40-414 Compiler Design

Bottom-Up Parsing-II

Lecture 9

LR (k) parsing (Revisited)

- SLR – Simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (Look-Ahead LR)
- SLR, LR, and LALR work exactly the same; only their parse tables are different.

LR(1) Grammar

Canonical sets of LR(1) items

- Number of states much larger than in the SLR construction
  LR(1) = Order of thousands for a standard prog. Lang.
  SLR(1) = Order of hundreds for a standard prog. Lang.

LALR(1) (lookahead-LR)

- A tradeoff:
  - Collapse states of the LR(1) table that have the same *core* (the "LR(0)" part of each state)
  - LALR never introduces a Shift/Reduce Conflict if LR(1) doesn't.
  - It might introduce a Reduce/Reduce Conflict (that did not exist in the LR(1))
  - Still much better than SLR(1) (larger set of languages)
  - but smaller than LR(1)

Conflict Example

Grammar	I_0		I_6	I_9
$S \rightarrow L=R$	I_0: $S' \rightarrow .S$	I_1: $S' \rightarrow S.$	I_6: $S \rightarrow L=.R$	I_9: $S \rightarrow L=R.$
$S \rightarrow R$	$S \rightarrow .L=R$		$R \rightarrow .L$	
$L \rightarrow *R$	$S \rightarrow .R$	I_2: $S \rightarrow L.=R$	$L \rightarrow .*R$	
$L \rightarrow id$	$L \rightarrow .*R$	$R \rightarrow L.$	$L \rightarrow .id$	
$R \rightarrow L$	$L \rightarrow .id$			
	$R \rightarrow .L$	I_3: $S \rightarrow R.$		
		I_4: $L \rightarrow *.R$	I_7: $L \rightarrow *R.$	
		$R \rightarrow .L$		
		$L \rightarrow .*R$	I_8: $R \rightarrow L.$	
		$L \rightarrow .id$		
		I_5: $L \rightarrow id.$		

Problem

FOLLOW(R)={=,$}

= → shift 6

= → reduce by $R \rightarrow L$

shift/reduce conflict

Conflict Example

$S \to AaAb$

$S \to BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

I_0: $S' \to .S$

$S \to .AaAb$

$S \to .BbBa$

$A \to .$

$B \to .$

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by $A \to \varepsilon$

a → reduce by $B \to \varepsilon$

reduce/reduce conflict

b → reduce by $A \to \varepsilon$

b → reduce by $B \to \varepsilon$

reduce/reduce conflict

LookAhead vesus Follow

In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:

if the $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)

In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

Grammar:

1) $S \rightarrow AaAb$
2) $S \rightarrow BbBa$
3) $A \rightarrow \varepsilon$
4) $B \rightarrow \varepsilon$

RMD: 1 3 3

$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$

Parse (Reverse of RMD):

$Aab \Rightarrow \varepsilon\ ab$ (correct follow is a)

$AaAb \Rightarrow Aa\ \varepsilon\ b$ (correct follow is b)

RMD: 2 4 4

$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$

Parse (Reverse of RMD):

$Bba \Rightarrow \varepsilon\ ba$

$BbBa \Rightarrow Bb\ \varepsilon\ a$

LR(1) item

To avoid some of invalid reductions, the states need to carry more information.

Extra information is put into a state by including a terminal symbol as a second component in an item.

A LR(1) item is:

$A \rightarrow \alpha . \beta, a$ where **a** is the lookahead of the LR(1) item

(**a** is a terminal or end-marker.)

LR(1) items

- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the lookahead a does not have any affect.
- When β is empty ($A \rightarrow \alpha.,a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain $A \rightarrow \alpha.,a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

  ...

  $A \rightarrow \alpha.,a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that ***closure*** and ***goto*** operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha . B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of *G*; then $B \rightarrow . \gamma, b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

