Bottom-Up Parsers

Lecture Nine

Bottom-Up Parsing

- Bottom-up parsing algorithms are in general more powerful than top-down methods. (For example, left recursion is not a problem in bottom-up parsing.)
- Almost all practical programming languages have an LR(1) grammar.
- All of the important bottom-up methods are really too complex for hand coding.
- Begin at the leaves, build the parse tree in small segments, combine the small trees to make bigger trees, until the root is reached.
- This process is called reduction of the sentence to the start symbol of the grammar

Bottom-Up Parsing

- The most general bottom-up algorithm is called LR(1) parsing:
  - the L indicates that the input is processed from left to right.
  - the R indicates that a rightmost derivation is produced.
  - the number 1 indicates that one symbol of lookahead is used.
- LR(0) parsing: no lookahead is need in making parsing decisions. (This is possible because a lookahead token can be examined after it appears on the parsing stack, and if this happens it does not count as lookahead.)
- SLR(1) (for simple LR(1) parsing) : An improvement on LR(0) parsing.
- LALR(1) parsing (for lookahead LR(1) parsing):A method that is slightly more powerful than SLR(1) parsing but less complex than general LR(1) parsing.

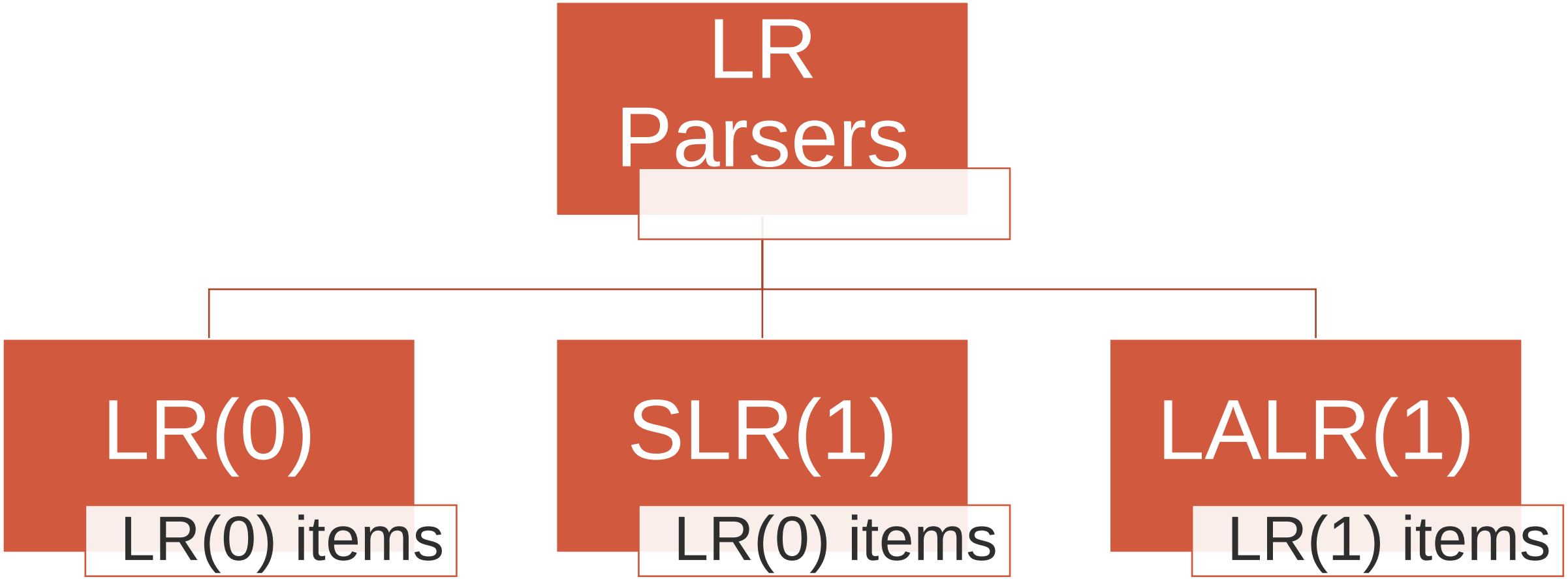
Bottom-Up Parsing

- A bottom-up parser uses an explicit stack to perform a parse similar to a nonrecursive top-down parser.
- The parsing stack will contain both tokens and nonterminals, and also 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 bottom-up parser has two possible actions (besides "accept"):

1. Shift a terminal from the front of the input to the top of the stack.

2. Reduce a string a at the top of the stack α to a nonterminal A. Given the Grammar Rule $A \rightarrow \alpha$

A bottom-up parser is thus sometimes called a shift-reduce parser.


