Compilation

0368-3133

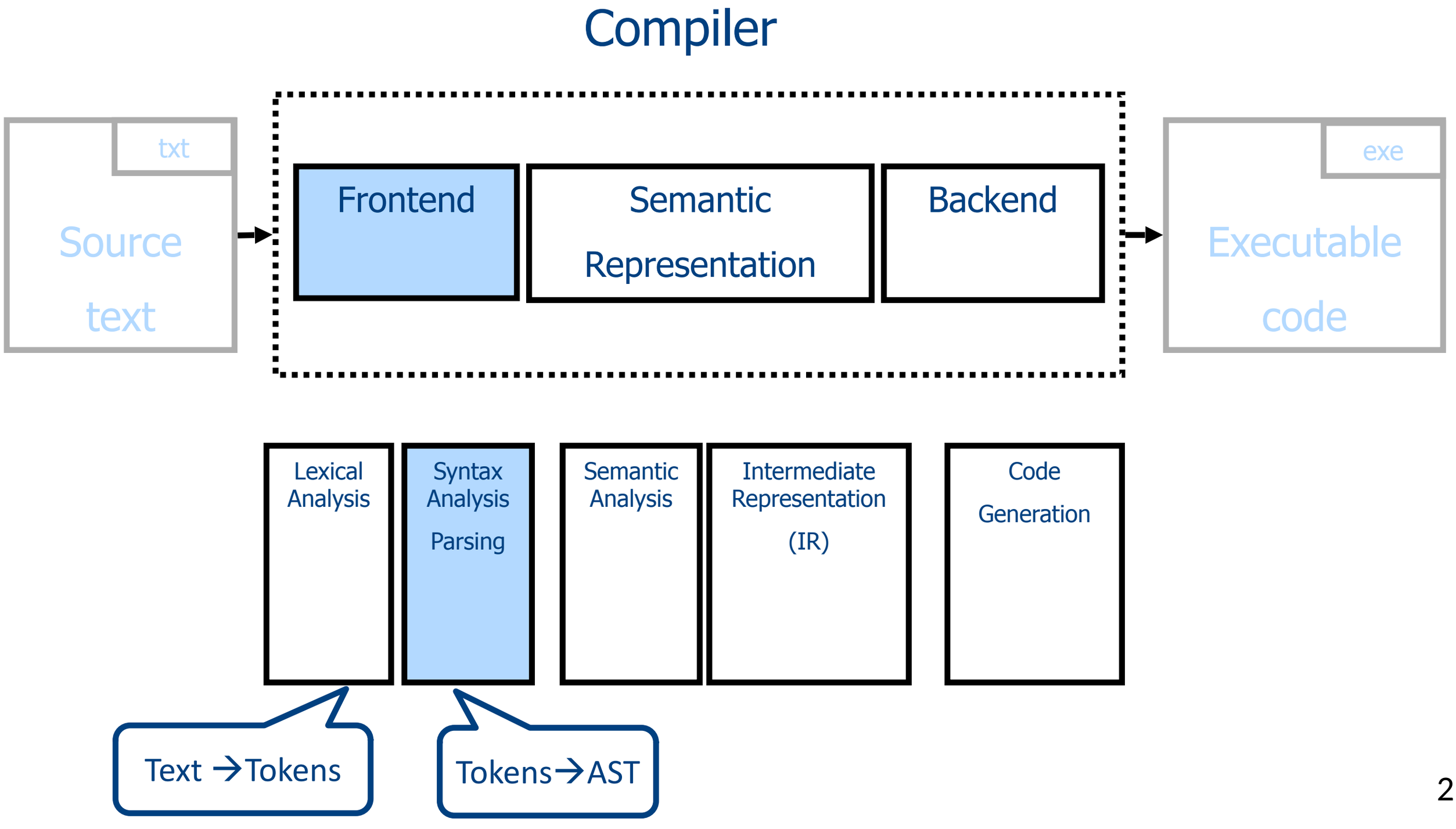
Lecture 4:

Syntax Analysis:

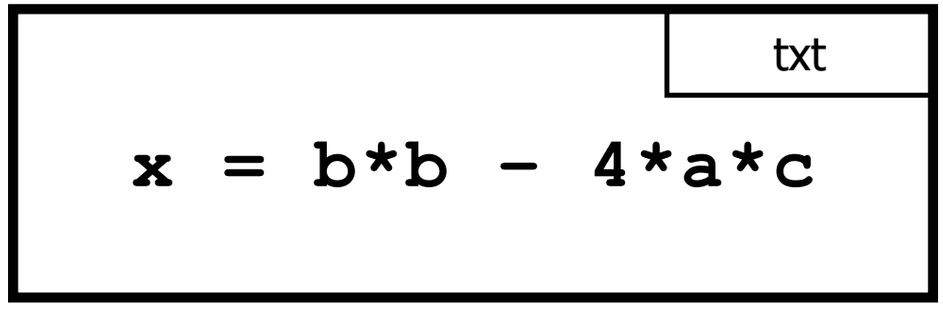
Bottom-Up Parsing (Part II)

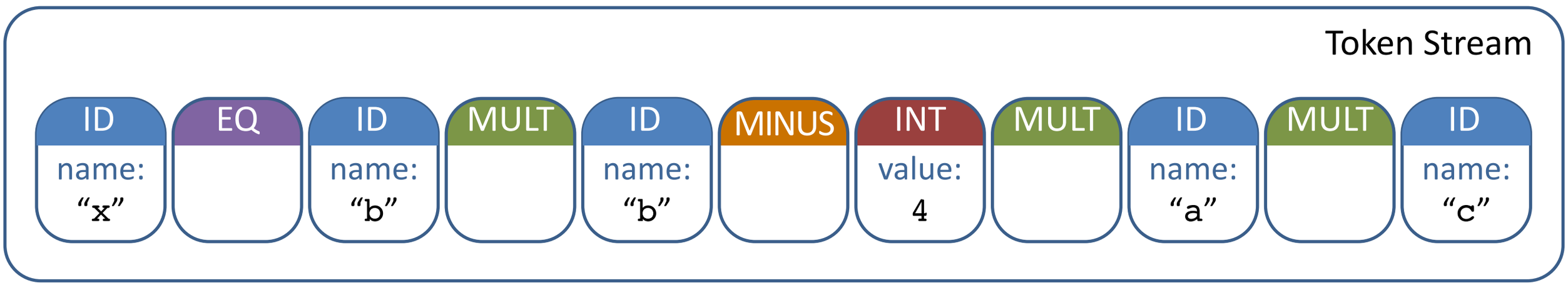
Slides' credit: Eran Yahav, Shachar Itzhaky, Roman Manevich

Conceptual Structure of a Compiler



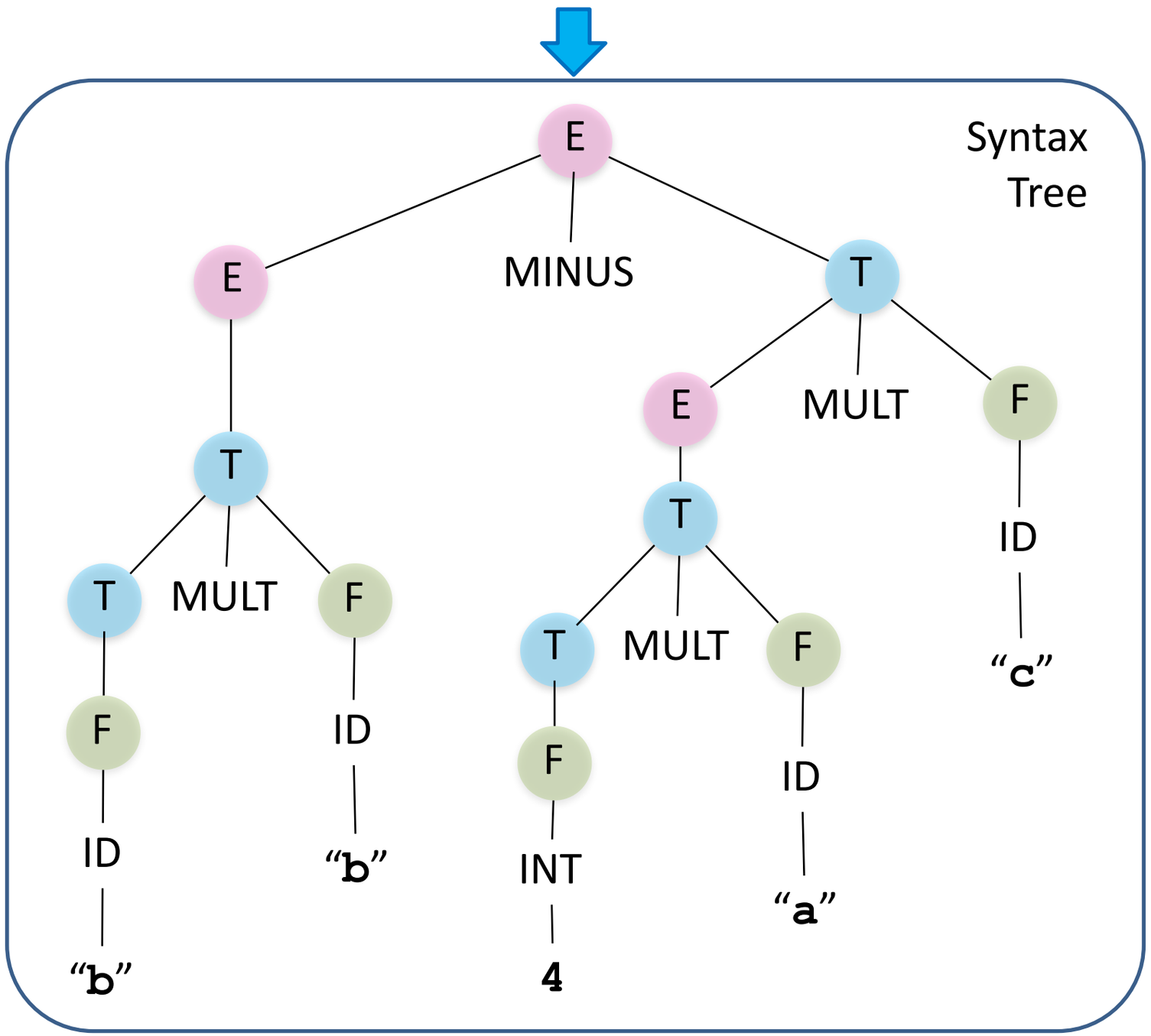
Lexing: from Characters to Tokens





Parsing: from Tokens to Syntax Tree

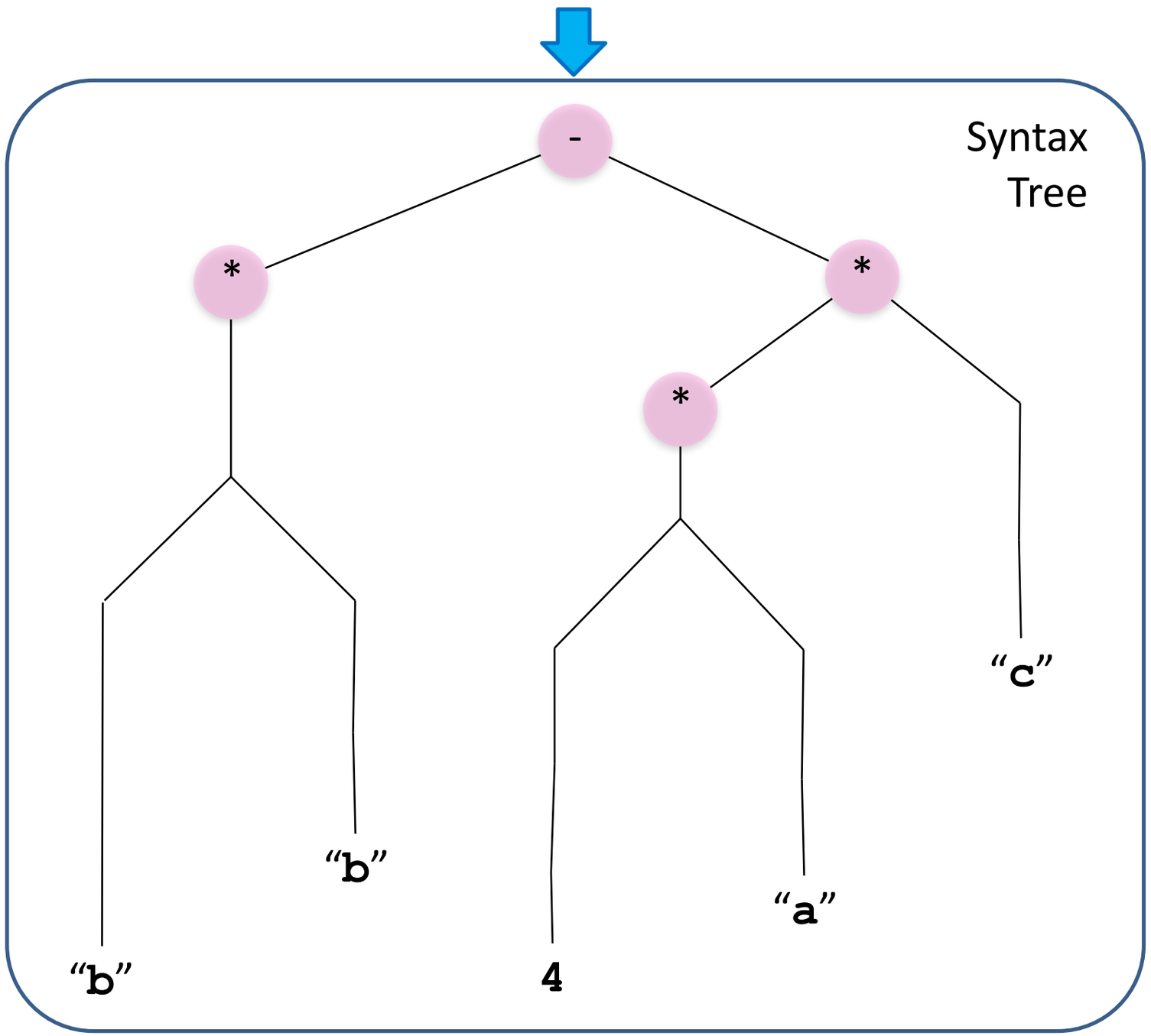
⟨ID,"**b**"⟩ ⟨MULT⟩ ⟨ID,"**b**"⟩ ⟨MINUS⟩ ⟨INT,**4**⟩ ⟨MULT⟩ ⟨ID,"**a**"⟩ ⟨MULT⟩ ⟨ID,"**c**"⟩



Lexical Analysis	Syntax Analysis	Sem. Analysis	Inter. Rep.	Code Gen.

Parsing: from Tokens to AST

⟨ID,"**b**"⟩ ⟨MULT⟩ ⟨ID,"**b**"⟩ ⟨MINUS⟩ ⟨INT,**4**⟩ ⟨MULT⟩ ⟨ID,"**a**"⟩ ⟨MULT⟩ ⟨ID,"**c**"⟩



Lexical Analysis	Syntax Analysis	Sem. Analysis	Inter. Rep.	Code Gen.

Bottom-Up Parsing

- Bottom-Up parsers
  - Scan input from left to right
  - Construct the parse tree in a bottom-up manner
    - Add a node when
      - All its leaves (tokens) have been read
      - All its children have been added
  - Reduce* the input to the start symbol
    - Reduction in reverse order of the rightmost derivation
  - Using a lookahead of *k* tokens

- A grammar that can be parsed like that is called an *LR(k)* grammar

* Reduction (step): βαμ ➡ βAμ if A ⟶ α be a production rule

Example

end-of-input marker

S → C$ | A$

A → BC | BB | BbC | CB

B → bb

C → cc

Rightmost derivation

w = bbbcc$

S ⇒

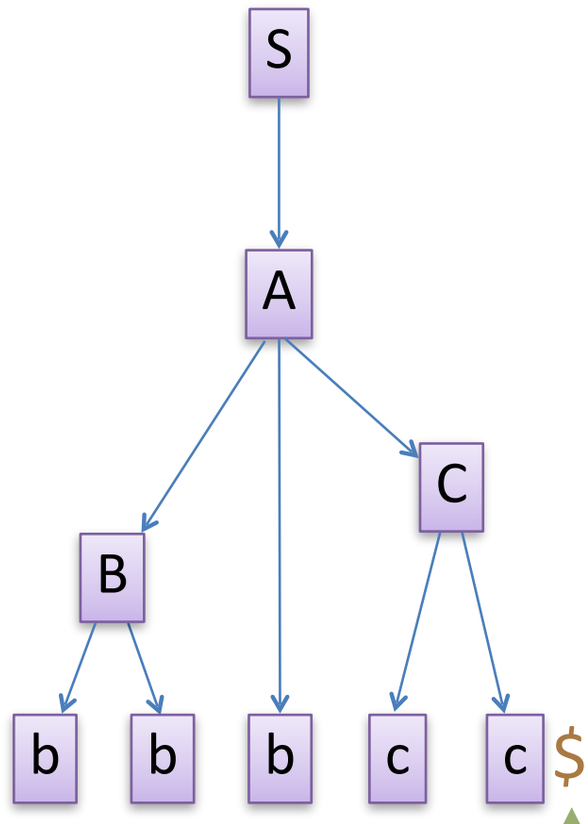
A$ ⇒

BbC$ ⇒

Bbcc$ ⇒

bbbcc$

Parse tree



not in the parse tree

Example

end-of-input marker

S → C$ | A$

A → BC | BB | BbC | CB

B → bb

C → cc

Rightmost derivation

w = bbbcc$

S ←

A$ ←

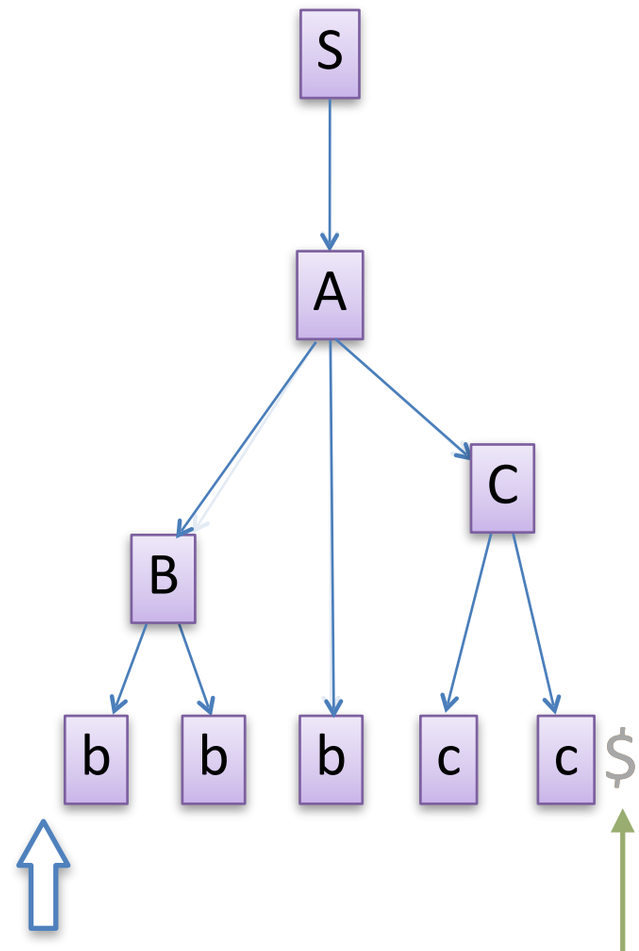
BbC$ ←

Bbcc$ ←

bbbcc$

desired reduction

Parse tree



How does the Parser Work?

Shift & Reduce (LR) Parsers

deterministic

The parser uses a **pushdown automaton** (DPDA)

$\in \Gamma^*$

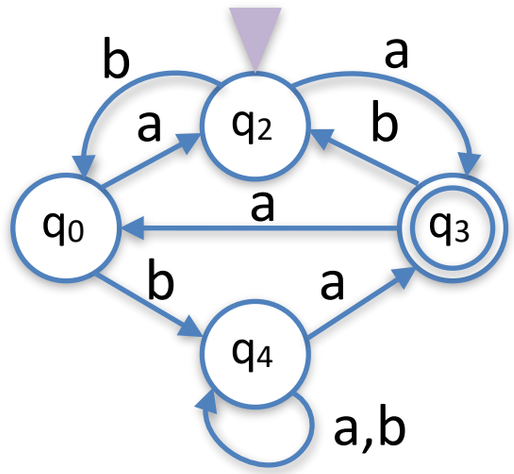
top → Y
X
X
Z

$(Q,\Gamma,\Sigma\cup\{\varepsilon\})\rightarrow(Q,\Gamma^*)$

Transition function

Stack

current state



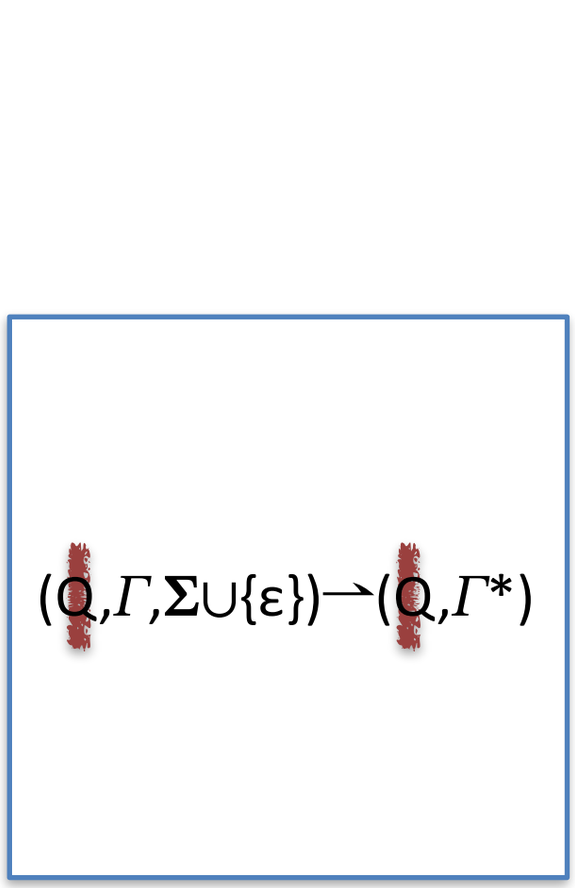
DFA

w = b a a b a a b a a b b a b a b a b b a $\in\Sigma^*$

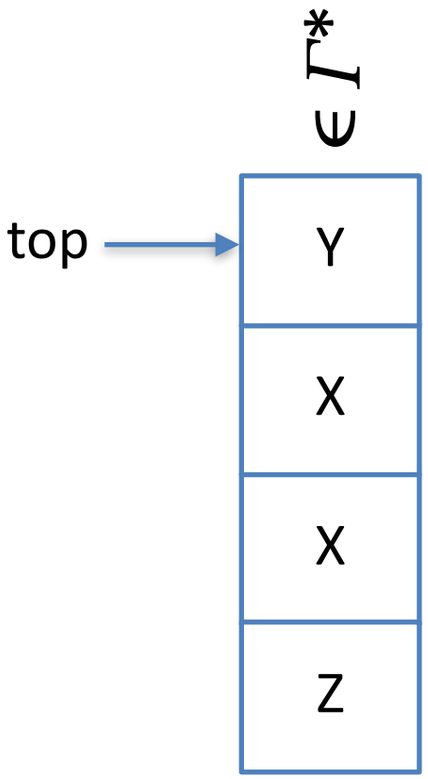
current position

Shift & Reduce (LR) Parsers

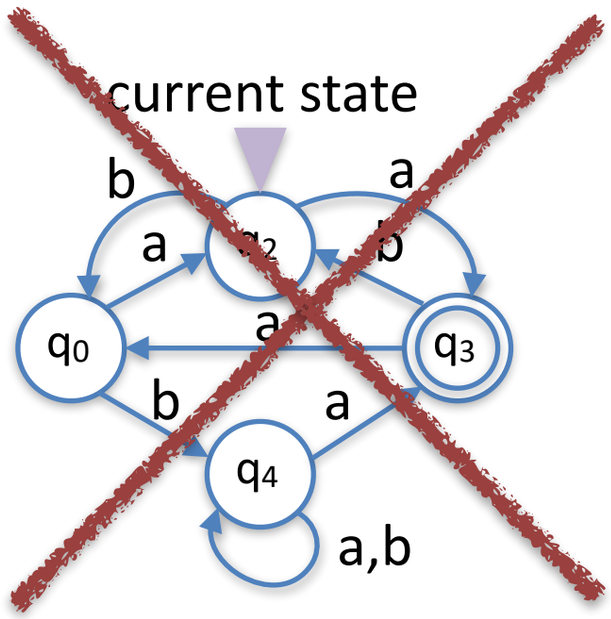
The parser uses a **deterministic** **pushdown automaton** (DPDA)



Transition function



Stack



DFA

Cheat

w = b a a b a a b a a b b a b a b a b b a ∈Σ*

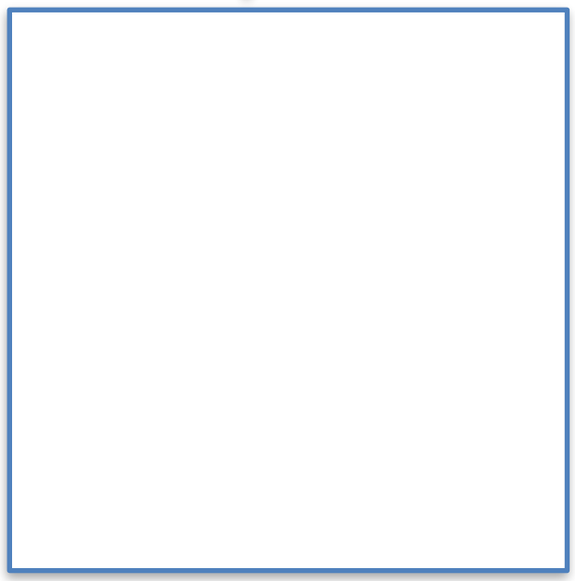
current position

Shift & Reduce (LR) Parsers

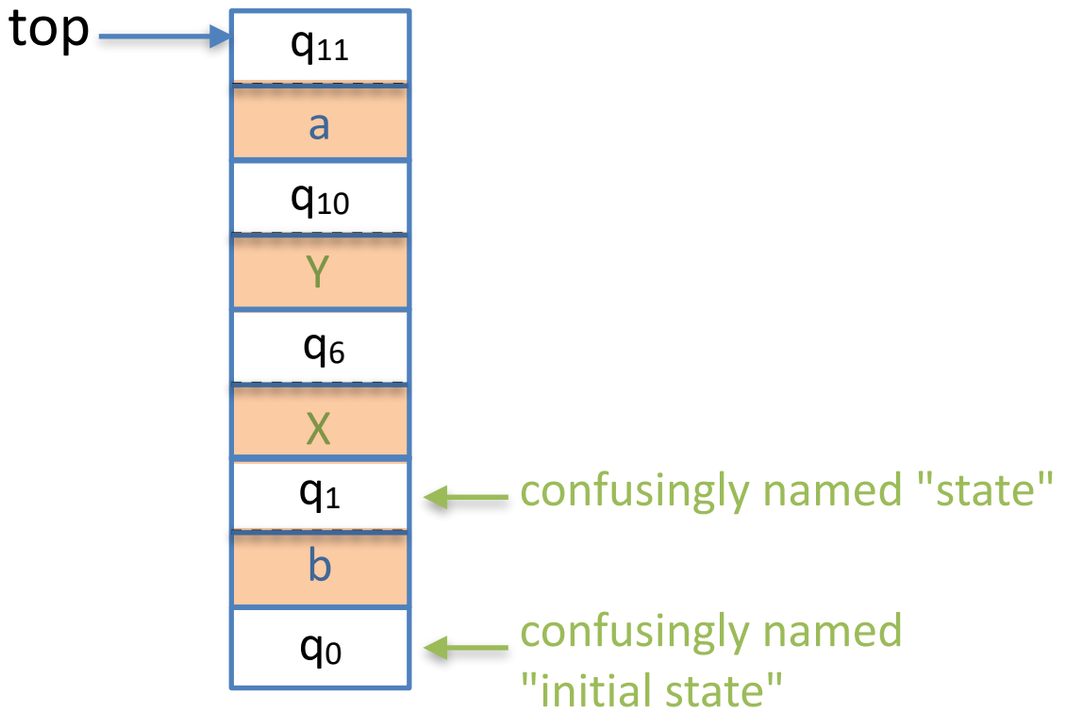
The parser uses a **pushdown automaton** (DPDA)

deterministic

Encodes a (confusingly named) "LR Automaton"



