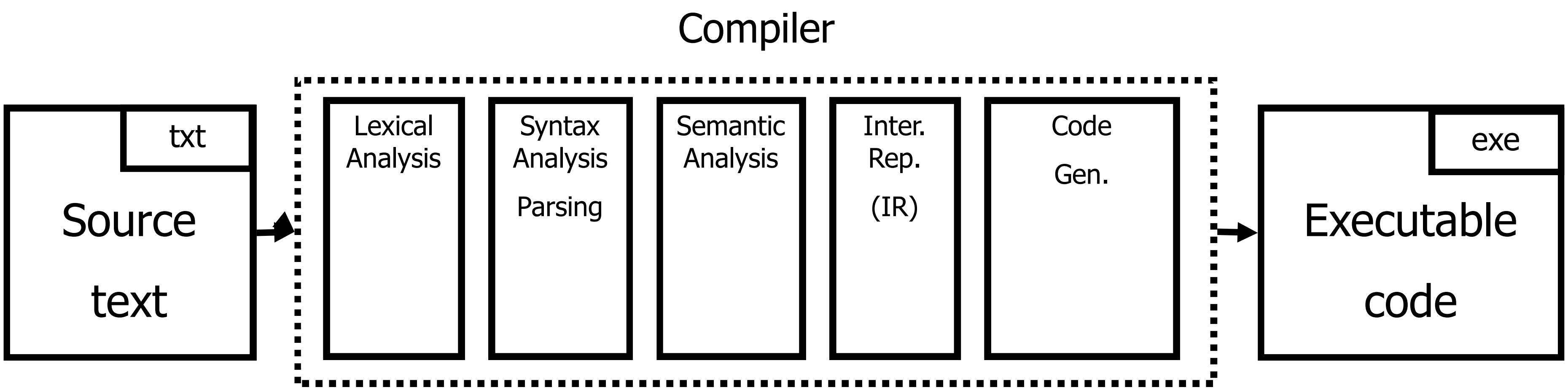
Lecture 04 – Syntax analysis: top-down and bottom-up parsing

THEORY OF COMPILATION

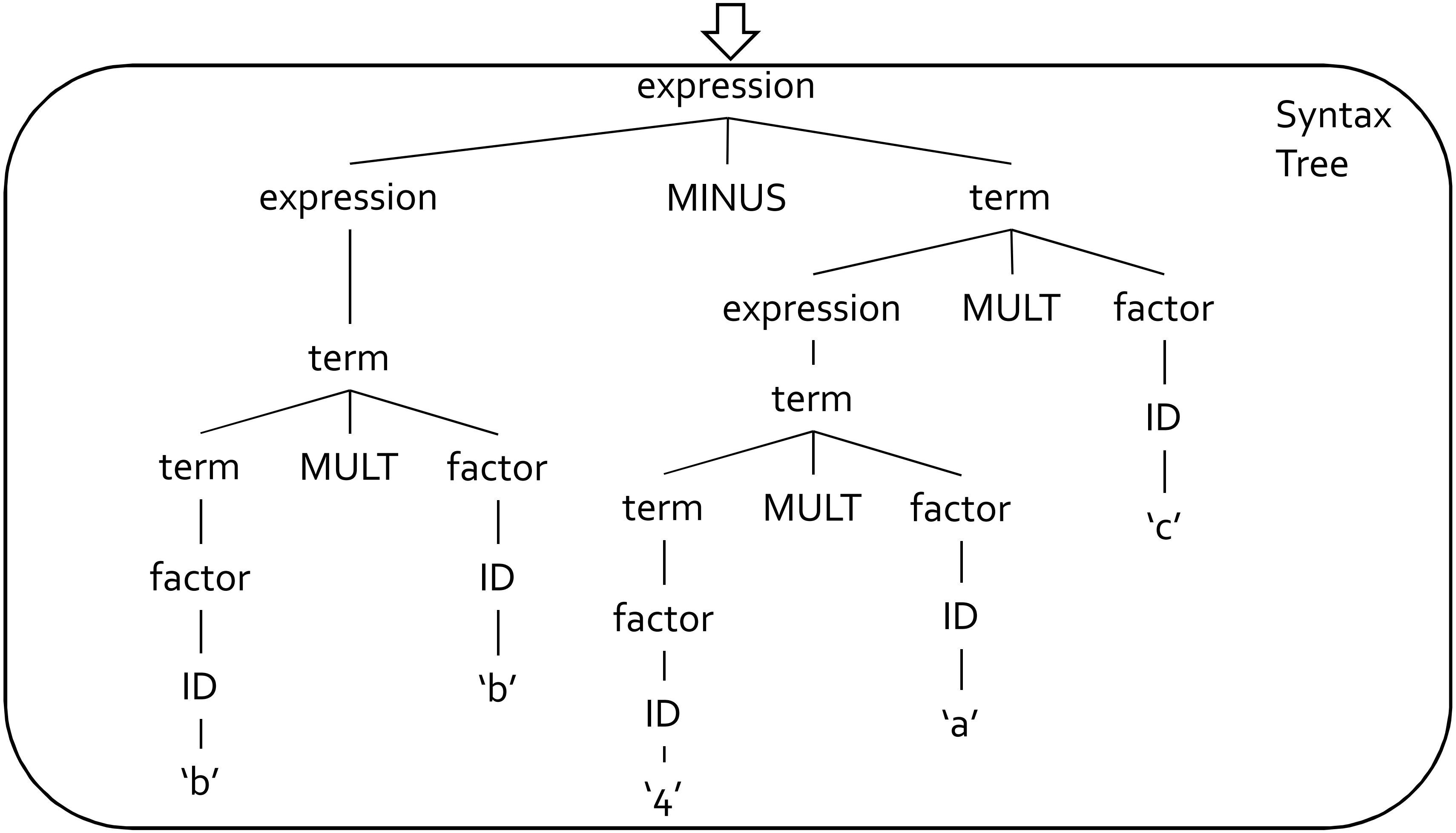
Eran Yahav

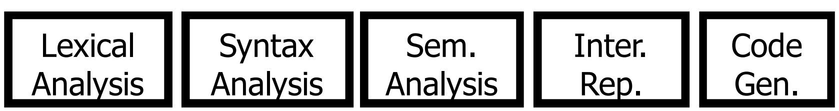
You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Inter.
Rep.
(IR)
Code
Gen.
exe
Executable
code

Last week: from tokens to AST

<ID,"x"> <EQ> <ID,"b"> <MULT> <ID,"b"> <MINUS> <INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">





Last week: context free grammars

G = (V,T,P,S)

