Compilation

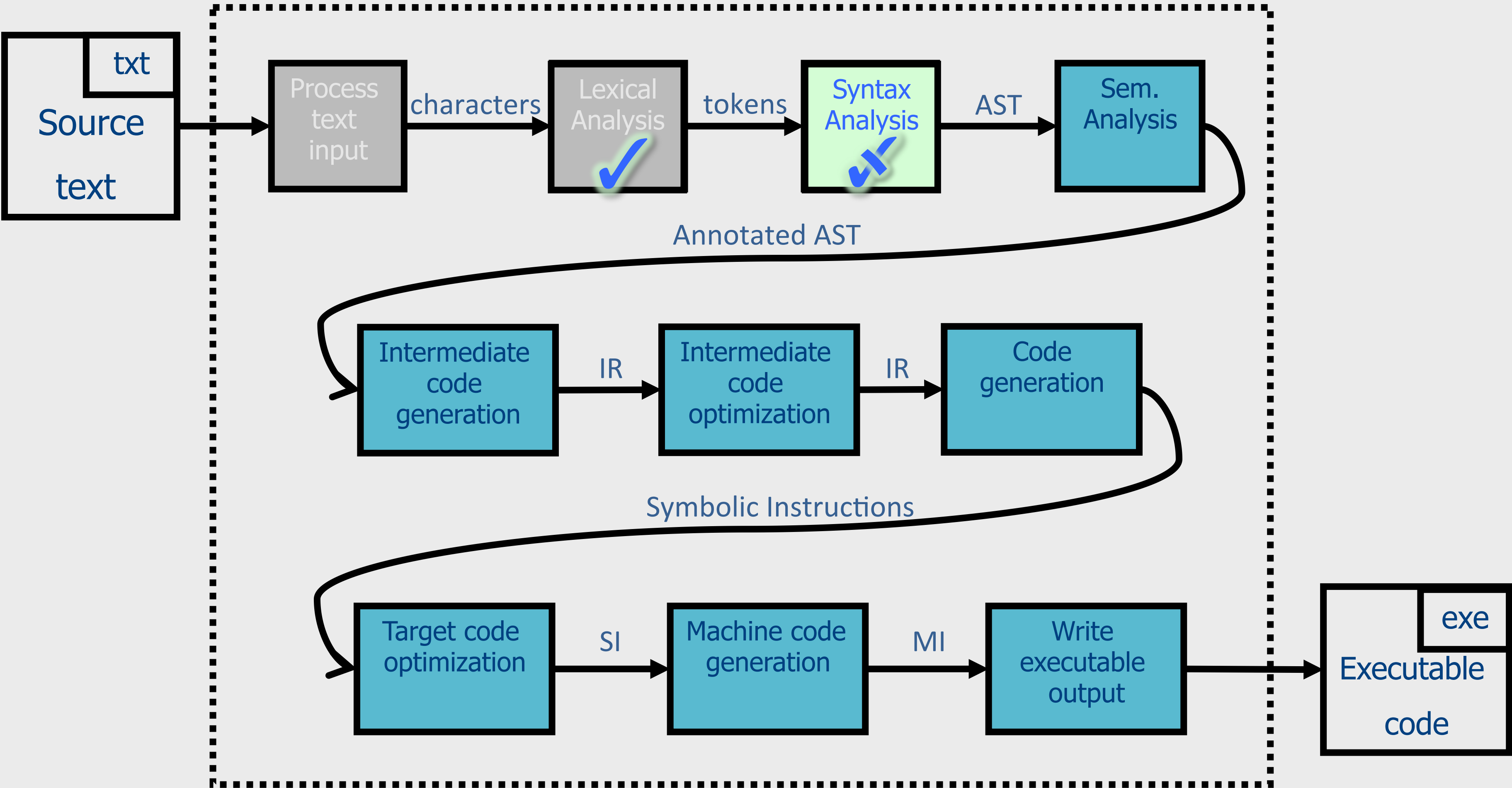
0368-3133 2014/15a

Lecture 5

Syntax Analysis: Bottom-Up parsing

Noam Rinetzky

Where are we?



Bottom-Up vs Top-Down

Top-Down Parsing

Reduction starts by choosing a derivation rule

Reduction is done in context

Bottom-Up vs Top-Down

Top-Down Parsing

Reduction starts by choosing a derivation rule

Reduction is done in context

Expr → Even * Even | Odd – Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9

| 6 * 8

used unused

Bottom-Up vs Top-Down

Top-Down Parsing

Reduction starts by choosing a derivation rule

Reduction is done in context

Expr → Even * Even | Odd – Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9

Expr

| 6 * 8

matched unmatched

Bottom-Up vs Top-Down

Top-Down Parsing

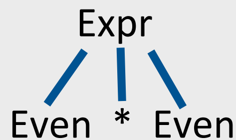
Reduction starts by choosing a derivation rule

Reduction is done in context

Expr → Even * Even | Odd – Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



6 * 8

matched unmatched

Bottom-Up vs Top-Down

Top-Down Parsing

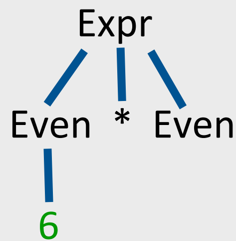
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



6 | * 8

matched unmatched

Bottom-Up vs Top-Down

Top-Down Parsing

Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Bottom-Up Parsing

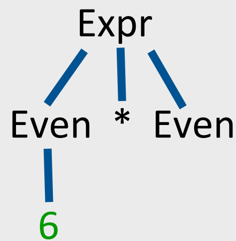
Reduction considers all possible reduction rule**s**

- Rule is determined at the end

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



Bottom-Up vs Top-Down

Top-Down Parsing

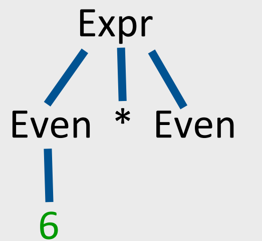
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



6 | * 8

matched unmatched

Bottom-Up Parsing

Reduction considers all possible reduction rule**s**

- Ruleis determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9

| 6 * 8

matched unmatched

Bottom-Up vs Top-Down

Top-Down Parsing

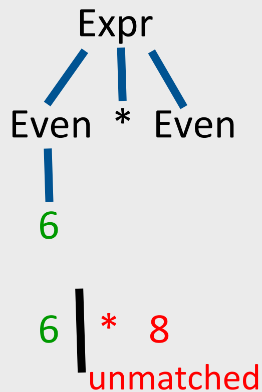
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



matched

Bottom-Up Parsing

Reduction considers all possible reduction rule**s**

- Ruleis determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9

Expr → Num * Num | Num + Num
Num → 0 | 1 | ...

Bottom-Up vs Top-Down

Top-Down Parsing

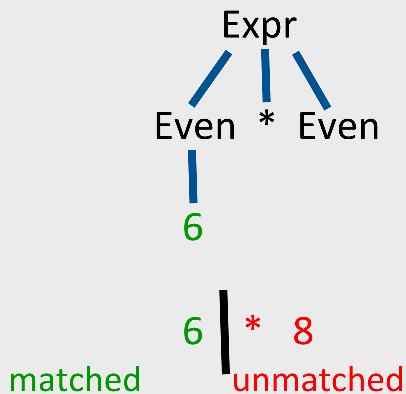
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



Bottom-Up Parsing

Reduction considers all possible reduction rules

- Ruleis determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9

Expr → Num * Num | Num + Num
Num → 0 | 1 | …

Bottom-Up vs Top-Down

Top-Down Parsing

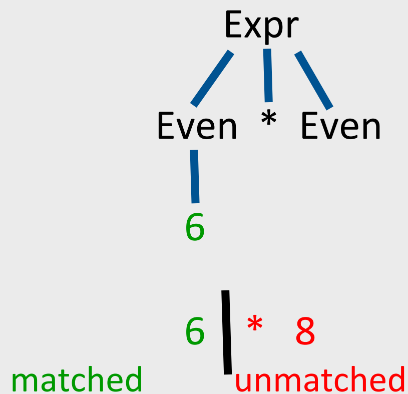
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



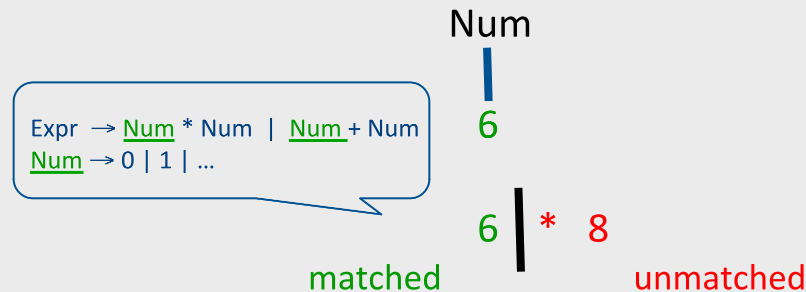
Bottom-Up Parsing

Reduction considers all possible reduction rules

- Ruleis determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9



Bottom-Up vs Top-Down

Top-Down Parsing

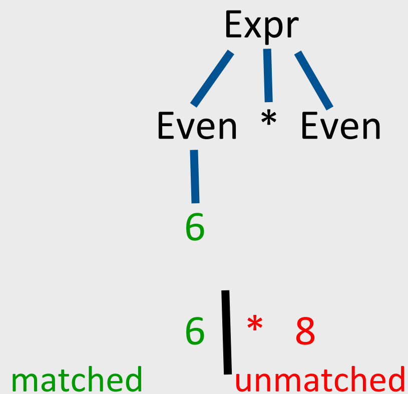
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



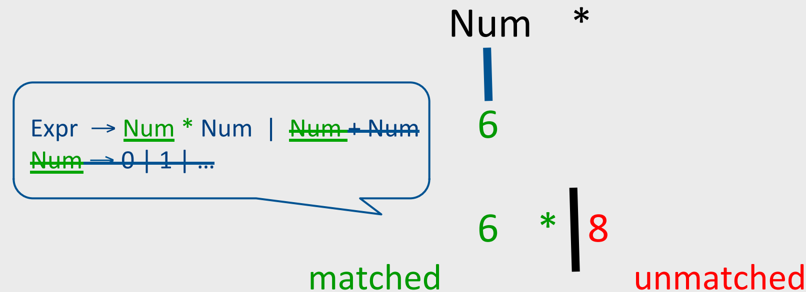
Bottom-Up Parsing

Reduction considers all possible reduction rules

- Ruleis determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9



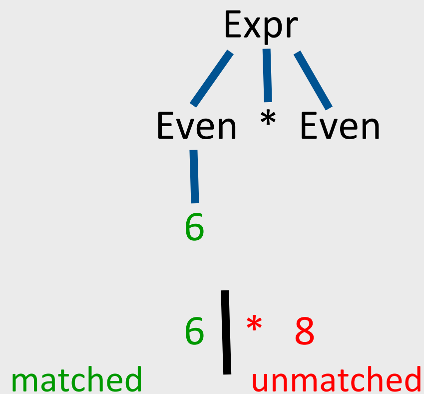
Bottom-Up vs Top-Down

Top-Down Parsing

Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd
Even → 0 | 2 | 4 | 6 | 8
Odd → 1 | 3 | 5 | 7 | 9

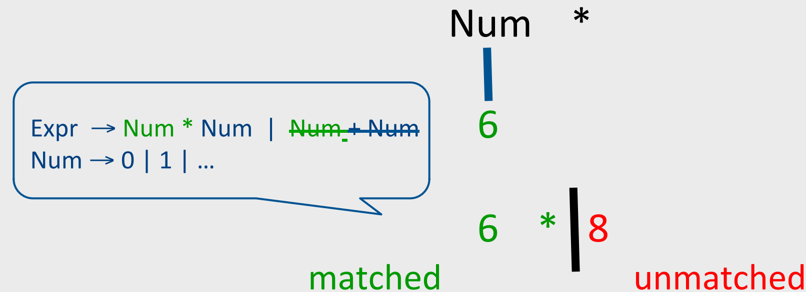


Bottom-Up Parsing

Reduction considers all possible reduction rule**s**

- Ruleis determined at the end

Expr → Num * Num | Num + Num
Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9



Bottom-Up vs Top-Down

Top-Down Parsing

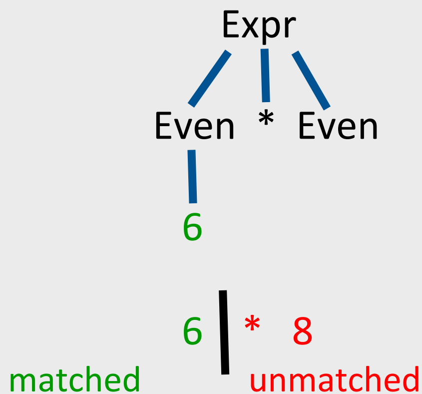
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



Bottom-Up Parsing

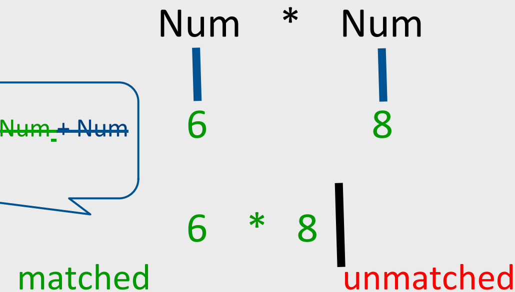
Reduction considers all possible reduction rule**s**

- Rule is determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9

Expr → Num * Num | ~~Num + Num~~
Num → 0 | 1 | …



Bottom-Up vs Top-Down

Top-Down Parsing

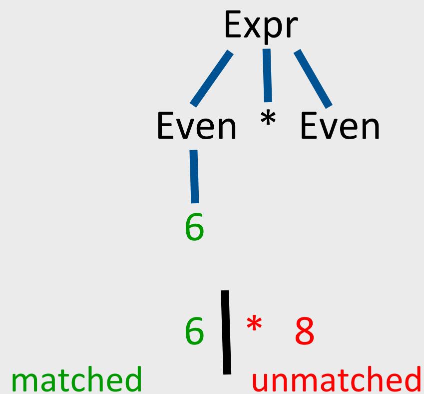
Reduction starts by choosing a derivation rule

