Syntax Analysis

Course Textbook (Chapter 4)

Faryal Shamsi

Department of Computer Science

Sukkur IBA University

Bottom-up Parsing (BUP)

LR(0) parsers

Bottom-up Parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

F ☾ E+T|T
T ☾ T*F|F
F ☾ (E)|id

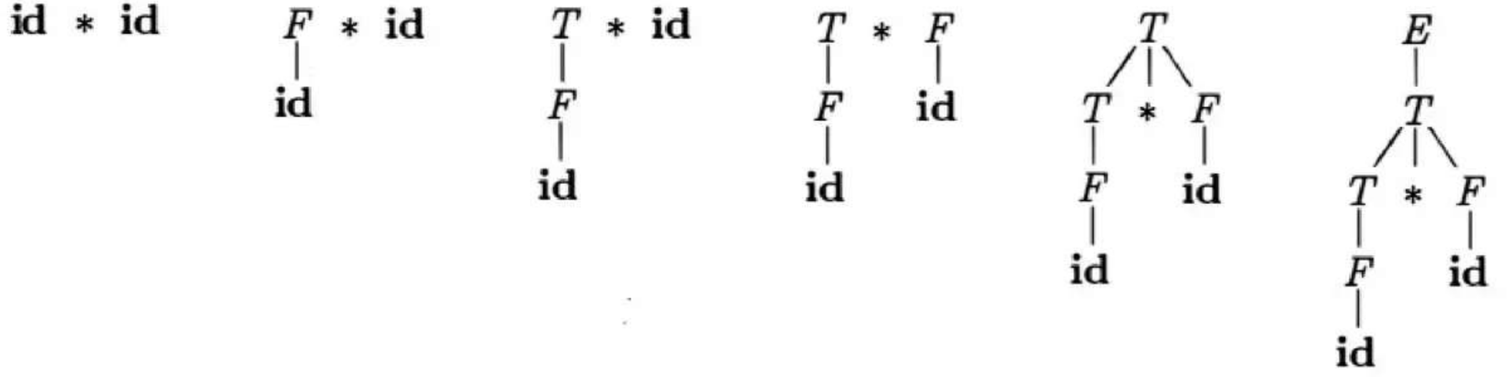


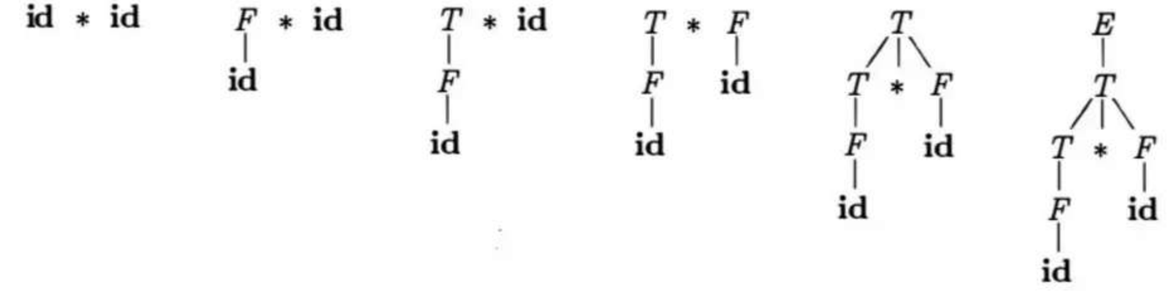
Figure 4.25: A bottom-up parse for **id * id**

Reductions

- At each reduction step , a specific substring matching the body of a production is replaced by the non-terminal at the head of that production $\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$

F ☾ E+T|T
T ☾ T*F|F
F ☾ (E)|id

  - First id reduced to F due to F ☾ id
  - Then F reduces to T due to T ☾ T production
  - Then Second id reduced to F
  - And so on



Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- We use $ to mark the bottom of the stack and also the right end of the input.
- Initially, the stack is empty, and the string w is on the input , as follows:

STACK	INPUT
$	w $

Shift Reduce Parser actions

Four possible actions a shift-reduce parser can make:

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

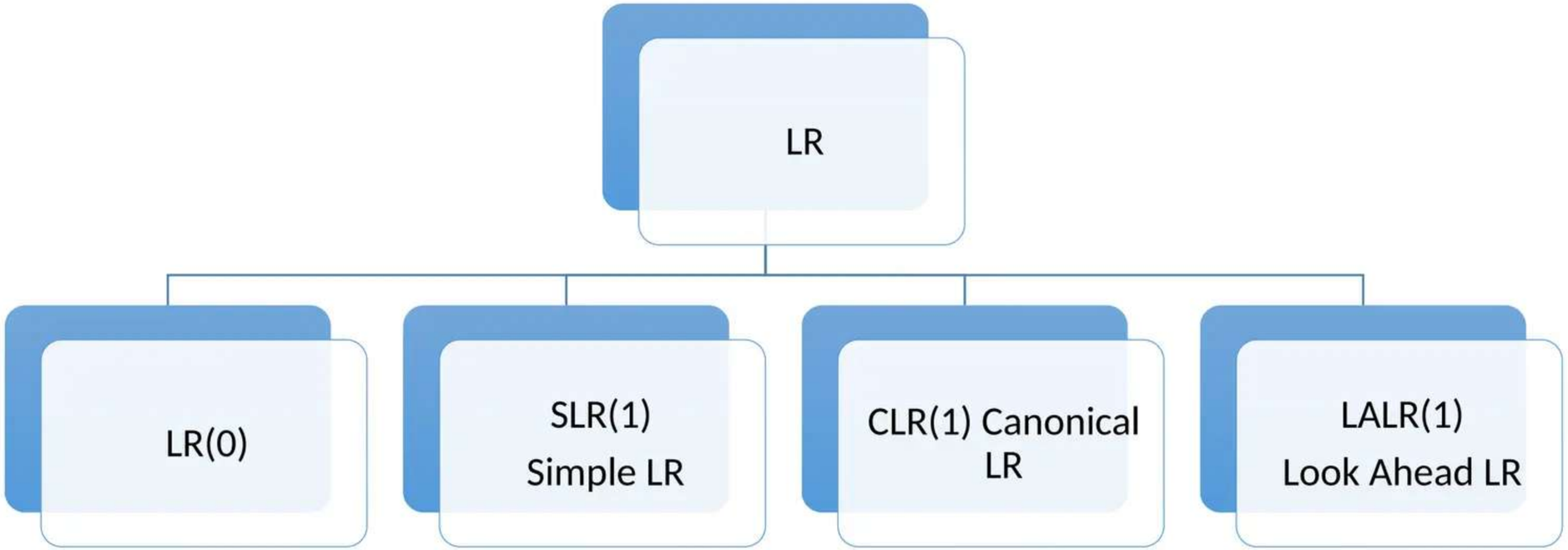
Conflict during Shift Reduce Parsing

- 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 to reduce (a shift/reduce conflict) ,
    - or cannot decide which of several reductions to make (a reduce /reduce conflict).

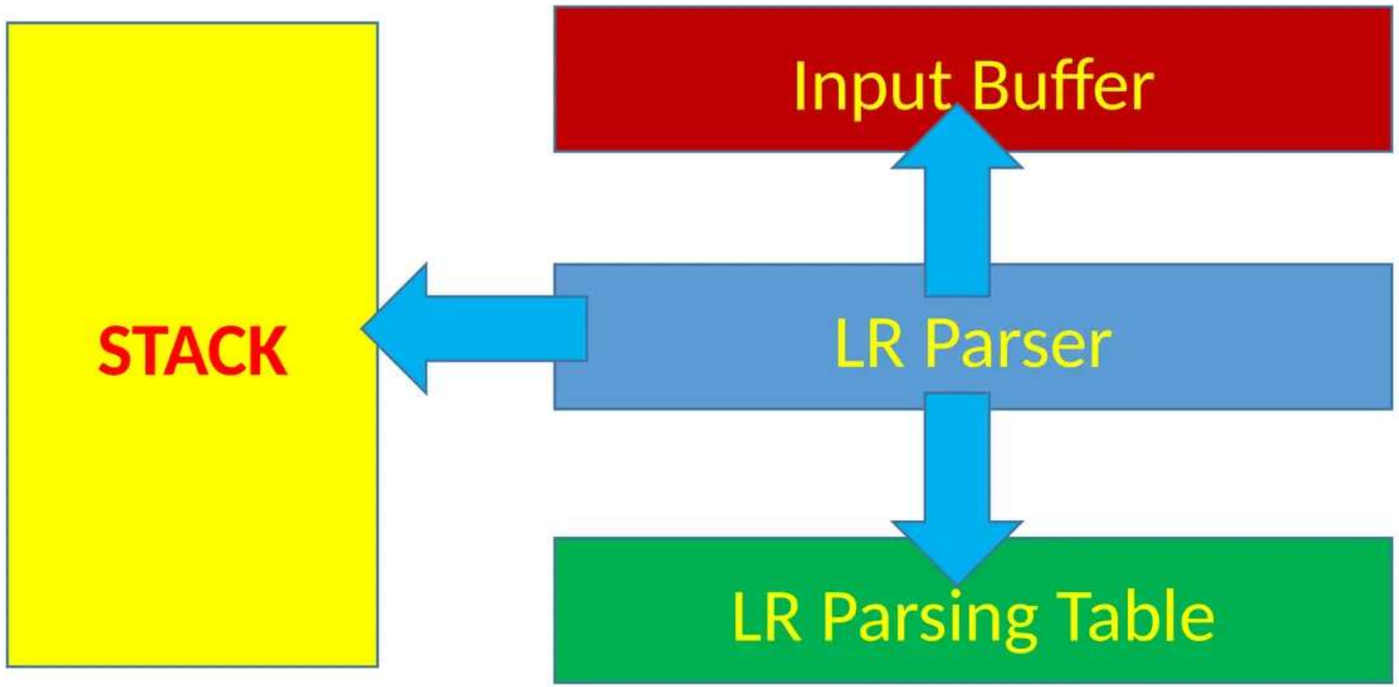
- We see some examples of SR and RR conflict in next lecture slides.

