Compilation
0368-3133 (Semester A, 2013/14)

Lecture 5: Syntax Analysis

(Bottom-Up Parsing)

Modern Compiler Design: Chapter 2.2

Slides credit: Roman Manevich, Mooly Sagiv, Jeff Ullman, Eran Yahav

Admin

- Next weeks: Trubowicz 101 (Law school)

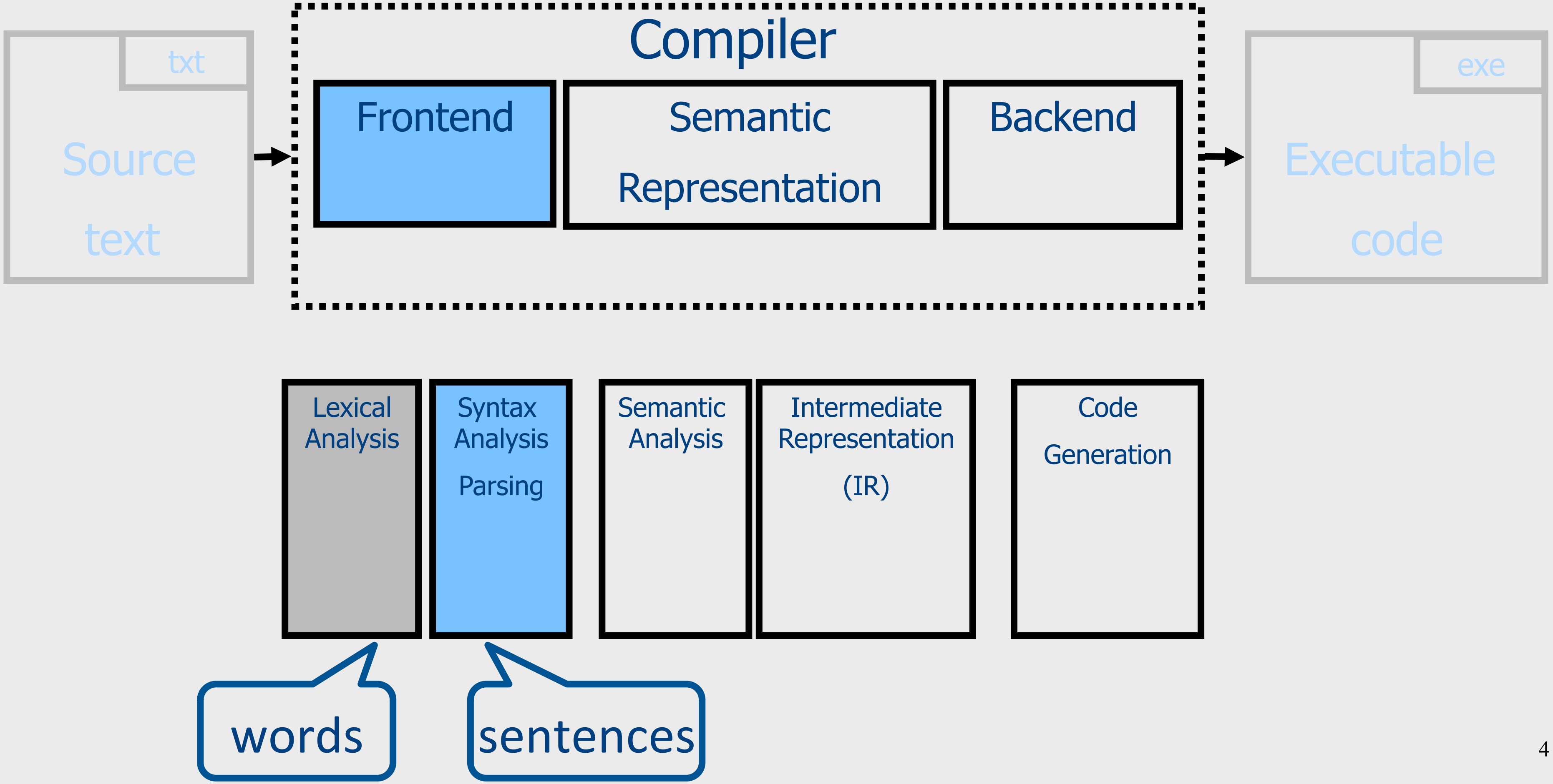
- Mobiles ...

What is a Compiler?

- "A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).
- The most common reason for wanting to transform source code is to create an executable program."

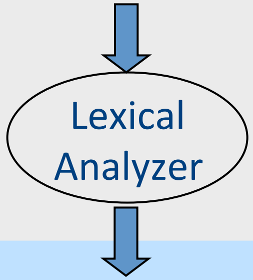
--Wikipedia

Conceptual Structure of a Compiler



From scanning to parsing

program text **((23 + 7) * x)**



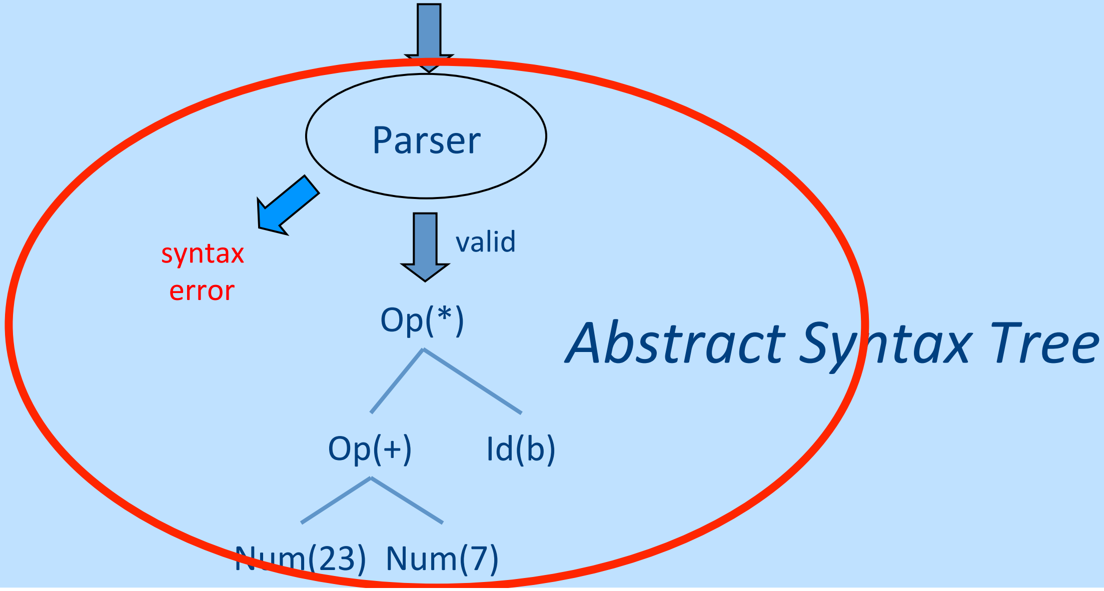
token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

Grammar:

E → ... | Id

Id → 'a' | ... | 'z'



Context Free Grammars

$G = (V,T,P,S)$

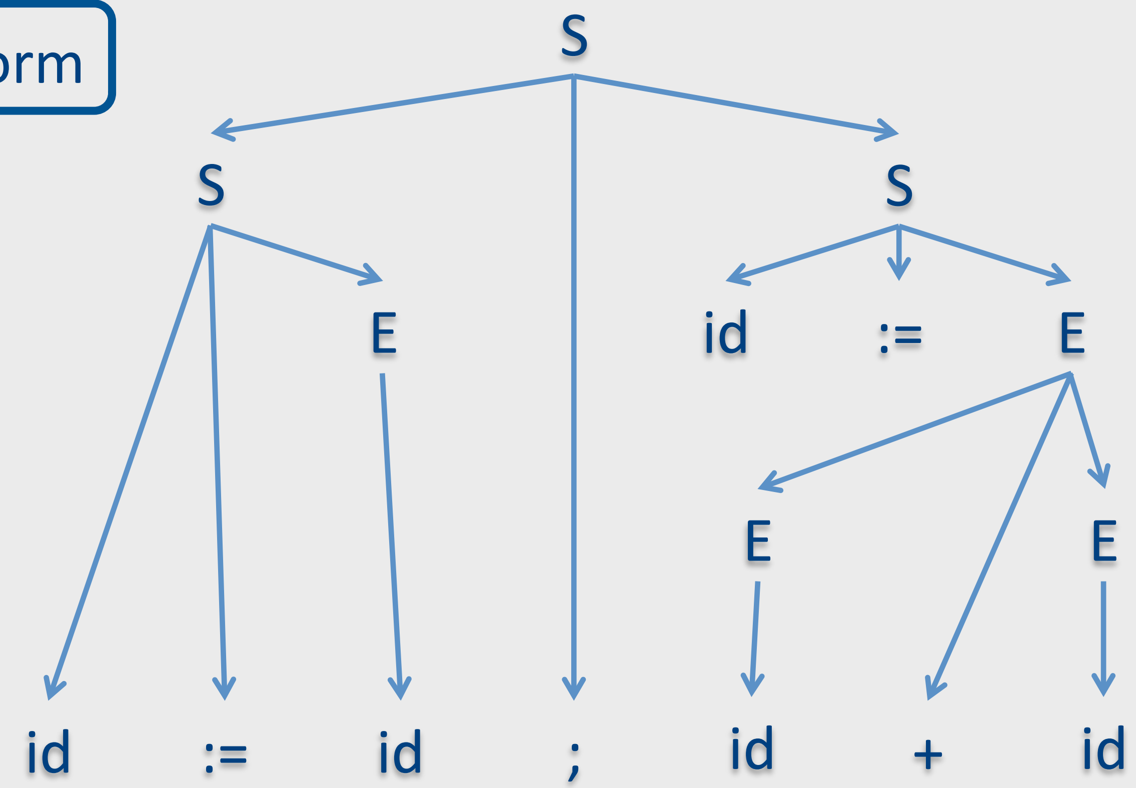
- V – non terminals (syntactic variables)
- T – terminals (tokens)
- P – derivation rules
  - Each rule of the form $V \rightarrow (T \cup V)^*$
- S – start symbol

Derivations & Parse Trees

```
S

S ; S

id := E;  S

id := id; S

id := id;id := E

id := id; id := E + E

id := id;id := E + id

id := id;id := id + id
x:= z ; y := x  + z
```

Sentential form

Derivation



Parse tree

Leftmost/rightmost Derivation

- Leftmost derivation
  - always expand leftmost non-terminal
- Rightmost derivation
  - Always expand rightmost non-terminal

Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method
  - Usually, not used in practice (though might change)
- Top-Down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-Up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

Intuition: Top-Down Parsing

- Begin with start symbol
- “Guess” the productions
- Check if parse tree yields user's program

Recursive Descent

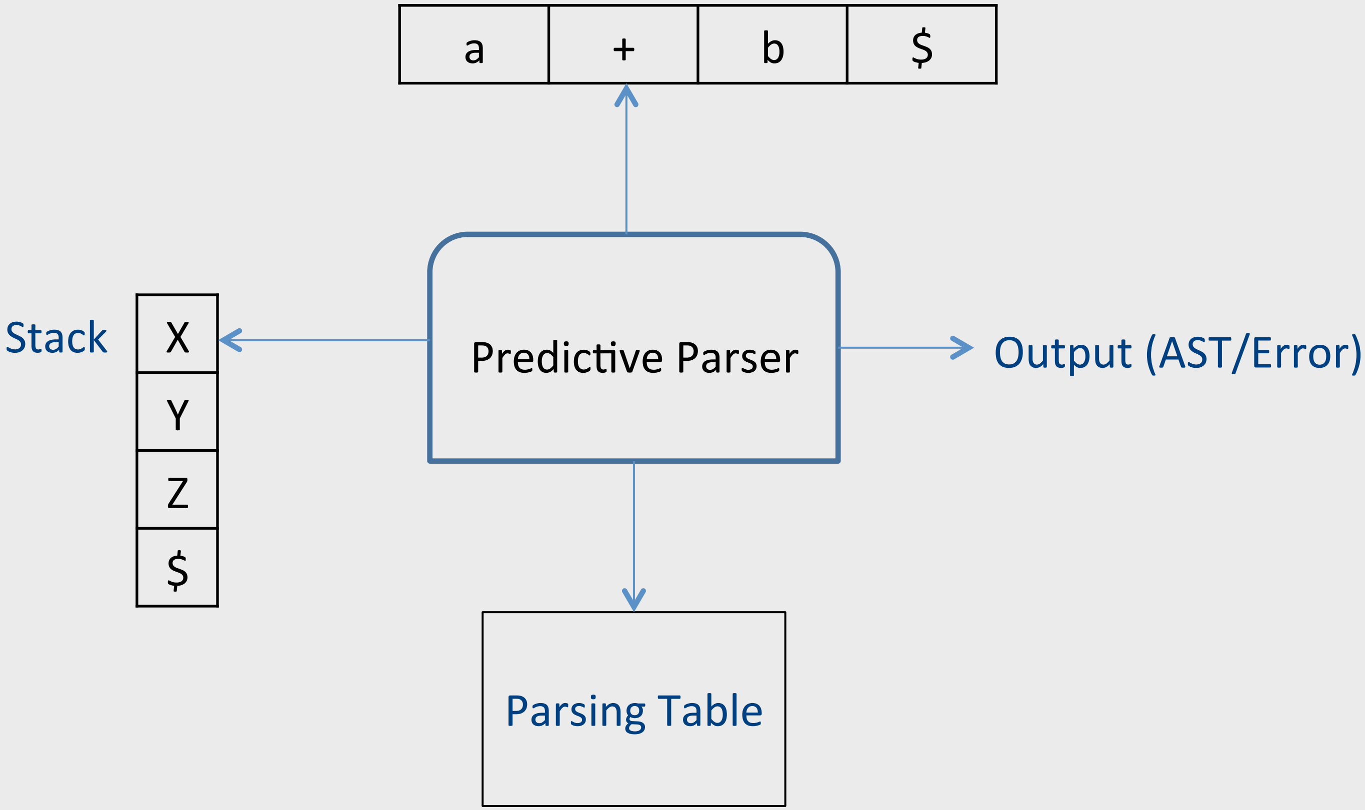
- Blind exhaustive search
  - Goes over all possible production rules
  - Read & parse prefixes of input
  - Backtracks if guesses wrong

- Implementation
  - A (recursive) function for every production rule
  - Backtracks

Predictive parsing

- Predicts which rule to use based on
  - Non terminal
  - Next k input symbols (look ahead)
    - Restricted grammars (LL(k))
- Implementation:
  - Predication stack: Expected future derivation
  - Transition table: Non terminal x terminal → Rule
    - FIRST(α) = The terminals that can appear first in some derivation for α
    - FOLLOW(X) = The tokens that can immediately follow X in some sentential form

Stack-based Predictive Parser



A reminder: Predictive parsing

- Predication stack: Expected future derivation
- Transition table: Non terminal x terminal → Rule
- Moves:
  - Predict
  - match

Running parser example

aacbb$

A → aAb | c

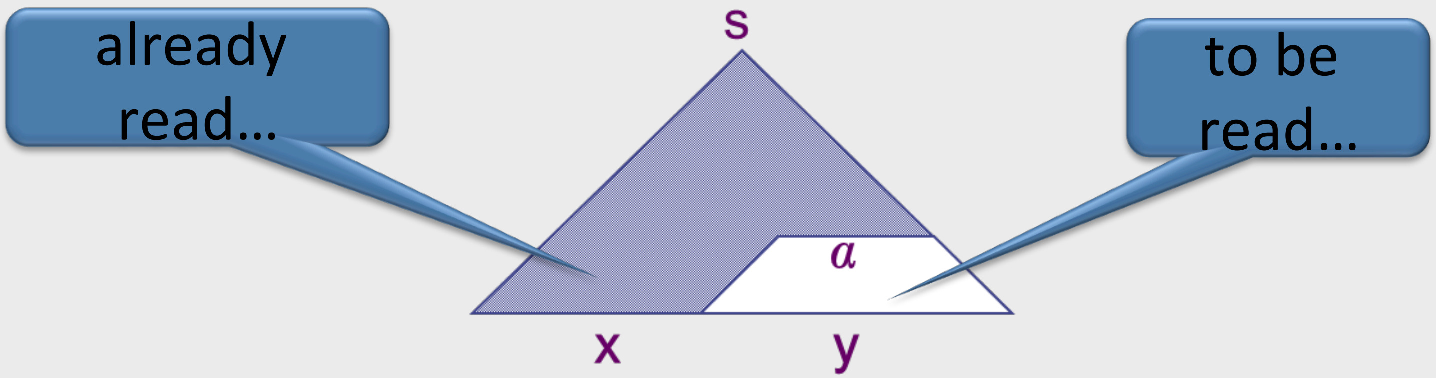
Input suffix	Stack content	Move
aacbb$	A$	predict(A,a) = A → aAb
aacbb$	aAb$	match(a,a)
acbb$	Ab$	predict(A,a) = A → aAb
acbb$	aAbb$	match(a,a)
cbb$	Abb$	predict(A,c) = A → c
cbb$	cbb$	match(c,c)
bb$	bb$	match(b,b)
b$	b$	match(b,b)
$	$	match($,$) – success

