Compiler Construction (CS-636)

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

(Week 10) Lecture 19 & 20

Objectives: Learning objectives of these lectures are

Students will able to understand:

- What is Bottom-Up Parser?
- What are types of Bottom-Up Parser?
- SLR(1) Parser Steps

1. Number the Production
2. Augment the given grammar (Augmented Grammar)
3. Find the Follow Set of Non-Terminals in given CFG
4. Draw the Canonical Collection
5. Create the Parsing Table
6. Stack Implementation
7. Draw Parse Tree

Text Book & Resources:

Compilers Principles Techniques and Tools (2nd Edition) by Alfread V. Aho, Ravi Sethi.

Videos Link:

https://youtu.be/m1mkjiUUm3A (Part 1)
https://youtu.be/48S73DRd8GU (Part 2)
https://youtu.be/0v0q84nTQdA (Part 3)
https://youtu.be/gEZIRmpgX9w (Part 4)
https://youtu.be/MeP_OaGHl6Y (Part 5)
https://youtu.be/-hyt3uBv6wI (Part 6)
https://youtu.be/Dn95j5rAJxc (Part 7)
https://youtu.be/ATEwm3Kr3do (Part 8)

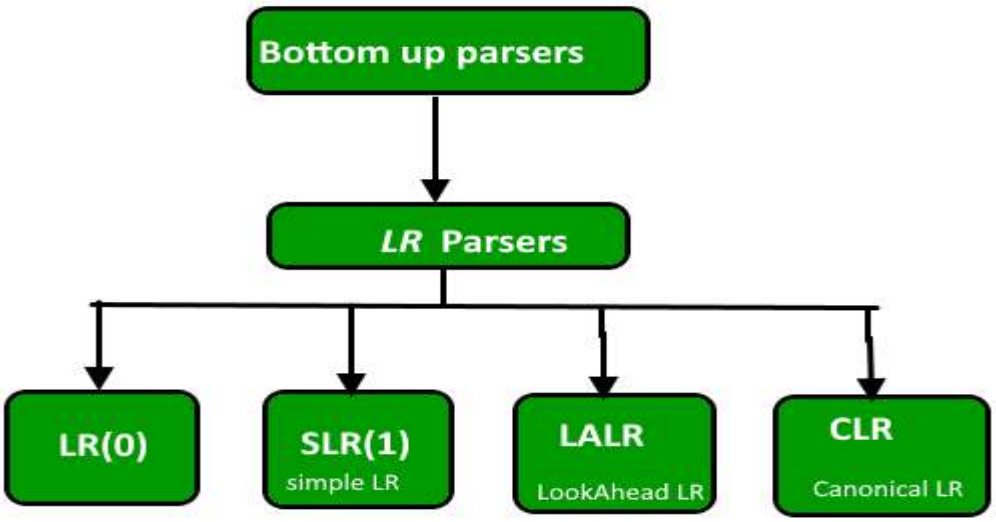
Compiler Construction (CS-636)

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Bottom Up or Shift Reduce Parsers

In the last week, we had discussed the Bottom Up Parsers and also discussed the following types of parsers. From these parsers we had discussed LR(0) Parser and in this week we will discuss the simple LR or SLR(1) Parser.

Classification (Types) of bottom up parsers



SLR (1) Parsing

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table. To construct SLR (1) parsing table, we use canonical collection of LR (0) item. In the SLR (1) parsing, we place the reduce move only in the follow set of left hand side non-terminals.

SLR Parser

We will first consider SLR(1) where the S stands for simple. 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 comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts. If we think back to the kind of conflicts we encountered in LR(0) parsing, it was the reduce actions that cause us grief. A state in an LR(0) parser can have at most one reduce action and cannot have both shift and reduce instructions. Since a reduce is indicated for any completed item, this dictates that each completed item must be in a state by itself. But let's revisit the assumption that if the item is complete, the parser must choose to reduce. Is that always appropriate? If we peeked at the next upcoming token, it may tell us something that invalidates that reduction. If the sequence on top of the stack could be reduced to the non-terminal A, what tokens do we expect to find as the next input? What tokens would tell us that the reduction is not appropriate? Perhaps Follow(A) could be useful here!

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

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. When filling in the table, we don't assume a reduce on all inputs as we did in LR(0), we selectively choose the reduction only when the next input symbols in a member of the follow set. To be more precise, here is the algorithm for SLR(1) table construction (note
all steps are the same as for LR(0) table construction except for 2a).