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

Example

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

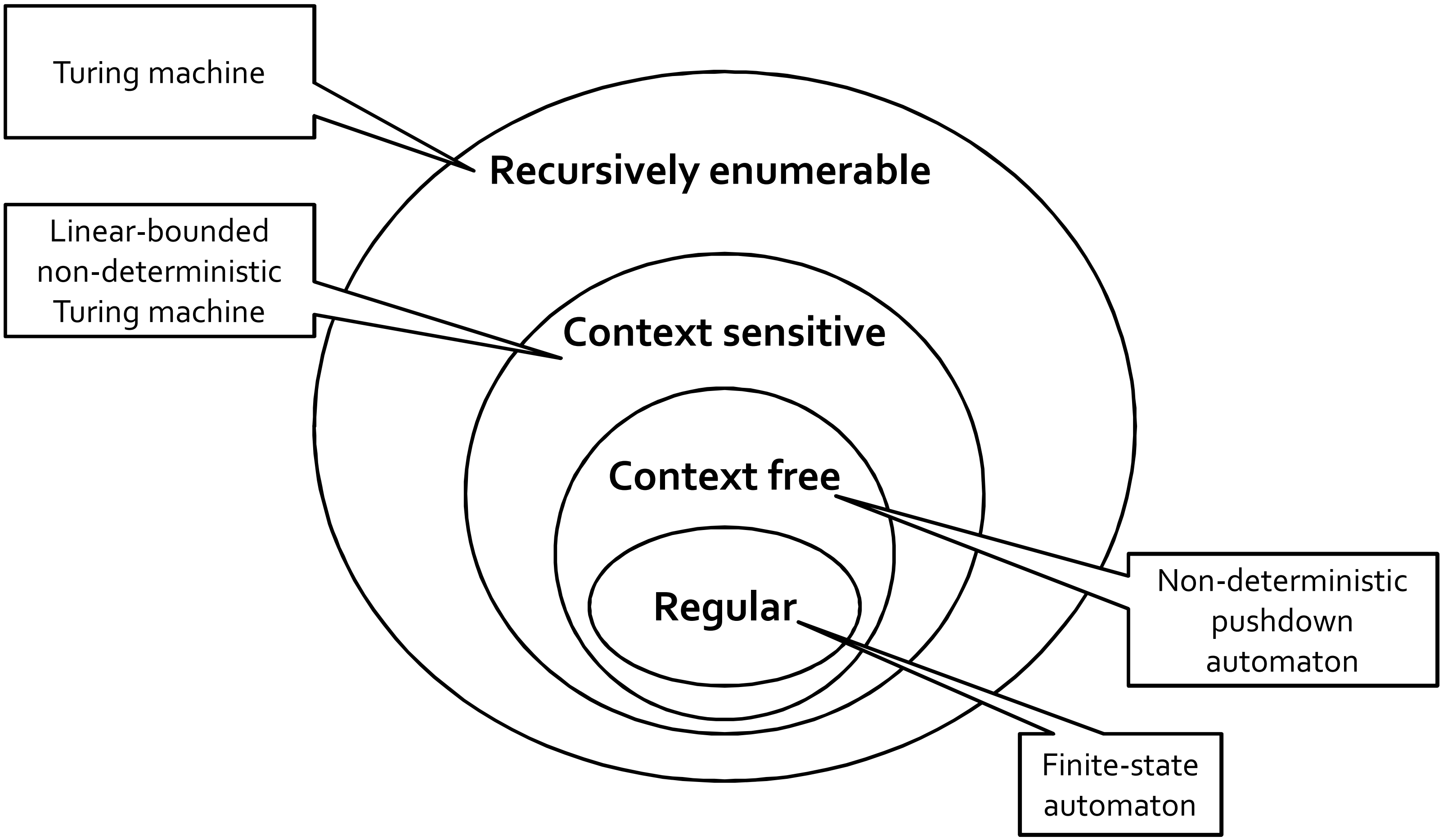
Last week: parsing

- A context free language can be recognized by a non-deterministic pushdown automaton

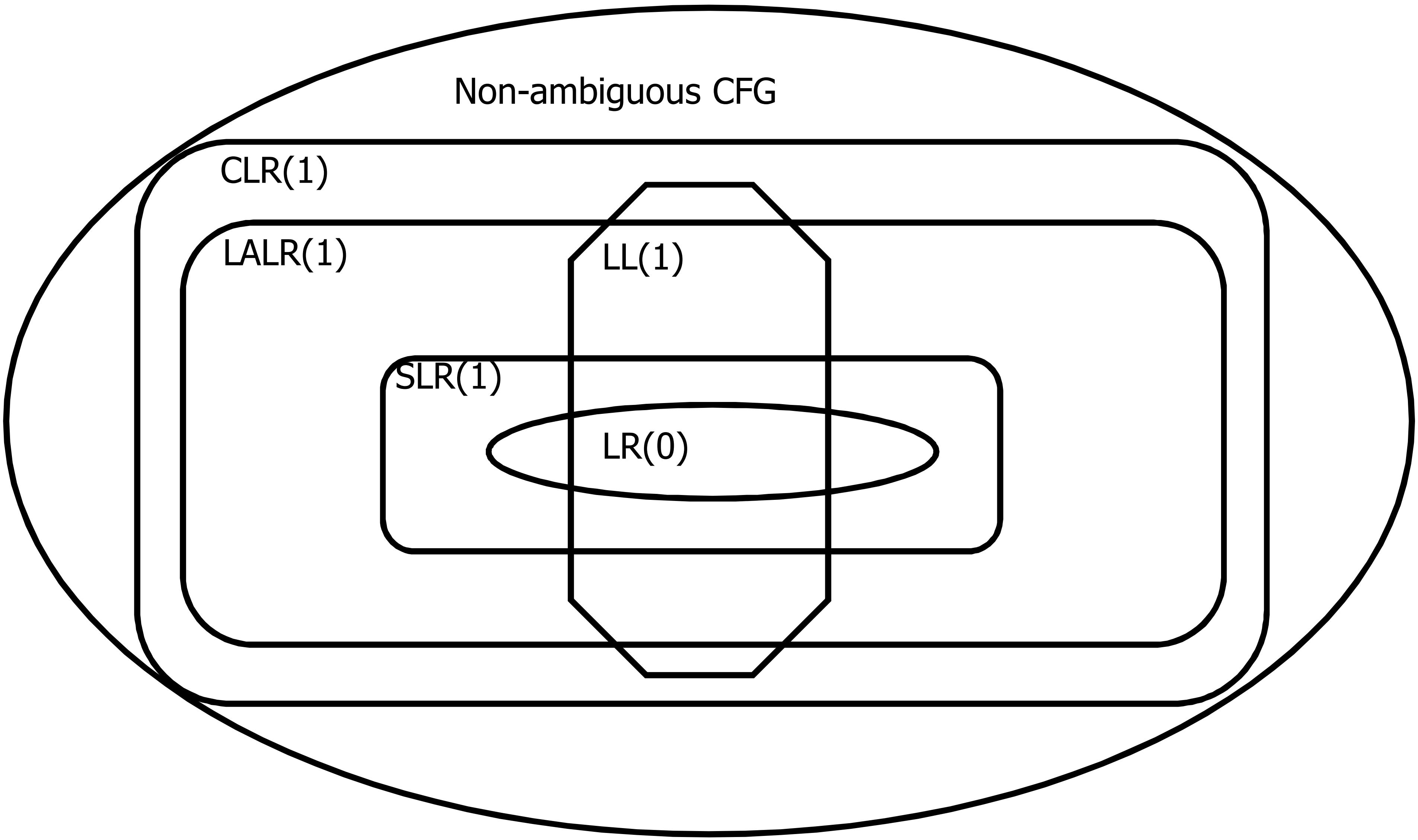
- Parsing can be seen as a search problem
  - Can you find a derivation from the start symbol to the input word?
  - Easy (but very expensive) to solve with backtracking

- We want efficient parsers
  - Linear in input size
  - Deterministic pushdown automata
  - We will sacrifice generality for efficiency