- Reduction is done in context of a rule

Expr → Even * Even | Odd + Odd

Even → 0 | 2 | 4 | 6 | 8

Odd → 1 | 3 | 5 | 7 | 9



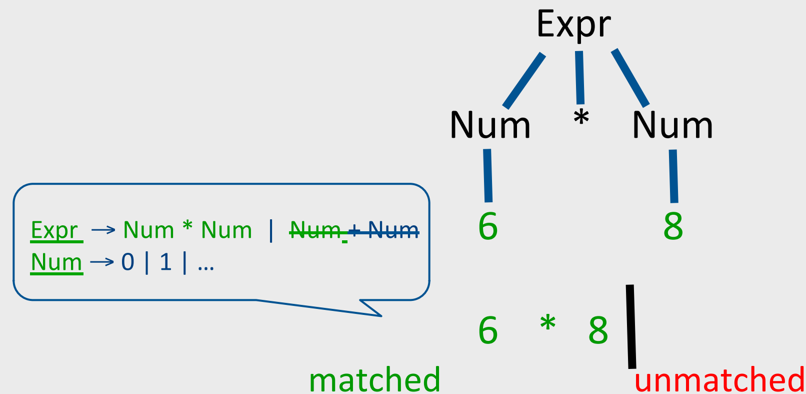
Bottom-Up Parsing

Reduction considers all possible reduction rules

- Rule is determined at the end

Expr → Num * Num | Num + Num

Num → 0 | 1 | 2 | 3| 4 | 5| 6 | 7 | 8 | 9



Bottom-up parsing: LR(k) Grammars

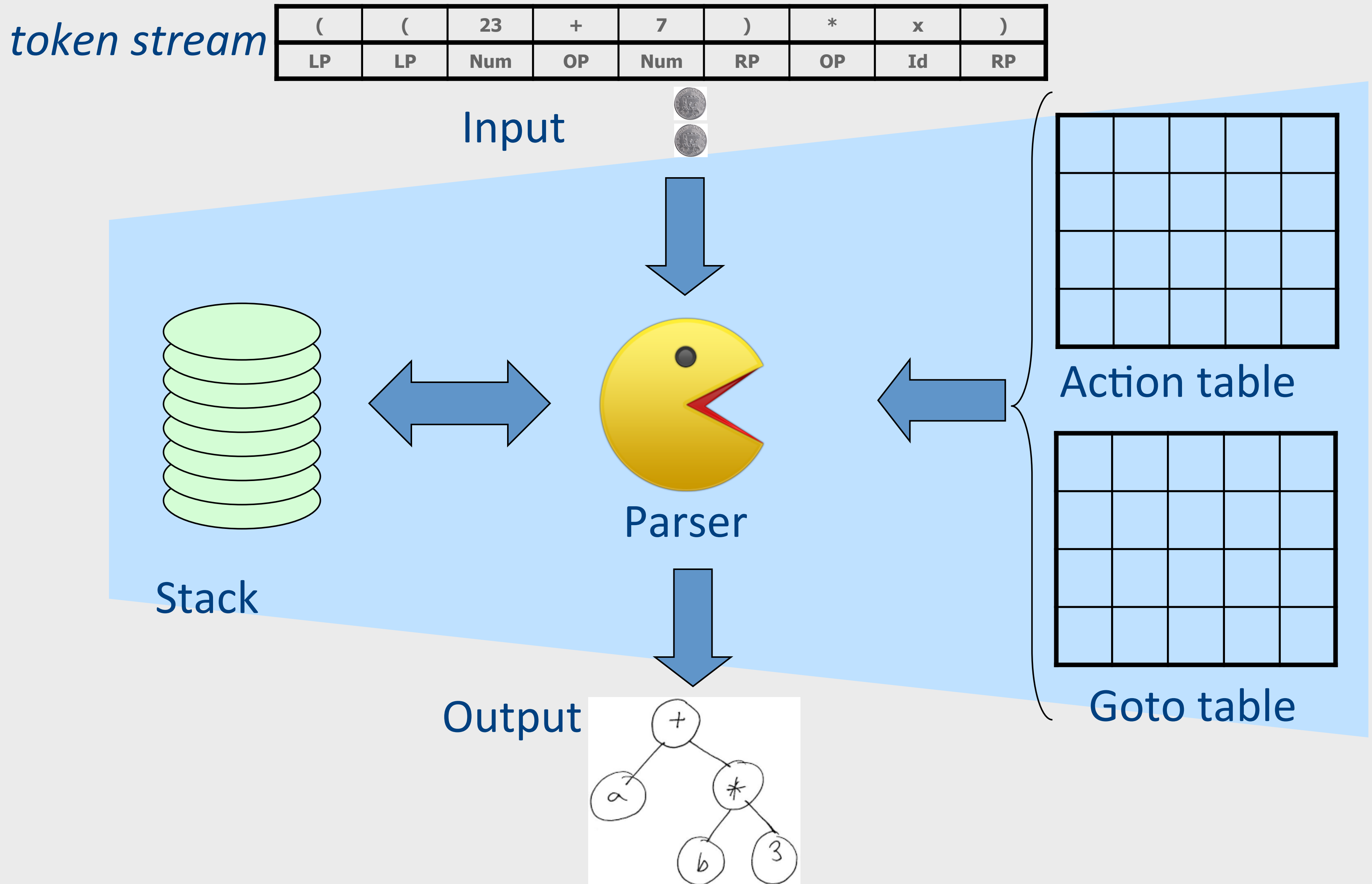
- A grammar is in the class LR(K) when it can be derived via:
  - Bottom-up derivation
  - Scanning the input from left to right (L)
  - Producing the rightmost derivation (R)
  - With lookahead of k tokens (k)

- We will look at LR(0) grammars

Terminology: Reductions & Handles

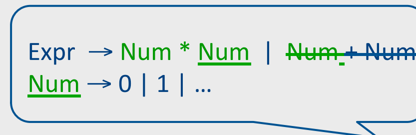
- The opposite of derivation is called *reduction*
  - Let $A \longrightarrow \alpha$ be a production rule
  - Derivation: $\beta A \mu \Rightarrow \beta \alpha \mu$
  - Reduction: $\beta \alpha \mu \Rightarrow \beta A \mu$
- A *handle* is the reduced substring
  - α is the handles for $\beta \alpha \mu$

How does the parser know what to do?



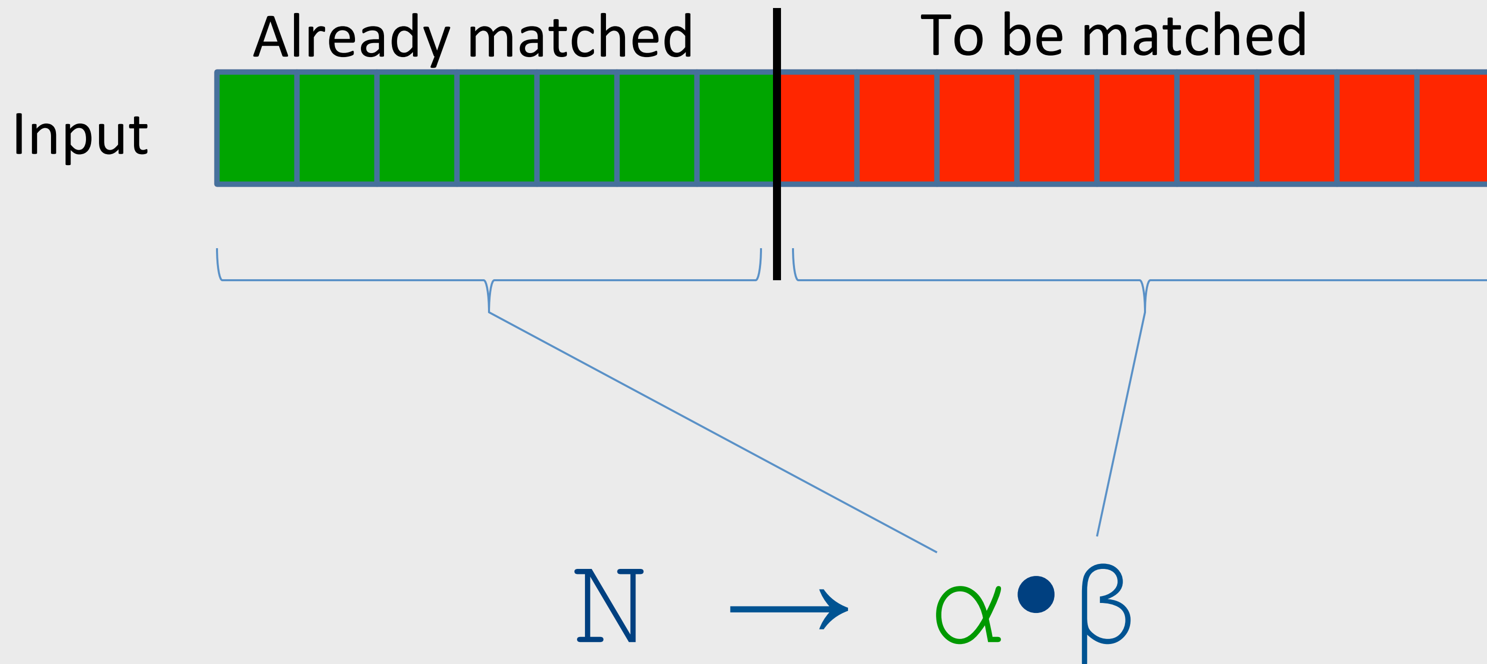
How does the parser know what to do?

- A state will keep the info gathered on handle(s)
  - A state in the "control" of the PDA
    - Also (part of) the stack alphabet
  - State = set of LR(0) items



- A table will tell it "what to do" based on current state and next token
  - The transition function of the PDA
- A stack will records the "nesting level"
  - Prefixes of handles

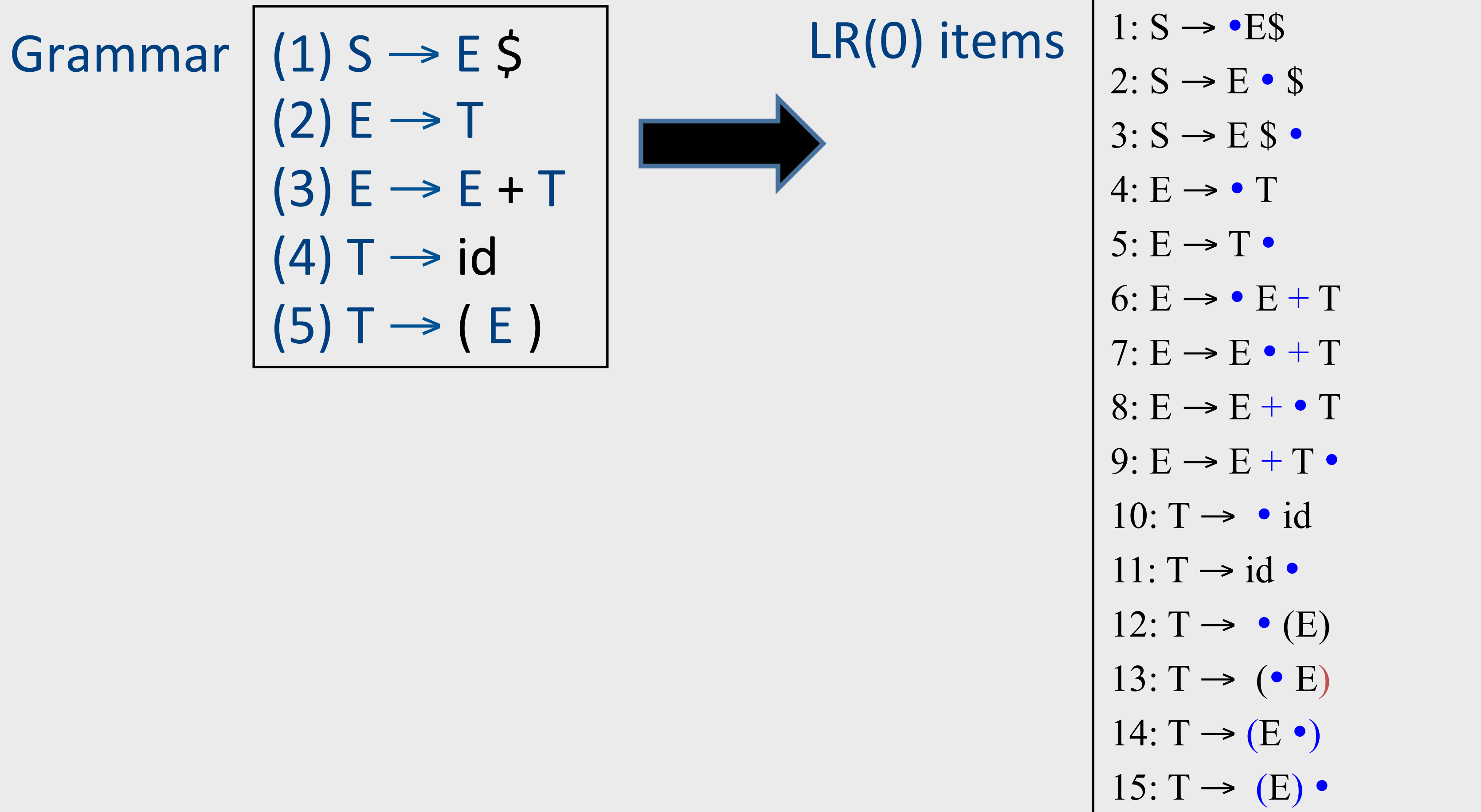
LR item



Hypothesis about αβ being a possible handle, so far we've matched α, expecting to see β

