Chapter 5
Bottom-Up Parsing

2022 Spring&Summer

Outline

A more powerful parsing technology

LR grammars -- more expressive than LL

- Construct right-most derivation of program
- Left-recursive grammars, virtually all programming languages
- Easier to express programming language syntax

Shift-reduce parsers

- Parsers for LR grammars
- Automatic parser generators (e.g., yacc, CUP)

5.2.3 The LR(0) Parsing Algorithm

- The parsing stack to store: *symbols* and *state numbers*.
- Pushing the new *state number* onto the parsing stack after each push of *a symbol*.

Parsing stack	Input
$0	Input String
$0 n2	Rest input string $

Parsing stack	Symbol stack	Input
0	$	Input String
02	$n	Rest input string $

5.2.3 The LR(0) Parsing Algorithm

- Definition

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 $A \rightarrow \alpha{\cdot}X\beta$ (X is a terminal). Then the action is to *shift* the current input token on to the stack.

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 $A \rightarrow \gamma{\cdot}$

- A *reduction* by the rule $S' \rightarrow S$, where S' is the start state,
- *Acceptance* if the input is empty
- *Error* if the input is not empty.

5.2.3 The LR(0) Parsing Algorithm

- A grammar is said to be LR(0) grammar if the above rules are unambiguous.
- A grammar is LR(0) ***if and only if***
  - Each state is a shift state(a state containing only “shift” items)
  - A reduce state containing a single complete item.

5.2.3The LR(0) Parsing Algorithm

Grammar:

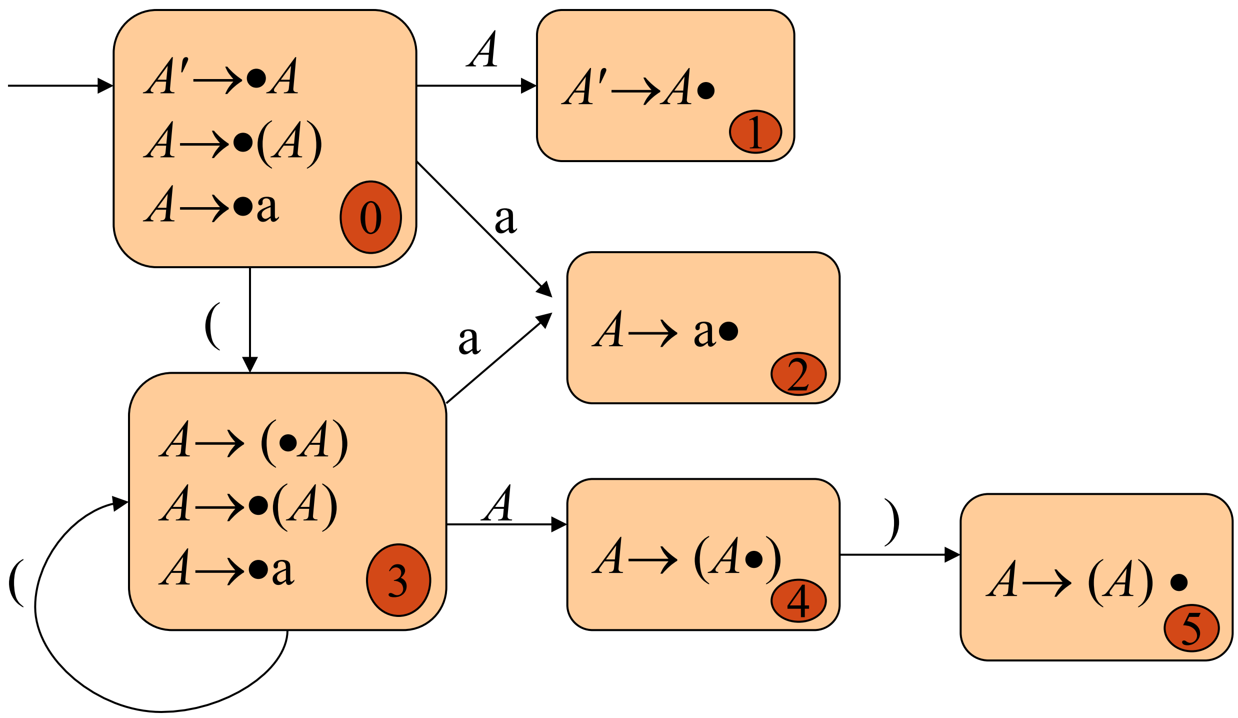
$S \rightarrow (\ L\ )\ |\ id$

$L \rightarrow S\ |\ L,\ S$

Derivation	stack	input	action
((a),b) ⇐	1	((a),b)	shift, goto 3
((a),b) ⇐	13	(a),b)	shift, goto 3
((a),b) ⇐	133	a),b)	shift, goto 2
((a),b) ⇐	1332	),b)	reduce S→id
((S),b) ⇐	1337	),b)	reduce L→S
((L),b) ⇐	1335	),b)	shift, goto 6
((L),b) ⇐	13356	,b)	reduce S→(L)
(S,b) ⇐	137	,b)	reduce L→S
(L,b) ⇐	135	,b)	shift, goto 8
(L,b) ⇐	1358	b)	shift, goto 9
(L,b) ⇐	13582	)	reduce S→id
(L,S) ⇐	13589	)	reduce L→L , S
(L) ⇐	135	)	shift, goto 6
(L) ⇐	1356		reduce S→(L)
S	14		done

