Bottom Up parsing

Prepared by
Deepthi S

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

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

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*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

STACK	INPUT
··· **id** (**id**	, **id**) ···

- 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*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

STACK	INPUT
··· **procid (id**	**, id)** ···

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→XYZ •

  time to reduce XYZ to A.
- So, there is a prodn A→ ∈. what is the item?

  A→ •

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

- Ex 2: S′ → S
  S → (S) S | ε
  - The grammar has 3 production choices.
  - The grammar has 8 items
    - S′ → .S
    - S → .(S) S
    - S → (S .) S
    - S → (S) S.
    - S′ → S.
    - S → (. S) S
    - S → (S) . S
    - S → .

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 + n
    - E → E + . n
    - E → .n
    - E′ → E.
    - E → E . + n
    - E → 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

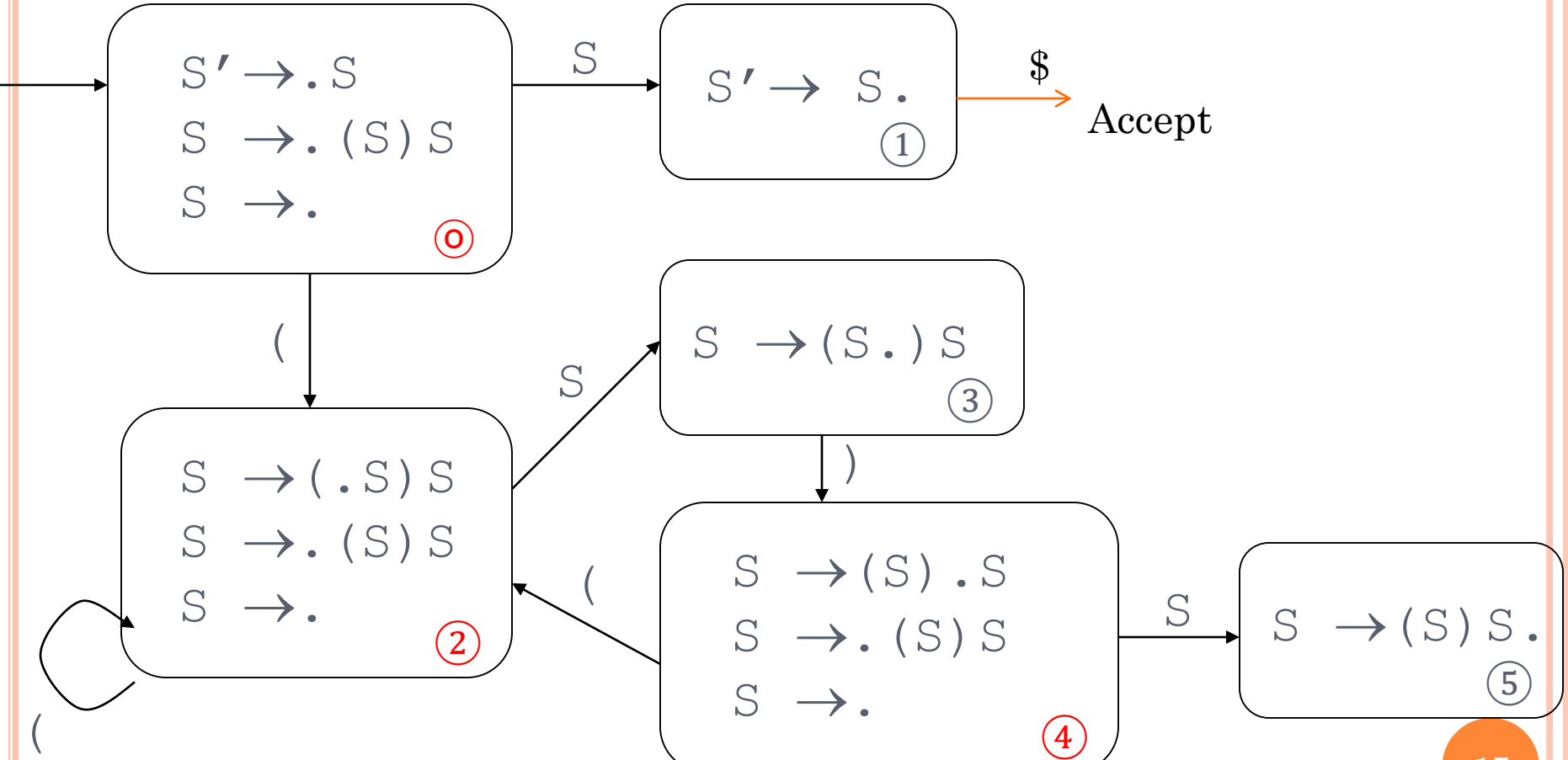
- goto.pdf

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.

