

CMPT 379

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	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

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 $S \Rightarrow_{rm}^* \alpha A \beta \Rightarrow_{rm} \alpha w \beta$ 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

$$E \Rightarrow_{\text{rm}}^* E + E \setminus^* \text{id}$$

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** $[s, X]$; $X \in N$

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

Action/Goto Table

		*	()	id	\$	T	F
0			S5		S8		2	1
1	R1	R1	R1	R1	R1	R1		
2	S3					Acc!		
3			S5		S8			4
4	R2	R2	R2	R2	R2	R2		
5			S5		S8		6	1
6	S3			S7				
7	R4	R4	R4	R4	R4	R4		
8	R3	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 3 $F \rightarrow id$, pop 8, goto [2,F]=1
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 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

Tracing LR: $\text{action}[s, a]$

- 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
- We use the notion of a dotted rule or item:

$$T \rightarrow T * \bullet F$$

- The dot is before **F**, so we predict all rules with **F** as the left-hand side

$$T \rightarrow T * \bullet F$$

$$F \rightarrow \bullet (T)$$

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

- This creates a configuration set (or item set)

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
- The closure property creates a configuration set (item set) from a dotted rule (item).

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

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

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

Compute **Successor**(I, “(“)

$$\{ F \rightarrow (\bullet T), T \rightarrow \bullet F, T \rightarrow \bullet T * F, \\ F \rightarrow \bullet \text{id}, F \rightarrow \bullet (T) \}$$

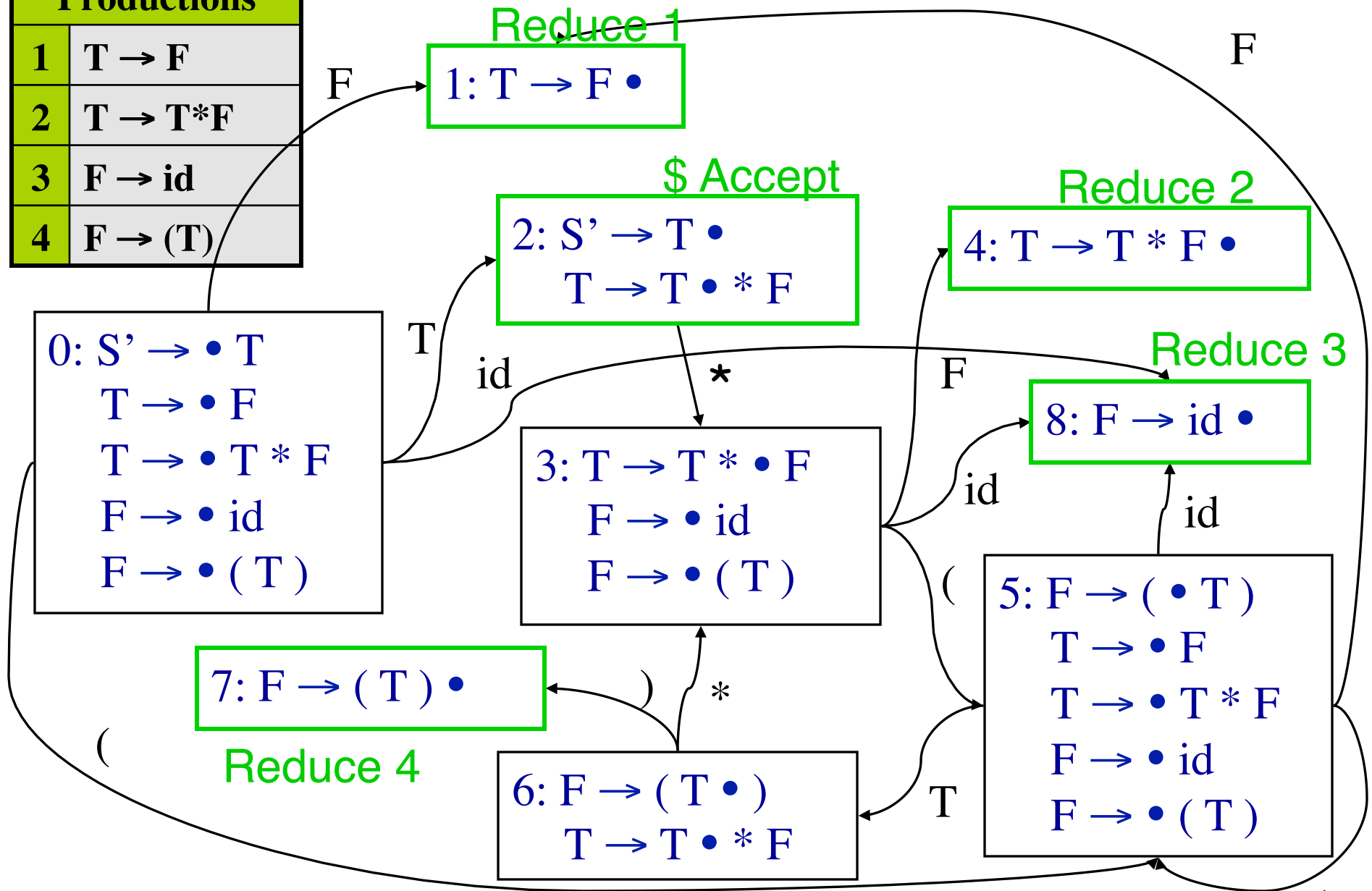
Sets-of-Items Construction

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 $F \rightarrow (T)$

0: $S' \rightarrow \bullet T$
 $T \rightarrow \bullet F$

	*	()	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		

Reduce 1

1: $T \rightarrow F \bullet$

\$ Accept

2: $S' \rightarrow T \bullet$
 $T \rightarrow T \bullet * F$

Reduce 2

4: $T \rightarrow T * F \bullet$

Reduce 3

8: $F \rightarrow id \bullet$

5: $F \rightarrow (\bullet T)$
 $T \rightarrow \bullet F$
 $T \rightarrow \bullet T * F$
 $F \rightarrow \bullet id$
 $F \rightarrow \bullet (T)$

