Bottom Up parsing

CONFLICTS IN SHIFT REDUCE PARSER

- Shift reduce conflict
- Reduce reduce conflict

Shift reduce Conflict

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

If we have a shift-reduce parser in configuration

Stack	Input
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots \$$

- We can resolve above conflict by giving preference to shift

Reduce reduce conflict

(1)	$stmt$	$\rightarrow$	**id** ($parameter\_list$)
(2)	$stmt$	$\rightarrow$	$expr$:= $expr$
(3)	$parameter\_list$	$\rightarrow$	$parameter\_list$, $parameter$
(4)	$parameter\_list$	$\rightarrow$	$parameter$
(5)	$parameter$	$\rightarrow$	**id**
(6)	$expr$	$\rightarrow$	**id** ($expr\_list$)
(7)	$expr$	$\rightarrow$	**id**
(8)	$expr\_list$	$\rightarrow$	$expr\_list$, $expr$
(9)	$expr\_list$	$\rightarrow$	$expr$

Stack	Input
$\cdots$ **id** (**id**	, **id**) $\cdots$

- Same syntax for function name and array
- LA returns **id** function name and array element.

REDUCE REDUCE CONFLICT[CONTD..]

Change this to procid

(1)	$stmt$	$\rightarrow$	$\textbf{id} \; (\; parameter\_list \;)$
(2)	$stmt$	$\rightarrow$	$expr \; := \; expr$
(3)	$parameter\_list$	$\rightarrow$	$parameter\_list \; , \; parameter$
(4)	$parameter\_list$	$\rightarrow$	$parameter$
(5)	$parameter$	$\rightarrow$	$\textbf{id}$
(6)	$expr$	$\rightarrow$	$\textbf{id} \; (\; expr\_list \;)$
(7)	$expr$	$\rightarrow$	$\textbf{id}$
(8)	$expr\_list$	$\rightarrow$	$expr\_list \; , \; expr$
(9)	$expr\_list$	$\rightarrow$	$expr$

STACK	INPUT
$\cdots \; \textbf{procid} \; (\; \textbf{id}$	$, \; \textbf{id} \;) \; \cdots$

LR PARSER

- Shift reduce parser is general class of bottom up parser.
- One level down in hierarchy , LR parser.
- Types of LR parsers
  - SLR parser : simple LR – basic
  - Canonical LR parser
  - LALR : lookahead LR parser
- More complex
- So difficult to construct in hand
- LR parser generator is usually used.

Why LR parsers?

- LR parser can be constructed to recognize most of the programming languages for which CFG can be written.
- LR parser works using non backtracking shift reduce technique.
- LR parser can detect a syntactic error as soon as it is possible.
- Class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsing

Items and LR(0) automaton

- How does a shift reduce parser know when to shift and when to reduce?

Ex -

Stack	Input	Action
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2\,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2\,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2\,\$$	shift
$\$\,T *$	$\mathbf{id}_2\,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

Reduce to E or shift

ITEMS AND LR(0) AUTOMATON[CONTD..]

- An LR parser make this decision by maintaining states to keep track of where are we in a parse.
- States represent set of "items".

- An LR(0) item of a grammar G is a prodn of G with a dot at some position of the body.
- An item indicates how much of a prodn we have seen at given point in the parsing process.

Items and LR(0) automaton[contd..]

- Production $A \rightarrow XYZ$

Items are

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet YZ$

$A \rightarrow XY \bullet Z$

$A \rightarrow XYZ \bullet$

- $A \rightarrow X \bullet YZ$ indicates that we have just parsed input string derivable from X and YZ are yet to be parsed.

ITEMS AND LR(0) AUTOMATON[CONTD..]

- An item indicates how much of a prodn we have seen at given point in the parsing process.
- $A \rightarrow XYZ \bullet$

  time to reduce XYZ to A.
- So, there is a prodn $A \rightarrow \epsilon$. what is the item?

  $A \rightarrow \bullet$

ITEMS AND LR(0) AUTOMATON[CONTD..]

- Ex 2: $S' \rightarrow S$

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

  - The grammar has 3 production choices.
  - The grammar has 8 items
    - $S' \rightarrow .S$ $\qquad$ $S' \rightarrow S.$
    - $S \rightarrow .(S) S$ $\qquad$ $S \rightarrow (. S) S$
    - $S \rightarrow (S .) S$ $\qquad$ $S \rightarrow (S) . S$
    - $S \rightarrow (S) S.$ $\qquad$ $S \rightarrow .$

ITEMS AND LR(0) AUTOMATON[CONTD..]

- Ex 3: E′ → E

  E → E + n | n

  - The grammar has 3 production choices.
  - The grammar has 8 items.
    - E′ → .E   E′ → E.
    - E → . E + n   E → E . + n
    - E → E + . n   E → E + n .
    - E → .n   E → n .

Terms related

- Canonical LR(0) collection
- LR(0) automaton
- Augmented grammar
- Kernel : S'→ .S + all items without dot at leftmost of RHS
- Non kernel : All items with dot at left end except S'→.S

Closure of item sets

- closure.pdf

- I – set of items for G
- Closure(I) – 2 rules
- Initially add every item in I to closure(I).
- If $A \rightarrow \alpha \bullet B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production then add item $B \rightarrow \bullet\gamma$

