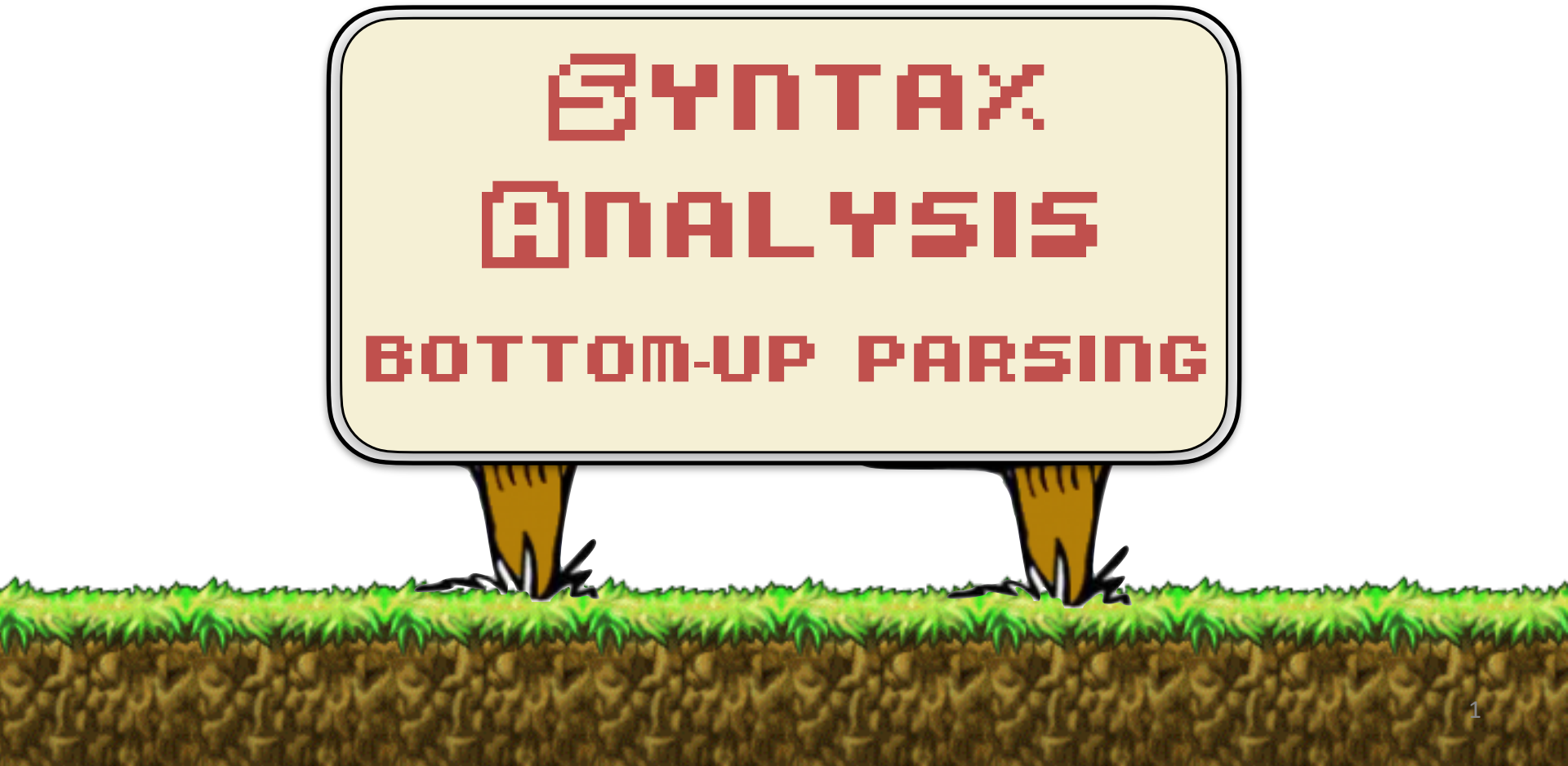
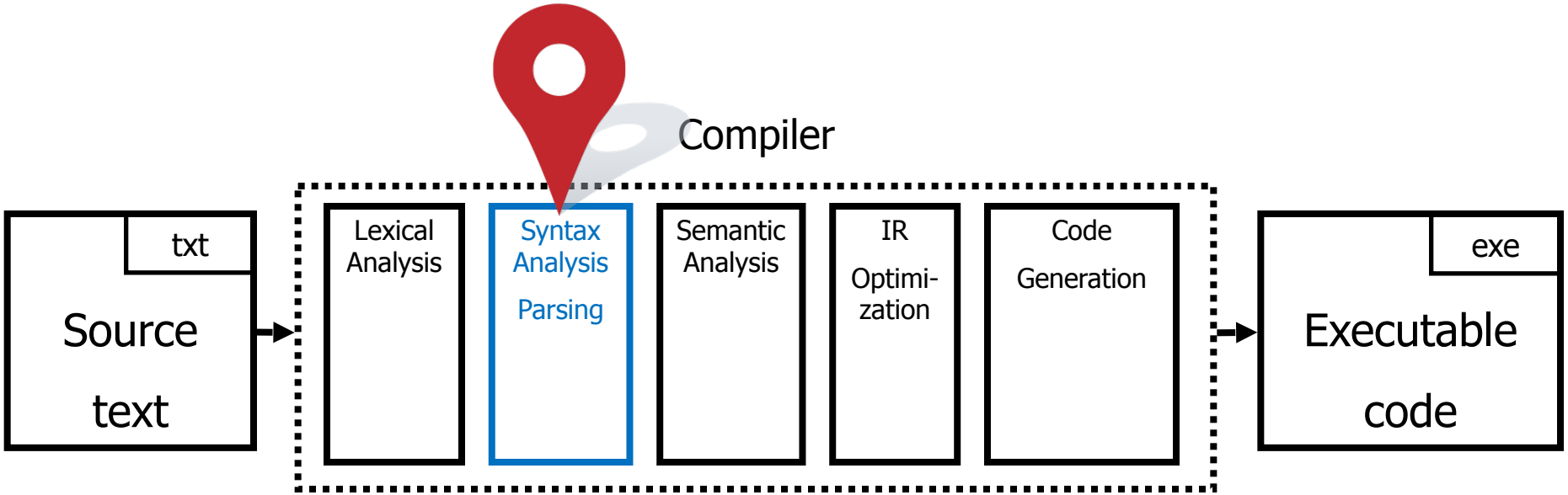
THEORY OF COMPILATION

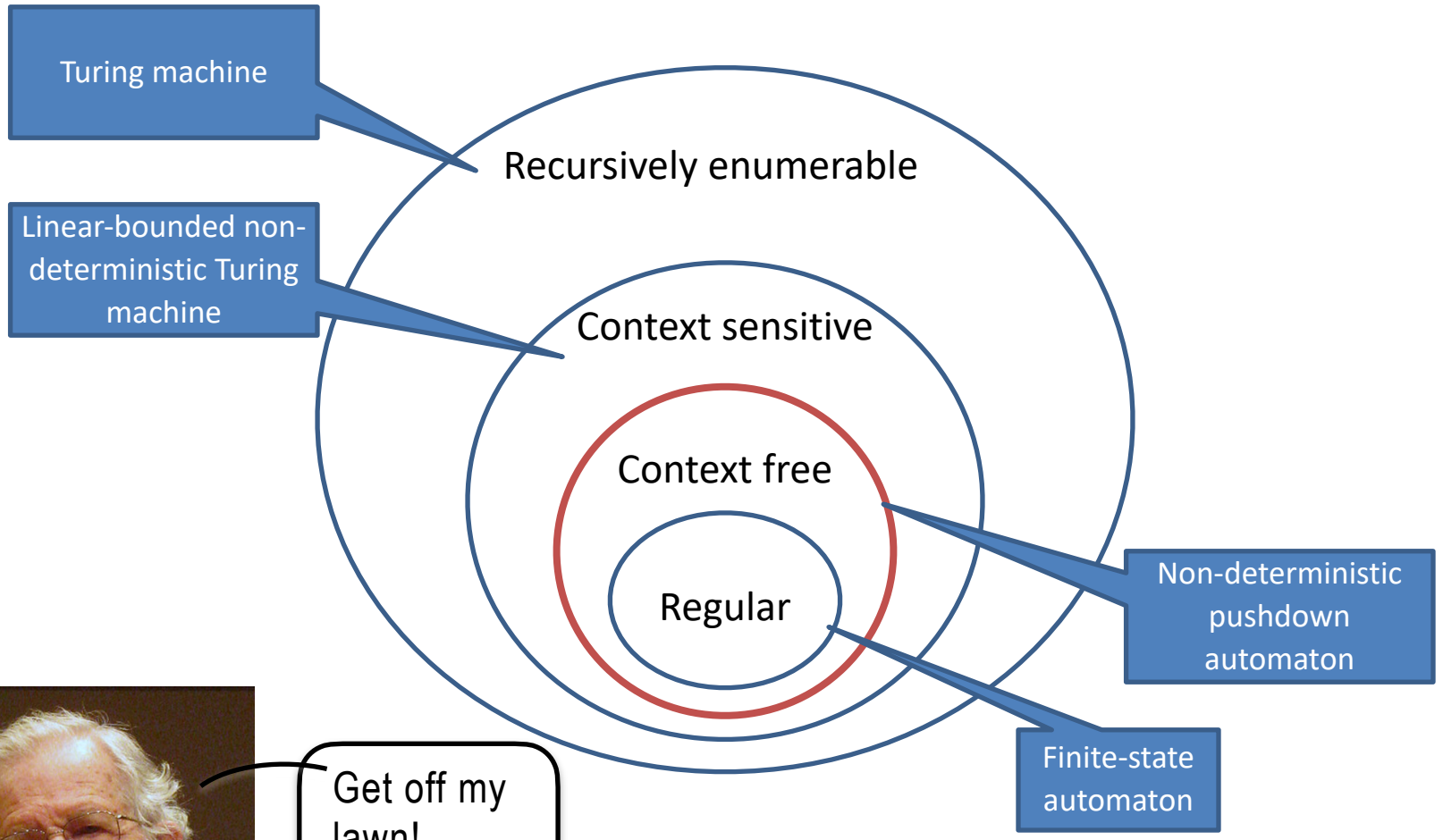
LECTURE 05



You are here


Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code
Generation
exe
Executable code

Parsing!


Turing machine
Recursively enumerable
Linear-bounded non-deterministic Turing machine
Context sensitive
Context free
Regular
Non-deterministic pushdown automaton
Finite-state automaton

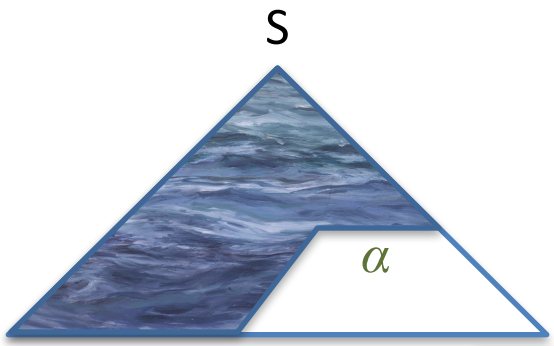

Get off my lawn!
Damn kids...

Last Time

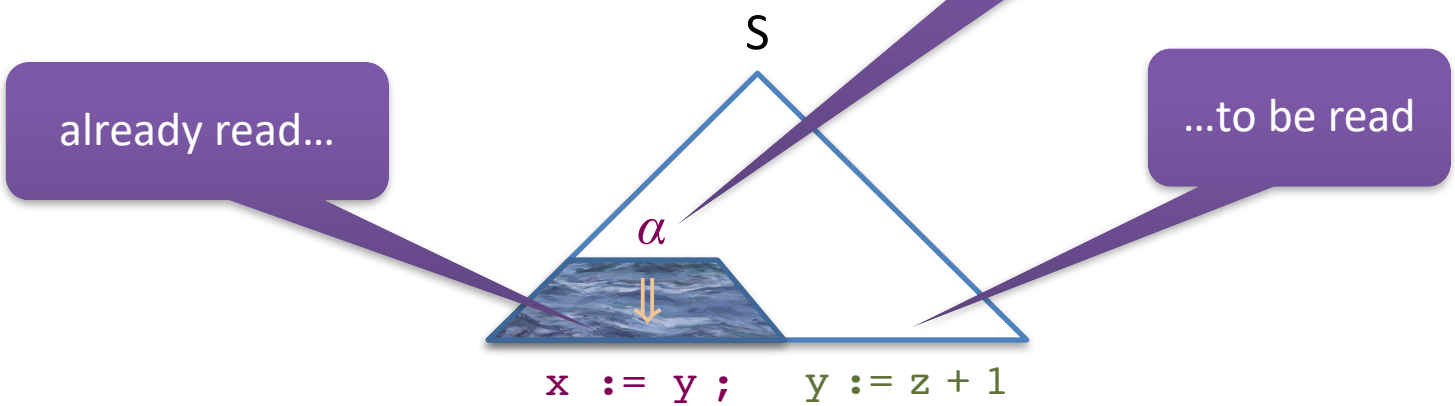
- Parsing
  - Top-down or bottom-up
- Top-down parsing
  - Recursive descent
  - LL(k) grammars
  - LL(k) parsing with pushdown automata
- LL(k) parsers
  - Cannot deal with common prefixes and left recursion
  - Left-recursion removal might result in complicated grammar

Efficient Parsers

- Top-down (predictive)



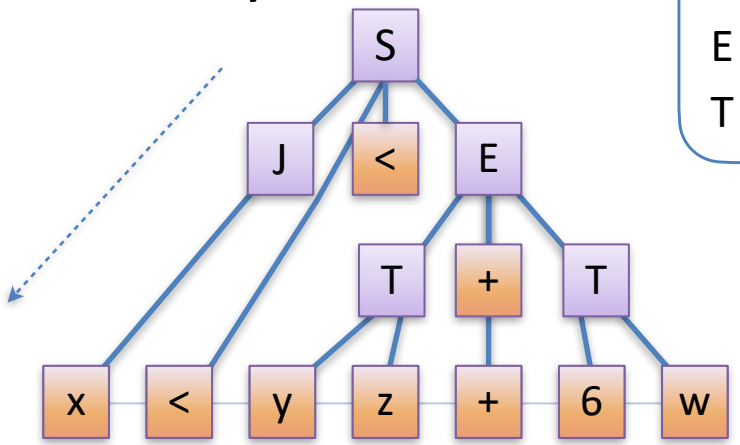
- Bottom-up (shift-reduce)



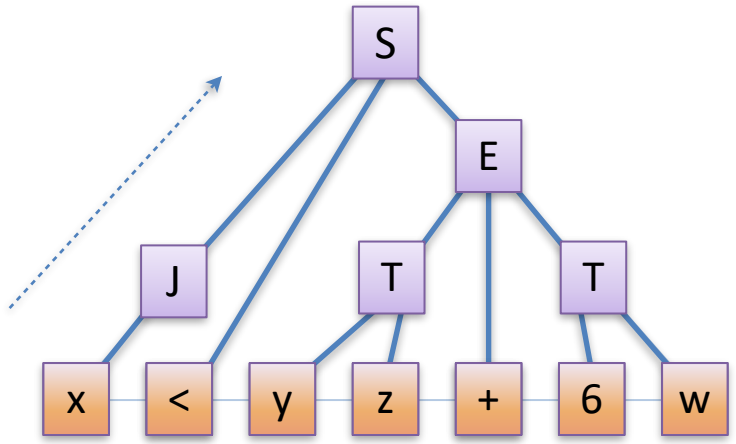
Parser Classes – Illustrated

- Top-down (predictive)

$S \rightarrow J < E$
$J \rightarrow ID$
$E \rightarrow T + T$
$T \rightarrow ID\ ID \mid NUM\ ID$



- Bottom-up (shift-reduce)



Example: a Simple Grammar

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

$$B \rightarrow 0 \mid 1$$

- Let us number the rules:

  (1) $E \rightarrow E * B$

  (2) $E \rightarrow E + B$

  (3) $E \rightarrow B$

  (4) $B \rightarrow 0$

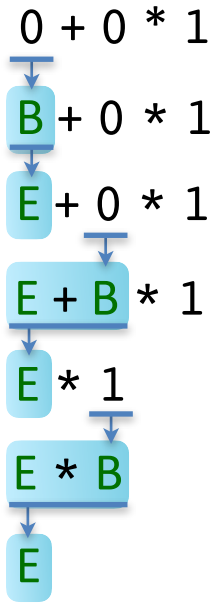
  (5) $B \rightarrow 1$

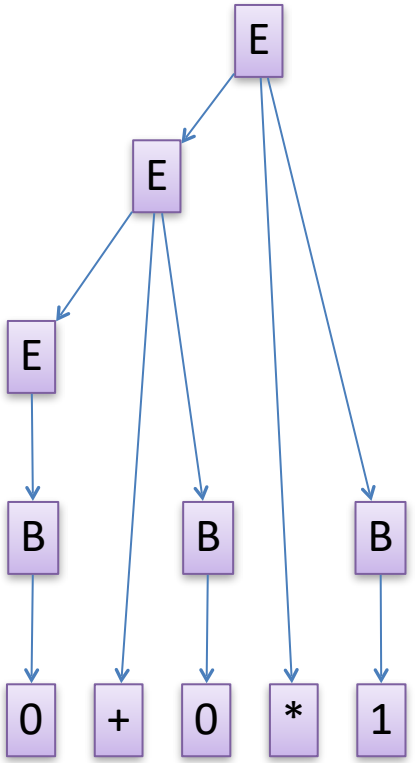
Goal: Reduce the Input to the Start Symbol

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

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

Rightmost derivation





Scan input left to right. Upon seeing a right-hand side of a rule, apply the rule "in reverse": replace the **right-hand side** with the **left-hand side** (which is a single non-terminal)

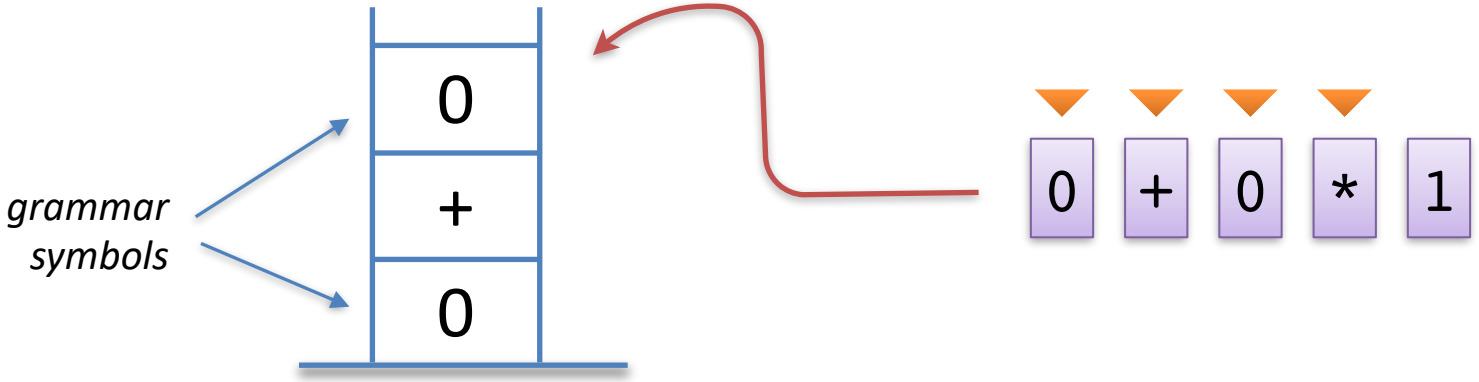
Shift & Reduce

$E \xrightarrow{(1)} E * B \mid \underset{(2)}{E + B} \mid \underset{(3)}{B}$

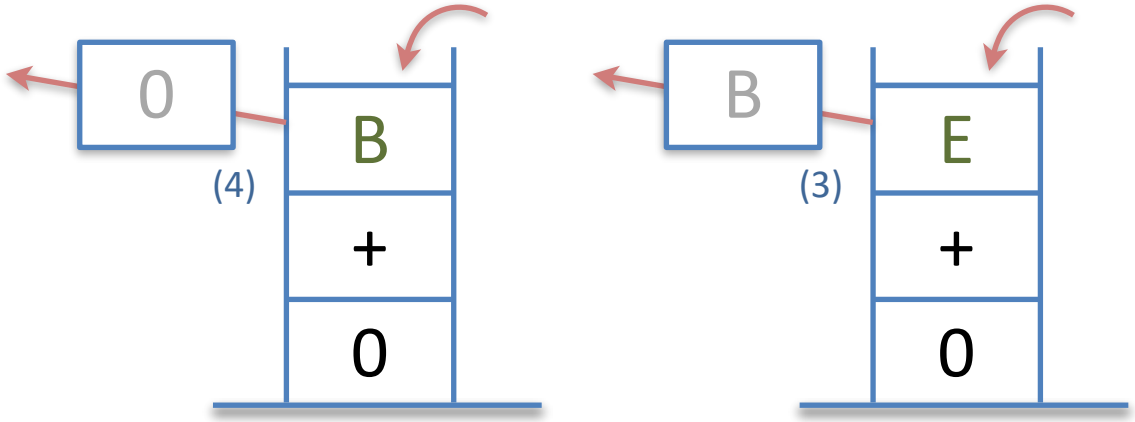
$B \xrightarrow{(4)} 0 \mid \underset{(5)}{1}$

In each step, we either shift a symbol from the input to the stack, or reduce according to one of the rules.