F

F

T

id

*

F

id

id

(

T

(

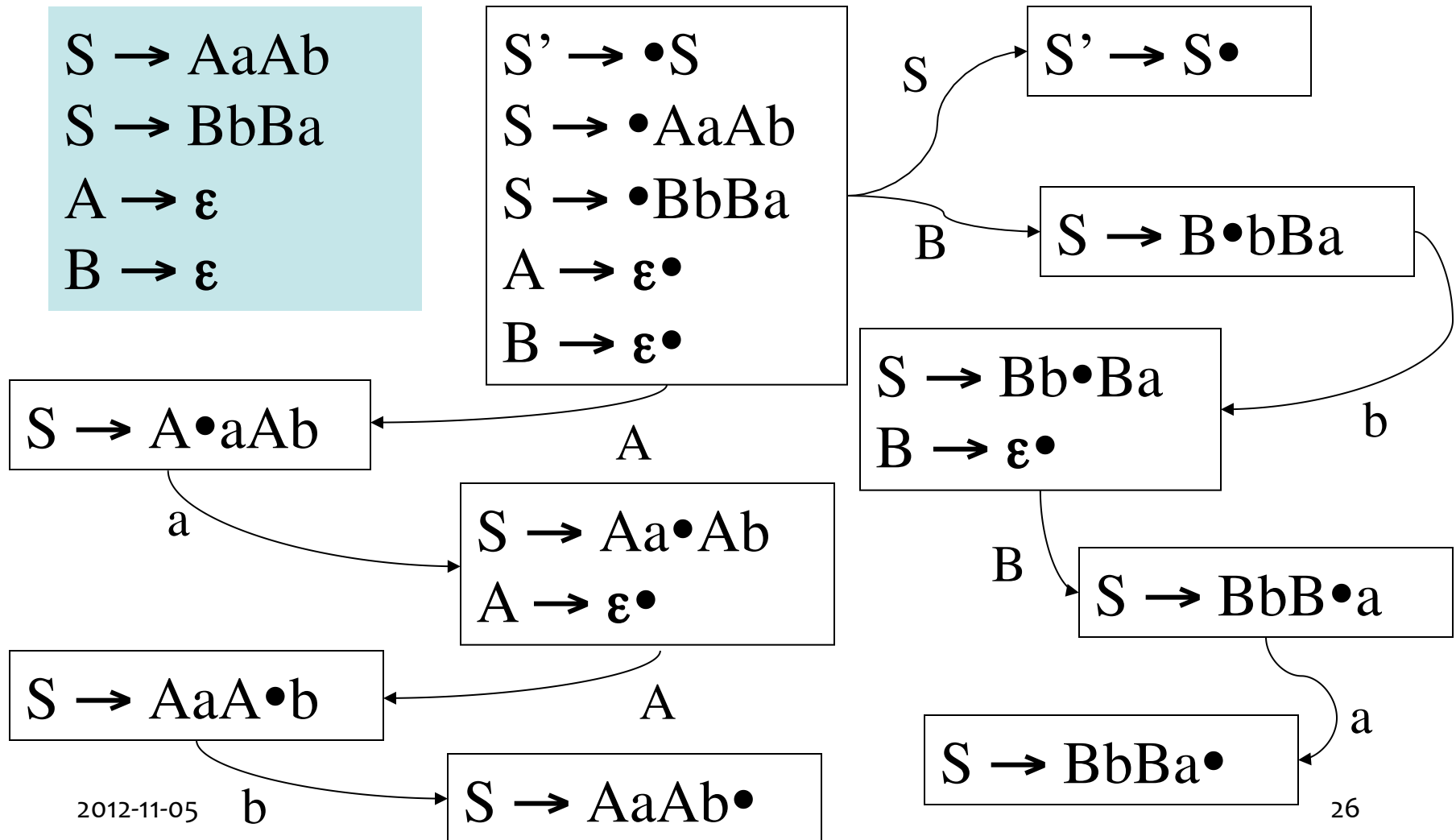
LR(0) Construction

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:

$S' \rightarrow T$

$T \rightarrow F$

$T \rightarrow T * F$

$T \rightarrow id$

$F \rightarrow id \mid (T)$

$F \rightarrow id = T ;$

11: $F \rightarrow id \bullet$

$F \rightarrow id \bullet = T$

Shift/reduce conflict

1: $F \rightarrow id \bullet$

$T \rightarrow id \bullet$

Reduce/Reduce conflict

Need more lookahead: SLR(1)

LR(o) Grammars

- An LR(o) grammar is a CFG such that the LR(o) construction produces a table without conflicts (a deterministic pushdown automata)
- $S \Rightarrow_{rm}^* \alpha A \beta \Rightarrow_{rm} \alpha w \beta$ and $A \rightarrow w$ then we can *prune the handle* w
 - pruning the handle means we can reduce αw to αA on the stack
- Every viable prefix αw can be recognized using the DFA built by the LR(o) construction

LR(o) Grammars

- Once we have a viable prefix on the stack, we can prune the handle and then restart the DFA to obtain another viable prefix, and so on ...
- In LR(o) pruning the handle can be done without any look-ahead
 - this means that in the rightmost derivation,
 - $S \Rightarrow^*_{rm} \alpha A \beta \Rightarrow_{rm} \alpha w \beta$ we reduce using a unique rule $A \rightarrow w$ without ambiguity, and without looking at β
- No ambiguous context-free grammar can be LR(o)

LR(o) Grammars \subset Context-free Grammars

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: $F \rightarrow id \bullet$
 $F \rightarrow id \bullet ++$
 $C \rightarrow id \bullet$

