LR Parsers

All grammars are not LL(1)

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

	a	c	d	$
X	X → a X → Y	X → Y	X → Y	
Y	Y → ε	Y → c Y → ε	Y → ε	
Z	Z→XYZ	Z→XYZ	Z → d Z→XYZ	Z→XYZ

FIRST (X) = {a, c, ε}
FIRST (Y) = {c, ε}}
FIRST (Z) = {d, a, c}
FOLLOW(X) = {c, d, a}
FOLLOW(Y) = {a, c, d}
FOLLOW(Z) = {$}

All grammars are not LL(1)

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

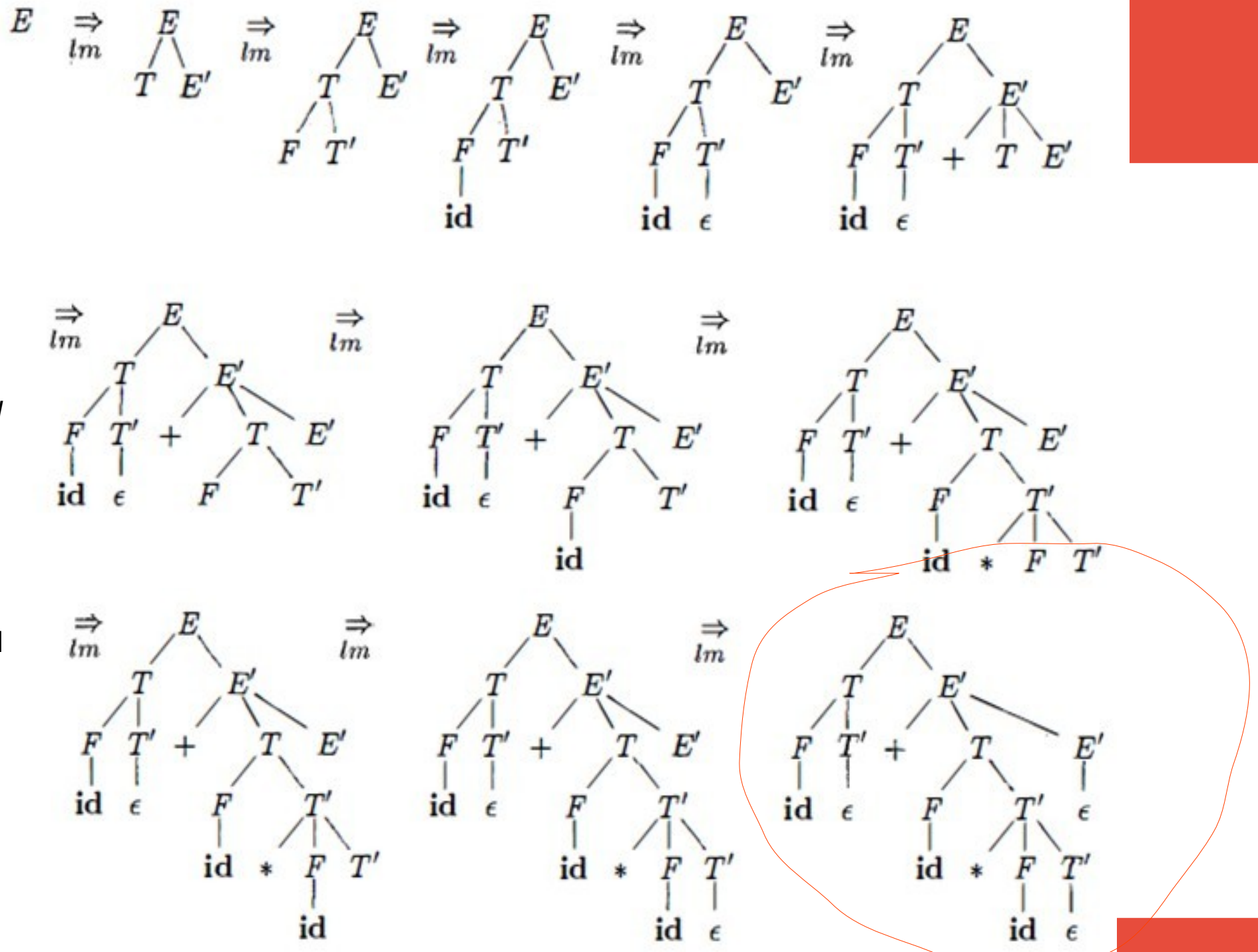
1. For each production $A \rightarrow \sigma 1 \mid \sigma 2 \mid \sigma 3 \ldots \mid \sigma n$,

