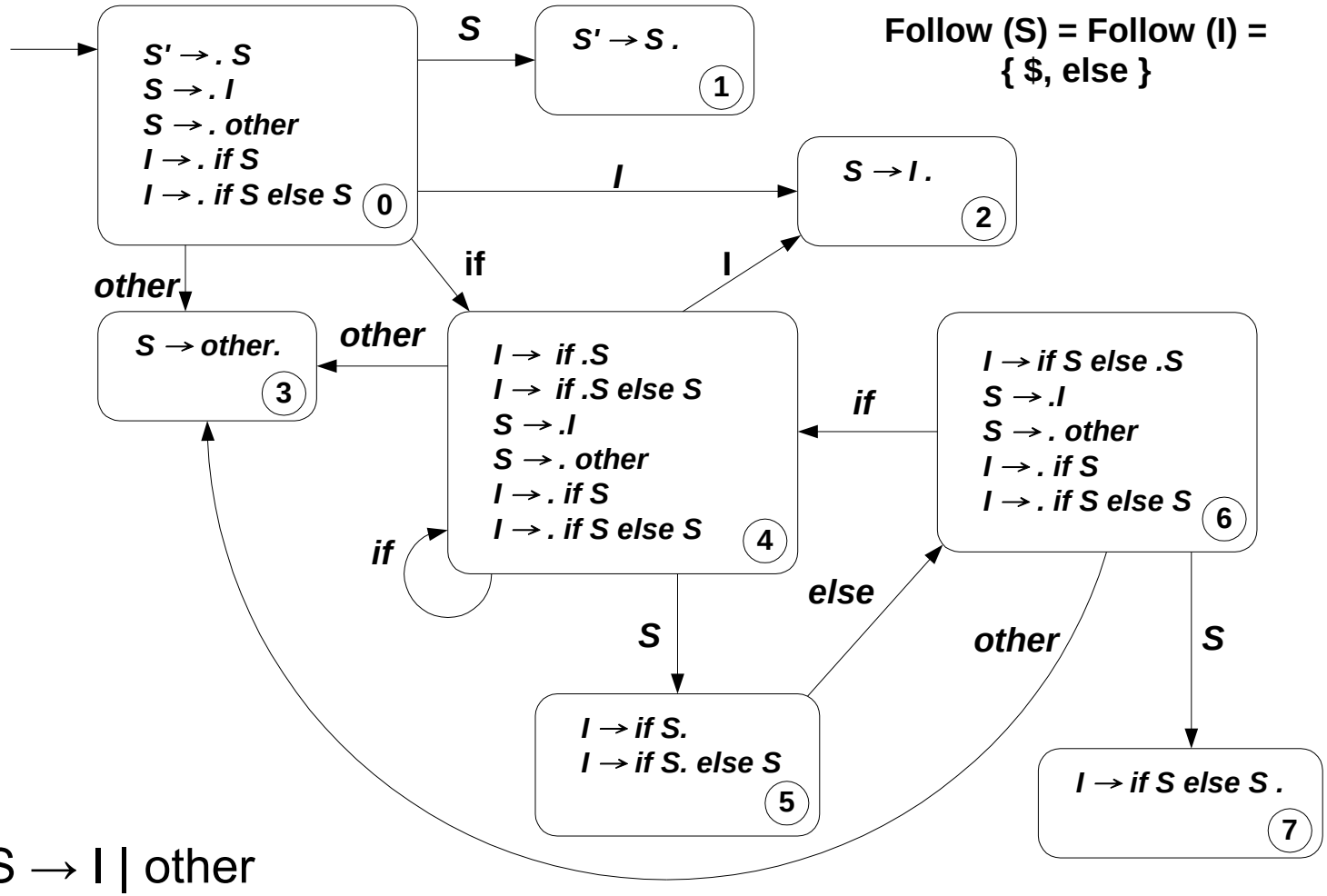
Bottom1



Follow (S) = Follow (I) =
{ $, else }

S → I | other

I → if S | if S else S

5 **Shift / Reduce conflict**

Reduce-Reduce conflict

stmt → call-stmt | assign-stmt

call-stmt → identifier

assign-stmt → var := exp

var → var [exp] | identifier

exp → var | number

==========

S → id | V := E

V → id

E → V | n

==========

Considering

S' → . S

S → . id

S → . V := E

V → . id

This state has a shift transition on id to the state

S → id.

