




# SYNTAX ANALYSIS

Bottom-up parsing



# Top-down vs. Bottom-up

- Top-down parsers start with the starting nonterminal and applies production rules until it generates the string that matches the input. In each stage, we have to decide which rule to expand.
- Bottom-up parsers start with the input string and works in reverse to find the production rules that must be used to derive the string.
- Top-down parsers use leftmost derivation while bottom-up parsers use rightmost derivation in the reverse order.

# Reductions

- We can think of bottom-up parsing as the process of “reducing” a string  $w$  to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

# Handles

- Bottom-up parsing performs a left-to-right scan of the input and constructs a rightmost derivation in reverse.
- “handle” is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	$F$	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
$T$	$T$	$E \rightarrow T$

# 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
\$	$w$ \$

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string of grammar symbols on top of the stack. It then reduces to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

# Shift-reduce parsing

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & ( E ) \mid \mathbf{id} \end{array}$$

STACK	INPUT	ACTION
\$	<b>id<sub>1</sub> * id<sub>2</sub> \$</b>	shift
\$ <b>id<sub>1</sub></b>	<b>* id<sub>2</sub> \$</b>	reduce by $F \rightarrow \mathbf{id}$
\$ $F$	<b>* id<sub>2</sub> \$</b>	reduce by $T \rightarrow F$
\$ $T$	<b>* id<sub>2</sub> \$</b>	shift
\$ $T *$	<b>id<sub>2</sub> \$</b>	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ $T$	\$	reduce by $E \rightarrow T$
\$ $E$	\$	accept

Configurations of a shift-reduce parser on input **id<sub>1</sub>\*id<sub>2</sub>**

# Conflicts 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 and also the next  $k$  input symbols, 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).

# Conflicts during shift-reduce parsing

- An ambiguous grammar can't be used in shift-reduce parsing.

<i>stmt</i>	→	<b>if</b> <i>expr</i> <b>then</b> <i>stmt</i>
		<b>if</b> <i>expr</i> <b>then</b> <i>stmt</i> <b>else</b> <i>stmt</i>
		<b>other</b>

- If we have a shift-reduce parser in configuration

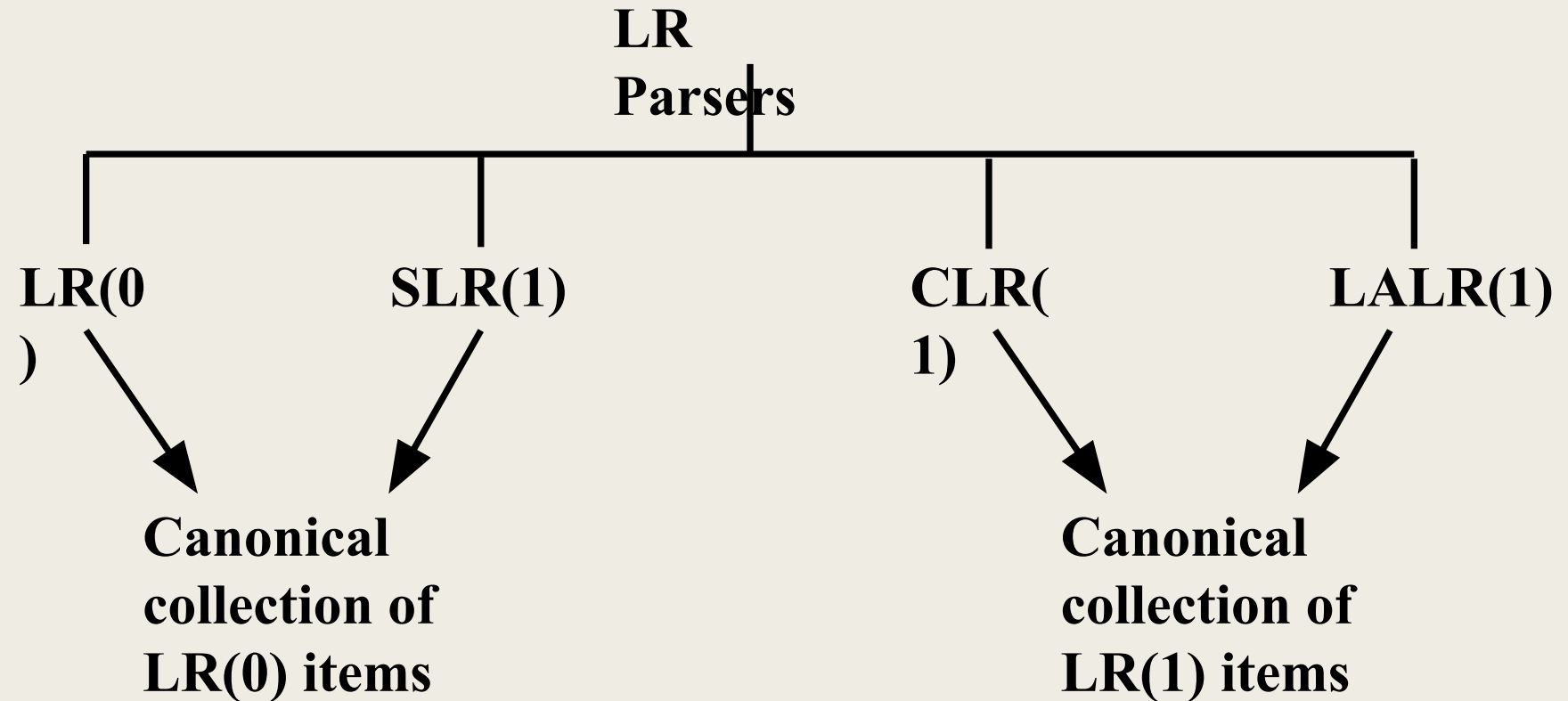
STACK
... <b>if</b> <i>expr</i> <b>then</b> <i>stmt</i>

INPUT
<b>else</b> ... \$

we cannot tell whether “**if** *expr* **then** *stmt*” is the handle or if we need to shift **else** and other symbols in the input to eventually find the handle “**if** *expr* **then** *stmt* **else** *stmt*.” We have a shift/reduce conflict.



# LR Parsing



# LR(0) parsing table

- To construct the LR(0) collection for a grammar, we define an augmented grammar and two functions CLOSURE and GOTO (just as we required to find FIRST and FOLLOW for the LL(1) parsing table).
- If  $G$  is a grammar with start symbol  $S$ , then  $G'$ , the augmented grammar for  $G$ , is  $G$  with a new start symbol  $S'$  and the production  $S' \rightarrow S$ . The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$ .

