LR Parsing

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

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: $S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

Input String: ccbca

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

Rightmost derivation for
id + id * id

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to - E$

$E \to \mathbf{id}$

$E \Rightarrow E * E$

$\Rightarrow E * \mathbf{id}$

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

$\Rightarrow E + \mathbf{id} * \mathbf{id}$ reduce with $E \to \mathbf{id}$

$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$ shift

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

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 is on the stack, α,w $\in$ (N U T)* 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 reduce αw to αA on the stack (called *pruning the handle* w)
  - 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

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 → F
2	T → T*F
3	F → id
4	F → (T)

Action/Goto Table

	*	(	)	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 → F
2	T → T*F
3	F → id
4	F → (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		

t	Input	A
0	**(id) * id $**	**Sh**
0 5	**id) * id $**	**Sh**
0 5 8	**) * id $**	**Re po**
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**

Trace “(id)*id”

Stack	Input	Action
0 1	*** id $**	**Reduce 1 T→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→id,** **pop 8, goto [3,F]=4**
0 2 3 4	**$**	**Reduce 2 T→T * F** **pop 4 3 2, goto [0,T]=2**
0 2	**$**	**Accept**

Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (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	Actio
0 1	*** id $**	**Reduc pop 1,**
0 2	*** id $**	**Shift S**
0 2 3	**id $**	**Shift S8**
0 2 3 8	**$**	**Reduce 3 F→id, pop 8, goto [3,F]=4**
0 2 3 4	**$**	**Reduce 2 T→T * F pop 4 3 2, goto [0,T]=2**
0 2	**$**	**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
- We use the notion of a dotted rule or item:

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

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

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

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

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

- This creates a configuration set (or item set)
  - Like NFA-to-DFA conversion

Closure

Closure property:

- If $\mathbf{T} \rightarrow \mathbf{X_1 \ldots X_i \bullet X_{i+1} \ldots X_n}$ is in set, and $\mathbf{X_{i+1}}$ is a nonterminal, then
  $\mathbf{X_{i+1} \rightarrow \bullet Y_1 \ldots Y_m}$ is in the set as well for all productions $\mathbf{X_{i+1} \rightarrow Y_1 \ldots Y_m}$
- 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 $S' \to S$
- Initial configuration set is

$$\text{closure}(S' \to \bullet\, S)$$

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

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

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

S' $\rightarrow \bullet$ T

T $\rightarrow \bullet$ T * F

T $\rightarrow \bullet$ F

F $\rightarrow \bullet$ id

F $\rightarrow \bullet$ (T)

S' $\rightarrow$ T
T $\rightarrow$ F | T * F
F $\rightarrow$ id | (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

$I = \{S' \rightarrow \bullet\, T,$

$T \rightarrow \bullet\, F,$

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

$F \rightarrow \bullet\, id,$

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

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

Compute **Successor**(I, "(")