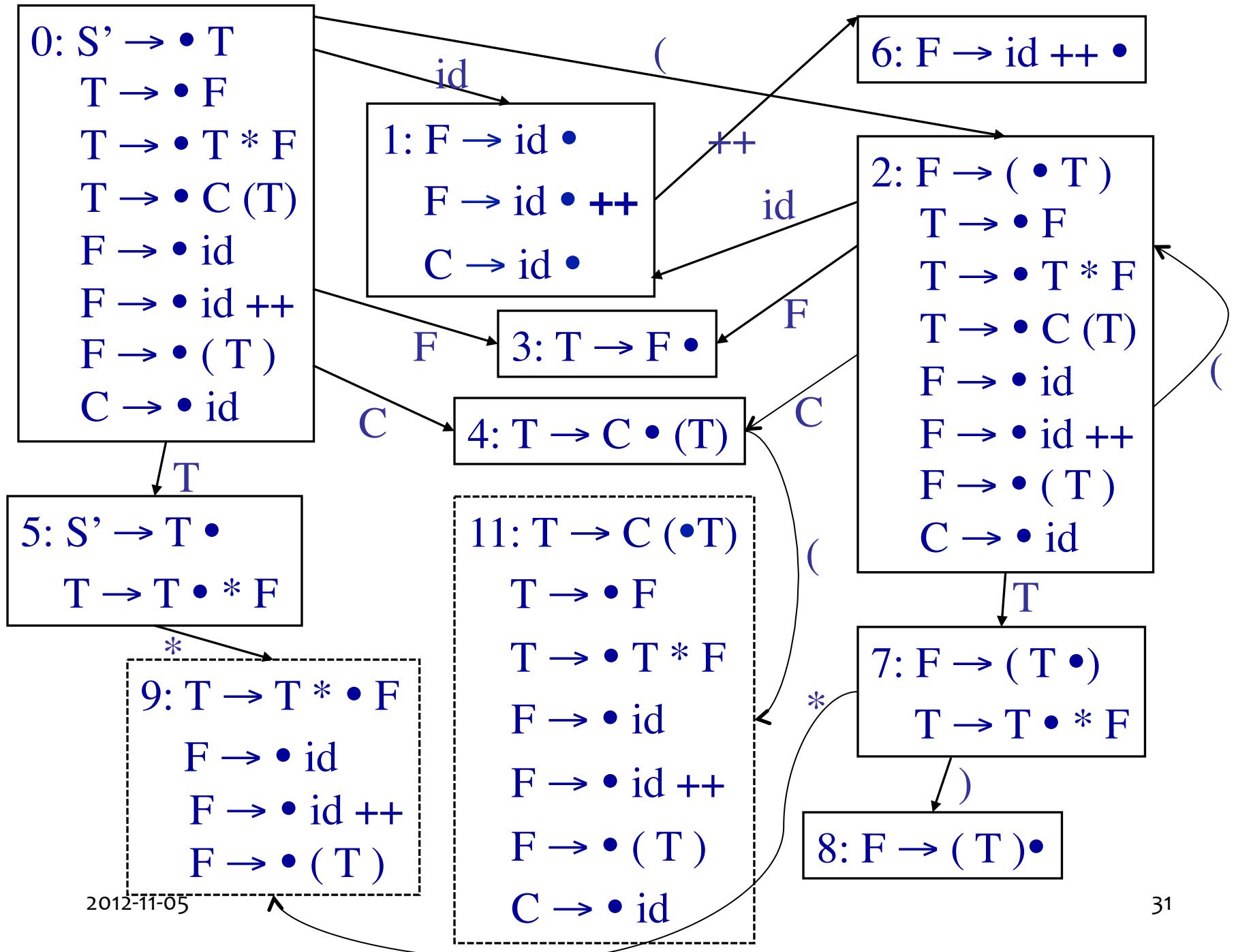
$\text{Follow}(F) = \{ *,), \$ \}$

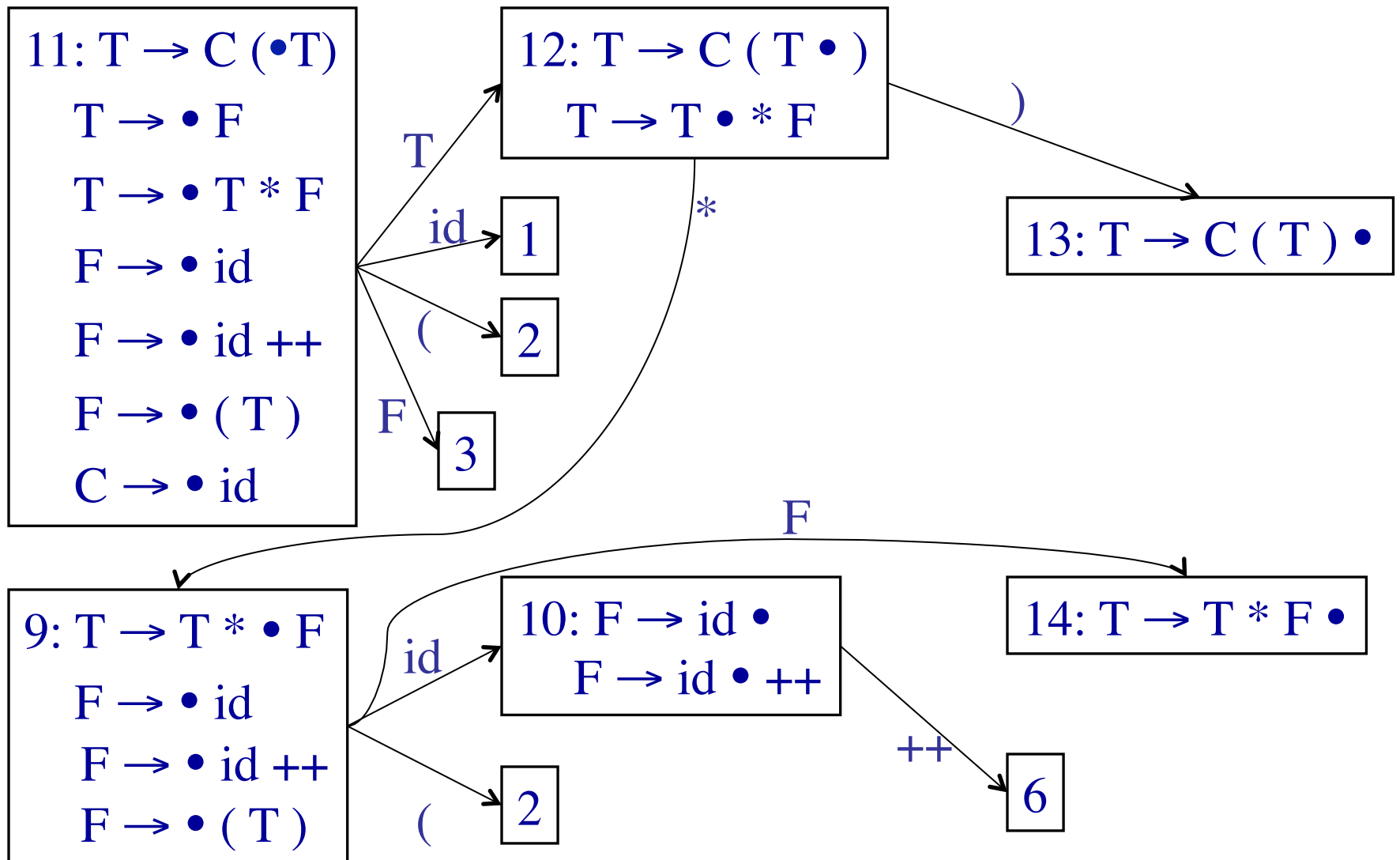
$\text{Follow}(C) = \{ (\}$

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

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

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





Productions	
1	$T \rightarrow F$
2	$T \rightarrow T * F$
3	$T \rightarrow C(T)$
4	$F \rightarrow id$
5	$F \rightarrow id ++$
6	$F \rightarrow (T)$
7	$C \rightarrow id$

	*	()	id	++	\$	T	F	C
0		S2		S1			5	3	4
1	R4	R7	R4		S2	R4			
2		S2		S1			7	3	4
3	R1		R1			R1			
4		S11							
5	S9					A			
6	R5		R5			R5			
7	S9		S8						
8	R6		R6			R6			
9		S2		S10				14	
10	R4		R4		S6	R4			
11		S2		S1			12	3	
12	S9		S13						
13	R3		R3			R3			
14	R2		R2			R2			

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)