5.2.3 The LR(0) Parsing Algorithm

- Example 5.9 Consider the grammar $A \to (\, A\,) \mid a$



5.2.3The LR(0) Parsing Algorithm

	Parsing stack	Input	Action
1	\$0	((a))\$	Shift
2	\$0 (3	(a))\$	Shift
3	\$0 (3 (3	a))\$	Shift
4	\$0 (3 (3 a2	))\$	Reduce $A \rightarrow a$
5	\$0 (3 (3 A4	))\$	Shift
6	\$0 (3 (3 A4)5	)\$	Reduce $A \rightarrow (A)$
7	\$0 (3 A4	)\$	Shift
8	\$0 (3 A4)5	\$	Reduce $A \rightarrow (A)$
9	\$0 A1	\$	Accept

5.2.3 The LR(0) Parsing Algorithm

- The DFA of sets of items and the actions : be combined into a parsing table.
- The LR(0) parsing becomes *a table-driven parsing* method.
- The table rows labeled with the states of the DFA.
- The columns to be labeled with "*shift*" and "*reduce*".

State	Action	Rule	Input			Goto
			(	a	)	A
0	Shift		3	2		1
1	Reduce	$A' \to A$				
2	Reduce	$A \to (A)$				
3	Shift		3	2		4
4	Shift				5	
5	Reduce	$A \to a$				

5.3 SLR(1) Parsing

Definition:

The SLR(1) parsing algorithm.

1. If state s contains any item of form $A \rightarrow \alpha{\cdot}X\beta$, 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 $A \rightarrow \alpha X{\cdot}\beta$.
2. If state s contains the complete item $A \rightarrow \gamma{\cdot}$, and the next token in the input string is in Follow(A), then the action is to reduce by the rule $A \rightarrow \gamma$.

5.3.1 The SLR(1) Parsing Algorithm

- A reduction by the rule $S' \rightarrow S$, where S' is the start state, this will happen only if the next input token is $\$$.
- 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.
- This state must contain an item of the form $B \rightarrow \alpha \cdot A\beta$. Push A onto the stack, and push the state containing the item $B \rightarrow \alpha A \cdot \beta$.

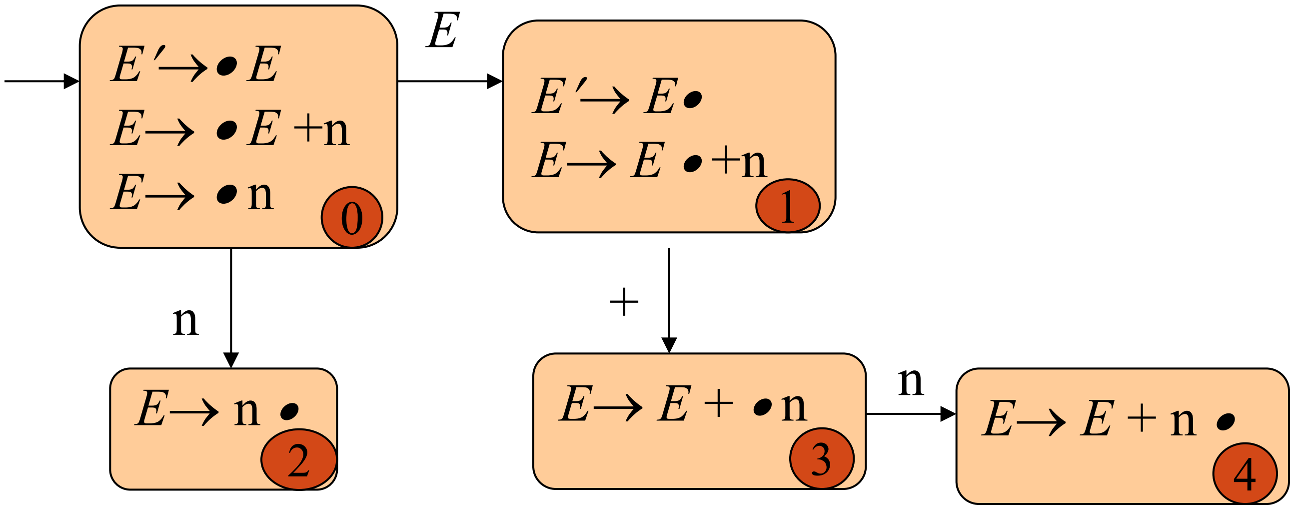
3. If the next input token is such that neither of the above two cases applies, an error is declared.