1. Construct F = {I0, I1, ... In}, the collection of LR(0) configuration sets for G'.

2. State i is determined from Ii. The parsing actions for the state are determined as follows:

a) If A → u• is in Ii then set Action[i,a] to reduce A →u for all a in Follow(A) (A is not S').

b) If S' → S• is in Ii then set Action[i,$] to accept.

c) If A → u•av is in Ii and successor(Ii, a) = Ij, then set Action[i,a] to shift j (a must be a terminal).

3. The goto transitions for state i are constructed for all non-terminals A using the rule: If successor(Ii, A) = Ij, then Goto [i, A] = j.

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

5. The initial state is the one constructed from the configurating set containing S' → •S.

In the SLR(1) parser, it is allowable for there to be both shift and reduce items in the same state as well as multiple reduce items. The SLR(1) parser will be able to determine which action to take as long as the follow sets are disjoint.

SLR(1) Grammars

A grammar is SLR(1) if the following two conditions hold for each configurating set:

1. For any item A → u•xv in the set, with terminal x, there is no complete item B → w• in that set with x in Follow(B). In the tables, this translates no shift reduce conflict on any state. This means the successor function for x from that set either shifts to a new state or reduces, but not both.

2. For any two complete items A → u• and B → v• in the set, the follow sets must be disjoint, e.g. Follow(A) ∩ Follow(B) is empty. This translates to no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which using only one token of look ahead.

All LR(0) grammars are SLR(1) but the reverse is not true, as the two extensions to our expression grammar demonstrated. The addition of just one token of lookahead and use of the follow set greatly expands the class of grammars that can be parsed without conflict.

SLR(1) Limitations

The SLR(1) technique still leaves something to be desired, because we are not using all the information that we have at our disposal. When we have a completed configuration (i.e., dot at the end) such as X → u•, we know that this corresponds to a situation in which we have u as a handle on top of the stack which we then can reduce, i.e., replacing u by X. We allow such a reduction whenever the next symbol is in Follow(X). However, it

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

may be that we should not reduce for every symbol in Follow(X), because the symbols below u on the stack preclude u being a handle for reduction in this case. In other words, SLR(1) states only tell us about the sequence on top of the stack, not what is below it on the stack. We may need to divide an SLR(1) state into separate states to differentiate the possible means by which that sequence has appeared on the stack. By carrying more information in the state, it will allow us to rule out these invalid reductions.

SLR(1) Parser Steps

1. Number the Production
2. Augment the given grammar (Augmented Grammar)
3. Find Follow Sets of Non-Terminals
4. Draw the Canonical Collection(DFD)
5. Create the Parsing Table
6. Stack Implementation
7. Draw Parse Tree

Example # 1

Given CFG

E → E + T | T
T → T * F | F
F → id

In this example we will perform all above steps to check whether the given grammar is SLR(1) conflict or not.

Step 1: Number the Production

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → id

Step 2: Augmented CFG

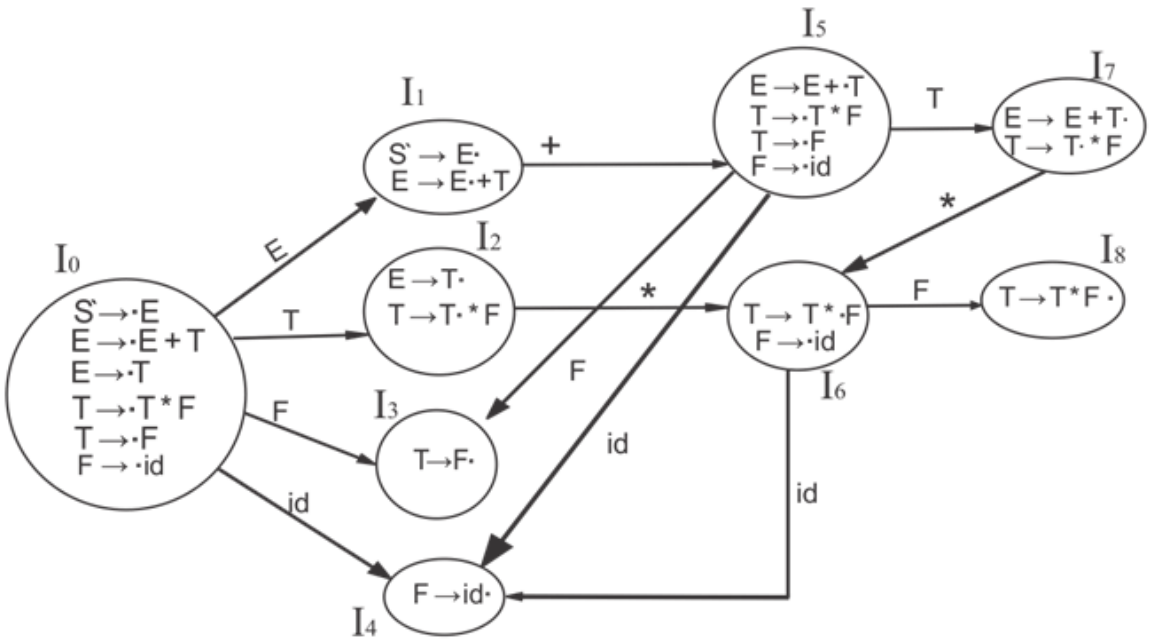
S'→E
E → E + T
E → T
T → T * F
T → F
F → id