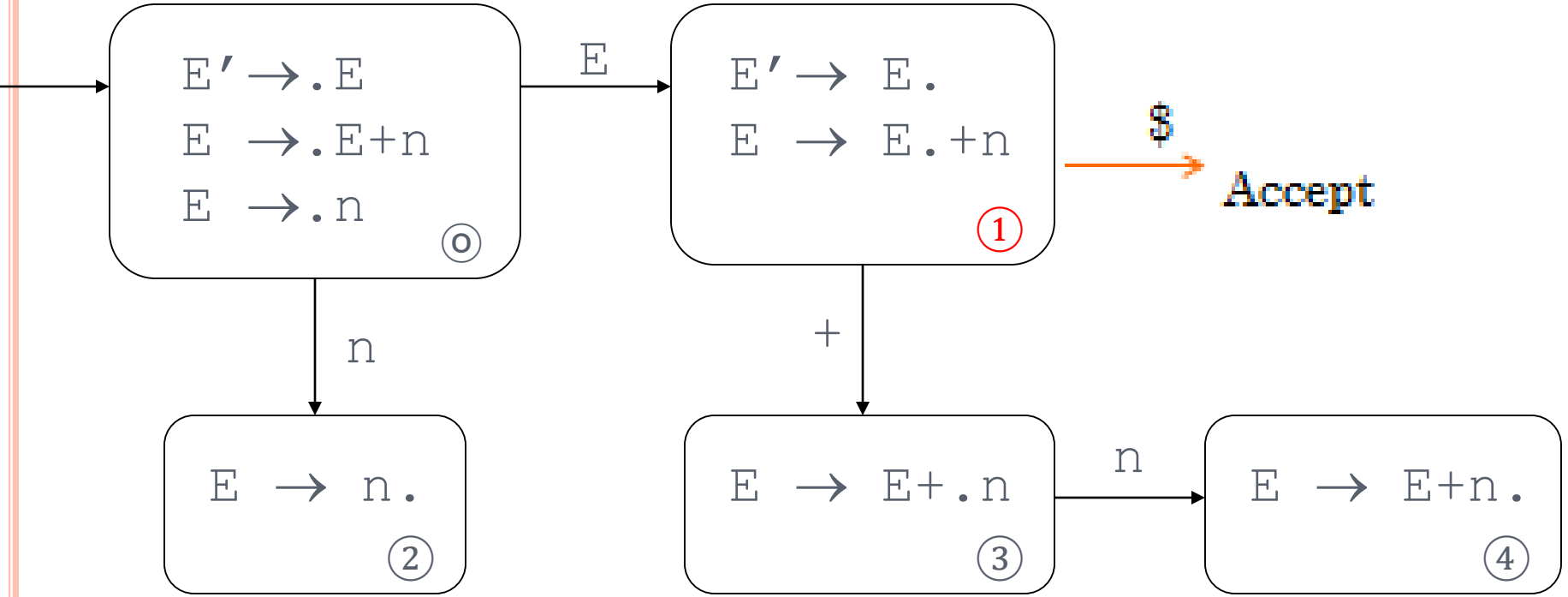
I – set of items

X – grammar symbol

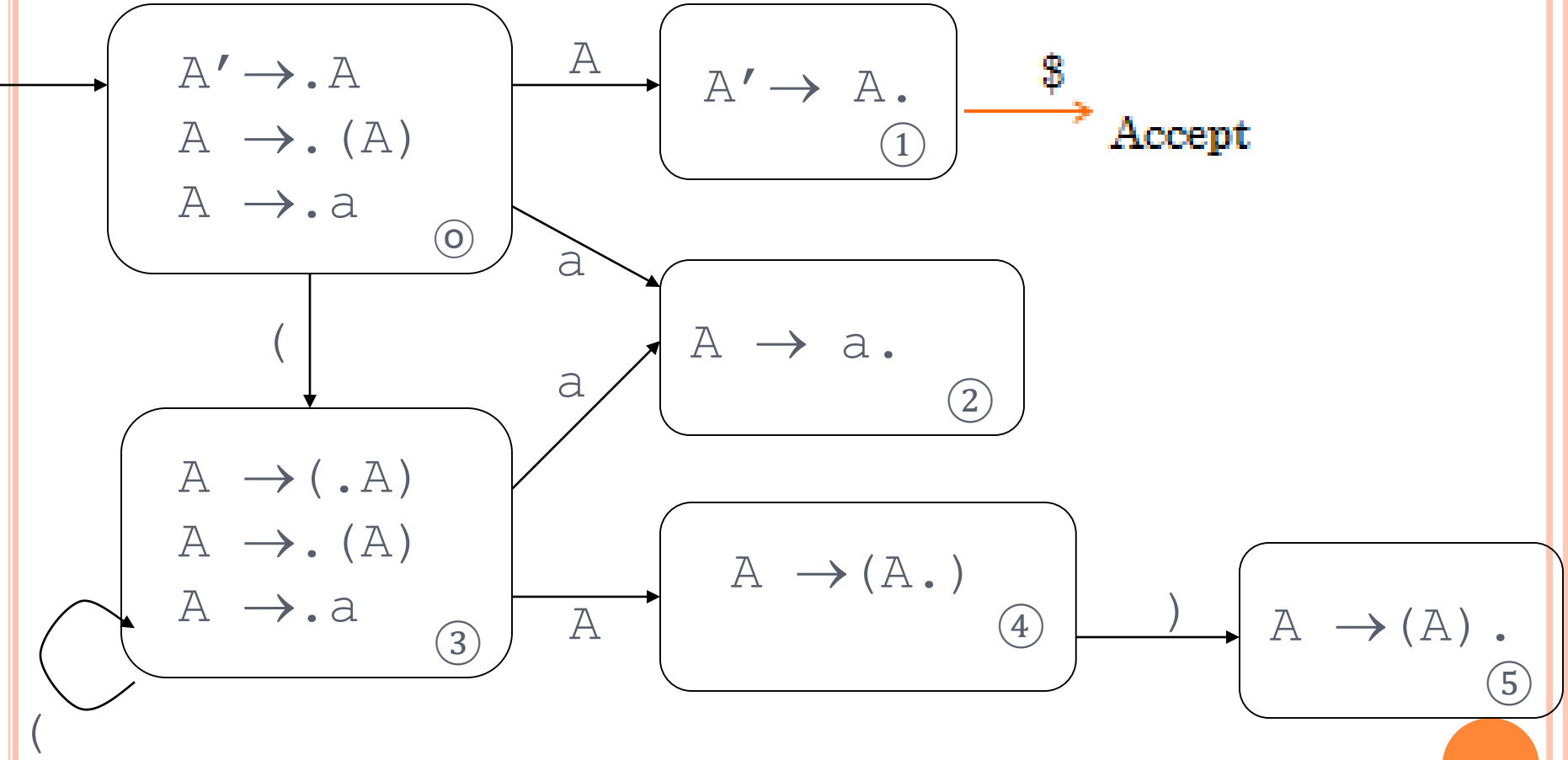
Example 1: DFA of LR(0) Items



Example 2: DFA of LR(0) Items



Example 3: DFA of LR(0) Items



Construct LR(0) automaton

- For grammar

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

SOLUTION

- Closure(E'→E)