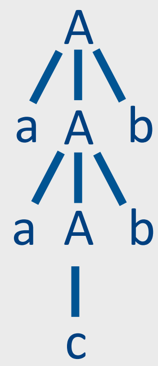
	a	b	c
A	A → aAb		A → c

Top-Down Predictive Parsing

A→aAb|c

aacbb$





Earley Parsing

- Parse arbitrary grammars in O(|input|3)
- Dynamic programming implementation of a recursive descent parser
  - S[N+1] Sequence of sets of "Earley states"
    - N = |INPUT|
    - Earley states is a sentential form + aux info
  - S[i] All parse tree that can be produced (by an RDP) after reading the first i tokens
    - S[i+1] built using S[0] … S[i]

Earley States

- s = < constituent, back >
  - constituent (dotted rule) for $A \rightarrow \alpha\beta$
    - $A \rightarrow \bullet\alpha\beta$ predicated constituents
    - $A \rightarrow \alpha\bullet\beta$ in-progress constituents
    - $A \rightarrow \alpha\beta\bullet$ completed constituents
  - back previous Early state in derivation

Earley Parser

```
Input = x[1…N]
S[0] = <E’→ •E, 0>; S[1] = … S[N] = {}
for  i = 0 ... N do
   until S[i] does not change do
      foreach s ∈ S[i]
         if  s = <A→…•a…, b>  and  a=x[i+1] then                   // scan
            S[i+1] =  S[i+1]  ∪ {<A→…a•…, b>  }
         if  s = <A→ … •X …, b> and X→α then                        // predict
            S[i] = S[i]  ∪  {<X→•α, i >  }
         if  s = < A→ … • , b>  and  <X→…•A…,  k>  ∈ S[b] then     // complete
            S[i] = S[i]  ∪{<X→…A•…,  k>  }
```

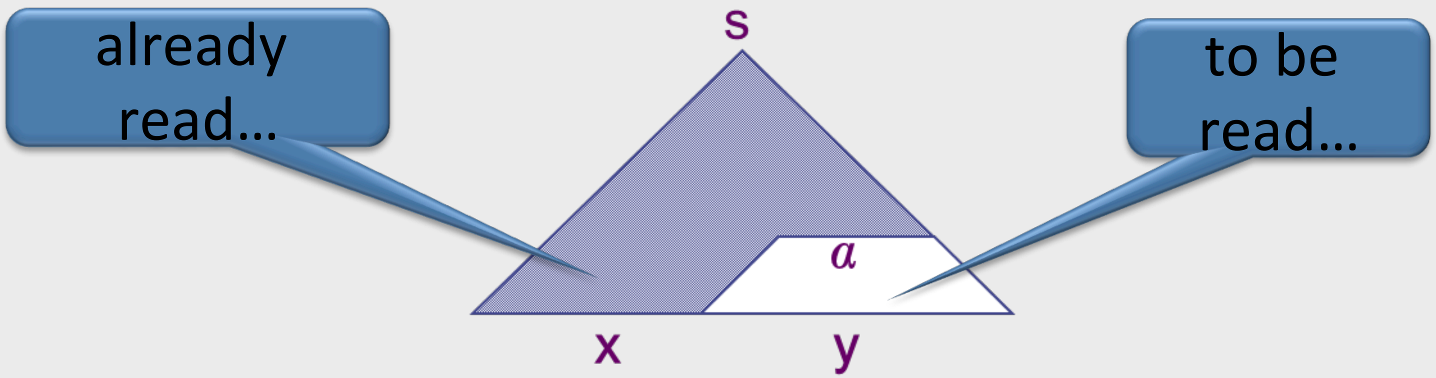
Bottom-Up Parsing

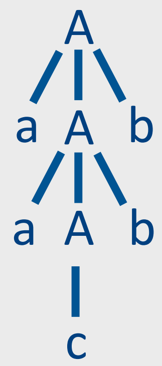
Top-Down vs Bottom-Up

A→aAb|c

aacbb$

- Top-down (predict match/scan-complete)



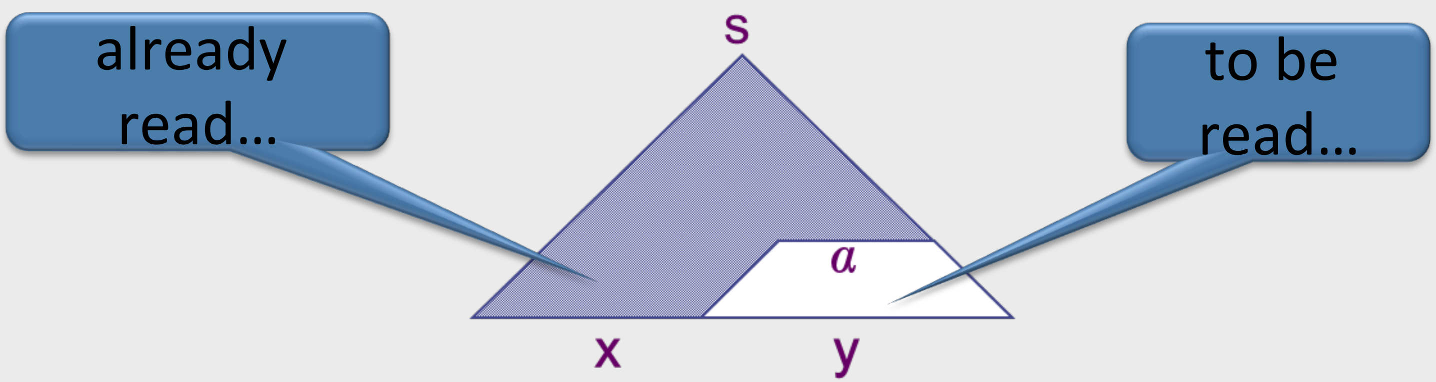


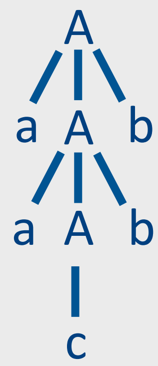
Top-Down vs Bottom-Up

A→aAb|c

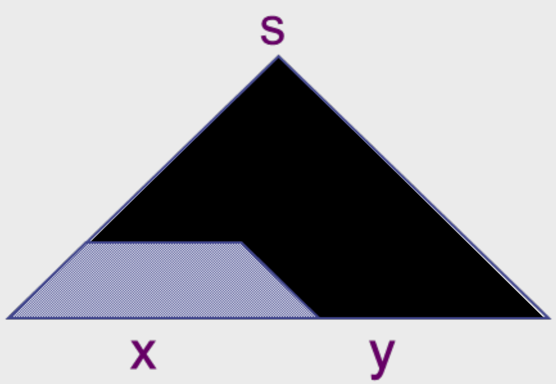
aacbb$

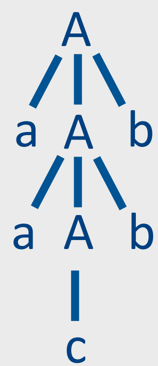
- Top-down (predict match/scan-complete)





- Bottom-up (shift reduce)





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

Terminology: Reductions & Handles

- The opposite of derivation is called *reduction*
  - Let A→α be a production rule
  - Derivation: βAμ ➔ βαμ
  - Reduction: βαμ ➔ βAμ

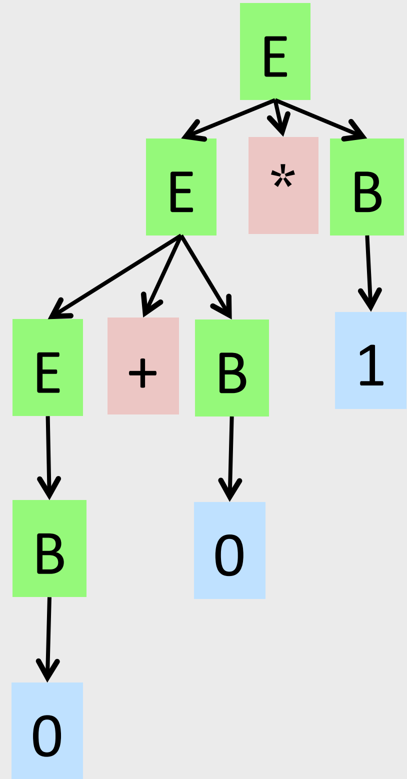
- A *handle* is the reduced substring
  - α is the handles for βαμ

Goal: Reduce the Input to the Start Symbol

E → E * B | E + B | B
B → 0 | 1

Example:

0 + 0 * 1
B + 0 * 1
E + 0 * 1
E + B * 1
E * 1
E * B
E



Go over the input so far, and upon seeing a right-hand side of a rule, "invoke" the rule and replace the right-hand side with the left-hand side (reduce)

Use Shift & Reduce

In each stage, we

shift a symbol from the input to the stack, or

reduce according to one of the rules.

Use Shift & Reduce

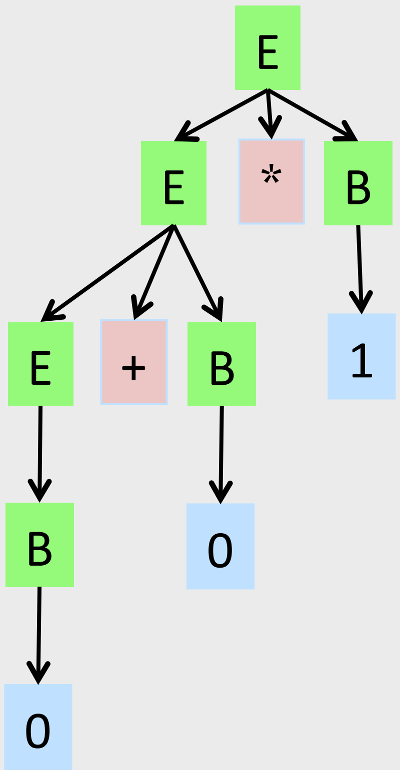
In each stage, we

- shift a symbol from the input to the stack, or
- reduce according to one of the rules.

Example: “0+0*1”

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

$B \rightarrow 0 \mid 1$

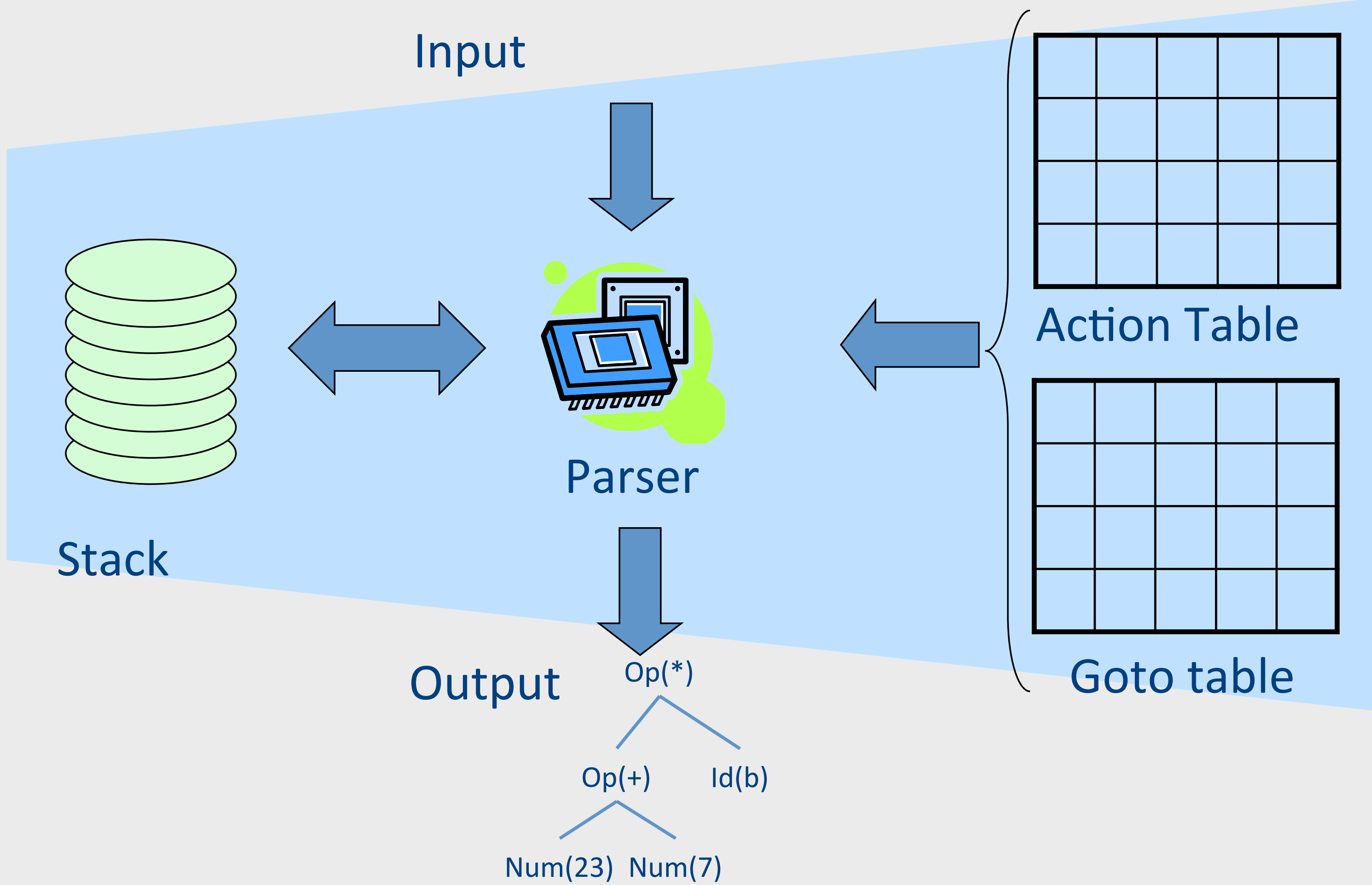


Stack	Input	action
	0+0*1$	shift
0	+0*1$	reduce
B	+0*1$	reduce
E	+0*1$	shift
E+	0*1$	shift
E+0	*1$	reduce
E+B	*1$	reduce
E	*1$	shift
E*	1$	shift
E*1	$	reduce
E*B	$	reduce
E	$	accept

How does the parser know what to do?

token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP



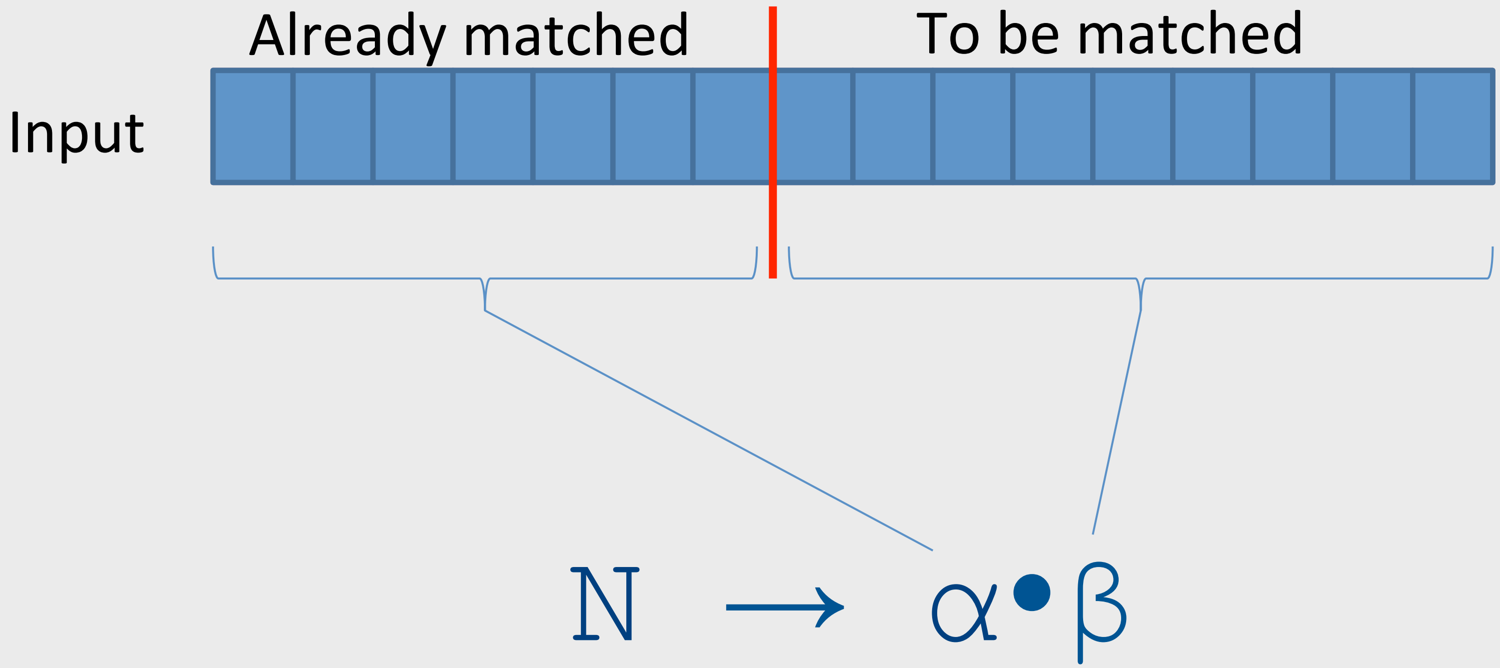
How does the parser know what to do?