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 3: Find the Follow Set of Non-Terminals

Non-Terminals	Follow Set
S'	{$}
E	{$,+}
T	{$,+,*}
F	{$,+,*}

Step 4: Canonical Collection (DFD)



Step 5: Parsing Table (R1: E→E+T , R2:E→T,R3: T→T*F, R4: T→F, R5: F→id)

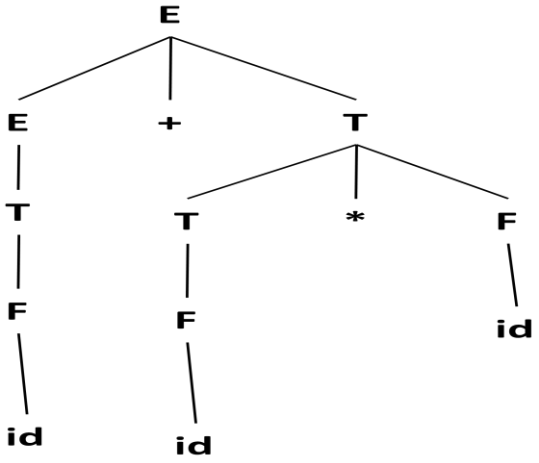
States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 6: Stack Implementation (String: id+id*id)

Stack	Input	Action
$0	id+id*id$	Shift id→S4
$0id4	+id*id$	Reduction (F→id) (R5)
$0F3	+id*id$	Reduction (T→F) (R4)
$0T2	+id*id$	Reduction (E→T) (R2)
$0E1	+id*id$	Shift +→S5
$0E1+5	id*id$	Shift id→S4
$0E1+5id4	*id$	Reduction (F→id) (R5)
$0E1+5F3	*id$	Reduction (T→F) (R4)
$0E1+5T7	*id$	Shift *→S6
$0E1+5T7*6	id$	Shift id→S4
$0E1+5T7*6id4	$	Reduction (F→id) (R5)
$0E1+5T7*6F8	$	Reduction (T→T*F) (R3)
$0E1+5T7	$	Reduction (E→E+T) (R1)
$0E1	$	Accept

Step 7: Draw Syntax Tree



Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Example # 2
Consider the grammar
S →AA
A → aA | b

In this example we will perform all above steps to check whether the given grammar is SLR(1) conflict or not.

