

LR Parsers

All grammars are not LL(1)

$Z \rightarrow d$
 $Z \rightarrow XYZ$
 $Y \rightarrow \epsilon$
 $Y \rightarrow c$
 $X \rightarrow Y$
 $X \rightarrow a$

	a	c	d	\$
X	$X \rightarrow a$ $X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	
Y	$Y \rightarrow \epsilon$	$Y \rightarrow c$ $Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow XYZ$	$Z \rightarrow XYZ$

$\text{FIRST}(X) = \{a, c, \epsilon\}$

$\text{FIRST}(Y) = \{c, \epsilon\}$

$\text{FIRST}(Z) = \{d, a, c\}$

$\text{FOLLOW}(X) = \{c, d, a\}$

$\text{FOLLOW}(Y) = \{a, c, d\}$

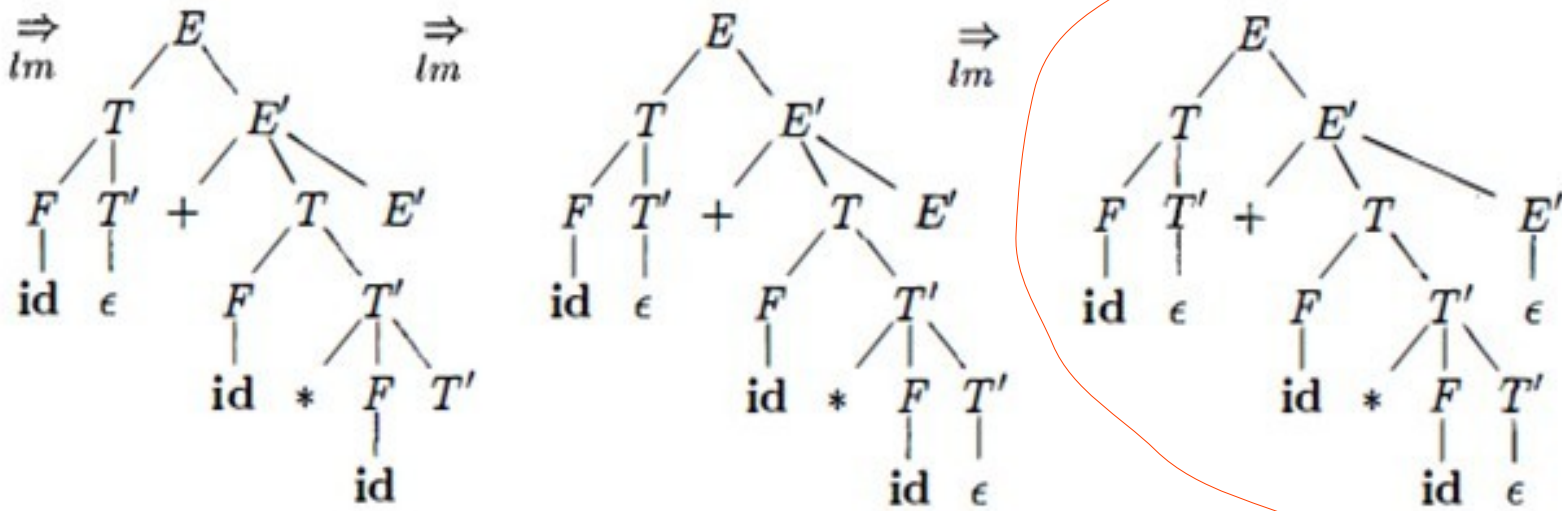
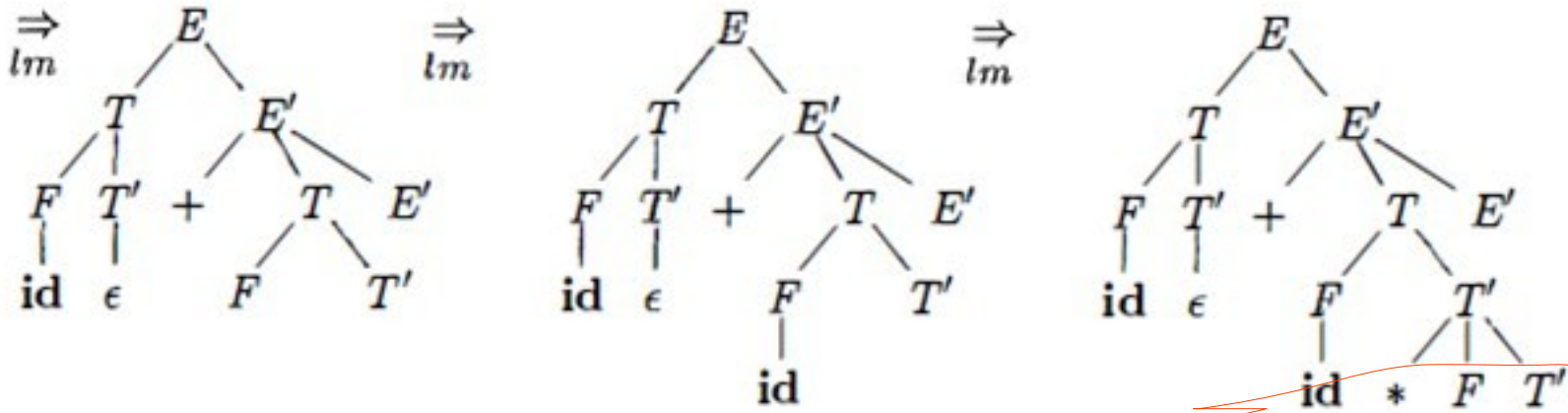
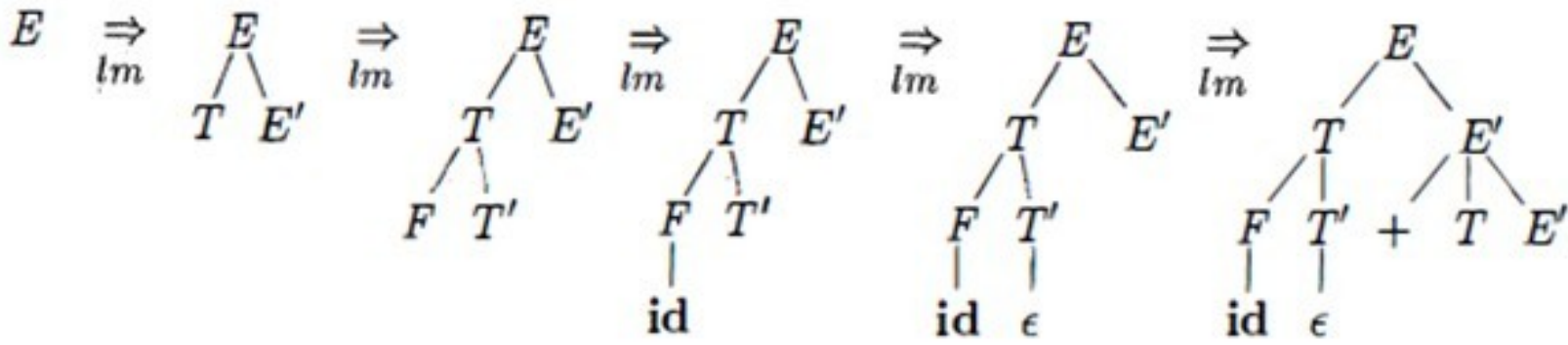
$\text{FOLLOW}(Z) = \{\$ \}$