5.3.1 The SLR(1) Parsing Algorithm

- A grammar is an SLR(l) grammar: the SLR(1) parsing rules results in no ambiguity.
- A grammar is SLR(1) if and only if, for any state s, the following two conditions are satisfied:
  1. 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).
  2. For any two complete items $A \rightarrow \alpha\cdot$ and $B \rightarrow \beta\cdot$ in s, Follow(A) $\cap$ Follow(B) is empty.

5.3.1 The SLR(1) Parsing Algorithm

- Example 5.10



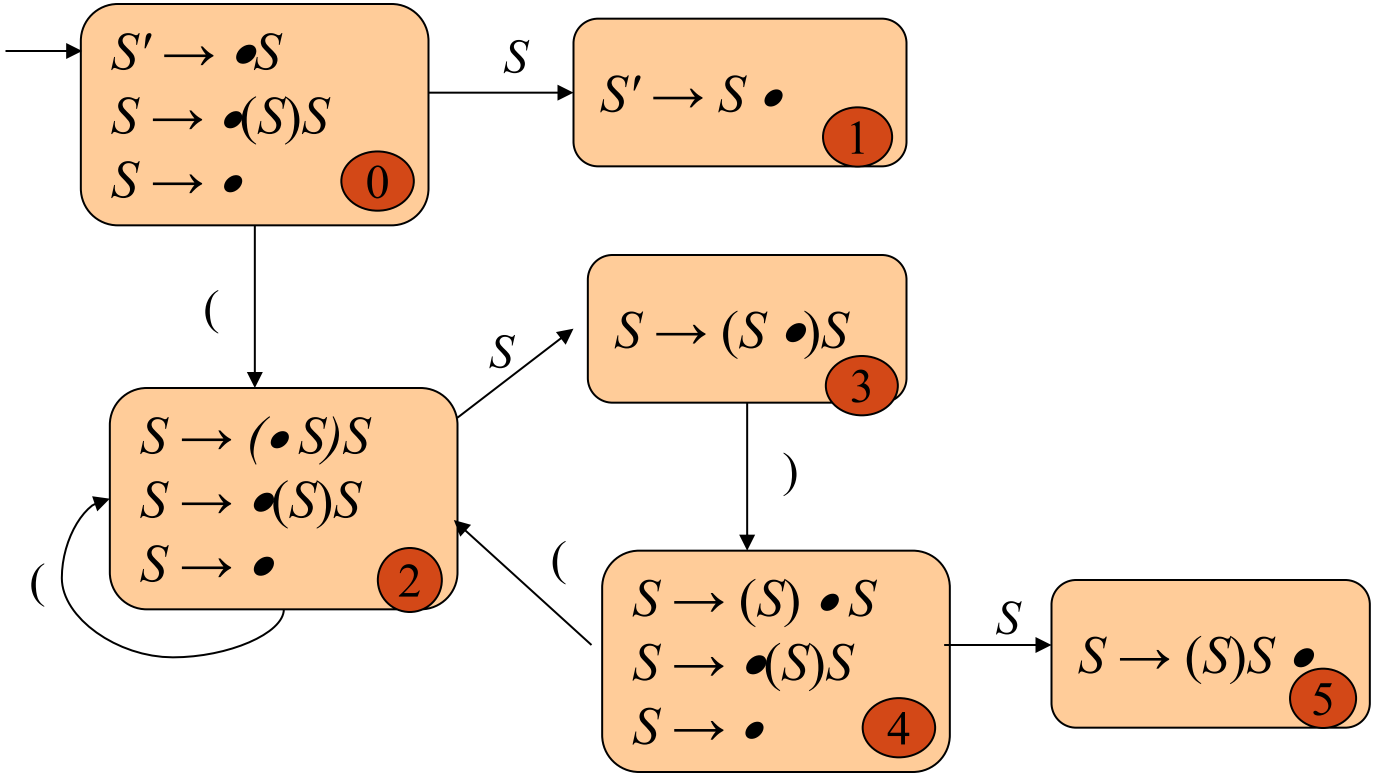
This grammar is not *LR(0)*, but it is *SLR(l)*.

Follow(E') ={$ } and Follow(E) = { $, +} .

5.3.1 The SLR(1) Parsing Algorithm

- Example 5. 11

The grammar of balanced parentheses, Follow(S') = {\$} and Follow(S)={\$,)}. Note how the non-LR(0) states 0, 2, and 4 have both shifts and reductions by the ε-production.



5.3.1 The SLR(1) Parsing Algorithm

- Note how the stack continues to grow until the final reductions.
- This is characteristic of bottom-up parsers in the presence of right-recursive rules such as $S \rightarrow (S)S$.
- Right recursion can cause stack overflow, and so is to be avoided if possible.