Goto function

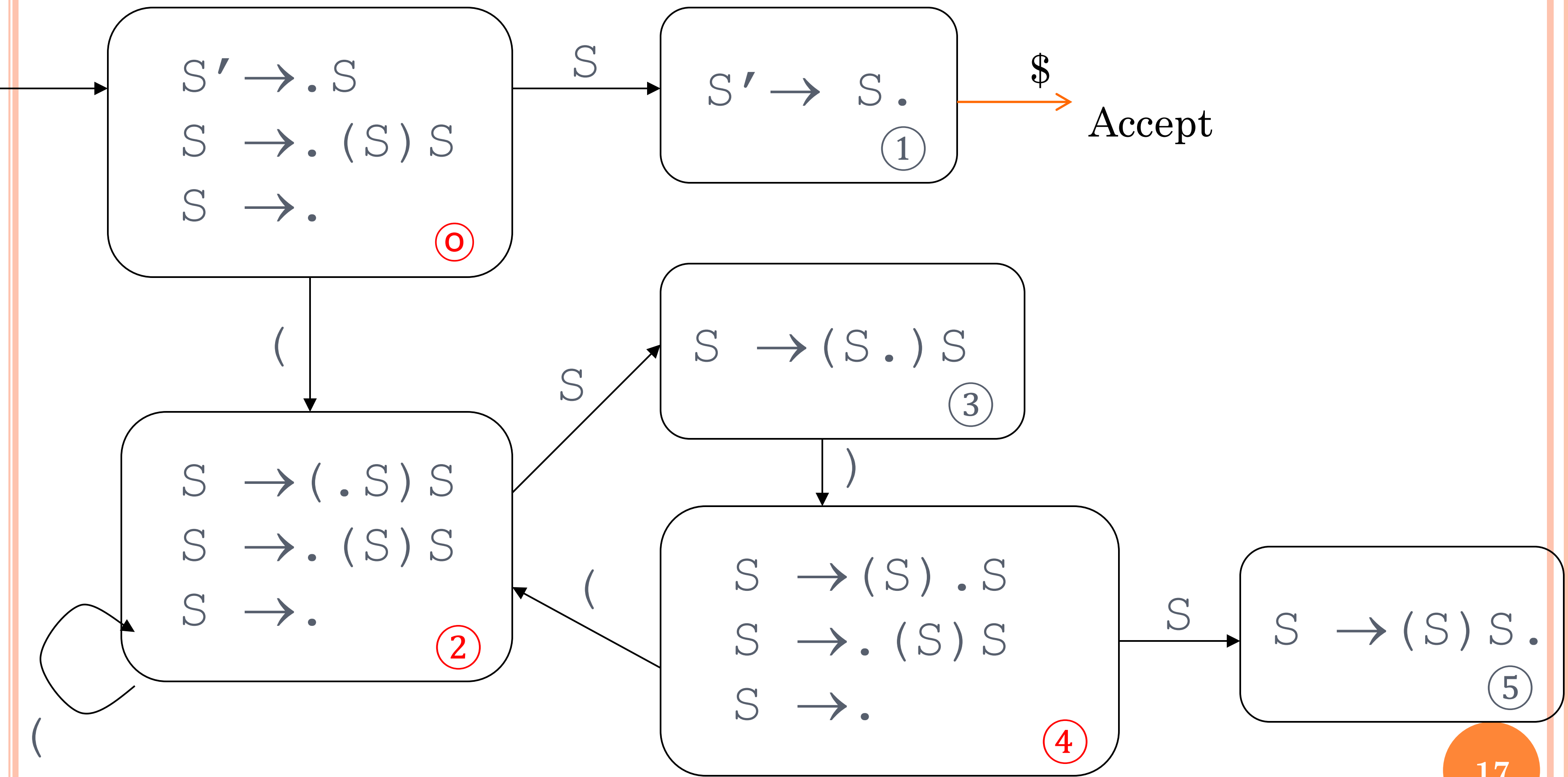
- goto.pdf

Definition : Goto(I,X) is closure of the set of all items [$A \rightarrow \alpha \bullet X \beta$] such that [$A \rightarrow \alpha X \bullet \beta$] is in I.