Step 1: Number the Production

1. S →AA
2. A → aA
3. A → b

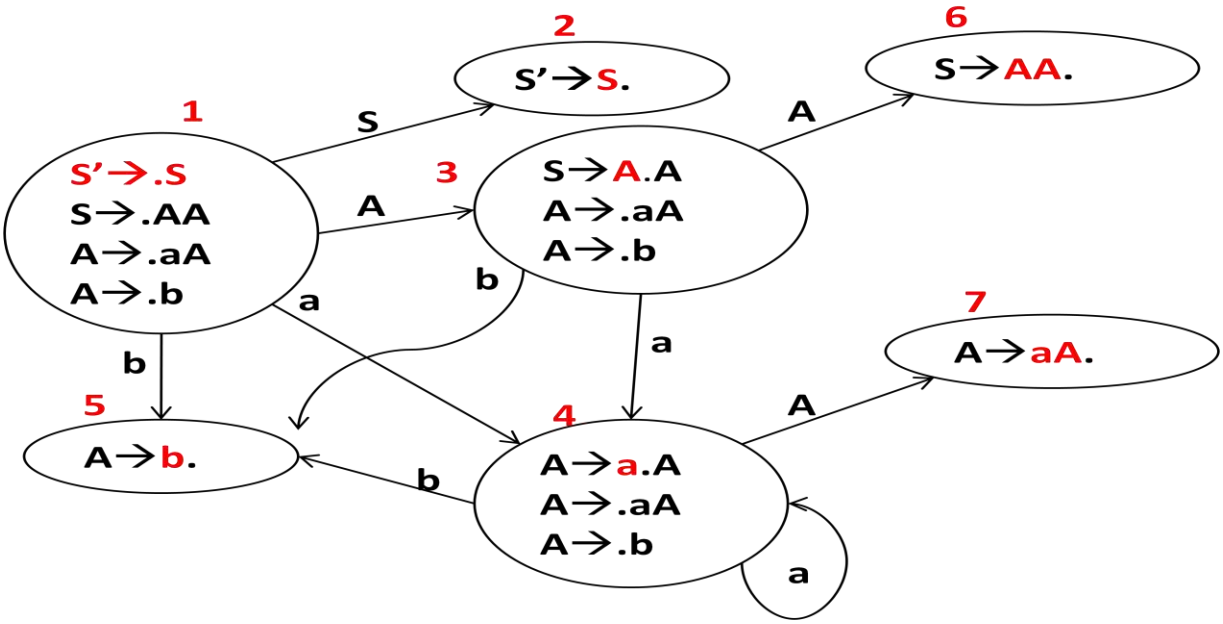
Step 2: Augmented CFG

S'→S
S →AA
A → aA
A → b

Step 3: Find the Follow Set of Non-Terminals

Non-Terminals	Follow Set
S'	{$}
S	{$}
A	{$, a , b}

Step 4: Canonical Collection



Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

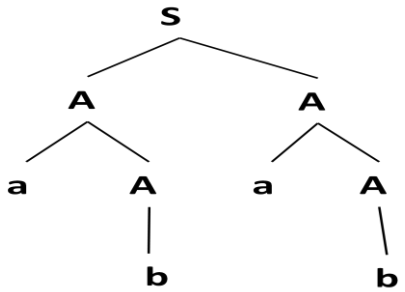
Step 5: Parsing Table (R1: S→AA , R2: A→aA , R3: A→b)

States	Action Part			Goto Part	
	a	**b**	**$**	**S**	**A**
1	S4	S5		2	3
2			Accept		
3	S4	S5			6
4	S4	S5			7
5	R3	R3	R3		
6			R1		
7	R2	R2	R2		

Step 6: Stack Implementation (String: **abab**)

Stack	Input	Action
$1	abab$	Shift a→S4
$1a4	bab$	Shift b→S5
$1a4b5	ab$	Reduction (A→b) (R3)
$1a4A7	ab$	Reduction (A→aA) (R2)
$1A3	ab$	Shift a→S4
$1A3a4	b$	Shift b→S5
$1A3a4b5	$	Reduction (A→b) (R3)
$1A3a4A7	$	Reduction (A→aA) (R2)
$1A3A6	$	Reduction (S→AA) (R1)
$1S2	$	Accept

Step 7: Draw Syntax Tree



Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Example # 3

In this example we will perform all above steps to check whether the given grammar is SLR(1) conflict or not.

Given CFG:

E→E+(E)

E→id

Given String:

id+(id)