  E'→.E

  E→.E+T

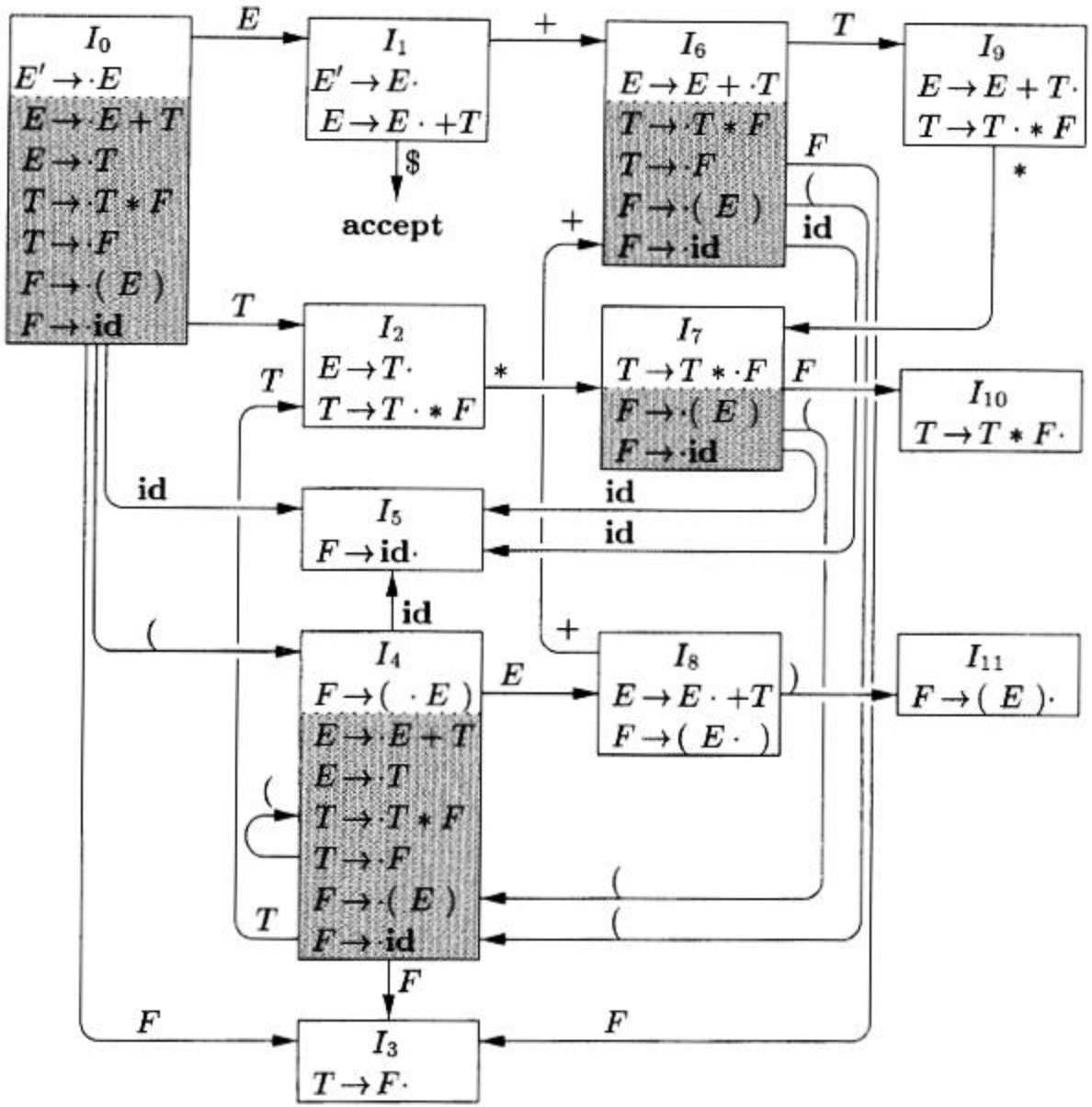
  E→.T

  T→.T*F

  T→.F

  F→.(E)