5.3.2 Disambiguating Rules for Parsing Conflicts

- Two kinds of parsing conflicts in SLR(1) parsing
  shift-reduce conflicts
  reduce-reduce conflicts.
- In the case of shift-reduce conflicts, there is a natural *disambiguating rule*: always prefer the *shift* over the *reduce*.
- The case of reduce-reduce conflicts is more difficult
  Such conflicts often (but not always) indicate an error in
  the design of the grammar.

5.3.2 Disambiguating Rules for Parsing Conflicts

- Example 5.12

  statement $\rightarrow$ *if-stmt*| *other*

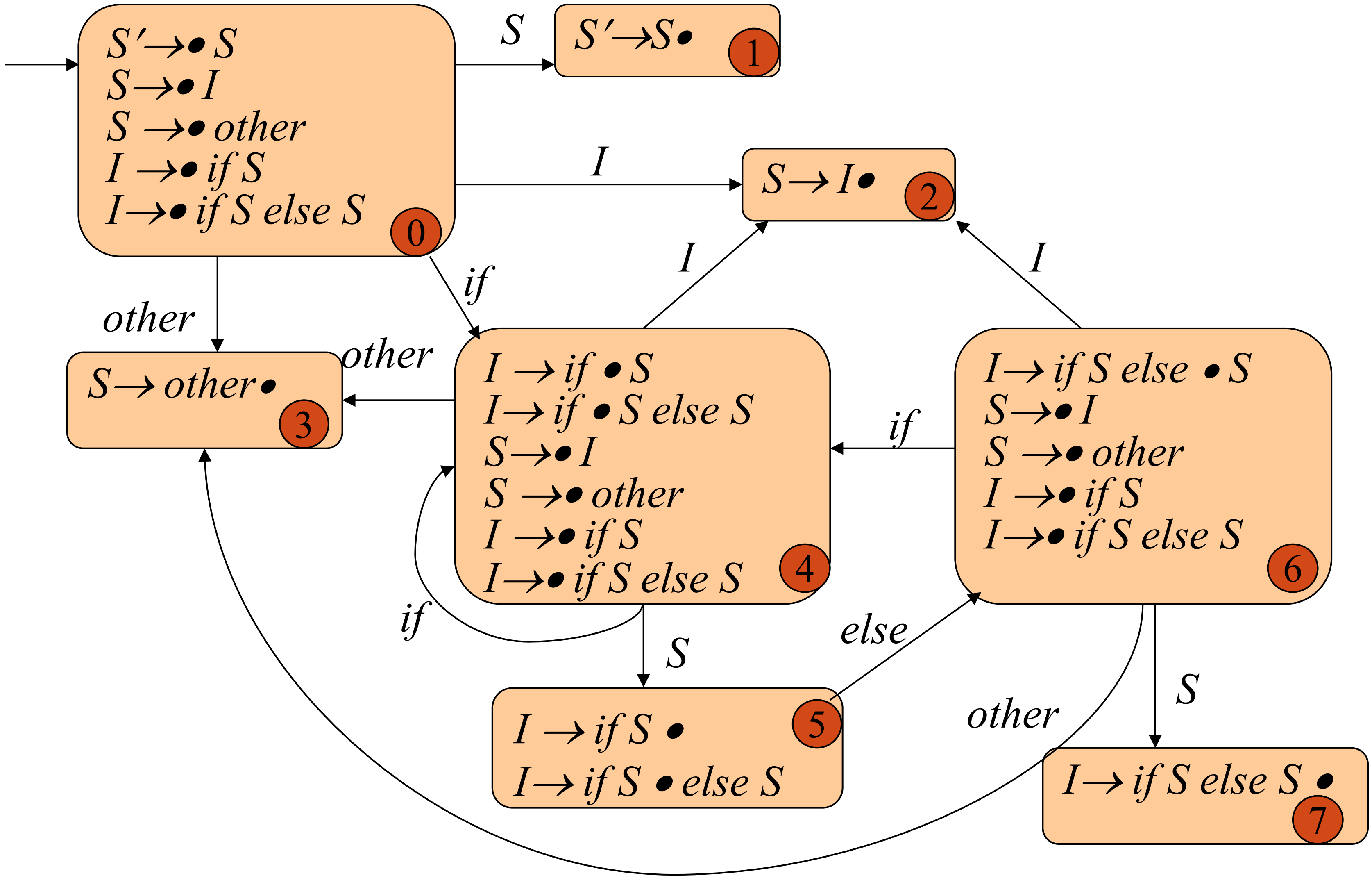
  if-stmt $\rightarrow$ *if (exp) statement* |*if (exp) statement else statement*

  exp $\rightarrow$ 0|1

  S $\rightarrow$ *I*| *other*

  I $\rightarrow$ *if S* | *if S else S*

5.3.2 Disambiguating Rules for Parsing Conflicts



- The conflict occurs in states 5 of the DFA.
- A disambiguating rule: *prefers the shift over the reduce*.