Chomsky Hierarchy



Grammar Hierarchy



LL(k) Parsers

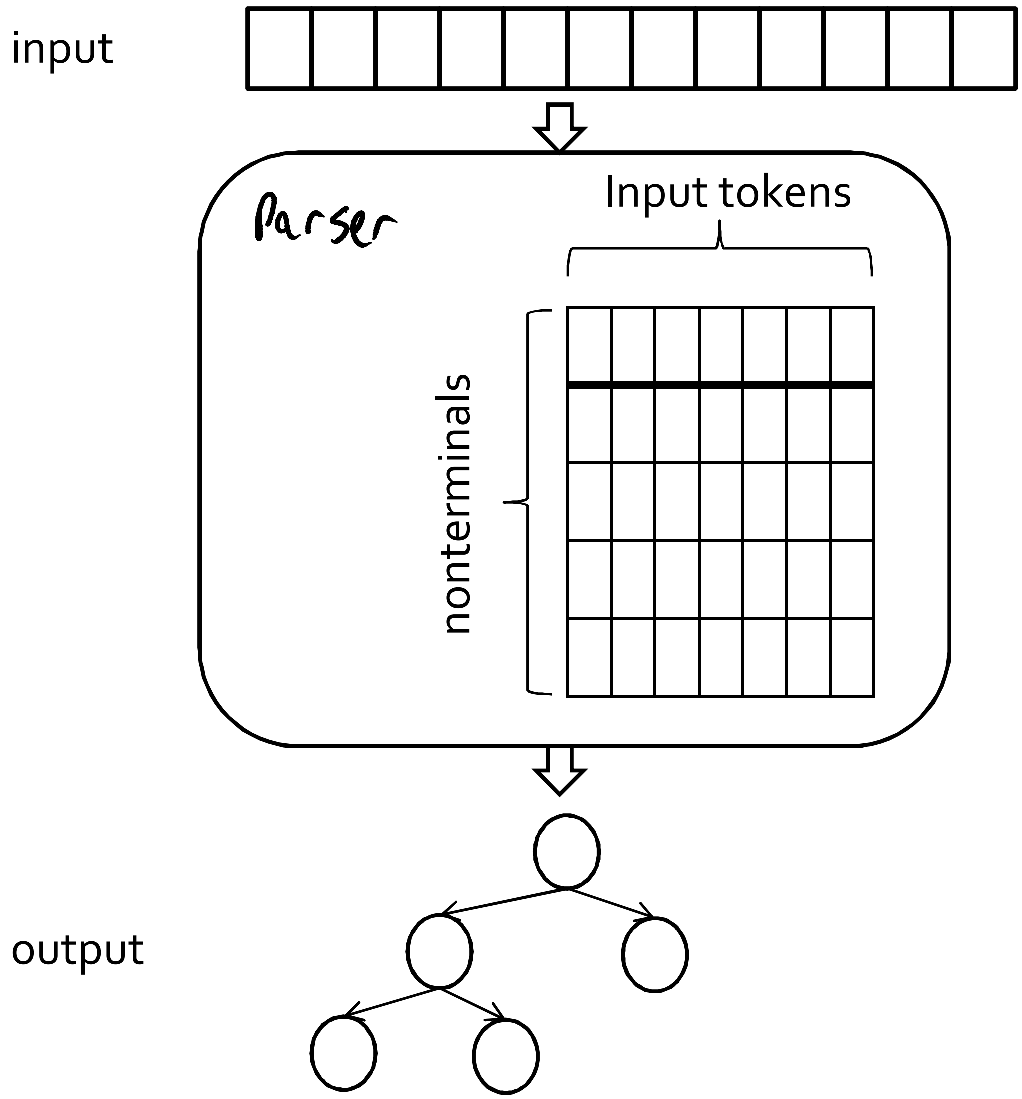
- Manually constructed
  - Recursive Descent

- Generated
  - Uses a pushdown automaton
  - Does not use recursion

LL(k) parsing with pushdown automata

- Pushdown automaton uses
  - Prediction stack
  - Input stream
  - Transition table
    - nonterminals x tokens -> production alternative
    - Entry indexed by nonterminal N and token t contains the alternative of N that must be predicated when current input starts with t

LL(k) parsing with pushdown automata



LL(k) parsing with pushdown automata

- Two possible moves
  - Prediction
    - When top of stack is nonterminal N, pop N, lookup table[N,t]. If table[N,t] is not empty, push table[N,t] on prediction stack, otherwise – syntax error
  - Match
    - When top of prediction stack is a terminal T, must be equal to next input token t. If (t == T), pop T and consume t. If (t ≠ T) syntax error

- Parsing terminates when prediction stack is empty. If input is empty at that point, success. Otherwise, syntax error

Example transition table

(1) E → LIT

(2) E → (E OP E)

(3) E → not E

(4) LIT → true

(5) LIT → false

(6) OP → and

(7) OP → or

(8) OP → xor

Which rule should be used

Input tokens

Nonterminals

	(	)	not	true	false	and	or	xor	$
E	2		3	1	1				
LIT				4	5				
OP						6	7	8	

Simple 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

Stack top on the left

Simple Example

abcbb$

A → aAb | c

Input suffix	**Stack content**	**Move**
abcbb$	A$	predict(A,a) = A → aAb
abcbb$	aAb$	match(a,a)
bcbb$	Ab$	predict(A,b) = ERROR

	a	**b**	**c**
A	A → aAb		A → c

Error Handling and Recovery

x = a * (p+q * (-b * (r-s);

- Where should we report the error?
- The valid prefix property
- Recovery is tricky
  - Heuristics for dropping tokens, skipping to semicolon, etc.

Error Handling in LL Parsers

c$

S → a c | b S

Input suffix	Stack content	Move
c$	S$	predict(S,c) = ERROR

- Now what?
  - Predict bS anyway "missing token b inserted in line XXX"

	a	b	c
S	S → a c	S → bS	

Error Handling in LL Parsers

c$

S → a c | b S

Input suffix	Stack content	Move
bc$	S$	predict(b,c) = S → bS
bc$	bS$	match(b,b)
c$	S$	Looks familiar?

- Result: infinite loop

	a	b	c
S	S → a c	S → bS	

Error Handling

- Requires more systematic treatment
- Enrichment
  - Acceptable-set method
  - Not part of course material

Summary so far

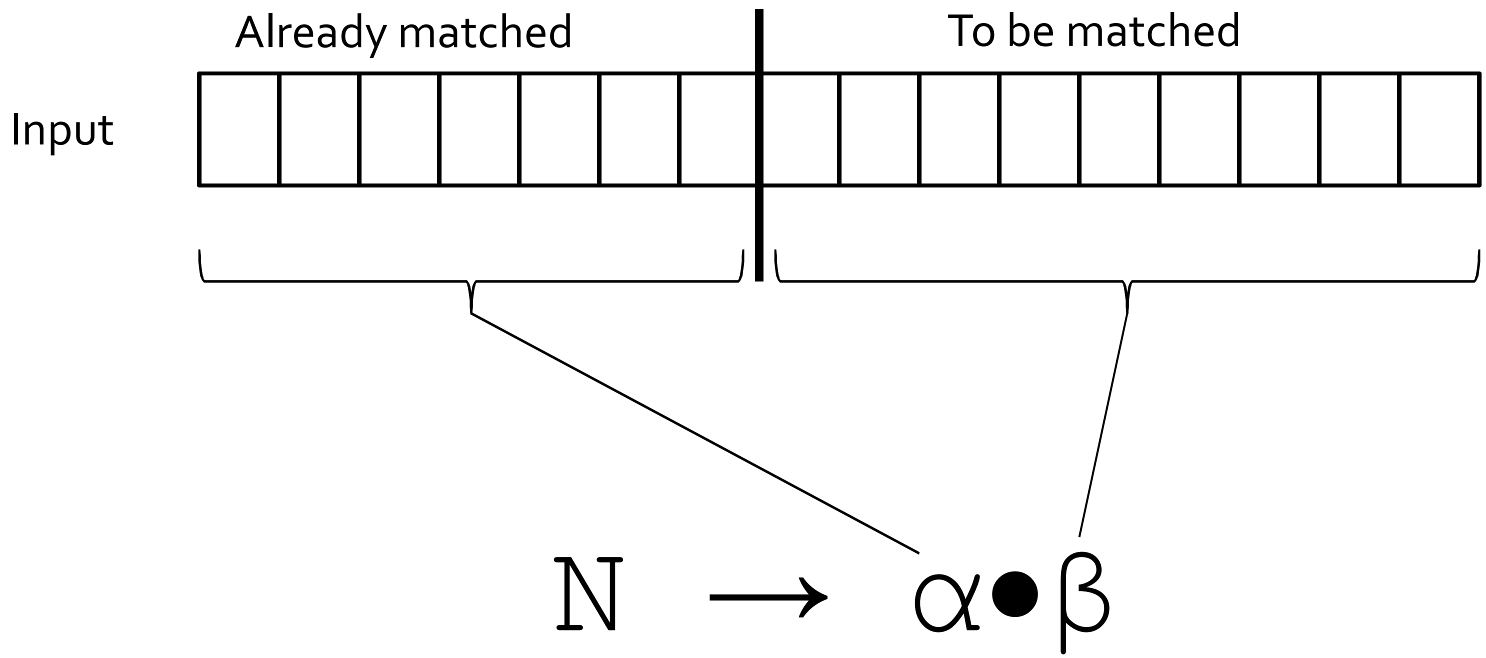
- Parsing
  - Top-down or bottom-up
- Top-down parsing
  - Recursive descent
  - LL(k) grammars
  - LL(k) parsing with pushdown automata
- LL(K) parsers
  - Cannot deal with left recursion
  - Left-recursion removal might result with complicated grammar