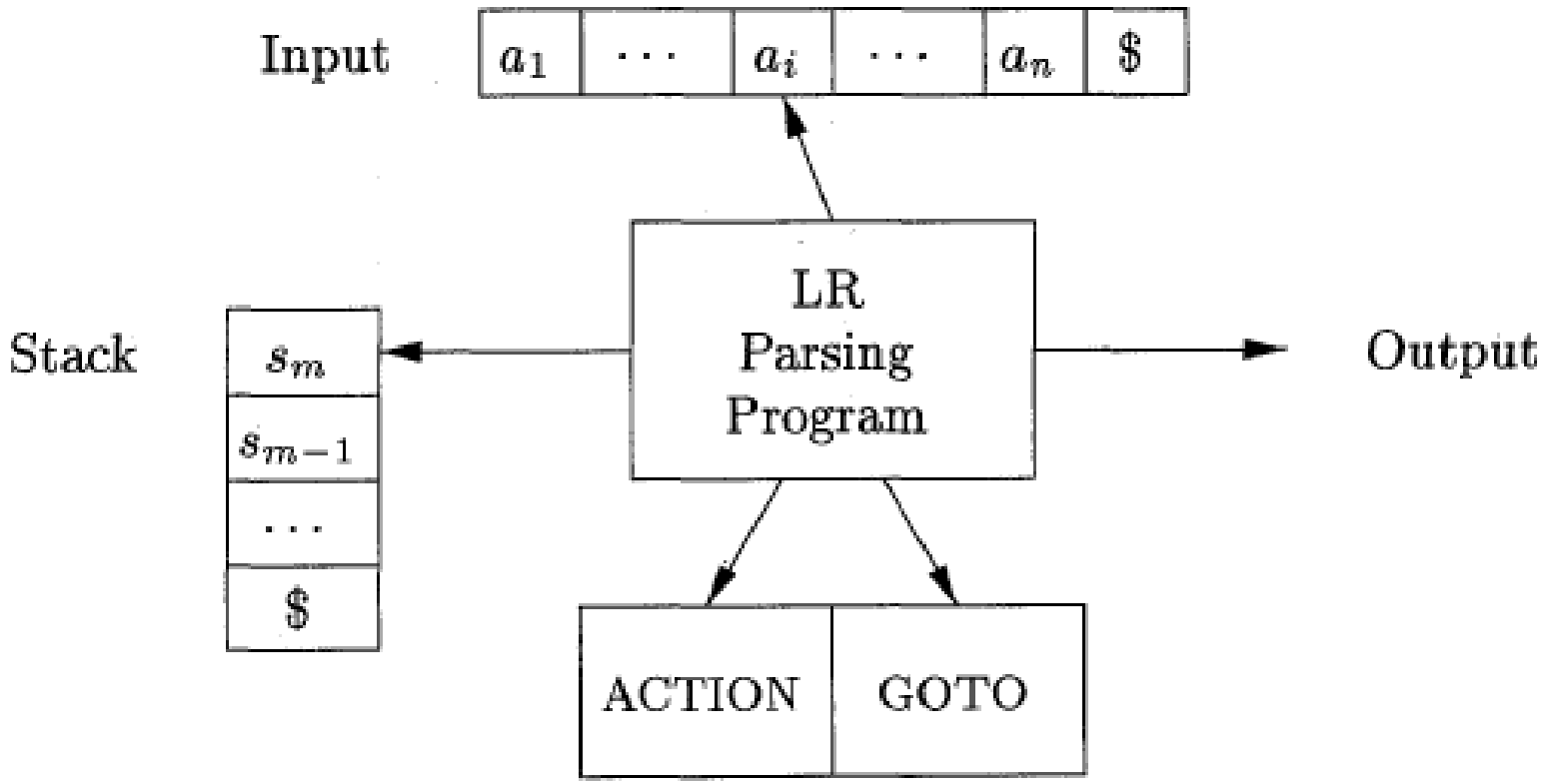
  F→.id


I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$
accept
E
T
F
$($
$\mathbf{id}$
$+$
$*$
$)$
$\$$