- A state will keep the info gathered on handle(s)
  - A state in the “control” of the PDA
  - Also (part of) the stack alpha beit

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:

Grammar

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

→

LR(0) items

1: S → •E$
2: S → E • $
3: S → E $ •
4: E → • T
5: E → T •
6: E → • E + T
7: E → E • + T
8: E → E + • T
9: E → E + T •
10: T → • i
11: T → i •
12: T → • (E)
13: T → (• E)
14: T → (E •)
15: T → (E) •

LR(0) items

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

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

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

States and LR(0) Items

- The state will "remember" the potential derivation rules given the part that was already identified
- For example, if we have already identified E then the state will remember the two alternatives:

(1) $E \rightarrow E * B$, (2) $E \rightarrow E + B$

- Actually, we will also remember where we are in each of them:

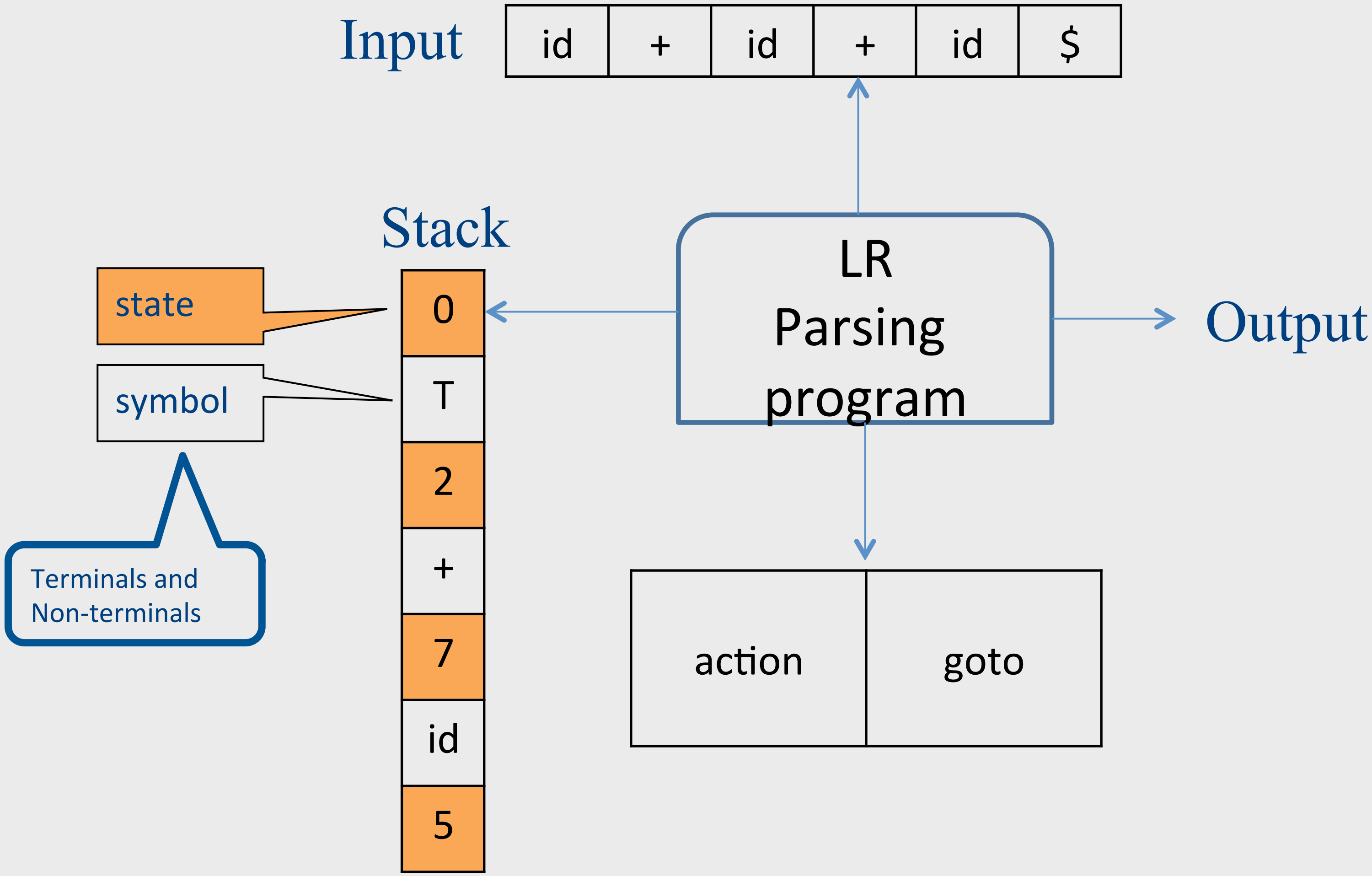
(1) $E \rightarrow E \bullet * B$, (2) $E \rightarrow E \bullet + B$

- A derivation rule with a location marker is called LR(0) item.
- The state is actually a set of LR(0) items. E.g., $q_{13} = \{ E \rightarrow E \bullet * B,\ E \rightarrow E \bullet + B \}$

Intuition

- Gather input token by token until we find a right-hand side of a rule and then replace it with the non-terminal on the left hand side
  - Going over a token and remembering it in the stack is a shift
    - Each shift moves to a state that remembers what we've seen so far
  - A reduce replaces a string in the stack with the non-terminal that derives it

Model of an LR parser

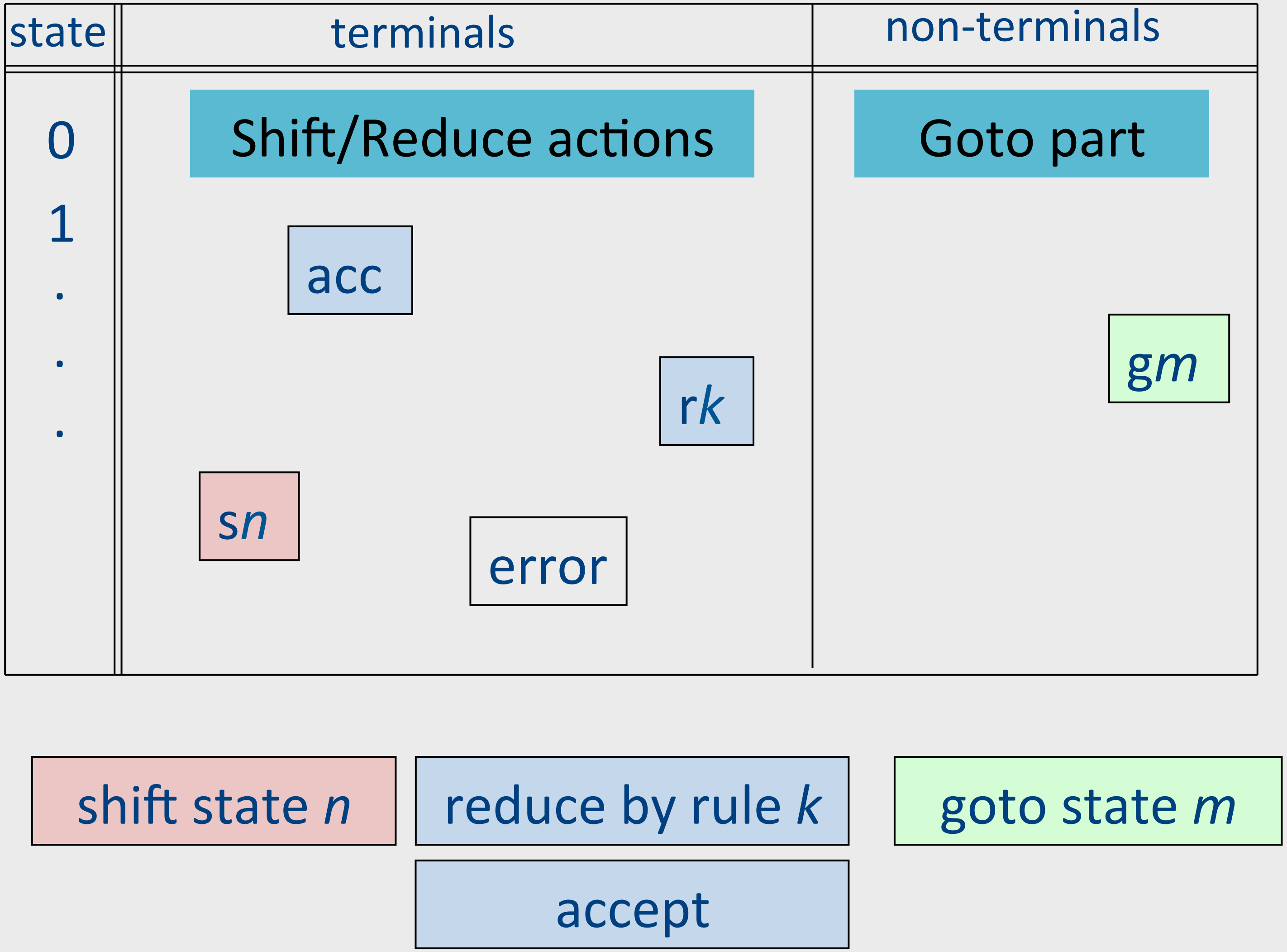


LR parser stack

- 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

Stack grows this way

Form of LR parsing table

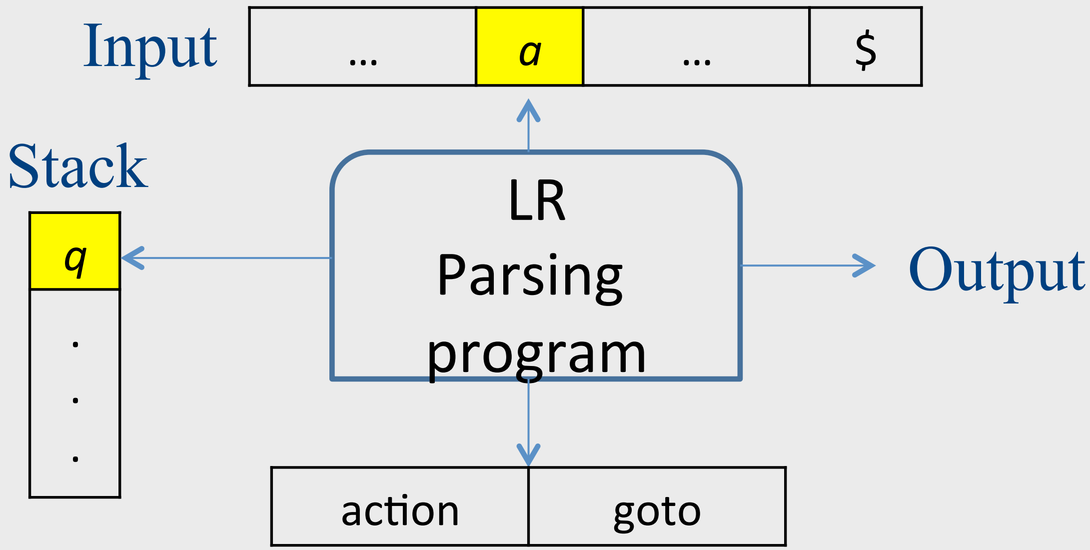


LR parser table example

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		

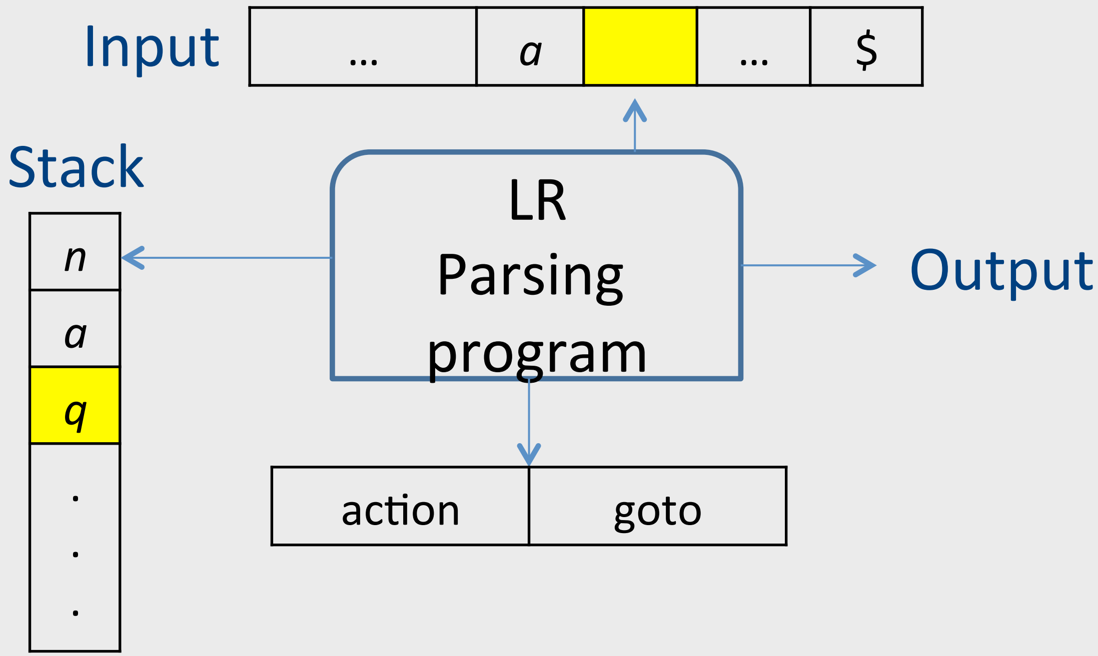
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Shift move