Shift



Reduce



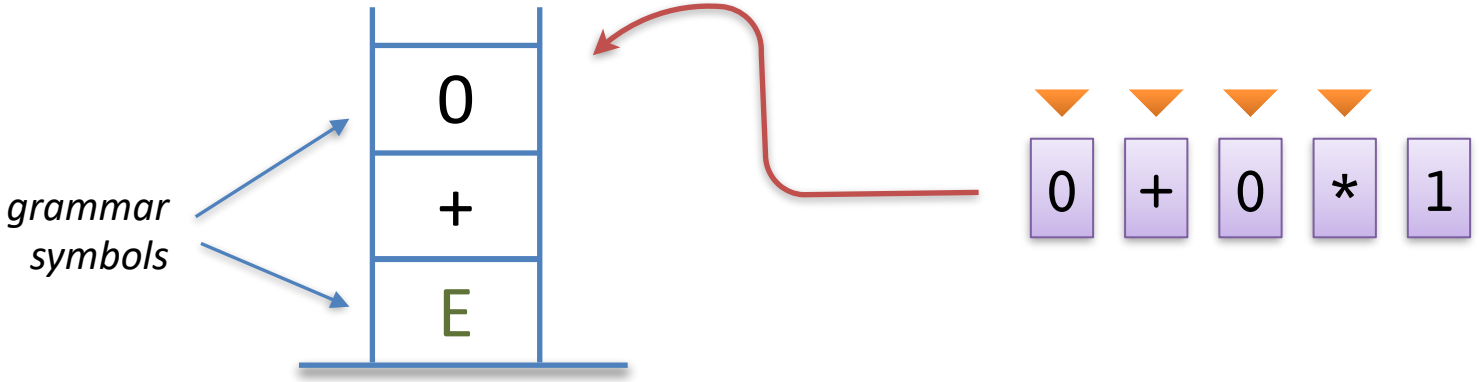
Shift & Reduce

$E \xrightarrow{(1)} E * B \mid \underset{(2)}{E + B} \mid \underset{(3)}{B}$

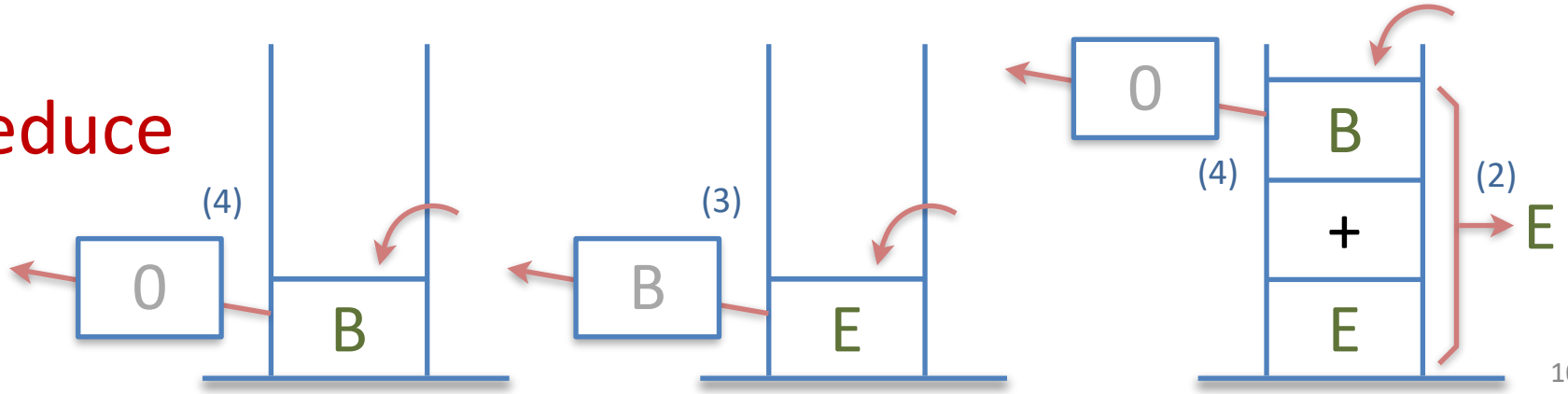
$B \xrightarrow{(4)} 0 \mid \underset{(5)}{1}$

In each step, we either shift a symbol from the input to the stack, or reduce according to one of the rules.

Shift



Reduce



Shift / Reduce Parser — Intuition

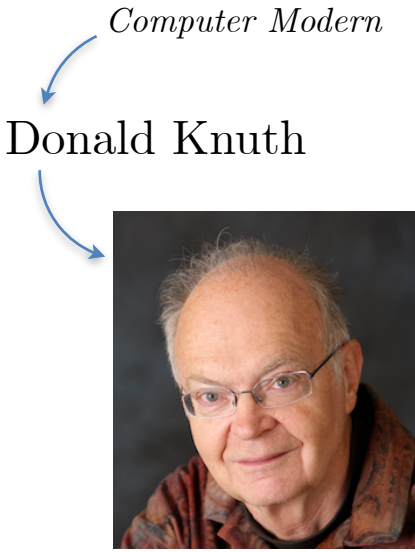
- Gather input token by token
  - until we find a right-hand side of some rule
  - then, replace it with the non-terminal on the left hand side
- Shift – consume token and push on stack
  - Each shift moves to a state that records what we've seen so far
- Reduce – replace a string α on the stack with a nonterminal N that derives it ($N \rightarrow \alpha$)

How does the parser know what to do?

- Pushdown Automaton!
  - A ***state*** will keep the info gathered so far
  - A ***table*** will tell it "what to do" based on current state and next token
  - Some info will be kept in a ***stack***

LR(*k*) Grammars

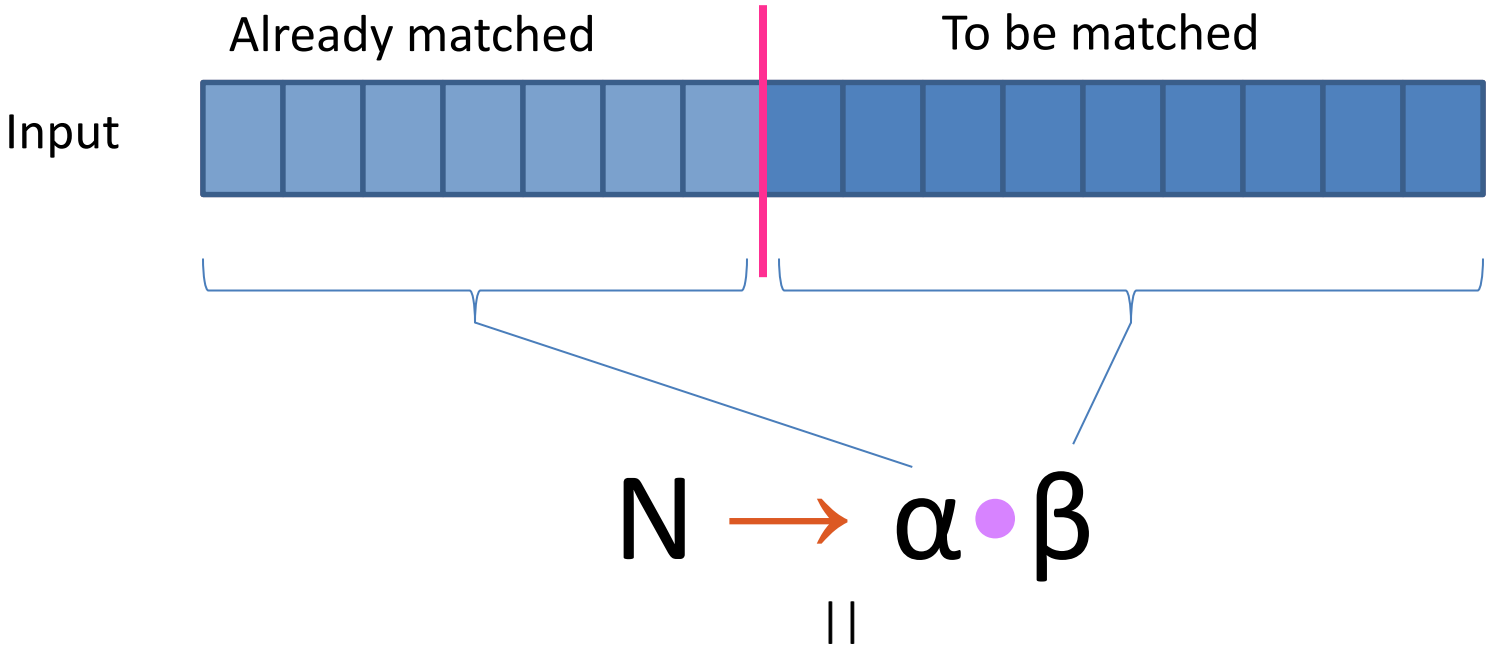
- A grammar is in the class LR(*k*) when it can be derived via:
  - Bottom-up analysis
  - Scanning the input from left to right (L)
  - Producing the rightmost derivation (R)
  - With lookahead of k tokens (*k*)



- A language is said to be LR(*k*) if it has an LR(*k*) grammar
- The simplest case is LR(0), which we discuss next

LR(0) Item

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



||

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

LR(0) Item

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

$B \rightarrow 0 \mid 1$

$E \rightarrow E \bullet * B$ Shift Item

$E \rightarrow E * B \bullet$ Reduce Item

Example: Parsing with LR(0) Items

end-of-input marker

$Z \rightarrow expr\ \$$

$expr \rightarrow term \mid expr + term$

$term \rightarrow ID \mid (expr)$

$Z \rightarrow E\ \$$

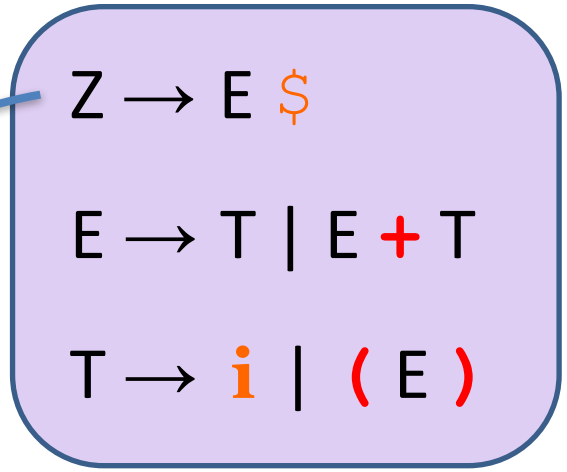
$E \rightarrow T \mid E + T$

$T \rightarrow i \mid (E)$

(just a shorthand of the grammar on top)

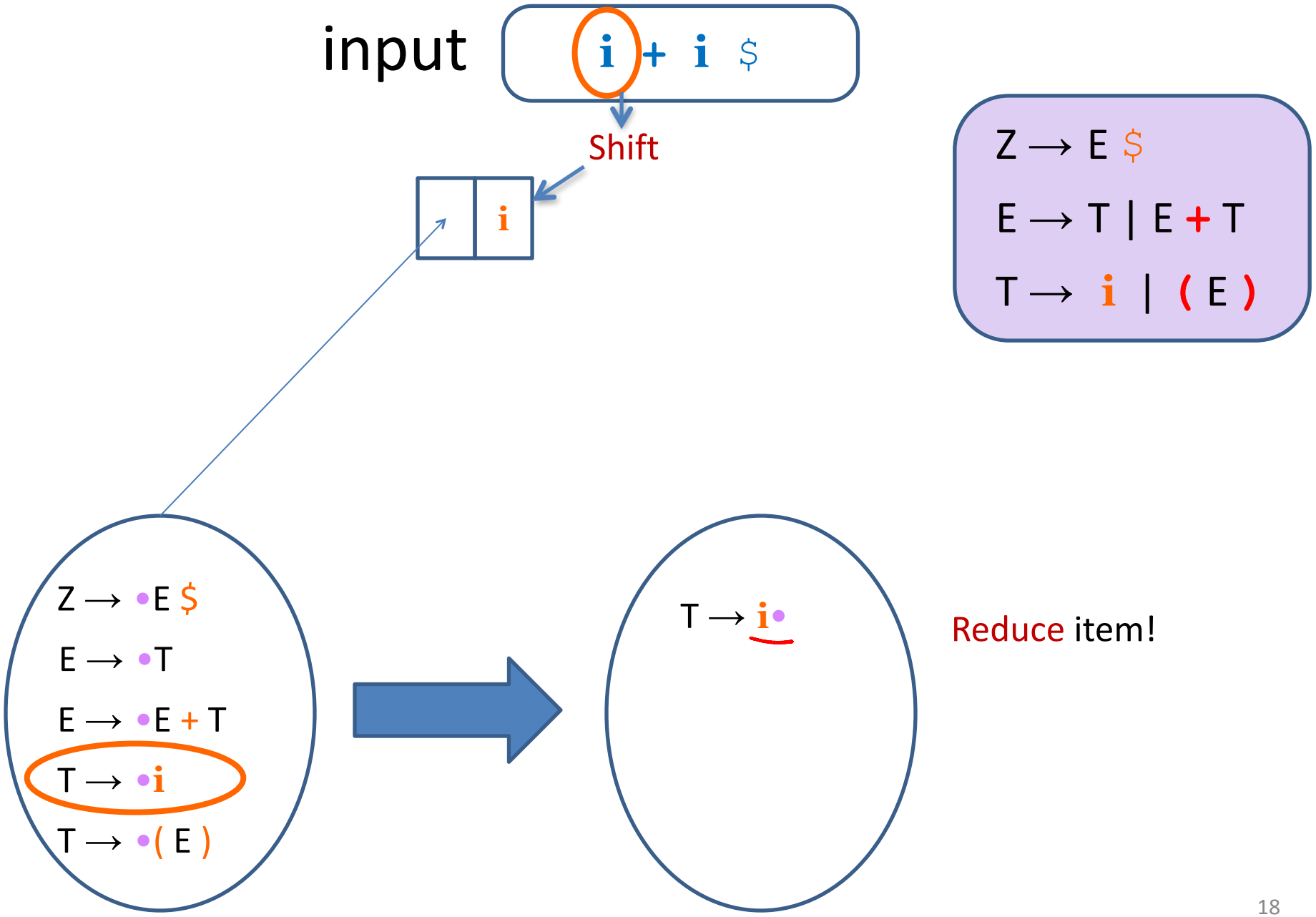
Example: Parsing with LR(0) Items

- LR(0) state =
  set of LR(0) items



Z → •E $
E → •T
E → •E + T
T → •i
T → •(E)

Initial state

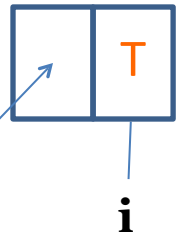

input
i + i $
Shift
i
Z → E $
E → T | E + T
T → i | (E)
Z → •E $
E → •T
E → •E + T
T → •i
T → •(E)
T → i•
Reduce item!

input **i + i $**

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

$T \rightarrow i \mid (\ E\ )$



$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

$E \rightarrow \bullet E + T$

$T \rightarrow \bullet i$

$T \rightarrow \bullet (\ E\ )$

$T \rightarrow i \bullet$

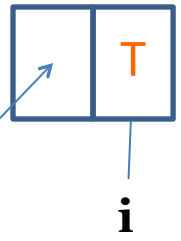
Reduce item!

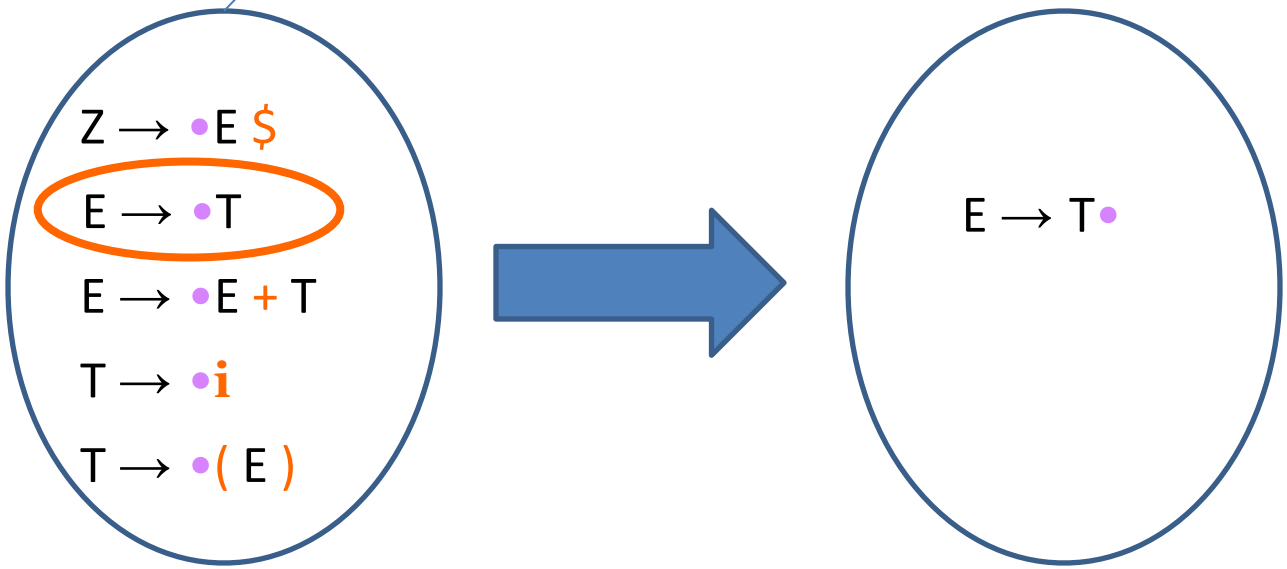
input **i + i $**

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

$T \rightarrow i \mid (E)$





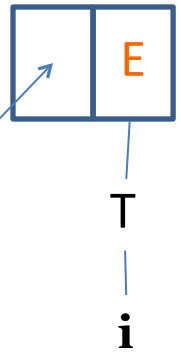
Reduce item!

