

# 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**

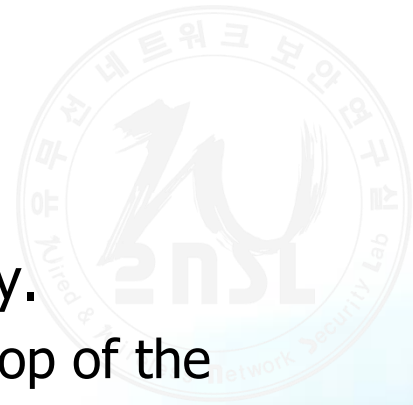
- LR(0) parsing
- LR(1) parsing : powerful but complex
- SLR(1) parsing : Simple LR(1)
- LALR(1) parsing : Lookahead LR(1)
  - More powerful than SLR(1), and less complex than LR(1)



# 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  $\alpha$  at the top of the stack to a nonterminal  $A$ , given the BNF choice  $A \rightarrow \alpha$ .
- If the stack contains the start symbol and the input is empty, accept.
- Shift-reduce parser



# Example: table 5.1

$$S' \rightarrow S$$

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

	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



# Example: table 5.2



$$E' \rightarrow E$$

$$E \rightarrow E + n \mid n$$

	Parsing stack	Input	Action
1	\$	<b>n + n</b> \$	shift
2	\$ <b>n</b>	+ <b>n</b> \$	reduce $E \rightarrow \mathbf{n}$
3	\$ $E$	+ <b>n</b> \$	shift
4	\$ $E$ +	<b>n</b> \$	shift
5	\$ $E$ + <b>n</b>	\$	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.
- Sometimes, the next token in the input may also need to be consulted as a lookahead.
- 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
  - Each of the intermediate strings of terminals and nonterminals in the rightmost derivation.
- 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.



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



<http://usecurity.hanyang.ac.kr>

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





# P.202 example 5.3

$$\begin{array}{l} S' \rightarrow S \\ S \rightarrow ( S ) S \mid \varepsilon \end{array}$$

$$\begin{array}{l} S' \rightarrow . S \\ S' \rightarrow S . \\ S \rightarrow . ( S ) S \\ S \rightarrow ( . S ) S \\ S \rightarrow ( S . ) S \\ S \rightarrow ( S ) . S \\ S \rightarrow ( S ) S . \\ S \rightarrow . \end{array}$$