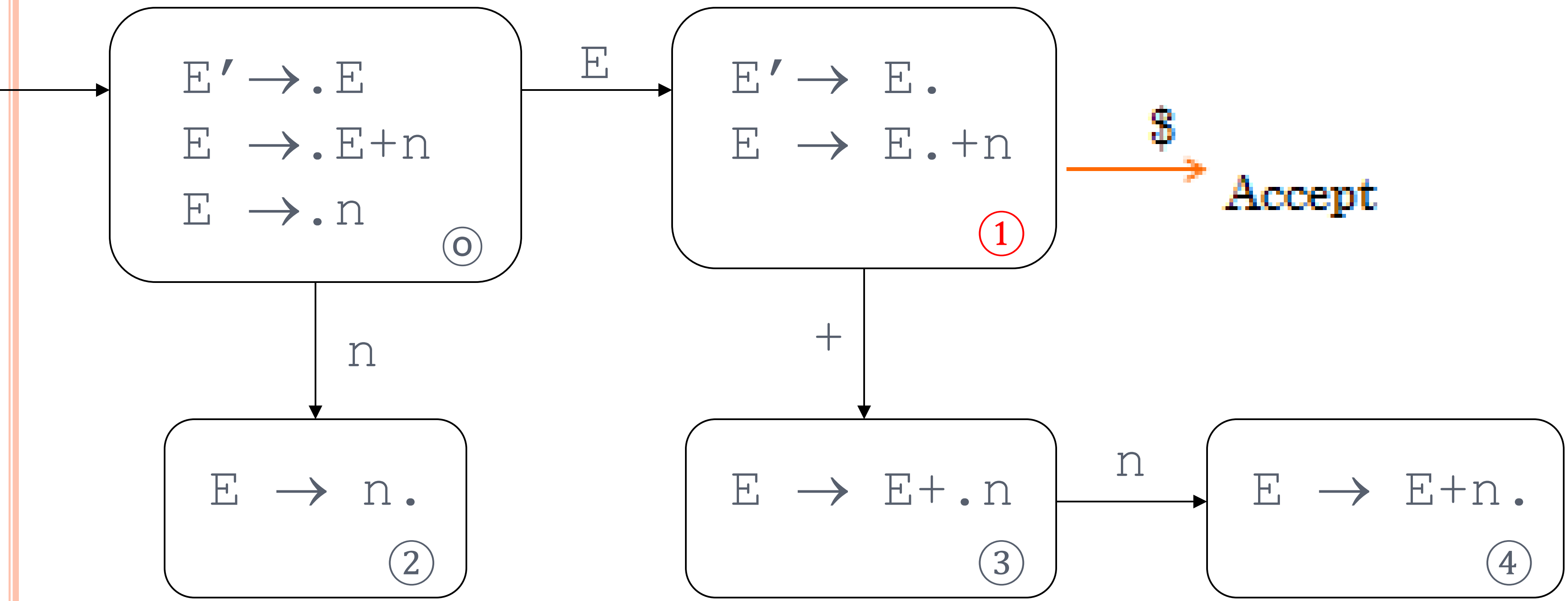
I – set of items

X – grammar symbol

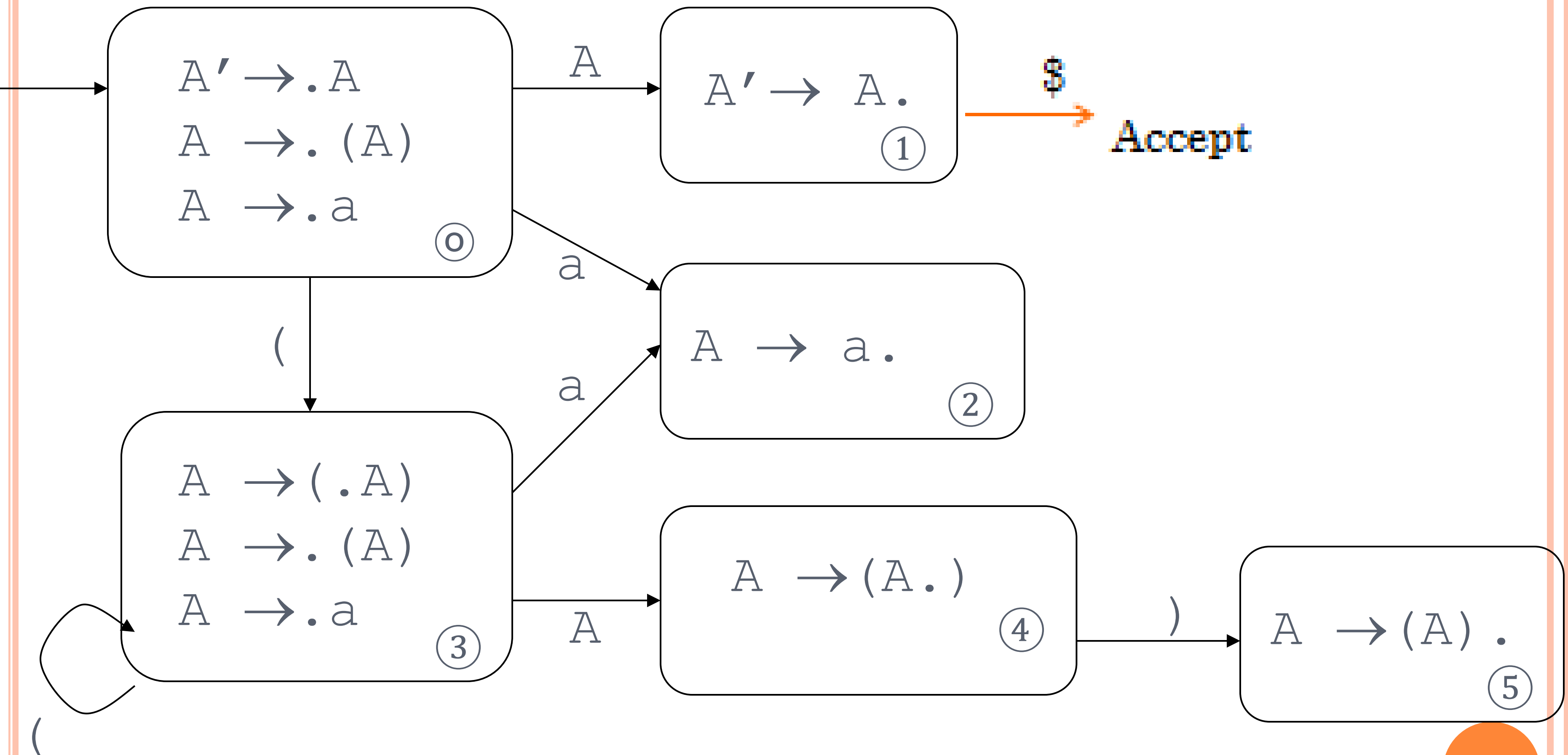
Example 1: DFA of LR(0) Items



EXAMPLE 2: DFA OF LR(0) ITEMS



Example 3: DFA of LR(0) Items



Construct LR(0) automaton

- For grammar

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
\mathrm{F} &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

SOLUTION

- Closure(E'→E)