$FIRST(\sigma_i) \cap FIRST(\sigma_j) = \emptyset$, for all i, j, i <> j

2. If nonterminal X derives an empty string, then

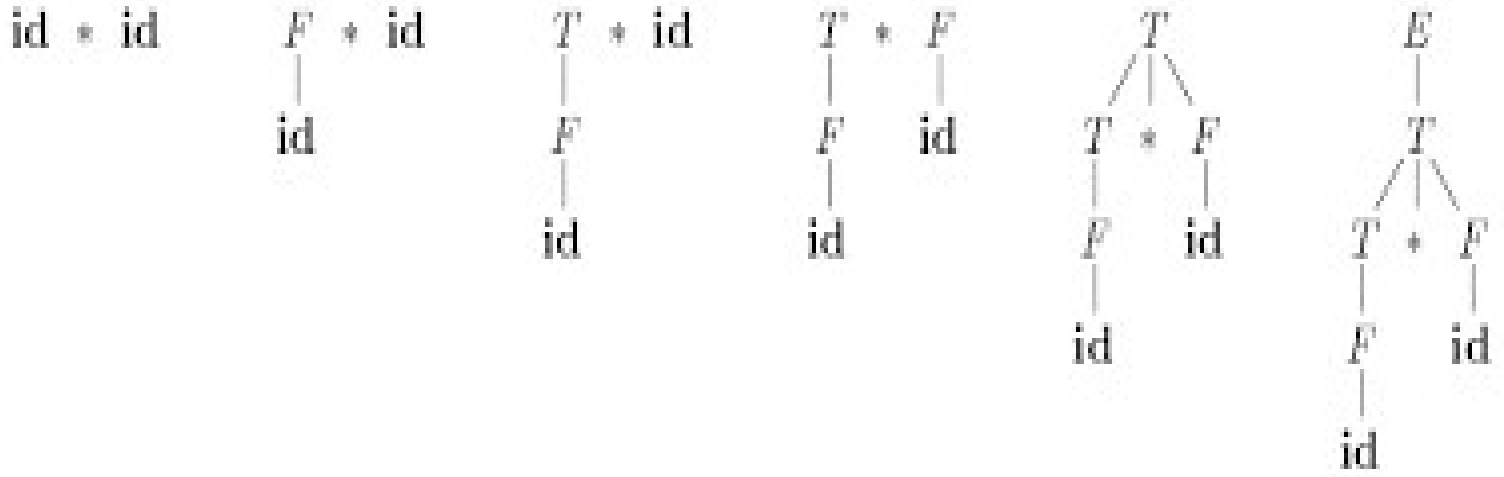
$FIRST(X) \cap FOLLOW(X) = \emptyset$

TOP DOWN PARSING



Bottom up parsing

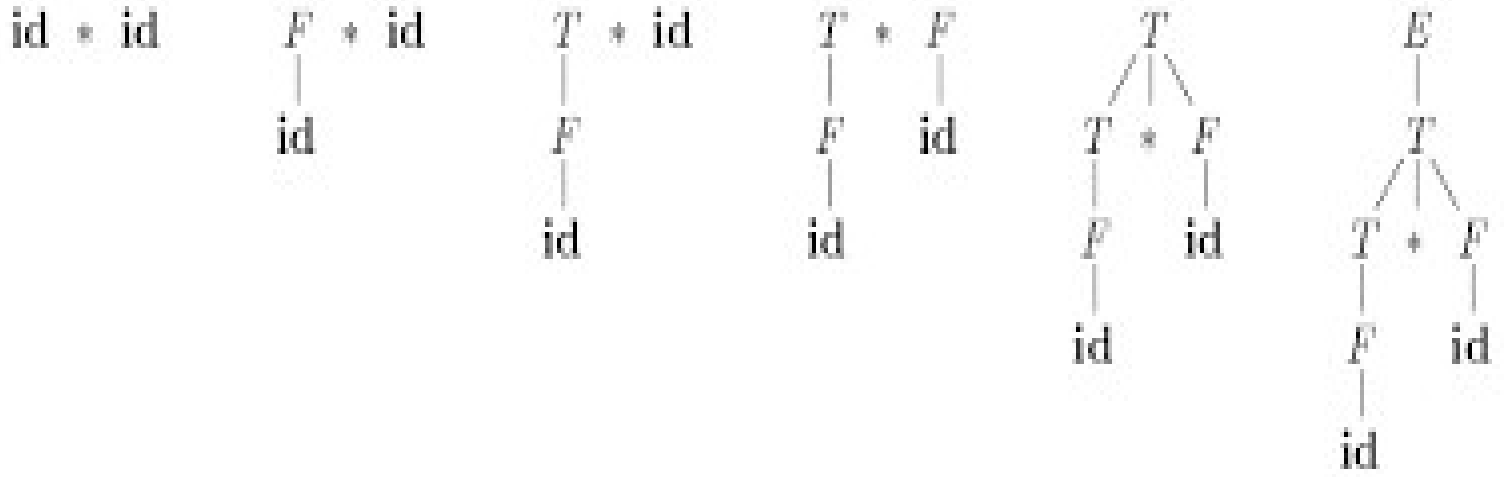
E → E + T | T
T → T * F | F
F → (E) | id



E => T => T * F => T * id => F * id => id * id

Bottom up parsing

E → E + T | T
T → T * F | F
F → (E) | id



E => T => T * F => T * id => F * id => id * 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 αβw 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	$

Viable Prefix- The sequence of symbols on the parsing stack

Stack	Input	Action
$	id * id $	shift
$ id	* id $	reduce F → id
$ F	* id $	shift
$ F	* id $	reduce T → F
$ T	* id $	shift
$ T *	id $	shift
$ T * id	$	reduce F → id
$ T * F	$	reduce T→T* F
$ T	$	reduce E → T
$ E	$	Accept

