Bottom-Up Parsing LR(0) and SLR(1)

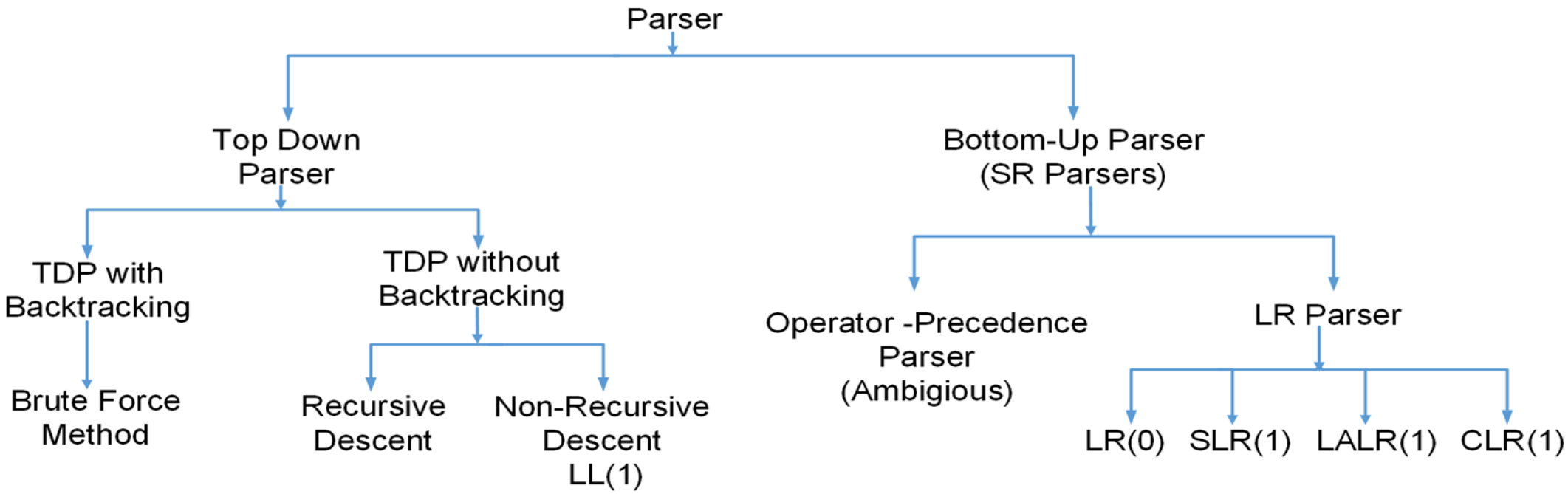
Baivab Das

Lecturer

Department of CSE

University of Asia Pacific

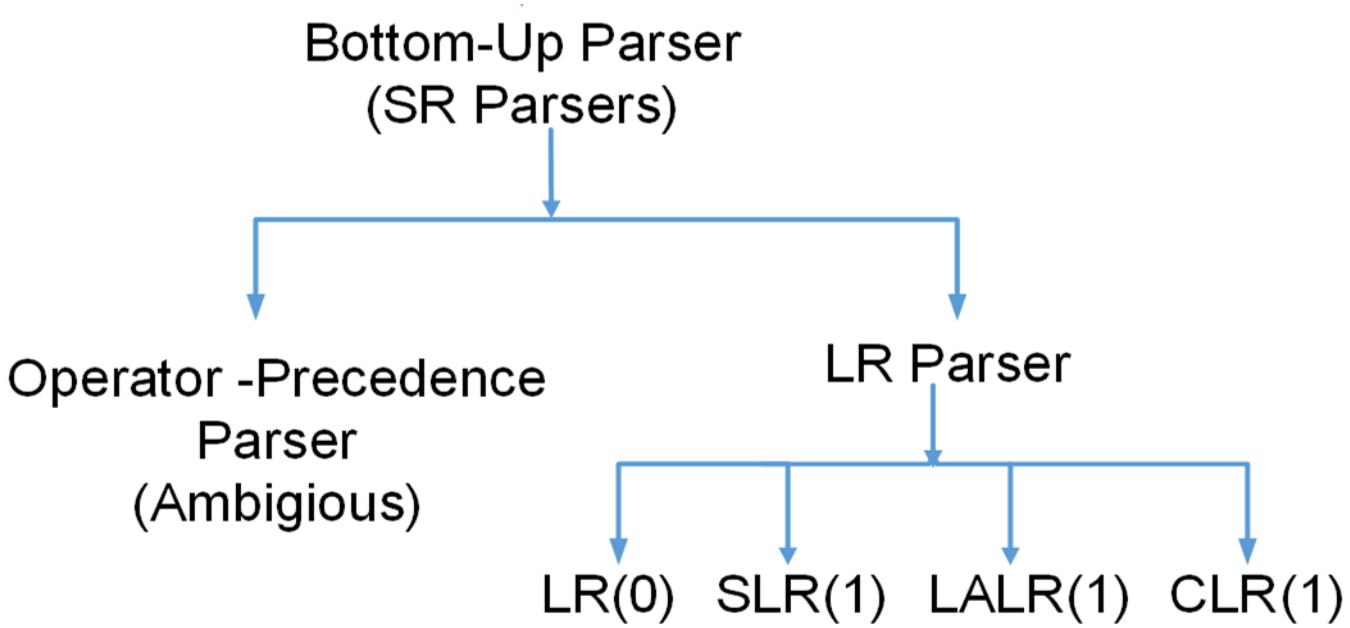
Parser Hierarchy



SR = Shift Reduce
Shift means pushing
Reduce means popping
So it uses a stack

LR = **L**eft-to-right, **R**ightmost derivation in reverse
In TDP without backtracking, LR and Non-Determinism is not accepted
Only Operator Precedence Parser can pass Ambiguous Grammars