LR Parser Types

- Following LR parsers are in ascending order from left to right by their efficiency.
  - Least efficient parser is LR(0) and
  - Highest efficient parser is LALR(1)

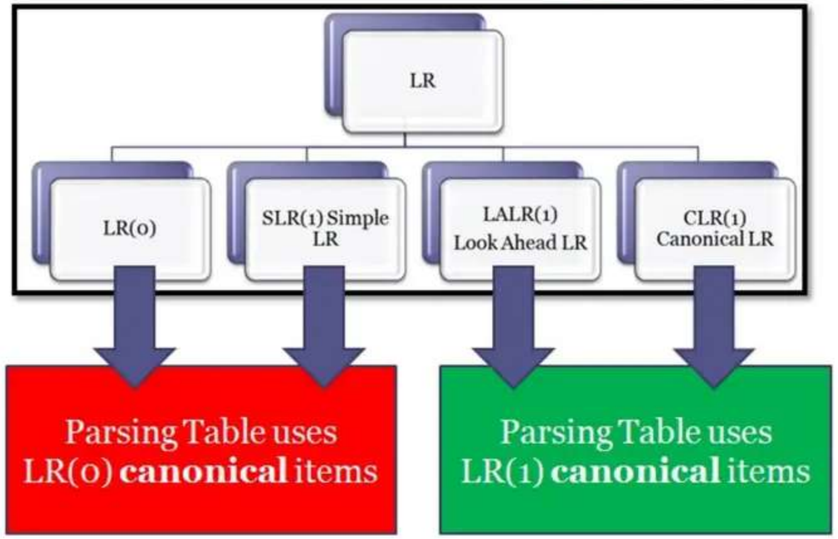


LR Parsers General Components



LR Parser

- “How to construct a parsing table?” is the only difference between all types of LR parsers.
- Parsing algorithm remain same for all types of LR parsers.
- In the construction of parsing table for:
  - LR(0) and SLR(1) parser we use canonical collection of **LR(0) items**.
  - LALR(1) and CLR(1) parser we use canonical collection of **LR(1) items**.



LR parser

- LR(0): Left to right scanning, **R**ight most derivation in reverse order.

- Simple LR / SLR(1) / SLR / LR(1)
- Lookahead LR / LALR(1) / LALR
- Canonical LR / CLR(1) /CLR
  - ONE is optional in LR parsers
  - But ZERO is mandatory to mentioned

Introduction to LR Parsing

- The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing;
- the "L" is for lef t-to-right scanning of the input ,
- the "R" for constructing a rightmost derivation in reverse, and
- the k for the number of input symbols of look ahead that are used in making parsing decisions.
- The cases k=0 or k=1 are of practical interest, and we shall only consider LR parsers with k <= 1 here.
- When (k) is **omitted**, k is assumed to be 1.

Why LR Parsers?

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR-parsing method is the most general non backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods.
3. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input .
4. LR grammars can describe more languages than LL grammars.

Drawback of LR

- The principal **drawback** of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar.

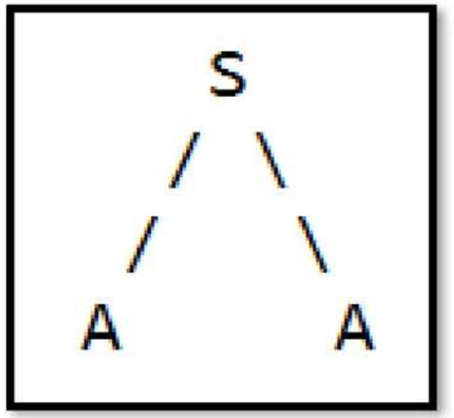
- A specialized tool , an LR parser generator, is needed.

- Fortunately, many such generators are available, and we shall discuss one of the most commonly used ones, Yacc,

What are LR items?

S ☾ . AA
This is called canonical collection of LR(0) item, when dot is there.

- Suppose
- S ☾ AA
- We use a DOT as shown in following production
- S ☾ . AA it means “not seen any thing”
- S ☾ A . A “Seen single A only”
- S ☾ AA . “Seen All”
- It actually shows reduction of AA . to S, i.e



LR(0) Parsing table Example

- In LL(1) parsing table construction we used FIRST and FOLLOW functions.
- While in LR(0) parsing table we will use **CLOSURE** and **GOTO** functions.

LR(0) Parsing table Example

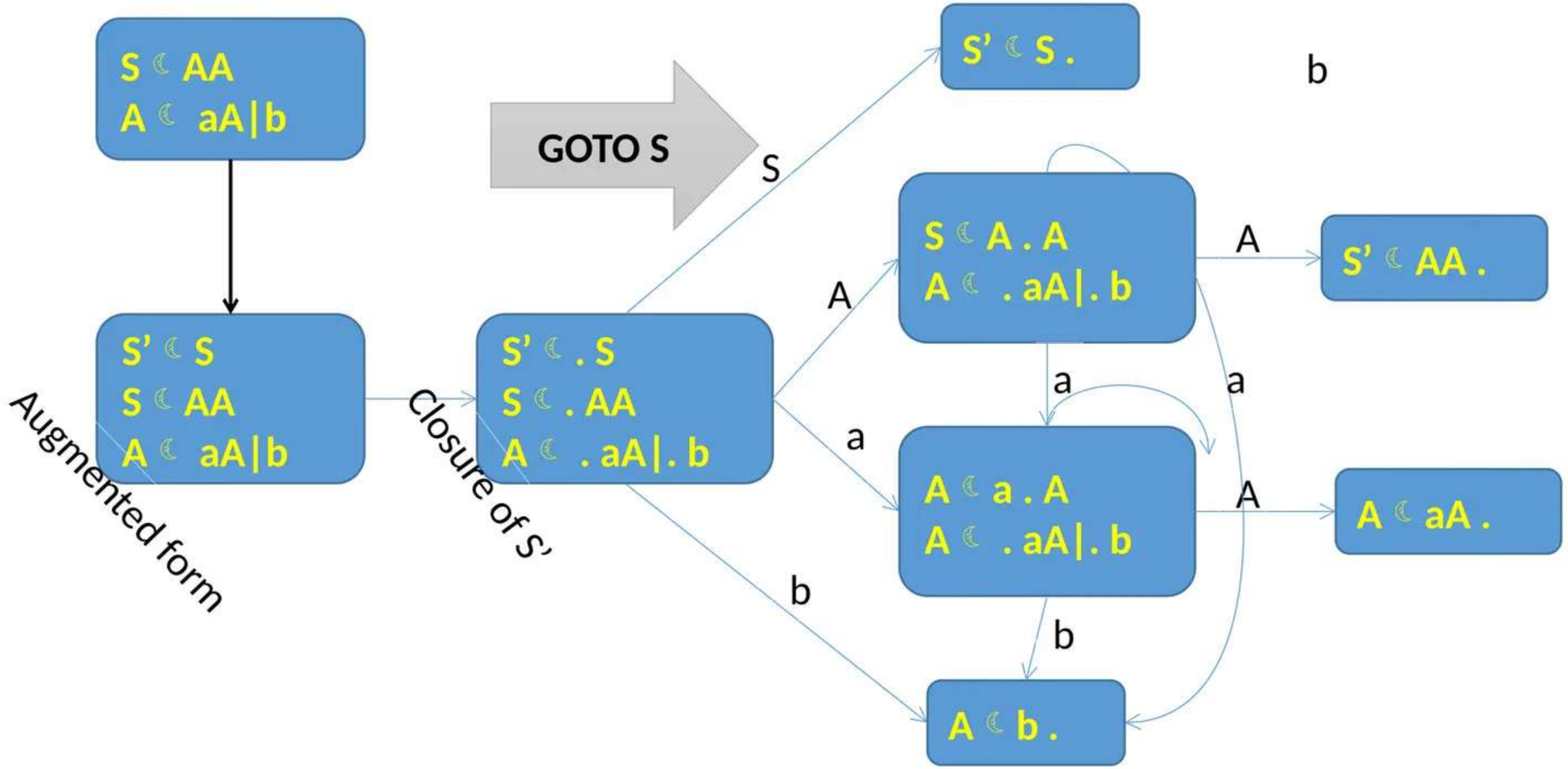
- S ☾ AA
- A ☾ aA|b
  - Convert into augmented grammar form
- S’ ☾ S
- S ☾ AA
- A ☾ aA|b
  - Perform CLOSURE
    - Closure of S’ is as follows
- S’ ☾ . S
- S ☾ . AA When we see dot on left of A, then add dot on all productions of A
- A ☾ . aA|.b