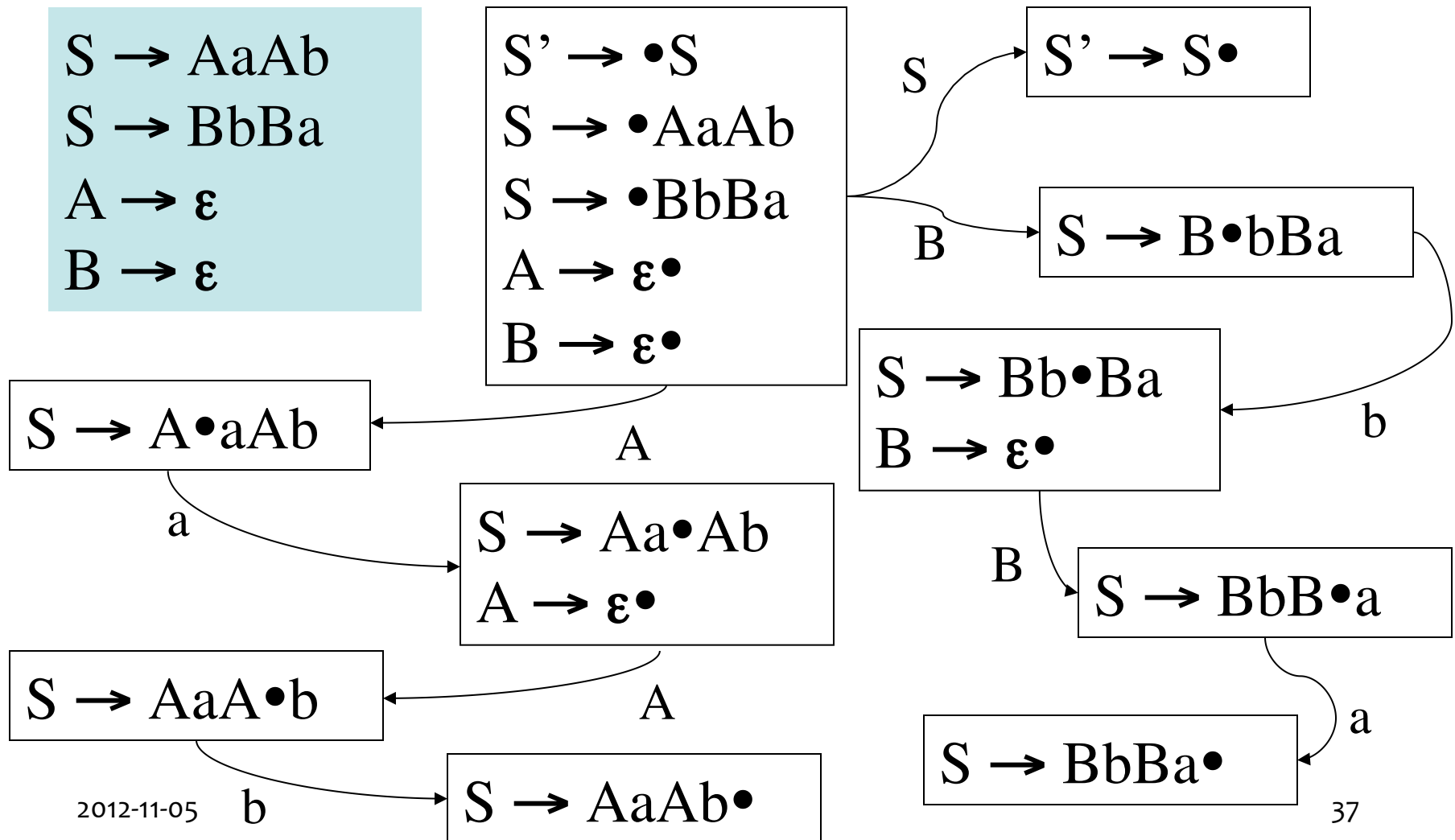
4. All entries not defined are errors
 5. Make sure I_0 is the initial state
- Note: SLR(1) only reduces $\{A \rightarrow \alpha \bullet\}$ if lookahead in $\text{Follow}(A)$
 - Shift and reduce items or more than one reduce item can be in the same configuration set as long as lookaheads are disjoint

SLR(1) Conditions

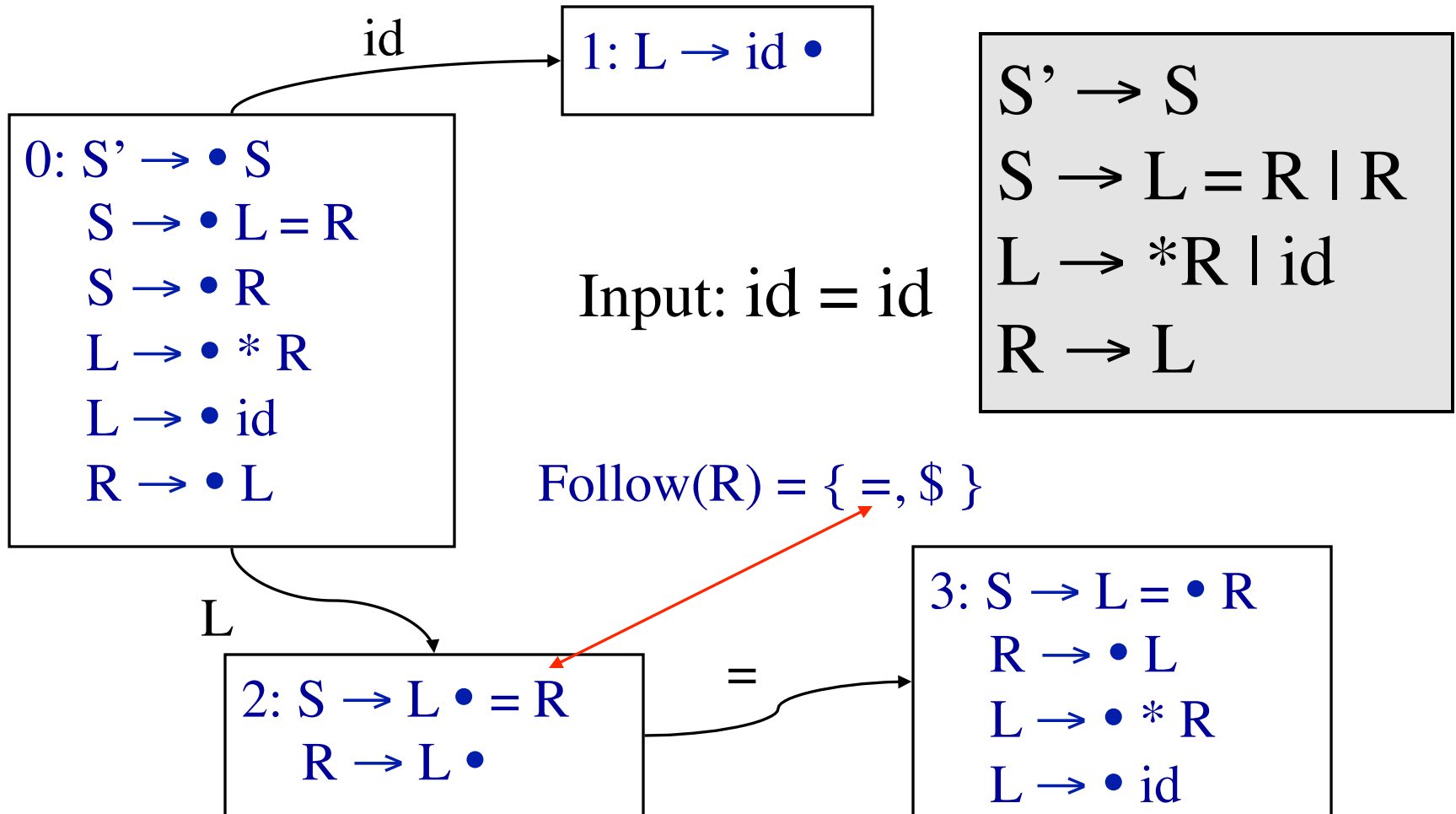
- A grammar is SLR(1) if for each configuration set:
 - For any item $\{A \rightarrow \alpha \bullet x \beta : x \in T\}$ there is no $\{B \rightarrow \gamma \bullet : x \in \text{Follow}(B)\}$
 - For any two items $\{A \rightarrow \alpha \bullet\}$ and $\{B \rightarrow \beta \bullet\}$ $\text{Follow}(A)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