      for each I in ***C*** and each grammar symbol X

          if goto(I,X) is not empty and not in ***C***

              add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \to \alpha.\beta, a_1$$

$$\ldots$$

$$A \to \alpha.\beta, a_n$$

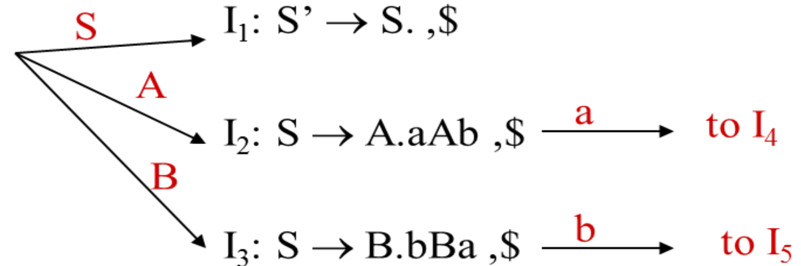
can be written as

$$A \to \alpha.\beta, \{a_1, a_2, \ldots, a_n\}$$

Canonical LR(1) Collection - Example

$S \to AaAb$
$S \to BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

I_0: $S' \to .S ,\$$
$S \to .AaAb ,\$$
$S \to .BbBa ,\$$
$A \to . ,a$
$B \to . ,b$



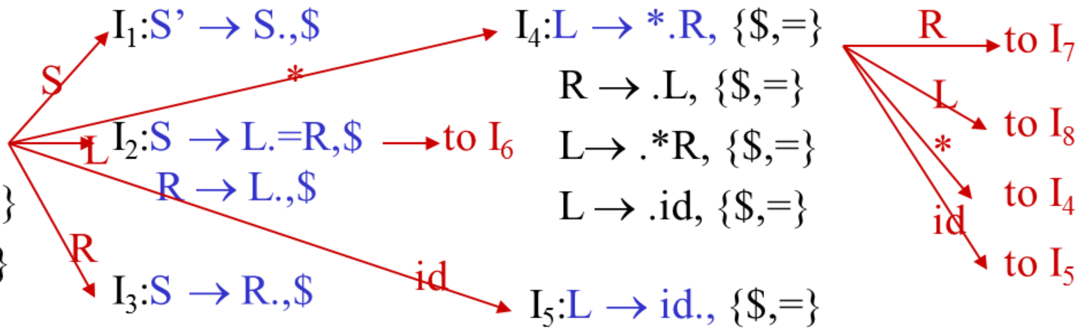
I_4: $S \to Aa.Ab ,\$$ $\xrightarrow{A}$ I_6: $S \to AaA.b ,\$$ $\xrightarrow{a}$ I_8: $S \to AaAb. ,\$$
$A \to . ,b$

I_5: $S \to Bb.Ba ,\$$ $\xrightarrow{B}$ I_7: $S \to BbB.a ,\$$ $\xrightarrow{b}$ I_9: $S \to BbBa. ,\$$
$B \to . ,a$

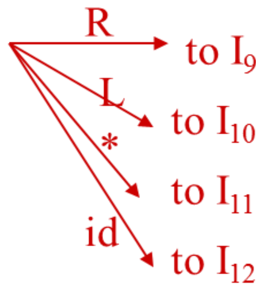
Canonical LR(1) Collection – Example 2

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_6:S → L=.R,$
R → .L,$
L → .*R,$
L → .id,$



to I_9
to I_{10}
to I_{11}
to I_{12}

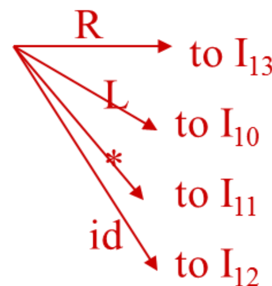
I_7:L → *R., {$,=}

I_8: R → L., {$,=}

I_9:S → L=R.,$

I_{10}:R → L.,$

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



to I_{13}
to I_{10}
to I_{11}
to I_{12}

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}

Construction of LR(1) Parsing Tables

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

   $C \leftarrow \{I_0,\dots,I_n\}$

- 2. Create the parsing action table as follows
  - If a is a terminal, $A \rightarrow \alpha \bullet a\beta, b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
  - If $A \rightarrow \alpha \bullet, a$ is in I_i , then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ where $A \neq S'$.
  - If $S' \rightarrow S \bullet, \$$ is in I_i , then action[i,$] is ***accept***.
  - If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S,\$$

LR(1) Parsing Tables

	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			

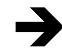
no shift/reduce or
no reduce/reduce conflict

so, it is a LR(1) grammar

LALR(1) Parsing Tables

- **LALR** stands for **LookAhead LR**.
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***YACC*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

LALR(1) Parsing Tables

Canonical LR(1) Parser  LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)

- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L\bullet=R,\$$ ➔ $S \rightarrow L\bullet=R$ (Core)

$R \rightarrow L\bullet,\$$ $R \rightarrow L\bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1:$L \rightarrow id\bullet,=$

➔ A new state: I_{12}: $L \rightarrow id\bullet,\{=,\$\}$

I_2:$L \rightarrow id\bullet,\$$ (have same core, merge the lookaheads)

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.

- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$A \rightarrow \alpha \bullet ,a$ and $B \rightarrow \beta \bullet a\gamma ,b$

- This means that a state of the canonical LR(1) parser must have:

$A \rightarrow \alpha \bullet ,a$ and $B \rightarrow \beta \bullet a\gamma ,c$

But, this state has also a shift/reduce conflict.

i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

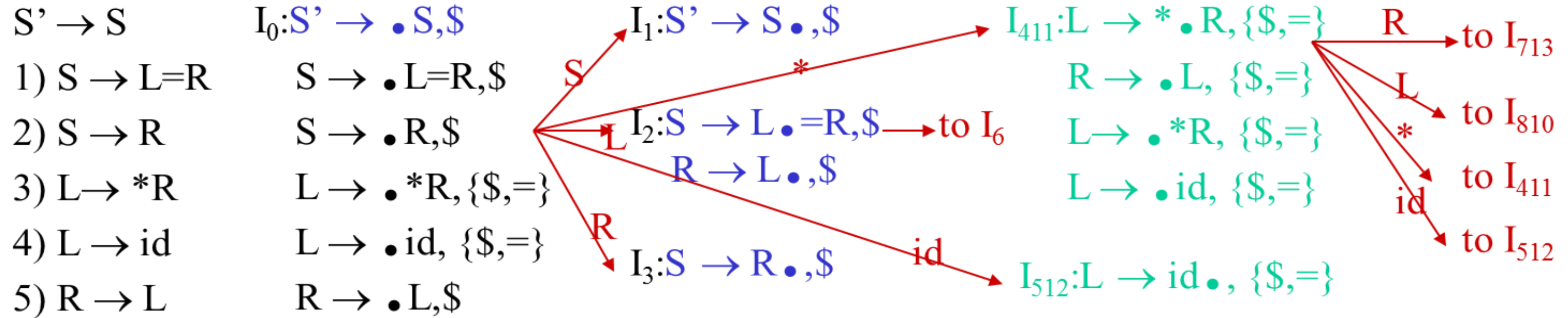
$I_1 : A \rightarrow \alpha \bullet ,a$ $\quad$ $I_2 : A \rightarrow \alpha \bullet ,b$

$\quad B \rightarrow \beta \bullet ,b$ $\quad$ $B \rightarrow \beta \bullet ,c$

$I_{12} : A \rightarrow \alpha \bullet , \{a,b\}$ ➔ reduce/reduce conflict

$\quad B \rightarrow \beta \bullet ,\{b,c\}$

Canonical LALR(1) Collection



I_6: $S \to L = \bullet R, \$$ — R: to I_9; L: to I_{810}; *: to I_{411}; id: to I_{512}

$R \to \bullet L, \$$

$L \to \bullet {*}R, \$$

$L \to \bullet id, \$$

I_9: $S \to L = R \bullet, \$$

I_{713}: $L \to {*}R \bullet, \{\$,=\}$

I_{810}: $R \to L \bullet, \{\$,=\}$

Same Cores

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parse Table

	id	*	=	$	S	L	R
0	s512	s411			1	2	3
1				acc			
2			s6	r5			
3				r2			
411	s512	s411				810	713
512			r4	r4			
6	s512	s411				810	9
713			r3	r3			
810			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

so, it is a LALR(1) grammar

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.