LR(0) Parsing table Example

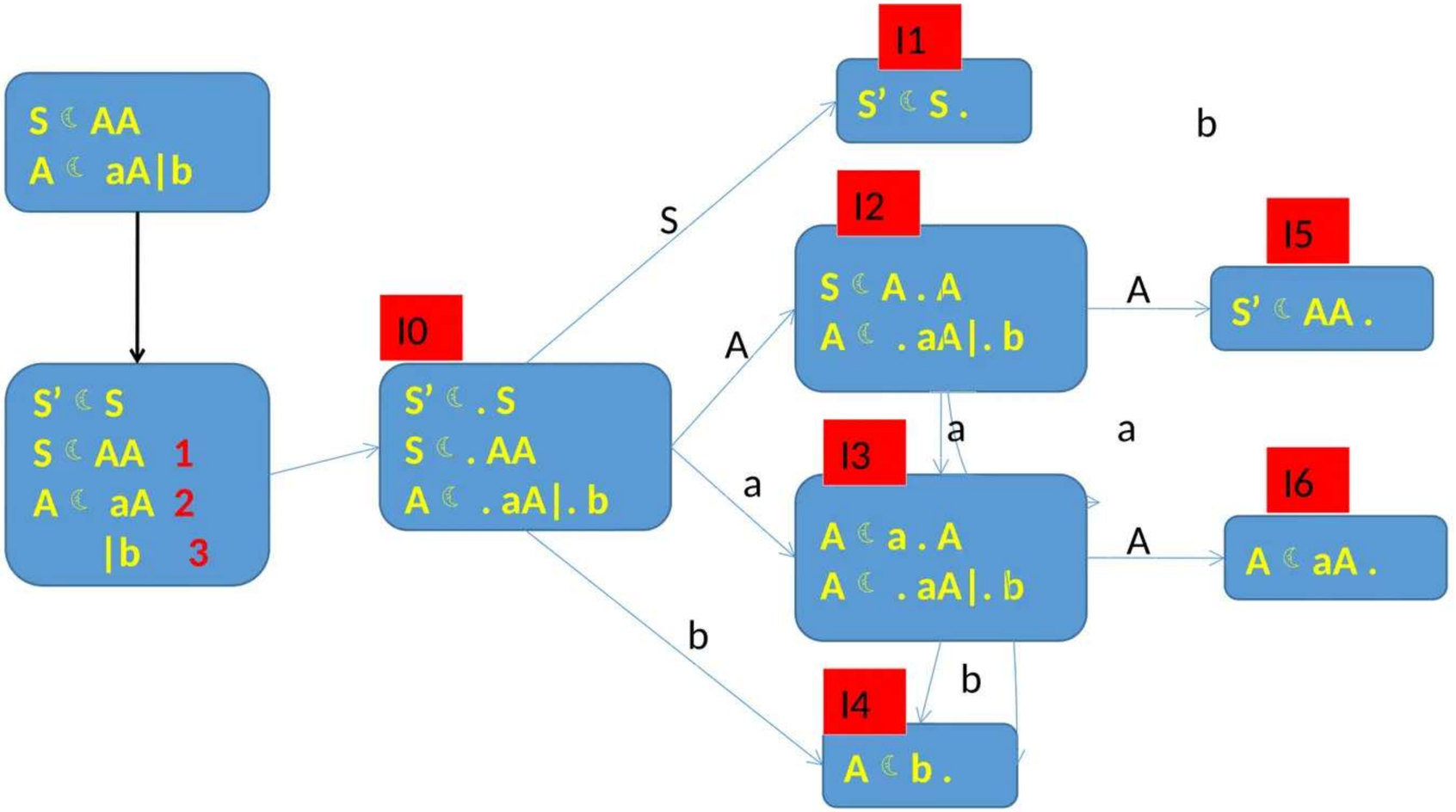
S’ ☾ S
S ☾ AA
A ☾ aA|b

- I_0
  - S’ ☾ . S
  - S ☾ . AA Dot is left of A, so that perform closure of A
  - A ☾ . aA|.b
- Perform GOTO
  - Shift dot one variable right for each terminal and nonterminal.
- I1 (GOTO for S)
  - S’ ☾ S. Its complete we have seen all productions
- I2 (GOTO for A)
  - S ☾ A . A Dot is left of A, so we perform closure of A
  - A ☾ a . A | .b
- And so on see next slide for complete solution.

LR(0) Parsing table Example



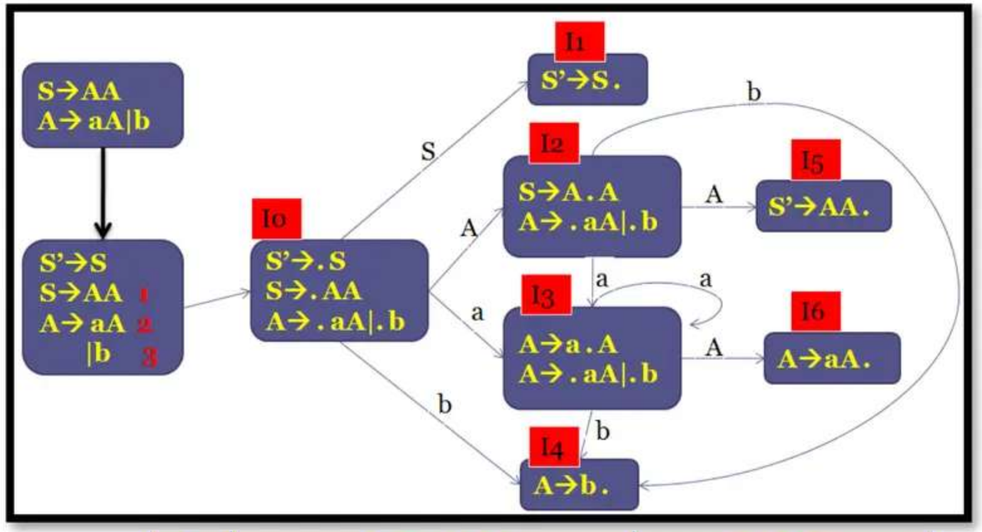
LR(0) Parsing table Example



- I0 to I6 are states called canonical collection of LR(0) items
- I1,I4,I5 and I6 are called completed items. i.e we have seen all productions and can be reduced.

LR(0) Parsing table Example

- Action part of table contains terminals
- GOTO part will contain Non-terminal
- S means Shift
- I0 to I6 are states



- I0 on S going to I1, so S contains 1 in table
- I0 on A going to I2, so A contains 1 in table

....

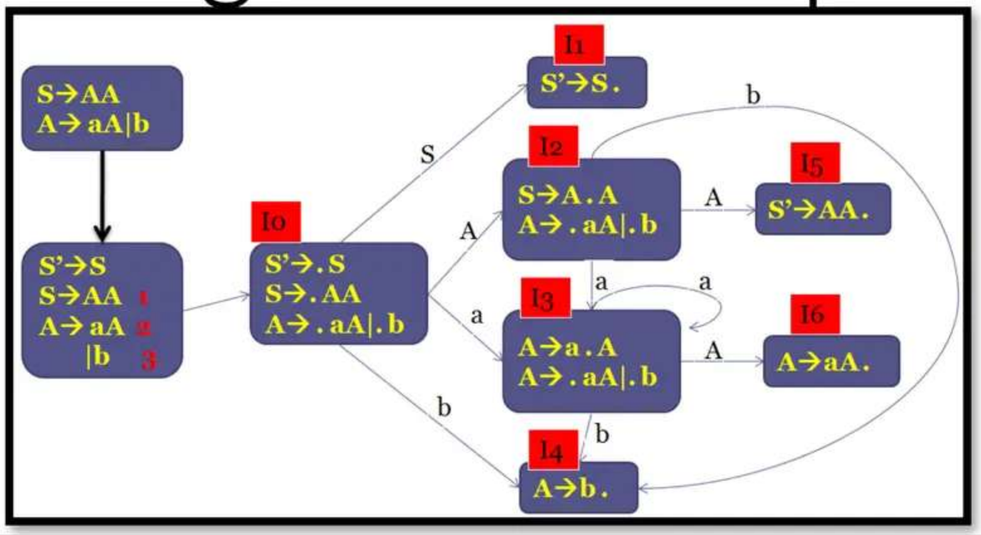
- I0 on a is going to I3, so a contains S3
- I1 on b is going to I4, so b contains S4

.....

I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1					
2	S3	S4		5	
3	S3	S4		6	
4					
5					
6					

- This table is common for all types of parser like LR(0), SLR(1), LALR(1),CLR(1)
- Table can vary for final items(having dot on right hand side) like I1,I4,I5,I6.
- Now assign numbers 1,2,3.. to the productions

LR(0) Parsing table Example



- Action part of table contains terminals
- GOTO part will contain Non-terminal
- S means Shift
- I0 to I6 are states

- I0 on S going to I1, so S contains 1 in table
- I0 on A going to I2, so A contains 1 in table

....