input **i + i** $

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

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



$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

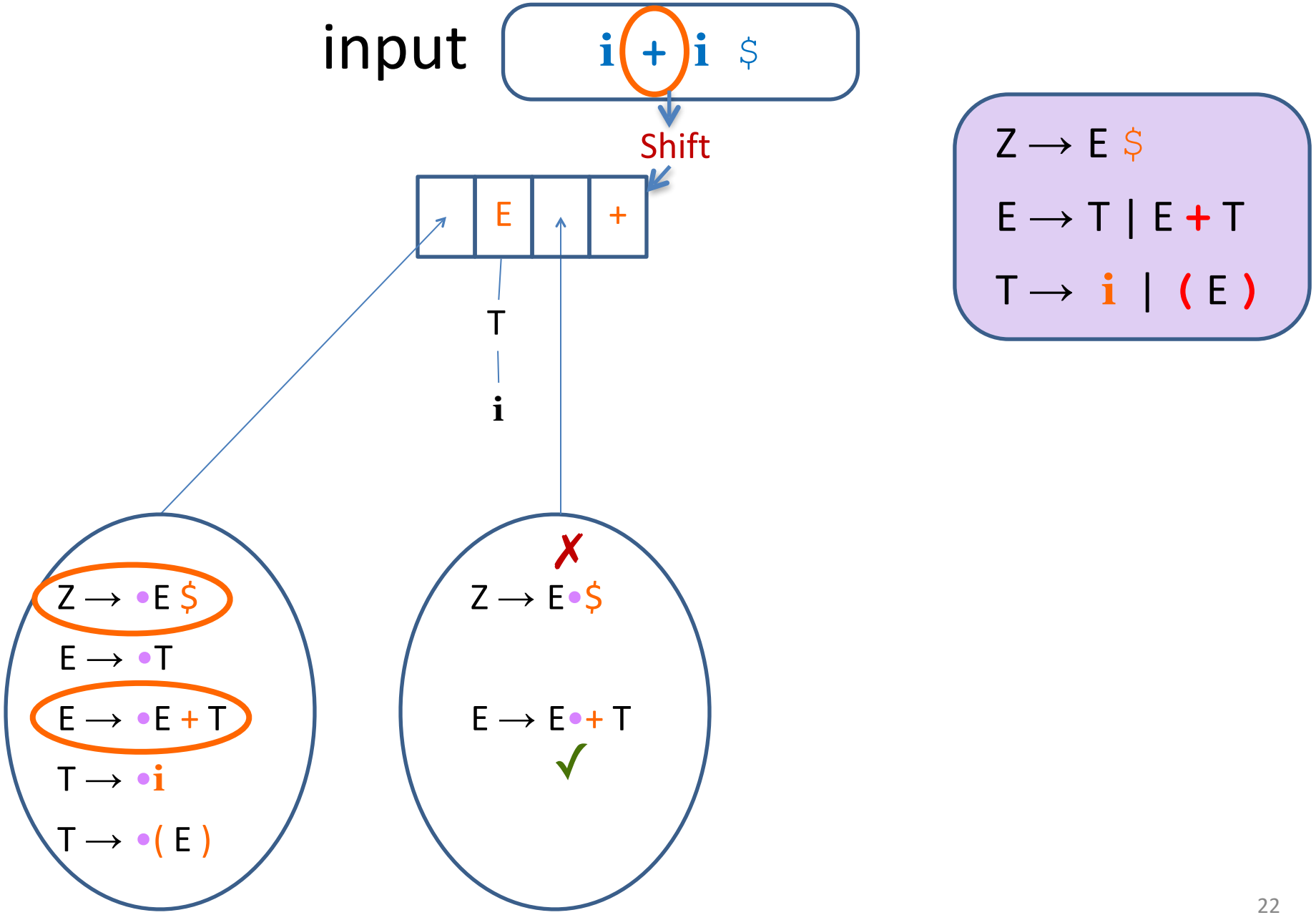
$E \rightarrow \bullet E + T$

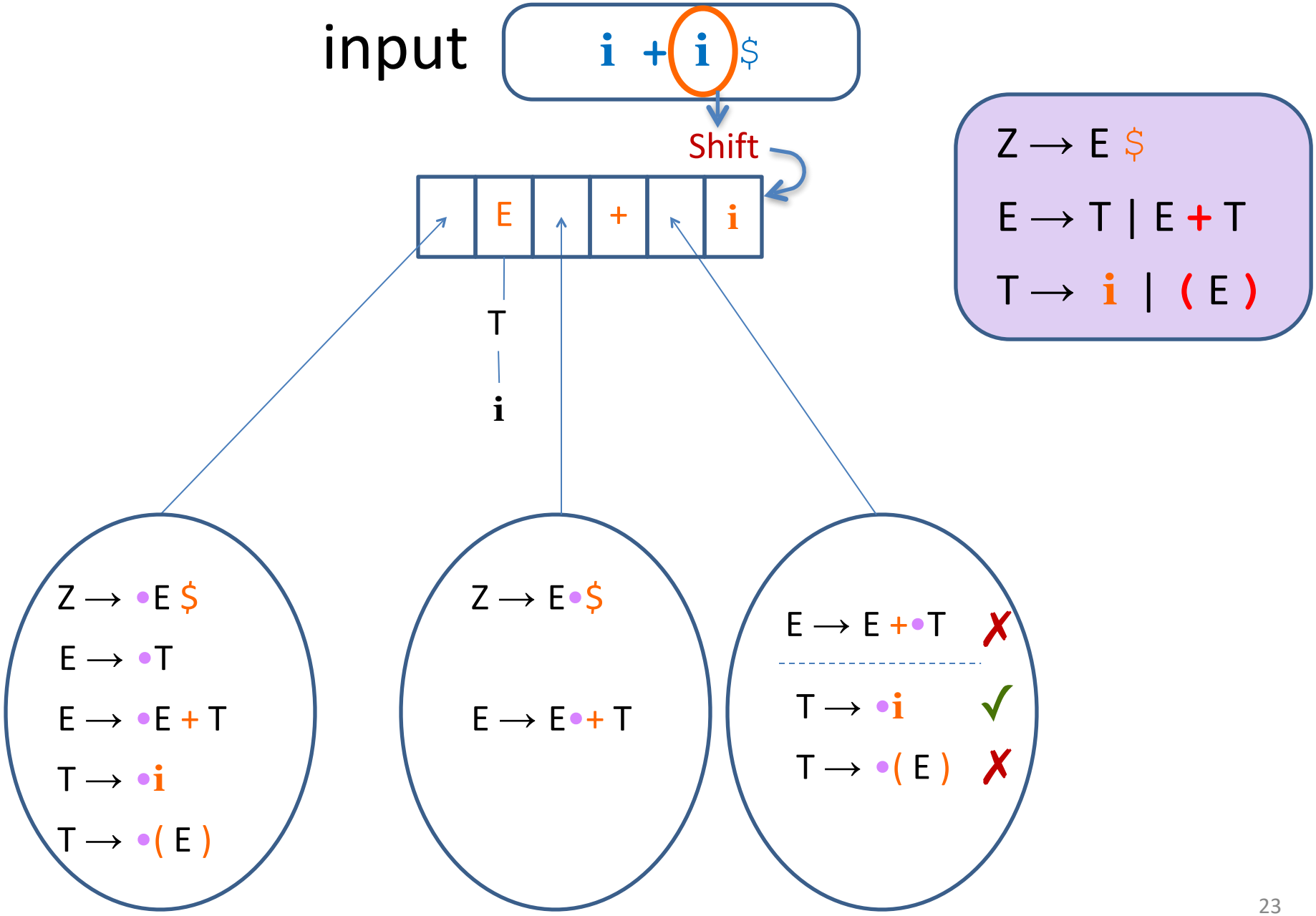
$T \rightarrow \bullet \mathbf{i}$

$T \rightarrow \bullet (\ E\ )$

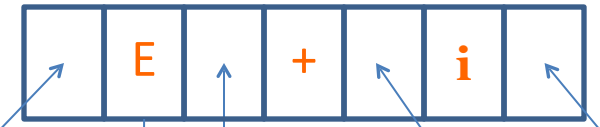
$E \rightarrow T\bullet$

Reduce item!


input
i + i $
Shift
E
+
T
i
Z → E $
E → T | E + T
T → i | (E)
Z → •E $
E → •T
E → •E + T
T → •i
T → •(E)
✗
Z → E•$
E → E•+ T
✓


input
i + i $
Shift
E
+
i
T
i
Z → E $
E → T | E + T
T → i | (E)
Z → •E $
E → •T
E → •E + T
T → •i
T → •(E)
Z → E•$
E → E•+ T
E → E +•T
T → •i
T → •(E)

input **i + i** $



T

i

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

$T \rightarrow$ **i** $\mid$ (E)

Reduce item!

$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

$E \rightarrow \bullet E + T$

$T \rightarrow \bullet$ **i**

$T \rightarrow \bullet (E)$

$Z \rightarrow E \bullet \$$

$E \rightarrow E \bullet + T$

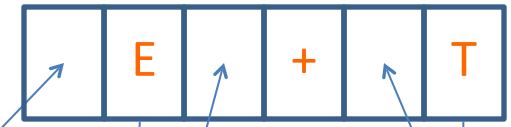
$E \rightarrow E + \bullet T$

$T \rightarrow \bullet$ **i**

$T \rightarrow \bullet (E)$

$T \rightarrow$ **i** $\bullet$

input **i + i** $



T

i

i

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

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

$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

$E \rightarrow \bullet E + T$

$T \rightarrow \bullet \mathbf{i}$

$T \rightarrow \bullet (\ E\ )$

$Z \rightarrow E \bullet \$$

$E \rightarrow E \bullet + T$

$E \rightarrow E + \bullet T$ ✓

$T \rightarrow \bullet \mathbf{i}$ ✗

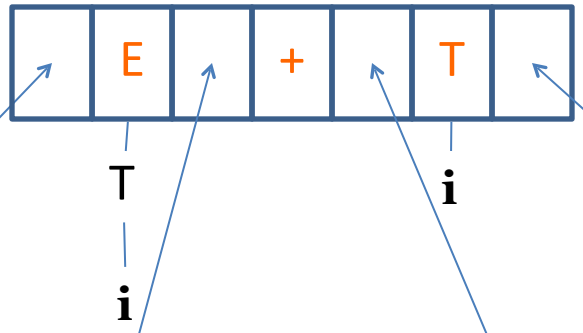
$T \rightarrow \bullet (\ E\ )$ ✗

input **i + i $**

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

$T \rightarrow i \mid (\ E\ )$



Reduce item!

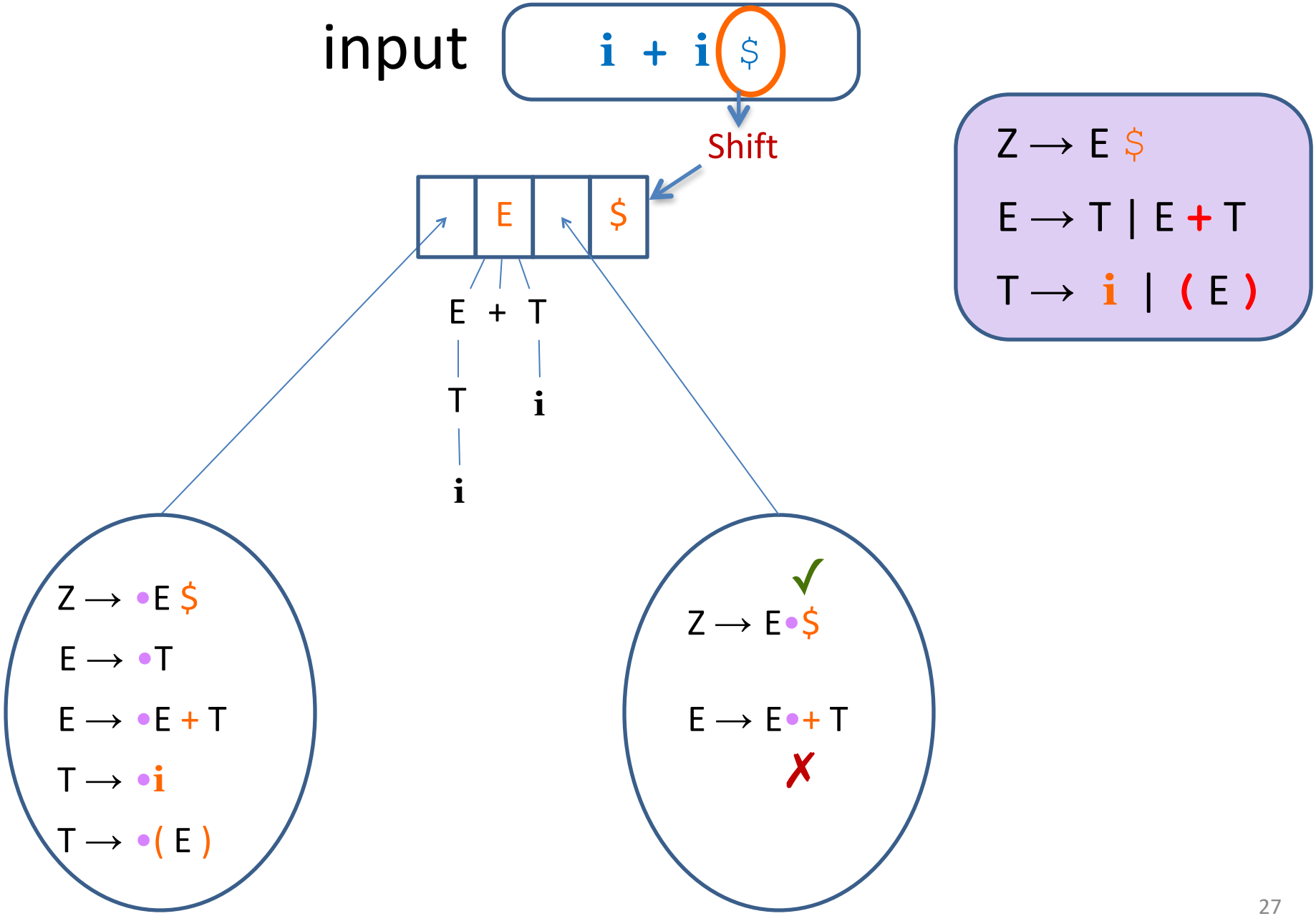
$Z \rightarrow \bullet E\ \$$
$E \rightarrow \bullet T$
$E \rightarrow \bullet E + T$
$T \rightarrow \bullet i$
$T \rightarrow \bullet (\ E\ )$

$Z \rightarrow E \bullet \$$
$E \rightarrow E \bullet + T$

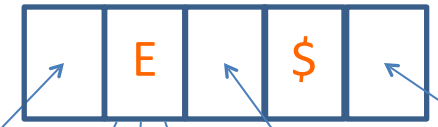
$E \rightarrow E + \bullet T$

$T \rightarrow \bullet i$
$T \rightarrow \bullet (\ E\ )$

$E \rightarrow E + T \bullet$


input
i + i $
Shift
E
$
E + T
T
i
i
Z → E $
E → T | E + T
T → i | (E)
Z → •E $
E → •T
E → •E + T
T → •i
T → •(E)
Z → E•$
E → E•+ T

input **i + i** $



E + T

T **i**

i

$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

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

$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

$E \rightarrow \bullet E + T$

$T \rightarrow \bullet \mathbf{i}$

$T \rightarrow \bullet (\ E\ )$

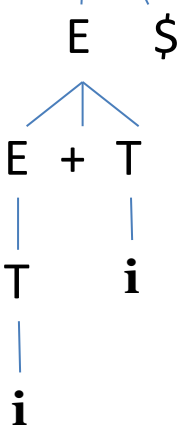
$Z \rightarrow E \bullet \$$

$E \rightarrow E \bullet + T$

$Z \rightarrow E\ \$ \bullet$

Reduce item!

input **i + i** $



$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

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

$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

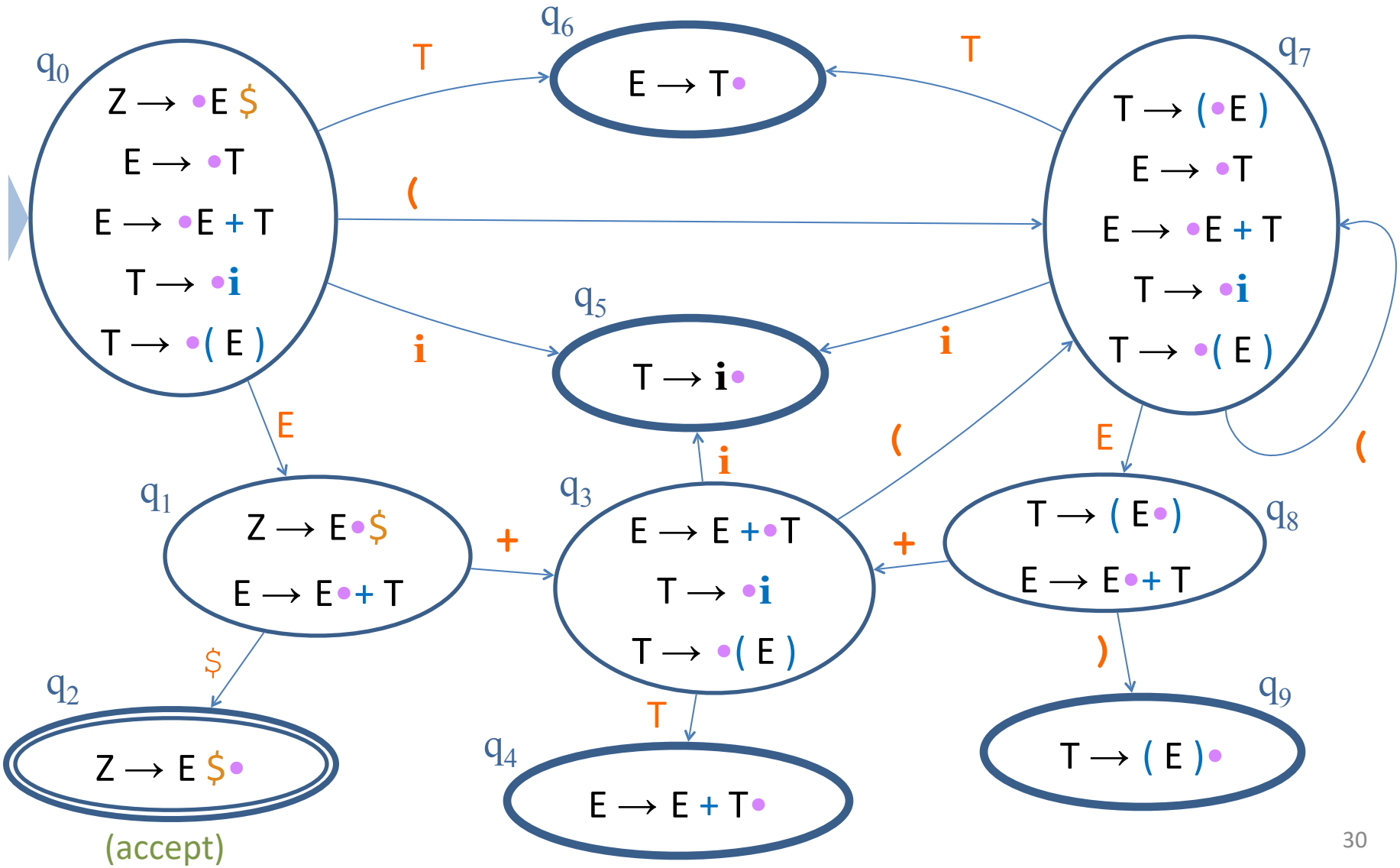
$E \rightarrow \bullet E + T$

$T \rightarrow \bullet \mathbf{i}$

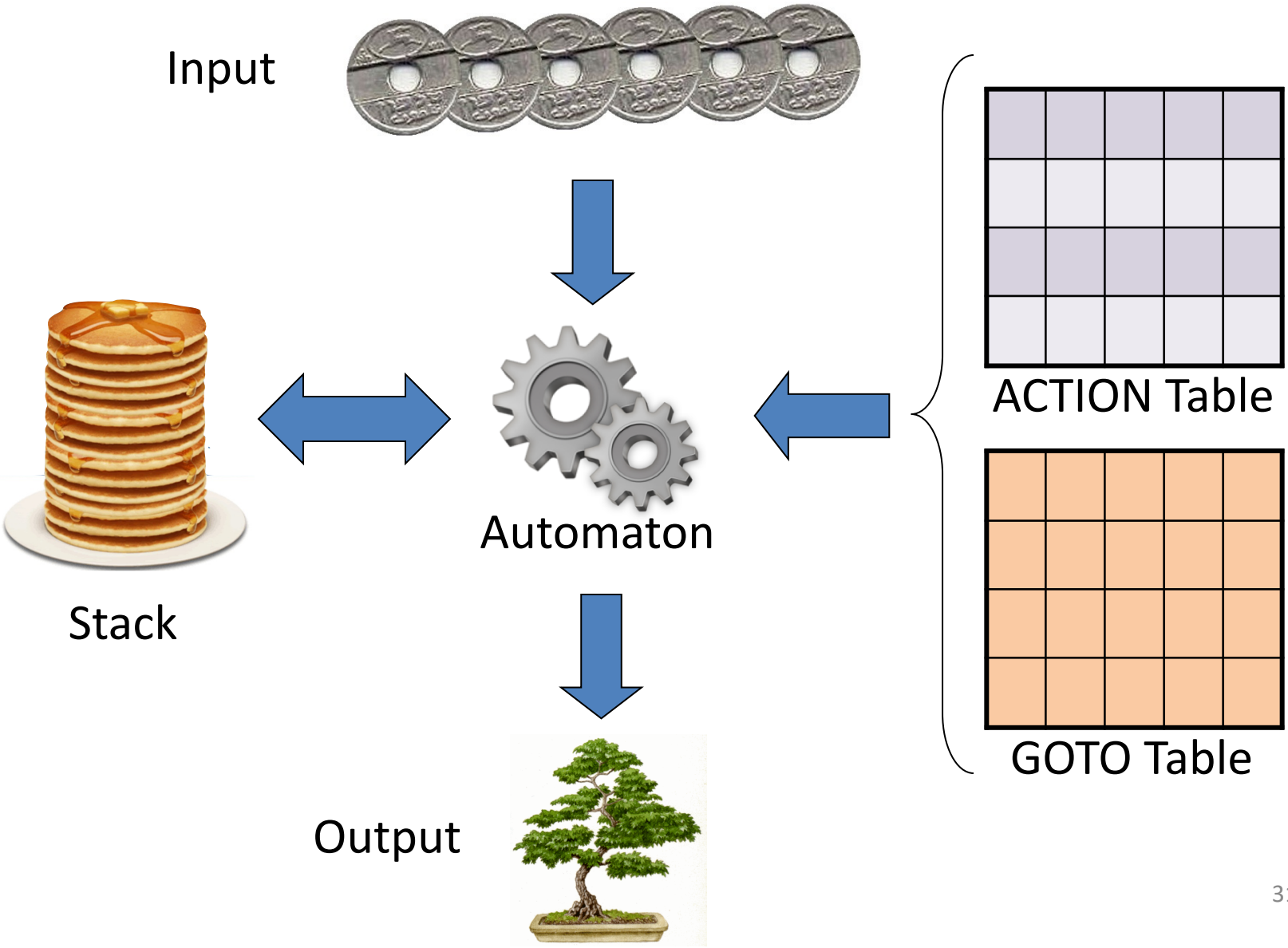
$T \rightarrow \bullet (\ E\ )$

Reducing the initial rule means accept

View as an LR(0) Automaton



LR(0) Parsing



Why do we need a stack?

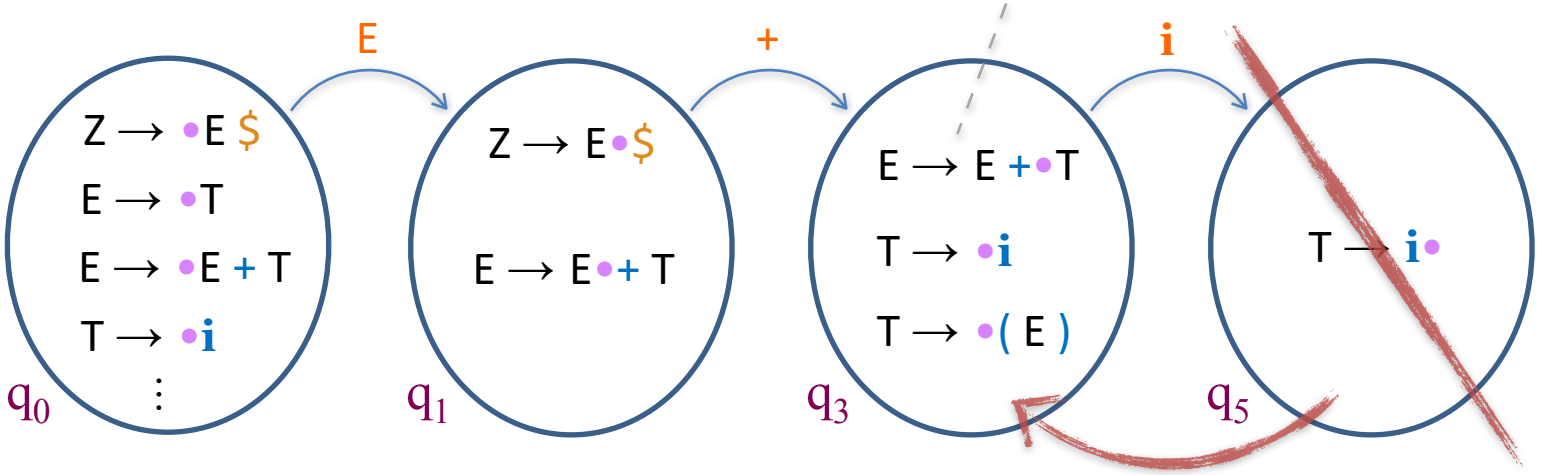
$Z \rightarrow E\ \$$

$E \rightarrow T \mid E + T$

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

- Suppose so far we have discovered $E \Rightarrow T \Rightarrow \mathbf{i}$ and $+$;
  So we have constructed the sentential form "$E +$".
- In the given grammar, this is expressed by the item:

$$E \rightarrow E + \bullet T$$

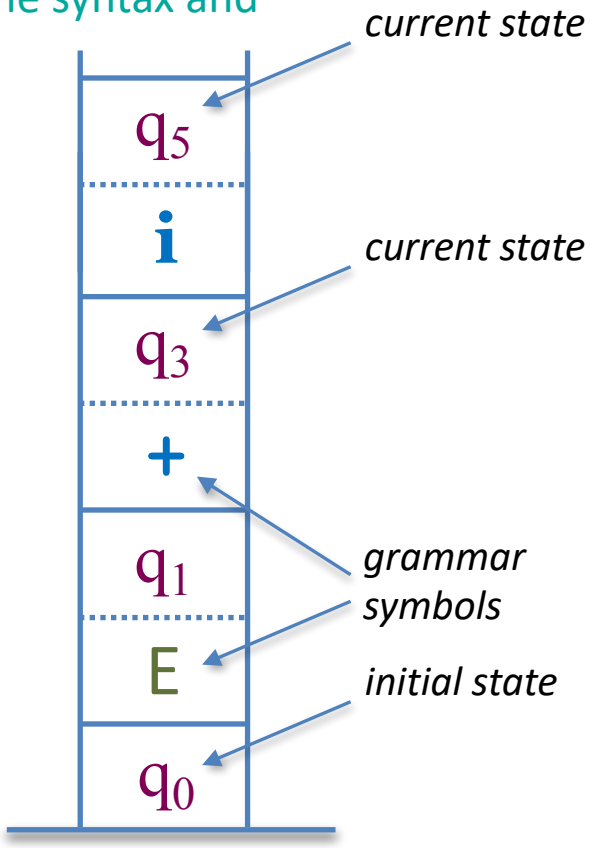


- Now, the next token is **i**, so we need to switch to another state q_5, in order to derive $T \Rightarrow \mathbf{i}$. (since $E + \mathbf{i}$ does not match any rule)

- After identifying T, we have to go back to q_3.
  So the path must be recorded *in memory*. The stack is that memory.

