift
$E+T*id	$	Reduce F→id
$E+T*F	$	Reduce T→T*F
$E+T	$	Reduce E→E+T
$E	$	Accept

2) SHIFT REDUCE PARSER

Example-2: Perform the Bottom-Up Parsing for the given string on the Grammar, i.e., shows the reduction for string a1 – (a2 + a3) on the following grammar:

S→ S + S | S – S | (S) | a

Example-3: Perform the Bottom-Up Parsing for the given string on the Grammar, i.e., shows the reduction for string 10201 on the following grammar:

S→ 0S0 | 1S1 | 2

Example-4: Perform the Bottom-Up Parsing for the given string on the Grammar, i.e., shows the reduction for string 32423 on the following grammar:

E→ 2E2 | 3E3 | 4

2) SHIFT REDUCE PARSING: CONFLICT

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol,

cannot decide whether to shift or reduce i.e. a **Shift / Reduce Conflict**

or

cannot decide which of several reductions to make i.e. **Reduce / Reduce Conflict.**

2) SHIFT REDUCE PARSING: CONFLICT

Example 1: Perform the Bottom-Up Parsing for the given string on the Grammar, i.e., shows the reduction for string abbcde on the following Grammar:

S → aAB

A → Abc | b

B → d

Shift or Reduce Conflict

Example 2: Perform the Bottom-Up Parsing for the given string on the Grammar, i.e., shows the reduction for string abbcde on the following Grammar:

S → aABe | b

A → Abc | b

B → d

Reduce by S or Reduce by A Conflict

3) VIABLE PREFIX IN BOTTOM UP PARSING

Viable Prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.

This clearly means that a viable prefix has a handle at its rightmost end. Not all prefixes of right sentential form can appear on the stack of a shift reduce parser.

Example –

A → B + id

B → (E)

Derivation for the string (E)+id

Bottom-up parser gives reverse rightmost derivation, so in the above example if we get the string **(E)+id** during the reverse rightmost derivation then the following operations are performed:

3) VIABLE PREFIX IN BOTTOM UP PARSING

Example-1: Derive for the string (E)+id by constructing a shift reduce parser as per the given context free grammar.

A → B + id

B → (E)

Operation performed	Stack	Comments
(.E)+id	(	shift (
(E.)+id	(E	shift E
(E).+id	(E)	shift)
B.+id	B	reduce (E) to B
B+.id	B +	shift +
B+id.	B + id	shift id
A	A	reduce B + id to A

3) VIABLE PREFIX IN BOTTOM UP PARSING

As we can see in the above table before shifting + to the stack we have reduced (E) to B. So we can only have (, (E, (E) on stack but we cannot have (E)+ on stack because (E) is a handle and the items in the stack cannot exceed beyond the handle.

Here **(and (E** are all viable prefixes for the handle (E) and only these prefixes are present in stack of shift reduce parser.

So we keep on shifting the items until we reach the handle or an error occurs. Once a handle is reached we reduce it with a non-terminal using the suitable production. Thus viable prefixes help in taking appropriate shift-reduce decisions. As long as stack contains these prefixes there cannot be any error.

3) VIABLE PREFIX IN BOTTOM UP PARSING

Example-2: Derive for the string bbbaa by constructing a shift reduce parser as per the grammar.

$S \rightarrow AA$

$A \rightarrow bA \mid a$

Reverse Rightmost Derivation with Handles	Viable Prefix	Comments
bbba**a**	b, bb, bbb, bbbba	Here, a is the handle so viable prefix cannot exceed beyond a.
bb**bA**a	b, bb, bbb, bbbA	Here, bA is the handle so viable prefix cannot exceed beyond bA.
b**bA**a	b, bb, bbA	Here, bA is the handle so viable prefix cannot exceed beyond bA.
bAa	b, bA	Here, bA is the handle so viable prefix cannot exceed beyond bA.
A**a**	A, aA	Here, a is the handle so viable prefix cannot exceed beyond a.
AA	A, AA	Here, AA is the handle so viable prefix cannot exceed beyond AA.
S		

4) OPERATOR PRCEDENCE PARSING

Operator Grammar:

A Grammar in which there is no Є in RHS of any production or no adjacent non terminals is called operator grammar.

Example: $E \rightarrow EAE \mid (E) \mid id$

$A \rightarrow + \mid * \mid -$

Above grammar is not operator grammar because right side ***EAE*** has consecutive non terminals.

In operator precedence parsing we define following disjoint relations:

Relation	Meaning
a<·b	a "gives precedence to" b
a=b	a "has the same precedence as" b
a·>b	a "takes precedence over" b

4) OPERATOR PRCEDENCE PARSING

Precedence and Associativity of Operators:

Operator	Precedence	Associative
↑	1	right
*, /	2	left
+, -	3	left

4) OPERATOR PRCEDENCE PARSING

Steps to Operator Precedence Parser

1. Find Leading and trailing of non terminal
2. Establish relation
3. Creation of table
4. Parse the string

4) OPERATOR PRCEDENCE PARSING: Leading and Trailing

Leading:-

Leading of a non terminal is the first terminal or operator in production of that non terminal.

Trailing:-

Trailing of a non terminal is the last terminal or operator in production of that non terminal.

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

Non terminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

4) OPERATOR PRCEDENCE PARSING: Rules to establish a relation

For

1. a = b, ⇒ aAb, where A is ϵ or a single non terminal [e.g : (E)]

2. a <• b ⇒ $Operator\ Non\ Terminal$ $then\ (Operator < \bullet\ Leading(NT))$ [e.g : +T]

3. a •>b ⇒ $Non\ Terminal\ Operator$ $then\ (Trailing(NT) \bullet > Operator)$ [e.g : E+]

4. $ <• Leading (start symbol)

5. Trailing (start symbol) •> $

4) OPERATOR PRCEDENCE PARSING: Creation of a Table

Example-1: Derive for the string id+id*id by operator precedence parsing for the grammar.

E→ E +T | T

T→ T *F | F

F→ id

Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

Step 2: Establish Relation

a <·b

Op NT	*Op <• Leading (NT)*
+T	+ <• {*, id}
*F	* <• {id}

Step3: Creation of Table

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

4) OPERATOR PRCEDENCE PARSING: Creation of a Table

Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Step 2: Establish Relation

$a \cdot\!> b$

$NT\ \ Op$	$(Trailing(NT)) \bullet > Op$
$E\ +$	$\{+, *, id\} \bullet > \ +$
$T\ *$	$\{*, id\} \bullet > *$

Step3: Creation of Table

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

SDPadiya
SSGMCE

4) OPERATOR PRCEDENCE PARSING: Creation of a Table

Step 1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

E→ E+ T| T

T→ T* F| F

F→ id

Step 2: Establish Relation

$<· Leading (start symbol)

$$\$ \lessdot \{+, *, id\}$$

Trailing (start symbol) .> $

$$\{+, *, id\} \gtrdot \$$$

Step3: Creation of Table

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

4) OPERATOR PRCEDENCE PARSING: Parse the String

Step 4: Parse the string using precedence table

Assign precedence operator between terminals

String: id+id*id

$ id+id*id $

$ <· id+id*id$

$ <· id ·> +id*id$

$ <· id ·> + <· id*id$

$ <· id ·> + <· id ·> *id$

$ <· id ·> + <· id ·> *<· id$

$ <· id ·> + <· id ·> *<· id ·> $

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

4) OPERATOR PRCEDENCE PARSING: Rules to establish a relation

Step 4: Parse the string using precedence table

1. Scan the input string until first **·>** is encountered.
2. Scan backward until **<·** is encountered.
3. The handle is string between **<·** and **·>**