V → id. (* Reduce / Reduce conflict *)

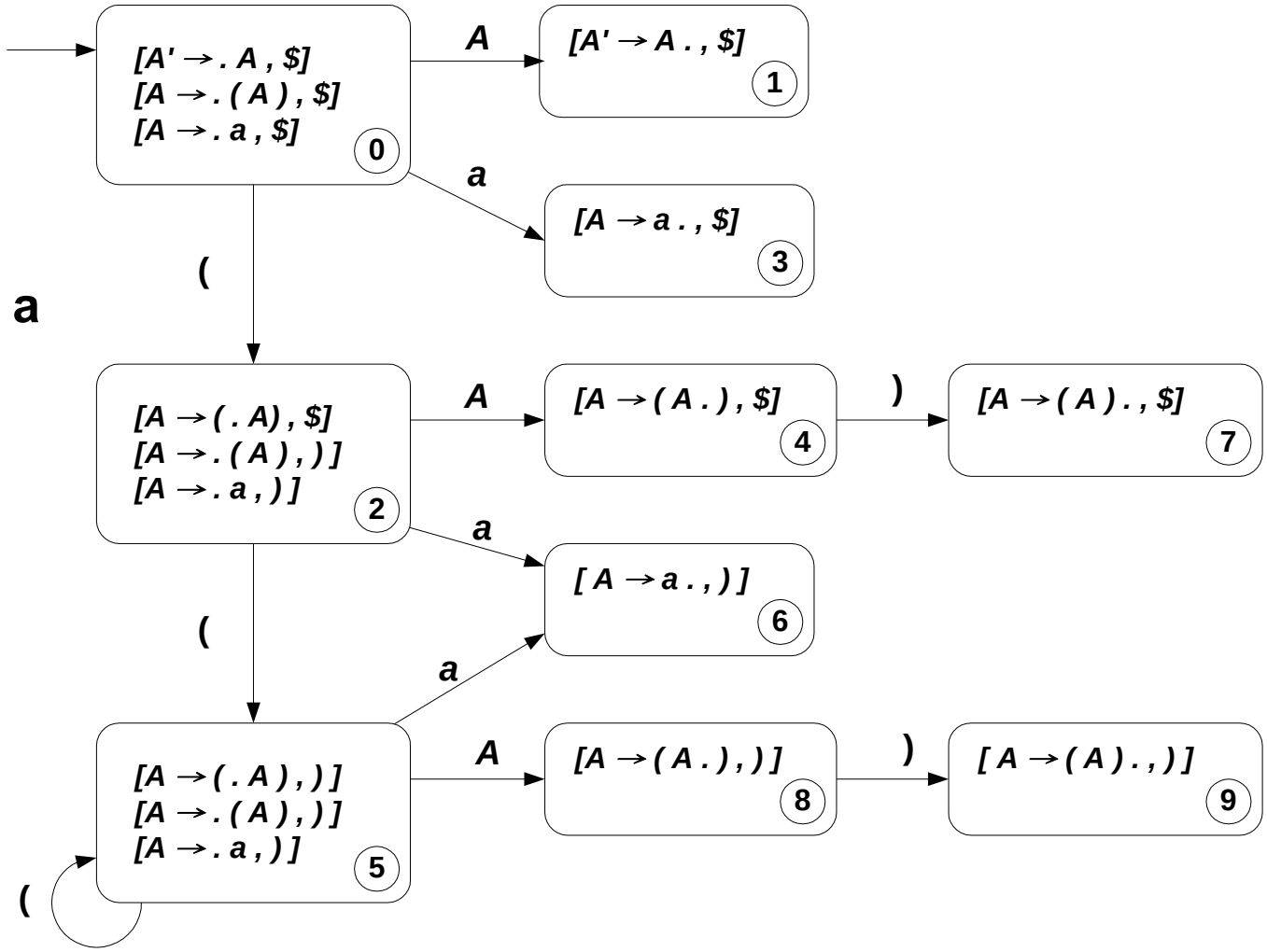
Definition of LR(1) transitions (part 1).

Given an LR(1) item [$A \rightarrow \alpha\ .X\ \gamma$, a], where X is any symbol (terminal), there is a transition on X to the item [$A \rightarrow \alpha\ X.\ \gamma$, a].

Definition of LR(1) transitions (part 2).

Given an LR(1) item [$A \rightarrow \alpha\ .B\ \gamma$, a], where B is a nonterminal, there are ε-transitions to items [B → .β, b] for every production B → β and *for every token b* in First (γa) .

A' → A
A → (A) | a



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(1) item of the form [A → α. X β, 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 → αX. β,a].