- I0 on a is going to I3, so a contains S3
- I1 on b is going to I4, so b contains S4

.....

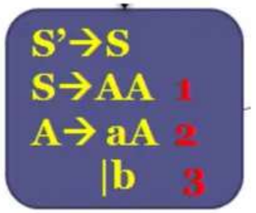
- I1 is augmented production, that means we have seen all. So I1 on $ is **accept** in table

→I4 is b. and it is 3rd production so 4th row uses r3 for a,b,$.

→I5 is AA. And its 1st production so 5th row uses r1 for a,b,$.....

I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

How to use Table in parsing



- Example:
- Reduce:
- INPUT: **aabb$**
- **STACK:**

$							

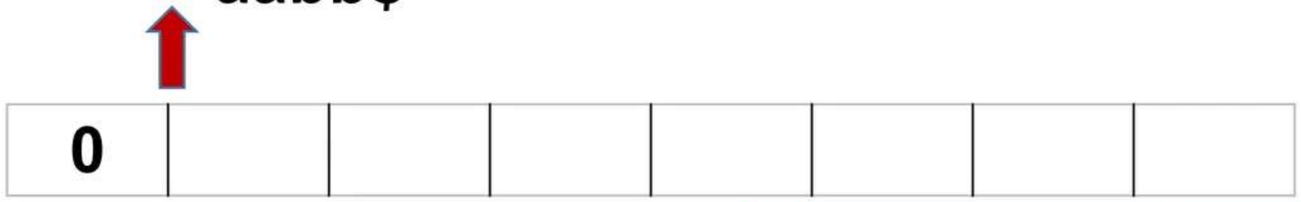
I	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

How to use Table in parsing

- Example:

S'→S
S→AA 1
A→aA 2
|b 3

- Reduce:
- INPUT: **aabb$**



- **STACK:**

•Assign numbers to each production
•Add $ to input string
•Show pointer in input on first letter
•Zero state will be on top of the stack

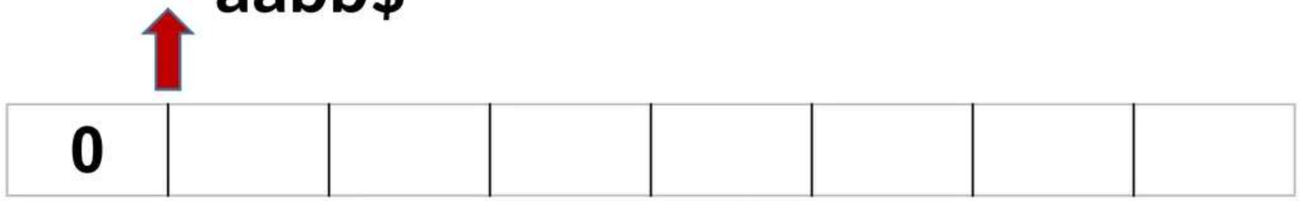
I	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

How to use Table in parsing

- Example:
- Reduce:
- INPUT:
- **STACK:**

S'→S
S→AA 1
A→ aA 2
|b 3

aabb$



- Top of the stack is **Zero** and input is **a**
- See **zero on a** in table, it is S3
- So PUSH a and 3,
- Move input pointer one ahead

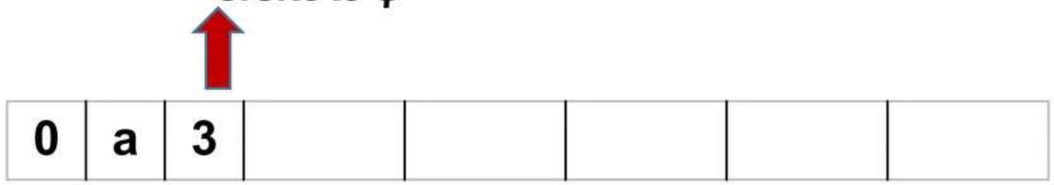
I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

How to use Table in parsing

- Example:

S'→S
S→AA 1
A→aA 2
|b 3

- Reduce:
- INPUT: **aabb$**

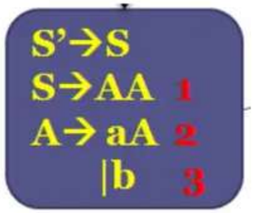


- **STACK:**

- Top of the stack is **3** and input is **a**
- See **3 on a** in table, it is S3
- So PUSH a and 3,
- Move input pointer one ahead

I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:



- Reduce:
- INPUT: **aabb$**

- **STACK:**

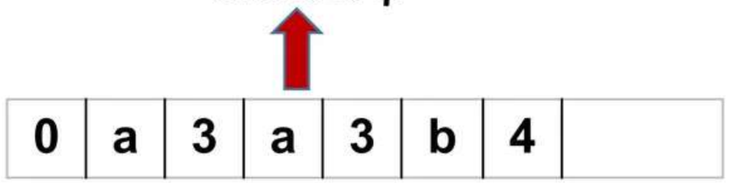
- Top of the stack is **3** and input is **b**
- See **3 on b** in table, it is S4
- So PUSH b and 4,
- Move input pointer one ahead

I	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	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3

- Reduce:
- INPUT: **aabb$**



- **STACK:**

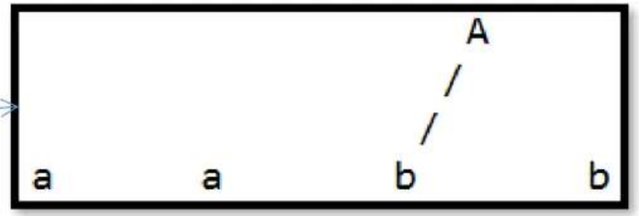
•Top of the stack is **4** and input is **b**
•See **4 on b** in table, it is r3
•R3 means see 3rd production i.e A ☾ b
•Size of production A ☾ |b| is one element so POP two/double elements.
•POP 4,b and PUSH A
•Reduce previous input that is aa**b**b to A
•Don't move input pointer one ahead

I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

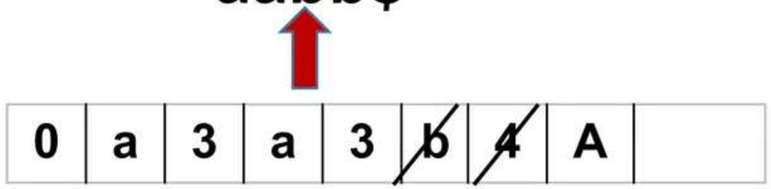
- Example:

S'→S
S→AA 1
A→aA 2
|b 3

- Reduce:



- INPUT: **aabb$**



- **STACK:**

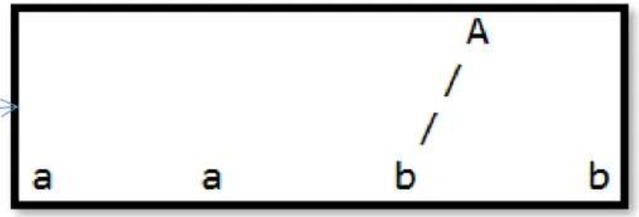
- We need state number for A so that
- See stack contains **3** and **A**
- See **3 on A** in table, it is 6
- PUSH 6
- Now 6 is the state number for A

I	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	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

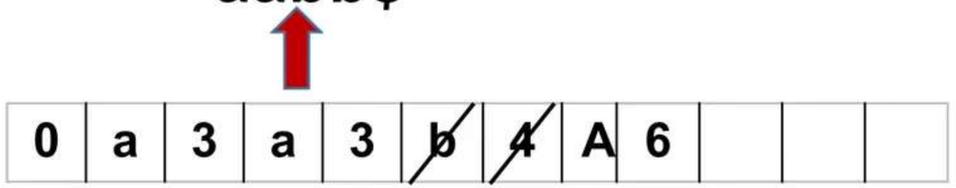
- Example:

S'→S
S→AA 1
A→aA 2
|b 3

- Reduce:



- INPUT: **aabb$**



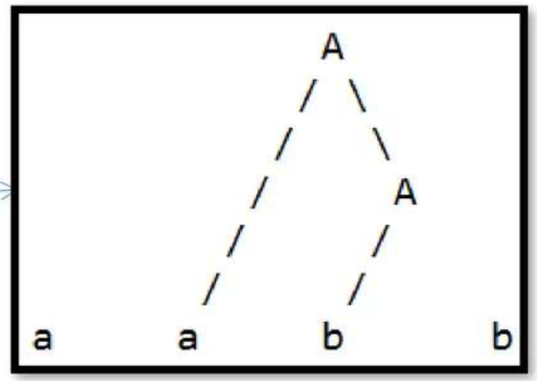
- **STACK:**

•Top of the stack is **6** and input is **b**
•See **6 on b** in table, it is r2
•r2 means see 2nd production i.e A ☾ aA
•Size of production A ☾ |aA| is two elements so POP four/double elements.
•POP 6,A,3,a and PUSH A
•Reduce (i.e A ☾ aA)previous input that is a**a**bb to A
•Don't move input pointer one ahead