# LR(0) items

- An LR(0) item of a grammar  $G$  is a production of  $G$  with a dot at some position of the body. Thus, production  $A \rightarrow XYZ$  yields the four items

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

- The production  $A \rightarrow \epsilon$  generates only one item,  $A \rightarrow \cdot$ .
- Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. This helps us decide whether it's the correct time to reduce.

# CLOSURE and GOTO

- Consider the augmented grammar:

$$\begin{aligned}S' &\rightarrow S \\S &\rightarrow AA \\A &\rightarrow aA \mid b\end{aligned}$$

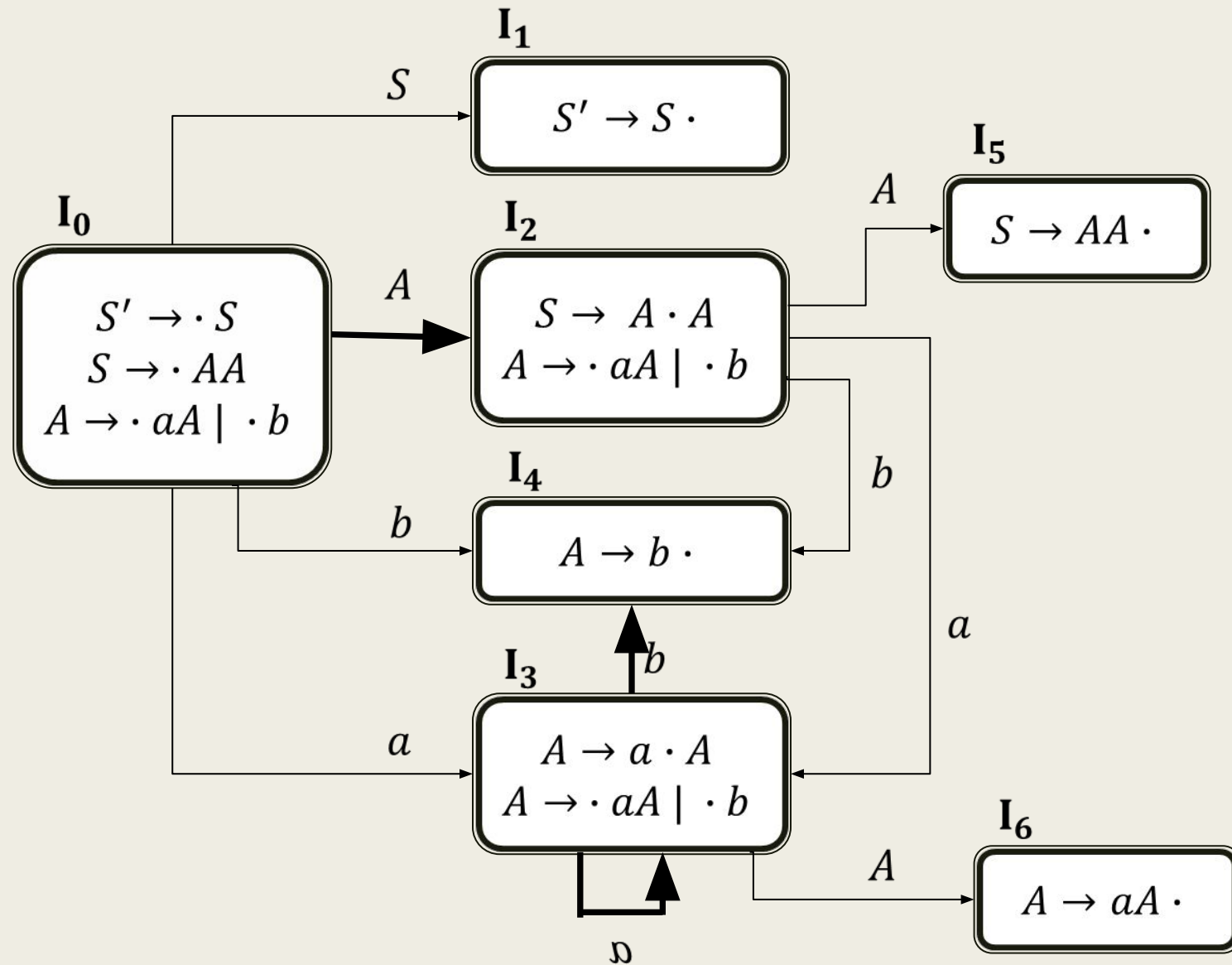
- $\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})$  results in:

$$\begin{aligned}S' &\rightarrow \cdot S \\S &\rightarrow \cdot AA \\A &\rightarrow \cdot aA \mid \cdot b\end{aligned}$$