All grammars are not LL(1)

A grammar is an LL(1) grammar if all productions conform to the following conditions:

1. For each production $A \rightarrow \sigma_1 \mid \sigma_2 \mid \sigma_3 \dots \mid \sigma_n$,
 $\text{FIRST}(\sigma_i) \cap \text{FIRST}(\sigma_j) = \emptyset$, for all $i, j, i \neq j$
2. If nonterminal X derives an empty string, then
 $\text{FIRST}(X) \cap \text{FOLLOW}(X) = \emptyset$

P A R S I N G



Bottom up parsing

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

$\text{id} * \text{id}$

$F * \text{id}$

\mid
 id

$T * \text{id}$

\mid
 F

\mid
 id

$T * F$

\mid
 F

\mid
 id

\mid
 id

T

\mid
 T

\mid
 F

\mid
 id

$*$

\mid
 F

\mid
 id

E

\mid
 T

\mid
 T

\mid
 F

\mid
 id

$*$

\mid
 F

\mid
 id

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$

Bottom up parsing

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

$\text{id} * \text{id}$

$F * \text{id}$

\mid
 id

$T * \text{id}$

\mid
 F

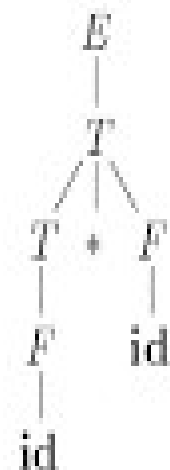
\mid
 id

$T * F$

\mid
 F

\mid
 id

\mid
 id



$E \Rightarrow \underline{T} \Rightarrow \underline{T} * F \Rightarrow T * \underline{\text{id}} \Rightarrow \underline{F} * \text{id} \Rightarrow \underline{\text{id}} * \text{id}$

Bottom up parsing

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

(The string, together with its position in the right sentential form where it occurs and the production used to reduce it).

Reducing β to A in $\alpha\beta w$ is termed as “pruning the handle”

Where w contains only terminal symbols.

Shift-reduce parsing

Step 1: Locating a substring

Step 2: Choosing a production

	Stack	Input
Initial condition	\$	w\$
Final condition	\$S	\$

<u>Stack</u>	<u>Input</u>	<u>Action</u>
\$	id * id \$	shift
\$ id	* id \$	reduce $F \rightarrow id$
\$ F	* id \$	shift
\$ F	* id \$	reduce $T \rightarrow F$
\$ T	* id \$	shift
\$ T *	id \$	shift
\$ T * id	\$	reduce $F \rightarrow id$
\$ T * F	\$	reduce $T \rightarrow T * F$
\$ T	\$	reduce $E \rightarrow T$
\$ E	\$	Accept

Viable Prefix- The sequence of symbols on the parsing stack

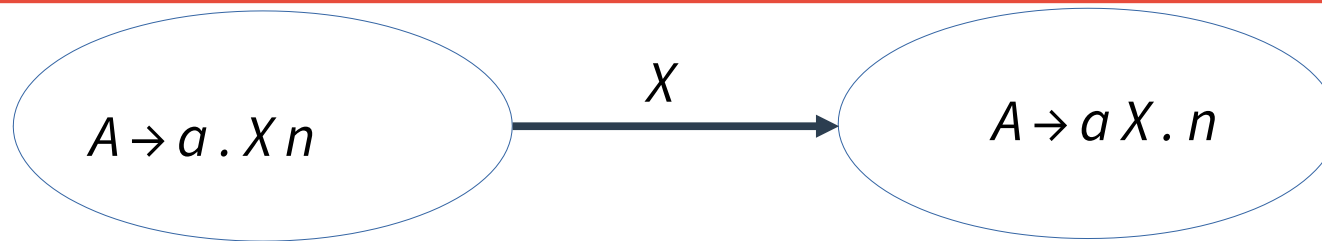
LR Parsing

- **LR(k) parsing, introduced by D. Knuth in 1965**
 - L is for Left-to-right scanning of the input
 - R is for reverse Rightmost derivation
 - k is the number of lookahead tokens
- **Most general non-backtracking shift-reduce parsing method**
- **Can be constructed to recognise virtually all programming language constructs**
- **The class of grammar is a proper superset of the class of grammars that can be parsed with *predictive parsers***