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

Is this grammar SLR(1)?



SLR limitation: lack of context



$$S' \rightarrow S$$

$$S \rightarrow L = R \mid R$$

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

$$R \rightarrow L$$

$$\text{Follow}(R) = \{ =, \$ \}$$

$$2: S \rightarrow L \bullet = R$$

$$R \rightarrow L \bullet$$

Find all lookaheads
for reduce $R \rightarrow L \bullet$

$$S' \rightarrow S \rightarrow R \rightarrow L \rightarrow \text{id}$$

\$

$$S' \rightarrow S \rightarrow L = R \rightarrow L \rightarrow \text{id}$$

\$

$$S' \rightarrow S \rightarrow R \rightarrow L \rightarrow *R \rightarrow L \rightarrow \text{id}$$

\$

$$S' \rightarrow S \rightarrow L = R \rightarrow L \rightarrow \text{id}$$

\$

Problem?

No! $R \rightarrow L \bullet$ reduce
and $S \rightarrow L \bullet = R$ do
not co-occur due to
the $L \rightarrow *R$ rule

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

$S \rightarrow B B$

$B \rightarrow a B \mid b$

$S \Rightarrow BB \Rightarrow BaB \Rightarrow Bab$
 $\Rightarrow aBab \Rightarrow aaBab \Rightarrow aaBab$