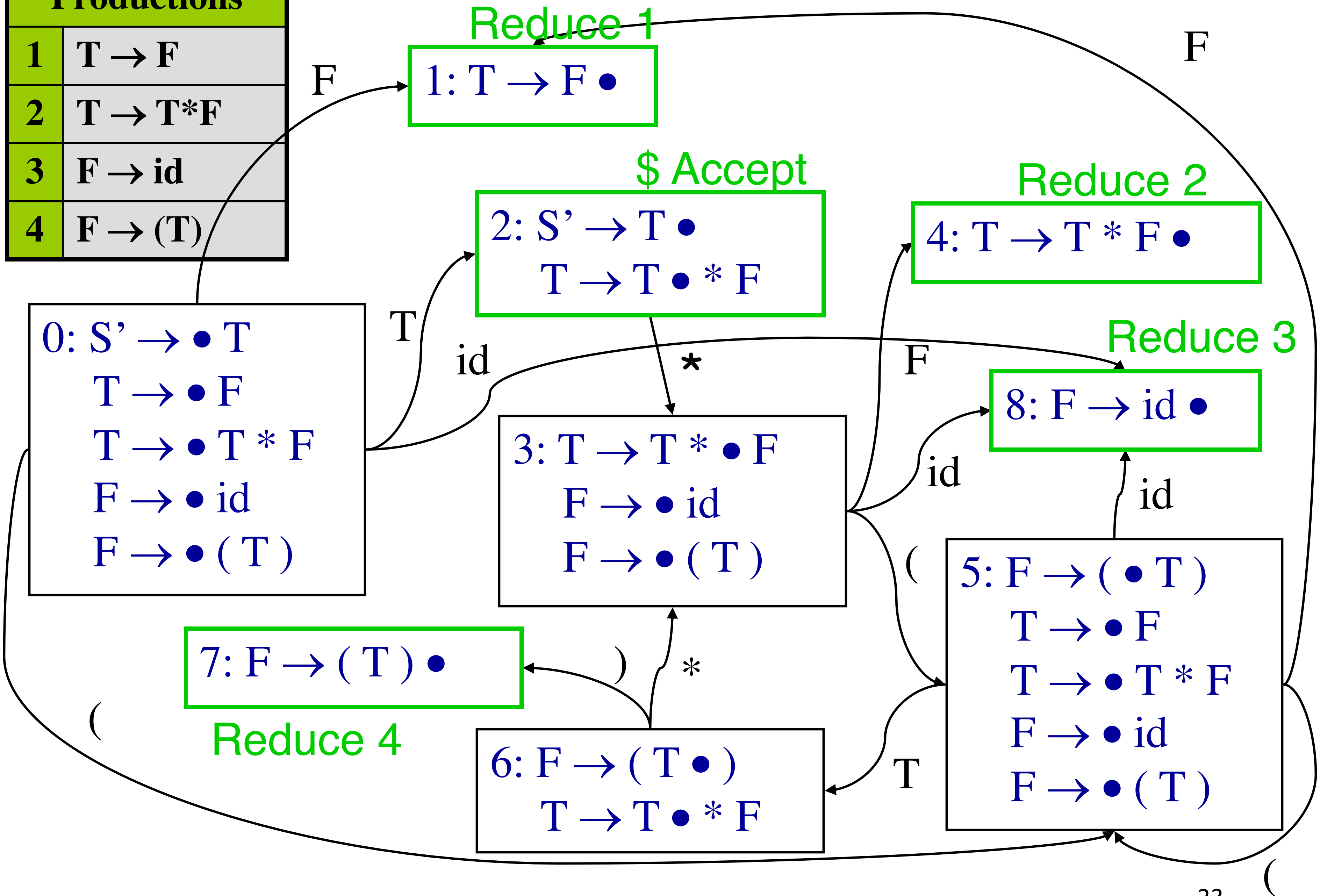
$\{ F \rightarrow (\bullet\, T), T \rightarrow \bullet\, F, T \rightarrow \bullet\, T * F,$
$F \rightarrow \bullet\, id, F \rightarrow \bullet\, (T) \}$