LR Parsing

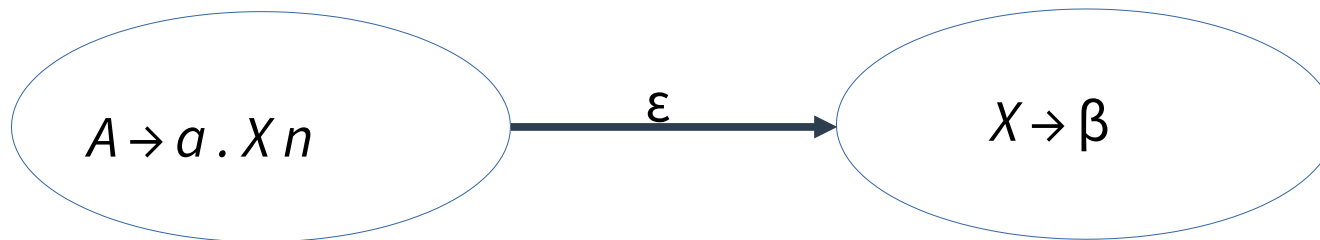
- **Different types:**
 - *Simple LR or SLR, the easiest method for constructing shift-reduce parsers*
 - *Canonical LR*
 - *LALR*
- **LR-parsers represent the DFA as a 2D – table**
- **Rows correspond to DFA states**
- **Columns correspond to terminals (action table) and non-terminals (goto table)**

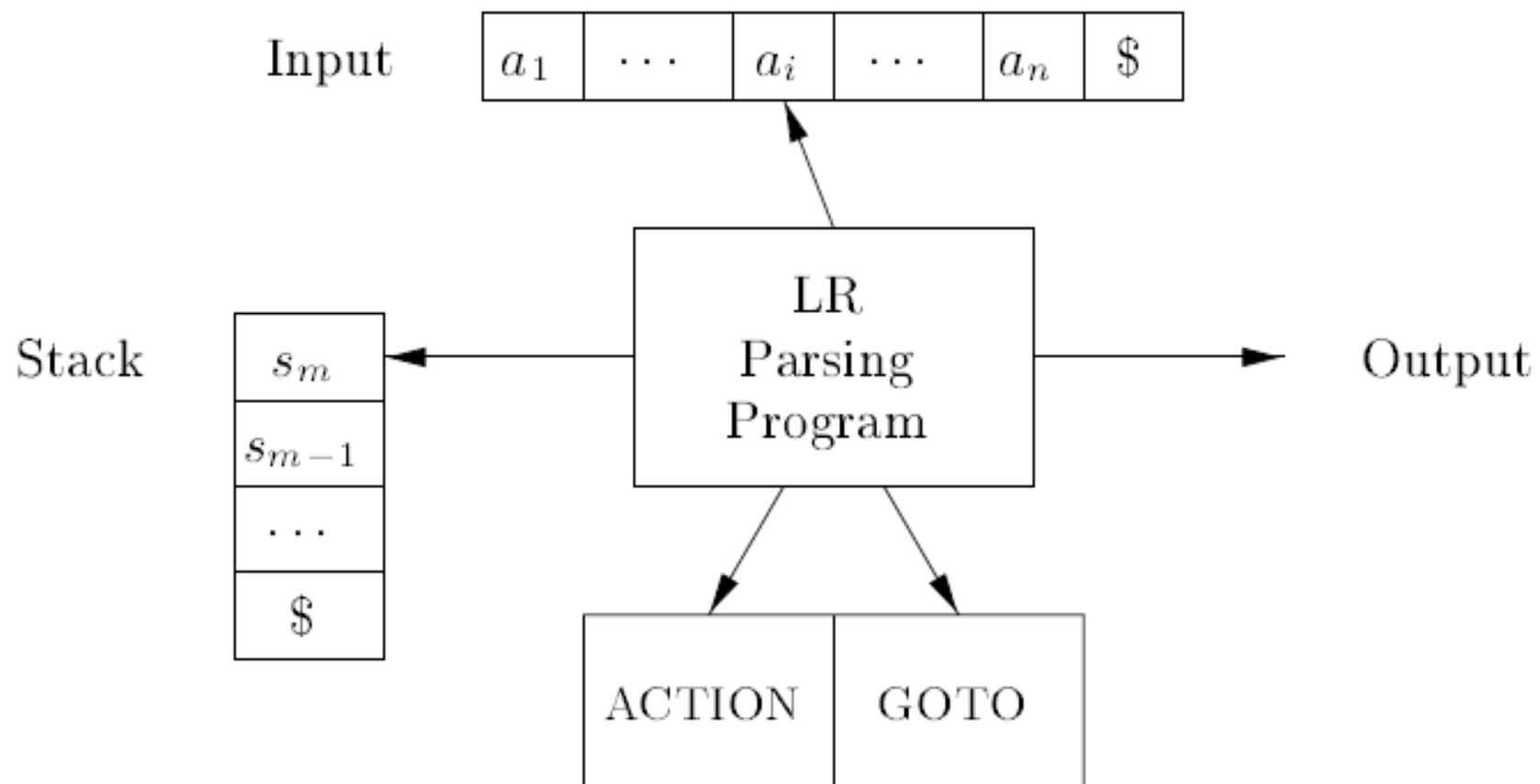
Transition between states



Two possibilities

1. If X is a terminal, then push X onto the stack
2. If X is a non-terminal, X will be pushed onto the stack during a reduction by $X \rightarrow \beta$.
3. Thus, we need to first recognize β
4. So for every production $X \rightarrow \beta$, we need to indicate that X can be produced by reducing (recognising) any of the right hand sides of such productions.
5. This is basically the following transition.