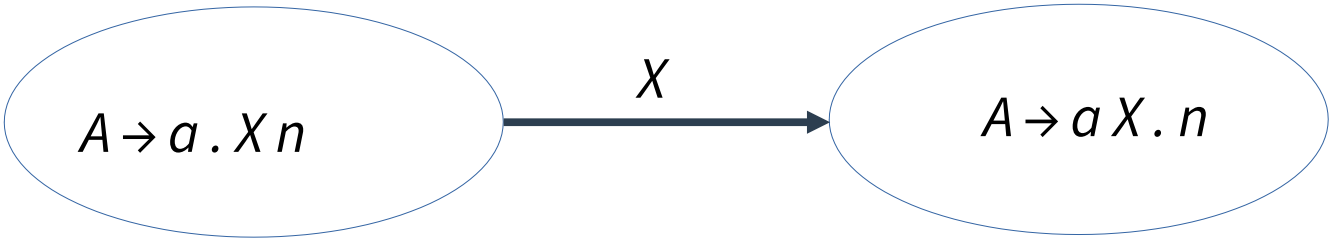
LR Parsing

- **LR(k) parsing, introduce 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

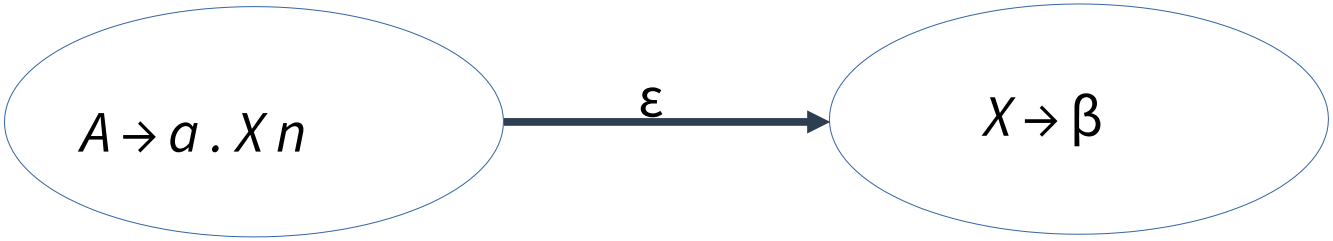
- **Different types:**
  - *Simple LR or SLR, the easiest method for constructing shift-reduce parsers*
  - *Canonical LR*
  - *LALR*
- **LR-parsers represent the DFA as a 2D – table**
- **Rows correspond to DFA states**
- **Columns correspond to terminals (action table) and non-terminals (goto table)**

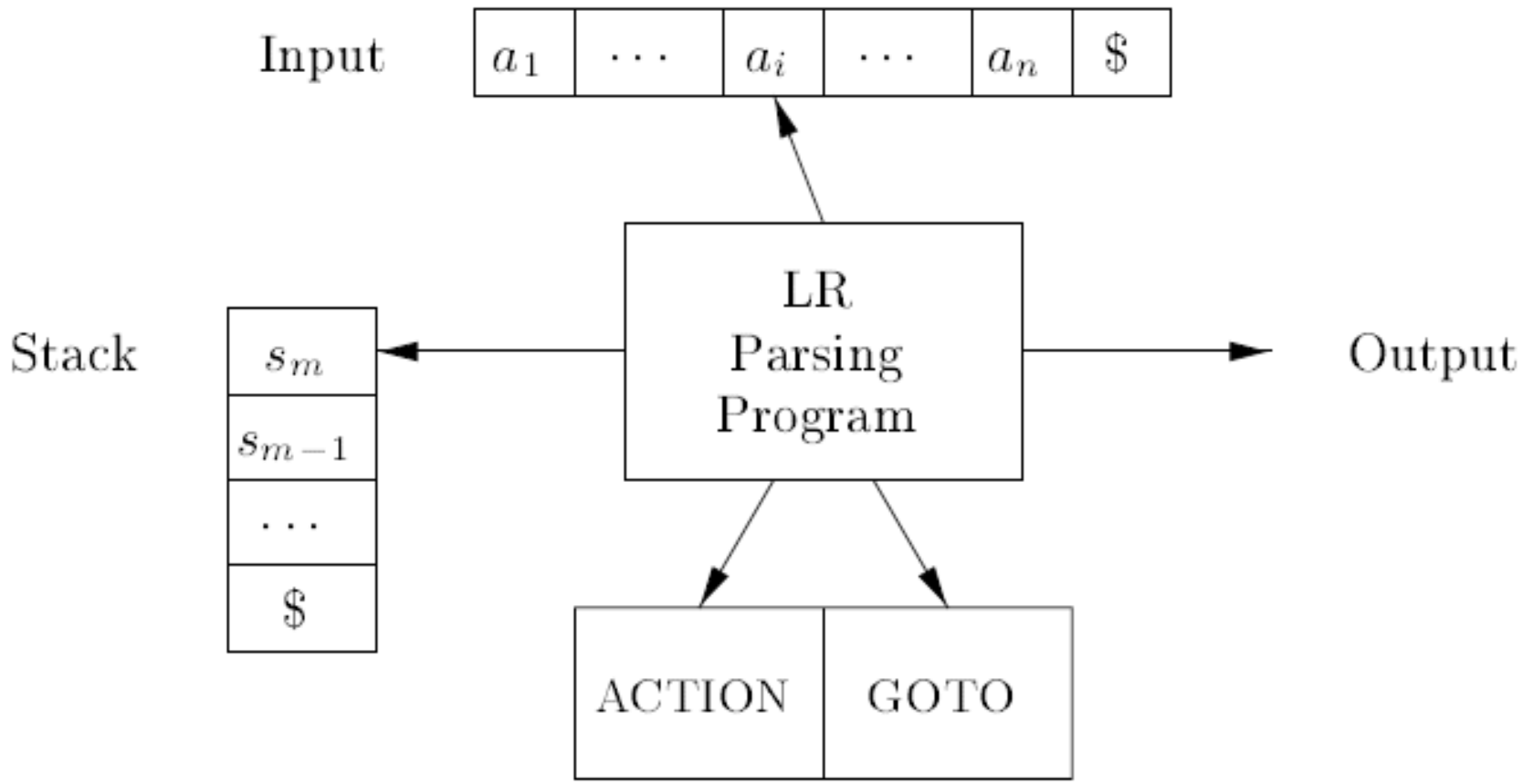
Transition between states



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 → β.
3. Thus, we need to first recognize β
4. So for every production X → β, 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.