  E'→.E

  E→.E+T

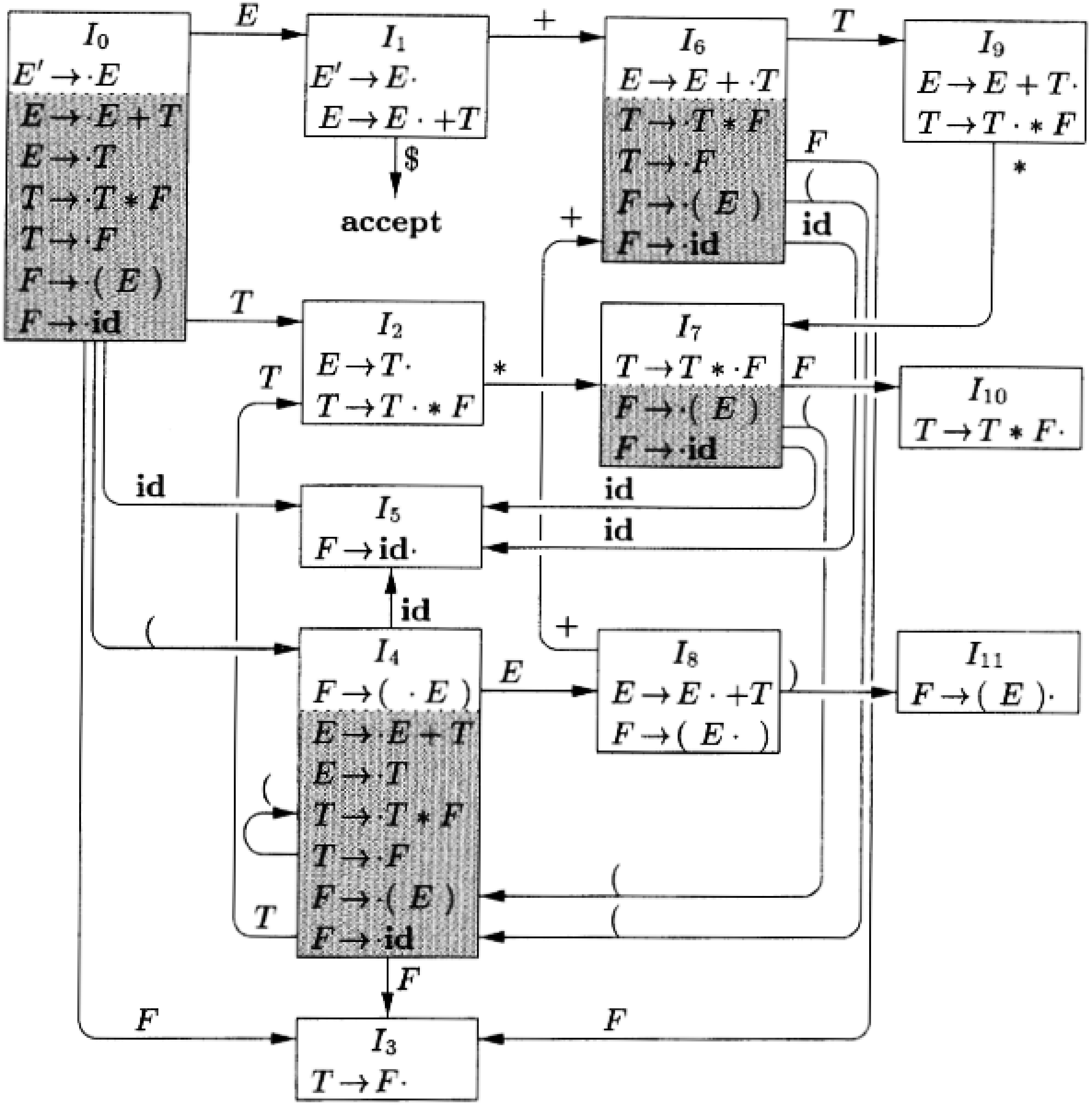
  E→.T

  T→.T*F

  T→.F

  F→.(E)