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

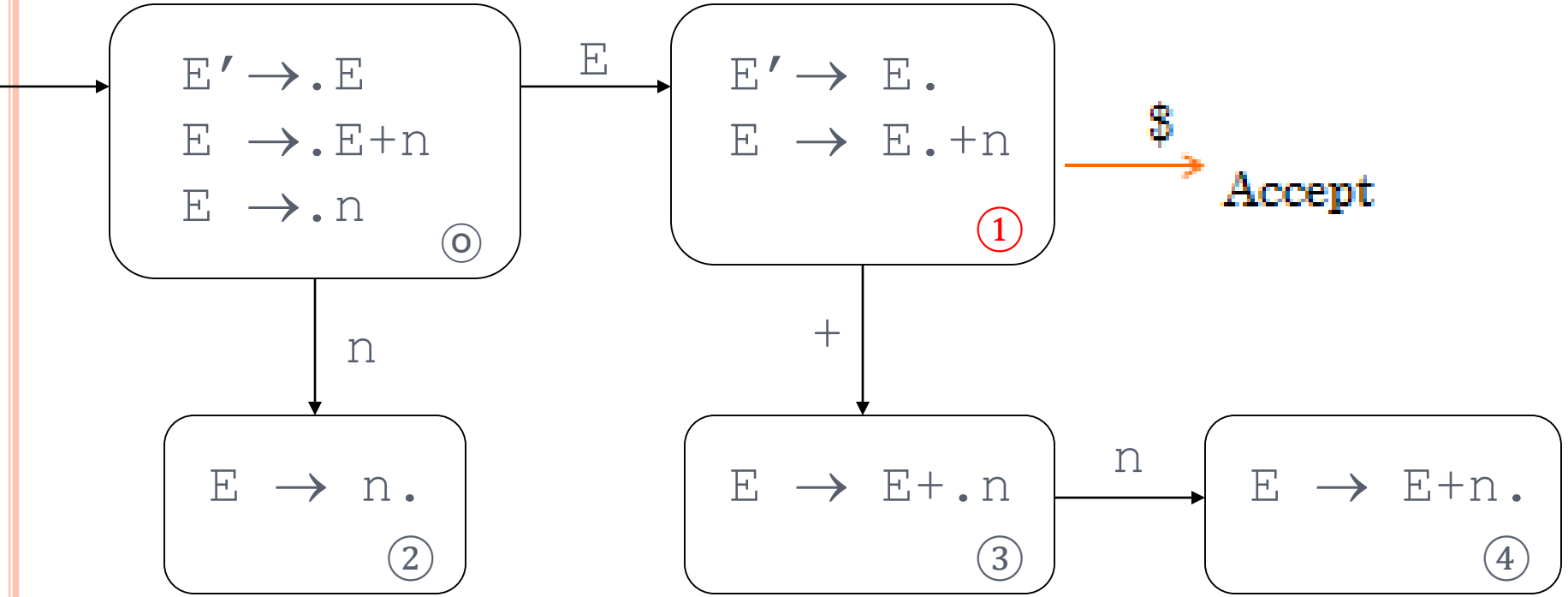
Basic function

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;
}
```

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

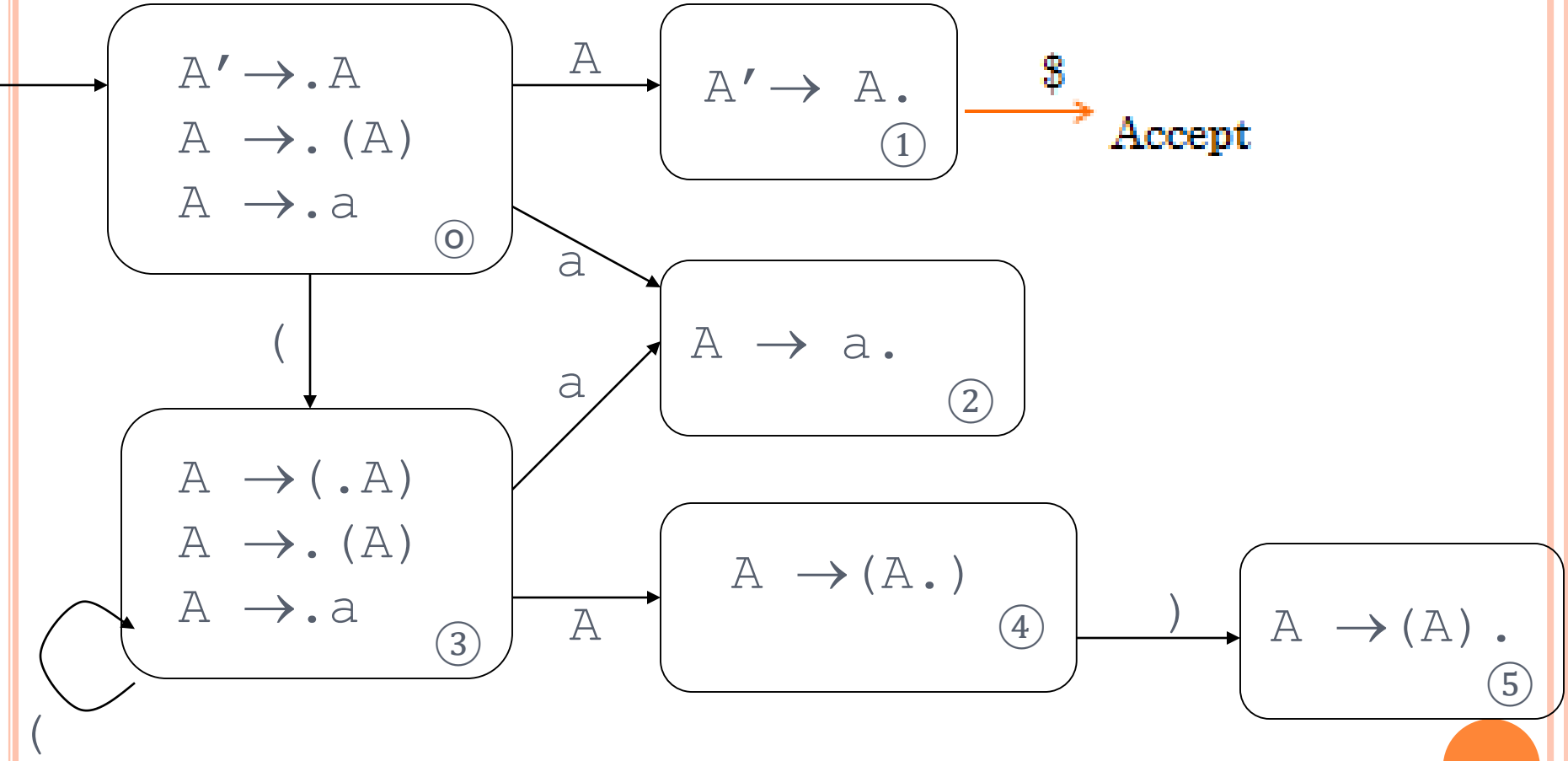
Follow(E) = {+,$}

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)

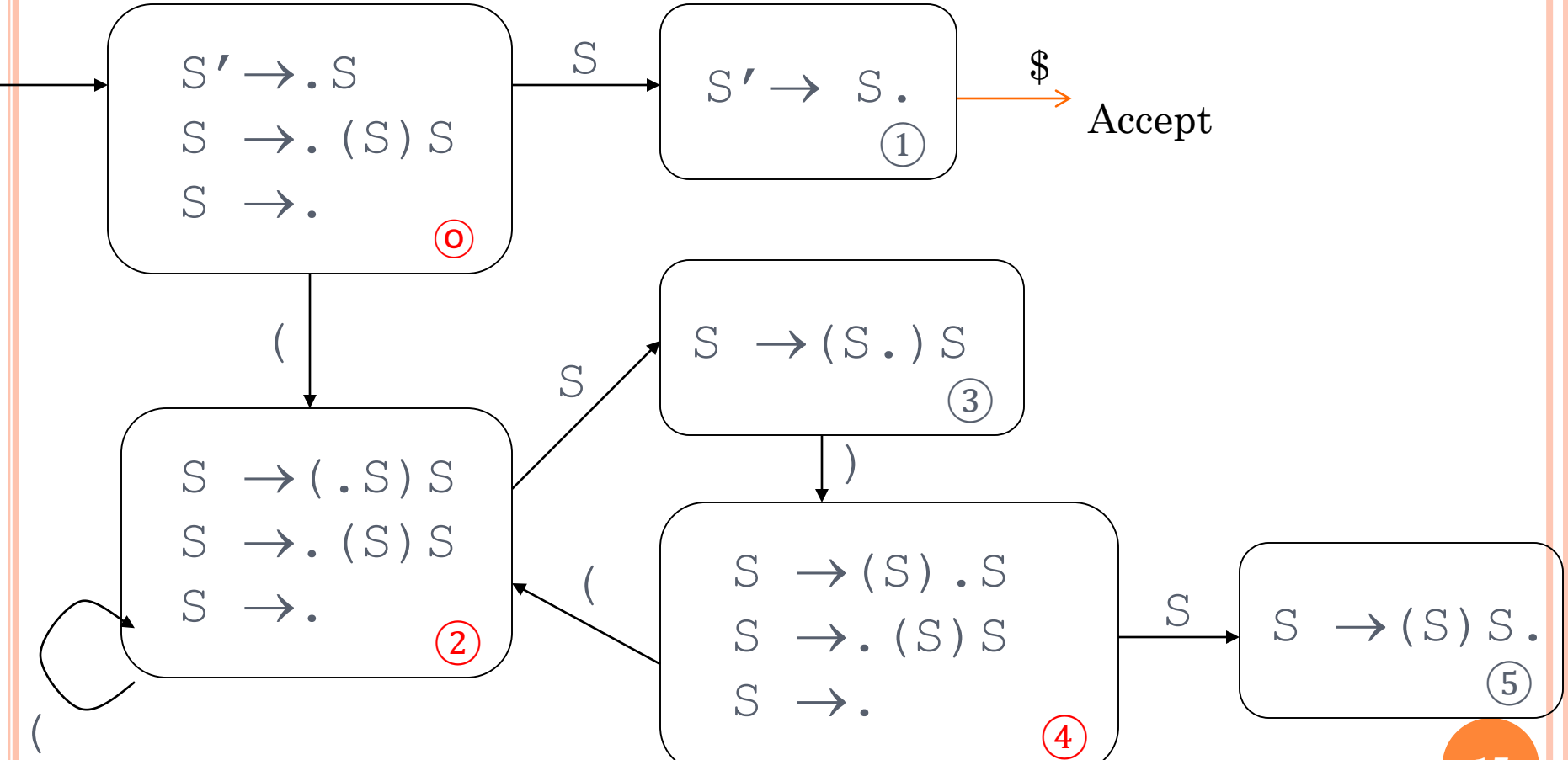
0.A′→A

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 →€

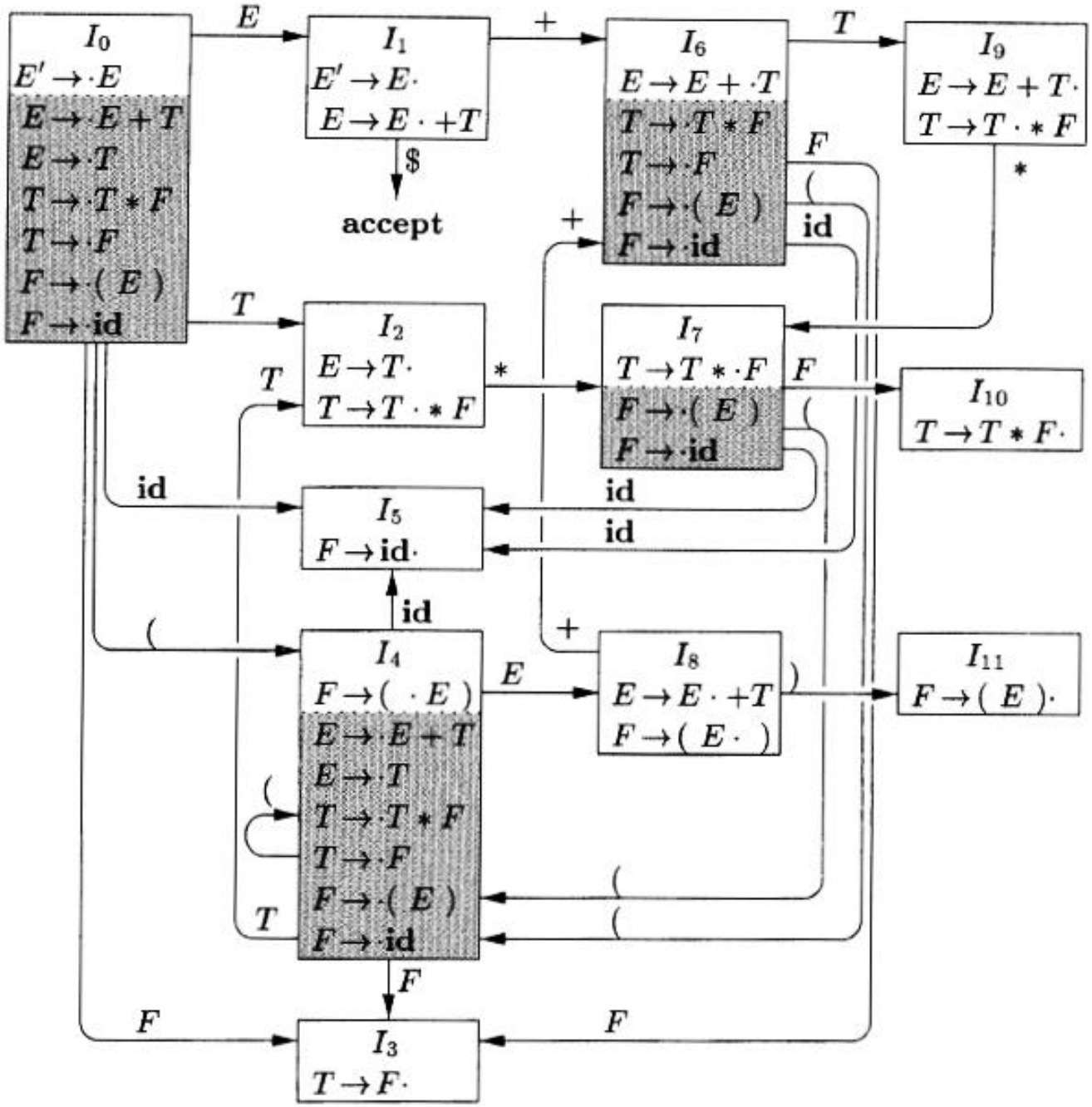
State	ACTION			GOTO
	(	)	$	S
0	s2	r2	r2	1
1			Accept	
2	s2	r2	r2	3
3		s4		
4	s2	r2	r2	5
5		r1	r1	

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

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


I_0
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_1
$E' \to E \cdot$
$E \to E \cdot + T$
I_2
$E \to T \cdot$
$T \to T \cdot * F$
