5. Bottom-Up Parsing

Contents

PART ONE
5.1 Overview of Bottom-Up Parsing
5.2 Finite Automata of LR(0) Items and LR(0) Parsing

PART TWO
5.3 SLR(1) Parsing
5.4 General LR(1) and LALR(1) Parsing
5.5 Yacc: An LALR(1) Parser Generator

5.1 Overview of Bottom-Up Parsing

- **A bottom-up parser uses an explicit stack to perform a parse**
  - **The parsing stack contains tokens, nonterminal as well as some extra state information**
  - **The stack is empty at the beginning of a bottom-up parse, and will contain the start symbol at the end of a successful parse**

- **A schematic for bottom-up parsing:**

$	**InputString $**	
…….		
$ StartSymbol	**$**	**accept**

- **A bottom-up parser has two possible actions (besides "accept"):**

  Shift **a terminal from the front of the input to the top of the stack**

  Reduce **a string α at the top of the stack to a nonterminal A, given the BNF choice $A \rightarrow \alpha$**

- **A bottom-up parser is thus sometimes called a shift-reduce parser**
- **One further feature of bottom-up parsers is that, grammars are always augmented with a new start symbol**

$$S' \rightarrow S$$

- **Example 5. 1 The augmented grammar for balanced parentheses:**

  $S' \rightarrow S$

  $S \rightarrow (S)S|\varepsilon$

- **A bottom-up parse of the string "()" using this grammar is given in following table**

	Parsing stack	Input	Action
1	$	()$	shift
2	$(	)$	reduce $S \rightarrow \varepsilon$
3	$(S	)$	shift
4	$(S)	$	reduce $S \rightarrow \varepsilon$
5	$(S)S	$	reduce $S \rightarrow (S)S$
6	$S	$	reduce $S' \rightarrow S$
7	$S'	$	accept

Steps	Parsing Stack	Input	Action
1	$S	( ) $	generate S→(S) S
2	$S)S(	( ) $	match
3	$S)S	)$	generate S→ ε
4	$S)	)$	match
5	$S	$	generate S→ ε
6	$	$	accept

Top-down Parsing vs. Bottom-up Parsing

	Parsing stack	Input	Action
1	$	( )$	shift
2	$(	)$	reduce S→ε
3	$(S	)$	shift
4	$(S)	$	reduce S→ε
5	$(S)S	$	reduce S → (S)S
6	$S	$	reduce S' → S
7	$S’	$	accept

- The handle of the right sentential form:
  - A string, together with the position in the right sentential form where it occurs, and the production used to reduce it.
- Determining the next handle is the main task of a shift-reduce parser.

	Parsing stack	input	Action
1	\$	n+n\$	shift
2	\$n	+n\$	reduce E→n
3	\$E	+n\$	shift
4	\$E+	n\$	shift
5	\$E+n	\$	reduce E→E + n
6	\$E	\$	reduce E'→E
7	\$E'	\$	accept

- **Note:**
  - The string of a handle forms a complete right-hand side of one production choice.
  - To be the handle, it is not enough for the string at the top of the stack to match the right-hand side of a production.
    - Indeed, if an ε-production is available for reduction, then its right-hand side (the empty string) is always at the top of the stack.
    - Reductions occur only when the resulting string is indeed a right sentential form.
  - For example, in step 3 of Table 5.1 a reduction by $S \rightarrow \varepsilon$ could be performed, but the resulting string (SS) is not a right sentential form, and thus ε is not the handle at this position in the sentential form (S) .

5.2 FINITE AUTOMATA OF LR(0) ITEMS AND LR(0) PARSING

5.2.1 LR(0) Items

- **An LR(0) item of a context-free grammar:**
  - **A production choice with a distinguished position in its right-hand side**
  - **Indicate the distinguished position by a period**
- **Example:**
  - **if $A \rightarrow \alpha$ is a production choice, and if β and γ are any two strings of symbols (including the empty string ε) such that $\beta\gamma = \alpha$**
  - **then $A \rightarrow \beta \cdot \gamma$ is an LR(0) item.**
- **These are called LR(0) items because they contain no explicit reference to lookahead**

Example 5.3 The grammar of Example 5. l :

$S' \rightarrow S$
$S \rightarrow (S)S|\varepsilon$

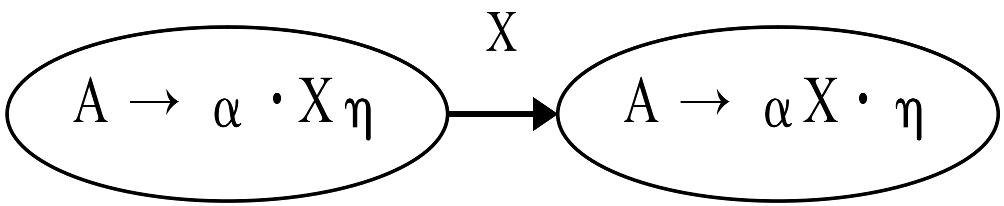
This grammar has three production choices and eight items:

$S' \rightarrow \cdot S$
$S' \rightarrow S\cdot$
$S \rightarrow \cdot(S)S$
$S \rightarrow (\cdot S)S$
$S \rightarrow (S\cdot)S$
$S \rightarrow (S)\cdot S$
$S \rightarrow (S)S\cdot$
$S \rightarrow \cdot$

5.2.2 Finite Automata of Items

- **The LR(0) items can be used as the states of a finite automaton**
  - **that maintains information about the parsing stack and the progress of shift-reduce parse.**
  - **This will start out as a nondeterministic finite automaton.**
- **From the NFA of LR(0) items to construct the DFA of sets of LR(0) items using the subset construction**
- ***The transitions of the NFA of LR(0) items***
  - **Consider the item $A \rightarrow \alpha \cdot \gamma$**
  - **Suppose γ begins with the symbol X, be either a token or a nonterminal**
  - **So that the item can be written as $A \rightarrow \alpha \cdot X\eta$.**

- **A transition on the symbol X from the state represented by this item $A \to \alpha \cdot X\eta$ to the state represented by the item $A \to \alpha X \cdot \eta$**



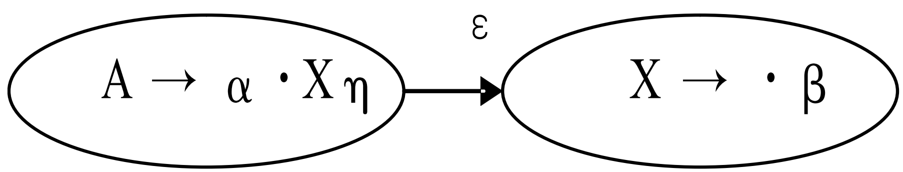
(1) If X is a token, then this transition corresponds to a shift of X from the input to the top of the stack.

(2) If X is a nonterminal, then the interpretation of this transition is more complex, since X will never appear as an input symbol.

