Chapter 5
Bottom-Up Parsing

한양대학교 컴퓨터공학부
임을규
2014년 2학기

Bottom-up parsing

- **Bottom-up parsing**
  - **Definition**
    - Parsing an input string of tokens by tracing out the steps in a rightmost derivation.
  - **Categories**
    - LR(0) parsing
    - LR(1) parsing : powerful but complex
    - SLR(1) parsing : Simple LR(1)
    - LALR(1) parsing : Lookahead LR(1)
      - More powerful than SLR(1), and less complex than LR(1)

L : input is processed from left to right
R : rightmost derivation
1 : one symbol of lookahead

Bottom-up parsing

- Outline
  - Initialization
    - The stack is empty.
  - Iteration of the followings until the stack is empty.
    - 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 → α.
  - If the stack contains the start symbol and the input is empty, accept.
  - Shift-reduce parser

Example: table 5.1

new symbol add $S' \to S$

$S \to (S) S \mid \varepsilon$

S` -> S -> (S)S -> (S) -> ()
rightmost derivation, reverse order

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

Example: table 5.2

$E' \rightarrow E$
$E \rightarrow E + \mathrm{n} \mid \mathrm{n}$

	Parsing stack	Input	Action
1	$	**n** + **n** $	shift
2	$ **n**	+ **n** $	reduce $E \rightarrow \mathbf{n}$
3	$ *E*	+ **n** $	shift
4	$ *E* +	**n** $	shift
5	$ *E* + **n**	$	reduce $E \rightarrow E + \mathbf{n}$
6	$ *E*	$	reduce $E' \rightarrow E$
7	$ *E*`	$	accept

Bottom-up Parsing

- Bottom-up parsers have less difficulty than top-down parsers with lookahead.
- However, a bottom-up parser may need to look deeper into the stack.

  this 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

- Sometimes, the next token in the input may also need to be consulted as a lookahead.

  because keeping track of the stack contents alone is not enough to be able to uniquely determine the next step in shift-reduce parse
  different shift-reduce parsing methods use the lookahead in different ways (results in parsers of varying power and complexity)

- A shift-reduce parser traces out a rightmost derivation of the input string, but the steps of the derivation occur in reverse order.

Bottom-up Parsing

- Right sentential form ex) S` -> S -> (S)S -> (S) -> ()
  - Each of the intermediate strings of terminals and nonterminals in the rightmost derivation. split between the parsing stack and the input during a shift-reduce parse
- Viable prefix
  - sequence of symbols on the parsing stack
- Handle
  - string in the right sentential form + production
- Determining the next handle in a parse is the main task of a shift-reduce parser.
  - The string at the top of the stack matches the right-hand side of a production.
  - Reduction occur when the resulting string is a right sentential form.
    - Step 3 of Table 5.1.

a shift-reduce parser will shift terminals from the input to the stack until it is possible to perform a reduction to obatin the next right sentential form

Finite automata of LR(0) items and LR(0) parsing

- LR(0) item (item for short) no explicit reference to lookahead
- Initial item
- Complete item
- augmented grammar
- Closure items
- Kernel items

P.202 example 5.3

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

. - mark symbol (현재까지 어디까지 string을 읽었는지 나타낸다.)

$$S' \to .S$$
$$S' \to S.$$
$$S \to .(S) S$$
$$S \to (.S) S$$
$$S \to (S.) S$$
$$S \to (S) .S$$
$$S \to (S) S.$$
$$S \to .$$

Finite automata of LR(0) items and LR(0) parsing

- LR(0) item (item for short)
- Initial item
- Complete item
- augmented grammar
- Closure items
- Kernel items

initial item
A -> .α - we may be about to recognize an A by using the grammar rule choice A ->α

complete item
A -> α. - α now resides on the top of the parsing stack and may be handle(reduce)

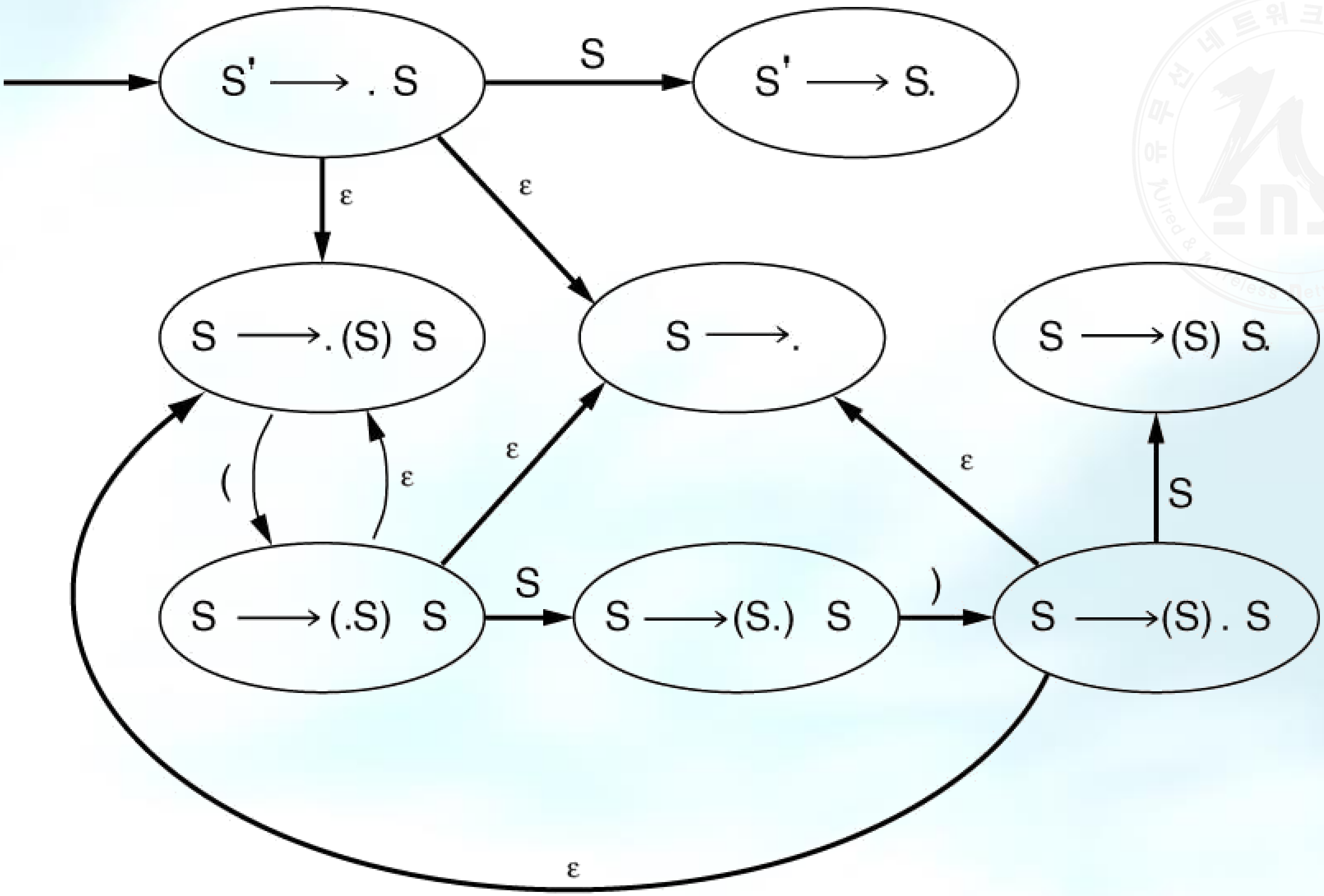
P.202 example 5.4

E’ → E
E → E + n | n

$E' \rightarrow .\, E$
$E' \rightarrow E\, .$
$E \rightarrow .\, E + \mathbf{n}$
$E \rightarrow E\, .\, + \mathbf{n}$
$E \rightarrow E + .\, \mathbf{n}$
$\mathrm{E} \rightarrow \mathrm{E} + \mathbf{n}\, .$
$E \rightarrow .\, \mathbf{n}$
$E \rightarrow \mathbf{n}\, .$