I_3
$T \to F \cdot$
I_4
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_5
$F \to \mathbf{id} \cdot$
I_6
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_7
$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$
I_8
$E \to E \cdot + T$
$F \to (E \cdot)$
I_9
$E \to E + T \cdot$
$T \to T \cdot * F$
I_{10}
$T \to T * F \cdot$
I_{11}
$F \to (E) \cdot$
E
T
F
+
*
(
)
id
$
accept

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		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \to$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \to F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \to$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \to T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \to T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \to$ **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**

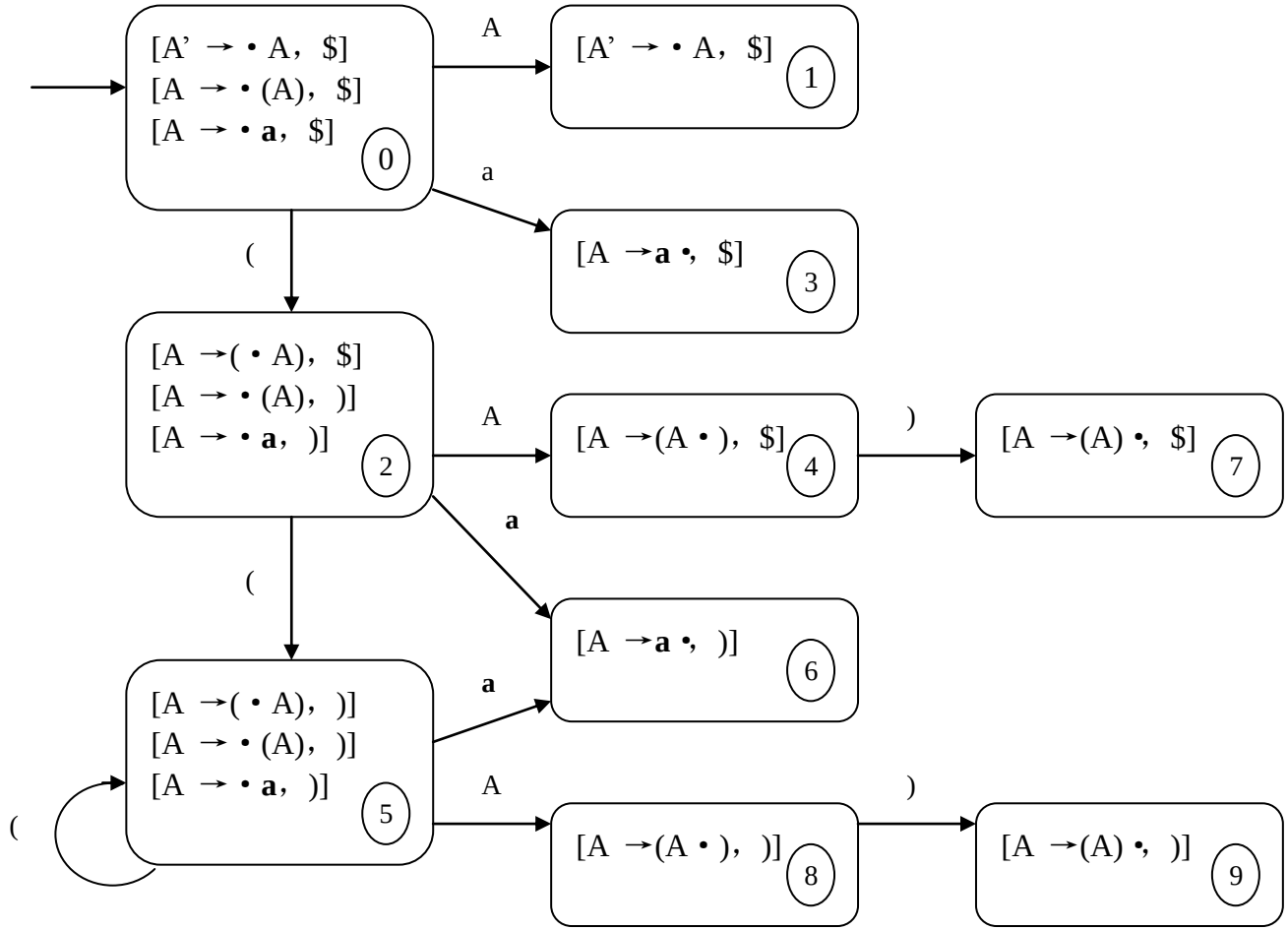
    The DFA items are an extension of LR(0) items

    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 → α·β, a]

    where A → α·βis an LR(0) item and a is a lookahead token

A→(A) | a LR(1) DFA



The Grammar:
(1) A→(A)
(2) A→ **a**

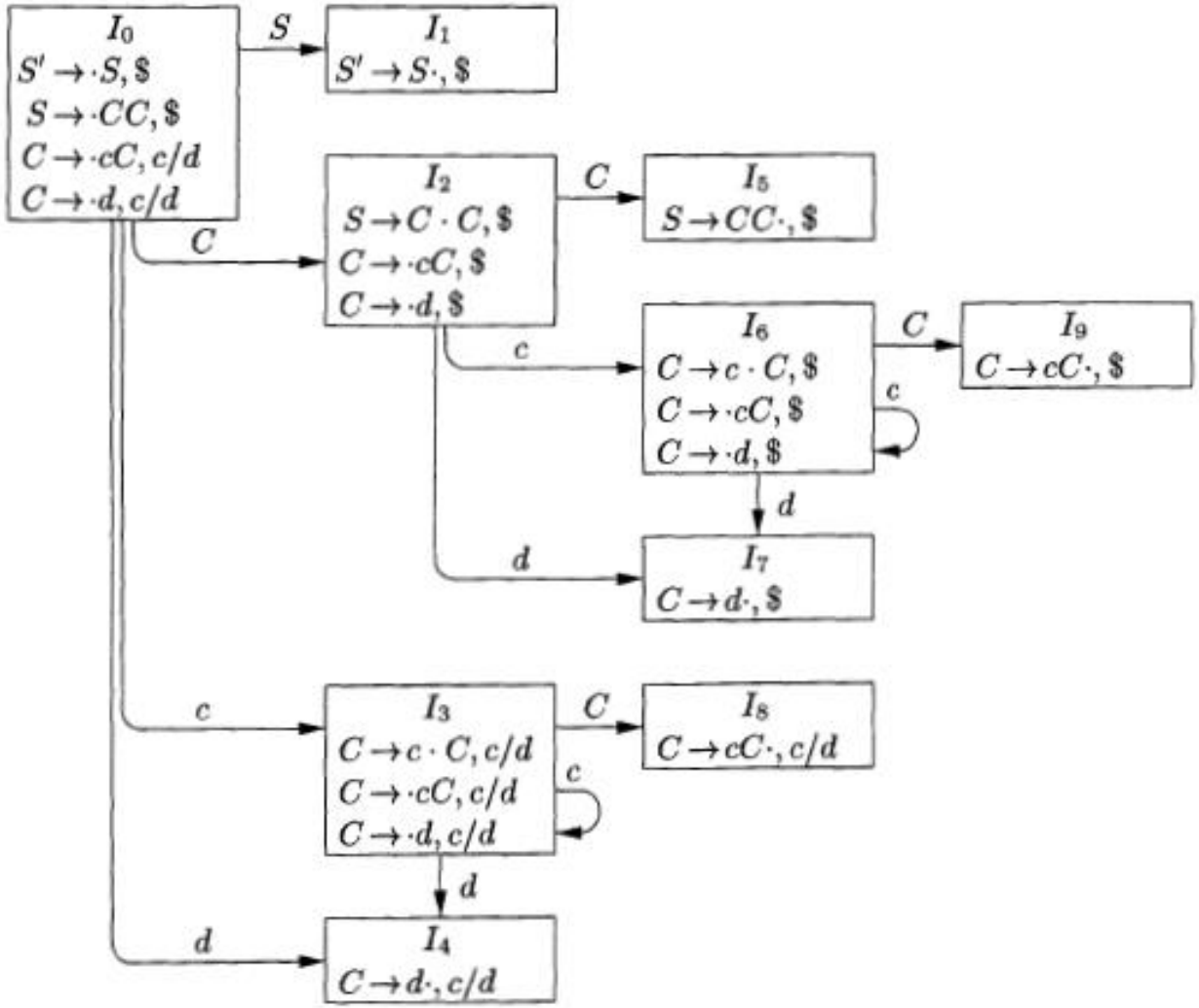
Parse table

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		

Parsing action

Stack	Symbols	Input	Action
$0		(a)$	shift
$02	(	a)$	Shift
$026	(a	)$	Reduce A->a
$024	(A	)$	Shift
$0247	(A)	$	Reduce A->(A)
$0 1	A	$	accept

S->CC
C->cC | d LR(1) DFA



Parse table

State	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

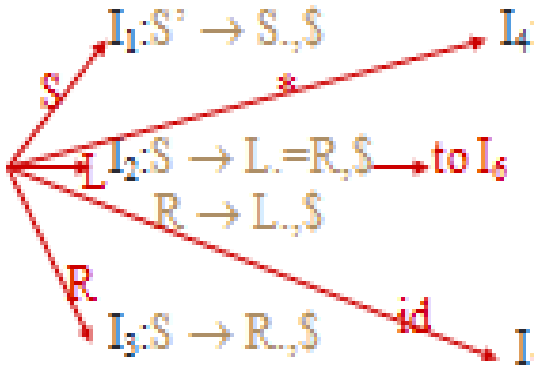
Parsing Action

Stack	Symbols	Input	Action
$0		ccdd$	Shift
$03	c	cdd$	shift
$033	cc	dd$	shift
$0334	ccd	d$	reduce C->d