- In fact, such a transition will **still correspond to the pushing of *X*** onto the stack during a parse;
- **This can only occur during a reduction by $X \to \beta$.**
  - Such a reduction **must be preceded by *the recognition*** of a β, and the state **given by the initial item** $X \to \cdot\beta$ represents the beginning of this process

    (the dot indicating that we are about to recognize a β,

    then for every item $A \to \alpha \cdot X\eta$ we must add an ε-transition)



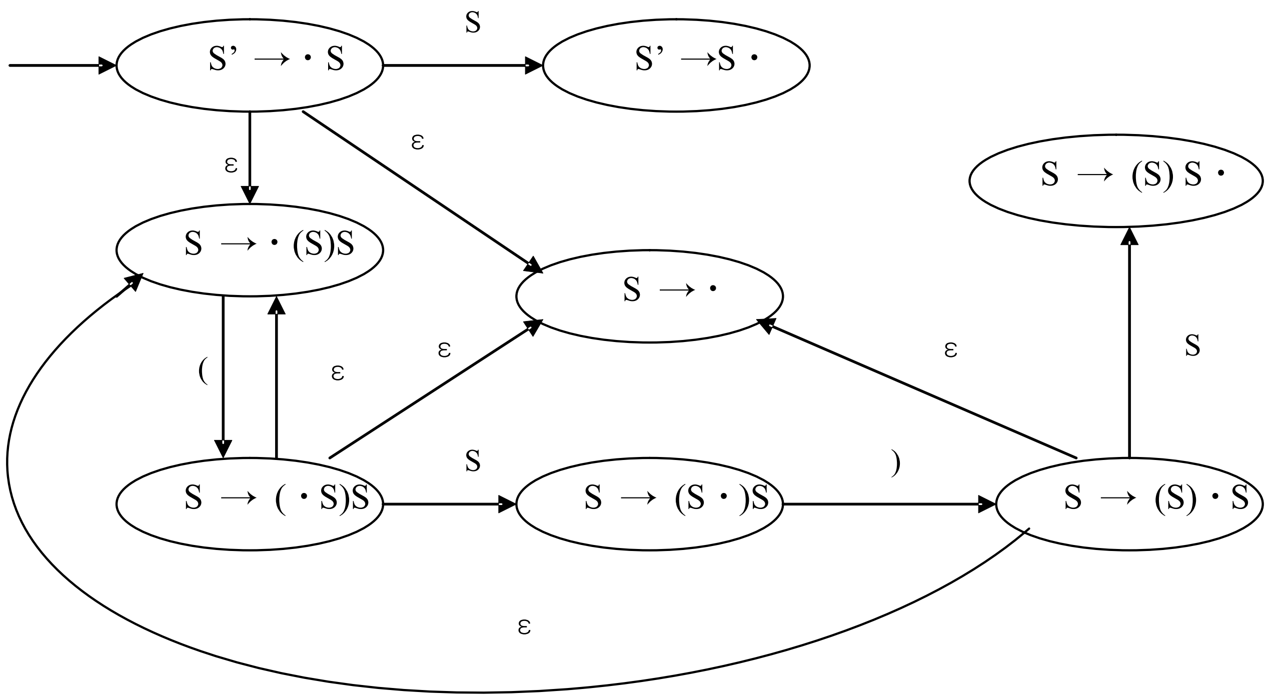
for every production choice $X \to \beta$ of X, indicating that **X can be produced** by recognizing **any of its production choices**.

- **The start state of the NFA should correspond to the initial state of the parser:**
  - **The stack is empty**
  - **Be about to recognize an S, where S is the start symbol of the grammar.**
  - **Any initial item $S \rightarrow \cdot\alpha$ could serve as a start state.**
  - **Unfortunately, there may be many such production choices for S.**

- **The solution is to augment the grammar by a single production $S' \rightarrow S$**
  - **S' becomes the start symbol of the augmented grammar, and the initial item $S' \rightarrow \cdot\, S$ becomes the only start state of the NFA.**

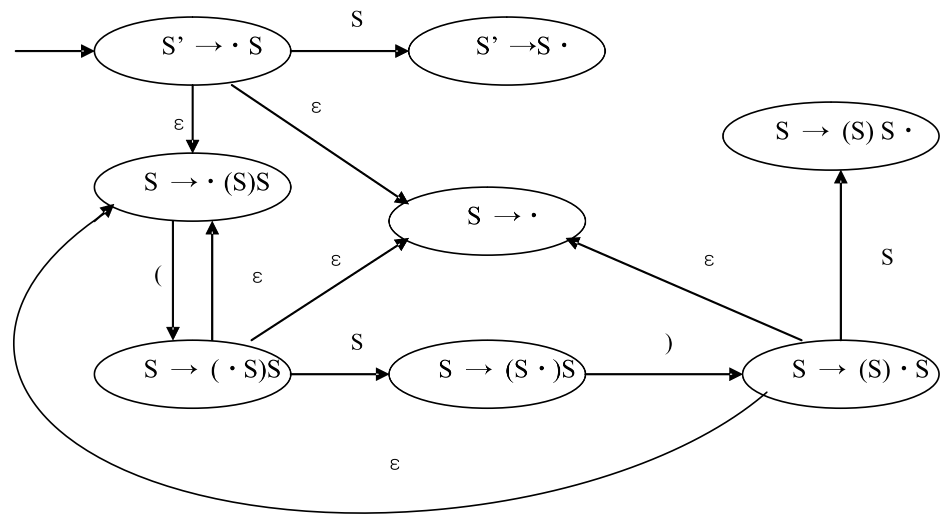
- **The NFA will have no accepting states at all:**
  - **The purpose of the NFA is to keep track of the state of a parse, not to recognize strings;**
  - **The parser itself will decide when to accept, and the NFA need not contain that information.**

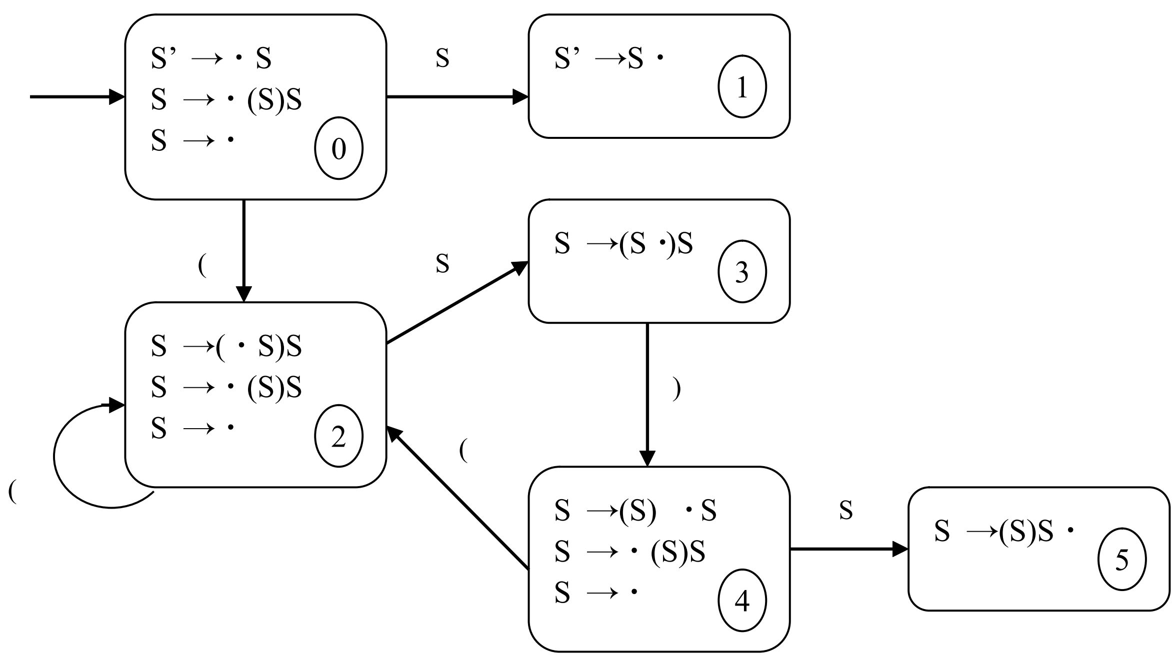
- **Example 5.5 The NFA with the eight LR(0) items of the grammar S→(S)S|ε**
- **Note : Every item with a dot before the non-terminal *S* has an *ε*-transition to every initial item of *S*.**



- ***In order to describe the use of items to keep track of the parsing state, one must construct the DFA of sets of items corresponding to the NFA of items according to the subset construction.***

Example 5.7 Consider the NFA of Figure 5.l.