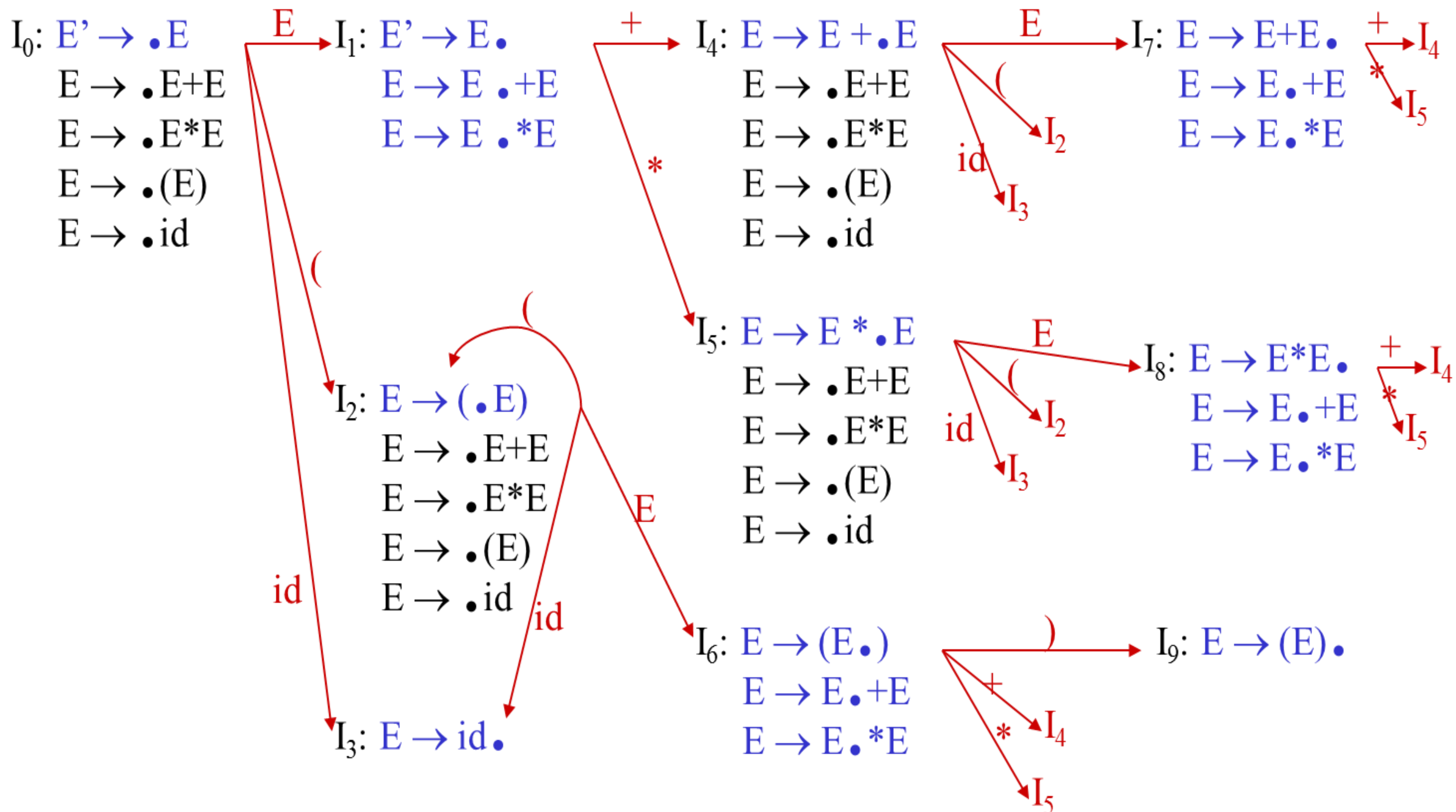
- Can we create LR-parsing tables for ambiguous grammars?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.

- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

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

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

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $,+,*,) }

State I_7 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

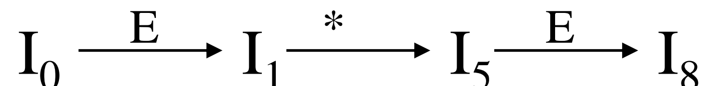
when current token is +
- shift ➔ + is right-associative
- reduce ➔ + is left-associative

when current token is *
- shift ➔ * has higher precedence than +
- reduce ➔ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_8 has shift/reduce conflicts for symbols + and *.



when current token is *
- shift → * is right-associative
- reduce → * is left-associative

when current token is +
- shift → + has higher precedence than *
- reduce → * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	Action							Goto
	id	**+**	*****	**(**	**)**	**$**		**E**
0	s3			s2				1
1		s4	s5			acc		
2	s3			s2				6
3		r4	r4		r4	r4		
4	s3			s2				7
5	s3			s2				8
6		s4	s5		s9			
7		r1	s5		r1	r1		
8		r2	r2		r2	r2		
9		r3	r3		r3	r3		

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).

- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - - The symbol **a** is simply in FOLLOW(A), but this may not work for all situations.

- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.

- This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
- - stmt, expr, block, ...

Phrase/Local Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - - missing operand
  - - unbalanced right parenthesis