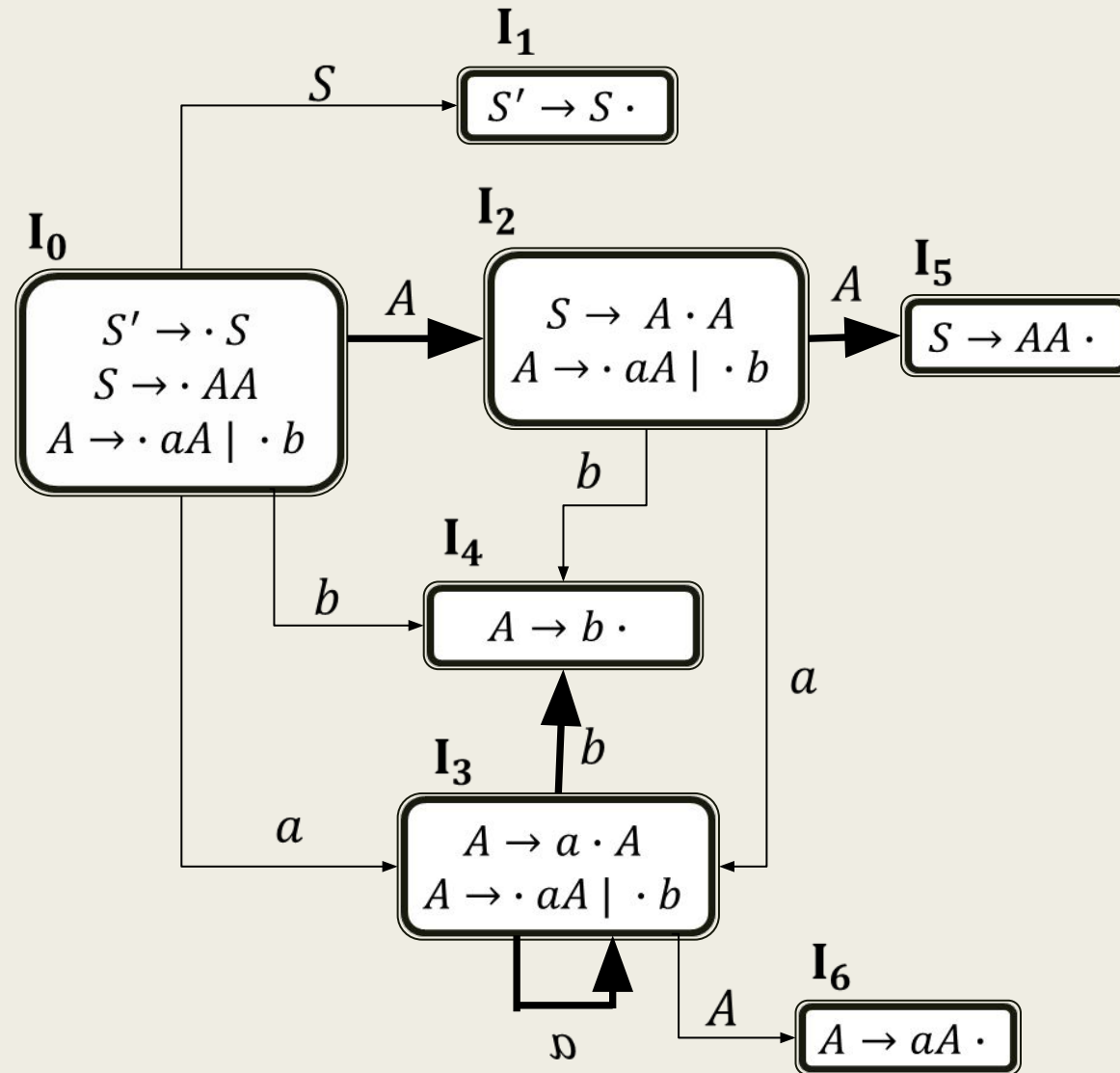
- $\text{GOTO}(\{[S \rightarrow \cdot AA], [A \rightarrow \cdot aA], [A \rightarrow \cdot b]\}, A)$  results in:

$$S \rightarrow A \cdot A$$

# LR(0) Automaton [Canonical collection of LR(0) items]



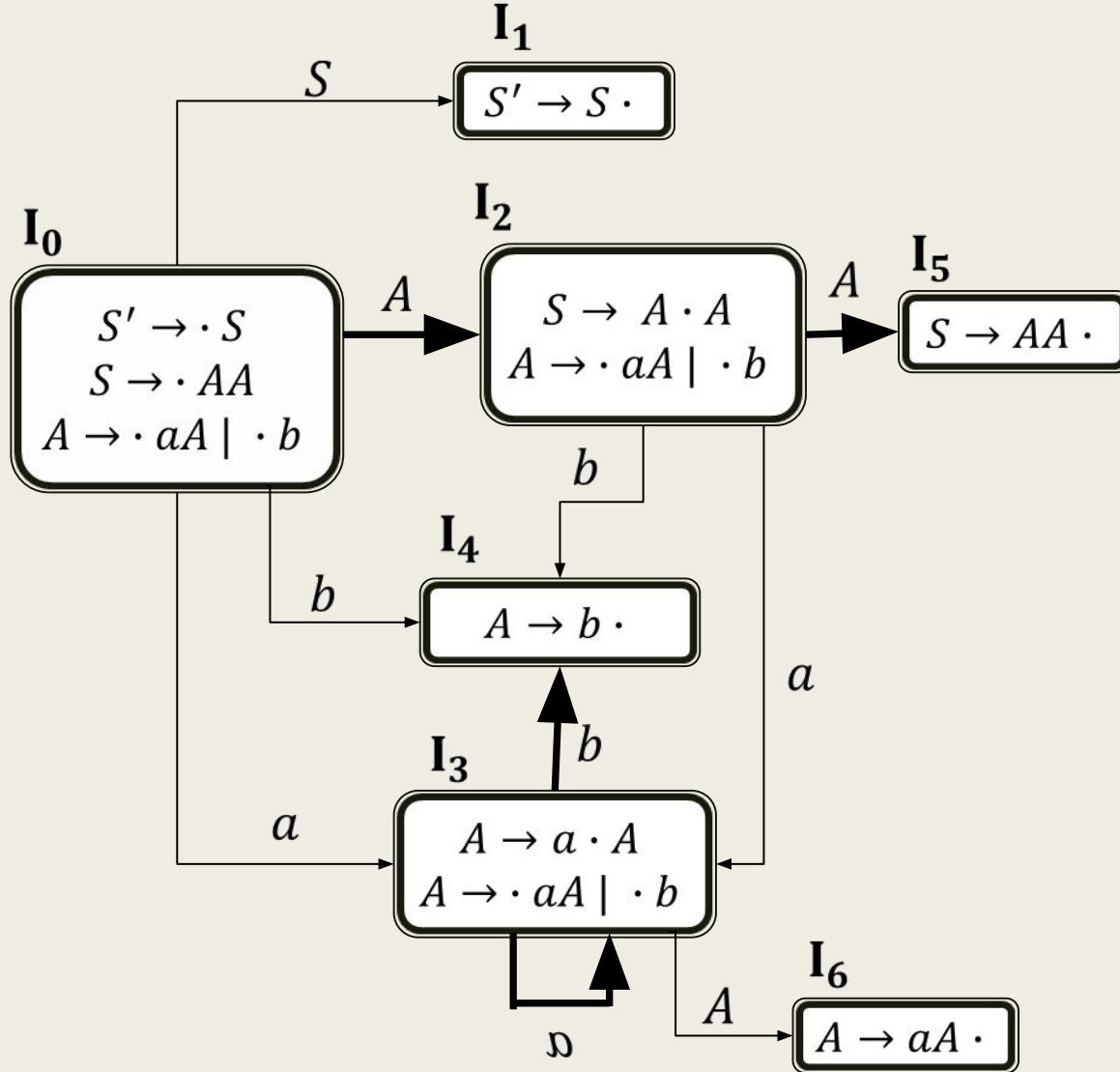
# LR(0) parsing table



LR(0) parser doesn't consider any lookahead, so any state where there is a rule with dot at the end means reduction is applied regardless of the symbol being currently seen.

	Actions			Go To	
0				1	2
1			accept		
2					5
3					6
4					
5					
6					

# SLR(1) parsing table



Don't place the reduction entries with the entire row, take the rule corresponding to the state and place the reduction entries only in the FOLLOW of the left side.