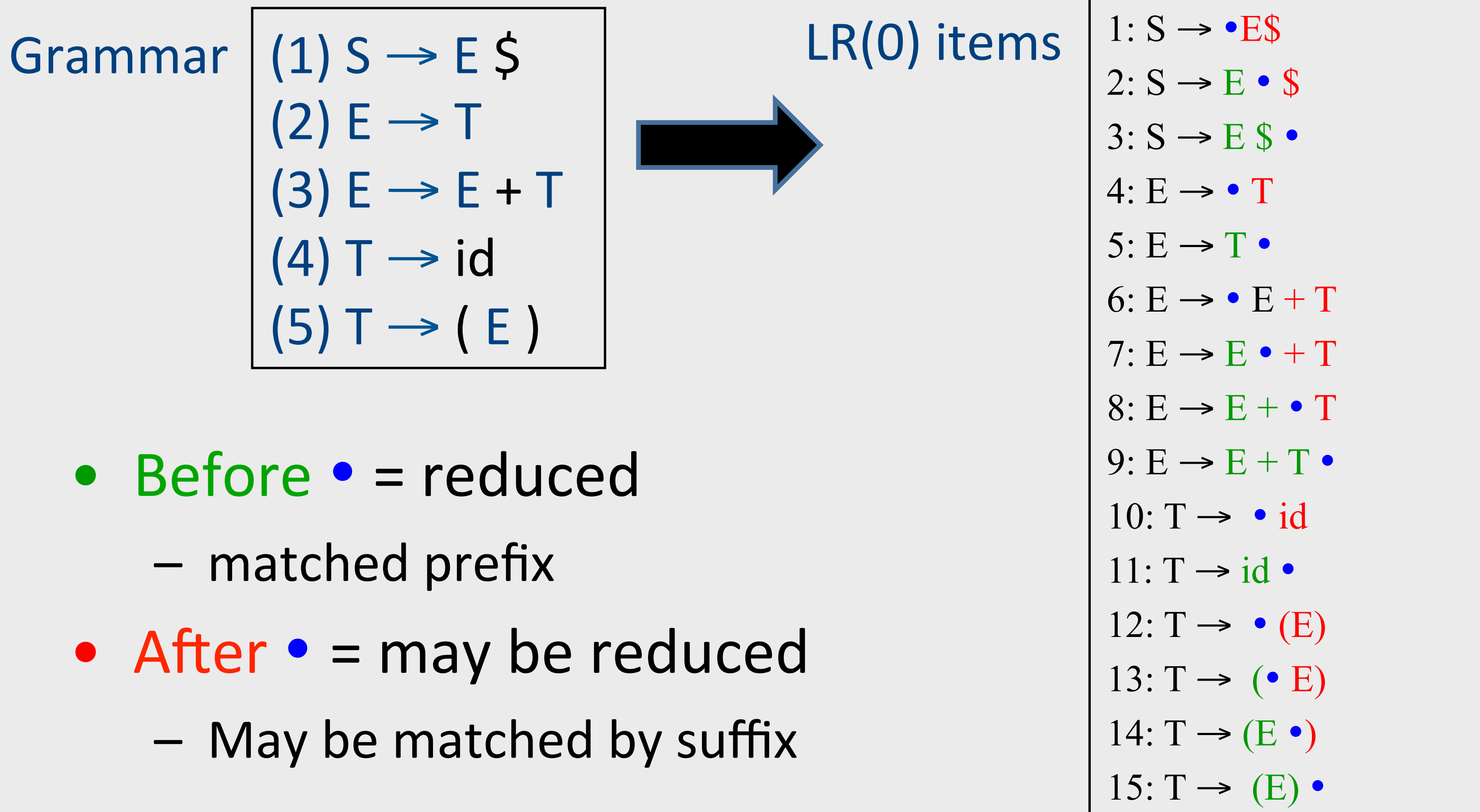
Example: LR(0) Items

- All items can be obtained by placing a dot at every position for every production:



Example: LR(0) Items

- All items can be obtained by placing a dot at every position for every production:



LR(0) Items

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

$B \rightarrow 0 \mid 1$

- A derivation rule with a location marker (●) is called LR(0) item

PDA States

$E \rightarrow E * B \mid E + B \mid B$
$B \rightarrow 0 \mid 1$

- A PDA state is a set of LR(0) items. E.g.,
  $q_{13} = \{ E \rightarrow E \bullet * B,\ E \rightarrow E \bullet + B,\ B \rightarrow 1 \bullet \}$

- Intuitively, if we matched 1,
  Then the state will remember the 3 possible alternatives rules and where we are in each of them

(1) $E \rightarrow E \bullet * B$ (2) $E \rightarrow E \bullet + B$ (3) $B \rightarrow 1 \bullet$

LR(0) Shift/Reduce Items

$N \longrightarrow \alpha \bullet \beta$ Shift Item

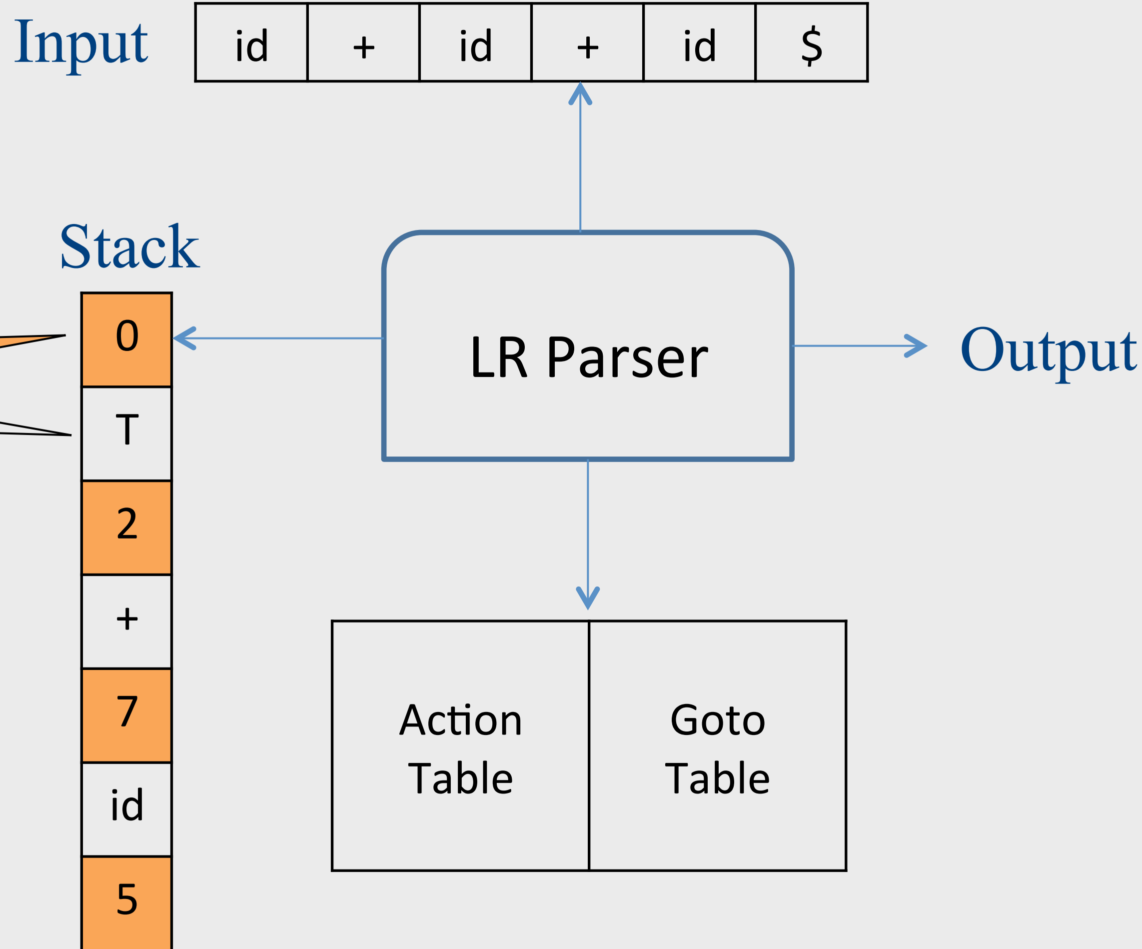
$N \longrightarrow \alpha \beta \bullet$ Reduce Item

Intuition

- Read input tokens left-to-right and remember them in the stack
- When a right hand side of a rule is found, remove it from the stack and replace it with the non-terminal it derives

- Remembering token is called shift
  - Each shift moves to a state that remembers what we've seen so far
- Replacing RHS with LHS is called reduce
  - Each reduce goes to a state that determines the context of the derivation

Model of an LR parser



LR parser stack

- Sequence made of state, symbol pairs
- For instance a possible stack for the grammar

$S \rightarrow E\ \$$

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow id$

$T \rightarrow (E)$

could be: 0 T 2 + 7 id 5

Form of LR parsing table

state	terminals	non-terminals
0	Shift/Reduce actions	Goto part
1	acc	
.	r*k*	g*m*
.	s*n*	
.	error	

shift state *n*

reduce by rule *k*

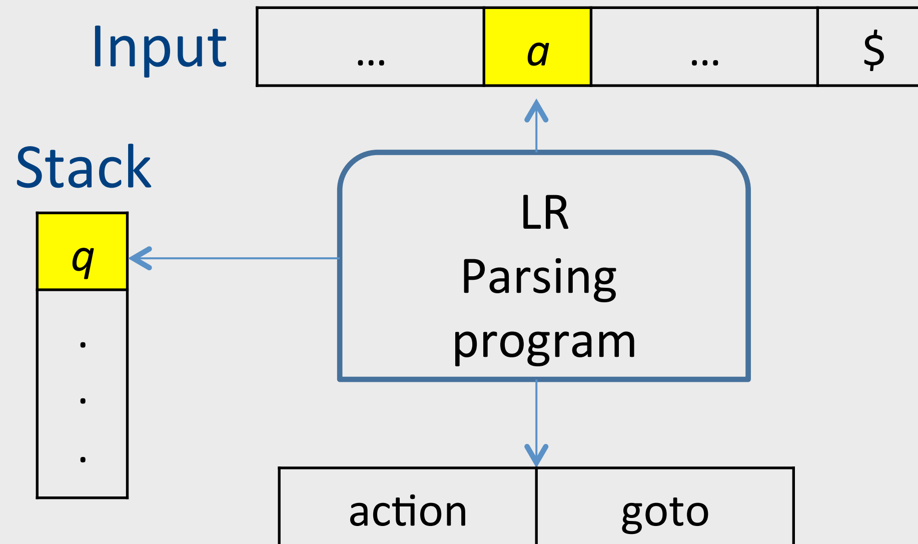
goto state *m*

accept

LR parser table example

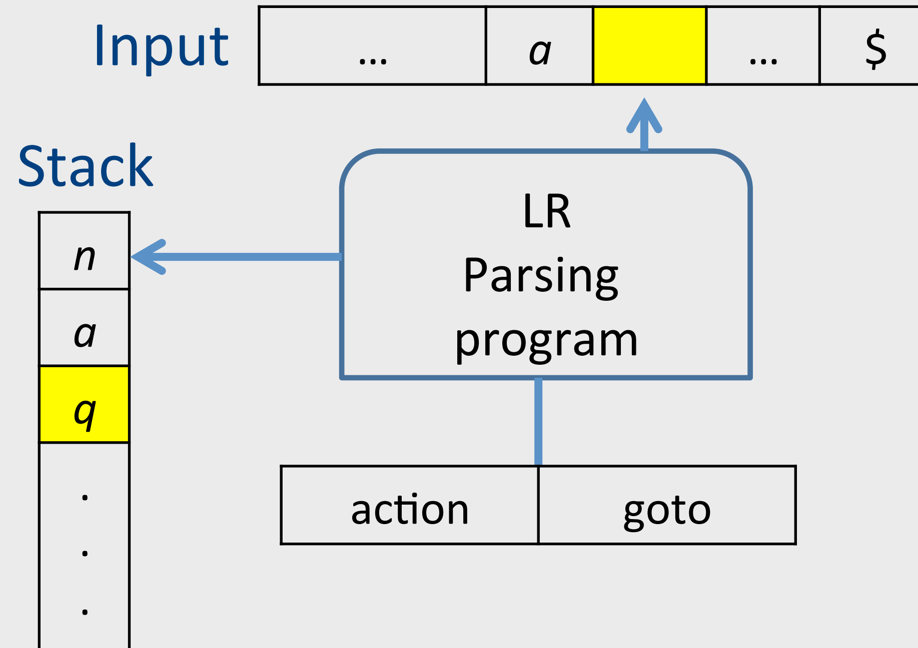
STATE	*action*					*goto*	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