The Stack

- The stack contains states and grammar symbols
  - ★ (*fun fact*: in a situation where we just want to check the syntax and return true/false, we could do without the symbols)
- The initial stack contains q_0 only
- The rest of the stack contains pairs of (state, token)
  or (state, nonterminal)



LR(0) Automaton Construction

- Typically a state consists of several LR items
- For example, if we identified a string that is reduced to E, then we may be in one of the following LR items:

$$E \rightarrow E \bullet + B \quad \text{or} \quad E \rightarrow E \bullet * B$$

- Therefore one state would be:

$$q = \{E \rightarrow E \bullet + B \, , \, E \rightarrow E \bullet * B\}$$

- But if the current state includes $E \rightarrow E + \bullet B$, then we must allow B to be derived too — **Closure**!

Construct the Closure

- Proposition: a closure set of LR(0) items has the following property — if the set contains an item of the form

$$A \longrightarrow \alpha \bullet B\ \beta$$

then it must *also* contain an item

$$B \longrightarrow \bullet \delta$$

for *each rule* of the form $B \longrightarrow \delta$ in the grammar.

- Building the closure set for a given item set is recursive, as δ may also begin with a variable.

Closure: an example

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

$B \rightarrow 0 \mid 1$

grammar

$C = \{ E \rightarrow E + \bullet B \}$

set C of LR items

- The closure of the set C is

$$clos(C) = \{ \; E \rightarrow E + \bullet B \;,$$
$$B \rightarrow \bullet 0 \;,$$
$$B \rightarrow \bullet 1 \; \}$$

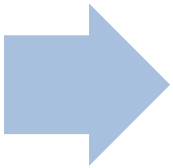
- This will become another parser state

Extended Grammar

- Goal: simple termination condition
  - Assume that the initial variable only appears in a single rule. This guarantees that the last reduction can be (easily) detected.
  - Any grammar can be (easily) extended to have such structure.

Example: the grammar

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



Can be extended into

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

end-of-input marker

The Initial State

- To build the ACTION/GOTO table, we go through all possible states that can be seen during derivation
  - Each state represents a (closure) set of LR(0) items

- The initial state q_0 is the closure of the initial rule
  - In our example the initial rule is S ⟶ ●E $, so:

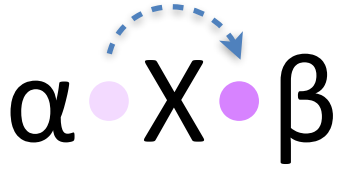
q_0 = clos({S ⟶ ●E $ }) =

{ S ⟶ ●E $, E ⟶ ●E * B , E ⟶ ●E + B ,
E ⟶ ●B , B ⟶ ●0 , B ⟶ ●1 }

- Next, we build all possible states that can be reached by following a *single symbol* (token or variable)

The Next States

- For every symbol (terminal or variable) X, and every possible state (closure set) *q*,
  1. Find all items in the set of q in which the dot is before an X. We denote this set by *q*|X
  2. Move the dot ahead of the X in all items in *q*|X
  3. Find the closure of the obtained set: this is the state into which we move from *q* upon seeing X



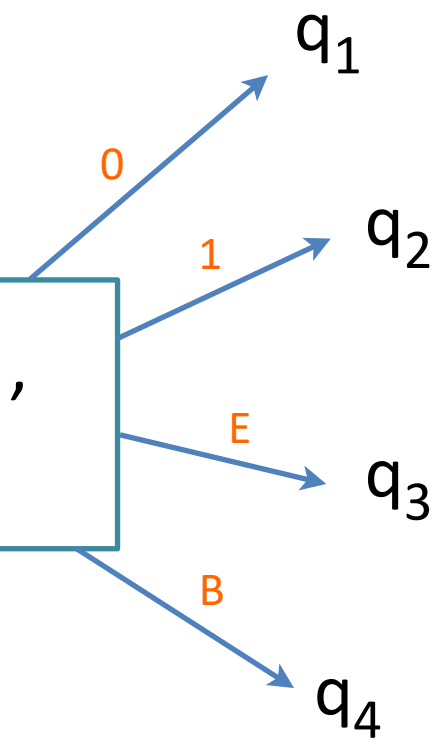
- Formally, the next set is defined on a set C and next symbol X:
  - C|X = { (N ⟶ α•Xβ) ∈ C }
  - step(C,X) = { N ⟶ αX•β | (N ⟶ α•Xβ) ∈ C}
  - nextSet(C,X) = clos(step(C,X))

The Next States

Recall that in our example

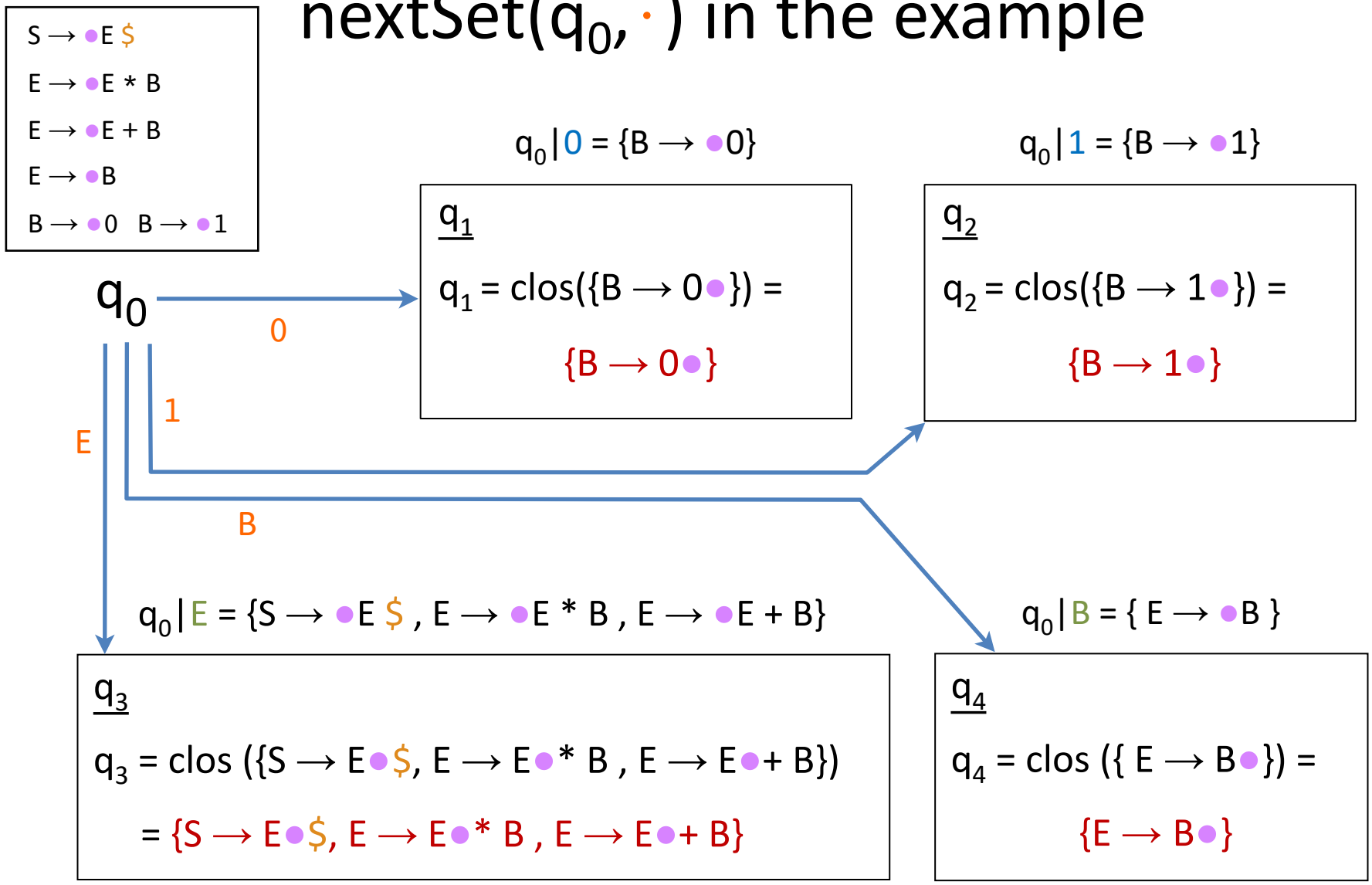
q_0 = clos({S → ●E }) =

{S → ●E $, E → ●E * B , E → ●E + B ,
E → ●B , B → ●0 , B → ●1}



Let us check which states are reachable from it.

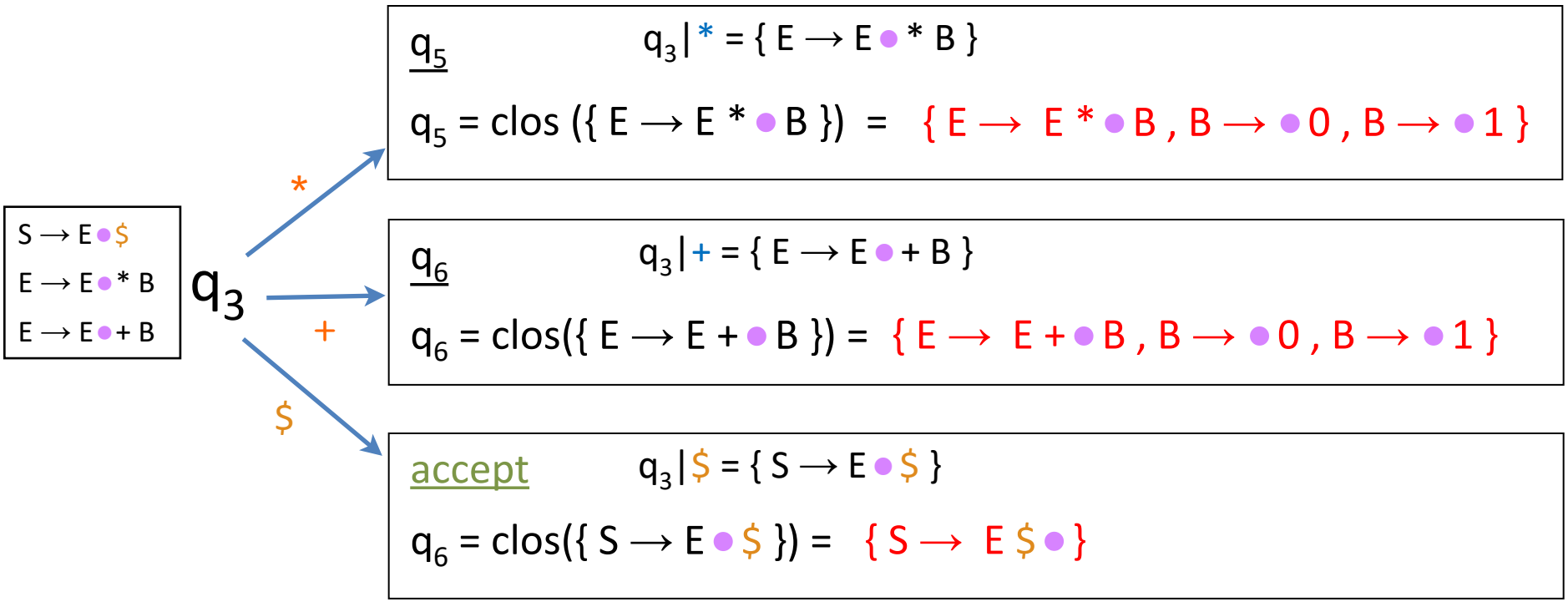
nextSet(q_0, $\cdot$) in the example



From these new states there are more reachable states

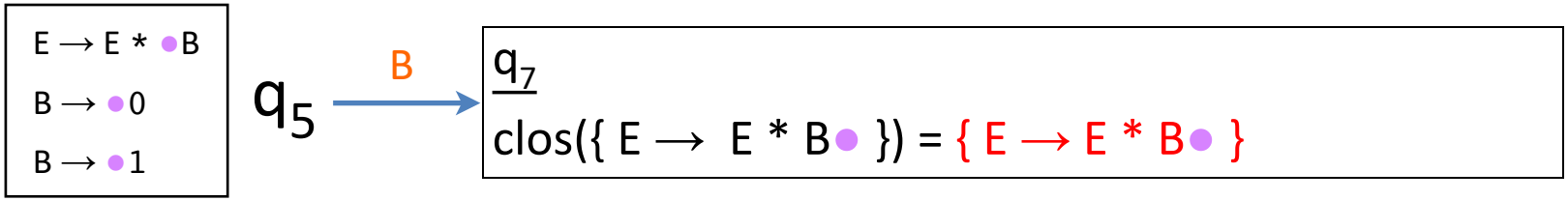
- From q_1, q_2, q_4, there are no steps because the dot is at the end of every item in their sets.

- From state q_3 we can reach the following three states —

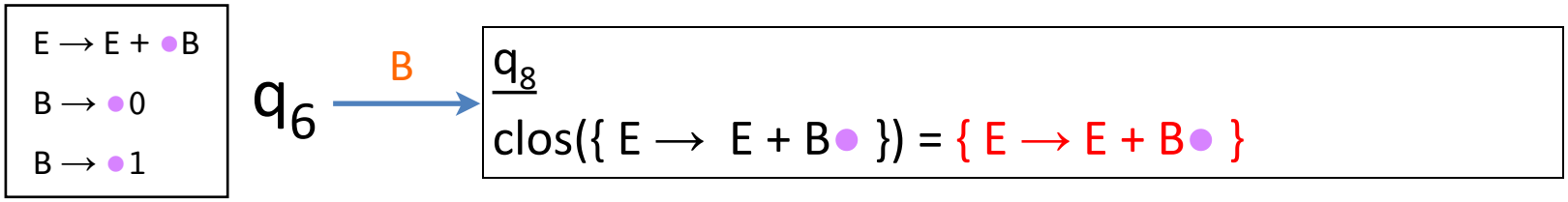


Finally

- From q_5 we can proceed with X=0, or X=1, or X=B.
- For X=0 we reach q_1 again; for X=1 we reach q_2.
- For X=B we get q_7:

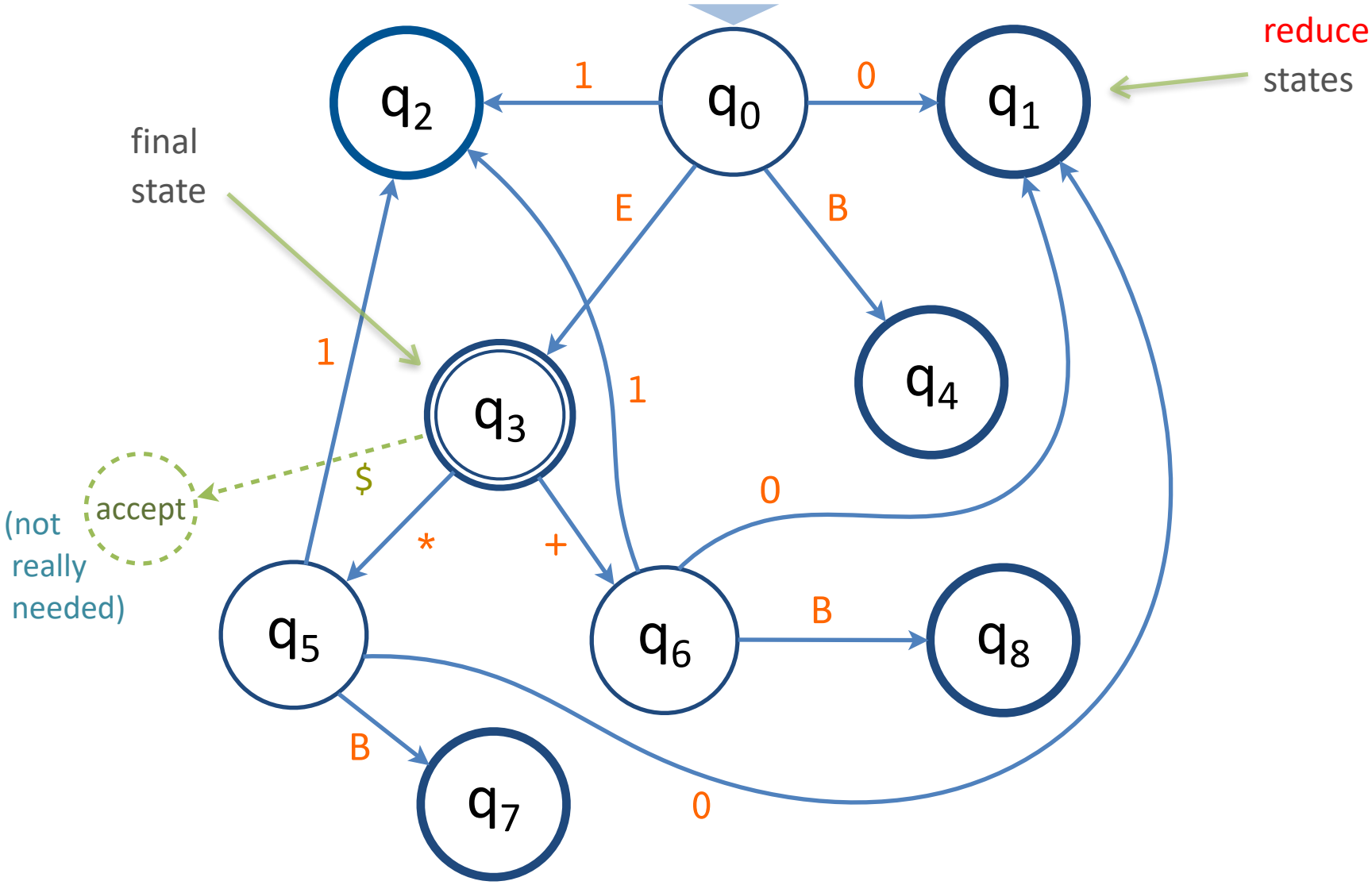


- Similarly, from q_6 with X=B we get q_8:



- These two states have no further steps. (Why?)

Automaton


reduce
states
final
state
q0
q1
q2
q3
q4
q5
q6
q7
q8
1
0
E
B
1
1
$
accept
(not
really
needed)
*
+
0
B
B
0

LR(0) Transition Table

	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		

shift "1" and go to state q_2

reduce by rule (4) B → 0

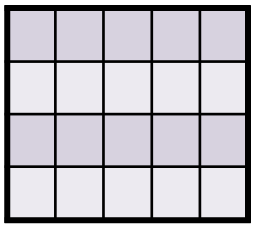
accept (*hurray*)

(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 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, σ] table tells us what to do based on current state q and next token σ:

  shift i : shift and move to q_i

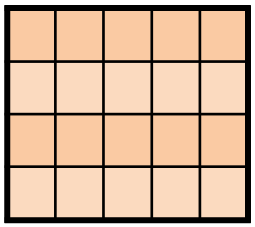
  reduce j: reduce according to production rule (j)

  also: accept and error conditions

  ⋮

  (empty cells)

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

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							

- 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).

Building the Tables: Shift

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

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

Building the Tables: Reduce

State	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		

For any state which includes an item

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

$A \rightarrow \alpha$ 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.

The 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 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.

