

# Example (Contd.)

$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid id$

Firststop+(E) = {+, \*, (, id}

Firststop+(T) = {\*, (, id}

Firststop+(F) = {(, id}

Lastop+(E) = {+, \*, ), id}

Lastop+(T) = {\*, ), id}

Lastop+(F) = {), id}

	\$	(	)	id	+	*
\$		<.		<.	<.	<.
(		<.	=	<.	<.	<.
)	.>		.>		.>	.>
id	.>		.>		.>	.>
+	.>	<.	.>	<.	.>	<.
*	.>	<.	.>	<.	.>	.>

# LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with  $k \leq 1$
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers



# LR Parsing Methods

- SLR – Simple LR. Easy to implement, less powerful
- Canonical LR – most general and powerful. Tedious and costly to implement, contains much more number of states compared to SLR
- LALR – Look Ahead LR. Mix of SLR and Canonical LR. Can be implemented efficiently, contains same number of states as simple LR for a grammar

# States of an LR parser

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
  - For  $A \rightarrow XYZ$  we have following items
    - $A \rightarrow .XYZ$
    - $A \rightarrow X.YZ$
    - $A \rightarrow XY.Z$
    - $A \rightarrow XYZ.$
  - In a state having  $A \rightarrow .XYZ$  we hope to see a string derivable from XYZ next on the input.
  - What about  $A \rightarrow X.YZ$ ?

# Constructing canonical LR(0) item sets

- Augmented grammar:
  - $G$  with addition of a production:  $S' \rightarrow S$
- Closure of item sets:
  - If  $I$  is a set of items,  $\text{closure}(I)$  is a set of items constructed from  $I$  by the following rules:
    - Add every item in  $I$  to  $\text{closure}(I)$
    - If  $A \rightarrow \alpha.B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production then add the item  $B \rightarrow \gamma$  to  $\text{closure}(I)$ .
- Example:

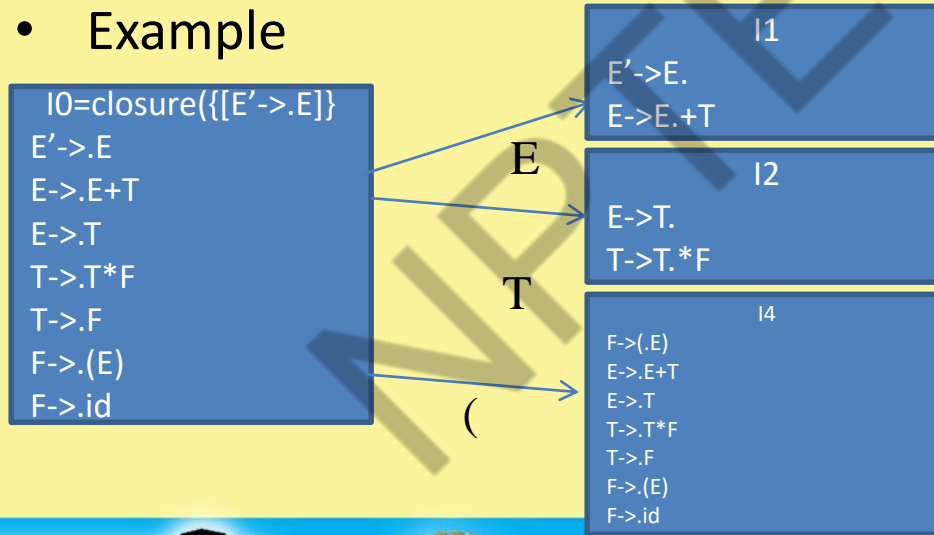
$E' \rightarrow E$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \text{id}$

$I_0 = \text{closure}(\{[E' \rightarrow .E]\})$

$E' \rightarrow .E$   
 $E \rightarrow .E + T$   
 $E \rightarrow .T$   
 $T \rightarrow .T * F$   
 $T \rightarrow .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .\text{id}$