Bottom-up Parsing

- LR(K)
- SLR
- LALR

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

LR Item



$$N \rightarrow \alpha \bullet \beta$$

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

LR Items

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

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

Example

$$Z \rightarrow \texttt{expr EOF}$$

$$\texttt{expr} \rightarrow \texttt{term} \mid \texttt{expr + term}$$

$$\texttt{term} \rightarrow \texttt{ID} \mid \texttt{(expr)}$$

$$Z \rightarrow E\ \$$$

$$E \rightarrow T \mid E + T$$

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

(just shorthand of the grammar on the top)

Example: Parsing with LR Items

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

	i		+		i		$

$Z \rightarrow \bullet E\ \$$

$E \rightarrow \bullet T$

$E \rightarrow \bullet E + T$

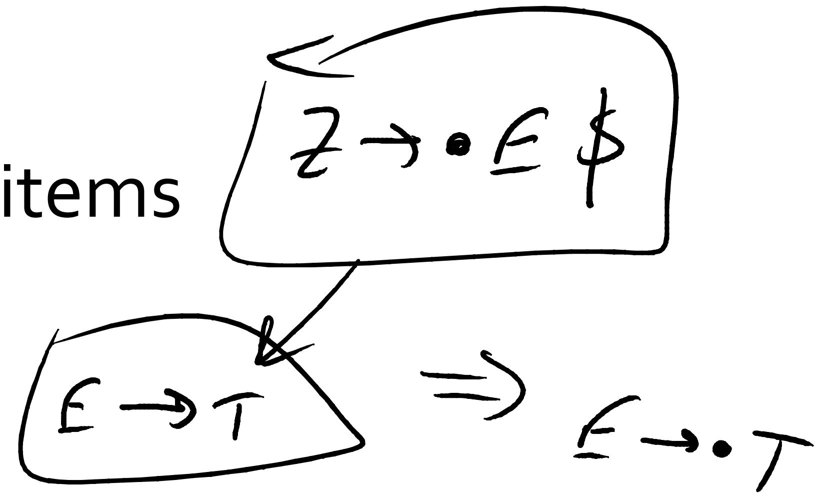
$T \rightarrow \bullet i$

$T \rightarrow \bullet (E)$

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

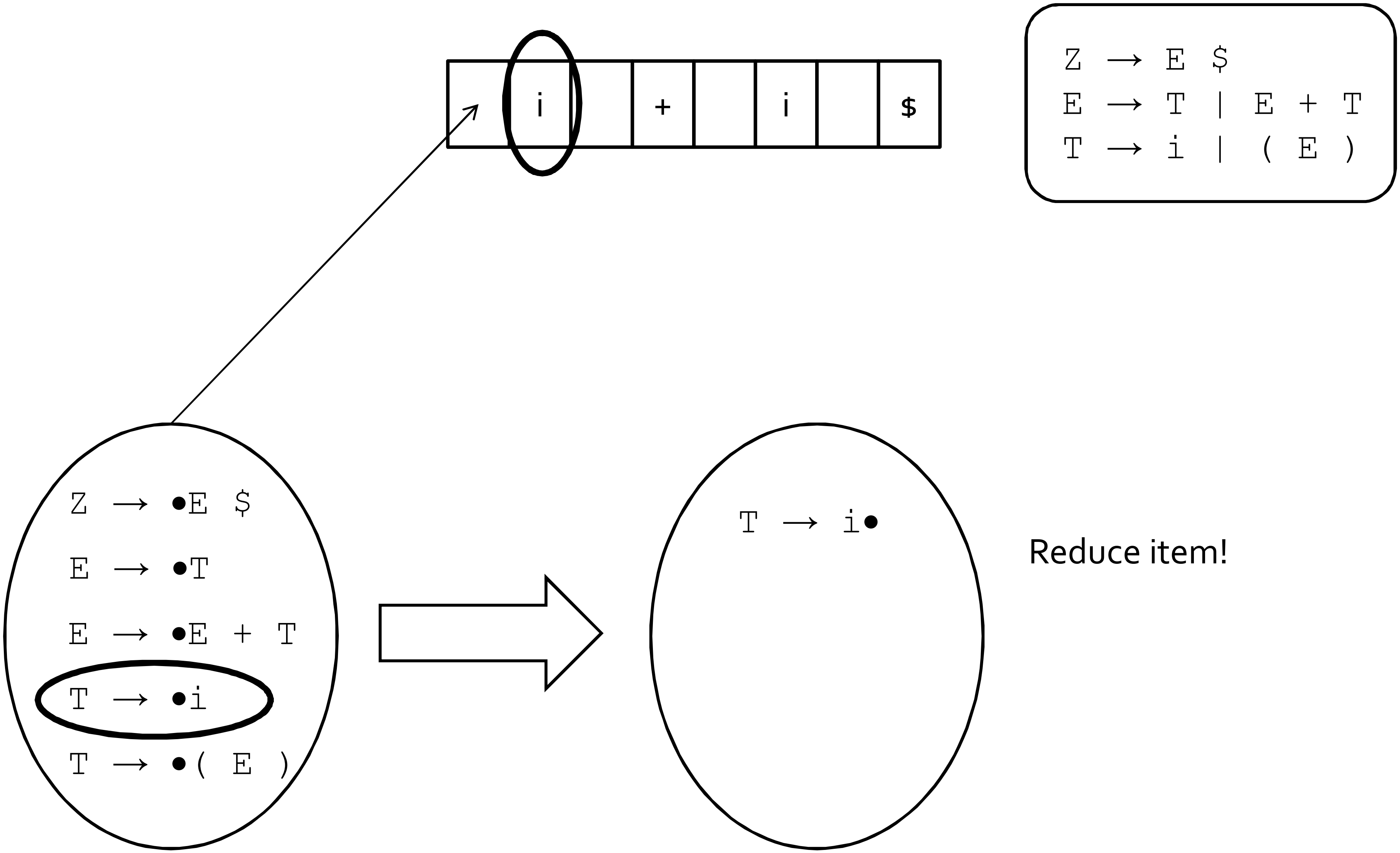


- If P → α•Nβ is in S
- then for each rule N → γ in the grammar S must also contain N → •γ

