Bottom Up parsing

Handle Pruning

- Handle is the substring which matches right side of the production and we can reduce such string by a non terminal on the LHS of the production.
- Reduction of a string or handle by a suitable Non terminal is called pruning.

Conflicts in shift reduce parser

- Shift reduce conflict
- Reduce reduce conflict

Shift reduce Conflict

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

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*	$\rightarrow$	**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
$\cdots$ **procid** (**id**	, **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 $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	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→ XYZ

Items are

A→ •XYZ

A→ X•YZ

A→ XY•Z

A→ XYZ•

- A→X • 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 $A \rightarrow \in$. 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
    - $S' \rightarrow .S$
    - $S' \rightarrow S.$
    - $S \rightarrow .(S) S$
    - $S \rightarrow (. S) S$
    - $S \rightarrow (S .) S$
    - $S \rightarrow (S) . S$
    - $S \rightarrow (S) S.$
    - $S \rightarrow .$

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

- Ex 3: $E' \rightarrow E$

  $E \rightarrow E + n \mid n$

  - The grammar has 3 production choices.
  - The grammar has 8 items.
    - $E' \rightarrow .E$ $\quad$ $E' \rightarrow E.$
    - $E \rightarrow .\ E + n$ $\quad$ $E \rightarrow E\ .\ + n$
    - $E \rightarrow E +\ .\ n$ $\quad$ $E \rightarrow E + n\ .$
    - $E \rightarrow .n$ $\quad$ $E \rightarrow n\ .$