Transition ~~function~~ table



Stack

$S \to bXYba\$$

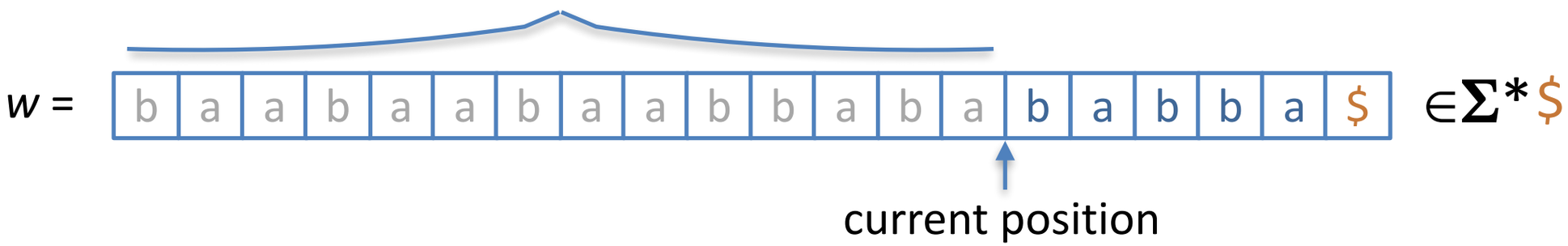
$X \to aabX \mid b$

$Y \to Yab \mid ab$

CFL

bXYa *

$S \Rightarrow^*_{rm} bXYa\,babba\$ \Rightarrow^*_{rm} w$



Shift & Reduce (LR) Parsers

The parser uses a **deterministic** **pushdown automaton** (DPDA)

The PDA maintains a stack of grammar symbols and states

In each step, we either shift or reduce

shift
- Moving a symbol from the input to the stack
- (computing & pushing a new state)

reduce
- Popping the symbols of the right-hand side of the reduced rule (and their paired states)
- Pushing the non-terminal in the left-hand side of the reduced rule
- (computing & pushing a new state)

Shift & Reduce (LR) Parsers

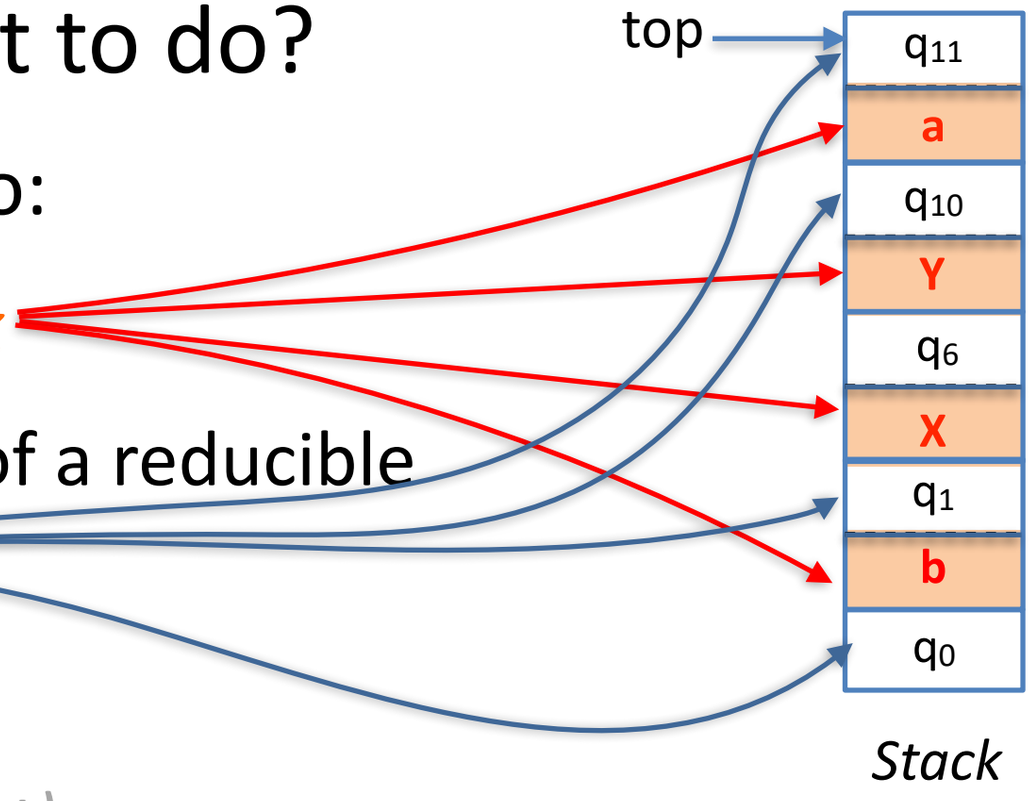
The parser uses a **pushdown automaton** (DPDA)

deterministic

How does the parser know what to do?

- Our progress record is divided in two:
  - The **reduction** so far is kept in a ***stack***
  - Our progress in discovering the RHS of a reducible rule is kept in a ***state***

    The ***state*** is kept in the **stack**!
  - *(We also keep the position in the input)*



- A ***transition table*** will tell the iterative algorithm "what to do" (shift? reduce?) based on the current (top) state and the next token(s)

Important Bottom-Up LR-Parsers

- LR(0) – simplest, explains basic ideas
- SLR(1) – simple, explains lookahead
- LR(1) – complicated, very powerful, expensive
- LALR(1) – complicated, powerful enough, used by automatic tools

LR(0) Items

For a production rule $N \rightarrow \alpha\ \beta$ in the grammar,

$A \rightarrow \bullet\alpha\beta$ $A \rightarrow \alpha\bullet\beta$ $(\beta \neq \varepsilon)$ Shift Item

$A \rightarrow \alpha\beta\bullet$ Reduce Item

LR(0) Items

initial rule

end-of-input marker

S' → S$

S → A | C

A → BC | BB | BbC | CB

B → bb

C → cc

A → •BbC A → B•bC A → Bb•C Shift Items

A → BbC• B → bb• Reduce Items

LR(0) States

S' → S$

S → A | C

A → BC | BB | BbC | CB

B → bb

C → cc

- LR(0) state = set of LR(0) items
- A shift state = a set of shift items
  - e.g., q_3 = {B → •bb , B → b•b}
  - There are some rules regulating which sets can be shift states.
    We'll discuss this in a minute
- A reduce state = a set containing a single reduce item
  - e.g., q_6 = {A → BC•}
  - We'll see later on why it must be a singleton state

LR(0) States - Closure Rules

- A shift state containing an item A $\rightarrow \alpha \bullet X\beta$, where X is a non-terminal, must include all items of the form X $\rightarrow \bullet \delta$ for every production rule in the grammar
- For a set of items *C*, *Clos(C)* denotes the closure of *C* according to the above rule

S' → S$
S → A | C
A → BC | BB | BbC | CB
B → bb
C → cc

e.g., {A → B•C, C → •cc} is a shift state, but {A → •BC} isn't

- The initial state is the closure of the initial rule
  - e.g., *Clos*({S' → •S$}) = {S' → •S$, S → •A, S → •C, A → •BC,
    A → •BB, A → •BbC, A → •CB,
    B → •bb, C → •cc}

LR(0) Transitions - nextState

- For every symbol (terminal <u>or</u> variable) K, and every (closed) set of items K
- nextSet(K,X) = *Clos*({N ⟶ αX•β | (N ⟶ α•Xβ) ∈ K})

S' ⟶ S$
S ⟶ A | C
A ⟶ BC | BB | BbC | CB
B ⟶ bb
C ⟶ cc

e.g., K= {A ⟶ •BC, C ⟶ •cc}

nextSet(K,c) = { C ⟶ c•c }

nextSet(K,B) = { A ⟶ B•C, C ⟶ •cc }

- For any possible state q, nextState(q,X) = nextSet(q,X)

LR(0) Actions

- The LR(0) state at the top of the stack determines the action of the PDA

- If the top state is a shift state, the PDA performs a shift
  - shift state = contains only shift items

- If the top state is a reduce state, the PDA performs a reduce
  - reduce state = contains only a (single) reduce item

LR(0) Actions

- shift moves (for token t)
  - Fire when the top state q contains a shift item(s) $X \rightarrow \alpha \bullet t \beta$ and the next input token is t
  - Determines the next top state q' based on the current top state q and the shifted token t: compute *nextState*(q,t)
  - Push t and q' to the stack

- reduce moves (for rule $N \rightarrow \alpha$)
  - Fire when the top state q contains a reduce item $N \rightarrow \alpha \bullet$
  - Pop α (and its paired states), resulting in a new top state q_r
  - Determines the next top state q' based on the discovered state q_r and N: compute *nextState*(q_r,N)
  - Push N and q' to the stack

LR(0) Actions

- shift moves (for token t)
  - Fire when the top state q contains a shift item(s) $X \rightarrow \alpha \bullet t\beta$ and the next input token is t — Need the stack; Looks at input
  - Determines the next top state q' based on the current top state q and the shifted token t: compute $nextState(q,t)$ — q' can be precomputed (for a given q,t)
  - Push t and q' to the stack — Need the stack

- reduce moves (for rule $N \rightarrow \alpha$)
  - Fire when the top state q contains a reduce item $N \rightarrow \alpha \bullet$ — Need the stack; Does **not** Look at input
  - Pop α (and its paired states), resulting in a new top state q_r — Need the stack
  - Determines the next top state q' based on the discovered state q_r and N: compute $nextState(q_r,N)$ — q' can be precomputed (for a given q_r,t)
  - Push N and q' to the stack — Need the stack

LR(0) Automaton

- We can pre-compute the set of possible states stored in the stack and the potential state transitions ahead of parsing time
- The pre-computation results in an **LR(0) automaton**

Simplified fixpint construction of the LR(0) automaton for G = (V,T,P,S)

States' := States := {InitialState}

Transitions = {}

do

States' := States ∪ {nextState(q,X) | q∈States, X∈ V ∪ T}

Transitions := Transitions ∪ {$q \xrightarrow{X} q'$ | q,q'∈ States, X∈ V ∪ T, q'=nextState(q)}

until States' = States

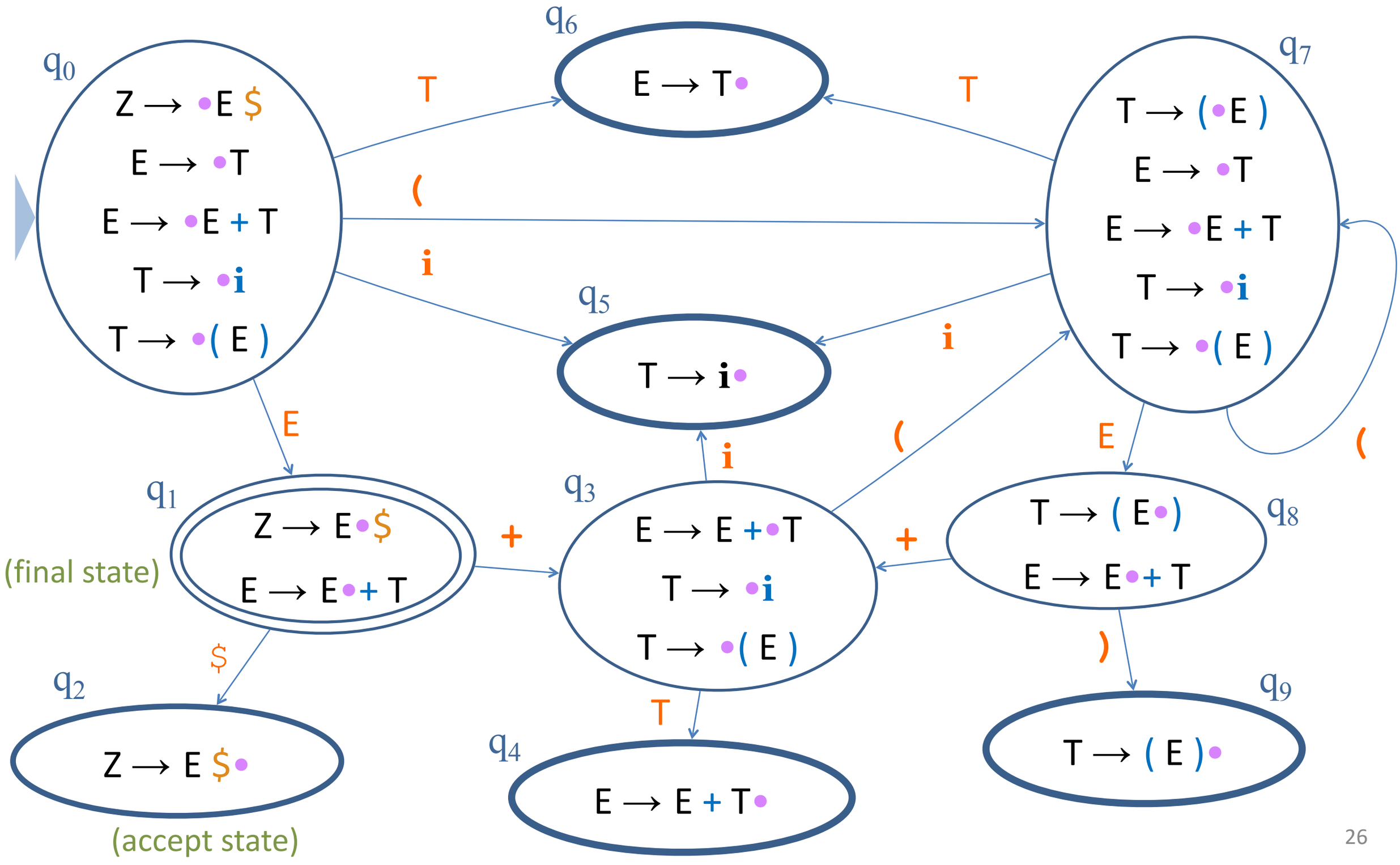
Example Grammar

$Z \rightarrow E\ \$$

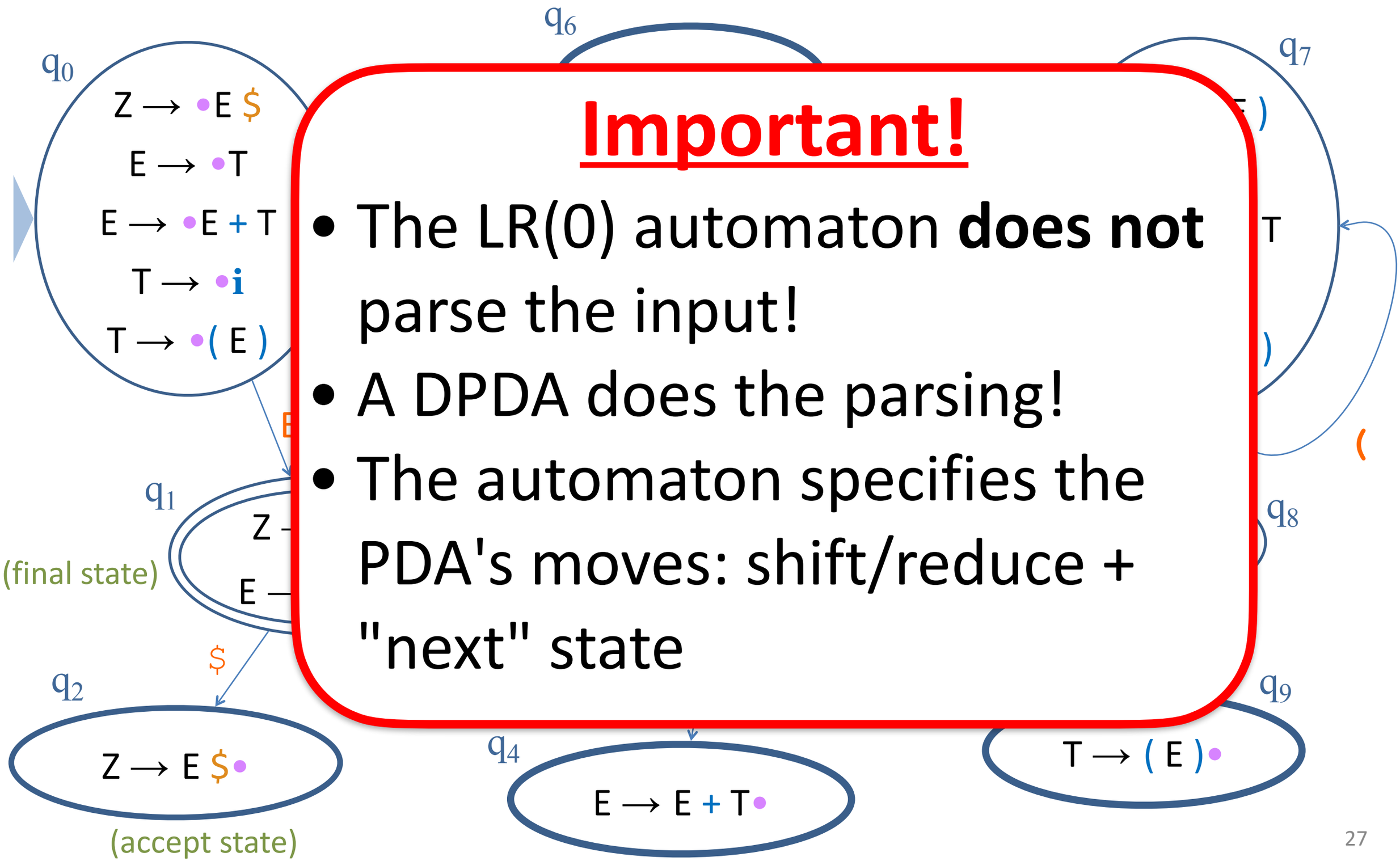
$E \rightarrow T \mid E + T$

$T \rightarrow \mathbf{i} \mid (\ E\ )$

LR(0) Automaton (Precomputed):



LR(0) Automaton (Precomputed):

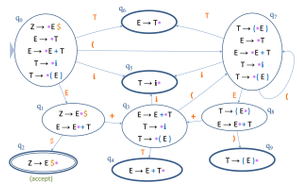


Important!

- The LR(0) automaton **does not** parse the input!
- A DPDA does the parsing!
- The automaton specifies the PDA's moves: shift/reduce + "next" state

How do parsers represent and use the LR(0) automaton?

LR(0) Parsing



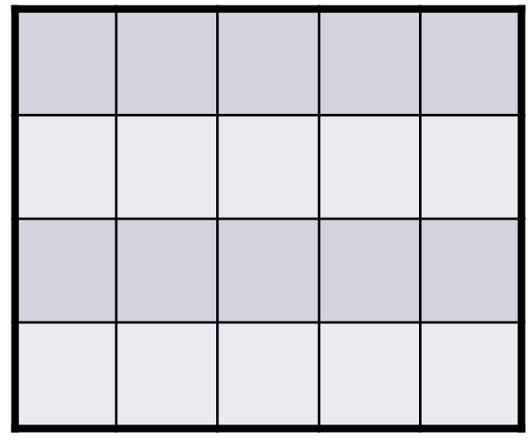
G

LR(0)-Automaton

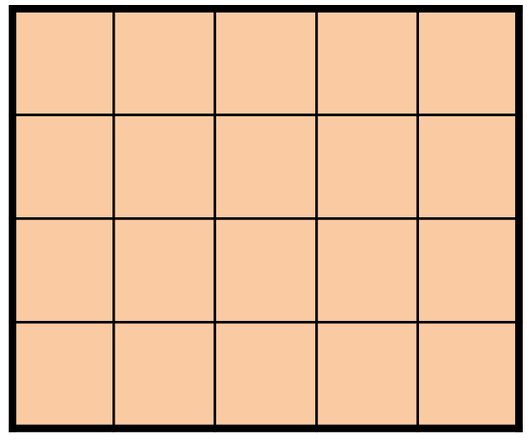
Input

Stack

PDA



ACTION Table



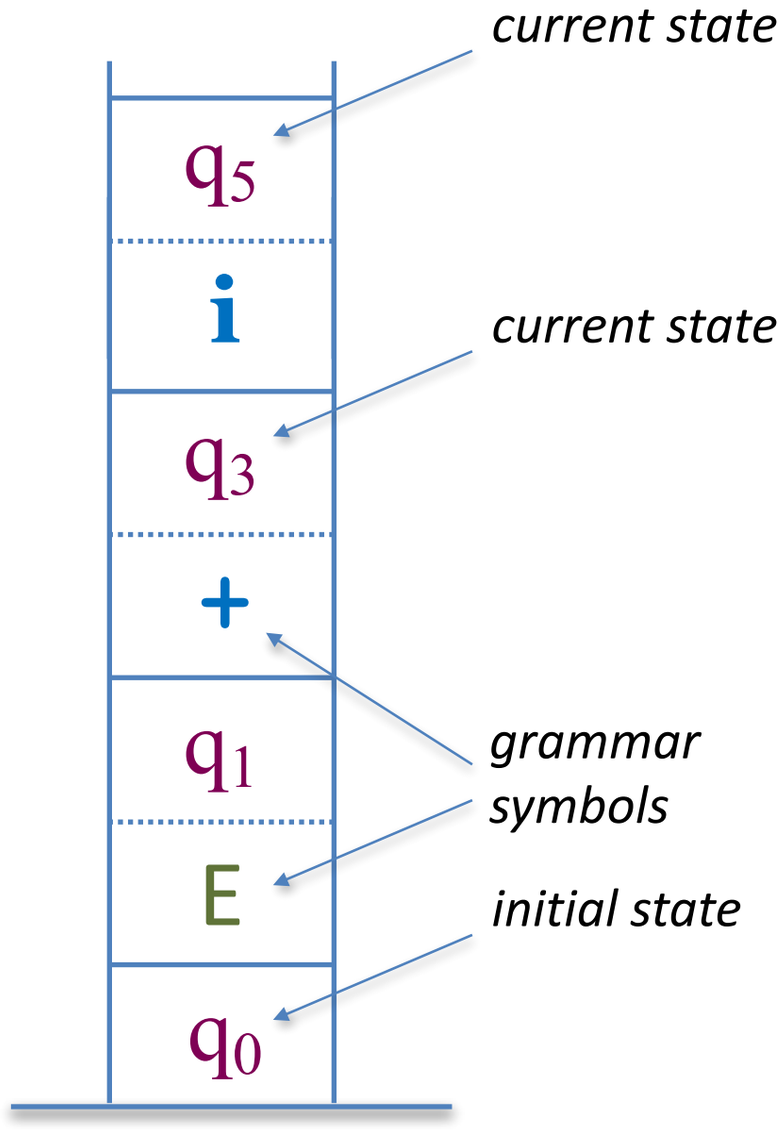
GOTO Table

Transition table

Output

The Stack

- The stack contains states and grammar symbols
- The initial stack contains q_0 only
- The rest of the stack contains pairs of
  - (state, token)
  - (state, nonterminal)



Aside: in a situation where we just want to check the syntax and return true/false, we could do without the symbols

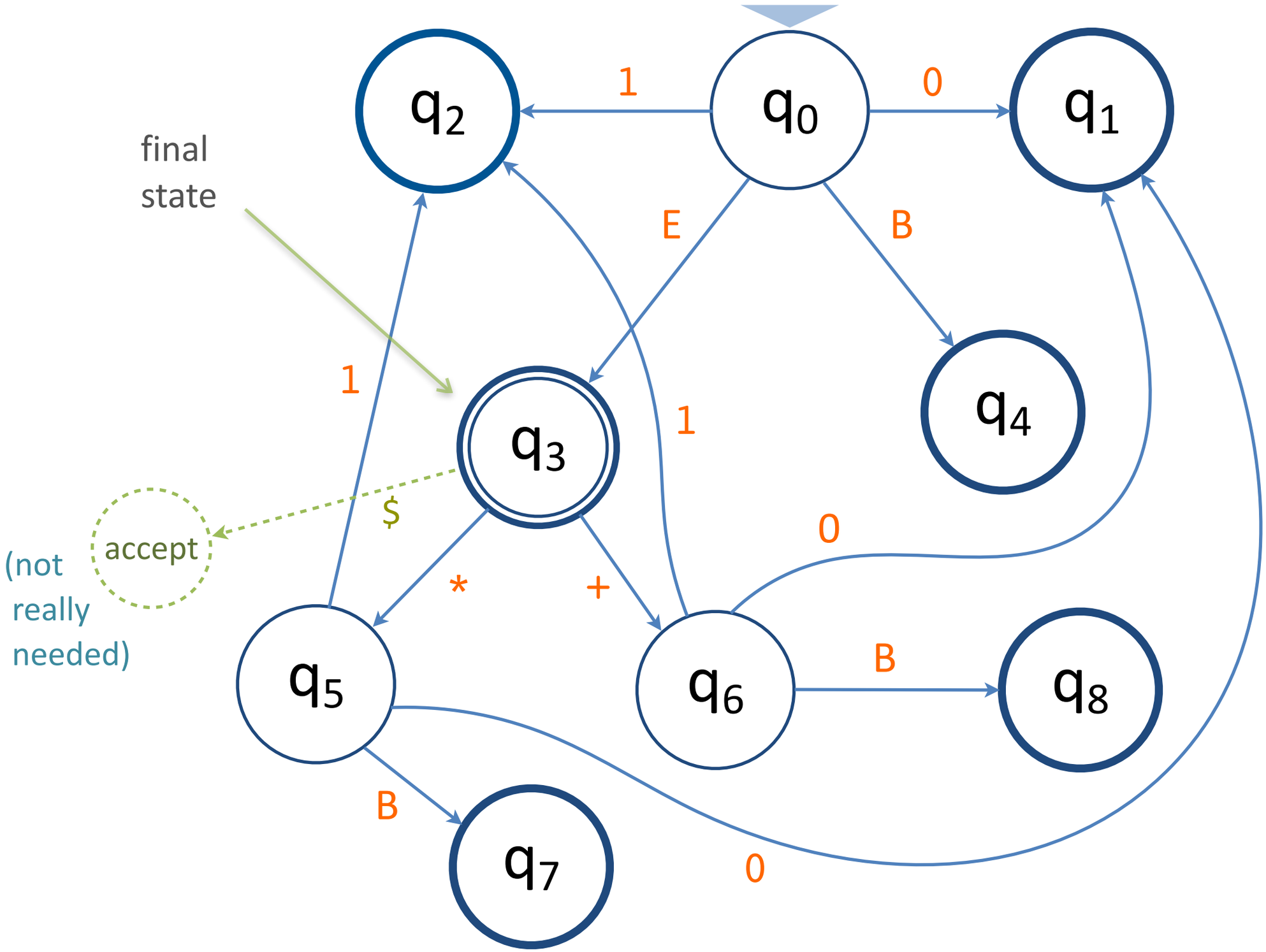
Aside II: sometimes we will look at the stack as an alternating sequence of states and symbols

Automaton ⇝ Tables

$S \rightarrow E\ \$$

$E \rightarrow E * B \mid E + B \mid B$

$B \rightarrow 0 \mid 1$



q_0

- $S \rightarrow \bullet E\ \$$
- $E \rightarrow \bullet E * B$
- $E \rightarrow \bullet E + B$
- $E \rightarrow \bullet B$
- $B \rightarrow \bullet 0$ $B \rightarrow \bullet 1$

q_4

- $E \rightarrow B \bullet$

q_3

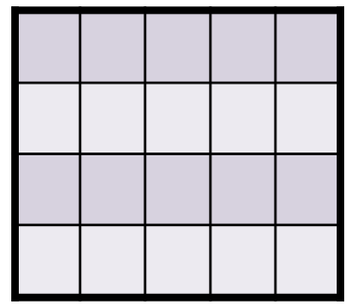
- $S \rightarrow E \bullet \$$
- $E \rightarrow E \bullet * B$
- $E \rightarrow E \bullet + B$

q_5

- $S \rightarrow E * \bullet B\ \$$
- $B \rightarrow \bullet 0$ $B \rightarrow \bullet 1$

• • •

The ACTION Table



- At each step we need to decide whether to shift the next token to the stack (and move to the appropriate state) or reduce a production rule from the grammar
- ACTION[q, t] table tells us what to do based on current state q and next token t:

shift i : shift and move to q_i

reduce j: reduce according to production rule (j)

also: accept and error conditions

(empty cells)

LR(0) Action Table

	ACTION				
	*	+	0	1	$
q_0			s1	s2	
q_1	r4	r4	r4	r4	r4
q_2	r5	r5	r5	r5	r5
q_3	s5	s6			acc
q_4	r3	r3	r3	r3	r3
q_5			s1	s2	
q_6			s1	s2	
q_7	r1	r1	r1	r1	r1
q_8	r2	r2	r2	r2	r2

shift "1" and go to state q_2

reduce by rule (4) B ⟶ 0

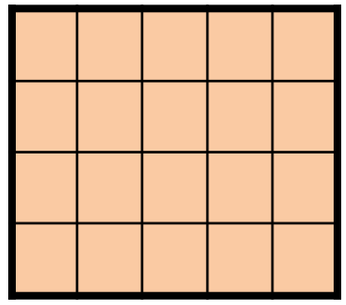
accept

(0) S ⟶ E $
(1) E ⟶ E * B
(2) E ⟶ E + B
(3) E ⟶ B
(4) B ⟶ 0
(5) B ⟶ 1

s i = shift to state q_i

r j = reduce using rule (j)

The GOTO Table



- Defines what to do on reduce actions
  - After reducing a right-hand side to the deriving non-terminal, we need to decide what the next state is
- This is determined by the previous state (which is on the stack) and the variable we got
  - Suppose we reduce according to $N \rightarrow \beta$;
  - We remove β from the stack, and look at the state q that is now at the top. GOTO[q, N] specifies the next state.