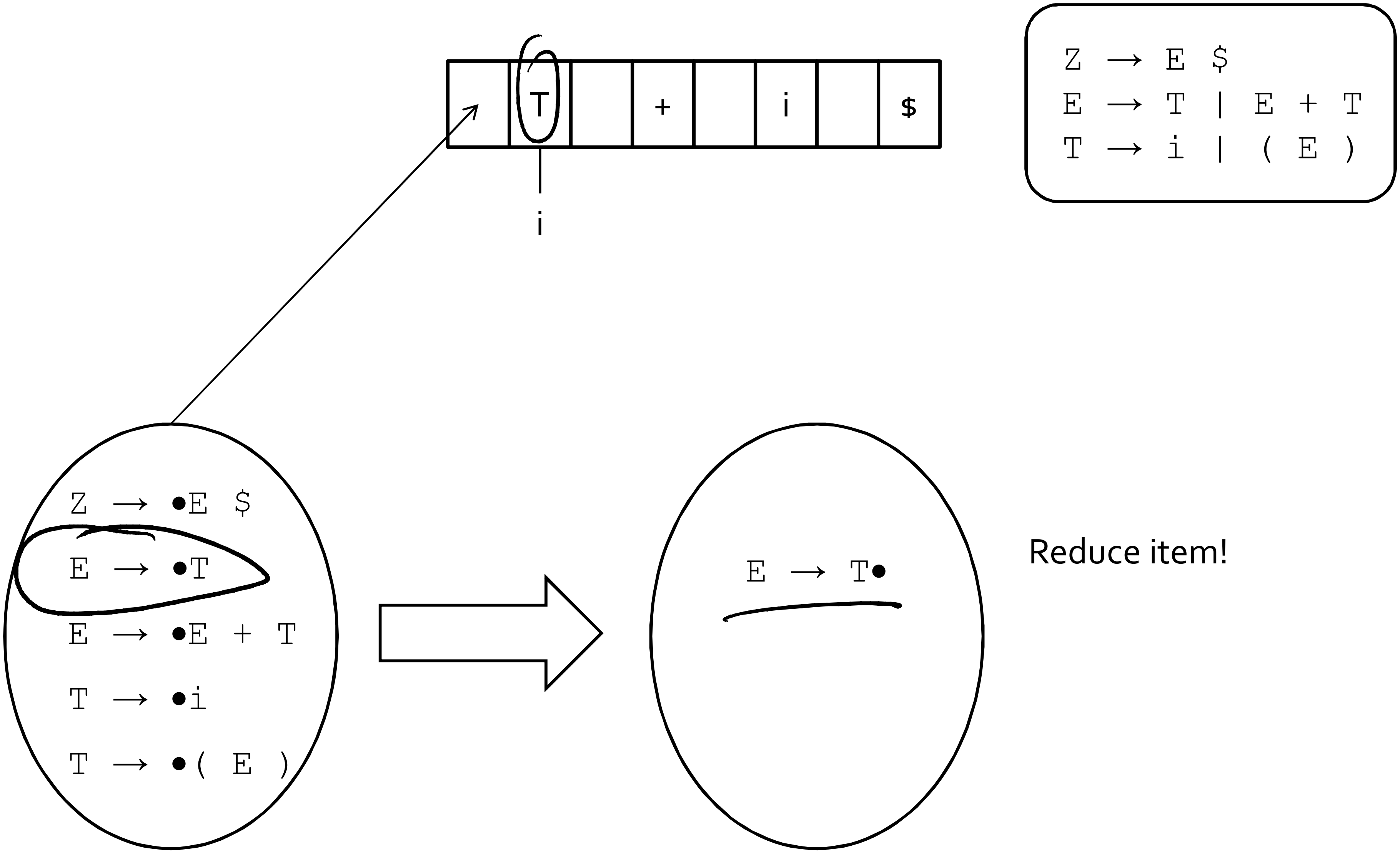
Z → E $
E → T
E → E + T
T → i
T → (E)