$\$ \underleftarrow{\underrightarrow{<\cdot\ Id\ \cdot>}} + <\cdot\ Id\ \cdot> * <\cdot\ Id\ \cdot> \$$	Handle id is obtained between <· and ·> Reduce this by F→id
$\$\ F + \underleftarrow{\underrightarrow{<\cdot\ Id\ \cdot>}} * <\cdot\ Id\ \cdot> \$$	Handle id is obtained between <· and ·> Reduce this by F→id
$\$\ F + F * \underleftarrow{\underrightarrow{<\cdot\ Id\ \cdot>}} \$$	Handle id is obtained between <· and ·> Reduce this by F→id
$ F + F * F $	Perform appropriate reductions of all non-terminals.
$ E + T * F $	Remove all non terminals.
$ + * $	Place relation between operators
$\$ <\cdot + \underleftarrow{\underrightarrow{<\cdot * >}}\$$	The * operator is surrounded by <· and ·>. This indicates * becomes handle so reduce by T→T*F.
$\$ \underleftarrow{\underrightarrow{<\cdot + >}}\$$	+ becomes handle. Hence reduce by E→E+T.
$ $	Parsing Done

C O M P I L E R D E S I G N

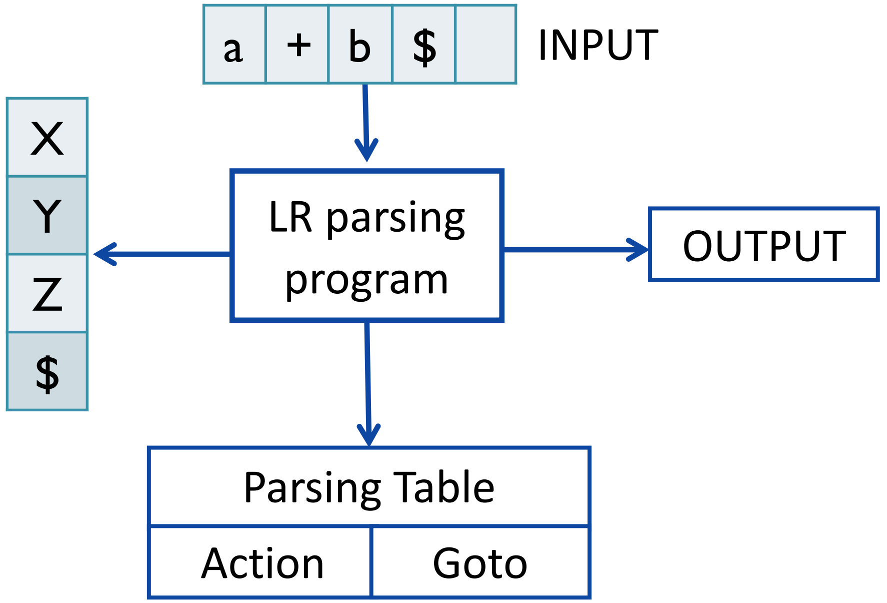
SDPadiya
SSGMCE

5) LR(k) PARSING

LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.

The technique is called LR(k) parsing:

1. The "L" is for left to right scanning of input symbol,
2. The "R" for constructing right most derivation in reverse,
3. The "k" for the number of input symbols of look ahead that are used in making parsing decision.



5) LR(k) PARSING

LR parser consists of an input, an output, a stack, a driver program, and a parsing table that have two parts: **action and goto.**

The driver program is same for all the LR parsers, only the parsing table changes from one-parser to another.

The parsing programs reads characters from an input buffer one at a time. The program uses a stack to store a string of the form X_0S_0, X_1S_1, X_2S_2.......X_mS_m.

Where,

Each X_i is a grammar symbol and

Each S_i is a symbol called a state.

5) LR(k) PARSING

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.

There are three widely used algorithms available for constructing an LR parser

1) LR(0) and SLR(1) – Simple LR Parser
2) CLR(1) – Canonical LR Parser
3) LALR(1) – Look-Ahead LR Parser

6) LR(0) PARSER

Steps to LR(0) Parsing

1. Augment given grammar.
2. Find LR(0) items by inserting '•' symbol at the first position for every production in G.
3. Number the Production.
4. Construct Canonical Collection of LR(0) items.
5. Construct LR(0) Parsing Table.
6. Stack Implementation.
7. Draw Parse Tree.

6) LR(0) PARSER

LR(0) items:

An LR(0) item (item) of a grammar G is a production of G with a dot at some position of the right side.

Thus, production $A \rightarrow XYZ$ yields the four items,

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet YZ$

$A \rightarrow XY \bullet Z$

$A \rightarrow XYZ \bullet$

The production $A \rightarrow \varepsilon$ generates only one item,

$A \rightarrow \bullet$

6) LR(0) PARSER

LR(0) items:

An item indicates how much of a production we have seen at a given point in the parsing process.

For example,

the first item $A \rightarrow \bullet XYZ$ above indicates that we hope to see a string derivable from **XYZ** next on the input.

the second item $A \rightarrow X \bullet YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ.

6) LR(0) PARSER

Augmented grammar:

If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and production **S' → S**.

The purpose of this new production is to indicate to the parser when it should stop parsing and announce acceptance of the input.

Example:

S → BA

B → aB | b

Solution: The augmented grammar of the given grammar is:-

S' → S	[0th production]
S → BA	[1st production]
B → aB	[2nd production]
B → b	[3rd production]

6) LR(0) PARSER

Augmented grammar and LR(0) items:

Example-1:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Solution: The augmented grammar of the given grammar is:-

$S' \rightarrow \bullet S$	[0th production]
$S \rightarrow \bullet AA$	[1st production]
$A \rightarrow \bullet aA$	[2nd production]
$A \rightarrow \bullet b$	[3rd production]

6) LR(0) PARSER

Augmented grammar and LR(0) items:

Example-2:

E → E

E → E + T | T

T → T * F | F

F → id

Solution: The augmented grammar of the given grammar is:-

E\` → •E

E → •E + T

E → •T

T → •T * F

T → •F

F → •id

6) LR(0) PARSER

Augmented grammar and LR(0) items:

Example-3:

E → T + E | T

T → id

Solution: The augmented grammar of the given grammar is:-

E` → •E

E → •T + E

E → •T

T → •id

6) LR(0) PARSER

Augmented grammar and LR(0) items:

Example-4:

E → BB

B → cB | d

Solution: The augmented grammar of the given grammar is:-

E\` → •E

E → •BB

B → •cB

B → •d

6) LR(0) PARSER

Augmented grammar and LR(0) items:

Example-5:

S → aAD | bBb | aBe | bAe

A → C

B → C

Solution: The augmented grammar of the given grammar is:-

S` → •S

S → •aAD

S → •bBb

S → •aBe

S → •bAe

6) LR(0) PARSER

Example-1: Derive for the string aaab by constructing a LR(0) parsing table for the given context free grammar.

