LR3: SLR(1) Parsing

LR Parsing

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

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)

FIRST and FOLLOW

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

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

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

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

Example First/Follow

$S \rightarrow AB$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

First(A) = {c, ε}

First(B) = {c}

First(cbB) =

First(ca) = {c}

First(S) = {c}

Follow(A) = {c}

Follow(A) ∩

First(c) = {c}

Follow(B) = {$}

Follow(S) = {$}

Example First/Follow

$S \rightarrow cAa$

$A \rightarrow cB \mid B$

$B \rightarrow bcB \mid \varepsilon$

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

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

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

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

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

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

SLR(1) : Simple LR(1) Parsing

S’ → T
T → F | T * F | C (T)
F → id | id ++ | (T)
C → id

What can the next symbol be when we reduce F → id ?

$S'\$ \Rightarrow T\$ \Rightarrow F\$ \Rightarrow id\underline{\$}$

$S'\$ \Rightarrow T\$ \Rightarrow T{*}F\$ \Rightarrow T{*}id\$ \Rightarrow F{*}id\$ \Rightarrow id\underline{*}id\$$

$S'\$ \Rightarrow T\$ \Rightarrow C(T)\$ \Rightarrow C(F)\$ \Rightarrow C(id\underline{)}\$$

The top of stack will be id and the next input symbol will be either $, or * or)

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

SLR(1) : Simple LR(1) Parsing

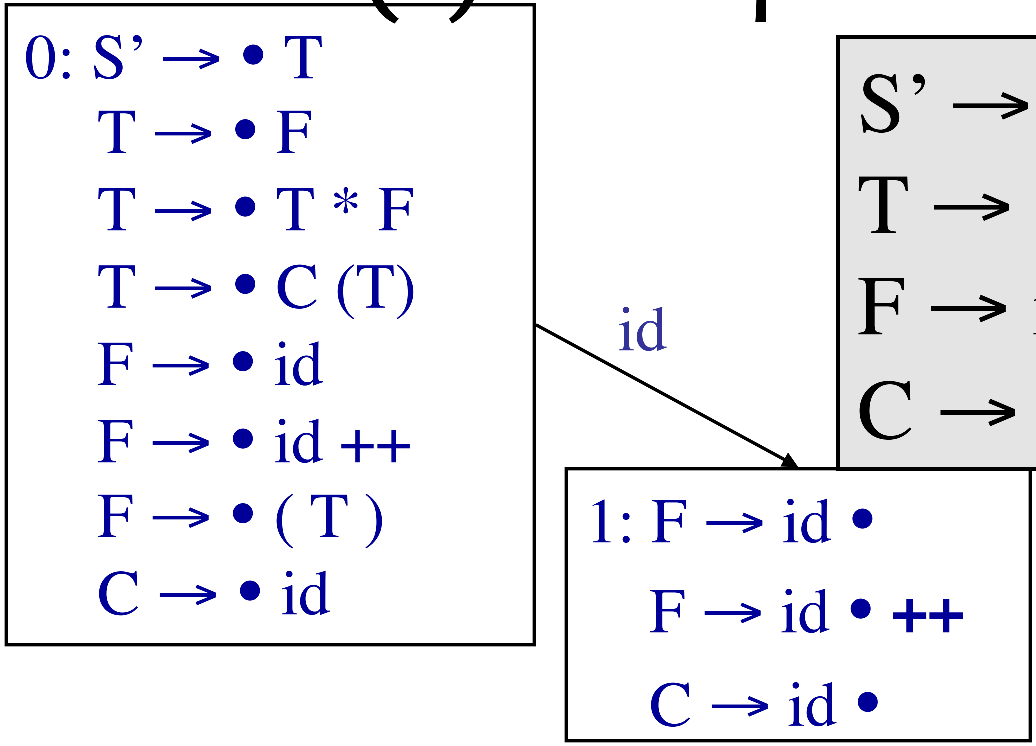
S’ → T
T → F | T * F | C (T)
F → id | id ++ | (T)
C → id

What can the next symbol be when we reduce C → id ?

$$S'\$ \Rightarrow T\$ \Rightarrow C(T)\$ \Rightarrow C(F)\$ \Rightarrow C(id) \Rightarrow id\underline{(}id)\$$$

Follow(C) = { (}

SLR(1) : Simple LR(1) Parsing



0: S’ → • T
T → • F
T → • T * F
T → • C (T)
F → • id
F → • id ++
F → • (T)
C → • id

id

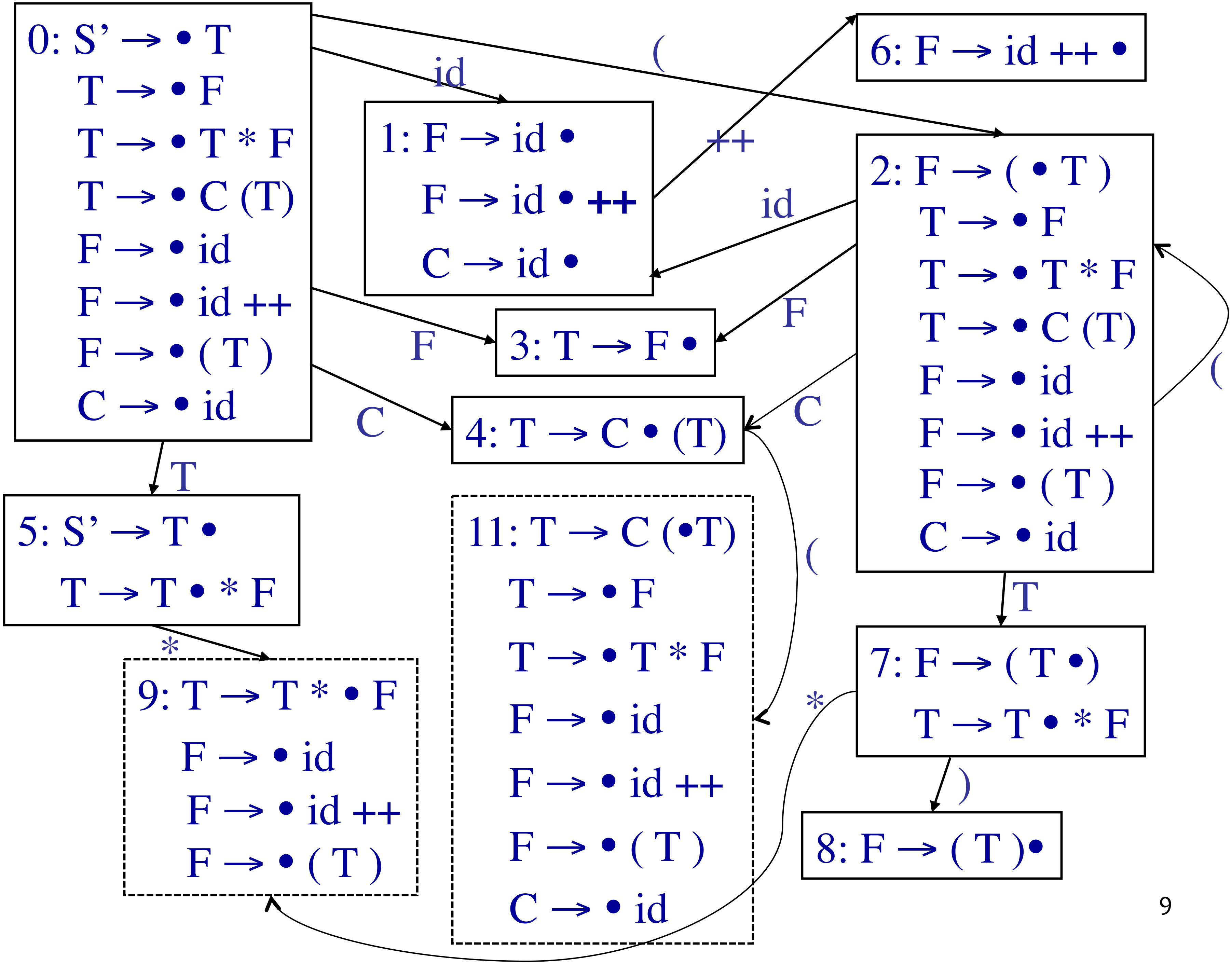
1: F → id •
F → id • ++
C → id •

S’ → T
T → F | T * F | C (T)
F → id | id ++ | (T)
C → id

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

Follow(C) = { (}

action[1,*]= action[1,)] = action[1,$] = Reduce F → id
action[1,(] = Reduce C → id
action[1,++] = Shift



0: S’ → • T
T → • F
T → • T * F
T → • C (T)
F → • id
F → • id ++
F → • (T)
C → • id