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

$S \Rightarrow BB \Rightarrow BaB \Rightarrow BaaB$

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

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

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

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

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

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

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

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

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

$S' \rightarrow S$

$S \rightarrow L = R \mid R$

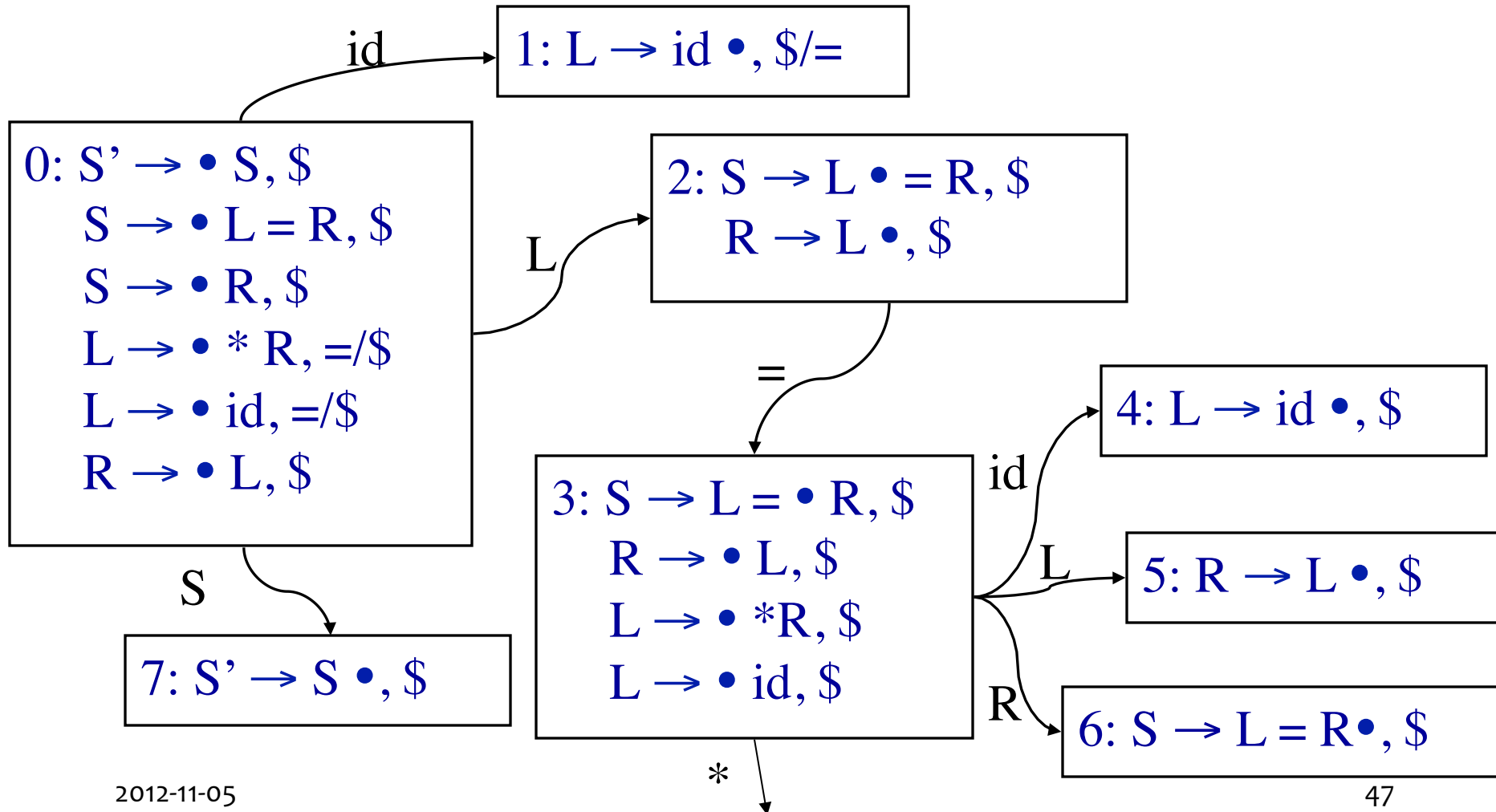
$L \rightarrow *R \mid \text{id}$

$R \rightarrow L$

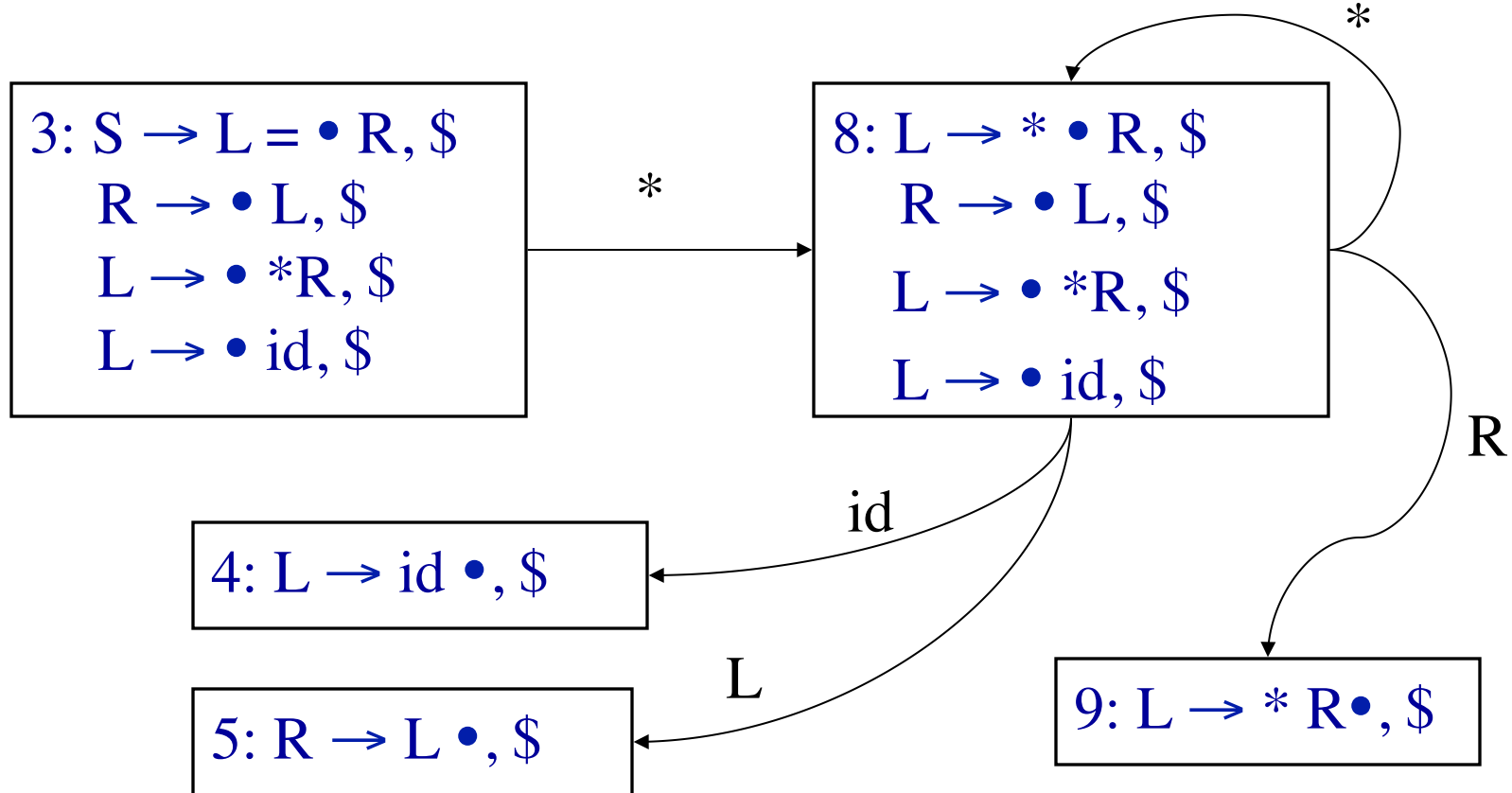
LR(1) Successor(C, X)

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

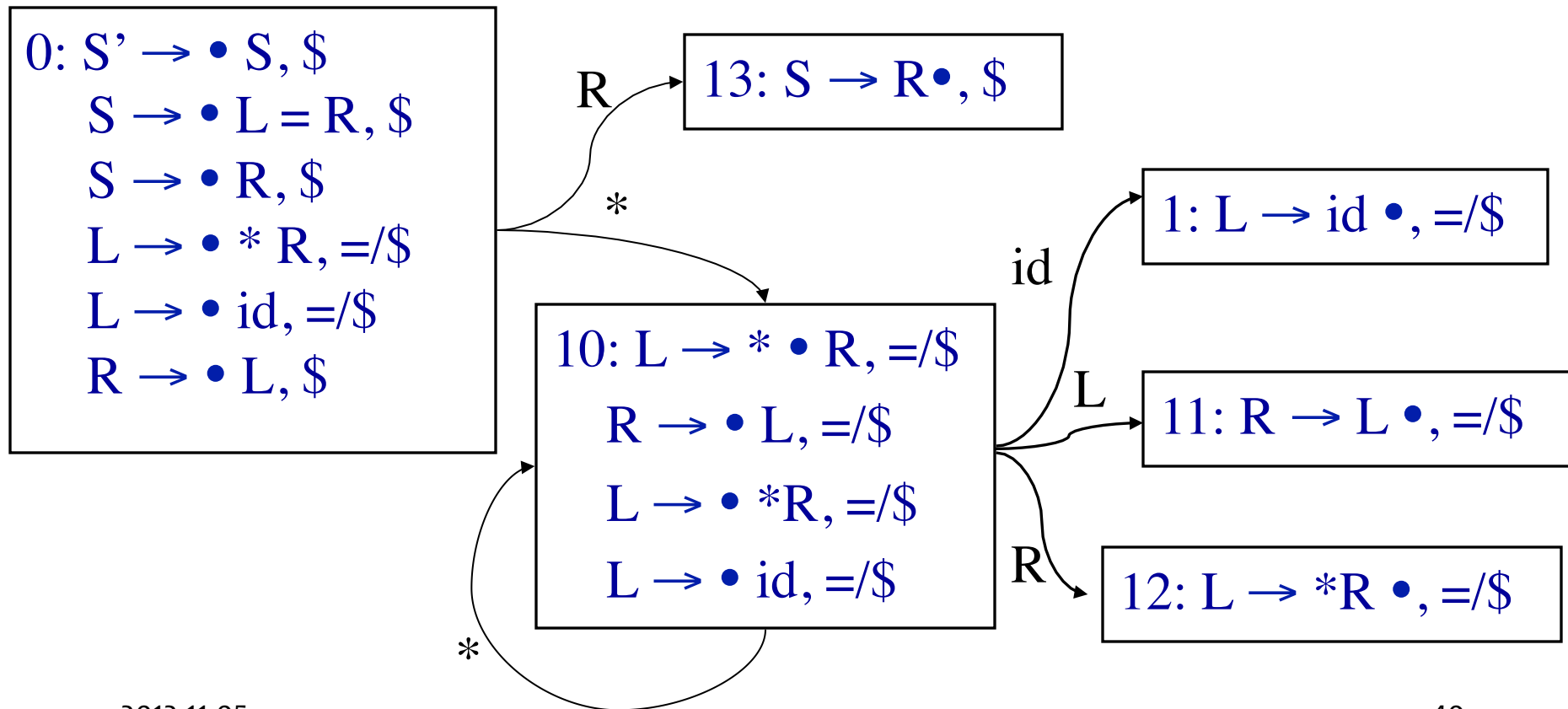
LR(1) Example



LR(1) Example (contd)