SLR Parsing

LR(0) - **zero** tokens of look-ahead

SLR - Simple LR: like LR(0), but uses FOLLOW sets to build more “precise” parsing tables

First we augment the grammar G with a new start symbol S' and a production

$S' \rightarrow S$

Closure operation -

For a set of items I , we construct $\text{closure}(I)$ as follows:

Every item in I is added to $\text{closure}(I)$

If $A \rightarrow \alpha . B \beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production, then $B \rightarrow . \gamma$ is added to $\text{closure}(I)$

Goto operation -

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

Example

Augmented Grammar:

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

If $I = \{ E' \rightarrow \bullet E \}$,

then $\text{CLOSURE}(I)$ is

$E' \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet \text{id}$

Example

Augmented Grammar:

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id}$

$I =$

$E' \rightarrow E \bullet$

$E \rightarrow E \bullet + T$

GOTO(I, +)

$E \rightarrow E + \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet \text{id}$

The Canonical LR(0) Collection -- Example

$I_0: E' \rightarrow .E$ $I_1: E' \rightarrow E.$ $I_6: E \rightarrow E+.T$

$E \rightarrow .E+T$

$E \rightarrow E.+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$I_2: E \rightarrow T.$

$T \rightarrow .F$

$T \rightarrow T.*F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_3: T \rightarrow F.$

$I_4: F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_5: F \rightarrow id.$

$I_9: E \rightarrow E+T.$

$T \rightarrow .T*F$

$T \rightarrow T.*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_{10}: T \rightarrow T*F.$

$I_7: T \rightarrow T*.F$

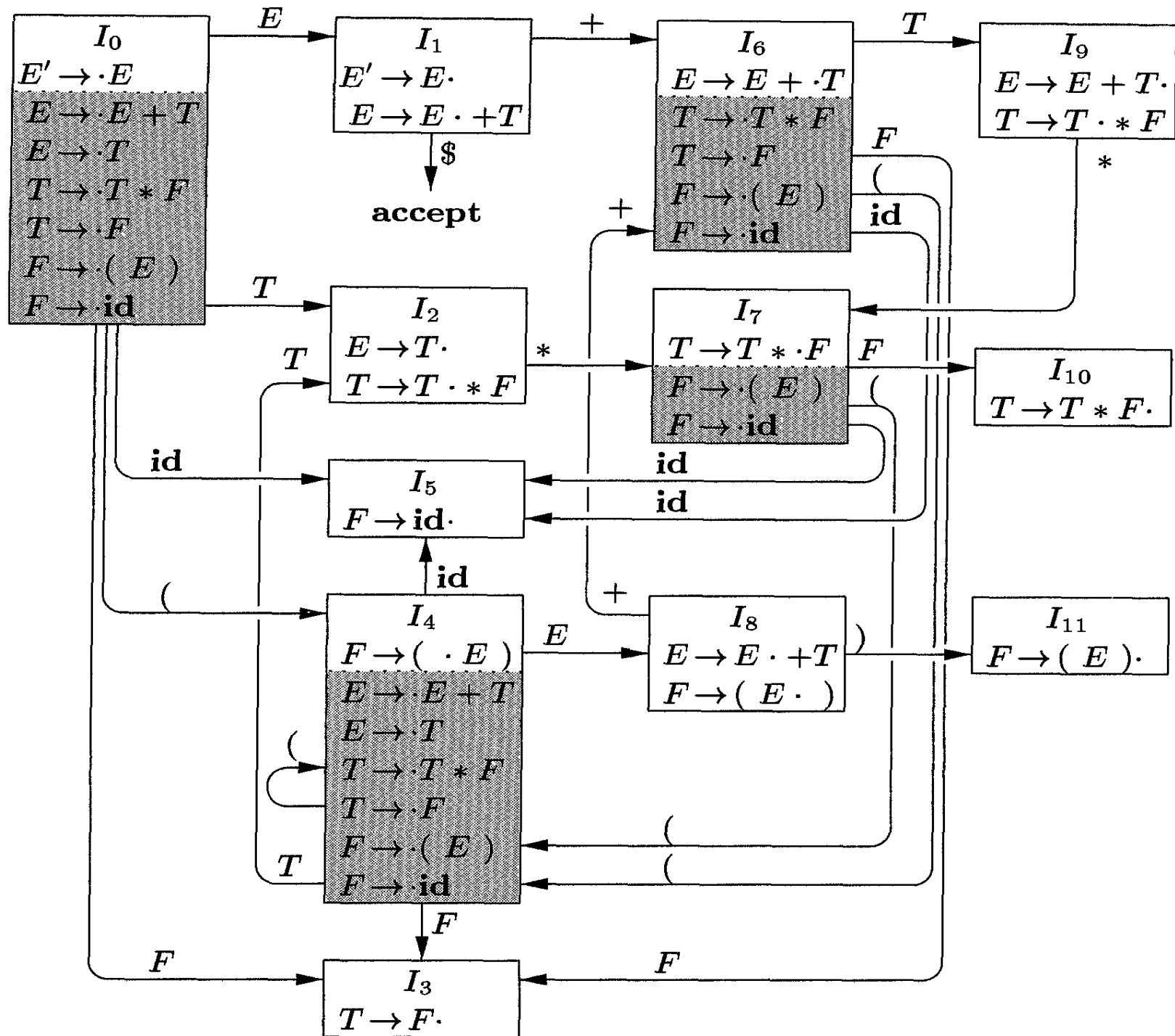
$F \rightarrow .(E)$

$F \rightarrow .id$

$I_{11}: F \rightarrow (E).$

$I_8: F \rightarrow (E.)$

$E \rightarrow E.+T$



Constructing SLR Parsing Table

Consider the grammar G'

1) Construct the canonical collection of sets of LR(0) items for G' ,

2) $C \leftarrow \{I_0, I_1 \dots I_n\}$

3) Create the parsing action table as follows:

1) If a is a terminal, $A \rightarrow \alpha . a \beta$ in I_i , and $\text{goto}(I_i, a) = I_j$, then $\text{action}[i, a] = \text{shift } j$

2) If $A \rightarrow \alpha .$ is in I_i , then $\text{action}[i, a]$ is **reduce** $A \rightarrow \alpha$ for all a in $\text{FOLLOW}(A)$, where $A \neq S'$

3) If $S' \rightarrow S$ is in I_i , then **action** $[i, \$]$ is **accept**

4) Create the parsing goto table

1) For all non-terminals A , if $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$