1: F → id •
F → id • ++
C → id •

2: F → (• T)
T → • F
T → • T * F
T → • C (T)
F → • id
F → • id ++
F → • (T)
C → • id

3: T → F •

4: T → C • (T)

5: S’ → T •
T → T • * F

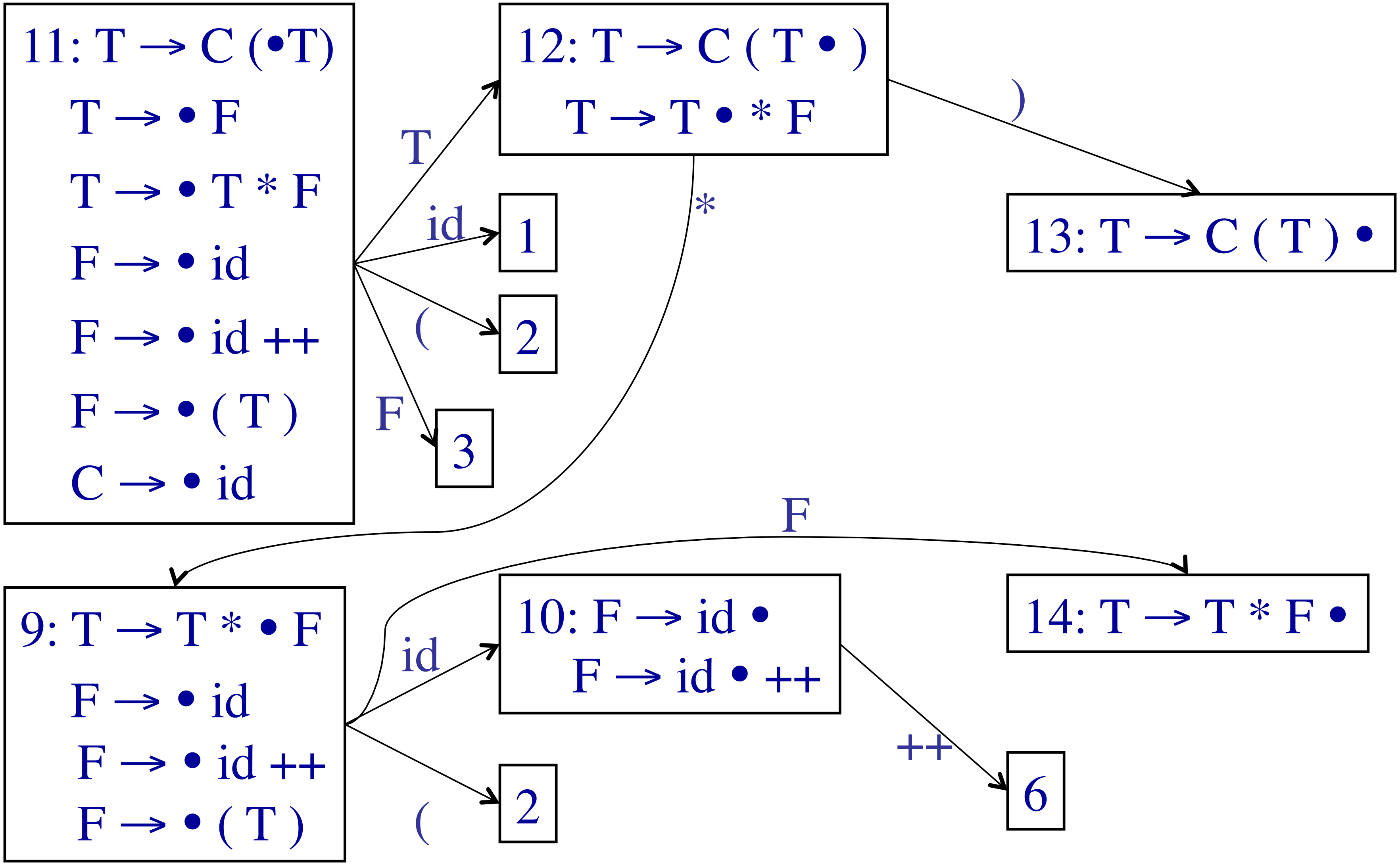
6: F → id ++ •

7: F → (T •)
T → T • * F

8: F → (T)•

9: T → T * • F
F → • id
F → • id ++
F → • (T)

11: T → C (•T)
T → • F
T → • T * F
F → • id
F → • id ++
F → • (T)
C → • id



11: T → C (•T)
T → • F
T → • T * F
F → • id
F → • id ++
F → • (T)
C → • id

12: T → C (T •)
T → T • * F

13: T → C (T) •

9: T → T * • F
F → • id
F → • id ++
F → • (T)

10: F → id •
F → id • ++

14: T → T * F •

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

If there is still conflicts under
These rules, grammar is not SLR

SLR Parsing

- Assume:
  - Stack contains α and next input is t
  - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta \bullet$
  - $t \in \text{Follow}(X)$
- Shift if
  - s contains item $X \rightarrow \beta \bullet t\omega$
  - If $X \rightarrow \beta \bullet$ is in s then t cannot be in Follow(X)

S’ → E

E → T + E

E → T

T → int

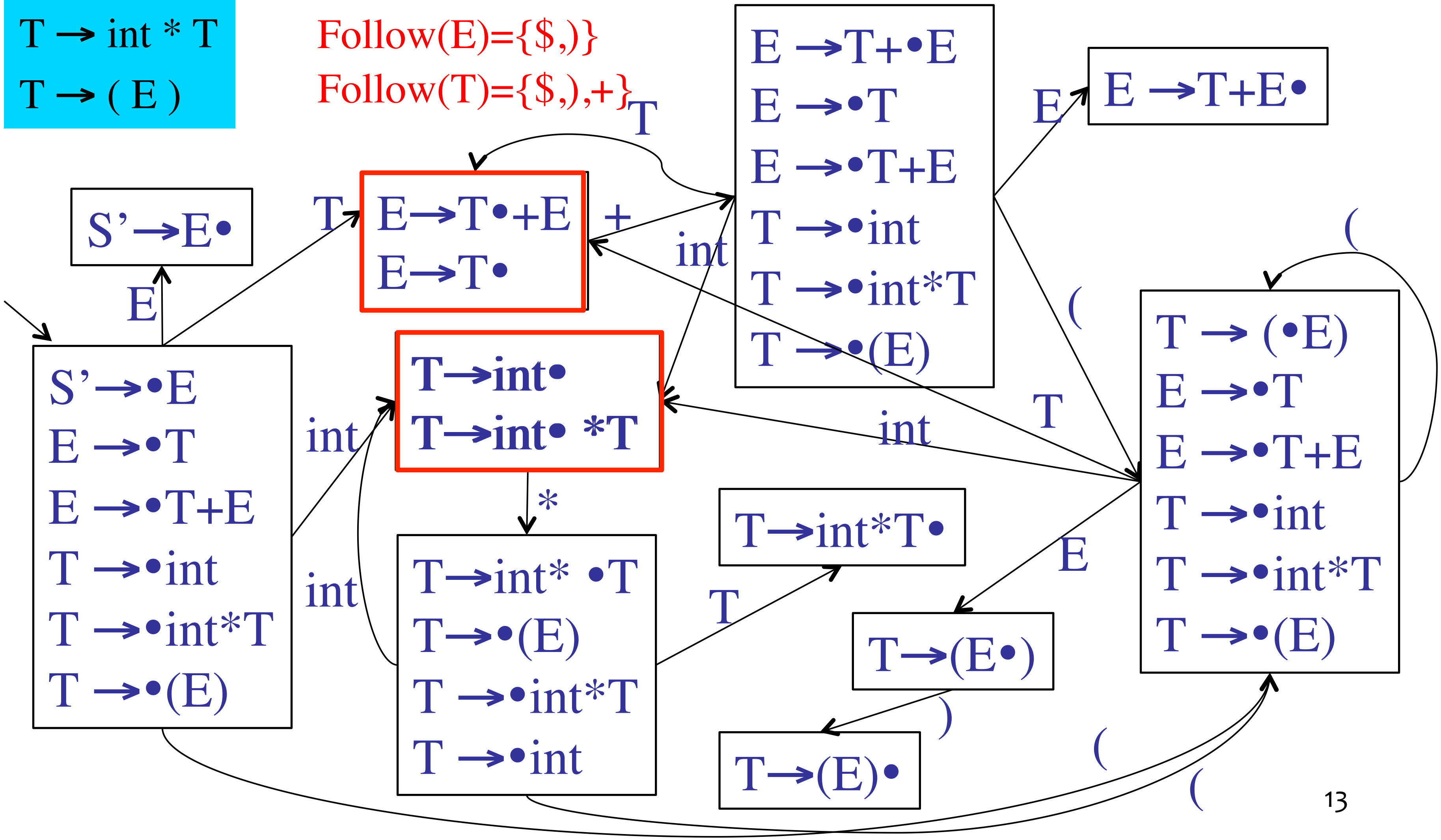
T → int * T

T → (E)