ε-closure ({ Z → •E $ }) = { Z → •E $, E → •T, E → •E + T, T → •i , T → •(E) }

Example: Parsing with LR Items

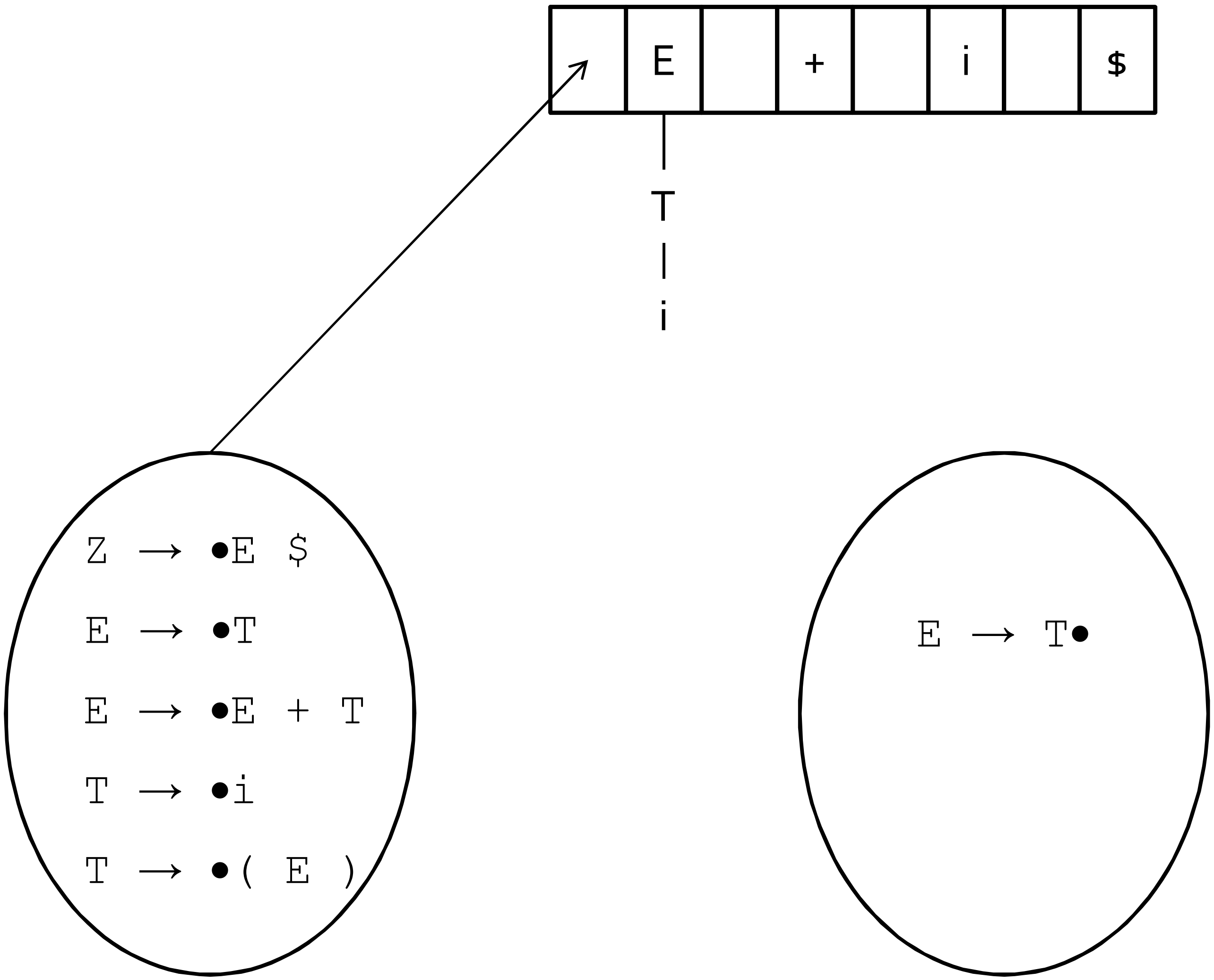


Example: Parsing with LR Items



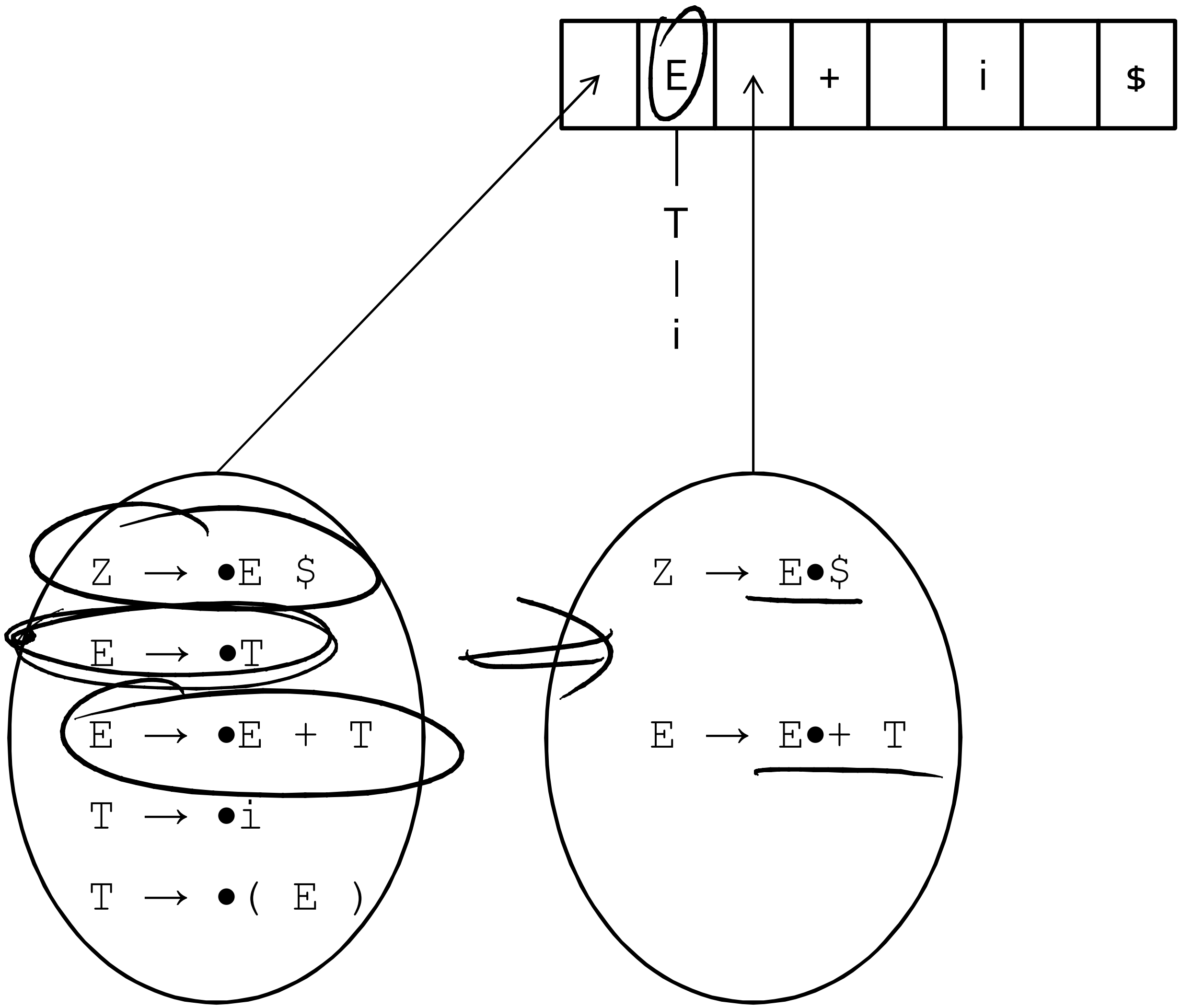
Example: Parsing with LR Items

Z → E $
E → T | E + T
T → i | (E)



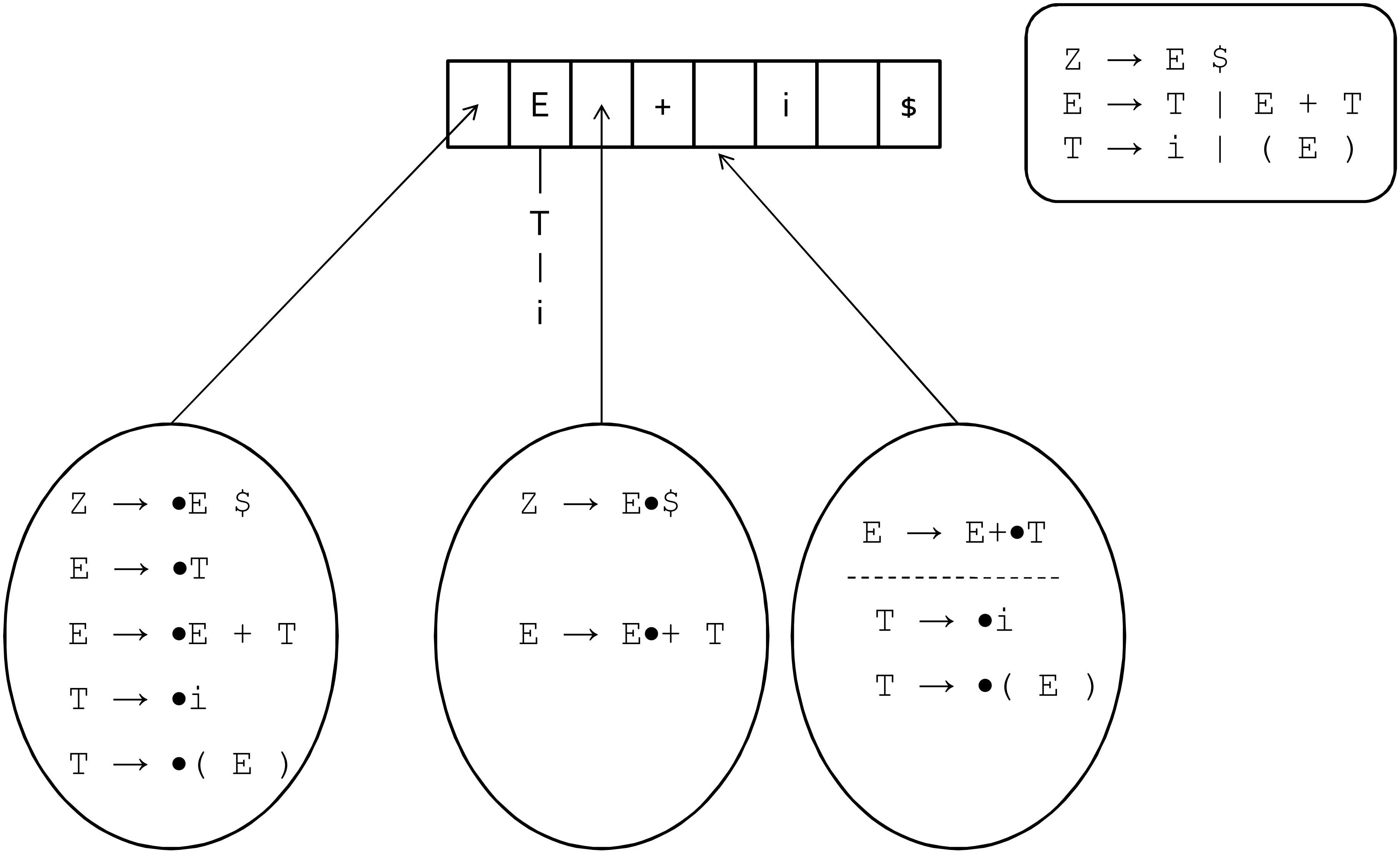
Reduce item!