Note – this can be a little confusing:

- q is the state **after** popping β
- N is the left-hand side of the rule just used in reduce

LR(0) Goto Table

q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8

GOTO	
E	B
g3	g4
	g7
	g8

if after popping the stack due to reduction to B, the top most state is q_0
then push q_4 to the stack

(0) S → E $
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

LR(0) Transition Table

	ACTION					GOTO	
	*	+	0	1	$	E	B
q_0			s1	s2		g3	g4
q_1	r4	r4	r4	r4	r4		
q_2	r5	r5	r5	r5	r5		
q_3	s5	s6			acc		
q_4	r3	r3	r3	r3	r3		
q_5			s1	s2			g7
q_6			s1	s2			g8
q_7	r1	r1	r1	r1	r1		
q_8	r2	r2	r2	r2	r2		

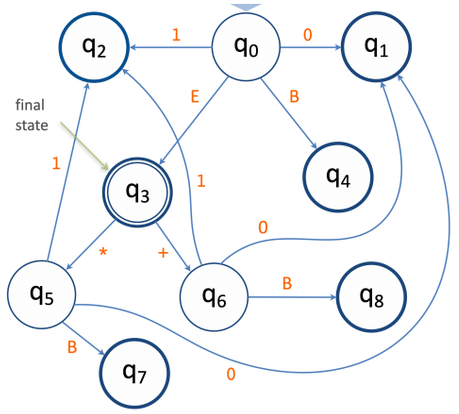
if after popping the stack due to reduction to B, the top most state is q_0
then push q_4 to the stack

(0) S ⟶ E $
(1) E ⟶ E * B
(2) E ⟶ E + B
(3) E ⟶ B
(4) B ⟶ 0
(5) B ⟶ 1

s i = shift to state q_i

r j = reduce using rule (j)

Building the Tables

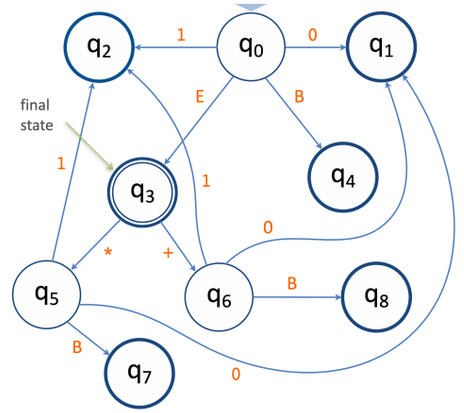


State	ACTION					GOTO	
	*	+	0	1	$	E	B
q_0			1	2		3	4
q_1							
q_2							
q_3	5	6					
q_4							
q_5			1	2			7
q_6			1	2			8
q_7							
q_8							
						E	B

- A row for each state.
- If there is a transition from q_i to q_j upon seeing X, then in row q_i and column x we write j.

(except X = $; see next slide)

Building the tables: accept



State	ACTION					GOTO	
	*	+	0	1	$	E	B
q_0			1	2		3	4
q_1							
q_2							
q_3	5	6			acc		
q_4							
q_5			1	2			7
q_6			1	2			8
q_7							
q_8							

- Add accept in column $ for each state that has the item $S \rightarrow E \bullet \$$

(so-called *final* states).

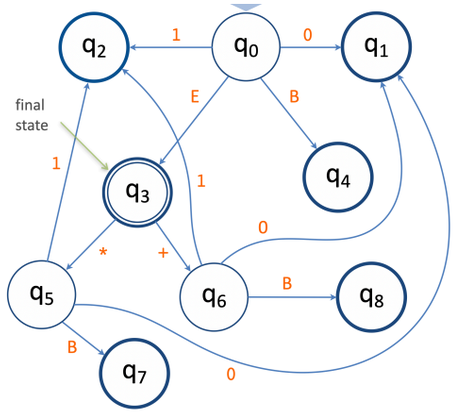
q_3

$S \rightarrow E \bullet \$$

$E \rightarrow E \bullet * B$

$E \rightarrow E \bullet + B$

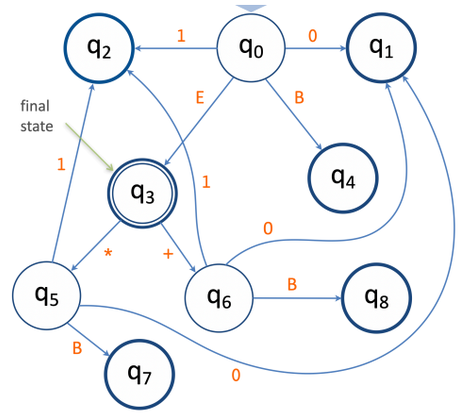
Building the Tables: GOTO



State	ACTION					GOTO	
	*	+	0	1	$	E	B
q_0			1	2		g3	g4
q_1							
q_2							
q_3	5	6			acc		
q_4							
q_5			1	2			g7
q_6			1	2			g8
q_7							
q_8							

- Any number n in the GOTO table becomes goto n.

Building the Tables: Shift

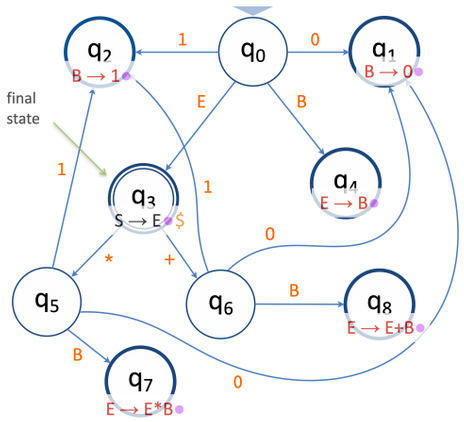


State	ACTION					GOTO	
	*	+	0	1	$	E	B
q_0			s1	s2		g3	g4
q_1							
q_2							
q_3	s5	s6			acc		
q_4							
q_5			s1	s2			g7
q_6			s1	s2			g8
q_7							
q_8							

- Any number n in the ACTION table becomes shift n.

(0) S ⟶ E $
(1) E ⟶ E * B
(2) E ⟶ E + B
(3) E ⟶ B
(4) B ⟶ 0
(5) B ⟶ 1

ing the Tables: Reduce



	ACTION					GOTO	
		+	0	1	$	E	B
q_0			s1	s2		g3	g4
q_1	r4	r4	r4	r4	r4		
q_2	r5	r5	r5	r5	r5		
q_3	s5	s6			acc		
q_4	r3	r3	r3	r3	r3		
q_5			s1	s2			g7
q_6			s1	s2			g8
q_7	r1	r1	r1	r1	r1		
q_8	r2	r2	r2	r2	r2		

For any state which includes an item

$A \longrightarrow \alpha\bullet$, such that

$A \longrightarrow \alpha$ is production rule (m):

Fill *all columns* of that state in the ACTION table with reduce m.

It means that — when a reduce is possible, we execute it *without* checking the next token.

k = 0

The Parsing Algorithm, Formally

- Initialize the stack to q_0
- Repeat until halting:
  - Let q = top of stack, t = next token;

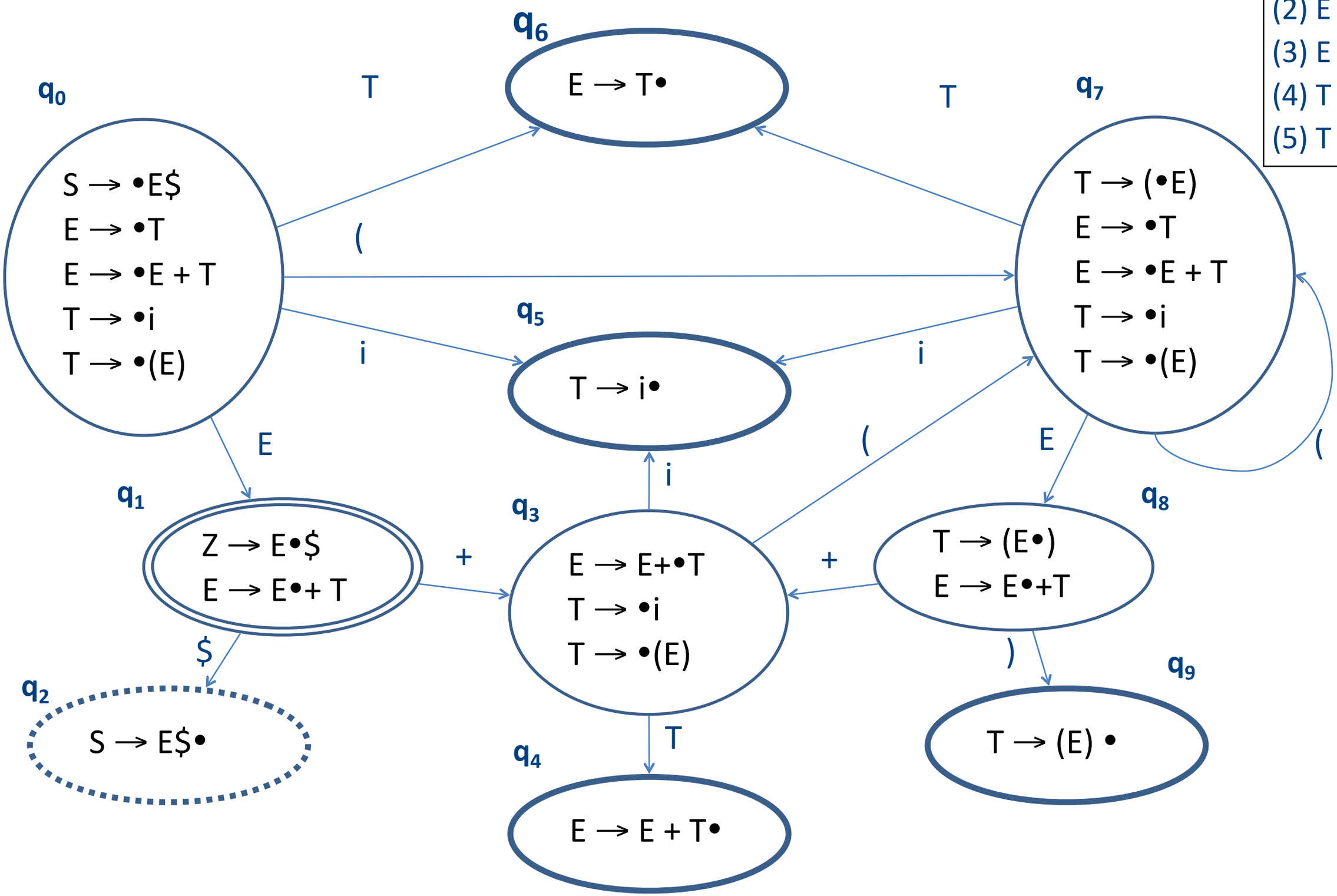
    Consider ACTION[q, t] —

    - "shift i":
      - Remove t from the input; push t and q_i on the stack.
    - "reduce j", where rule (j) is $N \longrightarrow \beta$:
      - Pop $|\beta|$ pairs from the stack;

        let q' = the state at the top of the stack *now*.
      - Push N and the state GOTO[q', N] on the stack.
    - "accept": halt successfully.
    - empty cell: halt with an error.

Example Run

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)



GOTO/ACTION Table

State	ACTION					GOTO	
	i	+	(	)	$	E	T
q_0	s5		s7			g1	g6
q_1		s3			acc		
q_3	s5		s7				g4
q_4	r3	r3	r3	r3	r3		
q_5	r4	r4	r4	r4	r4		
q_6	r2	r2	r2	r2	r2		
q_7	s5		s7			g8	g6
q_8		s3		s9			
q_9	r5	r5	r5	r5	r5		

(1) Z → E $

(2) E → T

(3) E → E + T

(4) T → i

(5) T → (E)

s*n* = shift to state *n*

r*m* = reduce using rule number (*m*)

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way

Stack	Input	Action
0	id + id $	s5

Initialize with state 0

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way

Stack	Input	Action
0	id + id $	s5

Initialize with state 0

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4

pop id 5

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way →

Stack	Input	Action
0	id + id $	s5
0 ~~id 5~~	+ id $	r4

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			~~acc~~		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

push T 6

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way →

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way →

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way →

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way →

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4
0 E 1 + 3 T 4	$	r3

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack grows this way

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4
0 E 1 + 3 T 4	$	r3
0 E 1	$	acc

S	*action*					*goto*	
	id	+	)	(	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

rn = reduce using rule number n
sm = shift to state m

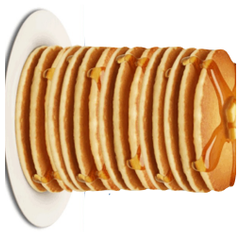
Exercise: Parsing: 0 * 0 + 1

	ACTION					GOTO	
	*	+	0	1	$	E	B
q_0			s1	s2		3	4
q_1	r4	r4	r4	r4	r4		
q_2	r5	r5	r5	r5	r5		
q_3	s5	s6			acc		
q_4	r3	r3	r3	r3	r3		
q_5			s1	s2			7
q_6			s1	s2			8
q_7	r1	r1	r1	r1	r1		
q_8	r2	r2	r2	r2	r2		

(0) S ⟶ E $
(1) E ⟶ E * B
(2) E ⟶ E + B
(3) E ⟶ B
(4) B ⟶ 0
(5) B ⟶ 1

- Initialize the stack to q_0
- Repeat until halting:
  - Let q = top of stack, t = next token;
    Consider ACTION[q, t] —
    - "shift i":
      - Remove t from the input; push t and q_i on the stack.
    - "reduce j", where rule (j) is N ⟶ β:
      - Pop |β| pairs from the stack; let q' = the state at the top of the stack *now*.
      - Push N and the state GOTO[q', N] on the stack.
- "accept": halt successfully.
- empty cell: halt with an error.

0 * 0 + 1 $



q_0

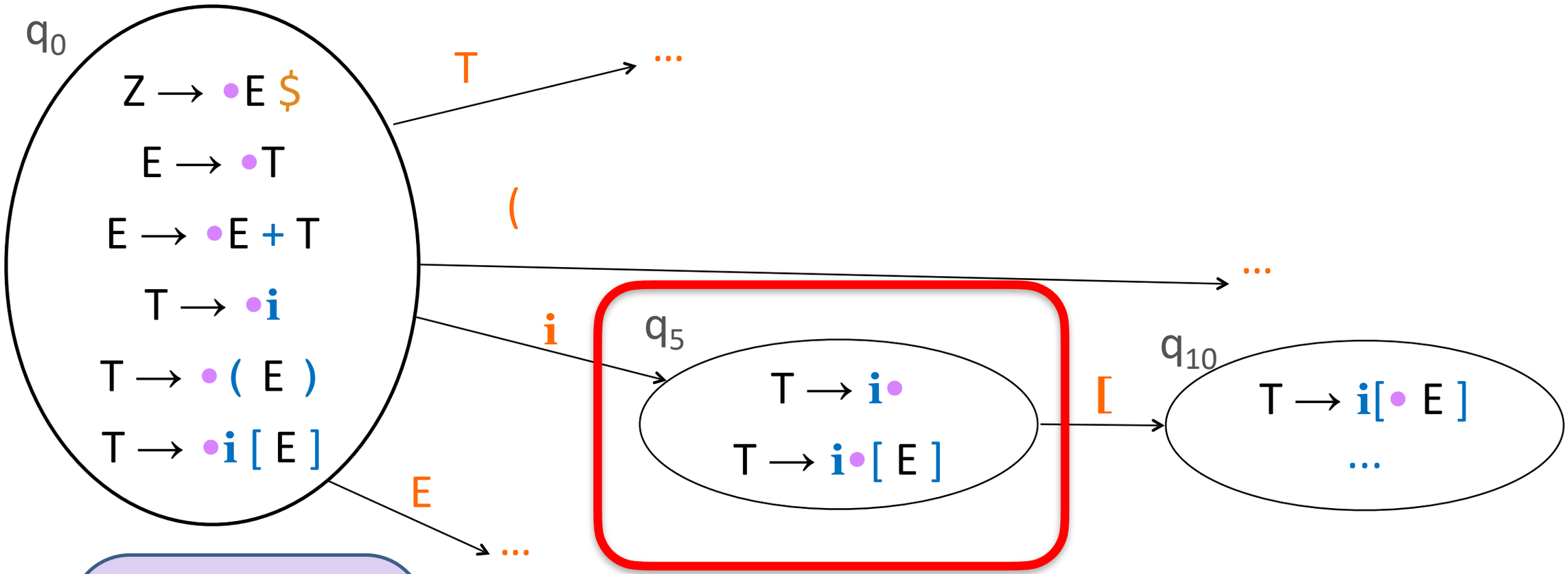
Are we done?

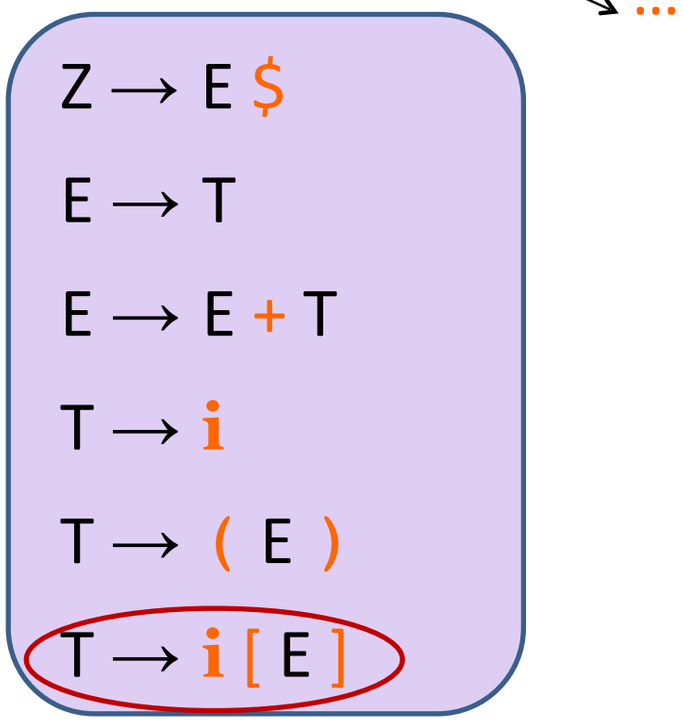
- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar

- …but the states are not always clear on what to do

⇒ **Cannot** make a *deterministic* ACTION table for every grammar

LR(0) Conflicts





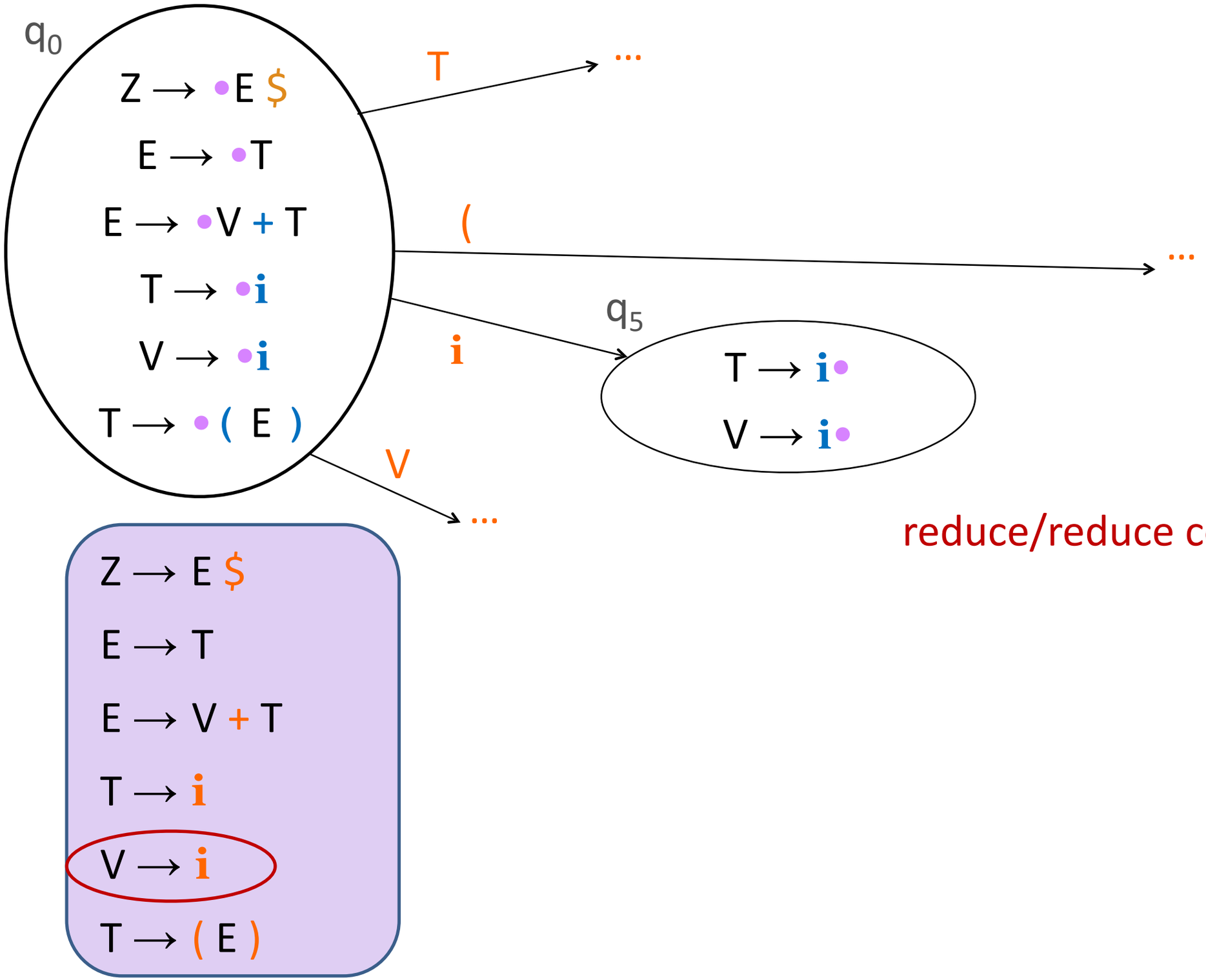
shift/reduce conflict

View in Action/Goto Table

- shift/reduce conflict…

	i	+	(	)	$	[	E	T
q_0	s5		s7				1	6
q_1		s3			acc			
q_2	r1	r1	r1	r1	r1	r1		
q_3	s5		s7					4
q_4	r3	r3	r3	r3	r3	r3		
q_5	r4	r4	r4	r4	r4	r4/s10		
q_6	r2	r2	r2	r2	r2	r2		
q_7	s5		s7				8	6
q_8		s3		s9				
q_9	r5	r5	r5	r5	r5	r5		

LR(0) Conflicts



reduce/reduce conflict

View in Action/Goto Table

- reduce/reduce conflict…

	i	+	(	)	$	E	T
q_0	s5		s7			1	6
q_1		s3			acc		
q_2	r1	r1	r1	r1	r1		
q_3	s5		s7				4
q_4	r3	r3	r3	r3	r3		
q_5	r4/r5	r4/r5	r4/r5	r4/r5	r4/r5		
q_6	r2	r2	r2	r2	r2		
q_7	s5		s7			8	6
q_8		s3		s9			
q_9	r5	r5	r5	r5	r5		

Can there be a shift/shift conflict?