SLR Parsing

Follow(E)={$,)}

Follow(T)={$,),+}



S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

S’→E•

E→T•+E
E→T•

T→int•
T→int• *T

T→int* •T
T→•(E)
T →•int*T
T →•int

E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

E →T+E•

T→int*T•

T→(E•)

T→(E)•

T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

SLR Parsing

- Let M be DFA for viable prefixes of G
- Let $|x_1 \ldots x_n\$$ be initial configuration
- Repeat until configuration is $S|\$$
  - Let $\alpha|\ \omega$ be current configuration
  - Run M on current stack α
  - If M rejects α, report parsing error
    - Stack α is not a viable prefix
  - If M accepts α with items I, let a be the next input
    - Shift $[X \rightarrow \beta \bullet a\ \gamma] \in I$
    - Reduce if $[X \rightarrow \beta \bullet] \in I$ and $a \in \text{Follow}(X)$
    - Report parsing error if neither applies

If there is any conflict in the last step (more than two valid action), grammar is not SLR(k) in practice k=1

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $		

S’ → E

E → T + E

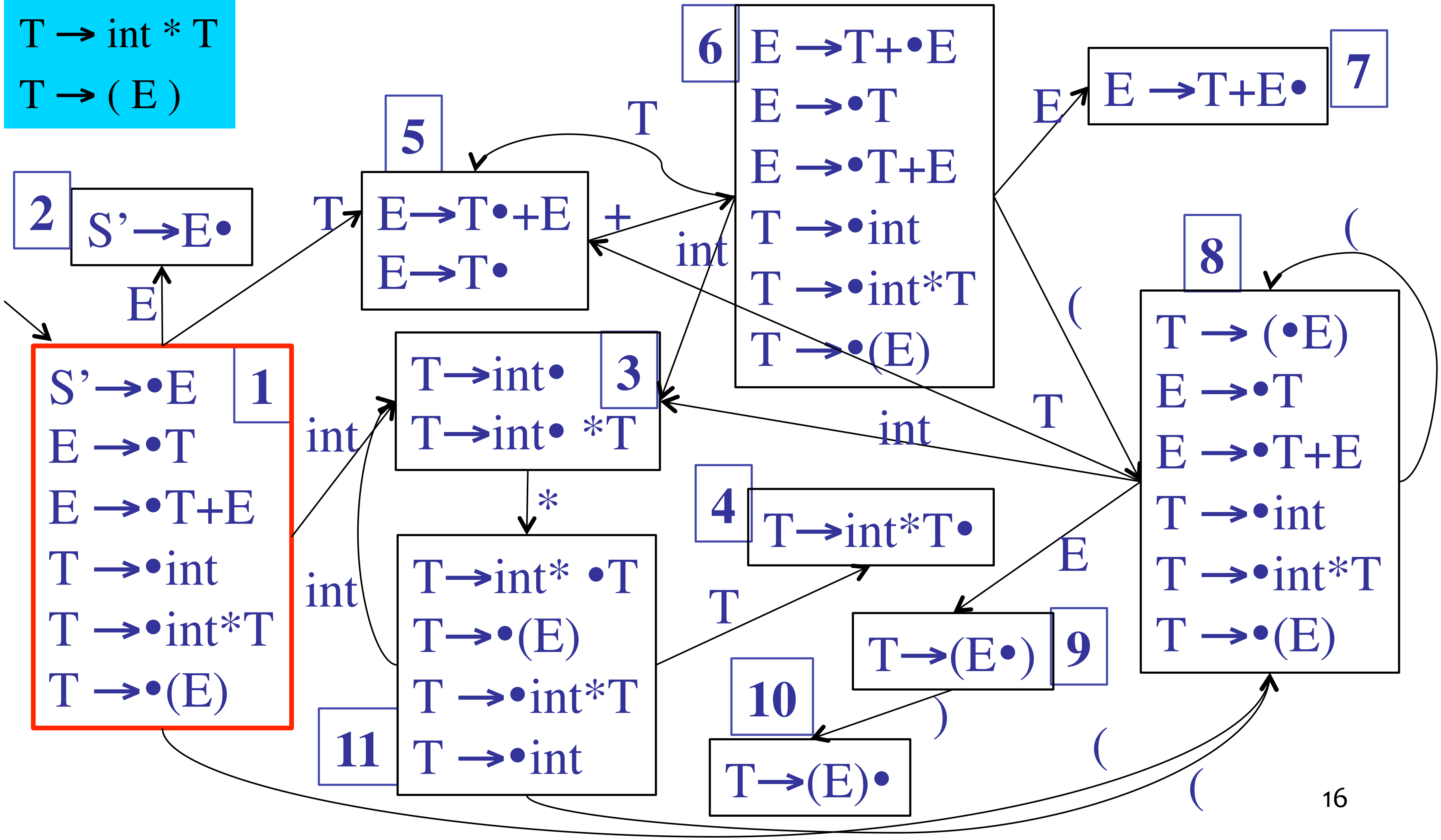
E → T

T → int

T → int * T

T → (E)

| int * int $



1
S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

2
S’→E•

3
T→int•
T→int• *T

4
T→int*T•

5
E→T•+E
E→T•

6
E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

7
E →T+E•

8
T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

9
T→(E•)

10
T→(E)•

11
T→int* •T
T→•(E)
T →•int*T
T →•int

Trace ‘int*int’

<table>
<tr><th>configuration (Stack|input)</th><th>DFA halt state</th><th>Action</th></tr>
<tr><td>| int * int $</td><td>1</td><td>Shift</td></tr>
<tr><td>int | * int $</td><td></td><td></td></tr>
</table>

S’ → E

E → T + E

E → T

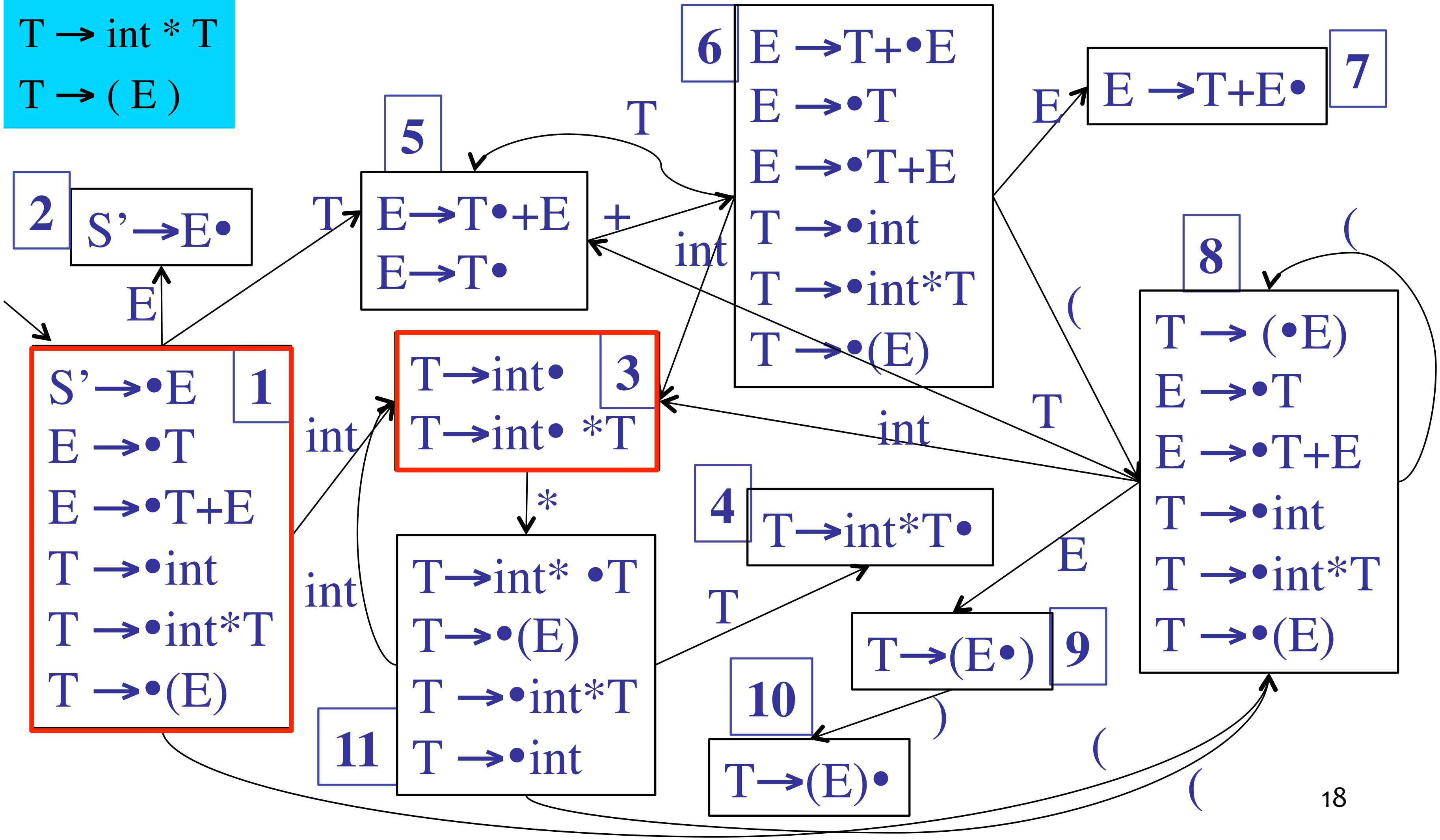
T → int

T → int * T

T → (E)