Input
a_1
$\cdots$
a_i
$\cdots$
a_n
$
LR
Parsing
Program
Stack
s_m
s_{m-1}
$\cdots$
$
Output
ACTION
GOTO

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' → S

Closure operation -

For a set of items I, we construct closure(I) as follows:

Every item in I is added to closure (I)

If A → α . B β is in closure (I) and B → γ is a production, then B → . γ is added to closure (I)

Goto operation -

Goto(I, X) is the closure of the set of all items [A → α X . β], such that [A→ α . X β] is in I.

Example

Augmented Grammar:

E' → E

E → E + T | T

T → T* F | F

F → (E) | id

If I = { E' → • E },

then CLOSURE(I) is

E' → • E

E → • E + T

E → • T

T → • T * F

T → • F

F → • (E)

F → • id

Example

Augmented Grammar:

E' → E

E → E + T | T

T → T* F | F

F → (E) | id

I =

E' → E •

E → E • + T

GOTO(I, +)

E → *E* + . *T*

T → . *T* * *F*

T → . *F*

F → . (*E*)

F → . **id**

The Canonical LR(0) Collection -- Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

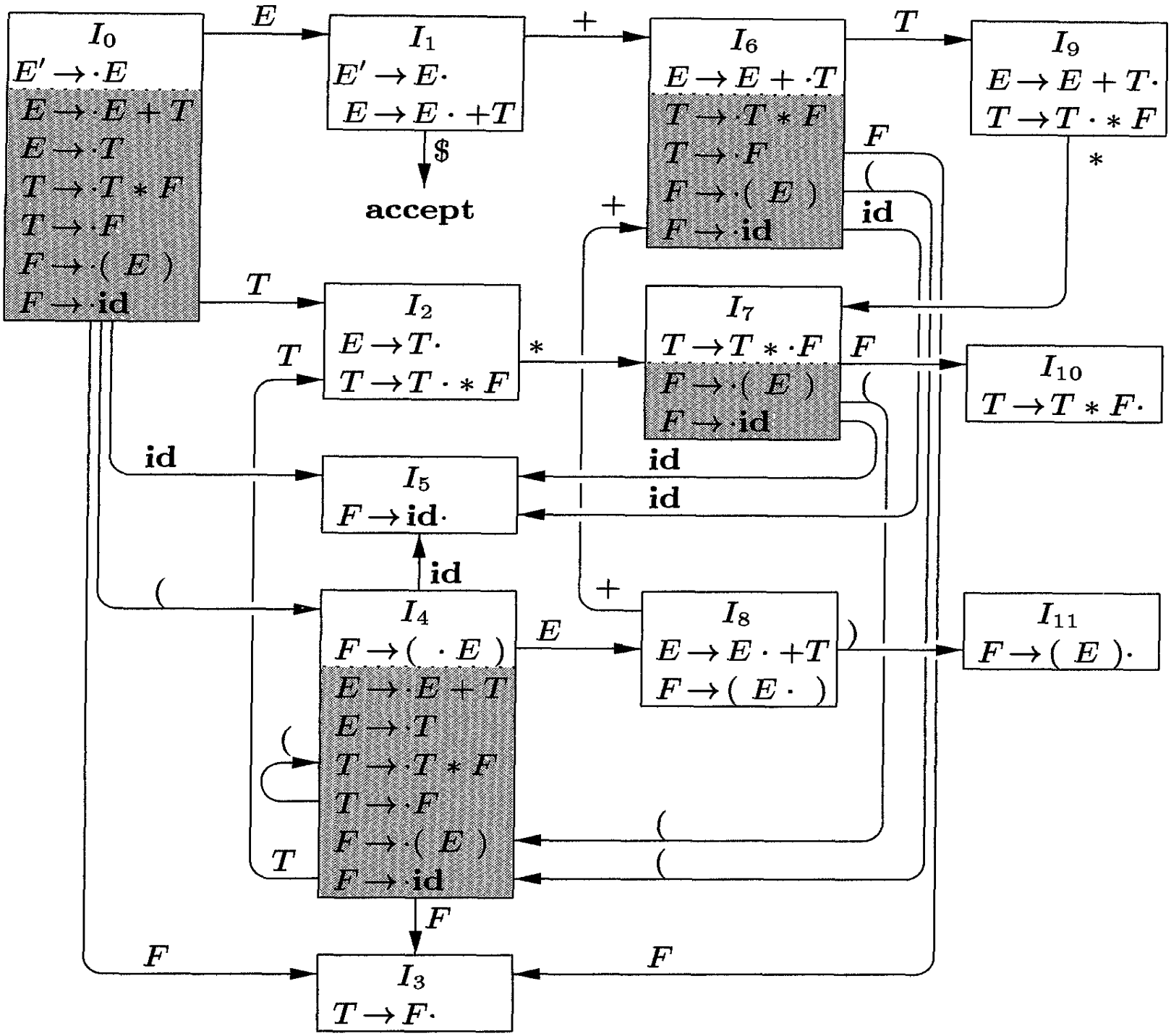
I_7: T → T*.F
F → .(E)
F → .id

I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).