Shift move



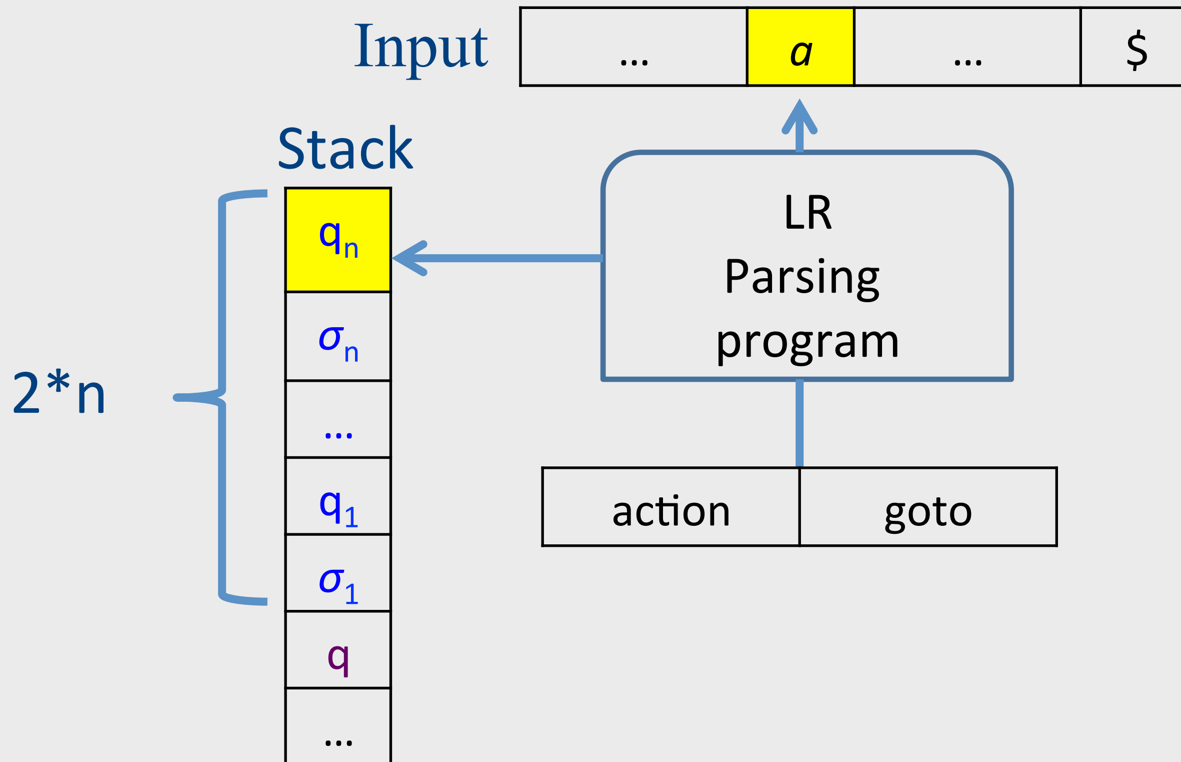
- action[q, a] = sn

Result of shift



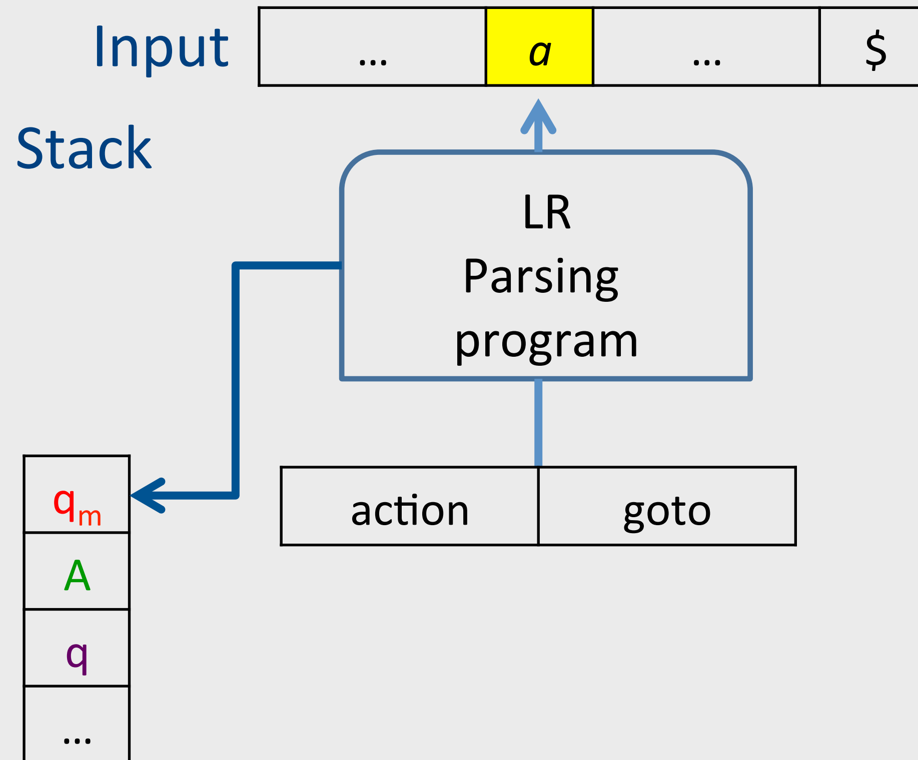
- action[q, a] = sn

Reduce move



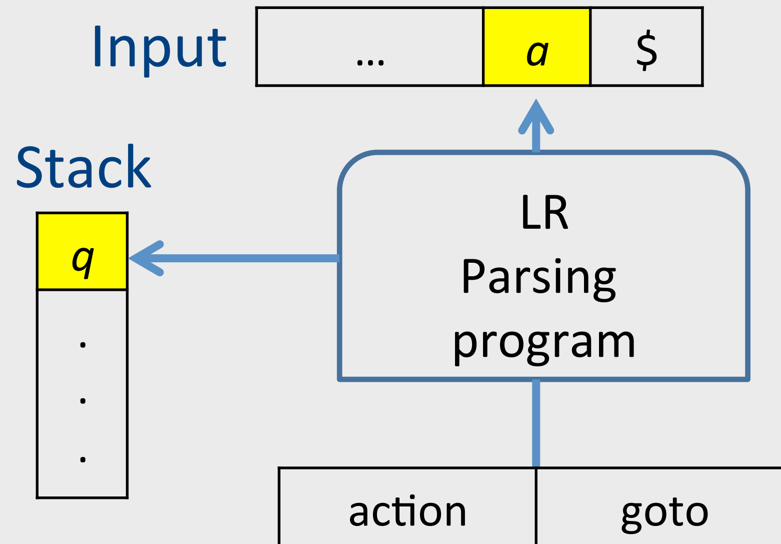
- action[q_n, a] = rk
- Production: (k) $A \longrightarrow \sigma_1 \ldots \sigma_n$
- Top of stack looks like $q_1\sigma_1 \ldots q_n\sigma_n$ for some $q_1 \ldots q_n$
- goto[q, A] = q_m

Result of reduce move



- action[q_n, a] = r*k*
- Production: (k) A ⟶ $\sigma_1 \ldots \sigma_n$
- Top of stack looks like $q_1\sigma_1 \ldots q_n\sigma_n$ for some $q_1 \ldots q_n$
- goto[q, A] = q_m

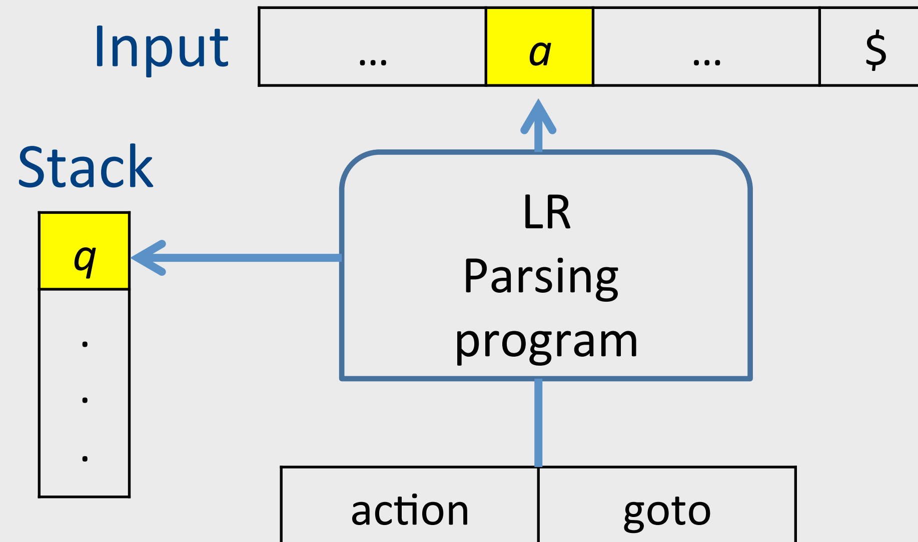
Accept move



If action[q, a] = accept

parsing completed

Error move



If action[q, a] = error (usually empty)

parsing discovered a syntactic error

Example

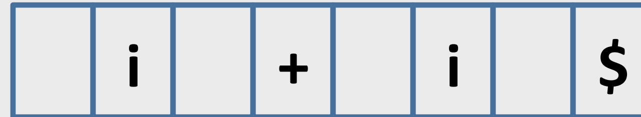
$$Z \rightarrow E \ \mathbf{\$}$$

$$E \rightarrow T \mid E \ \mathbf{+} \ T$$

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

Example: parsing with LR items

$Z \rightarrow E$ **$**
$E \rightarrow T$ | E **+** T
$T \rightarrow$ **i** | **(** E **)**



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

Why do we need these additional LR items?
Where do they come from?
What do they mean?

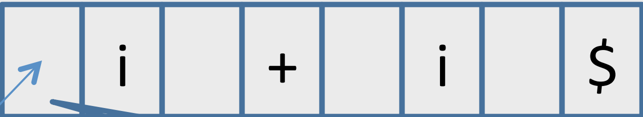
ε-closure

- Given a set S of LR(0) items
- If $P \longrightarrow \alpha \bullet N\beta$ is in state S
- then for each rule $N \longrightarrow \gamma$ in the grammar state S must also contain $N \longrightarrow \bullet\gamma$

$$\varepsilon\text{-closure}(\{Z \longrightarrow \bullet E\ \$\}) = \{\ Z \longrightarrow \bullet E\ \$,\ E \longrightarrow \bullet T,\ E \longrightarrow \bullet E + T,\ T \longrightarrow \bullet i\ ,\ T \longrightarrow \bullet (E)\ \}$$

```
Z → E $
E → T | E + T
T → i | ( E )
```

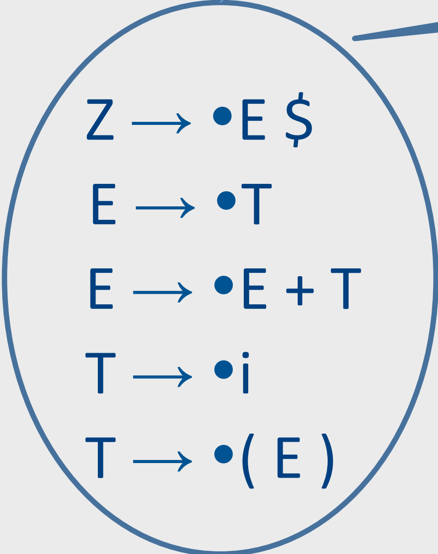
Example: parsing with LR items



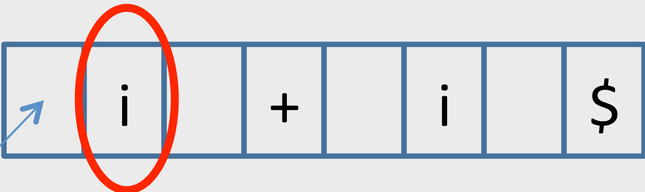
$Z \longrightarrow E\ \$$
$E \longrightarrow T \mid E + T$
$T \longrightarrow i \mid (\ E\ )$

Remember position from which we're trying to reduce

Items denote possible future handles

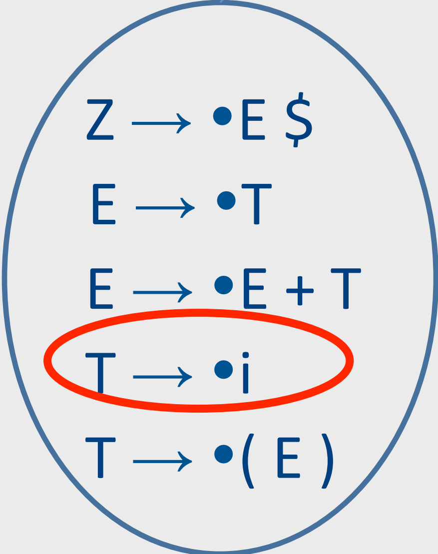


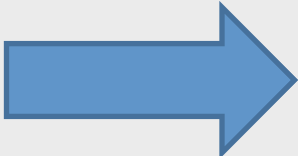
Example: parsing with LR items

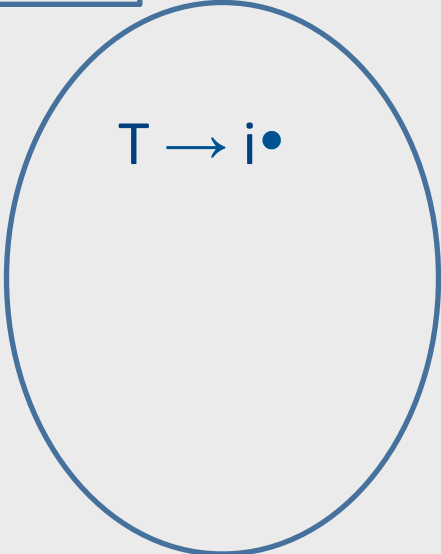


Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)

Match items with current token

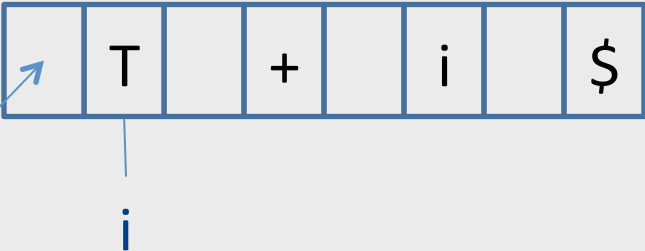




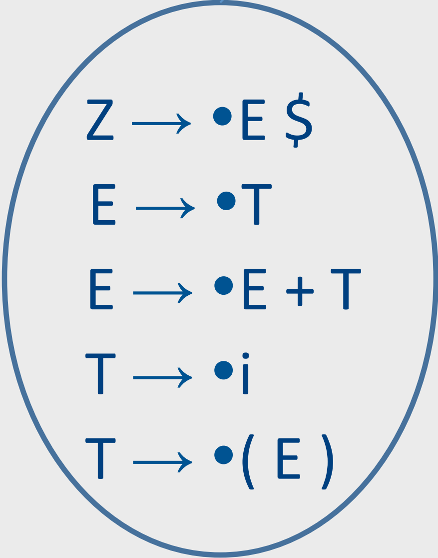


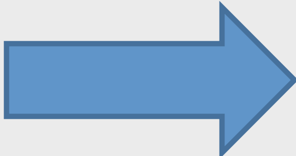
Reduce item!

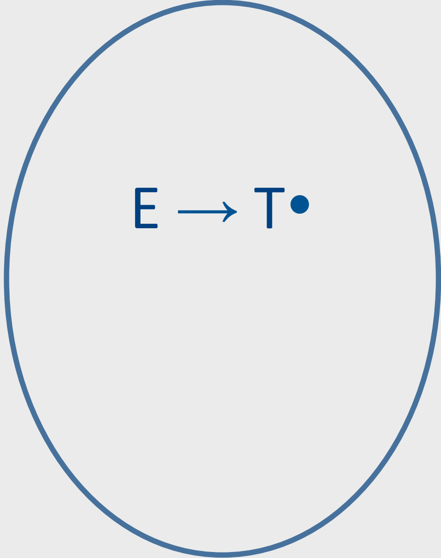
Example: parsing with LR items



Z → E $
E → T | E + T
T → i | (E)

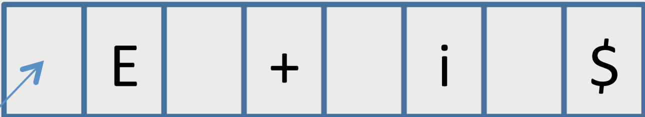






Reduce item!

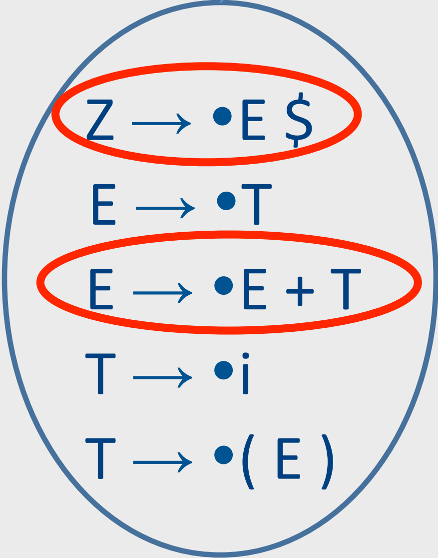
Example: parsing with LR items

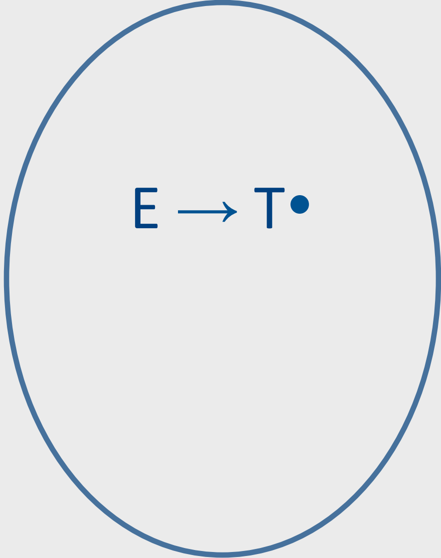


T

i

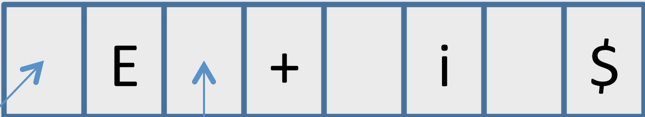
Z → E $
E → T | E + T
T → i | (E)





Reduce item!