Follow(T)={$,),+}

int | * int $



1
S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

2
S’→E•

3
T→int•
T→int• *T

4
T→int*T•

5
E→T•+E
E→T•

6
E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

7
E →T+E•

8
T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

9
T→(E•)

10
T→(E)•

11
T→int* •T
T→•(E)
T →•int*T
T →•int

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $	1	Shift
int | * int $	3 * ∉ Follow(T)	Shift
int * | int $		

S’ → E

E → T + E

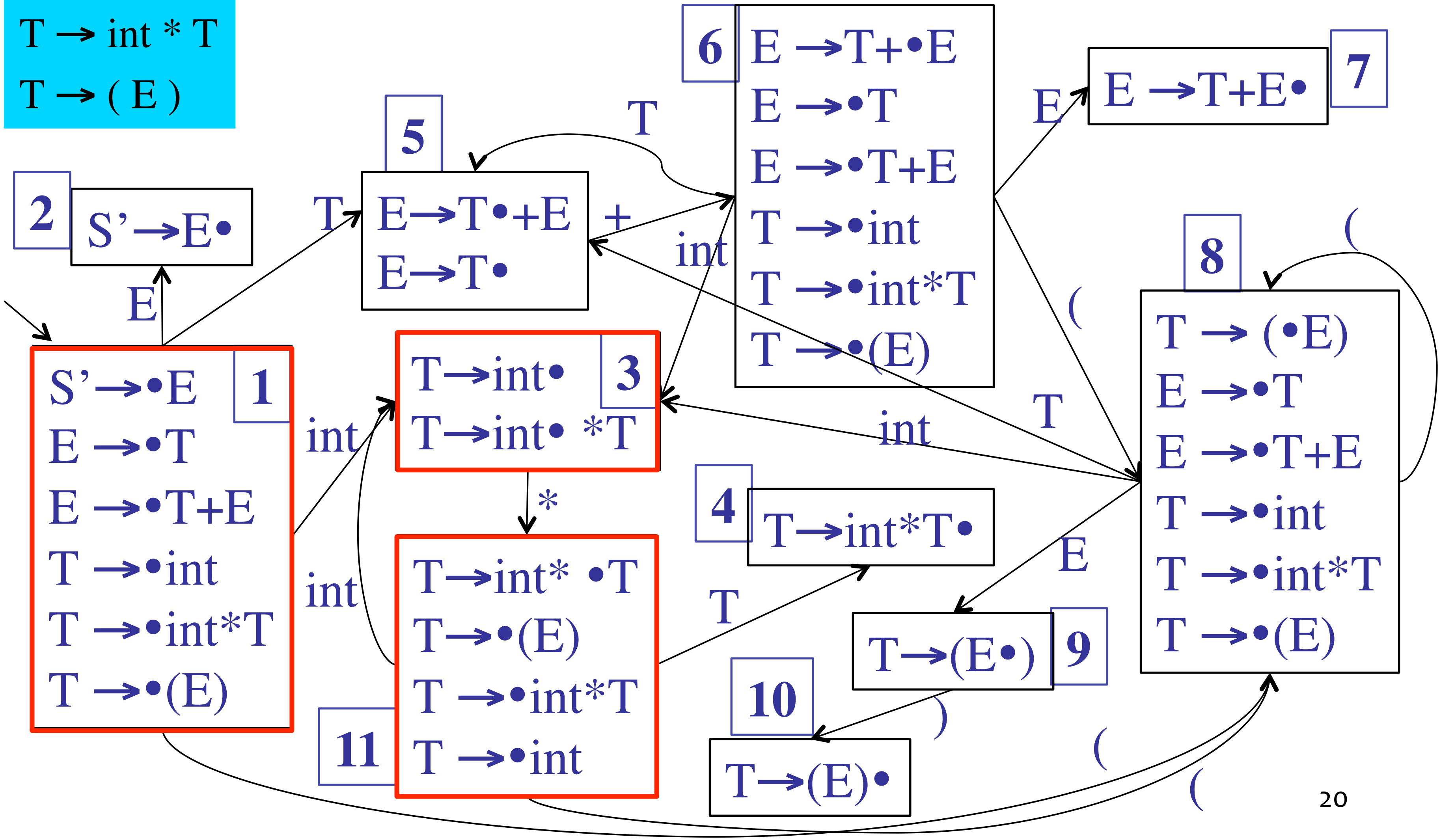
E → T

T → int

T → int * T

T → (E)

int * | int $



1
S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

2
S’→E•

3
T→int•
T→int• *T

4
T→int*T•

5
E→T•+E
E→T•

6
E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

7
E →T+E•

8
T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

9
T→(E•)

10
T→(E)•

11
T→int* •T
T→•(E)
T →•int*T
T →•int

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $	1	Shift
int | * int $	3 * ∉ Follow(T)	Shift
int * | int $	11	Shift
int * int | $		