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
$E' \to E \cdot$
$E \to E \cdot + T$
accept
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
$

Constructing SLR Parsing Table

Consider the grammar G'

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

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

3) Create the parsing action table as follows:

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

2) If $A \rightarrow \alpha .$ is in I_i, then action[i,a] is reduce $A \rightarrow \alpha$ for all a in FOLLOW(A), where A <> 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 $goto(I_i,A) = I_j$, then goto[i,A] = j

5) **All other entries will be marked as error**

6) Initial state of the parser contains S' → S

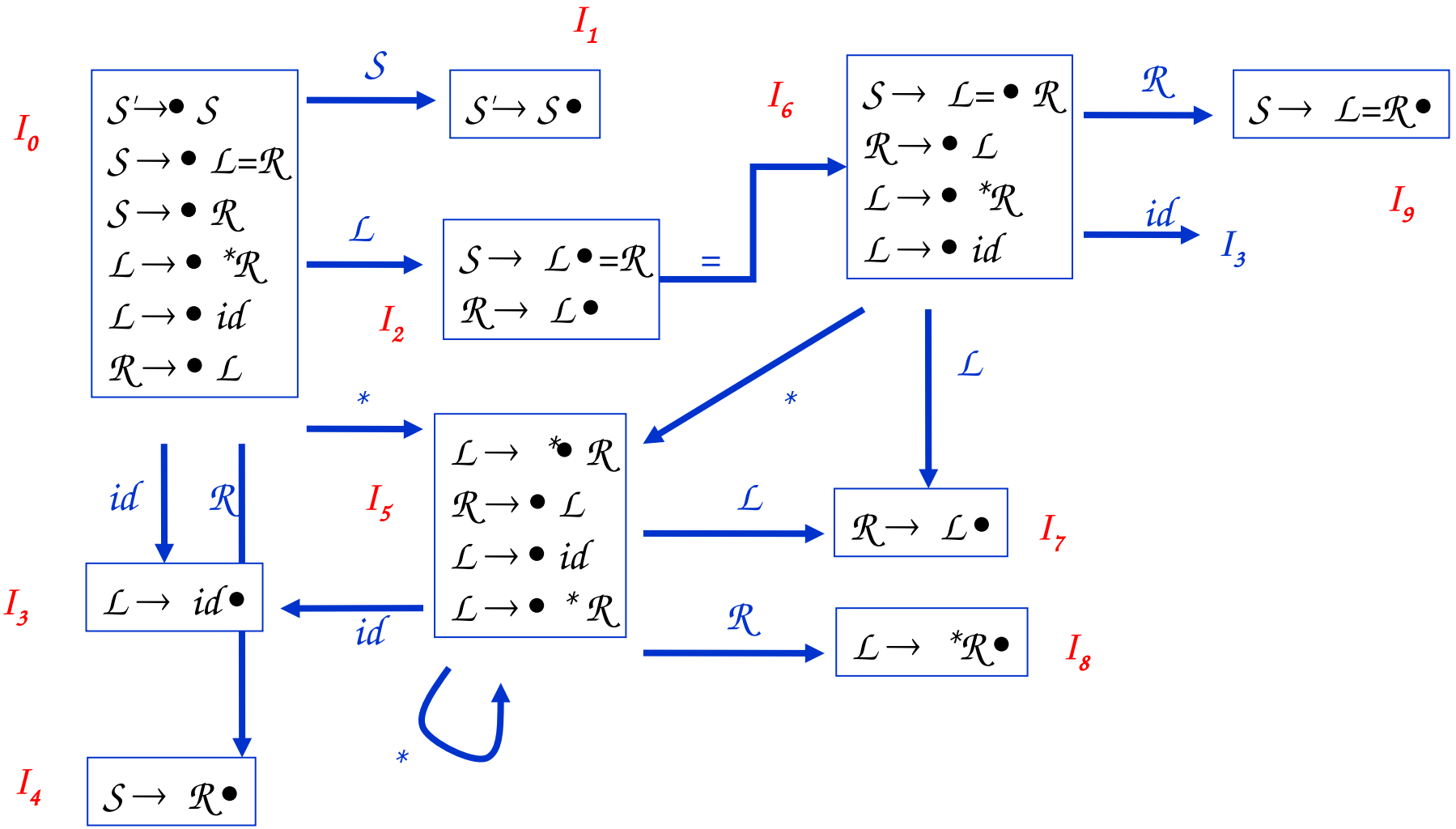
Parsing Tables of Expression Grammar

1. E' → E
2. E → E + T
3. E → T
4. T → T* F
5. T → F
6. F → (E)
7. F → id

	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				

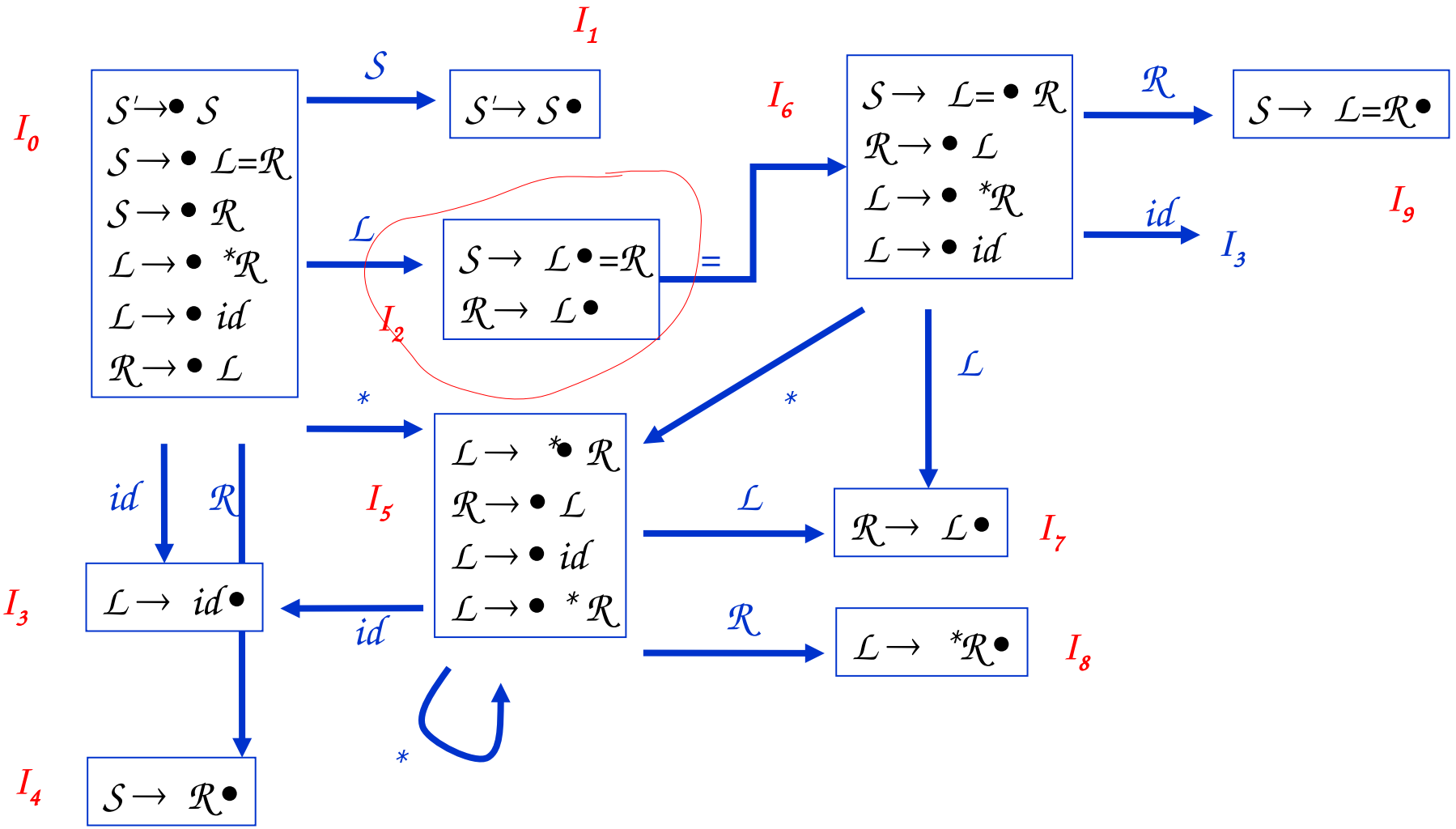
Parsing

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


I0
S′→• S
S→ • L=R
S→ • R
L→ • *R
L→ • id
R→ • L
S
I1
S′→ S•
L
I2
S→ L•=R
R→ L•
=
I6
S→ L=• R
R→ • L
L→ • *R
L→ • id
R
S→ L=R•
I9
id
I3
L
*
*
I5
L→ *• R
R→ • L
L→ • id
L→ • *R
id
R
I3
L→ id•
id
L
R→ L•
I7
R
L→ *R•
I8
*
I4
S→ R•

state	action			goto			
	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)			

state	action			goto			
	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)			


I0
S′→• S
S→ • L=R
S→ • R
L→ • *R
L→ • id
R→ • L
S
I1
S′→ S•
L
I2
S→ L•=R
R→ L•
=
I6
S→ L=• R
R→ • L
L→ • *R
L→ • id
R
S→ L=R•
I9
id
I3
L
*
*
I5
L→ *• R
R→ • L
L→ • id
L→ • *R
id
R
I3
L→ id•
id
L
R→ L•
I7
R
L→ *R•
I8
*
I4
S→ R•

Two possible scenarios

1)S’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *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’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *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’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *id=id

$\mathcal{S}' \rightarrow \mathcal{S}$

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

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

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

$\mathcal{L} \rightarrow \boldsymbol{id}$

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

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

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

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

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

Two possible scenarios

1)S’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *id=id

$\mathcal{S}' \rightarrow \mathcal{S}$

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

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

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

$\mathcal{L} \rightarrow id$

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

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

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

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

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

Two possible scenarios

1)S’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *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’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *id=id