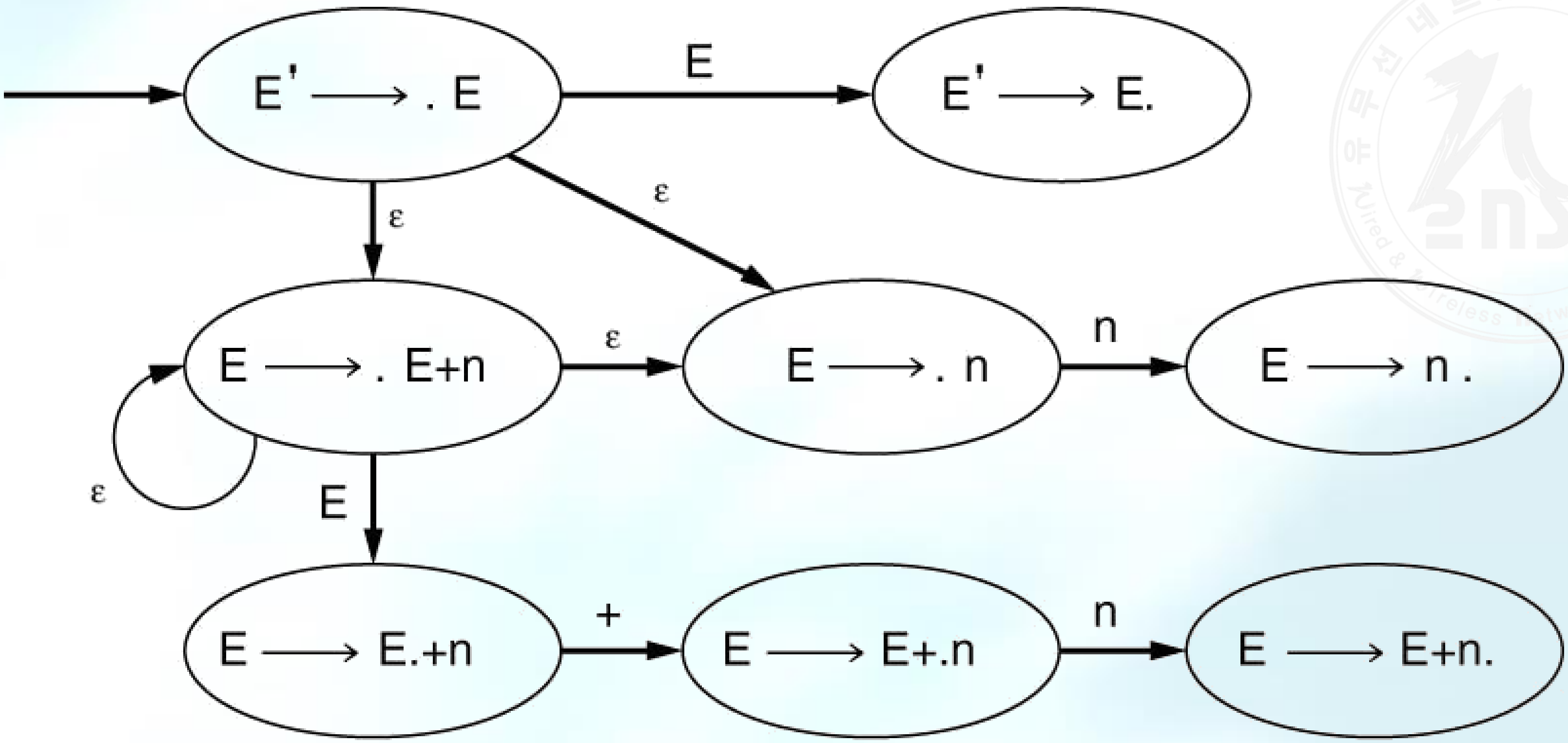
P.204 figure 5.1

start state을 유일하게 결정하기 위해 S` 도입 (augment grammar)