Bottom Up Parsers



SR = Shift Reduce
Shift means pushing
Reduce means popping
So, it uses a stack

SLR=Simple LR
LALR = Look Ahead LR
CLR = Canonical LR

SR Parsers

- SR Parser is a Bottom Up Parser
- So, it builds the parse tree from bottom to top
- From the input string '*w*' we need to retrieve the start symbol '**S**' of the grammar
- It use a stack and the initial configuration of the stack is:

Stack	Input
$	w$
⋮	⋮
⋮	⋮
⋮	⋮
↓	↓
$S	$

Actions in SR Parsers:

1. Shift
2. Reduce
3. Accept
4. Error

Actions used in SR Parsers

1. Shift: Parser shifts zero or more input symbols until handle *β*
2. Reduce: *β* is reduced to left hand side of te production
3. Accept: Announces successful completion
4. Error: Calls an error recovery routine

Handle is a substring which matches with the right-hand side of the production

$$A \rightarrow \beta$$

or

$$A \rightarrow abc$$

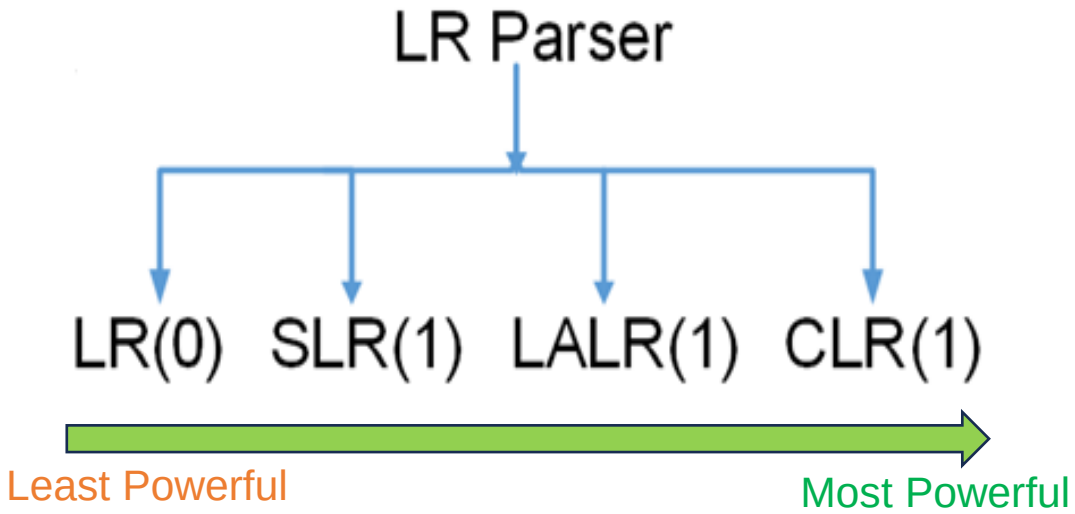
Here the handle is *abc*

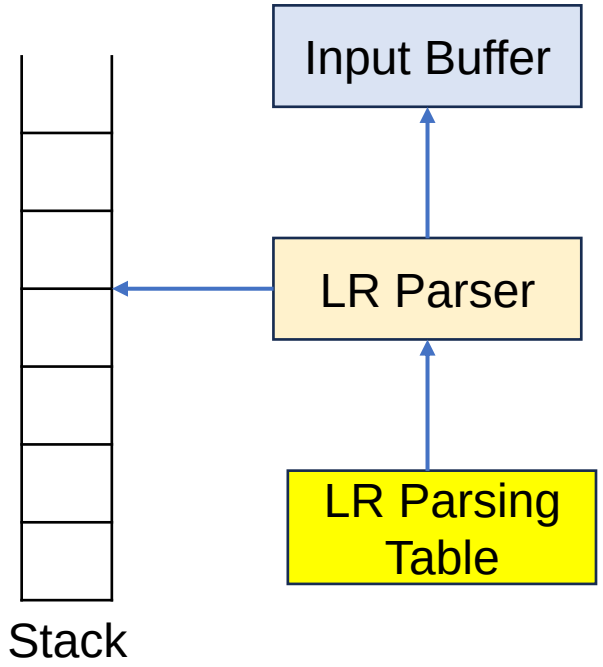
- So, *abc* will be replaced (reduced) by A
- The parser repeatedly performs the shift-reduce actions until the final configuration is reached