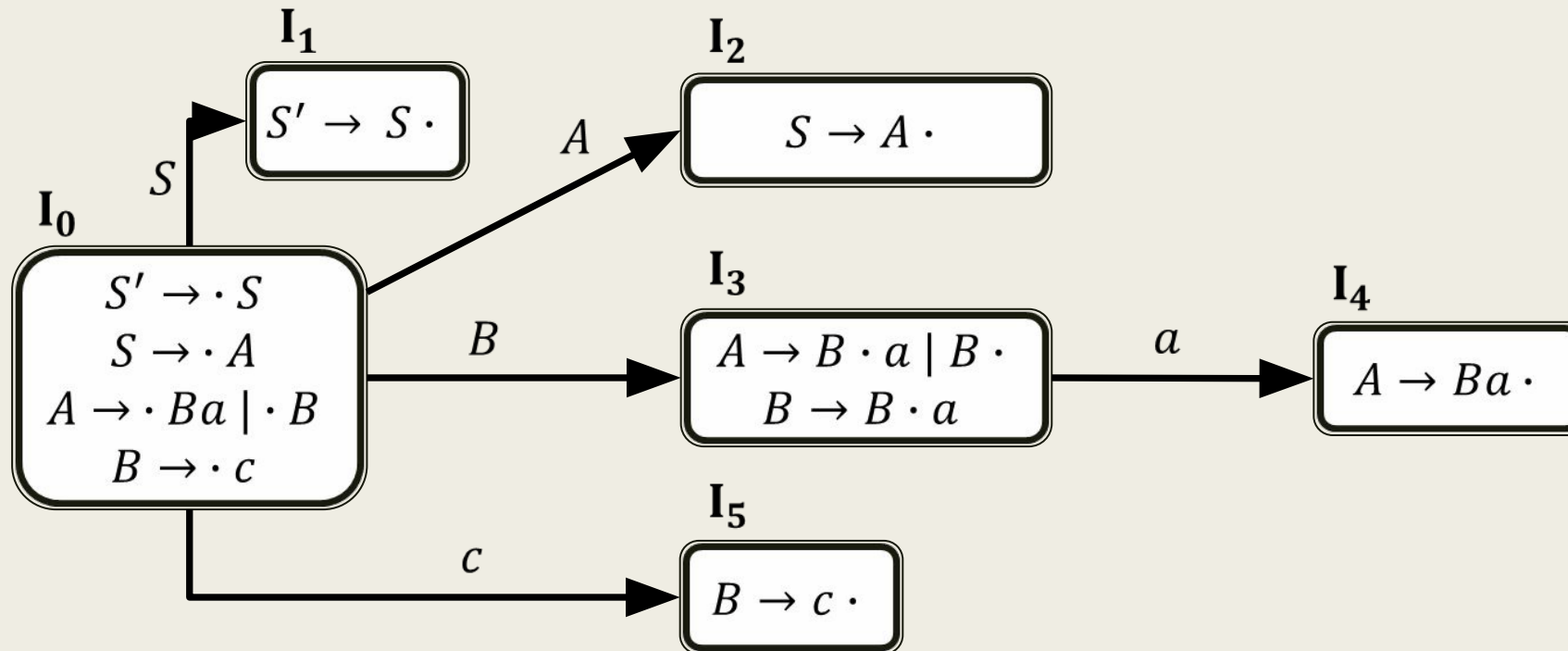
	Actions			Go To	
0				1	2
1			accept		
2					5
3					6
4					
5					
6					

# LR(0) vs. SLR(1) [Shift-reduce conflict]

- Let's consider the grammar:

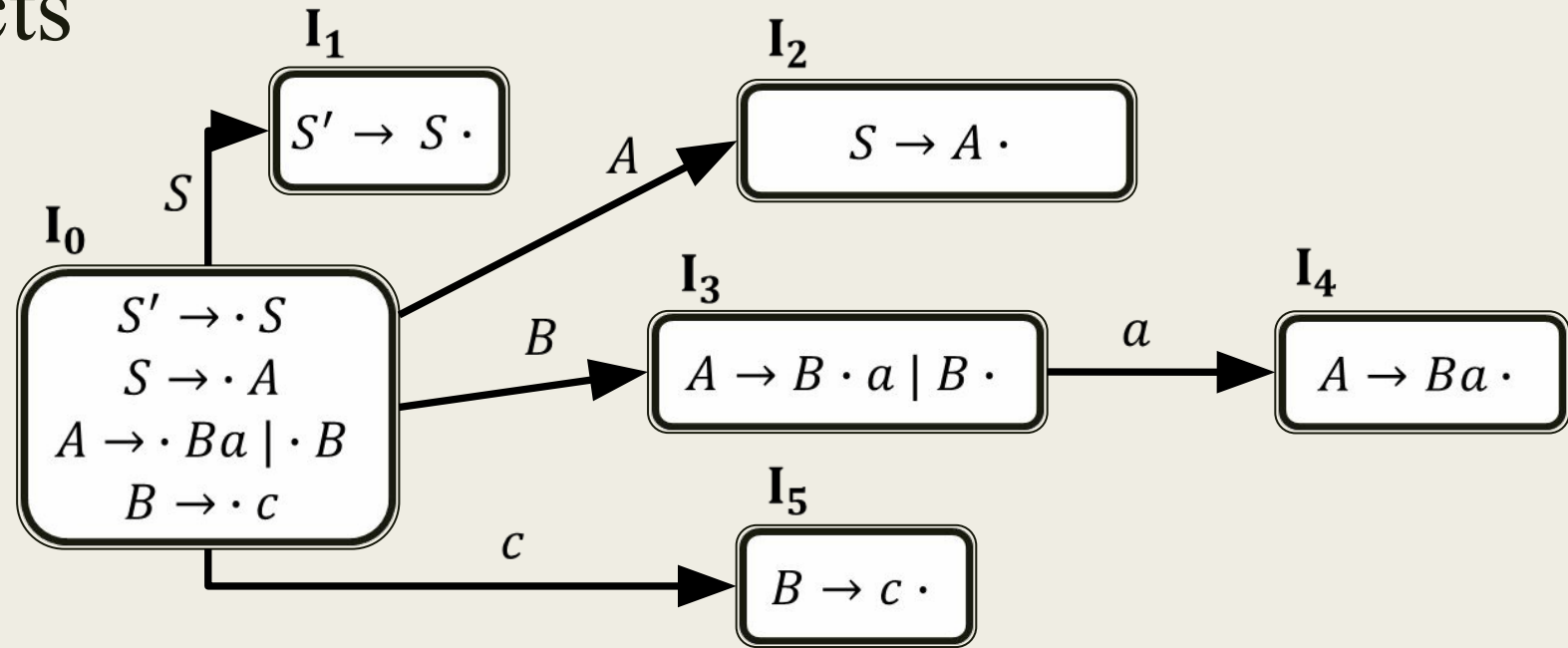
$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow Ba \mid B \\ B &\rightarrow c \end{aligned}$$