5.3.3 Limits of SLR(1) Parsing Power

- SLR(1) parsing is not quite powering enough

  Example 5. 13 :the statements extracted and simplified from Pascal (a similar situation occurs in C)

  stmt → *call-stmt* | *assign-stmt*

  call-stmt → ***identifier***

  assign-stmt → *var* **:=***exp*

  var →*var* **[** *exp* **]** | ***identifier***

  exp → *var* | ***number***

5.3.3 Limits of SLR(1) Parsing Power

$S' \rightarrow \cdot S$
$S \rightarrow \cdot \textbf{\textit{id}}$
$S \rightarrow \cdot V \textbf{:=} E$
$V \rightarrow \cdot \textbf{\textit{id}}$

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

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

Follow(S)={$} Follow(V)={:=, $}

The SLR(l) parsing algorithm calls for a reduction :

the rule $S \rightarrow$ ***id*** and the rule $V \rightarrow$ ***id*** under

input symbol $.

(This is a *reduce-reduce* conflict.)

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

- The difficulty with the SLR(1) method :
  Applies lookaheads after the construction of the DFA of LR(0) items
- The power of the general LR(1) method :
  It uses a new DFA that has the lookaheads built into its construction from the start.
- An LR(1) item is a pair consisting of an LR(0) item and a *lookahead* token.

5.4.1 Finite Automata of LR(1) Items

- Write LR(1) items using square brackets as

  $$[A \rightarrow \alpha.\beta,\ \ a]$$

  where $A \rightarrow \alpha\beta$ is an LR(0) item and a is a token (lookahead).
- The major difference between the *LR(0)* and *LR(1)* automata:

  Definition of the ε-transitions.

5.4.1 Finite Automata of LR(1) Items

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

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

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

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

5.4.1 Finite Automata of LR(1) Items

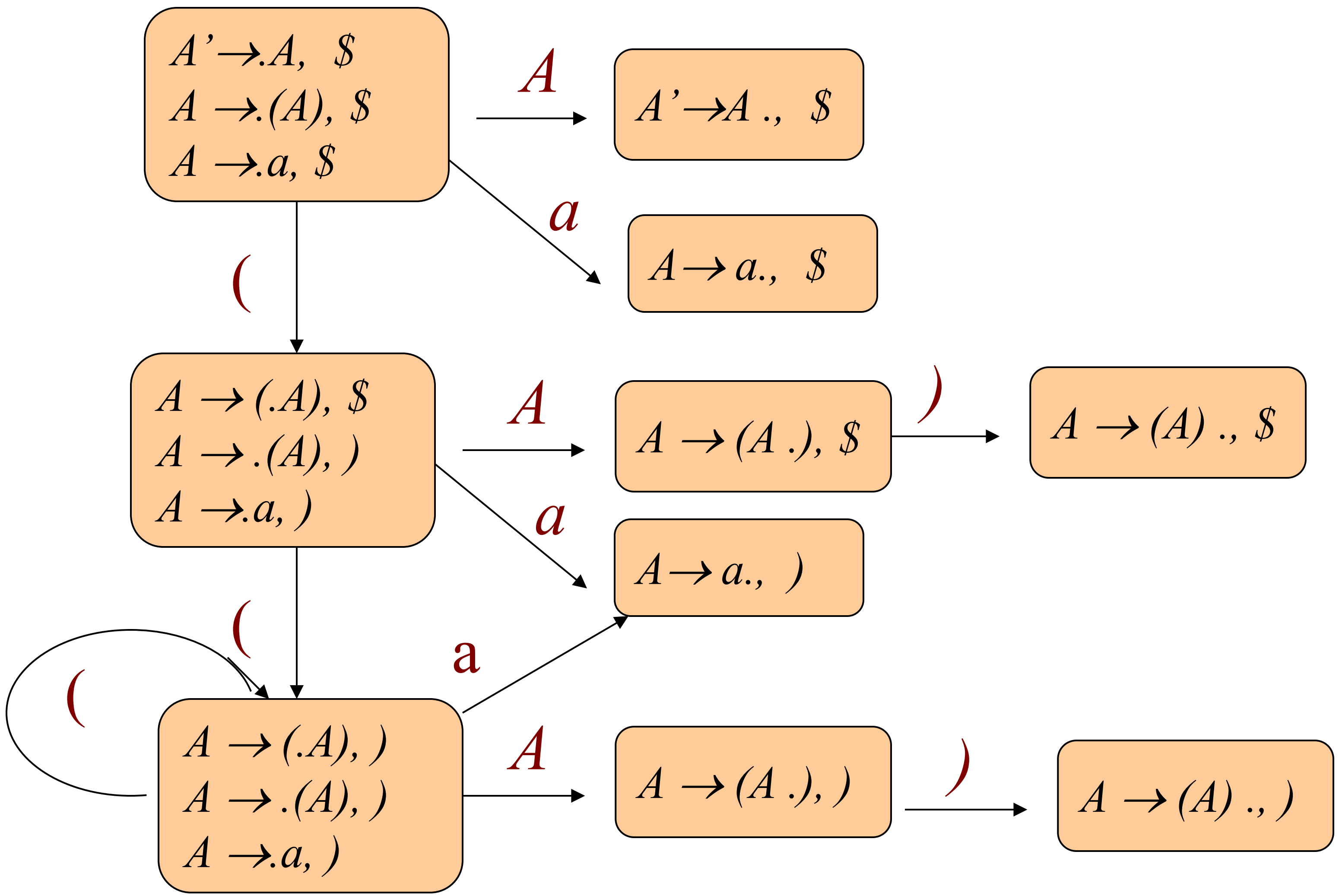
- The *start* state :