Stack	Input
$S	$

- Or an error occurs

LR Parsers





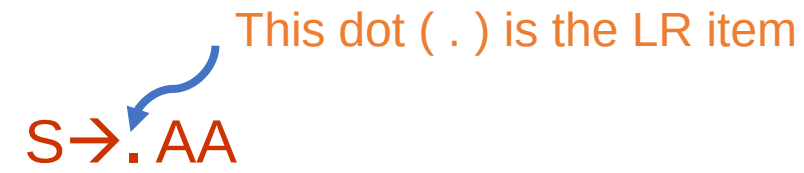
- For all the 4 parsers, the parsing algorithm is same
- Only change is the construction of **parsing table**
- **Canonical collection of LR(0) items** are used to construct the parsing table of LR(0) and SLR(1) parser
- **Canonical collection of LR(1) items** are used to construct the parsing table of LALR(1) and CLR(1) parser

LR(0) Items

- In LL(1) we have used First and Follow
- In LR Parsers, we have **Closure and Goto**
- Look at the following grammar:

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

  - We take the production S→AA and add a dot (.)
  - We add one more production to the grammar called **augmented grammar**



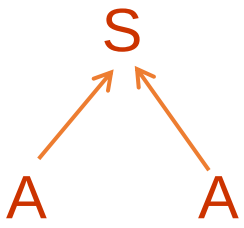
  - The dot means after the dot we have not seen anything

$$S \rightarrow A . A$$

  - The dot means before the dot we have **A** so we have seen **A**

$S \rightarrow A\ A\ .$

We have seen everything
from the right-hand side



- After we have seen everything in the right-hand side (handle), we apply the reduce operation
- Now, We add one more production to the grammar called **augmented grammar**

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA \mid b$

- We use the LR(0) items to determine the current state of the actions.
- We can assess till what part we have seen the input and if we can reduce or not

S' → . S
S → . AA
A → . aA | . b

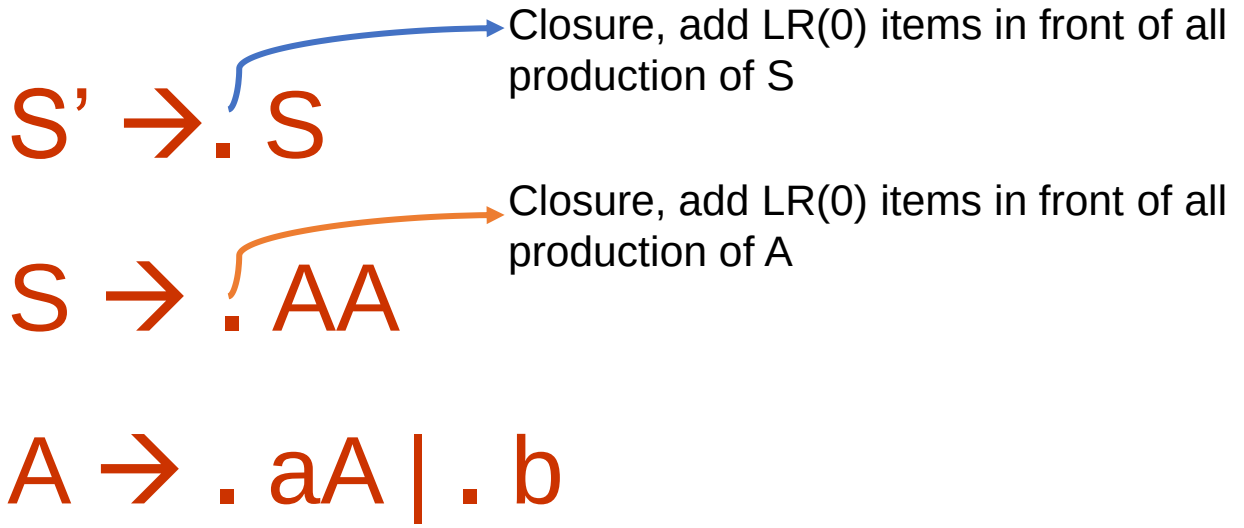
- Carefully observe the position of the dots (.)
- Try to understand the current position and what is already seen in the right hand side of the productions
- Now, we need to use two things: **CLOSURE and GOTO**

Understanding Closure

Let us start from the 1st production

S' →. S

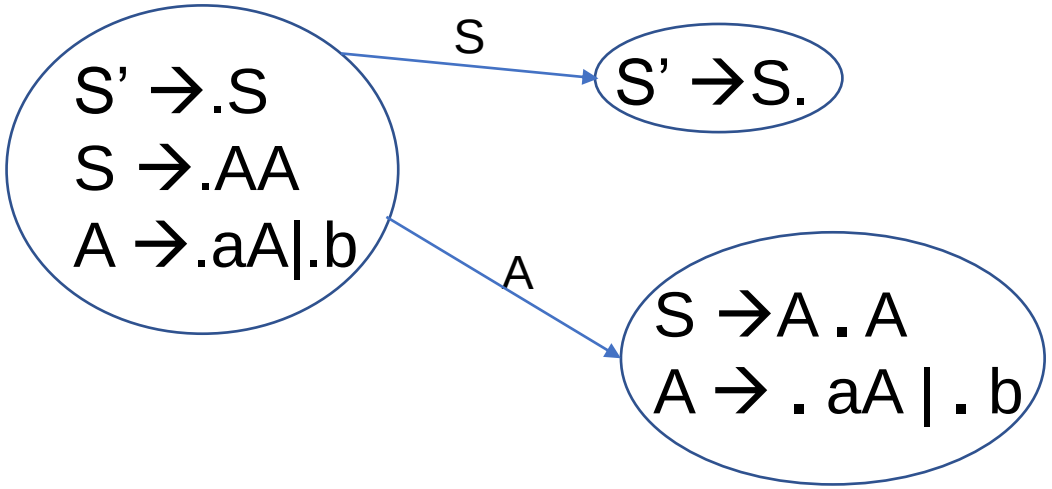
- There is a dot (.) in front of non-terminal (variable) S in the right hand side
- So, we need to add a dot (.) in all the productions of the non-terminal S