LR
Parsers
LR(0)
LR(0) items
SLR(1)
LR(0) items
LALR(1)
LR(1) items

Bottom-Up Parsing Example(1)

$$S \rightarrow (S) \quad S \mid \varepsilon$$

- Grammars are always augmented with a new start symbol. This means that if S is the start symbol, a new start symbol S' is added to the grammar, with a single unit production to the previous start symbol.

$$S' \rightarrow S$$

$$S \rightarrow (S) \quad S \mid \varepsilon$$

Bottom-Up Parsing Example(2)

$S' \rightarrow S$

$S \rightarrow (S) \quad S \mid \varepsilon$

	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

Bottom-Up Parsing

- A bottom-up parser can shift input symbols onto the stack until it determines what action to. However, a bottom-up parser may need to look deeper into the stack than just the top in order to determine what action to perform. For example, the previous table, line 5 has S on the top of the stack, and the parser performs a reduction by the production S → (S) S, while line 6 also has S on the top of the stack, but the parser performs a reduction by S'→S.
- To be able to know that S → (S) S is a valid reduction at step 5, we must know that the stack actually does contain the string (S) S at that point. Thus, bottom-up parsing requires arbitrary "stack lookahead.“
- 'Ihis is not nearly as serious as input lookahead, since the parser itself builds the stack and can arrange for the appropriate information to be available. The mechanism that will do this is a deterministic finite automaton of "items"

LR(0)items

- An LR(0) item of a context-free grammar is a production choice with a distinguished position in its right-hand side. We will indicate this distinguished position by a period (which, of course, becomes a meta symbol, not to be confused with an actual token).
- if $A \rightarrow \propto$ 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 $A \rightarrow \beta.\gamma$ is an LR(0) item. These are called LR(0) items because they contain no explicit reference to lookahead.

LR(0) items

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

This grammar has three production choices and eight items:

$$S' \rightarrow .S$$
$$S' \rightarrow S.$$
$$S \rightarrow .(S)S$$
$$S \rightarrow (.S)S$$
$$S \rightarrow (S.)S$$
$$S \rightarrow (S).S$$
$$S \rightarrow (S)S.$$
$$S \rightarrow .$$

LR(0) items

- The idea behind the concept of an item is that an item records an intermediate step in the recognition of the right-hand side of a particular grammar rule choice.
- In the grammar rule choice $A \rightarrow \propto with\ \alpha = \beta\gamma.$

$$A \rightarrow \beta.\gamma$$

means that β has already been seen and that it may be possible to derive the next input tokens from γ