  Augmenting the grammar with a new start symbol *S'*

  a new production *S'*—> *S*
- The start symbol of the NFA of *LR(1)* items becomes the item [*S'*—>.*S*, *$*].

5.4.1 Finite Automata of LR(1) Items

- Example 5.14 : Consider the grammar : $A \to (A) \mid a$



5.4.2 The LR(1) Parsing Algorithm

- The General LR(1) parsing algorithm :

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

  1. If state s: any LR(l) item of the form [A→α·Xβ,a], X is a terminal, and X is the next token in the input string

  2. If state s : the complete LR(1) item [A→α·,a] , the next token: in the input string is a

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

5.4.2 The LR(1) Parsing Algorithm

- A grammar is an LR(1) grammar :
  If the application of the above general LR(1) parsing rules results in no ambiguity.
- A grammar is LR(1) *if and only if*, for any state s. the following two conditions are satisfied.
  1. For any item $[A\rightarrow\alpha\cdot X\beta,a]$ in s with X a terminal, there is no item in s of the form $[B\rightarrow\gamma\cdot,X]$ (otherwise there is a *shift-reduce* conflict).
  2. There are no two items in s of the form $[A\rightarrow\alpha\cdot, a]$ and $[B\rightarrow\beta\cdot,a]$ (otherwise, there is a *reduce-reduce* conflict).

5.4.2 The LR(1) Parsing Algorithm

- Example 5.15 (1) $A \rightarrow (A)$ (2) $A \rightarrow a$

State	Input				Goto
	(	A	)	$	S
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		

