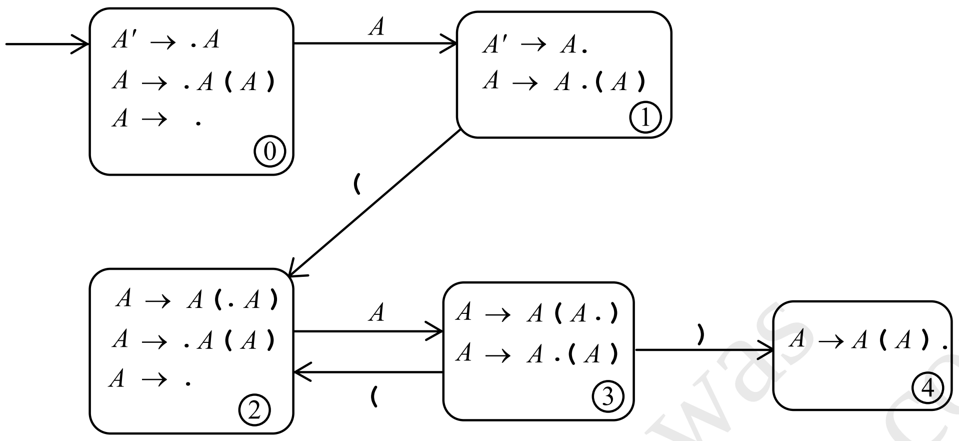
Compiler Construction: Principles and Practice
by Kenneth C. Louden

Chapter 5 Exercise Answers

Exercise 5.3

(a)



(b) The following table uses the grammar rule numbering (1) $A \to A$ **(** A **)** and (2) $A \to \varepsilon$, and is based on the fact that Follow(A) = { **(**, **)**, \$ }:

State	Input			Goto
	(	)	\$	*A*
0	r2	r2	r2	1
1	s2		accept	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

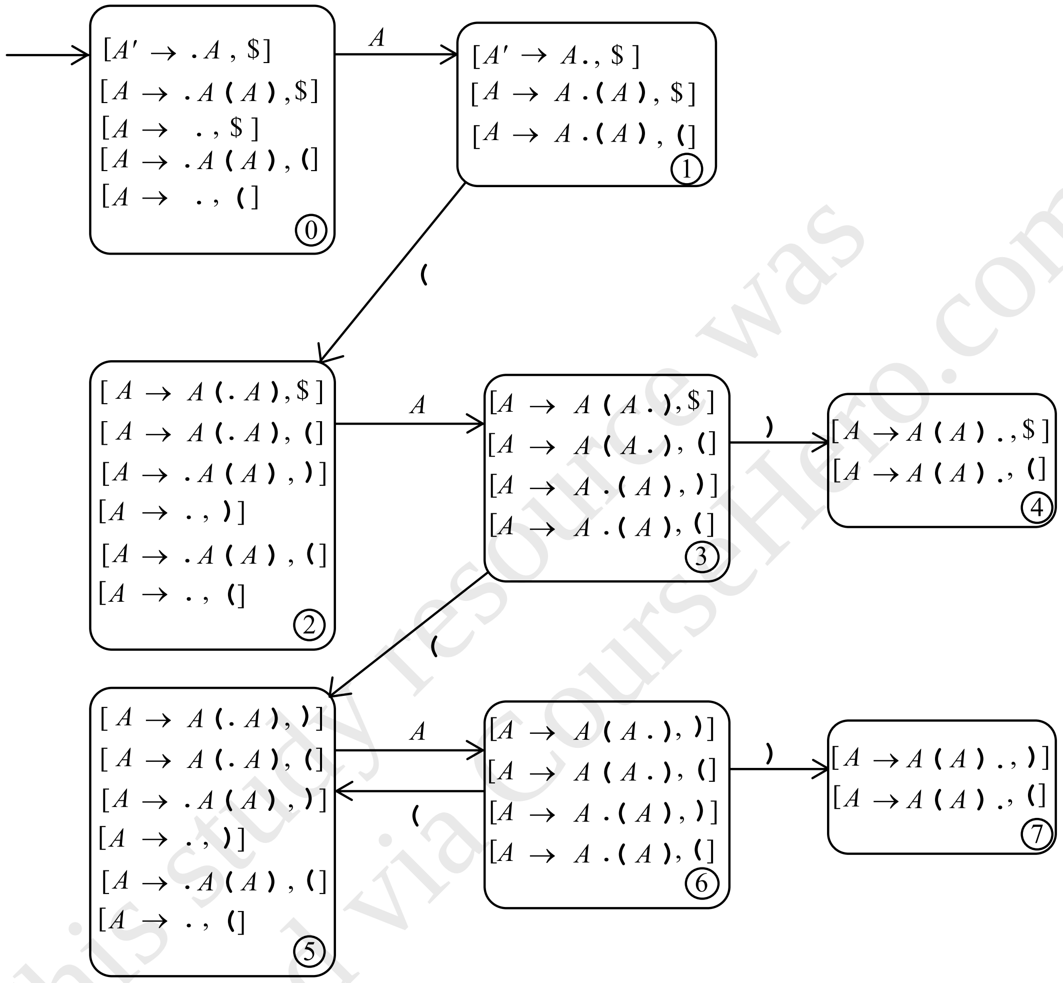
(c)