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

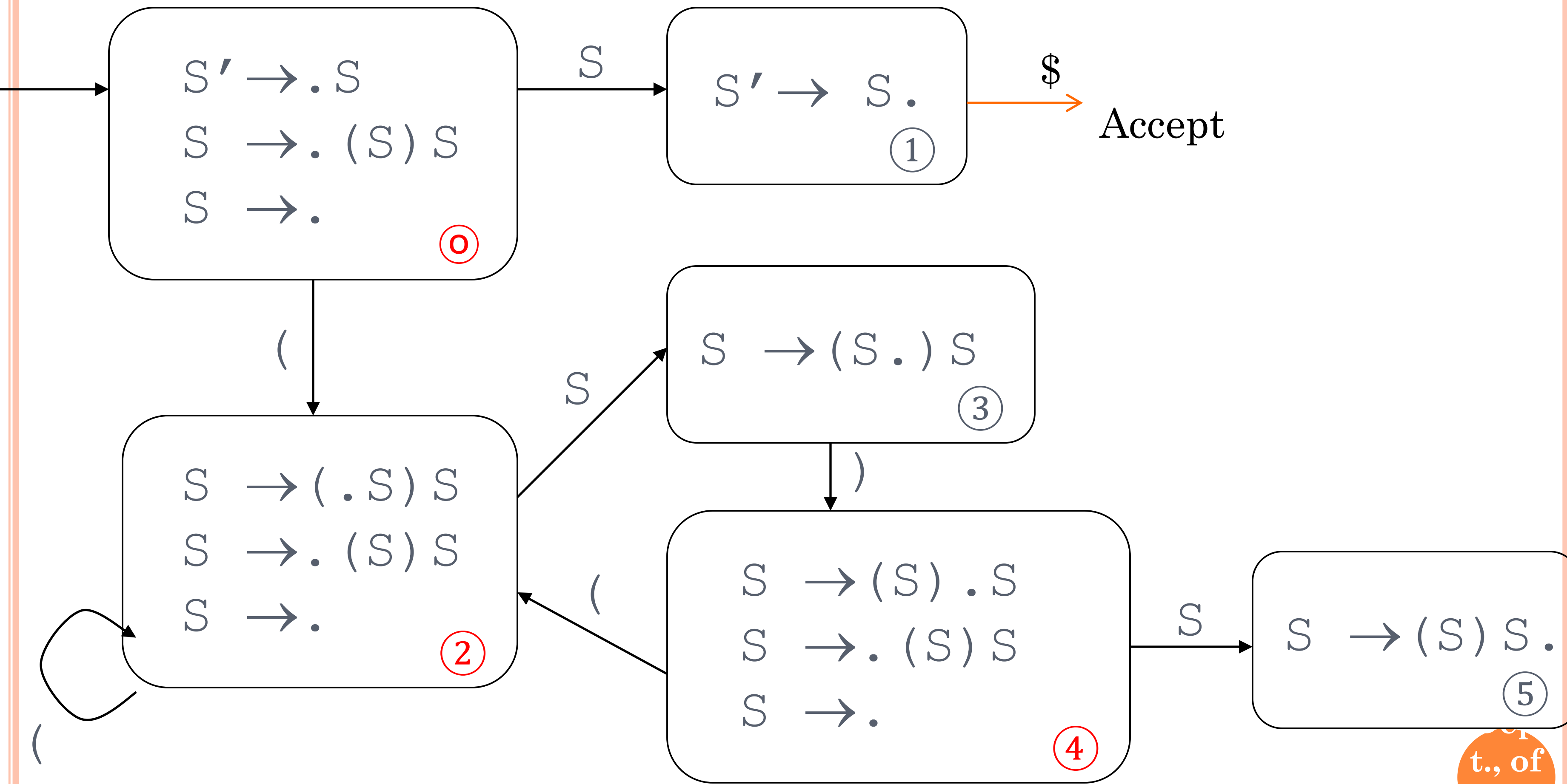
- goto.pdf

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