LR(0) *vs.* ε-Rules

- Whenever a nonterminal has an ε production, it will be reduced as soon as it is reached in the grammar (remember, *without looking at the next token*).
- If the variable has another production with a terminal prefix, there is an inherent shift/reduce conflict

$A \to \varepsilon$

$A \to a\ A$

✗ Not good

- $A \to \bullet$ — reduce item
- $A \to \bullet a\ A$ — shift item
- both are in the closure of any item of the form $\{P \to \alpha \bullet A \beta\}$

$A \to \varepsilon$

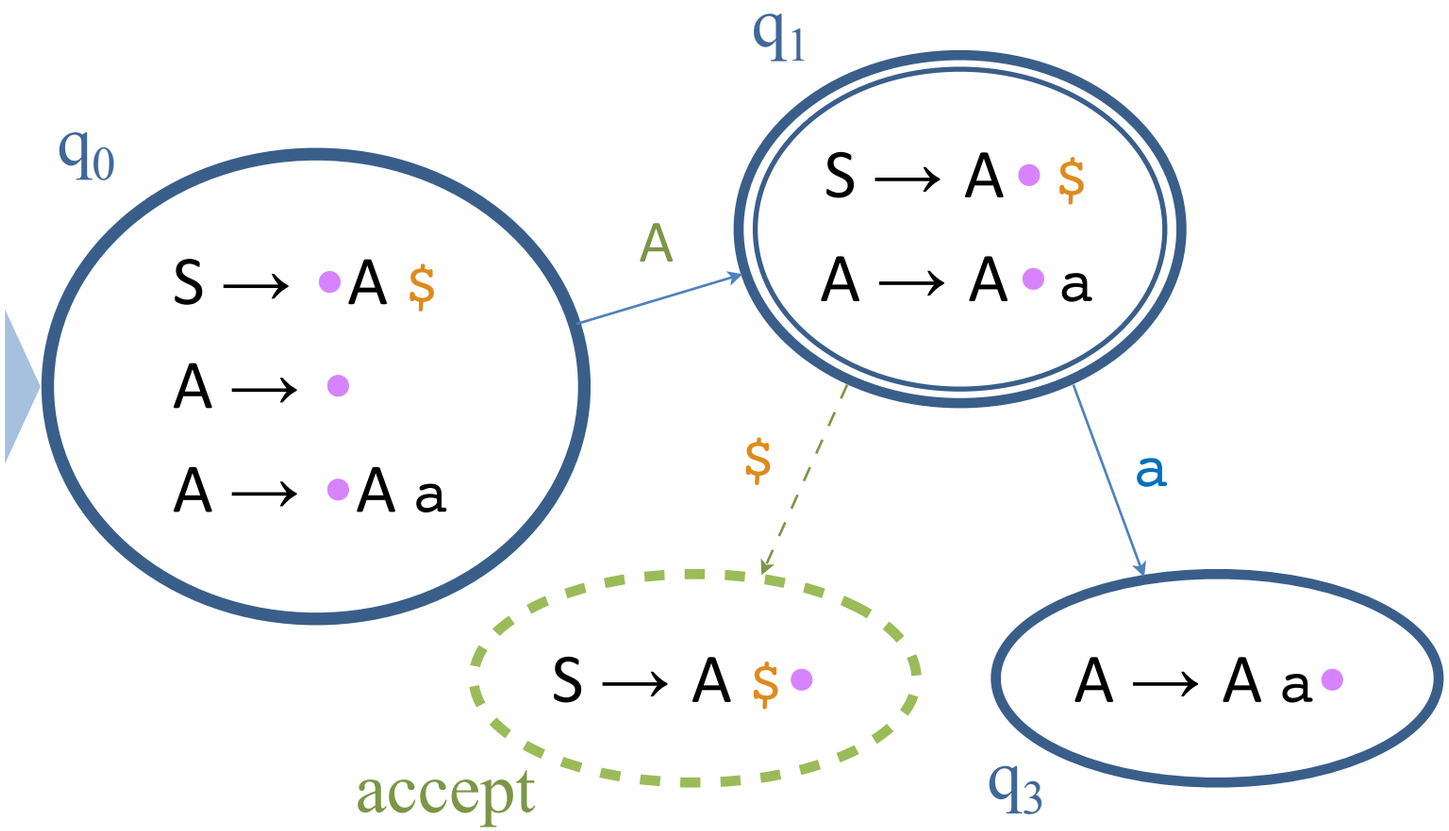
$A \to A\ a$

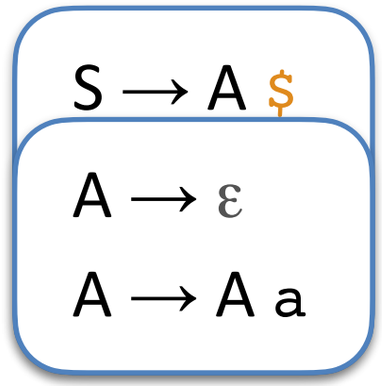
✓ This is fine

- $A \to \bullet$ — reduce item
- $A \to \bullet A\ a$
- No such thing as a shift/goto conflict

Good News: Left Recursion

- Left recursion is perfectly fine for LR(0) parsing and does not cause an infinite loop.





✓ This is fine

- A → • — reduce item
- A → •A a
- No such thing as a shift/goto conflict

input: aaa$ Transitions: $q_0 \rightsquigarrow$? ?

Can we do better?

Can we handle:

- Shift/reduce conflicts ?
- Reduce/reduce conflicts ?
- ε-rules: $A \rightarrow \varepsilon$ $A \rightarrow aA$?

Important Bottom-Up LR-Parsers

- LR(0) – simplest, explains basic ideas
- SLR(1) – simple, explains lookahead
- LR(1) – complicated, very powerful, expensive
- LALR(1) – complicated, powerful enough, used by automatic tools

LL(*k*) grammars: *k* = number of tokens we lookahead when we do a reduce operation

Back to Action/Goto Table

- Remember? Reductions ignore the input…

	i	+	(	)	$	E	T
q_0	s4		s6			g1	g5
q_1		s2			acc		
q_2	s4		s6				g3
q_3	r3	r3	r3	r3	r3		
q_4	r4	r4	r4	r4	r4		
q_5	r2	r2	r2	r2	r2		
q_6	s4		s6			g7	g5
q_7		s2		s8			
q_8	r5	r5	r5	r5	r5		

SLR(1) Grammars

- aka Simple LR(1) or SLR

- A string should only be reduced to a nonterminal N if the look-ahead is a token that can follow N
  - A reduce item $N \rightarrow \alpha\bullet$ is applicable only when the lookahead is in FOLLOW(N)

$$\text{FOLLOW}(N) = \{t \in T \mid S \Rightarrow^* \beta N t \gamma\} \cup \{\$ \mid S \Rightarrow^* \beta N\}$$

- Differs from LR(0) only on the original reduce rows
- Allows us to sometimes not reduce, instead shift (or do nothing — detect error sooner)

SLR Grammars and FOLLOW

$E \longrightarrow \overset{(1)}{E * B} \mid \overset{(2)}{E + B} \mid \overset{(3)}{B}$

$B \longrightarrow \underset{(4)}{0} \mid \underset{(5)}{1}$

Derivation (in reverse):

<u>0</u> + 0 * 1

<u>B</u> + 0 * 1 → ‘+’ ∈ FOLLOW(B)

E + <u>0</u> * 1

<u>E + B</u> * 1 → ‘*’ ∈ FOLLOW(B)

E * 1 → ‘*’ ∈ FOLLOW(E)

<u>E * B</u> → $ ∈ FOLLOW(B)

E

When reducing a sub-string to some nonterminal N, while the lookahead is a token a ⟶ “Na” is going to appear in the sentential form

GOTO/ACTION Table

	i	+	(	)	$	E	T
q_0	s4		s6			g1	g5
q_1		s2			acc		
q_2	s4		s6				g3
q_3		r3		r3	r3		
q_4	r4	r4	r4	r4	r4		
q_5	r2	r2	r2	r2	r2		
q_6	s4		s6			g7	g5
q_7		s2		s8			
q_8	r5	r5	r5	r5	r5		

(1) Z → E $

(2) E → T

(3) E → E + T

(4) T → i

(5) T → (E)

The tokens that can follow E are '+' ')' and '$'.

FOLLOW(E) = {+,), $}

GOTO/ACTION Table

	i	+	(	)	$	E	T
q_0	s4		s6			g1	g5
q_1		s2			acc		
q_2	s4		s6				g3
q_3		r3		r3	r3		
q_4	r4	r4	r4	r4	r4		
q_5		r2		r2	r2		
q_6	s4		s6			g7	g5
q_7		s2		s8			
q_8	r5	r5	r5	r5	r5		

(1) Z → E $

(2) E → T

(3) E → E + T

(4) T → i

(5) T → (E)

The tokens that can follow E are '+' ')' and '$'.

FOLLOW(E) = {+,), $}

GOTO/ACTION Table

	i	+	(	)	$	E	T
q_0	s4		s6			g1	g5
q_1		s2			acc		
q_2	s4		s6				g3
q_3		r3		r3	r3		
q_4		r4		r4	r4		
q_5		r2		r2	r2		
q_6	s4		s6			g7	g5
q_7		s2		s8			
q_8		r5		r5	r5		

(1) Z → E $

(2) E → T

(3) E → E + T

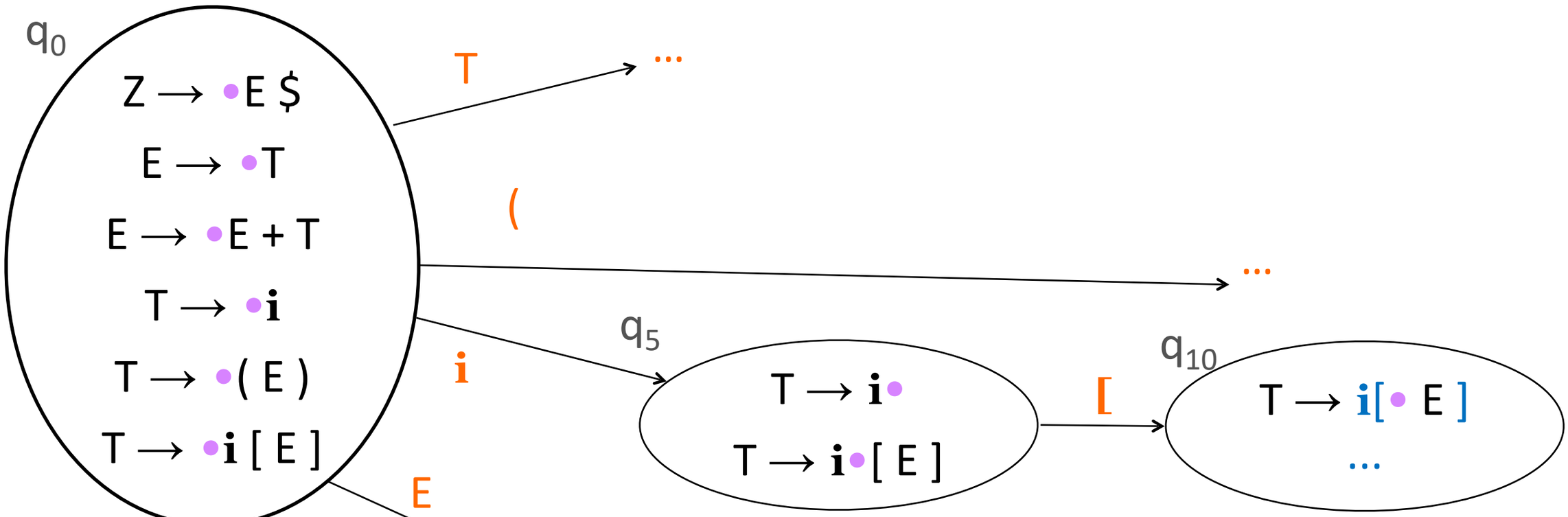
(4) T → i

(5) T → (E)

Same for T

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

Now let's add "T → **i** [E]"



Z → E $

E → T

E → E + T

T → **i**

T → (E)

T → **i** [E]

~~shift/reduce conflict~~

[∉ FOLLOW(T) = {+,), $}

Now let's add "T → **i** [E]"

	i	+	(	)	[	]	$	E	T
q_0	s4		s6					g1	g5
q_1		s2					acc		
q_2	s4		s6						g3
q_3		r3		r3			r3		
q_4		r4		r4	s9		r4		
q_5		r2		r2			r2		
q_6	s4		s6					g7	g5
q_7		s2		s8					
q_8		r5		r5			r5		
q_9	s4		s6					g10	
⋮									

yay 😊

(1) Z → E $

(2) E → T

(3) E → E + T

(4) T → **i**

(5) T → (E)

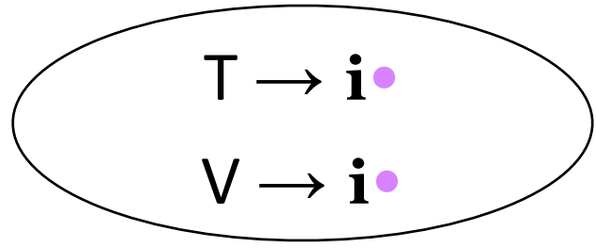
(6) T → **i** [E]

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

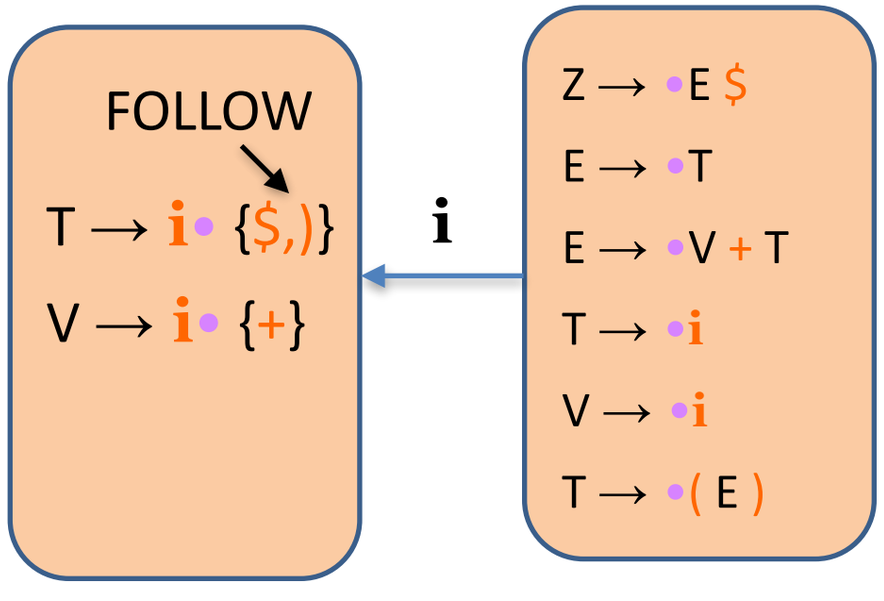
SLR(1): check next token when reducing

- Previous example demonstrates the elimination of a shift/reduce conflict
- Can eliminate reduce/reduce conflicts when conflicting rules' left-hand sides satisfy:

$$FOLLOW(T) \cap FOLLOW(V) = \emptyset$$



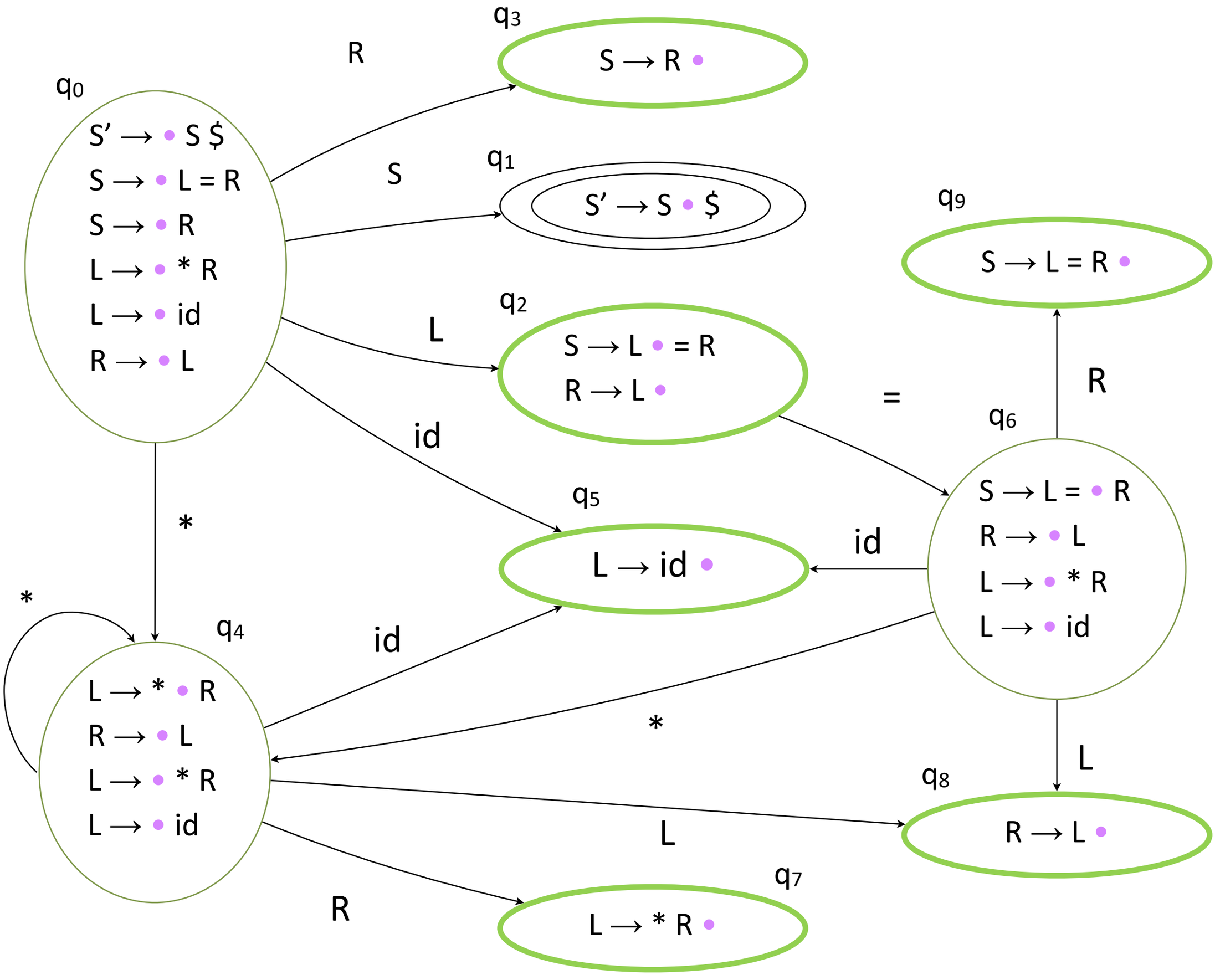
- But cannot resolve all shift/reduce and reduce/reduce conflicts!



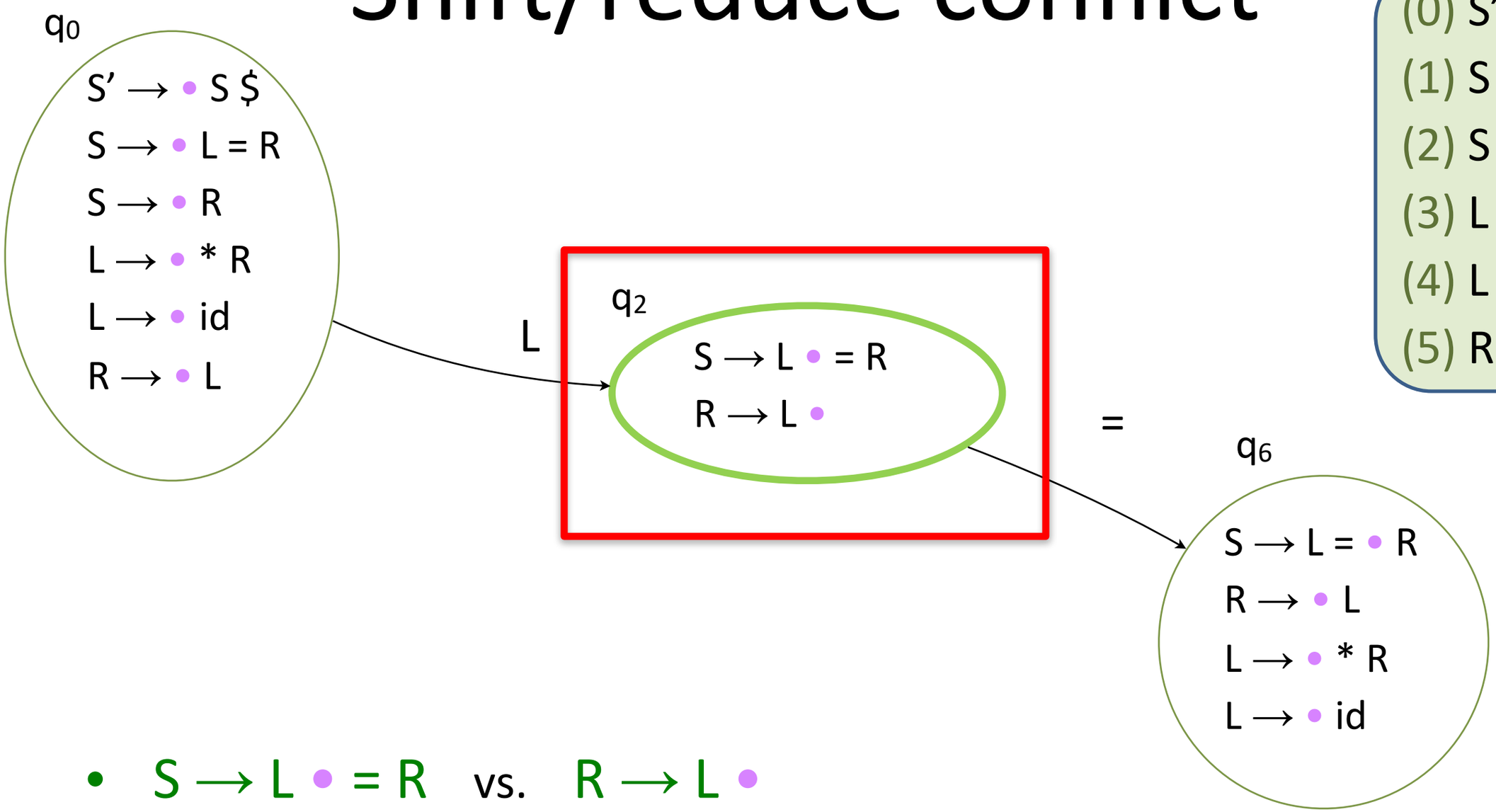
Consider this non-LR(0) grammar

(0) S’ → S $
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

id
***id
id = id
*id = id
id = *id
***id = **id
…


q0
S' → • S $
S → • L = R
S → • R
L → • * R
L → • id
R → • L
R
S
L
id
*
q3
S → R •
q1
S' → S • $
q2
S → L • = R
R → L •
q5
L → id •
q9
S → L = R •
=
q6
S → L = • R
R → • L
L → • * R
L → • id
R
id
q4
L → * • R
R → • L
L → • * R
L → • id
*
id
*
L
q8
R → L •
L
R
q7
L → * R •

Shift/reduce conflict



(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

- S → L • = R vs. R → L •
- FOLLOW(R) contains "="

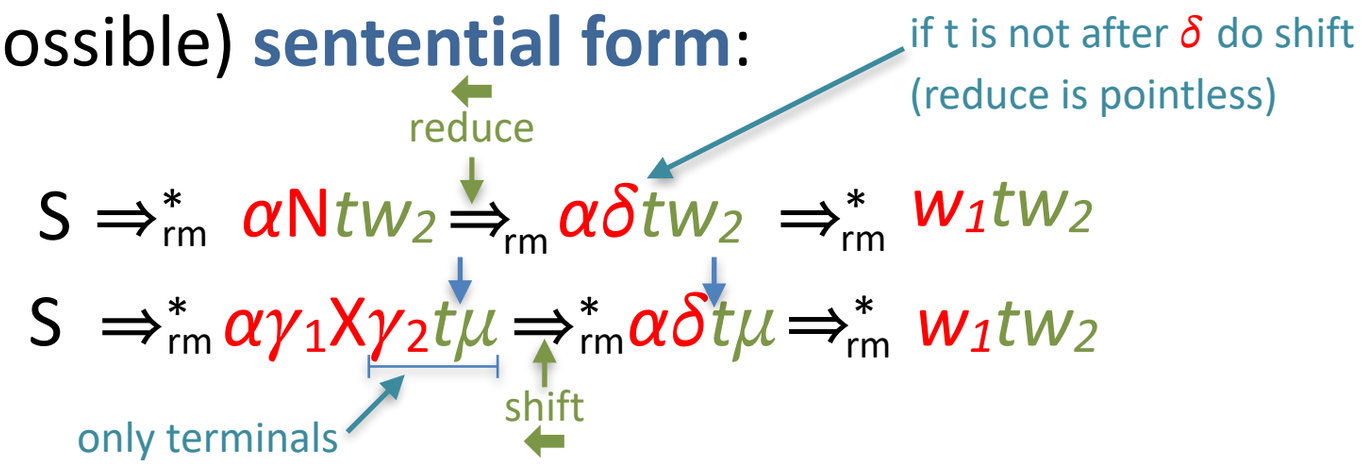
$$S \Rightarrow L = R \Rightarrow * R = R$$

⇒ SLR **cannot resolve** the conflict in this case

Resolving the Conflict

- In SLR(1): a reduce item $N \longrightarrow \delta\bullet$ is applicable when the lookahead is in FOLLOW(N)
- But when we reach a **reduce state**, there is a whole sequence of **reduction steps** that generate a prefix of (possible) **sentential form**:

$$input = w_1 t w_2 \rightarrow^* \alpha\delta \overset{?}{\rightarrow} \alpha N$$



- Given αN, we can ask what the next token of N may be in any sentential form whose prefix is αN:

$$\text{FOLLOW}(\alpha N) = \{\, t \mid S \Rightarrow^*_{rm} \alpha N t \beta \,\}$$

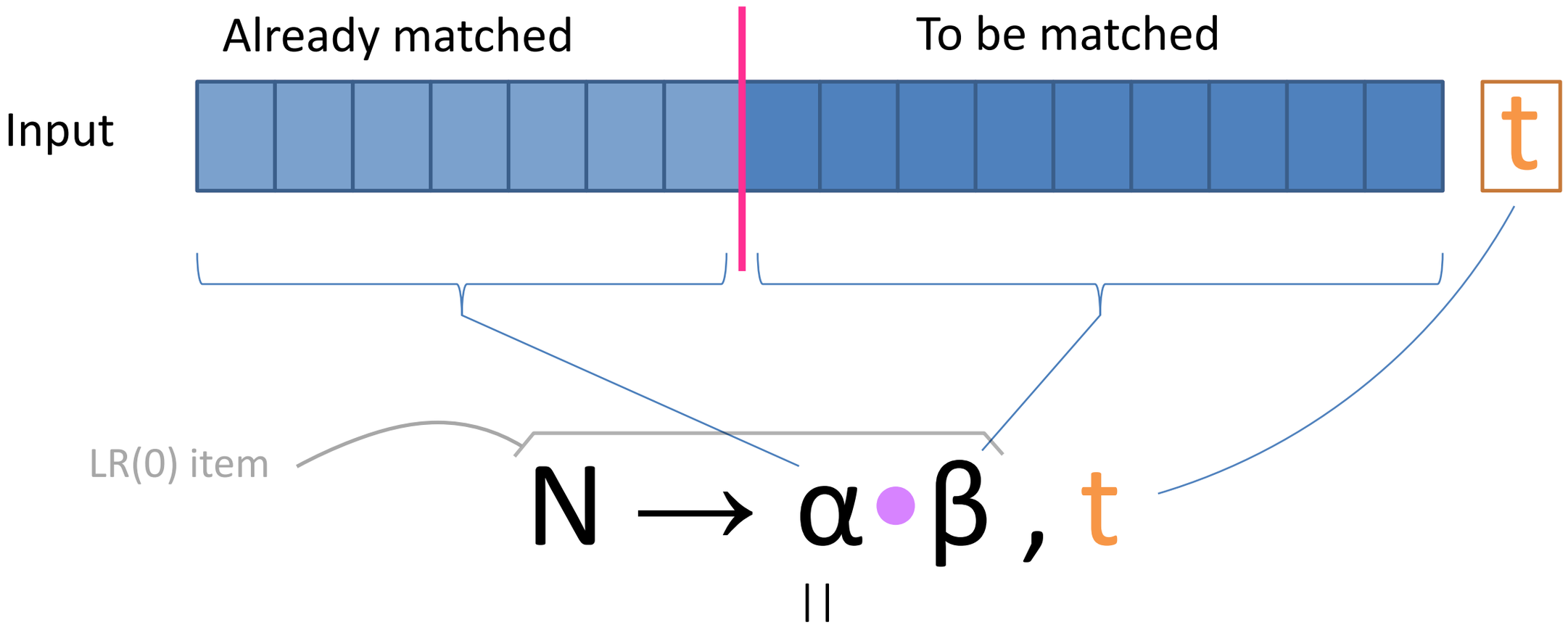
- Looking at FOLLOW(αN) is more restrictive than looking at the FOLLOW of the last variable N.