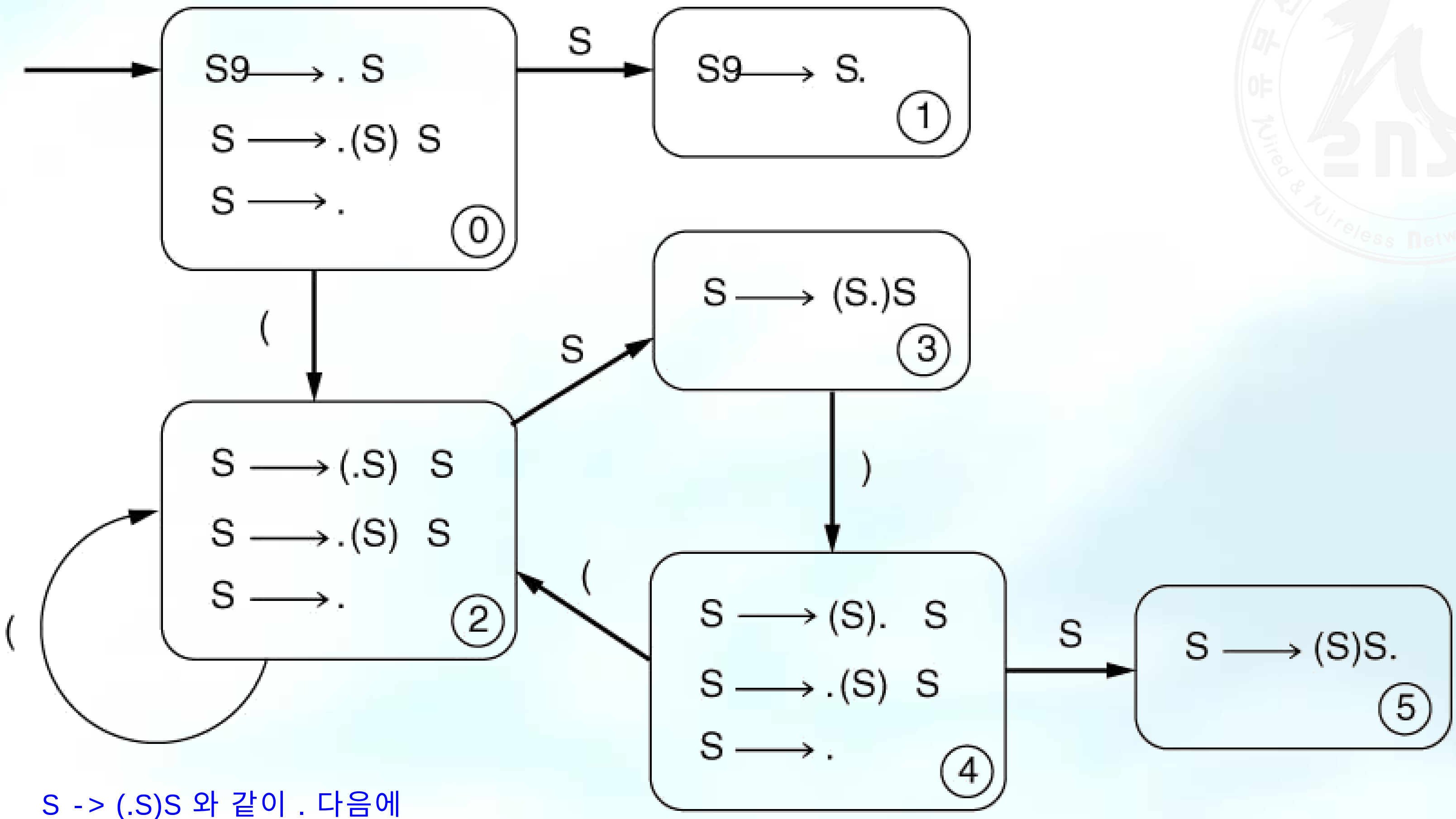


NFA의 accepting state 존재 X - parser가 accept인 지점을 결정해준다.

P.204 figure 5.2

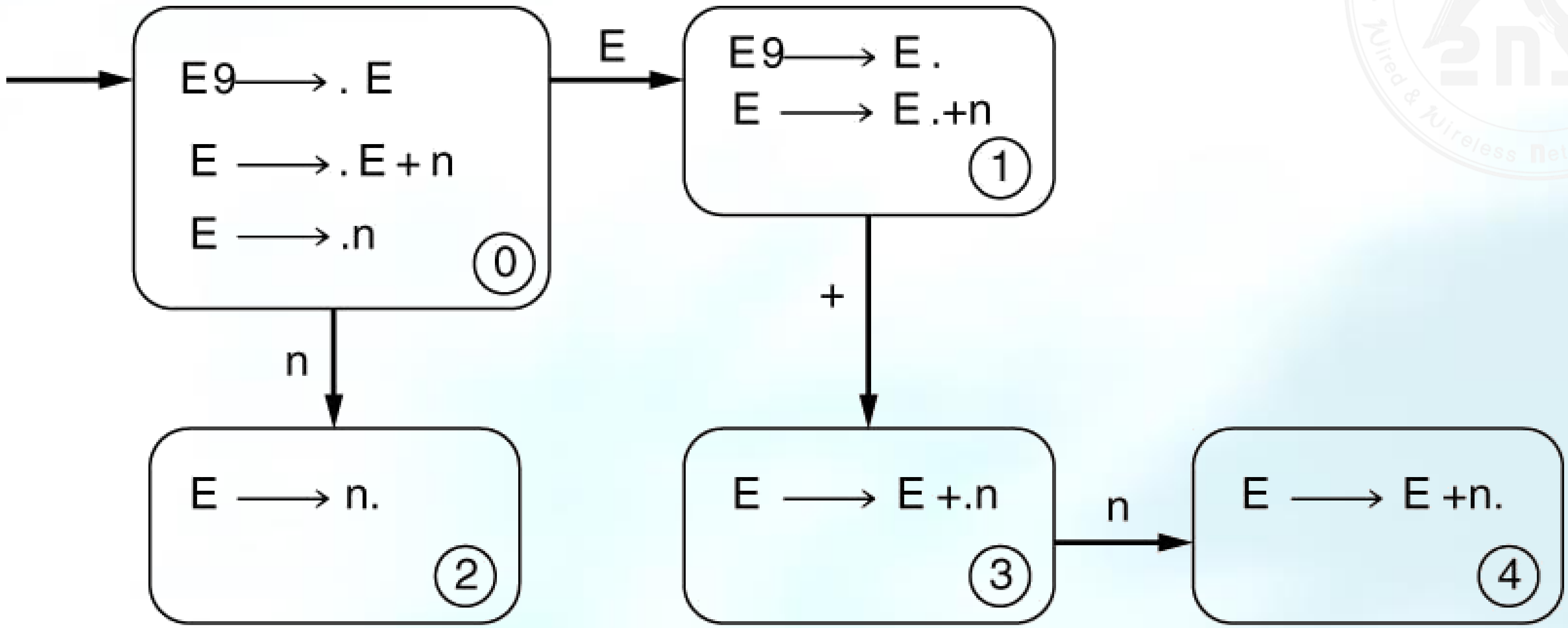


P.205 figure 5.3



S -> (.S)S 와 같이 . 다음에
nonterminal이 오면 derivation이 적용된다.
S -> . and S ->.(S)S 따라서 다음 두 grammar가
해당 state에 포함된다.

P.205 figure 5.4



Finite automata of LR(0) items and LR(0) parsing

- LR(0) item (item for short)
- Initial item
- Complete item
- augmented grammar
- Closure items
- Kernel items
  - Only kernel items need to be specified to completely characterize the DFA

DFA의 같은 state에 속한 item들의 관계
ex) E` -> .E - kernel items
E -> .E +n - closure items
E -> .n - closure items