5.2.3The LR(0) Parsing Algorithm

Definition: The LR (0) parsing algorithm

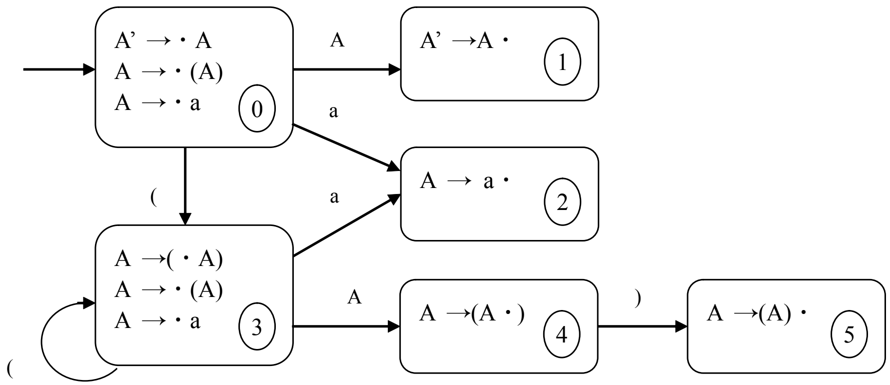
Let *s* be the current state (at the top of the parsing stack). Then actions are defined as follows:

1. If state *s* contains any item of the form $A \rightarrow \alpha \cdot X\beta$, where X is a terminal. Then the action is to *shift the current input token* onto the stack.
   - If this token is X and state s contains item $A \rightarrow \alpha \cdot X\beta$, then the new state to be pushed on the stack is the state containing the item $A \rightarrow \alpha X \cdot \beta$.
   - If this token is not X for some item in state s of the form just described, an error is declared.

2. If state s contains any complete item (an item of the form $A \rightarrow \gamma\cdot$), then the action is to reduce by the rule $A \rightarrow \gamma$

- **A reduction by the rule $S' \rightarrow S$, where S is the start symbol, is equivalent to acceptance, provided the input is empty, and error if the input is not empty**
- **In all other cases, the new state is computed as follows:**
  - **Remove the string γ and all of its corresponding states from the parsing stack**
  - **Back up in the DFA to the state from which the construction of γ began**
  - **Again, by the construction of the DFA, this state must contain an item of the form $B \rightarrow \alpha\cdot A\beta$**
  - **Push A onto the stack, and push (as the new state) the state containing the item $B \rightarrow \alpha A\cdot\beta$.**

Example 5.9 Consider the grammar: A → (A) | a



State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A'→A				
2	reduce	A→a				
3	shift		3	2		4
4	shift				5	
5	reduce	A→(A)				

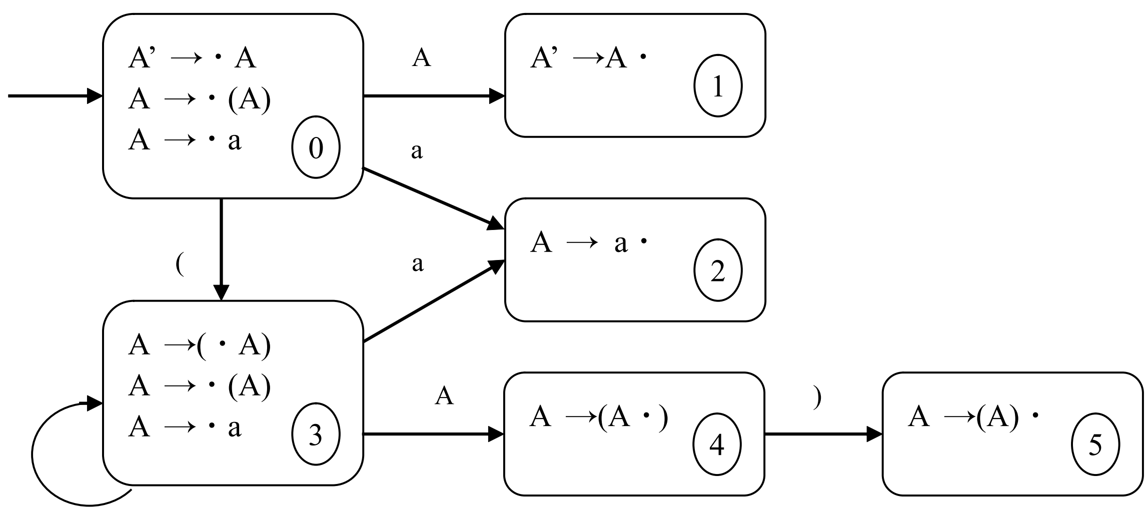
	Parsing stack	input	Action
1	$ 0	((a))$	shift
2	$ 0 (3	(a))$	shift
3	$ 0 (3 (3	a)$	shift
4	$ 0 (3 (3 a 2	))$	reduce A→a
5	$ 0 (3 (3 A 4	))$	shift
6	$ 0 (3 (3 A 4) 5	)$	reduce A→(A)
7	$ 0 (3 A 4	)$	shift
8	$ 0 (3 A 4) 5	$	reduce A→(A)
9	$ 0 A 1	$	accept

The parsing table

- The DFA of sets of items and the actions specified by the LR(0) parsing algorithm can be combined into a parsing table.
- Then, the LR(0) parsing becomes a table-driven parsing method.

An example of a parsing table

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A'→A				
2	reduce	A→a				
3	shift		3	2		4
4	shift				5	
5	reduce	A→(A)				



5.3 SLR(1) Parsing

SLR(1), called simple LR(1) parsing, uses the DFA of sets of LR(0)

SLR(1) increases the power of LR(0) parsing by using the next token in the input string

- **First, it consults the input token *before* a shift to make sure that an appropriate DFA transition exists**
- **Second, it uses the Follow set of a non-terminal to decide if a reduction should be performed**

5.3.1 The SLR(1) Parsing Algorithm

SLR(1) parsing algorithm

Let s be the current state,

actions are defined as follows: .

1.If state s contains any item of form $A \rightarrow \alpha \cdot X\beta$

where X is a terminal, and

X is the next token in the input string,

then to shift the current input token onto the stack,

and push the new state containing the item

$A \rightarrow \alpha X \cdot \beta$

2. If state s contains the complete item $A \rightarrow \gamma \cdot$,

and the next token in input string is in Follow(A)

then to reduce by the rule $A \rightarrow \gamma$

SLR(1) parsing algorithm

2. (Continue)

   A reduction by the rule $S' \rightarrow S$, is equivalent to acceptance;

   – This will happen only if the next input token is $.