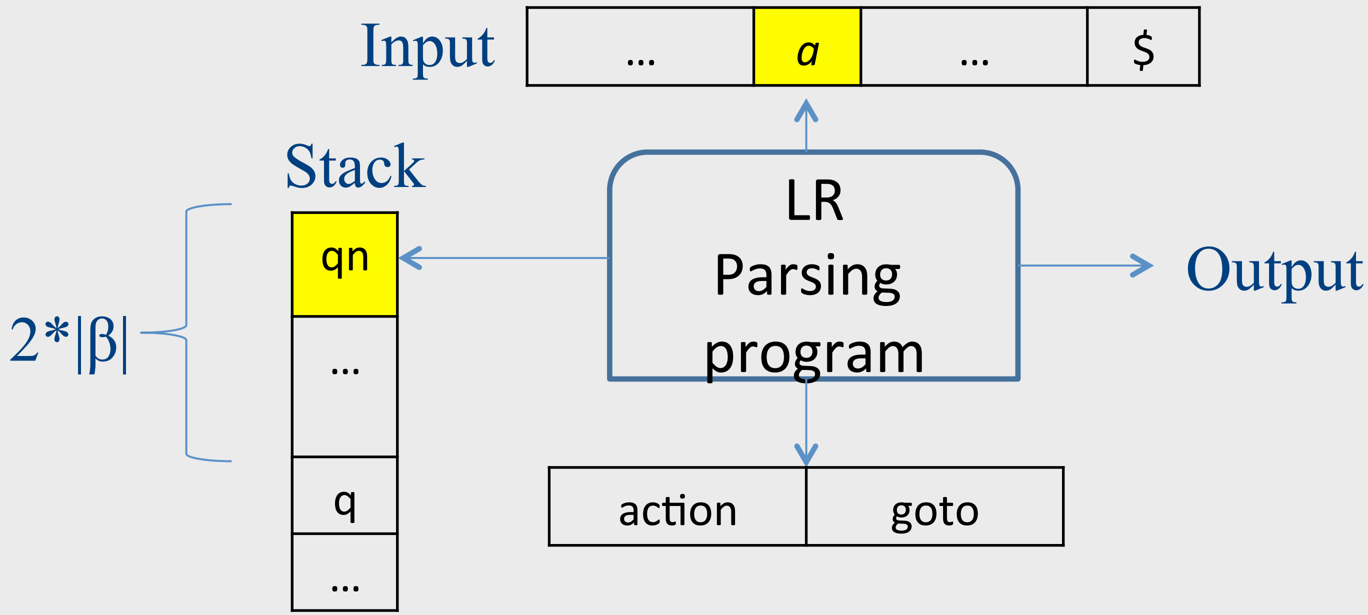
- If action[q, a] = sn

Result of shift



- If action[q, a] = sn

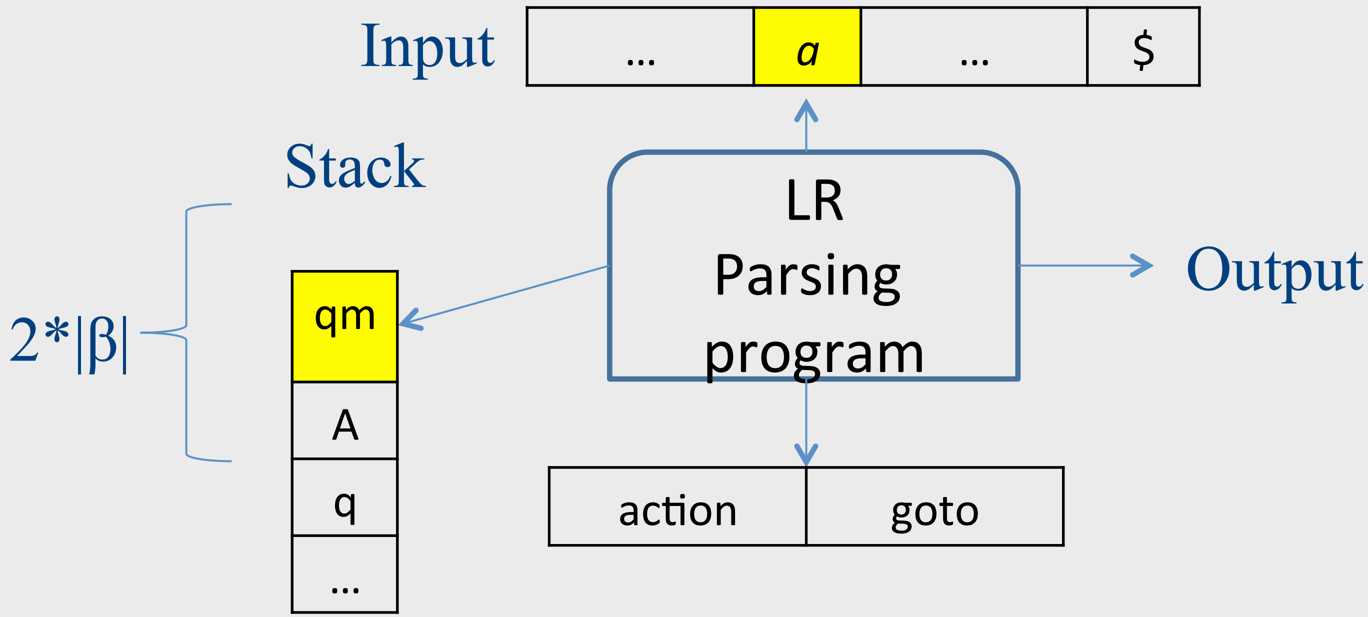
Reduce move



- If action[qn, a] = rk
- Production: (k) A →β
- If β= σ1… σn

  Top of stack looks like q1 σ1… qn σn
- goto[q, A] = qm

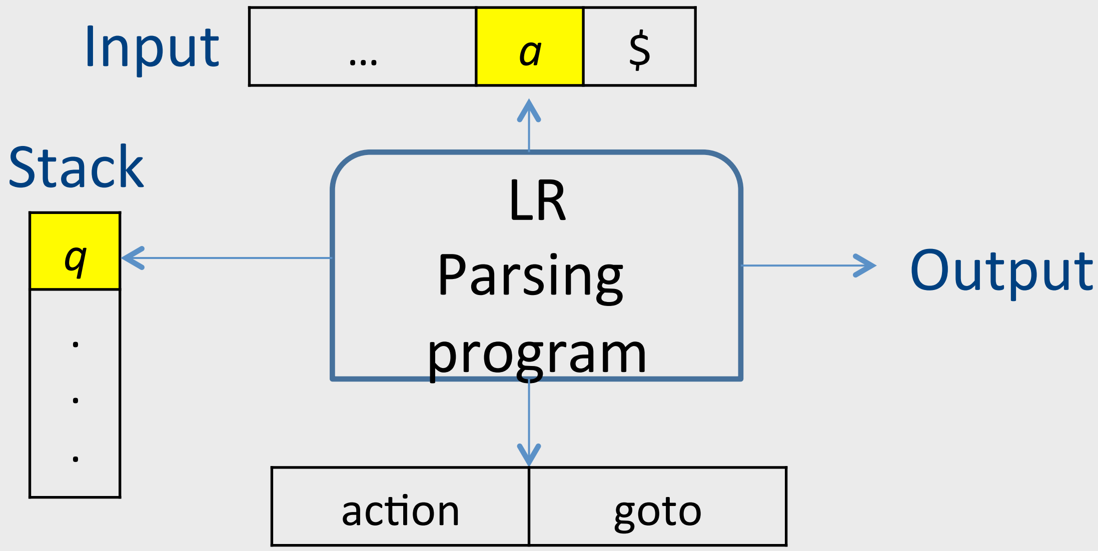
Result of reduce move



- If action[qn, a] = rk
- Production: (k) A →β
- If β= σ1… σn

  Top of stack looks like q1 σ1… qn σn
- goto[q, A] = qm

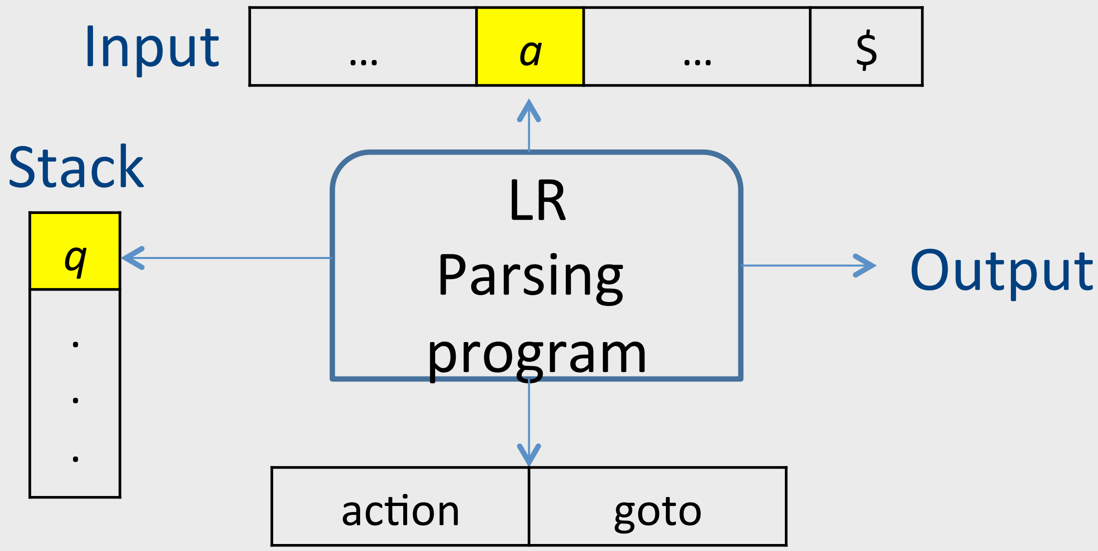
Accept move



If action[q, a] = accept

parsing completed

Error move



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

parsing discovered a syntactic error

Example

$$Z \longrightarrow E\ \mathtt{\$}$$

$$E \longrightarrow T \mid E\ \mathtt{+}\ T$$

$$T \longrightarrow \mathtt{i} \mid \mathtt{(}\ E\ \mathtt{)}$$

Example: parsing with LR items

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

	i		**+**		**i**		**$**

Z → •E **$**
E → •T
E → •E **+** T
T → •**i**
T → •**(** 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 $\longrightarrow$ α•Nβ is in S
- then for each rule N $\longrightarrow \gamma$ in the grammar
  S must also contain N $\longrightarrow$ •γ

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

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

Example: parsing with LR items

	i		+		i		$

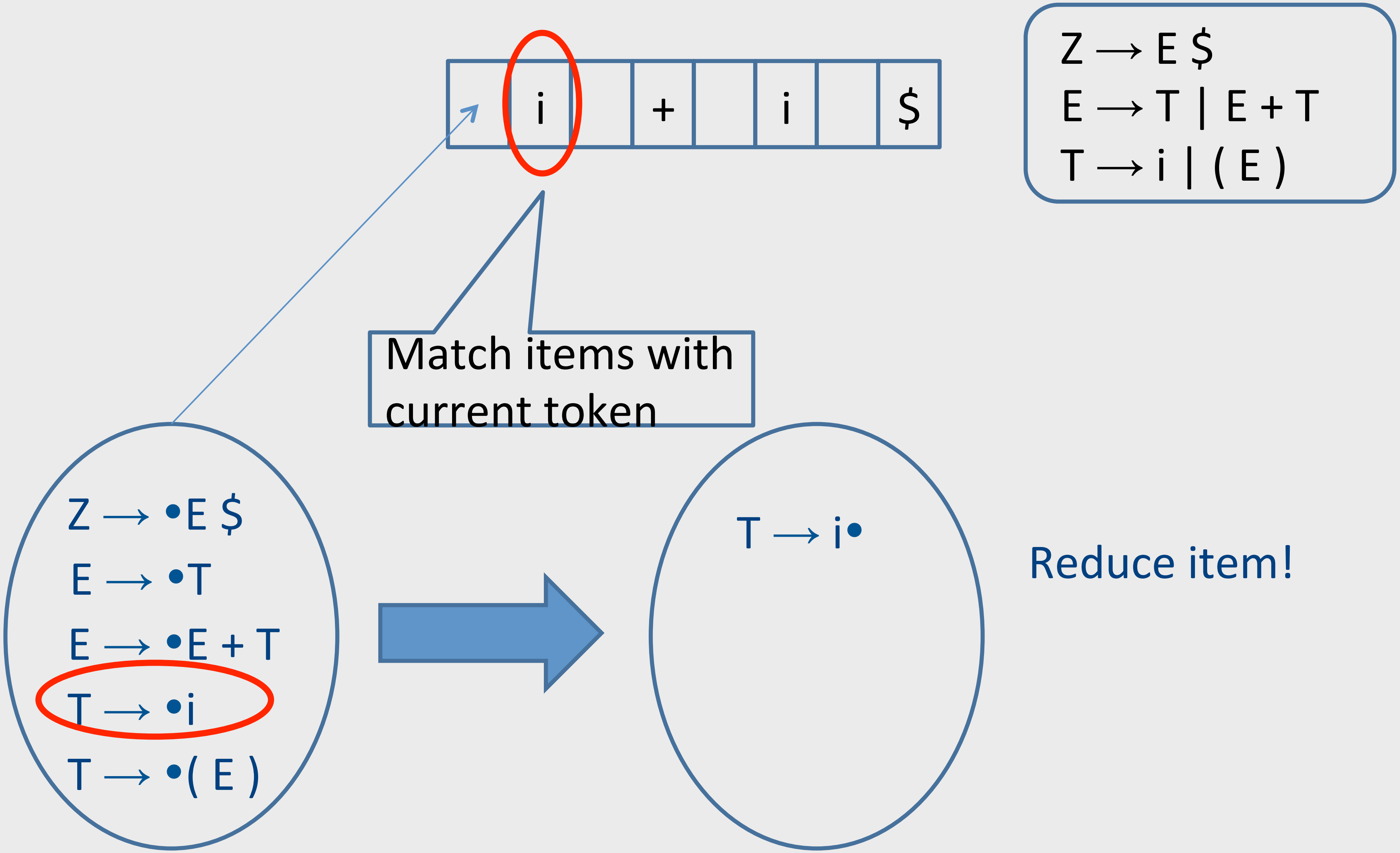
Remember position from which we're trying to reduce

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

Items denote possible future handles

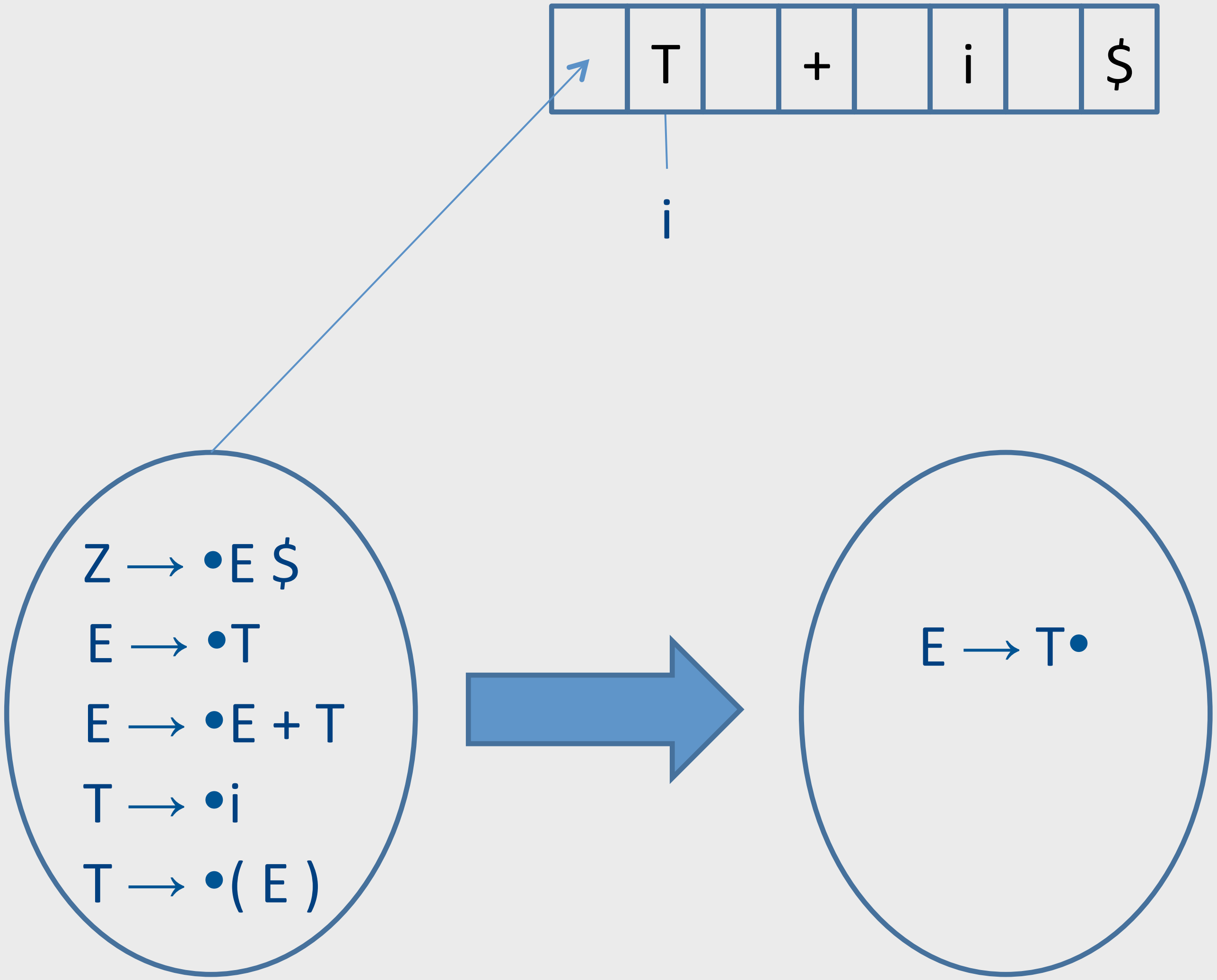
$Z \longrightarrow \bullet E\ \$$
$E \longrightarrow \bullet T$
$E \longrightarrow \bullet E + T$
$T \longrightarrow \bullet i$
$T \longrightarrow \bullet(\ E\ )$

Example: parsing with LR items



Example: parsing with LR items

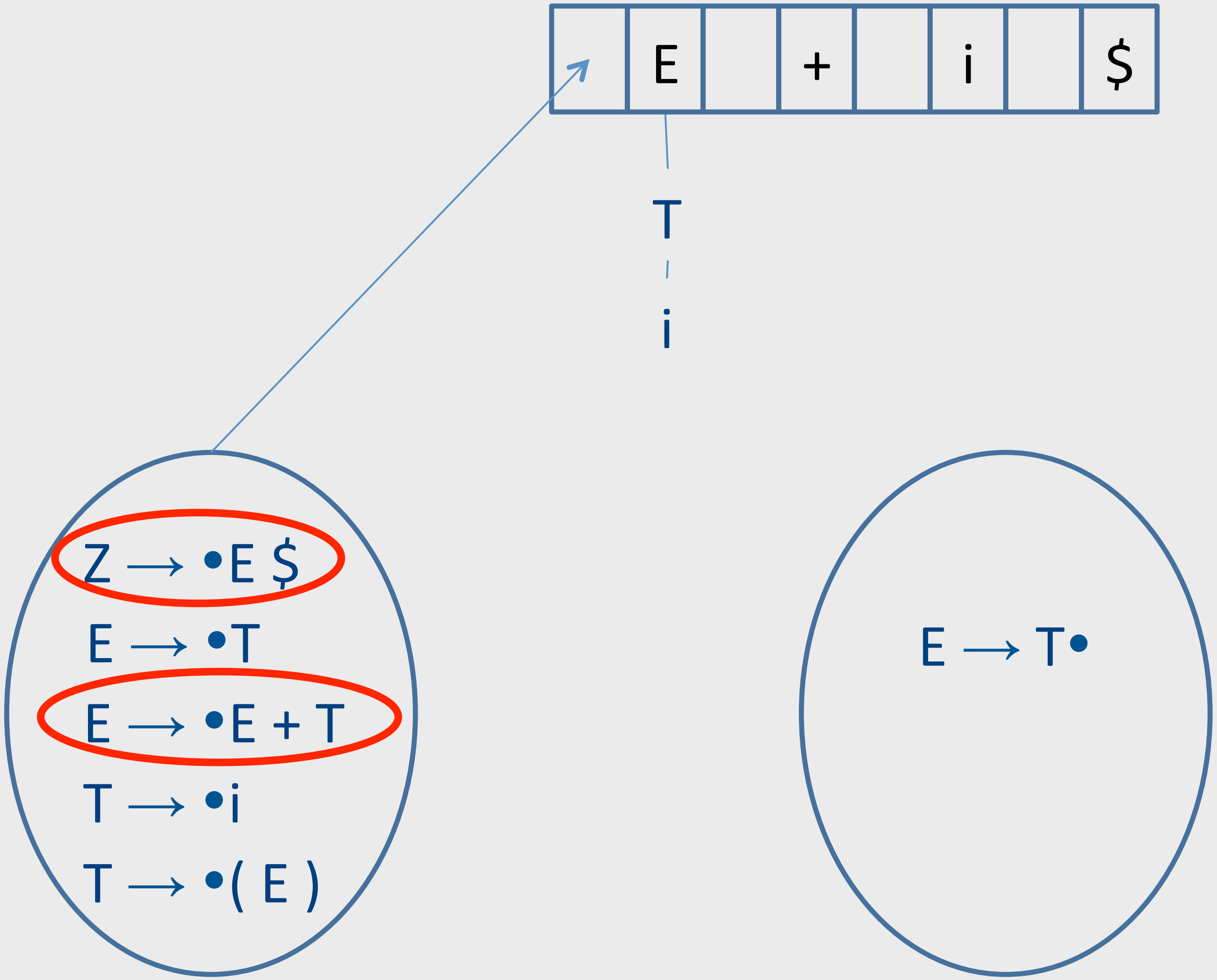
Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)



Reduce item!