Sets-of-Items Construction

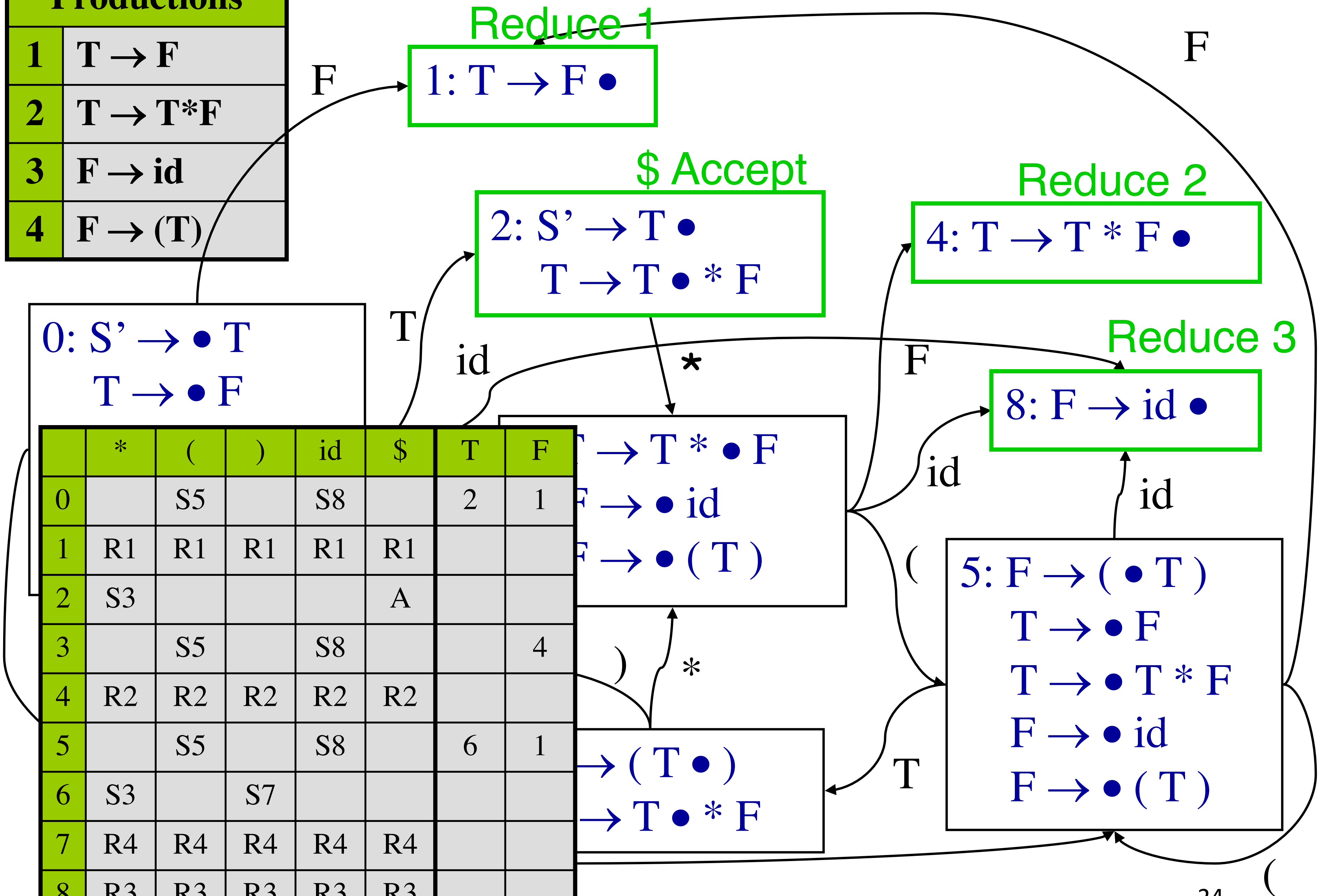
Family of configuration sets

function items(G')
    C = { closure({S' $\rightarrow \bullet$ S}) };
    do foreach I $\in$ C **do**
        foreach X $\in$ **(N $\cup$ T) do**
            C = C $\cup$ { **Successor**(I, X) };
    while C changes;

Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (T)



Productions	
1	T → F
2	T → T*F
3	F → id
4	F → (T)



	*	(	)	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		

LR(0) Construction

1. Construct $F = \{I_0, I_1, \ldots I_n\}$
2. a) if $\{A \rightarrow \alpha\bullet\} \in I_i$ and A != S'

   then action[i, _] := reduce $A \rightarrow \alpha$

   b) if $\{S' \rightarrow S\bullet\} \in I_i$

   then action[i,$] := accept

   c) if $\{A \rightarrow \alpha\bullet a\beta\} \in I_i$ and $\text{Successor}(I_i,a) = I_j$

   then action[i,a] := shift j
3. if $\text{Successor}(I_i,A) = I_j$ then goto[i,A] := j