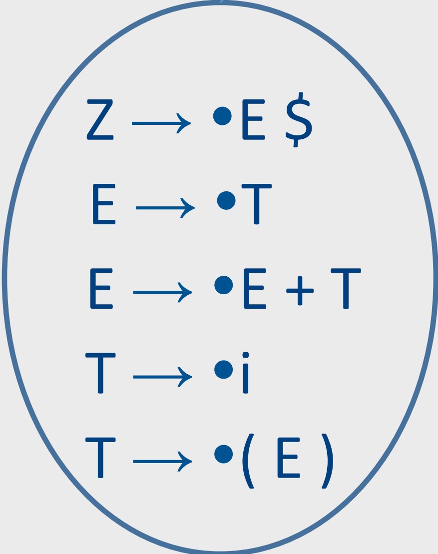
Example: parsing with LR items

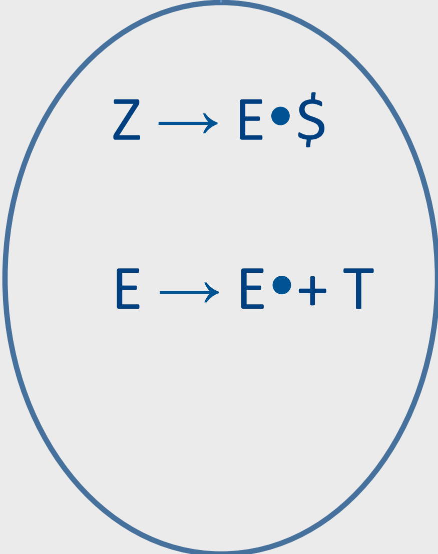


T

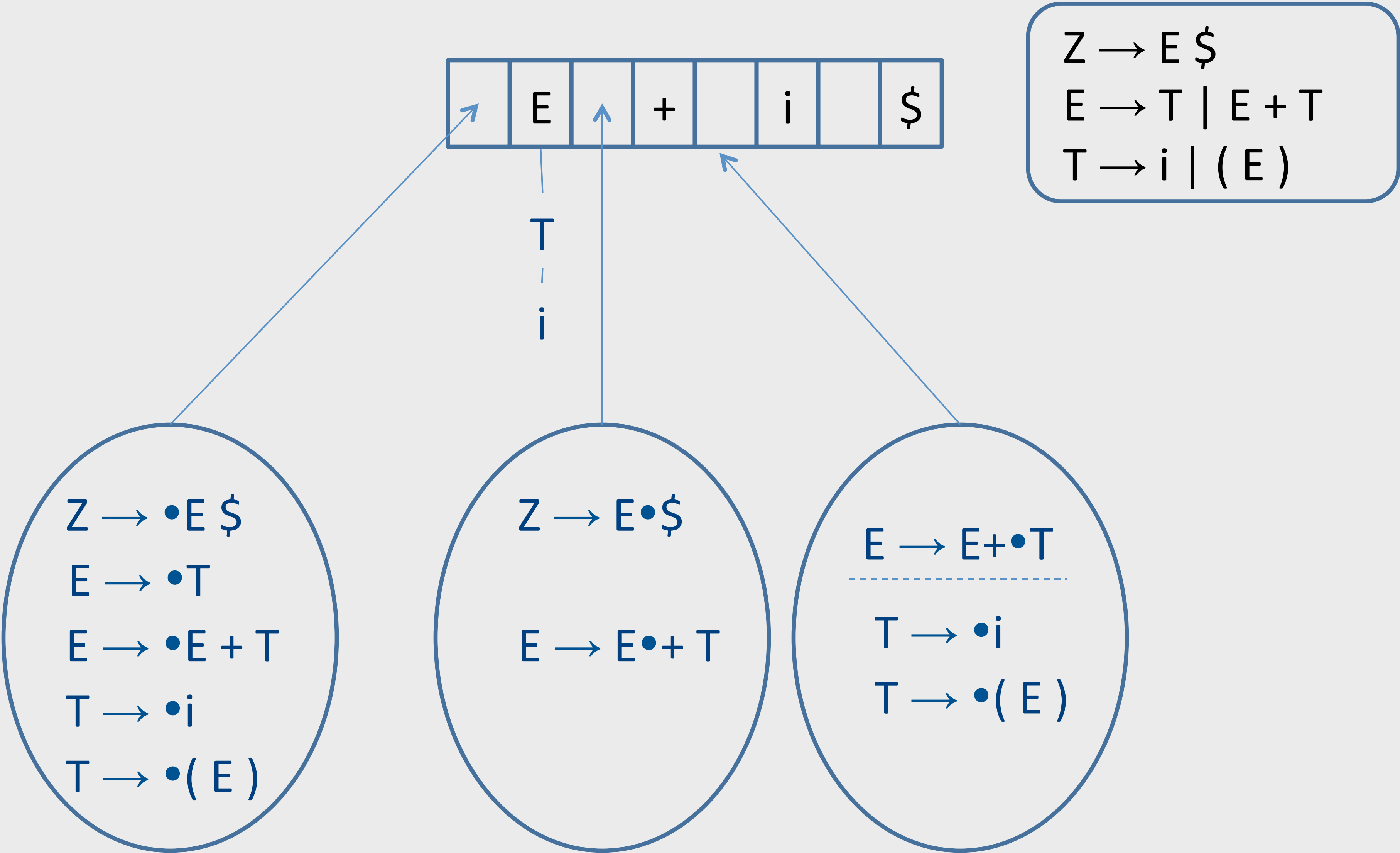
i

Z → E $
E → T | E + T
T → i | (E)

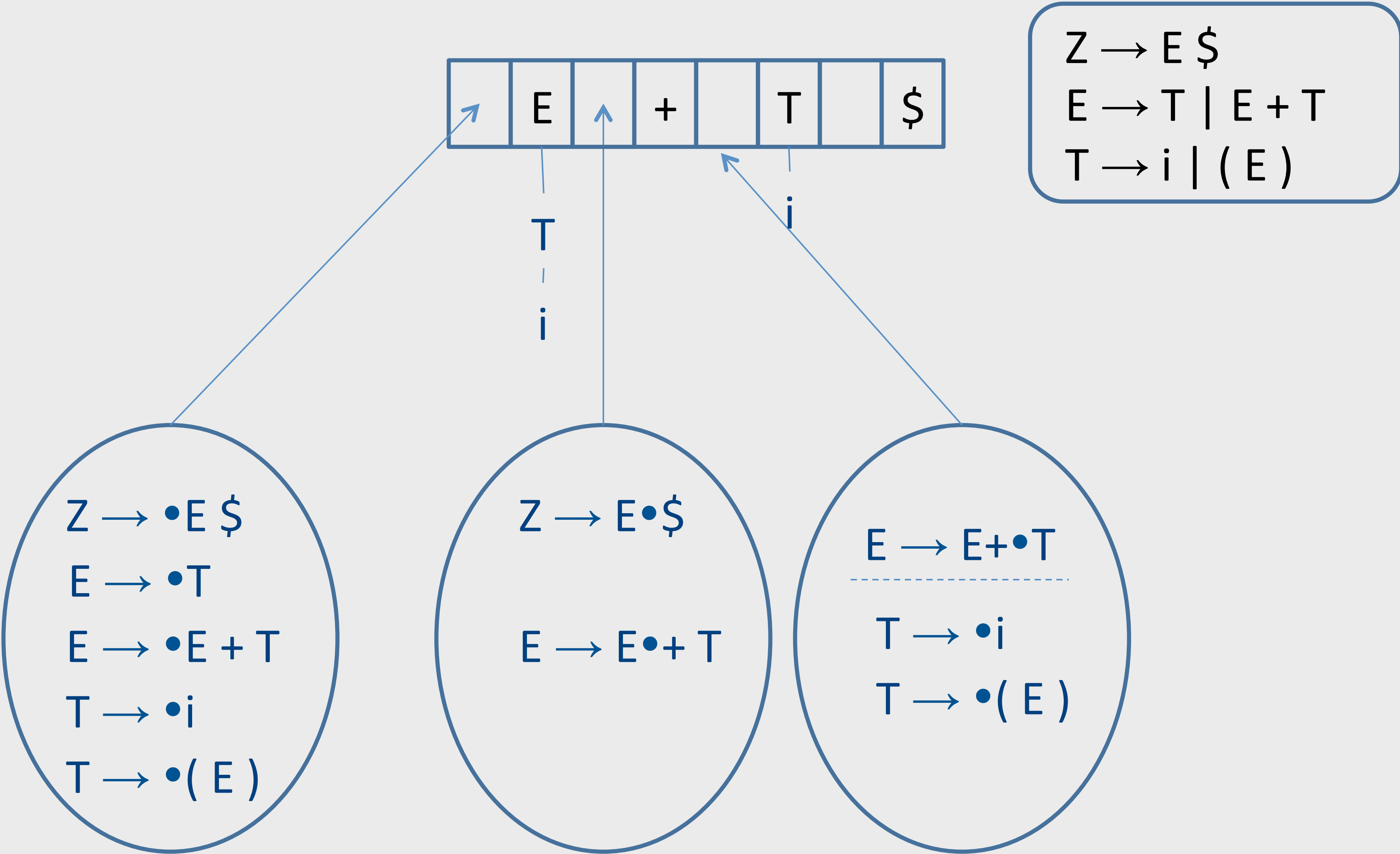




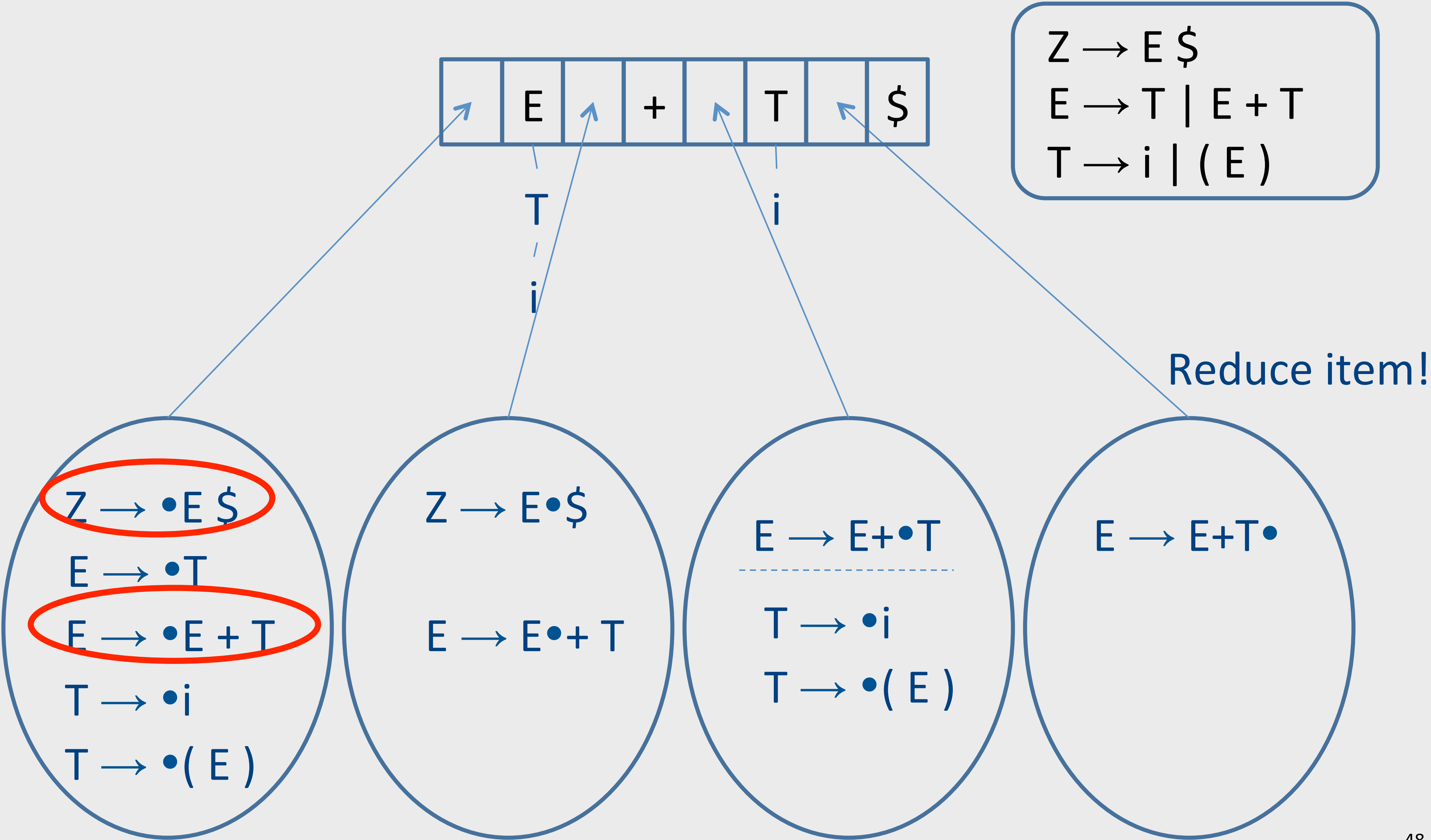
Example: parsing with LR items



Example: parsing with LR items



Example: parsing with LR items



Example: parsing with LR items

Z ⟶ E $

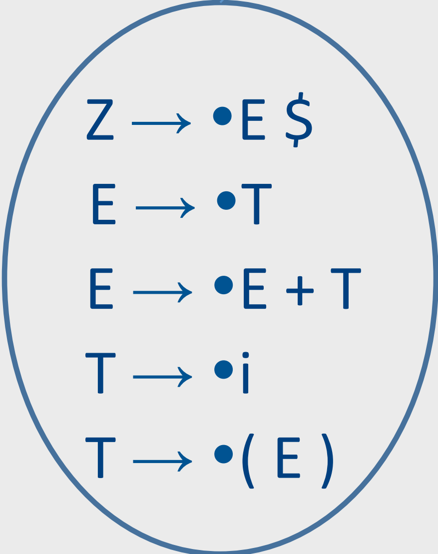
E ⟶ T | E + T

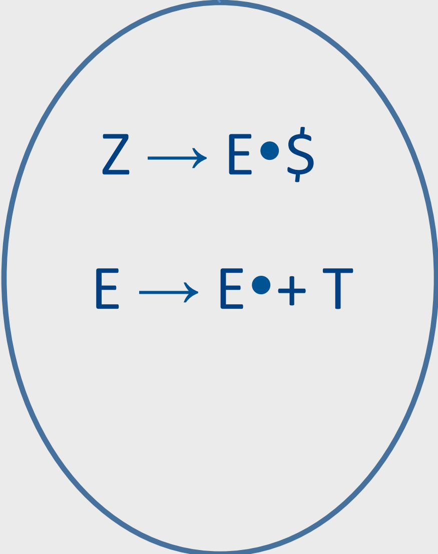
T ⟶ i | (E)

