Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.

- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

  $S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

  $\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.

  - At each shift action, the current symbol in the input string is pushed to a stack.

  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.

  - There are also two more actions: accept and error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string   ➔   the starting symbol

      reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.

- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

  Rightmost Derivation: $S \overset{*}{\underset{rm}{\Rightarrow}} \omega$

  Shift-Reduce Parser finds: $\omega \underset{rm}{\Leftarrow} \ldots \underset{rm}{\Leftarrow} S$

Shift-Reduce Parsing -- Example

S → aABb   input string: aaabb

A → aA | a   aaAbb

B → bB | b   aAbb   ⇓ reduction

                                           aABb

                                           S

$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

3

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.

    - But not every substring matches the right side of a production rule is handle

- A **handle** of a right sentential form $\gamma \ (\equiv \alpha\beta\omega)$ is

    a production rule $A \rightarrow \beta$ and a position of γ

    where the string β may be found and replaced by A to produce

    the previous right-sentential form in a rightmost derivation of γ.

$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A\omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S = \gamma_0 \xrightarrow[rm]{} \gamma_1 \xrightarrow[rm]{} \gamma_2 \xrightarrow[rm]{} \cdots \xrightarrow[rm]{} \gamma_{n-1} \xrightarrow[rm]{} \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$ Right-Most Derivation of `id+id*id`

$T \rightarrow T*F \mid F$ $E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

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

Right-Most Sentential Form	Reducing Production
`id+id*id`	$F \rightarrow id$
`F+id*id`	$T \rightarrow F$
`T+id*id`	$E \rightarrow T$
`E+id*id`	$F \rightarrow id$
`E+F*id`	$T \rightarrow F$
`E+T*id`	$F \rightarrow id$
`E+T*F`	$T \rightarrow T*F$
`E+T`	$E \rightarrow E+T$
`E`	

`Handles` are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-parser action:

  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

# A Stack Implementation of A Shift-Reduce Parser

Stack	Input	Action
$	id+id*id$ shift	
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Parse Tree

E 8
├─ E 3
│   └─ T 2
│   └─ F 1
│   └─ id
├─ +
└─ T 7
    ├─ T 5
    │   └─ F 4
    │   └─ id
    ├─ *
    └─ F 6
        └─ id
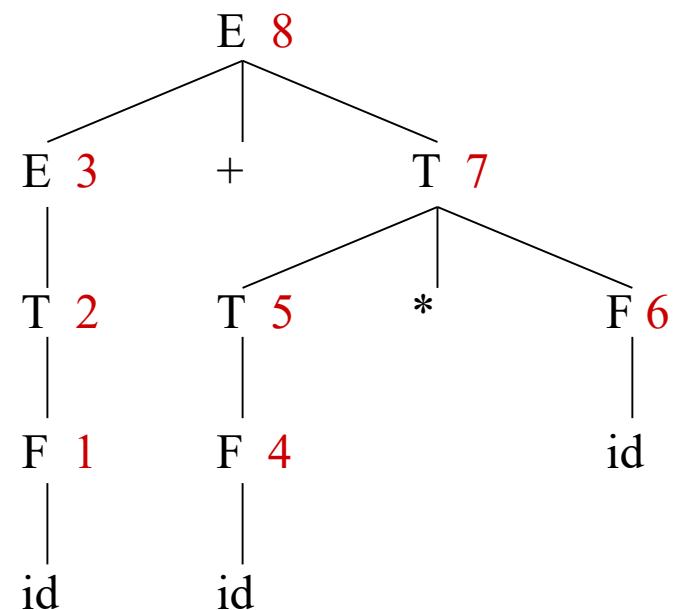
Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.

- Stack contents and the next input symbol may not decide action:
    - **shift/reduce conflict**: Whether make a shift operation or a reduction.
    - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.

- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

  left to right   right-most   k lookhead
  scanning   derivation
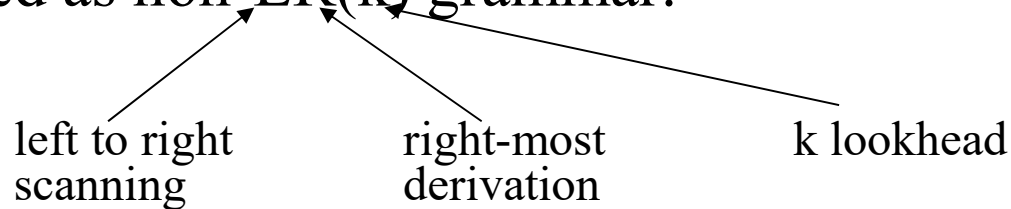
- An ambiguous grammar can never be a LR grammar.
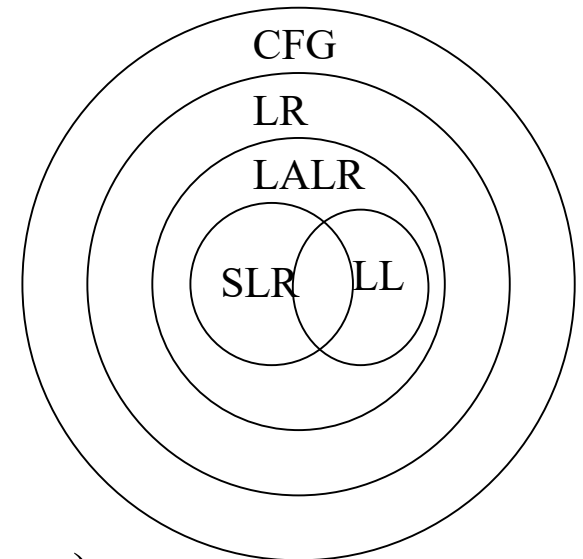
Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.

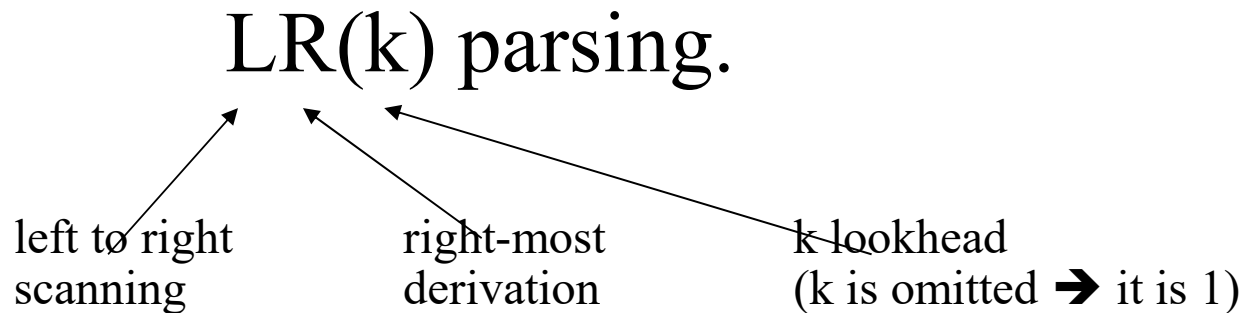2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - LR – most general LR parser
     - LALR – intermediate LR parser (lookhead LR parser)
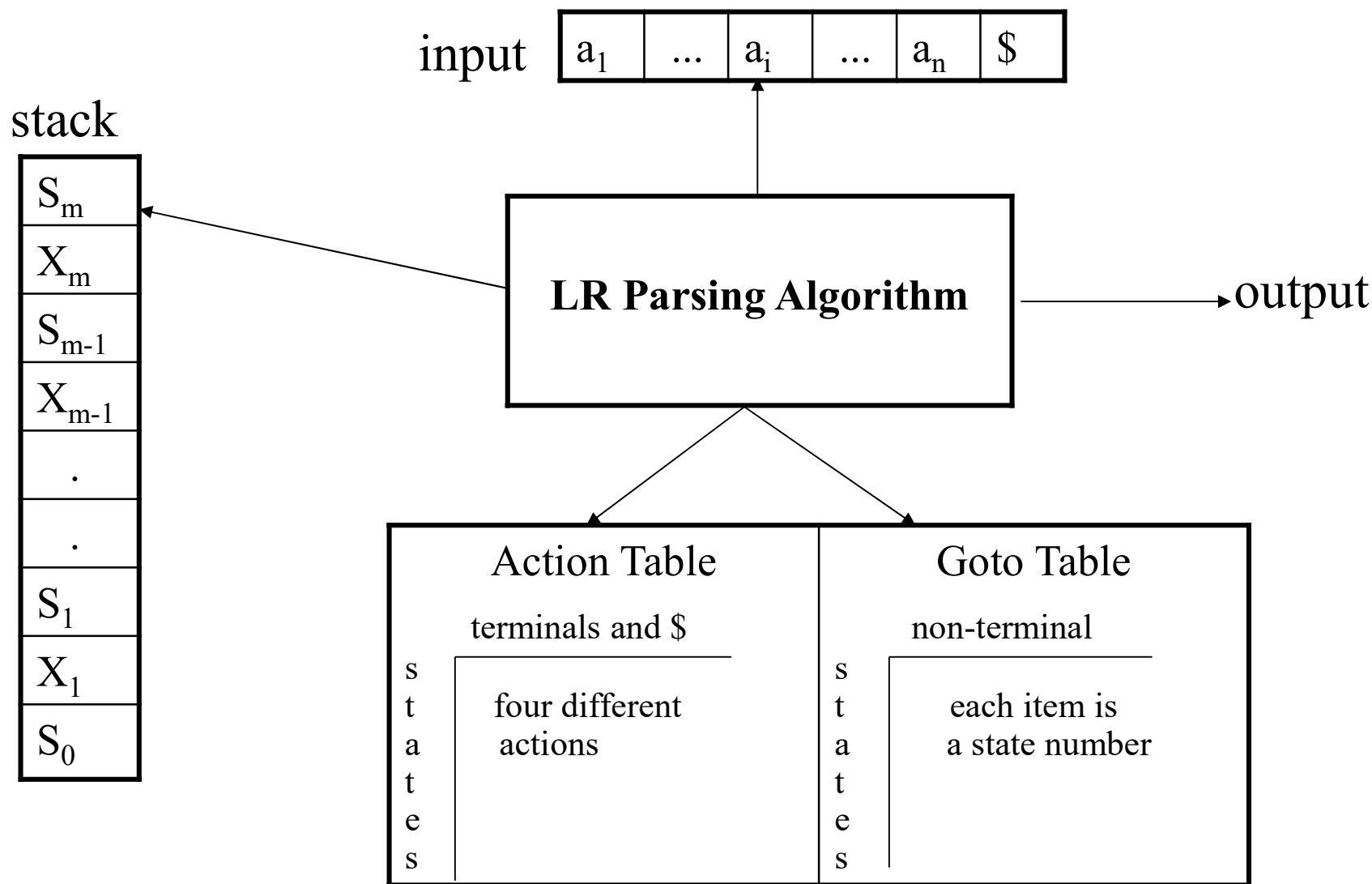   - SLR, LR and LALR work same, only their parsing tables are different.