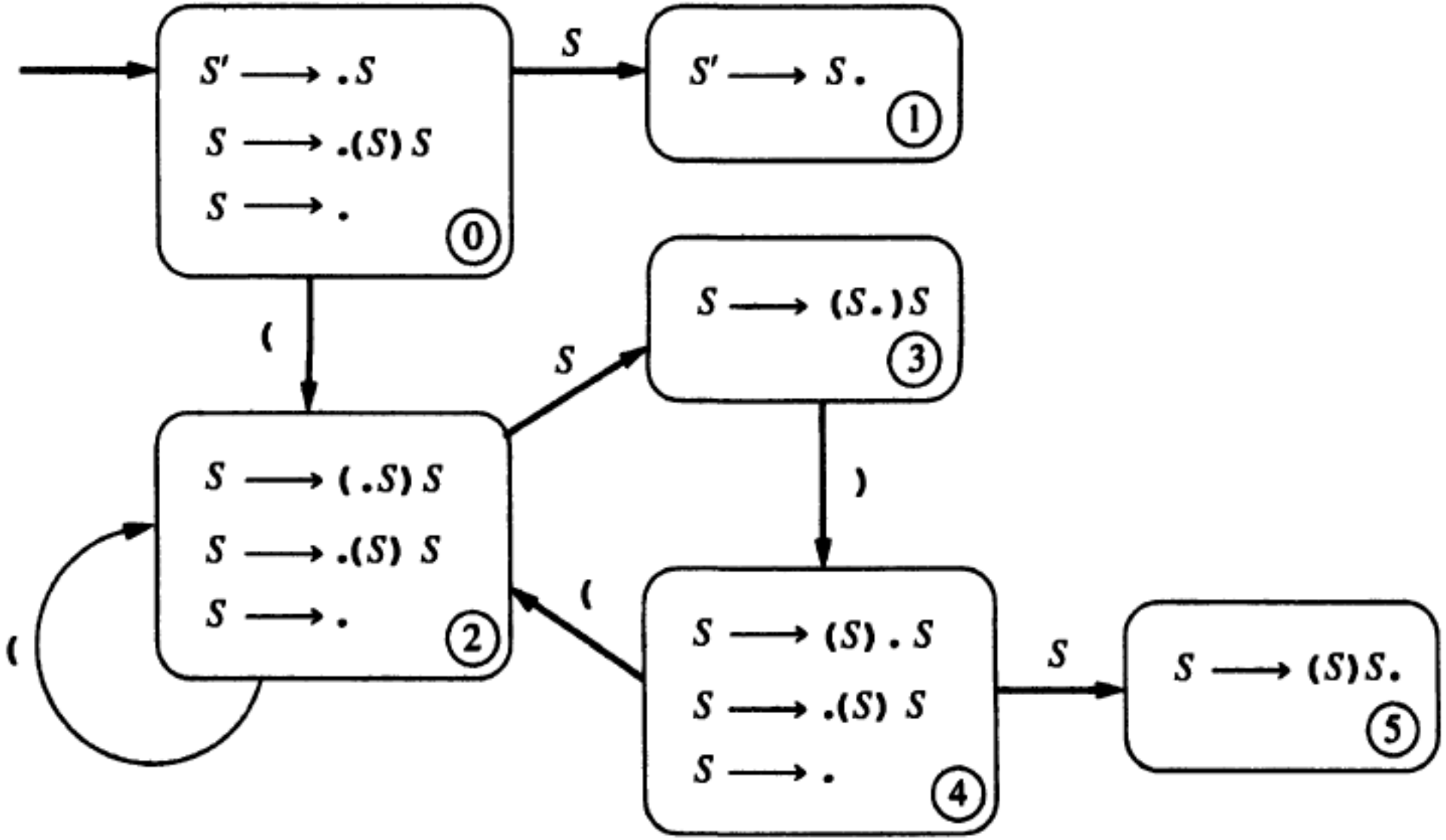
- In terms of the parsing stack, this means that β must appear at the top of the stack.

Finite Automata of LR(0) items and LR(0) Parsing

- 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 a shift-reduce parse. This will start out as a nondeterministic finite automaton. From this NFA of LR(0) items we can construct the DFA of sets of LR(0) items using the subset construction algorithm. It is also easy to construct the DFA of sets of LR(O) items directly.