GOTO/ACTION Table

State	ACTION					GOTO	
	i	+	(	)	$	E	T
q_0	s5		s7			1	6
q_1		s3			acc		
q_3	s5		s7				4
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			8	6
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)

sn = shift to state *n*

rm = reduce using rule number (*m*)

GOTO/ACTION Table

(1) Z → E $
(2) E → T
(3) E → E + T
(4) T → i
(5) T → (E)

	ACTION					GOTO	
	i	+	(	)	$	E	T
q_0	s5		s7			1	6
q_1		s3			acc		
q_3	s5		s7				4
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			8	6
q_8		s3		s9			
q_9	r5	r5	r5	r5	r5		

top is on the right

Stack	Input	Action
q_0	i + i $	s5
q_0 i q_5	+ i $	r4
q_0 T q_6	+ i $	r2
q_0 E q_1	+ i $	s3
q_0 E q_1 + q_3	i $	s5
q_0 E q_1 + q_3 i q_5	$	r4
q_0 E q_1 + q_3 T q_4	$	r3
q_0 E q_1	$	accept
(q_0 Z)		

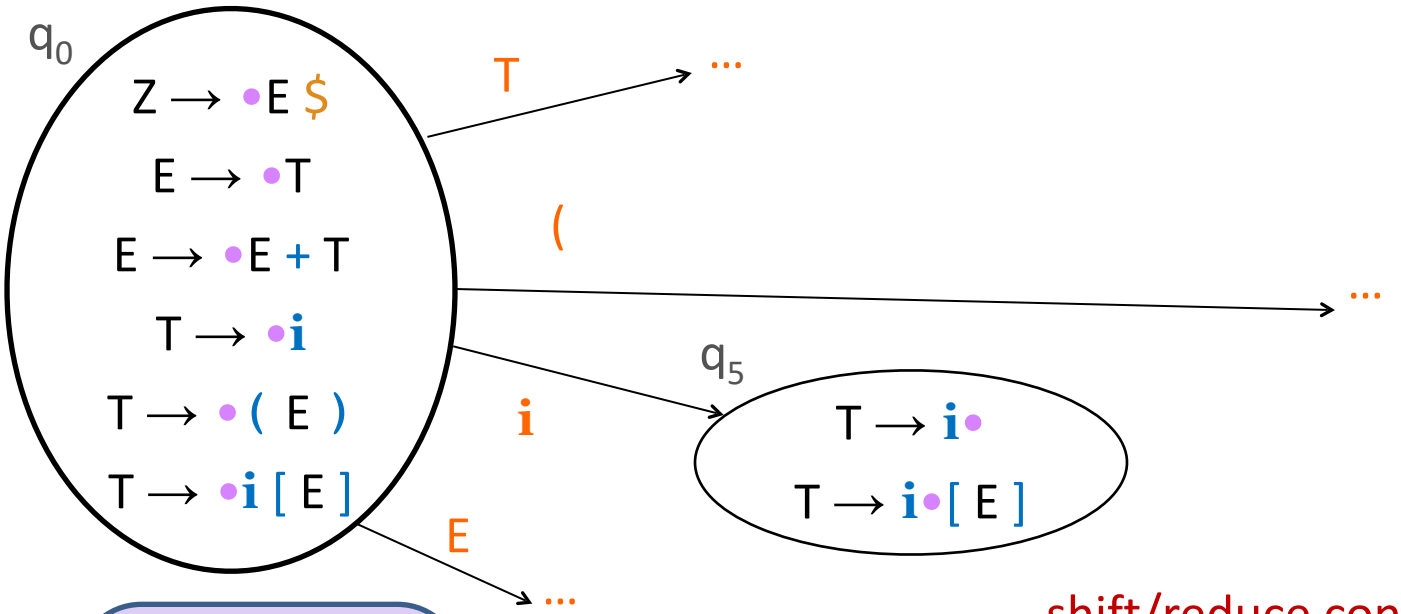
Are we done?

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

- …but it is not always clear what to do at each state

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

LR(0) Conflicts



shift/reduce conflict

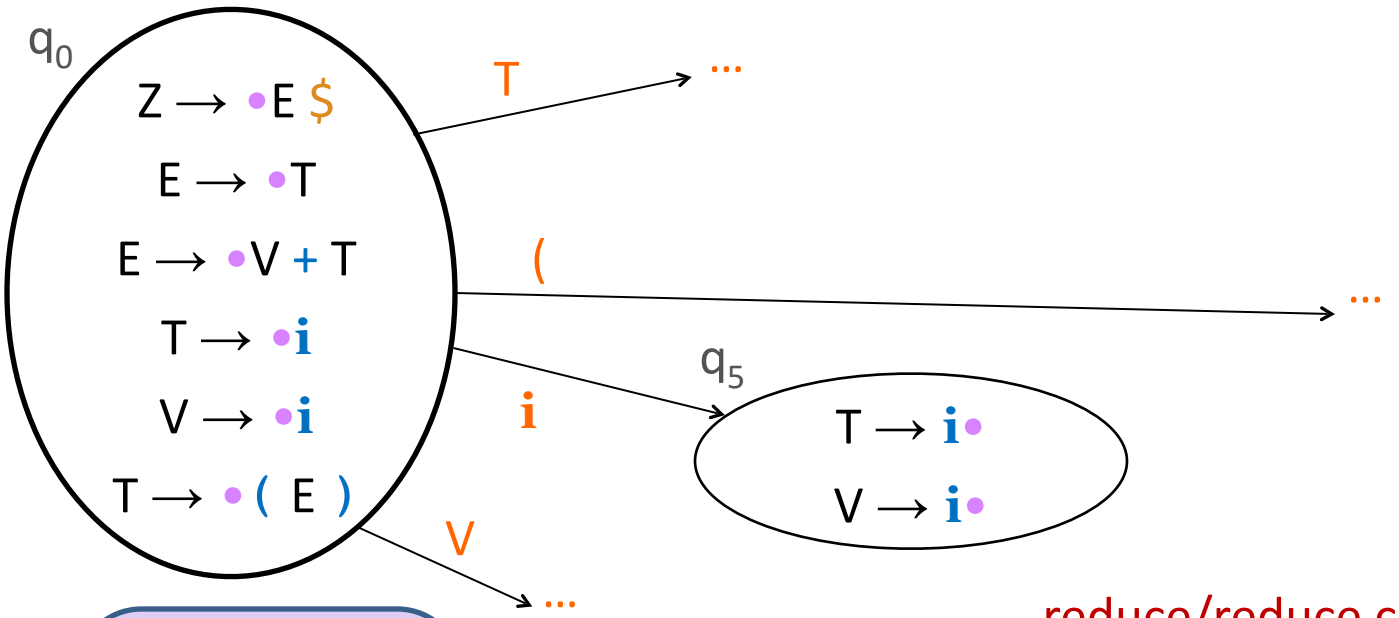
Z → E $
E → T
E → E + T
T → i
T → (E)
T → i [E]

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

Z → E $

E → T

E → V + T

T → i

V → i

T → (E)

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 $\{P \to \alpha \bullet A \beta\}$

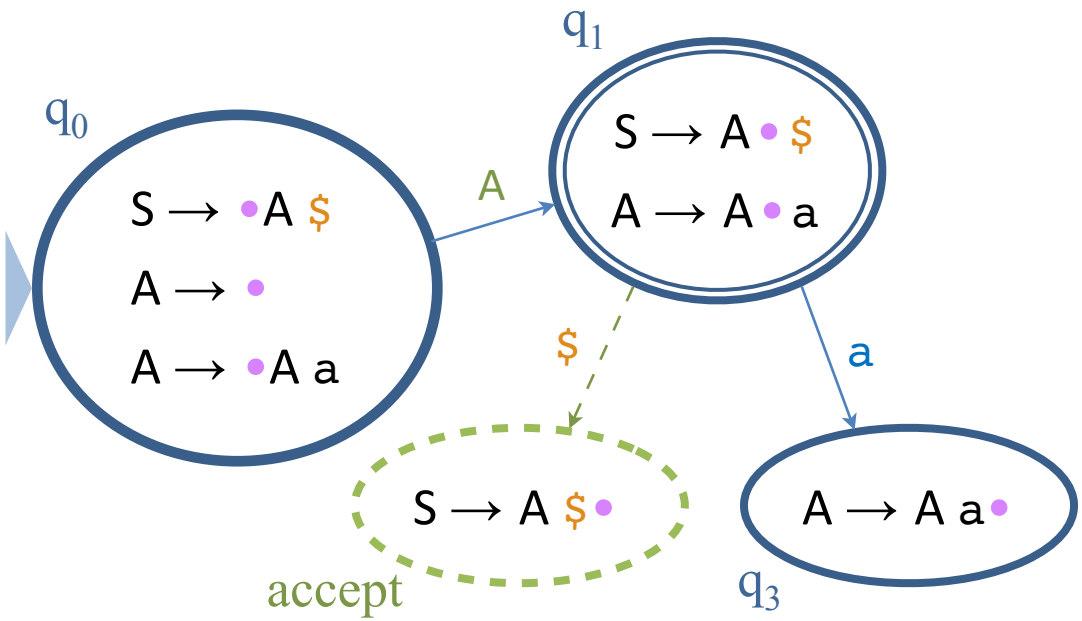
$A \to \varepsilon$
$A \to A\ a$

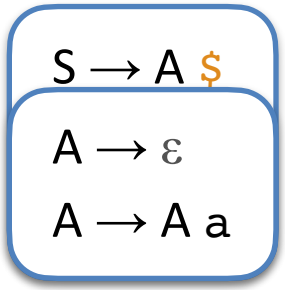
✓ This is fine

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

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

LR is More Powerful, But…

- Theorem: Any LL(k) language is also in LR(k) (and not vice versa); *i.e.*, LL(k) $\subset$ LR(k)

- **But** the lookahead is counted differently in the two cases:

With LL(k), the algorithm sees k tokens of the right-hand side of the rule and then must select the derivation rule

With LR(k), the algorithm sees the entire right-hand side of the derivation rule plus k more tokens

- LR(0) sees the *entire right-side*

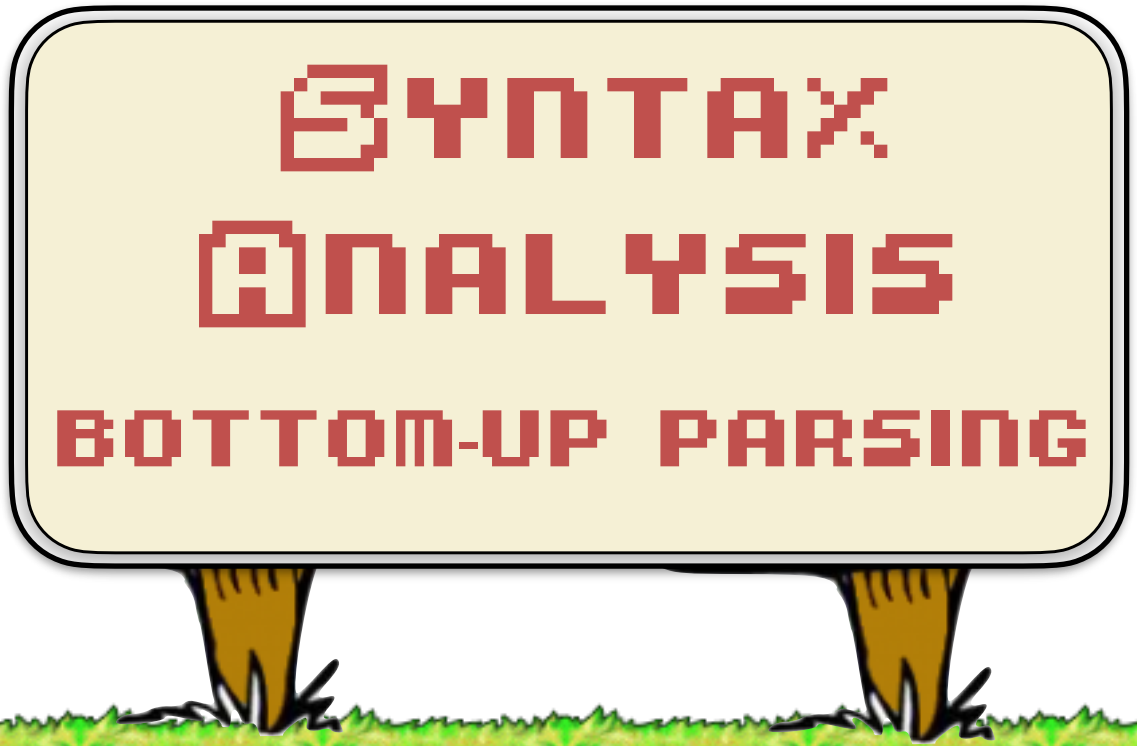
?

- The LR family of parsers is more popularly used today
  - *…but there are prominent examples of LL parsers in practice*

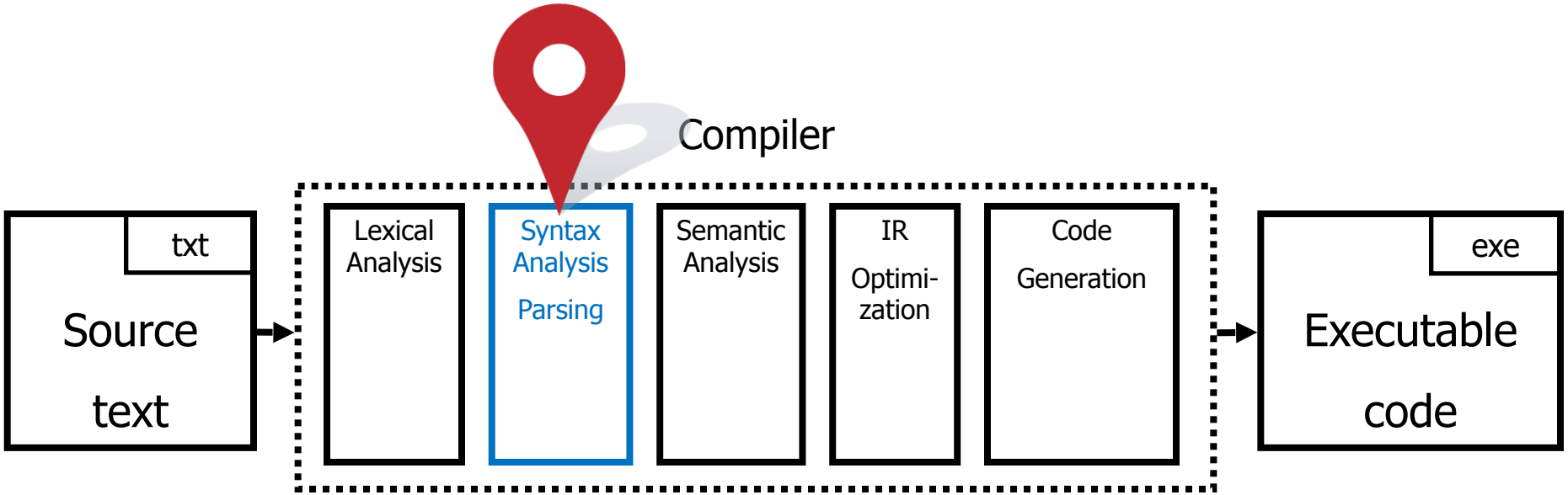
Coming Up

THEORY OF COMPILATION

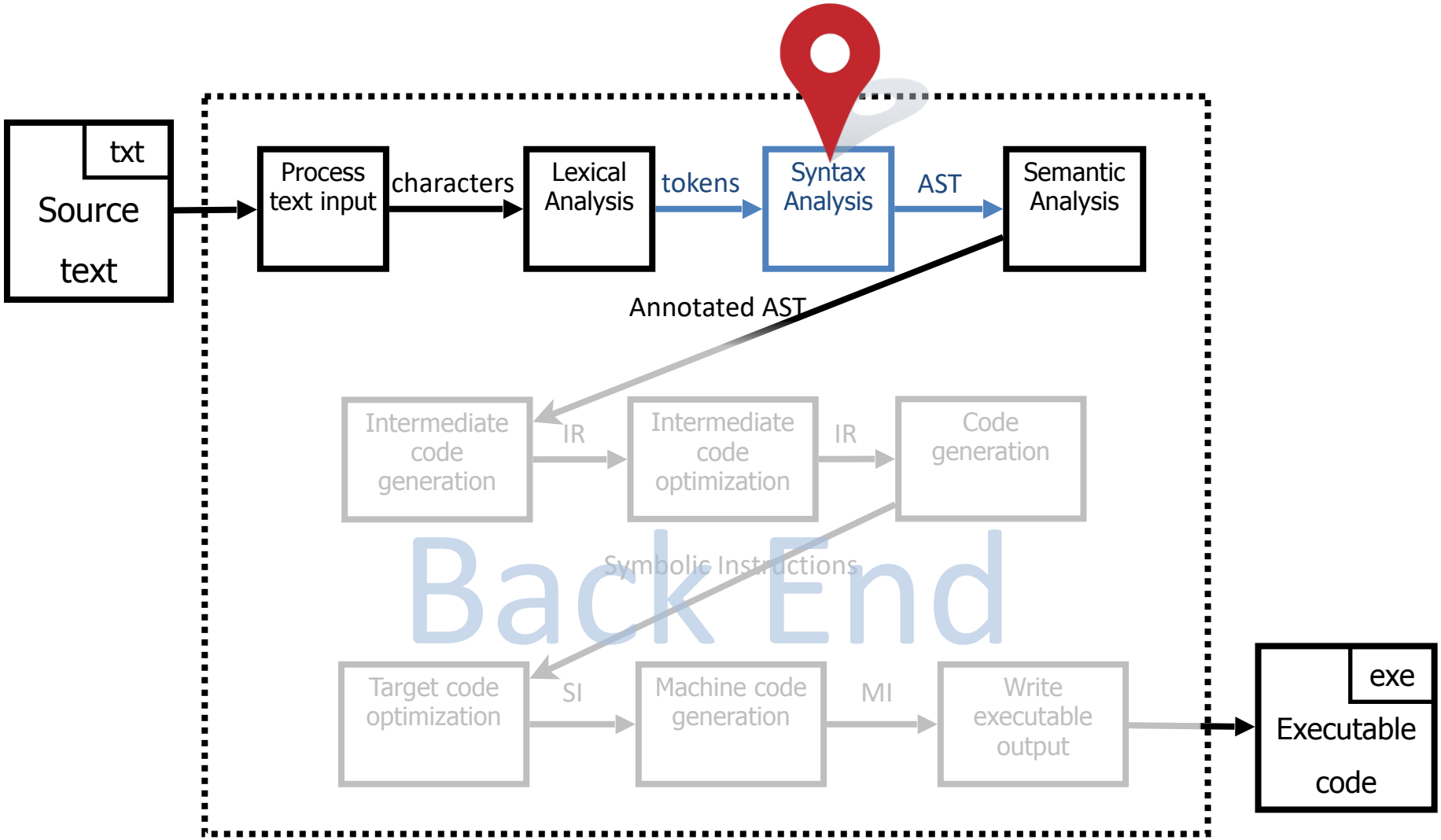
LECTURE 04



You are here

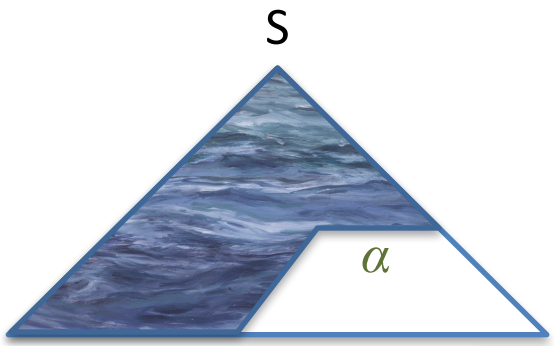

Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code Generation
exe
Executable code

You are here

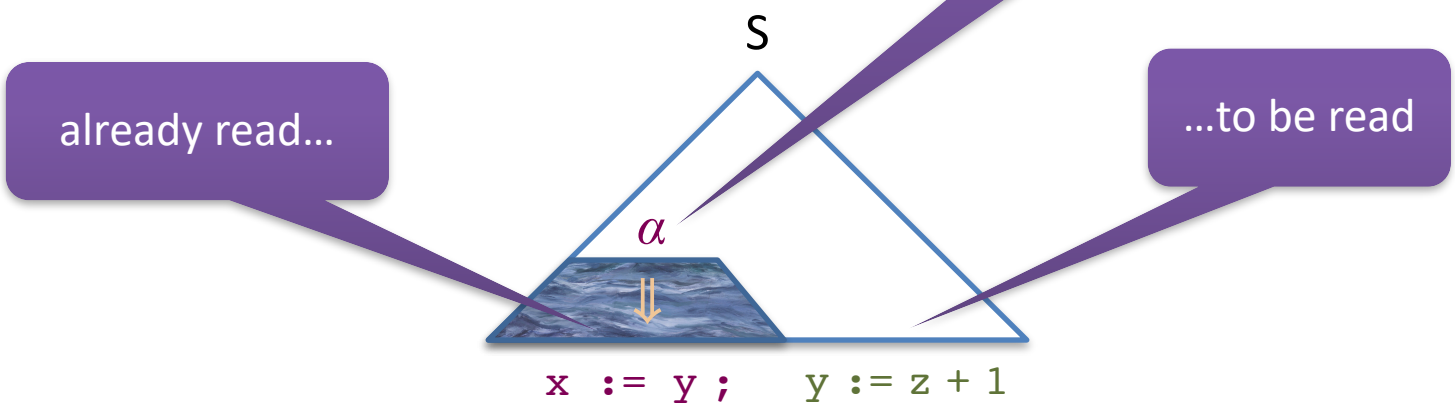

txt
Source text
Process text input
characters
Lexical Analysis
tokens
Syntax Analysis
AST
Semantic Analysis
Annotated AST
Intermediate code generation
IR
Intermediate code optimization
IR
Code generation
Back End
Symbolic Instructions
Target code optimization
SI
Machine code generation
MI
Write executable output
exe
Executable code

Efficient Parsers

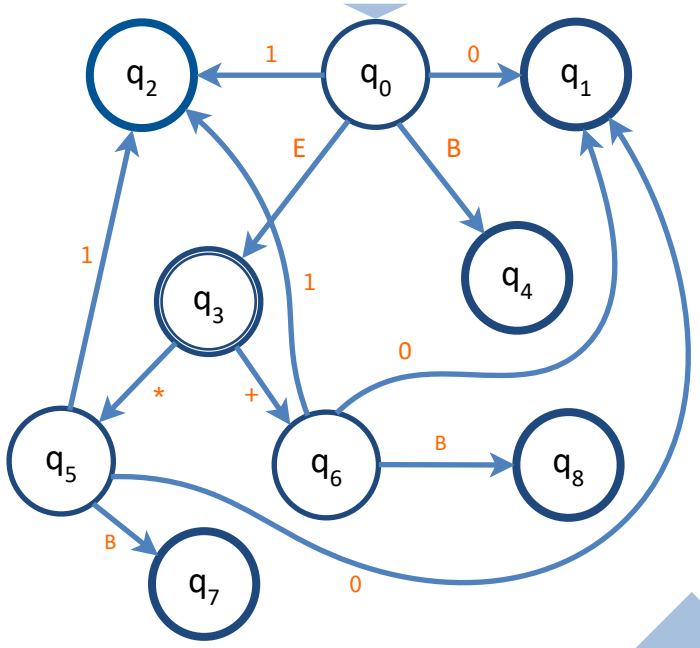
- Top-down (predictive)