CFG
LR
LALR
SLR LL

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

left to right   right-most   k lookhead
scanning   derivation   (k is omitted ➜ it is 1)

- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
    $$LL(1)\text{-Grammars} \subset LR(1)\text{-Grammars}$$
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsing Algorithm

input | a_1 | ... | a_i | ... | a_n | $ |

stack

| S_m |
| X_m |
| S_{m-1} |
| X_{m-1} |
| . |
| . |
| S_1 |
| X_1 |
| S_0 |

LR Parsing Algorithm

output

Action Table	Goto Table
terminals and $	non-terminal
s t a t e s	four different actions	s t a t e s	each item is a state number

A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(S_o\ X_1\ S_1\ ...\ X_m\ S_m,\quad a_i\ a_{i+1}\ ...\ a_n\ \$\ )$$

   Stack   Rest of Input
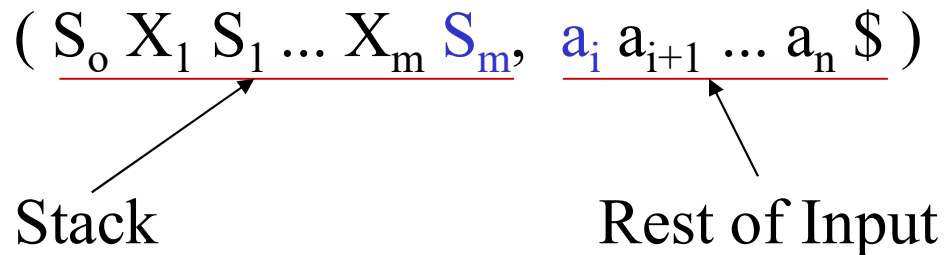
- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1\ ...\ X_m\ a_i\ a_{i+1}\ ...\ a_n\ \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) \Rightarrow (S_o X_1 S_1 ... X_m S_m \, a_i \, s, a_{i+1} ... a_n \$)$