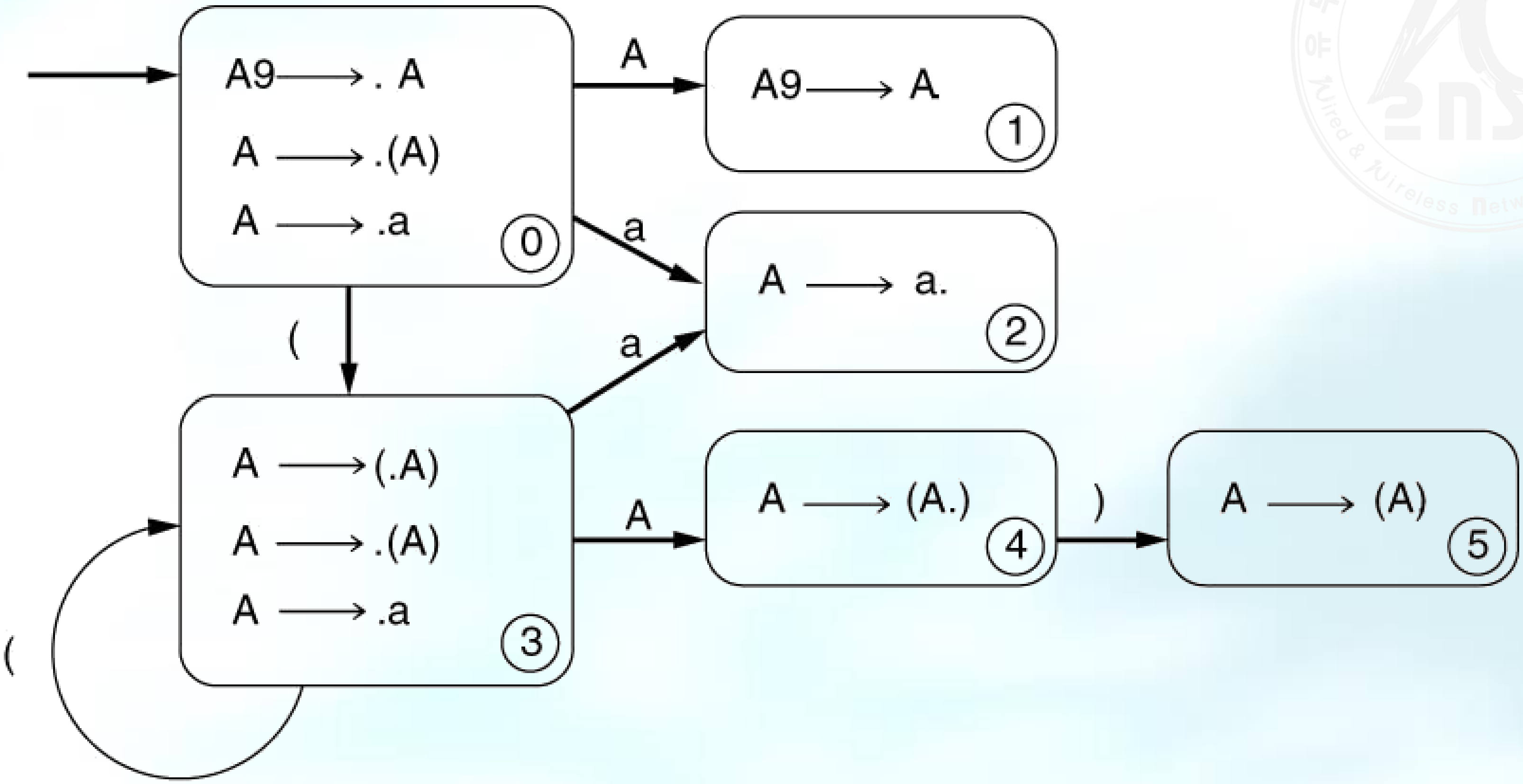
The LR(0) Parsing

Parsing stack	Input
$ 0 **n** 2	*Rest of InputString* $

The LR(0) Parsing

- A grammar is said to be an LR(0) grammar if a state has a complete item, then it can contain no other items. ambiguos가 없다.
- Definition
  - ...
  - If state s contains any complete item (an item of the form $A \rightarrow \gamma.$), then the action is to reduce by the rule $A \rightarrow \gamma$.
  - ...
- shift-reduce conflict 한 state에 A -> α. (reduce) A-> α.xβ(shift(x terminal)가 포함될 경우
- reduce-reduce conflict 한 state에 A-> α. (reduce) B->β. (reduce) 가 포함될 경우

P.208 figure 5.5



P.209 table 5.3

	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

Table-driven method!

P.209 table 5.4

reduce grammar rule

shift for nonterminal

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	$A \rightarrow (A)$				

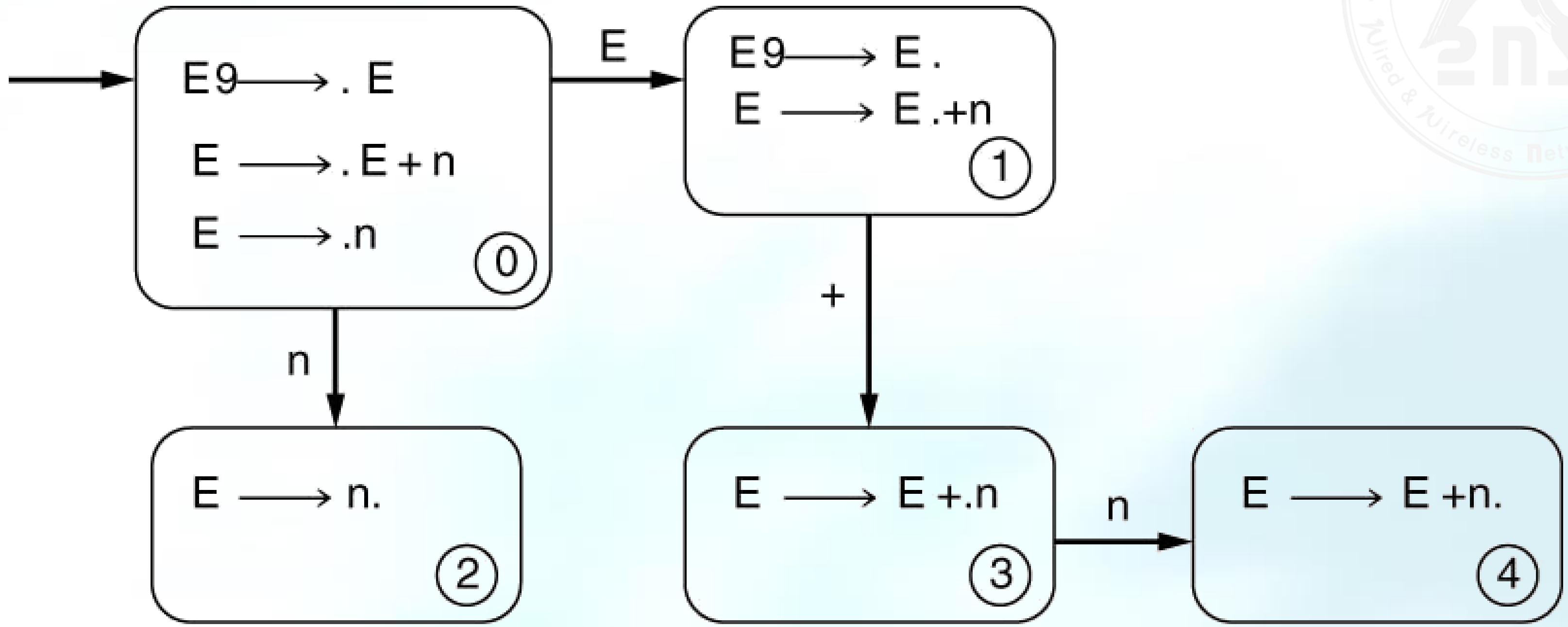
The SLR(1) Parsing

- Uses the DFA of sets of LR(0) items.
- shift-reduce and reduce-reduce conflicts can be minimized by consulting Follow sets.