- The LR(0) automaton for the grammar is:





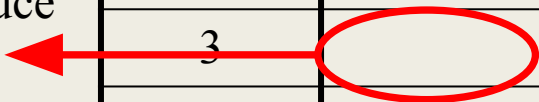
# Shift-reduce conflicts



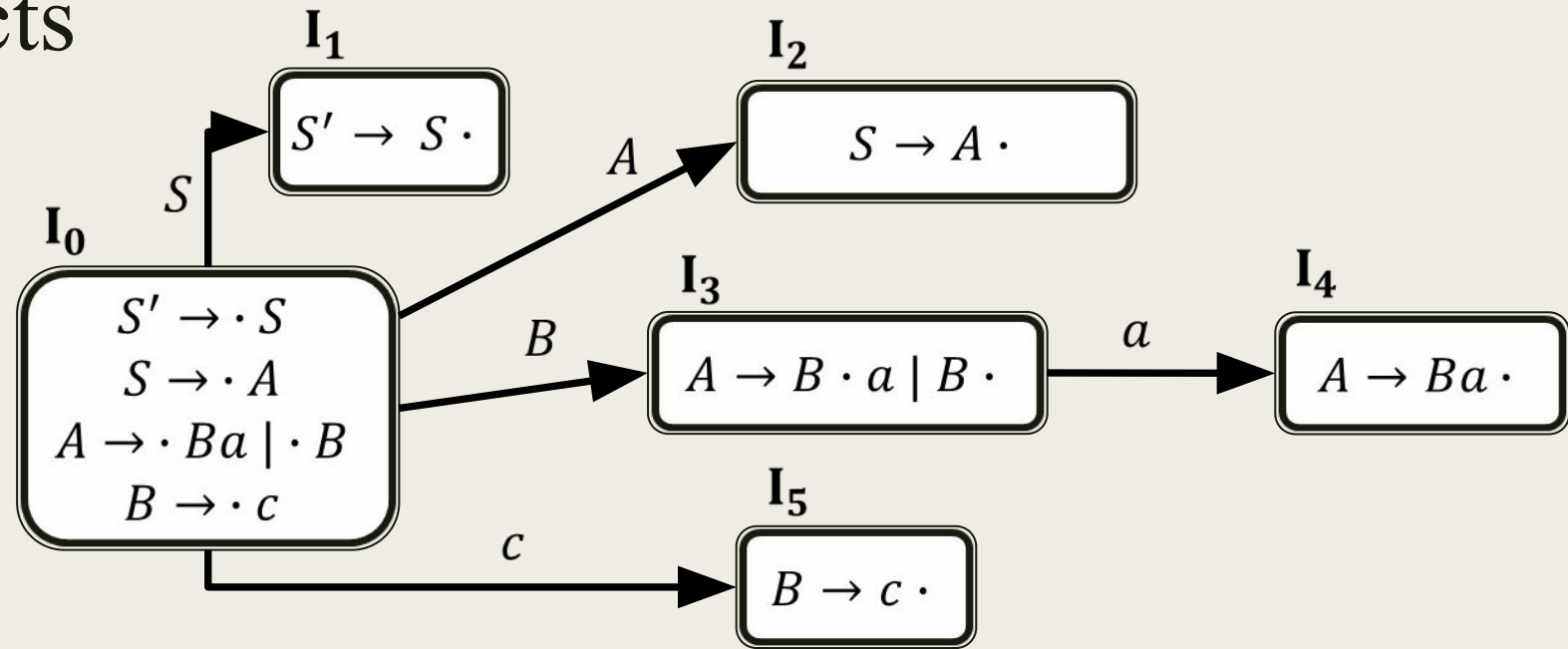
LR(0) parsing table

	Actions			Go To		
			\$			
0				1	2	3
1			accept			
2						
3						
4						
5						

Shift/reduce  
conflict



# Shift-reduce conflicts



SLR(1) parsing table

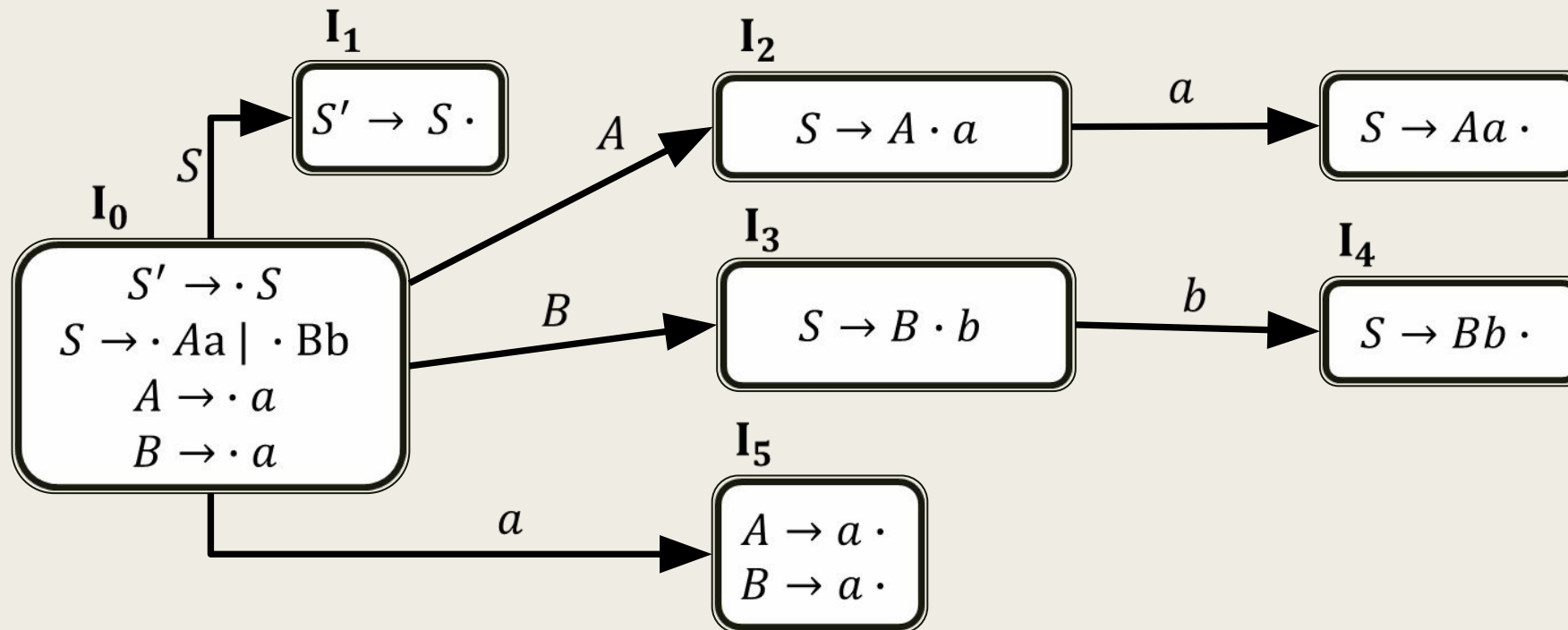
	Actions			Go To		
			\$			
0				1	2	3
1			accept			
2						
3						
4						
5						


# LR(0) vs. SLR(1) [Reduce-reduce conflict]

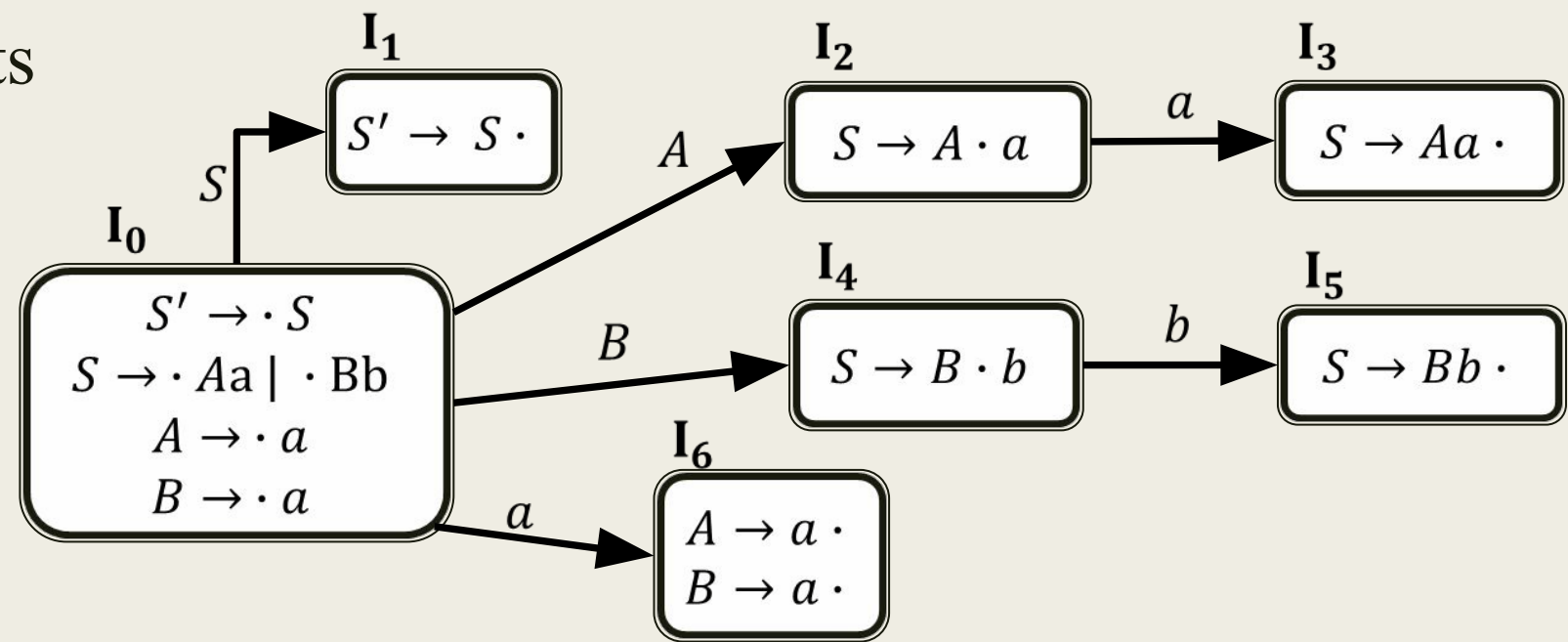
- Let's consider the grammar:

$$\begin{aligned} S &\rightarrow Aa \mid Bb \\ A &\rightarrow a \\ B &\rightarrow a \end{aligned}$$

- The LR(0) automaton for the grammar is:



# Reduce-reduce conflicts

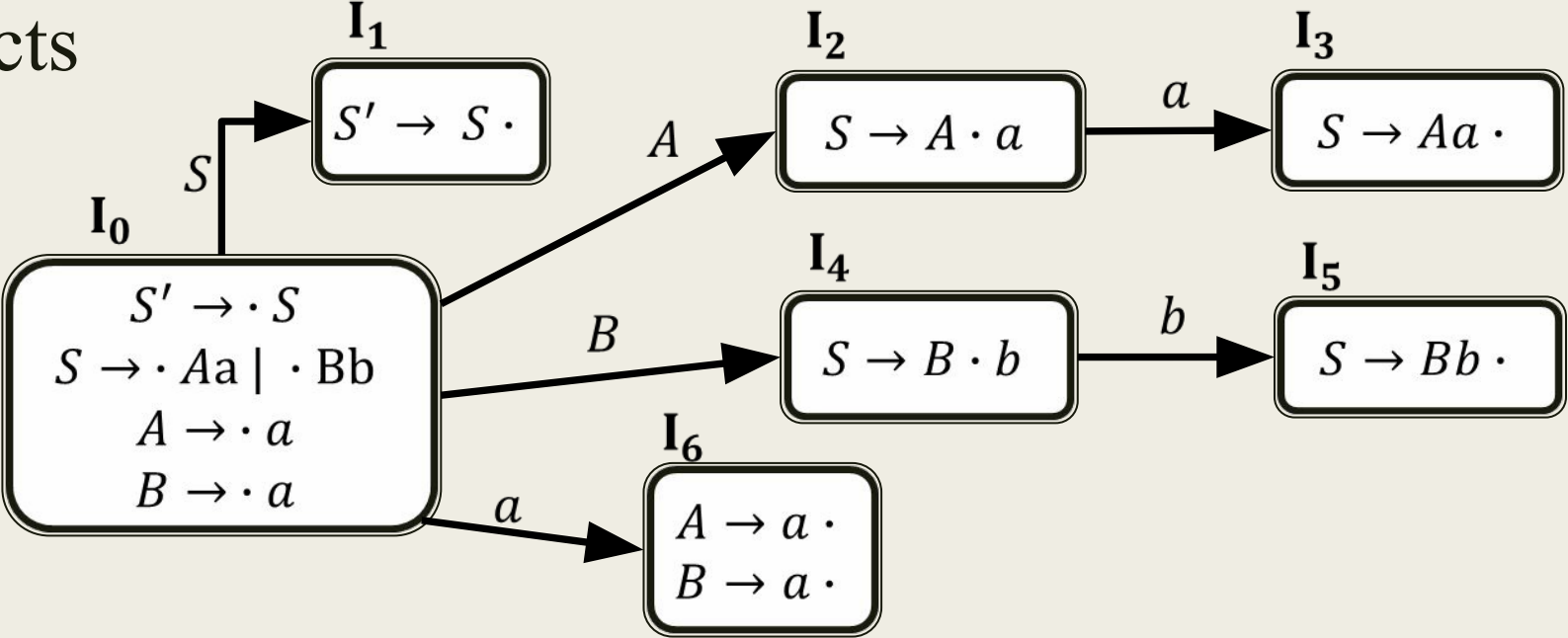


LR(0) parsing table

	Actions				Go To		
				\$			
0					1	2	4
1				accept			
2							
3							
4							
5							
6							

reduce/reduce  
conflict

# Reduce-reduce conflicts



SLR(1) parsing table

	Actions				Go To		
				\$			
0					1	2	4
1				accept			
2							
3							
4							
5							
6							


# CLR(1) (Canonical LR) parser

- We will use the canonical collection of LR(1) items.
- LR(1) item = LR(0) items + lookahead
- Let's consider the following:

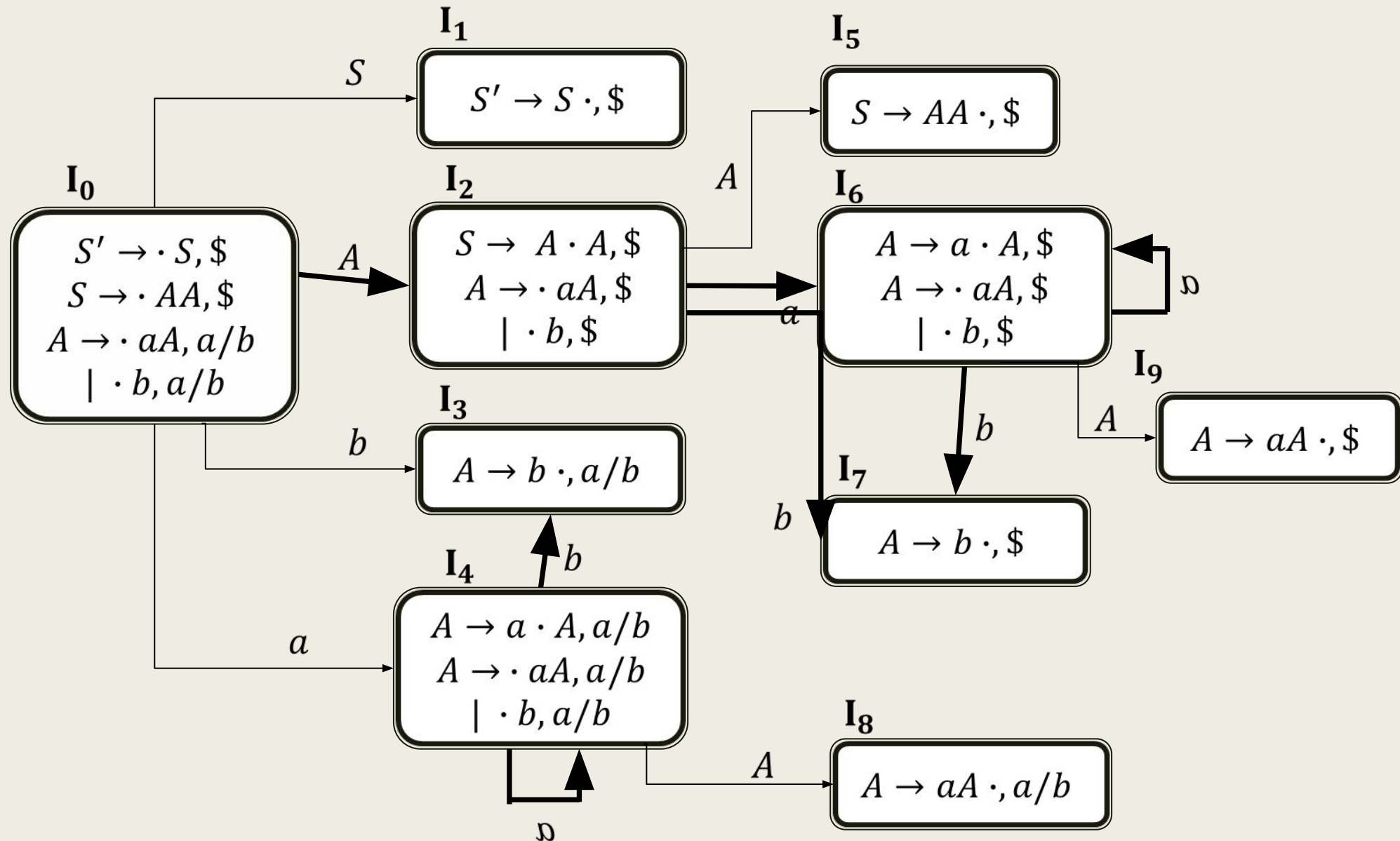
$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow aA \mid b \end{aligned}$$

- We augment the grammar to get:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow AA \\ A &\rightarrow aA \mid b \end{aligned}$$