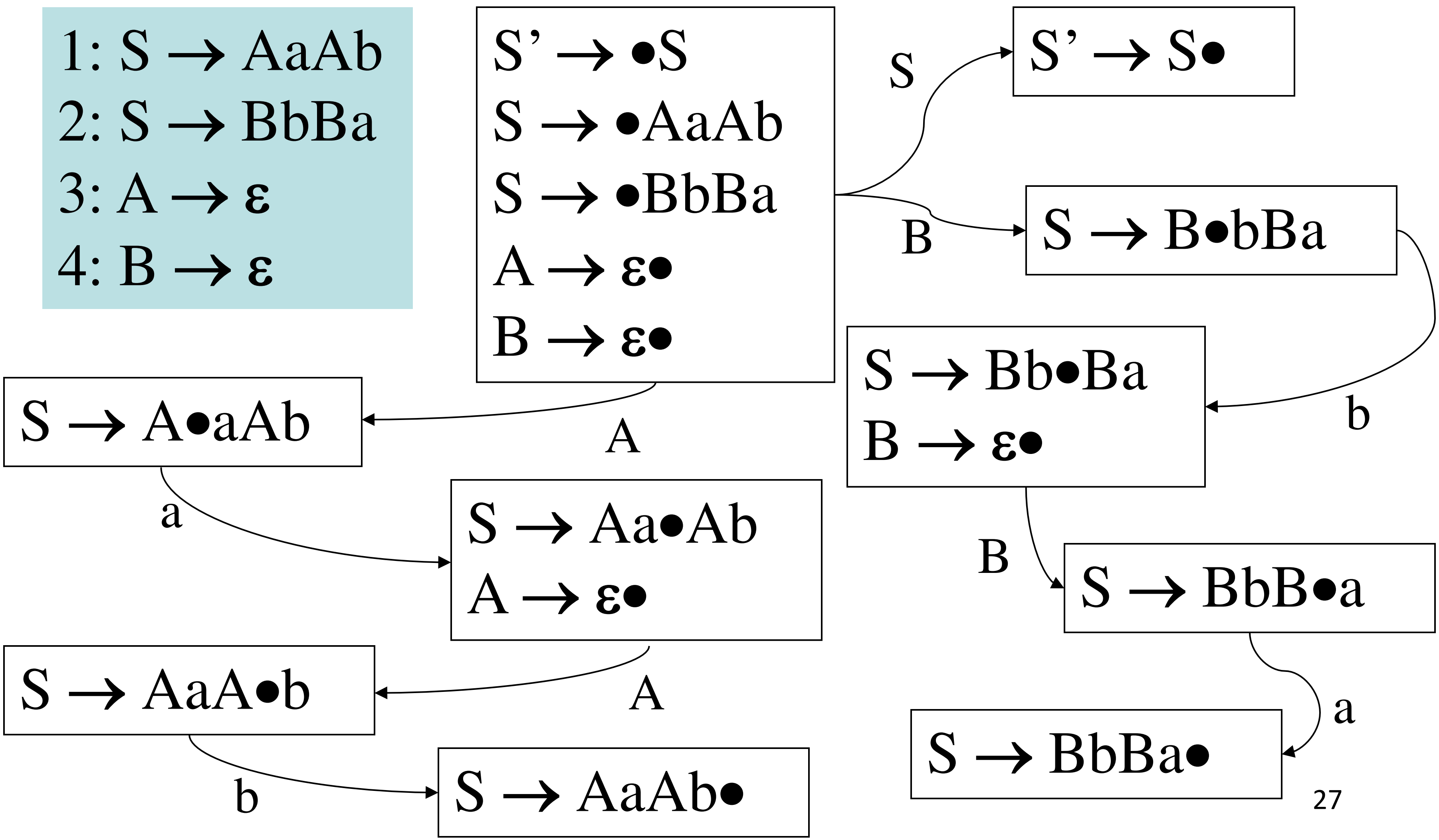
LR(0) Construction (cont'd)