- Definition
  - ...
  - 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 $A \rightarrow \gamma$.
  - …

shift-reduce conflict - in state $A \rightarrow \alpha.x\beta$ (x terminal) $B \rightarrow \gamma.$ 에서 $x \in$ Follow(B) 인 경우
input이 x인 경우 shift-reduce conflict 발생
reduce-reduce conflict - in state $A \rightarrow \alpha.$, $B \rightarrow \beta.$ 에서 Follow(A) 와 Follow(A) ∩ Follow(B) = not empty인 경우

P.205 figure 5.4



P.212 table 5.5 – SLR(1) parsing table

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 ($E \rightarrow E + n$)	

P.212 table 5.6

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

P.212 table 5.7 – SLR(1) parsing table

State	Input			Goto
	(	)	$	S
0	s2	r ($S \rightarrow \varepsilon$)	r ($S \rightarrow \varepsilon$)	1
1			accept	
2	s2	r ($S \rightarrow \varepsilon$)	r ($S \rightarrow \varepsilon$)	3
3		s4		
4	s2	r ($S \rightarrow \varepsilon$)	r ($S \rightarrow \varepsilon$)	5
5		r ($S \rightarrow (S) S$)	r ($S \rightarrow (S) S$)	

P.213 table 5.8

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

Disambiguating Rules for Parsing Conflicts

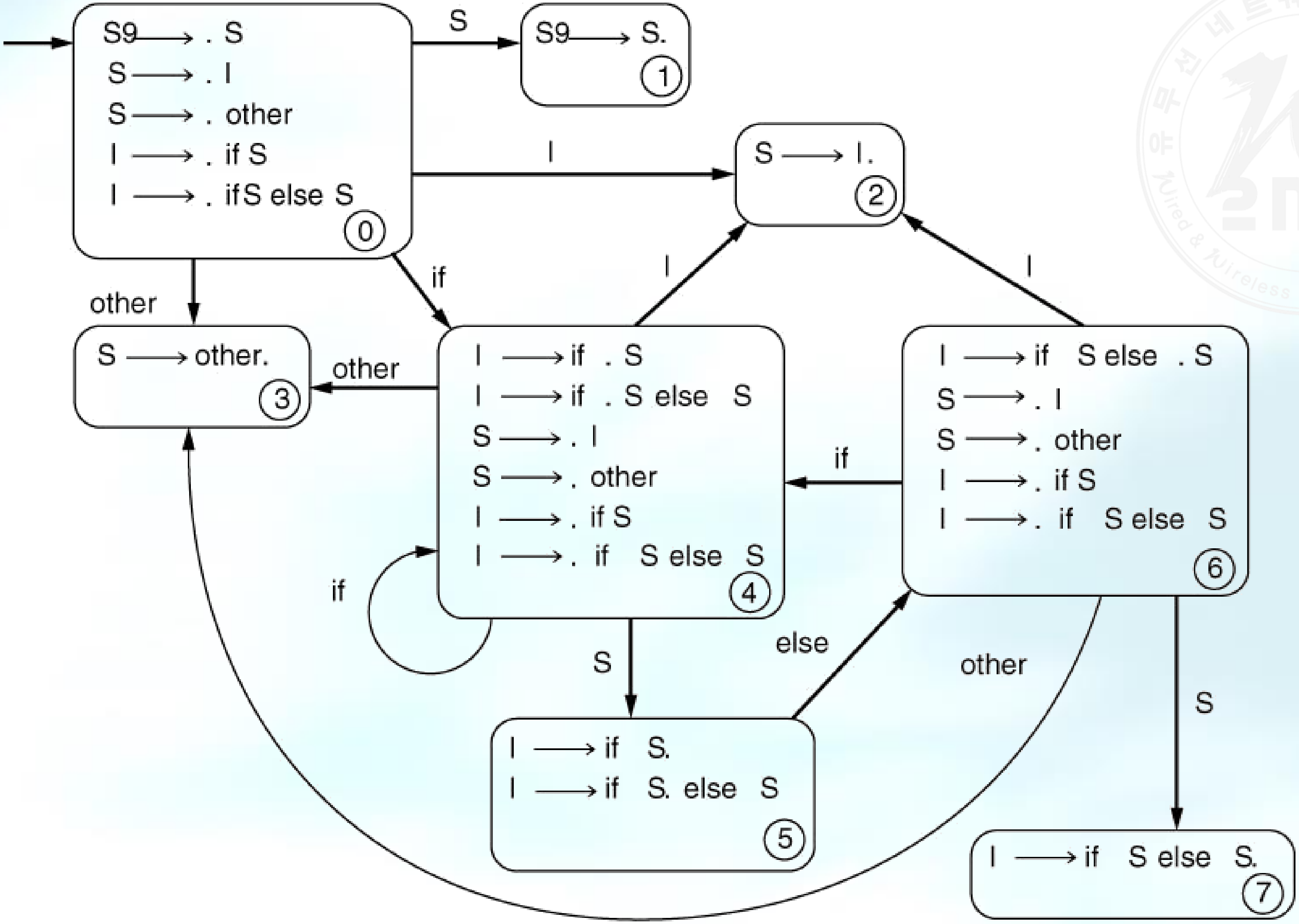
- P.213 example 5.12

statement → *if-stmt* | **other**

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

exp → 0 | 1

P.214 figure 5.6



P.215 table 5.9

- The shift is preferred over the reduction
  - What if the reduction is preferred?

State	Input				Goto	
	if	else	other	$	*S*	*I*
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		

P.215 example 5.13

- There are a few situations in which SLR(1) parsing is not quite powerful enough.

$$
\begin{aligned}
&\textit{stmt} \rightarrow \textit{call-stmt} \mid \textit{assign-stmt} \\
&\textit{call-stmt} \rightarrow \textit{identifier} \\
&\textit{assign-stmt} \rightarrow \textit{var} := \textit{exp} \\
&\textit{var} \rightarrow \textit{var}\ [\ \textit{exp}\ ] \mid \textit{identifier} \\
&\textit{exp} \rightarrow \textit{var} \mid \textit{number}
\end{aligned}
$$

Example 5.13

- Rules
  - $S \rightarrow id \mid V := E$
  - $V \rightarrow id$
  - $E \rightarrow V \mid n$
- A start state of the DFA
  - $S' \rightarrow .S$
  - $S \rightarrow .id$
  - $S \rightarrow .V := E$
  - $V \rightarrow .id$
- Conflict
  - $S \rightarrow id.$
  - $V \rightarrow id.$

Definition of LR(1) transitions

- Part 2
  - Given an LR(1) item [A → α.Bγ, 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).
- Examples
  - A → (A) | a
  - State 0: [A' → .A, $] First(ε$) = {$}