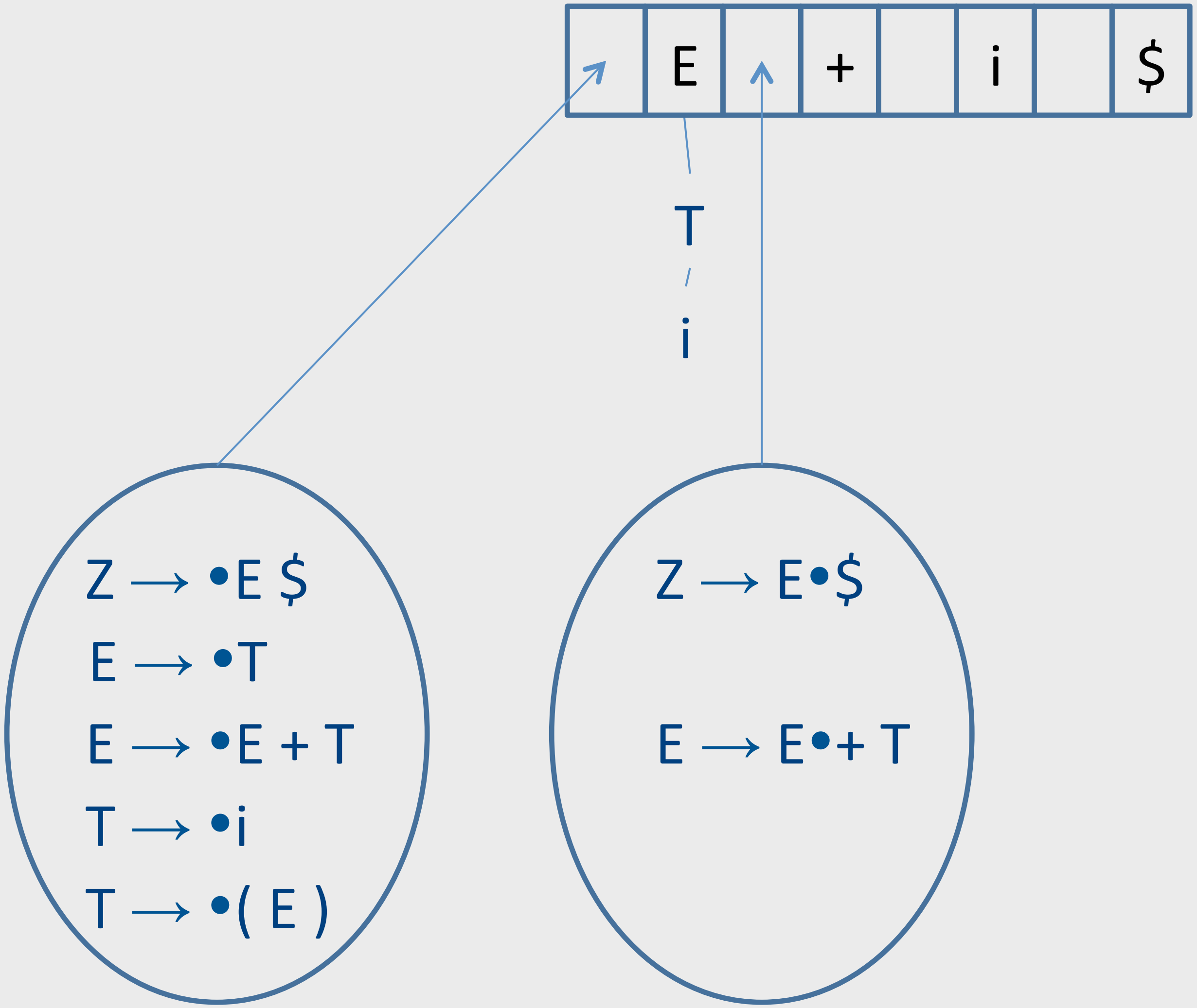
Example: parsing with LR items

Z → E $
E → T | E + T
T → i | (E)



Reduce item!

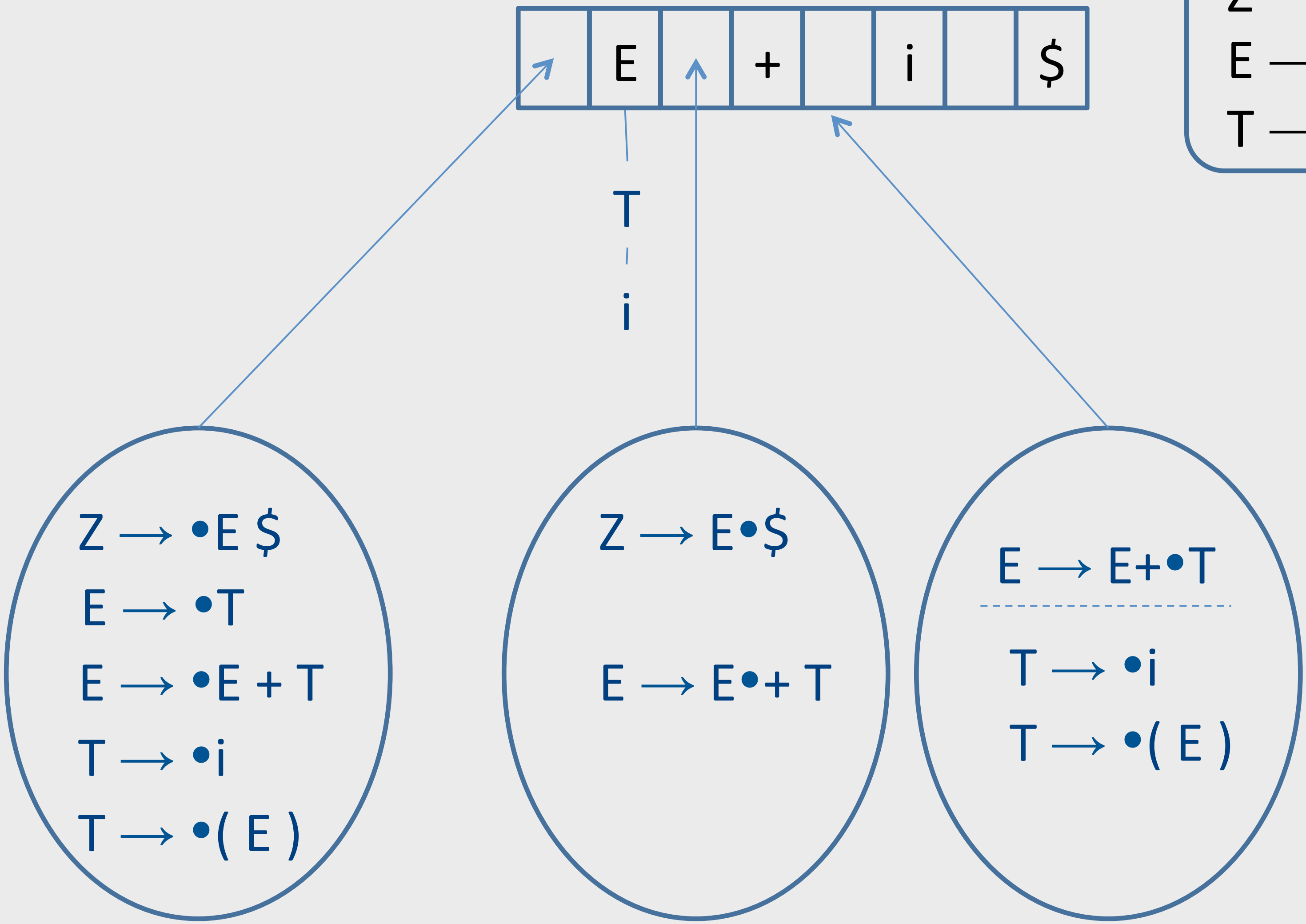
Example: parsing with LR items



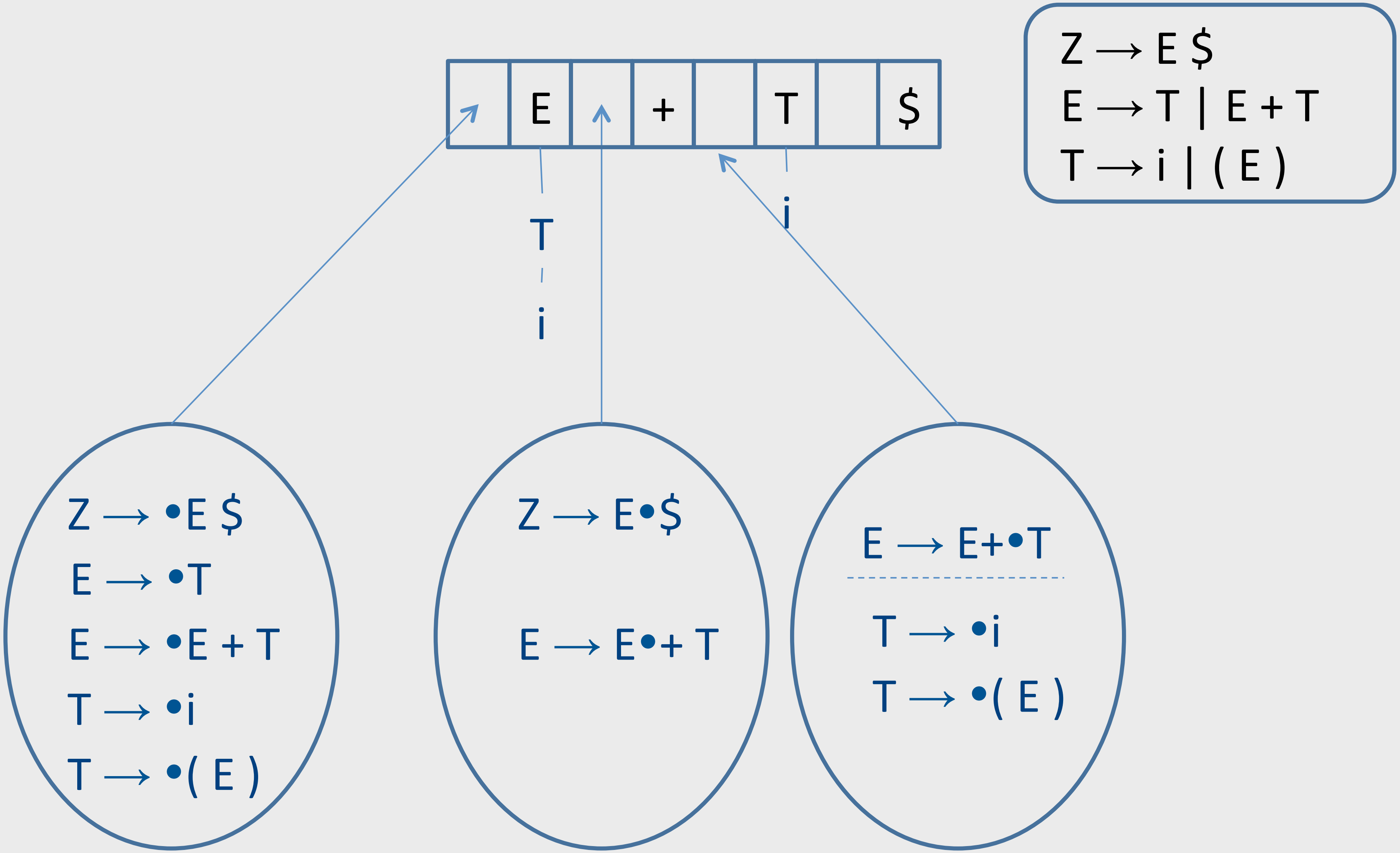
Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)

Example: parsing with LR items

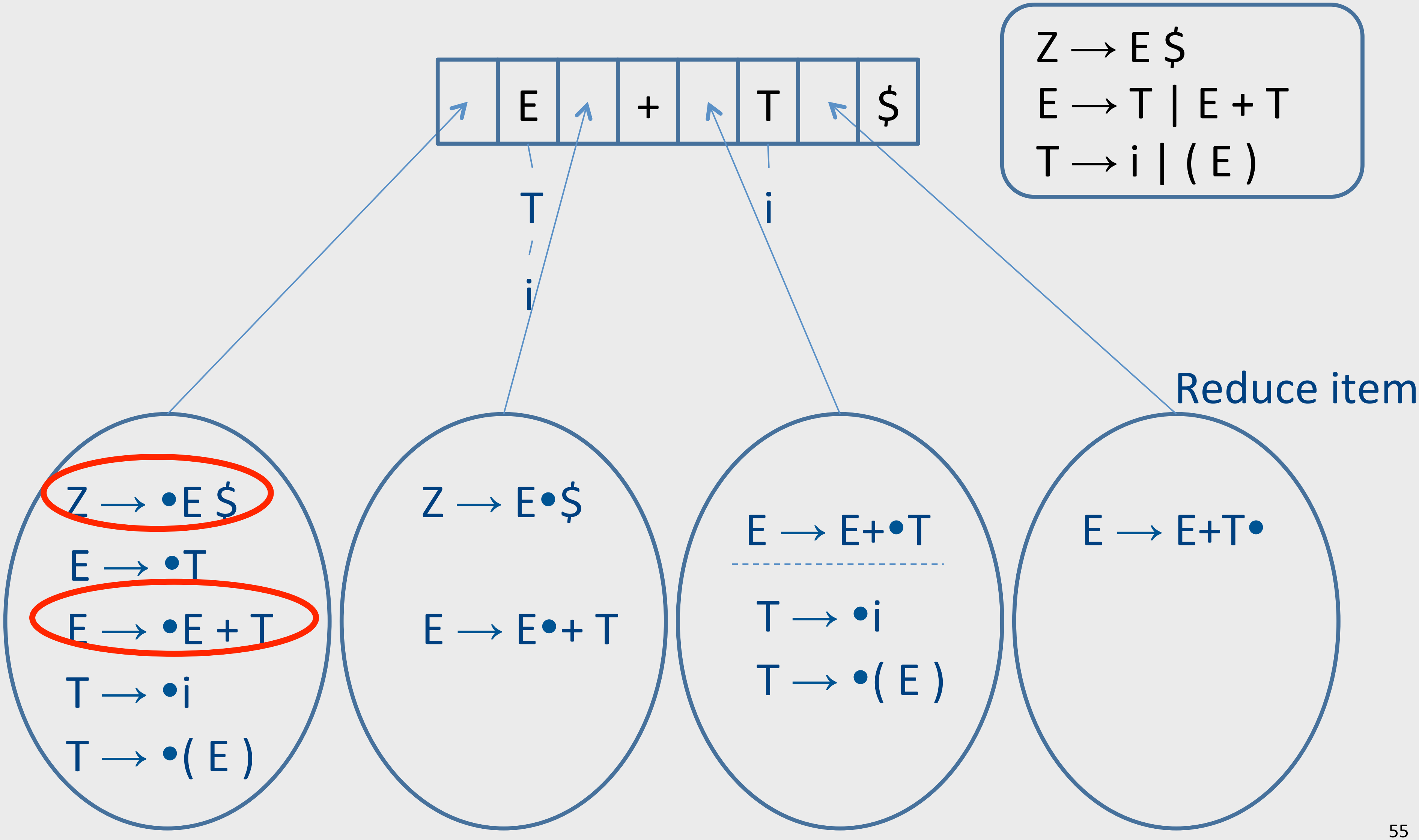
Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)