- Bottom-up (shift-reduce)



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



LR(0) Automaton

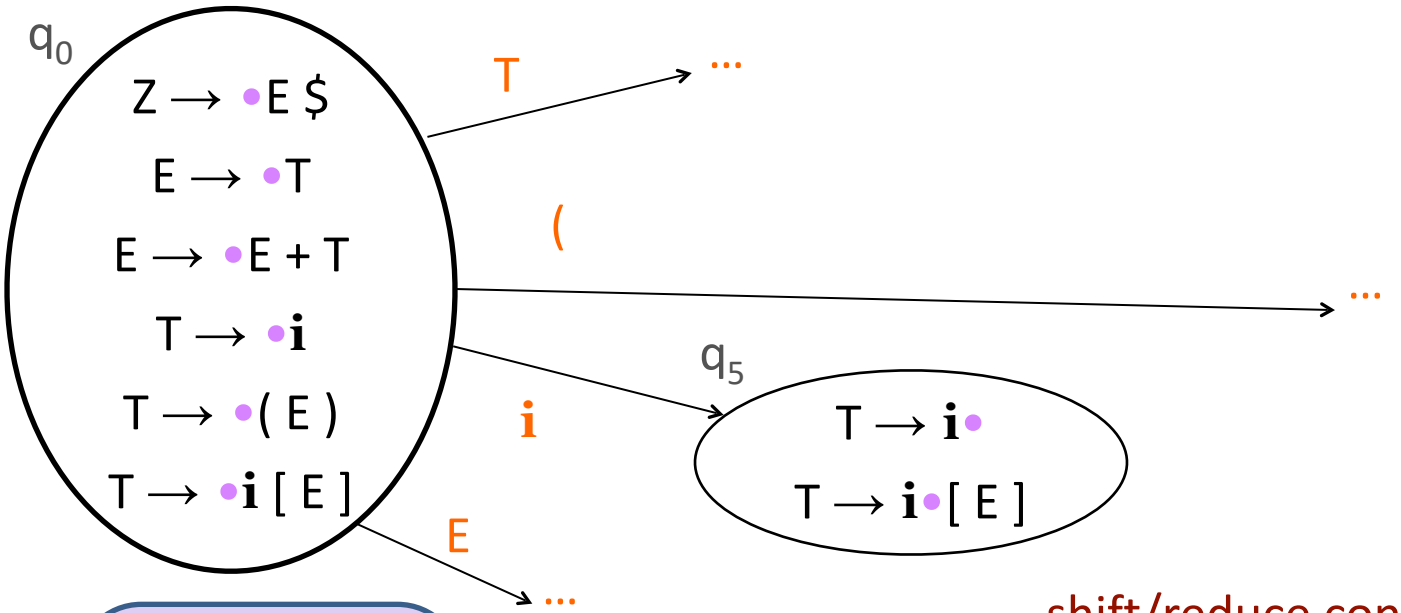
Input

	i	+	(	)	$	E	T
q_0	s5		s7			1	6
q_1		s3			s2		
q_2	r1	r1	r1	r1	r1	r1	r1
q_3	s5		s7				4
q_4	r3	r3	r3	r3	r3	r3	r3
q_5	r4	r4	r4	r4	r4	r4	r4
q_6	r2	r2	r2	r2	r2	r2	r2
q_7	s5		s7			8	6
q_8		s3		s9			
q_9	r5	r5	r5	r5	r5	r5	r5

LR(0) Parsing Algorithm

Output
Syntax Tree

Reminder – LR(0) Conflicts



shift/reduce conflict

Z → E $

E → T

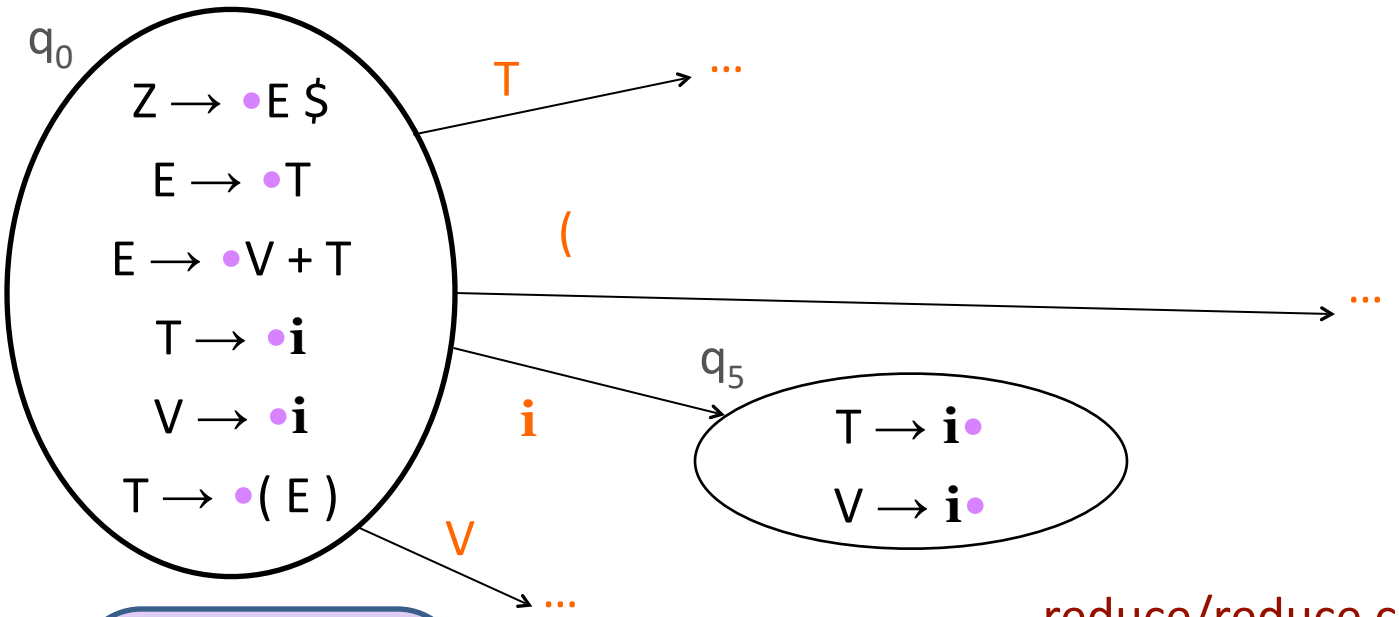
E → E + T

T → i

T → (E)

T → i [E]

Reminder – LR(0) Conflicts



reduce/reduce conflict

Z → E $

E → T

E → V + T

T → i

V → i

T → (E)

Back to Action/Goto Table

- Remember? Reductions ignore the input…

	i	+	(	)	$	E	T
q_0	s4		s6			1	5
q_1		s2			acc		
q_2	s4		s6				3
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			7	5
q_7		s2		s8			
q_8	r5	r5	r5	r5	r5		

SLR Grammars

- 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 look-ahead is in FOLLOW(N)

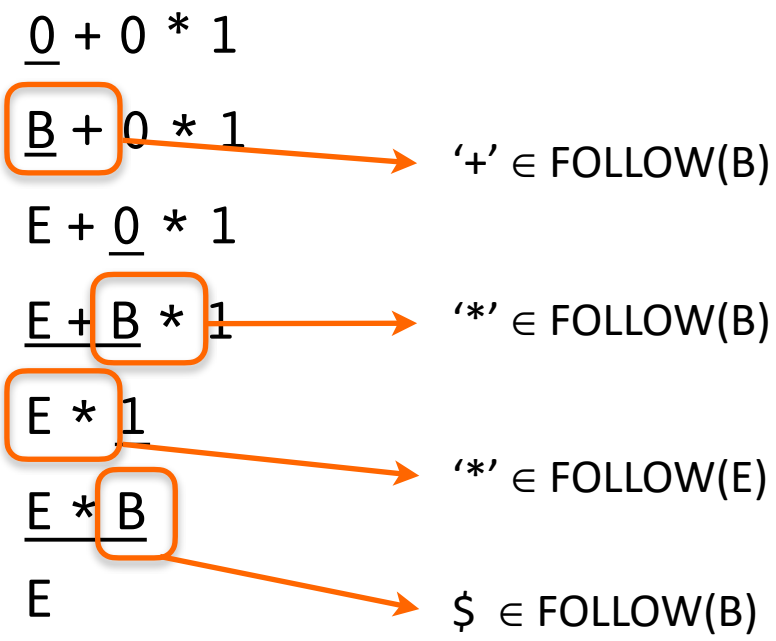
- Differs from LR(0) only on the original reduce rows
- Allows us to sometimes not reduce, instead shift (or do nothing — detect error sooner)

SLR Grammars and FOLLOW

$E \longrightarrow E * B \;(1) \mid E + B \;(2) \mid B \;(3)$

$B \longrightarrow 0 \;(4) \mid 1 \;(5)$

Derivation (in reverse):



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			1	5
q_1		s2			acc		
q_2	s4		s6				3
q_3		r3		r3	r3		
q_4	r4	r4	r4	r4	r4		
q_5	r2	r2	r2	r2	r2		
q_6	s4		s6			7	5
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			1	5
q_1		s2			acc		
q_2	s4		s6				3
q_3		r3		r3	r3		
q_4	r4	r4	r4	r4	r4		
q_5		r2		r2	r2		
q_6	s4		s6			7	5
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			1	5
q_1		s2			acc		
q_2	s4		s6				3
q_3		r3		r3	r3		
q_4		r4		r4	r4		
q_5		r2		r2	r2		
q_6	s4		s6			7	5
q_7		s2		s8			
q_8		r5		r5	r5		

(1) Z → E $

(2) E → T

(3) E → E + T

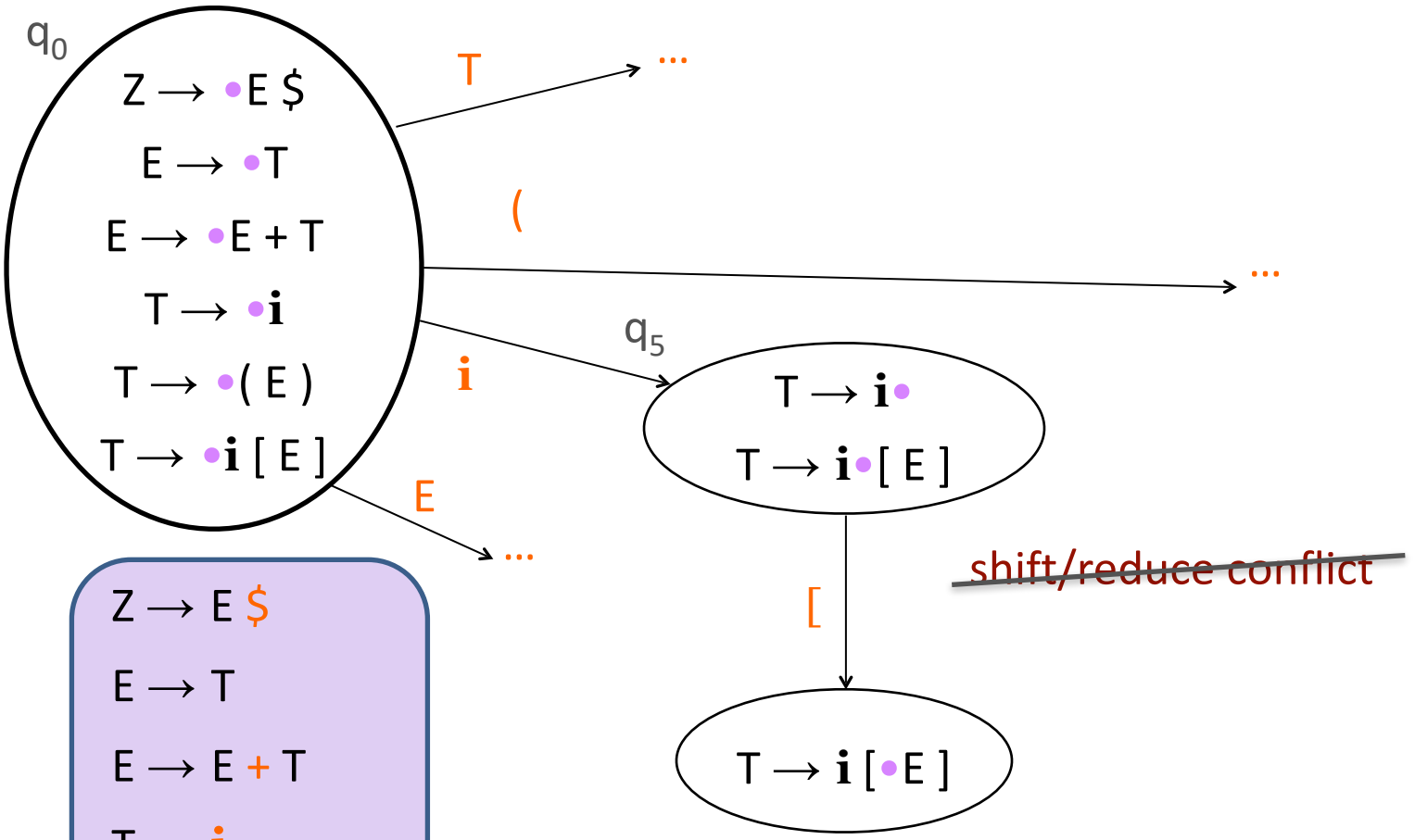
(4) T → i

(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]

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

Now let's add "T → i [E]"

(1) Z → E $
(2) E → T
(3) E → E + T
(4) T → i
(5) T → (E)
(6) T → i [E]

	i	+	(	)	[	]	$	E	T
q_0	s4		s6					1	5
q_1		s2					acc		
q_2	s4		s6						3
q_3		r3		r3			r3		
q_4		r4		r4	s9		r4		
q_5		r2		r2			r2		
q_6	s4		s6					7	5
q_7		s2		s8					
q_8		r5		r5			r5		
q_9	s4		s6					10	
⋮									

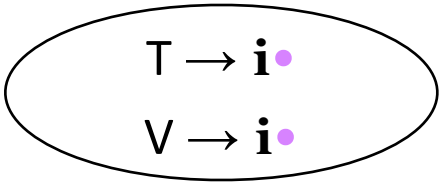
yay 😊

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

SLR: check next token when reducing

- Simple LR(1), or SLR(1), or SLR.
- Example demonstrates elimination of a shift/reduce conflict.
- Can eliminate reduce/reduce conflicts when conflicting rules' left-hand sides satisfy:

$$\text{FOLLOW}(T) \cap \text{FOLLOW}(V) = \emptyset.$$

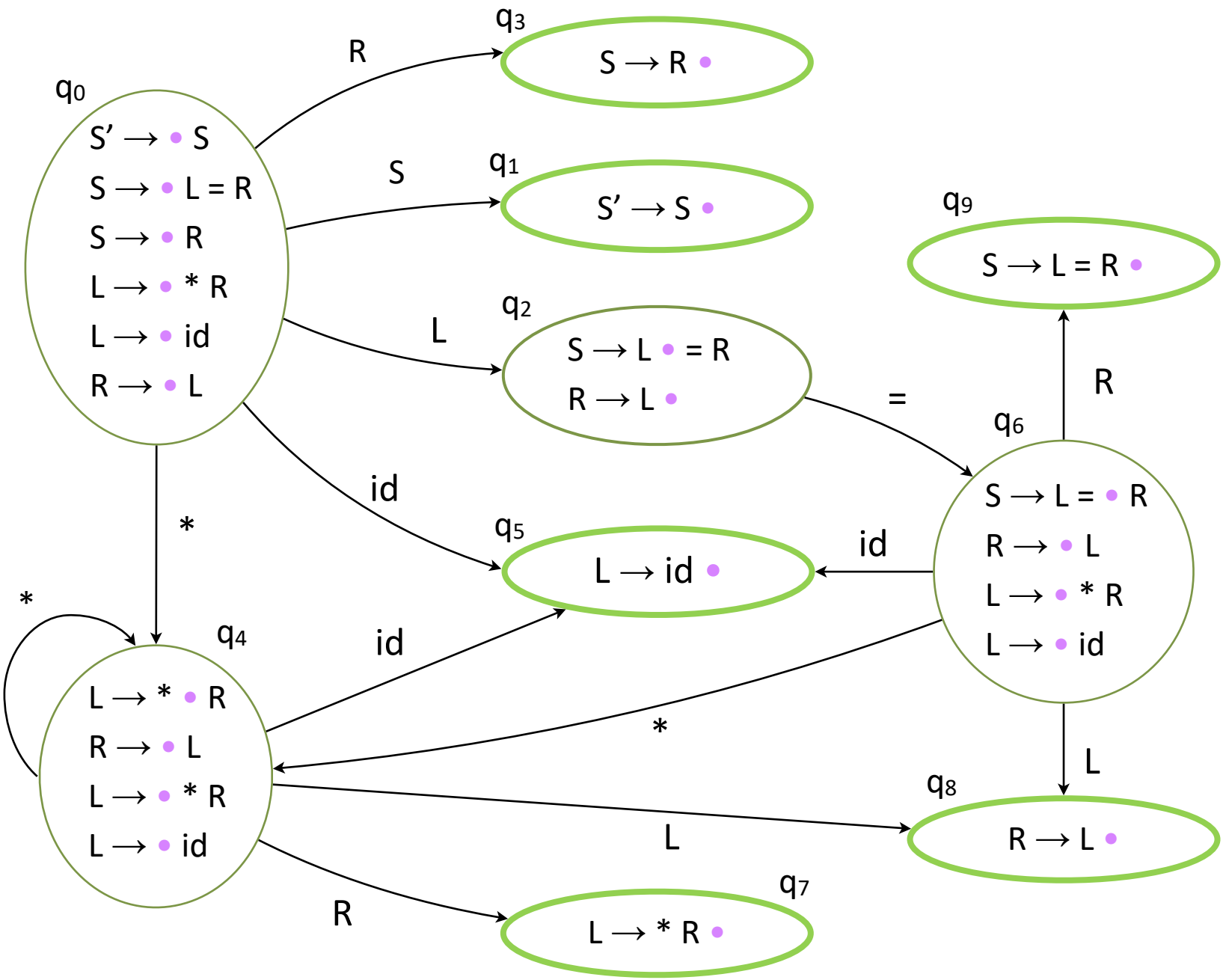


- But cannot resolve all conflicts.

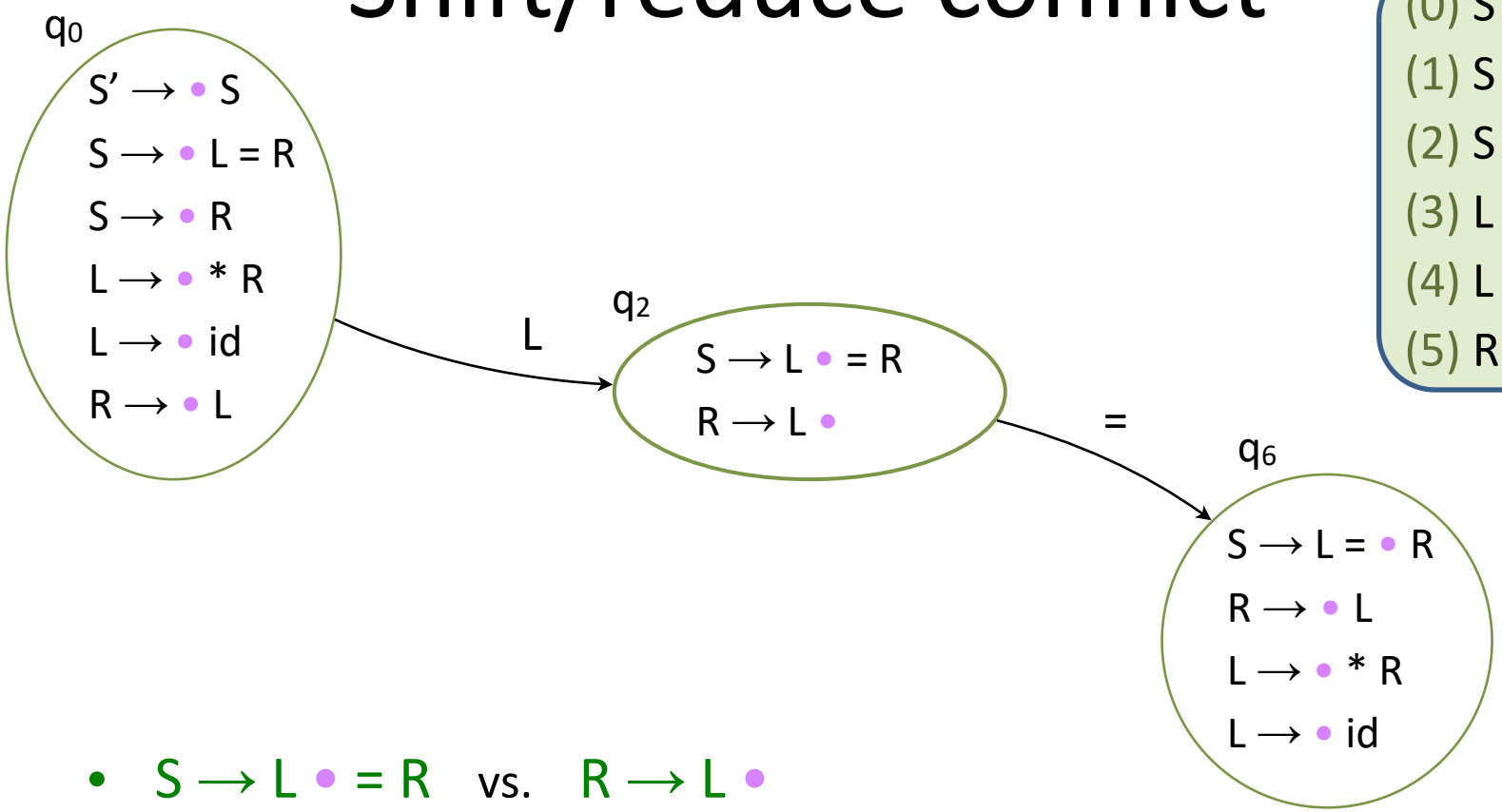
Z → E $
E → T
E → V + T
T → i
V → i
T → (E)