5) All other entries will be marked as error

6) *Initial state of the parser contains $S' \rightarrow S$*

Parsing Tables of Expression Grammar

Action Table

Goto Table

state	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				

1. $E' \rightarrow E$

2. $E \rightarrow E + T$

3. $E \rightarrow T$

4. $T \rightarrow T^* F$

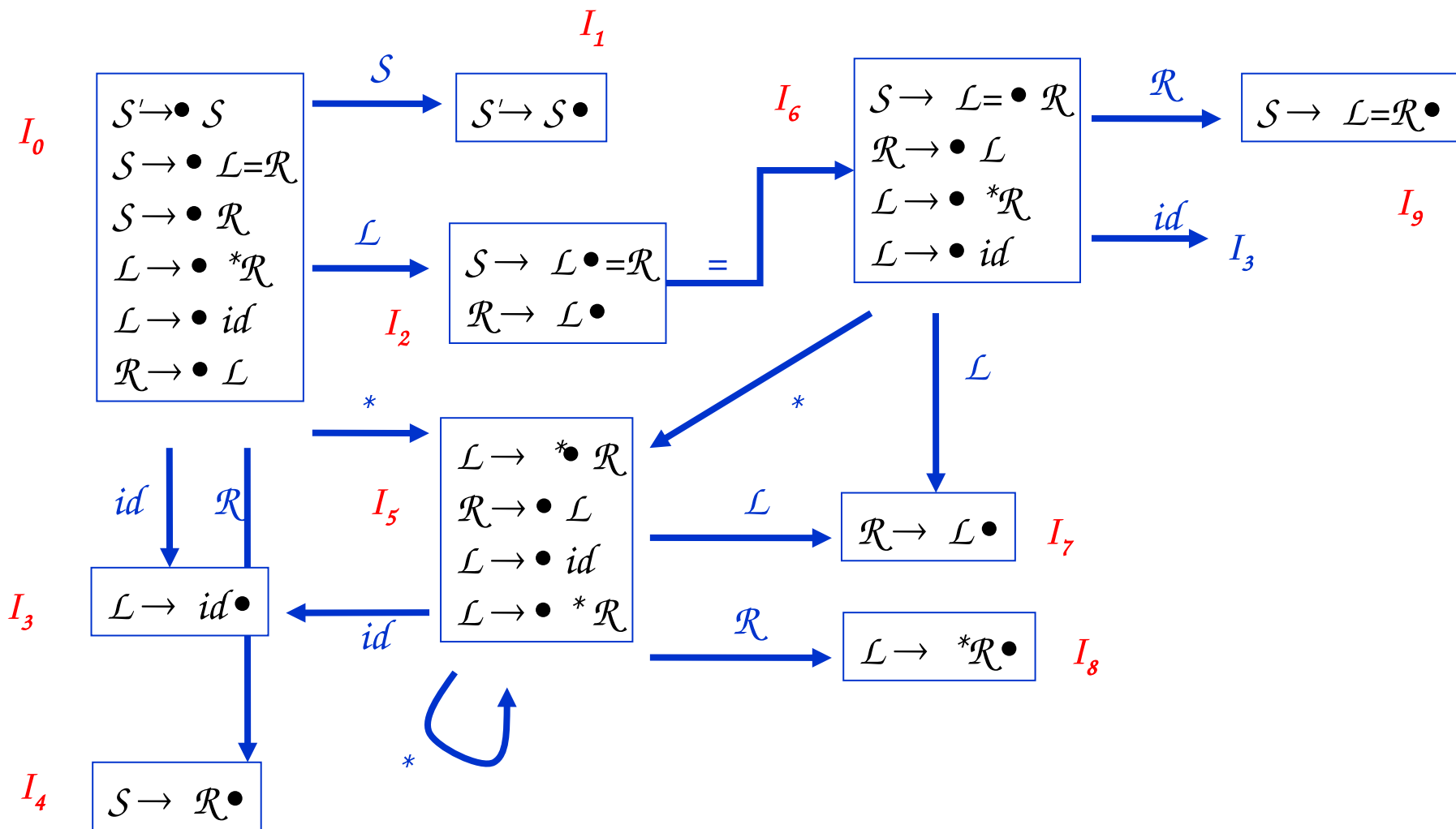
5. $T \rightarrow F$

6. $F \rightarrow (E)$

7. $F \rightarrow id$

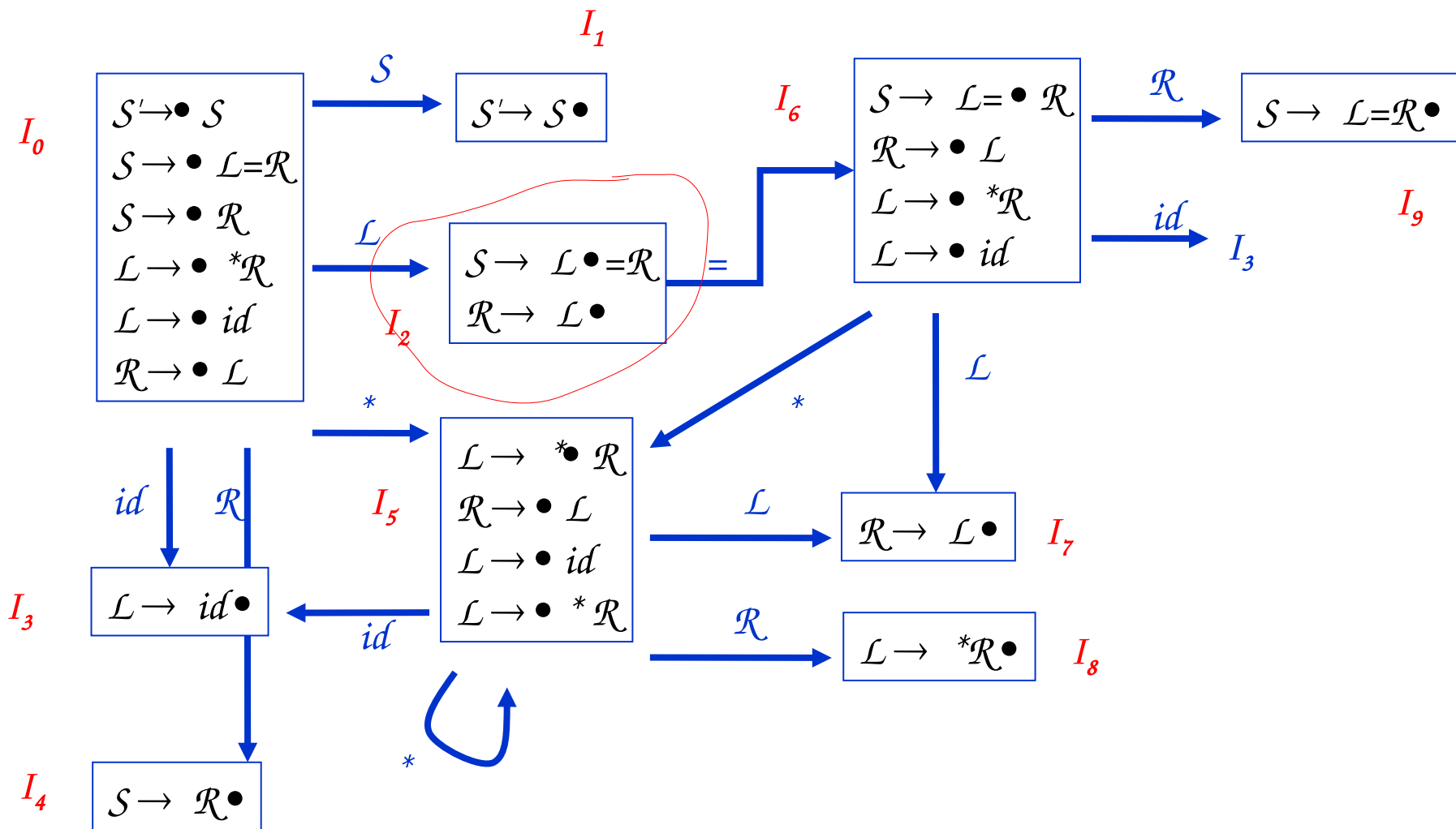
Parsing

Stack	Input	Parser Action
S0	id+id \$	Shift and move to state 5
S0 id S5	+id \$	Reduce $F \rightarrow id$
S0 F S3	+id \$	Reduce $T \rightarrow id$
S0 T S2	+id \$	Reduce $E \rightarrow T$
S0 E S1	+id \$	Shift and move to state 6
S0 E S1 + S6	id \$	Shift and move to state 5
S0 E S1 + S6 id S5	\$	Reduce $F \rightarrow id$
S0 E S1 + S6 F S3	\$	Reduce $T \rightarrow F$
S0 E S1 + S6 T S9	\$	Reduce $E \rightarrow E+T$
S0 E S1	\$	Accept



<i>state</i>	<i>id</i>	<i>action</i>	<i>goto</i>	<i>\$</i>	<i>S</i>	<i>L</i>	<i>R</i>
	<i>s3</i>	=	*				
0	<i>s3</i>		<i>s5</i>		1	2	4
1				accept			
2		$s6/r(\mathcal{R} \rightarrow \mathcal{L})$					
3		$r(\mathcal{L} \rightarrow id)$		$r(\mathcal{L} \rightarrow id)$			
4				$r(\mathcal{S} \rightarrow \mathcal{R})$			
5	<i>s3</i>		<i>s5</i>			7	8
6	<i>s3</i>		<i>s5</i>			7	9
7		$r(\mathcal{R} \rightarrow \mathcal{L})$		$r(\mathcal{R} \rightarrow \mathcal{L})$			
8		$r(\mathcal{L} \rightarrow *\mathcal{R})$		$r(\mathcal{L} \rightarrow *\mathcal{R})$			
9				$r(\mathcal{S} \rightarrow \mathcal{L} = \mathcal{R})$			

<i>state</i>	<i>id</i>	<i>action</i>	<i>goto</i>	<i>\$</i>	<i>S</i>	<i>L</i>	<i>R</i>
	<i>s3</i>	=	*				
0	<i>s3</i>		<i>s5</i>		1	2	4
1				accept			