This is the closure of S → .S

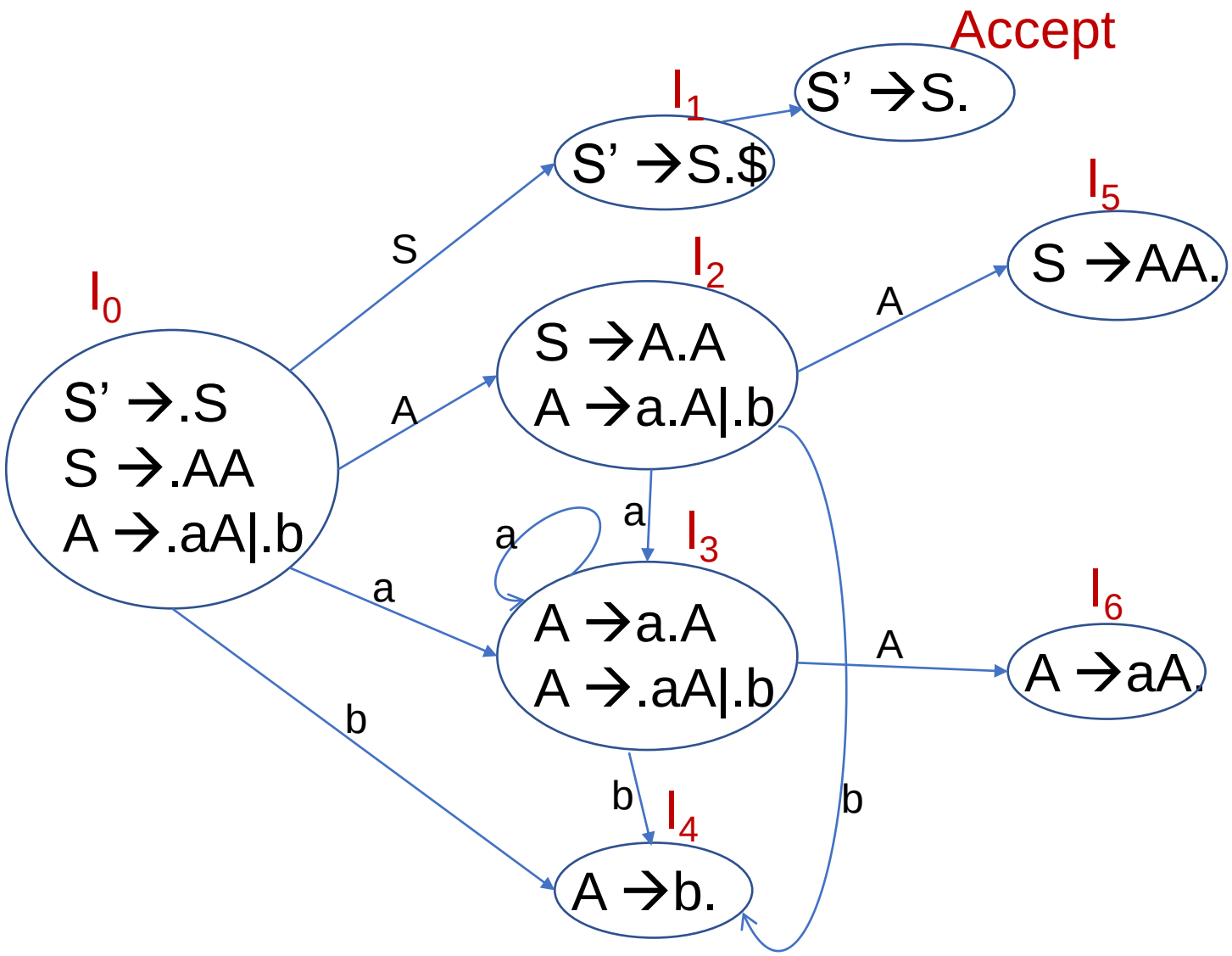
GOTO

- Goto is like a DFA
- We move the dot over the next symbol



Observe here, GOTO operation resulting a CLOSURE!