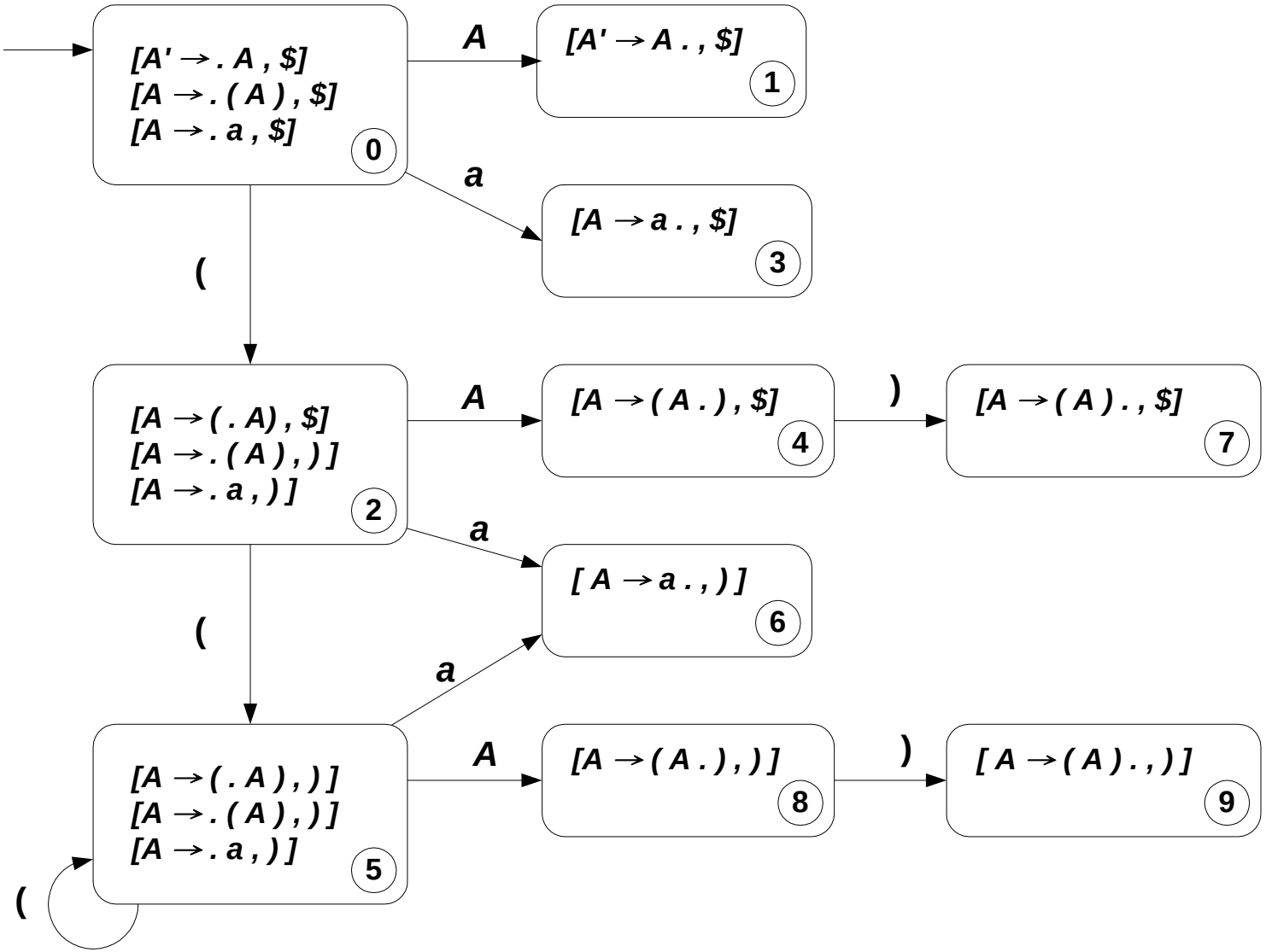
2. If state s contains the complete LR(1) item [A → α. , a], and the next token in the input string is a, then the action is to reduce by the rule A → α. A reduction by the rule S' → S, where S' is the start state, is equivalent to acceptance. (This will happen only if the next input token is $.) In the other cases, the new state is computed as follows. Remove the string α and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of α began. By construction, this state must contain an LR(1) item of the form [B → α. A β, b]. Push A onto the stack, and push the state containing the item [B → αA. β,b].

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

As with the previous methods, we say that a grammar is an LR(1) grammar if the application of the above general LR(1) parsing rules results in no ambiguity. In particular, a grammar is LR(1) if and only if, for any state s, the following two conditions are satisfied:

1. For any item [A → α. X β, a] in s with X a terminal, there is no item in s of the form [B → β. , X] (otherwise there is a shift-reduce conflict).