S → AA

A → aA | b

Solution:

Step 1) The augmented grammar of the given grammar is:-

S' → •S	[0th production]
S → •AA	[1st production]
A → •aA	[2nd production]
A → •b	[3rd production]

6) LR(0) PARSER

Step 2) Construct Canonical set of LR(0) items

S → AA

A → aA | b

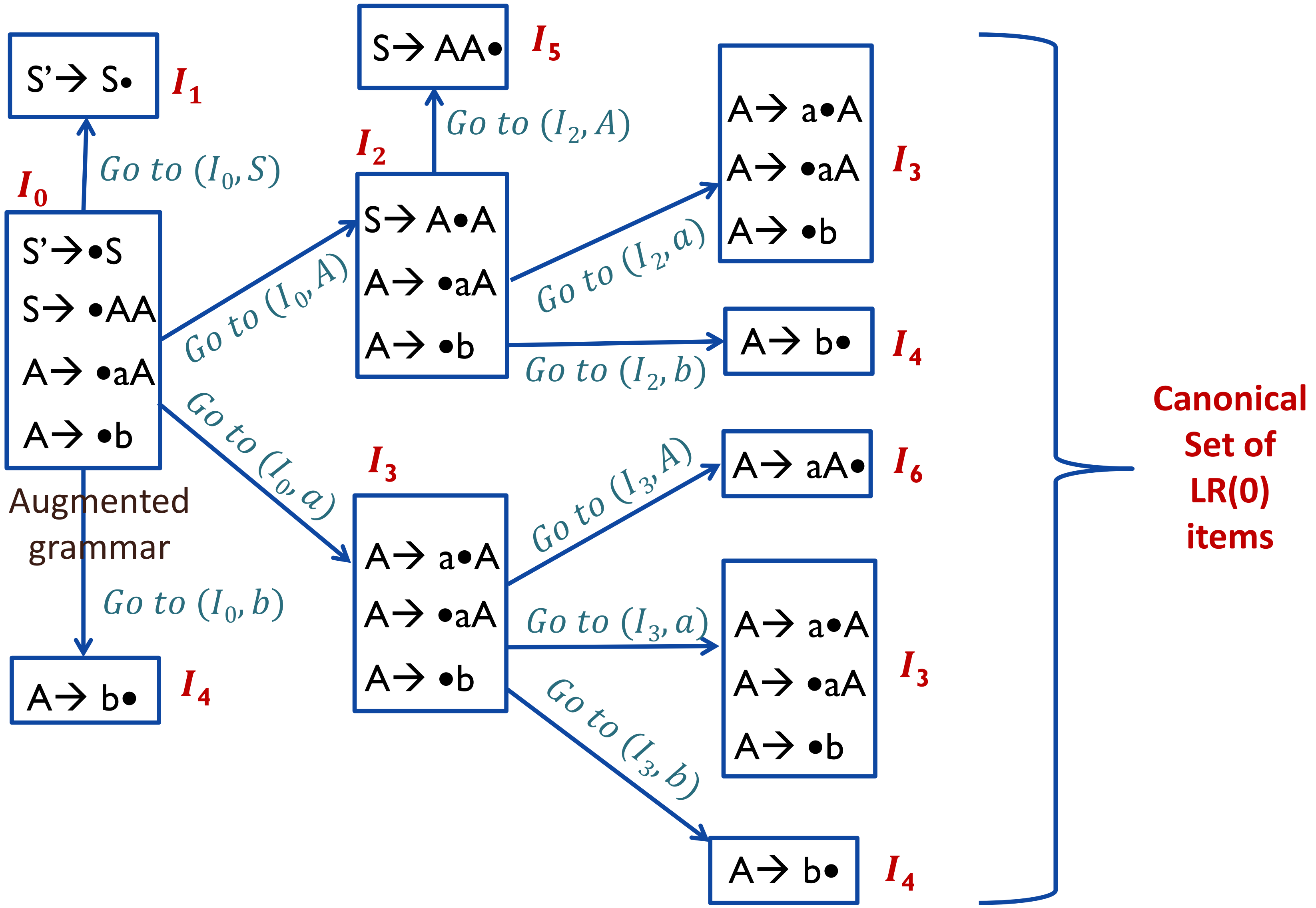
S' → •S

S → •AA

A → •aA

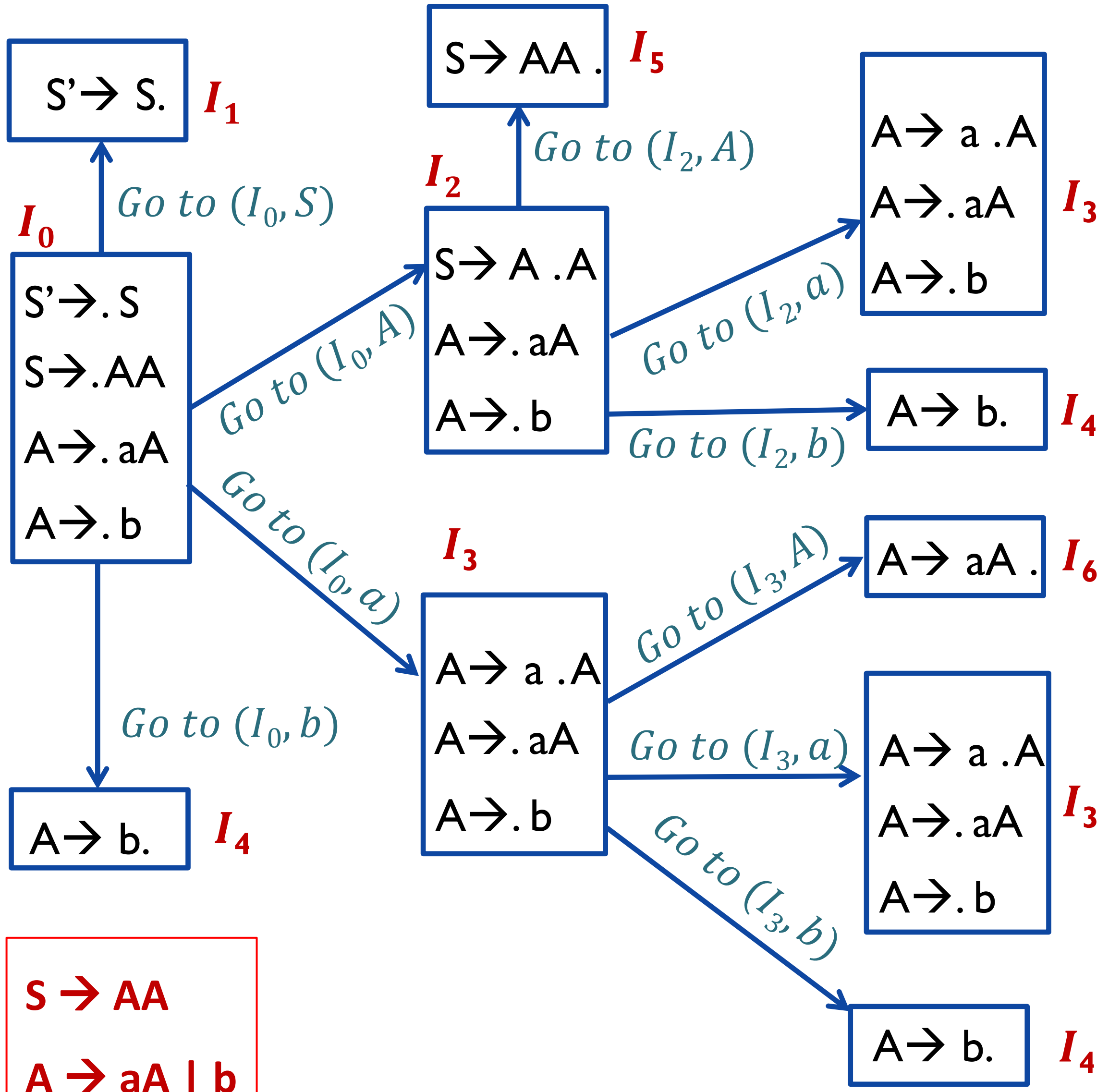
A → •b

Augmented grammar



6) LR(0) PARSER

Step 3) Construct LR(0) parsing table



S → AA

A → aA | b

$Follow(S) = \{\$\}$
$Follow(A) = \{a, b, \$\}$

	Action			**Go to**	
Item set	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

SDPadiya
SSGMCE

6) LR(0) PARSER

The augmented grammar of the given grammar is:-

S’ → •S	[0th production] = r0
S → •AA	[1st production] = r1
A → •aA	[2nd production] = r2
A → •b	[3rd production] = r3

Item set	Action			Go to	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Derive for the string “aaab”