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



<http://usecurity.hanyang.ac.kr>

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



# P.202 example 5.4



$$E' \rightarrow . E$$

$$E' \rightarrow E .$$

$$E \rightarrow . E + n$$

$$E \rightarrow E . + n$$

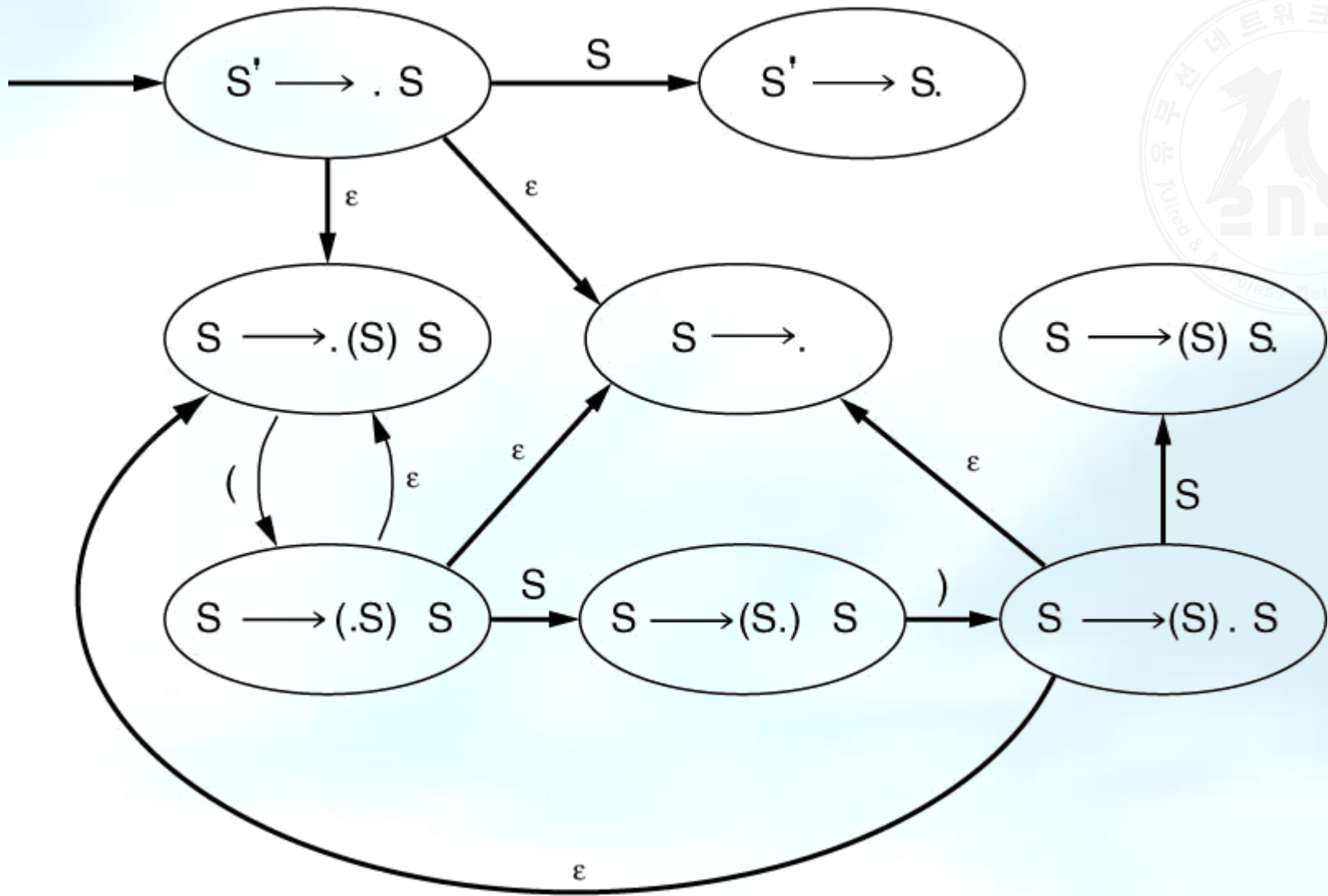
$$E \rightarrow E + . n$$

$$E \rightarrow E + n .$$