# Constructing canonical LR(0) item sets (cont.)

- Goto ( $I, X$ ) where  $I$  is an item set and  $X$  is a grammar symbol is closure of set of all items  $[A \rightarrow \alpha X \beta]$  where  $[A \rightarrow \alpha.X \beta]$  is in  $I$
- Example



# Closure algorithm

```
SetOfItems CLOSURE(I) {  
    J=I;  
    repeat  
        for (each item  $A \rightarrow \alpha.B\beta$  in J)  
            for (each production  $B \rightarrow \gamma$  of G)  
                if ( $B \rightarrow \gamma$  is not in J)  
                    add  $B \rightarrow \gamma$  to J;  
    until no more items are added to J on one round;  
    return J;  
}
```

# GOTO algorithm

```
SetOfItems GOTO(I,X) {  
    J=empty;  
    if (A- $\rightarrow$   $\alpha$ .X  $\beta$  is in I)  
        add CLOSURE(A- $\rightarrow$   $\alpha$ X.  $\beta$  ) to J;  
    return J;  
}
```

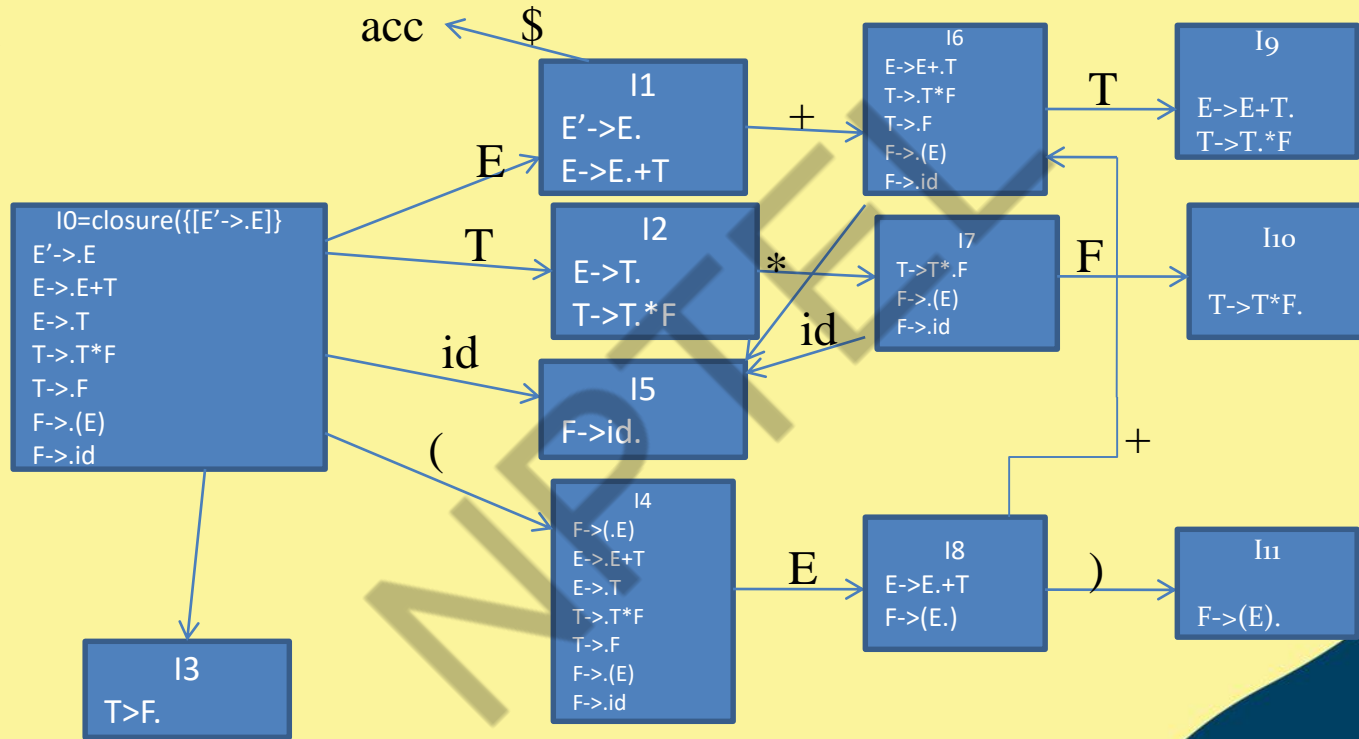


# LR(0) items

```
Void items( $G'$ ) {  
     $C = \text{CLOSURE}(\{[S' \rightarrow .S]\});$   
    repeat  
        for (each set of items  $I$  in  $C$ )  
            for (each grammar symbol  $X$ )  
                if ( $\text{GOTO}(I, X)$  is not empty and not in  $C$ )  