	Parsing stack	Input	Action
1	\$ 0	(()())\$	reduce 2
2	\$ 0 *A* 1	(()())\$	shift 2
3	\$ 0 *A* 1 (2	()())\$	reduce 2
4	\$ 0 *A* 1 (2 *A* 3	()())\$	shift 2
5	\$ 0 *A* 1 (2 *A* 3 (2	)())\$	reduce 2
6	\$ 0 *A* 1 (2 *A* 3 (2 *A* 3	)())\$	shift 4
7	\$ 0 *A* 1 (2 *A* 3 (2 *A* 3) 4	())\$	reduce 1
8	\$ 0 *A* 1 (2 *A* 3	())\$	shift 2
9	\$ 0 *A* 1 (2 *A* 3 (2	))\$	reduce 2
10	\$ 0 *A* 1 (2 *A* 3 (2 *A* 3	))\$	shift 4
11	\$ 0 *A* 1 (2 *A* 3 (2 *A* 3) 4	)\$	reduce 1
12	\$ 0 *A* 1 (2 *A* 3	)\$	shift 4
13	\$ 0 *A* 1 (2 *A* 3) 4	\$	reduce 1
14	\$ 0 *A* 1	\$	accept

(d) The grammar is *not* an LR(0) grammar, since state 1 has both a shift and a reduce (accept) action.

Exercise 5.4

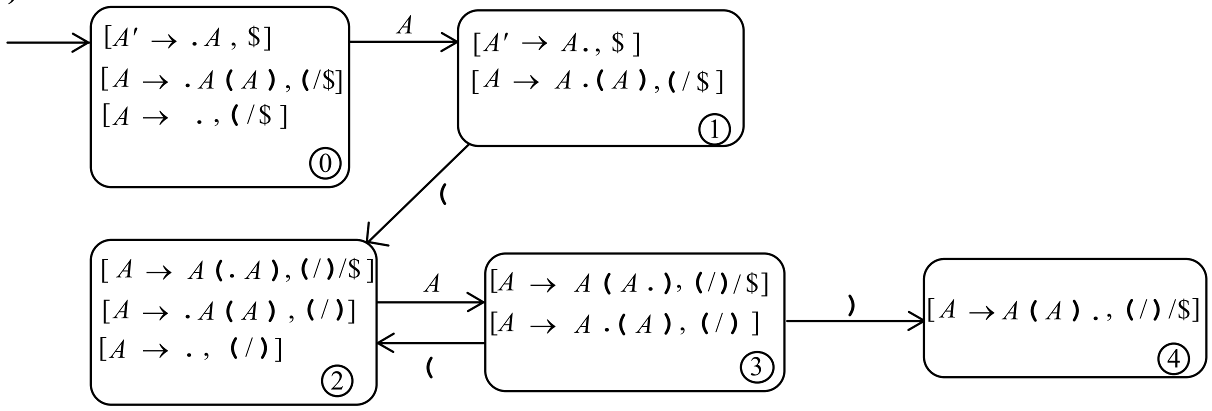
(a)



(b)

State	Input			Goto
	(	)	$	*A*
0	r2		r2	1
1	s2		accept	
2	r2	r2		3
3	s5	s4		
4	r1		r1	
5	r2	r2		6
6	s5	s7		
7	r1	r1		

(c)



(d)