$$\text{FOLLOW}(\alpha N) \subseteq \text{FOLLOW}(N) = \{\, t \in \text{FIRST}(\beta) \mid S \Rightarrow^*_{rm} \gamma N \beta \,\}$$

  - In a way, FOLLOW(N) merges look-ahead for all possible occurrences of N in a sentential form

LR(1) Item

LR(1) tracks FOLLOW(αN) by keeping a look-ahead with **each** LR(1) item

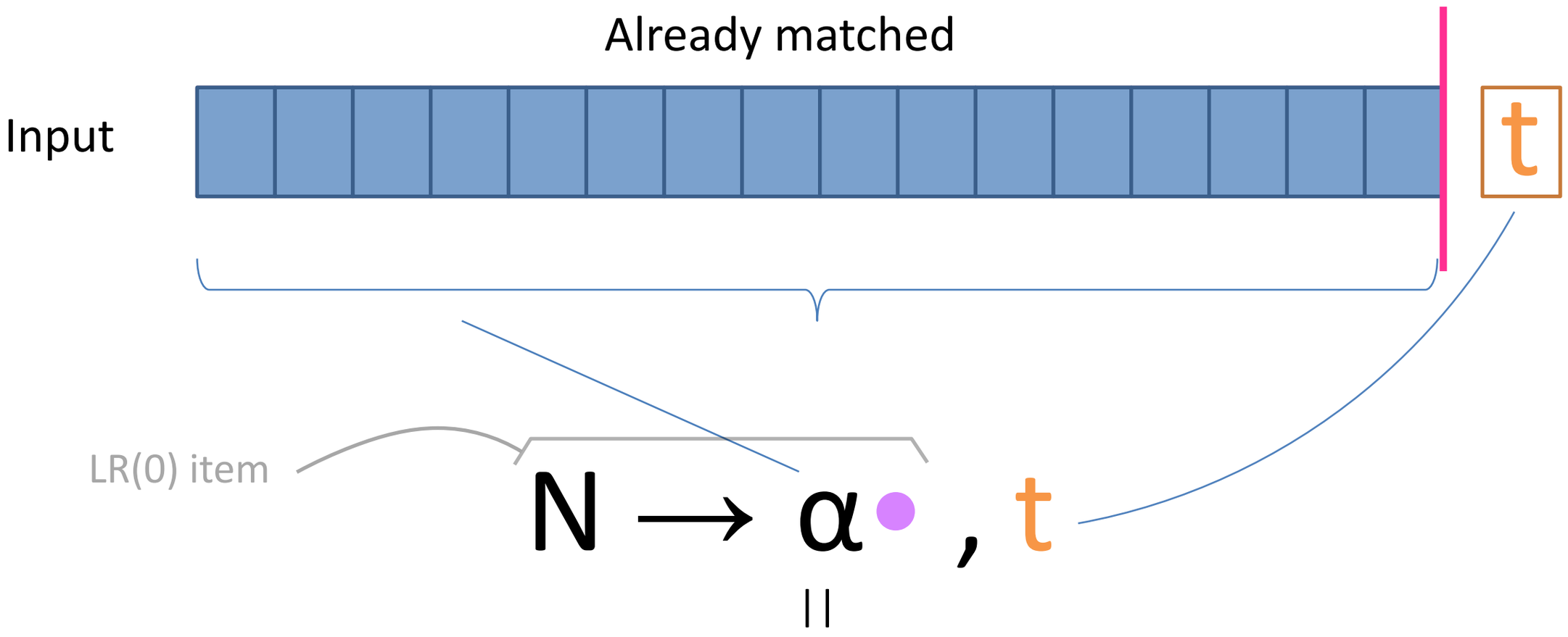


So far we've matched α, expecting to see β,

followed by the lookahead t

LR(1) Item

LR(1) tracks FOLLOW(αN) by keeping a look-ahead with **each** LR(1) item



So far we've matched α, if the next token is the lookahead t,

we should reduce α to N

LR(1) Item

- Example: the production L → id yields the following LR(1) items

(0) S’ → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

L → • id
L → id •

L → • id, *
L → • id, =
L → • id, id
L → • id, $
L → id •, *
L → id •, =
L → id •, id
L → id •, $

will never be generated, since *, id ∉ FOLLOW(L)

(by construction)

SLR

Reduce only when next token is in FOLLOW(L) = { = , $ }

LR(1)

Refines FOLLOW by using the RHS of the rules

Creating the states for LR(1)

- We start with the initial state:
  - q_0 will be the closure of: $\{S' \rightarrow \bullet S\ ,\ \$\}$

- Closure set for LR(1):

If the set contains an item of the form

$$A \rightarrow \alpha \bullet B\ \beta,\ c$$

then it must *also* contain an item

$$B \rightarrow \bullet \delta,\ d$$

for every production $B \rightarrow \delta$ and every token $d \in \text{FIRST}(\beta c)$

$$\begin{aligned}\text{FIRST}(\beta c) &= \{t \in T \mid \beta c \Rightarrow^* t\gamma\} \\ &= \text{FIRST}(\beta) \setminus \{\varepsilon\} \cup \{c \mid \beta \Rightarrow^* \varepsilon\}\end{aligned}$$

Closure of {S’ ⟶ ●S , $}

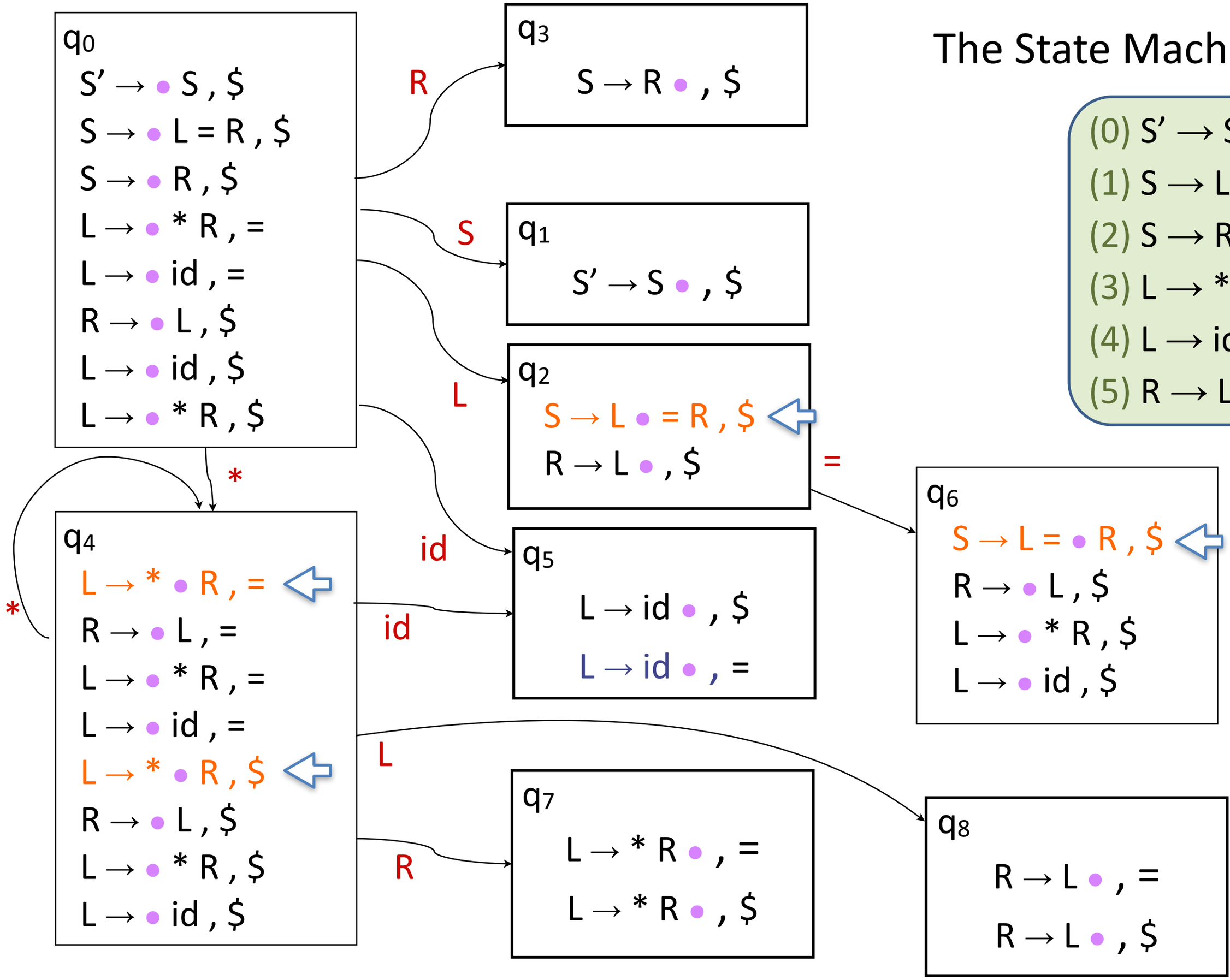
No need to have $ in the grammar explicitly

(0) S’ ⟶ S
(1) S ⟶ L = R
(2) S ⟶ R
(3) L ⟶ * R
(4) L ⟶ id
(5) R ⟶ L

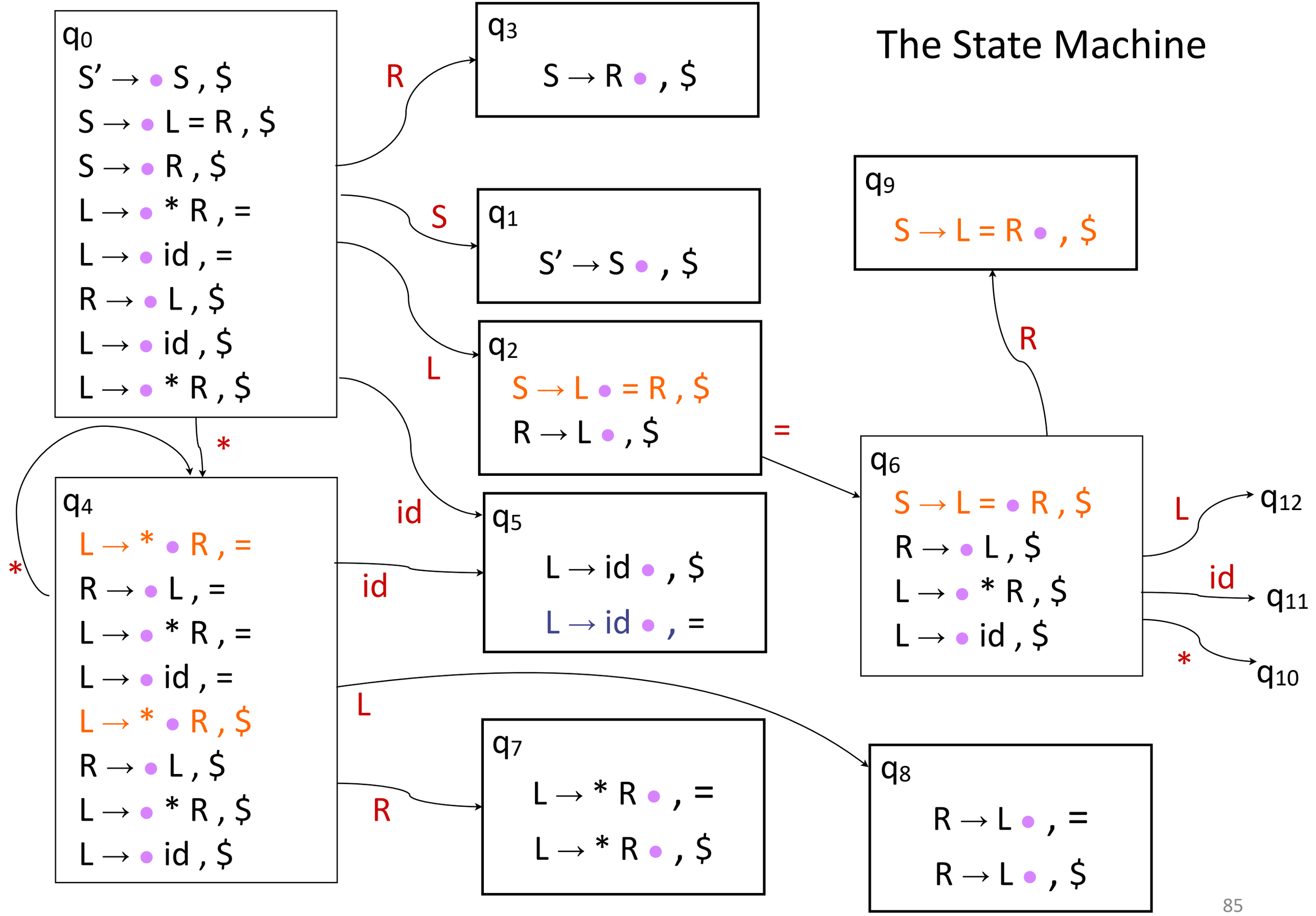
- We would like to add rules that start with S, but keep track of possible lookahead.
  - S’ ⟶ ● S , $
  - S ⟶ ● L = R , $ – Rules for S
  - S ⟶ ● R , $
  - L ⟶ ● * R , = – Rules for L
  - L ⟶ ● id , =
  - R ⟶ ● L , $ – Rules for R
  - L ⟶ ● * R , $ – More rules for L
  - L ⟶ ● id , $

The State Machine

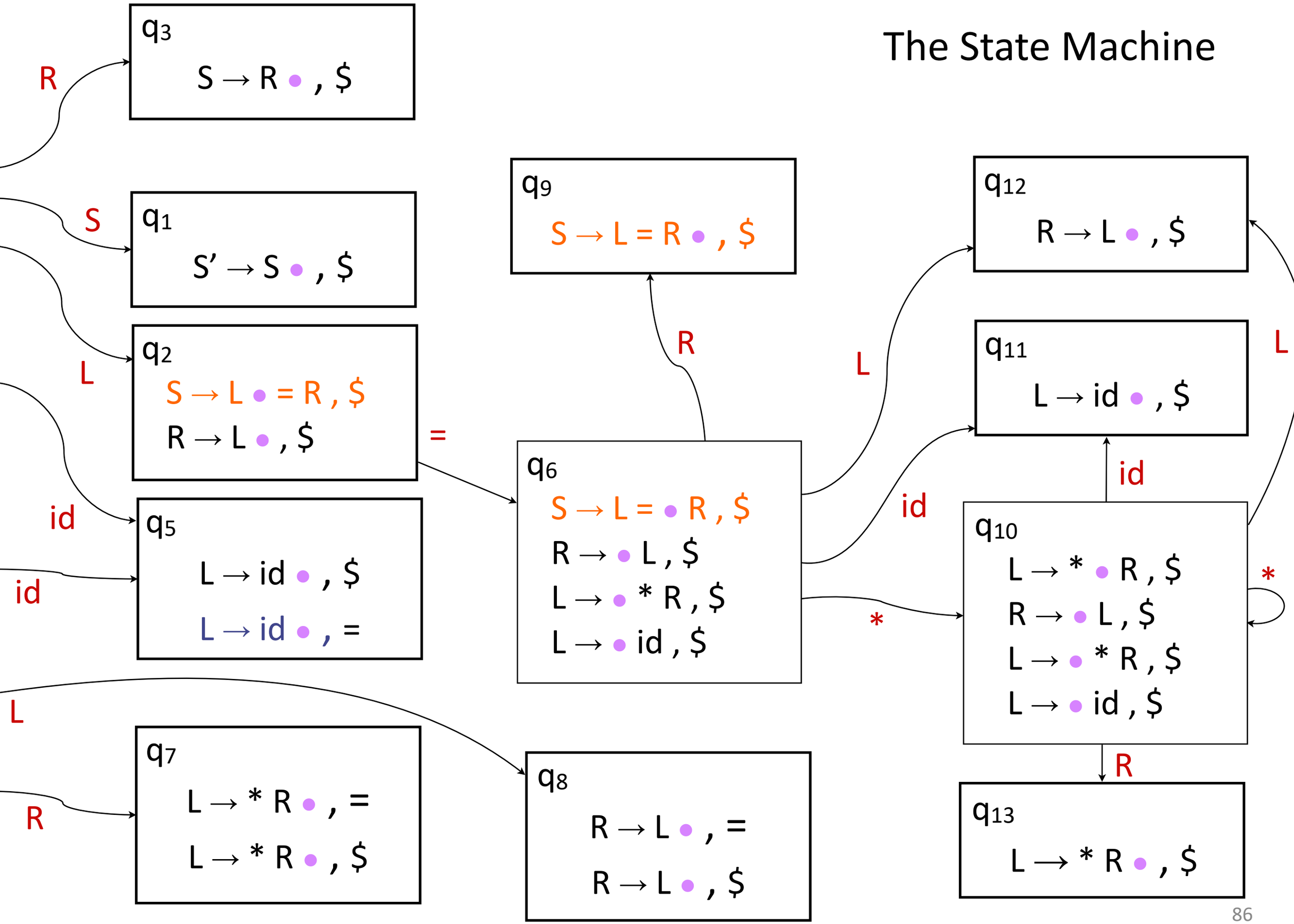
(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L



The State Machine



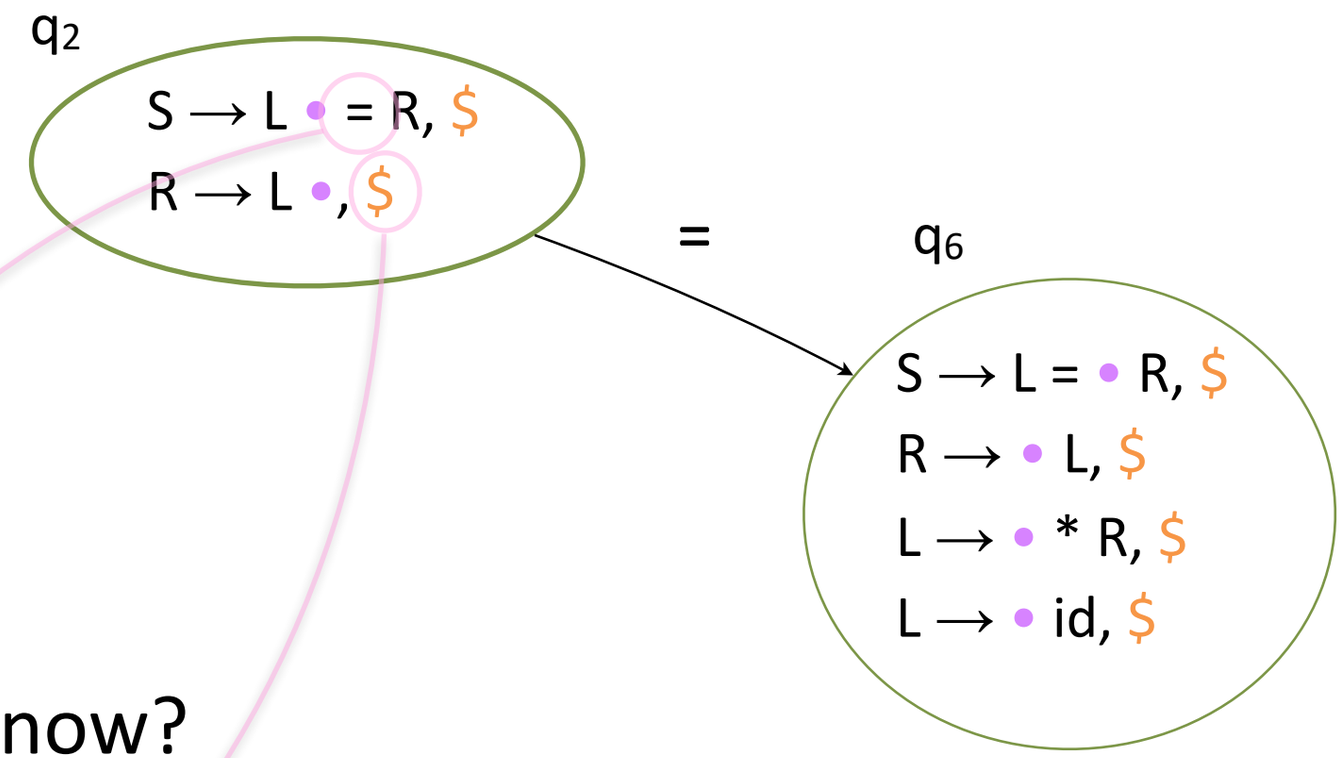
The State Machine



Back to the conflict

(0) S’ ⟶ S
(1) S ⟶ L = R
(2) S ⟶ R
(3) L ⟶ * R
(4) L ⟶ id
(5) R ⟶ L

FOLLOW(R) contains “=”



- Is there a conflict now?

State	ACTION				GOTO	
	*	=	id	$	E	T
q_2		s6		r5		

Building the Tables

- Similarly to LR(0) and SLR, we start with the automaton.
- Turn each transition in a token column to a shift.
- The variables' columns form the GOTO section.
- The "acc" is put in the $ column, for any state that contains ($S' \rightarrow S\bullet$, $) .
- For any state that contains an item of the form ($A \rightarrow \beta\bullet$, a), where $A \rightarrow \beta$ is rule number (*m*), use "reduce *m*" for the row of this state and the column of token a.

Building the LR(1) Table

	ACTION				GOTO		
	id	*	=	$	S	R	L
q_0	s5	s4			g1	g3	g2
q_1				acc			
q_2			s6	r5			
q_3				r2			
q_4	s5	s4				g7	g8
q_5			r4	r4			
q_6	s11	s10				g9	g12
q_7			r3	r3			
q_8			r5	r5			
q_9				r1			
q_{10}	s11	s10				g13	g12
q_{11}				r4			
q_{12}				r5			
q_{13}				r3			

(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

q_2
S → L • = R , $
R → L • , $

FOLLOW(R) = {=, $}

Bottom Up Parsing

✳LR(k)

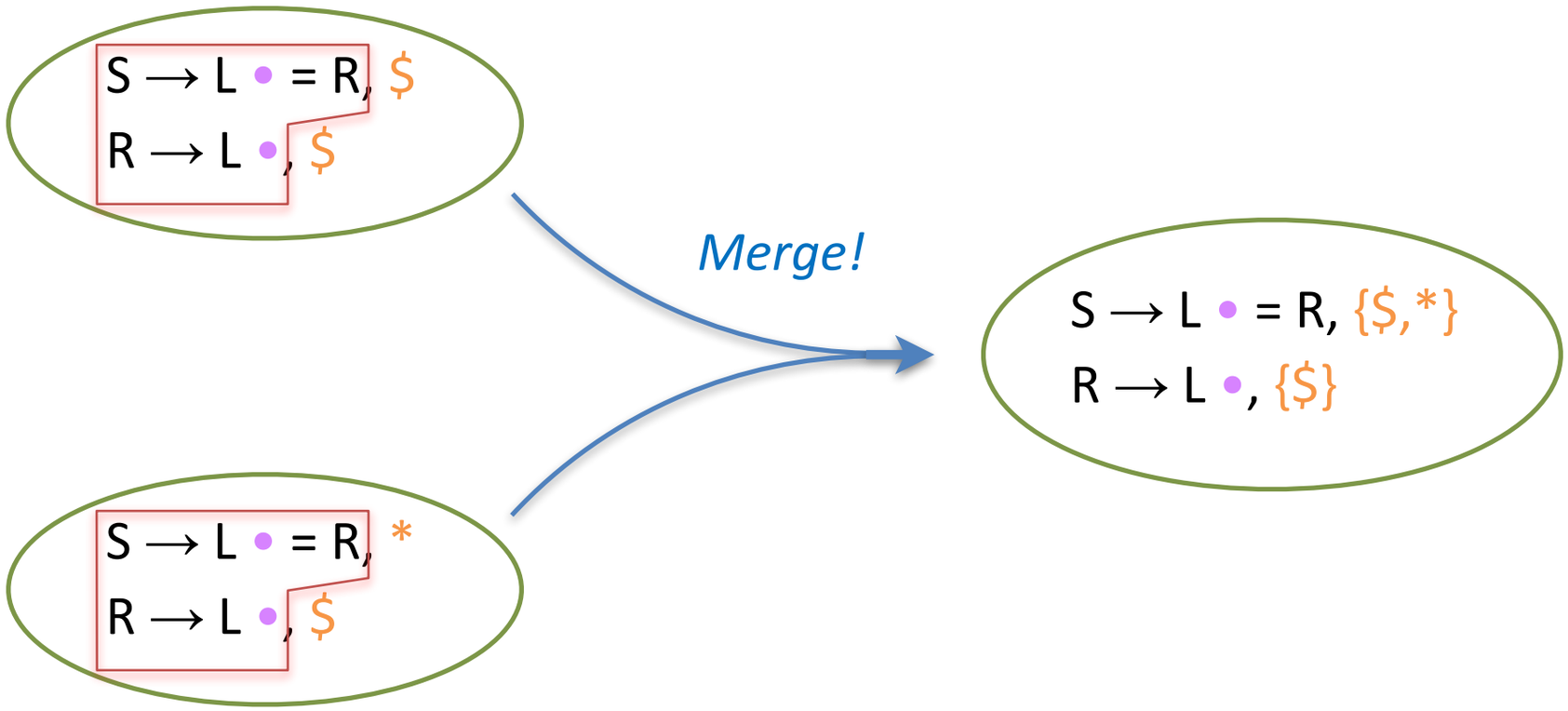
✳SLR

- All follow the same table+stack algorithm
- Differ on type of “LR Items”, translation to table

LR(0)	SLR(1)	LR(1)
$N \rightarrow \alpha \bullet \beta$	$N \rightarrow \alpha \bullet \beta$ $N \rightarrow \alpha \bullet$ {FOLLOW(N}	$N \rightarrow \alpha \bullet \beta$, σ

LALR

- Goal. LR(1) creates a lot of states that look similar. Can we reduce the number of states?
- Idea. Merge states that have exactly the same set of LR(0) items (**core**), representing the lookaheads as *sets*.