   In all other cases, remove the string γ and corresponding states from the parsing stack

   – Back up in the DFA to the state from which the construction of γ began.

   – This state must contain an item of the form $B \rightarrow \alpha \cdot A\beta$.

   Push A onto the stack, and the state containing the item $B \rightarrow \alpha A \cdot \beta$.

3. If the next input token is such that neither of the above two cases applies,

   – an error is declared

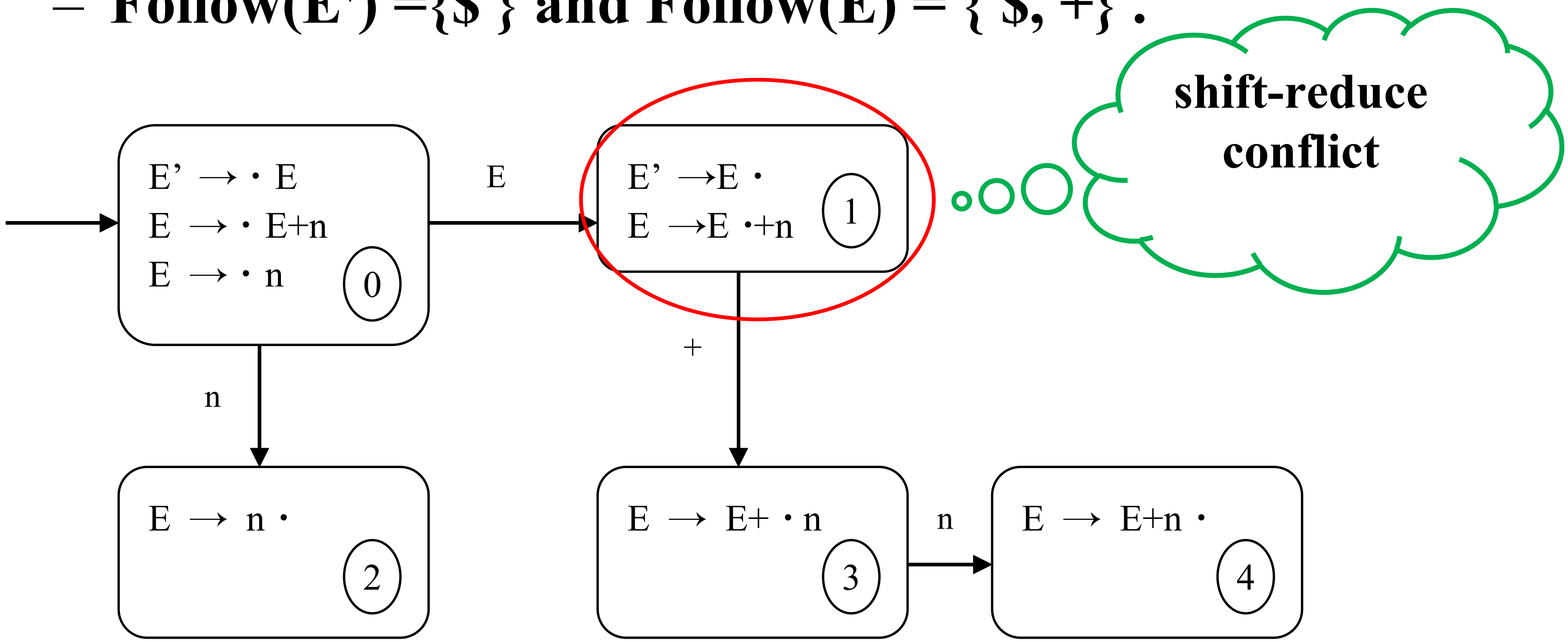
A grammar is SLR(1) if and only if, for any state s, the following two conditions are satisfied:

- For any item $A \rightarrow \alpha\cdot X\beta$ in s with X a terminal,
  There is no complete item $B \rightarrow \gamma.$ in s with X in Follow(B).
- For any two complete items $A \rightarrow \alpha\cdot$ and $B \rightarrow \beta\cdot$ in s, Follow(A) $\cap$ Follow(B) is empty.

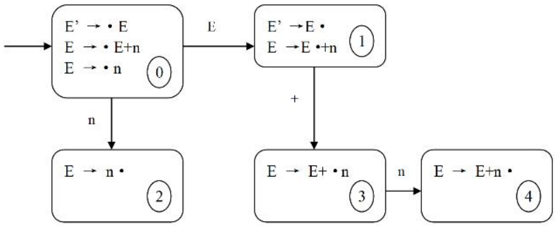
A violation of the first of these conditions represents a **shift-reduce conflict**

A violation of the second of these conditions represents a **reduce-reduce conflict**

- **Example 5. 10 Consider the grammar:**
  - **E'→E**
  - **E→E + n | n**
- **This grammar is not LR(0), but it is SLR(l).**
  - **The Follow sets for the nonterminals:**
  - **Follow(E') ={$ } and Follow(E) = { $, +} .**



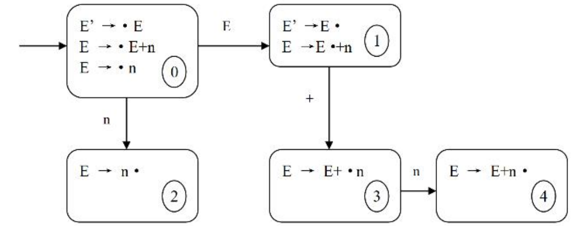
The SLR(1) parsing table for above Grammar



State	Input			Goto
	n	+	$	E
0	S2			1
1		$\boxed{\text{S3}}$	accept	
2		r($E \rightarrow n$)	r($E \rightarrow n$)	
3	S4			
4		r($E \rightarrow E+n$)	r($E \rightarrow E+n$)	

- A shift is indicated by the letter *s* in the entry, and a reduction by the letter *r*
  - In state 1 on input "+", a shift is indicated, together with a transition to state 3 (no reduction, because "+" ∉ follow(E'))
  - In state 2 on input "+", a reduction by production E → n is indicated

The parsing process for *n+n+n*



	Parsing stack	Input	Action
1	*\$0*	*n + n + n \$*	*shift 2*
2	*\$0 n 2*	*+n + n \$*	*reduce E →n*
3	*\$0E1*	*+n + n \$*	*shift3*
4	*\$0E1+3*	*n + n \$*	*shift4*
5	*\$0E1+3n4*	*+ n \$*	*reduce E → E+n*
6	*\$0E1*	*+ n \$*	*shift3*
7	*\$0E1+3*	*n \$*	*shift4*
8	*\$0E1+3n4*	*\$*	*reduce E → E+n*
9	*\$0E1*	*\$*	*accept*