E + T

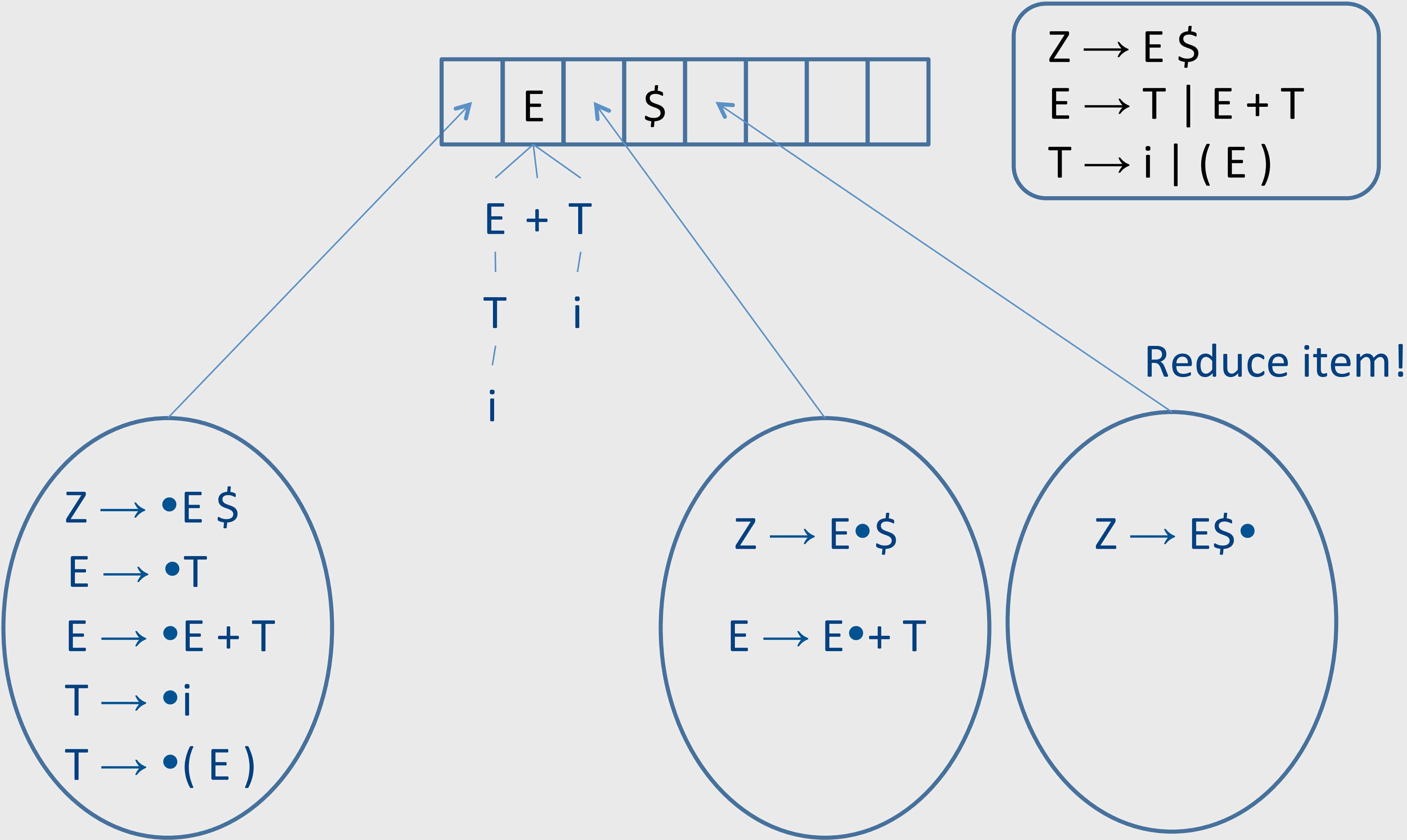
T i

i





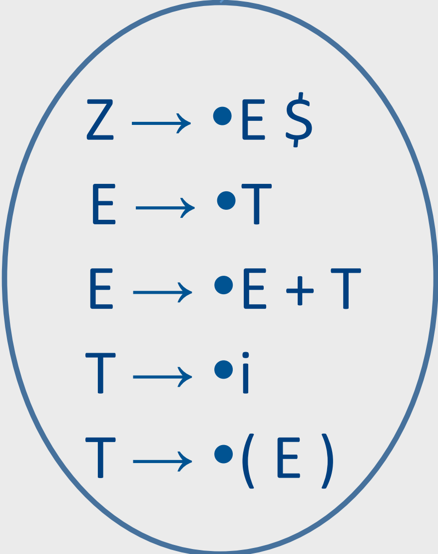
Example: parsing with LR items

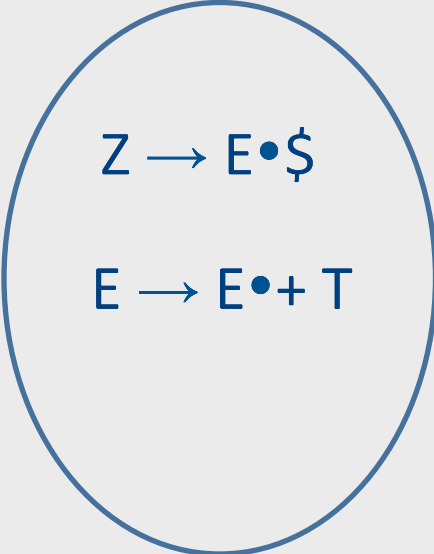


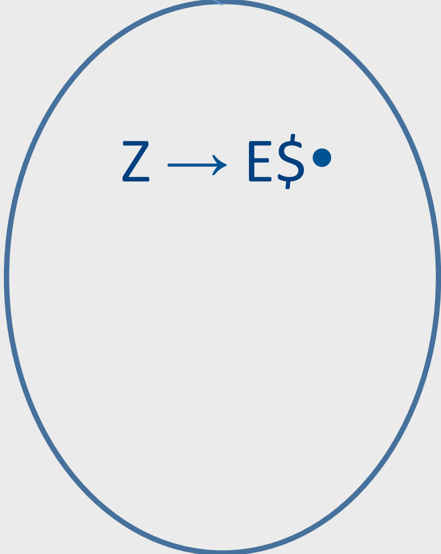
Example: parsing with LR items

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

Reduce item!







GOTO/ACTION tables

GOTO Table

empty – error move

ACTION Table

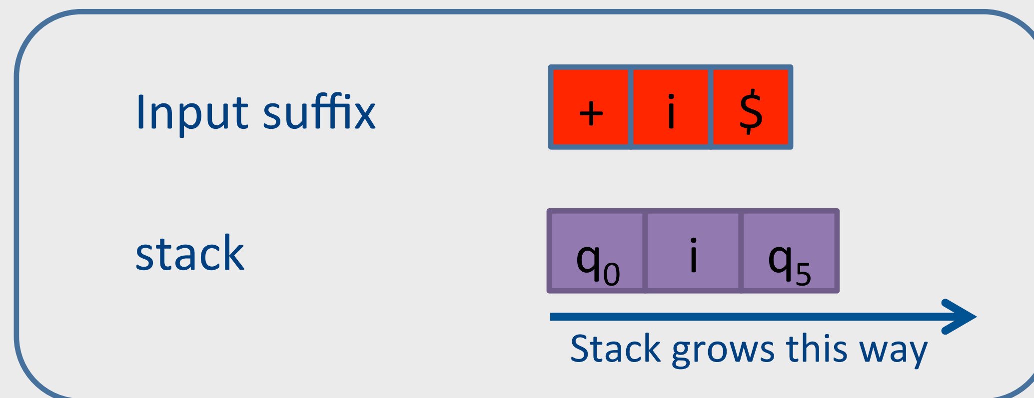
State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q1		q3			q2			shift
q2								Z⟶E$
q3	q5		q7				q4	Shift
q4								E⟶E+T
q5								T⟶i
q6								E⟶T
q7	q5		q7			q8	q6	shift
q8		q3		q9				shift
q9								T⟶E

LR(0) parser tables

- Two types of rows:
  - Shift row – tells which state to GOTO for current token
  - Reduce row – tells which rule to reduce (independent of current token)
    - GOTO entries are blank

LR parser data structures

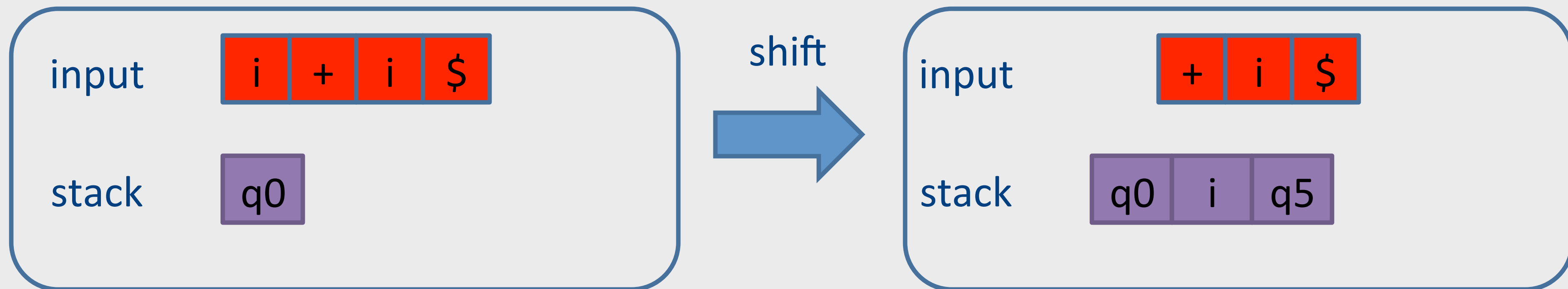
- Input – remainder of text to be processed
- Stack – sequence of pairs N, qi
  - N – symbol (terminal or non-terminal)
  - qi – state at which decisions are made



- Initial stack contains q0

LR(0) pushdown automaton

- Two moves: shift and reduce
- Shift move
  - Remove first token from input
  - Push it on the stack
  - Compute next state based on GOTO table
  - Push new state on the stack
  - If new state is error – report error

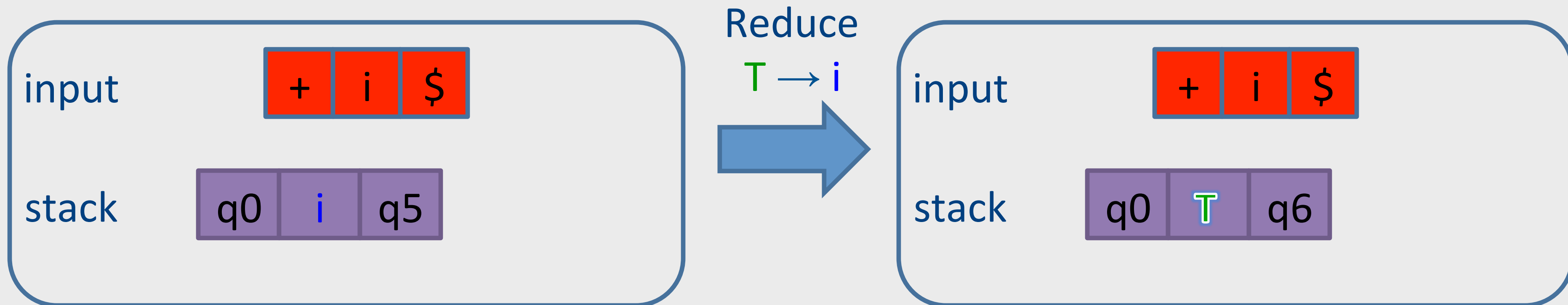


Stack grows this way

State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift

LR(0) pushdown automaton

- Reduce move
  - Using a rule N →α
  - Symbols in α and their following states are removed from stack
  - New state computed based on GOTO table (using top of stack, before pushing N)
  - N is pushed on the stack
  - New state pushed on top of N



Stack grows this way

State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift

GOTO/ACTION table

State	i	+	(	)	$	E	T
q0	s5		s7			s1	s6
q1		s3			s2		
q2	r1	r1	r1	r1	r1	r1	r1
q3	s5		s7				s4
q4	r3	r3	r3	r3	r3	r3	r3
q5	r4	r4	r4	r4	r4	r4	r4
q6	r2	r2	r2	r2	r2	r2	r2
q7	s5		s7			s8	s6
q8		s3		s9			
q9	r5	r5	r5	r5	r5	r5	r5

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

Warning: numbers mean different things!

rn = reduce using rule number n

sm = shift to state m

Parsing id+id$

Stack	Input	Action
0	id + id $	s5

Initialize with state 0

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

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	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$

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

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

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

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

Parsing id+id$

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

push T 6

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$

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

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

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	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	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	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	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	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	$	s2

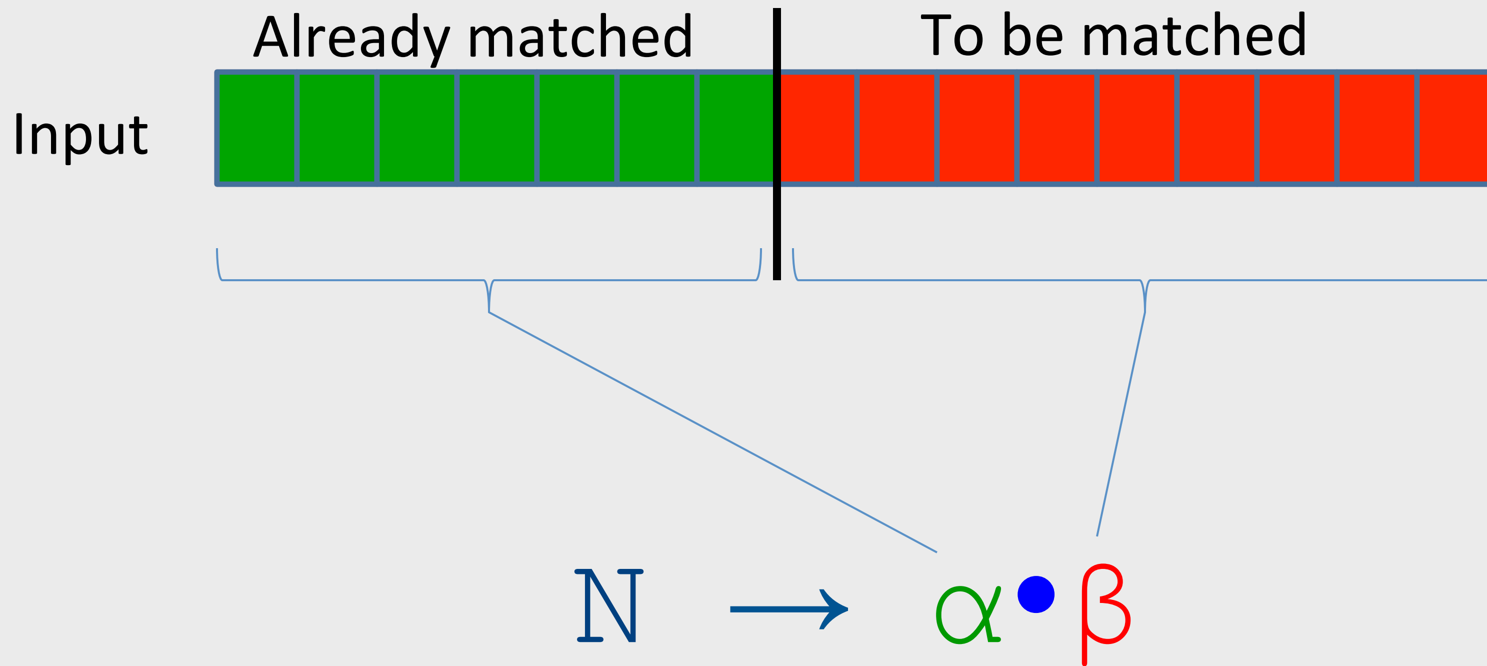
S	*action*					*goto*	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