Stack	Input	Action
$0	aaab$	Goto S3
$0a3	aab$	Goto S3
$0a3a3	ab$	Goto S3
$0a3a3a3	b$	Goto S4
$0a3a3a3b4	$	R3
$0a3a3a3A6	$	R2
$0a3a3A6	$	R2
$0a3A6	$	R2
$0A2	$	NOT ACCEPT

6) LR(0) PARSER

The augmented grammar of the given grammar is:-

S' → •S	[0th production] = r0
S → •AA	[1st production] = r1
A → •aA	[2nd production] = r2
A → •b	[3rd production] = r3

	Action			Go to	
Item set	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Derive for the string "aaabb"

Stack	Input	Action
$0	aaabb$	Goto S3
$0a3	aabb$	Goto S3
$0a3a3	abb$	Goto S3
$0a3a3a3	bb$	Goto S4
$0a3a3a3b4	b$	R3
$0a3a3a3A6	b$	R2
$0a3a3A6	b$	R2
$0a3A6	b$	R2
$0A2	b$	S4
$0A2b4	$	R3
$0A2A5	$	R1
$0S1	$	ACCEPT

SDPadiya
SSGMCE

6) LR(0) PARSER

Example-2: Construct a LR(0) parsing table for the given context free grammar.

E → T + E | T

T → id

Solution:

Step 1) The augmented grammar of the given grammar is:-

E' → •E	[0th production]
E → •T+E	[1st production]
E → •T	[2nd production]
T → •id	[3rd production]

6) LR(0) PARSER

Step 2) Construct Canonical set of LR(0) items

E → T + E | T

T → id

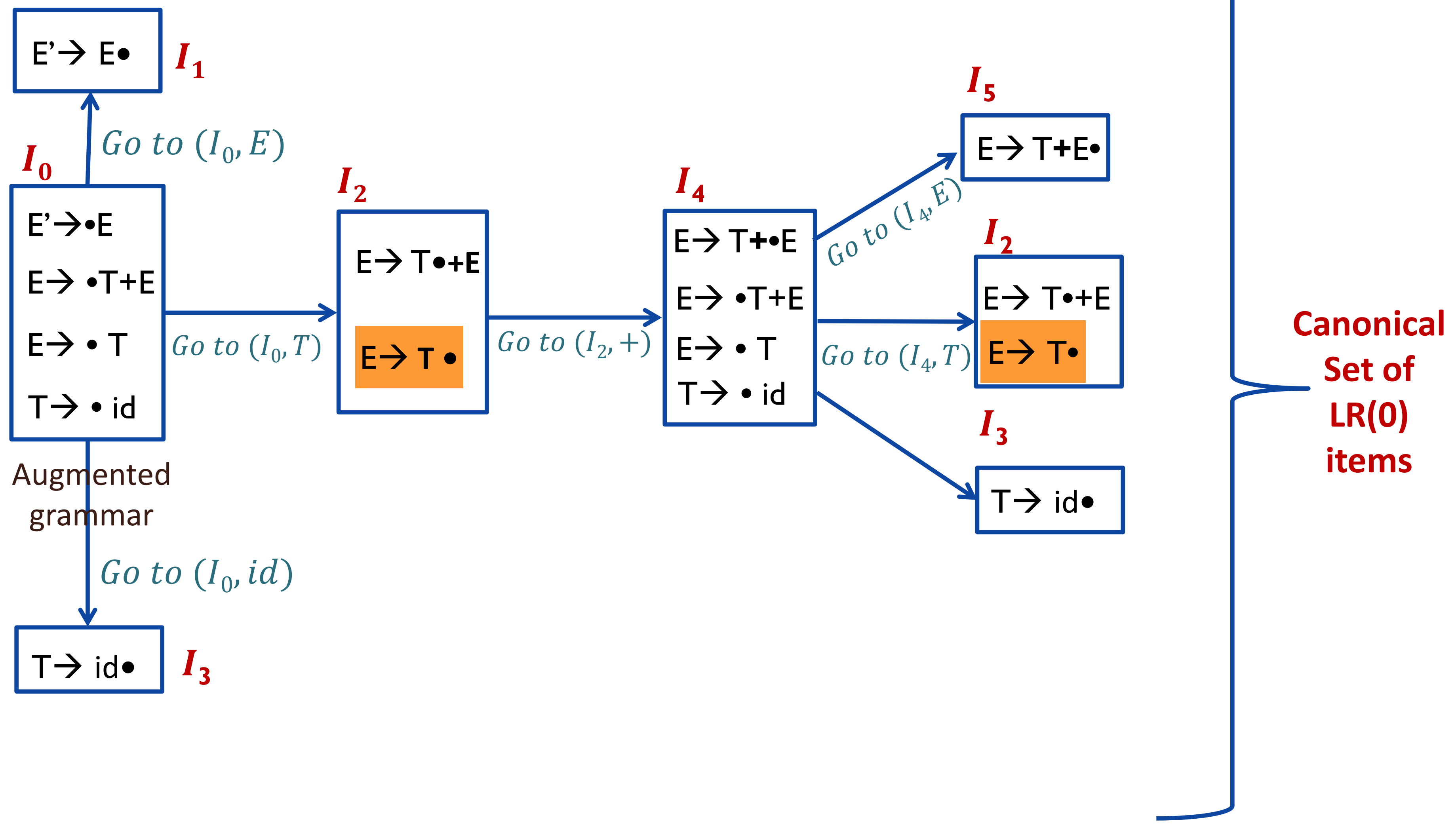
E' → •E

E → •T+E

E → •T

T → •id

Augmented grammar



6) LR(0) PARSER

The augmented grammar of the given grammar is:-

E’ → •E	[0th production]
E → •T+E	[1st production]
E → •T	[2nd production]
T → •id	[3rd production]

	Action			Go to	
Item Set	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		

Derive for the string “id + id + id”

Stack	Input	Action
$0	id+id+id$	Goto S3
$0id3	+id+id$	R3
$0T2	+id+id$	S4/R2
$0E1	+id+id$	NOT ACCEPT
$0T2+4	id+id$	S3
$0T2+4id3	+id$	R3
$0T2+4T2	+id$	S4/R2
$0T2+4E5	+id$	R1
$0E1	+id$	NOT ACCEPT

6) LR(0) PARSER

The augmented grammar of the given grammar is:-

E’ → •E	[0th production]
E → •T+E	[1st production]
E → •T	[2nd production]
T → •id	[3rd production]

	Action			Go to	
Item Set	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		

In the generated LR(0) parsing table, every cell does not has unique entry.

Cell $Go\ to\ (I_2, +)$ have two entries as S4/R2, it causes a shift or reduce conflict hence it is not a LR(0) grammar.

6) LR(0) PARSER

Example-3: Construct a LR(0) parsing table for the given context free grammar.

E → BB

B → cB | d

7) SLR PARSER

The Simple LR also called as SLR is first type of LR Parsing, it is easiest but weakest method among the available three.

It is the smallest class of grammar having few number of states.

A parsing table constructed by this method is called SLR parsing table,

LR parser using an SLR parsing table is called SLR parsing table, and

A grammar for which an SLR parser can be constructed is said to be an SLR grammar.

7) 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

7) SLR PARSER

Steps to construct SLR parser

1. Augment given grammar.
2. Number the Production.
3. Find FOLLOW() for each non terminals.
4. Find LR(0) items by inserting '•' symbol at the first position for every production in G.
5. Construct Canonical Collection of LR(0) items.
6. Construct SLR(1) Parsing Table (add reduce term for the FOLLOW() of left hand side non terminal).
7. Stack Implementation.
8. Draw Parse Tree.

NOTE

Add reduce only for the FOLLOW() of left hand side non terminals

7) SLR PARSER

Example-1: Check SLR(1) grammar by constructing a SLR(1) parsing table for the given context free grammar.