$\mathcal{S}' \rightarrow \mathcal{S}$

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

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

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

$\mathcal{L} \rightarrow id$

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

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

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

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

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

Two possible scenarios

1)S’ => S => L=R => L=L => L=id => id=id

2)S’ => S => L=R => L=L => L=id => *R=id

=> *L=id => *id=id

$$\mathcal{S}' \rightarrow \mathcal{S}$$
$$\mathcal{S} \rightarrow \mathcal{L}=\mathcal{R}$$
$$\mathcal{S} \rightarrow \mathcal{R}$$
$$\mathcal{L} \rightarrow *\mathcal{R}$$
$$\mathcal{L} \rightarrow id$$
$$\mathcal{R} \rightarrow \mathcal{L}$$

$$FOLLOW(S') = \{\$\}$$
$$FOLLOW(S) = \{\$\}$$
$$FOLLOW(L) = \{\$, =\}$$
$$FOLLOW(R) = \{\$, =\}$$

Two possible scenarios

S’ => S => L=R => L=L => L=id => id=id

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

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

S’ => S => L=R => L=L => L=id => *R=id => *L=id => *id=id

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

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

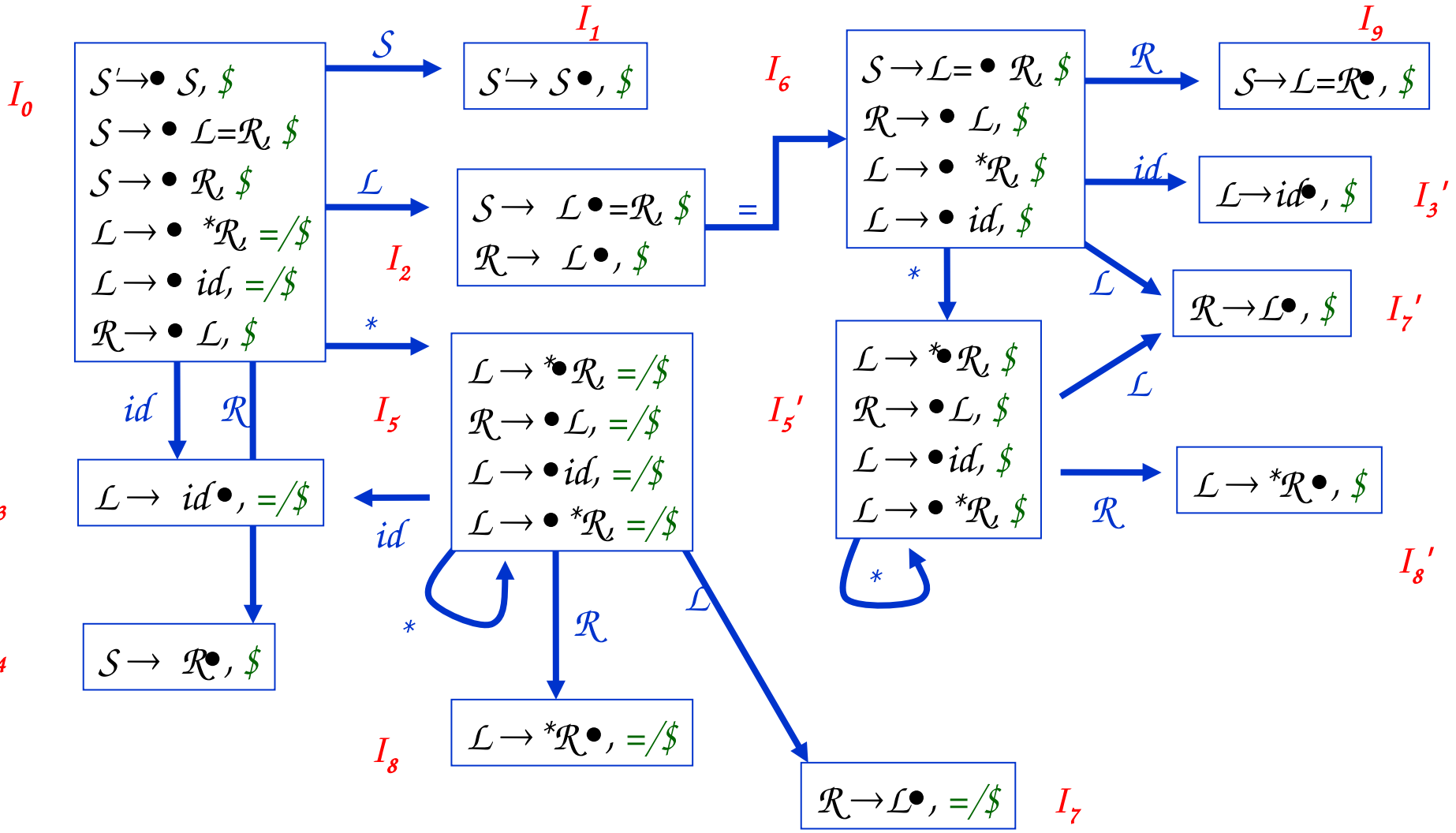
- The conflict occurred because we made a decision about when to reduce based on what token *may* follow a non-terminal at any time.
- However, the fact that a token t may follow a non-terminal N in some derivation does not necessarily imply that t will follow N in some other derivation.
- SLR parsing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