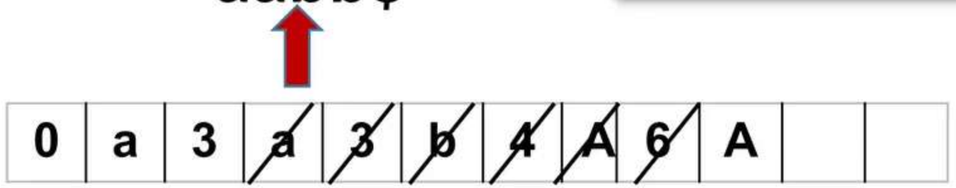
I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**



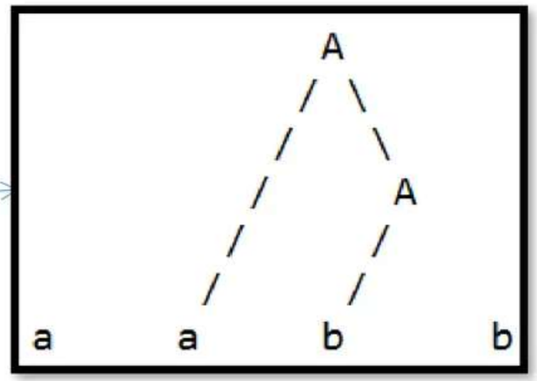
- **STACK:**

•We need state number for A so that
•See stack contains **A** and **3**
•See **3 on A** in table, it is 6
•PUSH 6
•6 is state number for A

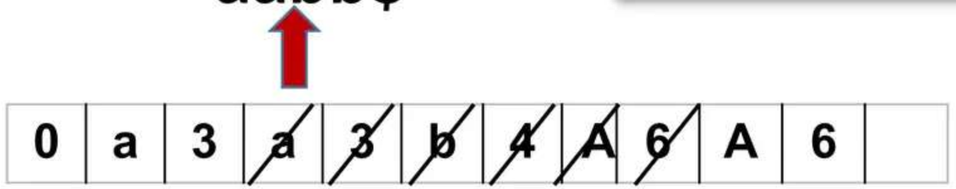
I	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**



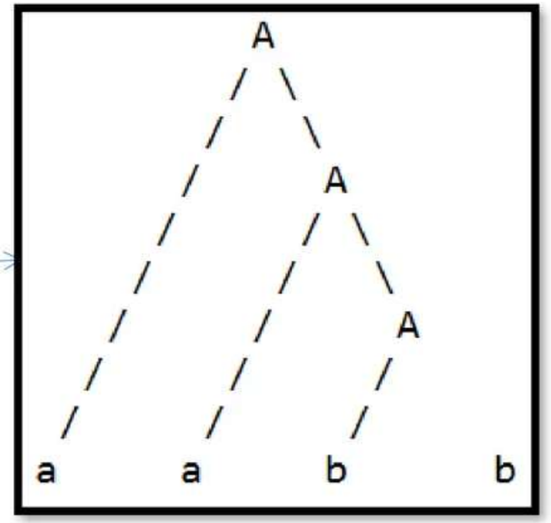
- **STACK:**

•Top of the stack is **6** and input is **b**
•See **6 on b** in table, it is r2
•r2 means see 2nd production i.e A ☾ aA
•Size of production A ☾ |aA| is two elements so POP four/double elements.
•POP 6,A,3,a and PUSH A
•Reduce (i.e A ☾ aA) previous input that is a**a**bb to A
•Don't move input pointer one ahead

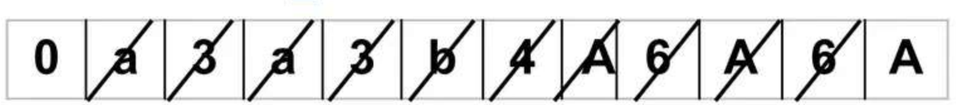
I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**



- **STACK:**

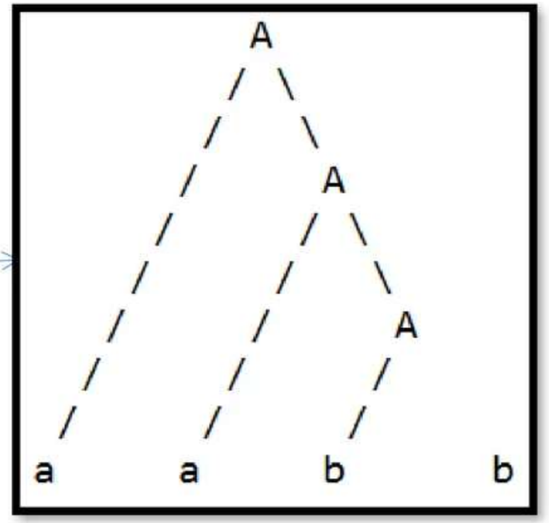
- We need state number for A so that
- See stack contains **A** and **0**
- See **0 on A** in table, it is 2
- PUSH 2
- 2 is state number for A

I	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	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

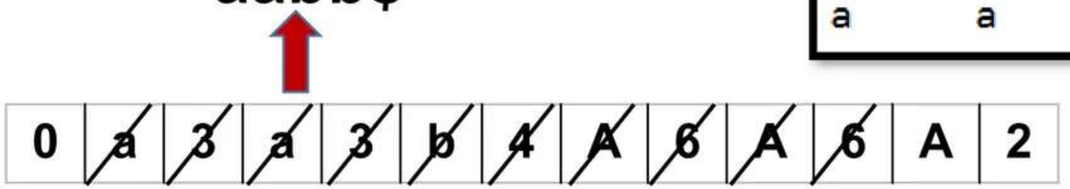
- Example:

S'→S
S→AA 1
A→aA 2
|b 3

- Reduce:



- INPUT: **aabb$**



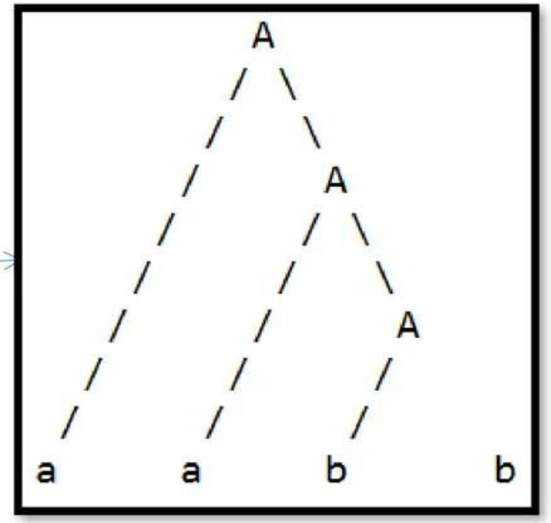
- **STACK:**

•Top of the stack is **2** and input is **b**
•See **2 on b** in table, it is S4
•So PUSH b and 4,
•Move input pointer one ahead

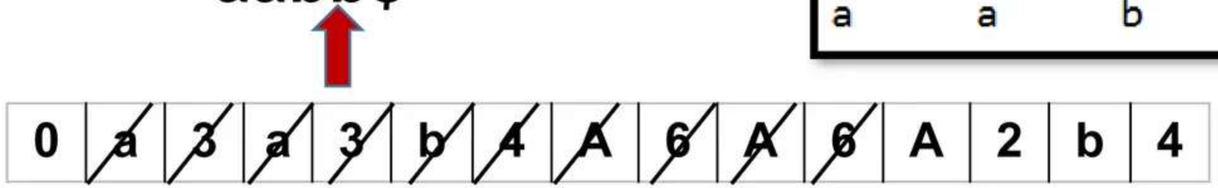
I	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**



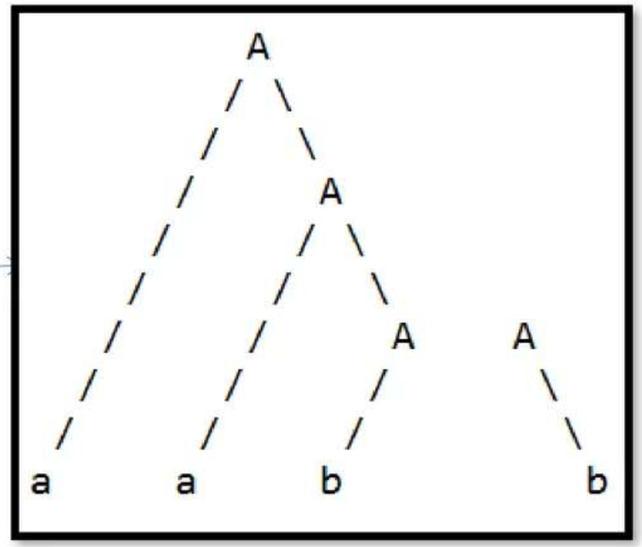
- **STACK:**

•Top of the stack is **4** and input is **$**
•See **4 on $** in table, it is r3
•r3 means see 3rd production i.e A ☾ b
•Size of production A ☾ |b| is one element so POP two/double elements.
•POP 4,b and PUSH A
•Reduce (i.e A ☾ b) previous input that is aab**b** to A
•Don't move input pointer one ahead