                    add  $\text{GOTO}(I, X)$  to  $C$ ;  
    until no new set of items are added to  $C$  on a round;  
}
```

# Example

$E' \rightarrow E$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid id$

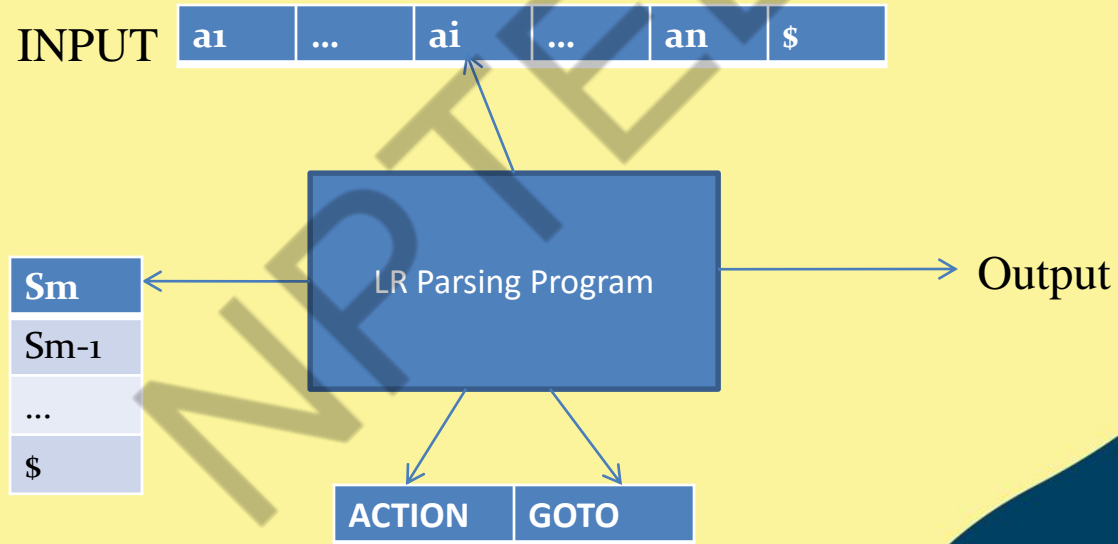


# Use of LR(0) automaton

- Example:  $\text{id}^*\text{id}$

Line	Stack	Symbols	Input	Action
(1)	o	\$	id*id\$	Shift to 5
(2)	o5	\$id	*id\$	Reduce by $F \rightarrow \text{id}$
(3)	o3	\$F	*id\$	Reduce by $T \rightarrow F$
(4)	o2	\$T	*id\$	Shift to 7
(5)	o27	\$T^*	id\$	Shift to 5
(6)	o275	\$T^*\text{id}	\$	Reduce by $F \rightarrow \text{id}$
(7)	o2710	\$T^*F	\$	Reduce by $T \rightarrow T^*F$
(8)	o2	\$T	\$	Reduce by $E \rightarrow T$
(9)	o1	\$E	\$	accept

# LR-Parsing model



# LR parsing algorithm

```
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-> $\beta$ ) {
        pop  $|\beta|$  symbols of the stack;
        let state t now be on top of the stack;
        push GOTO[t,A] onto the stack;
        output the production A-> $\beta$ ;
    } else if (ACTION[s,a]=accept) break; /* parsing is done */
    else call error-recovery routine;
}
```