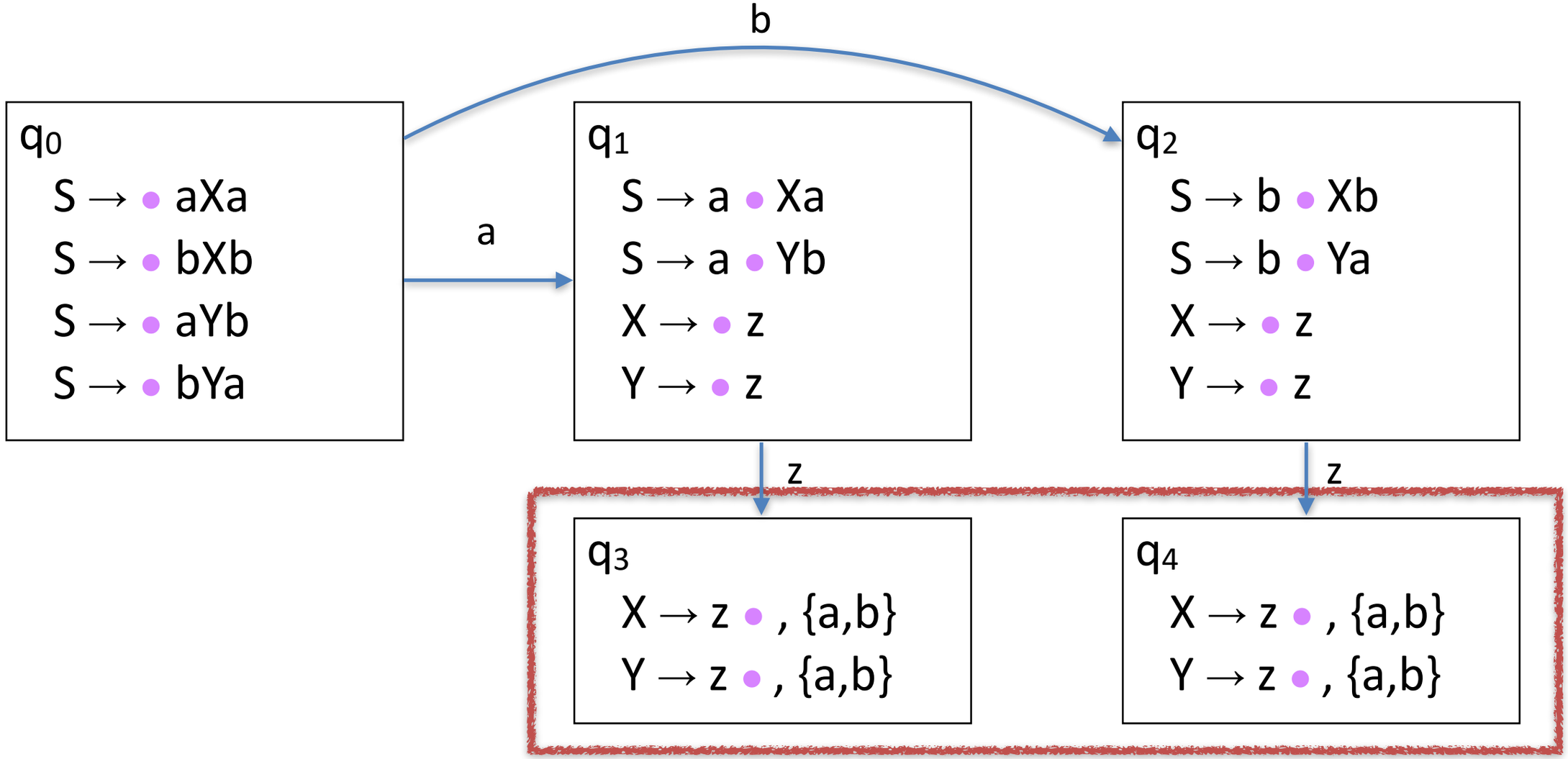


✳ Of course, we may get *more* conflicts as a result.

Example: SLR(1) Parsing

$$S \rightarrow aXa \mid bXb \mid aYb \mid bYa$$
$$X \rightarrow z$$
$$Y \rightarrow z$$

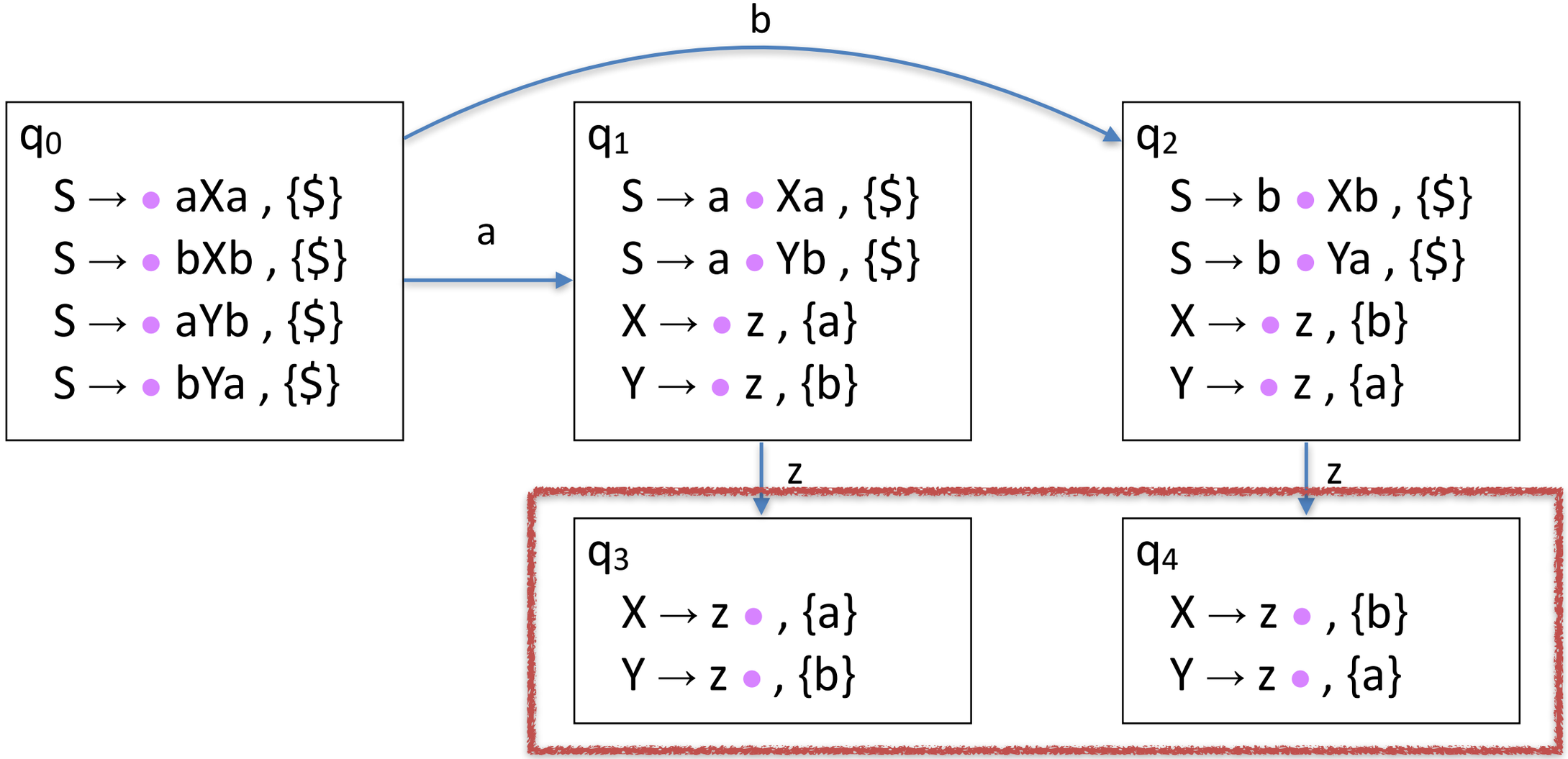
SLR(1) Automaton (partial)



Example: LR(1) Parsing

$$S \rightarrow aXa \mid bXb \mid aYb \mid bYa$$
$$X \rightarrow z$$
$$Y \rightarrow z$$

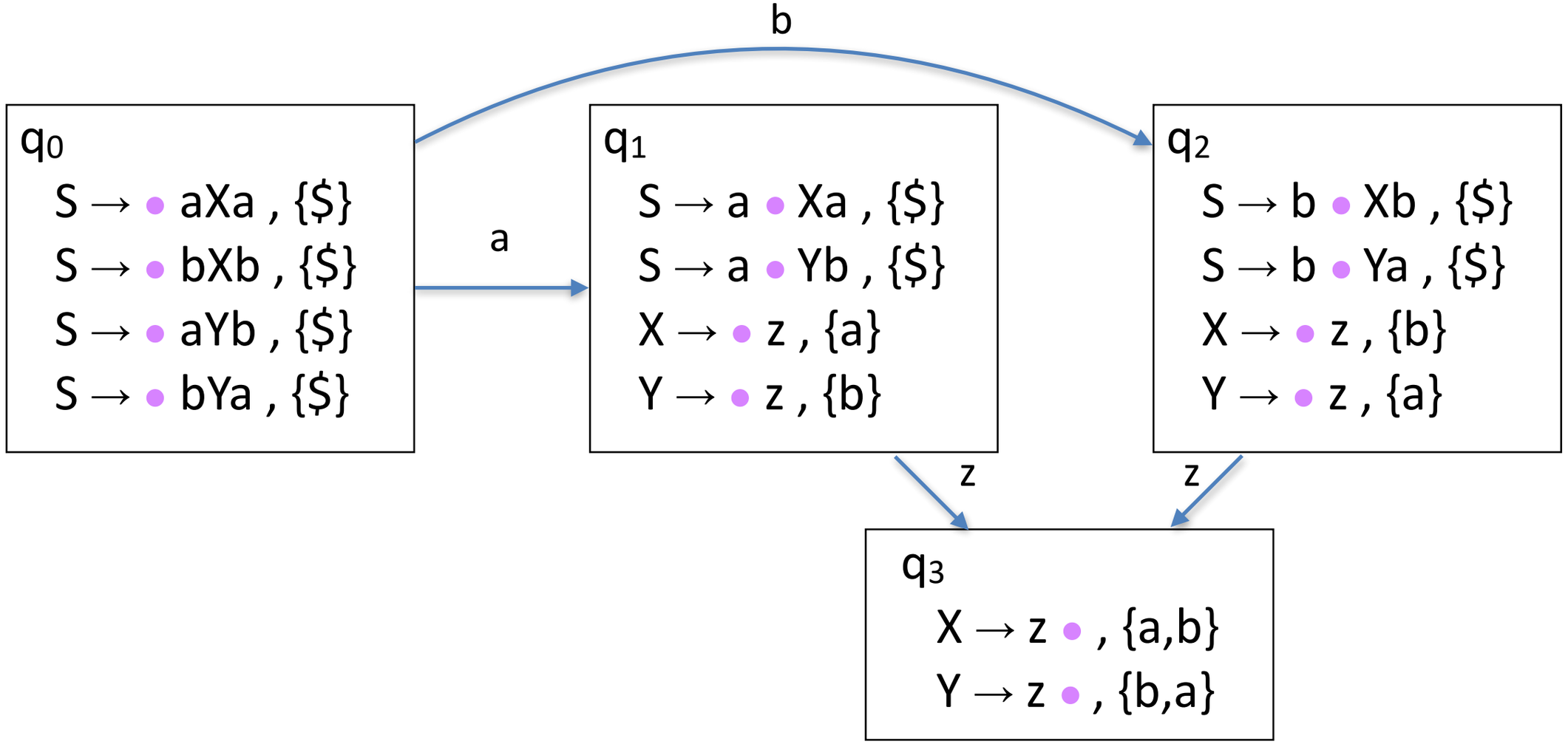
LR(1) Automaton (partial)



LALR Parsing

$$S \rightarrow aXa \mid bXb \mid aYb \mid bYa$$
$$X \rightarrow z$$
$$Y \rightarrow z$$

LR(1) Automaton (partial)



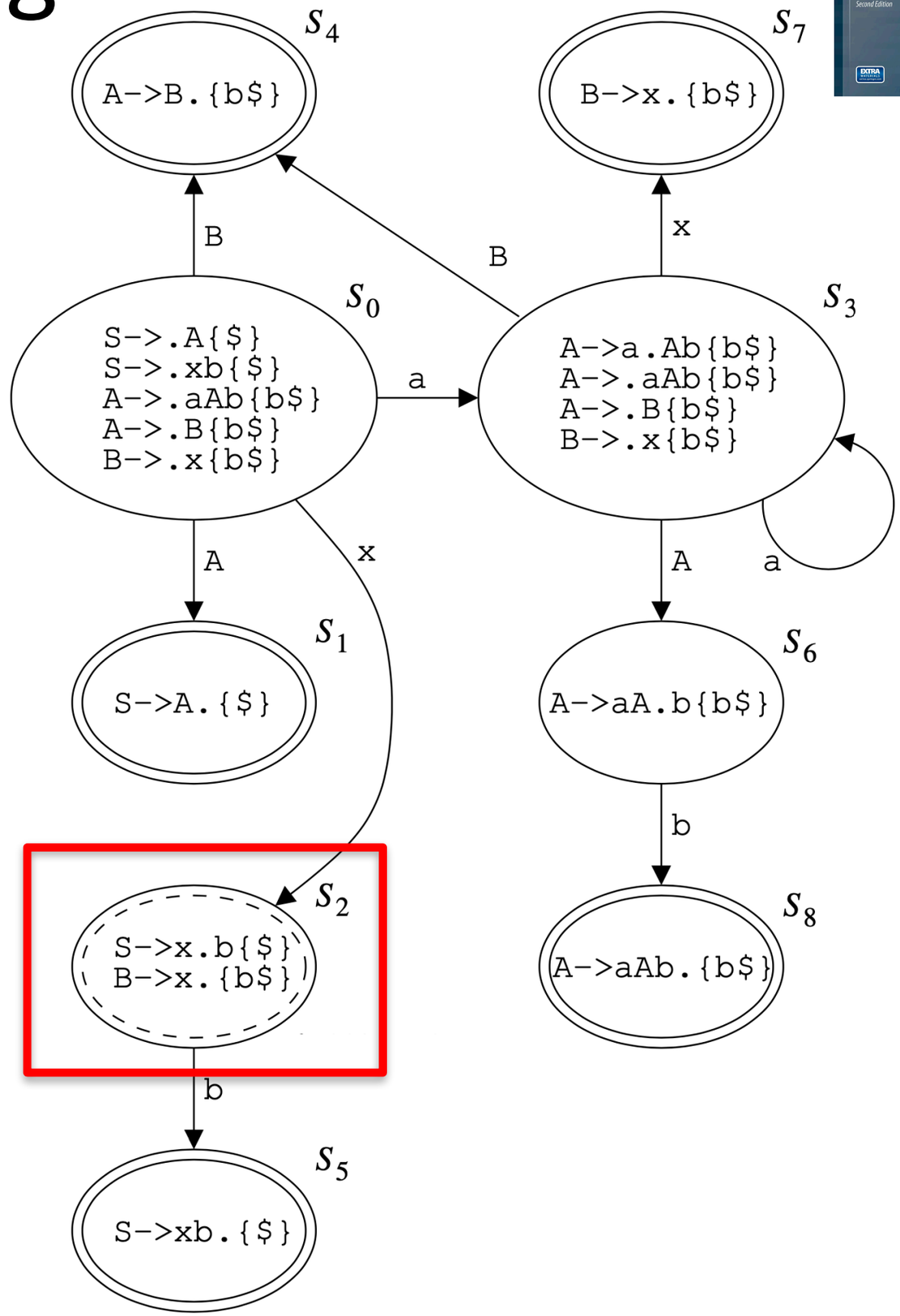
Example: SLR(1) Parsing

$$S \rightarrow A \mid xb$$
$$A \rightarrow aAb \mid B$$
$$B \rightarrow x$$

Remark: not LL(1) either — why?

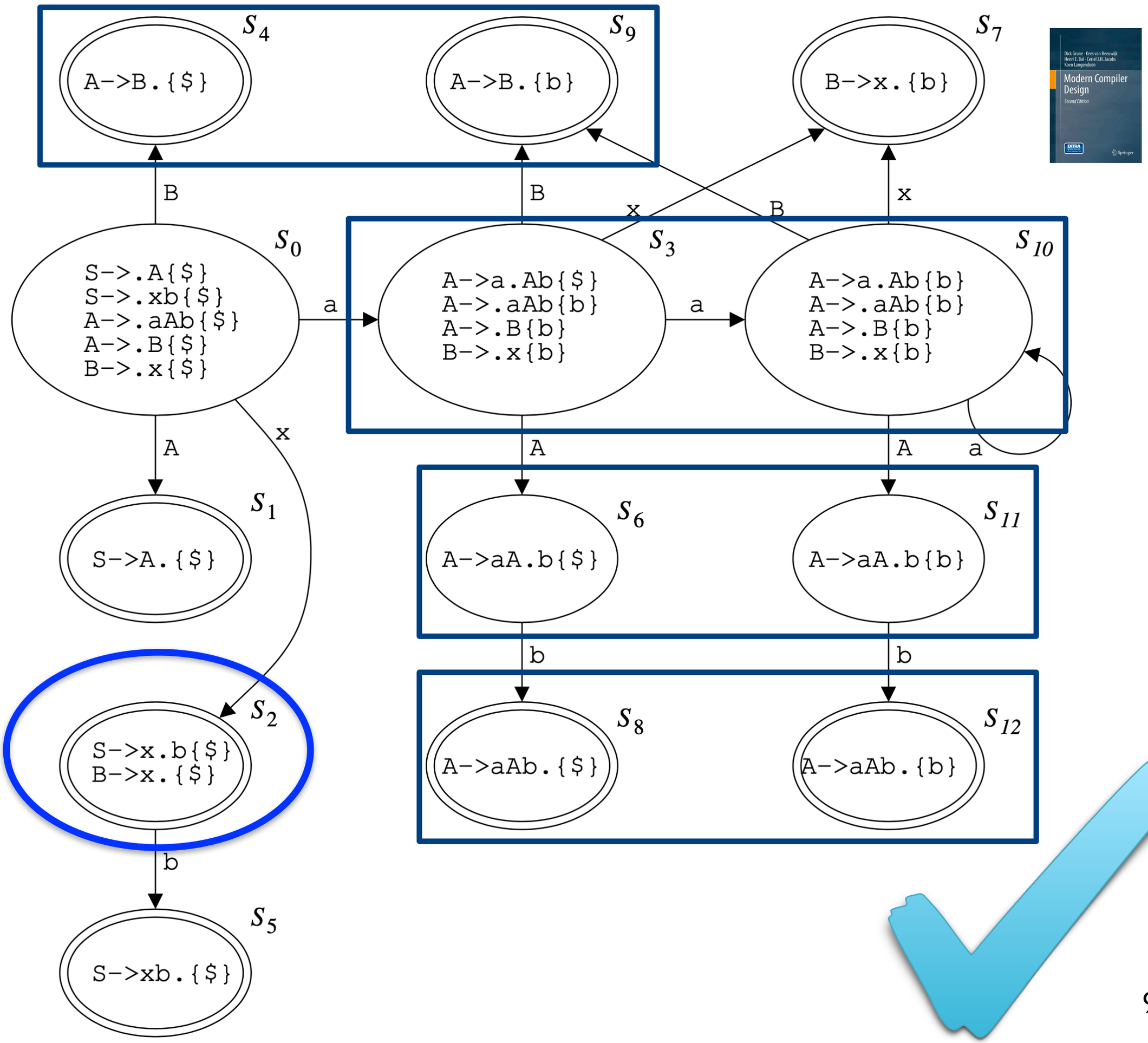
FOLLOW(S) = { $ }
FOLLOW(A) = { b $ }
FOLLOW(B) = { b $ }

Follow set is written in automaton inside { }

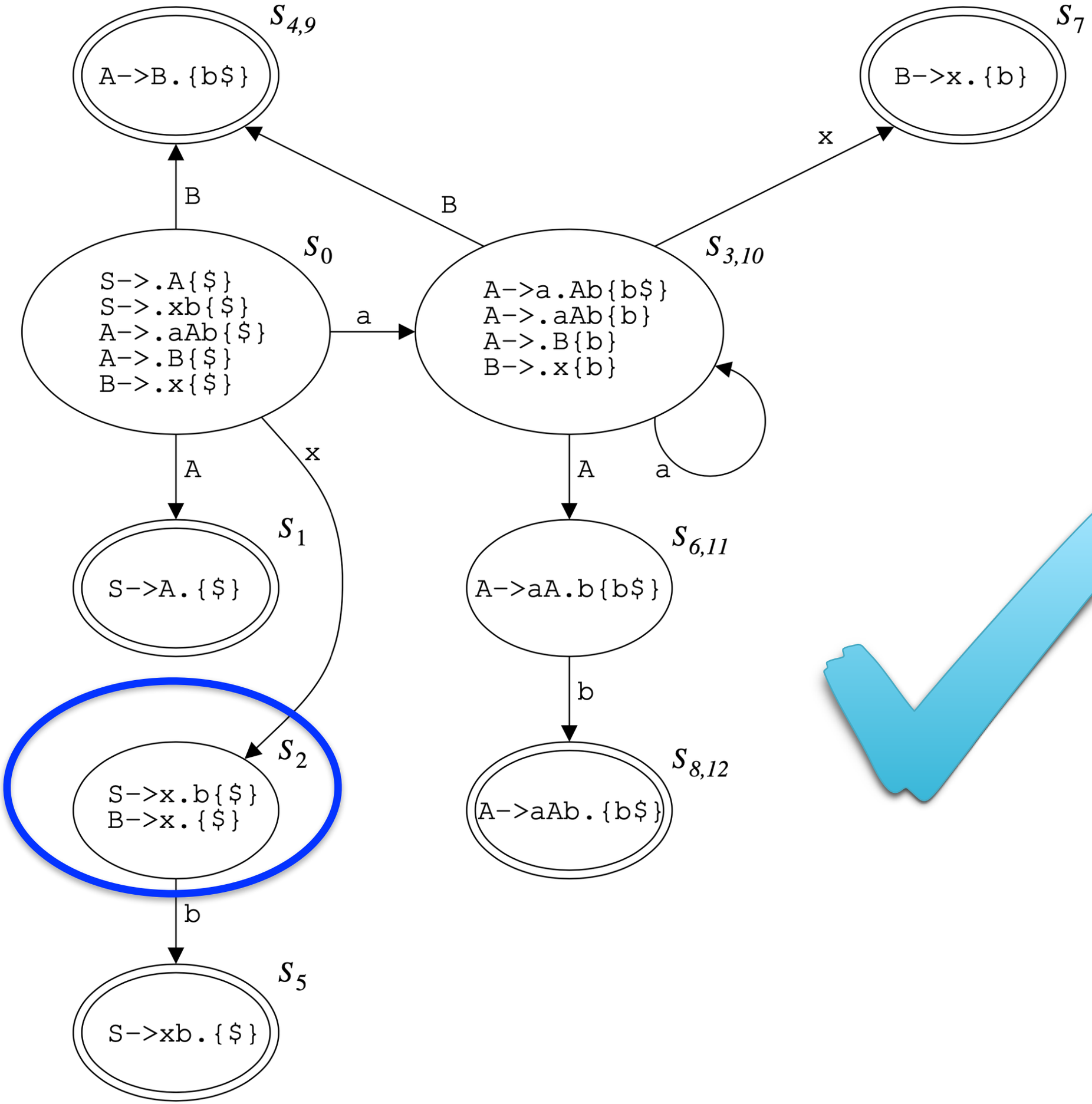


SLR(1) Automaton

Example: LR(1) Automaton



Example: LALR Automaton



Bottom-up Parsing

- LR(k)
- SLR
- LALR

- All follow the same pushdown-based algorithm
- Differ on granularity of lookahead in “LR Items”

LR(0) SLR(1)	LR(1)	LALR
$N \rightarrow \alpha \bullet \beta$	$N \rightarrow \alpha \bullet \beta$, t $N \rightarrow \alpha \bullet \beta$, t’	$N \rightarrow \alpha \bullet \beta$, {t,t’}

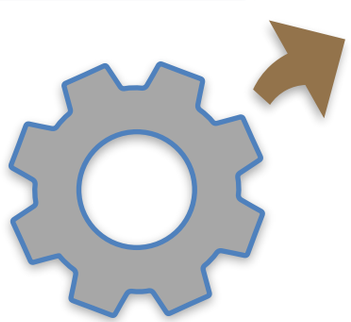
Building the Parse Tree

Building the Parse Tree

- Done at the time of reduce.

State	ACTION					GOTO	
	*	+	0	1	$	E	T
q_8	r2	r2			r2		

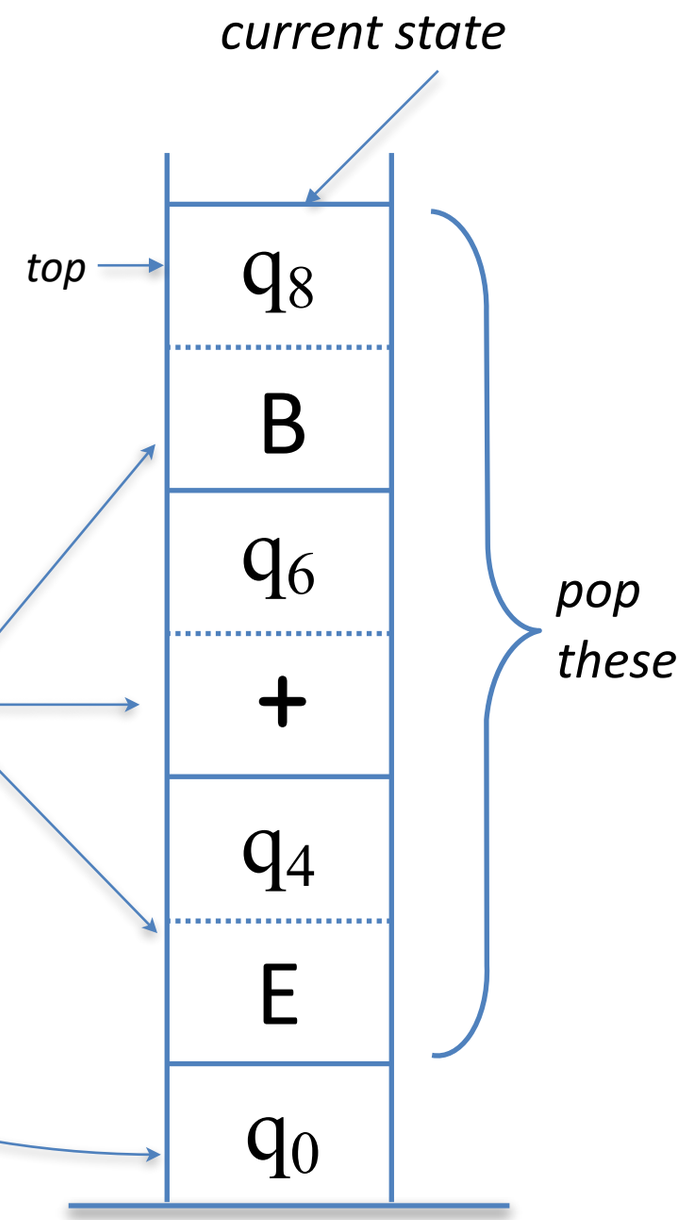
(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1



```
{:
   RESULT = new Node("E");
   RESULT.addChild(stack[top-6]);
   RESULT.addChild(stack[top-4]);
   RESULT.addChild(stack[top-2]);

   pop(6);

   next = GOTO[stack[top], "E"];
   push(RESULT);
   push(next);
:}
```



Building the Parse Tree ("CUP")

- Done at the time of reduce.