I	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	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**

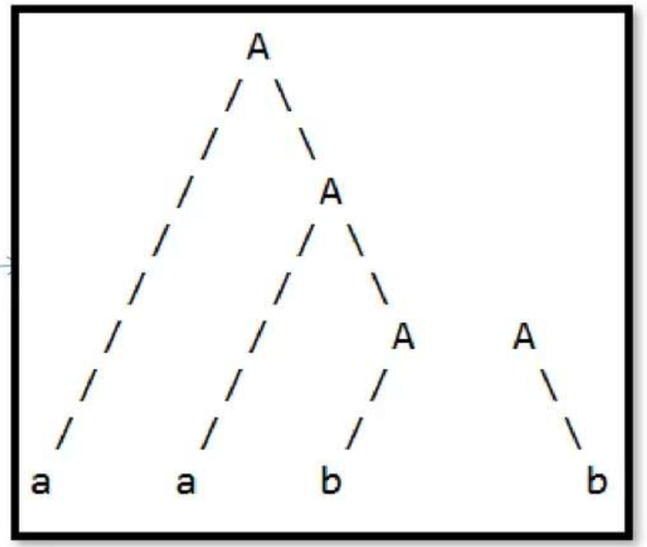
- **STACK:**

•See stack contains **A** and **2**
•See **2 on A** in table, it is 5
•PUSH 5

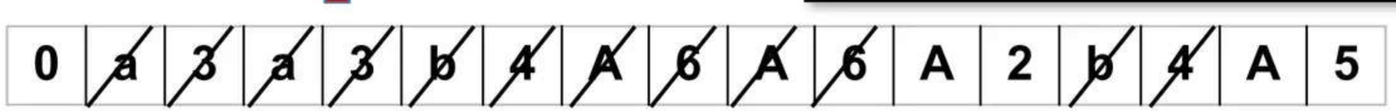
I	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**



- **STACK:**

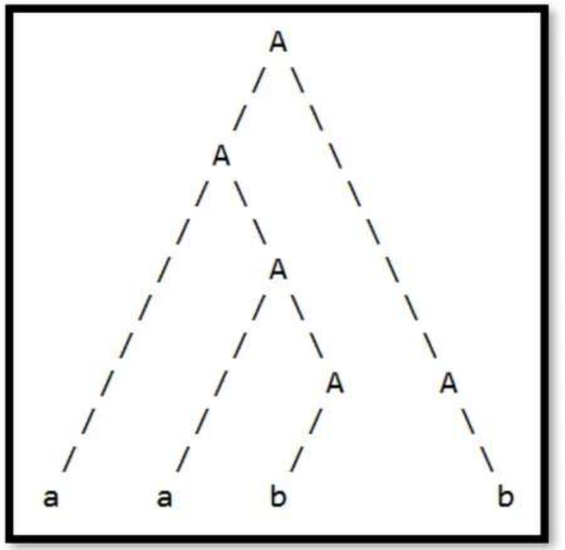
•Top of the stack is **5** and input is **$**
•See **5 on $** in table, it is **r1**
•r1 means see 1st production i.e S ☾ AA
•Size of production S ☾ |AA| is two elements so POP four/double elements.
•POP 5,A,2,A and PUSH S
•Reduce (i.e S ☾ AA)
•Don't move input pointer one ahead

I	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→ aA 2
|b 3

- Reduce:



- INPUT: **aabb$**

- **STACK:**

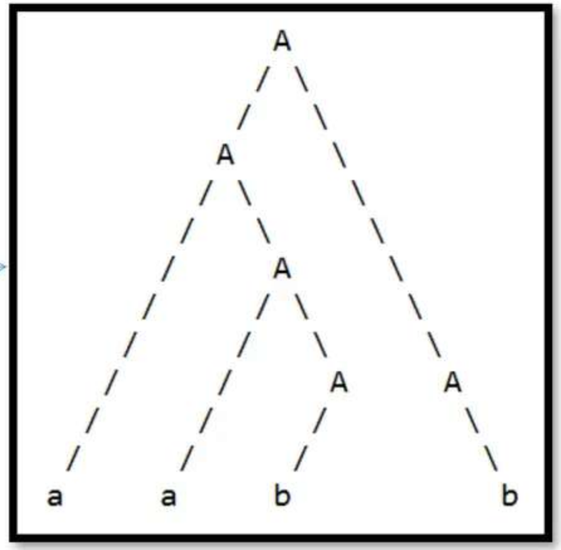
0	a	3	a	3	b	4	A	6	A	6	A	2	b	4	A	5	S

- See stack contains **S** and **0**
- See **0 on S** in table, it is 1
- PUSH 1

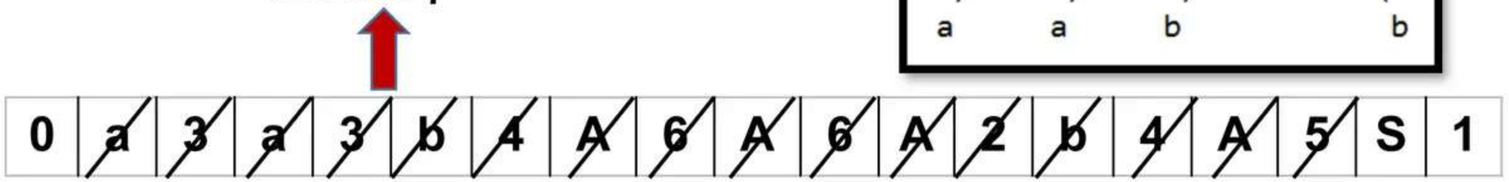
I	ACTION			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- Example:

S'→S
S→AA 1
A→aA 2
|b 3



- Reduce:
- INPUT: **aabb$**

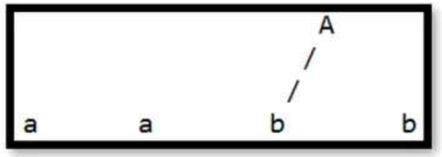


- STACK:

•See **1 on $** in table, it is **accepted**

I	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	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

What is meaning of LR(0)



- We have zero look ahead
- It doest mean that we are not looking any symbol at all.
- Lets understand by given examples for recent parse table
- The input was **aabb$**
- Here while reading last last **b** (i.e aab**b**$)**,** we reduced second last **b to A**. (i.e aa**b**b$)
- The 4th row in table is filled by r3 (three times)
  - It means that if you read any letter a,b or $ you can reduce second last variable, without knowing what is in the look ahead. That's why called zero look ahead
  - Due to r3 repetitions it may be difficult for LR(0) parser to detect errors more efficiently.
  - LR(0) is less efficient than SLR(1)
  - In the table see
    - I2 and $ is a blank entry that shows an error
    - I3 and $ is also an error

CW: Is LR(0)?

- Find Canonical collection of item
- Draw LR(0) parsing table.
- Show word dbbc is acceptable.
  - S ☾ dA|aB
  - A ☾ bA|c
  - B ☾ bB|c

LR Parsing Algorithm

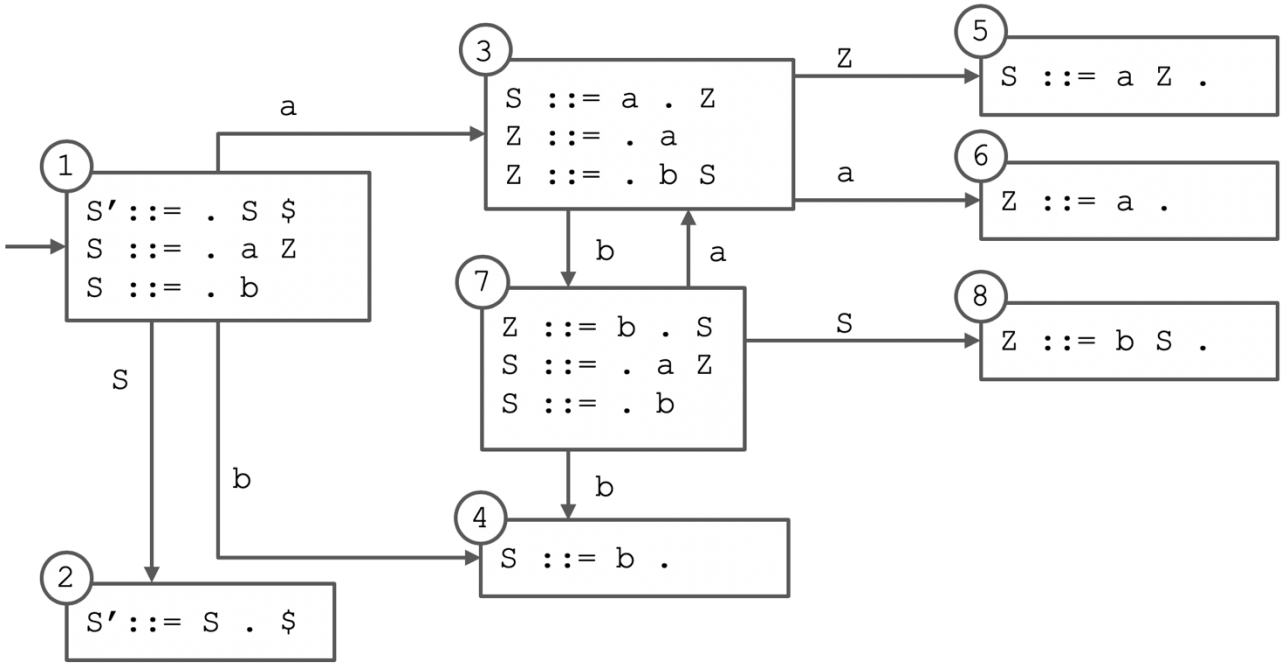
```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A → β then begin
        pop 2*|β| symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A → β
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

CSE 401 - LR Parsing Worksheet Sample Solutions - Week 3

Problem 1

a. Using the technique shown in lecture, construct the LR(0) state machine for this grammar. Remember to show the set of items that correspond to each state, including both any initial items and the resulting closure.