# Example

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			

Line	Stack	Symbols	Input	Action
(1)	0		id*id+id\$	Shift to 5
(2)	05	id	*id+id\$	Reduce by F->id
(3)	03	F	*id+id\$	Reduce by T->F
(4)	02	T	*id+id\$	Shift to 7
(5)	027	T*	id+id\$	Shift to 5
(6)	0275	T*id	+id\$	Reduce by F->id
(7)	02710	T*F	+id\$	Reduce by T->T*F
(8)	02	T	+id\$	Reduce by E->T
(9)	01	E	+id\$	Shift
(10)	016	E+	id\$	Shift
(11)	0165	E+id	\$	Reduce by F->id
(12)	0163	E+F	\$	Reduce by T->F
(13)	0169	E+T	\$	Reduce by E->E+T
(14)	01	E	\$	accept

- (0)  $E' \rightarrow E$
- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow id$

id\*id+id\$

# Constructing SLR parsing table

- Method
  - Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(0) items for  $G'$
  - State  $i$  is constructed from state  $li$ :
    - If  $[A \rightarrow \alpha.a\beta]$  is in  $li$  and  $\text{Goto}(li, a) = lj$ , then set  $\text{ACTION}[i, a]$  to “shift  $j$ ”
    - If  $[A \rightarrow \alpha.]$  is in  $li$ , then set  $\text{ACTION}[i, a]$  to “reduce  $A \rightarrow \alpha$ ” for all  $a$  in  $\text{follow}(A)$
    - If  $[S' \rightarrow .S]$  is in  $li$ , then set  $\text{ACTION}[i, \$]$  to “Accept”
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If  $\text{GOTO}(li, A) = lj$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made “error”
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S]$



# Example grammar which is not SLR

$S \rightarrow L=R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

I0  
 $S' \rightarrow .S$   
 $S \rightarrow .L=R$   
 $S \rightarrow .R$   
 $L \rightarrow .*R \mid$   
 $L \rightarrow .id$   
 $R \rightarrow .L$

I1  
 $S' \rightarrow S.$

I2  
 $S \rightarrow L.=R$   
 $R \rightarrow L.$

I3  
 $S \rightarrow R.$

I4  
 $L \rightarrow *.R$   
 $R \rightarrow .L$   
 $L \rightarrow .*R$   
 $L \rightarrow .id$

I5  
 $L \rightarrow id.$

I6  
 $S \rightarrow L=.R$   
 $R \rightarrow .L$   
 $L \rightarrow .*R$   
 $L \rightarrow .id$

I7  
 $L \rightarrow *R.$

I8  
 $R \rightarrow L.$

I9  
 $S \rightarrow L=R.$

Action = Shift 6  
 Reduce  $R \rightarrow L$



# More powerful LR parsers

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookaheads are introduced in LR(0) items

# LR(1) Grammar

- A grammar is said to be LR(1) if in a single left-to-right scan, we can construct a reverse rightmost derivation, while using atmost a single token lookahead to resolve ambiguities.
- LR(k) parsers use k token lookahead

# LR(k) item

- A pair  $[\alpha;\beta]$ , where
  - $\alpha$  is a production from  $G$  with a  $.$  at some position in the right hand side
  - $\beta$  is a lookahead string contains  $k$  symbols (terminals or  $\$$ )
- Several LR(1) items may have same core.  $[A \rightarrow X.YZ;a]$  and  $[A \rightarrow X.YZ;b]$  are represented together as  $[A \rightarrow X.YZ;\{a,b\}]$

# Usage of LR(1) Lookahead

- Carry them along to allow choosing correct reduction when there is any choice
- Lookaheads are bookkeeping, unless item has a . at right end
  - In  $[A \rightarrow X.YZ; a]$ ,  $a$  has no direct use
  - In  $[A \rightarrow XYZ.; a]$ ,  $a$  is useful
  - If there are two items  $[A \rightarrow XYZ.; a]$  and  $[B \rightarrow XYZ.; b]$ , we can decide between reducing to  $A$  or  $B$  by looking at limited right context