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


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

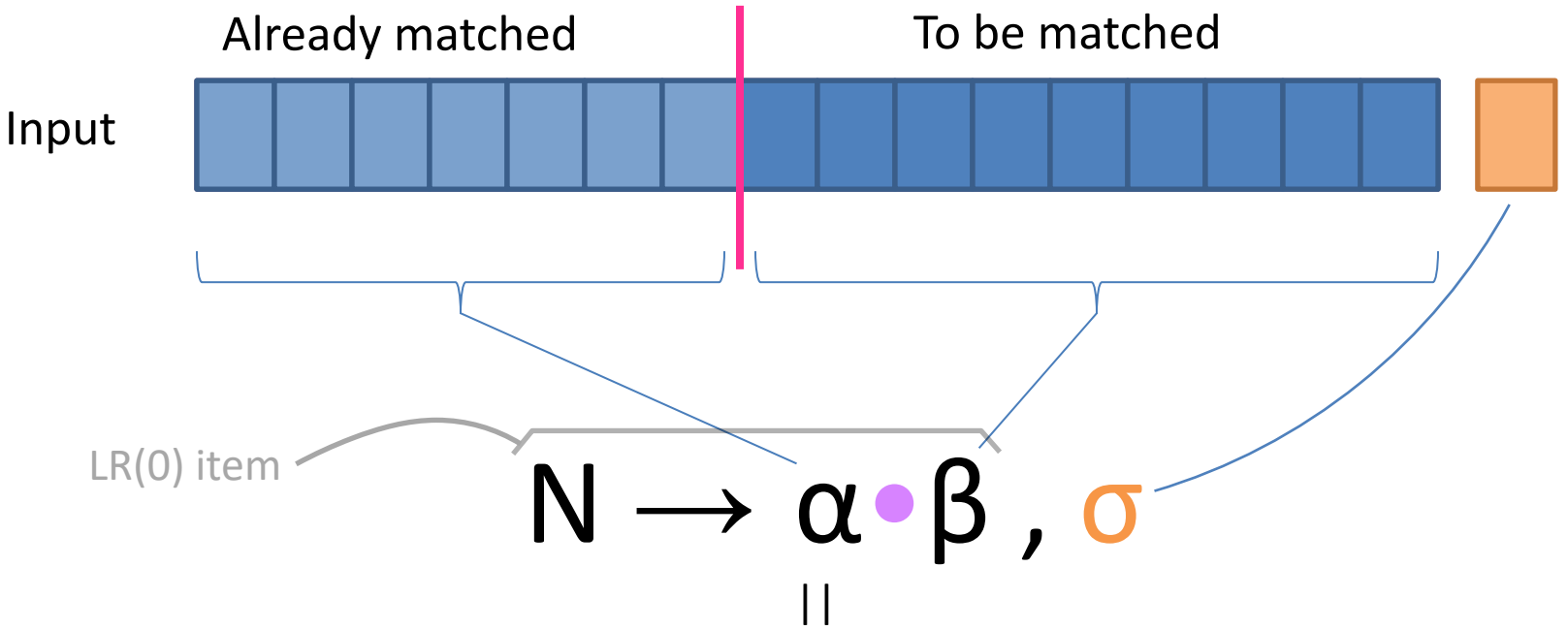
Resolving the Conflict

- In SLR: a reduce item $N \rightarrow \alpha \bullet$ is applicable when the lookahead is in FOLLOW(N).
- But there is a whole **sentential form** that we have discovered so far.
- We can ask what the next token may be given all **previous reductions**
- For example, even looking at the FOLLOW of the entire sentential form is more restrictive than looking at the FOLLOW of the last variable.
- In a way, FOLLOW(N) merges look-ahead for all possible occurrences of N:

$$\text{FOLLOW}(\sigma N) \subseteq \text{FOLLOW}(N)$$

- LR(1) keeps look-ahead with **each** LR item

LR(1) Item



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

followed by the lookahead σ

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

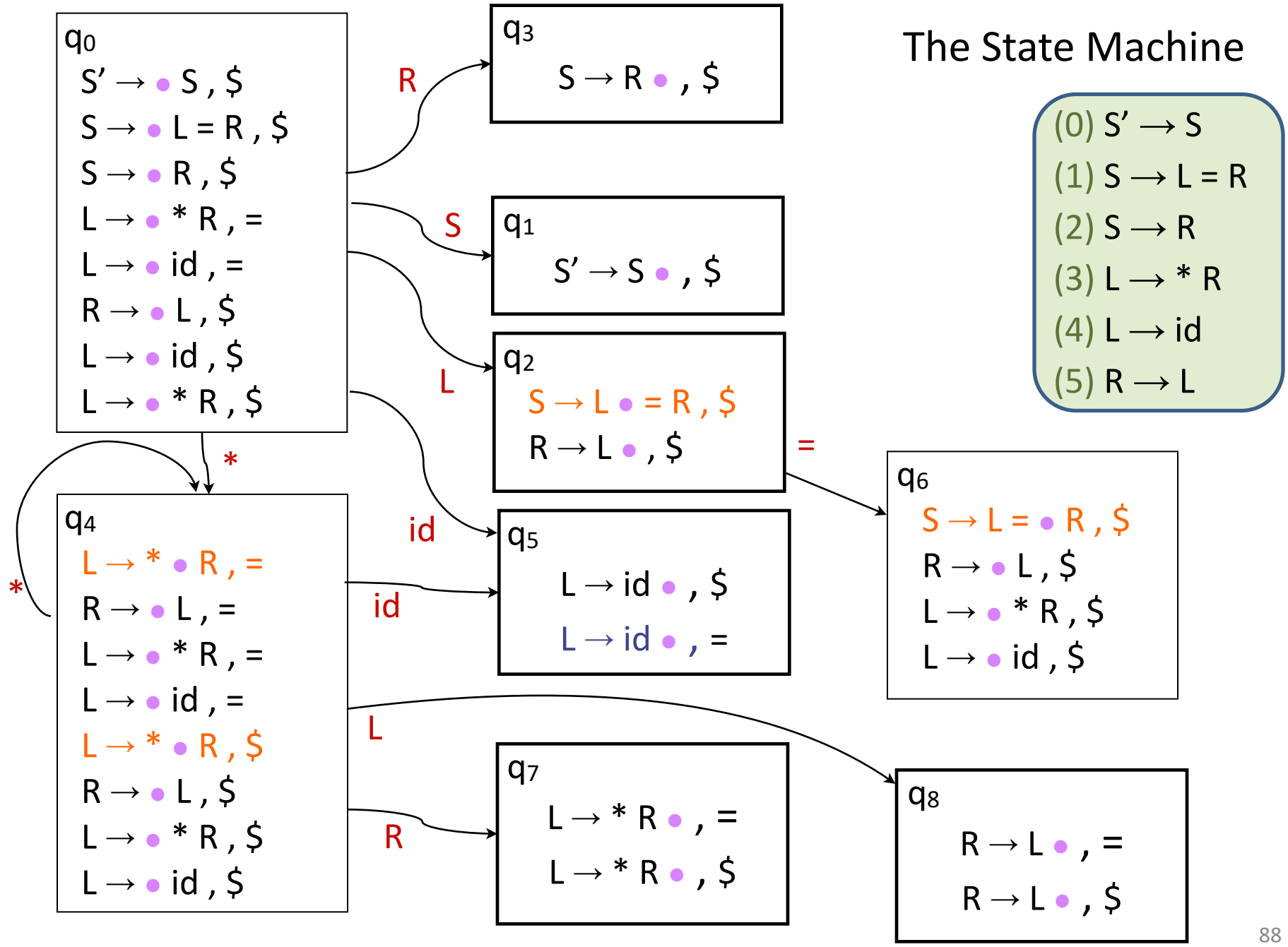
  $$B \rightarrow \bullet \delta,\ d$$

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

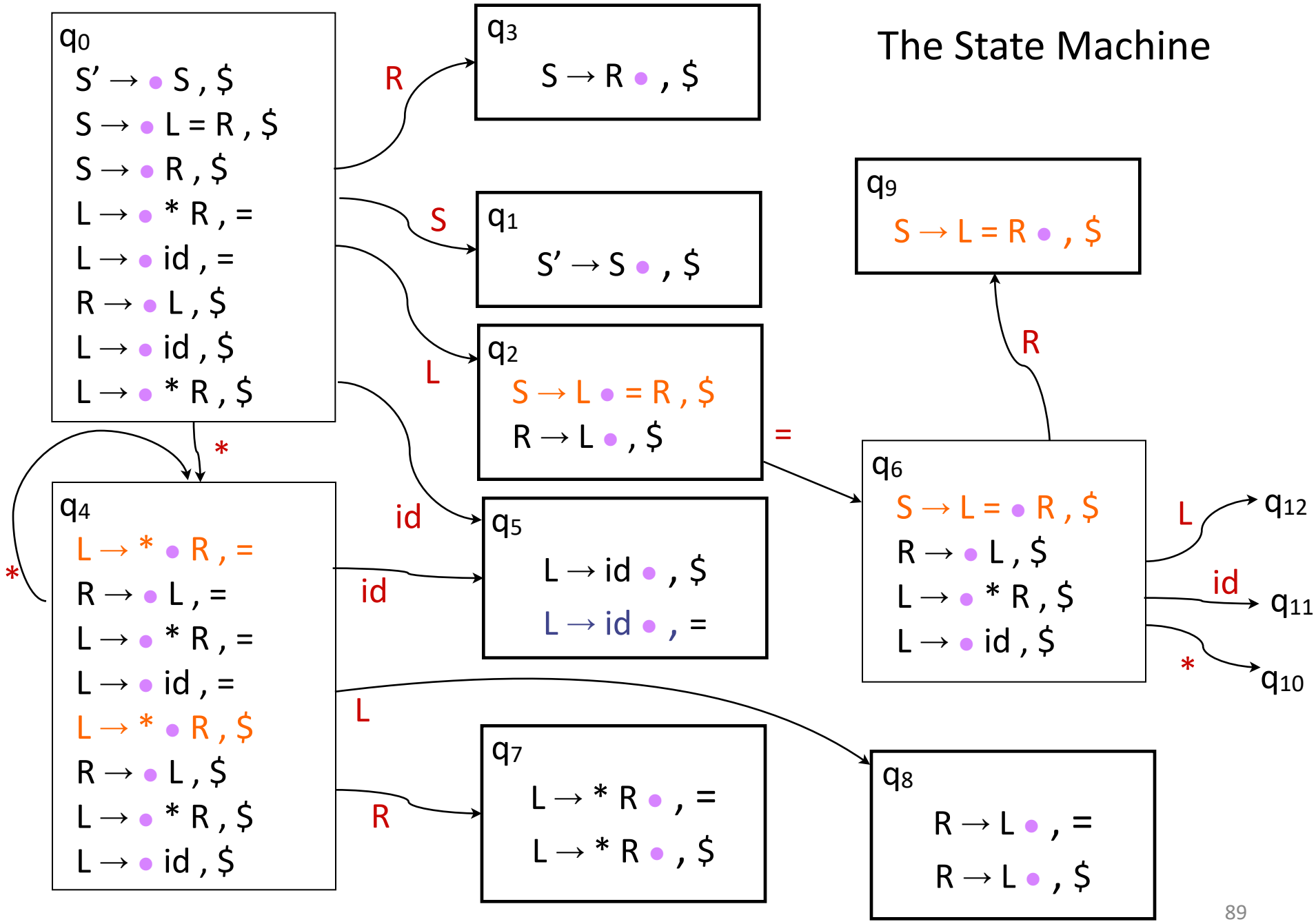
Closure of {S' → ●S , $}

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

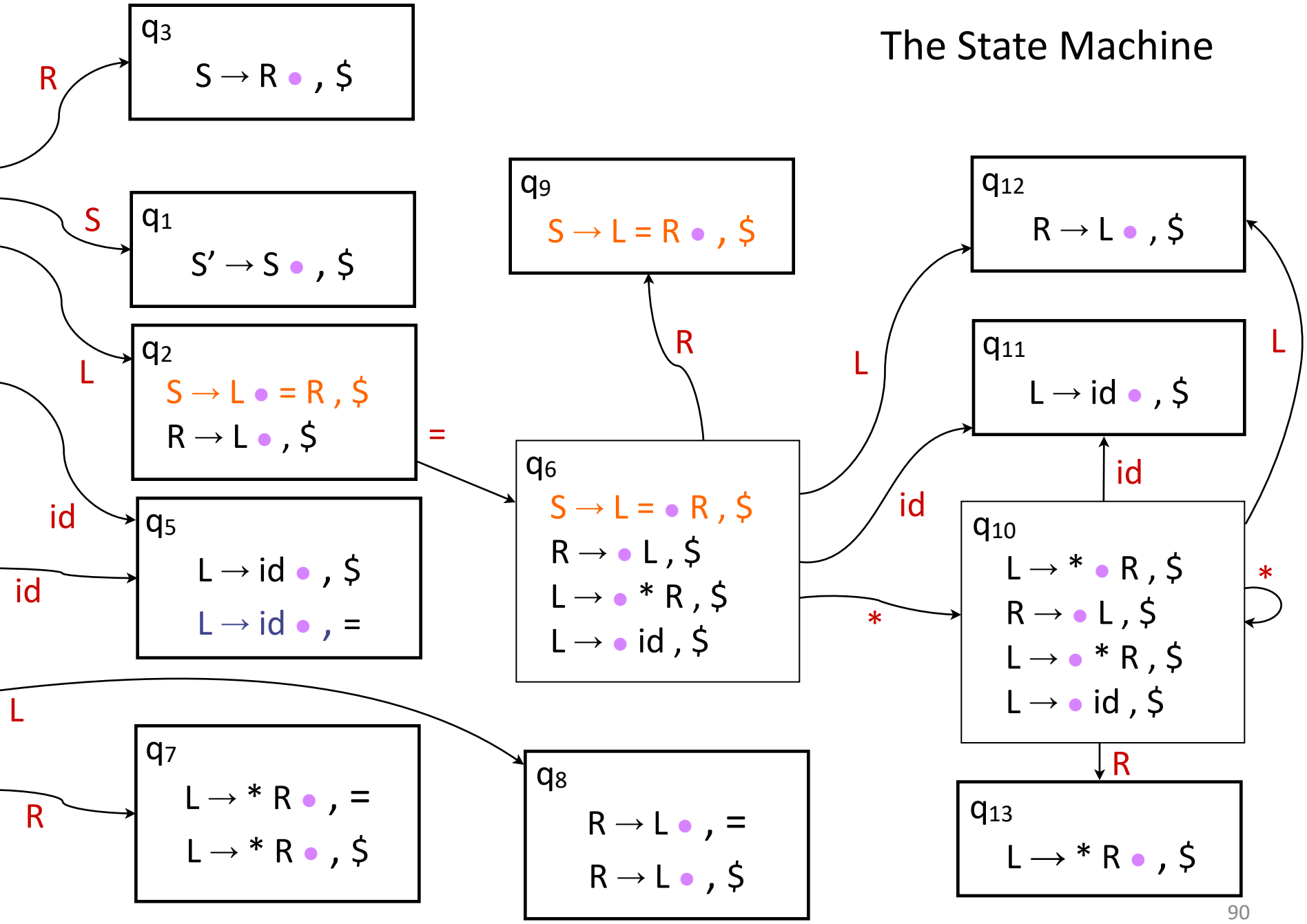
- We would like to add rules that start with S, but keep track of possible lookahead.
  - S' → ● S , $
  - S → ● L = R , $ – Rules for S
  - S → ● R , $
  - L → ● * R , = – Rules for L
  - L → ● id , =
  - R → ● L , $ – Rules for R
  - L → ● * R , $ – More rules for L
  - L → ● id , $


The State Machine
(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L
q0
S' → • S , $
S → • L = R , $
S → • R , $
L → • * R , =
L → • id , =
R → • L , $
L → • id , $
L → • * R , $
q1
S' → S • , $
q2
S → L • = R , $
R → L • , $
q3
S → R • , $
q4
L → * • R , =
R → • L , =
L → • * R , =
L → • id , =
L → * • R , $
R → • L , $
L → • * R , $
L → • id , $
q5
L → id • , $
L → id • , =
q6
S → L = • R , $
R → • L , $
L → • * R , $
L → • id , $
q7
L → * R • , =
L → * R • , $
q8
R → L • , =
R → L • , $
R
S
L
=
*
id
*
id
L
R

The State Machine

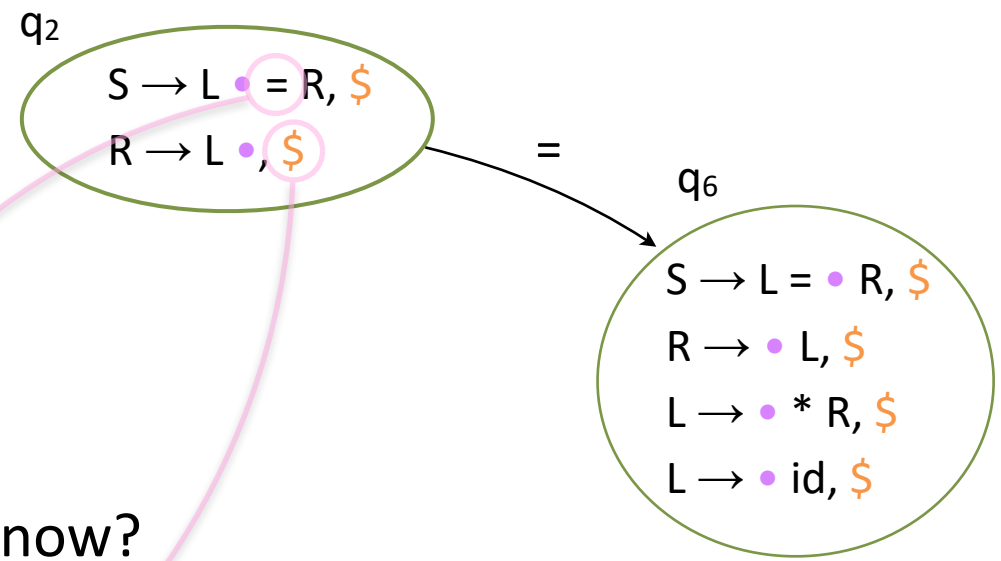


The State Machine



Back to the conflict

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



- 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.

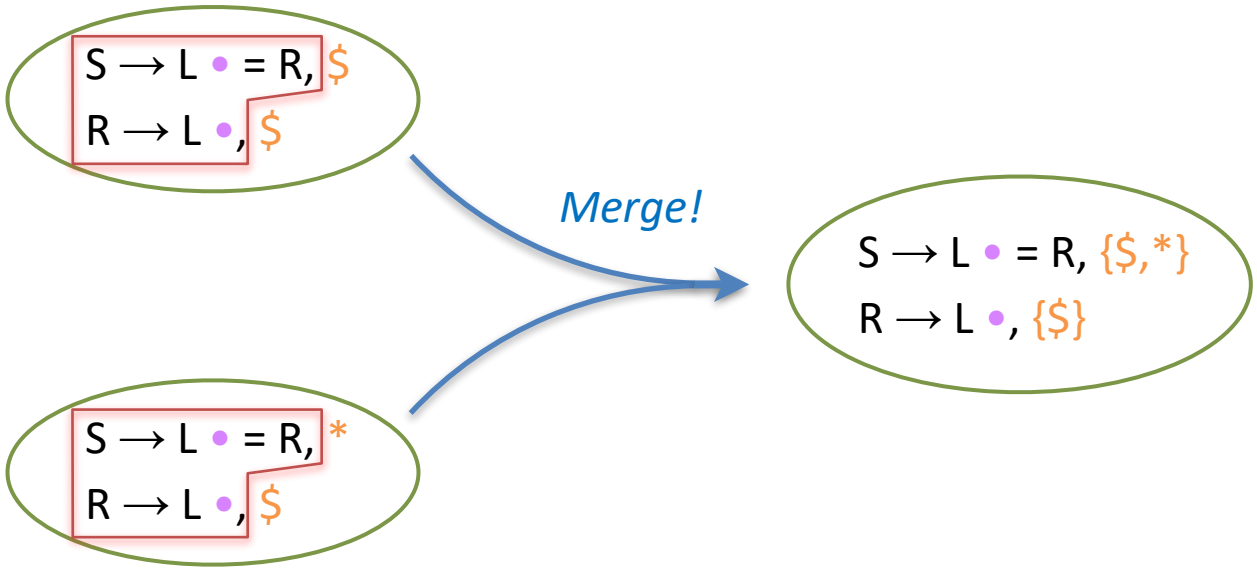
Building the Table

	ACTION				GOTO		
	id	*	=	$	S	R	L
q_0	s5	s4			1	3	2
q_1				acc			
q_2			s6	r5			
q_3				r2			
q_4	s5	s4				7	8
q_5			r4	r4			
q_6	s11	s10				9	12
q_7			r3	r3			
q_8			r5	r5			
q_9				r1			
q_{10}	s11	s10				13	12
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

One last thing: 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, representing the lookaheads as *sets*.