Finite Automata of LR(0) items and LR(0) Parsing

- DFA



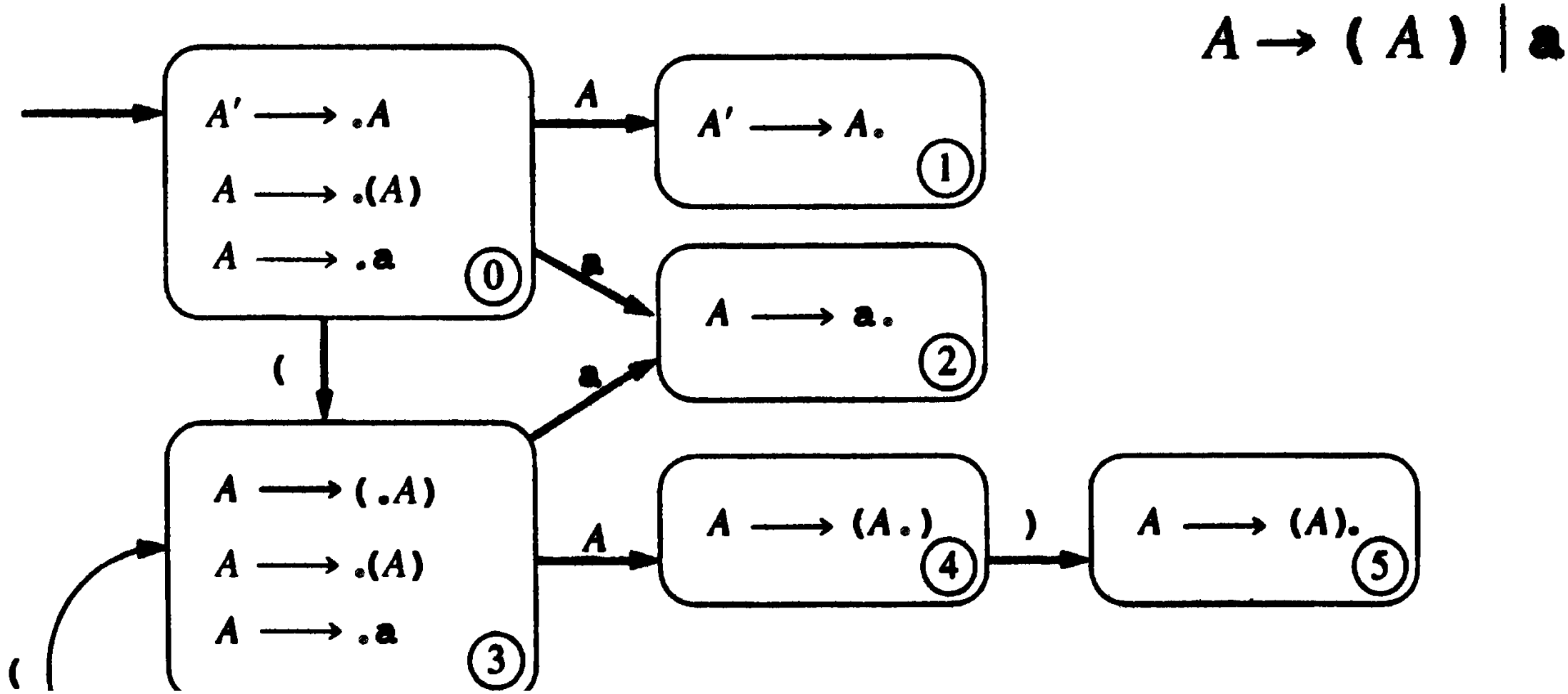
Finite Automata of LR(0) items and LR(0) Parsing

- **Closure Items**: items are added to a state during the e-closure step.
- **kernel items** : items are originated from the state of non-ε-transitions.
- All closure items are initial items
- The kernel items uniquely determine the state and its transitions. Thus, only kernel items need to be specified to completely characterize the DFA of sets of items. Parser generators that construct the DFA may, therefore, only report the kernel items (this is true of Yacc, for instance).

LR(0)Grammar

A grammar is said to be an **LR(0) grammar** if the above rules are unambiguous. This means that if a state contains a complete item $A \rightarrow \alpha.$, then it can contain no other items. Indeed, if such a state also contains a "shift" item $A \rightarrow \alpha . X \beta$ (X a terminal), then an ambiguity arises as to whether action (1) or action (2) is to be performed. This situation is called a **shift-reduce conflict**. Similarly, if such a state contains another complete item $B \rightarrow \beta.$, then an ambiguity arises as to which production to use for the reduction ($A \rightarrow \alpha$ or $B \rightarrow \beta$). This situation is called a **reduce-reduce conflict**. Thus, a grammar is LR(0) if and only if each state is a shift state (a state containing only "shift" items) or a reduce state containing a single complete item.

LR(0)Example DFA



LR(0)Example(2)

	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 $A \rightarrow$ a
5	$ 0 (3 (3 A 4	)) $	shift
6	$ 0 (3 (3 A 4) 5	) $	reduce $A \rightarrow$ (A)
7	$ 0 (3 A 4	) $	shift
8	$ 0 (3 A 4) 5	$	reduce $A \rightarrow$ (A)
9	$ 0 A 1	$	accept

LR(0)-Parsing table

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