# Closure algorithm

```
SetOfItems CLOSURE(I) {  
    J=I;  
    repeat  
        for (each item  $[A \rightarrow \alpha.B\beta;a]$  in J)  
            for (each production  $B \rightarrow \gamma$  of G and each terminal  $b$  in  $\text{First}(\beta a)$ )  
                if ( $[B \rightarrow \gamma;b]$  is not in J)  
                    add  $[B \rightarrow \gamma;b]$  to J;  
    until no more items are added to J on one round;  
    return J;  
}
```

# GOTO algorithm

```
SetOfItems GOTO(I,X) {  
    J=empty;  
    if ( $[A \rightarrow \alpha.X\beta;a]$  is in I)  
        add CLOSURE( $[A \rightarrow \alpha.X.\beta;a]$ ) to J;  
    return J;  
}
```

# Constructing LR(1) Parsing Table

- Method
  - Construct  $C=\{I_0, I_1, \dots, I_n\}$ , the collection of LR(1) items for  $G'$
  - State  $i$  is constructed from state  $li$ :
    - If  $[A \rightarrow \alpha.a\beta; b]$  is in  $li$  and  $\text{Goto}(li, a) = lj$ , then set  $\text{ACTION}[i, a]$  to “shift  $j$ ”
    - If  $[A \rightarrow \alpha.; a]$  is in  $li$ , then set  $\text{ACTION}[i, a]$  to “reduce  $A \rightarrow \alpha$ ”
    - If  $[S' \rightarrow S.; \$]$  is in  $li$ , then set  $\text{ACTION}[i, \$]$  to “Accept”
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If  $\text{GOTO}(li, A) = lj$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made “error”
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S; \$]$

# Example LR(1) Parser

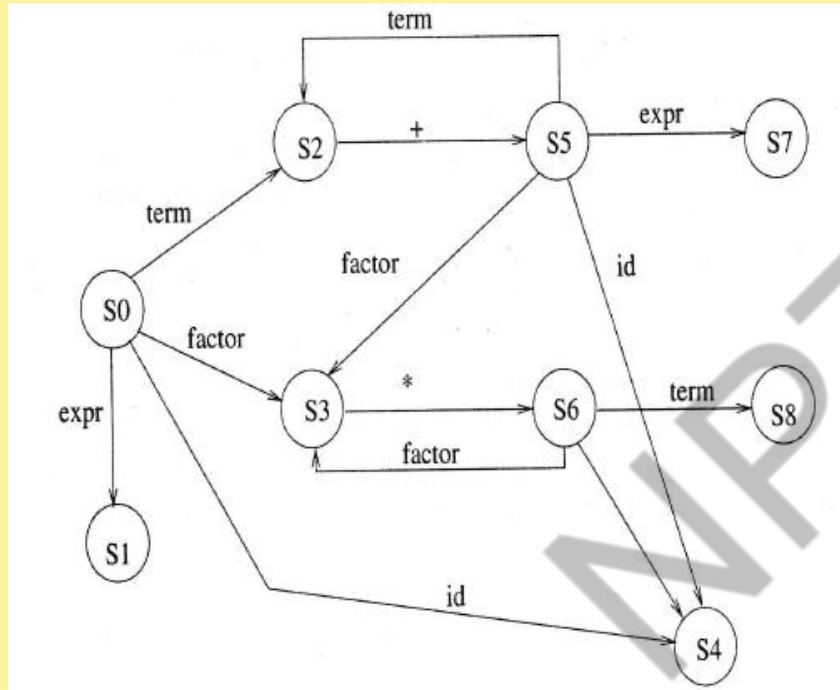
goal  $\rightarrow$  expr  
expr  $\rightarrow$  term + expr  
expr  $\rightarrow$  term  
term  $\rightarrow$  factor \* term  
term  $\rightarrow$  factor  
factor  $\rightarrow$  id