Example: Parsing with LR Items



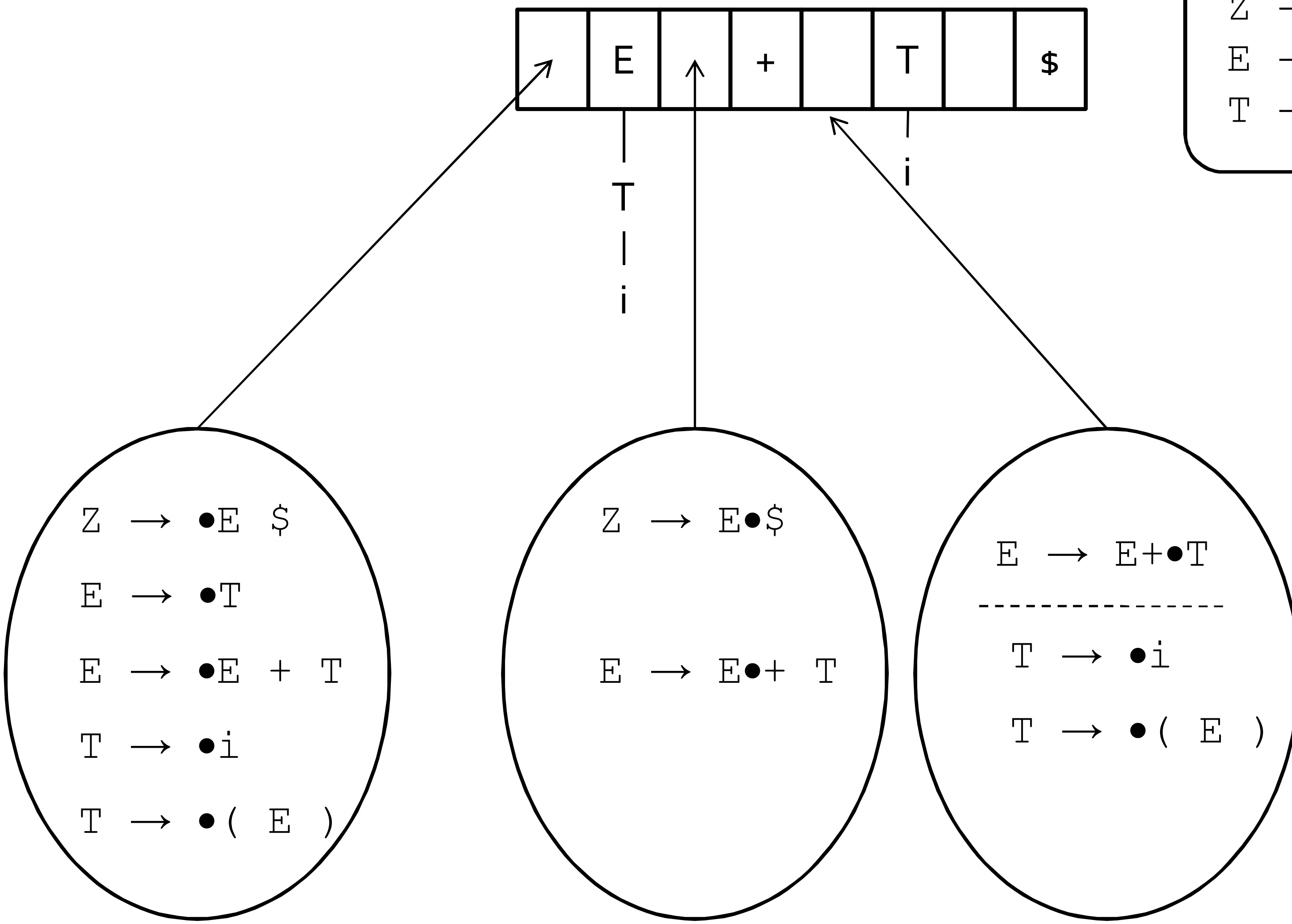
Z → E $
E → T | E + T
T → i | (E)

Example: Parsing with LR Items

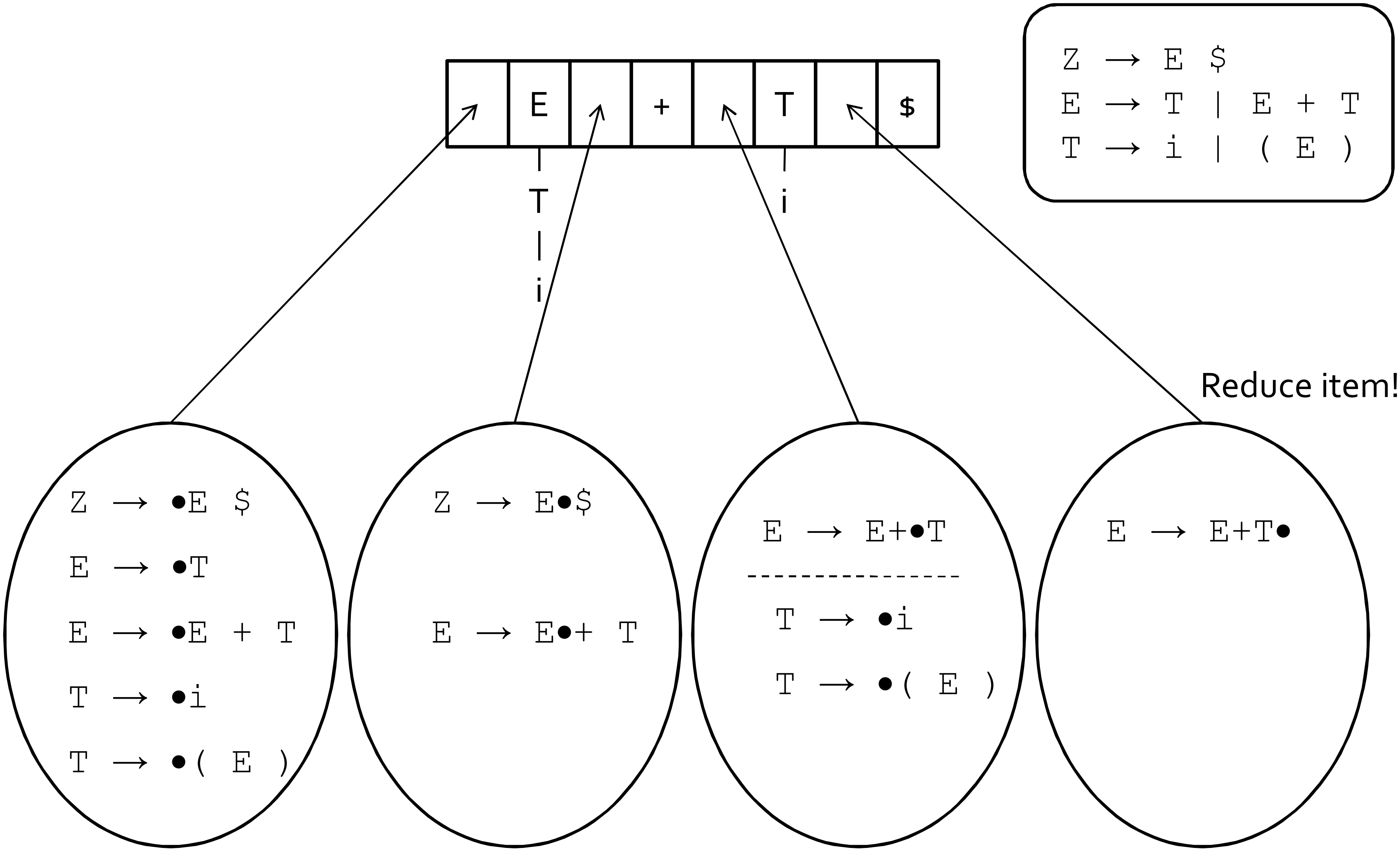


Example: Parsing with LR Items

Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)