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

Constructing an LR parsing table

- Construct a (determinized) transition diagram from LR items
- If there are conflicts – stop
- Fill table entries from diagram

LR item



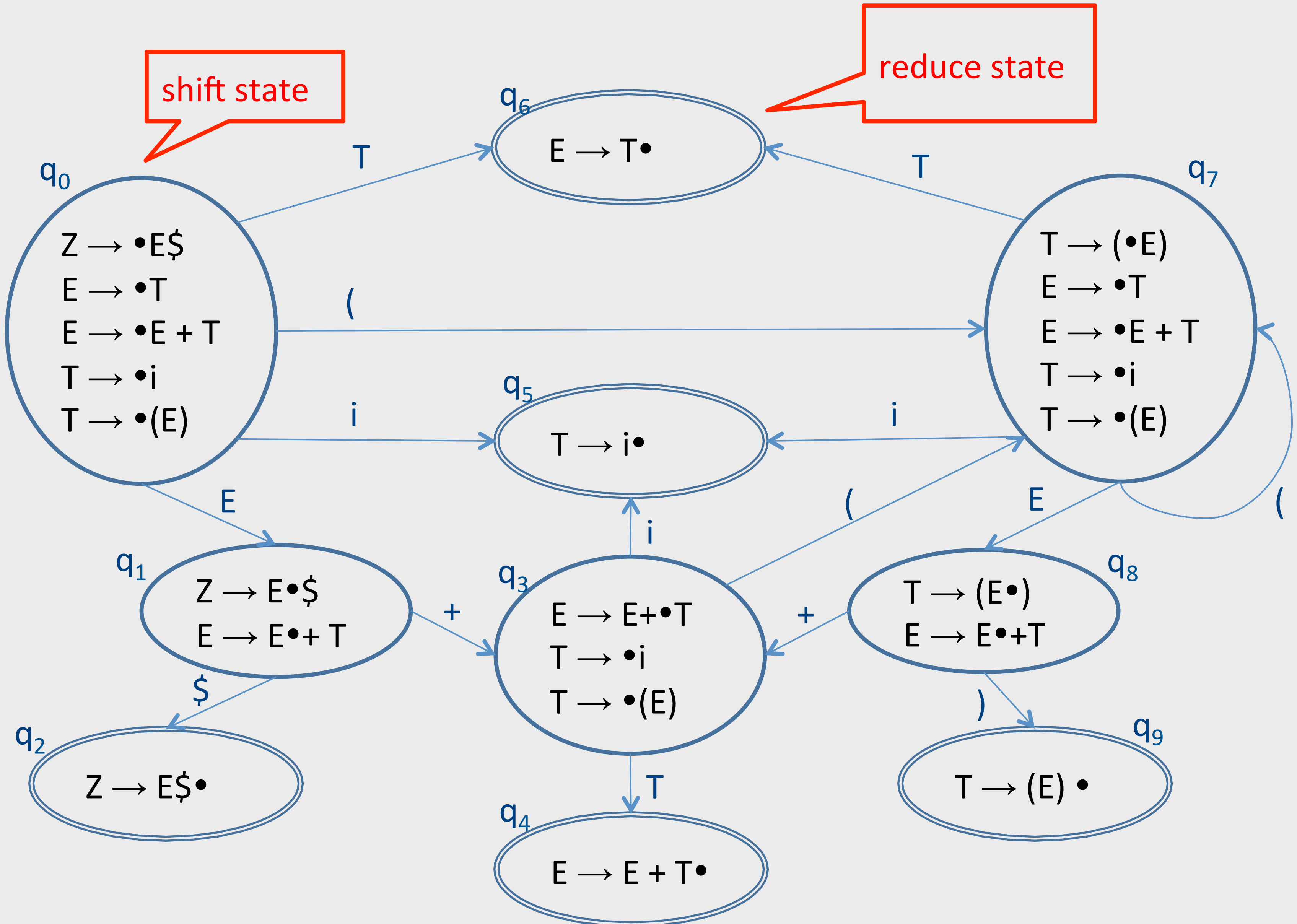
Hypothesis about αβ being a possible handle, so far we've matched α, expecting to see β

Types of LR(0) items

$N \longrightarrow \alpha \bullet \beta$ Shift Item

$N \longrightarrow \alpha \beta \bullet$ Reduce Item

LR(0) automaton example



Computing item sets

- Initial set
  - Z is in the start symbol
  - ε-closure({ Z→•α | Z→α is in the grammar })

- Next set from a set I and the next symbol X
  - step(I,X) = { N→αX•β | N→α•Xβ in the item set I}
  - nextSet(I,X) = ε-closure(step(I,X))

Operations for transition diagram construction

- Initial = $\{S' \rightarrow \bullet S\$\}$

- For an item set I
  $Closure(I) = Closure(I) \cup \{X \rightarrow \bullet \mu \text{ is in grammar} \mid N \rightarrow \alpha \bullet X \beta \text{ in } I\}$

- $Goto(I, X) = \{ N \rightarrow \alpha X \bullet \beta \mid N \rightarrow \alpha \bullet X \beta \text{ in } I\}$

Initial example

For simplicity: omitting S'

- Initial = {S → •E $}

Grammar

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

Closure example

- Initial = {S → •E $}
- Closure({S → •E $}) = {

    S → •E $

    E → •T

    E → •E + T

    T → •id

    T → •(E) }

Grammar

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

Goto example

Grammar

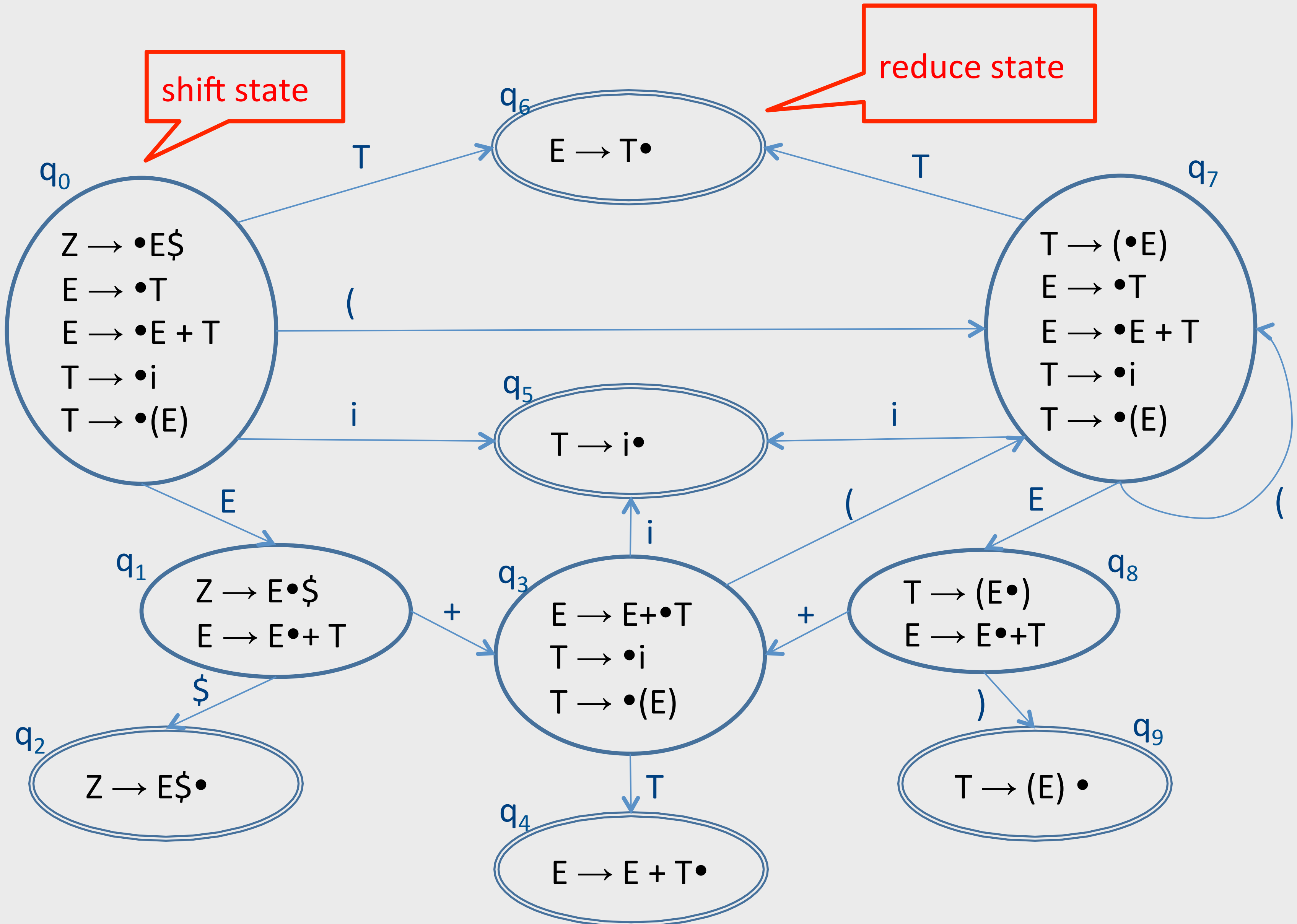
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

- Initial = {S → •E $}
- Closure({S → •E $}) = {
  S → •E $
  E → •T
  E → •E + T
  T → •id
  T → •(E) }
- Goto({S → •E $, E → •E + T, T → •id}, E) =
  {S → E• $, E → E• + T}

Constructing the transition diagram

- Start with state 0 containing item Closure({S $\rightarrow$ •E $})
- Repeat until no new states are discovered
  - For every state p containing item set I_p, and symbol N, compute state q containing item set I_q = Closure(goto(I_p, N))

LR(0) automaton example



Automaton construction example

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

q_0

$S \rightarrow \bullet E\$$

Initialize

Automaton construction example

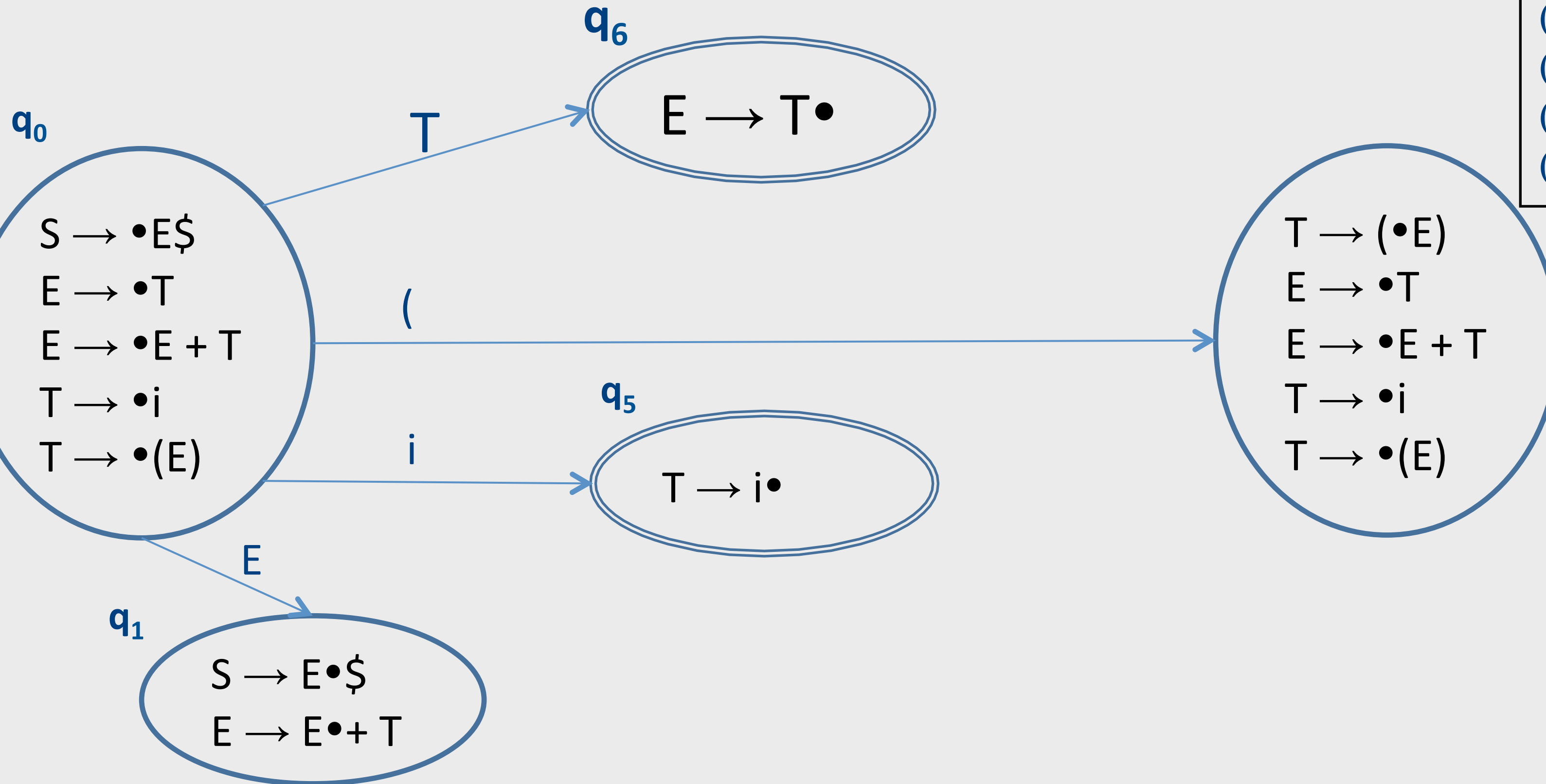
(1) $S \rightarrow E\ \$$
(2) $E \rightarrow T$
(3) $E \rightarrow E + T$
(4) $T \rightarrow id$
(5) $T \rightarrow (\ E\ )$