Step 1: Number the Production

1. E→E+(E)
2. E→id

Step 2: Augmented CFG

E'→E

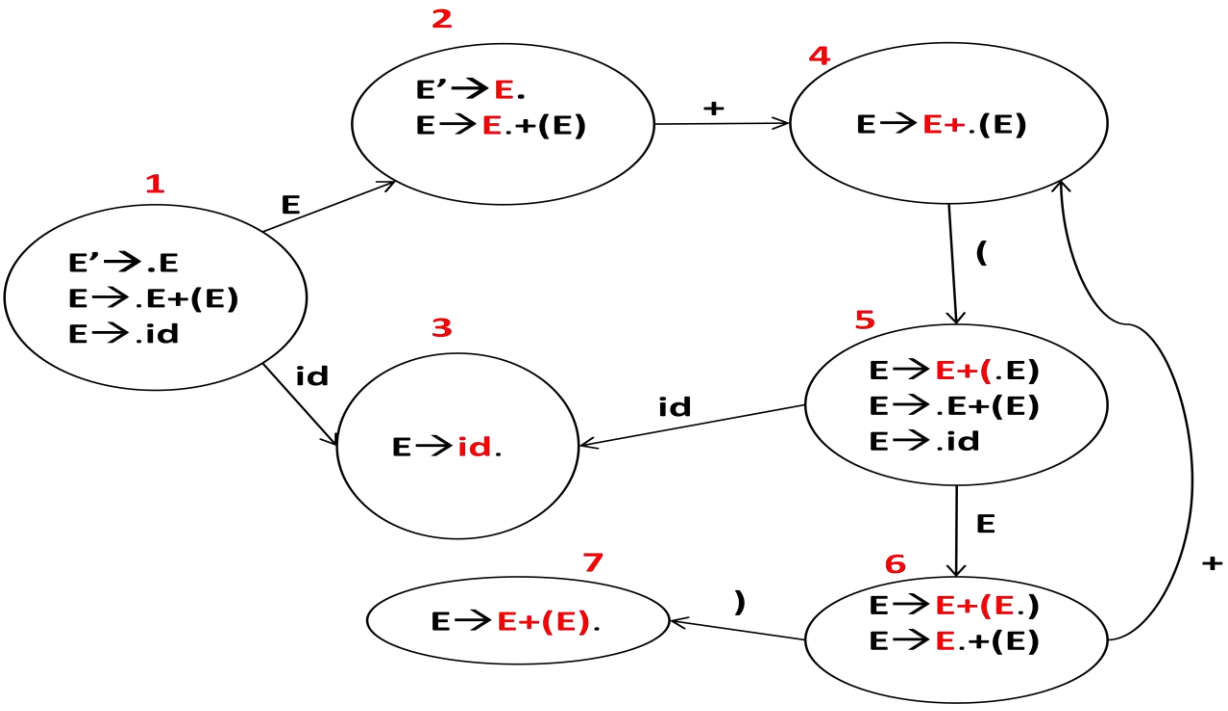
E→E+(E)

E→id

Step 3: Find the Follow Set of Non-Terminals

Non-Terminals	Follow Set
E'	{$}
E	{$, + ,) }

Step 4: Canonical Collection



Compiler Construction (CS-636)

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

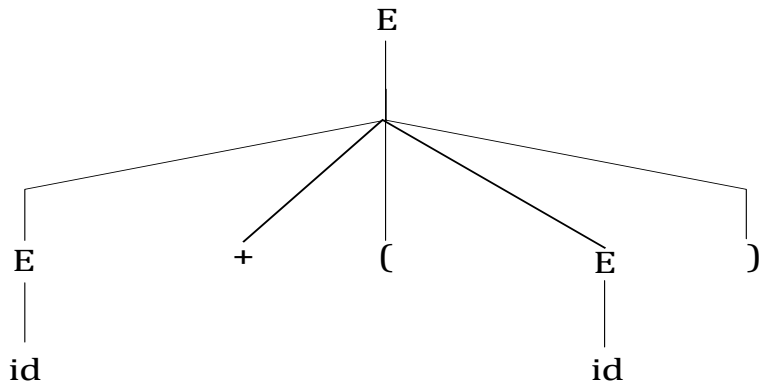
Step 5: Parsing Table (R1: E→E+(E) , R2: E→id)

States	**Action Part**					**Goto Part**
	+	**(**	**Id**	**)**	**$**	**E**
1			S3			2
2	S4				Accept	
3	R2			R2	R2	
4		S5				
5	S4		S3			6
6	S4			S7		
7	R1			R1	R1	

Step 6:Stack Implementation (String: **id+(id)**)

Stack	Input	Action
$1	id+(id)$	Shift id→S3
$1id3	+(id)$	Reduction (E→id) (R2)
$1E2	+(id)$	Shift +→S4
$1E2+4	(id)$	Shift (→S5
$1E2+4(5	id)$	Shift id→S3
$1E2+4(5id3	)$	Reduction E→id (R2)
$1E2+4(5E6	)$	Shift)→S7
$1E2+4(5E6)7	$	Reduction E→E+(E) (R1)
$1E2	$	Accept