S’ →S
S →AA
A →aA|b



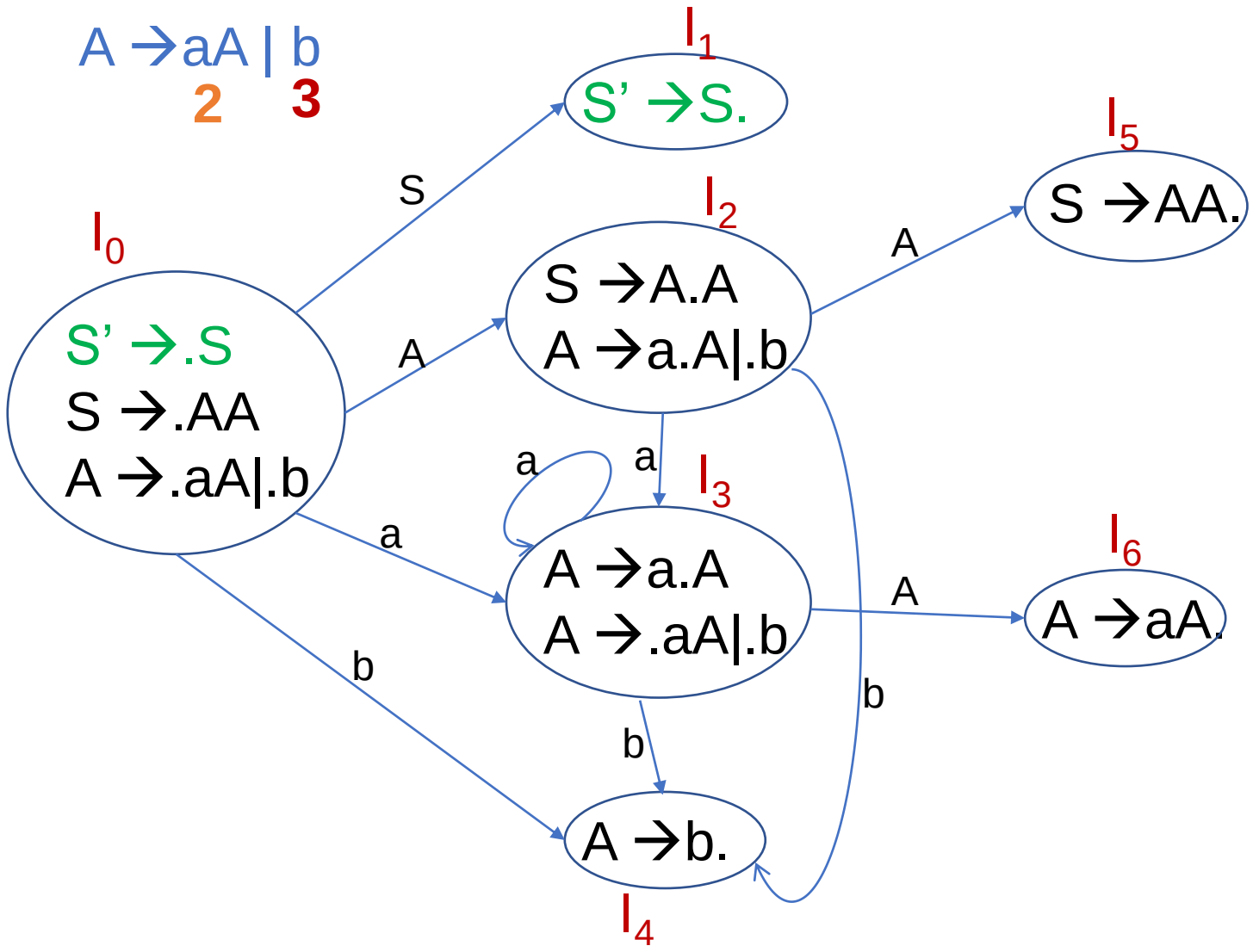
Whenever dot is in the rightmost side, it is called as final item