State	Input			Goto
	(	)	$	A
0	r2		r2	1
1	s2		accept	
2	r2	r2		3
3	s2	s4		
4	r1	r1	r1	

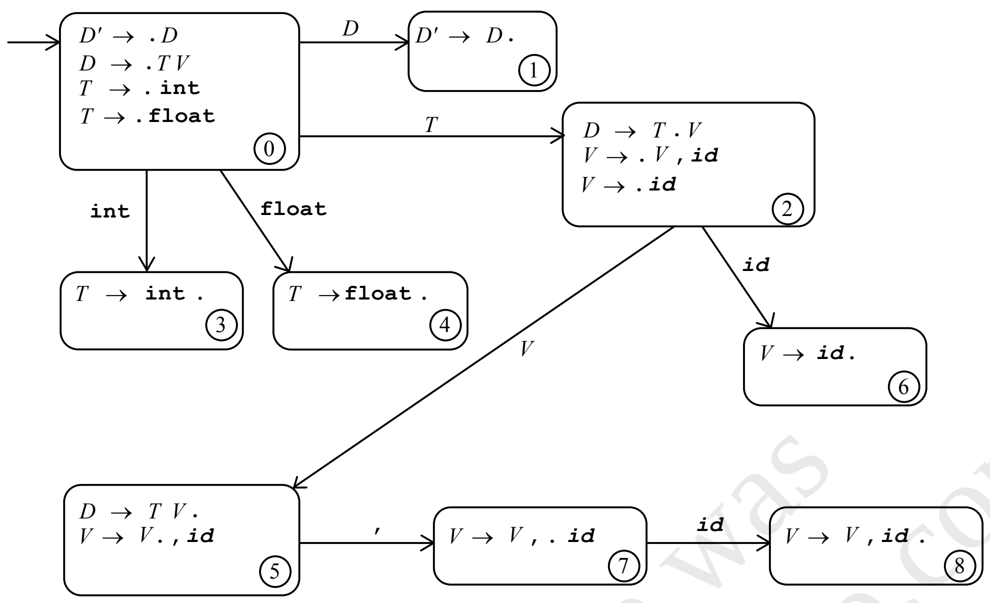
(e) As stated in the text, the only possible difference between the actions of an LR(1) parser and an LALR(1) parser is that, in the presence of errors, the LALR(1) parser may make some extra reductions before declaring error. Comparison of the tables of (b) and (d) shows that there are two situations in which the LALR(1) parser will make a reduction, while the LR(1) parser will declare error. First, when the LALR(1) and LR(1) parsers are both in state 4 and the next input is a right parenthesis, the LR(1) parser will declare error, while the LALR(1) parser will reduce by the rule $A \rightarrow A$ **(** A **)** first, and then will declare error. This will happen, for example, on the input **())**. The second case occurs when the LR(1) parser is in state 7 and the LALR(1) parser is in state 4, and the next input is $, in which case the LALR(1) again reduces by rule 1 before declaring error. This will happen, for example, on the input **(()**.

Exercise 5.8

(a) Right recursion causes the parsing stack to grow unnecessarily, so we rewrite the grammar as follows (condensing some of the names to save space):

$D \rightarrow T\ V$
$T \rightarrow$ **int** | **float**
$V \rightarrow\ V$ **,** ***id*** | ***id***

(b)