Example: parsing with LR items

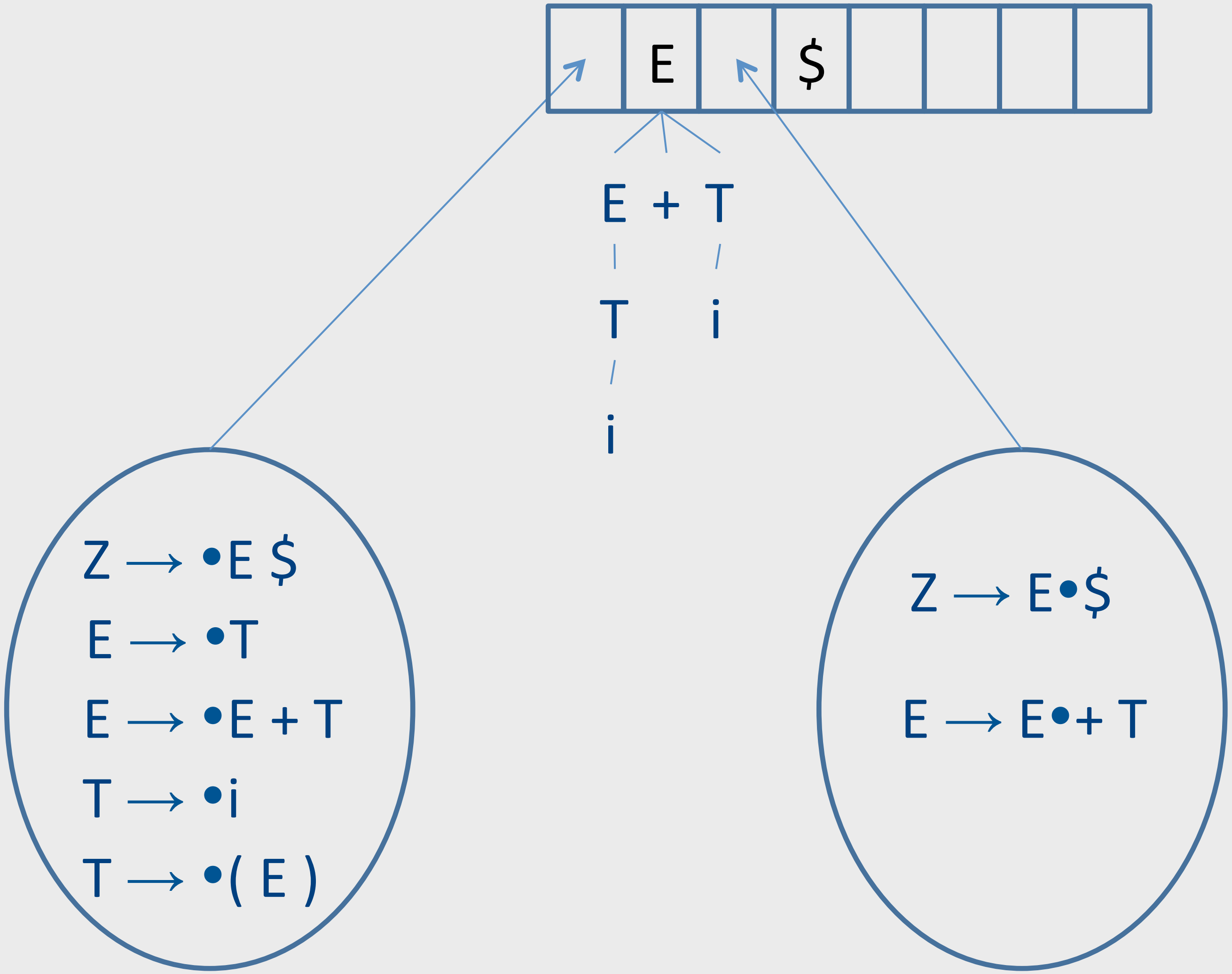


Example: parsing with LR items



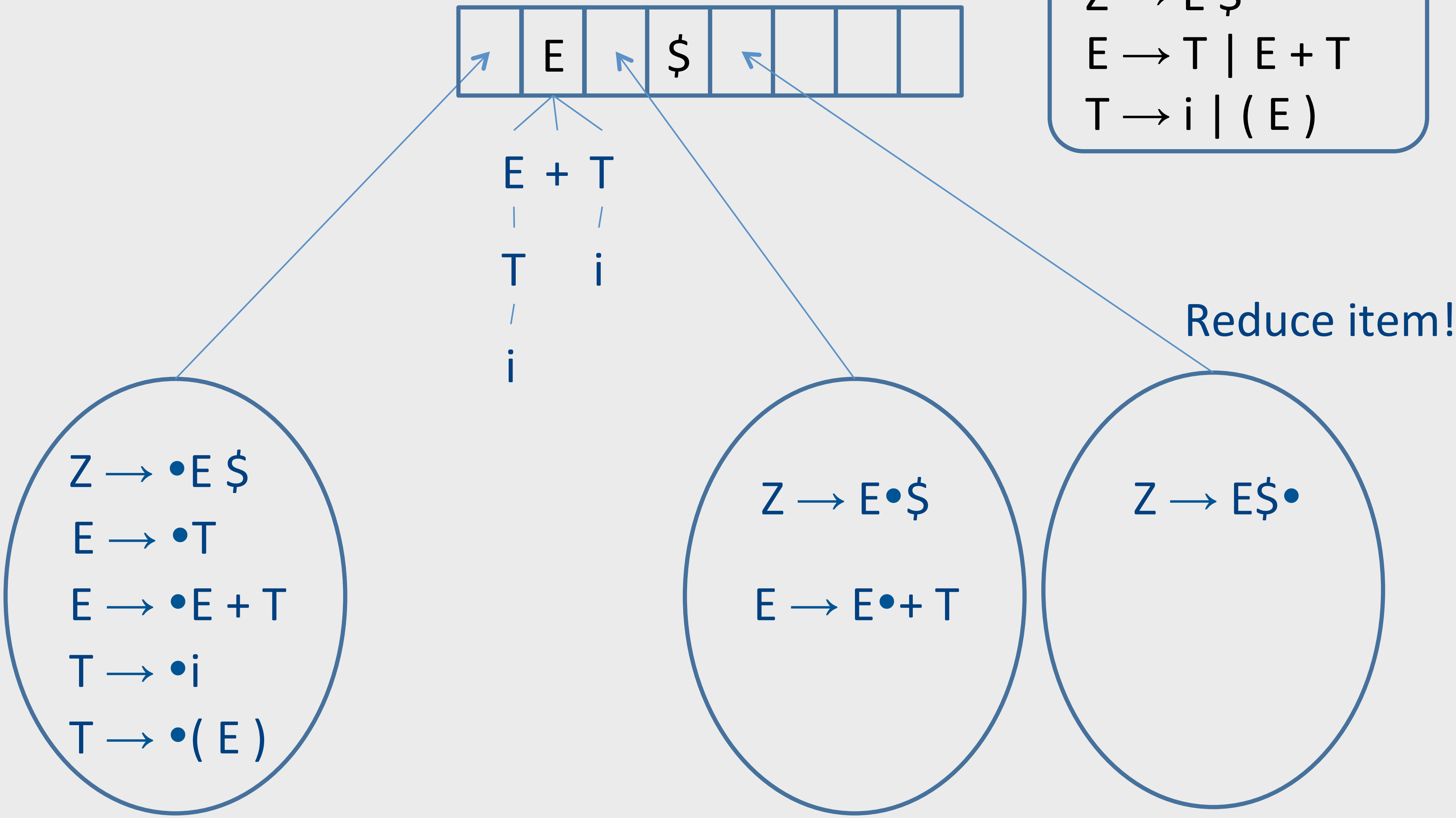
Example: parsing with LR items

Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)

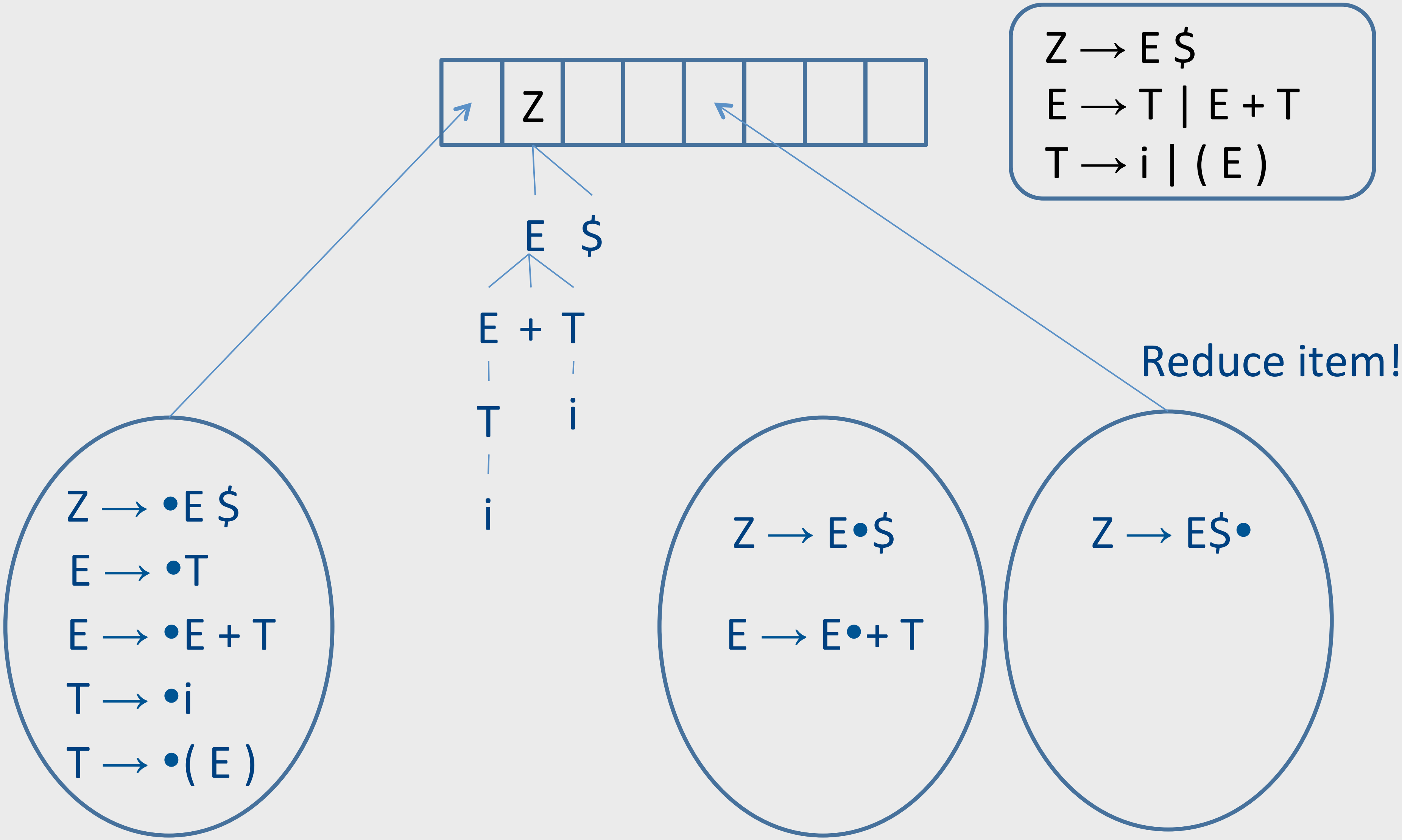


Example: parsing with LR items

Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)



Example: parsing with LR items



GOTO/ACTION tables

empty – error move

GOTO Table

ACTION Table

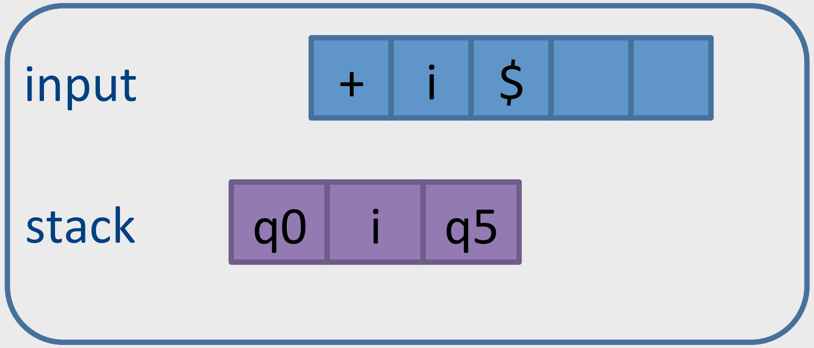
State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q1		q3			q2			shift
q2								Z⟶E$
q3	q5		q7				q4	Shift
q4								E⟶E+T
q5								T⟶i
q6								E⟶T
q7	q5		q7			q8	q6	shift
q8		q3		q9				shift
q9								T⟶E

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

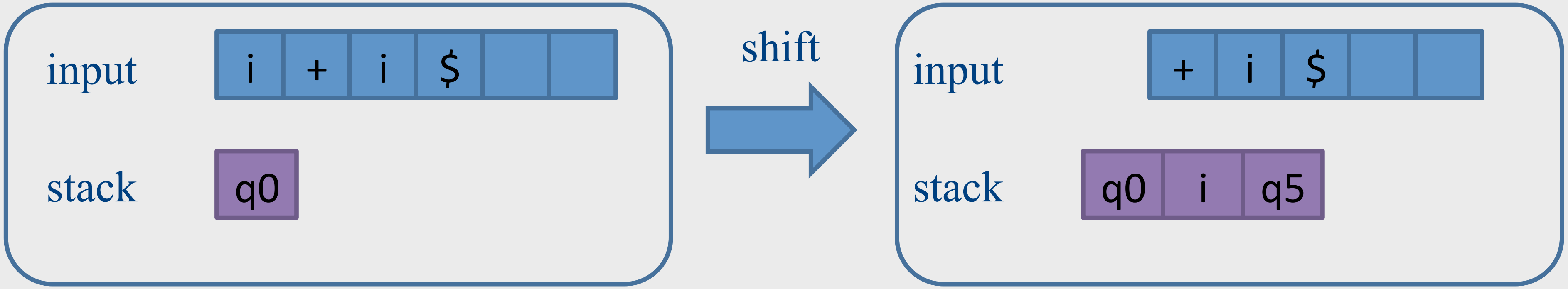
- Input – remainder of text to be processed
- Stack – sequence of pairs N, qi
  - N – symbol (terminal or non-terminal)
  - qi – state at which decisions are made



- Initial stack contains q0

LR(0) pushdown automaton

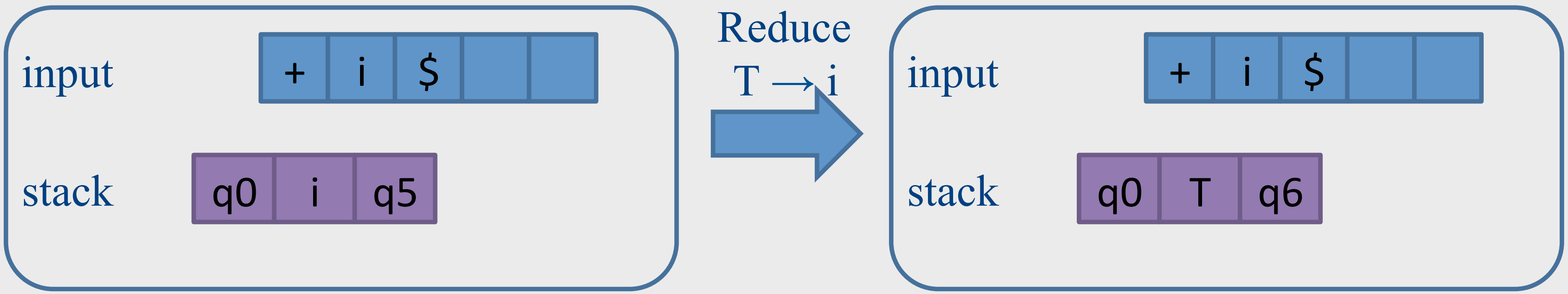
- Two moves: shift and reduce
- Shift move
  - Remove first token from input
  - Push it on the stack
  - Compute next state based on GOTO table
  - Push new state on the stack
  - If new state is error – report error