S’ → E

E → T + E

E → T

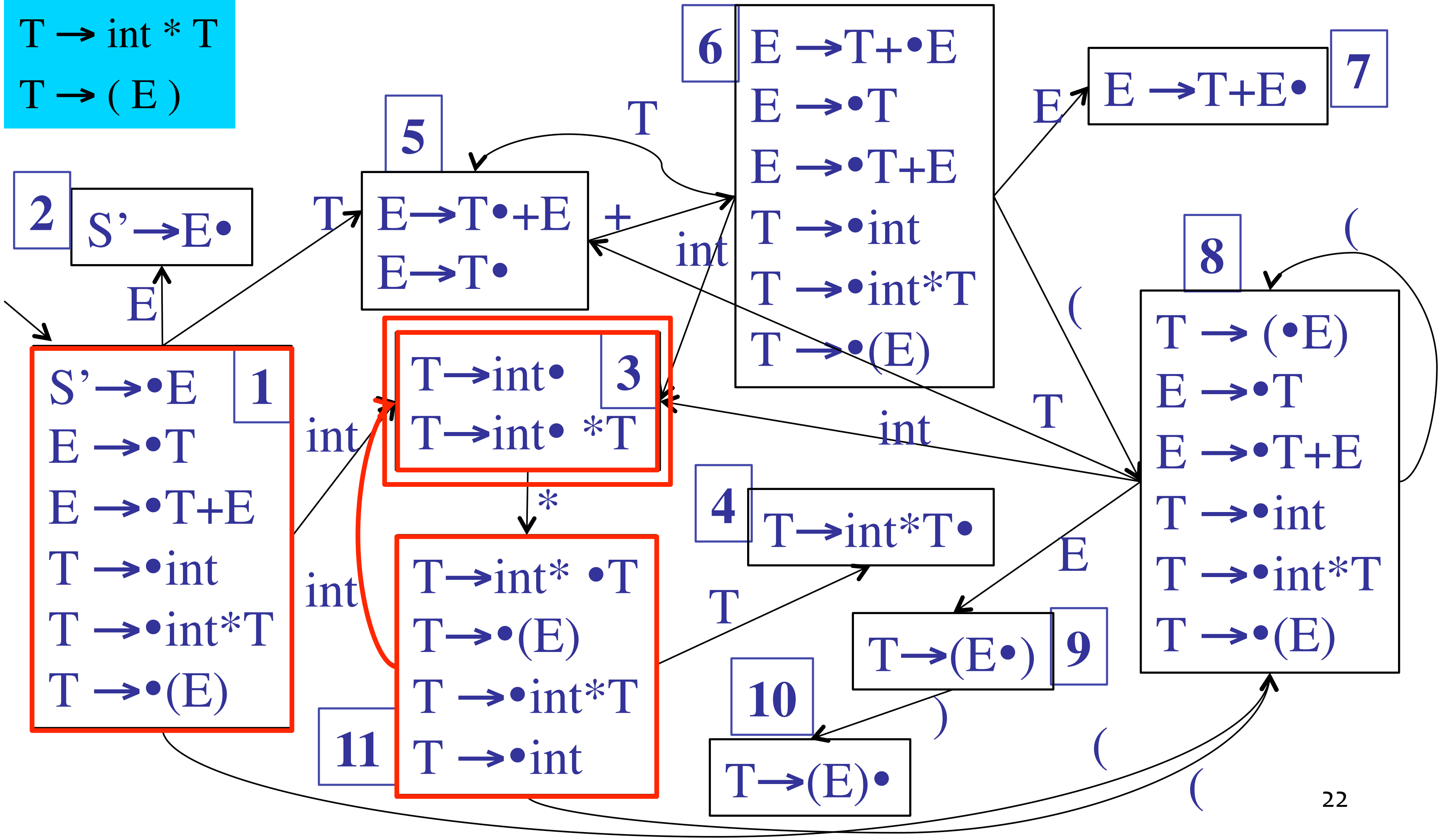
T → int

T → int * T

T → (E)

Follow(T)={$,),+}

int * int | $



1
S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

2
S’→E•

3
T→int•
T→int• *T

4
T→int*T•

5
E→T•+E
E→T•

6
E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

7
E →T+E•

8
T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

9
T→(E•)

10
T→(E)•

11
T→int* •T
T→•(E)
T →•int*T
T →•int

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $	1	Shift
int | * int $	3 $* \notin \text{Follow}(T)$	Shift
int * | int $	11	Shift
int * int | $	3 $\$ \in \text{Follow}(T)$	Reduce T → int
int * T | $		

S’ → E

E → T + E

E → T

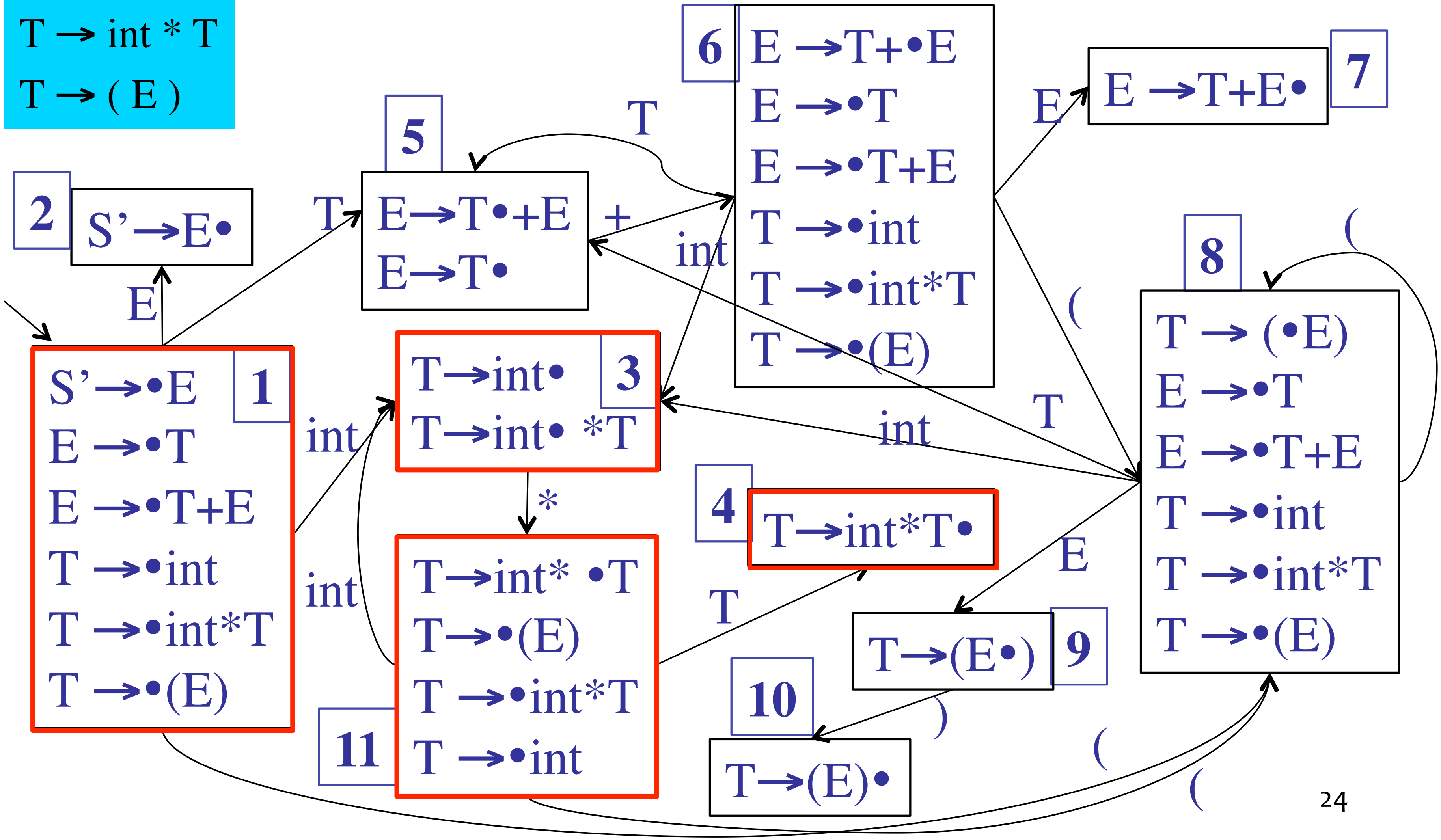
T → int

T → int * T

T → (E)

Follow(T)={$,),+}

int * T | $



1
S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

2
S’→E•

3
T→int•
T→int• *T

4
T→int*T•

5
E→T•+E
E→T•

6
E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

7
E →T+E•

8
T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

9
T→(E•)

10
T→(E)•

11
T→int* •T
T→•(E)
T →•int*T
T →•int

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $	1	Shift
int | * int $	3 * ∉ Follow(T)	Shift
int * | int $	11	Shift
int * int | $	3 $ ∈ Follow(T)	Reduce T → int
int * T | $	4 $ ∈ Follow(T)	Reduce T → int * T
T | $		