(c) In the following table we use the numbering

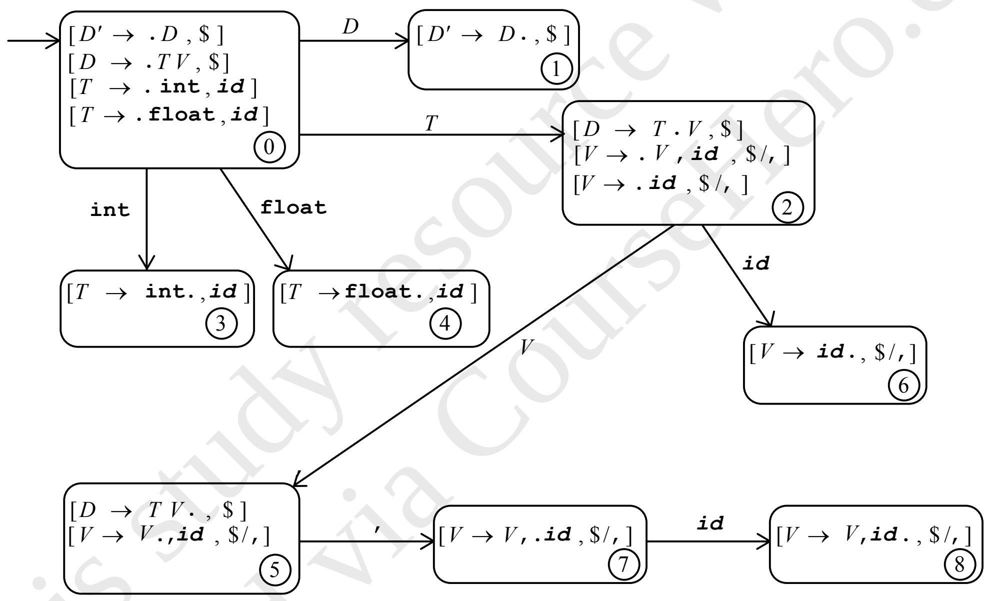
(1)	$D \to T\ V$
(2)	$T \to$ **int**
(3)	$T \to$ **float**
(4)	$V \to V$ **, *id***
(5)	$V \to$ ***id***

State	Input					Goto		
	int	**float**	***id***	**,**	$	*D*	*T*	*V*
0	s3	s4		s3		1	2	
1					accept			
2			s6					5
3			r2					
4			r3					
5				s7	r1			
6				r5	r5			
7			s8		r4			
8				r4	r4			

(d)

	Parsing stack	Input	Action
1	$ 0	**int x,y,z**$	shift 3
2	$ 0 **int** 3	**x,y,z**$	reduce 2
3	$ 0 *T* 2	**x,y,z**$	shift 6
4	$ 0 *T* 2 **id** 6	**,y,z**$	reduce 5
5	$ 0 *T* 2 *V* 5	**,y,z**$	shift 7
6	$ 0 *T* 2 *V* 5 **,** 7	**y,z**$	shift 8
7	$ 0 *T* 2 *V* 5 **,** 7 **id** 8	**,z**$	reduce 4
8	$ 0 *T* 2 *V* 5	**,z**$	shift 7
9	$ 0 *T* 2 *V* 5 **,** 7	**z**$	shift 8
10	$ 0 *T* 2 *V* 5 **,** 7 **id** 8	$	reduce 4
11	$ 0 *T* 2 *V* 5	$	reduce 1
12	$ 0 *D* 1	$	accept

(e)



(f) The LALR(1) parsing table is the same as the SLR(1) parsing table shown in (c).

__**Exercise 5.10**__

We use similar language to that on page 210, with appropriate modifications:

The SLR(1) parsing algorithm. Let *s* be the current state whose number is at the top of the parsing stack. Then actions are defined as follows:

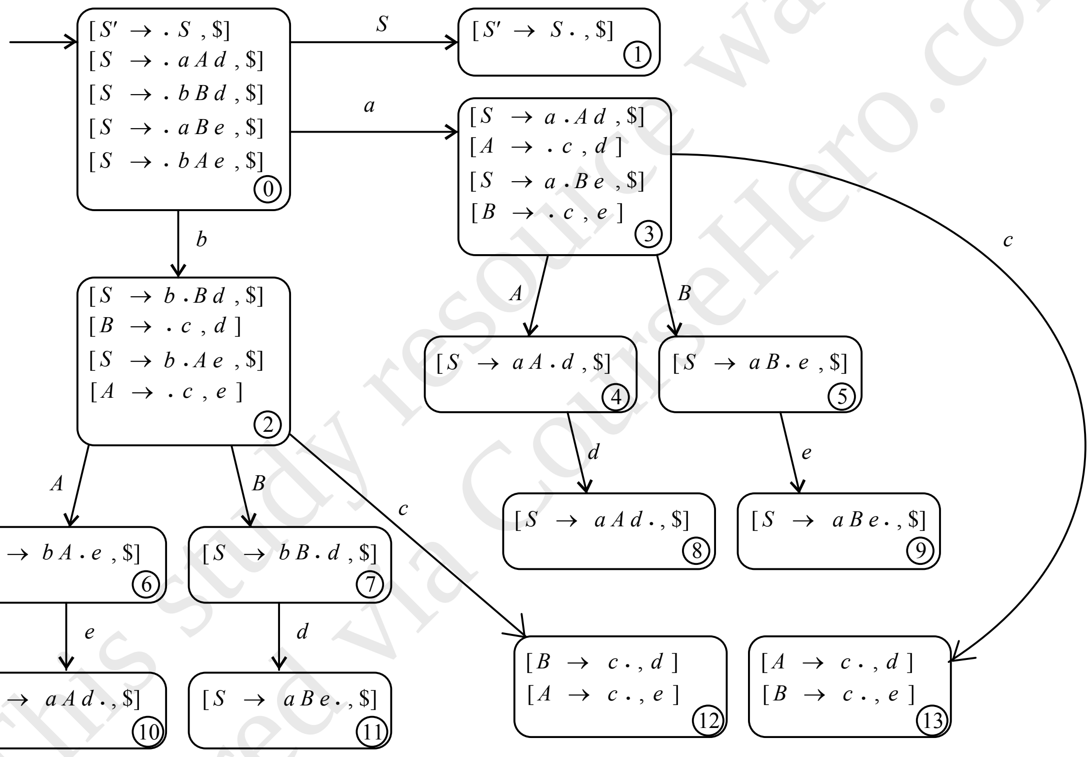
(1) If state *s* contains any item of the form $A \to \alpha . X \beta$, where *X* is a terminal, and *X* is the next token in the input string, then the action is to remove the current input token and push onto the stack the number of the state containing the item $A \to \alpha X . \beta$.