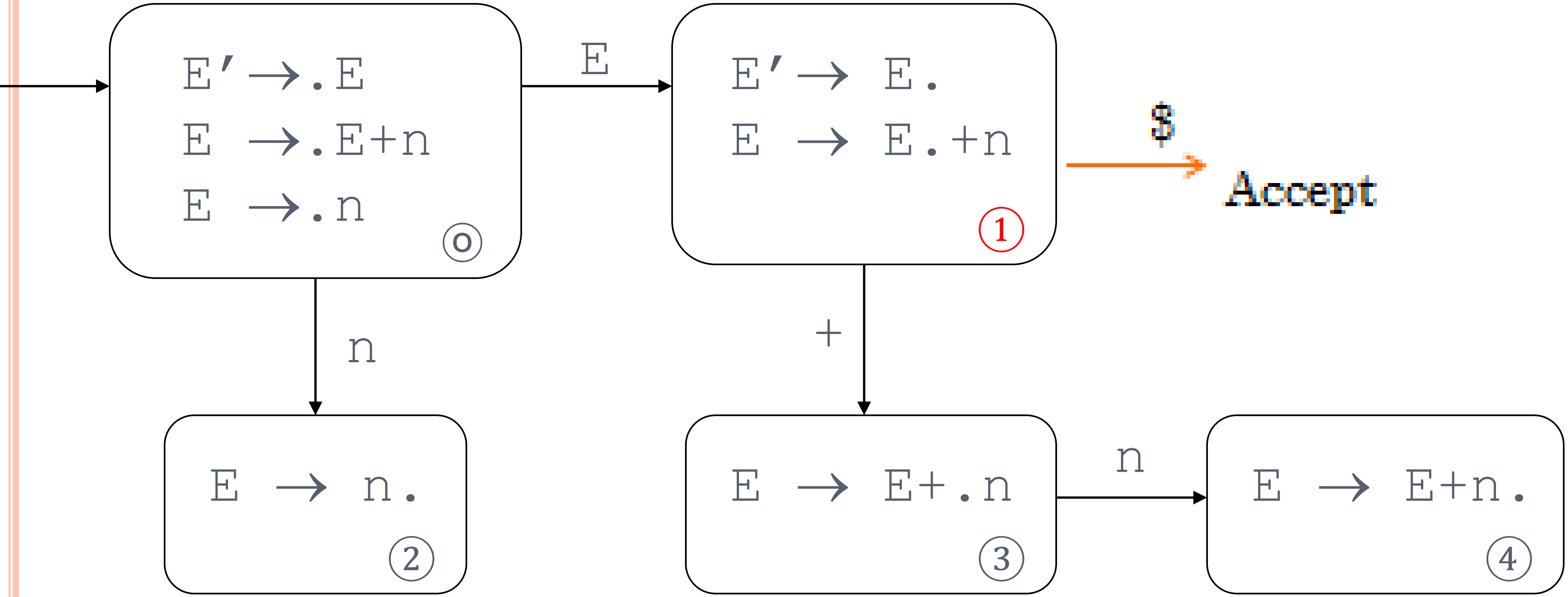
X – grammar symbol

Example 1: DFA of LR(0) Items

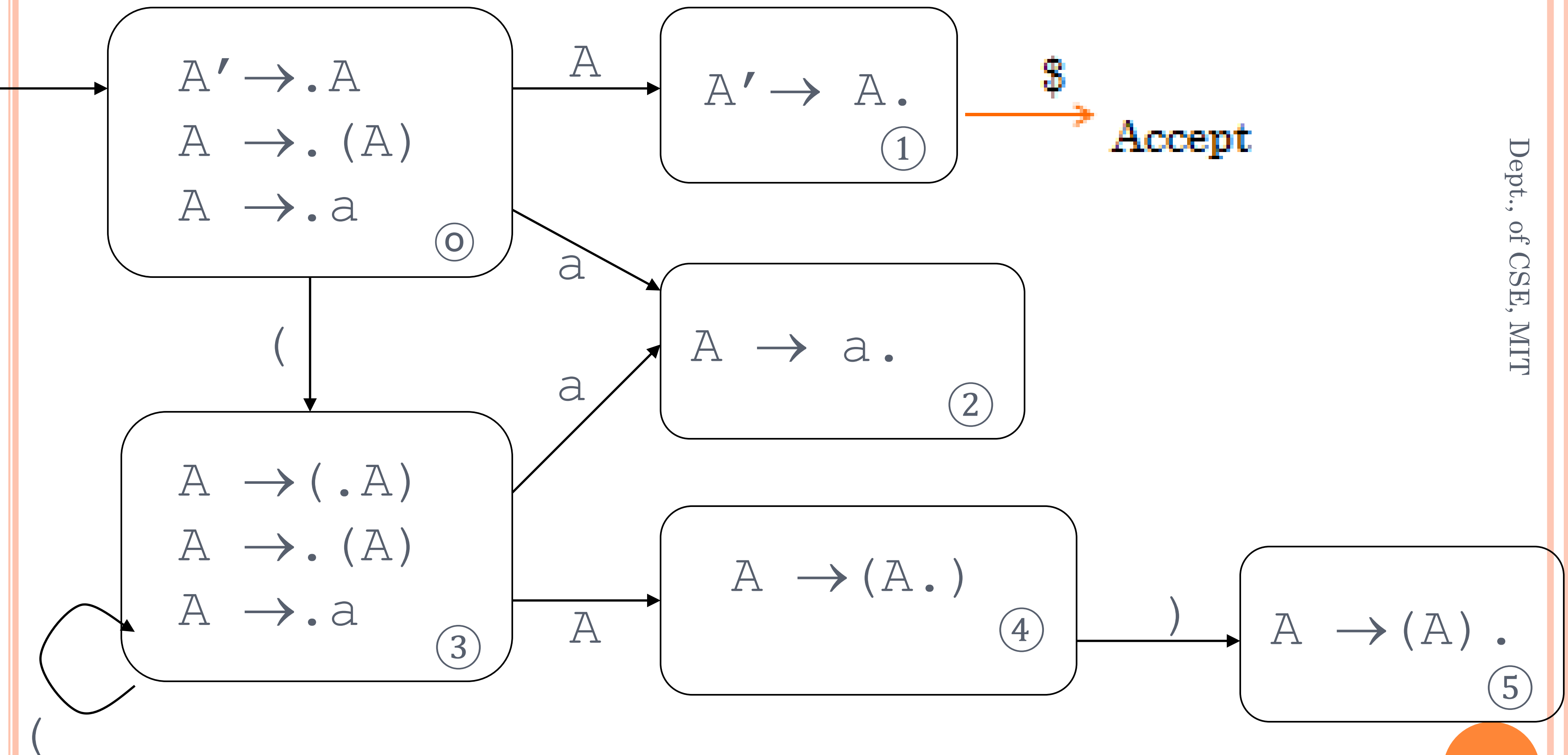


t., of CSE.