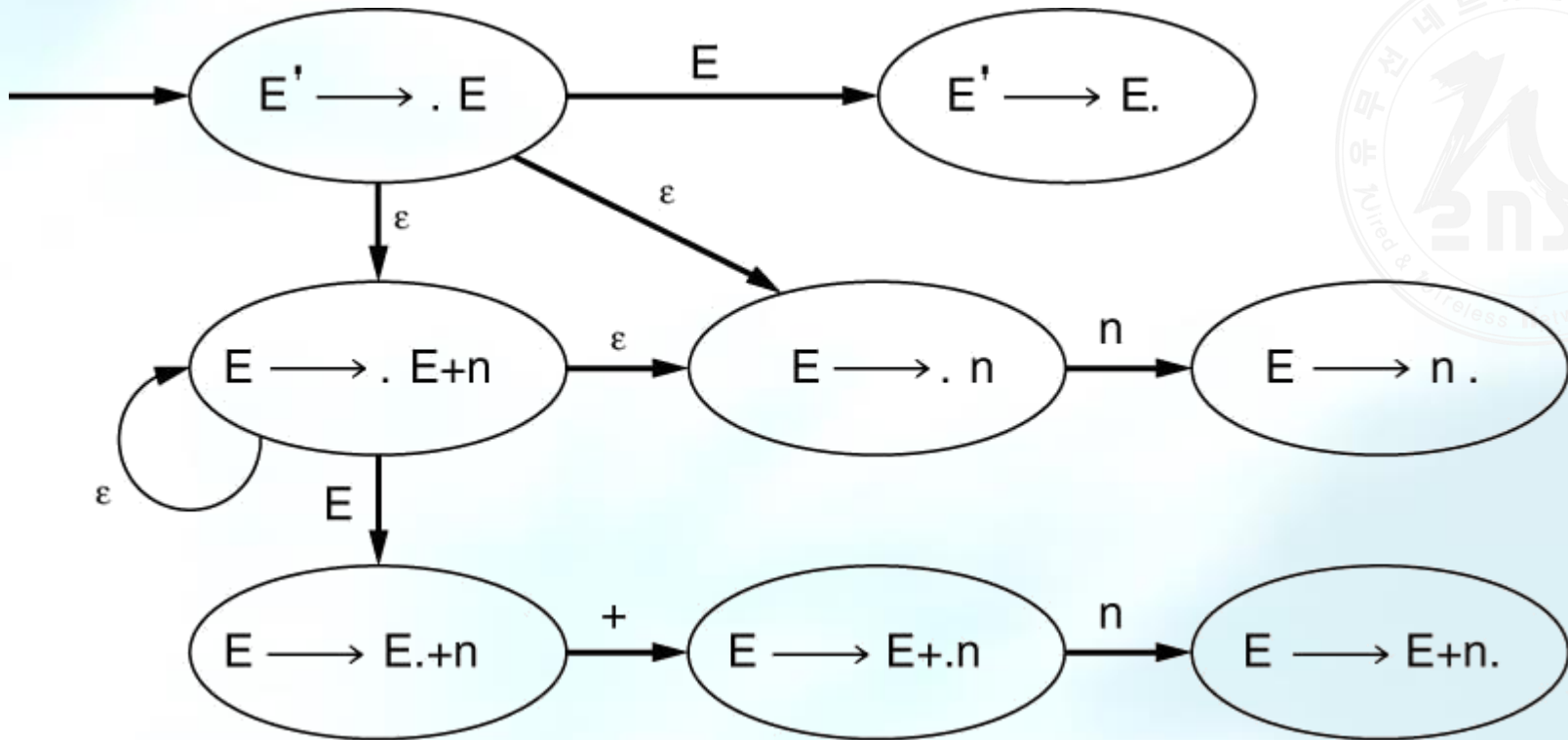
$$E \rightarrow . n$$

$$E \rightarrow n .$$

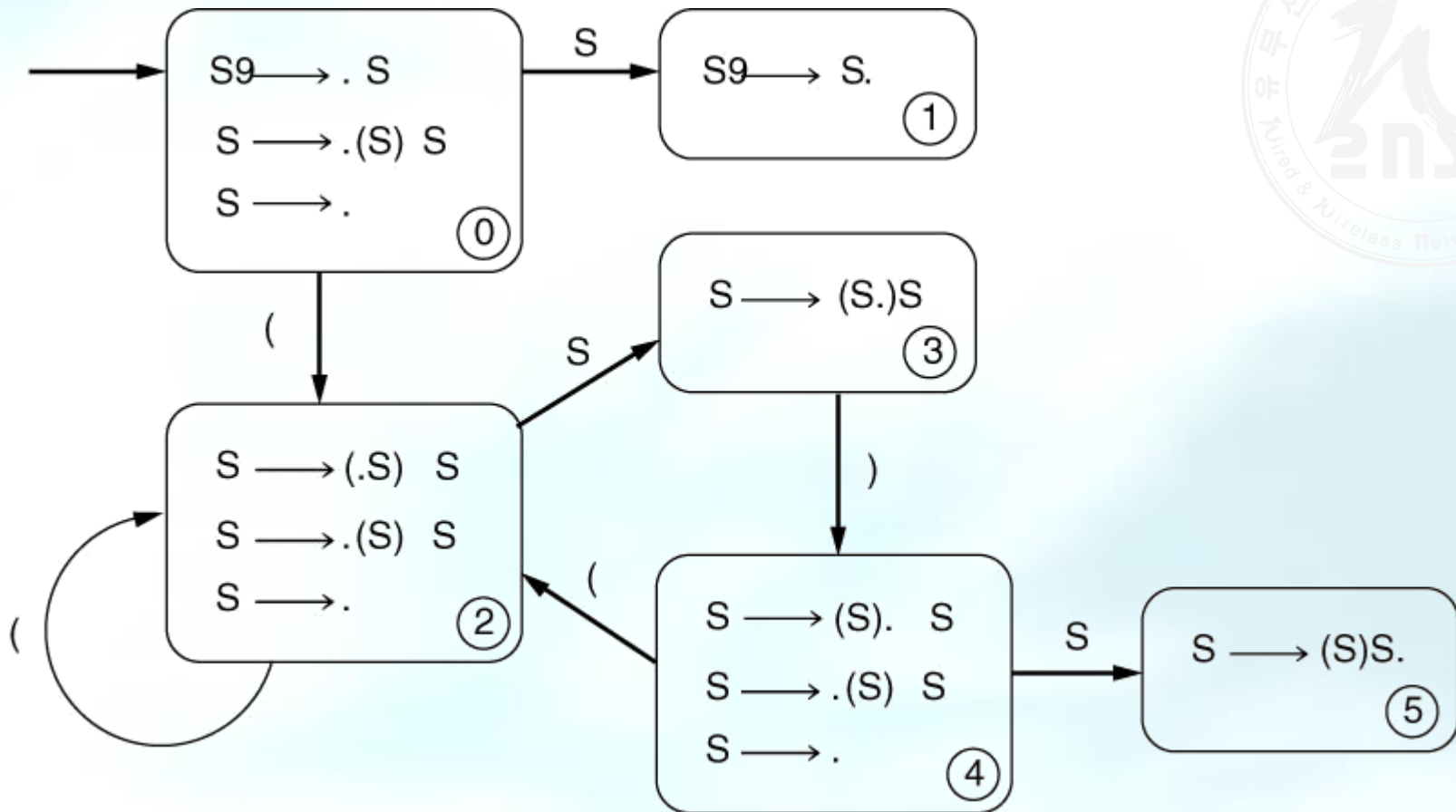
# P.204 figure 5.1



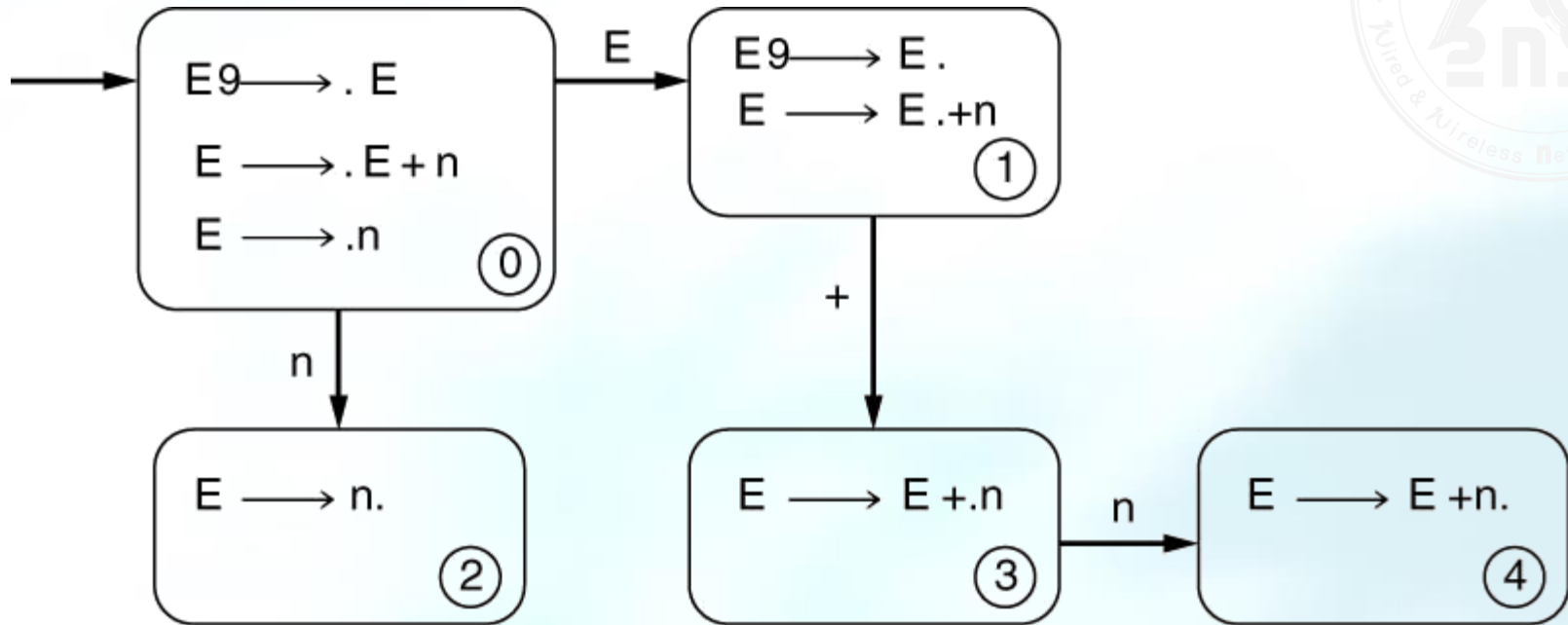
# P.204 figure 5.2



# P.205 figure 5.3



# P.205 figure 5.4



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



<http://usecurity.hanyang.ac.kr>

- 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





# The LR(0) Parsing



<http://usecurity.hanyang.ac.kr>

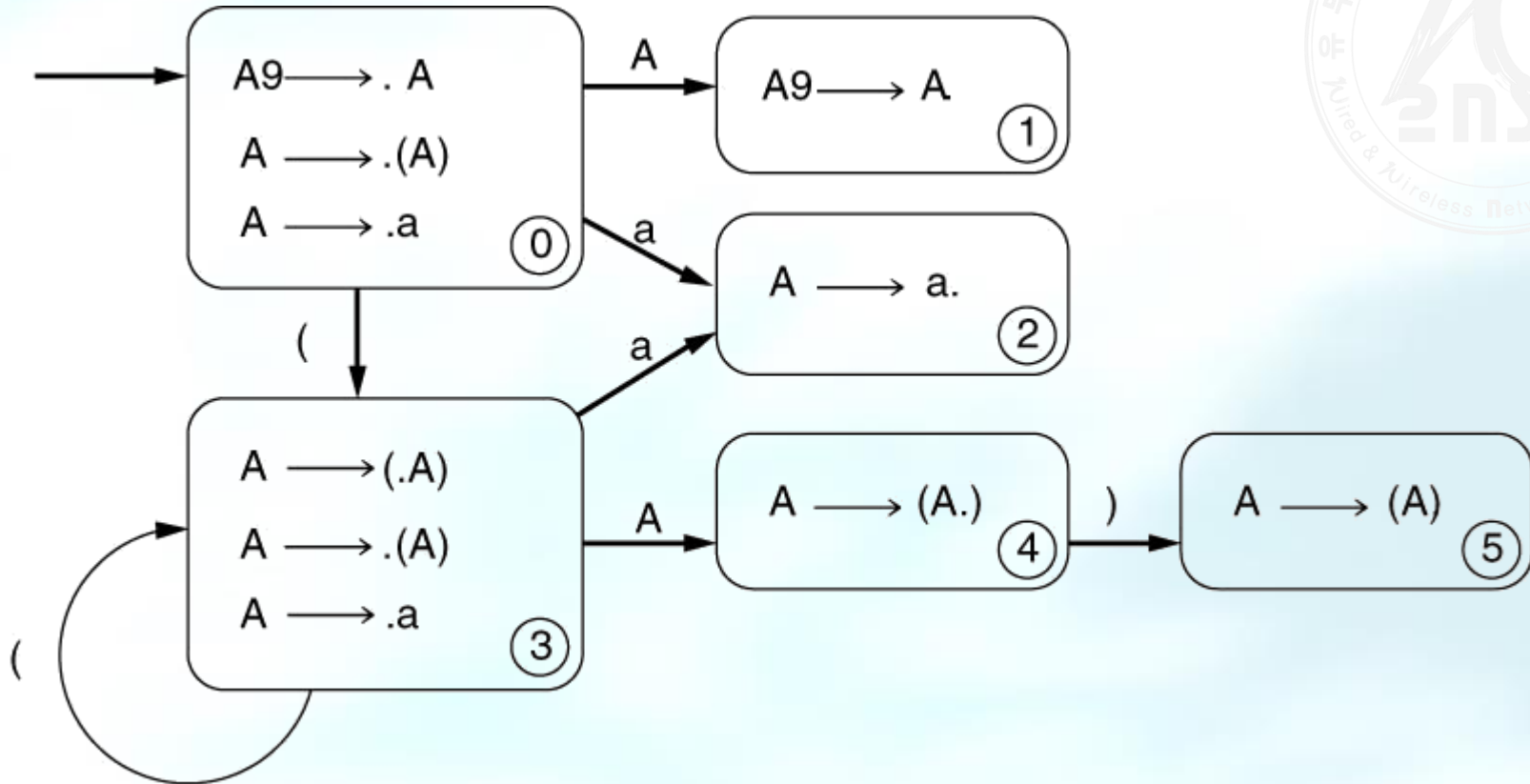


Parsing stack	Input
\$ 0 n 2	<i>Rest of InputString</i> \$

# 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.
- 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
- reduce-reduce conflict

# 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

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	$A \rightarrow ( A )$				

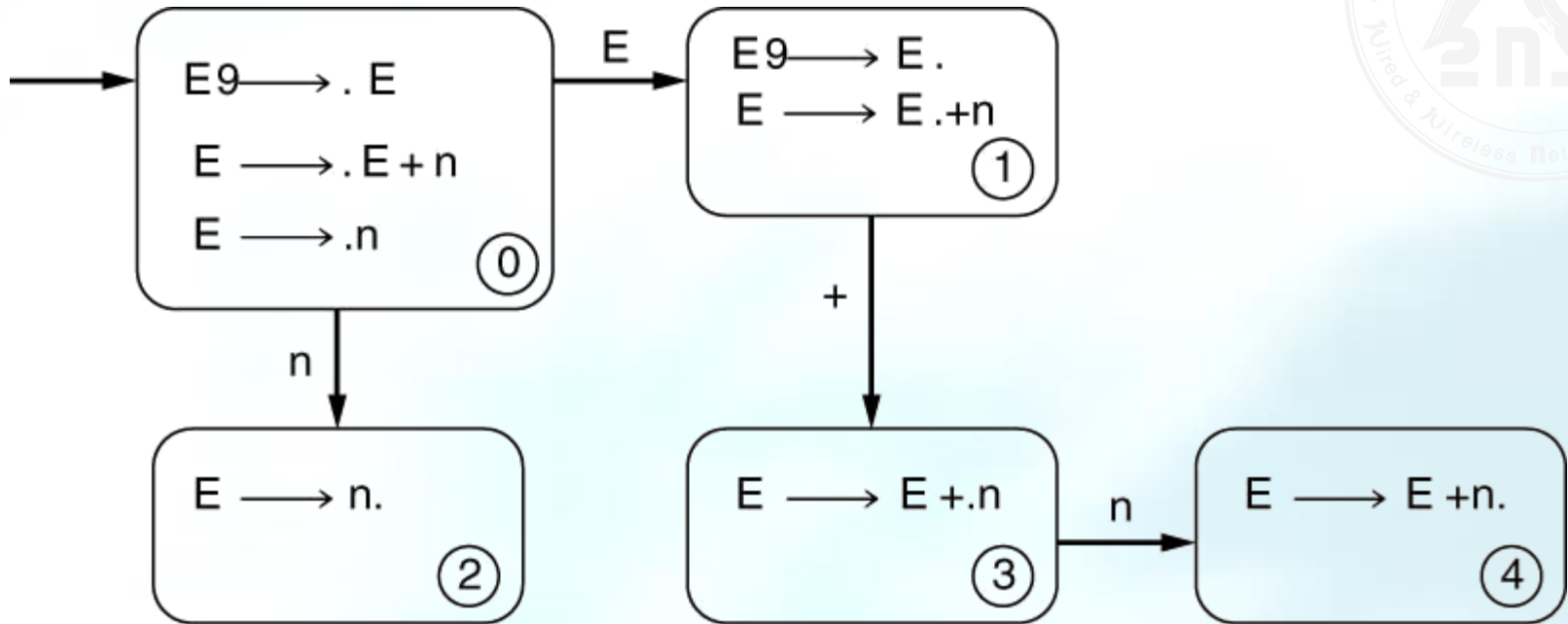
# The SLR(1) Parsing



<http://usecurity.hanyang.ac.kr>

- 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  $\text{Follow}(A)$ , then the action is to reduce by the rule  $A \rightarrow \gamma$ .
  - ...

# 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	<b>n + n + n \$</b>	shift 2
2	\$ 0 <b>n 2</b>	+ <b>n + n \$</b>	reduce $E \rightarrow \mathbf{n}$
3	\$ 0 $E 1$	+ <b>n + n \$</b>	shift 3
4	\$ 0 $E 1 + 3$	<b>n + n \$</b>	shift 4
5	\$ 0 $E 1 + 3$ <b>n 4</b>	+ <b>n \$</b>	reduce $E \rightarrow E + \mathbf{n}$
6	\$ 0 $E 1$	+ <b>n \$</b>	shift 3
7	\$ 0 $E 1 + 3$	<b>n \$</b>	shift 4
8	\$ 0 $E 1 + 3$ <b>n 4</b>	<b>\$</b>	reduce $E \rightarrow E + \mathbf{n}$
9	\$ 0 $E 1$	<b>\$</b>	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 $S \rightarrow \varepsilon$
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

# Disambiguating Rules for Parsing Conflicts



<http://usecurity.hanyang.ac.kr>

- P.213 example 5.12

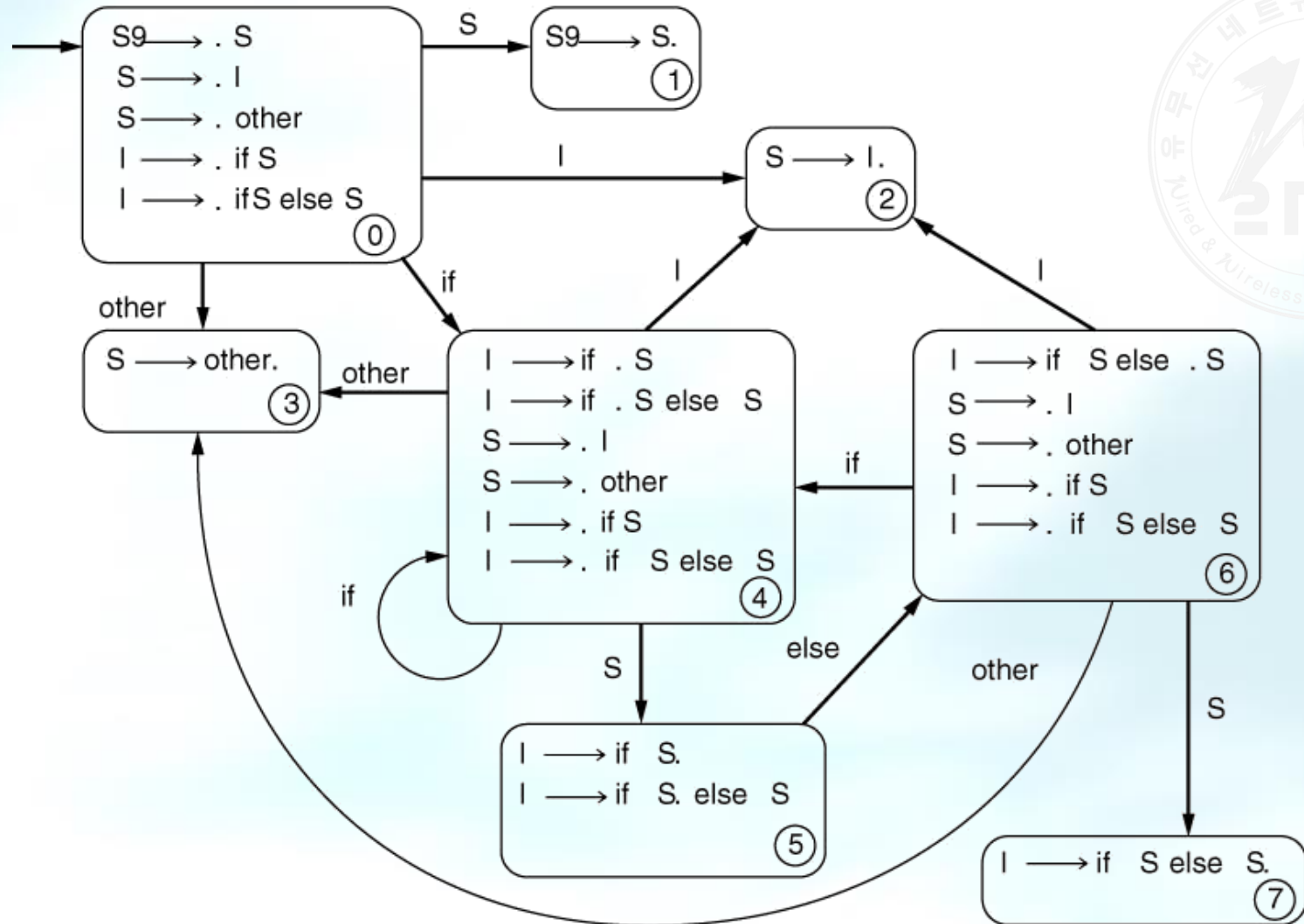
*statement*  $\rightarrow$  *if-stmt* | **other**