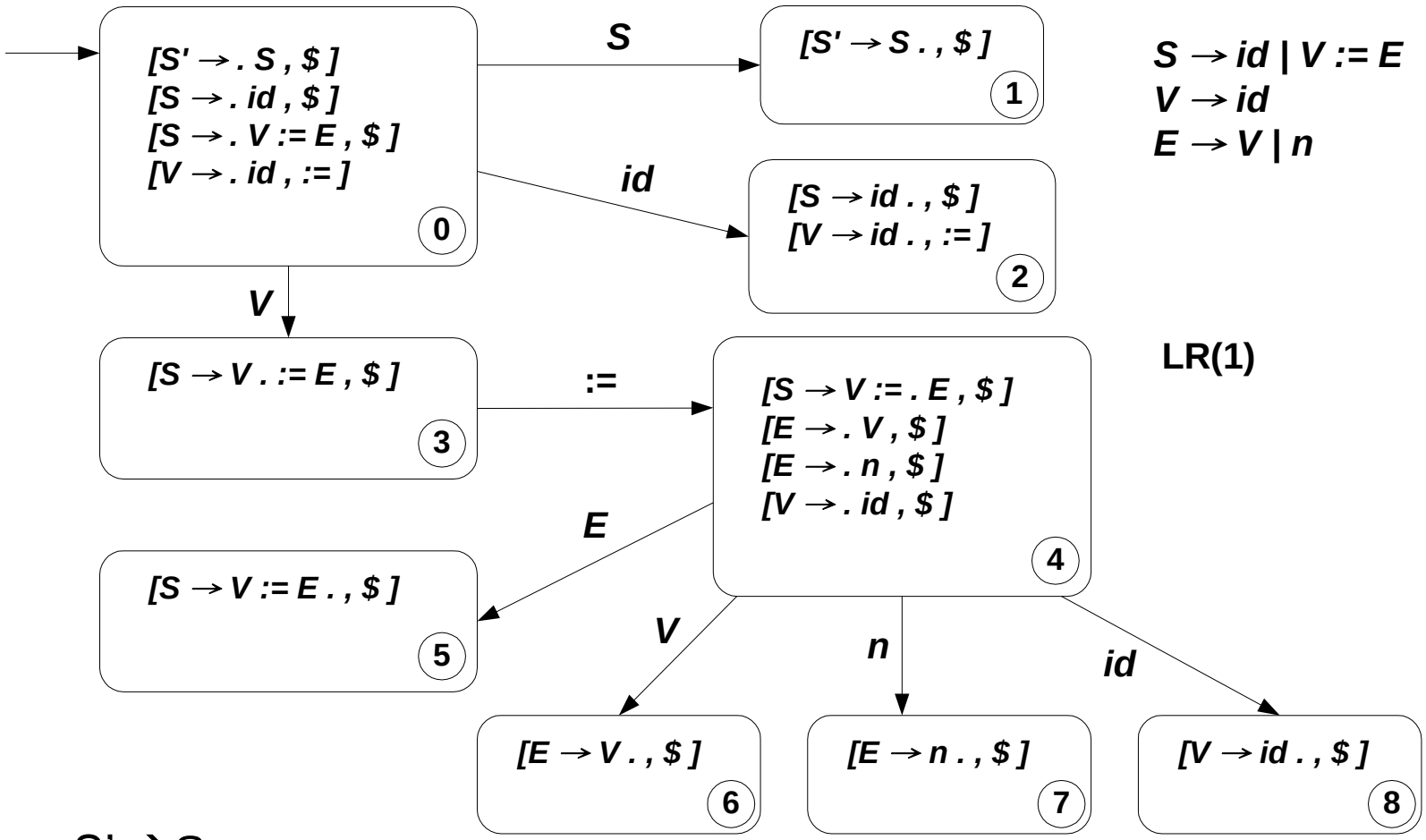
2. There are no two items in s of the form [A → α. , a] and [B → β. , a] (otherwise, there is a reduce-reduce conflict).


[A' → . A , $]
[A → . (A) , $]
[A → . a , $]
0
A
[A' → A . , $]
1
a
[A → a . , $]
3
(
[A → (. A) , $]
[A → . (A) ,)]
[A → . a ,)]
2
A
[A → (A .) , $]
4
)
[A → (A) . , $]
7
a
[A → a . ,)]
6
(
a
[A → (. A) ,)]
[A → . (A) ,)]
[A → . a ,)]
5
A
[A → (A .) ,)]
8
)
[A → (A) . ,)]
9
(

(1) A'→ A
(2) A → (A)
(3) A → a

LR(1)

State	Input				Goto
	(	a	)	$	A
0	s2	s3			1
1				accept	
2	s5	s6			4
3				R3	
4			s7		
5	s5	s6			8
6			R3		
7				R2	
8			s9		
9			R2		



S' →S

LR(1)

State 2 !!

The SLR(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 item of the form $A \rightarrow \alpha \, . \, X \, \beta$, 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 item $A \rightarrow \alpha \, X \, . \, \beta$.

2. If state *s* contains the complete item $A \rightarrow \gamma.$, and the next token in the input string is in Follow(*A*), 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 state, is equivalent to acceptance; this will happen only if the next input token is $. In all other cases, the new state is computed as follows. Remove the string α and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of α began. By construction, this state must contain an item of the form $B \rightarrow \gamma . A \beta$. Push *A* onto the stack, and push the state containing the item $B \rightarrow \alpha A . \beta$.

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

We say that a grammar is an **SLR(1) grammar** if the application of the above SLR(1) parsing rules results in no ambiguity. In particular, a grammar is SLR(1) if and only if, for any states *s*, the following two conditions are satisfied:

1. For any item $A \rightarrow \alpha \,.\, X\, \beta$ in *s* with *X* a terminal, there is no complete item $B \rightarrow \gamma.$ in *s* with *X* in Follow(*B*).