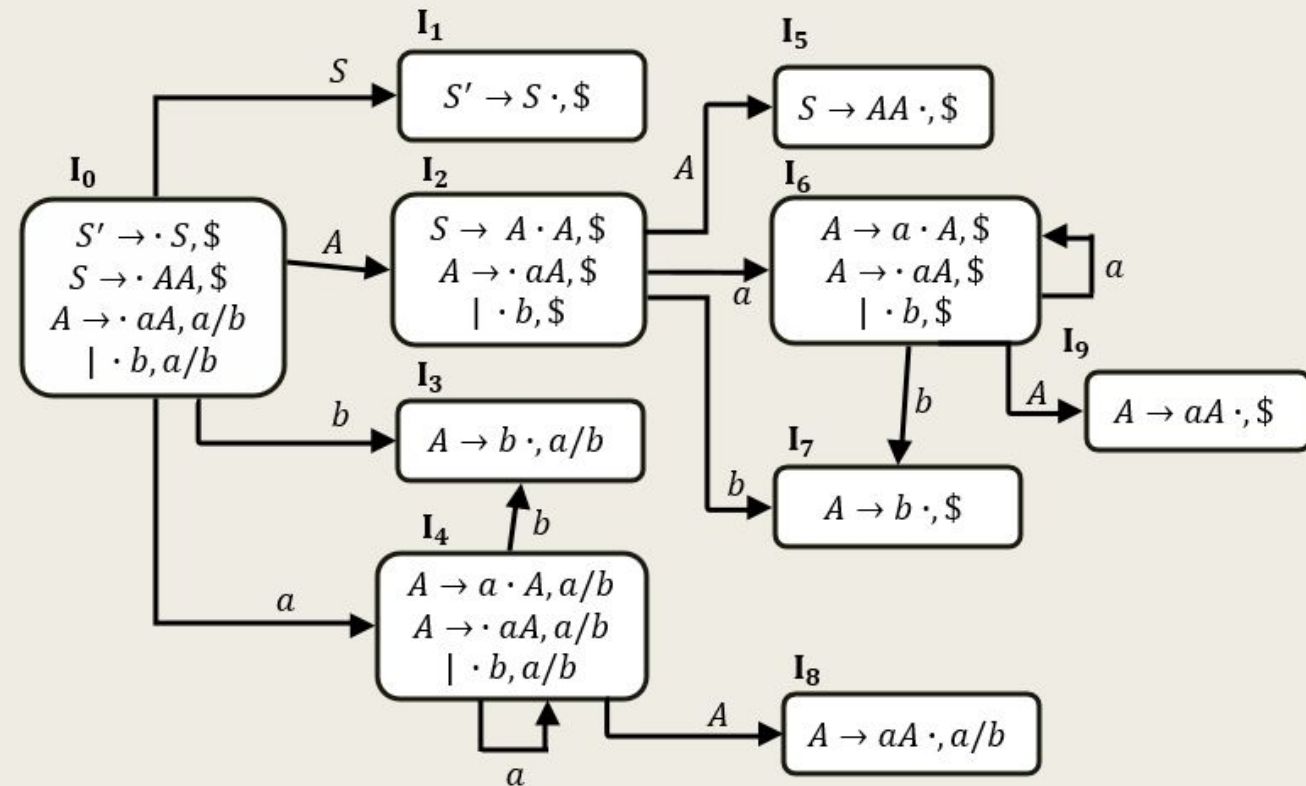
Let's construct the LR(1) automaton for this grammar.

# LR(1) Automaton [Canonical collection of LR(1) items]



# CLR(1) (canonical LR) parsing table

	Actions			Go To	
			\$		
0				1	2
1			accept		
2					5
3					
4					8
5					
6					9
7					
8					
9					

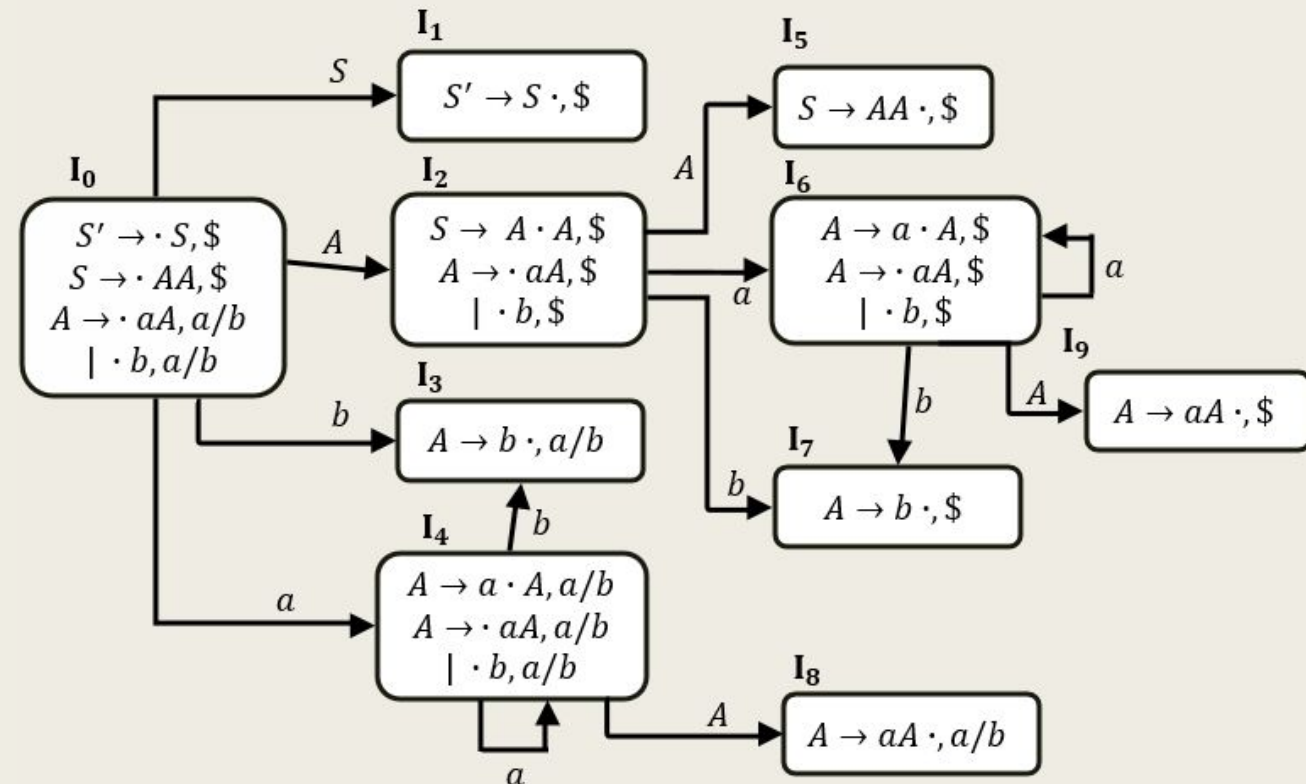




# LALR(1) (lookahead LR) parsing table

In LALR(1) we merge the states with common core if no conflicts arise and therefore, reduce the number of states as compared to CLR.

	Actions			Go To	
			\$		
0				1	2
1			accept		
2					5
37					
46					89
5					
89					



# Comparison of SLR(1) and LALR(1)

For this example, the SLR(1) table and LALR(1) table is the same as no conflicts arise while merging. The same will not be true for other grammars.

	Actions			Go To	
			\$		
0				1	2
1			accept		
2					5
37					
46					89
5					
89					

	Actions			Go To	
0				1	2
1			accept		
2					5
3					6
4					
5					
6					