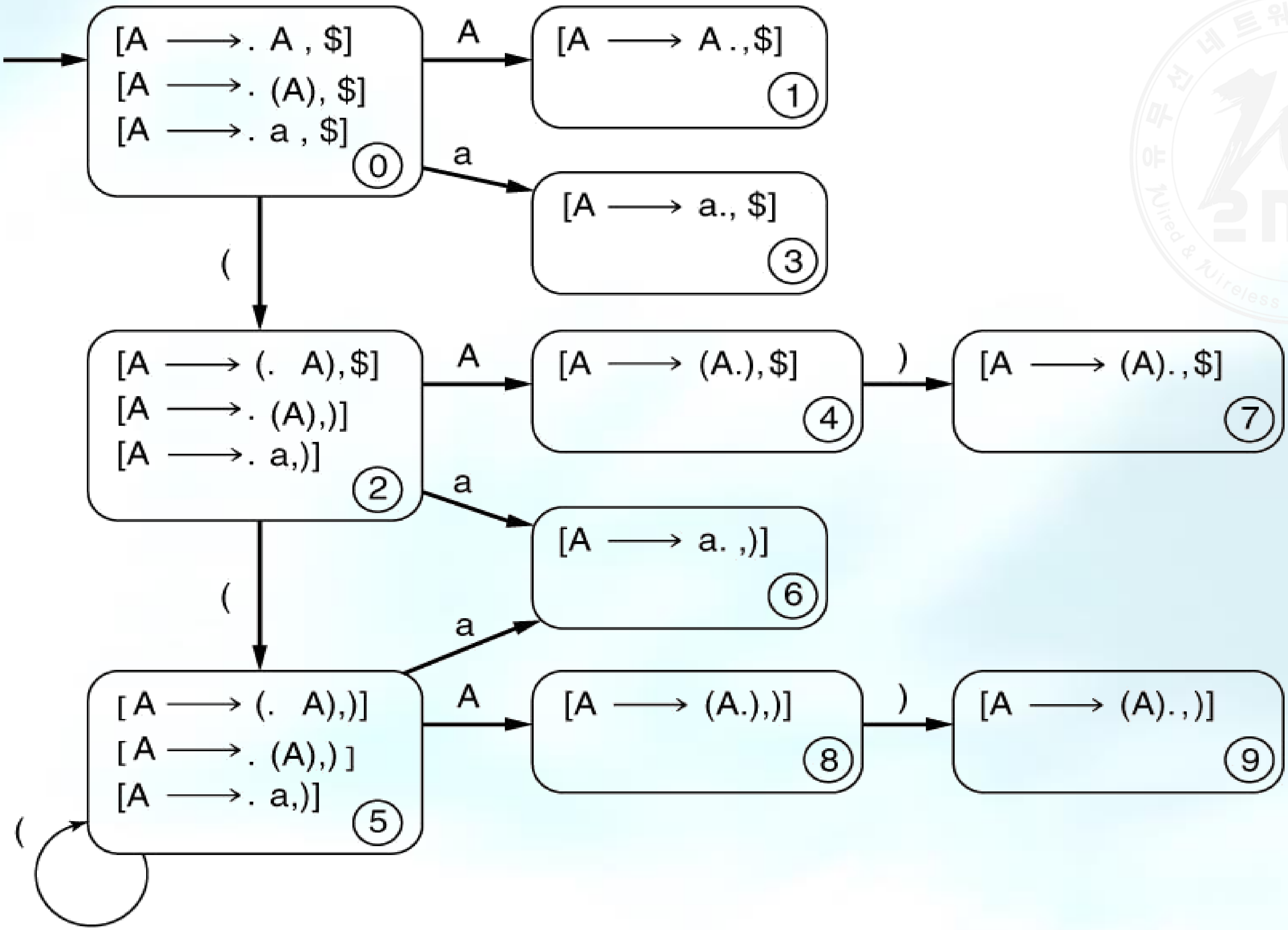
    [A → .(A), $]

    [A → .a , $]
  - State 2: [A → (. A), $] First($)) = {)}

    [A → .(A),)]

    [A → .a,)]

P.220 figure 5.7 – LR(1)



General LR(1) parsing algorithm

- ...
- 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. ... 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.Aβ, b]. Push A onto the stack, and push the state containing the item [B → αA.β, b].
- …

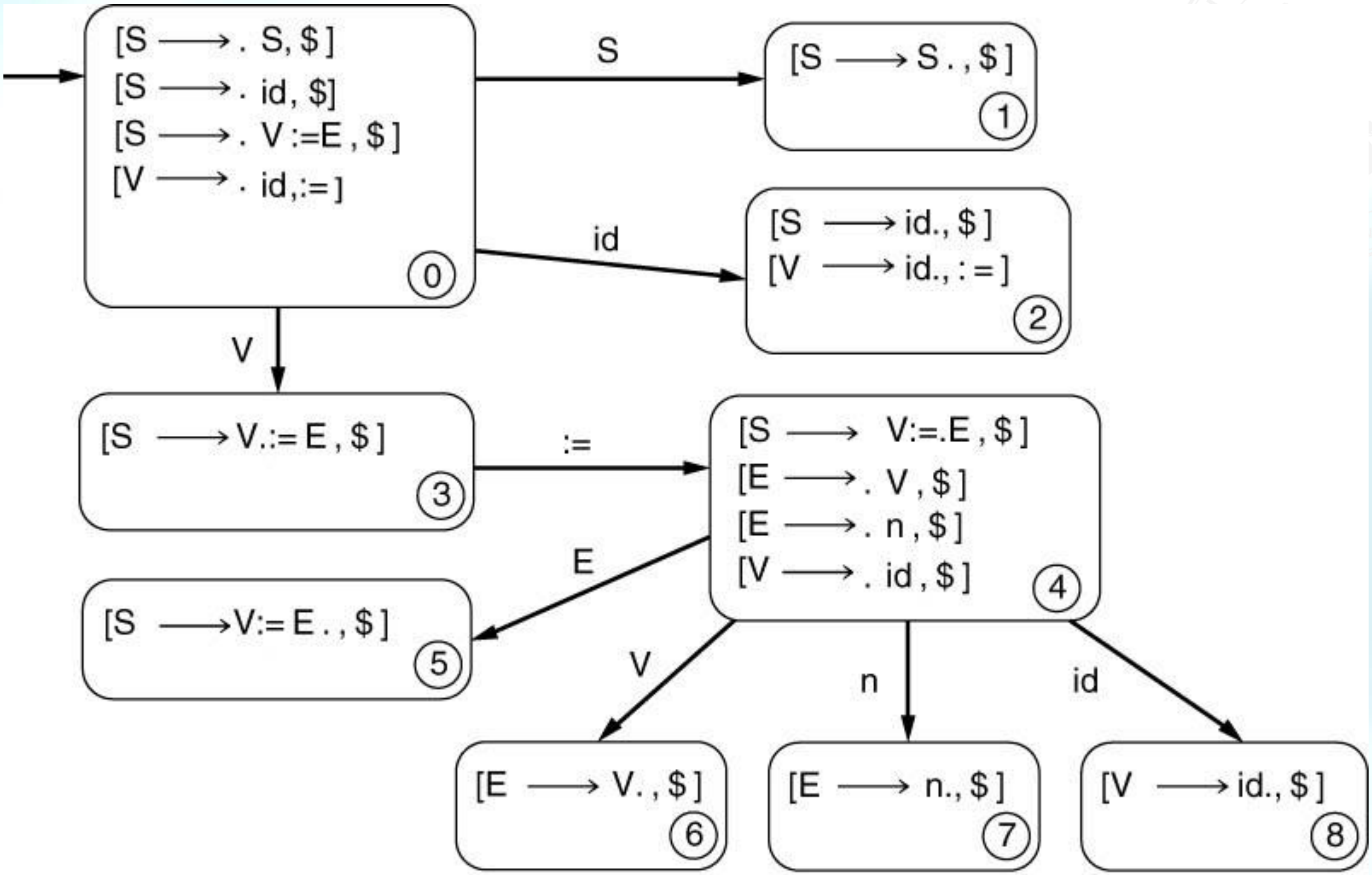
P.222 table 5.10

Table 5.10
General LR(1) parsing table for Example 5.14

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		

§

P.223 figure 5.8 – LR(1)