Step 7: Draw Syntax Tree



Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

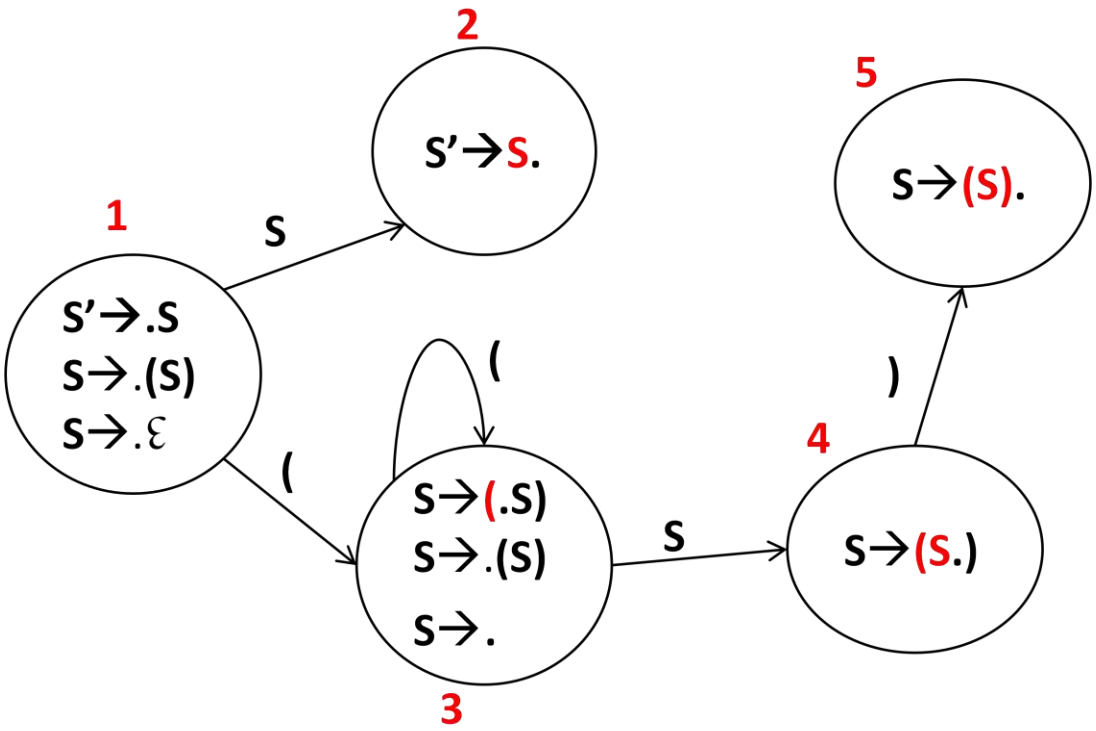
Example # 4
Given Grammar
S→(S) | ∈
Step # 1: Number the Production
1. S→(S)
2. S→∈

Step # 2: Augmented Grammar
S'→S
S→(S)
S→∈

Step 3: Find the Follow Set of Non-Terminals

Non-Terminals	Follow Set
S'	{$}
S	{$,) }

Step 4: Canonical Collection



Note: S→.∈ = S→. so there is to need to check next symbol

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 5: Parsing Table (R1: S→(S) , R2: S→∈)

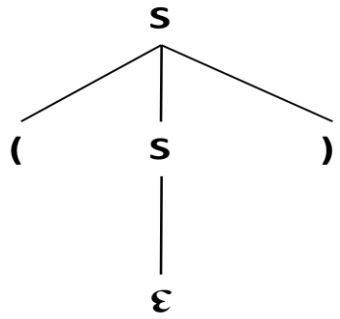
States	Action Part			Goto Part
	(	)	$	S
1	S3	R2	R2	2
2			Accept	
3	S3	R2	R2	4
4		S5		
5		R1	R1	

Step 6:Stack Implementation (String: ( ))

Stack	Input	Action
$1	( )$	Shift (→S3
$1(3	)$	Reduction (S→∈ (R2)
$1(3S4	)$	Shift)→S5
$1(3S4)5	$	Reduction (S→(S) (R1)
$1S2	$	Accept