E → T + E | T

T → id

Solution:

Step 1) The augmented grammar of the given grammar is:-

E' → •E	[0th production]
E → •T+E	[1st production]
E → •T	[2nd production]
T → •id	[3rd production]

Step 2) The FOLLOW() for each non terminal

FOLLOW(E) = {$}

FOLLOW(T) = {+, $}

6) SLR PARSER

Step 2) Construct Canonical set of LR(0) items

E → T + E | T

T → id

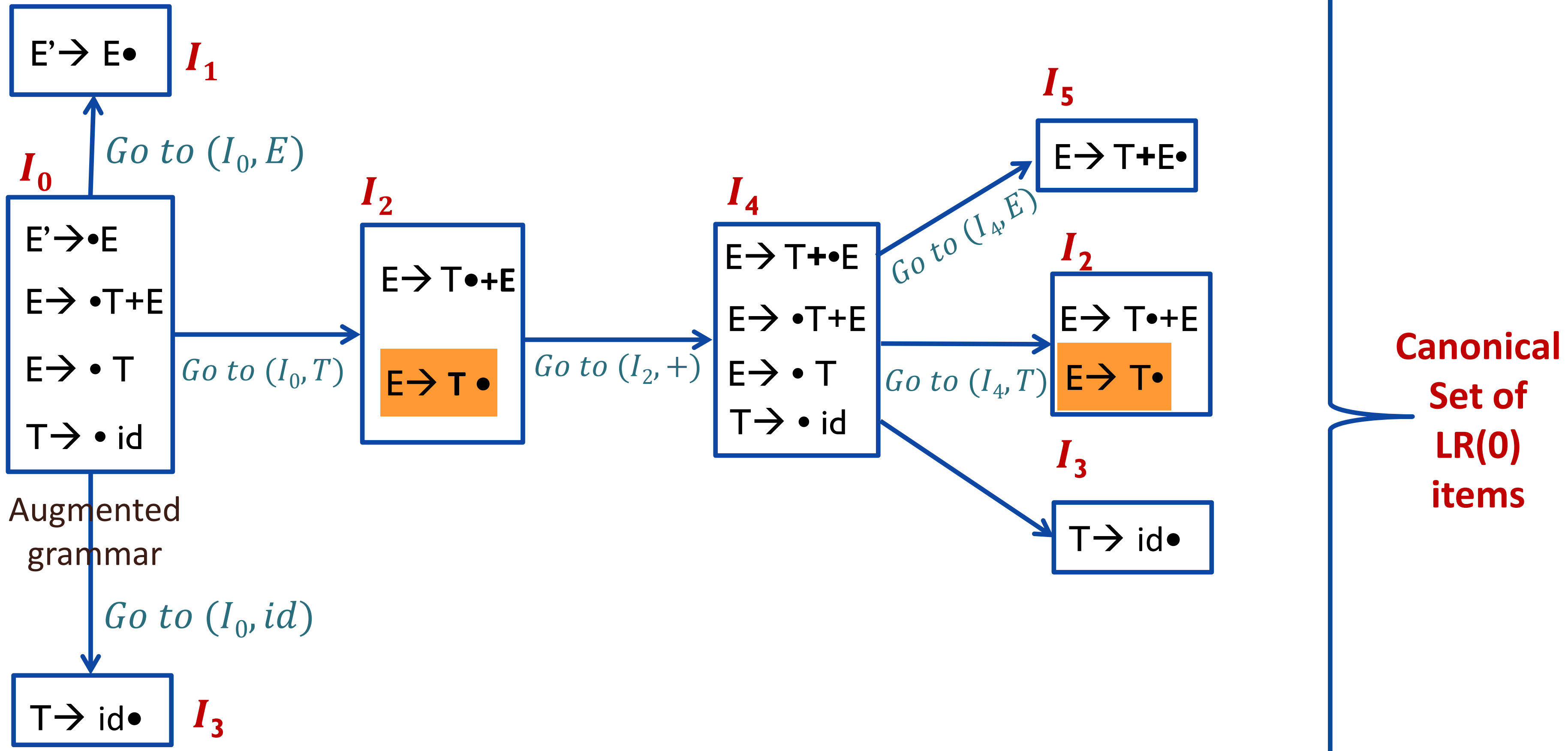
E' → •E

E → •T+E

E → •T

T → •id

Augmented grammar



6) SLR PARSER

The augmented grammar of the given grammar is:-

E’ → •E	[0th production]
E → •T+E	[1st production]
E → •T	[2nd production]
T → •id	[3rd production]

	Action			**Go to**	
Item Set	Id	+	$	E	T
0	S3			1	2
1			Accept		
2		S4	R2		
3		R3	R3		
4	S3			5	2
5			R1		

FOLLOW(E) = {$}

therefore only **(I2, $)** has **R2** entry.

FOLLOW(T) = {$, +}

therefore **(I3, +)** and **(I3, $)** have **R3** entry.

FOLLOW(E) = {$}

therefore **(I5, $)** has **R1** entry.

In the generated SLR(1) parsing table, every cell has unique entry, therefore it is a SLR(1) grammar.

6) SLR PARSER

Example-2: Construct the SLR Parsing table for the following grammar. Also, Parse the input string a * b + a.

E → E + T| T

T → TF| F

F → F*| a| b

Solution: Step 1) The augmented grammar is:-

E′ → • E	[0th production]
E → • E + T	[1st production]
E → • T	[2nd production]
T → • TF	[3rd production]
T → • F	[4th production]
F → • F *	[5th production]
F → • a	[6th production]
F → • b	[7th production]