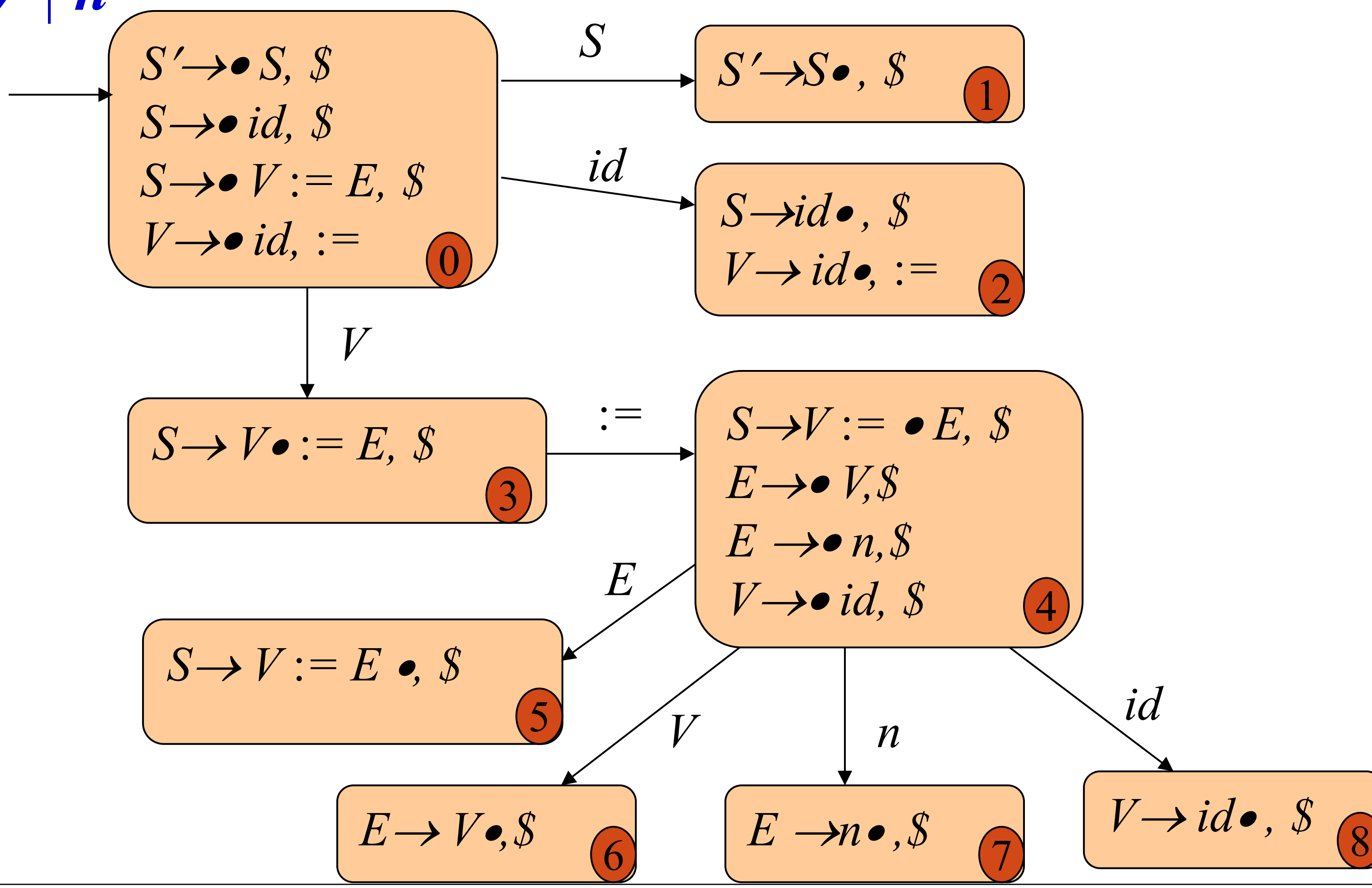
5.4.2 The LR(1) Parsing Algorithm

- Example 5.16

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

  $V \rightarrow \boldsymbol{id}$

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



5.4.3 LALR(1) Parsing

- The size of the DFA of sets of LR(1) items is too large
- The same set of first components
- Differing only in their second components (the lookahead symbols).

5.4.3 LALR(1) Parsing

- The LALR(1) parsing algorithm : it makes sense to identify all such states and combine their lookaheads.
- LALR(l) parsing retains some of the benefit of LR(l) parsing over SLR(l) parsing,
- Preserving the smaller size of the DFA of LR(0) items.

5.4.3 LALR(1) Parsing

- First principle of LALR(1) parsing

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

- Second principle of LALR(1) parsing

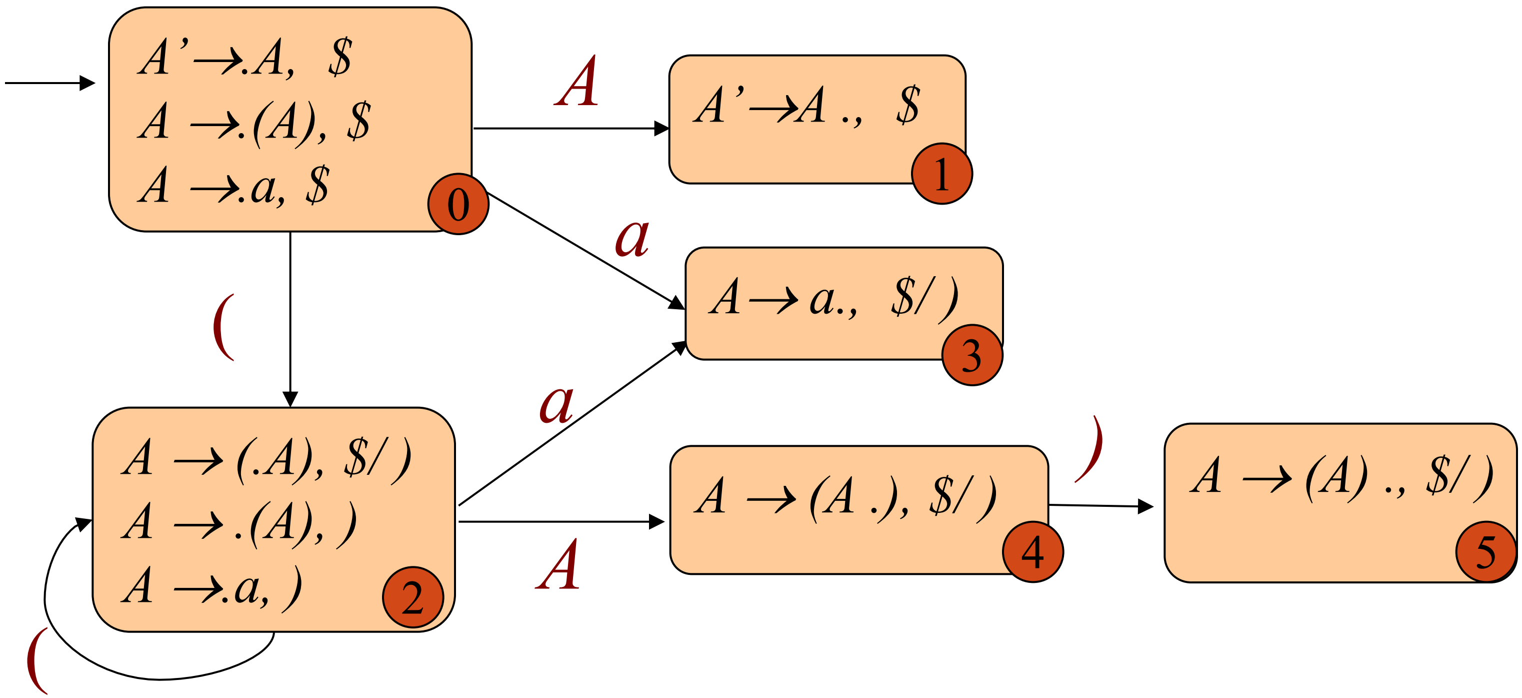
  Given two states s_1 and s_2 of the DFA of LR(l) items that have the same core, suppose there is a transition on the symbol X from s_1 to a state t_1. Then there is also a transition on X from state s_2 to a state t_2, and the states t_1 and t_2 have the same core.