EXAMPLE 2: DFA OF LR(0) ITEMS



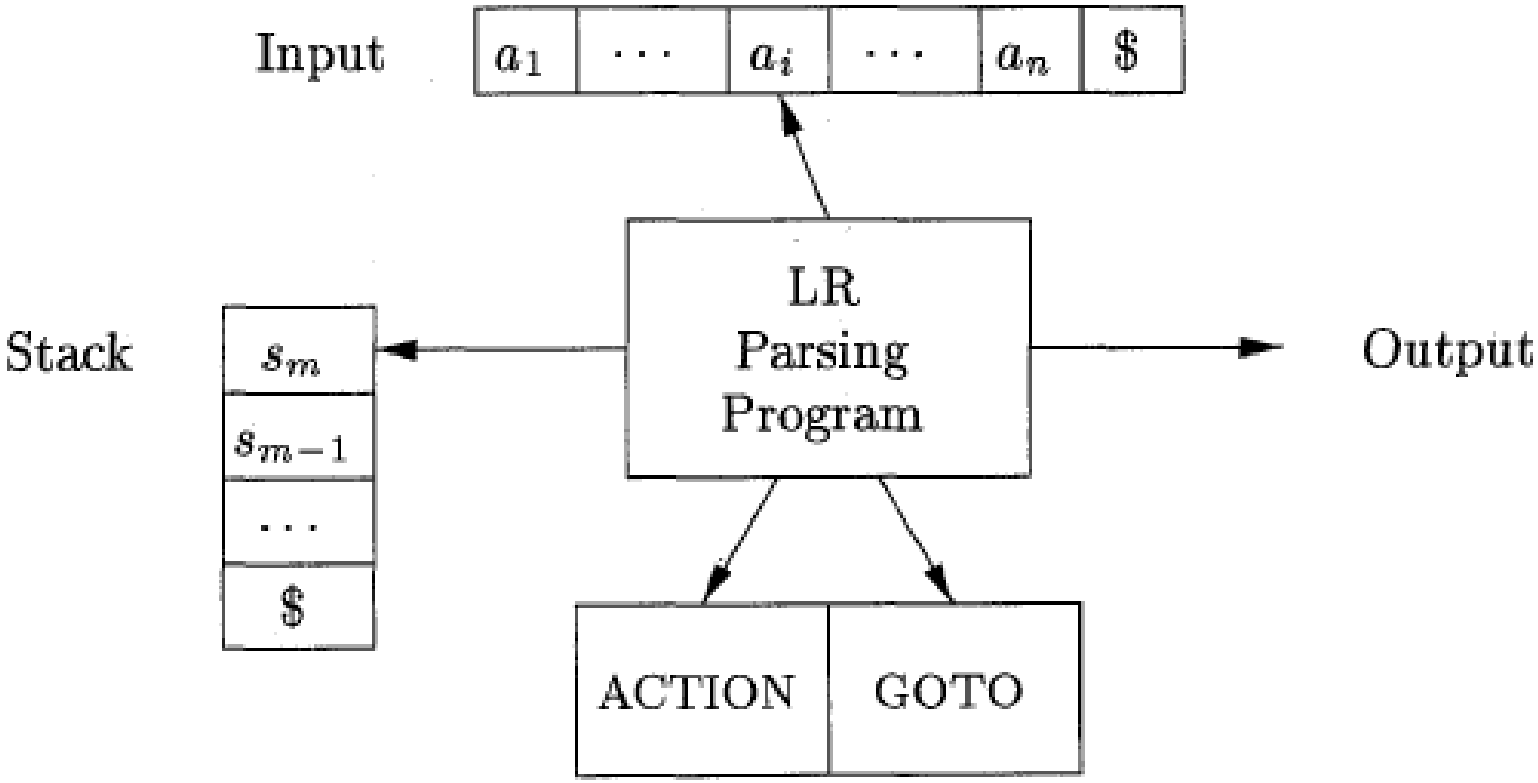
Example 3: DFA of LR(0) Items



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

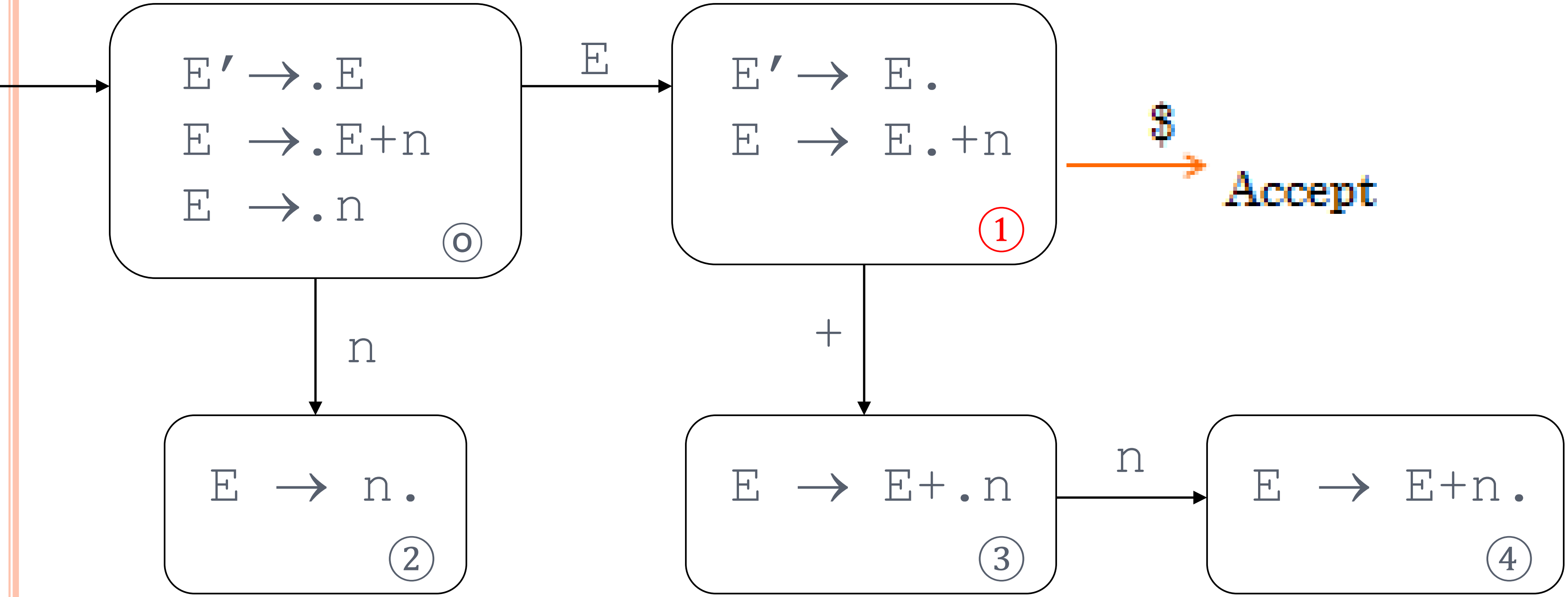
Basic function

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;
}