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



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$

(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		

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		

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

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		

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

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		

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

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		

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

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		

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		

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		

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		

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		

Parsing id+id$

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

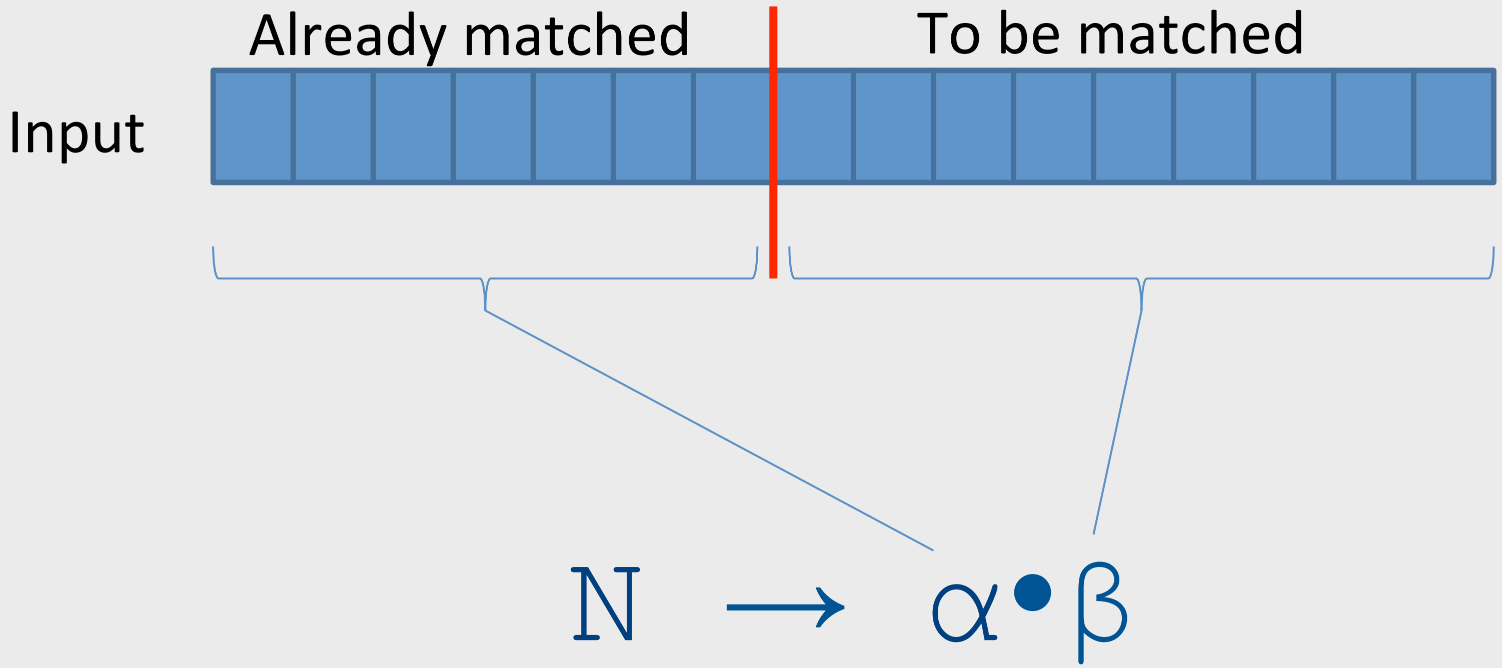
Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4
0 E 1 + 3 T 4	$	r3
0 E 1	$	s2

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		

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



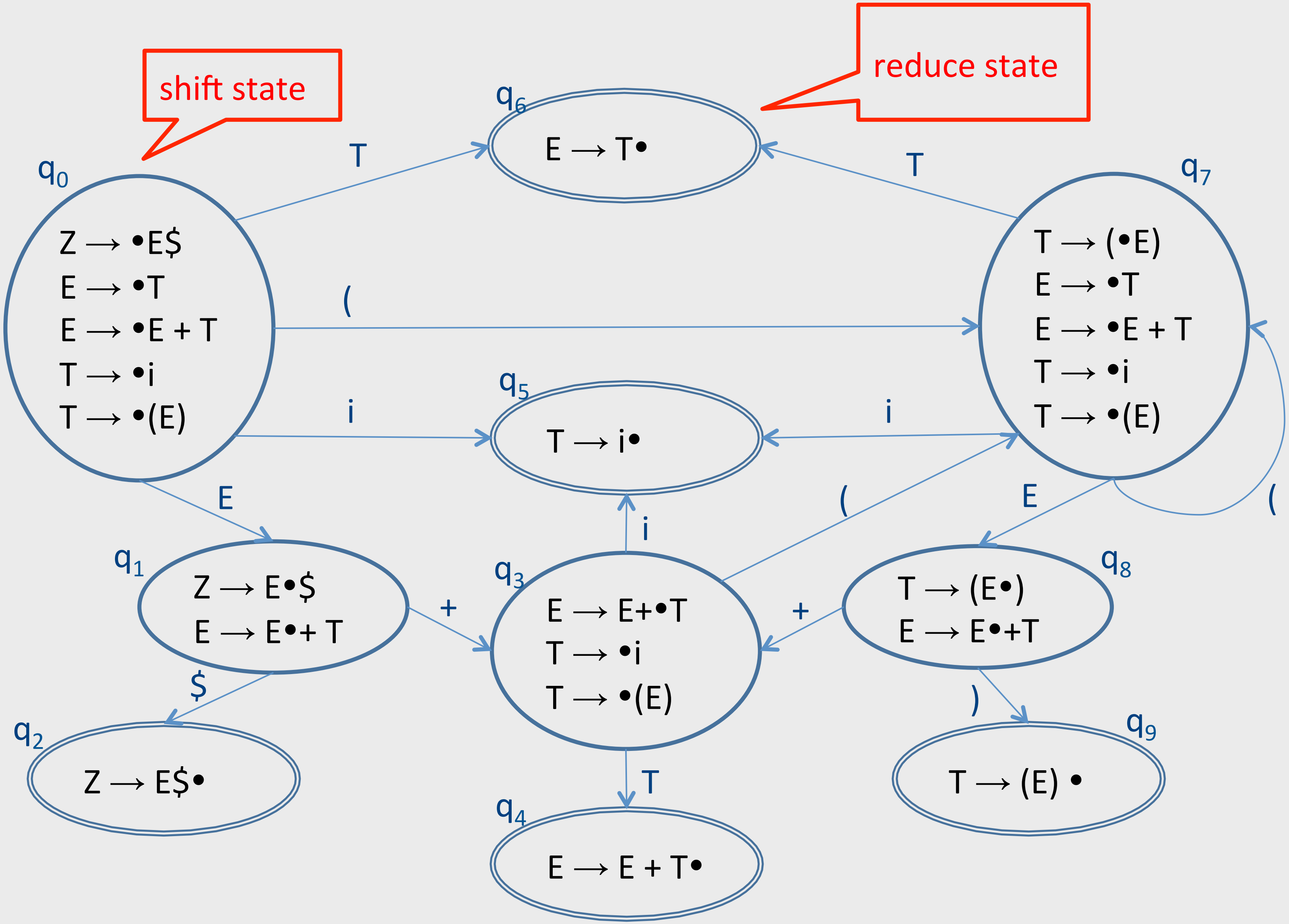
Hypothesis about αβ being a possible handle, so far we've matched α, expecting to see β

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({ $Z \rightarrow \bullet\alpha$ | $Z \rightarrow \alpha$ is in the grammar })
- Next set from a set S and the next symbol X
  - step(S,X) = { $N \rightarrow \alpha X \bullet \beta$ | $N \rightarrow \alpha \bullet X \beta$ in the item set S}
  - nextSet(S,X) = ε-closure(step(S,X))

Operations for transition diagram construction

- Initial = {S'⟶•S$}

- For an item set I
  Closure(I) = Closure(I) ∪
  {X⟶•μ is in grammar| N⟶α•Xβ in I}

- Goto(I, X) = { N⟶αX•β | N⟶α•Xβ in I}

Initial example

- Initial = {S → •E $}

Grammar

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

Closure 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 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