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

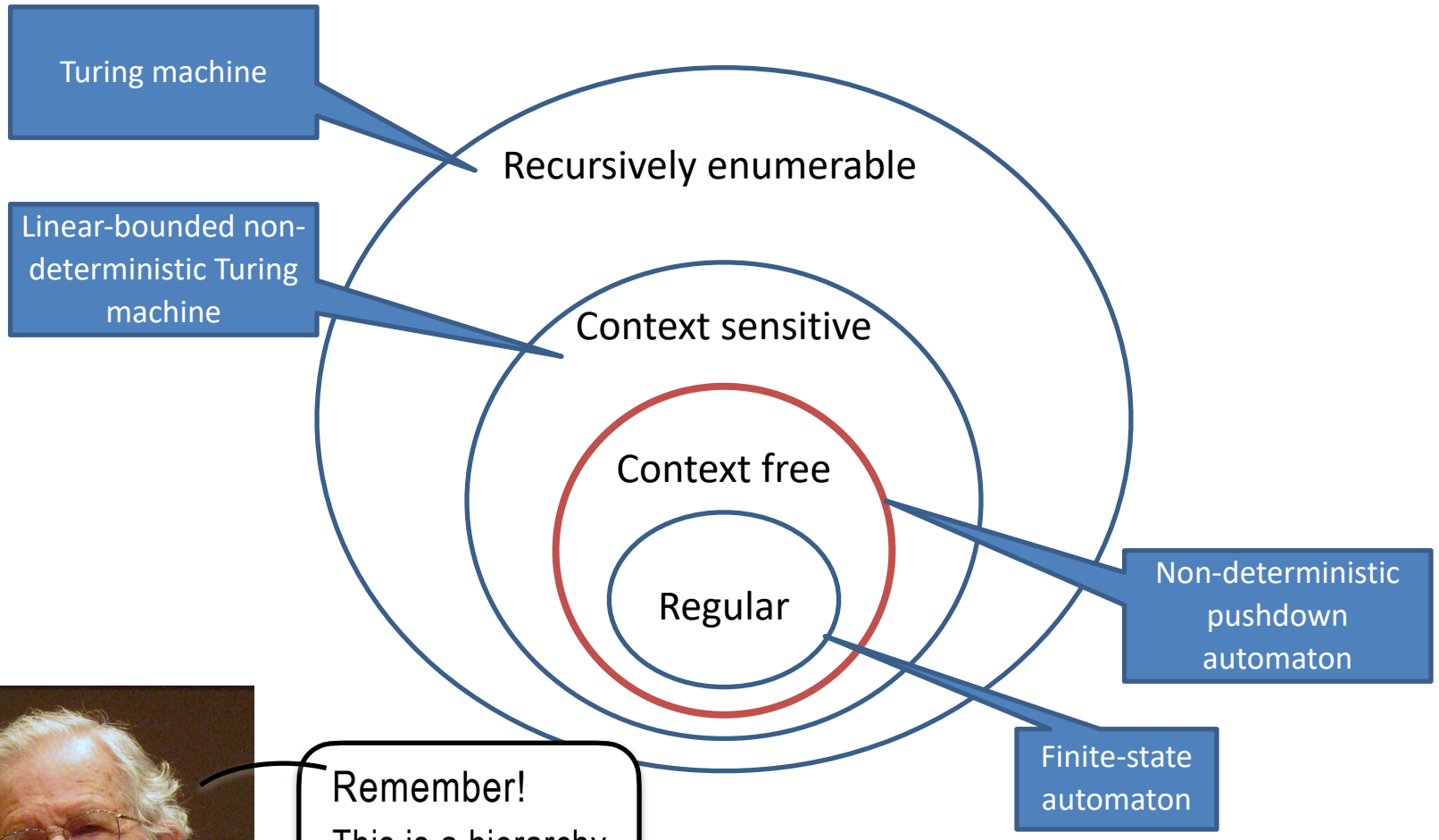
Bottom-up Parsing

- LR(k)
- SLR
- LALR

- All follow the same pushdown-based algorithm
- Differ on type of "LR Items"

LR(0) SLR(1)	LR(1)	LALR
$N \rightarrow \alpha \bullet \beta$	$N \rightarrow \alpha \bullet \beta , \sigma$	$N \rightarrow \alpha \bullet \beta , \{\sigma,\ldots\}$

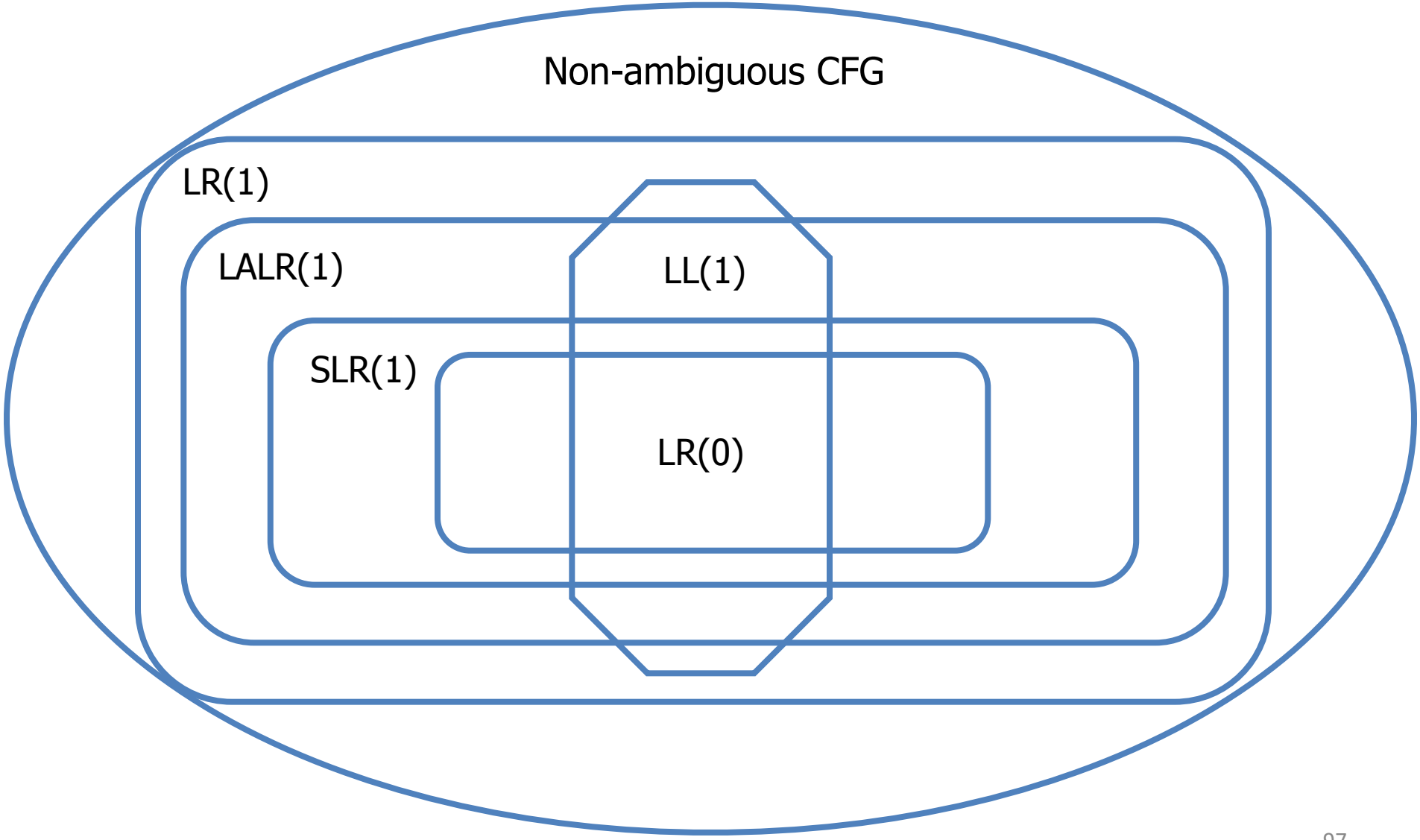
Chomsky Hierarchy



Remember!

This is a hierarchy of *languages*.

Grammar Hierarchy

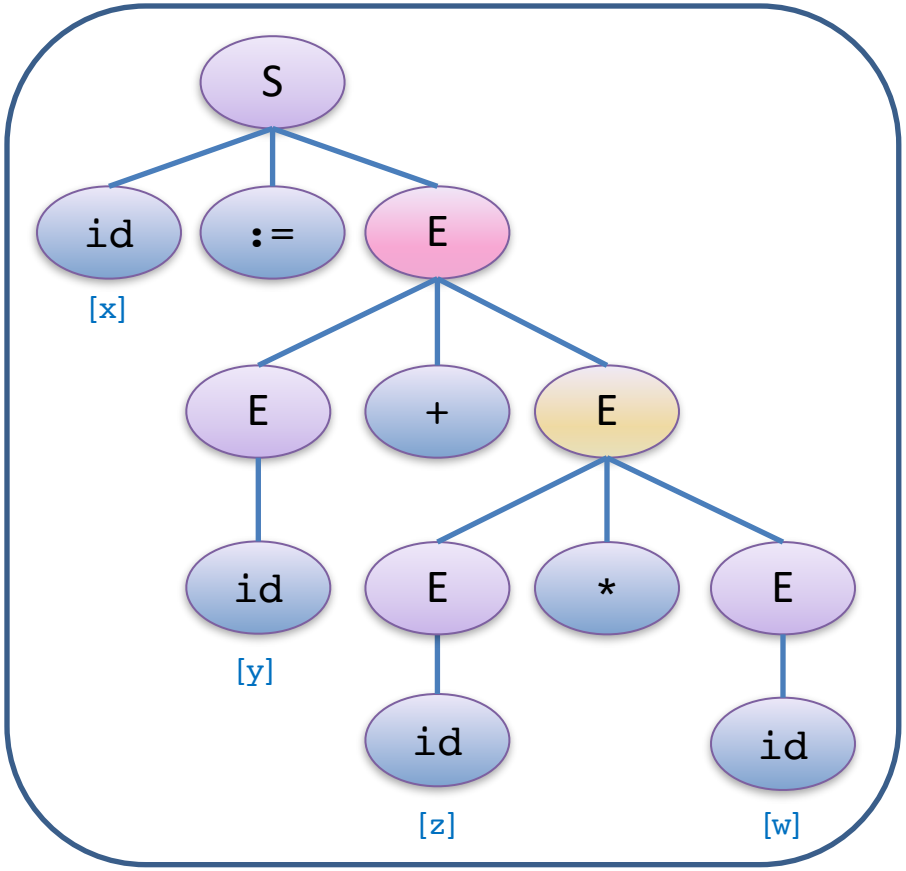


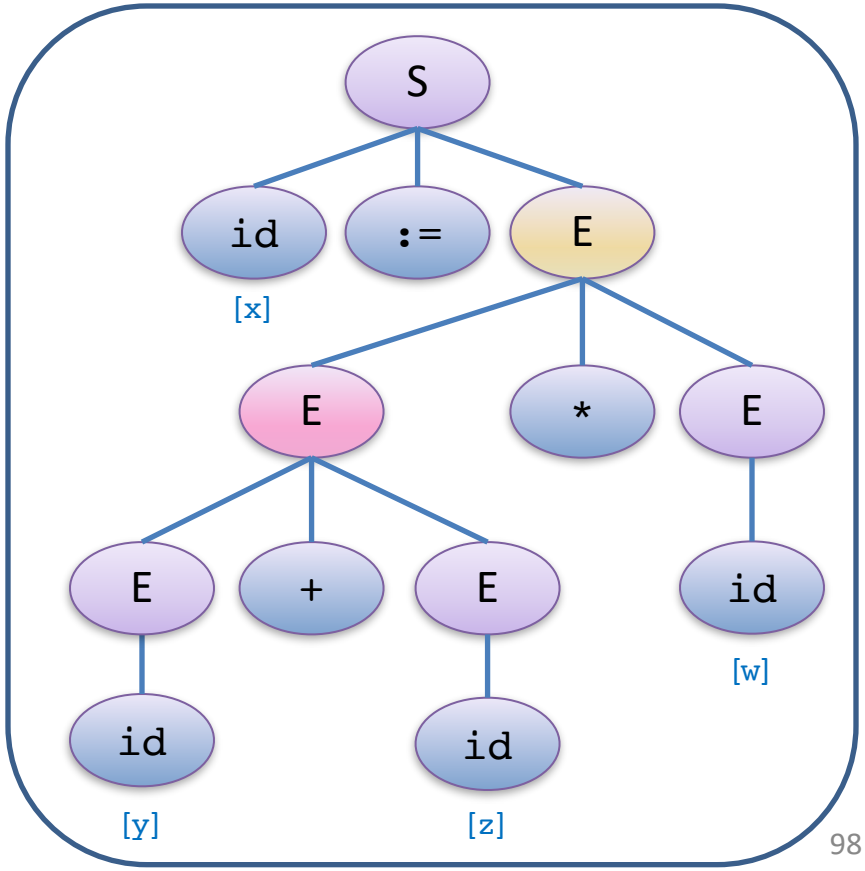
this is the same slide from lecture 3

Ambiguity

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

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





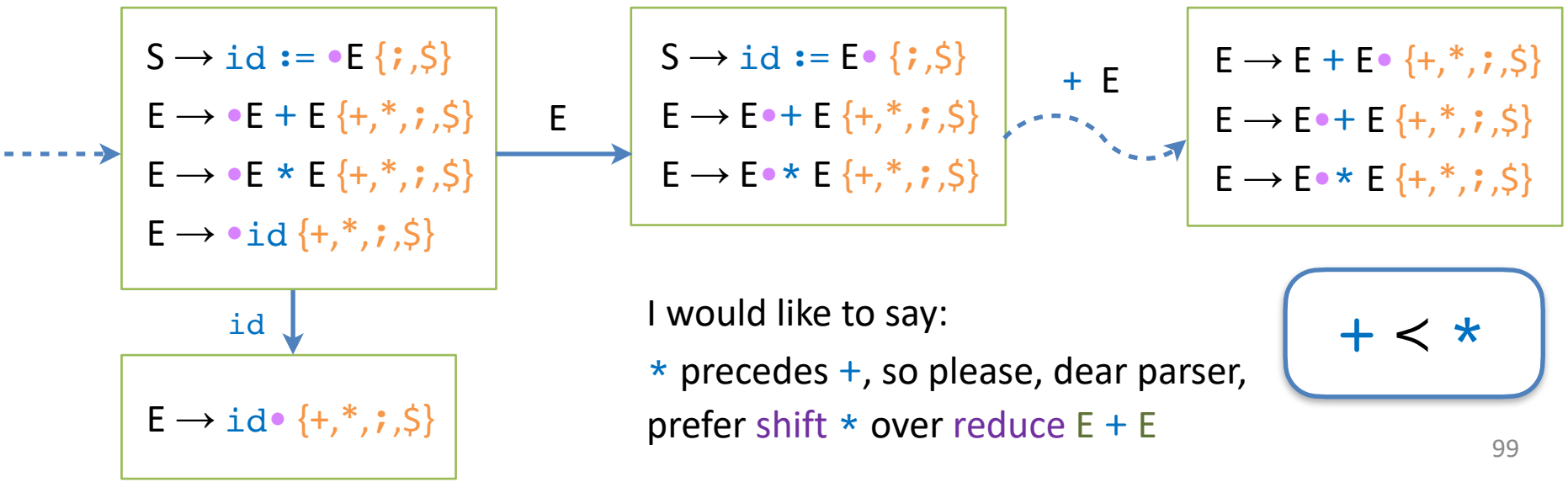
~~Ambiguity~~ Precedence

```
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

+ < *

Bison, an LR parser generator

- Input: a syntax definition file (essentially CFG)
- Output: LR table + a function `yyparse` that runs the PDA

```
/* appa.y */
%%
statement: expr '!' ;

expr: term '+' expr
    | term
    ;

term: id | num ;
%%
```

- ✴ Implements LALR — the lighter variant of LR(1)
- ✴ Offers *conflict resolution* in the form of operator precedence

  (to be covered in the tutorial)

Building the Parse Tree

- Done at the time of reduce.

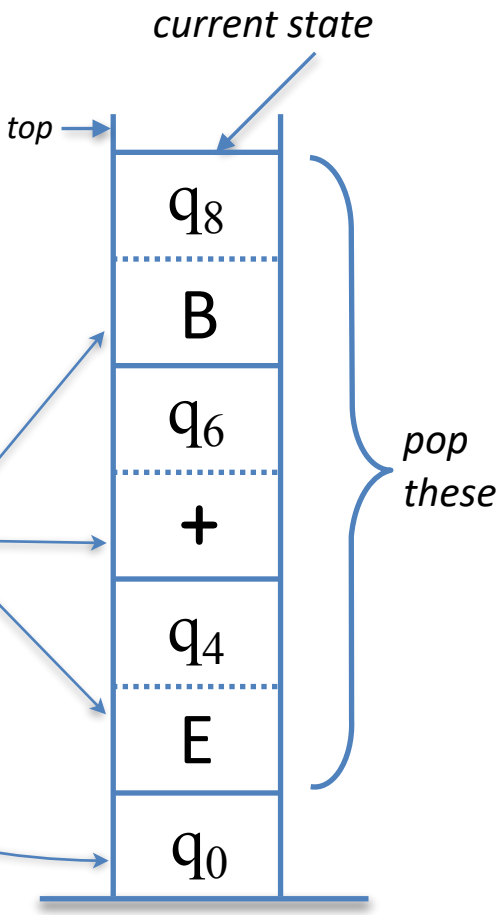
State	ACTION					GOTO	
	*	+	0	1	$	E	T
q_8	r2	r2			r2		

(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-1], "E"];
  push(result);
  push(next);
}
```



Building the Parse Tree

- Done at the time of reduce.

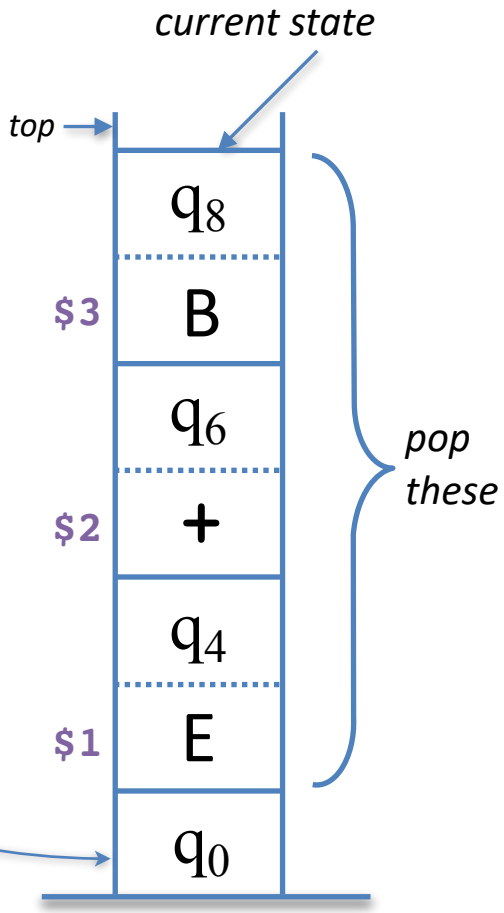
State	ACTION					GOTO	
	*	+	0	1	$	E	T
q_8	r2	r2			r2		

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

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

  pop(6);

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

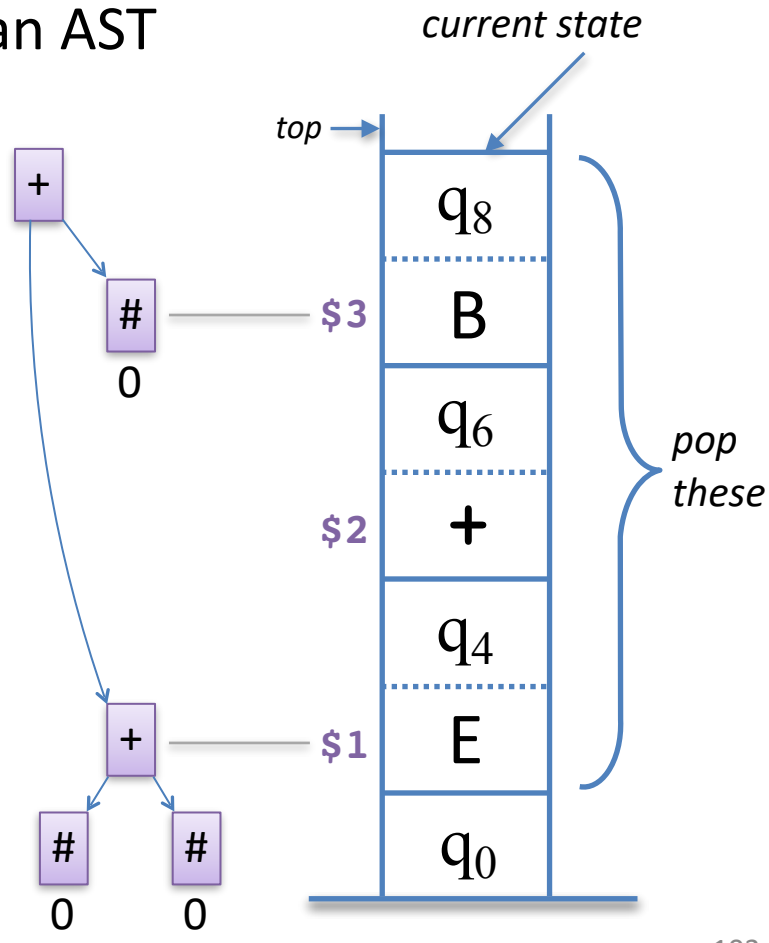


Building the **Abstract** Syntax Tree

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

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

```
E → E + B {
   $$ = new Node("+");
   $$.addChild($1);
   $$.addChild($3);
}
E → B {
   $$ = $1;
}
B → 0 {
   $$ = new Node("#");
   $$.value = 0;
}
```



A Note About 

- ANTLR = ANother Tool for Language Recognition
- LL(*) algorithm
  - Like LL(*k*) on steroids
  - Notable extensions:
    - Repeat operators — like in regex
    - Lookahead predicates — allow for unbounded lookahead at the cost of backtracking

✶ There's a nice demo at lab.antlr.org

* *LL(*): The Foundation of the ANTLR Parser Generator*, Parr and Fischer, PLDI 2011

Summary

- ✓ Bottom up derivation
- ✓ LR(k) can decide on a reduce after seeing the entire right side of the rule plus k look-ahead tokens.
  - ✓ particularly LR(0) – must reduce without lookahead.
- ✓ Using a table and a stack to derive.
- ✓ Definition of LR Items and the automaton.
- ✓ Creating the table from the automaton.
- ✓ LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm
- LALR: in the tutorial.

Next