  F→.id

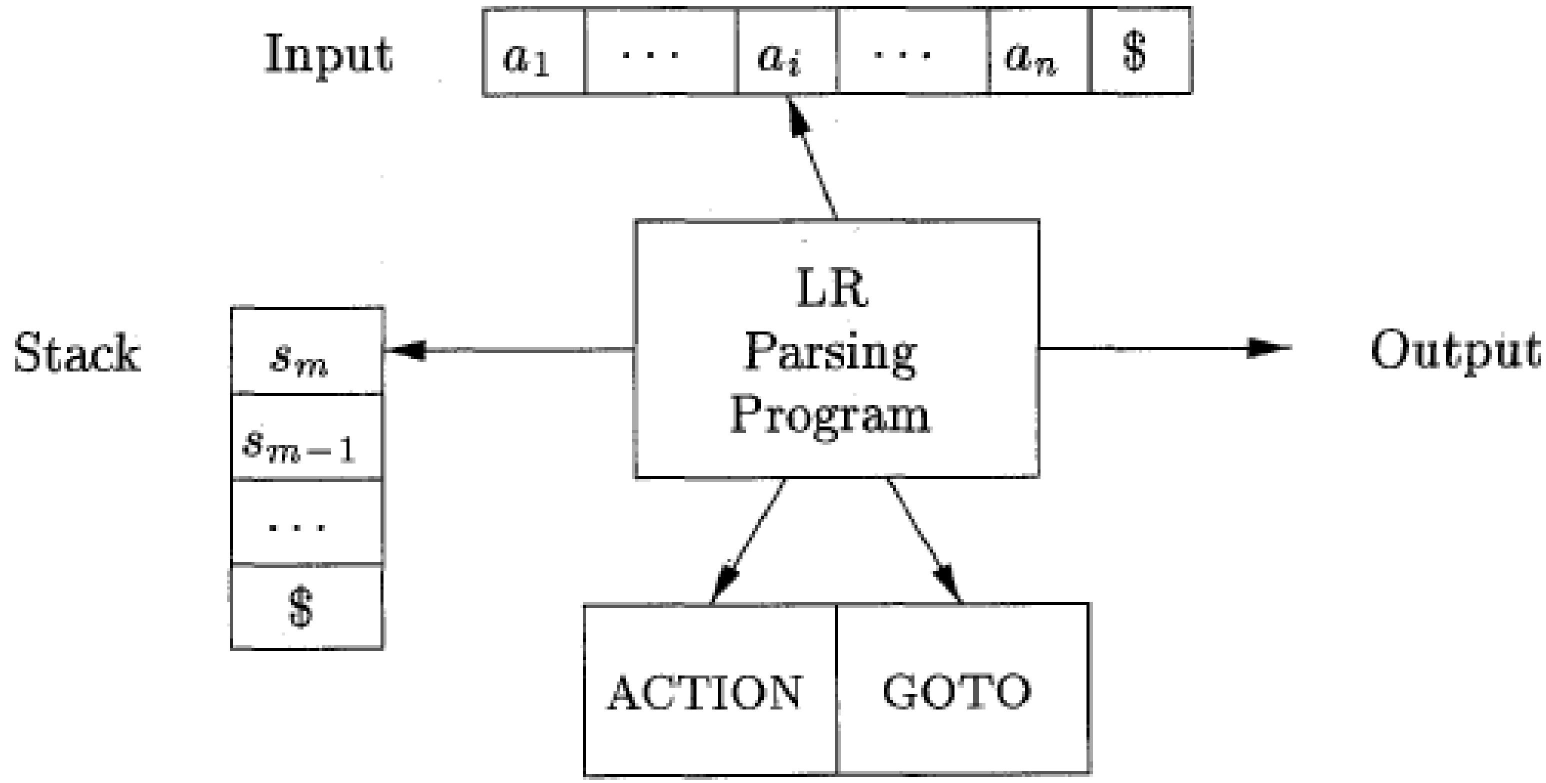


I0
E' → ·E
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E · +T
$
accept
I6
E → E + ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I9
E → E + T·
T → T · * F
I2
E → T·
T → T · * F
I7
T → T * ·F
F → ·(E)
F → ·id
I10
T → T * F·
I5
F → id·
I4
F → (· E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I8
E → E · +T
F → (E ·)
I11
F → (E)·
I3
T → F·
E
+
T
F
(
id
*

SLR(1)

- Simple LR parser
- Lookahead 1 - uses follow in construction of parse table
- Uses LR(0) items and DFA
- Parse table and parsing

LR parsing algorithm



- Stack maintains states rather than symbols.
- LR parser pushes states not symbols.

Construction of parse table for SLR(1)

- Write states of DFA as rows
- Has two parts – action and goto
- Under action, make columns for all terminals
- Under goto, make columns for all Non terminals
- For each state, refer DFA and fill table
  - Shift
  - Reduce
  - Accept
  - error
  - Goto entries