LR(1) Example (contd)



Productions	
1	$S \rightarrow L = R$
2	$S \rightarrow R$
3	$L \rightarrow * R$
4	$L \rightarrow \text{id}$
5	$R \rightarrow L$

	id	=	*	\$	S	L	R
0	S1		S10		7	2	13
1		R4		R4			
2		S3		R5			
3	S4		S8			5	6
4				R4			
5				R5			
6				R1			
7				Acc			
8	S4					5	9
9				R3			
10	S1		S10			11	12
11		R5		R5			
12		R3		R3			
13				R2			

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

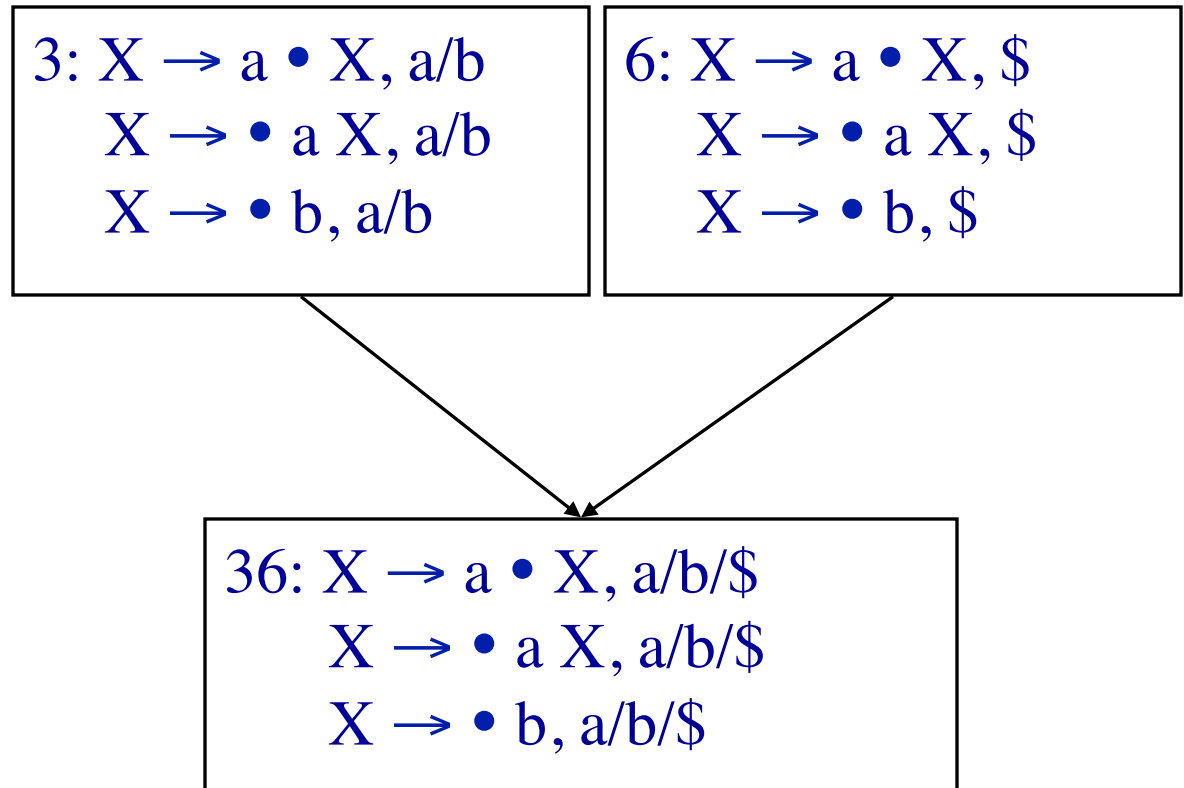
- A grammar is LR(1) if for each configuration set (itemset) the following holds:
 - For any item $[A \rightarrow \alpha \bullet x \beta, a]$ with $x \in T$ there is no $[B \rightarrow \gamma \bullet, x]$
 - For any two complete items $[A \rightarrow \gamma \bullet, a]$ and $[B \rightarrow \beta \bullet, b]$ then $a \neq b$.
- 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

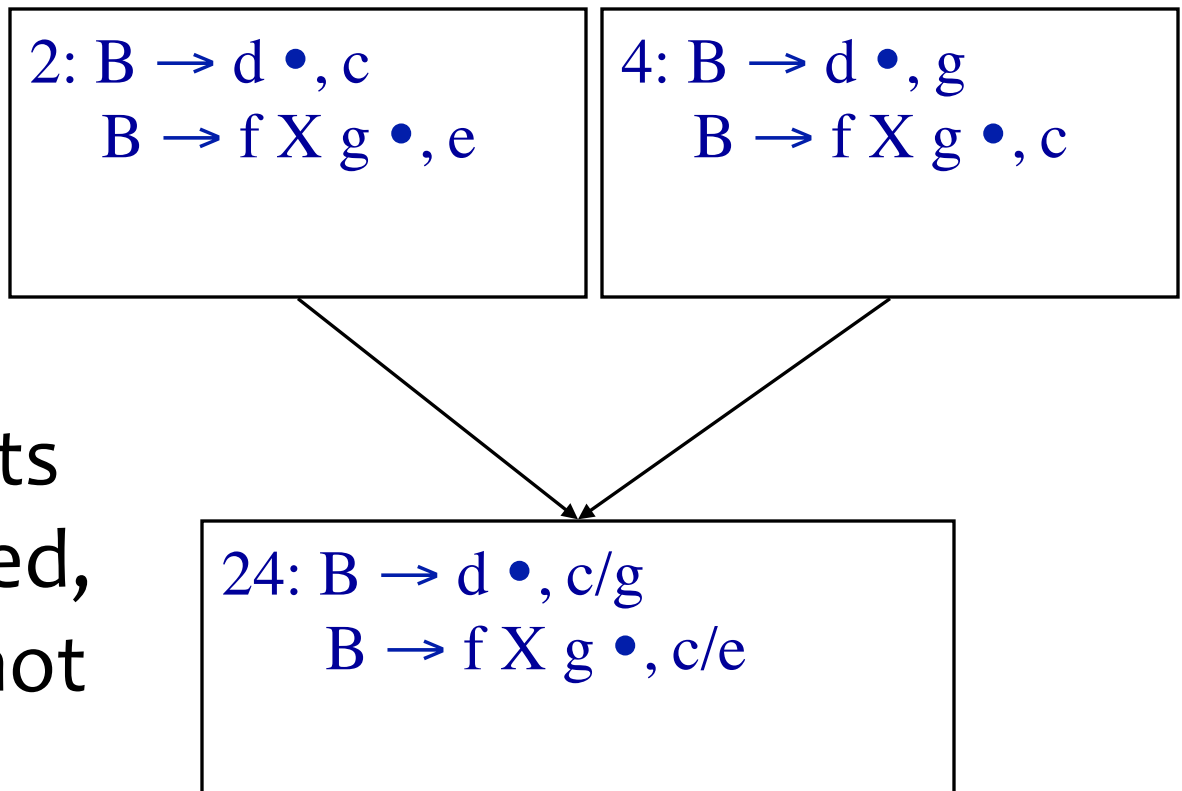
Merging States in LALR(1)