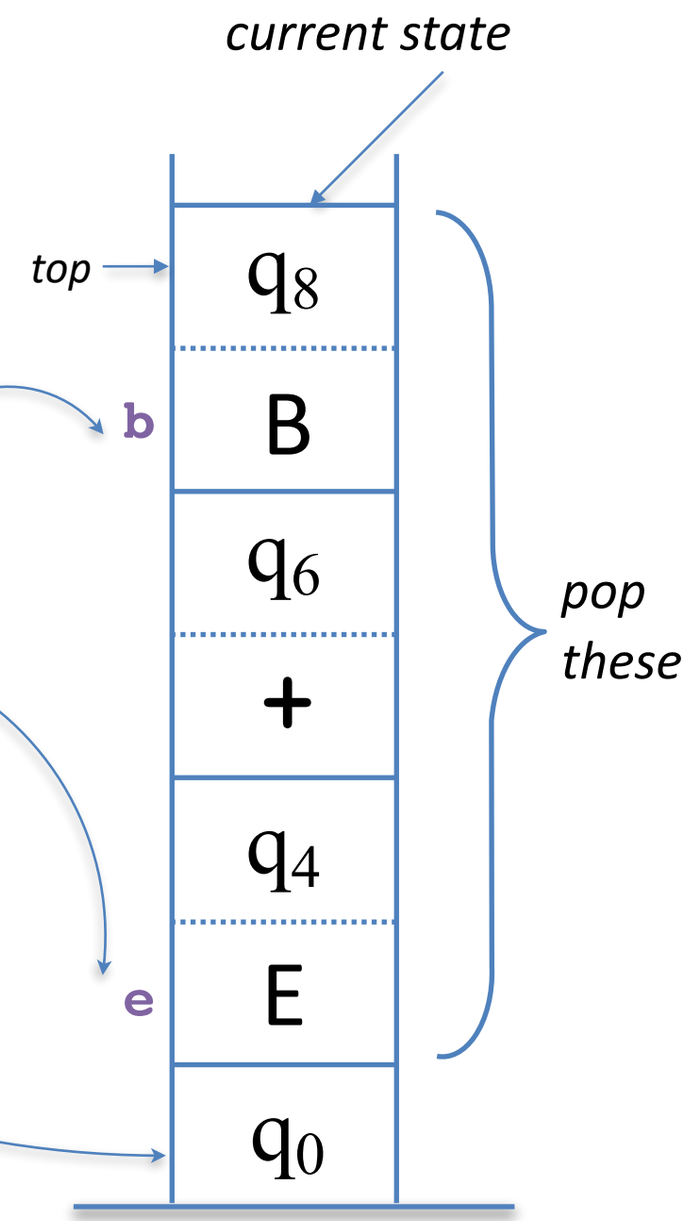
State	ACTION					GOTO	
	*	+	0	1	$	E	T
q_8	r2	r2			r2		

(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

```
E ::= E:e PLUS B:b {:
   RESULT = new Node("E");
   RESULT.addChild(e);
   RESULT.addChild(new Node("+"));
   RESULT.addChild(b);

   pop(6);

   next = GOTO[stack[top], "E"];
   push(result);
   push(next);
:}
```



Building the Parse Tree ("Bison")

- Done at the time of reduce.

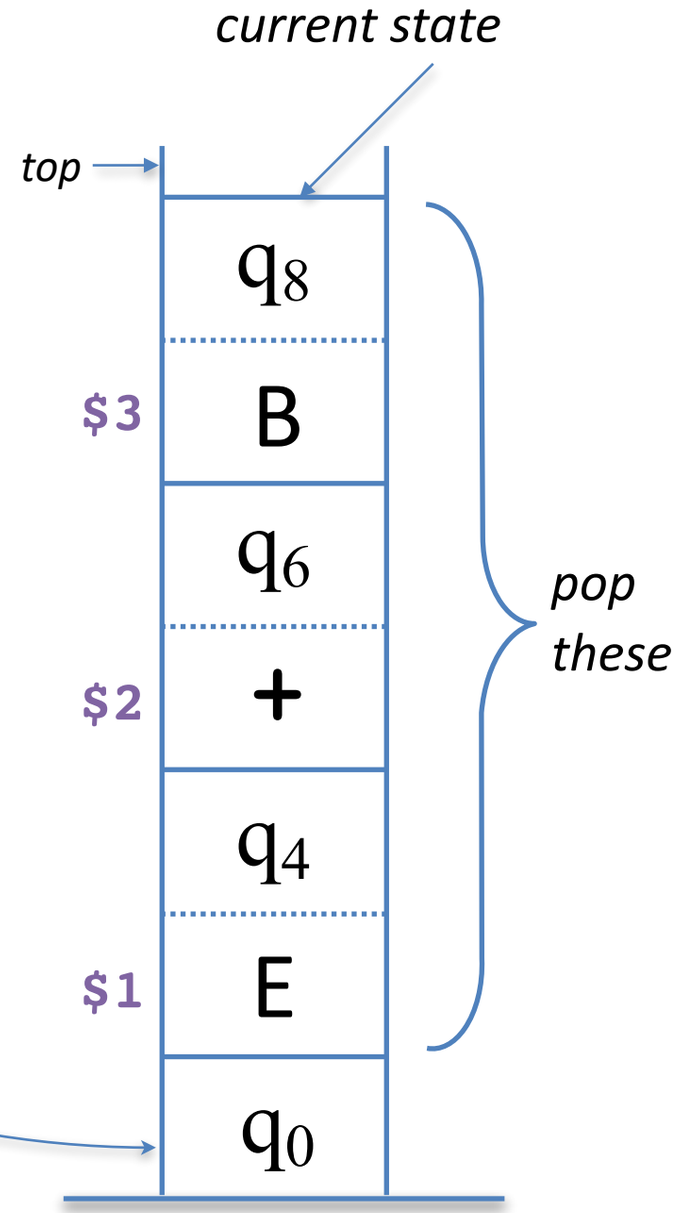
State	ACTION					GOTO	
	*	+	0	1	$	E	T
q_8	r2	r2			r2		

(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

```
{
   $$ = new Node("E");
   $$.addChild($1);
   $$.addChild($2);
   $$.addChild($3);

   pop(6);

   next = GOTO[stack[top-1], "E"];
   push($$);
   push(next);
}
```



Building an AST

- More useful representation of syntax tree
  - Less clutter
  - Actual level of detail depends on your design
- Basis for semantic analysis
- Later annotated with various information
  - Type information
  - Computed values
- Technically – a class hierarchy of abstract syntax tree nodes

Building the **Abstract** Syntax Tree

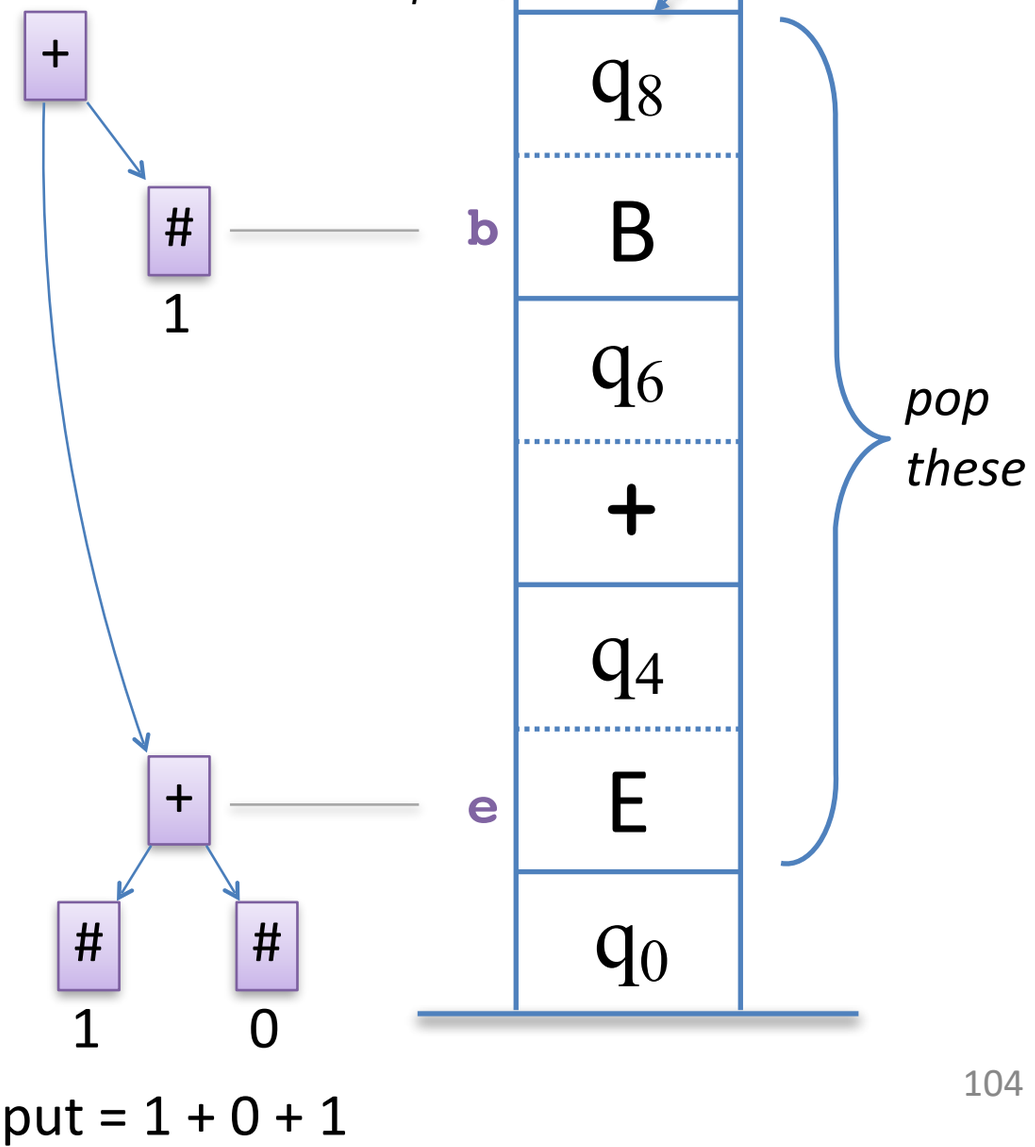
- Generally — just “skip over” the creation of some internal nodes and you get an AST

```
E ::= E:e PLUS B:b {:
   RESULT = new Node(“+”);
   RESULT.addChild(e);
   RESULT.addChild(b);
:}

   B:b {:
   RESULT = b;
:}

B ::= 0 {:
   RESULT = new Node(“#”);
   RESULT.value = 0;
:}
```

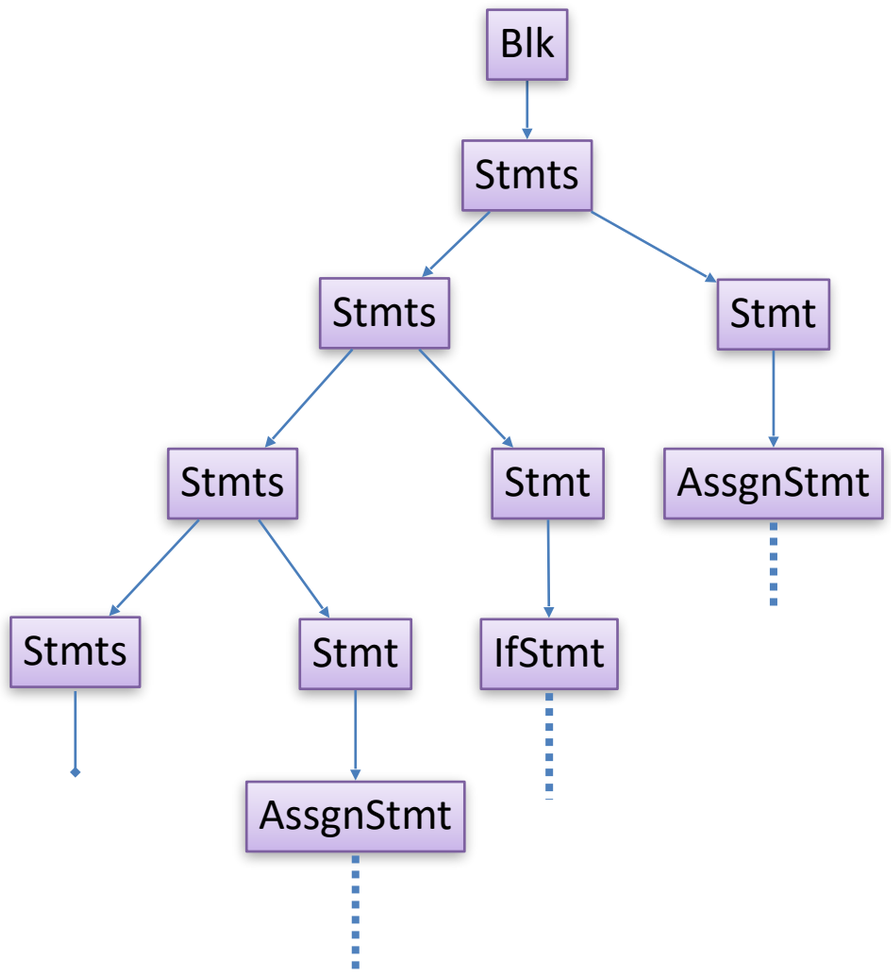
(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1



Building the **Abstract** Syntax Tree

- There is no "The AST" for a grammar
- However, clearly the parse tree is full of artifacts of the grammar we don't care about

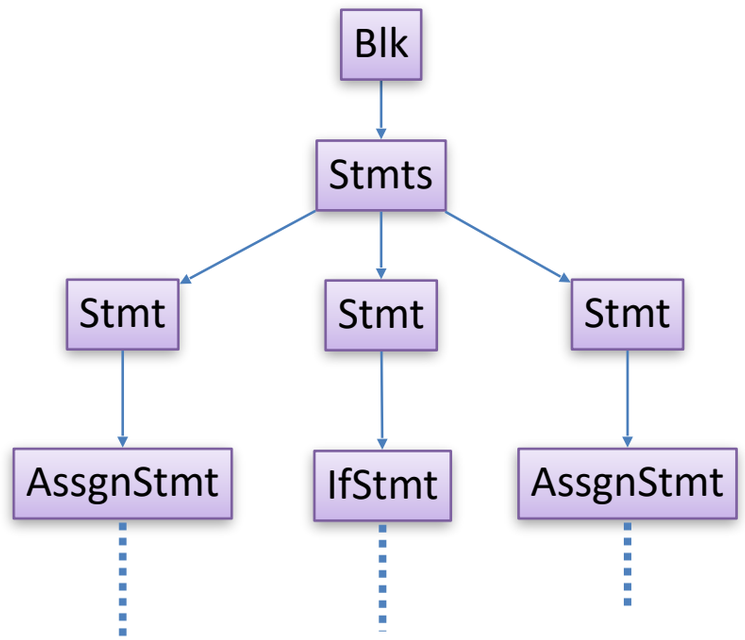
(0) Blk → Stmts
(1) Stmts → ε
(2) Stmts → Stmts Stmt
(3) Stmt → IfStmt
(4) Stmt → AssgnStmt
⋮



Building the **Abstract** Syntax Tree

- This is nicer:

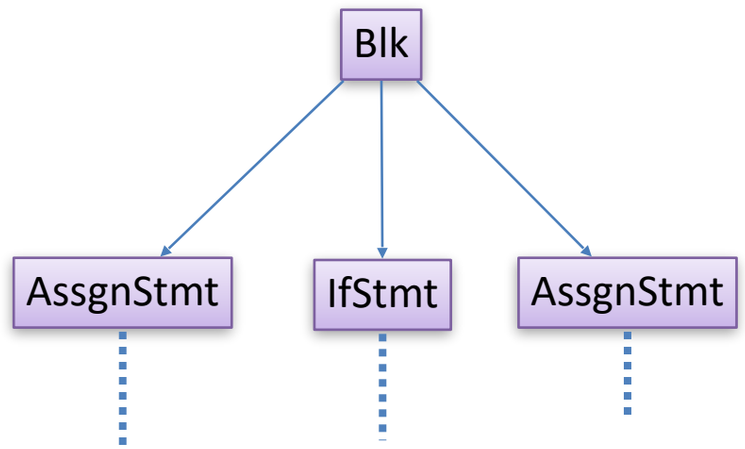
(0) Blk → Stmts
(1) Stmts → ε
(2) Stmts → Stmts Stmt
(3) Stmt → IfStmt
(4) Stmt → AssgnStmt
⋮



Building the **Abstract** Syntax Tree

- Even nicer:

(0) Blk → Stmts
(1) Stmts → ε
(2) Stmts → Stmts Stmt
(3) Stmt → IfStmt
(4) Stmt → AssgnStmt
⋮



Building the **Abstract** Syntax Tree

- We use the parser actions to skip over and merge nodes as we go

```
Blk → Stmts:ss  {
  RESULT = new Block();
  RESULT.statements = s.statements;
}
Stmts →ε  {
  RESULT = new StatementList();
}
Stmts → Stmts:ss Stmt:s  {
  RESULT = ss;
  RESULT.statements.push(s);
}
Stmt → IfStmt:ifs  {
  RESULT = ifs;
;
}
```

(0) Blk → Stmts
(1) Stmts → ε
(2) Stmts → Stmts Stmt
(3) Stmt → IfStmt
(4) Stmt → AssgnStmt
⋮

```
class Block {
public:
 vector<Statement> statements;
}

class StatementList{
public:
 vector<Statement> statements;
}

class Statement;

class IfStatement: public Statement {
…
}
```

Error Handling

Error Handling

MCD
S3.2

It has been suggested that with today's fast interactive systems, there is no point in continuing program processing after the first error has been detected, since the user can easily correct the error and then recompile in less time than it would take to read the next error message. But users like to have some idea of how many syntax errors there are left in their program; recompiling several times, each time expecting it to be the last time, is demoralizing. We therefore like to continue the parsing and give as many error messages as there are syntax errors. This means that we have to do error recovery.

Error Handling in LR Parsers

MCD
S3.5.10

Recovery **without** modifying the stack

- The **acceptable-set method**
  - Step 1: construct a set A of *acceptable* terminals based on the current state of the parser.
  - Step 2: discard input tokens until a token $t_A \in A$ is encountered.
  - Step 3: advance the parser to the next state in which it can consume t_A.

A = Correct tokens(top state)

don't need to do anything

Not practical: throws away important tokens and produce bad results

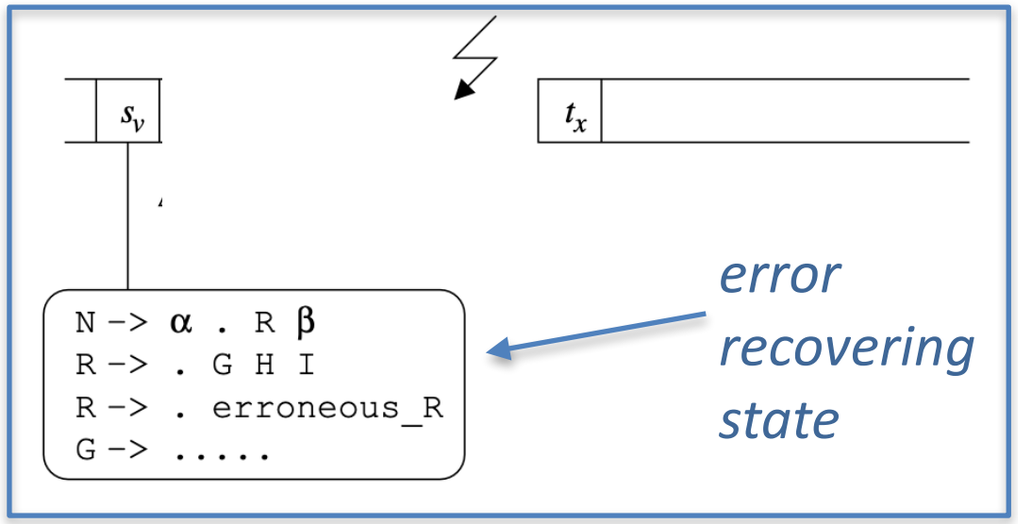
panic mode

Error Handling in LR Parsers

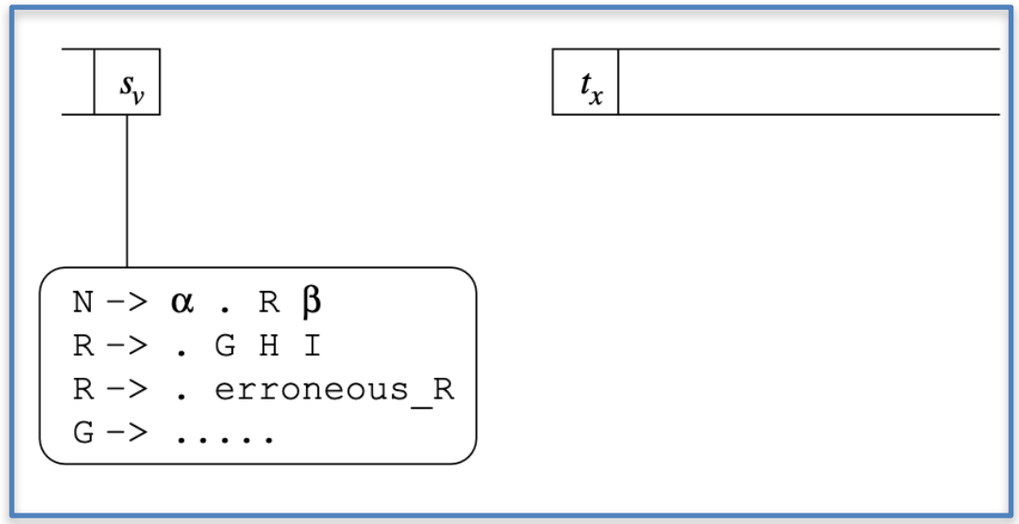
MCD
S3.5.10

Recovery **with** stack modification

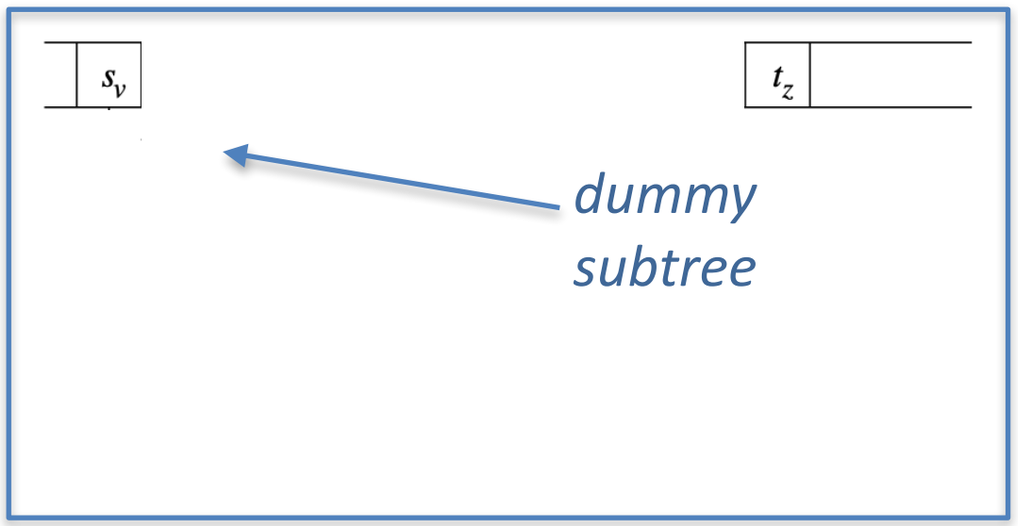
- **Error-recovering non-terminals**
  - Step 1: Define some non-terminal as error-recovering
    - "big names", e.g., declarations, statements
    - **R** → **G H I** | **errorneous_R**
    - Any state containing an item N → $\alpha \bullet R \beta$ is an error recovering state
  - Step 2: When an error occurs, given up on that non-terminal
    - Pop stack until finding an (error recovering) state with N → $\alpha \bullet R \beta$
    - Drop tokens until finding one that "ends" R



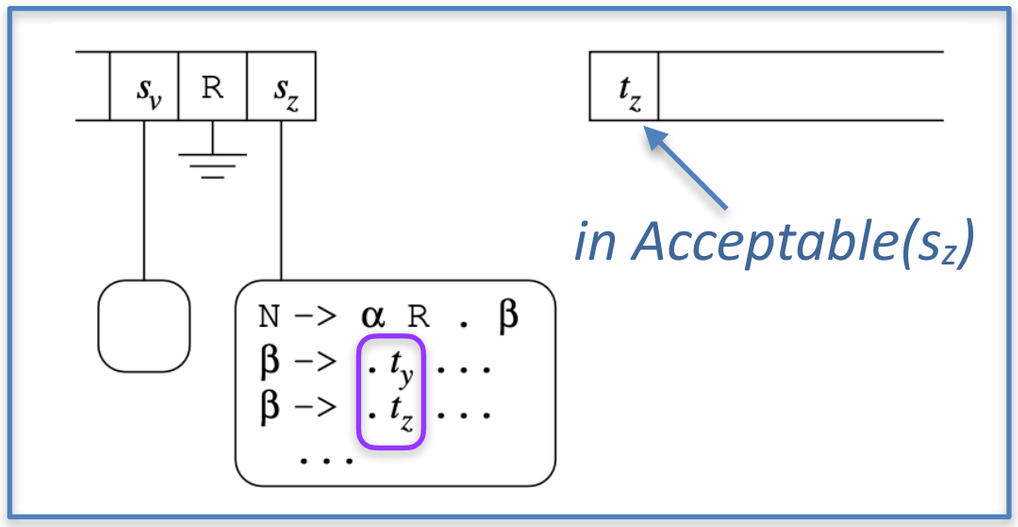
1. Detecting an error



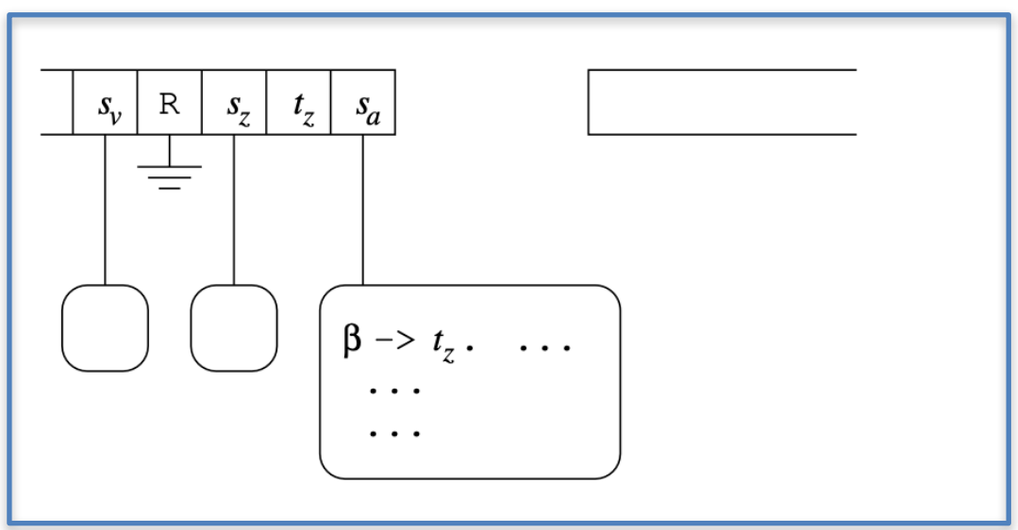
2. Popping to the latest R's error recovery state