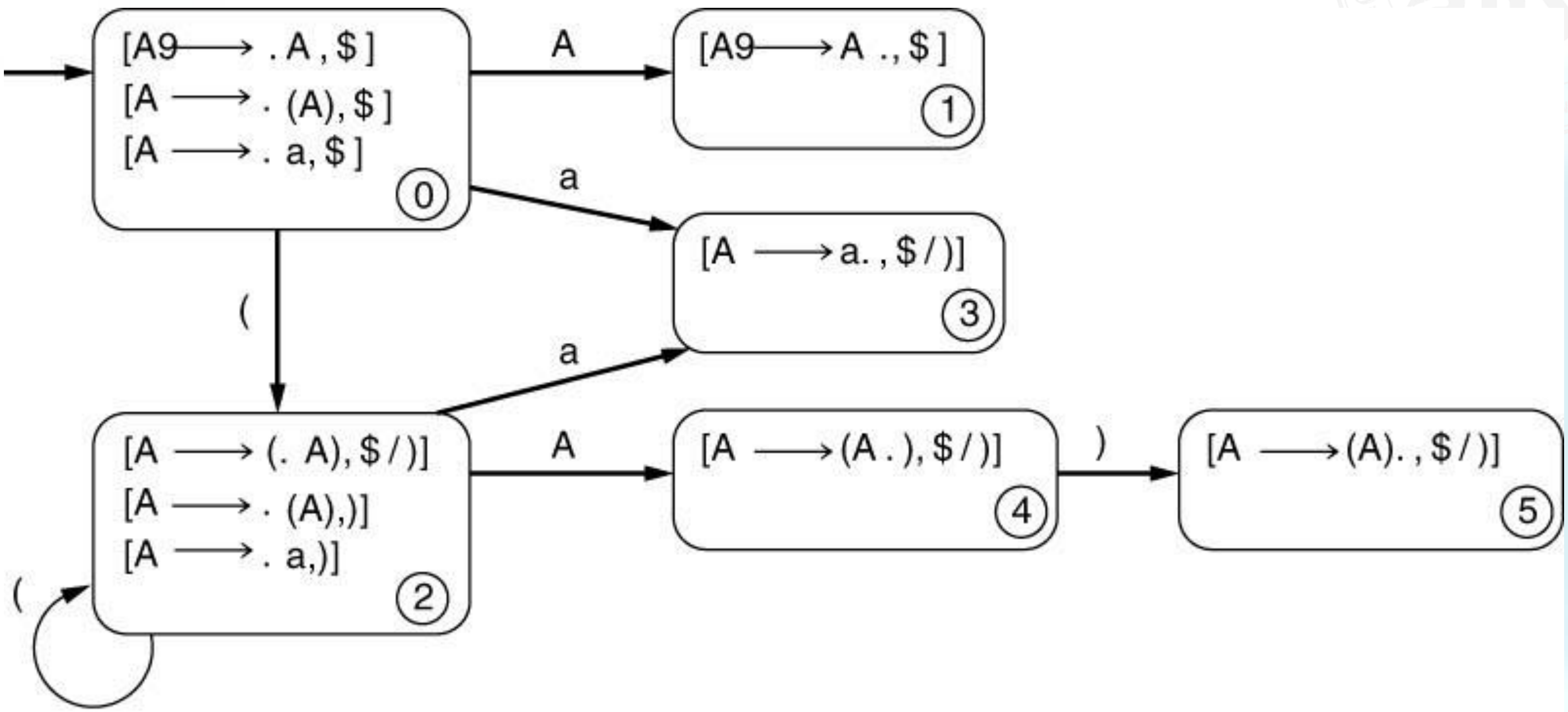


LALR(1) Parsing

- The size of the DFA of sets of LR(1) items is due in part to the existence of many different states with different lookahead symbols
- FIRST PRINCIPLE OF LALR(1) PARSING
  - The core of a state of the DFA of LR(1) items is a set 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 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.
- LALR(1) item
  - [A → α.β, a/b/c].

LR(1)을 LR(0)의 같은 core끼리 모은 후 , lookahead 합치기 !!

P.225 figure 5.9 – LALR(1)



Yacc

- A parser generator
- Yet another compiler-compiler
- Input
  - A specification of the syntax of a language
  - <filename>.y
- Output
  - A parse procedure for that language
  - y.tab.c, ytab.c or <filename>.tab.c

Yacc

- A specification file

  {definitions}
  %%
  {rules}
  %%
  {auxiliary routines}

Yacc basics

- Definitions
  - Information about the tokens, data types, and grammar rules.
  - Any C code that must go directly in the output file.

exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | *number*

Figure 5.10
Yacc definition for a simple calculator program

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp     { printf("%d\n",$1);}
        ; /* allows printing of the result */

exp     : exp '+' term  {$$ = $1 + $3;}
        | exp '-' term  {$$ = $1 - $3;}
        | term   {$$ = $1;}
        ;

term    : term '*' factor {$$ = $1 * $3;}
        | factor  {$$ = $1;}
        ;

factor       : NUMBER         {$$ = $1;}
             | '(' exp ')'    {$$ = $2;}
             ;

%%
```

Yacc basics

- Rules
  - Grammar rules
    in a modified BNF form
  - Actions in C code
    (used in a reduction)

exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | *number*

Figure 5.10
Yacc definition for a simple calculator program

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp     { printf("%d\n",$1);}
        ; /* allows printing of the result */

exp    : exp '+' term  {$$ = $1 + $3;}
       | exp '-' term  {$$ = $1 - $3;}
       | term   {$$ = $1;}
       ;

term   : term '*' factor {$$ = $1 * $3;}
       | factor  {$$ = $1;}
       ;

factor       : NUMBER         {$$ = $1;}
             | '(' exp ')'    {$$ = $2;}
             ;

%%
```

Yacc basics

- Auxiliary routines

exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | *number*

```
%%

main()
{ return yyparse();
}

int yylex(void)
{ int c;
  while((c = getchar()) == ' ');
  /* eliminates blanks */
  if ( isdigit(c) ) {
    ungetc(c,stdin);
    scanf("%d",&yylval);
    return(NUMBER);
  }
  if (c == '\n') return 0;
  /* makes the parse stop */
  return(c);
}

int yyerror(char * s)
{ fprintf(stderr,"%s\n",s);
  return 0;
} /* allows for printing of an error message */
```

Yacc options

- -d (heaDer file)
- -v (verbose)
  - y.output
  - yacc –v calc.y

```
%token NUMBER
%%
command     : exp
            ;
exp         : exp '+' term
            | exp '-' term
            | term
            ;
term        : term '*' factor
            | factor
            ;
factor      : NUMBER
            | '(' exp ')'
            ;
```