S’ → E

E → T + E

E → T

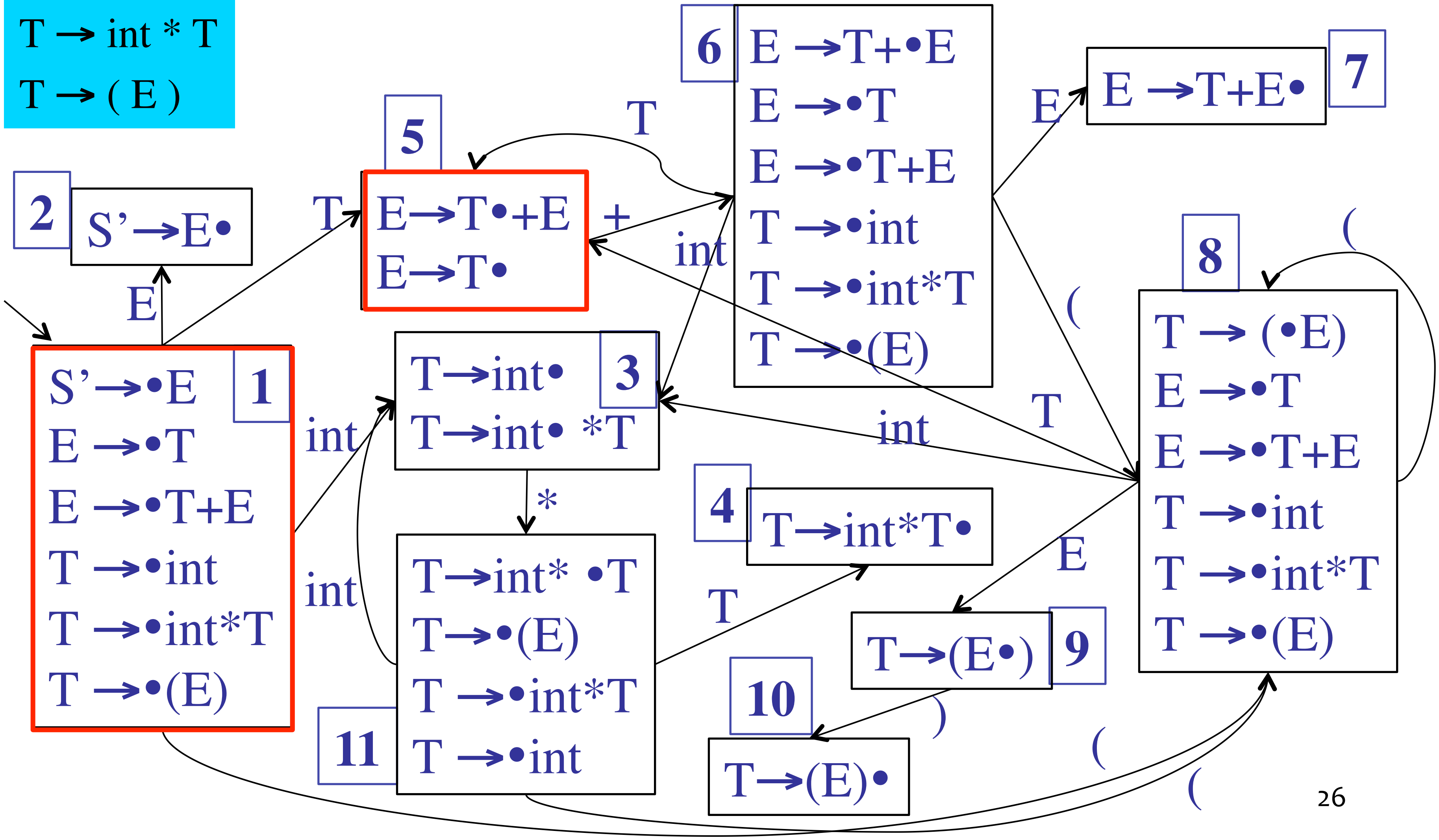
T → int

T → int * T

T → (E)

Follow(E)={$,)}

T | $



1
S’→•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

2
S’→E•

3
T→int•
T→int• *T

4
T→int*T•

5
E→T•+E
E→T•

6
E →T+•E
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

7
E →T+E•

8
T → (•E)
E →•T
E →•T+E
T →•int
T →•int*T
T →•(E)

9
T→(E•)

10
T→(E)•

11
T→int* •T
T→•(E)
T →•int*T
T →•int

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $	1	Shift
int | * int $	3 * ∉ Follow(T)	Shift
int * | int $	11	Shift
int * int | $	3 $ ∈ Follow(T)	Reduce T → int
int * T | $	4 $ ∈ Follow(T)	Reduce T → int * T
T | $	5 $ ∈ Follow(E)	Reduce E→T
E | $		

Trace ‘int*int’

configuration (Stack|input)	DFA halt state	Action
| int * int $	1	Shift
int | * int $	3 * ∉ Follow(T)	Shift
int * | int $	11	Shift
int * int | $	3 $ ∈ Follow(T)	Reduce T → int
int * T | $	4 $ ∈ Follow(T)	Reduce T → int * T
T | $	5 $ ∈ Follow(T)	Reduce E→T
E | $		Accept

S' → E

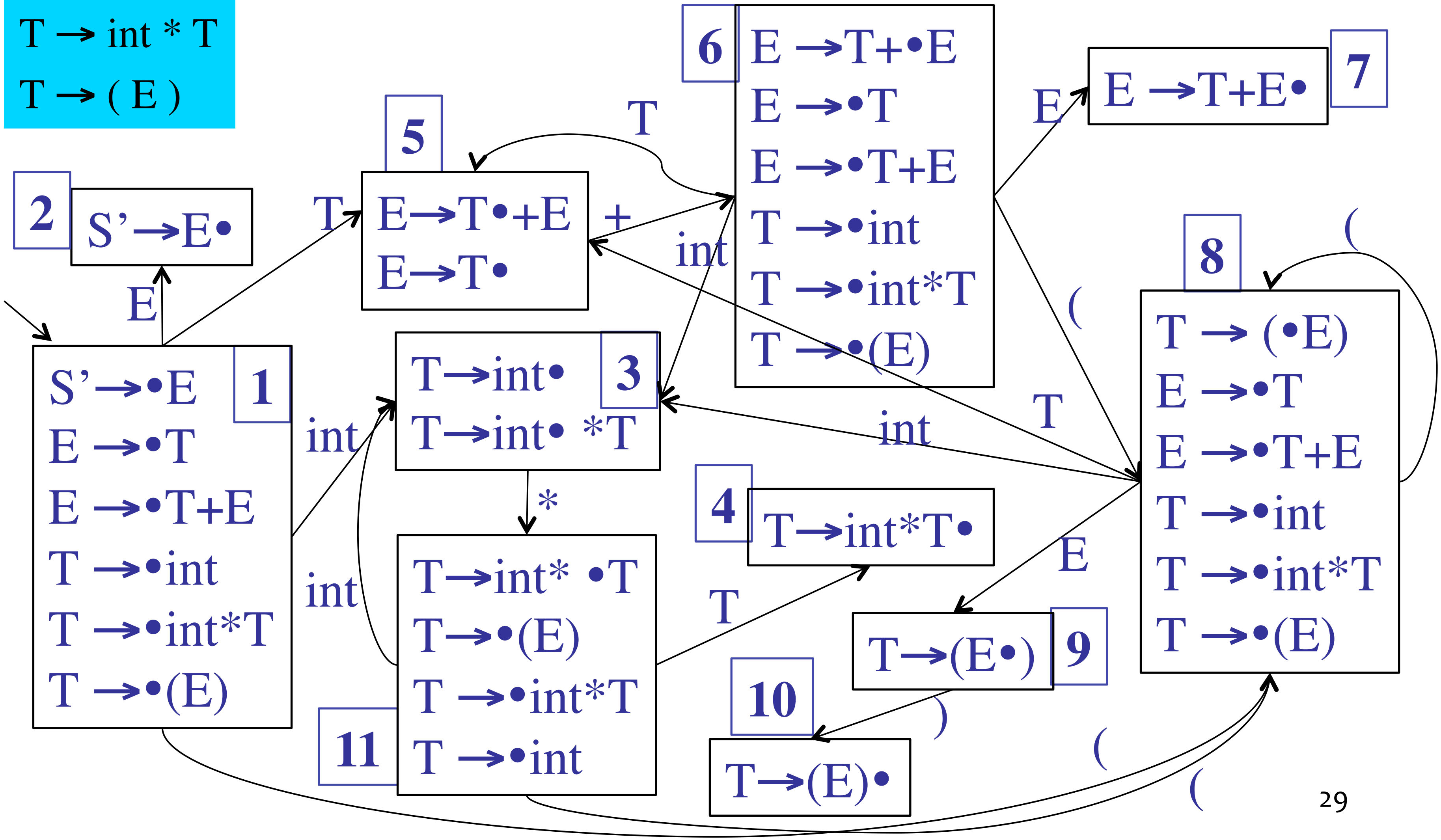
E → T + E

E → T

T → int

T → int * T

T → (E)



1

S'→•E

E →•T

E →•T+E

T →•int

T →•int*T

T →•(E)

2

S'→E•

3

T→int•

T→int• *T

4

T→int*T•

5

E→T•+E

E→T•

6

E →T+•E

E →•T

E →•T+E

T →•int

T →•int*T

T →•(E)

7

E →T+E•

8

T → (•E)

E →•T

E →•T+E

T →•int

T →•int*T

T →•(E)

9

T→(E•)

10

T→(E)•

11

T→int* •T

T→•(E)

T →•int*T

T →•int

Constructing SLR states

- Begin with item S’→•S, calculate related items (closure)
- Determine following states: what states can be reached on a single input token or non-terminal (GOTO)
- Construct closure of each resulting states