$I_0$  : [goal  $\rightarrow \cdot$ expr, \$], [expr  $\rightarrow \cdot$ term + expr, \$], [expr  $\rightarrow \cdot$ term, \$], [term  $\rightarrow \cdot$ factor \* term, {+, \$}], [term  $\rightarrow \cdot$ factor, {+, \$}], [factor  $\rightarrow$  id, {+, \*, \$}]  
 $I_1$  : [goal  $\rightarrow$  expr, \$]  
 $I_2$  : [expr  $\rightarrow$  term, \$], [expr  $\rightarrow$  term  $\cdot$  + expr, \$]  
 $I_3$  : [term  $\rightarrow$  factor, {+, \$}], [term  $\rightarrow$  factor  $\cdot$  \* term, {+, \$}]  
 $I_4$  : [factor  $\rightarrow$  id, {+, \*, \$}]  
 $I_5$  : [expr  $\rightarrow$  term +  $\cdot$ expr, \$], [expr  $\rightarrow \cdot$ term + expr, \$], [expr  $\rightarrow \cdot$ term, \$], [term  $\rightarrow$  factor \* term, {+, \$}], [term  $\rightarrow \cdot$ factor, {+, \$}], [factor  $\rightarrow \cdot$ id, {+, \*, \$}]  
 $I_6$  : [term  $\rightarrow$  factor \*  $\cdot$ term, {+, \$}], [term  $\rightarrow \cdot$ factor \* term, {+, \$}], [term  $\rightarrow \cdot$ factor, {+, \$}], [factor  $\rightarrow \cdot$ id, {+, \*, \$}]  
 $I_7$  : [expr  $\rightarrow$  term + expr, \$]  
 $I_8$  : [term  $\rightarrow$  factor \* term, {+, \$}]

LR(1) items



# Example LR(1) Parser (Contd.)



	id	+	*	\$	Expr	Term	factor
0	S4				1	2	3
1				Acc			
2		S5		R3			
3		S5	R6	R5			
4		R6	R6	R6			
5	S4				7	2	3
6	S4					8	3
7				R2			
8		R4		R4			

# LALR(1) Parsing

- Reduces number of states in an LR(1) parser
- Merges states differing only in lookahead sets
- SLR and LALR tables have same number of states
- For a C-like language, several hundred states in SLR and LALR parsers, several thousands for LR(1)

# Example

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

LR(1) items

$I_0 : [S' \rightarrow \cdot S, \$], [S \rightarrow \cdot CC, \$], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$

$I_1 : [S' \rightarrow S \cdot, \$]$

$I_2 : [S \rightarrow C \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$

$I_3 : [C \rightarrow c \cdot C, \{c, d\}], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$

$I_4 : [C \rightarrow d \cdot, \{c, d\}]$

$I_5 : [C \rightarrow CC \cdot, \$]$

$I_6 : [C \rightarrow c \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$

$I_7 : [C \rightarrow d \cdot, \$]$

$I_8 : [C \rightarrow cC \cdot, \{c, d\}]$

$I_9 : [C \rightarrow cC \cdot, \$]$

States  $I_4$  and  $I_7$ ,  $I_3$  and  $I_6$ ,  $I_8$  and  $I_9$  can be merged

$I_{47} : [C \rightarrow d \cdot; \{c, d, \$\}]$

$I_{36} : [C \rightarrow c \cdot C; \{c, d, \$\}], [C \rightarrow \cdot c; \{c, d, \$\}],$   
 $[C \rightarrow \cdot d; \{c, d, \$\}]$

$I_{89} : [C \rightarrow cC \cdot; \{c, d, \$\}]$

# LALR Construction – Step-by-Step Approach

- Sets of states constructed as in LR(1) method
- At each point where a new set is spawned, it may be merged with an existing set
- When a new state  $S$  is created, all other states are checked to see if one with the same core exists
- If not,  $S$  is kept; otherwise it is merged with the existing set  $T$  with the same core to form state  $ST$

# Using Ambiguous Grammars

$E \rightarrow E + E$   
 $E \rightarrow E * E$   
 $E \rightarrow (E)$   
 $E \rightarrow id$