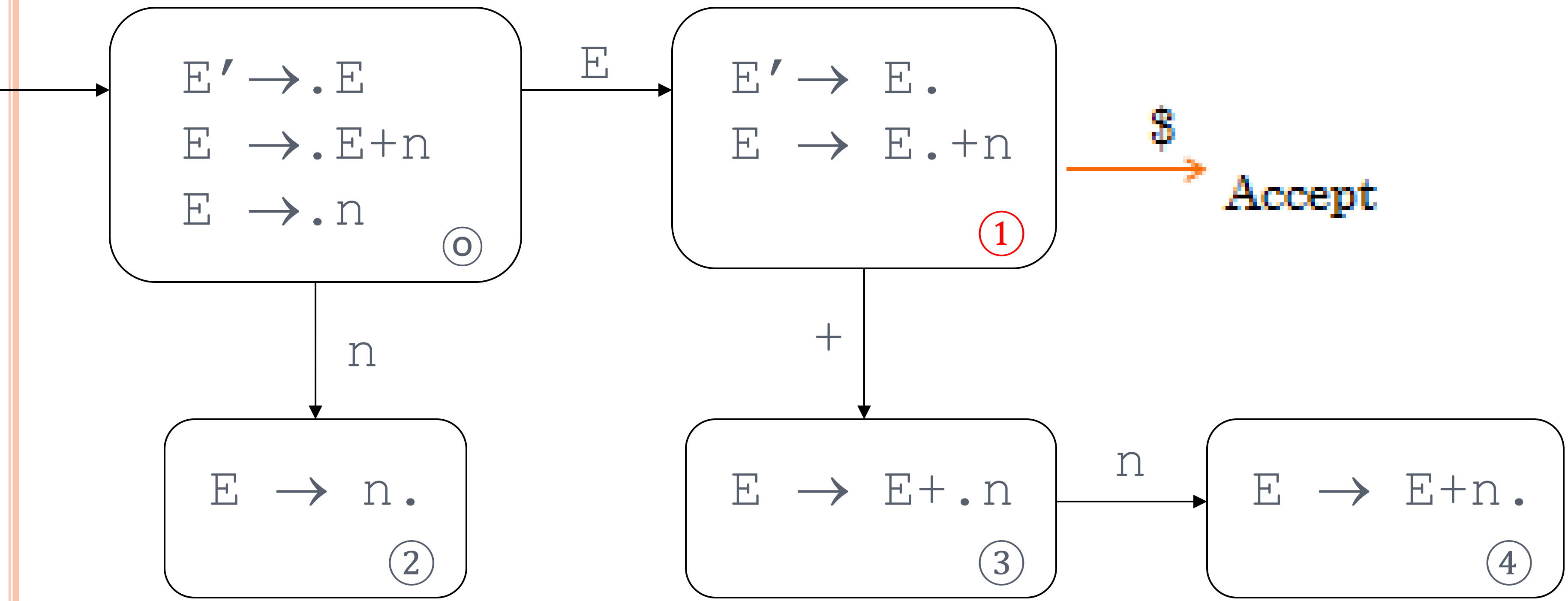
Basic function

LR PARSING

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /*
        else call error-recovery routine;
}
```

0) E1->E
1) E->E+n
2) E->n

EXAMPLE 2: DFA OF LR(0) ITEMS



SLR PARSE TABLE

0. E^1 -> E
1. E - > E+n
2. E - > n

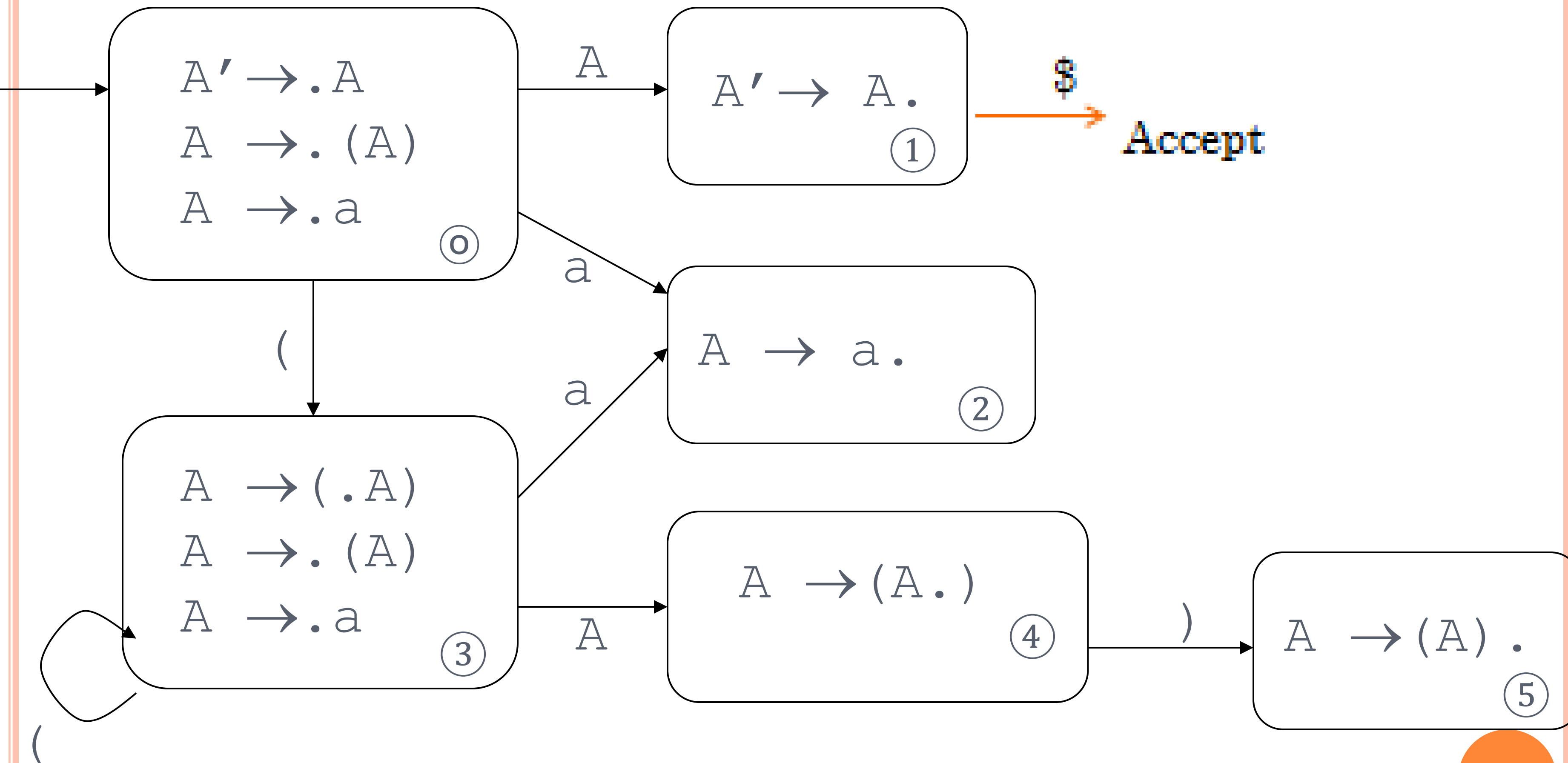
Follow(E) = {+,$}

State	ACTION			GOTO
	n	+	$	E
0	s2			1
1		s3	Accept	
2		r2	r2	
3	s4			
4		r1	r1	

Parsing action

Stack	symbols	input	action
\$0		n+n+n\$	

Example 3: DFA of LR(0) Items



Input: (a)

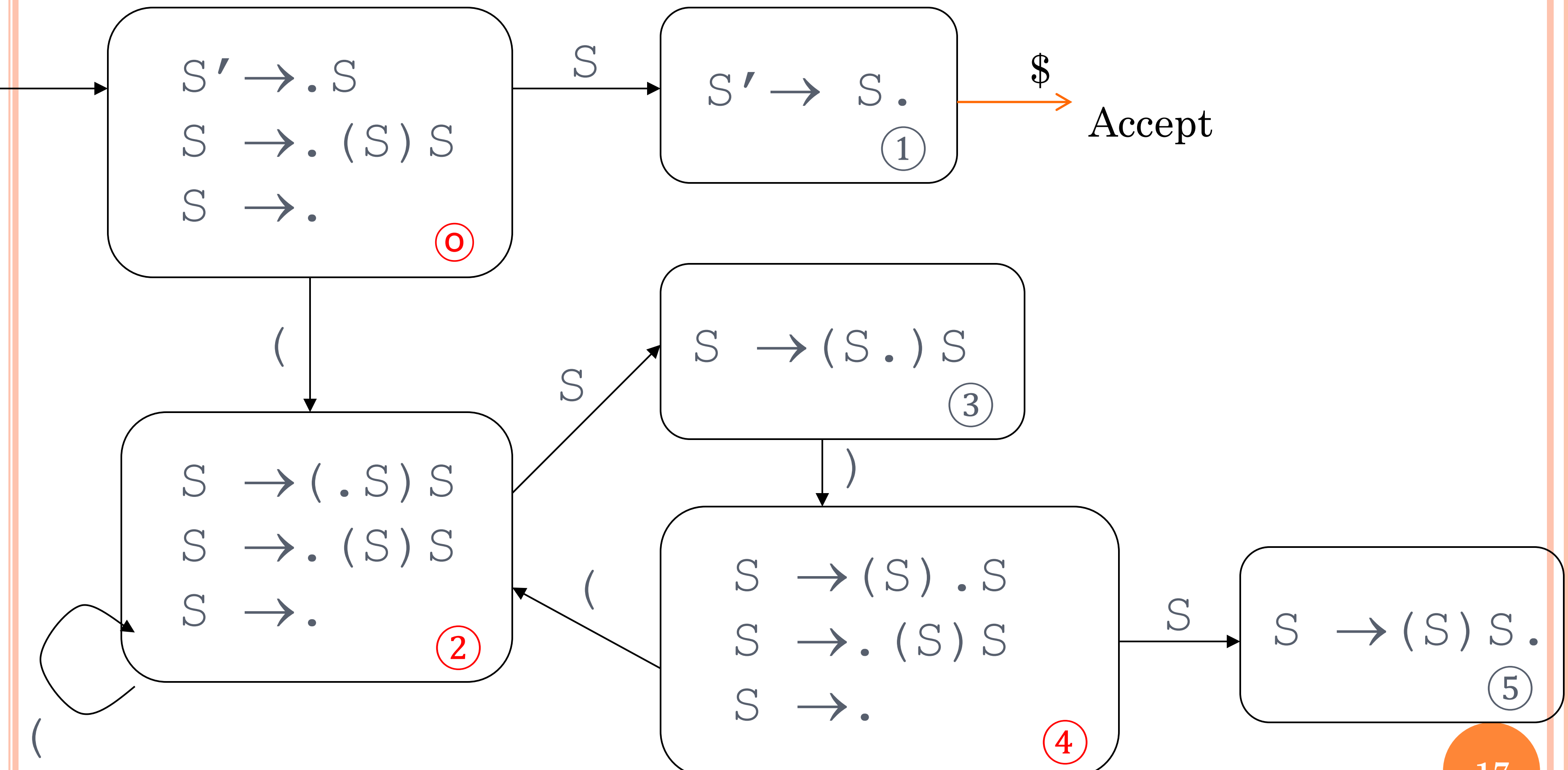
0. A’ → .A

1. A → .(A)

2. A → .a

State	ACTION				GOTO
	(	)	a	$	A
0	s3		s2		1
1				Accept	
2		r2		r2	
3	s3		s2		4
4		s5			
5		r1		r1	

EXAMPLE 1: DFA OF LR(0) ITEMS



0. S' →.S

1. S →.(S)S

2. S →.

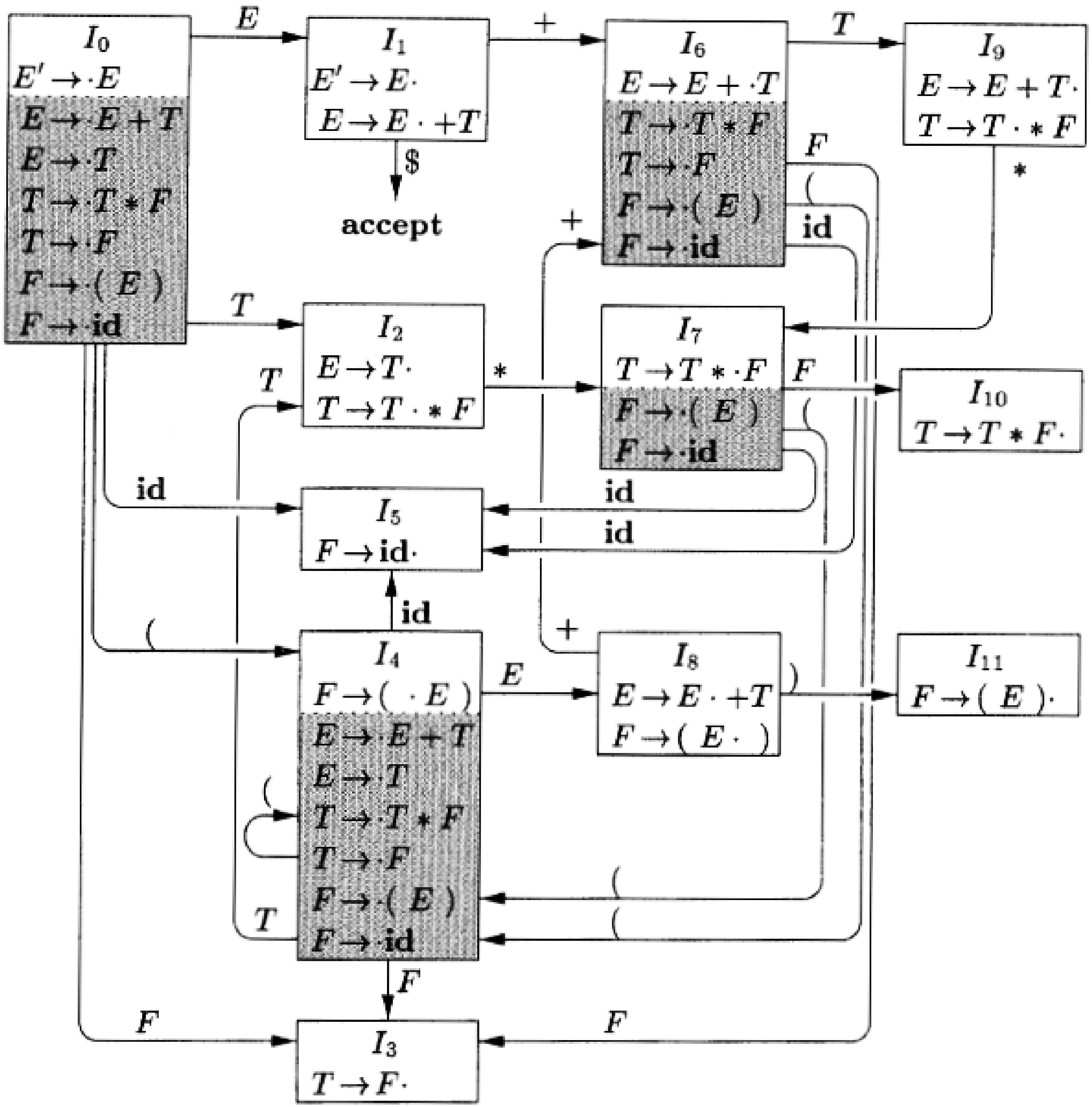
State	ACTION			GOTO
	(	)	$	S
0	s2	r2	r2	1
1			Accept	
2	s2	r2	r2	3
3		s4		
4	s2	r2	r2	5
5		r1	r1	

Input : ( )( )

Why to augment grammar

- To indicate parser when it should stop parsing and announce acceptance of the input.
- Single node
- Start symbol of the given grammar may have more than one definition.