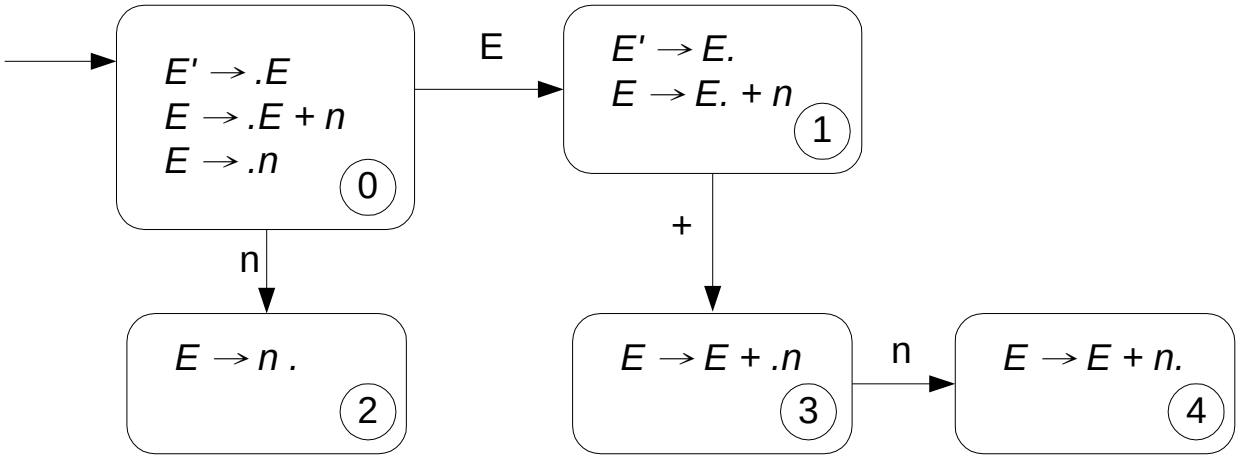
2. For any two complete items $A \rightarrow \alpha.$ and $B \rightarrow \beta.$ in s, Follow(*A*) ∩ Follow(*B*) is empty.

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**.

These two conditions are similar in spirit to the two conditions for LL(1) parsing stated in the previous chapter, except that, as with all shift-reduce parsing methods, decisions on which grammar rule to use can be delayed until the last possible moment, resulting in a more powerful parser.

A parsing table for SLR(1) parsing can also be constructed in a manner similar to that for LR(0) parsing described in the previous section. The differences are as follows.

Since a state can have both shifts and reduces in an SLR(1) parser (depending on the lookahead).



Follow (E') = { $ }; Follow (E) = { $, + }

State	Input			Goto
	n	+	$	E
0	s2			1
*1		s3	accept	
2		r ($E \rightarrow n$)	r ($E \rightarrow n$)	
3	s4			
4		r ($E \rightarrow E + n$)	r ($E \rightarrow E + n$)	

GoTo

	+	n	E	$
0		2	1	
1	3			
2				
3		4		
4				

Action

	+	n	$
0		S	
*1	S		A
2	R3		R3
3		S	
4	R2		R2

	Parsing stack	Input	Action
1	$ 0	$n + n + n$ $	shift 2
2	$ 0 n 2	+ n + n $	reduce $E \rightarrow n$
3	$ 0 E 1	+ n + n $	shift 3
4	$ 0 E 1 + 3	n + n $	shift 4
5	$ 0 E 1 + 3 n 4	+ n $	reduce $E \rightarrow E + n$
6	$ 0 E 1	+ n $	shift 3
7	$ 0 E 1 + 3	n $	shift 4
8	$ 0 E 1+ 3 n 4	$	reduce $E \rightarrow E + n$
9	$ 0 E 1	$	accept