2		<i>s6/r(R→L)</i>					
3		<i>r(L→id)</i>		<i>r(L→id)</i>			
4				<i>r(S→R)</i>			
5	<i>s3</i>		<i>s5</i>			7	8
6	<i>s3</i>		<i>s5</i>			7	9
7		<i>r(R→L)</i>		<i>r(R→L)</i>			
8		<i>r(L→*R)</i>		<i>r(L→*R)</i>			
9				<i>r(S→L=R)</i>			



Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow \mathcal{L}=\mathcal{R}$

$S \rightarrow \mathcal{R}$

$\mathcal{L} \rightarrow *\mathcal{R}$

$\mathcal{L} \rightarrow id$

$\mathcal{R} \rightarrow \mathcal{L}$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(\mathcal{L}) = \{\$, =\}$

$FOLLOW(\mathcal{R}) = \{\$, =\}$

Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow L=R$

$S \rightarrow \mathcal{R}$

$\mathcal{L} \rightarrow *\mathcal{R}$

$\mathcal{L} \rightarrow id$

$\mathcal{R} \rightarrow \mathcal{L}$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(\mathcal{L}) = \{\$, =\}$

$FOLLOW(\mathcal{R}) = \{\$, =\}$

Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow \mathcal{L}=\mathcal{R}$

$S \rightarrow \mathcal{R}$

$\mathcal{L} \rightarrow *\mathcal{R}$

$\mathcal{L} \rightarrow id$

$\mathcal{R} \rightarrow \mathcal{L}$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(\mathcal{L}) = \{\$, =\}$

$FOLLOW(\mathcal{R}) = \{\$, =\}$

Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(L) = \{\$, =\}$

$FOLLOW(R) = \{\$, =\}$

Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(L) = \{\$, =\}$

$FOLLOW(R) = \{\$, =\}$

Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(L) = \{\$, =\}$

$FOLLOW(R) = \{\$, =\}$

Two possible scenarios

1) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow id=id$

2) $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id$
 $\Rightarrow *L=id \Rightarrow *id=id$