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

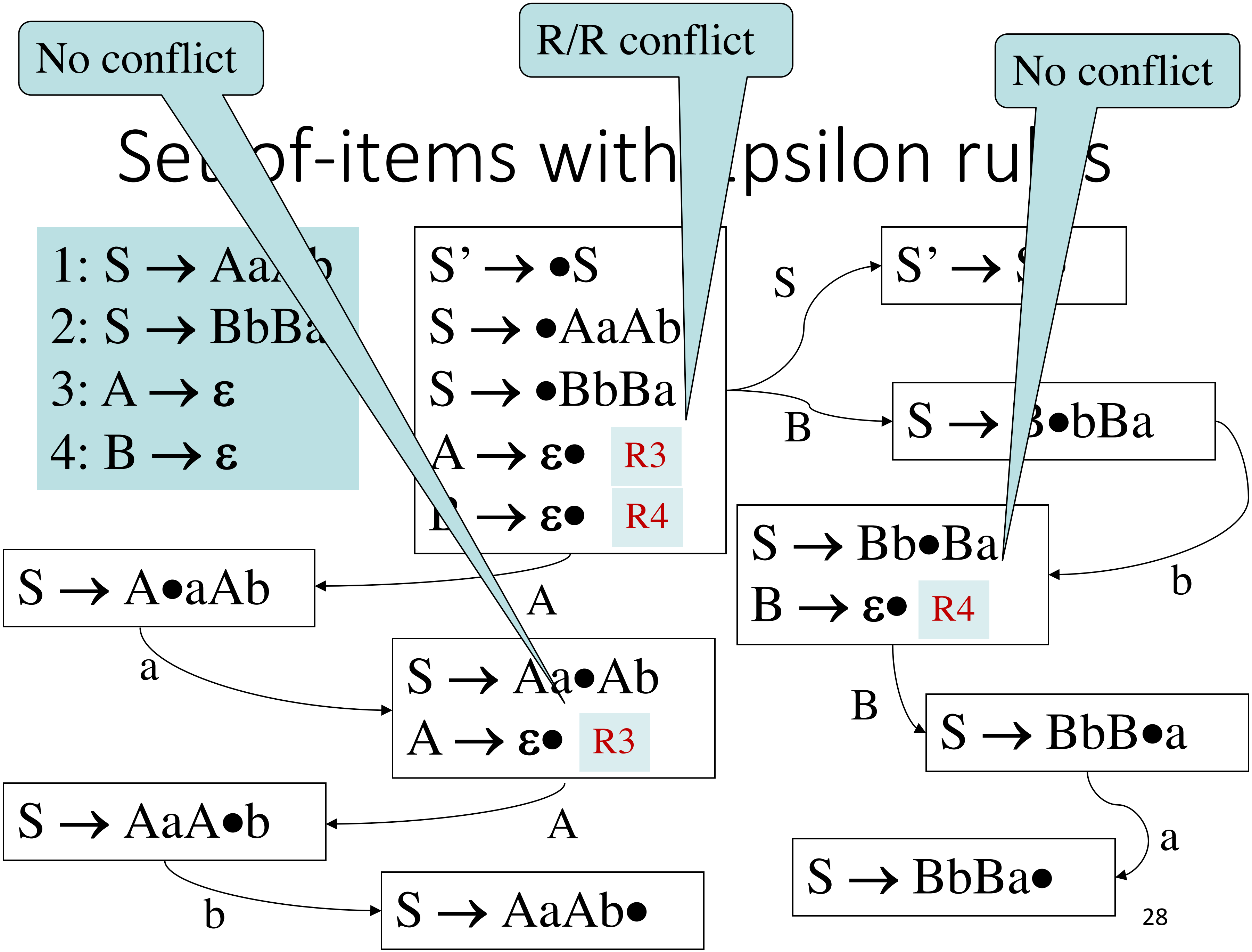
Set-of-items with Epsilon rules

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



Sets-of-items with epsilon rules

1: S → AaAb
2: S → BbBa
3: A → ε
4: B → ε



LR(0) conflicts:

S' → T
T → F
T → T * F
T → id
F → id | (T)
F → id = T ;

11: F → id •
F → id • = T
Shift/reduce conflict

1: F → id •
T → id •
Reduce/Reduce conflict

Need more lookahead: SLR(1)

Viable Prefixes

- γ is a viable prefix if there is some ω such that $\gamma | \omega$ is a state of a shift-reduce parser

$$\text{stack} \rightarrow \boxed{\gamma} \,|\, \boxed{\omega} \leftarrow \text{rest of input}$$

- Important fact: A viable prefix is a prefix of a handle
- An LR(0) item $[X \rightarrow \alpha \bullet \beta]$ says that
  - α is on top of the stack (α is a suffix of γ)
  - The parser is looking for an X
  - Expects to find input string derived from β
- We can recognize viable prefixes via a NfA (DFA)
  - States of NFA are LR(0) items
  - States of DFA are sets of LR(0) items (LR(0) states)

LR(0) Grammars

- An LR(0) grammar is a CFG such that the LR(0) 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 recognized using the DFA built by the LR(0) construction

LR(0) 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(0) 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(0)

LR(0) Grammars $\subset$ Context-free Grammars