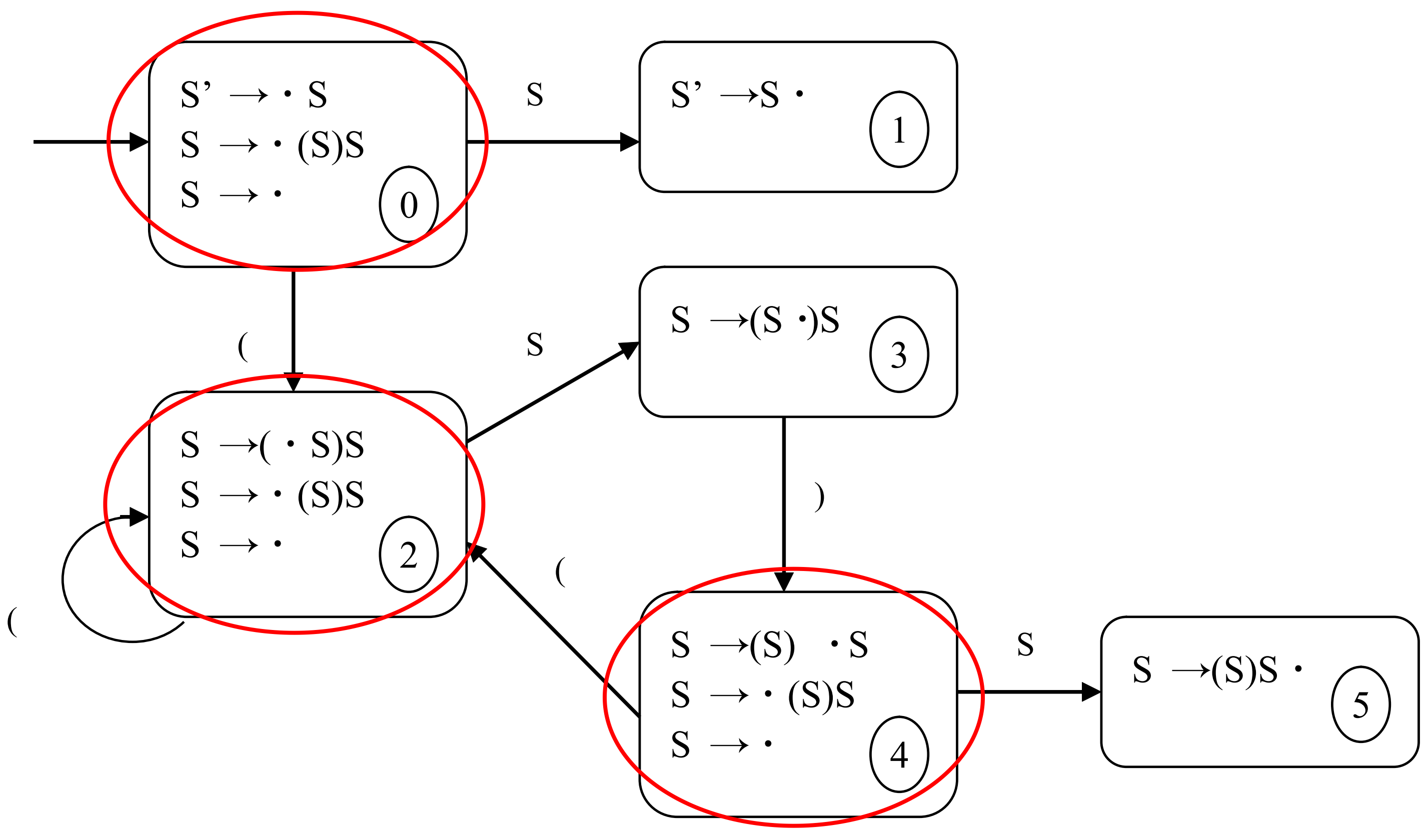
Example 5. 11 Consider the grammar of balanced parentheses

$S' \rightarrow S$

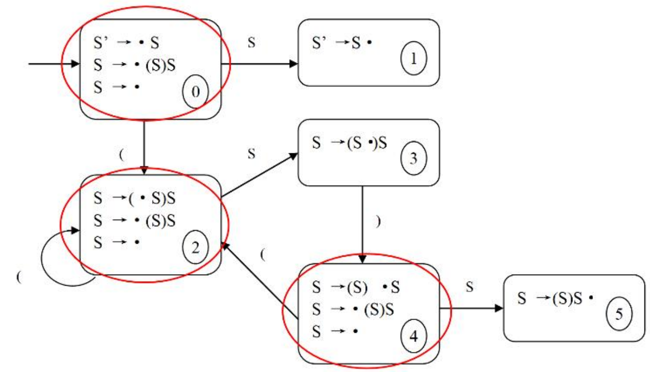
$S \rightarrow (S)S|\varepsilon$

Follow sets computation yields:

– Follow(S') = {\$} and Follow(S)={\$,)}.



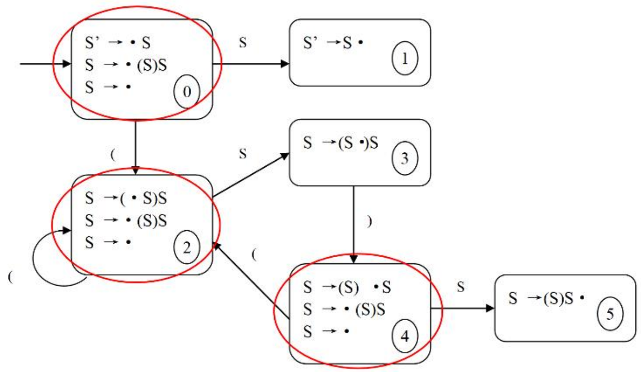
- **The SLR(l) parsing table is as follows, where**
  - **The non-LR(0) states 0, 2, and 4 have both shifts and reductions by $S \rightarrow \varepsilon$.**



State	Input			Goto
	(	)	$	S
0	s2	r(S → ε)	r(S → ε)	1
1			accept	
2	s2	r(S → ε)	r(S → ε)	3
3		s4		
4	s2	r(S → ε)	r(S → ε)	5
5		r(S → (S)S)	r(S → (S)S)	

Follow(S') = {$} and Follow(S)={$,)}

- **The steps to parse the string "()()"**



	Parsing stack	Input	Action
1	*\$0*	*( ) ( )\$*	*shift 2*
2	*\$0(2*	*) ( )\$*	*reduce* S → *ε*
3	*\$0(2S3*	*) ( )\$*	*shift4*
4	*\$0(2S3)4*	*( )\$*	*shift2*
5	*\$0(2S3)4(2*	*)\$*	*reduce* S → *ε*
6	*\$0(2S3)4(2S3*	*) \$*	*shift4*
7	*\$0(2S3)4(2S3)4*	*\$*	*reduce* S → *ε*
8	*\$0(2S3)4(2S3)4S5*	*\$*	*reduce* S → (S)S
9	*\$0(2S3)4S5*	*\$*	*reduce* S → (S)S
10	*\$0S1*	*\$*	*accept*

5.3.3 Limits of SLR(1) Parsing Power

- Example 5. 13 Consider the following grammar rules for statements.