```

Example 2: DFA of LR(0) Items

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



SLR PARSE TABLE

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

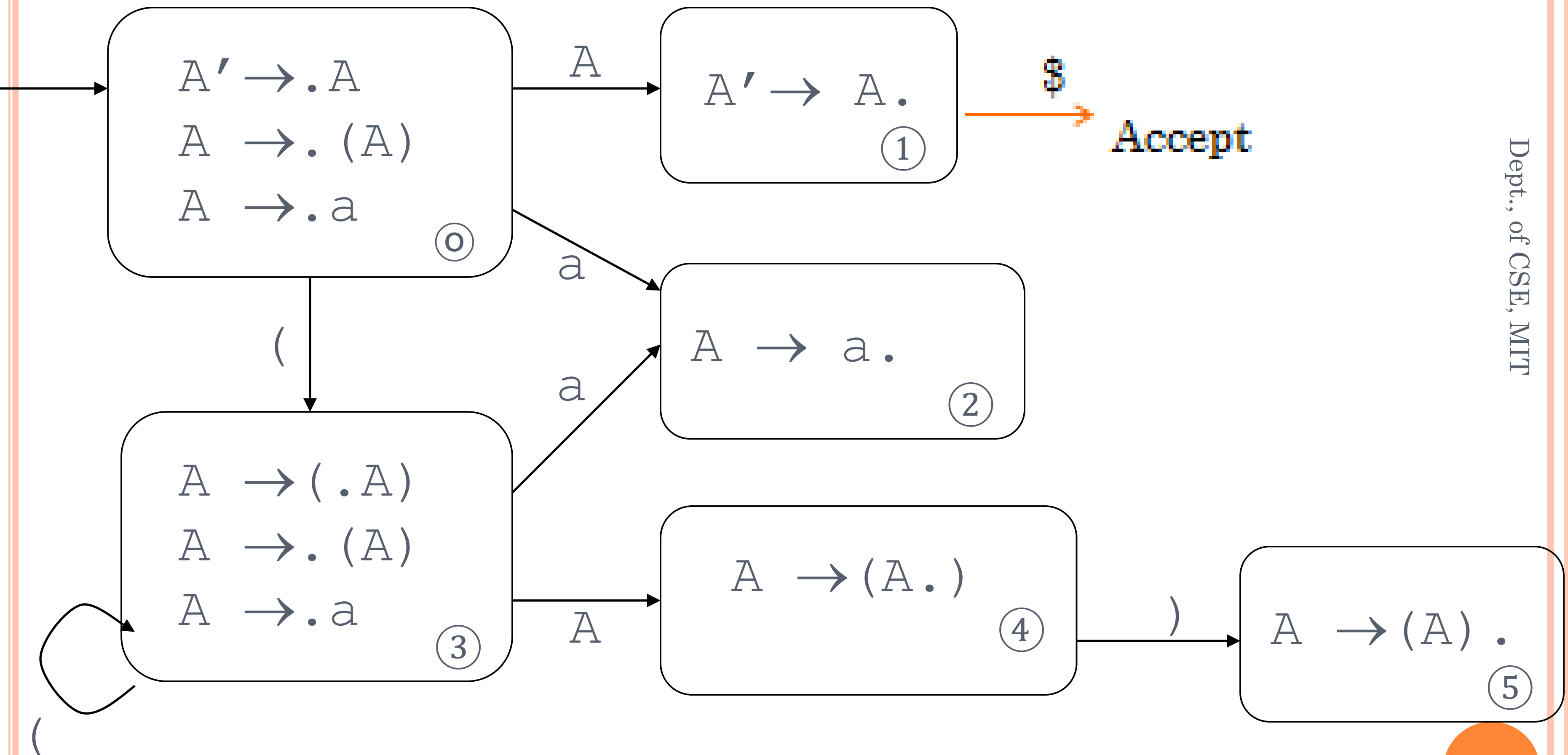
Follow(E) = {+,$}

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