$S' \rightarrow S$

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$FOLLOW(S') = \{\$ \}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(L) = \{\$, = \}$

$FOLLOW(R) = \{\$, = \}$

Two possible scenarios

$S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id$
 $\Rightarrow id=id$

$S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id$
 $\Rightarrow *R=id \Rightarrow *L=id \Rightarrow *id=id$

Stack	Input	• Action
\$id	=id\$	Reduce
\$L	=id\$	Shift
\$L=	id\$	Shift
\$L=id	\$	Reduce
\$L=L	\$	Reduce
\$L=R	\$	

Stack	Input	• Action
\$*id	=id\$	Reduce
\$*L	=id\$	Reduce

$FOLLOW(L) = \{\$, =\}$

Two types of Conflicts in LR parsing

- shift/reduce conflict
 - On some particular lookahead it is possible to shift or reduce
 - The if/else ambiguity would give rise to a shift/reduce conflict
- reduce/reduce
 - This occurs when a state contains more than one handle that may be reduced on the same lookahead.

Two types of Conflicts in LR parsing

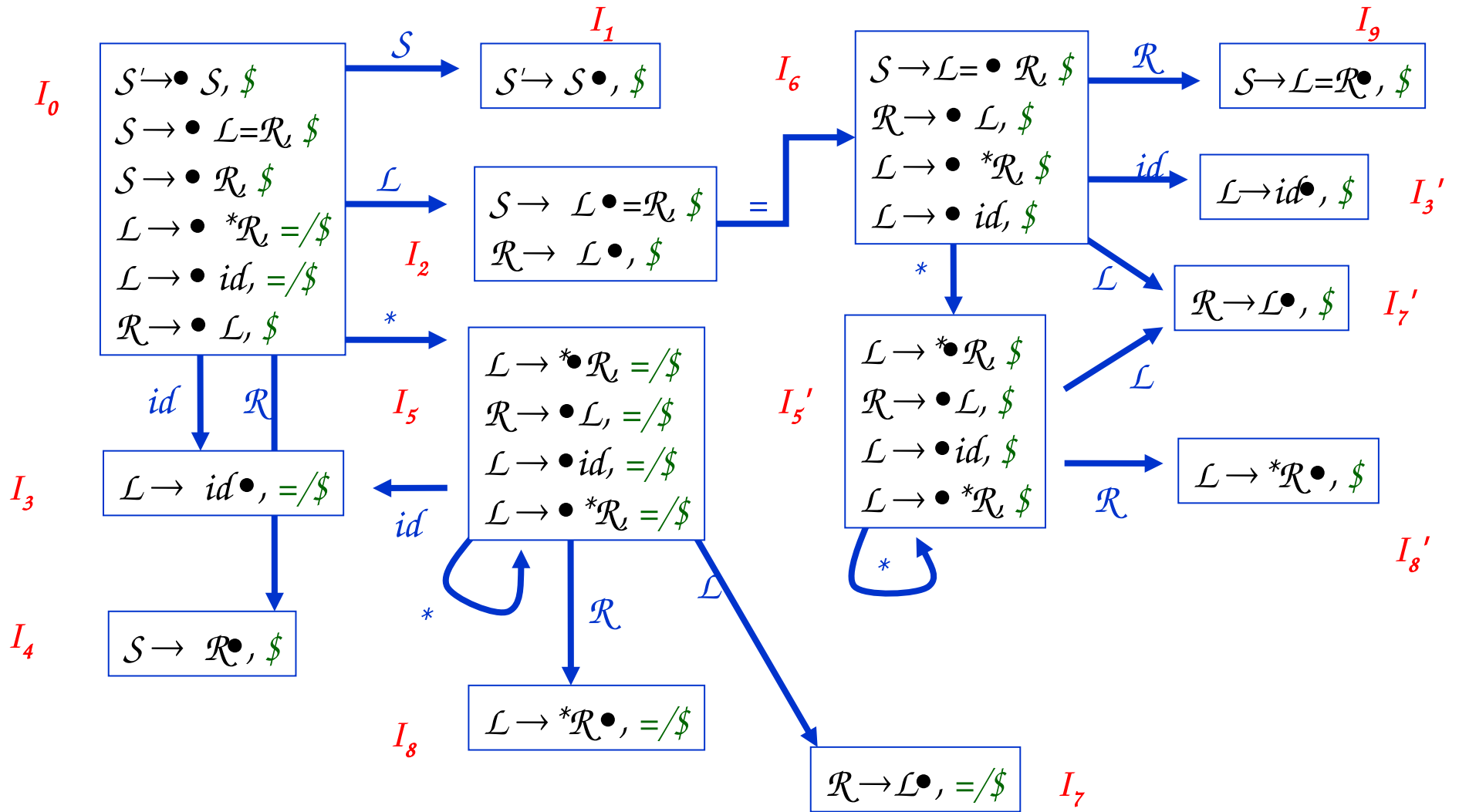
- The conflict occurred because we made a decision about when to reduce based on what token *may* follow a non-terminal at any time.
- However, the fact that a token t may follow a non-terminal N in some derivation does not necessarily imply that t will follow N in some other derivation.
- SLR parsing does not make a distinction.
- Solution : instead of using general FOLLOW information, try to keep track of exactly what tokens may follow a non-terminal in each possible derivation and perform reductions based on that knowledge.