  $stmt \rightarrow call\text{-}stmt \mid assign\text{-}stmt$

  $call\text{-}stmt \rightarrow \boldsymbol{identifier}$

  $assign\text{-}stmt \rightarrow var\ \boldsymbol{:=} exp$

  $var \rightarrow \boldsymbol{identifier}$

  $exp \rightarrow var \mid \boldsymbol{number}$

- Simplify this situation to the following grammar without changing the basic situation:

  $S \rightarrow \boldsymbol{id} \mid V\ \boldsymbol{:=}\ E$

  $V \rightarrow \boldsymbol{id}$

  $E \rightarrow V \mid \boldsymbol{n}$

- To show how this grammar results in parsing conflict in SLR(l) parsing, consider the start state of the DFA of sets of items:

$$S' \rightarrow \cdot S$$
$$S \rightarrow \cdot \boldsymbol{id}$$
$$S \rightarrow \cdot V \mathbin{\boldsymbol{:=}} E$$
$$V \rightarrow \cdot \boldsymbol{id}$$

- This state has a shift transition on *id* to the state

$$S \rightarrow \boldsymbol{id}\cdot$$
$$V \rightarrow \boldsymbol{id}\cdot$$

- Now, Follow(S)={\$} and Follow(V)={:=, \$}
- Thus, the SLR(l) parsing algorithm calls for a reduction in this state by both the rule $S \rightarrow \boldsymbol{id}$ and the rule $V \rightarrow \boldsymbol{id}$ under input symbol \$.
  - (this is a reduce-reduce conflict)

5.4 General LR(1) and LALR(1) Parsing

5.4.1 Finite Automata of LR(1) Items

- The SLR(1) method:
  - Applies lookaheads after the construction of the DFA of LR(0) items
  - **The construction of DFA ignores lookaheads**
- The general LR(1) method:
  - **Using a new DFA with the lookaheads built into its construction**

    The DFA items are an extension of LR(0) items

    LR(1) items include a single lookahead token in each item.
  - A pair consisting of an LR(0) item and a lookahead token.

    LR(1) items using square brackets as $[A \rightarrow \alpha \cdot \beta, a]$

    where $A \rightarrow \alpha \cdot \beta$ is an LR(0) item and a is a lookahead token

- The definition of transitions between LR(l) items
  - Similar to the LR(0) transitions except keeping track of lookaheads
  - As with LR(0) items including ε-transitions, to build a DFA's states are sets of items that are ε-closures
  - **The difference between the LR(0) and LR(1) automata comes in the definition of the ε-transitions**

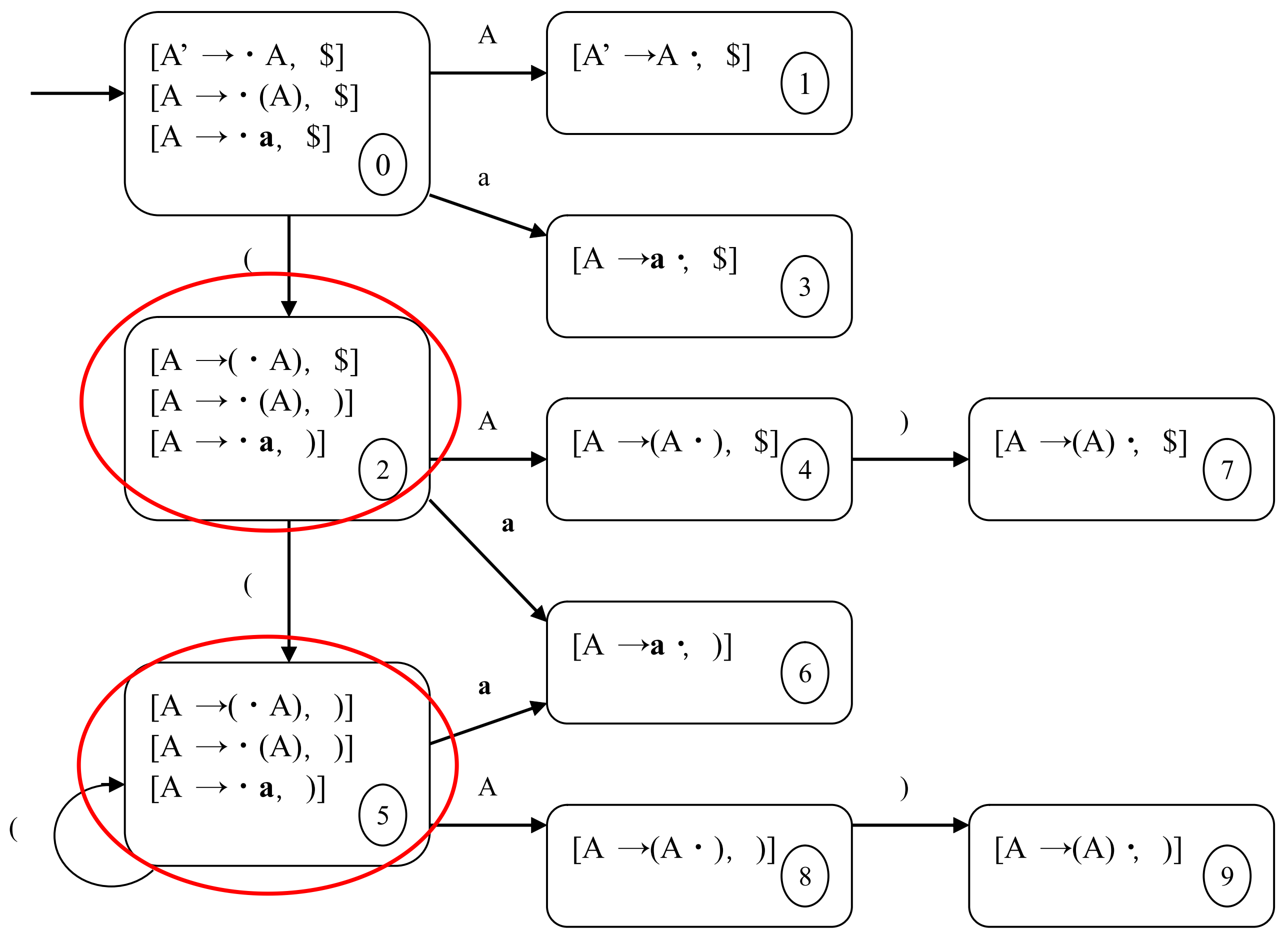
Definition

- Definition of LR(1) transitions (part 1).
  - Given an LR(1) item $[A \rightarrow \alpha \cdot X\gamma, a]$, where X is any symbol (terminal or nonterminal),
  - There is a transition on X to the item $[A \rightarrow \alpha X \cdot \gamma, a]$
- Definition of LR(1) transitions (part 2).
  - Given an LR(1) item $[A \rightarrow \alpha \cdot B\gamma, a]$, where B is a nonterminal,
  - There are ε-transitions to items $[B \rightarrow \cdot\beta, b]$ for *every production* $B \rightarrow \beta$ and *for every token b* in First(γa).

- **Example 5.14** Consider the grammar

  A→(A) | a

- The DFA of sets of LR(1) items



5.4.2 The LR(1) Parsing Algorithm

- Need to complete the discussion of general LR(1) parsing by restating the parsing algorithm based on the new DFA construction
- Only need to restate the SLR(1) parsing algorithm, except that it **uses the lookahead tokens in the LR(1) items instead of the Follow set**

The General LR(1) parsing algorithm

Let s be the current state (at the top of the parsing stack),

Then actions are defined as follows:

1. If state s contains any LR(l) item of the form $[A \rightarrow \alpha \cdot X\beta, a]$, where X is a terminal, and X is the next token in the input string, then the action is to shift the current input token onto the stack,

   and the new state to be pushed on the stack is the state containing the LR(1) item $[A \rightarrow \alpha X \cdot \beta, a]$.