Yacc options – verbose option


```
state 0
        $accept : _command $end

        NUMBER  shift 5
        (  shift 6
        .  error

        command  goto 1
        exp  goto 2
        term  goto 3
        factor  goto 4

state 1
        $accept :  command_$end

        $end  accept
        .  error

state 2
        command :  exp_    (1)
        exp :  exp_+ term
        exp :  exp_- term

        +  shift 7
        -  shift 8
        .  reduce 1
```

```
state 3
        exp :  term_    (4)
        term :  term_* factor

        *  shift 9
        .  reduce 4

state 4
        term :  factor_    (6)

        .  reduce 6

state 5
        factor :  NUMBER_    (7)

        .  reduce 7
```

Yacc options

```
state 6
        factor :   (_exp )

        NUMBER   shift 5
        (   shift 6
        .   error

        exp   goto 10
        term   goto 3
        factor   goto 4

state 7
        exp :   exp +_term

        NUMBER   shift 5
        (   shift 6
        .   error

        term   goto 11
        factor   goto 4
```

```
state 8
        exp :   exp -_term

        NUMBER   shift 5
        (   shift 6
        .   error

        term   goto 12
        factor   goto 4

state 9
        term :   term *_factor

        NUMBER   shift 5
        (   shift 6
        .   error

        factor   goto 13
```

Yacc options


```
state 10
     exp :  exp_+ term
     exp :  exp_- term
     factor :  ( exp_)

     +   shift 7
     -   shift 8
     )   shift 14
     .   error

state 11
     exp :  exp + term_    (2)
     term :  term_* factor

     *   shift 9
     .   reduce 2

state 12
     exp :  exp - term_    (3)
     term :  term_* factor

     *   shift 9
     .   reduce 3

state 13
     term :  term * factor_   (5)

     .   reduce 5

state 14
     factor :  ( exp )_    (8)

      .   reduce 8

8/127 terminals, 4/600 nonterminals
9/300 grammar rules, 15/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
9/601 working sets used
memory: states, etc. 36/2000, parser 11/4000
9/601 distinct lookahead sets
6 extra closures
18 shift entries, 1 exceptions
8 goto entries
4 entries saved by goto default
Optimizer space used: input 50/2000, output 218/4000
218 table entries, 202 zero
maximum spread: 257, maximum offset: 43
```

Yacc options

Table 5.11
Parsing table corresponding to the Yacc output of Figure 5.12

State	Input							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				

Parsing Conflicts and Disambiguating Rules

- shift/reduce conflict
  - shift > reduce
  - Table 5.9 (p. 215) for example 5.12 (p. 213)

- reduce/reduce conflict
  - The reduction rule listed first is preferred.

Reduce-Reduce Conflict

Figure 5.13
Yacc output file for the grammar of Example 5.18

```
state 0
        $accept : _S $end

        a  shift 4
        .  error

        S  goto 1
        A  goto 2
        B  goto 3

state 1
        $accept :  S_$end

        $end  accept
        .   error

state 2
        S  :  A_    (1)

        .  reduce 1

state 3
        S  :  B_    (2)

        .  reduce 2
```

```
4: reduce/reduce conflict (red'ns 3 and 4 ) on $end
state 4
        A  :  a_    (3)
        B  :  a_    (4)

        .  reduce 3

Rule not reduced:   B  :  a

3/127 terminals, 3/600 nonterminals
5/300 grammar rules, 5/1000 states
0 shift/reduce, 1 reduce/reduce conflicts reported
...
```

$$S \rightarrow A \mid B$$
$$A \rightarrow a$$
$$B \rightarrow a$$

Operator precedence and associativity

Figure 5.14
Yacc specification for a simple calculator with ambiguous grammar and precedence and associativity rules for operators

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%left '+' '-'
%left '*'

%%

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

exp     : NUMBER        {$$ = $1;}
        | exp '+' exp   {$$ = $1 + $3;}
        | exp '-' exp   {$$ = $1 - $3;}
        | exp '*' exp   {$$ = $1 * $3;}
        | '(' exp ')'   {$$ = $2;}
        ;

%%
/* auxiliary procedure declarations as in Figure 5.10 */
```

Tracing the Execution of a Yacc Parser

- #define YYDEBUG 1

Figure 5.15
Tracing output using `yydebug` for the Yacc parser generated by Figure 5.10, given the input `2+3`

```
Starting parse
Entering state 0
Input: 2+3
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0
Entering state 3
Next token is '+'
Reducing via rule 4, term -> exp
state stack now 0
Entering state 2
Next token is '+'
Shifting token '+', Entering state 7
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0 2 7
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0 2 7
Entering state 11
Now at end of input.
Reducing via rule 2, exp '+' term -> exp
state stack now 0
Entering state 2
Now at end of input.
Reducing via rule 1, exp -> command
5
state stack now 0
Entering state 1
Now at end of input.
```

Arbitrary Value Types in Yacc


```
%{
    …
    #define YYSTYPE double
    …
%}
```

Arbitrary Value Types in Yacc


```
...
%token NUMBER

%union { double val;
         char op; }

%type <val> exp term factor NUMBER

%type <op> addop mulop

%%
command : exp       { printf("%d\n",$1);}
        ;
exp   : exp op term { switch ($2) {
                       case '+': $$ = $1 + $3; break;
                       case '-': $$ = $1 - $3; break;
                       }
                     }
      | term   {$$ = $1;}
      ;
op : '+' { $$ = '+'; }
   | '-' { $$ = '-'; }
   ;
```

exp → *exp addop term* | *term*

addop → + | -

Embedded Actions in Yacc


```
decl    : type { current_type = $1; }
          var_list
        ;
type    : INT { $$ = INT_TYPE; }
        | FLOAT {  $$ = FLOAT_TYPE; }
        ;
var_list : var_list ',' ID
             { setType(tokenString,current_type);}
         | ID
             { setType(tokenString,current_type);}
         ;
```

Embedded Actions in Yacc

Table 5.12

Yacc internal names and definition mechanisms

Yacc internal name	Meaning/Use
`y.tab.c`	Yacc output file name
`y.tab.h`	Yacc-generated header file containing token definitions
`yyparse`	Yacc parsing routine
`yylval`	value of current token in stack
`yyerror`	user-defined error message printer used by Yacc
`error`	Yacc error pseudotoken
`yyerrok`	procedure that resets parser after error
`yychar`	contains the lookahead token that caused an error
`YYSTYPE`	preprocessor symbol that defines the value type of the parsing stack
`yydebug`	variable which, if set by the user to 1, causes the generation of runtime information on parsing actions

Yacc definition mechanism	Meaning/Use
`%token`	defines token preprocessor symbols
`%start`	defines the start nonterminal symbol
`%union`	defines a union **YYSTYPE**, allowing values of different types on parser stack
`%type`	defines the variant union type returned by a symbol
`%left %right %nonassoc`	defines the associativity and precedence (by position) of operators