$Follow(E) = \{+, *, ), \$\}$

$I0: E' \rightarrow .E$   
 $E \rightarrow .E + E$   
 $E \rightarrow .E * E$   
 $E \rightarrow .(E)$   
 $E \rightarrow .id$

$I2: E \rightarrow (.E)$   
 $E \rightarrow .E + E$   
 $E \rightarrow .E * E$   
 $E \rightarrow .(E)$   
 $E \rightarrow .id$

$I4: E \rightarrow E + .E$   
 $E \rightarrow .E + E$   
 $E \rightarrow .E * E$   
 $E \rightarrow .(E)$   
 $E \rightarrow .id$

$I1: E' \rightarrow E.$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E$

$I3: E \rightarrow id.$

$I5: E \rightarrow E * .E$   
 $E \rightarrow (.E)$   
 $E \rightarrow .E + E$   
 $E \rightarrow .E * E$   
 $E \rightarrow .(E)$   
 $E \rightarrow .id$

$I6: E \rightarrow (E.)$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E$

$I7: E \rightarrow E + .E$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E$

$I8: E \rightarrow E * .E$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E$

$I9: E \rightarrow (E).$

STATE	ACTION						GO TO
	id	+	*	(	)	\$	E
0	S <sub>3</sub>			S <sub>2</sub>			1
1		S <sub>4</sub>	S <sub>5</sub>			Acc	
2	S <sub>3</sub>		S <sub>2</sub>				6
3		R <sub>4</sub>	R <sub>4</sub>		R <sub>4</sub>	R <sub>4</sub>	
4	S <sub>3</sub>			S <sub>2</sub>			7
5	S <sub>3</sub>			S <sub>2</sub>			8
6		S <sub>4</sub>	S <sub>5</sub>				
7		R <sub>1</sub> / S <sub>4</sub>	S <sub>5</sub> / R <sub>1</sub>		R <sub>1</sub>	R <sub>1</sub>	
8		R <sub>2</sub> / S <sub>4</sub>	R <sub>2</sub> / S <sub>5</sub>		R <sub>2</sub>	R <sub>2</sub>	
9		R <sub>3</sub>	R <sub>3</sub>		R <sub>3</sub>	R <sub>3</sub>	

# Error Recovery in LR Parsing

- Undefined entries in LR parsing table means error
- Proper error messages can be flashed to the user
- Error handling routines can be made to modify the parser stack by
  - popping out some entries
  - pushing some desirable entries into the stack
- Brings parser at a descent stage from which it can proceed further
- Enables detection of multiple errors and flashing them to the user for correction



# Error Recovery – Example

$E' \rightarrow E$

$E \rightarrow E + E \mid E * E \mid id$

$Follow(E) = \{+, *, \$\}$

10:  $\{[E' \rightarrow E], [E \rightarrow E + E], [E \rightarrow E * E], [E \rightarrow id]\}$

11:  $\{[E' \rightarrow E], [E \rightarrow E + E], [E \rightarrow E * E]\}$

12:  $\{[E \rightarrow id]\}$

13:  $\{[E \rightarrow E + E], [E \rightarrow E + E], [E \rightarrow E * E], [E \rightarrow id]\}$

14:  $\{[E \rightarrow E * E], [E \rightarrow E + E], [E \rightarrow E * E], [E \rightarrow id]\}$

15:  $\{[E \rightarrow E + E], [E \rightarrow E + E], [E \rightarrow E * E]\}$

16:  $\{[E \rightarrow E + E], [E \rightarrow E + E], [E \rightarrow E * E]\}$

17:  $\{[E \rightarrow E * E], [E \rightarrow E + E], [E \rightarrow E * E]\}$

State	ACTION				GOTO
	id	+	*	\$	E
0	S2	e1	e1	e1	1
1	e2	S3	S4	Acc	
2	e2	R3	R3	R3	
3	S2	e1	e1	e1	5
4	S2	e1	e1	e1	6
5	e2	R1	S4	R1	
6	e2	R2	R2	R2	

e1: Seen operator or end of string  
while expecting id

e2: Seen id while expecting  
operator