2. If state s contains the complete LR(1) item $[A \rightarrow \alpha \cdot, a]$, and the next token in the input string is a.

   then the action is to reduce by the rule $A \rightarrow \alpha$.

   – A reduction by the rule $S' \rightarrow S$, where S is the start state, is equivalent to acceptance.
   – (This will happen only if the next input token is \$.)

The General LR(1) parsing algorithm (cont.)

In the other cases, the new state is computed as follows.

- Remove the string α and all of its corresponding states from the parsing stack;
- back up in the DFA to the state from which the construction of α began.
- By construction, this state must contain an LR(l) item of the form $[B \rightarrow \alpha \cdot A\beta, b]$.
- Push A onto the stack, and push the state containing the item $[B \rightarrow \alpha A \cdot \beta, b]$.

3. If the next input token is such that neither of the above two cases applies,

- an error is declared.

- A grammar is LR(1) if and only if, for any state s, the following two conditions are satisfied.
  1. For any item $[A \to \alpha \cdot X\beta, a]$ in s with X a terminal, there is no item in s of the form $[B \to \gamma \cdot, X]$ (otherwise there is a shift-reduce conflict).
  2. There are no two items in s of the form $[A \to \alpha \cdot, a]$ and $[B \to \alpha \cdot, a]$ (otherwise, there is a reduce-reduce conflict).

The Grammar:

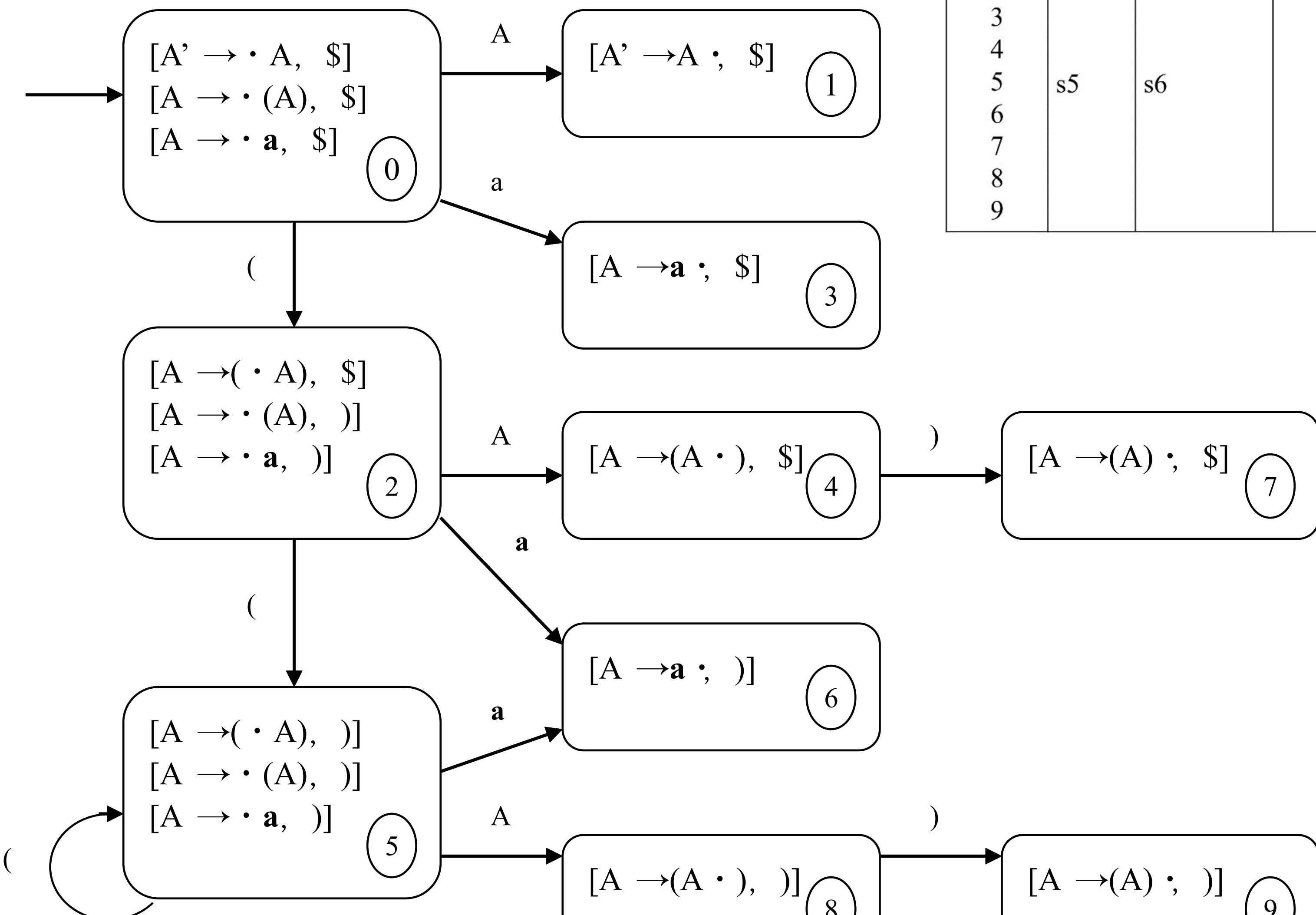
(1) A→(A)

(2) A→***a***

State	Input				Goto
	(	*a*	)	$	A
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