```
0. S′::= S $
1. S ::= a Z
2. S ::= b
3. Z ::= a
4. Z ::= b S
```



b. Based on your state machine, build the corresponding LR(0) parse table. Start by filling in the ACTION and GOTO headers with the grammar's terminals and non-terminals, respectively, then give each state in your state machine a number and use it to fill out one row of the table.

STATE	ACTION			GOTO	
	a	**b**	**$**	**S**	**Z**
1	s3	s4		g2	
2			acc		
3	s6	s7			g5
4	r2	r2	r2		
5	r1	r1	r1		
6	r3	r3	r3		
7	s3	s4		g8	
8	r4	r4	r4		

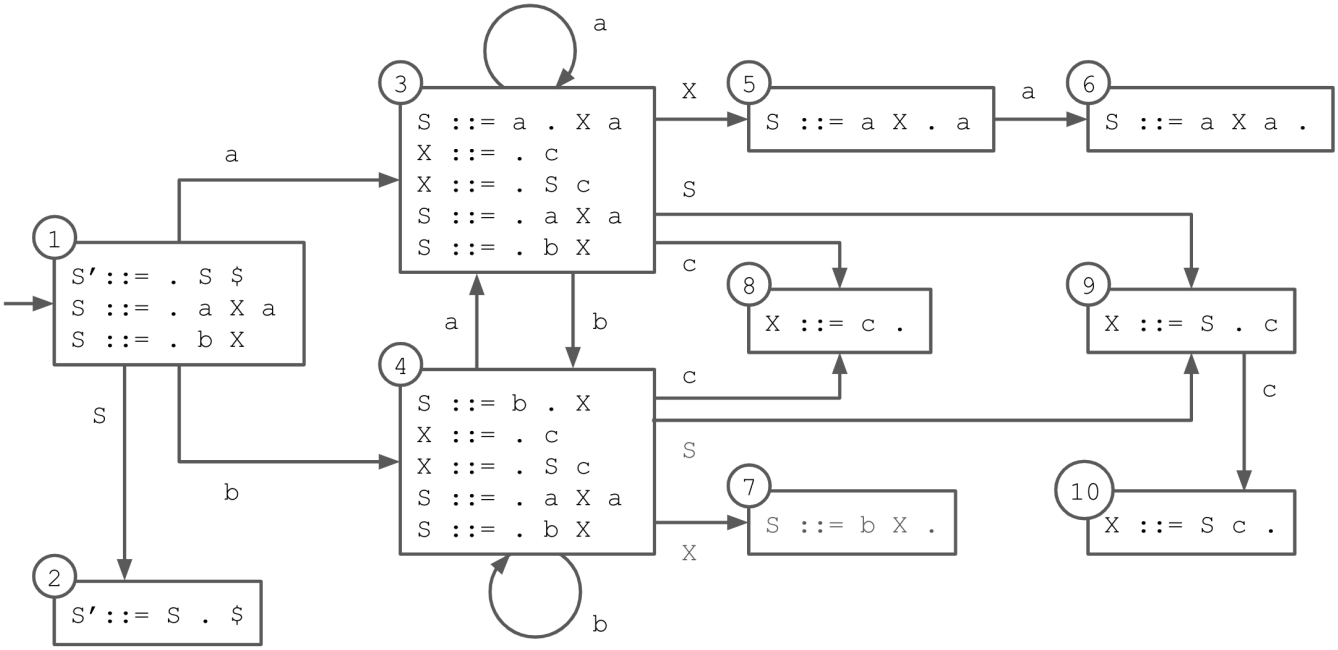
c. Finally, use your table to parse the provided input, keeping track of both the stack and the remaining input at each step in a table such as the one below. For clarity, you will probably find it easiest to push both the current state and the corresponding symbol onto the stack at each point, although a real parser would only need to keep track of the states. The initial state (s_1) has already been inserted onto the stack for you.

STACK	**INPUT**	**ACTION**
$0 s1	**a b a b b $**	SHIFT
$0 s1 a s3	b a b b $	SHIFT
$0 s1 a s3 b s7	a b b $	SHIFT
$0 s1 a s3 b s7 a s3	b b $	SHIFT
$0 s1 a s3 b s7 a s3 b s7	b $	SHIFT
$0 s1 a s3 b s7 a s3 b s7 b s4	$	REDUCE
$0 s1 a s3 b s7 a s3 b s7 S s8	$	REDUCE
$0 s1 a s3 b s7 a s3 Z s5	$	REDUCE
$0 s1 a s3 b s7 S s8	$	REDUCE
$0 s1 a s3 Z s5	$	REDUCE
$0 s1 S s2	$	ACCEPT

Problem 2

a. Using the technique shown in lecture, construct the LR(0) state machine for this grammar. Remember to show the set of items that correspond to each state, including both any initial items and the resulting closure.

```
0. S'::= S $
1. S ::= a X a
2. S ::= b X
3. X ::= c
4. X ::= S c
```



b. Based on your state machine, build the corresponding LR(0) parse table. Start by filling in the ACTION and GOTO headers with the grammar's terminals and non-terminals, respectively, then give each state in your state machine a number and use it to fill out one row of the table.

STATE	ACTION				GOTO	
	a	b	c	$	S	X
1	s3	s4			g2	
2				acc		
3	s3	s4	s8		g9	g5
4	s3	s4	s8		g9	g7
5	s6					
6	r2	r2	r2	r2		
7	r3	r3	r3	r3		
8	r4	r4	r4	r4		
9			s10			
10	r5	r5	r5	r5		

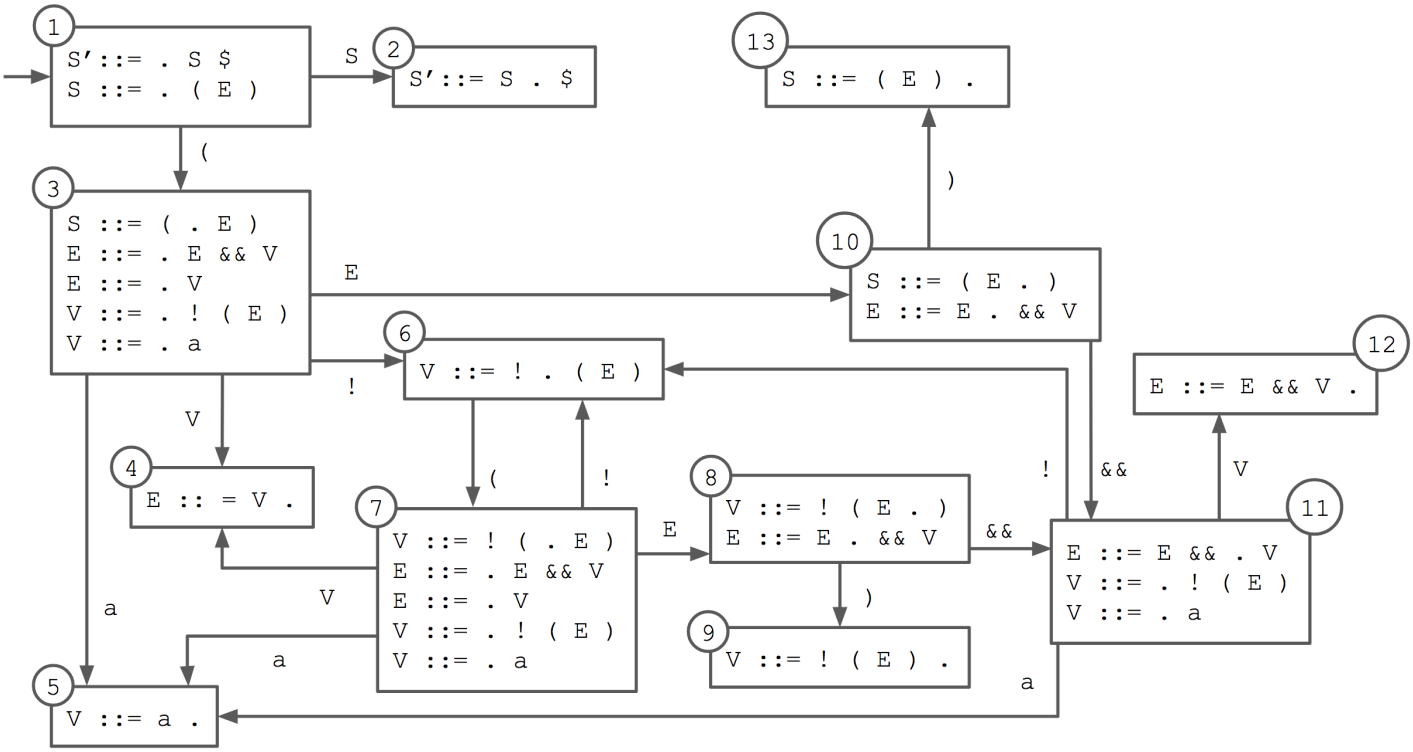
c. Finally, use your table to parse the provided input, keeping track of both the stack and the remaining input at each step in a table such as the one below. For clarity, you will probably find it easiest to push both the current state and the corresponding symbol onto the stack at each point, although a real parser would only need to keep track of the states. The initial state (s_1) has already been pushed onto the stack for you.

STACK	**INPUT**	**ACTION**
$ 0 s_1	**a b c c a $**	SHIFT
$ 0 s_1 a s_3	b c c a $	SHIFT
$ 0 s_1 a s_3 b s_4	c c a $	SHIFT
$ 0 s_1 a s_3 b s_4 c s_8	c a $	REDUCE
$ 0 s_1 a s_3 b s_4 X s_7	c a $	REDUCE
$ 0 s_1 a s_3 S s_9	c a $	SHIFT
$ 0 s_1 a s_3 S s_9 c s_{10}	a $	REDUCE
$ 0 s_1 a s_3 X s_5	a $	SHIFT
$ 0 s_1 a s_3 X s_5 a s_6	$	REDUCE
$ 0 s_1 S s_2	$	ACCEPT

Problem 2

a. Using the technique shown in lecture, construct the LR(0) state machine for this grammar. Remember to show the set of items that correspond to each state, including both any initial items and the resulting closure.