./

Step 7: Draw Syntax Tr/ee/



Example # 5

Given Grammar

S→ SS+ | SS*| a

Step # 1: Number the Production

1. S→SS+
2. S→ SS*
3. S→a

Step # 2: Augmented Grammar

S'→S

S→SS+

S→ SS*

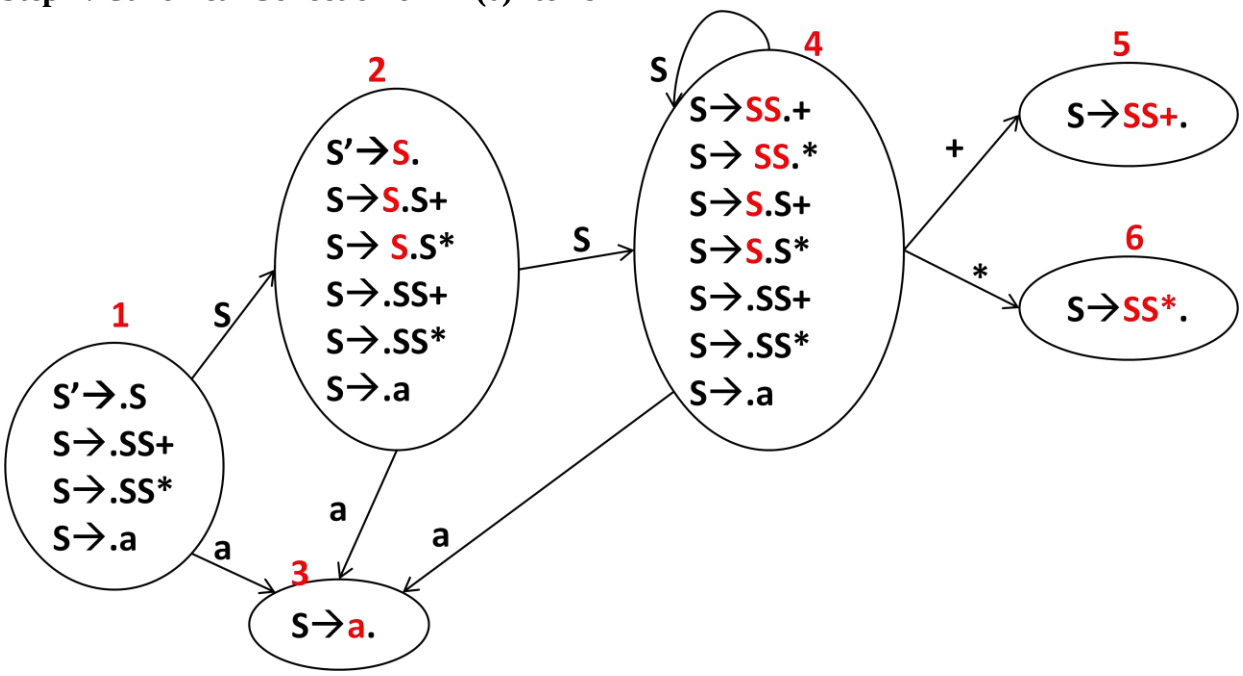
S→a

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 3: Find the Follow Set of Non-Terminals

Non-Terminals	Follow Set
S'	{$}
S	{$, a,+,* }

Step 4: Canonical Collection of LR(0) Items



Step 5: Parsing Table (R1: S→SS+ , R2: S→SS* , R3: S→a)

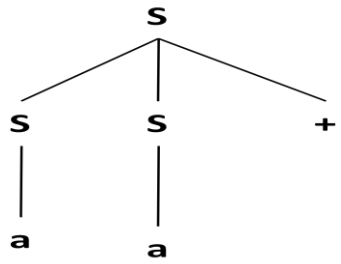
States	Action Part				GoTo Part
	+	*	a	$	S
1			S3		2
2			S3	Accept	4
3	R3	R3	R3	R3	
4	S5	S6	S3		4
5	R1	R1	R1	R1	
6	R2	R2	R2	R2	

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step6: Stack Implementation (String: aa+)

Stack	Input	Action
$1	aa+$	Shift a→S3
$1a3	a+$	Reduction R3(S→a)
$1S2	a+$	Shift a→S3
$1S2a3	+$	Reduction R3(S→a)
$1S2S4	+$	Shift +→S5
$1S2S4+5	$	Reduction R1(S→SS+)
S1S2	$	Accept