2. **reduce A$\rightarrow\beta$** (or **rn** where n is a production number)
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s_{m-r},A]**

   $(S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) \Rightarrow (S_o X_1 S_1 ... X_{m-r} S_{m-r} A \, s, a_i ... a_n \$)$

   - Output is the reducing production reduce A$\rightarrow\beta$

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop $2|\beta|$ (=r) items from the stack; let us assume that $\beta = Y_1 Y_2 ... Y_r$
- then push **A** and **s** where **s=goto[s_{m-r},A]**

$(S_o X_1 S_1 ... X_{m-r} S_{m-r} Y_1 S_{m-r} ... Y_r S_m, a_i a_{i+1} ... a_n \$)$

$\quad \Rightarrow (S_o X_1 S_1 ... X_{m-r} S_{m-r} A s, a_i ... a_n \$)$

- In fact, $Y_1 Y_2 ... Y_r$ is a handle.

$X_1 ... X_{m-r} A a_i ... a_n \$ \Rightarrow X_1 ... X_m Y_1 ... Y_r a_i a_{i+1} ... a_n \$$

(SLR) Parsing Tables for Expression Grammar

Action Table   Goto Table

1) $E \rightarrow E+T$

2) $E \rightarrow T$

3) $T \rightarrow T*F$

4) $T \rightarrow F$

5) $F \rightarrow (E)$

6) $F \rightarrow id$

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				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: $A \rightarrow \bullet aBb$

  (four different possibility) $A \rightarrow a \bullet Bb$

                                      $A \rightarrow aB \bullet b$

                                      $A \rightarrow aBb \bullet$

- Sets of LR(0) items will be the states of action and goto table of the SLR parser.

- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.

- *Augmented Grammar*:

  G' is G with a new production rule S'$\rightarrow$S where S' is the new starting symbol.

The Closure Operation

- If I is a set of LR(0) items for a grammar G, then **closure(I)** is the set of LR(0) items constructed from I by the two rules:

  1. Initially, every LR(0) item in I is added to closure(I).
  2. If A $\rightarrow \alpha \bullet B\beta$ is in closure(I) and B$\rightarrow \gamma$ is a production rule of G; then B$\rightarrow \bullet \gamma$ will be in the closure(I).
     We will apply this rule until no newer LR(0) items can be added to closure(I).

The Closure Operation -- Example

E' → E   closure({E' → •E}) =

E → E+T   { E' → •E   ⟵—————— kernel items

E → T   E → •E+T

T → T*F   E → •T

T → F   T → •T*F

F → (E)   T → •F

F → id   F → •(E)

                                    F → •id   }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I
    then every item in **closure($\{A \rightarrow \alpha X \bullet \beta\}$)** will be in goto(I,X).

Example:

I ={ 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 $\rightarrow$ $\bullet$ id }

goto(I,E) = { E' $\rightarrow$ E $\bullet$, E $\rightarrow$ E $\bullet$ +T }

goto(I,T) = { E $\rightarrow$ T $\bullet$, T $\rightarrow$ T $\bullet$ *F }

goto(I,F) = {T $\rightarrow$ F $\bullet$ }

goto(I,() = { F $\rightarrow$ ($\bullet$ E), E $\rightarrow$ $\bullet$ E+T, E $\rightarrow$ $\bullet$ T, T $\rightarrow$ $\bullet$ T*F, T $\rightarrow$ $\bullet$ F,

            F $\rightarrow$ $\bullet$ (E), F $\rightarrow$ $\bullet$ id }

goto(I,id) = { F $\rightarrow$ id $\bullet$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- *Algorithm*:

    C is { closure({S'→ • S}) }

    repeat the followings until no more set of LR(0) 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.

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
  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).

Transition Diagram (DFA) of Goto Function

I_0 $\xrightarrow{E}$ I_1 $\xrightarrow{+}$ I_6 $\xrightarrow{T}$ I_9 $\xrightarrow{*}$ to I_7

I_6 $\xrightarrow{F}$ to I_3

I_6 $\xrightarrow{(}$ to I_4

I_6 $\xrightarrow{id}$ to I_5

I_0 $\xrightarrow{T}$ I_2 $\xrightarrow{*}$ I_7 $\xrightarrow{F}$ I_{10}

I_7 $\xrightarrow{(}$ to I_4

I_7 $\xrightarrow{id}$ to I_5

I_0 $\xrightarrow{F}$ I_3

I_0 $\xrightarrow{(}$ I_4 $\xrightarrow{E}$ I_8 $\xrightarrow{)}$ I_{11}

I_8 $\xrightarrow{+}$ to I_6

I_0 $\xrightarrow{id}$ I_5

I_4 $\xrightarrow{id}$ I_5

I_4 $\xrightarrow{T}$ to I_2

I_4 $\xrightarrow{F}$ to I_3

I_4 $\xrightarrow{(}$ to I_4

Constructing SLR Parsing Table
(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $$C \leftarrow \{I_0,...,I_n\}$$

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

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				

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.

- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).

- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule `i` or `j` for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

I_0: $S' \rightarrow .S$

$S \rightarrow .L=R$

$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .id$

$R \rightarrow .L$

I_1: $S' \rightarrow S.$

I_2: $S \rightarrow L.=R$

$R \rightarrow L.$

I_3: $S \rightarrow R.$

I_6: $S \rightarrow L=.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .id$

I_9: $S \rightarrow L=R.$

I_4: $L \rightarrow *.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .id$

I_7: $L \rightarrow *R.$

I_8: $R \rightarrow L.$

Problem

FOLLOW(R)={=,$}

= shift 6

reduce by $R \rightarrow L$

shift/reduce conflict

I_5: $L \rightarrow id.$

Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$I_0: S' \rightarrow .S$

$\quad S \rightarrow .AaAb$

$\quad S \rightarrow .BbBa$

$\quad A \rightarrow .$

$\quad B \rightarrow .$

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a $\longrightarrow$ reduce by $A \rightarrow \varepsilon$

$\quad\searrow$ reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b $\longrightarrow$ reduce by $A \rightarrow \varepsilon$

$\quad\searrow$ reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

Constructing Canonical LR(1) Parsing Tables

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

  - if the $A \rightarrow \alpha \bullet$ 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.

$S \rightarrow AaAb$ $S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$ $S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$ $Aab \Rightarrow \varepsilon\ ab$ $Bba \Rightarrow \varepsilon\ ba$

$B \rightarrow \varepsilon$ $AaAb \Rightarrow Aa\ \varepsilon\ b$ $BbBa \Rightarrow Bb\ \varepsilon\ a$

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:

$$A \rightarrow \alpha \bullet \beta, a$$

where **a** is the look-head of the LR(1) item

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

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha \bullet \beta, a$) is not empty, the look-head does not have any affect.

- When β is empty $(A \rightarrow \alpha \bullet, a)$, 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 $A \rightarrow \alpha \bullet, a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

$$...$$

$$A \rightarrow \alpha \bullet, a_n$$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are like 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

$$A \rightarrow \alpha \cdot \beta, a_1$$

$$...$$

$$A \rightarrow \alpha \cdot \beta, a_n$$

can be written as

$$A \rightarrow \alpha \cdot \beta, a_1/a_2/.../a_n$$

Canonical LR(1) Collection -- Example

$S \rightarrow AaAb$

$S \rightarrow BbBa$
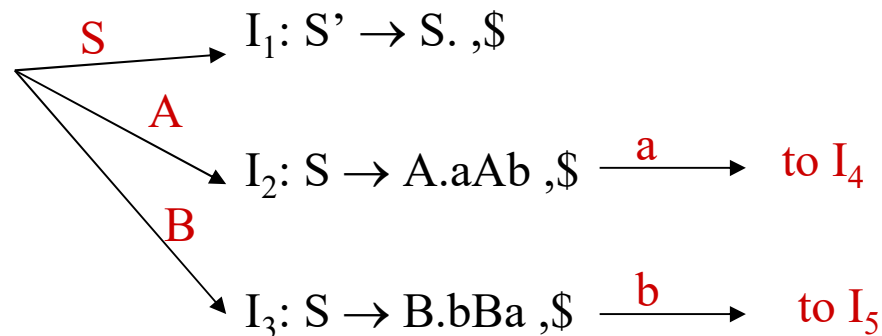
$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$,$

$S \rightarrow .AaAb$,$

$S \rightarrow .BbBa$,$

$A \rightarrow .$,a

$B \rightarrow .$,b

$\xrightarrow{S}$ I_1: $S' \rightarrow S.$,$

$\xrightarrow{A}$ I_2: $S \rightarrow A.aAb$,$ $\xrightarrow{a}$ to I_4

$\xrightarrow{B}$ I_3: $S \rightarrow B.bBa$,$ $\xrightarrow{b}$ to I_5

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

$A \rightarrow .$,b

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

$B \rightarrow .$,a

Canonical LR(1) Collection – Example2

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.,$/=

R → to I_7

L → to I_8

* → to I_4

id → to I_5

I_6:S → L=.R,$

R → .L,$

L → .*R,$

L → .id,$

R → to I_9

L → to I_{10}

* → to I_{11}

id → 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,$

R → to I_{13}

L → to I_{10}

* → to I_{11}

id → 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'.
   $$C \leftarrow \{I_0,...,I_n\}$$

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 $A \rightarrow \alpha$*** where $A \neq S'$.
   - If $S' \rightarrow S \bullet, \$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
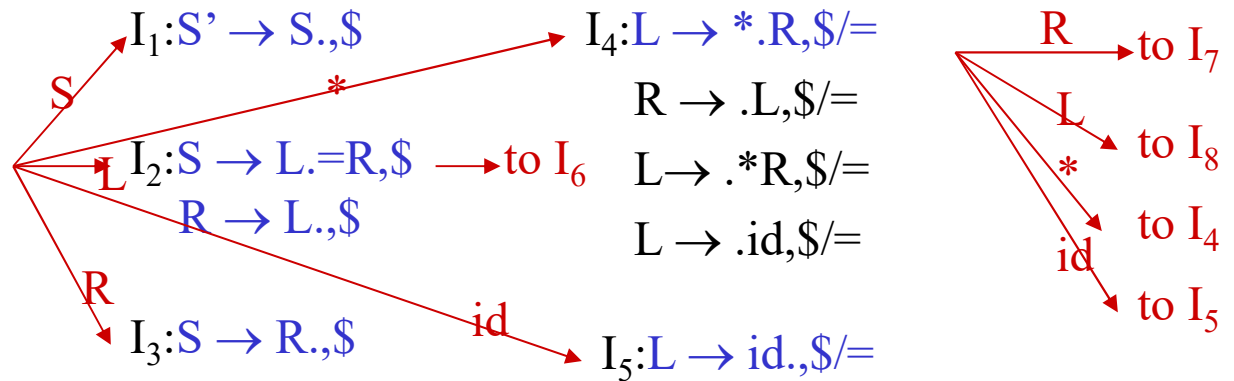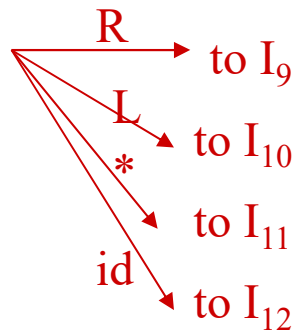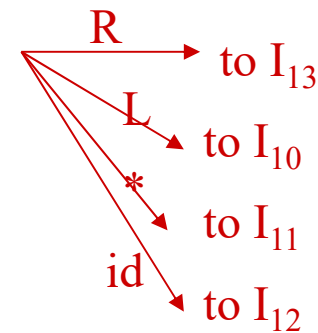
3. Create the parsing goto table
   - for all non-terminals A, if goto$(I_i,A)=I_j$ then goto[i,A]=j

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 – (for Example2)

	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 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.

Creating LALR 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: $I_{12} : L \rightarrow id \bullet ,=$

                         ➜ $L \rightarrow id \bullet ,\$$

$I_2 : L \rightarrow id \bullet ,\$$ have same core, merge them

- 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.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

$$C=\{I_0,...,I_n\} \;\Rightarrow\; C'=\{J_1,...,J_m\} \qquad \text{where } m \leq n$$

- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores
    $\Rightarrow$ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

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:

$$A \rightarrow \alpha\bullet,a \quad \text{and} \quad B \rightarrow \beta\bullet a\gamma,b$$

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

$$A \rightarrow \alpha\bullet,a \quad \text{and} \quad B \rightarrow \beta\bullet a\gamma,c$$

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.

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

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

$$\Downarrow$$

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

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

Canonical LALR(1) Collection – Example2

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_{411}: L → *•R,\$/=
R → •L,\$/=
L → •*R,\$/=
L → •id,\$/=

I_{512}: L → id•,\$/=

R → to I_{713}
L → to I_{810}
* → to I_{411}
id → to I_{512}

I_6: S → L=•R,\$
R → •L,\$
L → •*R,\$
L → •id,\$

R → to I_9
L → to I_{810}
* → to I_{411}
id → to I_{512}

I_9: S → L=R•,\$

I_{713}: L → *R•,\$/=

I_{810}: R → L•,\$/=

Same Cores
I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	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			

no shift/reduce or
no reduce/reduce conflict

$$\Downarrow$$

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.

$$E \rightarrow E+E \mid E*E \mid (E) \mid id \quad \Rightarrow$$

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$
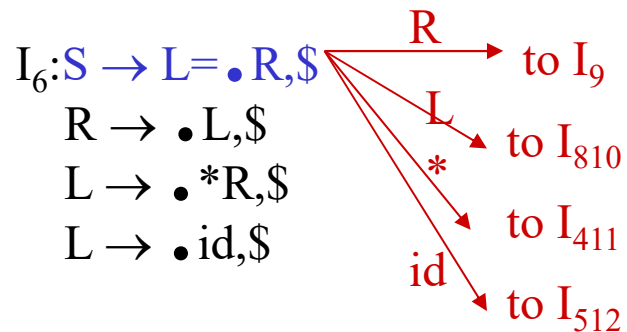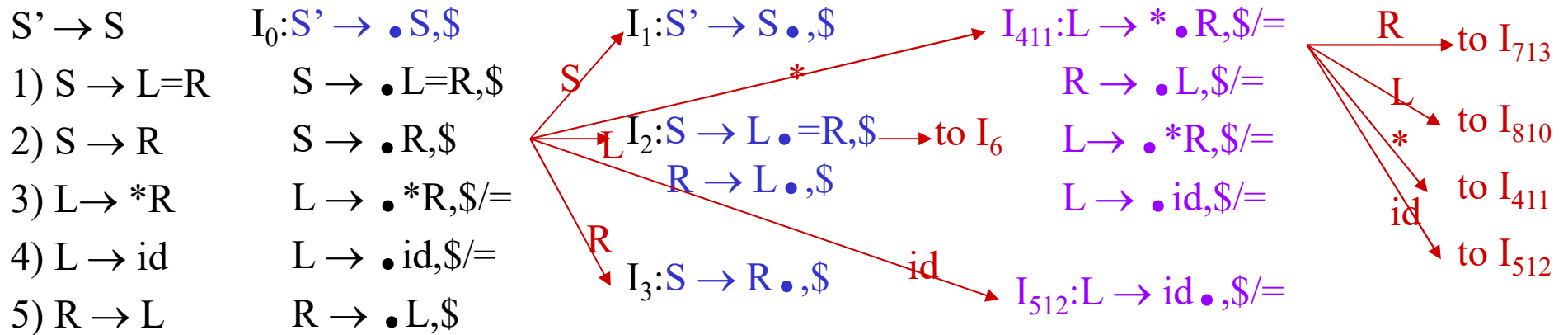
Error Recovery in LR Parsing

- 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-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

Summary Course Plan

Week	Plan	Remarks
Week 1, 2 and 3 (Jan 1-Jan 22)	• **Introduction to Compilers** • **Lexical Analysis:** • Regular Expressions -DFA, NFA • LEX specification	**Assignment #1**

Week 4,5 &6 (Jan 25-Feb 12)	• **Syntax Analysis**
	• **Parsing Techniques:**
	  • Top down parsing
	  • Bottom up parsing
		MidSem (Feb 15-20)

Week 7, 8 and 9 (Feb 15-Mar 5)	• **Syntax Directed Translations** • **Type Checking**	
Week 10 & 11 (Mar 8 – Mar 19)	• **Intermediate code generation** • **Runtime environment**	**Assignment #2**
Week 12, 13, 14 (Mar 22 – Apr 9)	• **Code Generation** • **Code optimization**	Apr 13 – Classes end
Apr 15-Apr 30	**End Sem Exams**	Apr 24-25 (Evaluation and checking of answer sheets)