I0
S′→• S, $
S → • L=R, $
S → • R, $
L → • *R, =/$
L → • id, =/$
R → • L, $
S
I1
S′→ S•, $
L
I2
S → L•=R, $
R → L•, $
=
I6
S → L= • R, $
R → • L, $
L → • *R, $
L → • id, $
R
I9
S → L=R•, $
id
L → id•, $
I3′
*
L
R → L•, $
I7′
I5′
L → *•R, $
R → •L, $
L → •id, $
L → • *R, $
L
R
L → *R•, $
I8′
*
id
R
I3
L → id•, =/$
I4
S → R•, $
I5
L → *•R, =/$
R → •L, =/$
L → •id, =/$
L → • *R, =/$
*
id
R
L
I8
L → *R•, =/$
R → L•, =/$
I7

Canonical LR(1) parsing

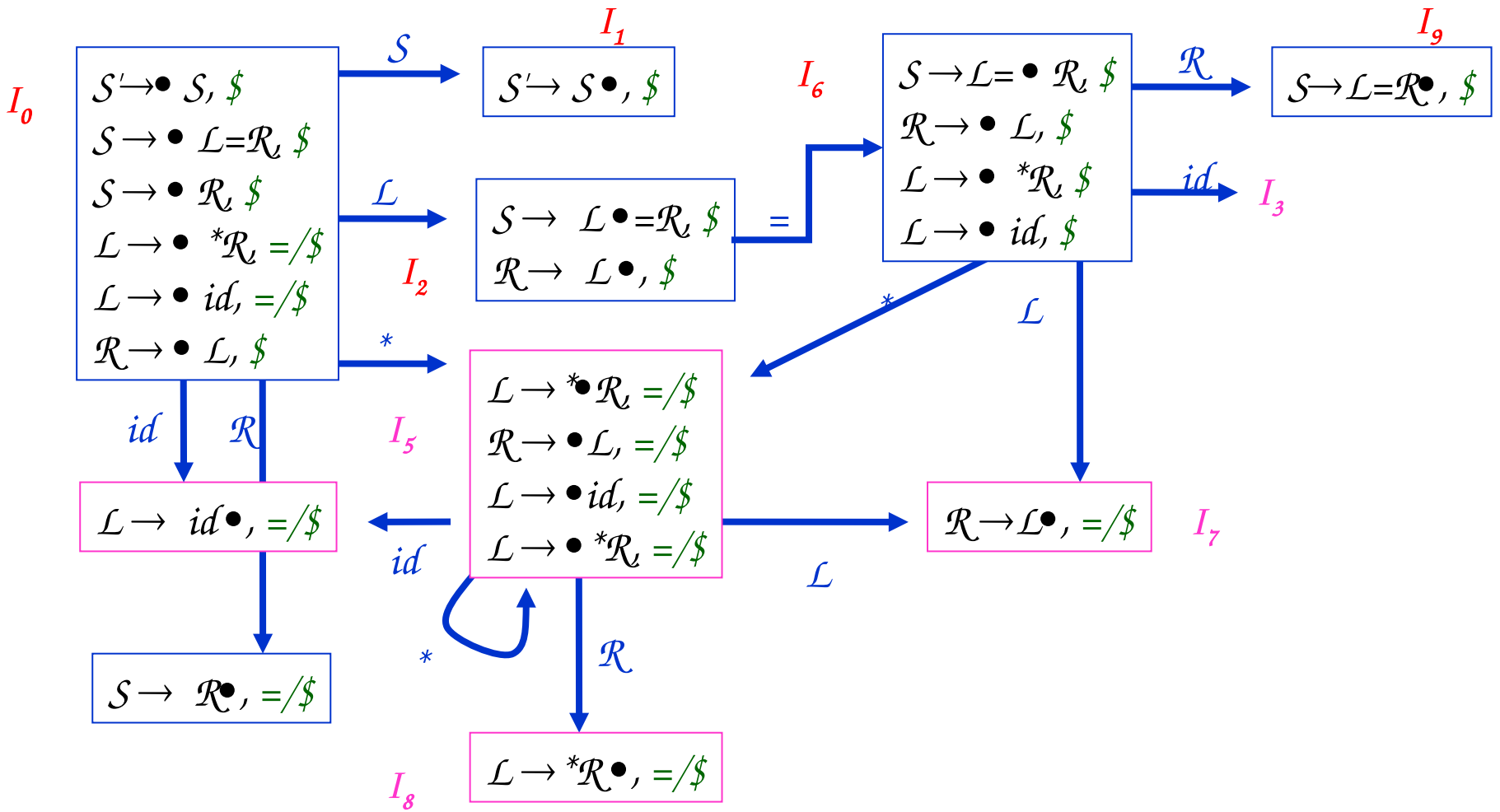
- Rules for creating the table is same as SLR, except we now use the possible lookahead tokens saved in each state, instead of the FOLLOW sets.
- The conflicts that appeared in the SLR parser is now resolved.
- However, the LR(1) parser has many more states. This is not very practical.

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


I0
S'→• S, $
S → • L=R, $
S → • R, $
L → • *R, =/$
L → • id, =/$
R → • L, $
S
I1
S'→ S•, $
L
I2
S → L•=R, $
R → L•, $
=
I6
S → L=• R, $
R → • L, $
L → • *R, $
L → • id, $
R
I9
S → L=R•, $
id
I3
*
L
id
R
I5
L → *•R, =/$
R → •L, =/$
L → •id, =/$
L → • *R, =/$
*
L
id
I7
R → L•, =/$
L → id•, =/$
S → R•, =/$
*
R
I8
L → *R•, =/$

	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: