LR(0) and SLR(1) Parsers

C.Naga Raju

B.Tech(CSE),M.Tech(CSE),PhD(CSE),MIEEE,MCSI,MISTE

Professor

Department of CSE

YSR Engineering College of YVU

Proddatur

Contents

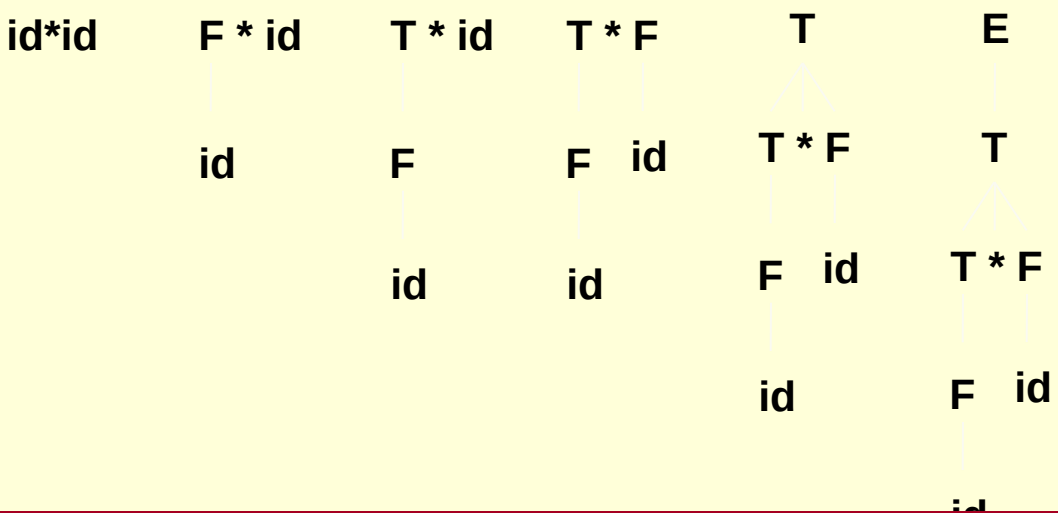
- Introduction to bottom up parsers
- LR(0) Parser
- Example problem
- Gate Questions and solutions
- SLR(1) Parser
- Example problem
- Gate Questions and solutions

Bottom up Parser

- Construction of parse tree for any given input string beginning at the bottom and working towards the root is called bottom up parser

For example the given input string : id*id

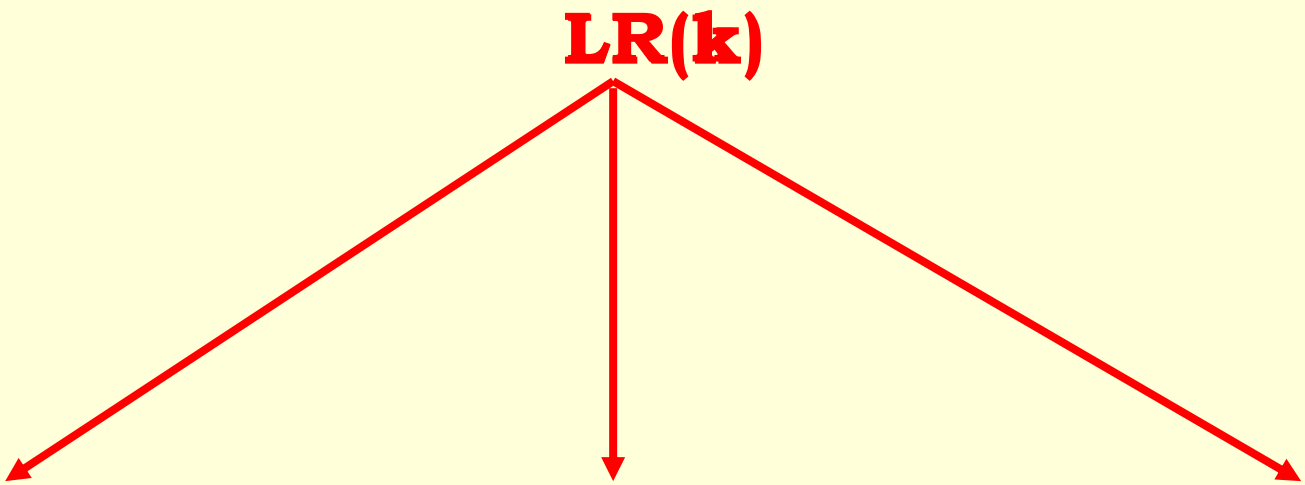
E -> E + T | T
T -> T * F | F
F -> (E) | id



Shift-Reduce Parsing

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, if a specific substring is matched then the body of a production is replaced by the Non Terminal at the head of the production A—>ac/b
- The key decisions during bottom-up parsing are about when to reduce and what production should 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=>T=>T*F=>T*id=>F*id=>id*id

LR Parsers



Left-to-Right scan

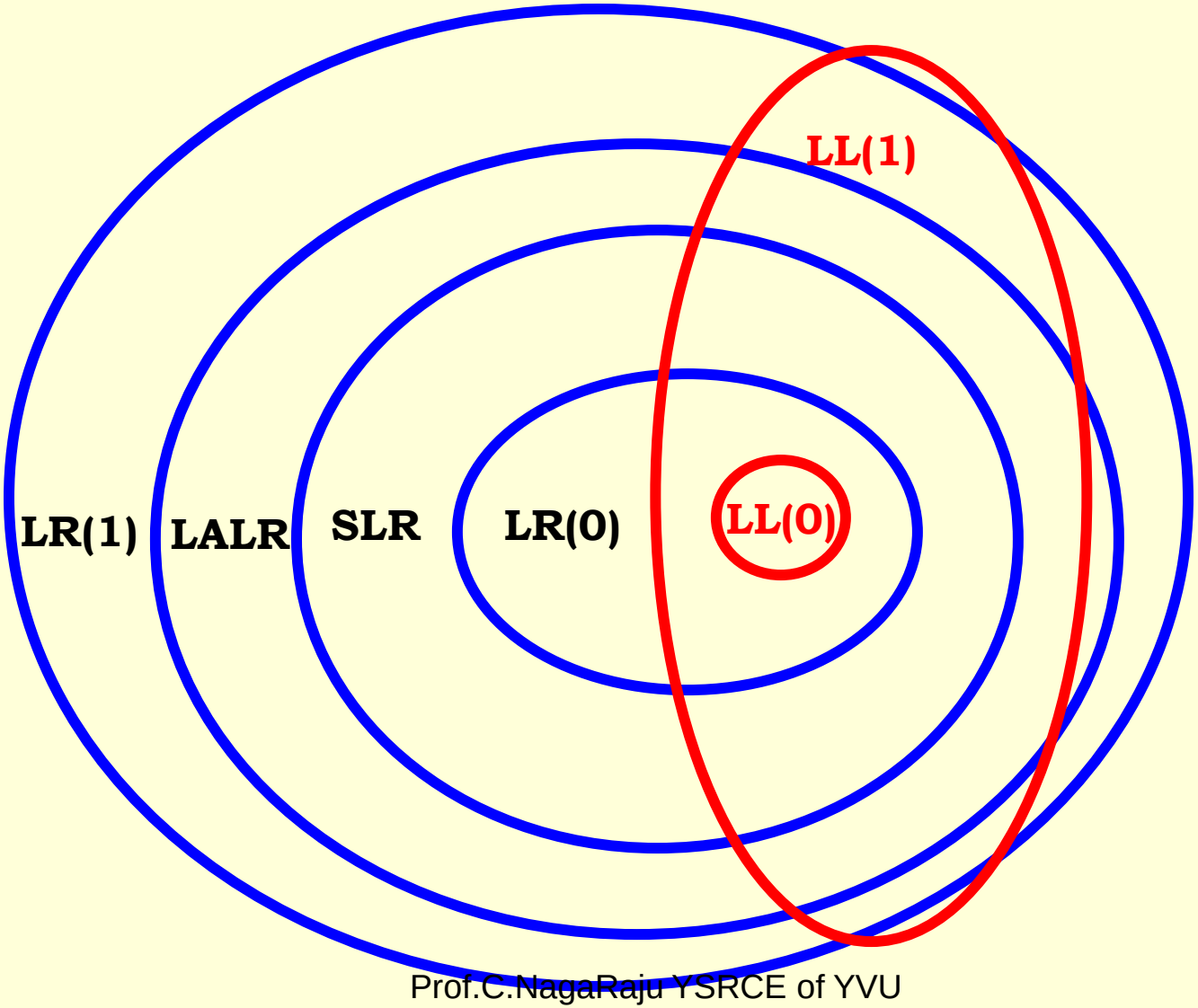
Rightmost Derivation In Reverse

Number Of Input Symbols Of Look Ahead

❖ Types of LR Parsers

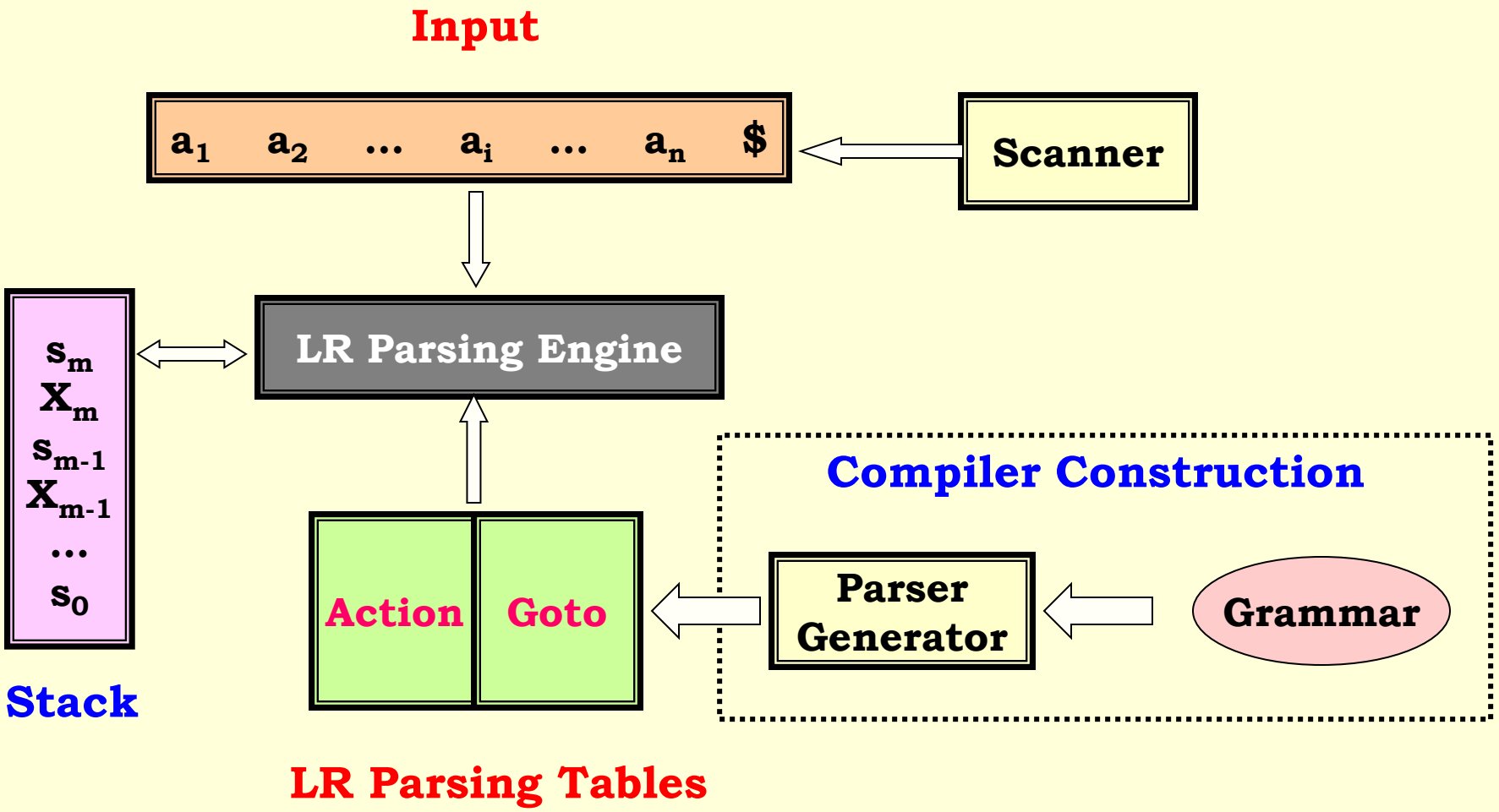
1. LR(0) Parser
2. Simple LR-Parser (SLR)
3. Canonical LR Parser (CLR)
4. LALR Parser.

Comparison of LL & LR Methods



- **Advantages of LR Parsers**
- LR parsers are constructed to recognize all Programming Languages
- The LR-parsing is Non-Backtracking Shift-Reduce Parser
- An LR parser can detect a syntactic errors
- It scans input string from left-to-right and use left most derivation in reverse

•The LR Parsing Algorithm



- The LR parser consists of 1) Input 2)Output 3)Stack 4) Driver Program 5) Parsing Table
- The Driver Program is same for all LR Parsers.
- Only the Parsing Table changes from one parser to the other.
- In CLR method the stack holds the states from the LR(0)automation and canonical LR and LALR methods are same

- The Driver Program uses the Stack to store a string

$$\mathbf{s_0X_1s_1X_2\ldots X_ms_m}$$

  - ✓ Where s_m is the Top of the Stack.
  - ✓ The S_k's are State Symbols
  - ✓ The X_i's are Grammar Symbols.
  - ✓ Together State and Grammar Symbols determine a Shift-reduce Parsing Decision.

- The Parsing Program reads characters from an Input Buffer one at a time
- The Current Input Symbols are used to index the parsing table and determine the shift-reduce parsing decision
- In an implementation, the grammar symbols need not appear on the stack

Parse Table

- The LR Shift-Reduce Parsers can be efficiently implemented by computing a table to guide the processing
- The Parsing Table consists of two parts:
  1. A Parsing **Action Function** and
  2. A **GOTO function**.

❖ The Action Table

❖ The Action Table specifies the actions of the parser (e.g., shift or reduce), for the given parse state and the next token

- ✓ Rows are State Names;
- ✓ Columns are Terminals

LR Driver Program

- The **LR driver Program** determines S_m, the state on top of the stack and a_i, the Current Input symbol.
- It then consults **Action[S_m, a_i]** which can take one of four values:
  - ✓ **Shift**
  - ✓ **Reduce**
  - ✓ **Accept**
  - ✓ **Error**

- If **Action[S_m, a_i]** = Shift S
  - ✓ Where S is a State, then the Parser pushes a_i and S on to the Stack.
- If **Action[S_m, a_i]** = Reduce A → β,
  - ✓ Then a_i and S_m are replaced by A
  - ✓ if S was the state appearing below a_i in the Stack, then **GOTO[S, A]** is consulted and the state pushed onto the stack.

- If **Action[S_m, a_i]** = Accept,
  - ✓ Parsing is completed
- If **Action[S_m, a_i]** = Error,
  - ✓ The Parser discovered an Error.

❖ GOTO Table

❖ The GOTO table specifies which state to put on top of the stack after a reduce

- ✓ Rows are State Names;
- ✓ Columns are Non-Terminals

- The **GOTO Table** is important to find out the next state after every reduction.

- The **GOTO Table** is indexed by a state of the parser and a Non Terminal (Grammar Symbol)

  ex : GOTO[S, A]

- The **GOTO Table** simply indicates what the next state of the parser if it has recognized a certain Non Terminal.

- **Right Sentential Form** is a Sentential Form in a Rightmost Derivation

  Example: (S)S , ((S)S)

- **Viable Prefix** is a sequence of symbols on the parsing stack

  Example:

  - ✓ (S)S, (S), (S, (,
  - ✓ ((S)S, ((S), ((S , ((, (

LR(0) Parser

- The LR Parser is a Shift-reduce Parser that makes use of a Deterministic Finite Automata, recognizing the Set Of All Viable Prefixes by reading the stack from Bottom To Top.
- if a Finite-State Machine that recognizes viable prefixes of the right sentential forms is constructed, it can be used to guide the handle selection in the Shift-reduce Parser.

- **Handle:** Handle is a substring that matches the body of a production
- Handle is a Right Sentential Form + position where reduction can be performed + production used for reduction

Example

(S) S. With $S \rightarrow \epsilon$

(S) .S With $S \rightarrow S$

((S) S .) With $S \rightarrow (S) S$

- **Handle pruning :** Handle pruning specifies that the reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Augmented Grammar

- If G is a Grammar with Start Symbol S, the Augmented Grammar G' is G with a New Start Symbol S`, and New Production S` →S$.

- The Purpose of the Augmented Grammar is to indicate to the parser when it should stop parsing and announce acceptance of the input

LR(0) Items

An LR(0) Item of a Grammar G is a Production of G with a Dot (•) at some position of the right side.

.Production A → XYZ yields the Four items:

1. A→•XYZ We hope to see a string derivable from XYZ next on the input.
2. A→X•YZ We have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ next on the input.

3. $A \rightarrow XY \bullet Z$

4. $A \rightarrow XYZ \bullet$

- The production $A \rightarrow \varepsilon$ generates only one item, $A \rightarrow \bullet$.
- Each of this item is a Viable prefixes
- **Closure Item :** An Item created by the closure operation on a state.
- **Complete Item** : An Item where the Item Dot is at the end of the RHS.

LR(0) Example

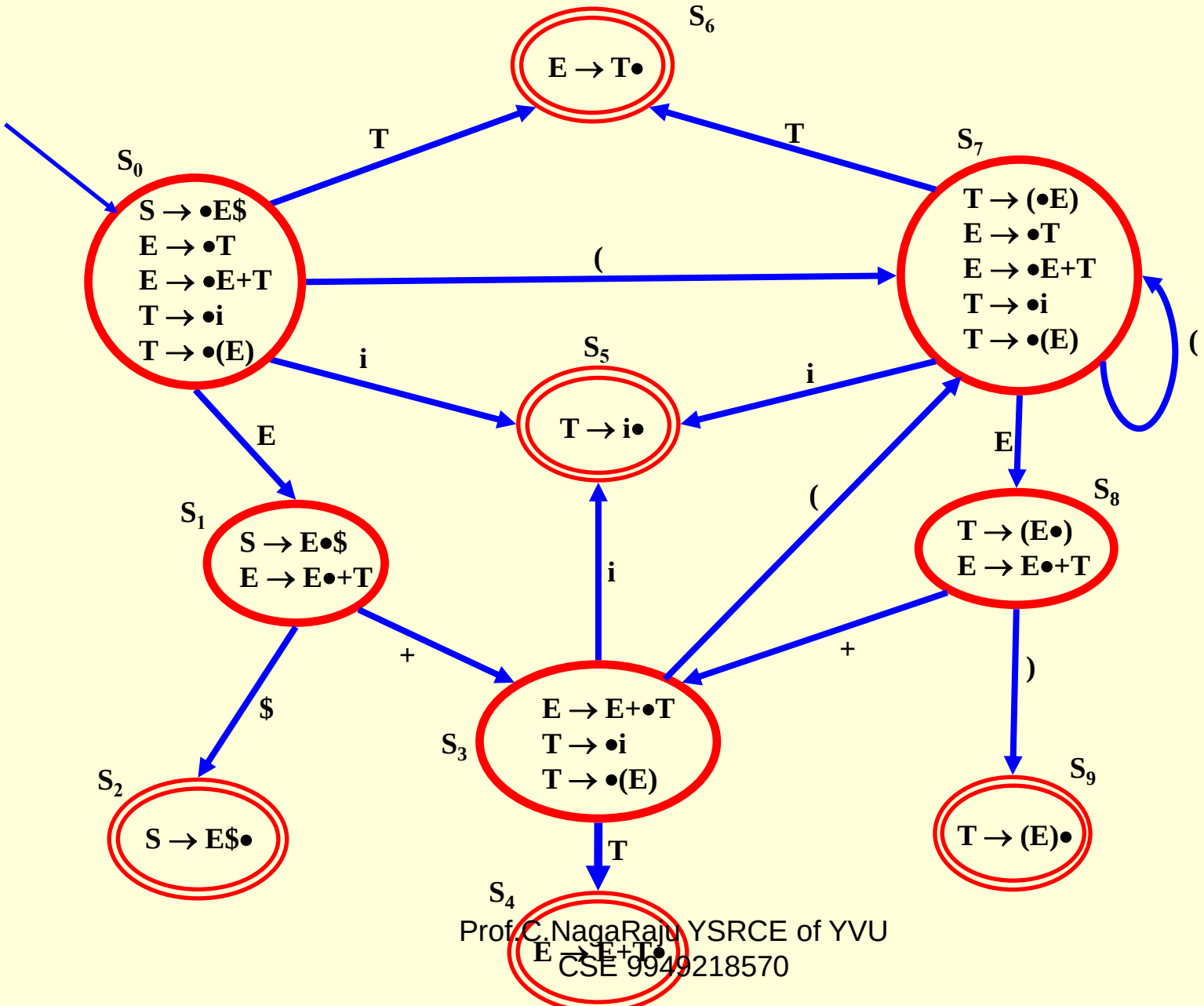
Context Free Grammar:

S → E **rule1: (S→S')**
E → T
E → E + T
T → i
T → (E)

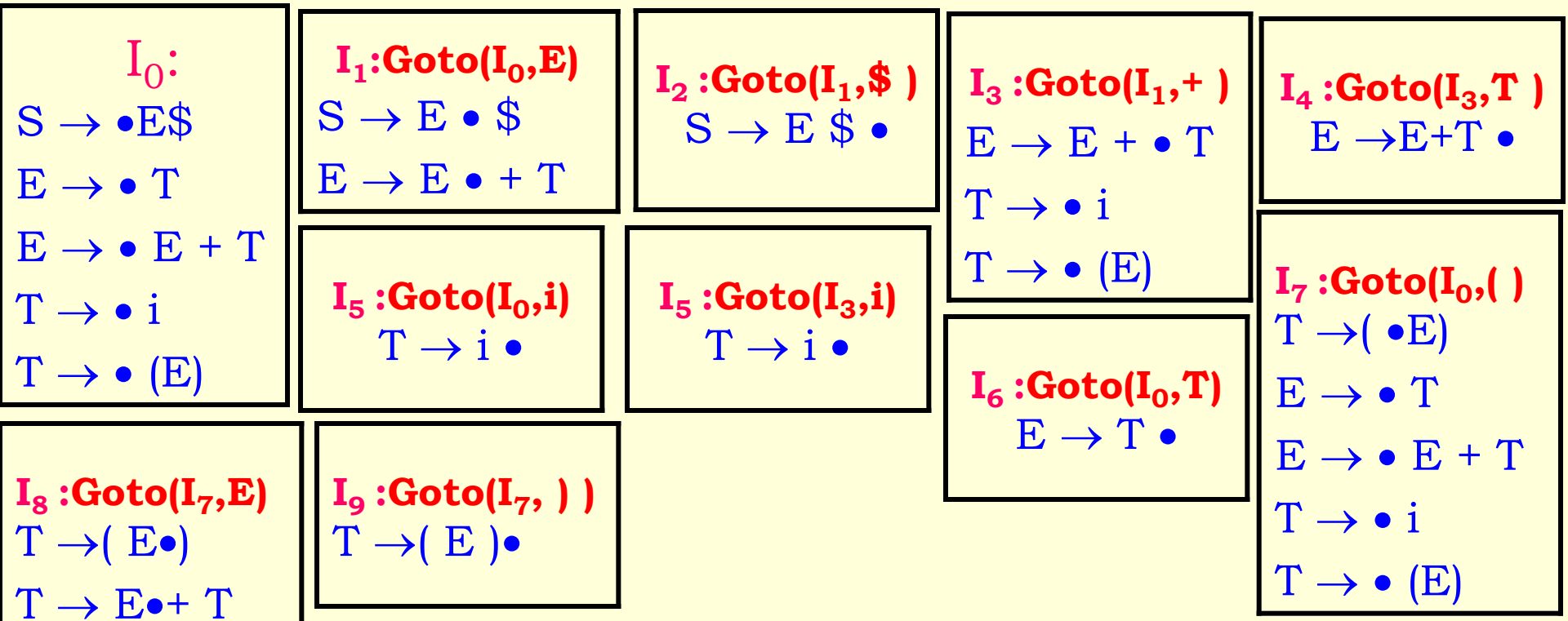
Augumented Grammar

S → •E
E → •T
E → •E+T
T → •i
T → •(E)

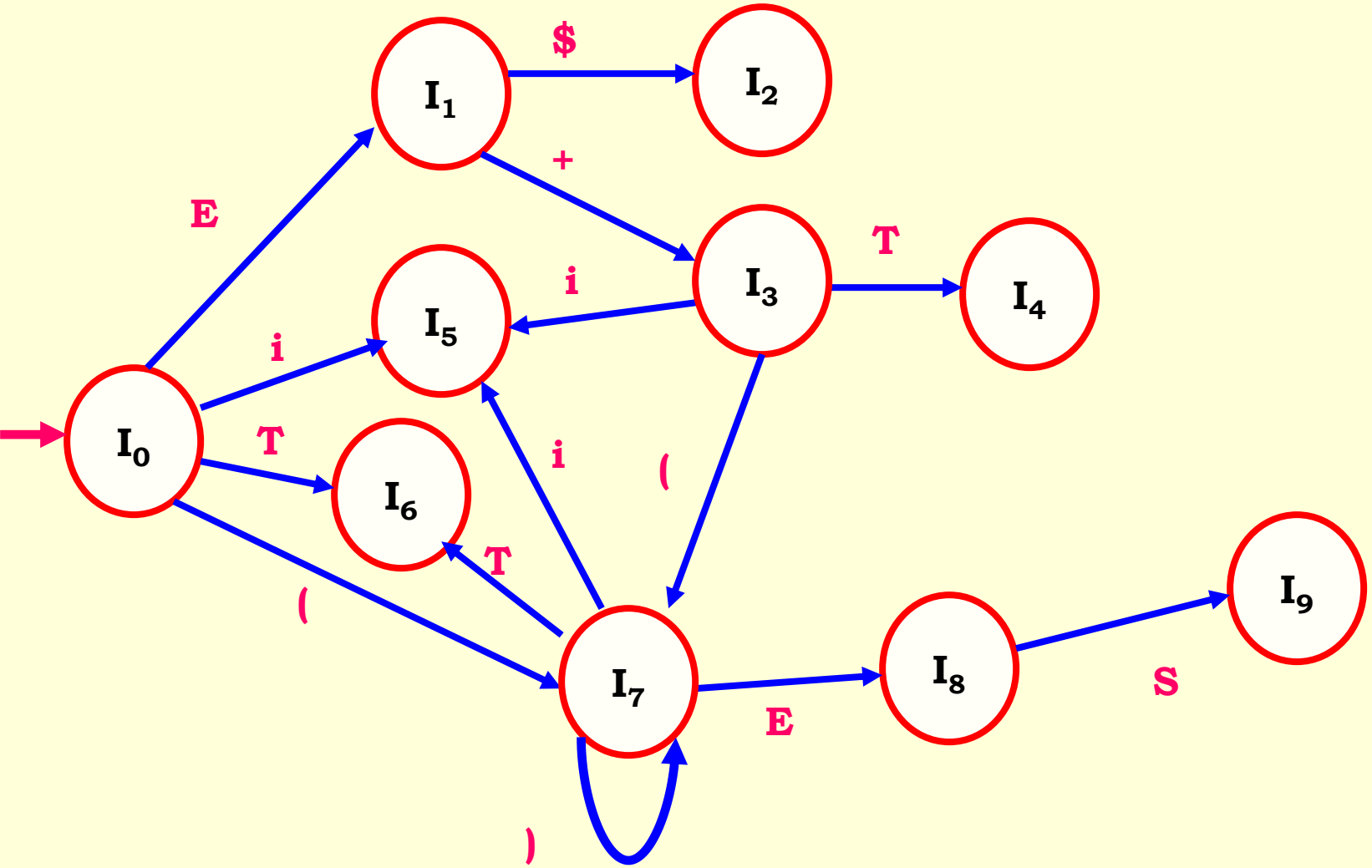
Construction of LR(0) Closure Items



Construction of LR(0) Items



Construction of DFA for LR(0) Items



Construction of LR(0) Parsing Table

States	Input							
	Action Part (Terminals)					Goto Part (Non-Terminals)		
	i	+	(	)	$	E	T	
0	5		7			1	6	Shift
1		3			2			Shift
2	S→E$							Reduce
3	5		7				4	Shift
4	E→E+T							Reduce
5	T→i							Reduce
6	E→T							Reduce
7	5		7			8	6	Shift
8		3		9				Shift
9	T→(E)							Reduce

String Acceptance

States	Input							
	Action Part (Terminals)					Goto Part (Non-Terminals)		
	i	+	(	)	$	E	T	
0	5		7			1	6	Shift
1		3			2			Shift
2	S→E$							Reduce
3	5		7				4	Shift
4	E→E+T							Reduce
5	T→i							Reduce
6	E→T							Reduce
7	5		7			8	6	Shift
8		3		9				Shift
9	T→(E)							Reduce

Stack	Input	Action
S_0	i + (i + i) $	Shift
S_0 i S_5	+ (i + i) $	Reduce by T→i
S_0 T S_6	+ (i + i) $	Reduce by E→T
S_0 E S_1	+ (i + i) $	Shift
S_0 E S_1 + S_3	(i + i) $	Shift

String Acceptance

States	Input							
	Action Part (Terminals)					Goto Part (Non-Terminals)		
	i	+	(	)	$	E	T	
0	5		7			1	6	Shift
1		3			2			Shift
2	S→E$							Reduce
3	5		7				4	Shift
4	E→E+T							Reduce
5	T→i							Reduce
6	E→T							Reduce
7	5		7			8	6	Shift
8		3		9				Shift
9	T→(E)							Reduce

Stack	Input	Action
S_0 E S_1 + S_3	(i + i) $	shift
S_0 E S_1 + S_3 (S_7	i + i) $	shift
S_0 E S_1 + S_3 (S_7iS_5	+ i)$	reduce by T→i
S_0 E S_1 + S_3 (S_7TS_6	+ i)$	reduce by E→T
S_0 E S_1 + S_3 (S_7ES_8	+ i)$	shift

String Acceptance

States	Input							
	Action Part (Terminals)					Goto Part (Non-Terminals)		
	i	+	(	)	$	E	T	
0	5		7			1	6	Shift
1		3			2			Shift
2	S→E$							Reduce
3	5		7				4	Shift
4	E→E+T							Reduce
5	T→i							Reduce
6	E→T							Reduce
7	5		7			8	6	Shift
8		3		9				Shift
9	T→(E)							Reduce

Stack	Input	Action
S_0 E S_1 + S_3 (S_7ES_8	+ i) $	shift
S_0 E S_1 + S_3 (S_7ES_8+S_3	i) $	shift
S_0 E S_1 + S_3 (S_7ES_8+S_3iS_5	) $	reduce by T→i
S_0 E S_1 + S_3 (S_7ES_8+S_3TS_4	) $	reduce by E→E+T
S_0 E S_1 + S_3 (S_7ES_8	) $	shift

String Acceptance

States	Input							
	Action Part (Terminals)					Goto Part (Non-Terminals)		
	i	+	(	)	$	E	T	
0	5		7			1	6	Shift
1		3			2			Shift
2	S→E$							Reduce
3	5		7				4	Shift
4	E→E+T							Reduce
5	T→i							Reduce
6	E→T							Reduce
7	5		7			8	6	Shift
8		3		9				Shift
9	T→(E)							Reduce

Stack	Input	Action
S_0 E S_1 + S_3 (S_7ES_8	)$	shift
S_0 E S_1 + S_3 (S_7ES_8)S_9	$	reduce by T→(E)
S_0 E S_1 + S_3TS_4	$	reduce by E→E+T
S_0 E S_1	$	shift
S_0 E $S_1$$S_2$		reduce by S→E$
S_0 S		accept

Draw backs of LR(0) Parser

- LR(0) is the Simplest Technique in the LR family.
- LR(0) Parsers are too weak to be of practical use
- LR(0) accepts only small class of LR(0) grammar because if conflicts occurs.
- The Fundamental Limitation of LR(0) is that no look ahead tokens are used.
- LR(0) Parsing is the weakest and it is not used much in practice because of its limitations.

GATE QUESTIONS AND SOLUTIONS

Question 1

Consider the grammar

$E \rightarrow E + n \mid E \times n \mid n$

For a sentence $n + n \times n$, the handles in the right-sentential form of the reductions are ? (GATE 2005)

A. n, E + n and E + n × n

B. n, E + n and E + E × n

C. n, n + n and n + n × n

D. n, E + n and E × n

Explanation

- E → E * n {Applying E → E * n }
- E→ E + n * n {Applying E → E + n }
- E→ n + n * n {Applying E → n } Hence, the handles in right sentential form is n, E + n and E × n.
- Hence **Option D is the right choice**

Question 2

A bottom-up parser generates:

A. Left-most derivation in reverse
B. Right-most derivation in reverse
C. Left-most derivation
D. Right-most derivation

Explanation

- A bottom-up parser generates right-most derivation in reverse
- **Option (B) is correct.**

Question 3

Consider the following statements related to compiler construction :

I. Lexical Analysis is specified by context-free grammars and implemented by pushdown automata.

II. Syntax Analysis is specified by regular expressions and implemented by finite-state machine.

Which of the above statement(s) is/are correct ?

A. Only I
B. Only II
C. Both I and II
D. Neither I nor II

Explanation

- Both statements are wrong for detailed information on lexical analysis and syntax analysis
- **option (D) is correct.**

Question 4

Which of these is true about LR parsing?

A. Is most general non-backtracking shift-reduce parsing
B. It is still efficient
C. Both a and b
D. None of the mentioned

Explanation

- LR parsers are a type of bottom-up parsers that efficiently handle deterministic context-free languages in guaranteed linear time.
- **option (C) is correct.**

Question 5

Which of the following is incorrect for the actions of A LR-Parser

I) shift s

ii) reduce A->ß

iii) Accept

iv) reject?

A. Only I)

B. I) and ii)

C. I), ii) and iii)

D. I), ii) , iii) and iv)

Explanation

- Only reject out of the following is a correct LR parser action
- **Option C is correct**

Question 6

If a state does not know whether it will make a shift operation or reduction for a terminal is called

A. Shift/reduce conflict
B. Reduce /shift conflict
C. Shift conflict
D. Reduce conflict

Explanation

- As the name suggests that the conflict is between shift and reduce hence it is called shift reduce conflict
- **Option A is correct**

Question 7

When there is a reduce/reduce conflict?

A. If a state does not know whether it will make a shift operation using the production rule i or j for a terminal.
B. If a state does not know whether it will make a shift or reduction operation using the production rule i or j for a terminal.
C. If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal.
D. None of the mentioned

Explanation

- It occurs when If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal.
- **Option C is correct**

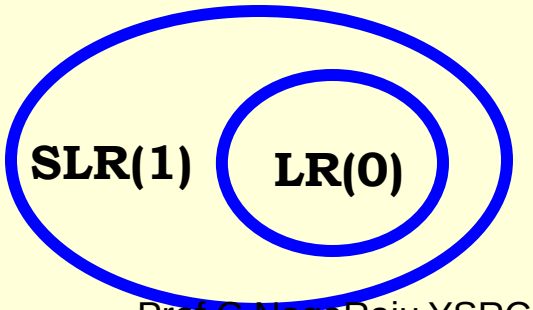
SLR(1) Parser

- We will find that it allows for a much larger class of grammars to be parsed.
- SLR(1) Parser is used for accepting the certain grammar which is not accepted by LR(0) parser
- The letters "SLR" stand for "Simple", "Left" and "Right".
  - ✓ "Left" indicates that the input is read from left to right and
  - ✓ The "Right" indicates that a right-derivation is built.

- SLR(1) Parser stands for Simple LR(1).
- SLR(1) parsers use the same LR(0) Configurating Sets and have the Same Table Structure and Parser Operation, So everything you've already learned about LR(0) applies here.
- The difference in SLR(1) Parser with LR(0) Parser comes in Assigning Table Actions,

- SLR(1) Parsers are going to use one token of lookahead to eliminate the conflicts.
- In LR(0) parsing, Reduce Actions that cause the problem
- The Simple Improvement that SLR(1) makes on the basic LR(0) parser is to reduce only if the next input token is a member of the **Follow Set** of the non-terminal being reduced.

- SLR(1) parser will perform a reduce action for configuration **B→a•** if the lookahead symbol is in the set **Follow(B)**
- A Grammar is an SLR(1) grammar if there is no conflict in the grammar.
- Clearly SLR(1) is a proper superset of LR(0)



Example: SLR(1) Parser

❖ Construct the **SLR(1) Parser** for the Following **Grammar**

Context Free Grammar:

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

Step 1: Define a Augmented Grammar

Context Free Grammar:

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Augmented Grammar:

$E' \rightarrow E\#$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

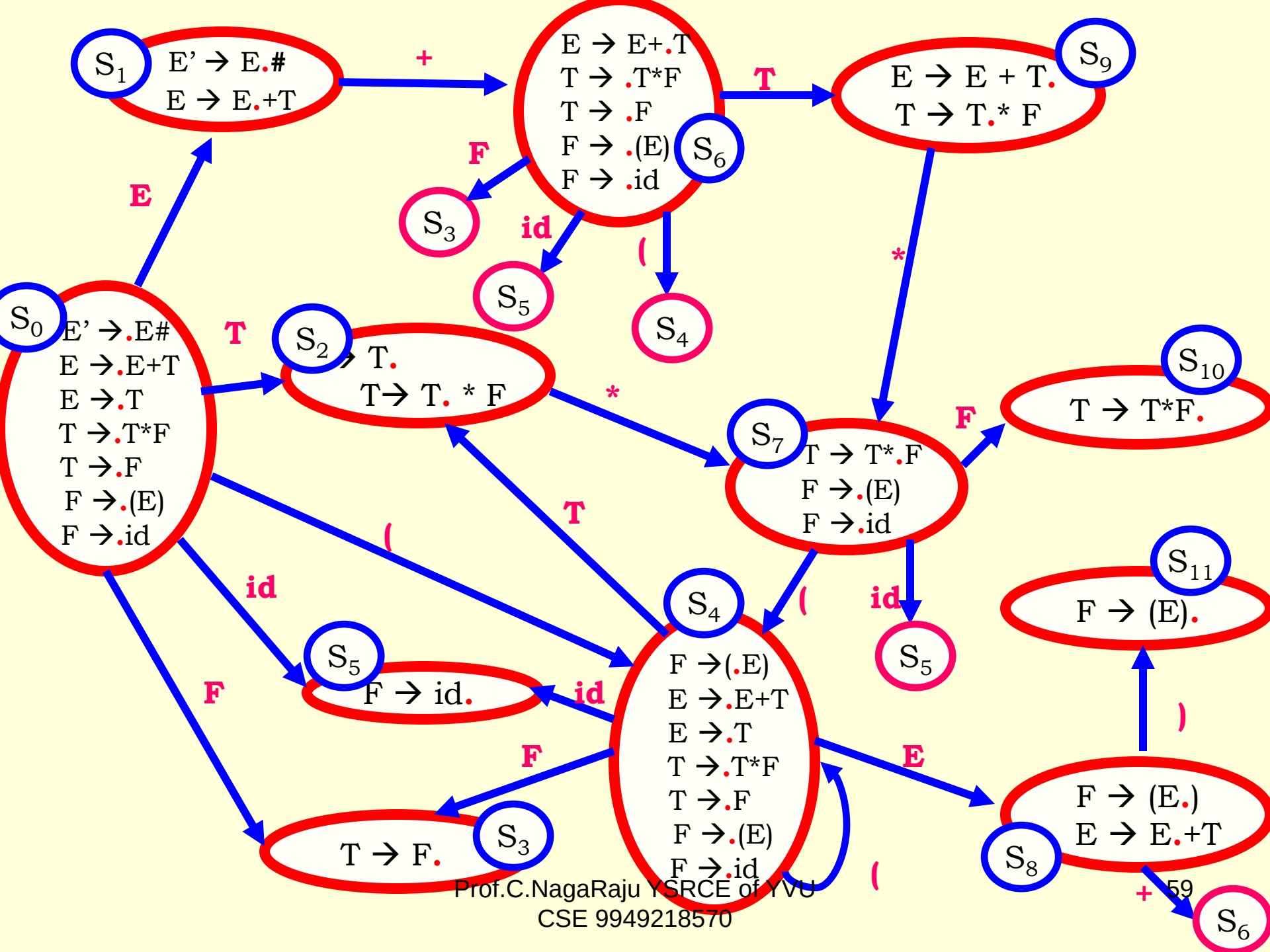
$T \rightarrow F$

$F \rightarrow (E)$

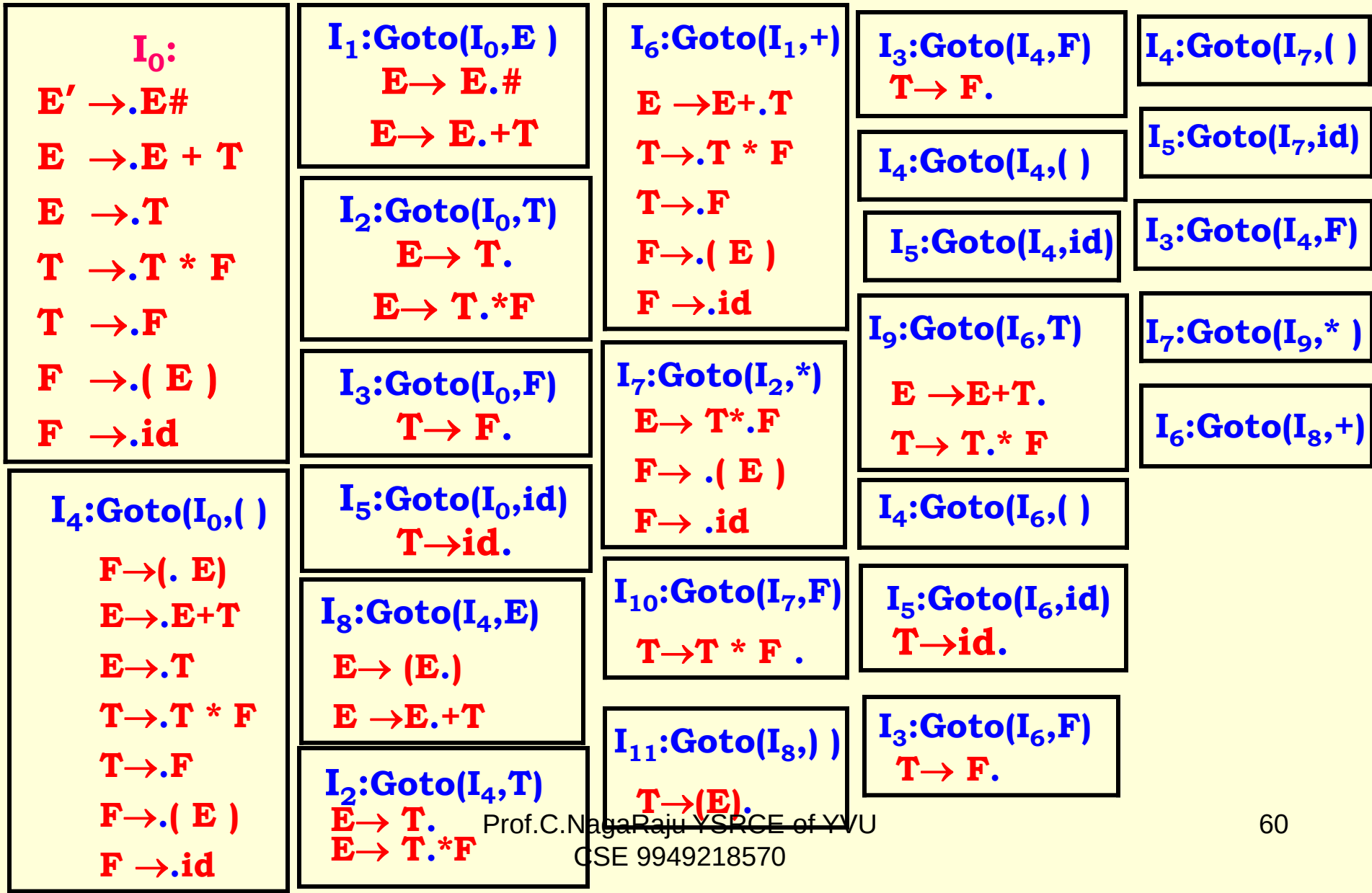
$F \rightarrow id$

Step2 : Constructing SLR(1) Automaton

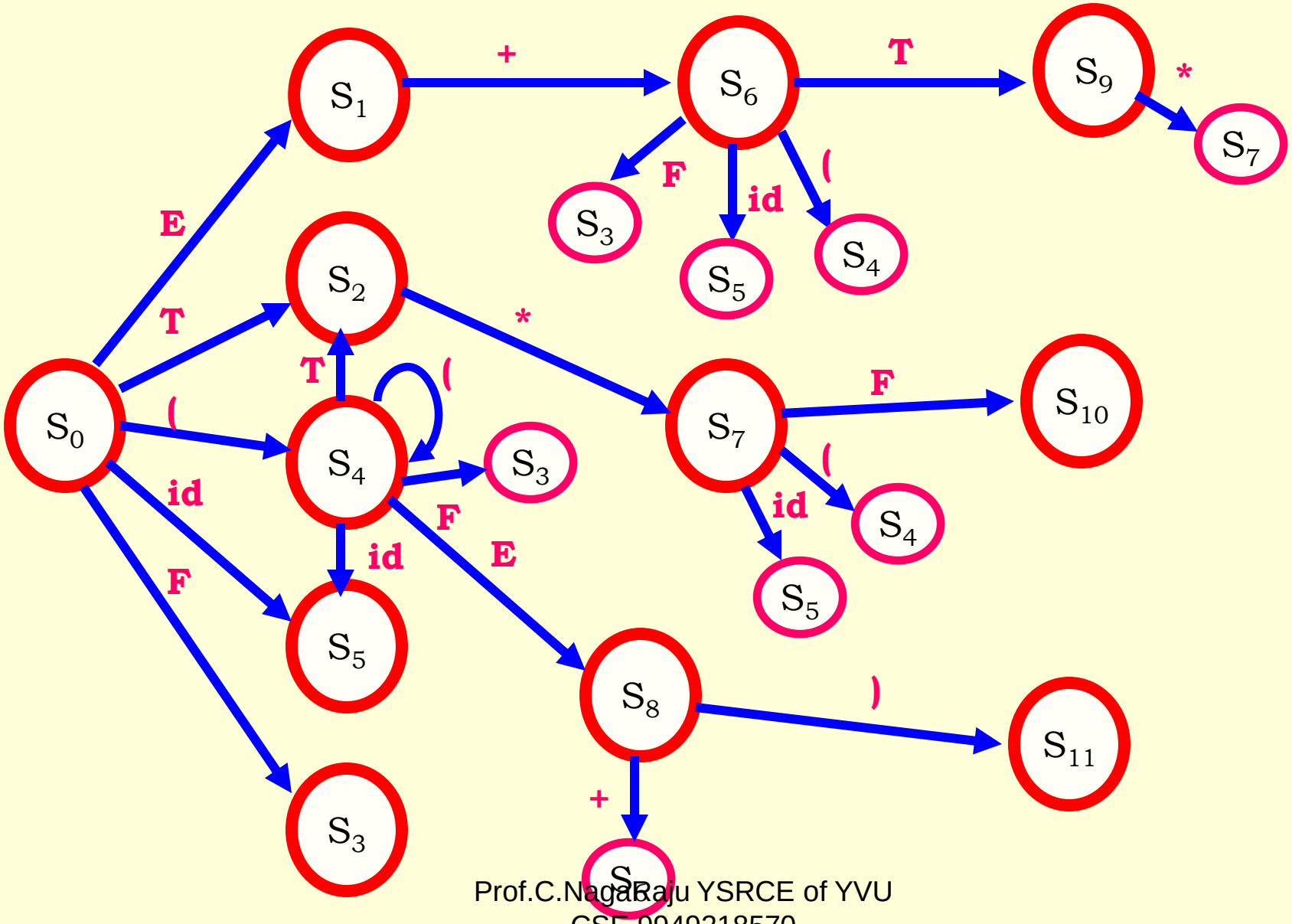
Context Free Grammar	**Adding the SLR(1) Item**
$E' \rightarrow E\#$	$E' \rightarrow \bullet E\#$
$E \rightarrow E + T$	$E \rightarrow \bullet E + T$
$E \rightarrow T$	$E \rightarrow \bullet T$
$T \rightarrow T * F$	$T \rightarrow \bullet T * F$
$T \rightarrow F$	$T \rightarrow \bullet F$
$F \rightarrow (E)$	$F \rightarrow \bullet (E)$
$F \rightarrow id$	$F \rightarrow \bullet id$


S1
E' → E.#
E → E.+T
+
E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id
S6
T
S9
E → E + T.
T → T.* F
F
S3
id
S5
(
S4
*
E
S0
E' →.E#
E →.E+T
E →.T
T →.T*F
T →.F
F →.(E)
F →.id
T
S2
T.
T→ T. * F
*
S7
T → T*.F
F →.(E)
F →.id
F
S10
T → T*F.
T
(
id
S5
F → id.
id
S4
F →(.E)
E →.E+T
E →.T
T →.T*F
T →.F
F →.(E)
F →.id
(
id
S5
S11
F → (E).
)
F
F
E
T → F.
S3
S8
F → (E.)
E → E.+T
(
+
S6
Prof.C.NagaRaju YSRCE of YVU
CSE 9949218570
59

Constructing the SLR(1) Items



Construction of DFA for SLR(1) Items



Construction of Follow Function

E′ →.E#

E →.E + T (r1)

E →.T (r2)

T →.T * F (r3)

T →.F (r4)

F →.(E) (r5)

F →.id (r6)

Follow (E) = { # , + ,) }

Follow (T) = { * , # , + ,) }

Follow (F) = { * , # , + ,) }

Constructing the SLR(1) Parsing Table

States	Input								
	Action Part						Goto Part		
	id	+	*	(	)	$	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			

Input Acceptance

String Acceptance id * id + id$

STACK	INPUT	ACTION
(1) 0	id * id + id $	shift
(2) 0 id 5	* id + id $	

STATE	action					goto		
	id	+	*	(	)	$	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			

In S_0 with input symbol id, shift the symbol onto the stack and enter state S_5

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(2)	0 id 5	* id + id $	reduce by $F \rightarrow$ id
(3)	0	* id + id $	

S_5 *with input symbol* *

Reduce using grammar production6

(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	action						goto		
	id	+	*	(	)	$	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			

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(2)	0 id 5	* id + id $	reduce by $F \rightarrow id$
(3)	0 F 3	* id + id $	reduce by $T \rightarrow F$

S_5 with input symbol *

Reduce using grammar production6

(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	action						goto		
	id	+	*	(	)	$	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			

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(3)	0 F 3	* id + id $	reduce by $T \rightarrow F$
(4)	0 T 2	* id + id $	shift

Reduction exposes S_0 and goto of S_0 gives next state for the leading non-terminal, T. The next state is S_2

STATE	action						goto		
	id	+	*	(	)	$	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			

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

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(4)	0 T 2	* id + id $	shift
(5)	0 T 2 * 7	id + id $	shift

STATE	action						goto		
	id	+	*	(	)	$	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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \text{id}$

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(5)	0 *T* 2 * 7	id + id $	shift
(6)	0 *T* 2 * 7 id 5	+ id $	reduce by *F* → id

STATE	action					goto		
	id	+	*	(	)	$	*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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

STACK	INPUT	ACTION
(6) 0 *T* 2 * 7 id 5	+ id $	reduce by *F* → id

STATE	action						goto		
	id	+	*	(	)	$	*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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