LR(1) items

- In LR(1) items, we also save all possible lookaheads
- The closure function for LR(1) items is defined as follows:

For each item $A \rightarrow \alpha \bullet B \beta$, x in state I ,
each production $B \rightarrow \gamma$ in the grammar,
and each terminal b in $\text{FIRST}(\beta x)$,
add $B \rightarrow \bullet \gamma, b$ to I

If a state contains core item $B \rightarrow \bullet \gamma$ with multiple possible lookaheads b_1, b_2, \dots , we write $B \rightarrow \bullet \gamma, b_1/b_2$ as shorthand for $B \rightarrow \bullet \gamma, b_1$ and $B \rightarrow \bullet \gamma, b_2$



Canonical LR(1) parsing

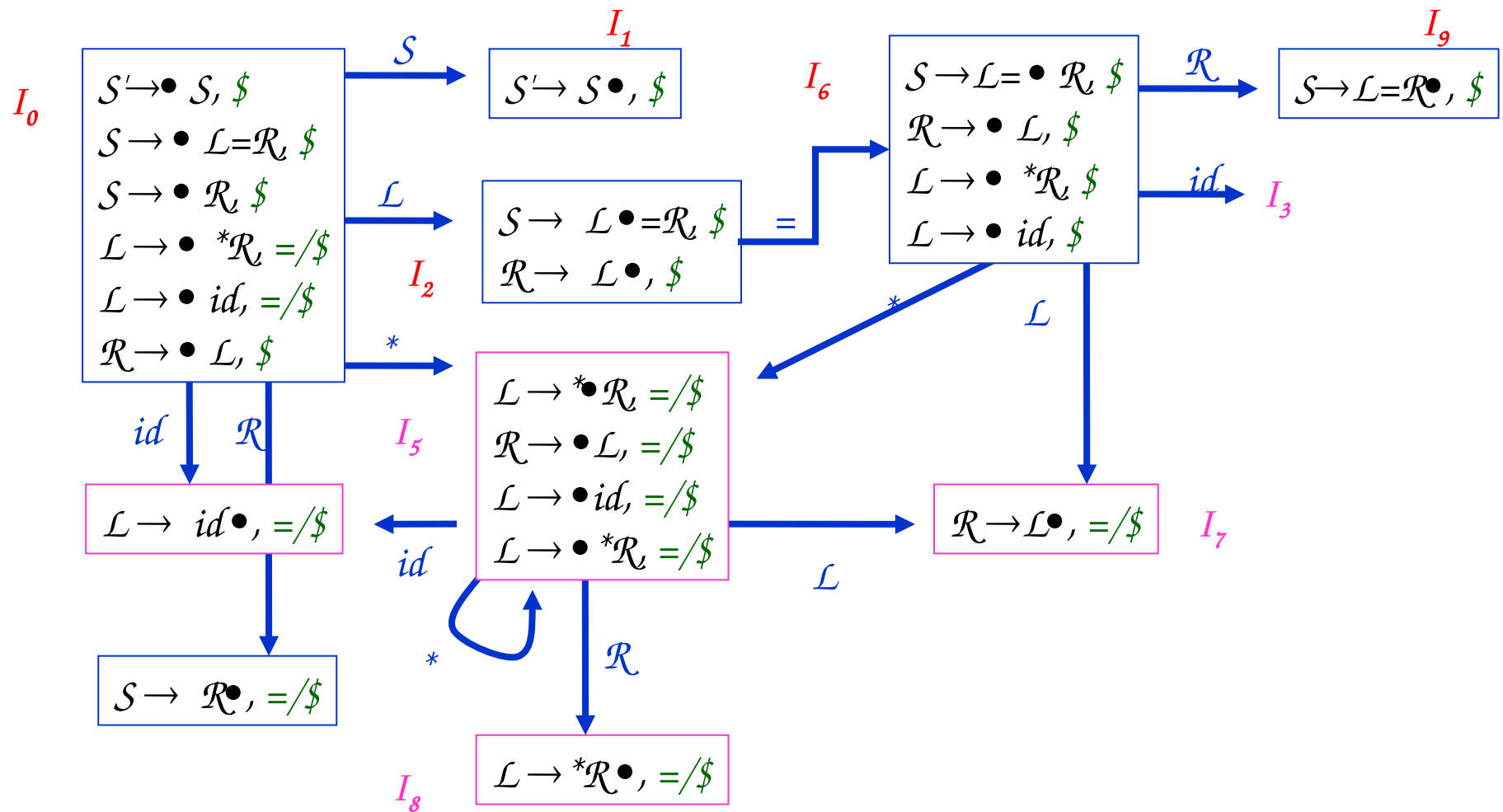
- Rules for creating the table is same as SLR, except we now use the possible lookahead tokens saved in each state, instead of the FOLLOW sets.
- The conflicts that appeared in the SLR parser is now resolved.
- However, the LR(1) parser has many more states. This is not very practical.

Canonical LR Parsing Table

	id	*	=	\$		S	L	R
0	S3	S5				1	2	4
1				accept				
2			S6	R(R->L)				
3			R(L->id)	R(L->id)				
4				R(S->R)				
5	S3	S5					7	8
6	S3'	S5'					7'	9
7			R(R->L)	R(R->L)				
8			R(L->*R)	R(L->*R)				
9				R(S->L=R)				
3'				R(L->id)				
5'	S3'	S5'						
7'				R(R->L)				
8'				R(L->*R)				

LALR Parsing

- It is required to reduce the number of states in an LR(1) parser.
- Some states in the LR(1) automaton have the same core items and differ only in the possible lookahead information.
- They also have similar transitions.
- For example, states I_3 and I_3' , I_5 and I_5' , I_7 and I_7' , I_8 and I_8' are states with same core items and similar transitions.
- Such states are merged in LALR parser.
- In our example - SLR : 10 states, LR(1): 14 states, LALR(1) : 10 states



	id	*	=	\$		S	L	R
0	S3	S5				1	2	4
1				accept				
2			S6	R(R->L)				
3			R(L->id)	R(L->id)				
4				R(S->R)				
5	S3	S5					7	8
6	S3'	S5'					7'	9
7			R(R->L)	R(R->L)				
8			R(L->*R)	R(L->*R)				
9				R(S->L=R)				

Conflicts in LALR(1) parsing

- LALR(1) parsers cannot introduce shift/reduce conflicts.
 - Such conflicts are caused when a lookahead is the same as a token on which we can shift.
- LALR(1) parsers can introduce reduce/reduce conflicts.
- Here's a situation when this might happen:

