

CMPT 755

Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

Parsing - Roadmap

- Parser:
 - decision procedure: builds a parse tree
- Top-down vs. bottom-up
- LL(1) – Deterministic Parsing
 - recursive-descent
 - table-driven
- LR(k) – Deterministic Parsing
 - LR(0), SLR(1), LR(1), LALR(1)
- Parsing arbitrary CFGs – Polynomial time parsing

Top-Down vs. Bottom Up

Grammar: $S \rightarrow A B$

Input String: ccbca

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Top-Down/leftmost		Bottom-Up/rightmost	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

Top-Down: Backtracking

$S \rightarrow A B$

$A \rightarrow c \mid \epsilon$

$B \rightarrow cbB \mid ca$

True/False

$S \Rightarrow^* cbca?$

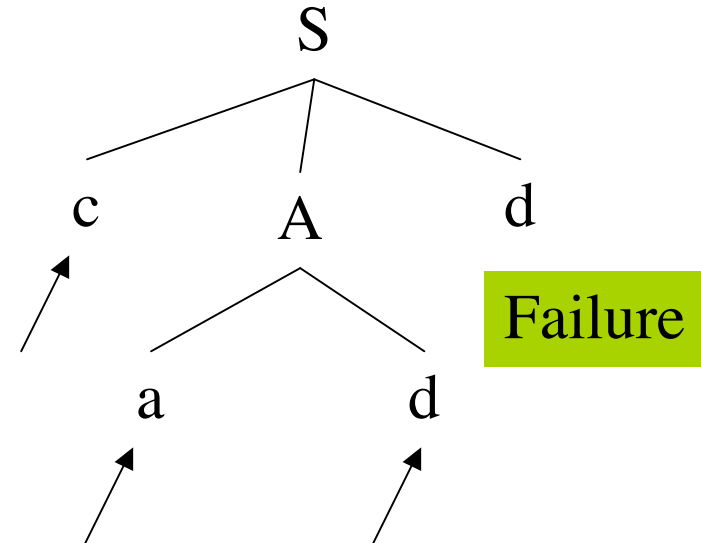
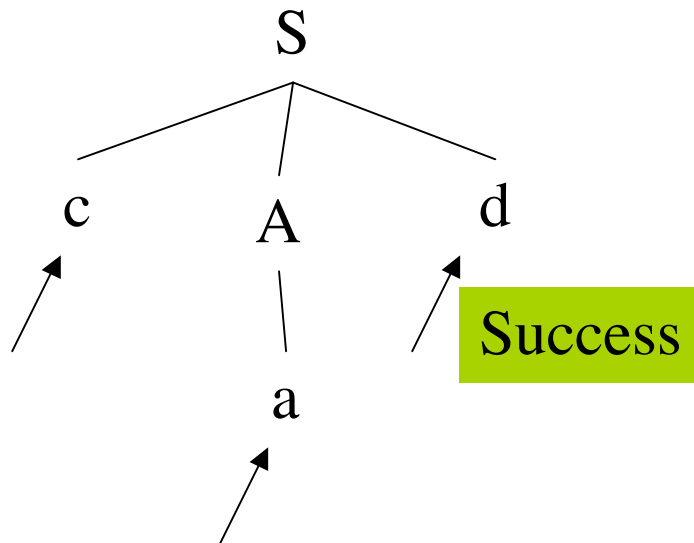
S	cbca	try $S \rightarrow AB$
AB	cbca	try $A \rightarrow c$
cB	cbca	match c
B	bca	dead-end, try $A \rightarrow \epsilon$
ϵB	cbca	try $B \rightarrow cbB$
cbB	cbca	match c
bB	bca	match b
B	ca	try $B \rightarrow cbB$
cbB	ca	match c
bB	a	dead-end, try $B \rightarrow ca$
ca	ca	match c
a	a	match a, Done!

Backtracking

$S \rightarrow cAd \mid c$
 $A \rightarrow a \mid ad$

Input: cad

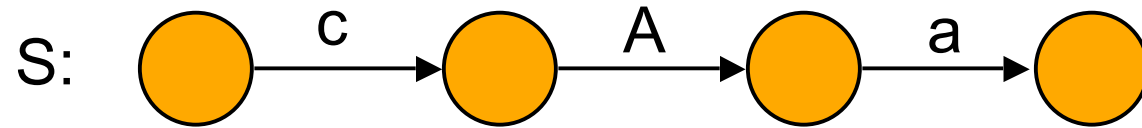
$S \rightarrow cAd \mid c$
 $A \rightarrow ad \mid a$



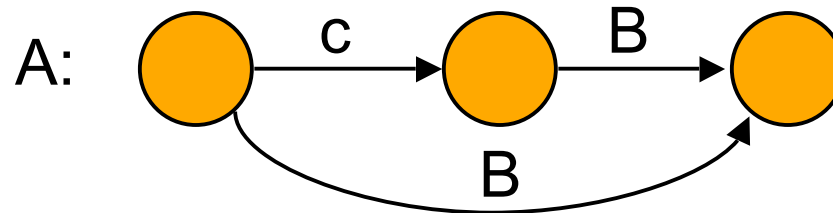
Recursive descent parser does not backtrack into rules that succeed

Transition Diagram

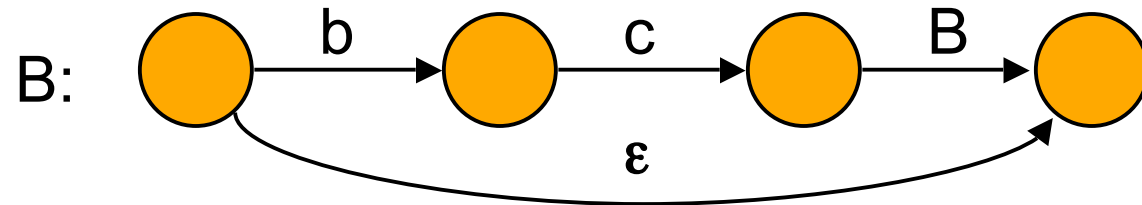
$S \rightarrow cAa$



$A \rightarrow cB \mid B$



$B \rightarrow bcB \mid \varepsilon$



Predictive Top-Down Parser

- Knows which production to choose based on single lookahead symbol
- Need LL(1) grammars
 - First L: reads input Left to right
 - Second L: produce Leftmost derivation
 - 1: one symbol of lookahead
- Can't have left-recursion
- Must be left-factored (no left-factors)
- Not all grammars can be made LL(1)

Leftmost derivation for **id + id * id**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \text{id}$

$E \Rightarrow \textcolor{blue}{E} + E$

$\Rightarrow \text{id} + \textcolor{blue}{E}$

$\Rightarrow \text{id} + \textcolor{blue}{E} * E$

$\Rightarrow \text{id} + \text{id} * \textcolor{blue}{E}$

$\Rightarrow \text{id} + \text{id} * \text{id}$

$E \Rightarrow^*_{\text{lm}} \text{id} + E * E$

Predictive Parsing Table

Productions	
1	$T \rightarrow F T'$
2	$T' \rightarrow \epsilon$
3	$T' \rightarrow * F T'$
4	$F \rightarrow \text{id}$
5	$F \rightarrow (T)$

	*	()	id	\$
T		$T \rightarrow F T'$		$T \rightarrow F T'$	
T'	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F		$F \rightarrow (T)$		$F \rightarrow \text{id}$	

Trace “(id)*id”

	*	()	id	\$
T		$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F		$F \rightarrow (T)$		$F \rightarrow id$	

Stack	Input	Output
\$T	(id)*id\$	
\$T'F	(id)*id\$	$T \rightarrow F T'$
\$T')T((id)*id\$	$F \rightarrow (T)$
\$T')T	id)*id\$	
\$T')T'F	id)*id\$	$T \rightarrow F T'$
\$T')T'id	id)*id\$	$F \rightarrow id$
\$T')T')*id\$	
\$T'))*id\$	$T' \rightarrow \epsilon$

Trace “(id)*id”

	*	()	id	\$
T		$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F		$F \rightarrow (T)$		$F \rightarrow id$	

Stack	Input	Output
\$T'	*id\$	
\$T'F*	*id\$	$T' \rightarrow * F T'$
\$T'F	id\$	
\$T'id	id\$	$F \rightarrow id$
\$T'	\$	
\$	\$	$T' \rightarrow \epsilon$

Table-Driven Parsing

```
stack.push($); stack.push(S);  
a = input.read();  
forever do begin  
    X = stack.peak();  
    if X = a and a = $ then return SUCCESS;  
    elseif X = a and a != $ then  
        pop X; a = input.read();  
    elseif X != a and X ∈ N and M[X,a] then  
        pop X; push right-hand side of M[X,a];  
    else ERROR!  
end
```

Predictive Parsing table

- Given a grammar produce the predictive parsing table
- We need to know for all rules $A \rightarrow \alpha \mid \beta$ the lookahead symbol
- Based on the lookahead symbol the table can be used to pick which rule to push onto the stack
- This can be done using two sets: FIRST and FOLLOW

FIRST and FOLLOW

$a \in \text{FIRST}(\alpha)$ if $\alpha \Rightarrow^* a\beta$

if $\alpha \Rightarrow^* \epsilon$ then $\epsilon \in \text{FIRST}(\alpha)$

$a \in \text{FOLLOW}(A)$ if $S \Rightarrow^* \alpha A a \beta$

$a \in \text{FOLLOW}(A)$ if $S \Rightarrow^* \alpha A \gamma a \beta$

and $\gamma \Rightarrow^* \epsilon$

Conditions for LL(1)

- Necessary conditions:
 - no ambiguity
 - no left recursion
 - Left factored grammar
- A grammar G is LL(1) iff - whenever $A \rightarrow \alpha \mid \beta$
 1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
 2. $\alpha \Rightarrow^* \varepsilon$ implies $\neg(\beta \Rightarrow^* \varepsilon)$
 3. $\alpha \Rightarrow^* \varepsilon$ implies $\text{First}(\beta) \cap \text{Follow}(A) = \emptyset$

proc First(α : string of symbols)

```
// assume  $\alpha = X_1 X_2 X_3 \dots X_n$   
if  $X_1 \in \mathbf{T}$  then First( $\alpha$ ) :=  $\{X_1\}$   
else begin  
   $i := 1$ ; First( $\alpha$ ) := First( $X_1$ ) \  $\{\epsilon\}$ ;  
  while  $X_i \Rightarrow^* \epsilon$  do begin  
    if  $i < n$  then  
      First( $\alpha$ ) := First( $\alpha$ )  $\cup$  First( $X_{i+1}$ ) \  $\{\epsilon\}$ ;  
    else  
      First( $\alpha$ ) := First( $\alpha$ )  $\cup \{\epsilon\}$ ;  
     $i := i + 1$ ;  
  end  
end
```


proc First(X); modified

```
foreach  $X \in T$  do First(X) := X;  
foreach  $p \in P : X \rightarrow \epsilon$  do First(X) :=  $\{\epsilon\}$ ;  
repeat foreach  $X \in N, p : X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$  do  
  begin  $i := 1$ ;  
    while  $Y_i \Rightarrow^* \epsilon$  and  $i \leq n$  do begin  
      First(X) := First(X)  $\cup$  First( $Y_i$ )  $\setminus \{\epsilon\}$ ;  
       $i := i + 1$ ;  
    end  
    if  $i = n + 1$  then First(X) := First(X)  $\cup \{\epsilon\}$ ;  
    else First(X) := First(X)  $\cup$  First( $Y_i$ );  
until no change in any First(X);
```

proc Follow(N: non-terminal)

Follow(S) := {\$};

repeat

foreach $p \in P$ **do**

case $p = A \rightarrow \alpha B \beta$ **begin**

 Follow(B) := Follow(B) \cup First(β) \ $\{\epsilon\}$;

if $\epsilon \in \text{First}(\beta)$ **then**

 Follow(B) := Follow(B) \cup Follow(A);

end

case $p = A \rightarrow \alpha B$

 Follow(B) := Follow(B) \cup Follow(A);

until no change in any Follow(N)

Example First/Follow

$$S \rightarrow AB$$

$$A \rightarrow c \mid \varepsilon$$

Not an LL(1) grammar

$$B \rightarrow cbB \mid ca$$

$$\text{First}(A) = \{c, \varepsilon\}$$

$$\text{Follow}(A) = \{c\}$$

$$\text{First}(B) = \{c\}$$

$$\text{Follow}(A) \cap$$

$$\text{First}(cbB) =$$

$$\text{First}(c) = \{c\}$$

$$\text{First}(ca) = \{c\}$$

$$\text{Follow}(B) = \{\$ \}$$

$$\text{First}(S) = \{c\}$$

$$\text{Follow}(S) = \{\$ \}$$

Converting to LL(1)

$S \rightarrow AB$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Note that grammar
is regular: $c? (cb)^* ca$

$c (c b c b \dots c b) c a$
 $(c b c b \dots c b) c a$



$c c (b c b \dots c b c) a$
 $c (b c b \dots c b c) a$

same as:

$c c? (bc)^* a$

$S \rightarrow cAa$

$A \rightarrow cB \mid B$

$B \rightarrow bcB \mid \varepsilon$

Verifying LL(1) using F/F sets

$$S \rightarrow cAa$$

$$A \rightarrow cB \mid B$$

$$B \rightarrow bcB \mid \varepsilon$$

$$\text{First}(A) = \{b, c, \varepsilon\}$$

$$\text{Follow}(A) = \{a\}$$

$$\text{First}(B) = \{b, \varepsilon\}$$

$$\text{Follow}(B) = \{a\}$$

$$\text{First}(S) = \{c\}$$

$$\text{Follow}(S) = \{\$\}$$

Building the Parse Table

- Compute First and Follow sets
- For each production $A \rightarrow \alpha$
 - foreach $a \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A,a]$
 - If $\epsilon \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A,b]$ for each b in $\text{Follow}(A)$
 - If $\epsilon \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A,\$]$ if $\$ \in \text{Follow}(\alpha)$
 - All undefined entries are errors

Revisit conditions for LL(1)

- A grammar G is LL(1) iff - whenever $A \rightarrow \alpha \mid \beta$
 1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
 2. $\alpha \Rightarrow^* \varepsilon$ implies $\neg(\beta \Rightarrow^* \varepsilon)$
 3. $\alpha \Rightarrow^* \varepsilon$ implies $\text{First}(\beta) \cap \text{Follow}(A) = \emptyset$
- No more than one entry per table field

Error Handling

- Reporting & Recovery
 - Report as soon as possible
 - Suitable error messages
 - Resume after error
 - Avoid cascading errors
- Phrase-level vs. Panic-mode recovery

Panic-Mode Recovery

- Skip tokens until *synchronizing set* is seen
 - Follow(A)
 - garbage or missing things after
 - Higher-level start symbols
 - First(A)
 - garbage before
 - Epsilon
 - if nullable
 - Pop/Insert terminal
 - “auto-insert”
- Add “synch” actions to table

Summary so far

- LL(1) grammars
 - necessary conditions
 - No left recursion
 - Left-factored
- Not all languages can be generated by LL(1) grammar
- LL(1) grammars can be parsed by simple predictive recursive-descent parser
 - Alternative: table-driven top-down parser

Bottom-up parsing overview

- Start from terminal symbols, search for a path to the start symbol
- Apply shift and reduce actions: postpone decisions
- LR parsing:
 - L: left to right parsing
 - R: rightmost derivation (in reverse or bottom-up)
- $LR(0) \rightarrow SLR(1) \rightarrow LR(1) \rightarrow LALR(1)$
 - 0 or 1 or k lookahead symbols

Actions in Shift-Reduce Parsing

- Shift
 - add terminal to parse stack, advance input
- Reduce
 - If αw on stack, and $A \rightarrow w$, and there is a $\beta \in T^*$ such that $S \Rightarrow_{\text{rm}}^* \alpha A \beta \Rightarrow_{\text{rm}} \alpha w \beta$ then we can *prune the handle* w ; we reduce αw to αA on the stack
 - αw is a *viable prefix*
- Error
- Accept

Questions

- When to shift/reduce?
 - What are valid handles?
 - Ambiguity: Shift/reduce conflict
- If reducing, using which production?
 - Ambiguity: Reduce/reduce conflict

Rightmost derivation for **id + id * id**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \text{id}$

$E \Rightarrow E * E$

$\Rightarrow E * \text{id}$

$\Rightarrow E + E * \text{id}$

$\Rightarrow E + \text{id} * \text{id}$

$\Rightarrow \text{id} + \text{id} * \text{id}$

reduce with $E \rightarrow \text{id}$

shift

$E \Rightarrow^*_{\text{rm}} E + E * \text{id}$

LR Parsing

- Table-based parser
 - Creates rightmost derivation (in reverse)
 - For “less massaged” grammars than LL(1)
- Data structures:
 - Stack of states/symbols $\{s\}$
 - Action table: **action** $[s, a]$; $a \in T$
 - Goto table: **goto** $[s, X]$; $X \in N$

Action/Goto Table

Productions	
1	$T \rightarrow F$
2	$T \rightarrow T * F$
3	$F \rightarrow id$
4	$F \rightarrow (T)$

	*	()	id	\$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				Acc!		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Trace “(id)*id”

Stack	Input	Action
0	(id) * id \$	Shift S5
0 5	id) * id \$	Shift S8
0 5 8) * id \$	Reduce 3 F→id, pop 8, goto [5,F]=1
0 5 1) * id \$	Reduce 1 T→ F, pop 1, goto [5,T]=6
0 5 6) * id \$	Shift S7
0 5 6 7	* id \$	Reduce 4 F→ (T), pop 7 6 5, goto [0,F]=1
0 1	* id \$	Reduce 1 T → F pop 1, goto [0,T]=2

Productions	
1	$T \rightarrow F$
2	$T \rightarrow T * F$
3	$F \rightarrow id$
4	$F \rightarrow (T)$

“(id)*id”

	*	()	id	\$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

	Input	Action
0	(id) * id \$	Shift S5
0 5	id) * id \$	Shift S8
0 5 8) * id \$	Reduce 2 $T \rightarrow T * F$, pop 8 5, goto [0,T]=2
0 5 1) * id \$	Reduce 1 $T \rightarrow F$, pop 1, goto [5,T]=6
0 5 6) * id \$	Shift S7
0 5 6 7	* id \$	Reduce 4 $F \rightarrow (T)$, pop 7 6 5, goto [0,F]=1
0 1	* id \$	Reduce 1 $T \rightarrow F$, pop 1, goto [0,T]=2

Trace “(id)*id”

Stack	Input	Action
0 1	* id \$	Reduce 1 $T \rightarrow F$, pop 1, goto [0,T]=2
0 2	* id \$	Shift S3
0 2 3	id \$	Shift S8
0 2 3 8	\$	Reduce 3 $F \rightarrow id$, pop 8, goto [3,F]=4
0 2 3 4	\$	Reduce 2 $T \rightarrow T * F$ pop 4 3 2, goto [0,T]=2
0 2	\$	Accept

Productions	
1	$T \rightarrow F$
2	$T \rightarrow T * F$
3	$F \rightarrow id$
4	$F \rightarrow (T)$

“(id)*id”

	*	()	id	\$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				A		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Stack	Input	Action
0 1	* id \$	Reduce 3 $F \rightarrow id$, pop 1,
0 2	* id \$	Shift S
0 2 3	id \$	Shift S8
0 2 3 8	\$	Reduce 2 $T \rightarrow T * F$, pop 8, goto [3,F]=4
0 2 3 4	\$	Reduce 3 $F \rightarrow id$, pop 4 3 2, goto [0,T]=2
0 2	\$	Accept

Tracing LR: **action**[s, a]

- case **shift** u :
 - push state u
 - read new a
- case **reduce** r :
 - lookup production $r: X \rightarrow Y_1..Y_k$;
 - pop k states, find state u
 - push **goto**[u, X]
- case **accept**: done
- no entry in action table: **error**

Configuration set

- Each set is a parser state
- Consider

$$\mathbf{T \rightarrow T * \bullet F}$$

$$\mathbf{F \rightarrow \bullet (T)}$$

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

- Like NFA-to-DFA conversion

Closure

Closure property:

- If $T \rightarrow X_1 \dots X_i \bullet X_{i+1} \dots X_n$ is in set, and X_{i+1} is a nonterminal, then $X_{i+1} \rightarrow \bullet Y_1 \dots Y_m$ is in the set as well for all productions $X_{i+1} \rightarrow Y_1 \dots Y_m$
- Compute as fixed point

Starting Configuration

- Augment Grammar with S'
- Add production $S' \rightarrow S$
- Initial configuration set is
 $\text{closure}(S' \rightarrow \bullet S)$

Example: $I = \text{closure}(S' \rightarrow \bullet T)$

$S' \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

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

$F \rightarrow \bullet (T)$

$S' \rightarrow T$

$T \rightarrow F \mid T * F$

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

Successor(I, X)

Informally: “move by symbol X”

1. move dot to the right in all items where dot is before X
2. remove all other items
(viable prefixes only!)
3. compute closure

Successor Example

$$\begin{aligned} I = \{ & S' \rightarrow \bullet T, \\ & T \rightarrow \bullet F, \\ & T \rightarrow \bullet T * F, \\ & F \rightarrow \bullet \text{id}, \\ & F \rightarrow \bullet (T) \} \end{aligned}$$

$\begin{aligned} S' &\rightarrow T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow \text{id} \mid (T) \end{aligned}$

Compute **Successor**(I, “(“)

$$\begin{aligned} \{ & F \rightarrow (\bullet T), T \rightarrow \bullet F, T \rightarrow \bullet T * F, \\ & F \rightarrow \bullet \text{id}, F \rightarrow \bullet (T) \} \end{aligned}$$

Sets-of-Items Construction

Family of configuration sets

function items(G')

$C = \{ \text{closure}(\{S' \rightarrow \bullet S\}) \};$

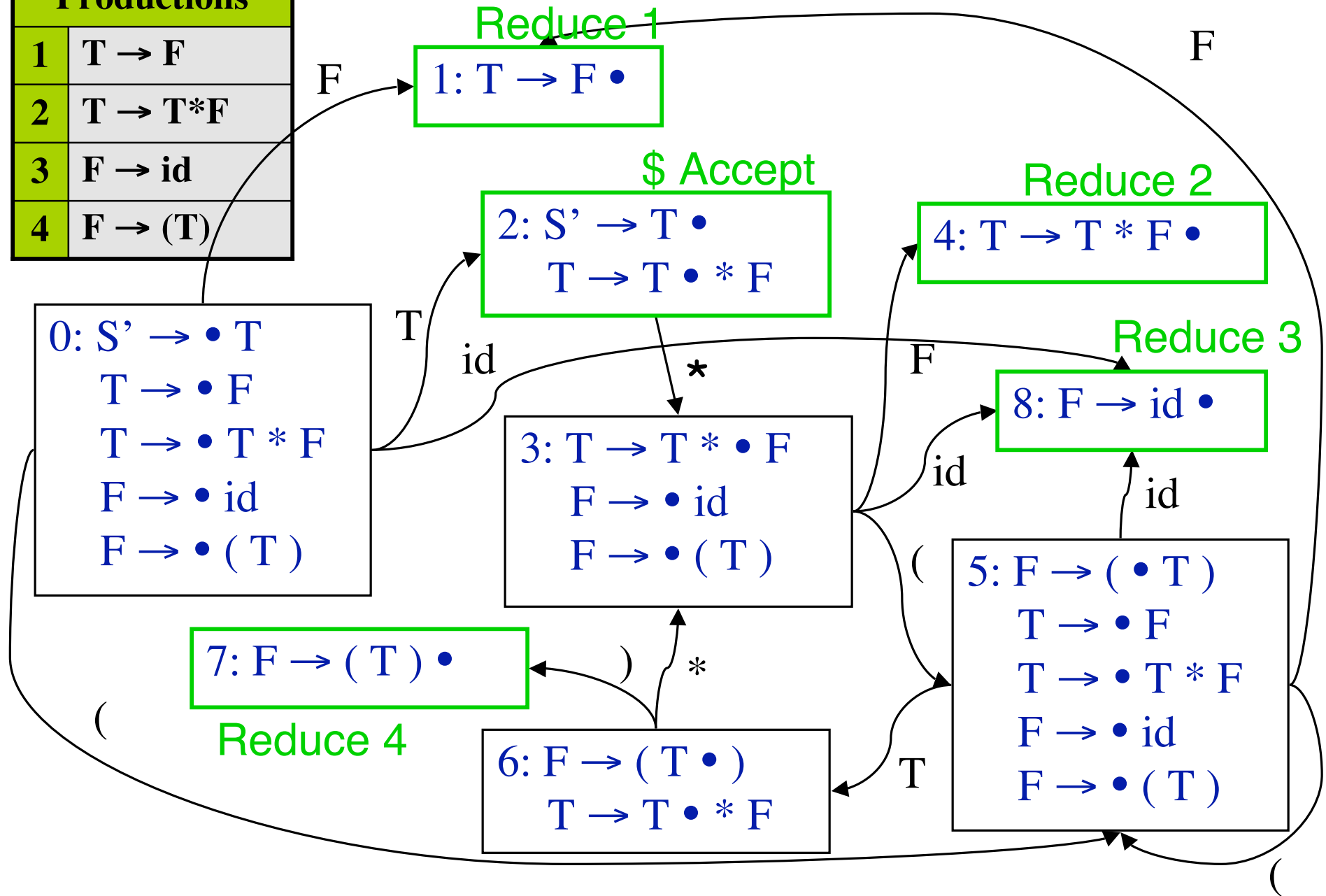
do **foreach** $I \in C$ **do**

foreach $X \in (N \cup T)$ **do**

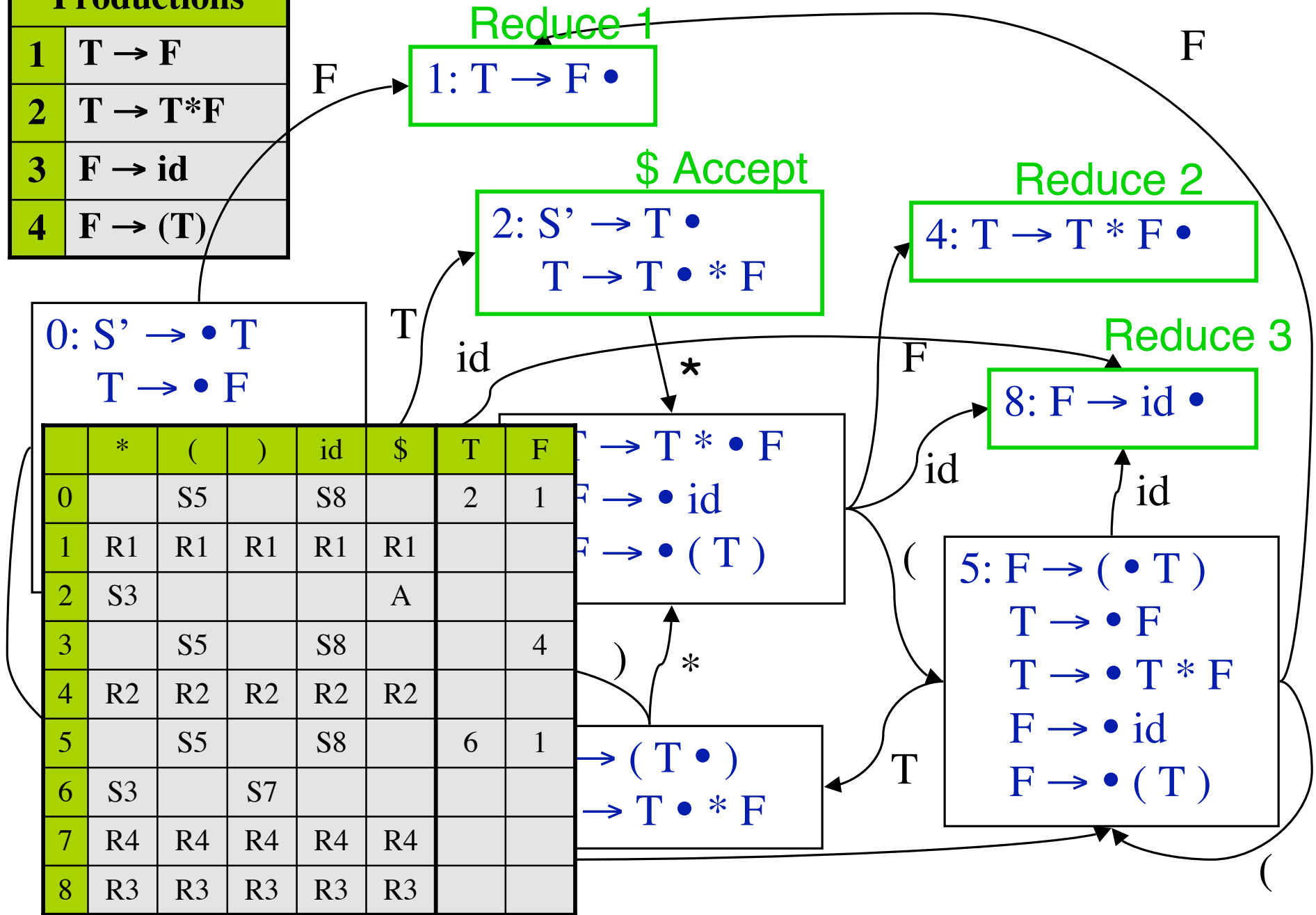
$C = C \cup \{ \text{Successor}(I, X) \};$

while C changes;

Productions	
1	$T \rightarrow F$
2	$T \rightarrow T * F$
3	$F \rightarrow id$
4	$F \rightarrow (T)$



Productions	
1	$T \rightarrow F$
2	$T \rightarrow T * F$
3	$F \rightarrow id$
4	$F \rightarrow (T)$



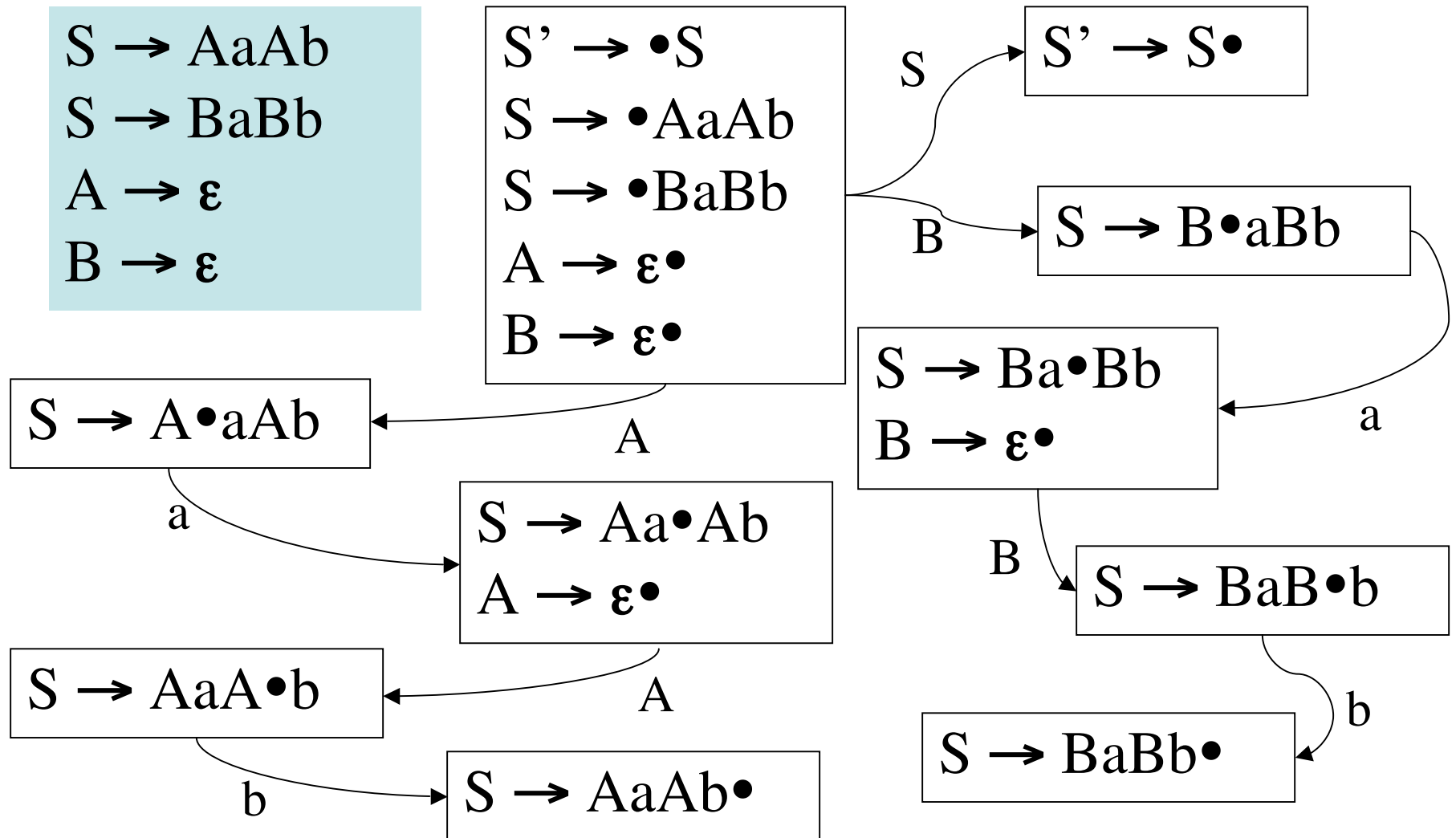
LR(0) Construction

1. Construct $F = \{I_0, I_1, \dots, I_n\}$
2. a) if $\{A \rightarrow \alpha \bullet\} \in I_i$ and $A \neq S$
then $\text{action}[i, _] := \text{reduce } A \rightarrow \alpha$
b) if $\{S' \rightarrow S \bullet\} \in I_i$
then $\text{action}[i, \$] := \text{accept}$
c) if $\{A \rightarrow \alpha \bullet a \beta\} \in I_i$ and $\text{Successor}(I_i, a) = I_j$
then $\text{action}[i, a] := \text{shift } j$
3. if $\text{Successor}(I_i, A) = I_j$ then $\text{goto}[i, A] := j$

LR(0) Construction (cont'd)

4. All entries not defined are errors
 5. Make sure I_0 is the initial state
- Note: LR(0) always reduces if $\{A \rightarrow \alpha \bullet\} \in I_i$, no lookahead
 - Shift and reduce items can't be in the same configuration set
 - Accepting state doesn't count as reduce item
 - At most one reduce item per set

Set-of-items with Epsilon rules



LR(0) conflicts:

$S' \rightarrow F$
 $F \rightarrow id \mid (T)$
 $F \rightarrow id = T ;$
 $T \rightarrow T * F$
 $T \rightarrow id$

5: $F \rightarrow id \bullet$

$F \rightarrow id \bullet = T$

Shift/reduce conflict

2: $F \rightarrow id \bullet$

$T \rightarrow id \bullet$

Reduce/Reduce conflict

Need more lookahead: SLR(1)

SLR(1) : Simple LR(1) Parsing

0: $S' \rightarrow \bullet T$

$T \rightarrow \bullet F$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet C (T)$

$F \rightarrow \bullet id$

$F \rightarrow \bullet id ++$

$F \rightarrow \bullet (T)$

$C \rightarrow \bullet id$

id

$S' \rightarrow T$

$T \rightarrow F \mid T * F \mid C (T)$

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

$C \rightarrow id$

1: $F \rightarrow id \bullet$

$F \rightarrow id \bullet ++$

$C \rightarrow id \bullet$

$\text{Follow}(F) = \{ *,), \$ \}$

$\text{Follow}(C) = \{ (\}$

$\text{action}[1,*] = \text{action}[1,)] = \text{action}[1,\$] = \text{Reduce } F \rightarrow id$

$\text{action}[1,(] = \text{Reduce } C \rightarrow id$

$\text{action}[1,++] = \text{Shift}$

SLR(1) Construction

1. Construct $F = \{I_0, I_1, \dots, I_n\}$
2. a) if $\{A \rightarrow \alpha \bullet\} \in I_i$ and $A \neq S'$
then $\text{action}[i, b] := \text{reduce } A \rightarrow \alpha$
for all $b \in \text{Follow}(A)$
b) if $\{S' \rightarrow S \bullet\} \in I_i$
then $\text{action}[i, \$] := \text{accept}$
c) if $\{A \rightarrow \alpha \bullet a \beta\} \in I_i$ and $\text{Successor}(I_i, a) = I_j$
then $\text{action}[i, a] := \text{shift } j$
3. if $\text{Successor}(I_i, A) = I_j$ then $\text{goto}[i, A] := j$

SLR(1) Construction (cont'd)

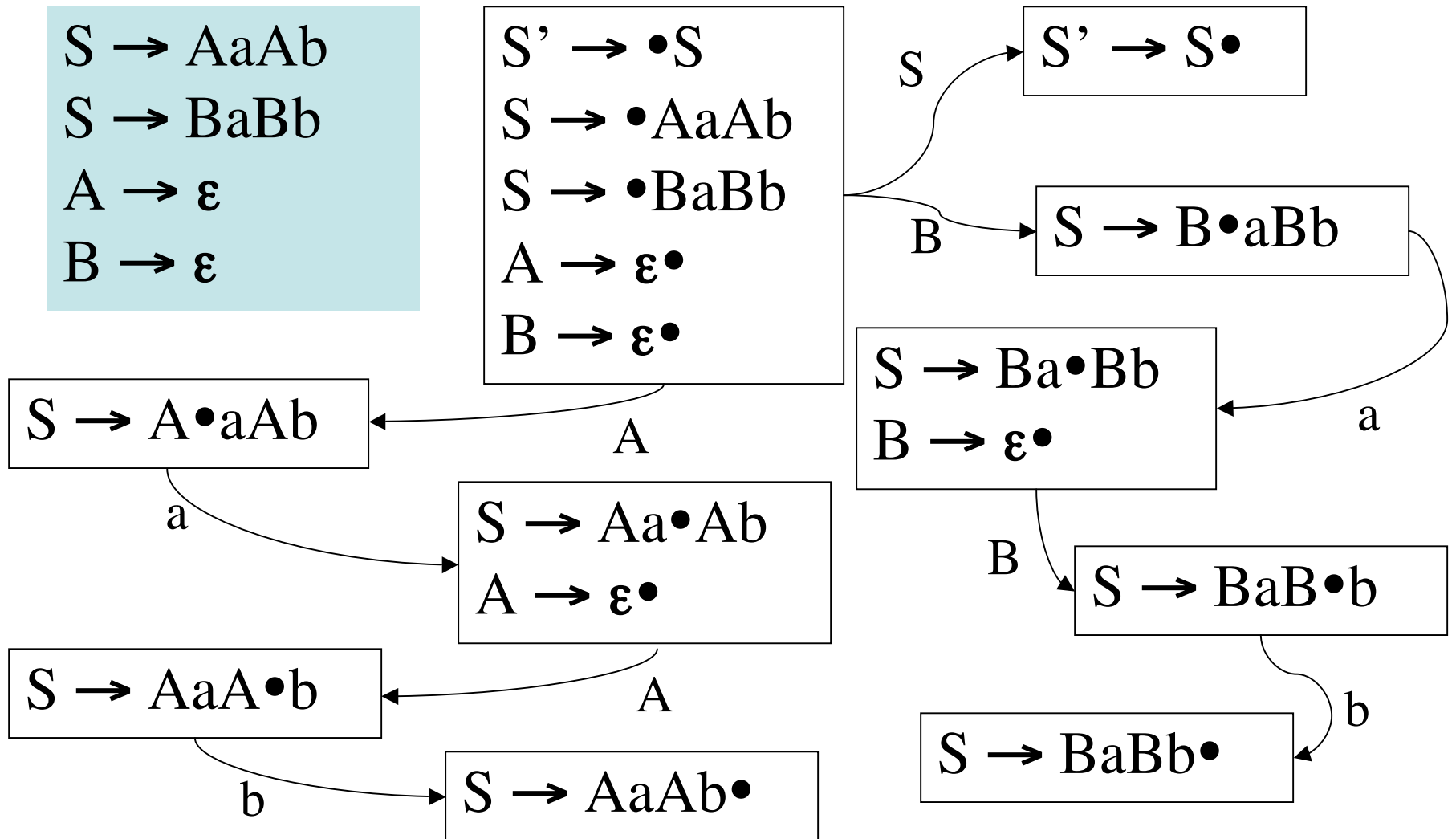
4. All entries not defined are errors
 5. Make sure I_0 is the initial state
- Note: SLR(1) only reduces $\{A \rightarrow \alpha \bullet\}$ if lookahead in $\text{Follow}(A)$
 - Shift and reduce items or more than one reduce item can be in the same configuration set as long as lookaheads are disjoint

SLR(1) Conditions

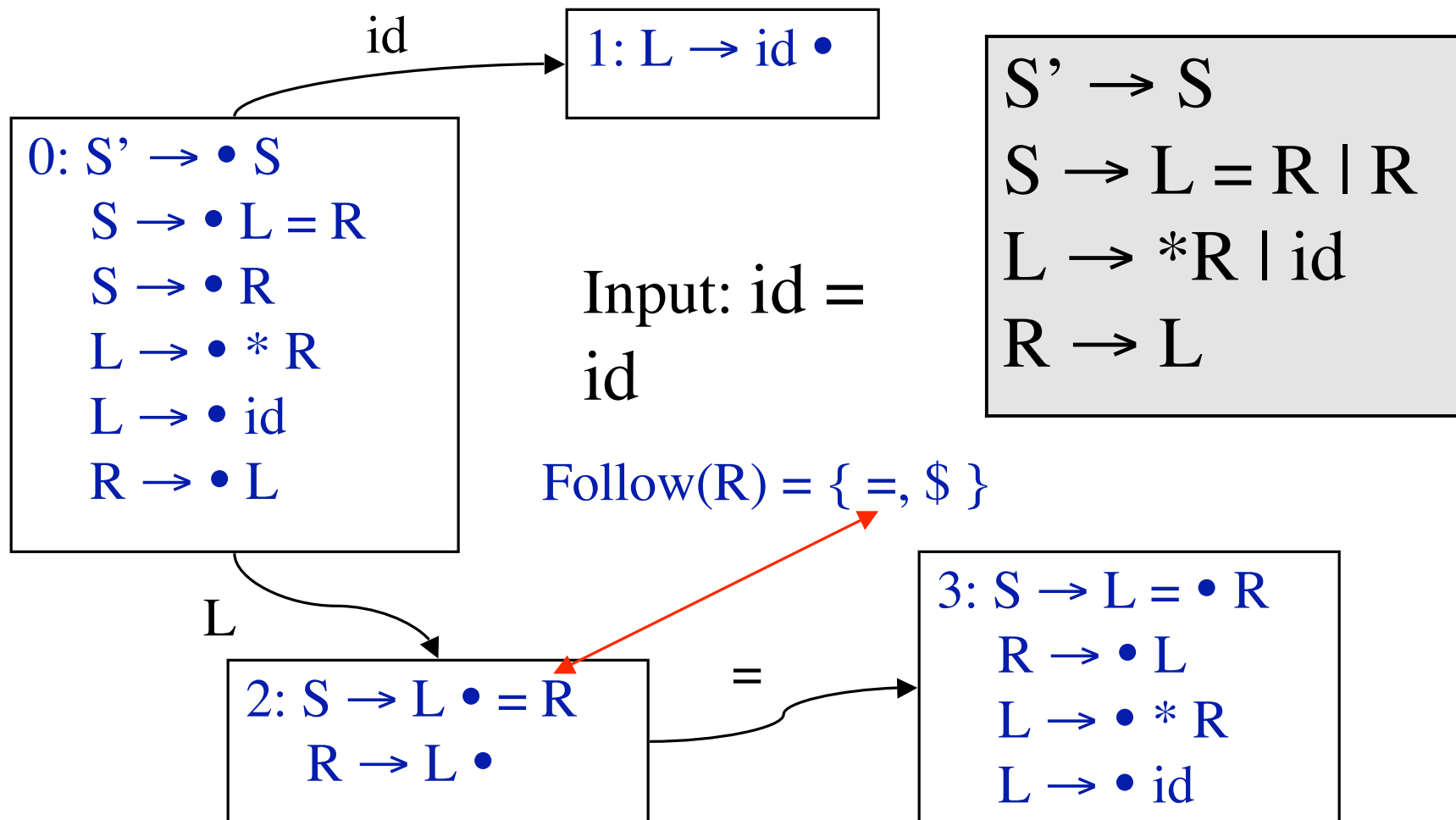
- A grammar is SLR(1) if for each configuration set:
 - For any item $\{A \rightarrow \alpha \bullet x \beta : x \in T\}$ there is no $\{B \rightarrow \gamma \bullet : x \in \text{Follow}(B)\}$
 - For any two items $\{A \rightarrow \alpha \bullet\}$ and $\{B \rightarrow \beta \bullet\}$ $\text{Follow}(A) \cap \text{Follow}(B) = \emptyset$

LR(0) Grammars \subset SLR(1) Grammars

Is this grammar SLR(1)?



SLR limitation: lack of context



Solution: Canonical LR(1)

- Extend definition of configuration
 - Remember lookahead
- New closure method
- Extend definition of Successor

LR(1) Configurations

- $[A \rightarrow \alpha \bullet \beta, a]$ for $a \in T$ is valid for a viable prefix $\delta\alpha$ if there is a rightmost derivation $S \Rightarrow^* \delta A \eta \Rightarrow^* \delta \alpha \beta \eta$ and $(\eta = a\gamma)$ or $(\eta = \varepsilon \text{ and } a = \$)$
- Notation: $[A \rightarrow \alpha \bullet \beta, a/b/c]$
 - if $[A \rightarrow \alpha \bullet \beta, a], [A \rightarrow \alpha \bullet \beta, b], [A \rightarrow \alpha \bullet \beta, c]$ are valid configurations

LR(1) Configurations

$S \rightarrow B B$

$B \rightarrow a B \mid b$

- $S \Rightarrow_{\text{rm}}^* aaBab \Rightarrow_{\text{rm}} aaaBab$
- Item $[B \rightarrow a \bullet B, a]$ is valid for viable prefix aaa
- $S \Rightarrow_{\text{rm}}^* BaB \Rightarrow_{\text{rm}} BaaB$
- Also, item $[B \rightarrow a \bullet B, \$]$ is valid for viable prefix Baa

LR(1) Closure

Closure property:

- If $[A \rightarrow \alpha \bullet B\beta, a]$ is in set, then $[B \rightarrow \bullet \gamma, b]$ is in set if $b \in \text{First}(\beta a)$
- Compute as fixed point
- Only include contextually valid lookaheads to guide reducing to B

Starting Configuration

- Augment Grammar with S' just like for LR(0), SLR(1)
- Initial configuration set is
$$I = \text{closure}([S' \rightarrow \bullet S, \$])$$

Example: $\text{closure}([S' \rightarrow \bullet S, \$])$

$[S' \rightarrow \bullet S, \$]$

$[S \rightarrow \bullet L = R, \$]$

$[S \rightarrow \bullet R, \$]$

$[L \rightarrow \bullet * R, =]$

$[L \rightarrow \bullet \text{id}, =]$

$[R \rightarrow \bullet L, \$]$

$[L \rightarrow \bullet * R, \$]$

$[L \rightarrow \bullet \text{id}, \$]$

$S' \rightarrow S$

$S \rightarrow L = R \mid R$

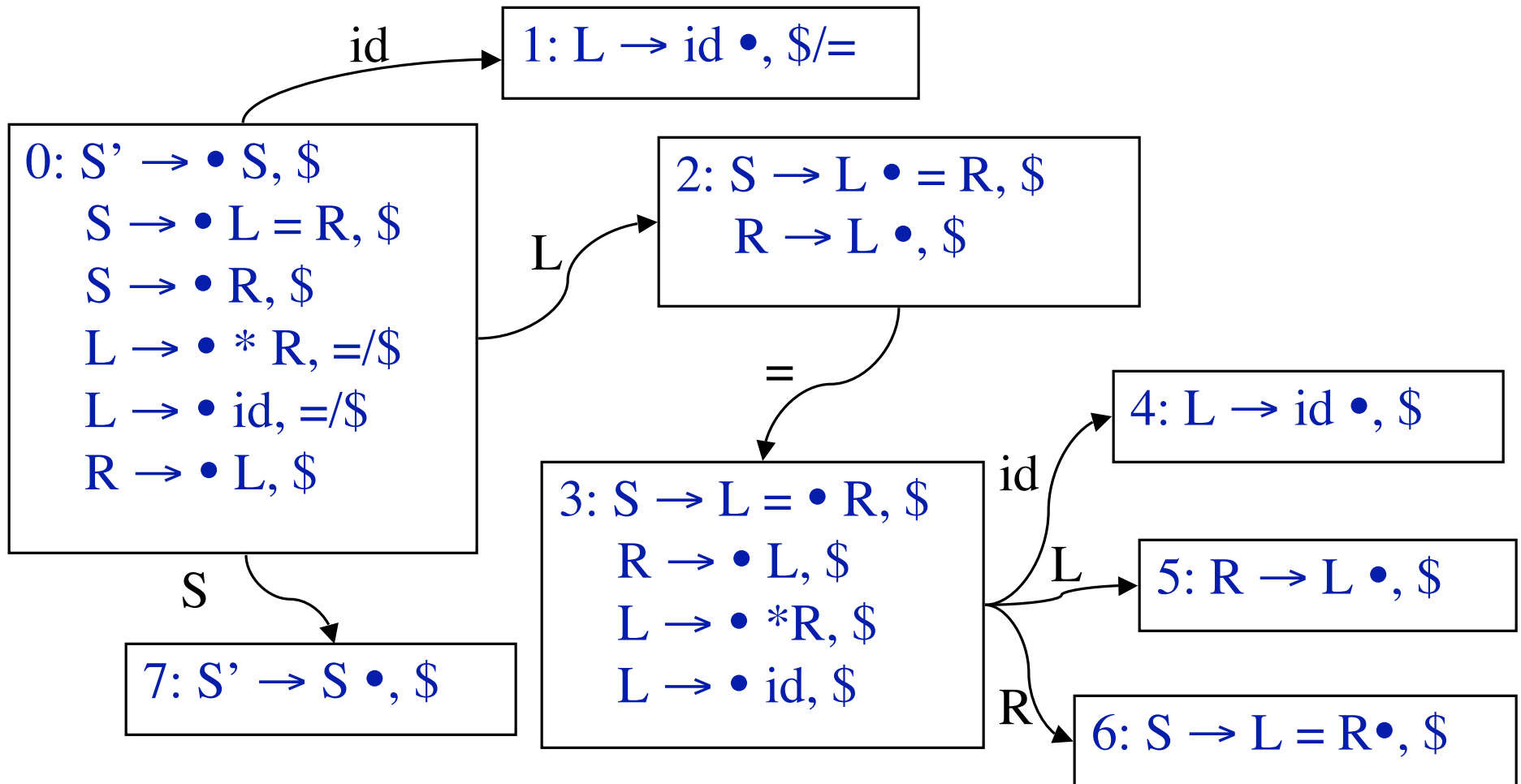
$L \rightarrow *R \mid \text{id}$

$R \rightarrow L$

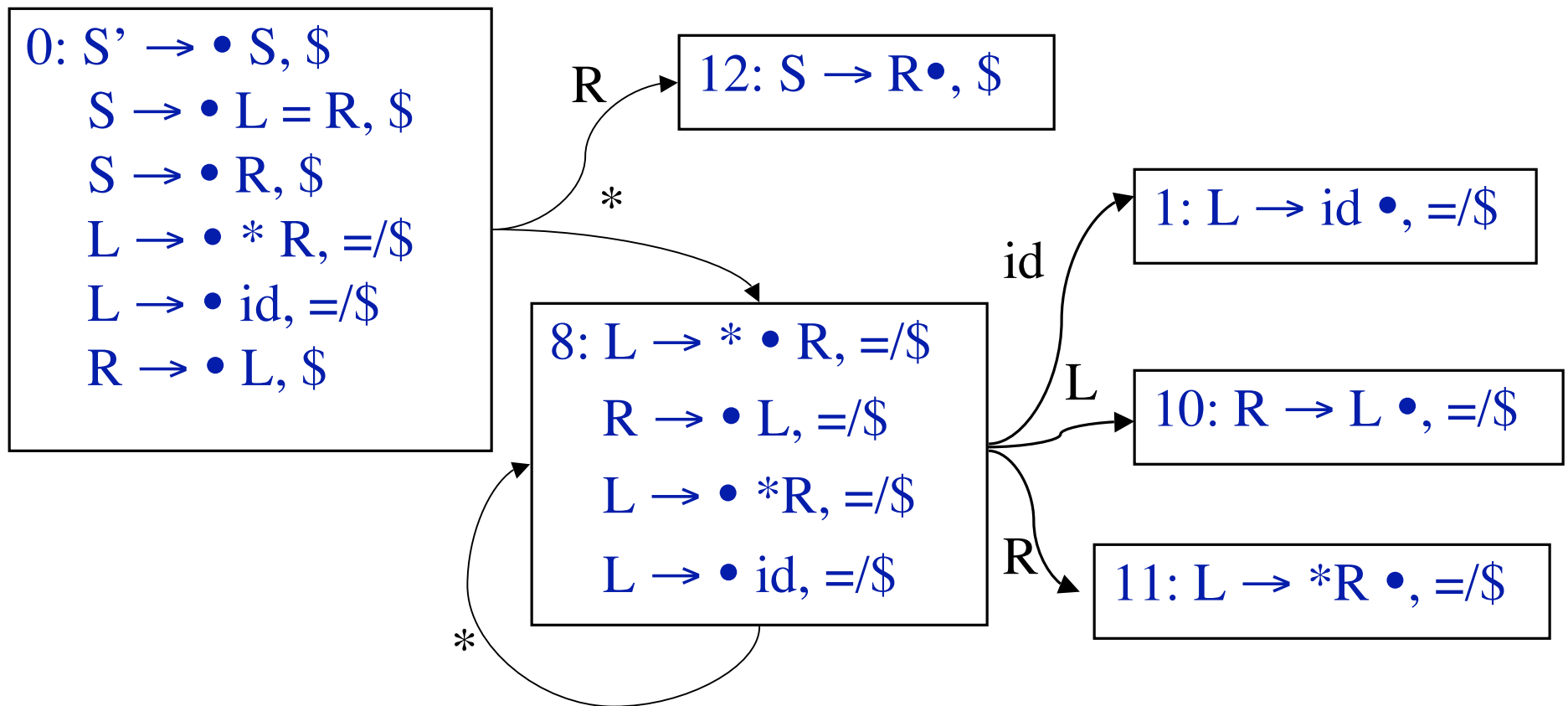
LR(1) Successor(C, X)

- Let $I = [A \rightarrow \alpha \bullet B\beta, a]$
- $\text{Successor}(I, B)$
 $= \text{closure}([A \rightarrow \alpha B \bullet \beta, a])$

LR(1) Example: $*id = id$



LR(1) Example: $*id = id$



LR(1) Construction

1. Construct $F = \{I_0, I_1, \dots, I_n\}$
2.
 - a) if $[A \rightarrow \alpha \bullet, a] \in I_i$ and $A \neq S'$
then $\text{action}[i, a] := \text{reduce } A \rightarrow \alpha$
 - b) if $[S' \rightarrow S \bullet, \$] \in I_i$
then $\text{action}[i, \$] := \text{accept}$
 - c) if $[A \rightarrow \alpha \bullet a \beta, b] \in I_i$ and $\text{Successor}(I_i, a) = I_j$
then $\text{action}[i, a] := \text{shift } j$
3. if $\text{Successor}(I_i, A) = I_j$ then $\text{goto}[i, A] := j$

LR(1) Construction (cont'd)

4. All entries not defined are errors
 5. Make sure I_0 is the initial state
- Note: LR(1) only reduces using $A \rightarrow \alpha$ for $[A \rightarrow \alpha\bullet, a]$ if a follows
 - LR(1) states remember context by virtue of lookahead
 - Possibly many states!
 - LALR(1) combines some states

LR(1) Conditions

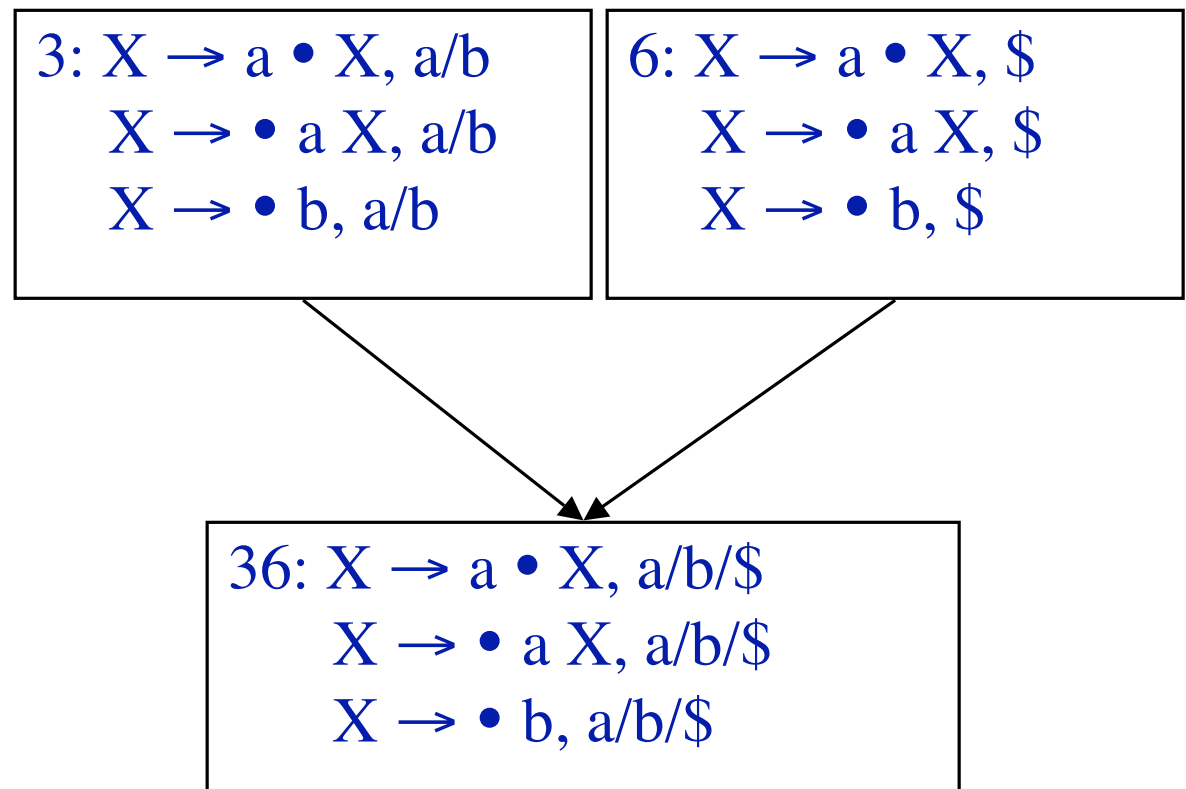
- A grammar is LR(1) if for each configuration set holds:
 - For any item $[A \rightarrow \alpha \bullet x \beta, a]$ with $x \in T$ there is no $[B \rightarrow \gamma \bullet, x]$
 - For any two complete items $[A \rightarrow \gamma \bullet, a]$ and $[B \rightarrow \beta \bullet, b]$ it follows $a \neq b$.
- Grammars:
 - $LR(0) \subset SLR(1) \subset LR(1) \subset LR(k)$
- Languages expressible by grammars:
 - $LR(0) \subset SLR(1) \subset LR(1) = LR(k)$

Canonical LR(1) Recap

- LR(1) uses left context, current handle and lookahead to decide when to reduce or shift
- Most powerful parser so far
- LALR(1) is practical simplification with fewer states

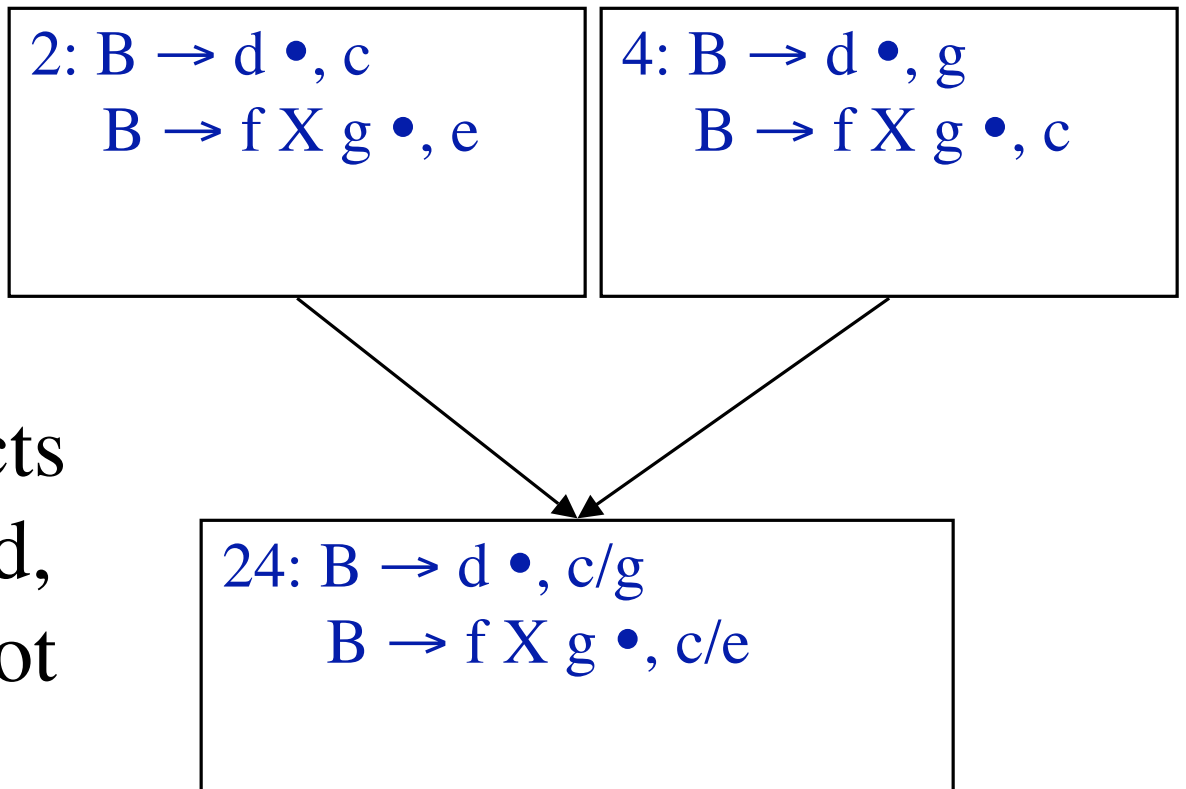
Merging States in LALR(1)

- $S' \rightarrow S$
 $S \rightarrow XX$
 $X \rightarrow aX$
 $X \rightarrow b$
- Same **Core Set**
- Different lookaheads



R/R conflicts when merging

- $B \rightarrow d$
 $B \rightarrow f X g$
 $X \rightarrow \dots$



- If R/R conflicts are introduced, grammar is not LALR(1)!

LALR(1)

- LALR(1) Condition:
 - Merging in this way does not introduce reduce/reduce conflicts
 - Shift/reduce can't be introduced
- Merging brute force or step-by-step
- More compact than canonical LR, like SLR(1)
- More powerful than SLR(1)
 - Not always merge to full Follow Set

S/R & ambiguous grammars

- Lx(k) Grammar vs. Language
 - Grammar is Lx(k) if it can be parsed by Lx(k) method
 - according to criteria that is specific to the method.
 - A Lx(k) grammar may or may not exist for a language.
- Even if a given grammar is not LR(k), shift/reduce parser can *sometimes* handle them by accounting for ambiguities
 - Example: ‘dangling’ else
 - Preferring shift to reduce means matching inner ‘if’

Dangling ‘else’

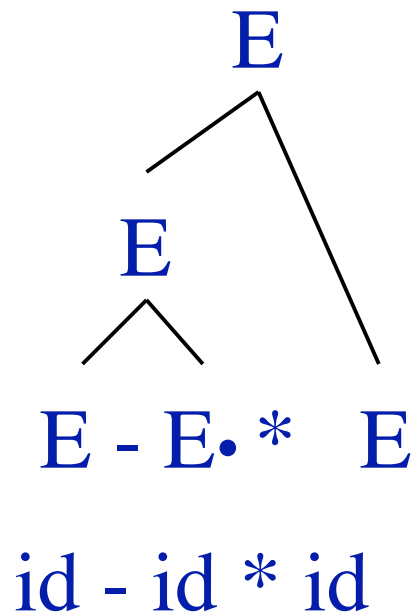
1. $S \rightarrow \text{if } E \text{ then } S$
 2. $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- Viable prefix “if E then if E then S”
 - Then read else
 - Shift “else” (means go for 2)
 - Reduce (reduce using production #1)
 - NB: dangling else as written above is ambiguous
 - NB: Ambiguity can be resolved, but there’s still no LR(k) grammar

Precedence & Associativity

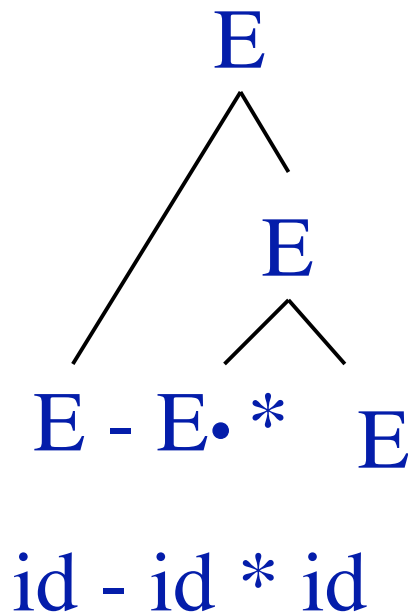
- Consider

$$E \rightarrow E - E \mid E * E \mid \text{id}$$

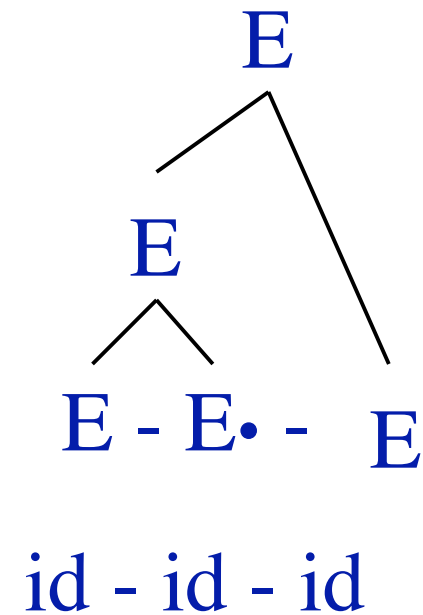
Reduce



Shift



Reduce



Precedence Relations

- Let $A \rightarrow w$ be a rule in the grammar
- And b is a terminal
- In some state q of the LR(1) parser there is a shift-reduce conflict:
 - either reduce with $A \rightarrow w$ or shift on b
- Write down a rule, either:
 $A \rightarrow w, < b$ or $A \rightarrow w, > b$

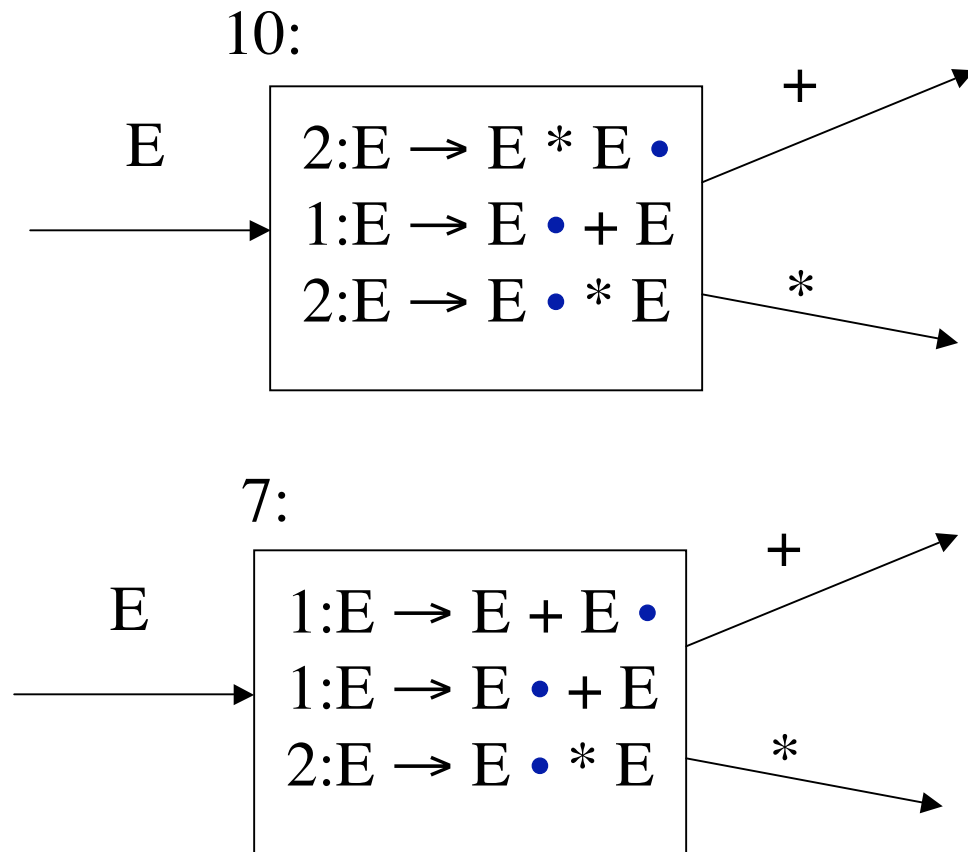
Precedence Relations

- $A \rightarrow w, < b$ means rule has less precedence and so we shift if we see b in the lookahead
- $A \rightarrow w, > b$ means rule has higher precedence and so we reduce if we see b in the lookahead
- If there are multiple terminals with shift-reduce conflicts, then we list them all:
 $A \rightarrow w, > b, < c, > d$

Precedence Relations

- Consider the grammar
 $E \rightarrow E + E \mid E * E \mid (E) \mid a$
- Assume left-association so that $E+E+E$ is interpreted as $(E+E)+E$
- Assume multiplication has higher precedence than addition
- Then we can write precedence rules/relns:
 $E \rightarrow E + E, > +, < *$
 $E \rightarrow E * E, > +, > *$

Precedence & Associativity



$E \rightarrow E + E, > +, < *$
 $E \rightarrow E * E, > +, > *$

	$+$	$*$
7	R1	Shift
10	R2	R2

Handling S/R & R/R Conflicts

- Have a conflict?
 - No? – Done, grammar is compliant.
- Already using most powerful parser available?
 - No? – Upgrade and goto 1
- Can the grammar be rearranged so that the conflict disappears?
 - While preserving the language!

Conflicts revisited (cont'd)

- Can the grammar be rearranged so that the conflict disappears?
 - No?
 - Is the conflict S/R and does shift-to-reduce preference yield desired result?
 - Yes: Done. (Example: dangling else)
 - Else: Bad luck
 - Yes: Is it worth it?
 - Yes, resolve conflict.
 - No: live with default or specified conflict resolution (precedence, associativity)

Compiler (parser) compilers

- Rather than build a parser for a particular grammar (e.g. recursive descent), write down a grammar as a text file
- Run through a compiler compiler which produces a parser for that grammar
- The parser is a program that can be compiled and accepts input strings and produces user-defined output

Compiler (parser) compilers

- For LR parsing, all it needs to do is produce action/goto table
 - Yacc (yet another compiler compiler) was distributed with Unix, the most popular tool. Uses LALR(1).
 - Many variants of yacc exist for many languages
- As we will see later, translation of the parse tree into machine code (or anything else) can also be written down with the grammar
- Handling errors and interaction with the lexical analyzer have to be precisely defined

Parsing CFGs

- Consider the problem of parsing with arbitrary CFGs
- For any input string, the parser has to produce a parse tree
- The simpler problem: print **yes** if the input string is generated by the grammar, print **no** otherwise
- This problem is called *recognition*

CKY Recognition Algorithm

- The Cocke-Kasami-Younger algorithm
- As we shall see it runs in time that is polynomial in the size of the input
- It takes space polynomial in the size of the input
- **Remarkable fact:** it can find all possible parse trees (exponentially many) in polynomial time

Chomsky Normal Form

- Before we can see how CKY works, we need to convert the input CFG into Chomsky Normal Form
- CNF means that the input CFG G is converted to a new CFG G' in which all rules are of the form:

$$A \rightarrow B C$$

$$A \rightarrow a$$

Epsilon Removal

- First step, remove epsilon rules

$$A \rightarrow B C$$

$$C \rightarrow \varepsilon \mid C D \mid a$$

$$D \rightarrow b \quad B \rightarrow b$$

- After ε -removal:

$$A \rightarrow B \mid B C D \mid B a$$

$$C \rightarrow D \mid C D D \mid a D \mid C D \mid a$$

$$D \rightarrow b \quad B \rightarrow b$$

Removal of Chain Rules

- Second step, remove chain rules

$$A \rightarrow B C \mid C D C$$
$$C \rightarrow D \mid a$$
$$D \rightarrow d \quad B \rightarrow b$$

- After removal of chain rules:

$$A \rightarrow B a \mid B D \mid a D a \mid a D D \mid D D a \mid D D D$$
$$D \rightarrow d \quad B \rightarrow b$$

Eliminate terminals from RHS

- Third step, remove terminals from the rhs of rules

$$A \rightarrow B a C d$$

- After removal of terminals from the rhs:

$$A \rightarrow B N_1 C N_2$$

$$N_1 \rightarrow a$$

$$N_2 \rightarrow d$$

Binarize RHS with Nonterminals

- Fourth step, convert the rhs of each rule to have two non-terminals

$$A \rightarrow B N_1 C N_2$$

$$N_1 \rightarrow a$$

$$N_2 \rightarrow d$$

- After converting to binary form:

$$A \rightarrow B N_3 \quad N_1 \rightarrow a$$

$$N_3 \rightarrow N_1 N_4 \quad N_2 \rightarrow d$$

$$N_4 \rightarrow C N_2$$

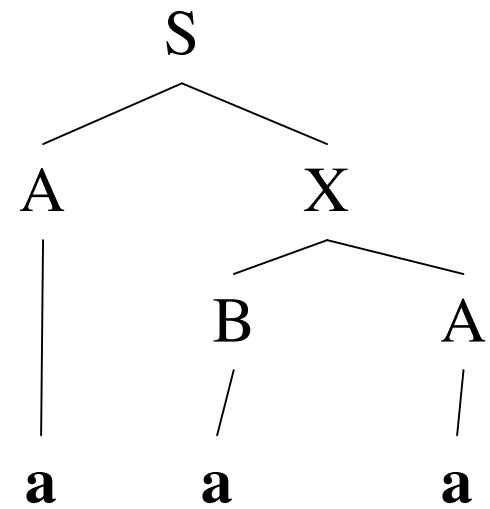
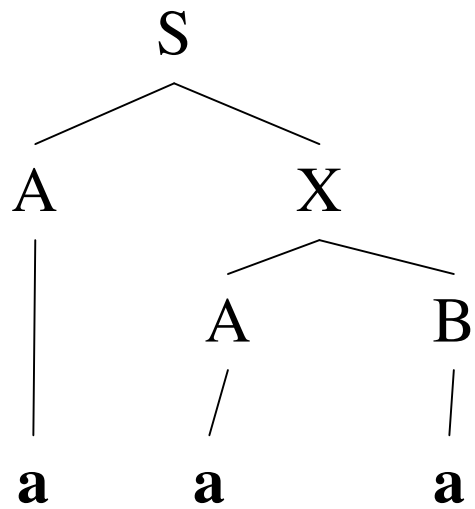
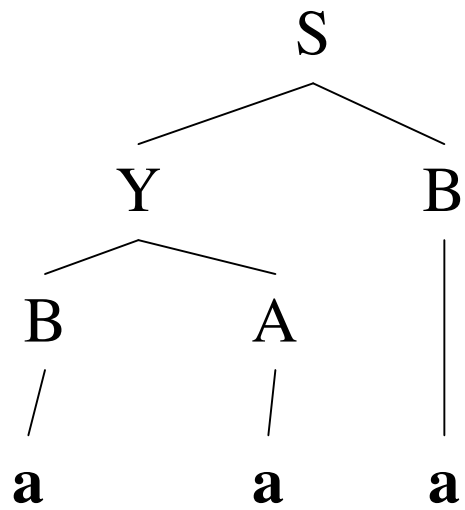
CKY algorithm

- We will consider the working of the algorithm on an example CFG and input string
- Example CFG:
$$S \rightarrow A X \mid Y B$$
$$X \rightarrow A B \mid B A \quad Y \rightarrow B A$$
$$A \rightarrow a \quad B \rightarrow a$$
- Example input string: *aaa*

CKY Algorithm

	0	1	2	3
0		A, B $A \rightarrow a$ $B \rightarrow a$	X, Y $X \rightarrow A B \mid B A$ $Y \rightarrow B A$	S $S \rightarrow A_{(0,1)} X_{(1,3)}$ $S \rightarrow Y_{(0,2)} B_{(2,3)}$
1			A, B $A \rightarrow a$ $B \rightarrow a$	X, Y $X \rightarrow A B \mid B A$ $Y \rightarrow B A$
2				A, B $A \rightarrow a$ $B \rightarrow a$
		a	a	a

Parse trees



CKY Algorithm

Input string **input** of size n

Create a 2D table **chart** of size n^2

for $i=0$ **to** $n-1$

chart $[i][i+1] = A$ **if** there is a rule $A \rightarrow a$ and **input** $[i]=a$

for $j=2$ **to** N

for $i=j-2$ **downto** 0

for $k=i+1$ **to** $j-1$

chart $[i][j] = A$ **if** there is a rule $A \rightarrow B C$ **and**

chart $[i][k] = B$ **and** **chart** $[k][j] = C$

return *yes* **if** **chart** $[0][n]$ has the start symbol

else return *no*

CKY algorithm summary

- Parsing arbitrary CFGs
- For the CKY algorithm, the time complexity is $O(|G|^2 n^3)$
- The space requirement is $O(n^2)$
- The CKY algorithm handles arbitrary ambiguous CFGs
- All ambiguous choices are stored in the chart
- For compilers we consider parsing algorithms for CFGs that do not handle ambiguous grammars

GLR – Generalized LR Parsing

- Works for any CFG (just like CKY algorithm)
 - Masaru Tomita [1986]
- If you have shift/reduce conflict, just clone your stack and shift in one clone, reduce in the other clone
 - proceed in lockstep
 - parser that get into error states die
 - merge parsers that lead to identical reductions (graph structured stack)

Parsing - Summary

- Parsing arbitrary CFGs: $O(n^3)$ time complexity
- Top-down vs. bottom-up
- Lookahead: FIRST and FOLLOW sets
- LL(1) – Parsing: $O(n)$ time complexity
 - recursive-descent and table-driven predictive parsing
- LR(k) – Parsing : $O(n)$ time complexity
 - LR(0), SLR(1), LR(1), LALR(1)
- Resolving shift/reduce conflicts
 - using precedence, associativity