Step 2) The FOLLOW() for each non terminal

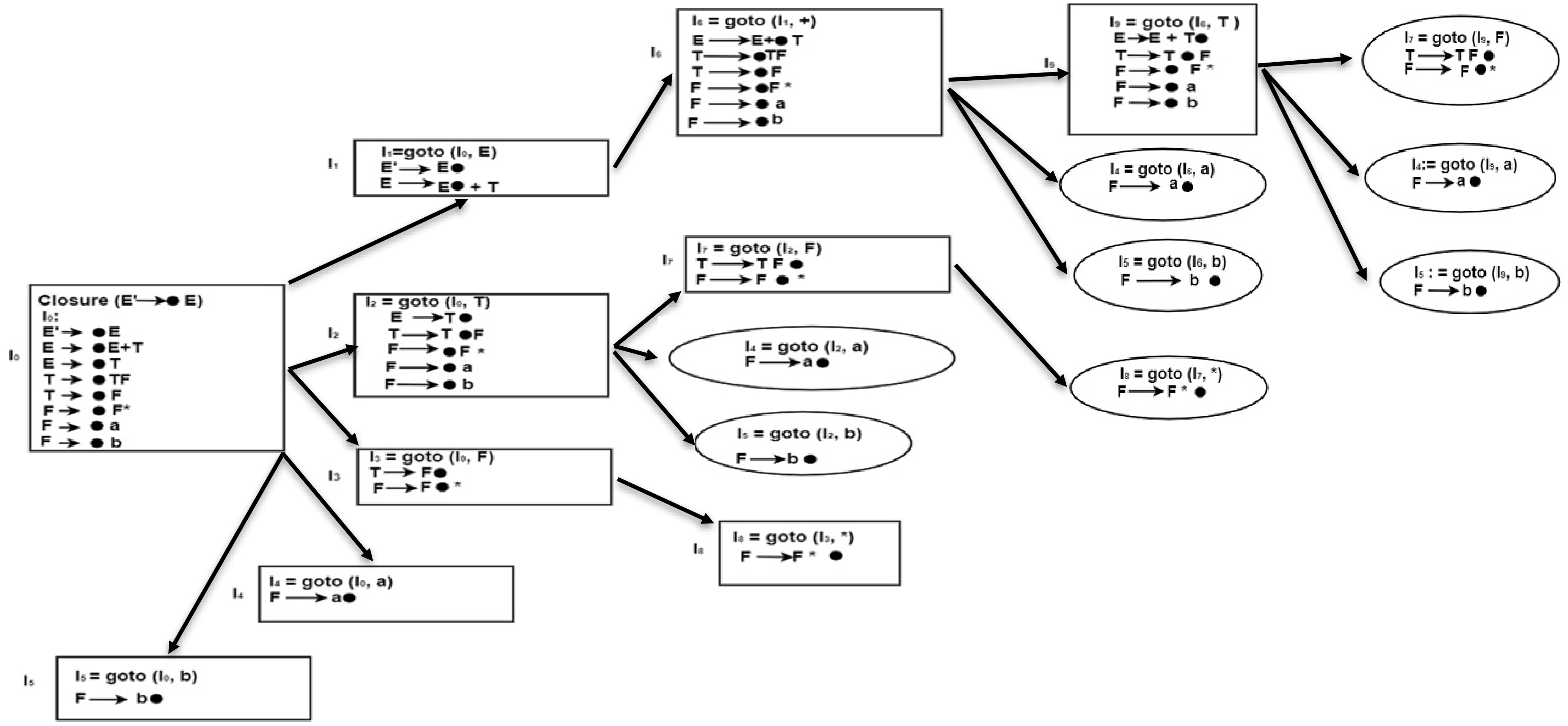
FOLLOW(E) = {+, $}

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

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

COMPILER DESIGN

6) SLR PARSER



SDPadiya
SSGMCE

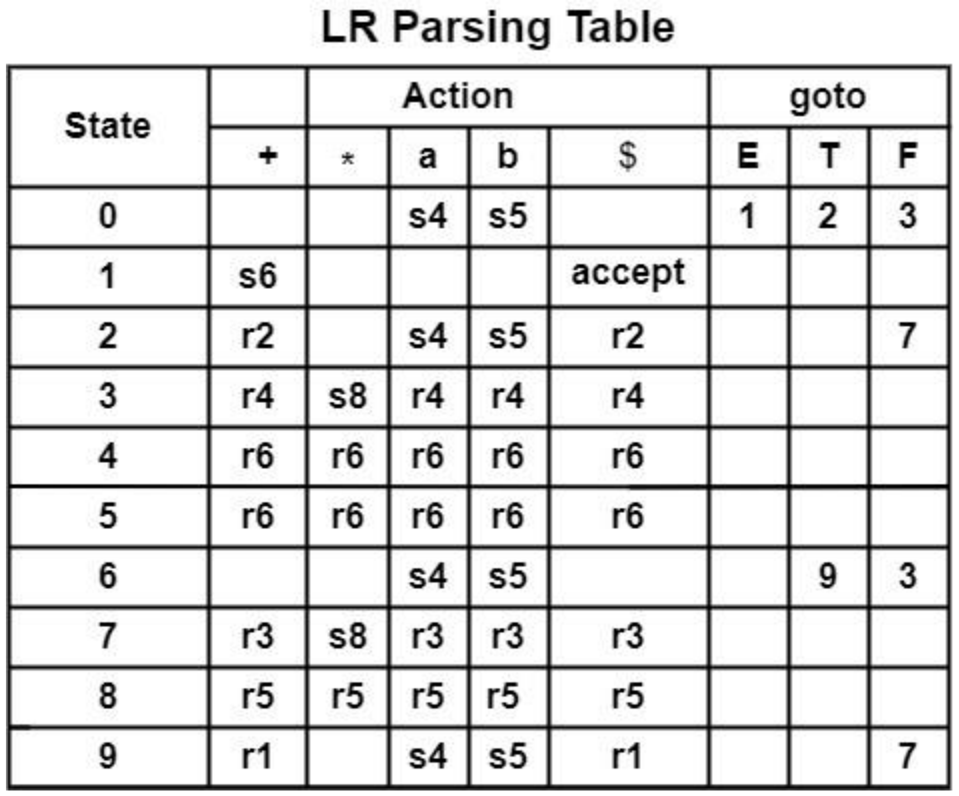
6) SLR PARSER

Step 2) The FOLLOW() for each non terminal

FOLLOW(E) = {+, $}

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

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



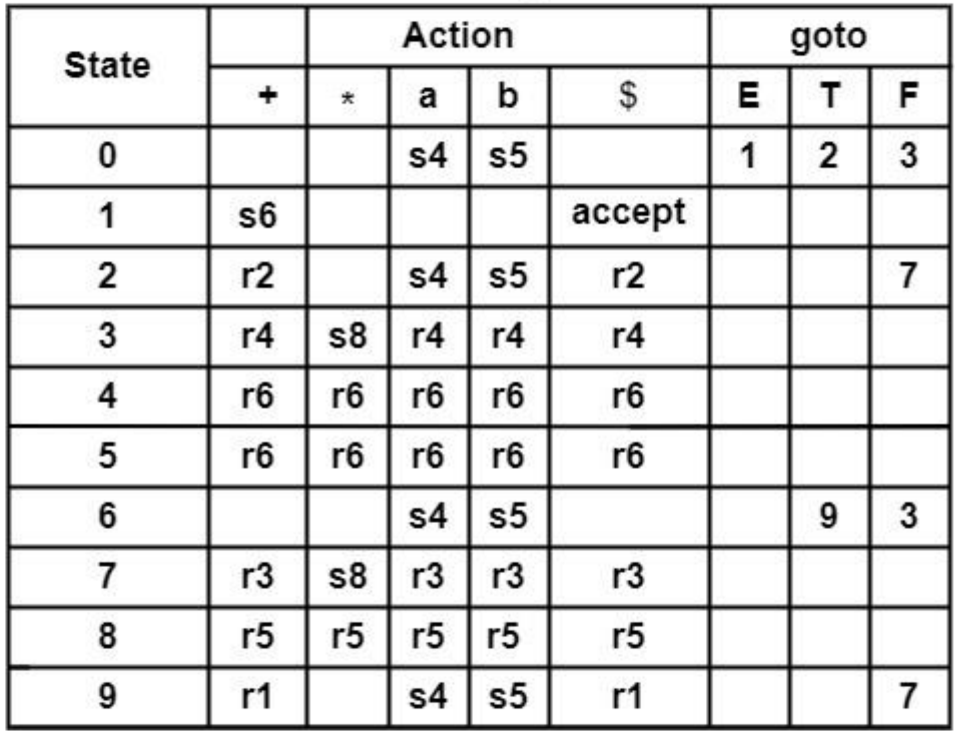
LR Parsing Table

State		Action				goto		
	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				accept			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r6	r6	r6	r6	r6			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

COMPILER DESIGN

6) SLR PARSER

LR Parsing Table



State	Action					goto		
	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				accept			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r6	r6	r6	r6	r6			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

Stack	Input String	Action
0	a * b + a $	Shift
0 a 4	* b + a $	Reduce by F → a.
0 F 3	* b + a $	Shift
0 F 3 * 8	b + a $	Reduce by F → F ∗
0 F 3	b + a $	Reduce by T → F
0 T 2	b + a $	Shift
0 T 2 b 5	+a $	Reduce by F → b
0 T 2 F 7	+a $	Reduce by T → TF
0 T 2	+a $	Reduce by E → T
0 E 1	+a $	Shift
0 E 1 + 6	a$	Shift
0 E 1 + 6 a 4	$	Reduce by F → a
0 E 1 + 6 F 3	$	Reduce by T → F
0 E 1 + 6 T 9	$	Reduce by E → E + T
0 E 1	$	Accept

SDPadiya
SSGMCE

7) CLR PARSER

Steps to construct CLR parser

1. Augment given grammar.
2. Number the Production.
3. Find LOOKAHEAD for each non terminals.
4. Find CLR(1) items by inserting '•' symbol at the first position for every production in G.
5. Construct Canonical Collection of CLR(1) items.
6. Construct CLR(1) Parsing Table (add reduce term for the lookahead symbols of left hand side non terminal).
7. Stack Implementation.
8. Draw Parse Tree.