STACK	INPUT	ACTION
(6) 0 T 2 * 7	+ id $	reduce by F → id
(7) 0 T 2 * 7 F 10		

Reduction exposes S_7 and goto of S_7 gives next state for the leading non-terminal, F. The next state is S_{10}

STATE	action						goto		
	id	+	*	(	)	$	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			

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

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(7)	0 *T* 2 * 7 *F* 10	+ id $	Reduce T → T*F
(8)	0 *T* 2	+ id $	

(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	action						goto		
	id	+	*	(	)	$	*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			

String Acceptance id * id + id$

STACK	INPUT	ACTION
(8) 0 T 2	+ id $	Reduce E → T

STATE	action						goto		
	id	+	*	(	)	$	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			

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

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(8)	0 *T*	+ id $	Reduce E → T
(9)	0 *E* 1	+ id $	

STATE	action						goto		
	id	+	*	(	)	$	*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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(9)	0 E 1	+ id $	Shift
(10)	0 E 1 + 6	id $	

STATE	action						goto		
	id	+	*	(	)	$	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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

STACK	INPUT	ACTION
(10) 0 E 1 + 6	id $	Shift
(11) 0 E 1 + 6 id 5	$	

STATE	action						goto		
	id	+	*	(	)	$	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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(11)	0 E 1 + 6 id 5	$	Reduce F → id
(12)	0 E 1 + 6 F	$	

(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	action						goto		
	id	+	*	(	)	$	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			

String Acceptance id * id + id$

STACK	INPUT	ACTION
(12) 0 E 1 + 6 F 3	$	
(12) 0 E 1 + 6 F 3	$	

(1)	$E \rightarrow E + T$
(2)	$E \rightarrow T$
(3)	$T \rightarrow T * F$
(4)	$T \rightarrow F$
(5)	$F \rightarrow (E)$
(6)	$F \rightarrow \mathbf{id}$

STATE	action						goto		
	id	+	*	(	)	$	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			

String Acceptance id * id + id$

STACK	INPUT	ACTION
(12) 0 E 1 + 6 F 3	$	Reduce T → F
(13) 0 E 1 + 6 T	$	

STATE	action						goto		
	id	+	*	(	)	$	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			

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

String Acceptance id * id + id$

STACK	INPUT	ACTION
(13) 0 E 1 + 6 T 9	$	

STATE	action						goto		
	id	+	*	(	)	$	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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(13)	0 E 1 + 6 T 9	$	Reduce E→E+T
(14)	0 E 1	$	

STATE	action						goto		
	id	+	*	(	)	$	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			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

String Acceptance id * id + id$

STACK	INPUT	ACTION
(13) 0 E 1	$	Accept

(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	action						goto		
	id	+	*	(	)	$	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			

String Acceptance id * id + id$

	STACK	INPUT	ACTION
(1)	0	**id** * **id** + **id**$	shift
(2)	0 **id** 5	* **id** + **id**$	reduced by $F \rightarrow$ **id**
(3)	0 F 5	* **id** + **id**$	reduced by $T \rightarrow F$
(4)	0 T 2	* **id** + **id**$	shift
(5)	0 T 2 * 7	**id** + **id**$	shift
(6)	0 T 2 * 7 **id** 5	+ **id**$	reduced by $F \rightarrow$ **id**
(7)	0 T 2 * 7 F 10	+ **id**$	reduced by $T \rightarrow T*F$
(8)	0 T 2	+ **id**$	reduced by $E \rightarrow T$
(9)	0 E 1	+ **id**$	shift
(10)	0 E 1 + 6	**id**$	shift
(11)	0 E 1 + 6 **id** 5	$	reduced by $F \rightarrow$ **id**
(12)	0 E 1 + 6 F 3	$	reduced by $T \rightarrow F$
(13)	0 E 1 + 6 T 9	$	$E \rightarrow E + T$
(14)	0 E 1	$	accept

Observation

- the Shift/Reduce Conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input = has seen a string reducible to L.

- That is "R=" cannot be a part of any Right Sentential Form. So when "L" appears on the top of stack and "=" is the current character of the input buffer , we can not reduce "L" into "R".

- Every SLR Grammar is Unambiguous, but Every Unambiguous Grammar is not a SLR grammar.

- If the SLR parsing table of a grammar G has a Conflict, we say that Grammar is not SLR Grammar.

Drawbacks

- SLR(1) parsers cannot parse some LR grammars.
- The Main Problem of SLR(1) Parser is that Lookahead Information is added to LR(0) parser at the end of construction based on FOLLOW sets
- In SLR(1) parsing, we reduce **A→α** for ANY lookahead **a Є FOLLOW(A),** which is too general such that sometimes a reduction cannot occur for some **a Є FOLLOW(A)**

Drawbacks

- if the Grammar contains Reduce/Reduce Conflict then, we say that Grammar is not SLR(1) Grammar.

GATE QUESTIONS AND SOLUTIONS

Question 1

- Consider the following two statements: P: Every regular grammar is LL(1) Q: Every regular set has a LR(1) grammar Which of the following is TRUE? (GATE 2007)

A. Both P and Q are true
B. P is true and Q is false
C. P is false and Q is true
D. Both P and Q are false

Explanation

- A regular grammar can also be ambiguous also For example, consider the following grammar, S → aA/a A → aA/ε In above grammar, string 'a' has two leftmost derivations.
- (1) S → aA (2) S → a S->a (using A->ε) And LL(1) parses only unambiguous grammar, so statement P is False.
- Statement Q is true is for every regular set, we can have a regular grammar which is unambiguous so it can be parse by LR parser.
- So **option C** is correct choice

Question 2

A canonical set of items is given below

$S \rightarrow L. > R$

$Q \rightarrow R.$

On input symbol < the set has? (GATE 2014)

A. a shift-reduce conflict and a reduce-reduce conflict.
B. a shift-reduce conflict but not a reduce-reduce conflict.
C. a reduce-reduce conflict but not a shift-reduce conflict.
D. neither a shift-reduce nor a reduce-reduce conflict

Explanation

- The question is asked with respect to the symbol ' < ' which is **not present** in the given canonical set of items.
- Hence it is neither a shift-reduce conflict nor a reduce-reduce conflict on symbol '<'.
- So **option D** is correct choice
- But if the question would have asked with respect to the symbol ' > ' then it would have been a shift-reduce conflict.

Question 3

Which of the following is true?

A. Canonical LR parser is LR (1) parser with single look ahead terminal
B. All LR(K) parsers with K > 1 can be transformed into LR(1) parsers.
C. Both (A) and (B)
D. None of the above

Explanation

- Canonical LR parser is LR (1) parser with single look ahead terminal. All LR(K) parsers with K > 1 can be transformed into LR(1) parsers.
- **Option (C) is correct.**

Question 4

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. Which is an exception?

A. Closure and goto operations work a little bit different
B. Closure and goto operations work similarly
C. Closure and additive operations work a little bit different
D. Closure and associatively operations work a little bit different

Explanation

- Closure and goto do work differently in case of LR (0) and LR (1)
- **Option (A) is correct.**

Question 5

When ß (in the LR(1) item A -> ß.a,a) is not empty, the look-head

A. Will be affecting.
B. Does not have any affect.
C. Shift will take place.
D. Reduction will take place.

Explanation

- There is no terminal before the non terminal beta
- **Option (B) is correct.**

Question 6

When ß is empty (A -> ß.,a), the reduction by A-> a is done

A. If next symbol is a terminal
B. Only If the next input symbol is a
C. Only If the next input symbol is A
D. Only if the next input symbol is a

Explanation

- The next token is considered in this case it’s a
- **Option (D) is correct.**

Thank U