```
0. S'::= S $
1. S ::= ( E )
2. E ::= E && V
3. E ::= V
4. V ::= ! ( E )
5. V ::= a
```



b. Based on your state machine, build the corresponding LR(0) parse table. Start by filling in the ACTION and GOTO headers with the grammar's terminals and non-terminals, respectively, then give each state in your state machine a number and use it to fill out one row of the table.

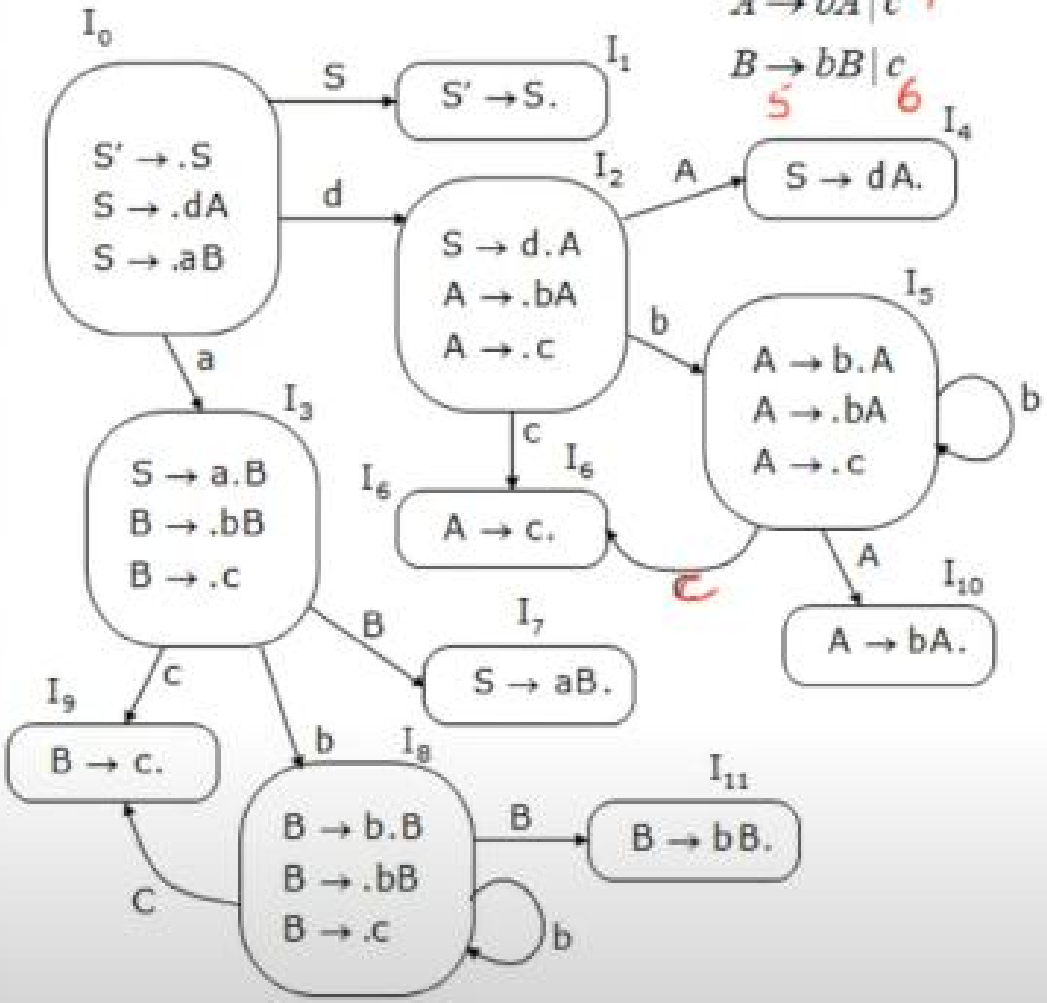
STATE	ACTION						GOTO		
	(	)	&&	!	a	$	S	E	V
1	s3						g2		
2						acc			
3				s6	s5			g10	g4
4	r4	r4	r4	r4	r4	r4			
5	r6	r6	r6	r6	r6	r6			
6	s7								
7				s6	s5			g8	g4
8		s9	s11						
9	r5	r5	r5	r5	r5	r5			
10		s13	s11						
11				s6	s5				g12
12	r3	r3	r3	r3	r3	r3			
13	r2	r2	r2	r2	r2	r2			

c. Finally, use your table to parse the provided input, keeping track of both the stack and the remaining input at each step in a table such as the one below. For clarity, you will probably find it easiest to push both the current state and the corresponding symbol onto the stack at each point, although a real parser would only need to keep track of the states. The initial state (s_1) has already been pushed onto the stack for you.

STACK	**INPUT**	**ACTION**
$ 0 s_1	**(! (a && a)) $**	SHIFT
$ 0 s_1 (s_3	! (a && a)) $	SHIFT
$ 0 s_1 (s_3 ! s_6	(a && a)) $	SHIFT
$ 0 s_1 (s_3 ! s_6 (s_7	a && a)) $	SHIFT
$ 0 s_1 (s_3 ! s_6 (s_7 a s_5	&& a)) $	REDUCE
$ 0 s_1 (s_3 ! s_6 (s_7 V s_4	&& a)) $	REDUCE
$ 0 s_1 (s_3 ! s_6 (s_7 E s_8	&& a)) $	SHIFT
$ 0 s_1 (s_3 ! s_6 (s_7 E s_8 && s_{11}	a)) $	SHIFT
$ 0 s_1 (s_3 ! s_6 (s_7 E s_8 && s_{11} a s_5	)) $	REDUCE
$ 0 s_1 (s_3 ! s_6 (s_7 E s_8 && s_{11} V s_{12}	)) $	REDUCE
$ 0 s_1 (s_3 ! s_6 (s_7 E s_8	)) $	SHIFT
$ 0 s_1 (s_3 ! s_6 (s_7 E s_8) s_9	) $	REDUCE
$ 0 s_1 (s_3 V s_4	) $	REDUCE
$ 0 s_1 (s_3 E s_{10}	) $	SHIFT
$ 0 s_1 (s_3 E s_{10}) s_{13}	$	REDUCE
$ 0 s_1 S s_2	$	ACCEPT

Construction of LR(0) Parsing table

$S' \to S_1$ 0
$S \to dA \mid aB$ 2
$A \to bA \mid c$ 3 4
$B \to bB \mid c$ 5 6



	Action					Goto		
	a	b	c	d	$	S	A	B
I_0	I_3			I_2		I_1		
I_1					acc			
I_2		I_5	I_6				I_4	
I_3		I_8	I_9				I_7	
I_4	r_1	r_1	r_1	r_1	r_1			
I_5		I_5	I_6				I_{10}	
I_6	r_4	r_4	r_4	r_4	r_4			
I_7	r_2	r_2	r_2	r_2	r_2			
I_8		I_8	I_9					I_{11}
I_9	r_6	r_6	r_6	r_6	r_6			
I_{10}	r_3	r_3	r_3	r_3	r_3			
I_{11}	r_5	r_5	r_5	r_5	r_5			