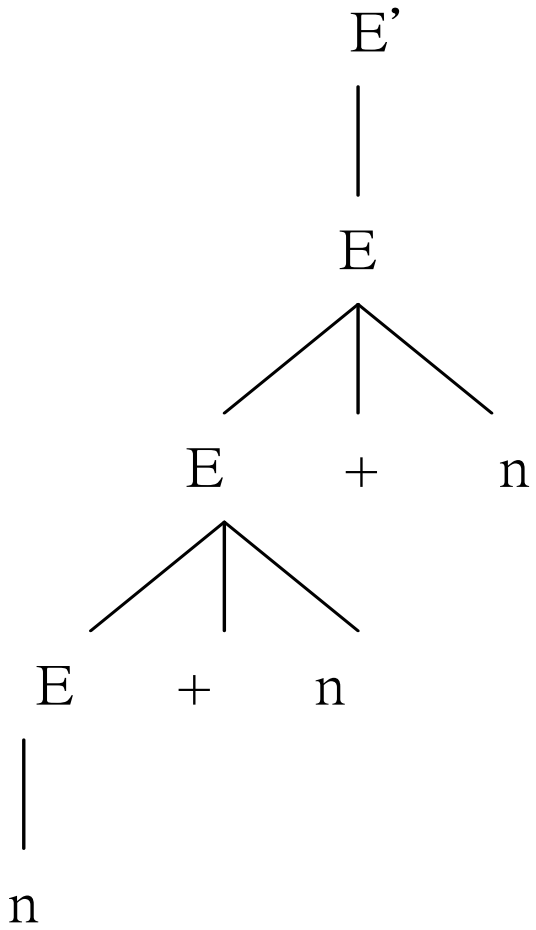
n + n + n

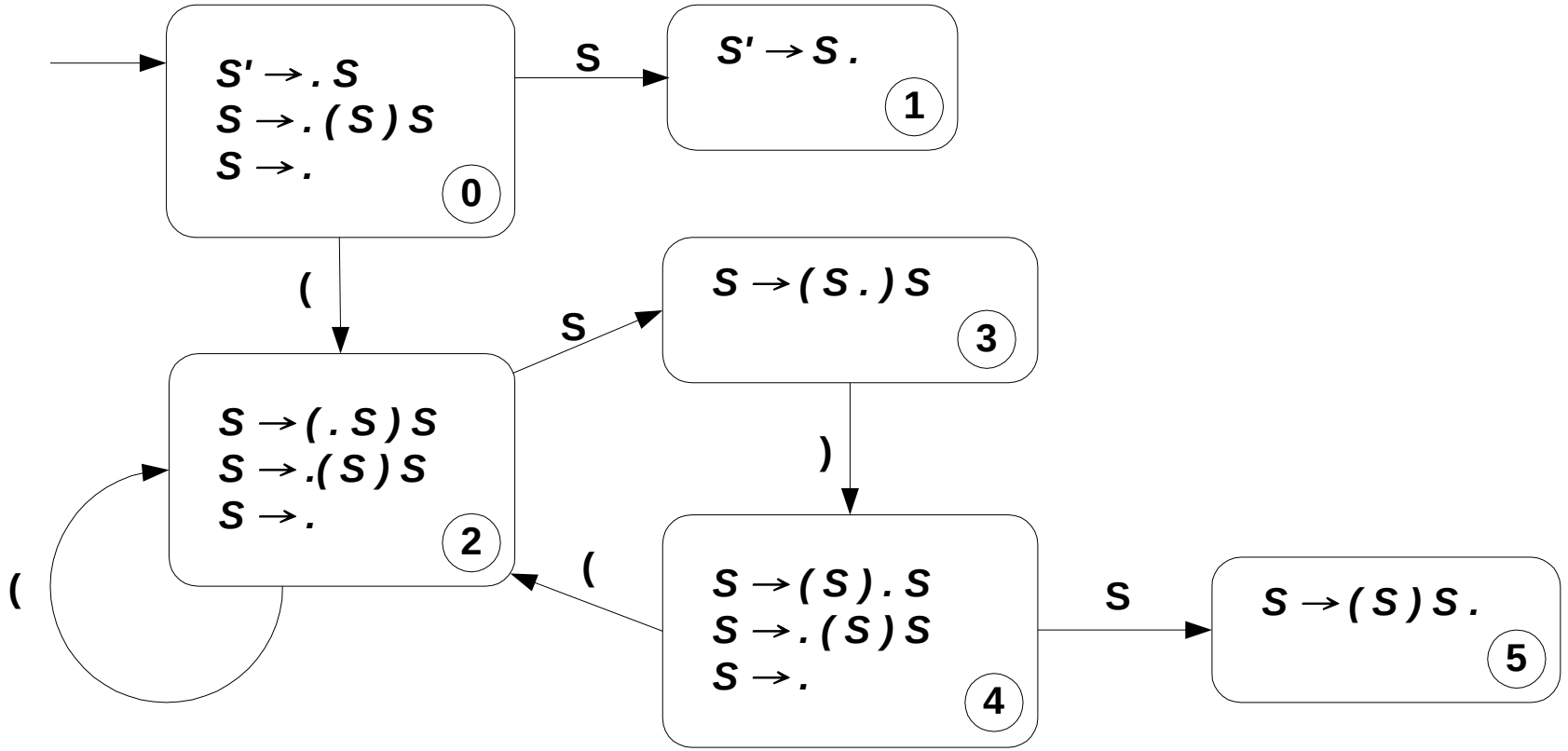
→ E + n + n

→ E + n

→ E

→ E’




S′ → . S
S → . (S) S
S → .
0
S
S′ → S .
1
(
S → (. S) S
S → .(S) S
S → .
2
S
S → (S .) S
3
)
(
S → (S) . S
S → . (S) S
S → .
4
S
S → (S) S .
5
(

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

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)	