S’ →S

S →AA 1

A →aA | b

2 3



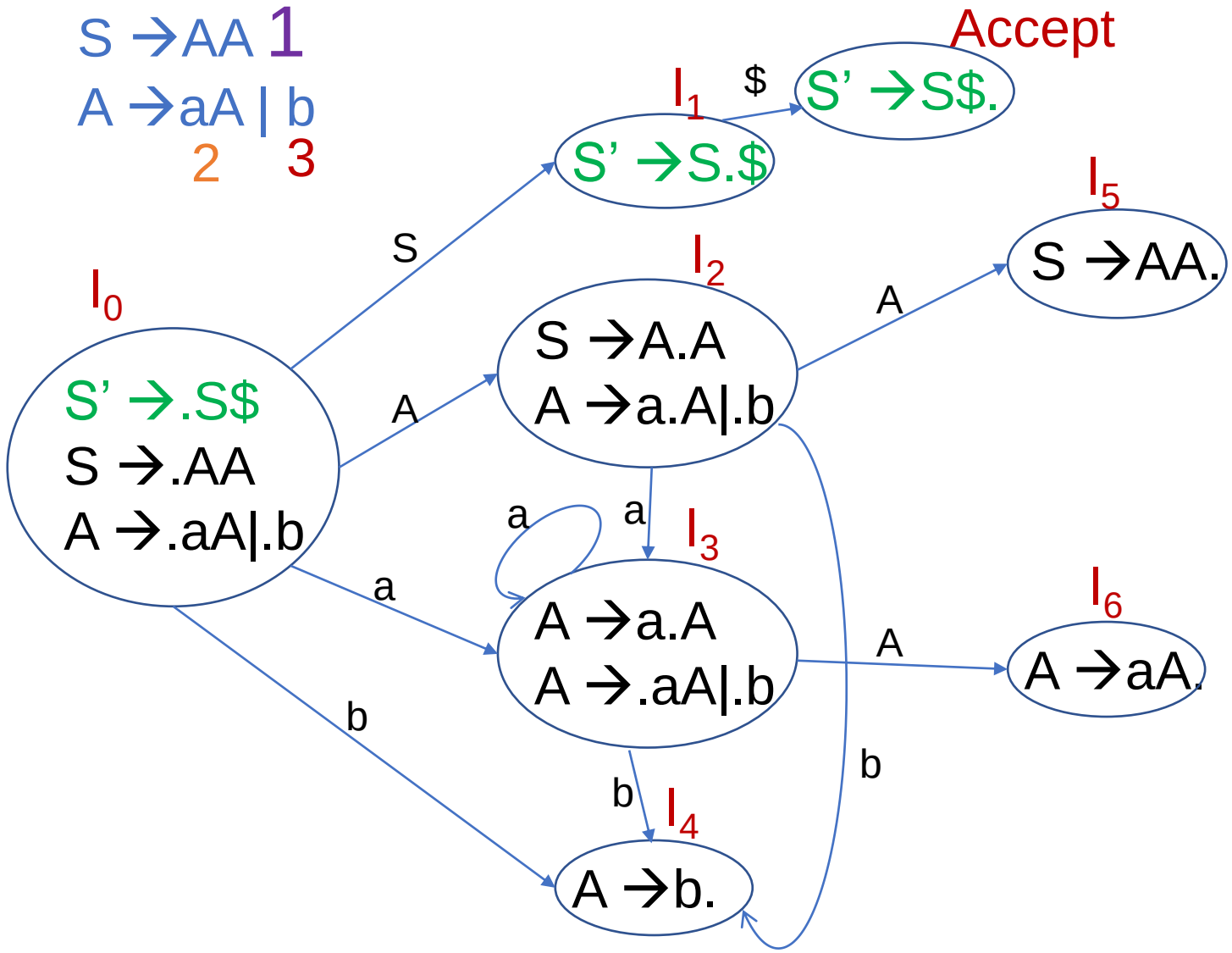
LR(0) Parsing Table

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

S' →S

S →AA 1

A →aA | b
2 3



LR(0) Parsing Table

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Parsing an Input String ***aabb***

S → AA 1
A → aA | b
2 3

Stack	Input	Action
0	aabb$	Shift 3
0a3	abb$	Shift 3
0a3a3	bb$	Shift 4
0a3a3b4	b$	Reduce by A→b
0a3a3A6	b$	Reduce by A→aA
0a3A6	b$	Reduce by A→aA
0A2	b$	Shift 4
0A2b4	$	Reduce by A→b
0A2A5	$	Reduce by S→AA
0S1	$	Accept

	Action			Goto	
	a	b	$	A	S
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

SLR(1): Simple LR

- Main difference between LR(0) and SLR(1) is in terms of reduce move
- Both use canonical collection of LR(0) items but the reduce move in the parsing table is different
- SLR(1) does not reduce all the symbols
- The reduce move will only happen when the lefthand side if followed
- 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 nonterminal being reduced