NOTE

Add reduce only for the LOOKAHEAD symbols of left hand side non-terminals

7) CLR PARSER

Problem 1: Obtain the lookahead to prepare CLR(1) parsing table.

E → BB

B → cB| d

Solution: Step 1) The augmented grammar of the given grammar is:-

[0th production]	E′ → •**E, $**	$\text{Lookahead} = \text{FOLLOW}\ (\bullet E) = \text{First}(E') = \$$
[1st production]	E → •**BB, $**	$\text{Lookahead} = \text{FOLLOW}(\bullet \text{B}) = \text{First}(B) = \$$
[2nd production]	C → •**cC, c|d**	$\text{Lookahead} = \text{FOLLOW}(\bullet \text{c}) = \text{First}(C) = \boldsymbol{c} \mid \boldsymbol{d}$
[3rd production]	C → •**d, c|d**	$\text{Lookahead} = \text{FOLLOW}(\bullet \text{d}) = \text{First}(C) = \boldsymbol{c} \mid \boldsymbol{d}$

7) CLR PARSER

Problem 2: Obtain the lookahead to prepare CLR(1) parsing table.

S → AA

A → aA

A → b

Solution: Step 1) The augmented grammar of the given grammar is:-

[0th production]	S′ → **•S, $**	$\text{Lookahead} = \text{FOLLOW}\ (\bullet S) = \text{First}(S') = \$$
[1st production]	A → **•AA, $**	$\text{Lookahead} = \text{FOLLOW}(\bullet A) = \text{First}(A) = \$$
[2nd production]	A → **•aA, a|b**	$\text{Lookahead} = \text{FOLLOW}(\bullet \text{a}) = \text{First}(A) = \boldsymbol{a} \mid \boldsymbol{b}$
[3rd production]	A → **•b, a|b**	$\text{Lookahead} = \text{FOLLOW}(\bullet \text{b}) = \text{First}(A) = \boldsymbol{a} \mid \boldsymbol{b}$

7) CLR PARSER

Problem 3: Construct the CLR Parsing table for the following grammar. Also, Parse the input string cccdd.

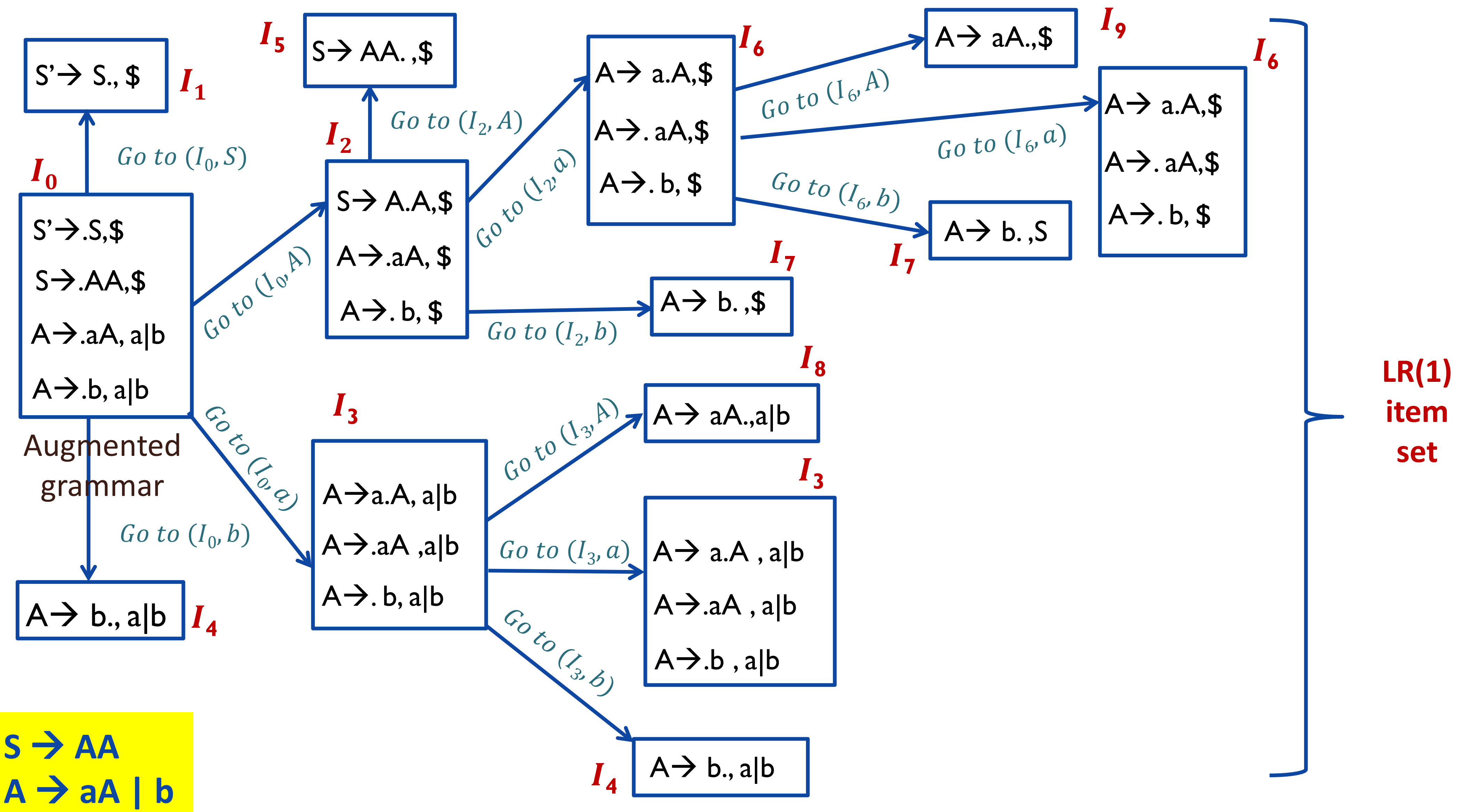
S → CC

C → cC| d

Solution: Step 1) The augmented grammar of the given grammar is:-

[0th production]	S′ → •**S**, $	$\text{Lookahead} = \text{First}(S') = \$$
[1st production]	S → •**CC**, $	$\text{Lookahead} = \text{First}(S') = \$$
[2nd production]	C → •**cC**, c|d	$\text{Lookahead} = \text{First}(C) = \boldsymbol{c} \mid \boldsymbol{d}$
[3rd production]	C → •**d**, c|d	$\text{Lookahead} = \text{First}(C) = \boldsymbol{c} \mid \boldsymbol{d}$