SLR(1) Construction

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

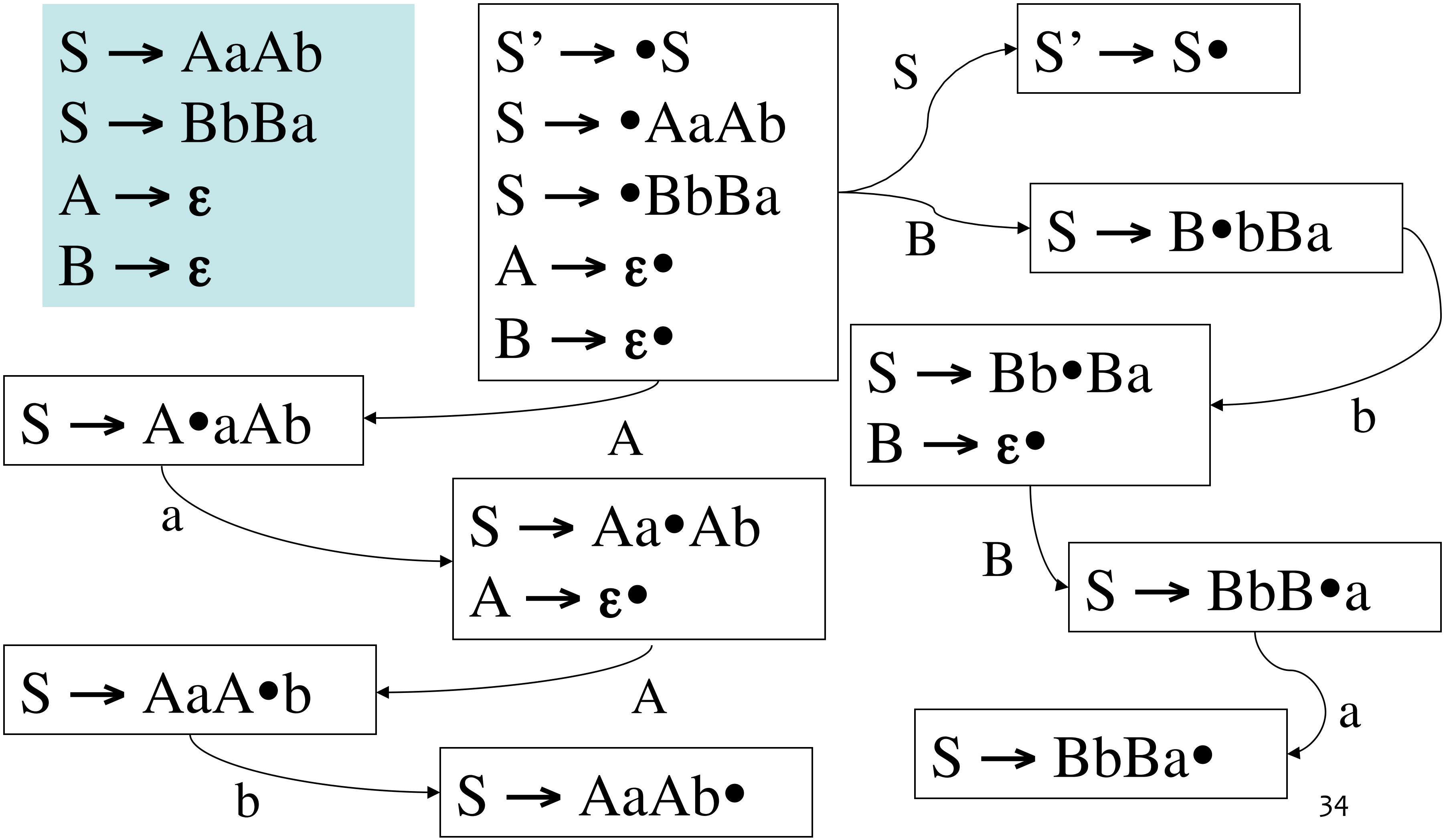
SLR(1) Conditions

- 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 ⊂ SLR(1) Grammars

Is this grammar SLR(1)?

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



$S' \rightarrow \bullet S$
$S \rightarrow \bullet AaAb$
$S \rightarrow \bullet BbBa$
$A \rightarrow \varepsilon\bullet$
$B \rightarrow \varepsilon\bullet$

$S' \rightarrow S\bullet$

$S \rightarrow B\bullet bBa$

$S \rightarrow Bb\bullet Ba$
$B \rightarrow \varepsilon\bullet$

$S \rightarrow BbB\bullet a$

$S \rightarrow BbBa\bullet$

$S \rightarrow A\bullet aAb$

$S \rightarrow Aa\bullet Ab$
$A \rightarrow \varepsilon\bullet$

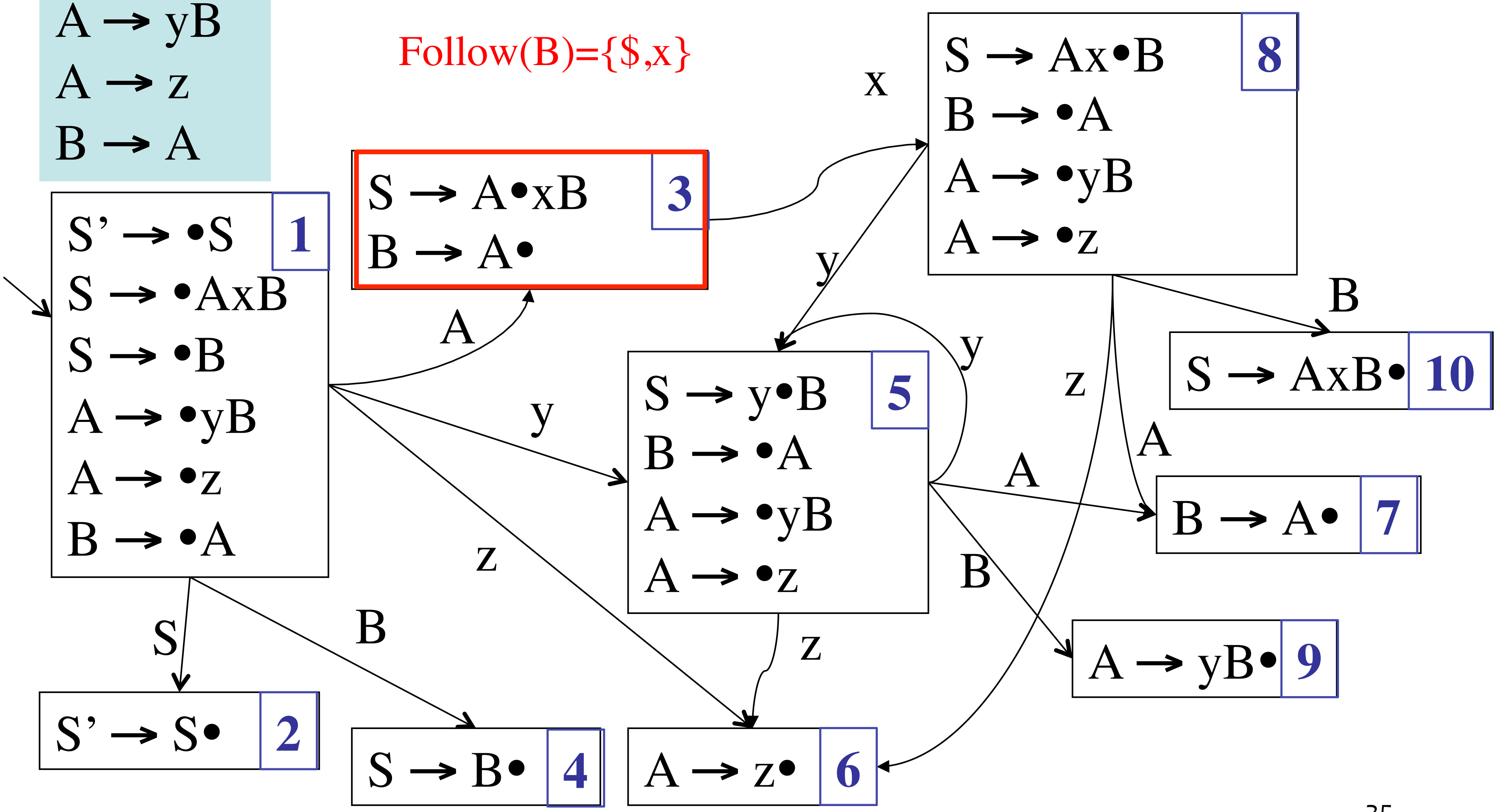
$S \rightarrow AaA\bullet b$

$S \rightarrow AaAb\bullet$

S’→ S
S → AxB
S → B
A → yB
A → z
B → A

Is this grammar SLR(1)?