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

*exp*  $\rightarrow$  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	\$	<i>S</i>	<i>I</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



<http://usecurity.hanyang.ac.kr>

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

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

## Example 5.13

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





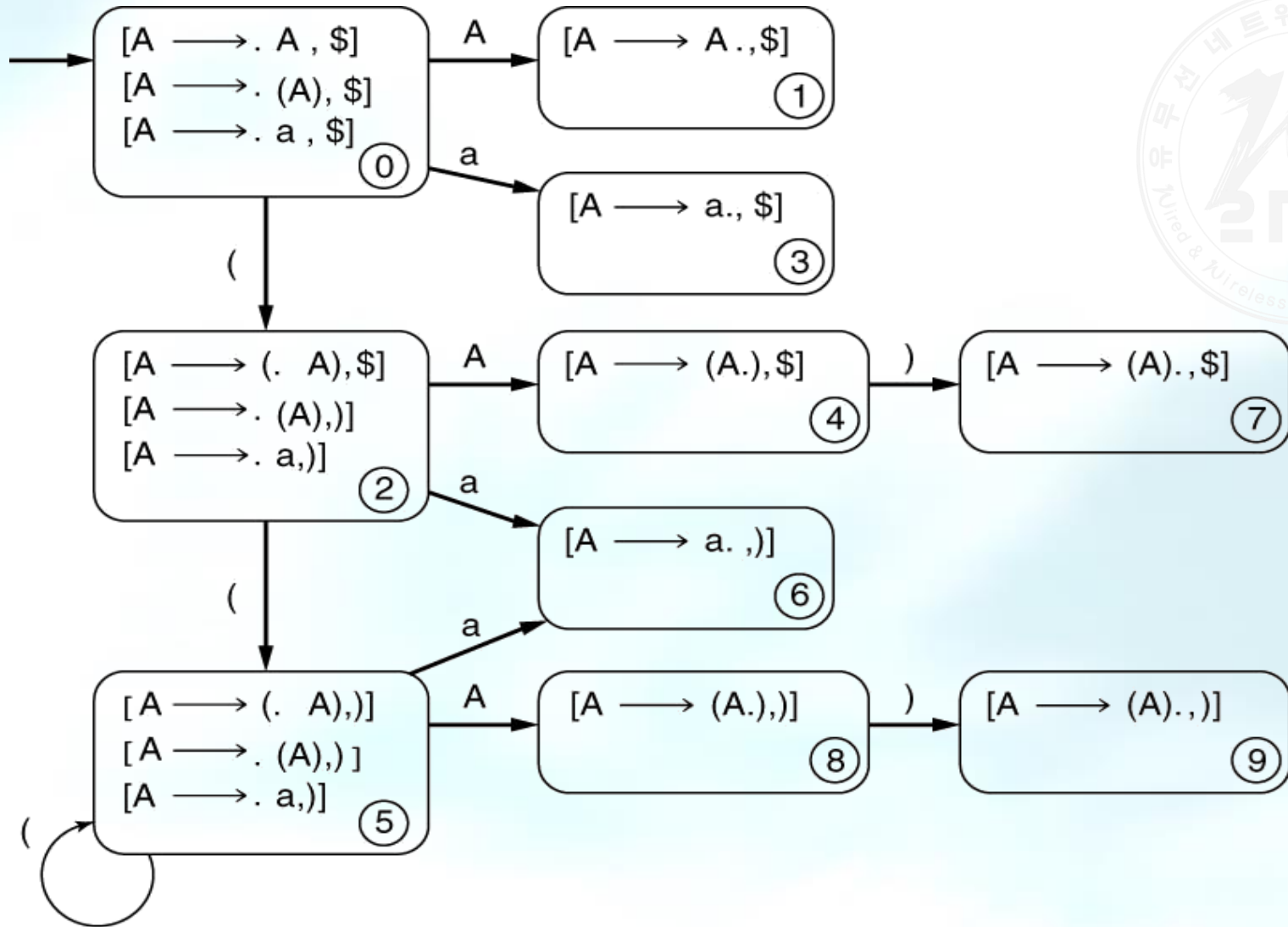
# Definition of LR(1) transitions

- Part 2
  - Given an LR(1) item  $[A \rightarrow \alpha.B\gamma, a]$ , where  $B$  is a nonterminal, there are  $\epsilon$ -transitions to items  $[B \rightarrow \cdot\beta, b]$  for every production  $B \rightarrow \beta$  and for every token  $b$  in  $\text{First}(\gamma a)$ .
- Examples
  - $A \rightarrow ( A ) \mid a$
  - State 0:  $[A' \rightarrow \cdot A, \$]$ 

$$\begin{array}{l} [A \rightarrow \quad \quad \quad ] \\ [A \rightarrow \quad \quad \quad ] \end{array}$$
  - State 2:  $[A \rightarrow ( \cdot A ), \$]$ 

$$\begin{array}{l} [A \rightarrow \quad \quad \quad ] \\ [A \rightarrow \quad \quad \quad ] \end{array}$$

# P.220 figure 5.7 – LR(1)



# General LR(1) parsing algorithm

- ...
- If state  $s$  contains the complete LR(1) item  $[A \rightarrow \alpha., a]$ , and the next token in the input string is  $a$ , then the action is to reduce by the rule  $A \rightarrow \alpha.$  ... The new state is computed as follows. Remove the string  $\alpha$  and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of  $\alpha$  began. By construction, this state must contain an LR(1) item of the form  $[B \rightarrow \alpha.A\beta, b]$ . Push  $A$  onto the stack, and push the state containing the item  $[B \rightarrow \alpha.A.\beta, b]$ .
- ...