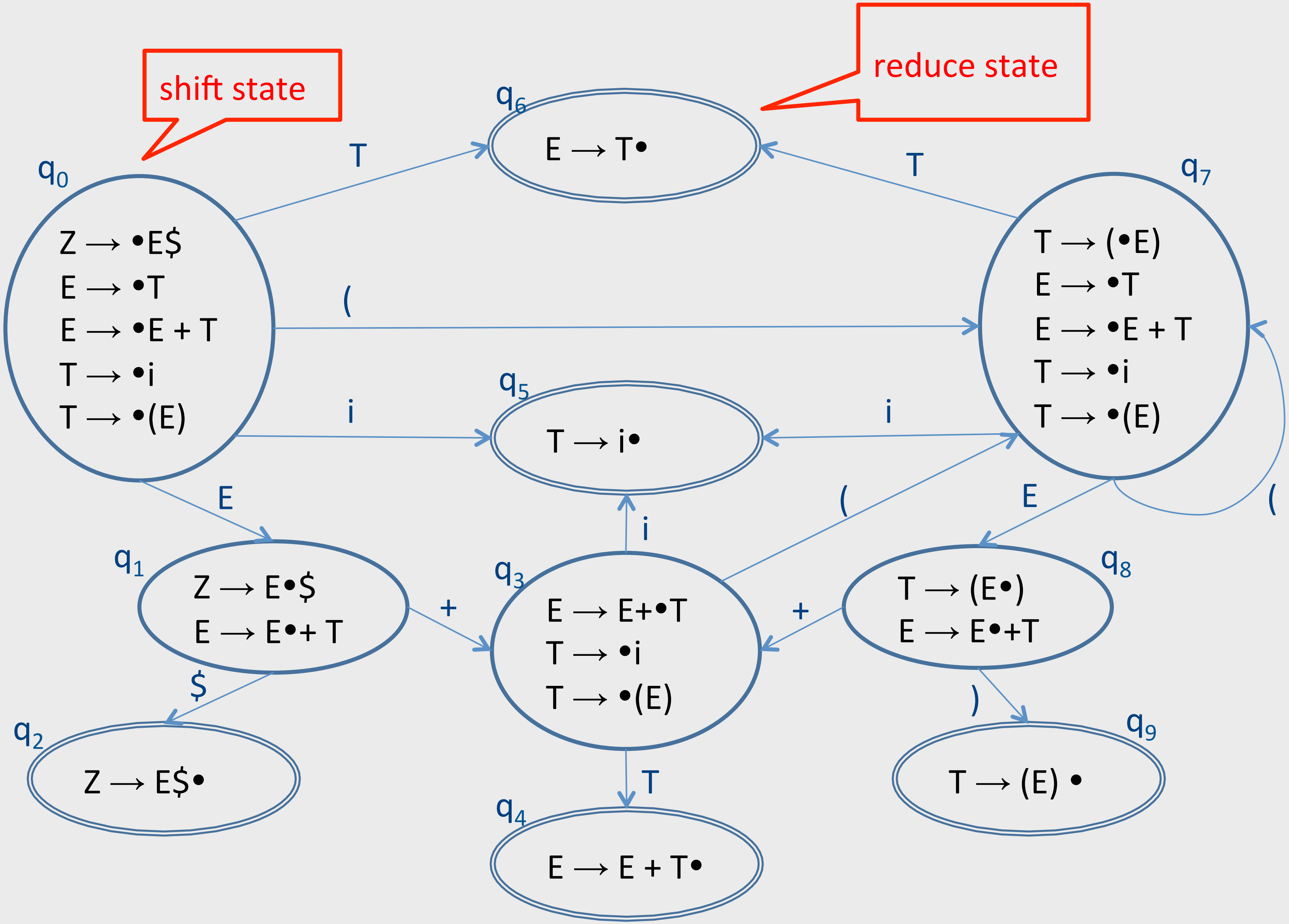
  T → •id

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

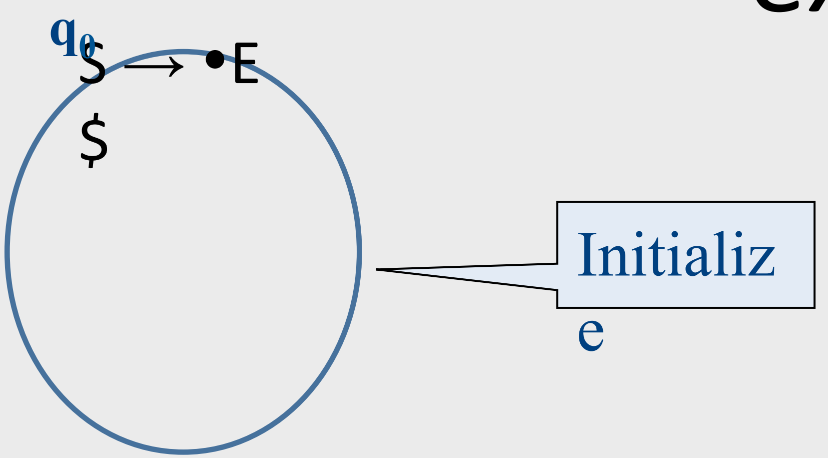
Constructing the transition diagram

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

LR(0) automaton example



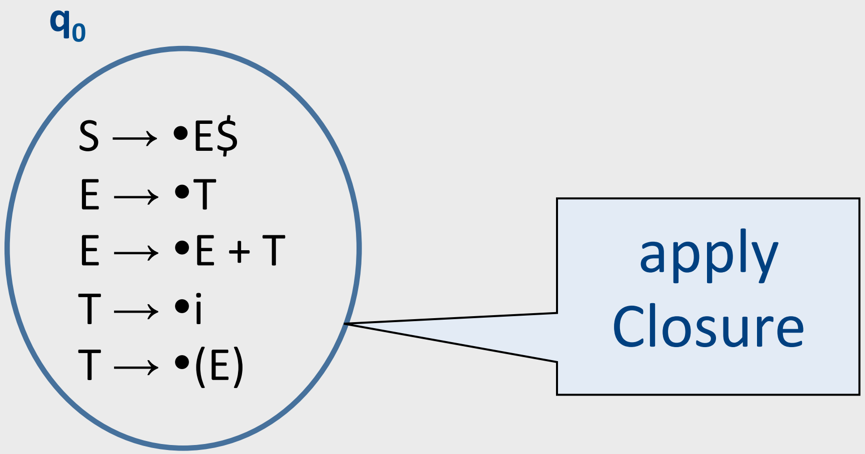
Automaton construction example



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

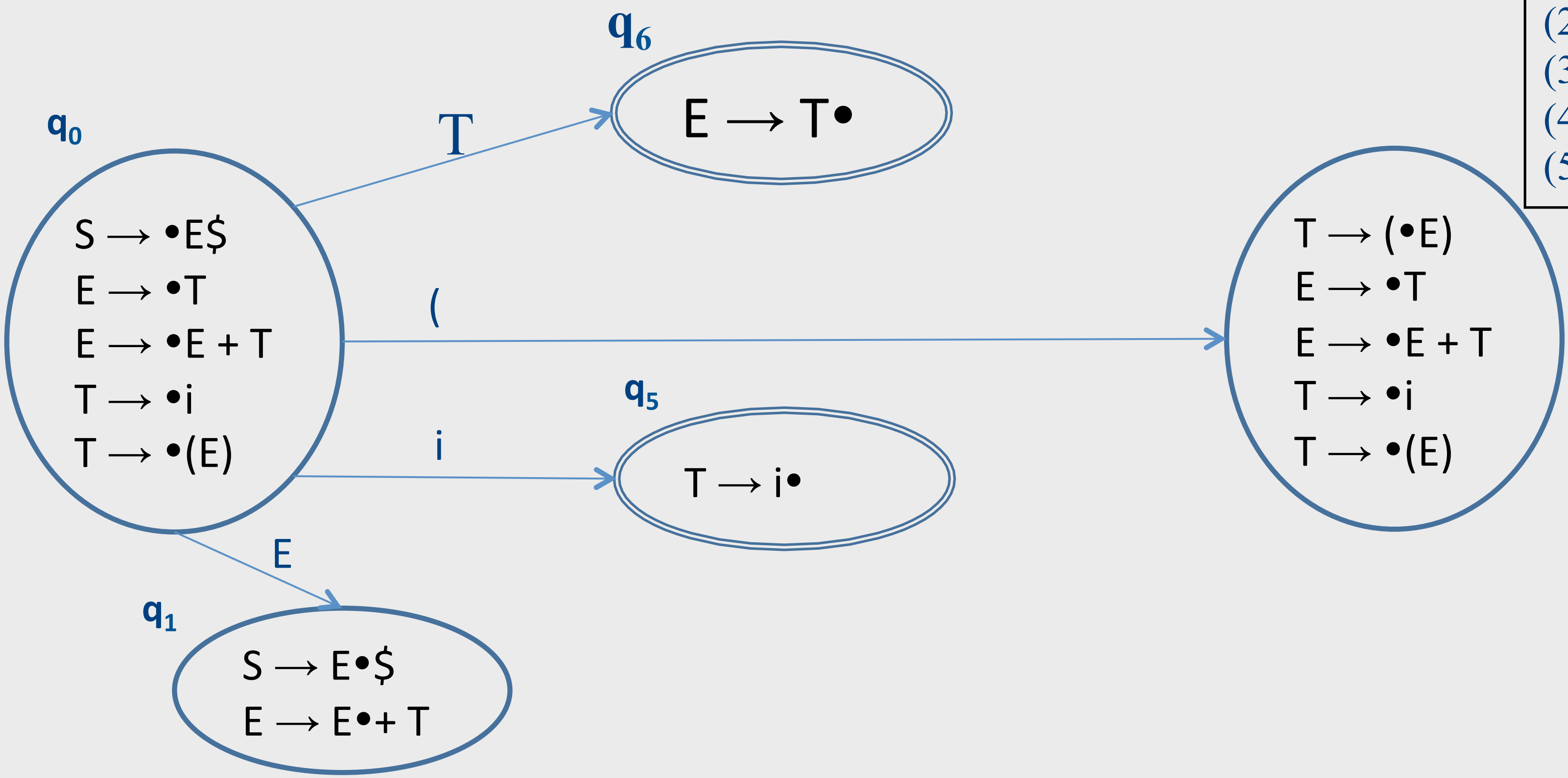
Automaton construction example

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

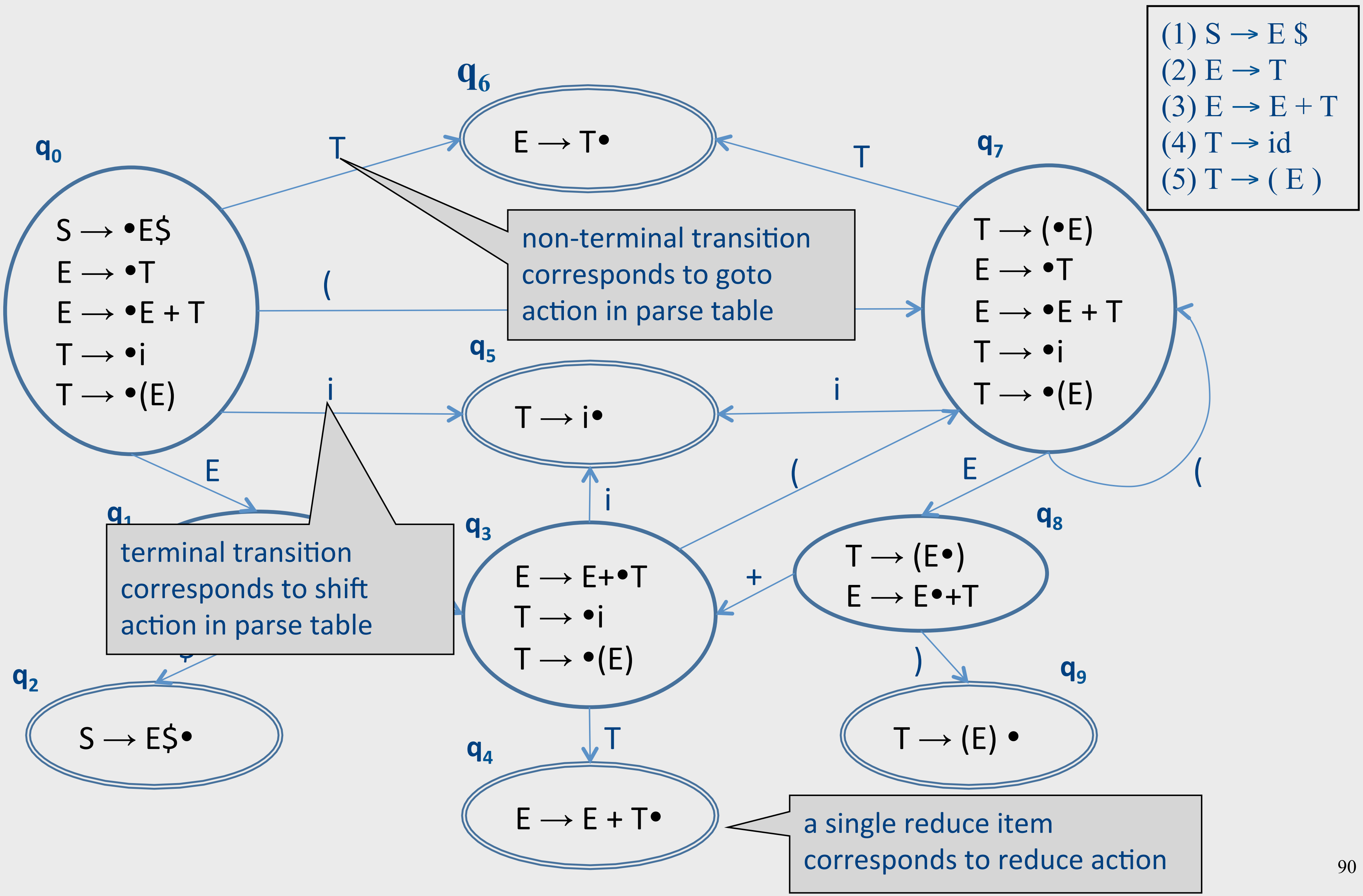


Automaton construction example

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



Automaton construction example

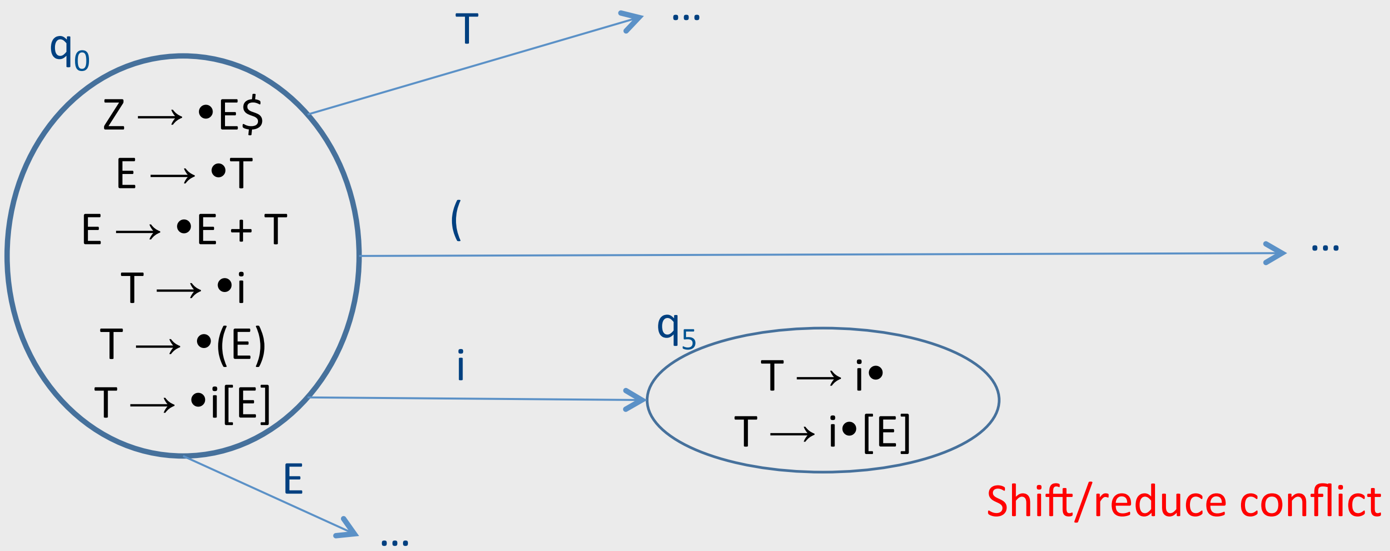


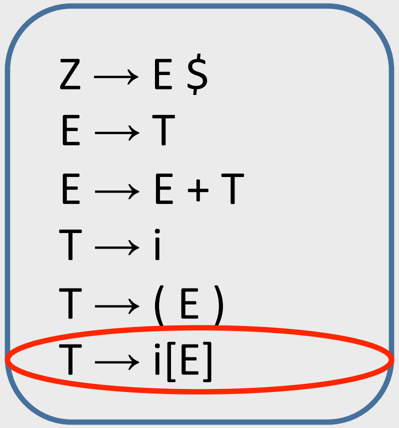
Are we done?

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

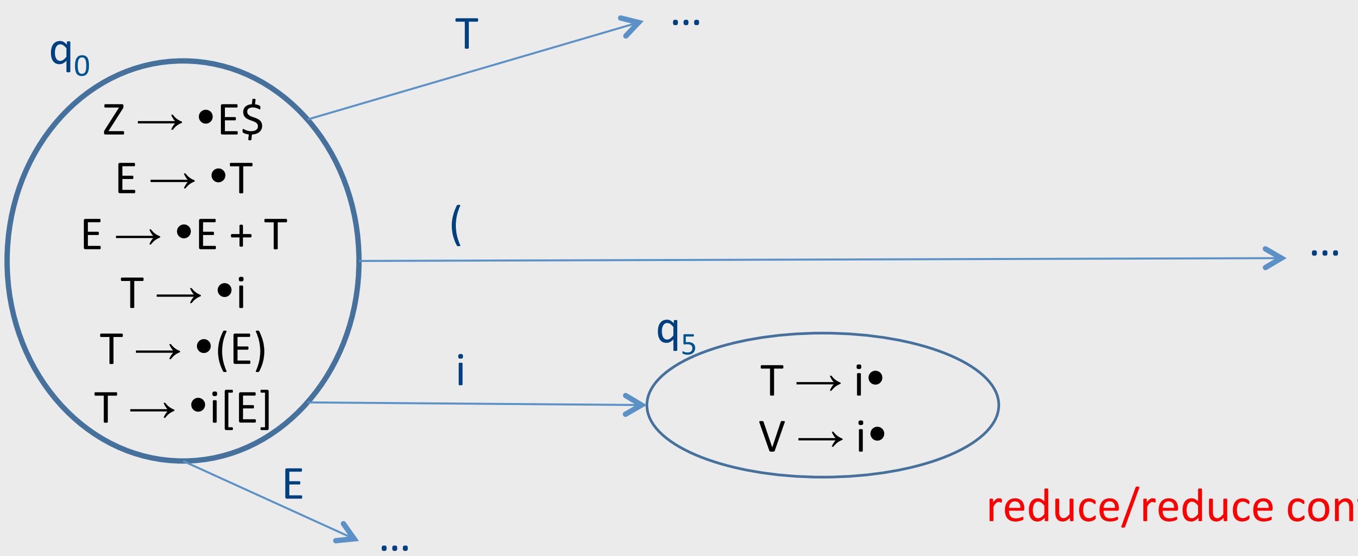
- Cannot make a deterministic ACTION table for every grammar

LR(0) conflicts





LR(0) conflicts



reduce/reduce conflict

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

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
  - $A \rightarrow \varepsilon\bullet$ can always be predicted from $P \rightarrow \alpha\bullet A\beta$

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 ➔* βNb.
  - 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)

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

SLR – use 1 token look-ahead

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, which is unlikely in practice

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

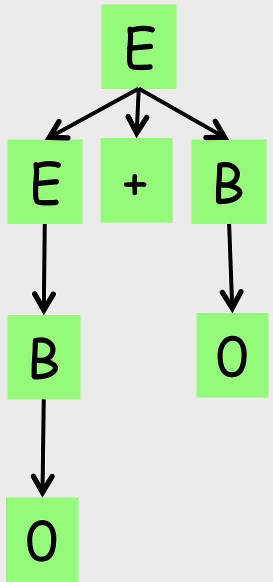
Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method
  - Usually, not used in practice (though might change)
- Top-Down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-Up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

Question

- Why do we need the stack?
- Why can we use FSM to make parsing decisions?

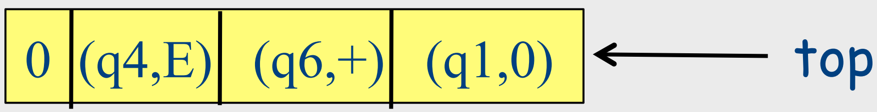
Why do we need a stack?



- Suppose so far we have discovered E → B → 0 and gather information on "E +".
- In the given grammar this can only mean

  E → E + ● B

- Suppose state q_6 represents this possibility.
- Now, the next token is 0, and we need to ignore q_6 for a minute, and work on B → 0 to obtain E+B.
- Therefore, we push q_6 to the stack, and after identifying B, we pop it to continue.



See you next time

- Here!

GOTO/ACTION table

st	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 )
```

top is on the right

Stack	Input	Action
q0	i + i $	s5
q0 i q5	+ i $	r4
q0 T q6	+ i $	r2
q0 E q1	+ i $	s3
q0 E q1 + q3	i $	s5
q0 E q1 + q3 i q5	$	r4
q0 E q1 + q3 T q4	$	r3
q0 E q1	$	s2
q0 E q1 $ q2		r1
q0 Z		

Example

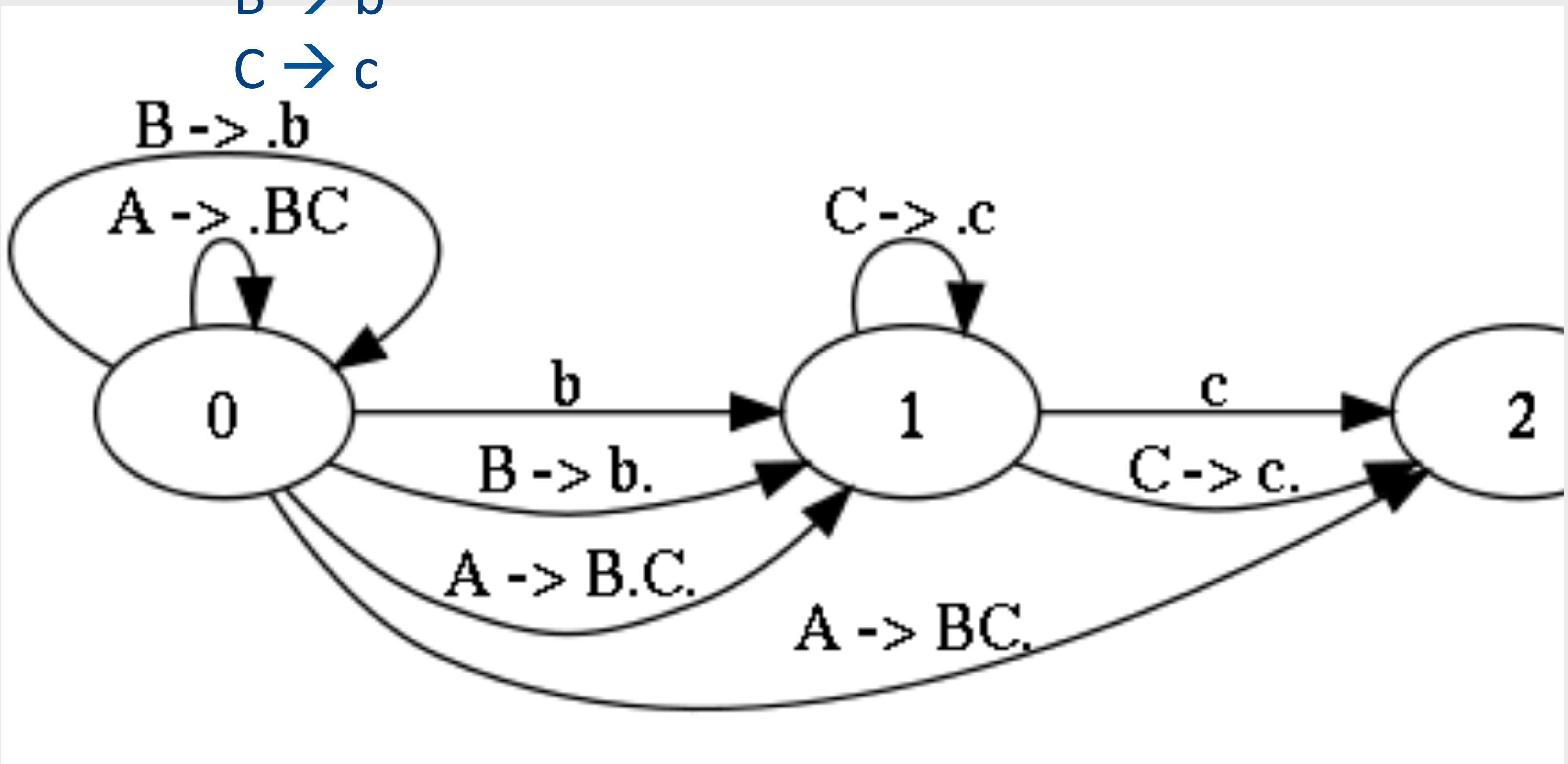
S_0		S_1	
$S' \to \bullet E \quad ,0$ $E \to \bullet E + E \quad ,0$ $E \to \bullet n \quad ,0$	n	$\mathbf{E \to n\bullet \quad ,0}$ $S' \to E\bullet \quad ,0$ $E \to E \bullet +E \quad ,0$	
		S_2	
	+	$E \to E + \bullet E \quad ,0$ $E \to \bullet E + E \quad ,2$ $E \to \bullet n \quad ,2$	
		S_3	
	n	$\mathbf{E \to n\bullet \quad ,2}$ $\mathbf{E \to E + E\bullet \quad ,0}$ $E \to E \bullet +E \quad ,2$ $\mathbf{S' \to E\bullet \quad ,0}$	

FIGURE 1. Earley sets for the grammar $E \to E + E \mid n$ and the input `n + n`. Items in bold are ones which correspond to the input's derivation.

Earley with Pictures

Grammar: A → BC
B → b
C → c

Input: bc



Earley Parsing in Pictures

Grammar: S → E

E → T + id | id

T → E

Input: id + id + id