$0		n+n+n$	Shift
$02	n	+n+n$	Reduce E→n
$01	E	+n+n$	Shift
$013	E+	n+n$	Shift
$0134	E+n	+n$	Reduce E→E+n
$01	E	+n$	Shift
$013	E+	n$	Shift
$0134	E+n	$	Reduce E→E+n
$01	E	$	Accept

Example 3: DFA of LR(0) Items



Input: (a)

Dept., of CSE, MIT

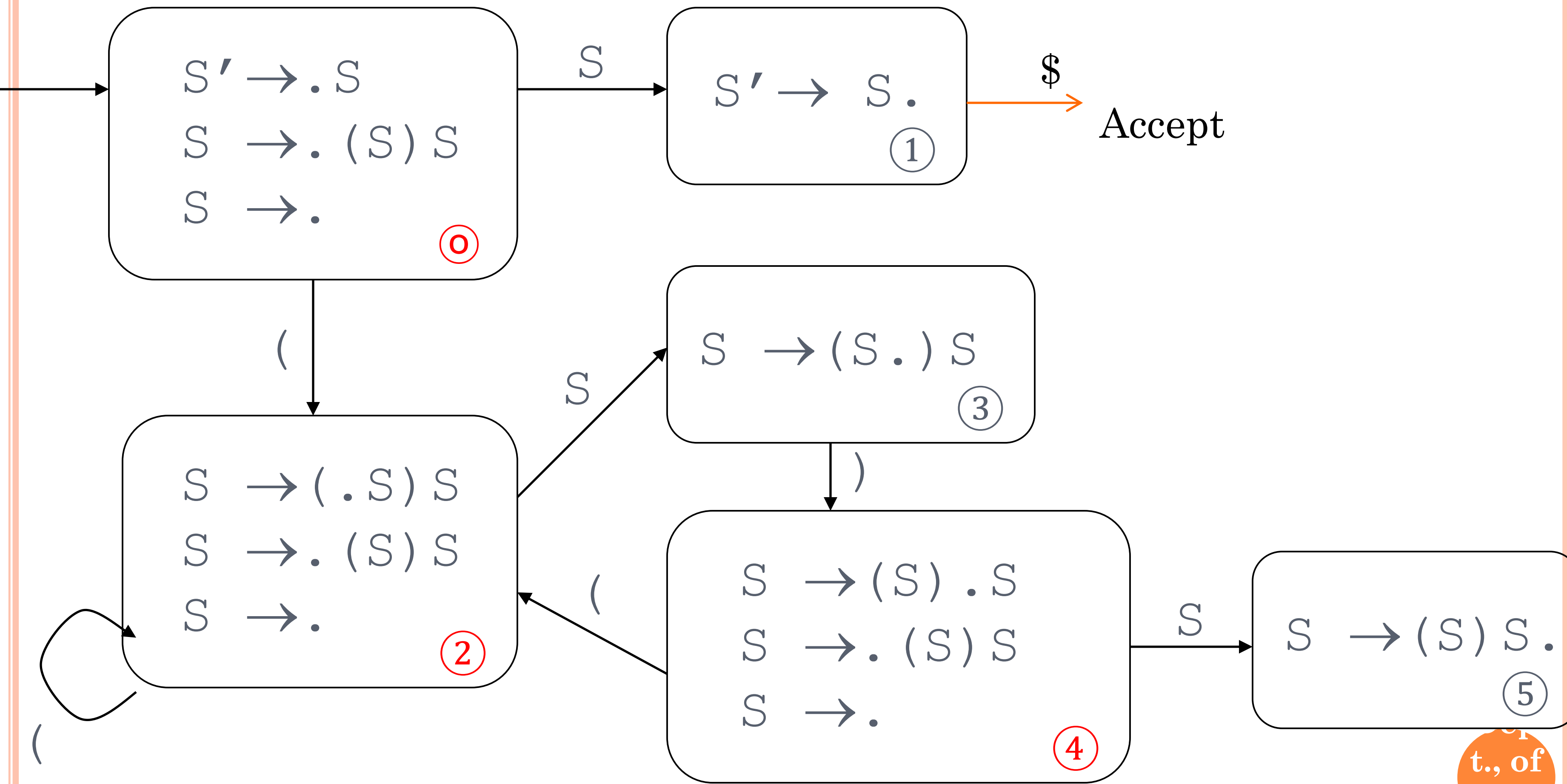