$033 8	ccC	d$	reduce C->cC
$03 8	cC	d$	reduce c->cC
$02	C	d$	shift
$027	Cd	$	reduce C->d
$02 5	CC	$	reduce S->CC
$0 1	S	$	Accept

Canonical LR(1) Collection

S' → S
1) S → L=R
2) S → R
3) L → *R
4) L → id
5) R → L

I_0: S' → .S,$
S → .L=R,$
S → .R,$
L → .*R,$/=
L → .id,$/=
R → .L,$

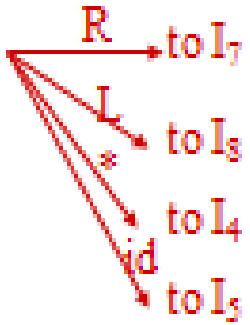


I_1: S' → S.,$

I_2: S → L.=R,$ → to I_6
R → L.,$

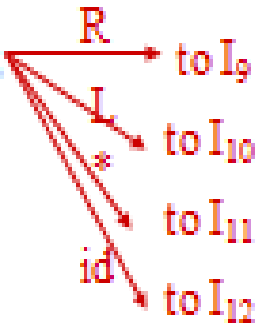
I_3: S → R.,$

I_4: L → *.R,$/=
R → .L,$/=
L → .*R,$/=
L → .id,$/=



I_5: L → id.,$/=

I_6: S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$



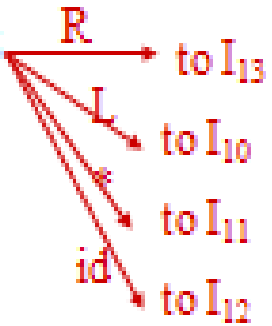
I_7: L → *R.,$/=

I_8: R → L.,$/=

I_9: S → L=R.,$

I_{10}: R → L.,$

I_{11}: L → *.R,$
R → .L,$
L → .*R,$
L → .id,$



I_{12}: L → id.,$

I_{13}: L → *R.,$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			