# P.222 table 5.10



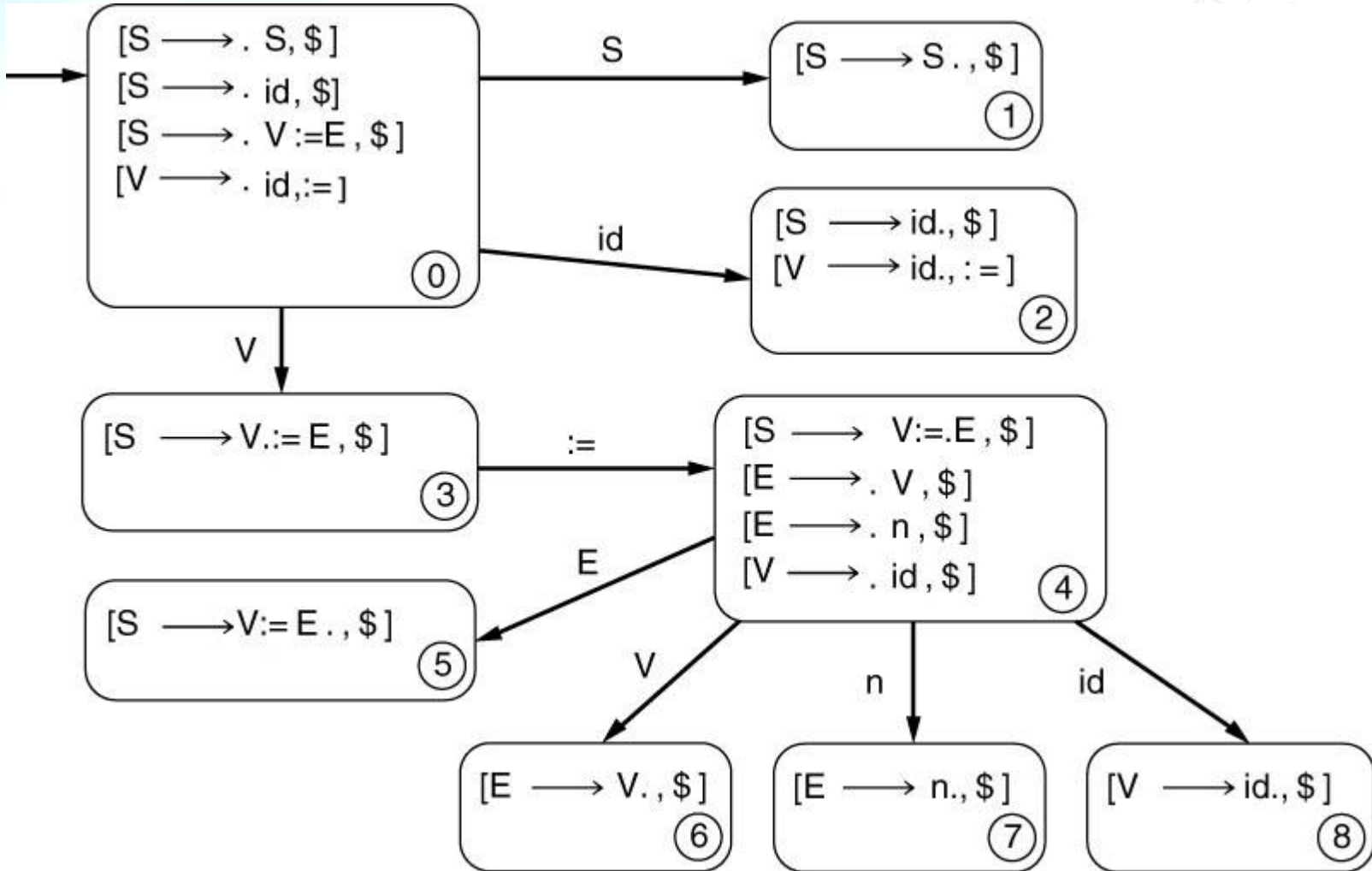
Table 5.10

General LR(1) parsing table  
for Example 5.14

State	Input				Goto
	(	a	)	\$	
0	s2	s3		accept	1
1					
2	s5	s6		r2	4
3					
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