Step 7: Draw Syntax Tree



Example # 6
Given Grammar
S→ (L) | a
L→ L,S | S
Step # 1: Number the Production

1. S→ (L)
2. S→ a
3. L→ L,S
4. L→ S

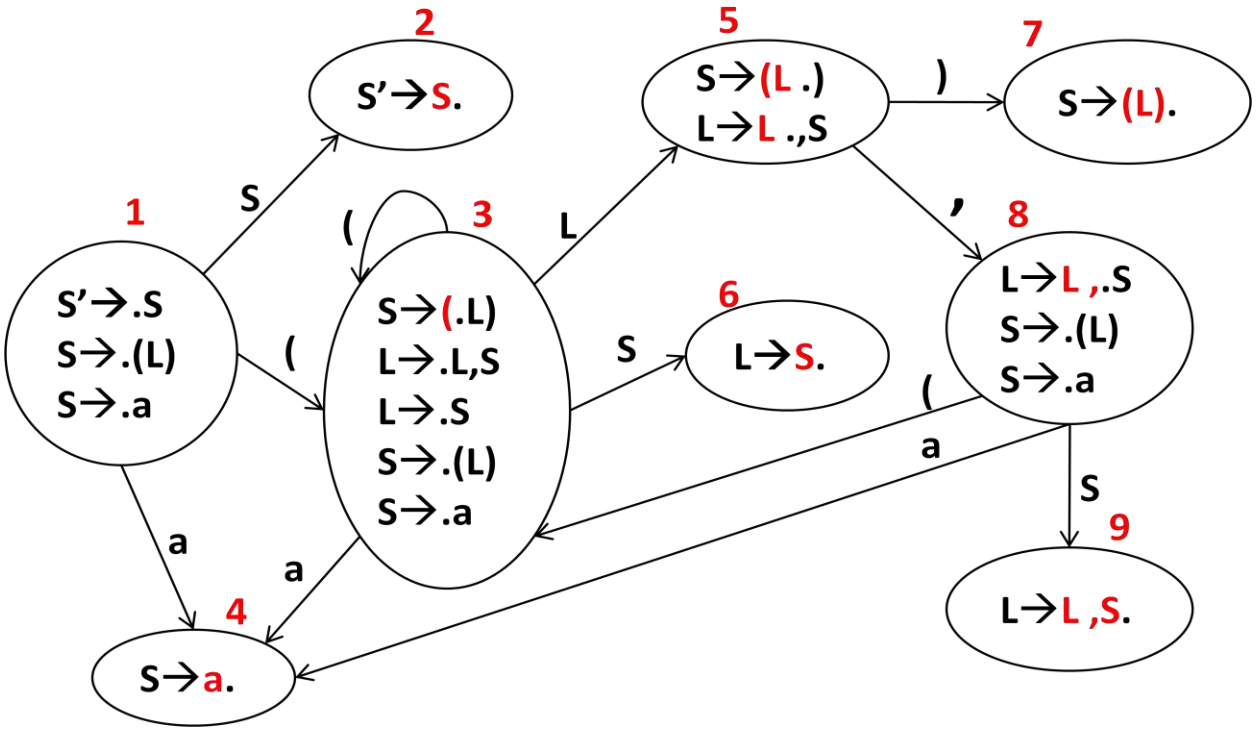
Step # 2: Augmented Grammar
S'→S
S→ (L)
S→ a
L→ L,S
L→ S

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 3: Find the Follow Set of Non-Terminals

Non-Terminals	Follow Set
S'	{$}
S	{$,) , ,}
L	{) , ,}

Step 4: Canonical Collection



Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 5: Parsing Table (R1: S→(L) , R2: S→a , R3: L→L,S , R4: L→S)

States	Terminal Part					Goto Part	
	(	)	a	,	$	S	L
1	S3		S4			2	
2					Accept		
3	S3		S4			6	5
4		R2		R2	R2		
5		S7		S8			
6		R4		R4			
7		R1		R1	R1		
8	S3		S4			9	
9		R3		R3			

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Step 6:Stack Implementation (String: (a,a))

Stack	Input	Action
$1	(a , a)$	Shift (→S3
$1(3	a , a)$	Shift a→S4
$1(3a4	,a)$	Reduction R2(S→a)
$1(3S6	,a)$	Reduction R4(L→S)
$1(3L5	,a)$	Shift , →S8
$1(3L5,8	a)$	Shift a →S4
$1(3L5,8a4	)$	Reduction R2(S→a)
$1(3L5,8S9	)$	Reduction R3(L→L,S)
$1(3L5	)$	Shift) →S7
$1(3L5)7	$	Reduction R1(S→(L)
$1S2	$	Accept

Step 7: Draw Syntax Tree