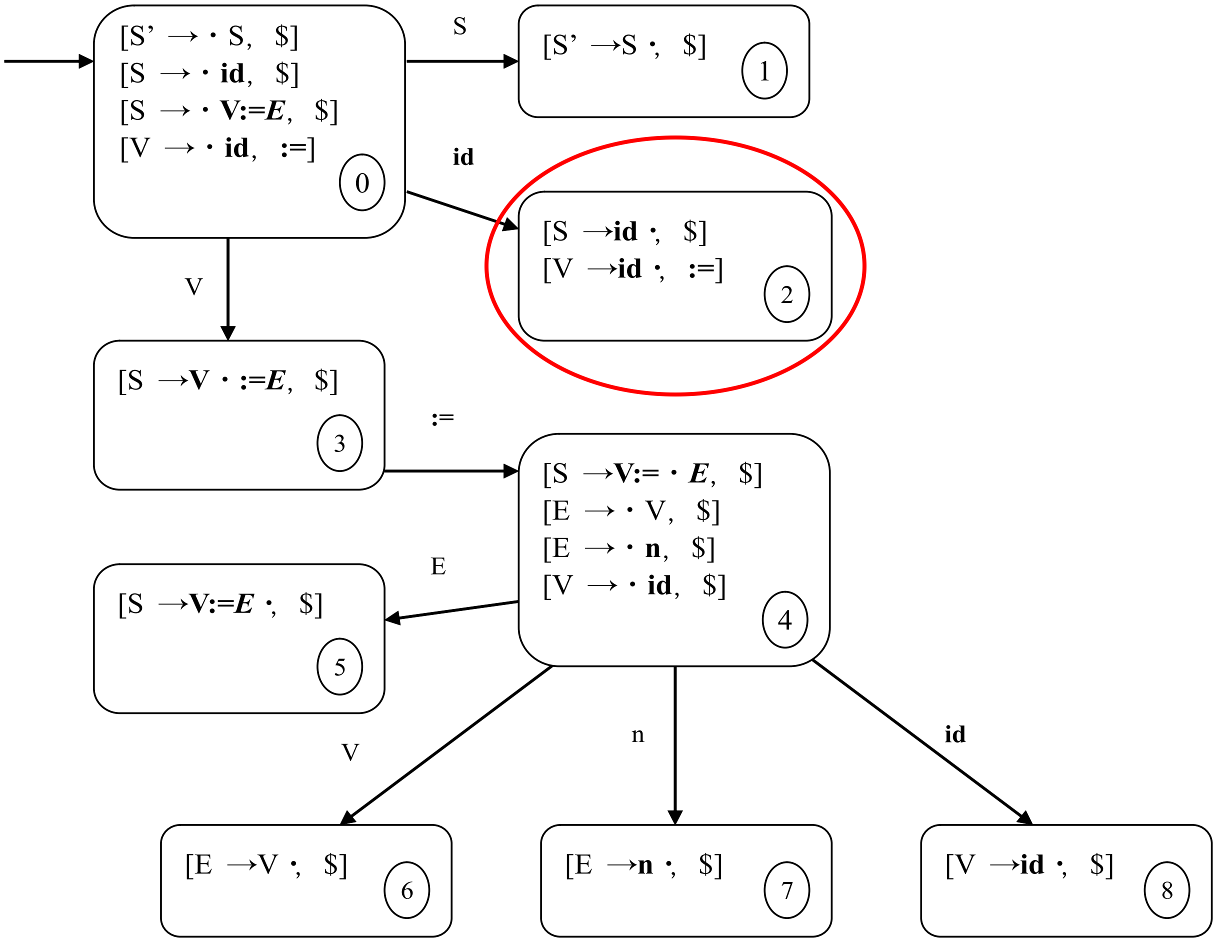
The DFA of LR(1) item

State	Input				Goto
	(	*a*	)	$	A
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		



- **Example 5.16** The grammar of Example 5. 13 in simplified form:

$$S \rightarrow \boldsymbol{id} \mid V := E$$
$$V \rightarrow \boldsymbol{id}$$
$$E \rightarrow V \mid \boldsymbol{n}$$



5.4.3 LALR(1) Parsing

- In the DFA of sets of LR(1) items, many different states have the same set of first components in their items (the LR(0) items), and different second components (the lookahead symbols)

- The LALR(1) (LookAhead LR) parsing algorithm:
  - Identify all such states and combine their lookaheads;
  - Then, we have a DFA identical to the DFA of LR(0) items, except the each state consists of items with sets of lookaheads.
- In the case of complete items, these lookahead sets are often smaller than the corresponding Follow sets.

- LALR(l) parsing retains some of the benefit of LR(l) parsing over SLR(l) parsing, while preserving the smaller size of the DFA of LR(0) items
- Formally, the core of a state of the DFA of LR(l) items is the set of LR(0) items consisting of the first components of all LR(1) items in the state

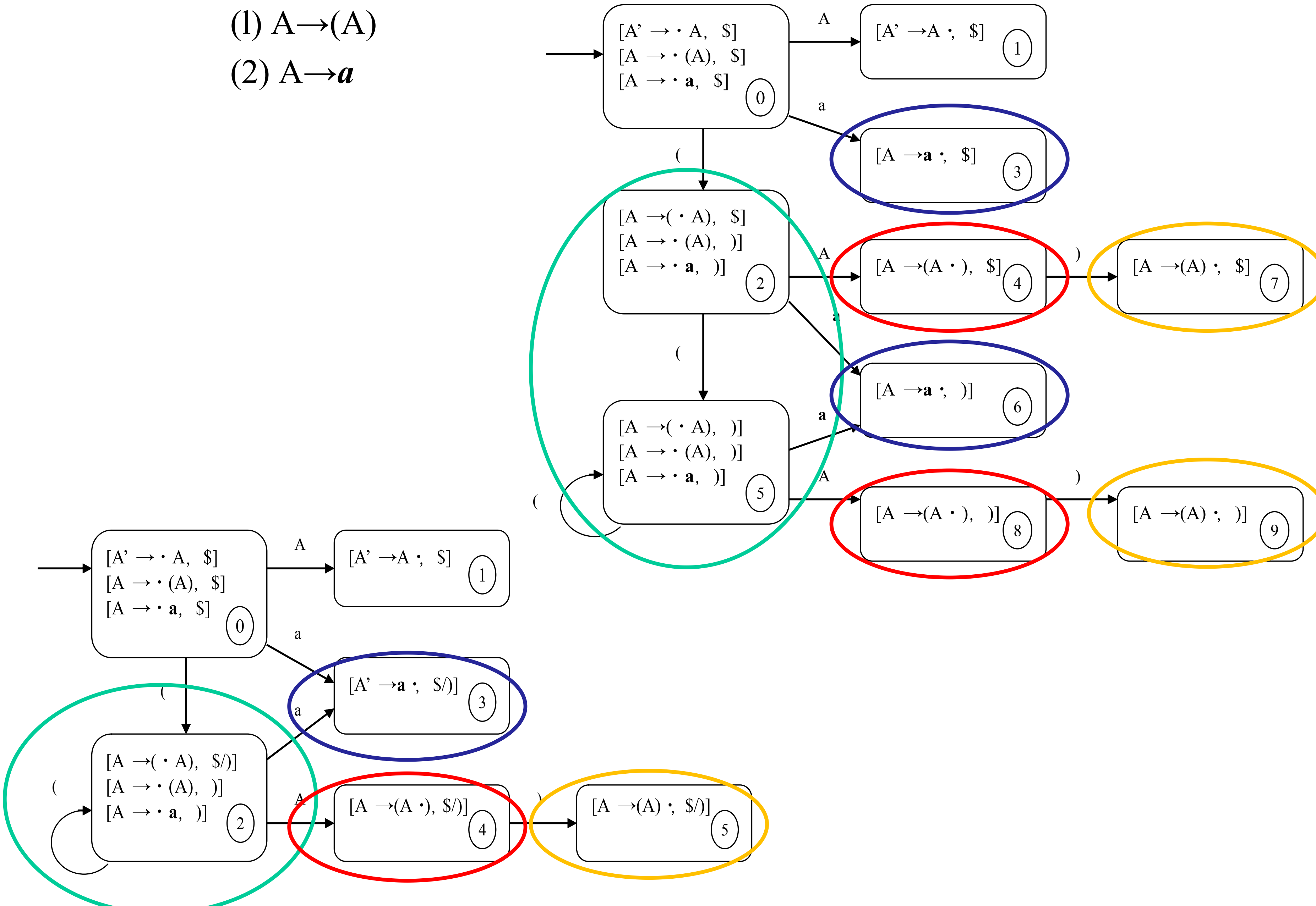
- Constructing the DFA of LALR(l) items:
  - Construct from the DFA of LR(l) items by identifying all states that have the same core
  - And forming the union of the lookahead symbols for each LR(0) item

- Each LALR(l) item in this DFA will have an LR(0) item as its first component and a set of lookahead tokens as its second component.

- **Example 5.17** Consider the grammar of Example 5.14.

(1) A→(A)

(2) A→***a***



LR(1) DFA
vs.
LALR(1) DFA
vs.
LR(0) DFA