- $S' \rightarrow S$
 $S \rightarrow XX$
 $X \rightarrow aX$
 $X \rightarrow b$
- 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

- $L_x(k)$ Grammar vs. Language
 - Grammar is $L_x(k)$ if it can be parsed by $L_x(k)$ method
 - according to criteria that is specific to the method.
 - A $L_x(k)$ 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’

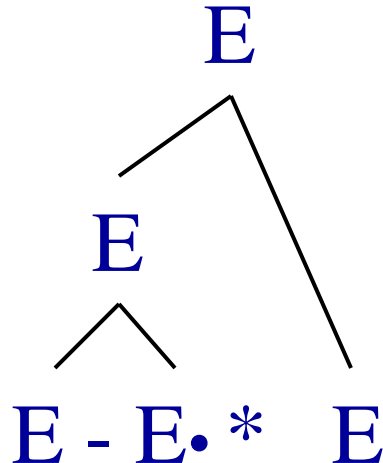
1. $S \rightarrow \text{if } E \text{ then } S$
2. $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
 - Viable prefix “if E then if E then S”
 - Then read else
 - Shift “else” (means go for 2)
 - Reduce (reduce using production #1)
 - NB: dangling else as written above is ambiguous
 - NB: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

Precedence & Associativity

- Consider

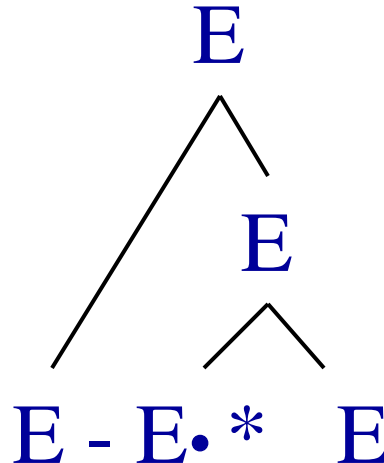
$$E \rightarrow E - E \mid E * E \mid id$$

Reduce



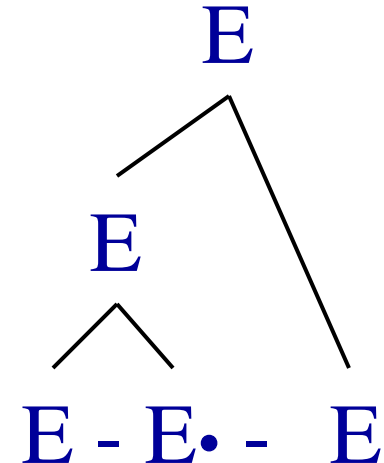
$id - id * id$

Shift



$id - id * id$

Reduce



$id - id - id$

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 $A \rightarrow w, > b$

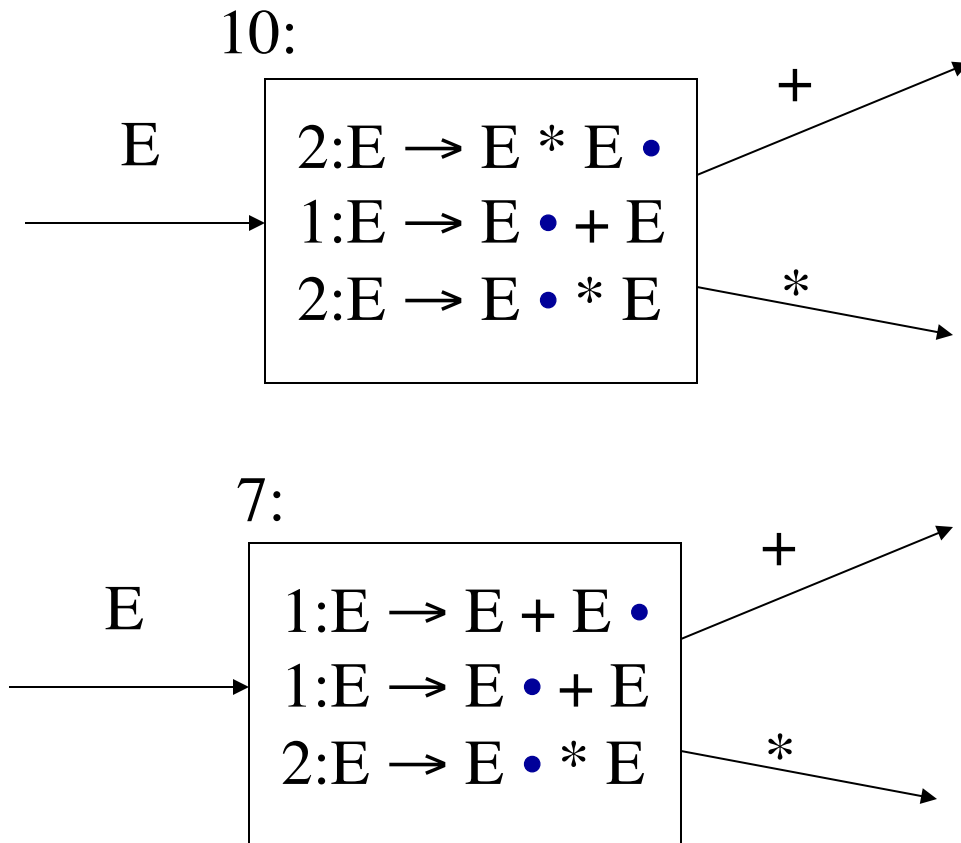
Precedence Relations

- $A \rightarrow w, < b$ 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 - Summary

- 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

Set-of-items with Epsilon rules