q_0

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

apply Closure

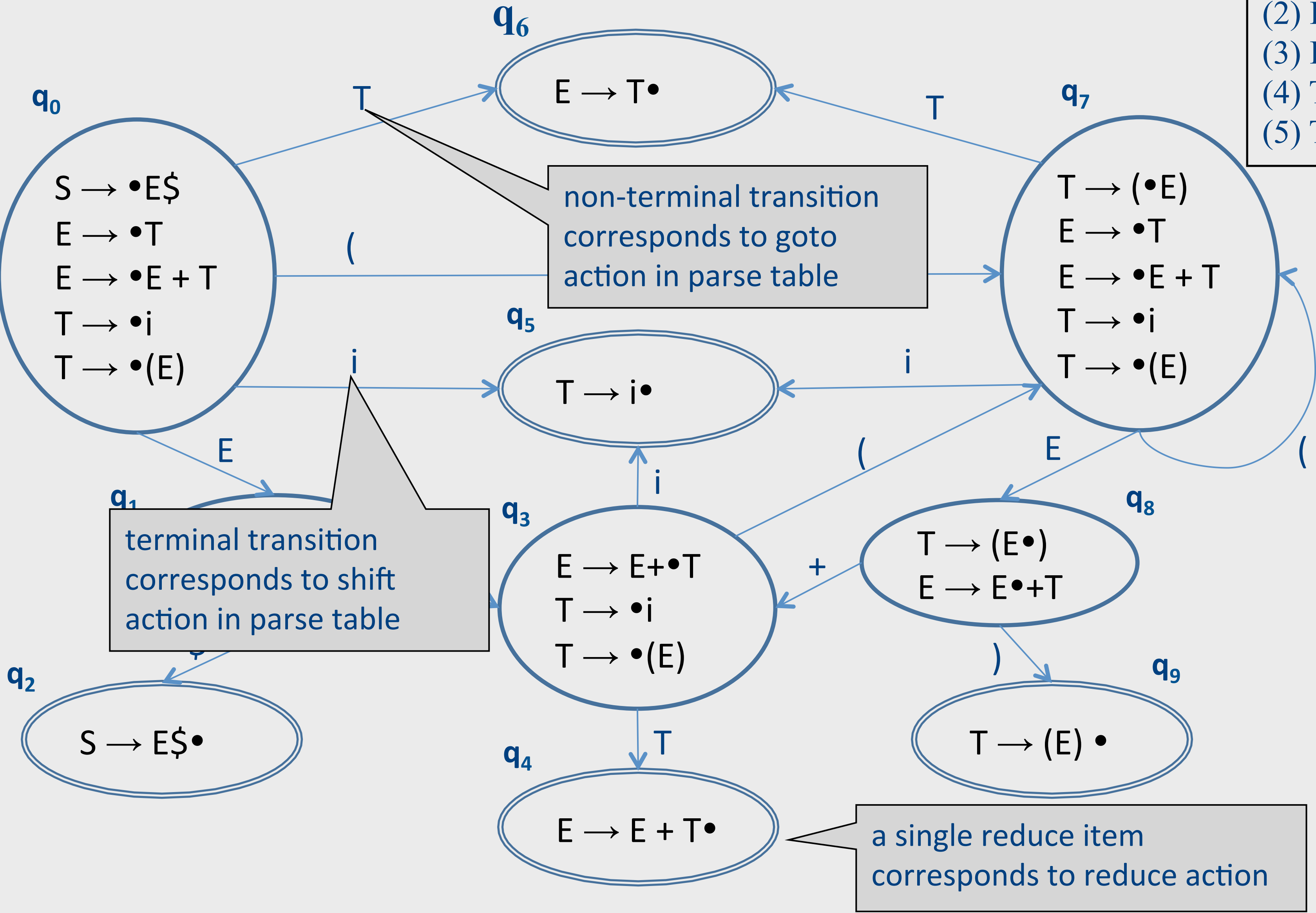
Automaton construction example



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

Automaton construction example

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

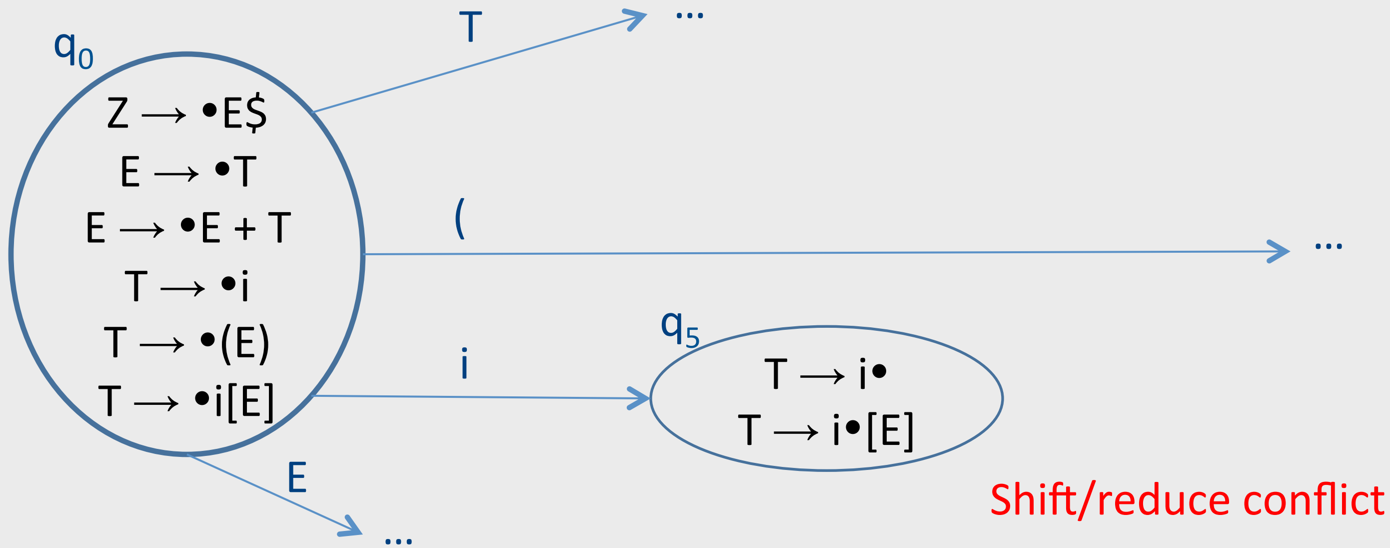


Are we done?

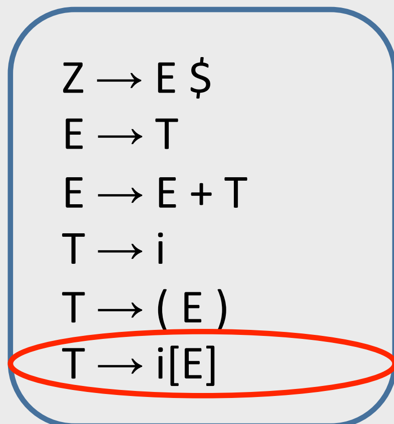
- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar

- Cannot make a deterministic ACTION table for every grammar

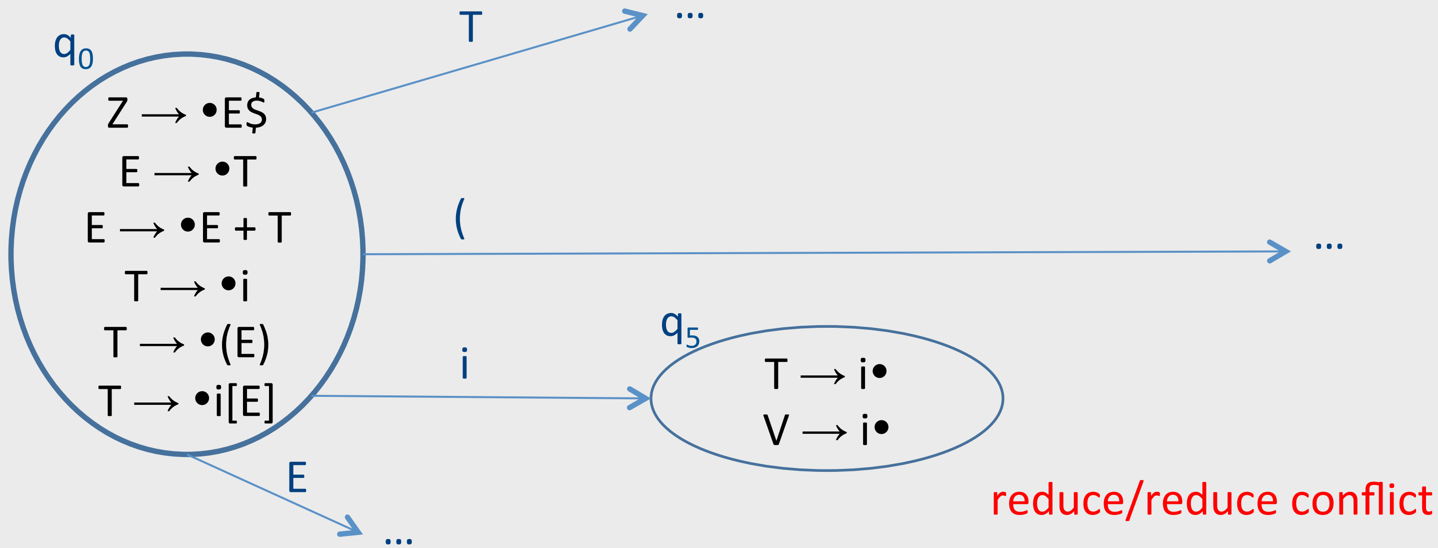
LR(0) conflicts

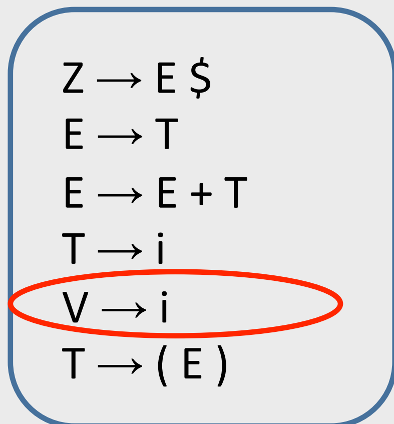


Shift/reduce conflict



LR(0) conflicts





LR(0) conflicts

- Any grammar with an ε-rule cannot be LR(0)

- Inherent shift/reduce conflict
  - $A \rightarrow \varepsilon \bullet$ – reduce item
  - $P \rightarrow \alpha \bullet A \beta$ – shift item
  - $A \rightarrow \varepsilon \bullet$ in Closure ($P \rightarrow \alpha \bullet A \beta$)

Conflicts

- Can construct a diagram for every grammar but some may introduce conflicts
- shift-reduce conflict: an item set contains at least one shift item and one reduce item
- reduce-reduce conflict: an item set contains two reduce items

LR variants

- LR(0) – what we’ve seen so far
- SLR(0)
  - Removes infeasible reduce actions via FOLLOW set reasoning
- LR(1)
  - LR(0) with one lookahead token in items
- LALR(0)
  - LR(1) with merging of states with same LR(0) component

LR (0) GOTO/ACTIONS tables

GOTO table is indexed by state and a grammar symbol from the stack

GOTO Table

ACTION Table

State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q1		q3			q2			shift
q2								Z⟶E$
q3	q5		q7				q4	Shift
q4								E⟶E+T
q5								T⟶i
q6								E⟶T
q7	q5		q7			q8	q6	shift
q8		q3		q9				shift
q9								T⟶E

ACTION table determined <u>only</u> by state, <u>ignores input</u>

SLR parsing

- A handle should not be reduced to a non-terminal N if the lookahead is a token that cannot follow N
- A reduce item $N \rightarrow \alpha\bullet$ is applicable only when the lookahead is in FOLLOW(N)
  - If b is not in FOLLOW(N) we just proved there is no derivation $S \Rightarrow^* \beta N b$.
  - Thus, it is safe to remove the reduce item from the conflicted state
- Differs from LR(0) only on the ACTION table
  - Now a row in the parsing table may contain both shift actions and reduce actions and we need to consult the current token to decide which one to take

SLR action table

Lookahead token from the input

State	i	+	(	)	[	]	$
0	shift		shift				
1		shift					accept
2							
3	shift		shift				
4		E⟶E+T		E⟶E+T			E⟶E+T
5		T⟶i		T⟶i	**shift**		T⟶i
6		E⟶T		E⟶T			E⟶T
7	shift		shift				
8		shift		shift			
9		T⟶(E)		T⟶(E)			T⟶(E)

SLR – use 1 token look-ahead

vs.

state	action
q0	shift
q1	shift
q2	
q3	shift
q4	E⟶E+T
q5	T⟶i
q6	E⟶T
q7	shift
q8	shift
q9	T⟶E

LR(0) – no look-ahead

```
… as before…
T → i
T → i[E]
```

LR(1) grammars

- In SLR: a reduce item $N \longrightarrow \alpha \bullet$ is applicable only when the lookahead is in FOLLOW(N)
- But FOLLOW(N) merges lookahead for all alternatives for N
  - Insensitive to the context of a given production

- LR(1) keeps lookahead with each LR item
- Idea: a more refined notion of follows computed per item

LR(1) items

- LR(1) item is a pair
  - LR(0) item
  - Lookahead token
- Meaning
  - We matched the part left of the dot, looking to match the part on the right of the dot, followed by the lookahead token
- 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

LR(0) items

[L → ● id]
[L → id ●]

LR(1) items

[L → ● id, *]
[L → ● id, =]
[L → ● id, id]
[L → ● id, $]
[L → id ●, *]
[L → id ●, =]
[L → id ●, id]
[L → id ●, $]

LALR(1)

- LR(1) tables have huge number of entries
- Often don't need such refined observation
  - (and cost)
- Idea: find states with the same LR(0) component and merge their lookaheads component as long as there are no conflicts
- LALR(1) not as powerful as LR(1) in theory but works quite well in practice
  - Merging may not introduce new shift-reduce conflicts, only reduce-reduce

Summary

LR is More Powerful than LL

- Any LL(k) language is also in LR(k), i.e., LL(k) ⊂ LR(k).
  - LR is more popular in automatic tools

- But less intuitive

- Also, the lookahead is counted differently in the two cases
  - In an LL(k) derivation the algorithm sees the left-hand side of the rule + k input tokens and then must select the derivation rule
  - In LR(k), the algorithm "sees" all right-hand side of the derivation rule + k input tokens and then reduces
    - LR(0) sees the entire right-side, but no input token

Grammar Hierarchy