Goto

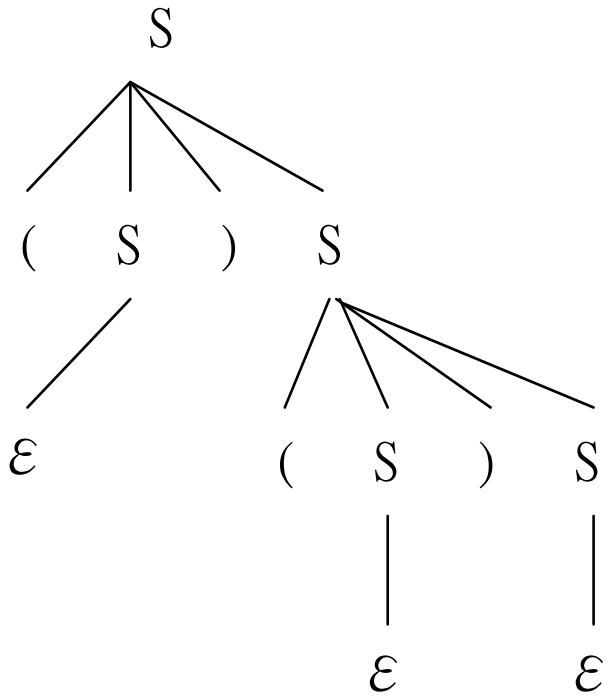
	(	)	S
0	2		1
1			
2	2		3
3		4	
4	2		5
5			

Action

	(	)	$
0	S	R3	R3
1			A
*2	S	R3	R3
3		S	
*4	S	R3	R3
5			R2

	Parsing stack	Input	Action
1	$ 0	( ) ( ) $	shift 2
2	$ 0 (2	) ( ) $	reduce $S \rightarrow \varepsilon$
3	$ 0 (2 *S* 3	) ( ) $	shift 4
4	$ 0 (2 *S* 3) 4	( ) $	shift 2
5	$ 0 (2 *S* 3) 4 (2	) $	reduce $S \rightarrow \varepsilon$
6	$ 0 (2 *S* 3) 4 (2 *S* 3	) $	shift 4
7	$ 0 (2 *S* 3) 4 (2 S 3) 4	$	reduce $S \rightarrow \varepsilon$
8	$ 0 (2 S 3) 4 (2 *S* 3) 4 *S* 5	$	reduce $S \rightarrow (S) S$
9	$ 0 (2 *S* 3) 4 *S* 5	$	reduce $S \rightarrow (S) S$
10	$ 0 *S* 1	$	accept

( ) ( )
→ (S) ( )
→ (S) (S)
→ (S) (S) S
→ (S) S
→ S



statement → *if-stmt* | ***other***

if-stmt → ***if*** *(exp)* *statement*

| ***if*** *(exp)* *statement* ***else*** *statement*

exp → ***0*** | ***1***

S → *I* | ***other***

I → ***if*** *S* | ***if*** *S* ***else*** *S*

(1) *S* → *I*

(2) *S* → ***other***

(3) *I* → ***if*** *S*

(4) *I* → ***if*** *S* ***else*** *S*

State	Input				Goto	
	if	**else**	**other**	$	*S*	*l*
0	s4		s3		1	2
1				accept		
2		r1		r1		
3		r2		r2		
4	s4		s3		5	2
5		s6		r3		
6	s4		s3		7	2
7		r4		r4		

SLR(1) -- refer page 2 (shift-reduce conflict in state 5 has been resolved)

HOWEVER➔ Limits of SLR(1) Parsing Power:

stmt → call-stmt | assign-stmt
call-stmt → identifier
assign-stmt → var := exp
var → var [exp] | identifier
exp → var | number

This grammar models statements which can be either calls to parameterless procedures, or assignments of expressions to variables. Note that both assignments and procedure calls begin with an identifier. It is not until either the end of the statement or the token := is seen that a parser can decide whether the statement is an assignment or a call.

S → id | V := E
V → id
E → V | n

To show how this grammar results in a parsing conflict in SLR(1) parsing, consider the start state of the DFA of sets of items:

S' → . S
S → . id
S → . V := E
V → . Id

This state has a shift transition on id to the state

S → id.
V → id. (* Reduce / Reduce conflict *)

Now, Follow(S) = {$} and Follow(V) = {:=, $}
(:= because of the rule V → V := E, and $ because an E can be a V). Thus, the SLR(1) parsing algorithm calls for a reduction in this state by both the rule S → id and the rule V → id under input symbol $. (This is a reduce-reduce conflict.) This parsing conflict is actually a “phony” problem caused by the weakness of the SLR(1) method. Indeed, the reduction by V → id should never be made in this state when the input is $, since a variable can never occur at the end of a statement until after the token := is seen and shifted.