0. A' → .A

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

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

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

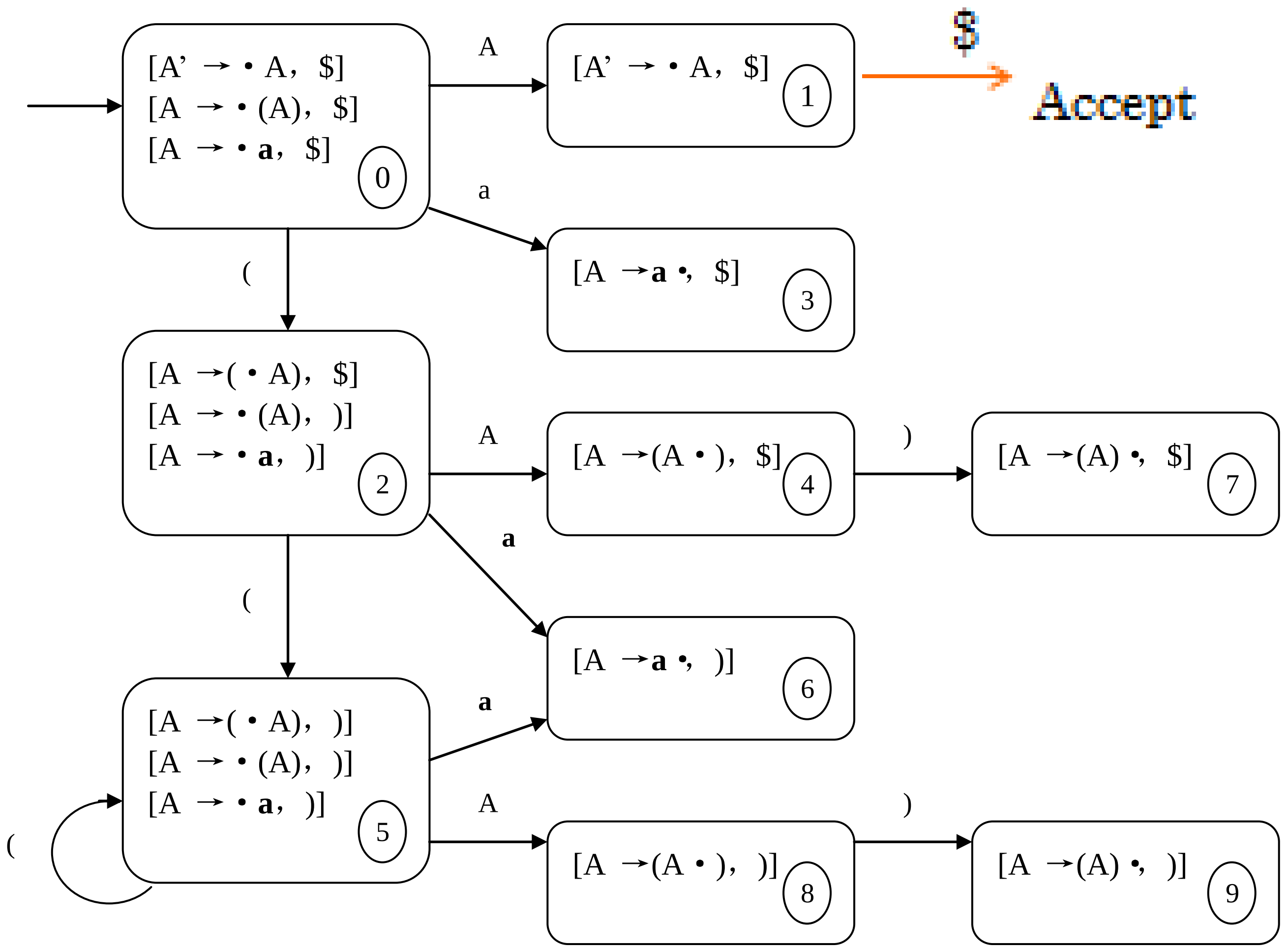
    where $A \rightarrow \alpha \cdot \beta$ is an LR(0) item and a is a lookahead token

CLR(1)

The Grammar:

(1) A→(A)

(2) A→**a**



Dept., of CSE, MIT

The Grammar:

(l) A→(A)

(2) A→**a**

State	Input				Goto
	(	a	)	$	A
0	s3	s2			1
1				accept	
3	s6	s5			4
2				r2	
4			s9		
6	S6	S5			7
5			r2		
9				r1	
7			s8		
8			r1		