(2) If state *s* contains the complete item $A \to \alpha\,.$, and the next token in the input is in Follow(A), then the action is to reduce by the rule $A \to \alpha$. A reduction by the rule $S' \to S$, where *S* is the start state, is equivalent to acceptance. In all other cases, the new state is computed as follows. Remove *n* state numbers from the parsing stack, where *n* is the number of symbols in α. Correspondingly, back up in the DFA to the state whose number is now at the top of the stack. By construction, this state must contain an item of the form $B \to \alpha\,.\,A\,\beta$. Push the number of the state containing the item $B \to \alpha\,A\,.\,\beta$ onto the stack.

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

Exercise 5.12

The DFA of sets of LR(1) items for this grammar is:



From this DFA it can be seen that there are no parsing conflicts, so this grammar is LR(1). In particular, the reductions by $A \to c$ and $B \to c$ in states 12 and 13 occur for different lookaheads. However, these two states would be combined in the DFA of LALR(1) items (since they have the same set of LR(0) items), and then a reduce-reduce parsing conflict results, since reductions by both $A \to c$ and $B \to c$ would be indicated for lookaheads *d* and *e*. Thus, the grammar is not LALR(1).

Exercise 5.14

Recall that a prefix of a right sentential form is a viable prefix if it occurs on the parsing stack during a bottom-up parse. Thus, let $S \Rightarrow^* \alpha \Rightarrow^* w$ be a right-most derivation with right sentential form α, and suppose that β is a prefix of α, so that α = βγ. Now assume that β does not extend beyond the handle of α (we assume an unambiguous grammar, so handles are unique). Then the position of the handle is either at the end of β, in which case β is on the stack and is a viable prefix, or the position of the handle is within γ, in which case β still occurs on the stack at some point prior to the reduction by the handle of α, followed by a series of shifts and reduces that place the handle on the top of the stack. Thus, if β does not extend beyond the handle, it must be a viable prefix.

Conversely, suppose that β is a viable prefix, and so occurs on the stack during the derivation that leads from *S* to α. Then a (possibly empty) series of shifts and reductions will occur after β appears on the stack, resulting in the appearance of the handle on the top of the stack. Thus, β cannot extend beyond the handle.

Exercise 5.18

Numbering the rules in the order written

(1) $A \rightarrow AA$
(2) $A \rightarrow$ **(** A **)**
(3) $A \rightarrow \varepsilon$

and running Yacc with the -verbose option gives the following parsing table:

State	Input			Goto
	(	)	$	*A*
0	s1	r3	r3	2
1	s1	r3	r3	3
2	s1	r3	accept	4
3	s1	s5	r3	4
4	s1	r1	r1	4
5	r2	r2	r2	

We note in this table that all tokens are shifted at some point, and all rules are used in a reduction. Thus, there is no obvious error in this table. Indeed, comparing this table to the table of Exercise 5.3 (above) shows that it parses all nested parenetheses correctly, despite the ambiguity. The main reason Yacc succeeds on this grammar is that, in the face of a reduce-reduce conflict in state 4 between rules 1 and 3, it prefers a reduction by rule 1. If it had reduced by rule 3 in state 4 on any input, then an infinite loop results, since the goto is to state 4 itself.