LALR(1)

FIRST PRINCIPLE OF LALR(1) PARSING

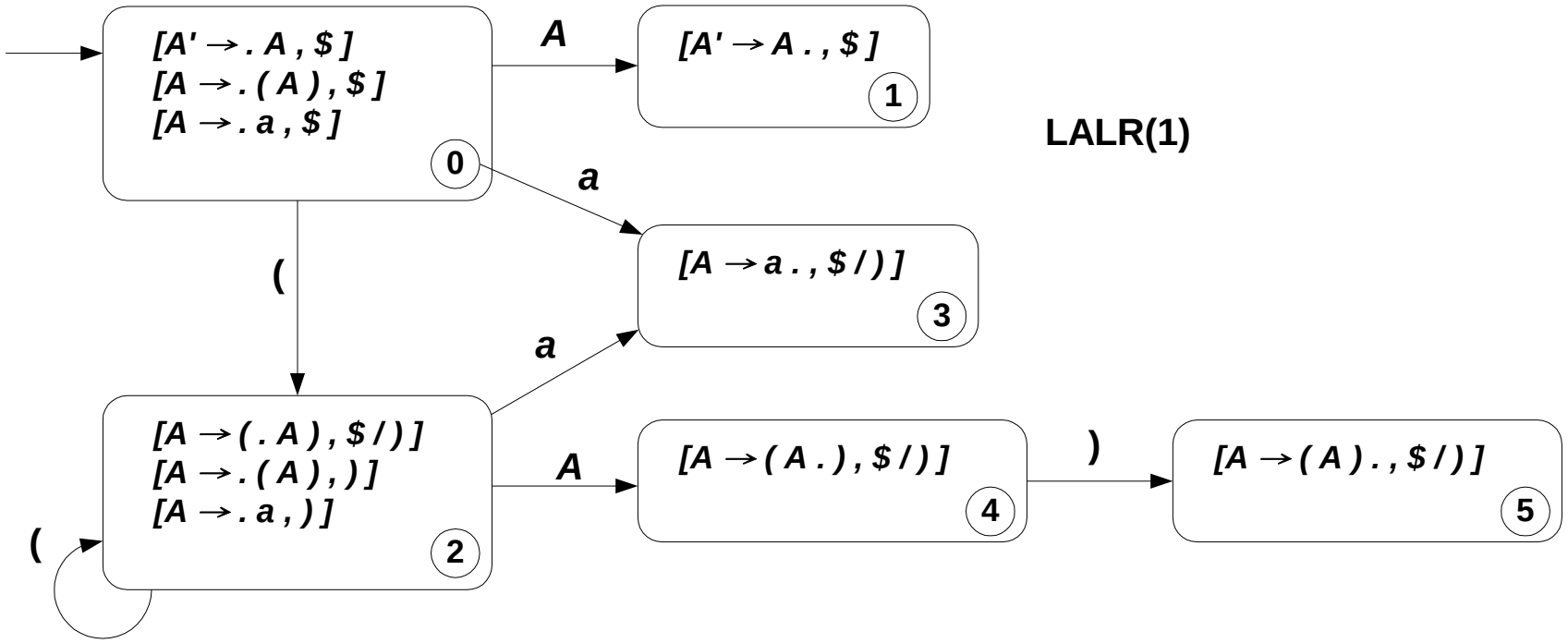
The core of a state of the DFA of LR(1) items is a state of the DFA of LR(0) items.

SECOND PRINCIPLE OF LALR(1) PARSING

Given two states s1 and s2 of the DFA of LR(1) items that have the same core, suppose there is a transition on the symbol X from s1 to a state t1. Then there is also a transition on X from state s2 to a state t2, and the states t1 and t2 have the same core.

The algorithm for LALR(1) parsing using the condensed DFA of LALR(1) items is identical to the general LR(1) parsing algorithm described in the previous section. As before, we call a grammar an LALR(1) grammar if no parsing conflicts arise in the LALR(1) parsing algorithm. It is possible for the LALR(1) construction to create parsing conflicts that do not exist in general LR(1) parsing, but this rarely happens in practice.

Indeed, if a grammar is LR(1), then the LALR(1) parsing table cannot have any shift-reduce conflicts; there may be reduce-reduce conflicts, however (see the Exercises). Nevertheless, if a grammar is SLR(1), then it certainly is LALR(1), and LALR(1) parsers often do as well as general LR(1) parsers in removing typical conflicts that occur in SLR(1) parsing. For example, the non-SLR(1) grammar of Example 5.16 is LALR(1): the DFA of LR(1) items of Figure 5.8 is also the DFA of LALR(1) items. If, as in this example, the grammar is already LALR(1), the only consequence of using LALR(1) parsing over general LR parsing is that, in the presence of errors, some spurious reductions may be made before error is declared. For example, we see from Figure 5.9 that, given the erroneous input string a), an LALR(1) parser will perform the reduction A → a before declaring error, while a general LR(1) parser will declare error immediately after a shift of the token a.



LALR(1)

new state 2: merge old states 2 and 5

new state 3: merge old states 3 and 6 (refer page 5)

new state 4: merge old states 4 and 8

new state 5: merge old states 7 and 9