\$

# P.223 figure 5.8 – LR(1)



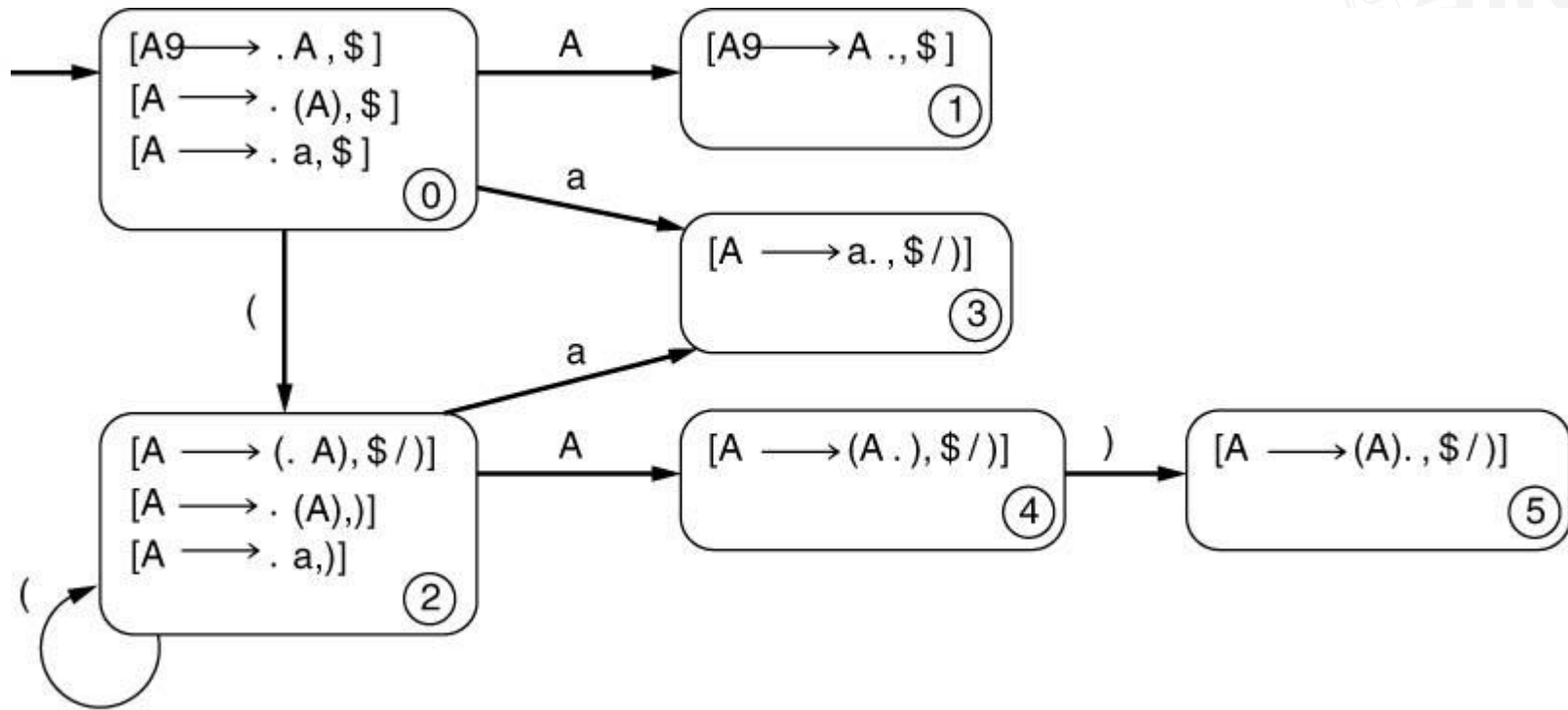
# LALR(1) Parsing



<http://usecurity.hanyang.ac.kr>

- 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  $s_1$  and  $s_2$  of the DFA of LR(1) items that have the same core, suppose there is a transition on  $X$  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.
- LALR(1) item
  - $[A \rightarrow \alpha.\beta, a/b/c]$ .

# 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





- 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*  $\rightarrow$  *exp addop term* | *term*

*addop*  $\rightarrow$  + / -

*term*  $\rightarrow$  *term mulop factor* | *factor*

*mulop*  $\rightarrow$  \*

*factor*  $\rightarrow$  (*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)

$exp \rightarrow exp \text{ addop } term \mid term$

$addop \rightarrow + \mid -$

$term \rightarrow term \text{ mulop } factor \mid factor$

$mulop \rightarrow *$

$factor \rightarrow (exp) \mid number$

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; }
        ;

%%
```

## 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

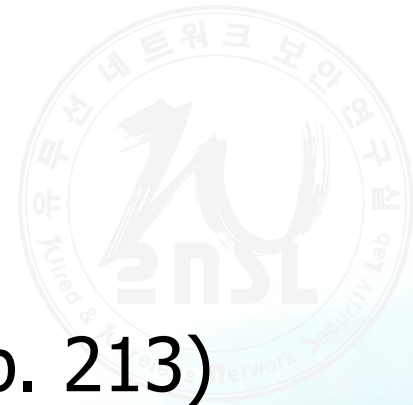
State	Input							Goto			
	NUMBER	(	+	-	*	)	\$	command	exp	term	factor
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			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



<http://usecurity.hanyang.ac.kr>

- 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



<http://usecurity.hanyang.ac.kr>

```
%{  
...  
#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
```

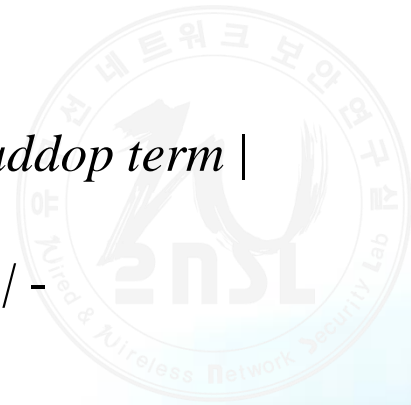
```
%%

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
	<b>y.tab.c</b>	Yacc output file name
	<b>y.tab.h</b>	Yacc-generated header file containing token definitions
	<b>yyparse</b>	Yacc parsing routine
	<b>yyval</b>	value of current token in stack
	<b>yyerror</b>	user-defined error message printer used by Yacc
	<b>error</b>	Yacc error pseudotoken
	<b>yyerrok</b>	procedure that resets parser after error
	<b>yychar</b>	contains the lookahead token that caused an error
	<b>YYSTYPE</b>	preprocessor symbol that defines the value type of the parsing stack
	<b>yydebug</b>	variable which, if set by the user to 1, causes the generation of runtime information on parsing actions
Yacc definition mechanism		Meaning/Use
	<b>%token</b>	defines token preprocessor symbols
	<b>%start</b>	defines the start nonterminal symbol
	<b>%union</b>	defines a union <b>YYSTYPE</b> , allowing values of different types on parser stack
	<b>%type</b>	defines the variant union type returned by a symbol
	<b>%left %right %nonassoc</b>	defines the associativity and precedence (by position) of operators