Exercise 5.20

All operations receive the same precedence, and all operations become right associative (since shifts are always favored over reductions).

Exercise 5.22

(a) Using the heuristics listed on page 246, the actions of the parser are as follows, resulting in an infinite loop (*E* = *exp*, *T* = *term*, *F* = *factor*, **N** = **NUMBER**):

	Parsing stack	Input	Action
1	$ 0	**(*2** $	shift 6
2	$ 0 **(** 6	***2** $	error (goto 3)
3	$ 0 **(** 6 *T* 3	***2** $	shift 9
4	$ 0 **(** 6 *T* 3 ***** 9	**2** $	shift 5
5	$ 0 **(** 6 *T* 3 ***** 9 **N** 5	$	reduce 7
6	$ 0 **(** 6 *T* 3 ***** 9 *F* 13	$	reduce 5
7	$ 0 **(** 6 *T* 3	$	reduce 4
8	$ 0 **(** 6 *E* 10	$	error (pop to 6)
9	$ 0 **(** 6	$	error (goto 4)
10	$ 0 **(** 6 *F* 4	$	reduce 6
11	$ 0 **(** 6 *T* 3	$	reduce 4
12	...	$	...

(b) Obviously we would like to suggest a way to avoid the infinite loop. One easy way would be to simply count reductions since the last shift and stop after some arbitrary number. Another way would be to mark the stack at the first error, and to stop if the marker is on the top of the stack and the same action as before occurs. More difficult is to suggest a way to continue the error recovery after a loop is detected. One way would be to pop the state from the stack at which the first error was seen, and begin again to apply the rules on page 246. In the example of part (a) this would simply result in the stack being emptied. Another possibility might be to look for a shift from the state where the loop began that immediately allows a further reduction. In the example, from state 10 a shift to state 14 on right parenthesis would in fact be exactly right. However, this is not always guaranteed to work either.

Exercise 5.24

(a) The replacement rule in this exercise should actually be

```
command :  exp       {printf("%d\n",$1);}
        |  exp error {printf("%d\n",$1);}
        ;
```

since we do not want to always insist that there be an error. With this rule, the behavior of a Yacc-generated parser on the input `2 3` is as follows. First, the 2 is parsed normally until the nonterminal `exp` is on the stack. Now the normal behavior would be to reduce `exp` to `command` and print the value 2. However, the lookahead is now a number, and there are two possible lookaheads, $ and `error`, with a reduction to `command` on $ and a shift of `error`. Since the lookahead is not $, an error is declared, an error message is printed, and the `error` token is shifted onto the stack. Now `exp error` is on the stack, and the parser uses the second rule above to reduce them to `command`, printing the value 2. Of course, the number 3 is still in the input, and only $ can follow a command, so the parser registers another error (without, however, printing a message, since the parser is still in error mode). However, the parsing stack is empty, so it cannot be popped. Thus, the input is popped, revealing the $. Now the parser cannot take any more actions, and so exits in error mode.

(b) The replacement rule in this exercise should actually be

```
command :  exp       {printf("%d\n",$1);}
        |  error exp {printf("%d\n",$2);}
        ;
```

since, as in (a), we do not want to always insist that there be an error. With this rule, the behavior of a Yacc-generated parser on the input **`2 3`** is as follows. As before, the 2 is shifted, and reductions occur until **`exp`** is on the stack. Now the parser makes a further reduction from **`exp`** to **`command`** (and prints 2), since the lookahead is a number (which is an error), but Yacc has extended the reduction as the default action to all non-shift cases. Note that **`error`** cannot be shifted at this point. Now **`command`** is at the top of the stack and a number is the next input token. At this point Yacc declares error and prints an error message. Since **`error`** cannot be correctly shifted in this state, Yacc pops the parsing stack, revealing the start state. From that state it is possible to shift **`error`** (by the error production), and so Yacc does so. Now the number 3 can be correctly shifted and reduced to **`exp`**, and then with **`error exp`** on the stack, Yacc reduces these to **`command`**, print the value 3 and ending the parse successfully.