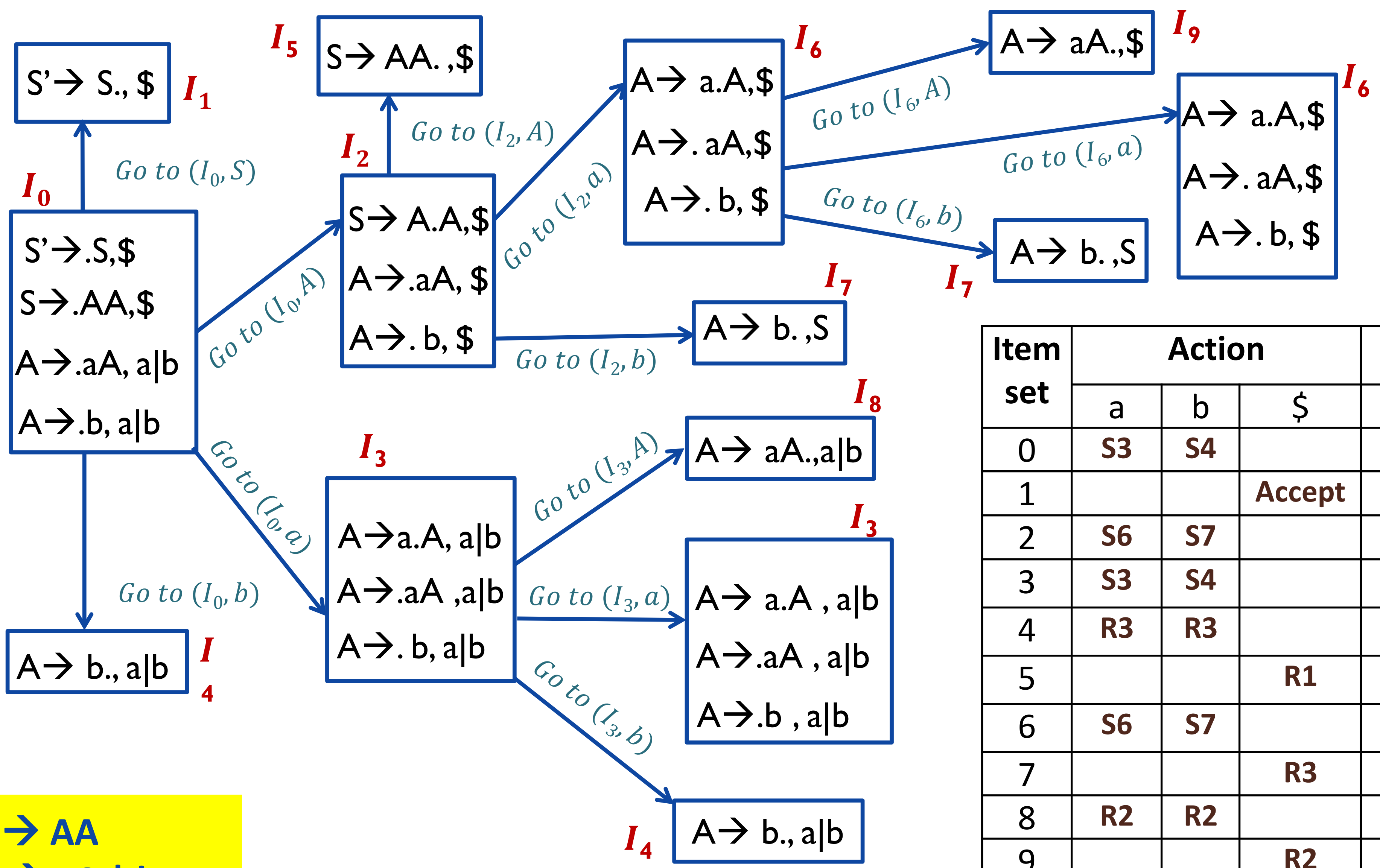
7) CLR PARSER



S → AA
A → aA | b

COMPILER DESIGN

7) CLR PARSER

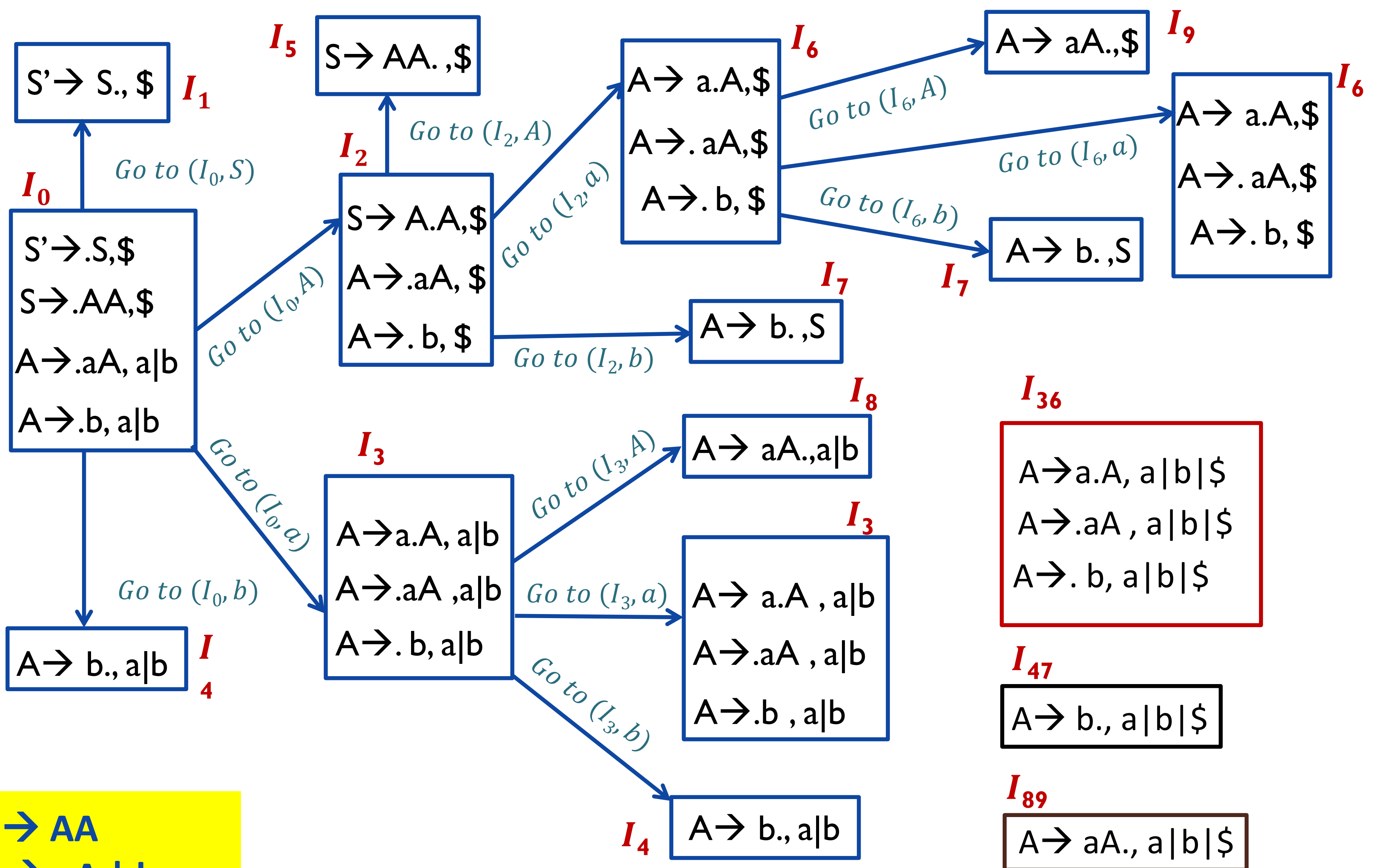


Item set	Action			Go to	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

S → AA
A → aA | b

SDPadiya
SSGMCE

8) LALR PARSER



S → AA
A → aA | b

8) LALR PARSER

I_{36}

A→a.A, a|b|$

A→.aA , a|b|$

A→. b, a|b|$

I_{47}

A→ b., a|b|$

I_{89}

A→ aA.,a|b|$

Item set	Action			Go to	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
~~6~~	~~S6~~	~~S7~~			~~9~~
~~7~~			~~R3~~		
8	R2	R2			
~~9~~			~~R2~~		

CLR Parsing Table

Item set	Action			Go to	
	a	b	$	S	A
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

LALR Parsing Table

Thank You

Prof. S. D. Padiya
padiyasagar@gmail.com
SSGMCE, Shegaon