Follow(B)={$,x}



1
S’ → •S
S → •AxB
S → •B
A → •yB
A → •z
B → •A

2
S’ → S•

3
S → A•xB
B → A•

4
S → B•

5
S → y•B
B → •A
A → •yB
A → •z

6
A → z•

7
B → A•

8
S → Ax•B
B → •A
A → •yB
A → •z

9
A → yB•

10
S → AxB•

1) S → AxB
2) S → B
3) A → yB
4) A → z
5) B → A

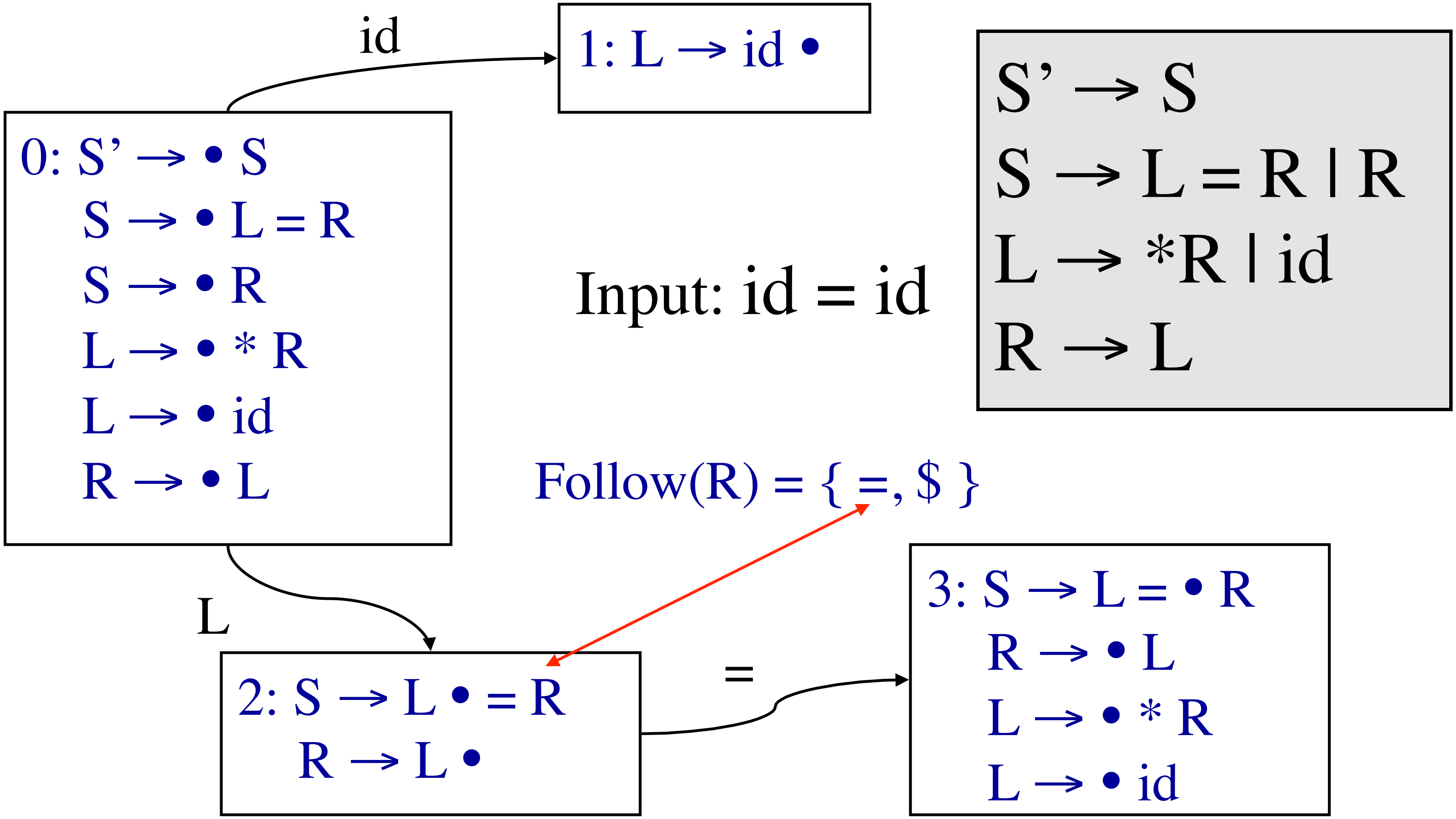
SLR Parsing Table

Grammar is not SLR

Reduce is a bad choice

	x	y	z	$	S	A	B
1		S5	S6		2	3	4
2				ACC!			
3	S8,R5			R5			
4				R2			
5		S5	S6			7	9
6	R4			R4			
7	R5			R5			
8		S5	S6			7	10
9	R3			R3			
10				R1			

SLR limitation: lack of context



0: S’ → • S
S → • L = R
S → • R
L → • * R
L → • id
R → • L

1: L → id •

2: S → L • = R
R → L •

3: S → L = • R
R → • L
L → • * R
L → • id

S’ → S
S → L = R | R
L → *R | id
R → L

Input: id = id

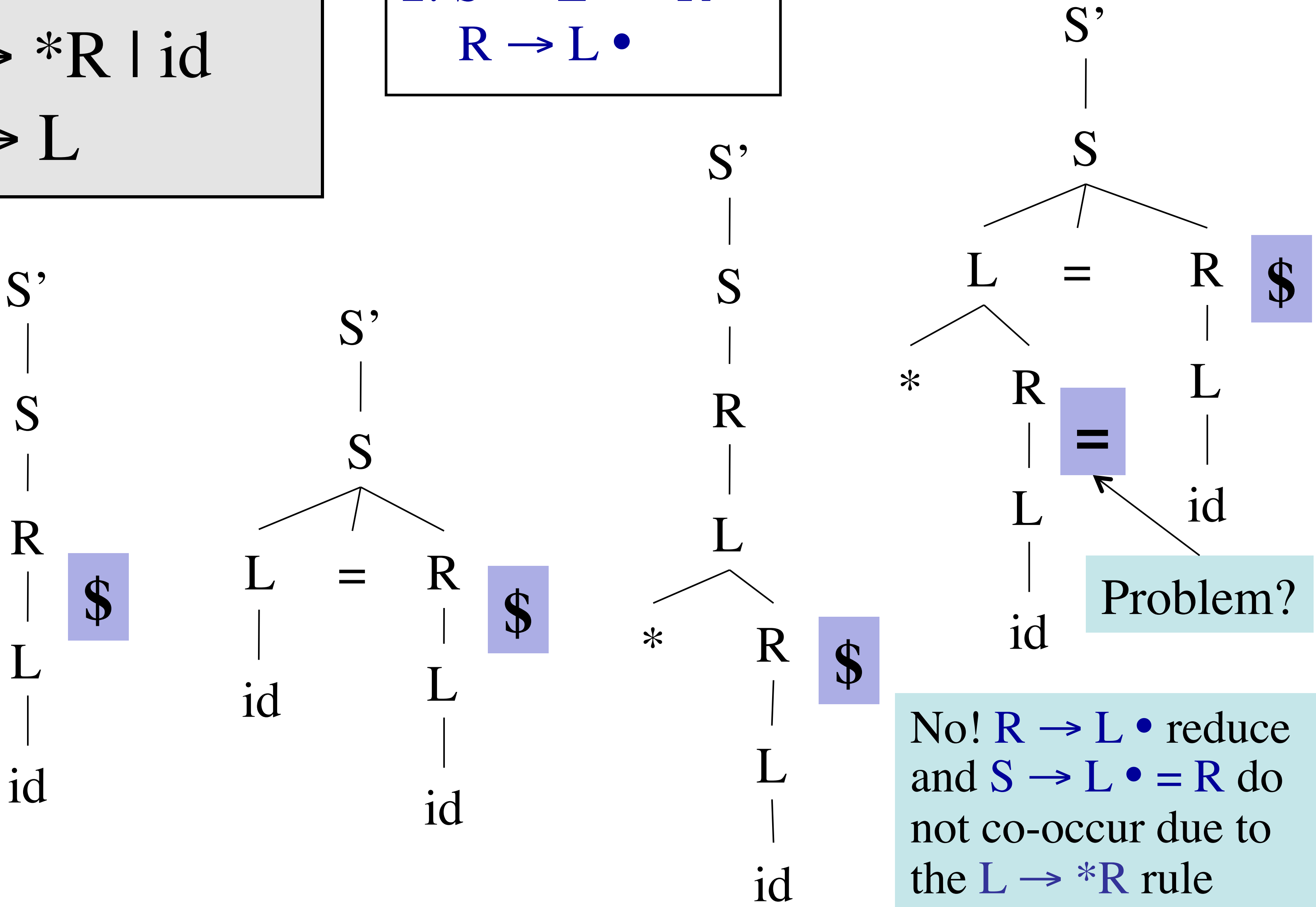
Follow(R) = { =, $ }

S’ → S
S → L = R | R
L → *R | id
R → L

Follow(R) = { =, $ }

2: S → L • = R
R → L •

Find all lookaheads for reduce R → L •



Problem?

No! R → L • reduce and S → L • = R do not co-occur due to the L → *R rule

Solution: Canonical LR(1)

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

LR(1) Parsing

- Limit introduced by SLR parsing in using Follow set to decide reductions
- Idea: augment LR items with 1 character lookahead [B → A•, $] making an LR(1) item
  - Reduce to B only if lookahead token is $
- More accurate than just Follow set
- Similar to SLR parsing just use LR(1) items rather than LR(0) items