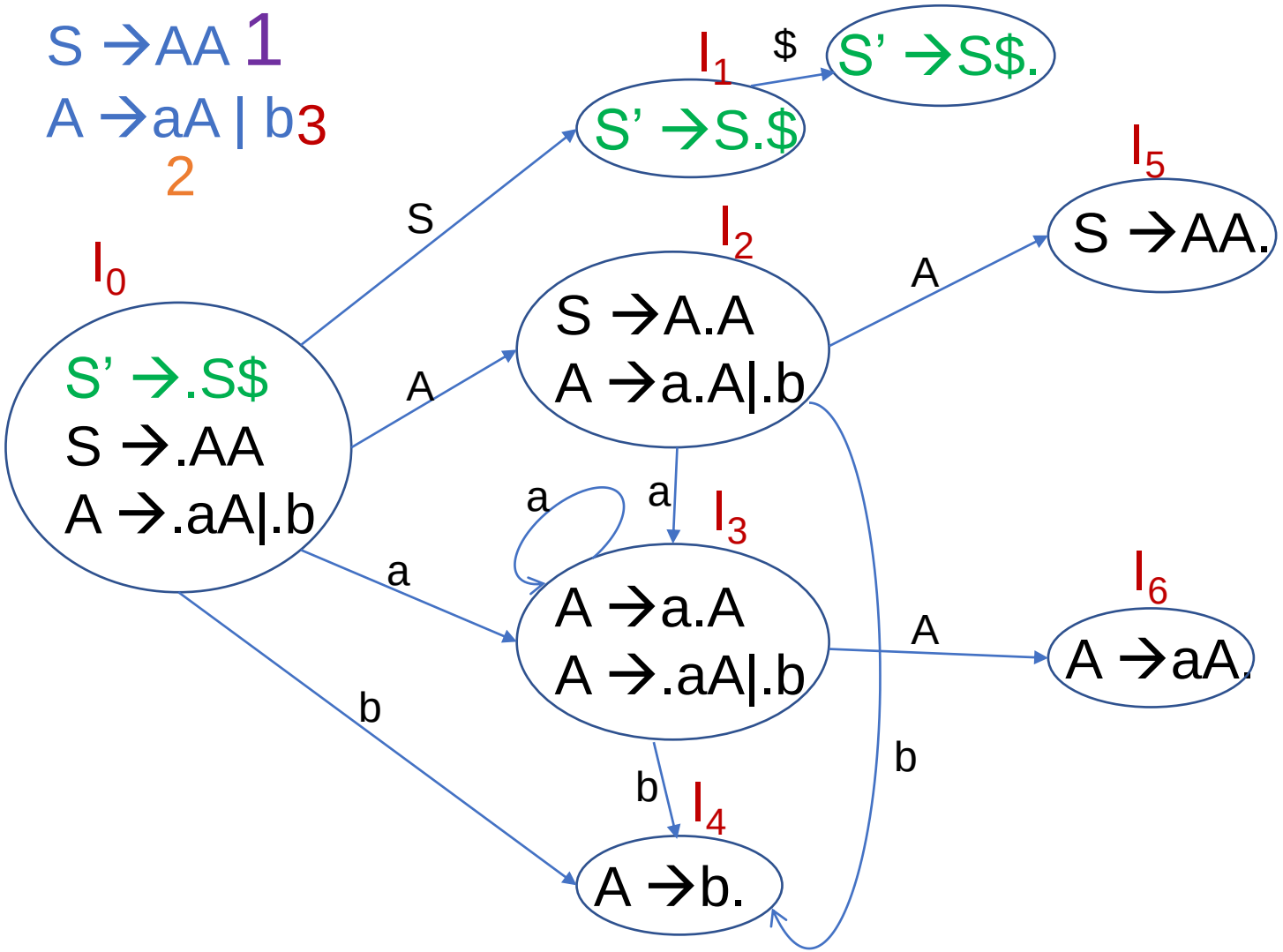
SLR(1) Parsing Table

S' →S

S →AA 1

A →aA | b3

2



NT	FIRST	FOLLOW
S	{a,b}	{$}
A	{a,b}	{a,b,$}

SLR(1) Parsing Table

	Action			**Goto**	
	a	**b**	**$**	**A**	**S**
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

LR(0) vs SLR(1)

LR(0) Parsing Table

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

SLR(1) Parsing Table

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	s_3	s_4		2	1
1			Accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

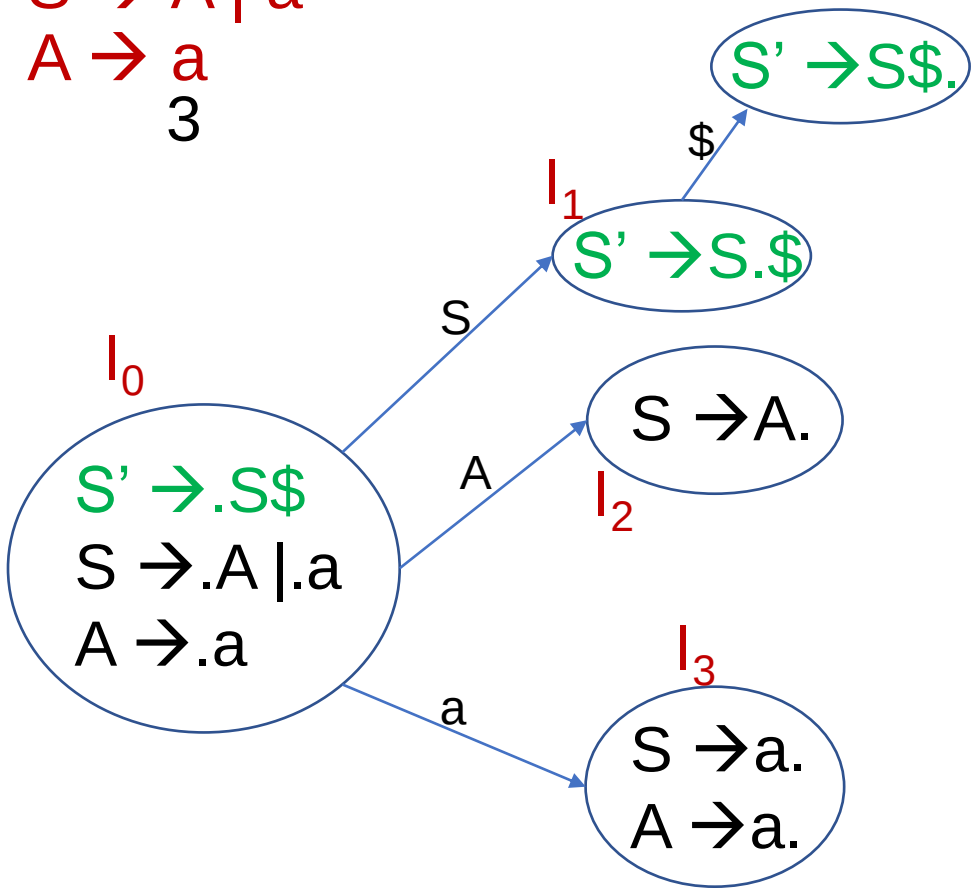
- In terms of shift and goto moves, both SLR(1) and LR(0) is same
- We can observe the difference in the reduce moves
- In both parsing tables, the empty cells are errors
- As SLR(1) is having less number of reduce moves, if there is any error, it will detect the error faster

Are All Grammars are LR(0)?