5.4.3 LALR(1) Parsing

- These two principles allow us to construct the DFA of LALR(l) items
  - Identifying all states that have the same core
  - Forming the union of the lookahead symbols for each LR(0) item.

5.4.3 LALR(1) Parsing

- Example 5.1 7 Consider the grammar of Example 5.14. (l) $A\rightarrow(A)$ (2) $A\rightarrow a$



5.4.3 LALR(1) Parsing

- A grammar is an LALR(l) grammar if no parsing conflicts arise in the LALR(l) parsing algorithm.
- If a grammar is LR(l), then the LALR(l) parsing table cannot have any *shift-reduce* conflicts, there may be *reduce-reduce* conflicts.

$$\begin{array}{l}[A \to \alpha.\ \ a] \\ [B \to \alpha.\ \ b]\end{array} \qquad \begin{array}{l}[A \to \alpha.\ \ b] \\ [B \to \alpha.\ \ a]\end{array} \quad \Longrightarrow \quad \begin{array}{l}[A \to \alpha.\ \ a/b] \\ [B \to \alpha.\ \ a/b]\end{array}$$

5.4.3 LALR(1) Parsing

- If a grammar is SLR(l), then it certainly is LALR(l)
- LALR(1) parsers often do as well as general LR(1) parsers in removing typical conflicts that occur in SLR(l) parsing.
- If the grammar is already LALR(1), the only consequence of using LALR(1) parsing over general LR parsing as following. in the presence of errors, Some spurious reductions may be made before error is declared.
- Compute the DFA of LALR(l) items directly from the DFA of LR(0) items through a process of *propagating lookaheads*.

5.7 Error Recovery in Bottom-up Parsers

- A bottom-up parser will *detect an error* when a blank (or error) entry is detected in the parsing table.
- Errors should be detected as soon as possible.
- This goal conflicts with an equally important one: reducing the size of the parsing table.
- An LR(l) parser can, for example, detect errors earlier than an LALR(l) or SLR(1) parser, and these latter can detect errors earlier than an LR(0) parser.

5.7 Error Recovery in Bottom-up Parsers

- A good error recovery in bottom-up parsers: removing symbol from either the parsing stack or the input or both.
- There are three possible alternative actions:

    1. *Pop* a state from the stack.

    2. Successively *pop* tokens from the input until a token is seen for which we can restart the parse.

    3. *Push* a new state onto the stack.

5.7 Error Recovery in Bottom-up Parsers

When an error occurs is as follows:

1. Pop states from the parsing stack until a state is found with nonempty *Goto* entries.
2. If there is a legal action on the current input token from one of the *Goto* states, push that state onto the stack and restart the parse.
3. If there is no legal action on the current input token from one of the *Goto* states, advance the input.

5.7 Error Recovery in Bottom-up Parsers

There are several possible solutions (infinite loop)

1、Insist on a *shift* action from a *Goto* state in step 2. （too restrictive ）

2、If the next legal move is a reduction, to set a flag that causes the parser to keep track of the sequence of states during the following reductions

3、If the same state recurs, to pop stack states until the original state is removed.

5.7 Error Recovery in Bottom-up Parsers

Example 5.19 Input: *(2+*)*

分 析 栈	输 入	动 作
...	...	...
$0 (6 E 10+7	*) $	错误:
		压入T, goto 11
$0 (6 E 10+7 T 11	*) $	移进9
$0 (6 E 10+7 T 11 * 9	) $	错误:
		压入F, goto 13
$0 (6 E 10+7 T 11 * 9 F 13	) $	用T→T * F 归约
...	...	...

5.7 Error Recovery in Bottom-up Parsers

状　态	输　入							Goto			
	NUMBER	(	+	−	*	)	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		S14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

Homework of Chapter 5

5.8 Consider the following grammar

declaration → type var-list

type → int | float

var-list → identifier,var-list | identifier

- a. Rewrite it in a form more suitable for bottom-up parsing.
- b. Construct the DFA of LR(0) items for the rewritten grammar.
- c. Construct the SLR(1) parsing table for the rewritten grammar.

Homework of Chapter 5

5.12 Show that the following grammar is LR(1) but not LALR(1):

$$S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$$

$$A \rightarrow c$$

$$B \rightarrow c$$