- It may be difficult to judge whether whole string is parsed.
- May also be part of other production

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$


I0
E' → ·E
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E · +T
$
accept
I6
E → E + ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I9
E → E + T·
T → T · * F
I2
E → T·
T → T · * F
I7
T → T * ·F
F → ·(E)
F → ·id
I10
T → T * F·
I5
F → id·
I4
F → (·E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I8
E → E · +T
F → (E ·)
I11
F → (E)·
I3
T → F·
E
+
T
F
(
id
*
id
id
id
+
+
F
F
F

SLR PARSE TABLE CONTRUCTION

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4					
1		S6							
2			S7						
3									
4	S5			S4					
5									
6	S5			S4					
7	S5			S4					
8		S6			S11				
9			S7						
10									
11									

SLR parse table construction

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4					
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4					
5		r6	r6		r6	r6			
6	S5			S4					
7	S5			S4					
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

SLR parse table construction

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Input : id*id+id

Moves of LR parser

	Stack	Symbols	Input	Action
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

Ex4:

S-> Aa | bAc | dc | bda

A->d

Step 1: Augment Grammar

Step 2: Find start state of DFA

S^1 -> ●S

S-> ●Aa

S-> ●bAc

S-> ●dc

S-> ●bda

A-> ●d

Step 3: Draw DFA

Step 4: construct Parse table

Step 5: Show parsing action

Ex 5:

S - > L=R | R

L -> *R | id

R ->L

Ex 6:

S-> a | ↑ | (T)

T-> T, S | S

Limitations of SLR(1)

- Applies lookaheads after the construction of the DFA of LR(0) items
- **The construction of DFA ignores lookaheads**

- The general LR(1) method:
  - **Using a new DFA with the lookaheads built into its construction**

    The DFA items are an extension of LR(0) items

    LR(1) items include a single lookahead token in each item.
  - A pair consisting of an LR(0) item and a lookahead token.

    LR(1) items using square brackets as $[A \rightarrow \alpha \cdot \beta, a]$

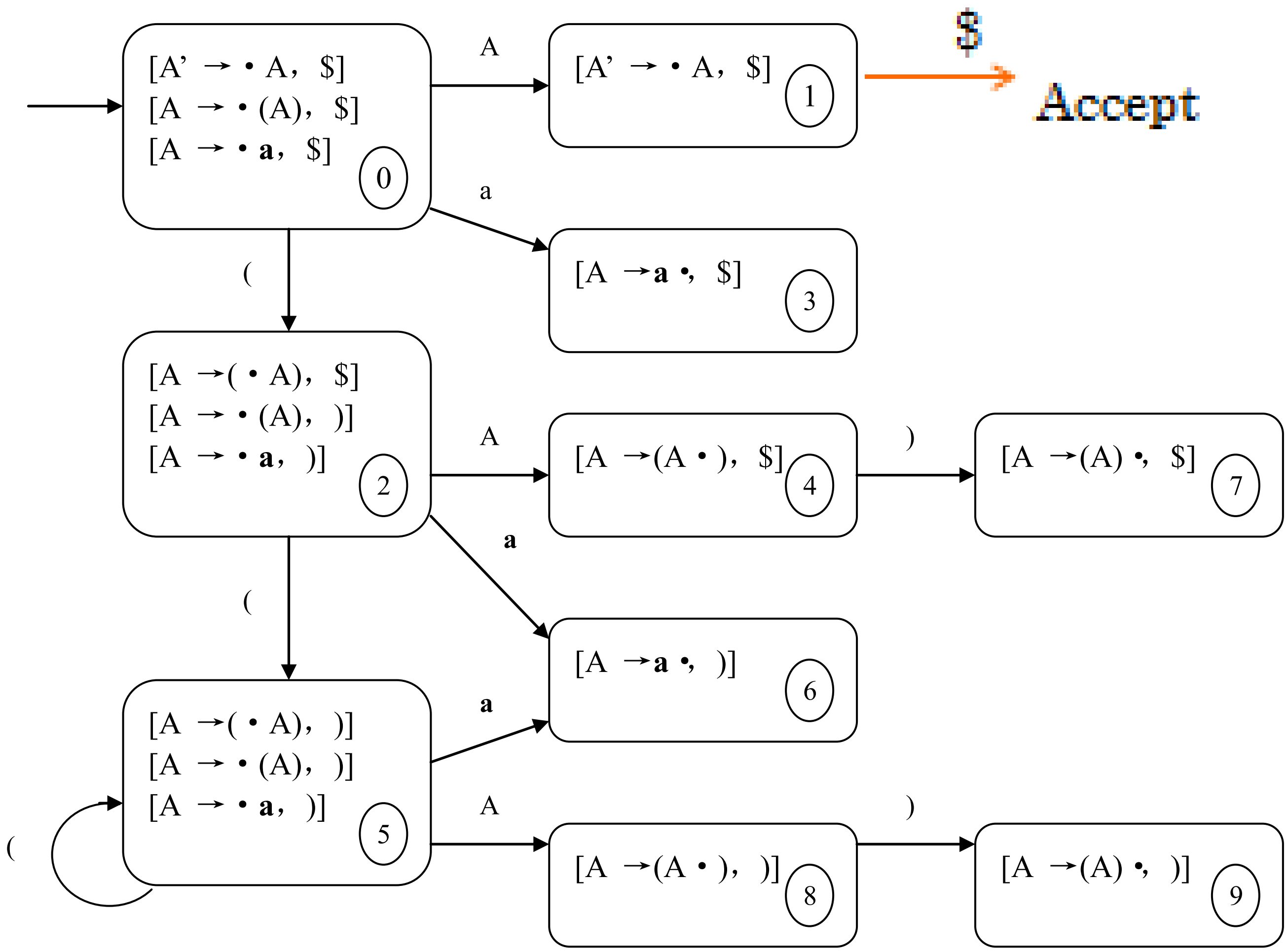
    where $A \rightarrow \alpha \cdot \beta$ is an LR(0) item and a is a lookahead token

CLR(1)

The Grammar:

(l) A→(A)

(2) A→**a**



The Grammar:

(l) A→(A)

(2) A→**a**

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		