3. Repairing the stack



4. Repairing the input



4. Restarting the parser

✓ Cannot loop

✓ Restricts damage to parse tree to a known place

✗ Though discards corrctly parsed subtrees

✓ Reasonable chance to successfully continue

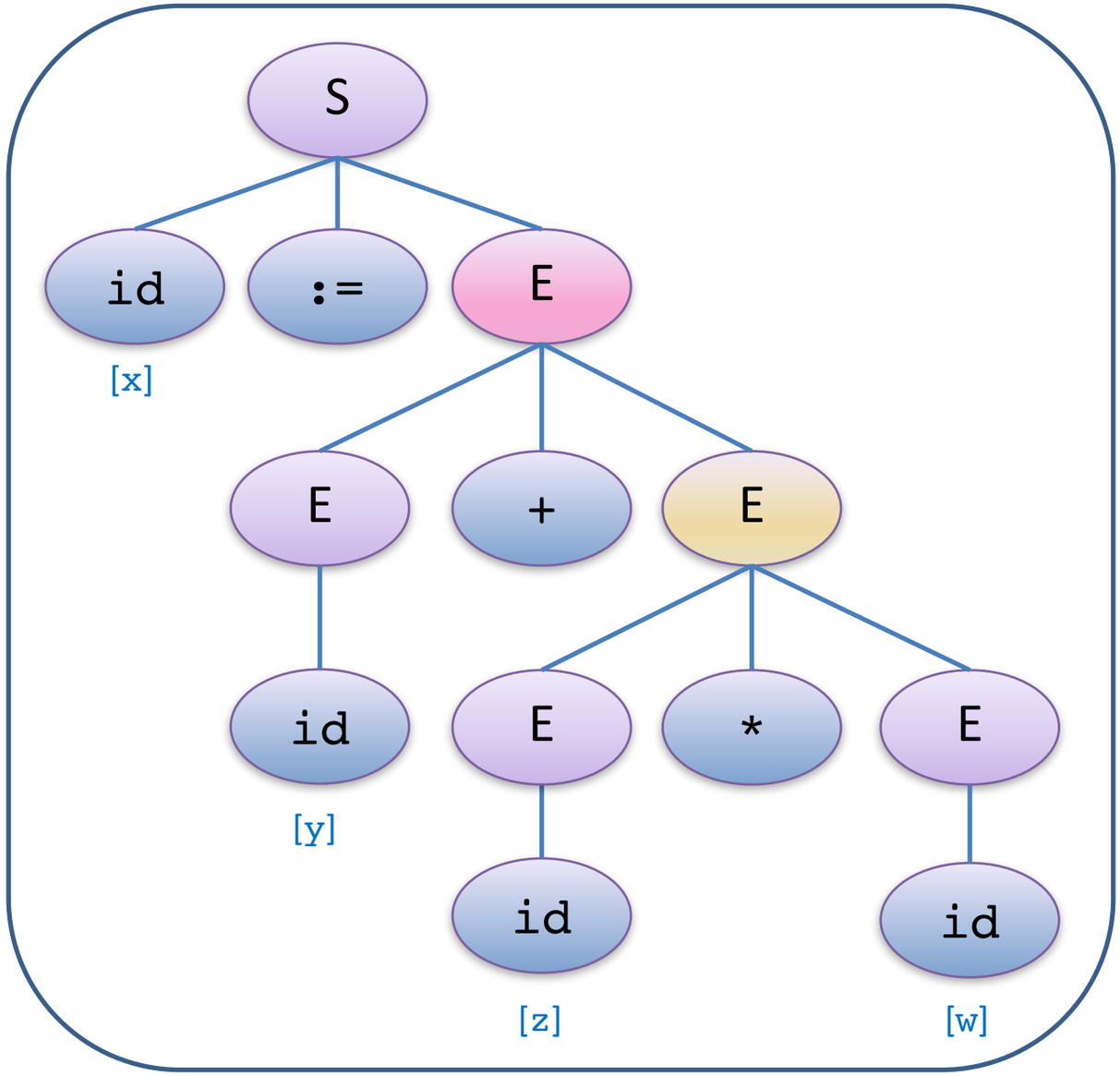
- Future phases must deal with dummy trees
- Must not generate code

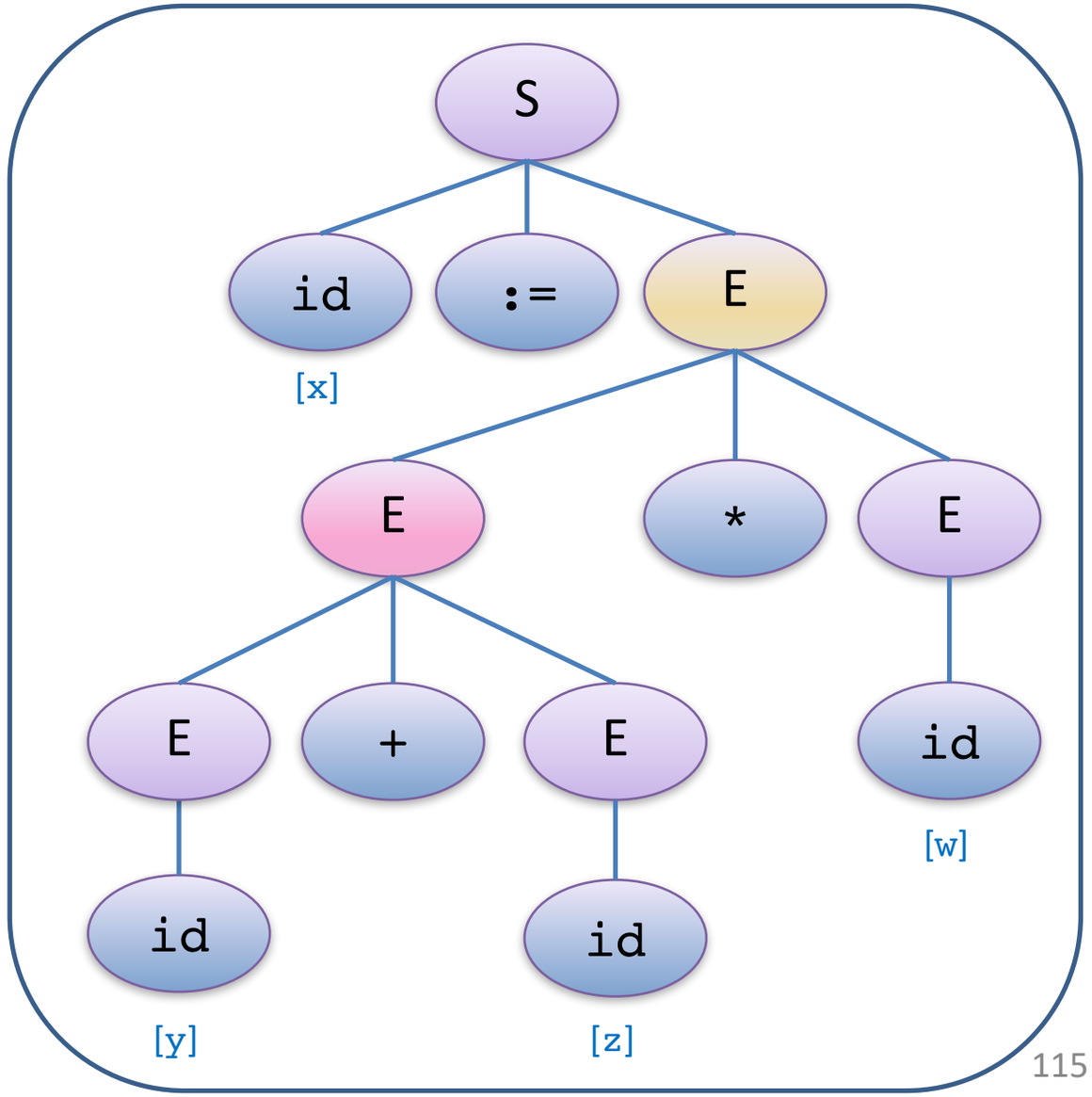
Ad Hock Treatment of Ambiguity

Ambiguity

```
x := y + z * w
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```





Precedence

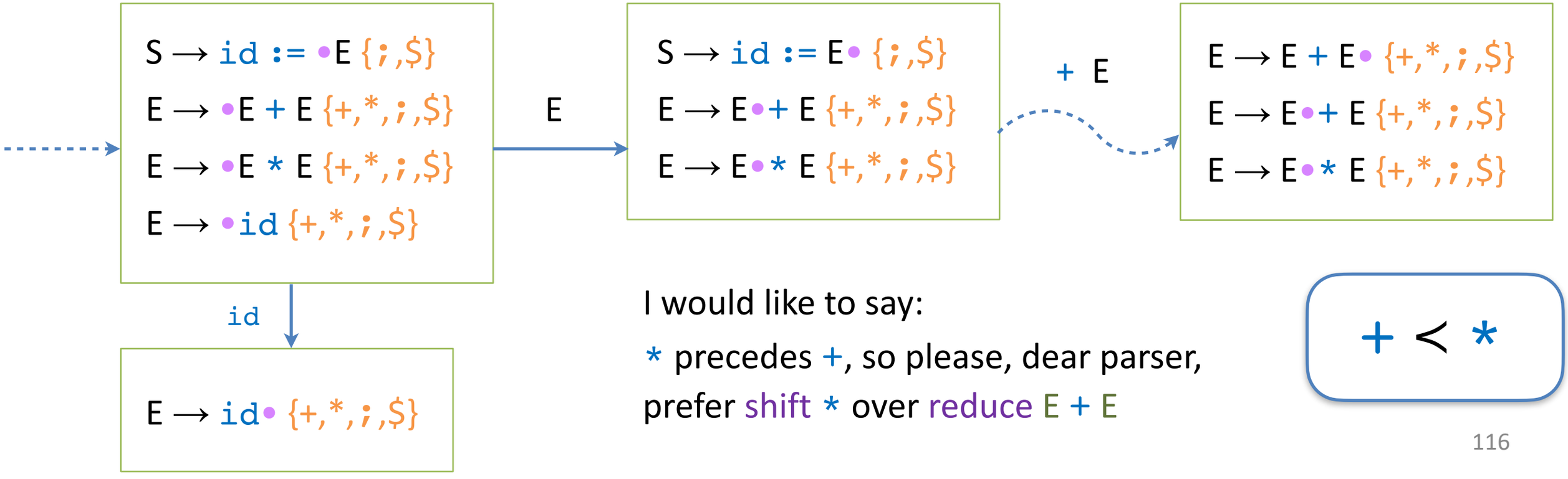
~~Ambiguity~~

```
x := y + z * w
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```

S → id := E → id := E + E → id := E + E * E → id := E + E * id → id := E + id * id → id := id + id * id

id := id + id * id ➡ id := E + id * id ➡ id := E + E * id ➡ id := E + E * E ➡ id := E + E ➡ id := E ➡ S



I would like to say:

* precedes +, so please, dear parser,

prefer shift * over reduce E + E

+ < *

Associativity

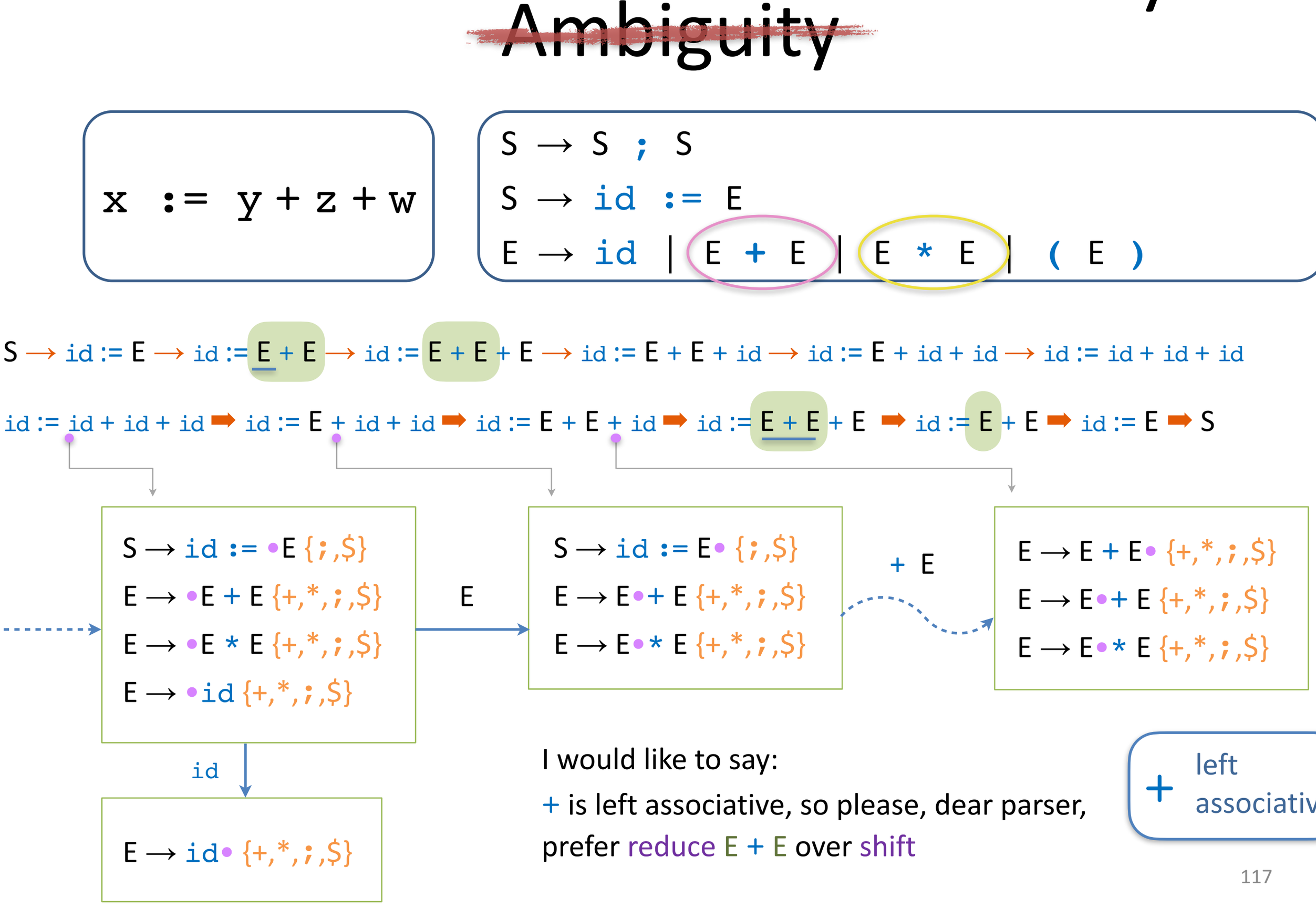
~~Ambiguity~~

```
x := y + z + w
```

```
S → S ; S
S → id := E
E → id | E + E | E * E | ( E )
```

S → id := E → id := E + E → id := E + E + E → id := E + E + id → id := E + id + id → id := id + id + id

id := id + id + id ➡ id := E + id + id ➡ id := E + E + id ➡ id := E + E + E ➡ id := E + E ➡ id := E ➡ S



I would like to say:

+ is left associative, so please, dear parser,

prefer reduce E + E over shift

\+ left associative

"Automatically" Resolving Conflicts in LR Parser

- **Shift-reduce conflicts**
  - Highest (user-specified) precedence wins:
    - $P \longrightarrow \alpha \bullet \circledast \beta \{ \ldots. \}$
    - $Q \longrightarrow \mu \bowtie R \bullet \{ .. \circledast .. \}$ or $Q \longrightarrow \mu \bowtie \bullet \{ .. \circledast .. \}$

$\bowtie < \circledast$

$\circledast < \bowtie$

  - (User-specified) associativity wins:
    - $P \longrightarrow \alpha \bullet \circledast \beta \{ \ldots. \}$
    - $Q \longrightarrow \mu \circledast R \bullet \{ .. \circledast .. \}$ or $Q \longrightarrow \mu \circledast \bullet \{ .. \circledast .. \}$

$\circledast$ right associative

$\circledast$ left associative

  - Default: use shift

a bit more general: allow for different terminals with equal precedence

“Automatically” Resolving Conflicts in LR Parser

- **Shift-reduce conflicts**
  - Highest (user-specified) precedence win:
    - $P \longrightarrow \alpha \bullet \circledast \beta$ { …. }
    - $Q \longrightarrow \mu \bowtie R \bullet$ { .. $\circledast$ ..} or $Q \longrightarrow \mu \bowtie \bullet$ { .. $\circledast$ ..}

$\bowtie < \circledast$

$\circledast < \bowtie$

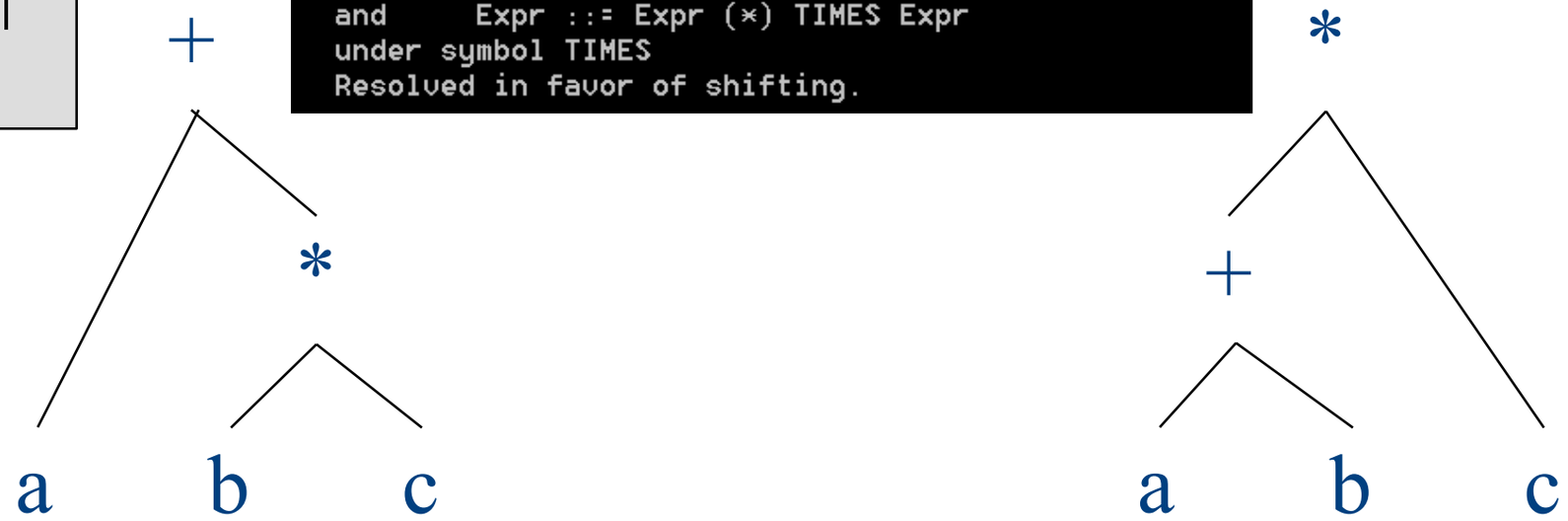
  - Solves if/else ambiguity
    - if_S $\longrightarrow$ `if` (BoolExpr) S $\cdot$ { … `else` …}
    - if_S $\longrightarrow$ `if` (BoolExpr) S $\cdot$ `else` S { … `else` …}

- **Reduce-reduce conflicts**
  - First rule wins

Ambiguities as conflicts in CUP

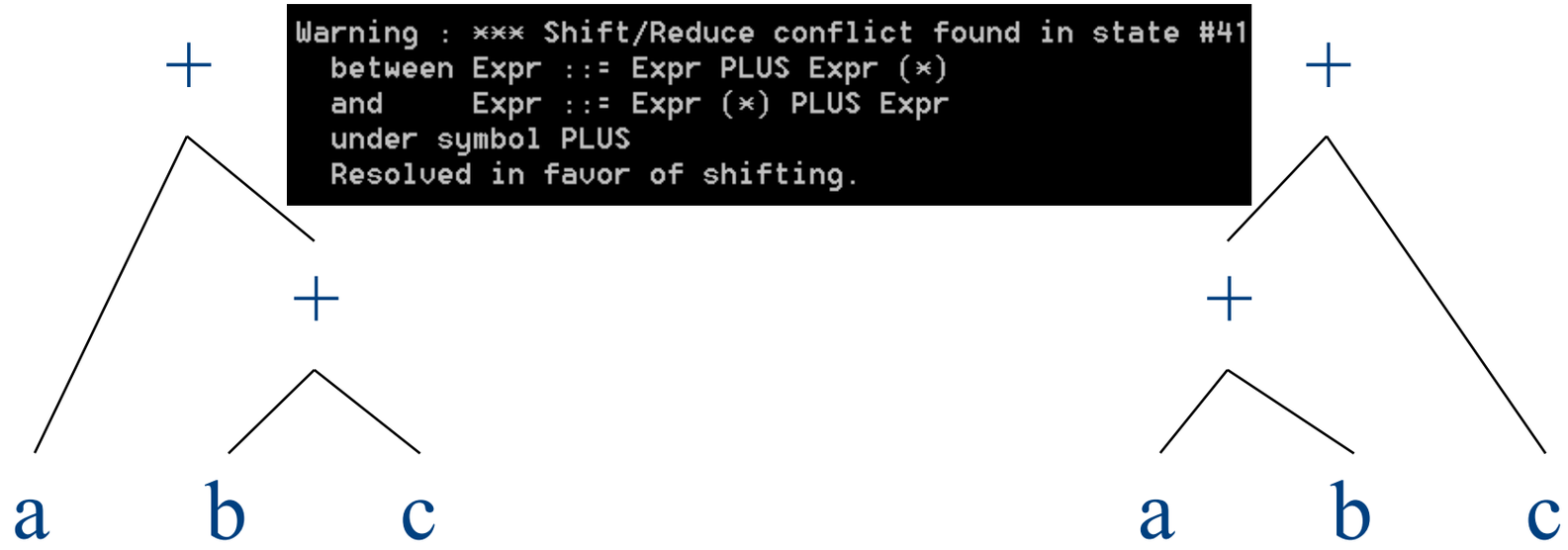
```
E → id |
    E + E |
    E * E
```

```
Warning : *** Shift/Reduce conflict found in state #41
  between Expr ::= Expr PLUS Expr (*)
  and     Expr ::= Expr (*) TIMES Expr
  under symbol TIMES
  Resolved in favor of shifting.
```



a + b * c

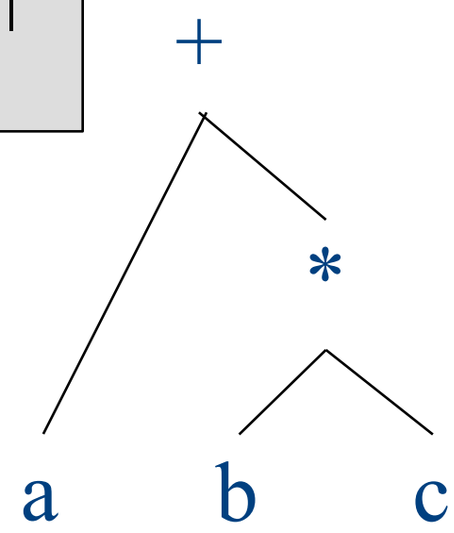
```
Warning : *** Shift/Reduce conflict found in state #41
  between Expr ::= Expr PLUS Expr (*)
  and     Expr ::= Expr (*) PLUS Expr
  under symbol PLUS
  Resolved in favor of shifting.
```



a + b + c

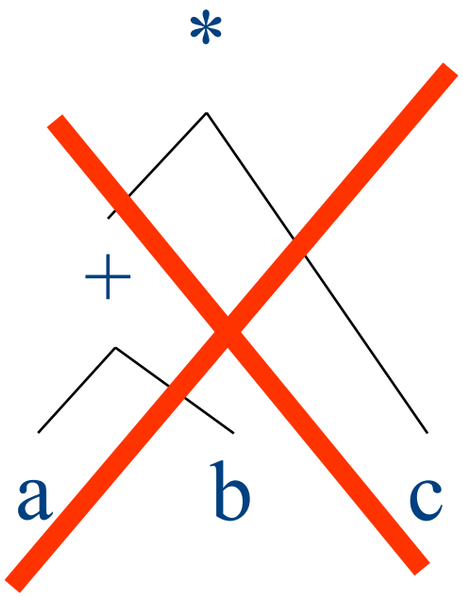
Ambiguities as conflicts in CUP

```
E → id |
    E + E |
    E * E
```

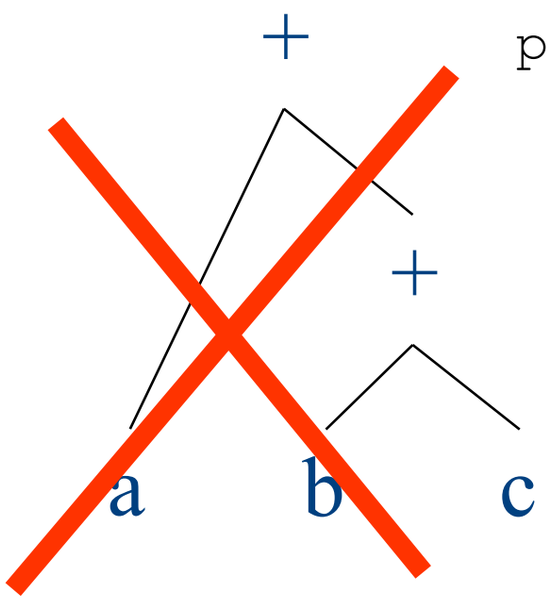



```
precedence left PLUS
precedence left MULT
```

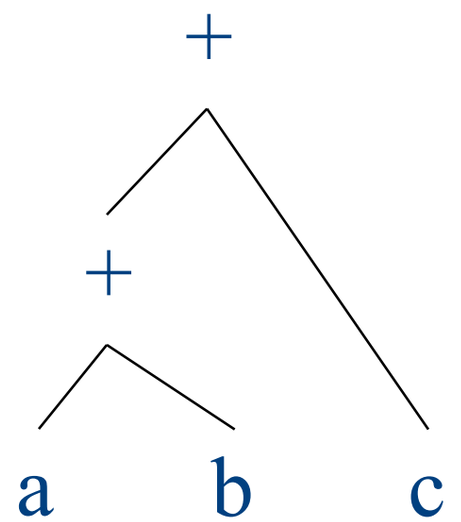
Increasing precedence



a + b * c




```
precedence left PLUS
```



a + b + c

Automated Parser Generation (e.g., via CUP)

Will be discussed more thoroughly in the recitation