- No, not all grammar are LR(0)
- There are two types of conflicts:
- Shift-Reduce (SR) Conflict
- Reduce-Reduce (RR) Conflict
- If a grammar is having SR conflict or RR conflict, it cannot be LR(0)
- If a grammar is LR(0) it will definitely be SLR(1) but not vise-versa

$S \xrightarrow{1} A \mid a^{2}$

$A \rightarrow a$ 3



Observe, I_0 to I_3 is happening on a. Here, we can see two productions on 'a'. As it is a DFA, it cannot have multiple transition from a same symbol

NT	FIRST	FOLLOW
S	{a}	{$}
A	{a}	{$}

LR(0) Parsing Table

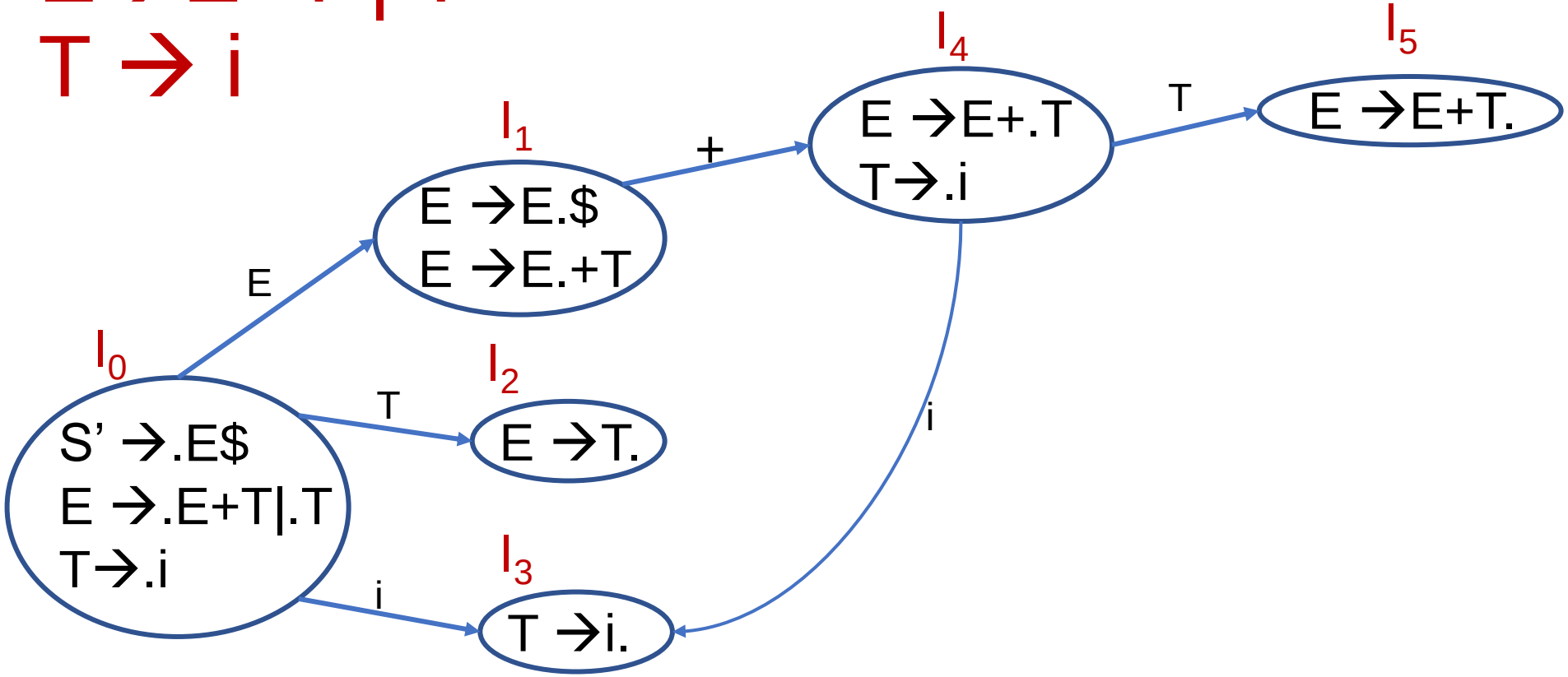
	Action		Goto	
	a	**$**	**A**	**S**
0	s_3		2	1
1		Accept		
2	r_1	r_1		
3	r_2/r_3	r_2/r_3		

SLR(1) Parsing Table

	Action		Goto	
	a	**$**	**A**	**S**
0	s_3		2	1
1		Accept		
2		r_1		
3		r_2/r_3		

Here, the grammar is not LL(1)
The grammar is neither LR(0) nor SLR(1)
Ambiguous grammar

E→E+T | T
T → i



$$E \rightarrow T+E \mid T$$
$$T \rightarrow i$$

1
S→ AaAb|BbBa
2
3 A→ ε
4 B→ ε

Remember,

A→ .ε or A→ ε.

Can be written as A→.

I_0

S'→.S
S→ .AaAb | .BbBa
A→ .ε
B→ .ε

NT	FOLLOW
S	{$}
A	{a,b}
B	{a,b}

- There is conflict as there are two reduce moves here
- So, the grammar will not be LR(0).
- If we want to check if the grammar will be SLR(1) or not, let us check the reduce moves for I_0

	Action		
	A	**b**	**$**
0	r_3/r_4	r_3/r_4	

- So, the grammar is nor SLR(1) either
- But, check if the grammar is LL(1)

S→AS | b
A → SA | a

- Check if the grammar is LL(1), LR(0) and SLR(1) or not

S → Aa | bAc | dc | bda
A → d

- Check if the grammar is LL(1), LR(0) and SLR(1) or not