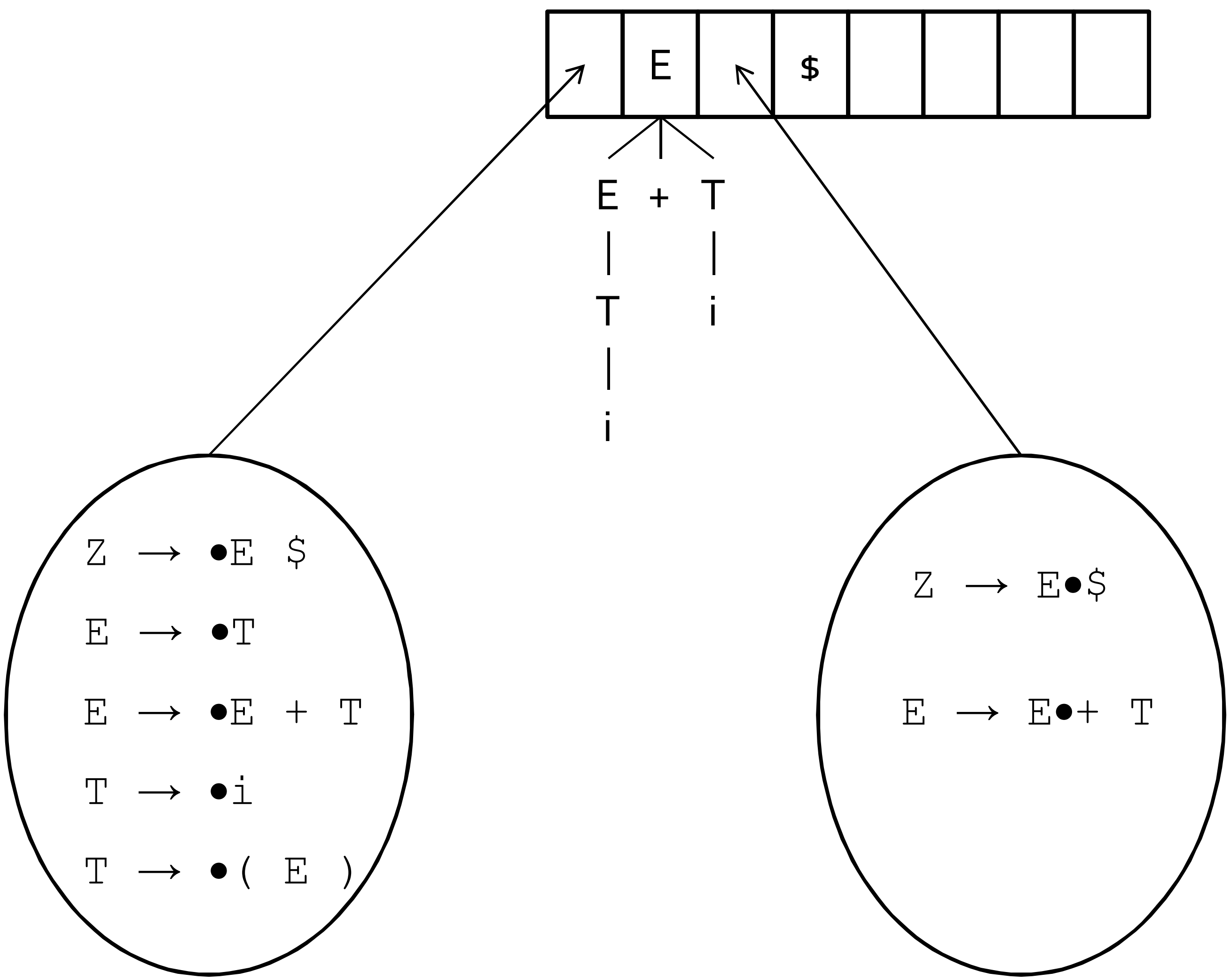


Example: Parsing with LR Items

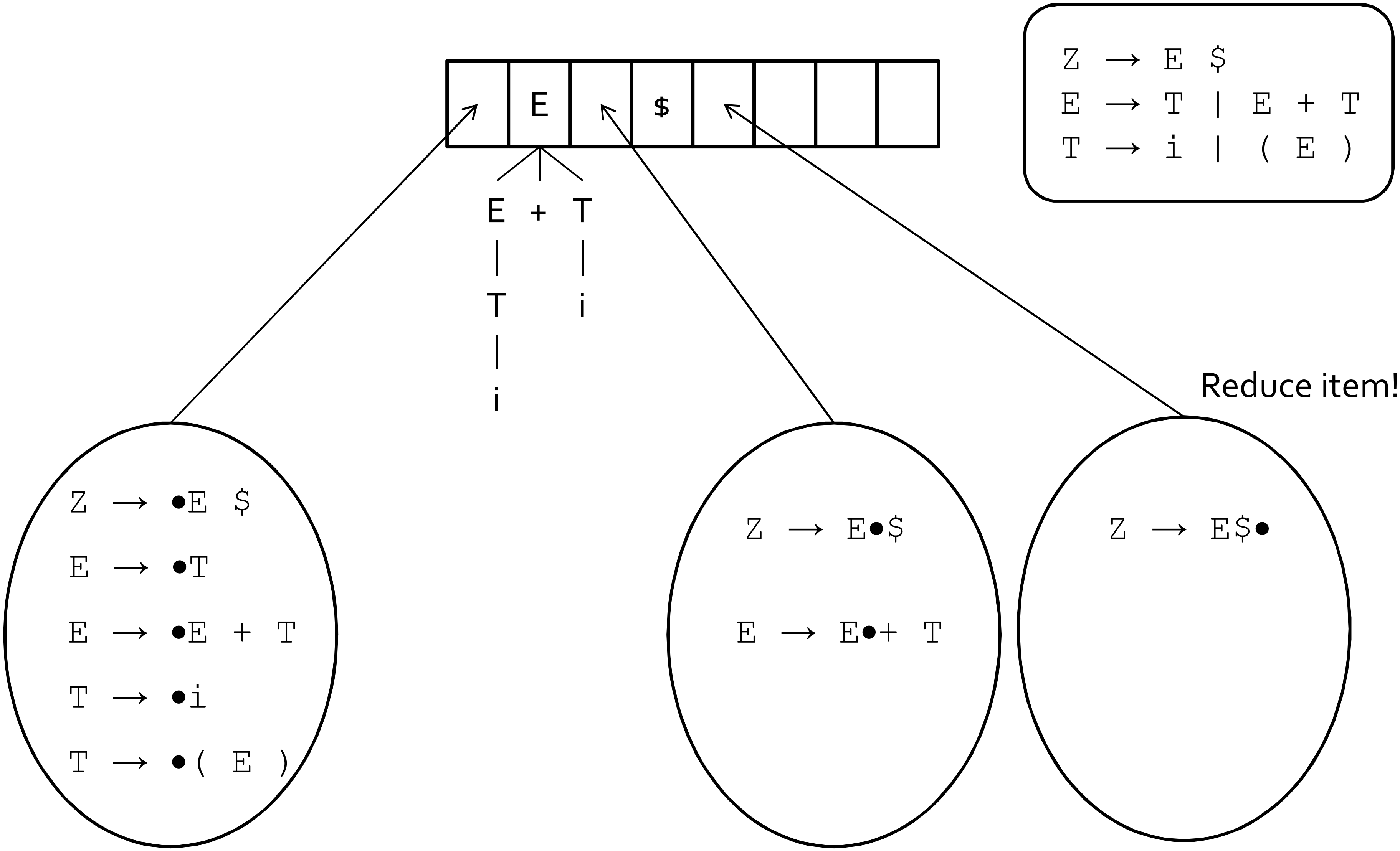


Example: Parsing with LR Items

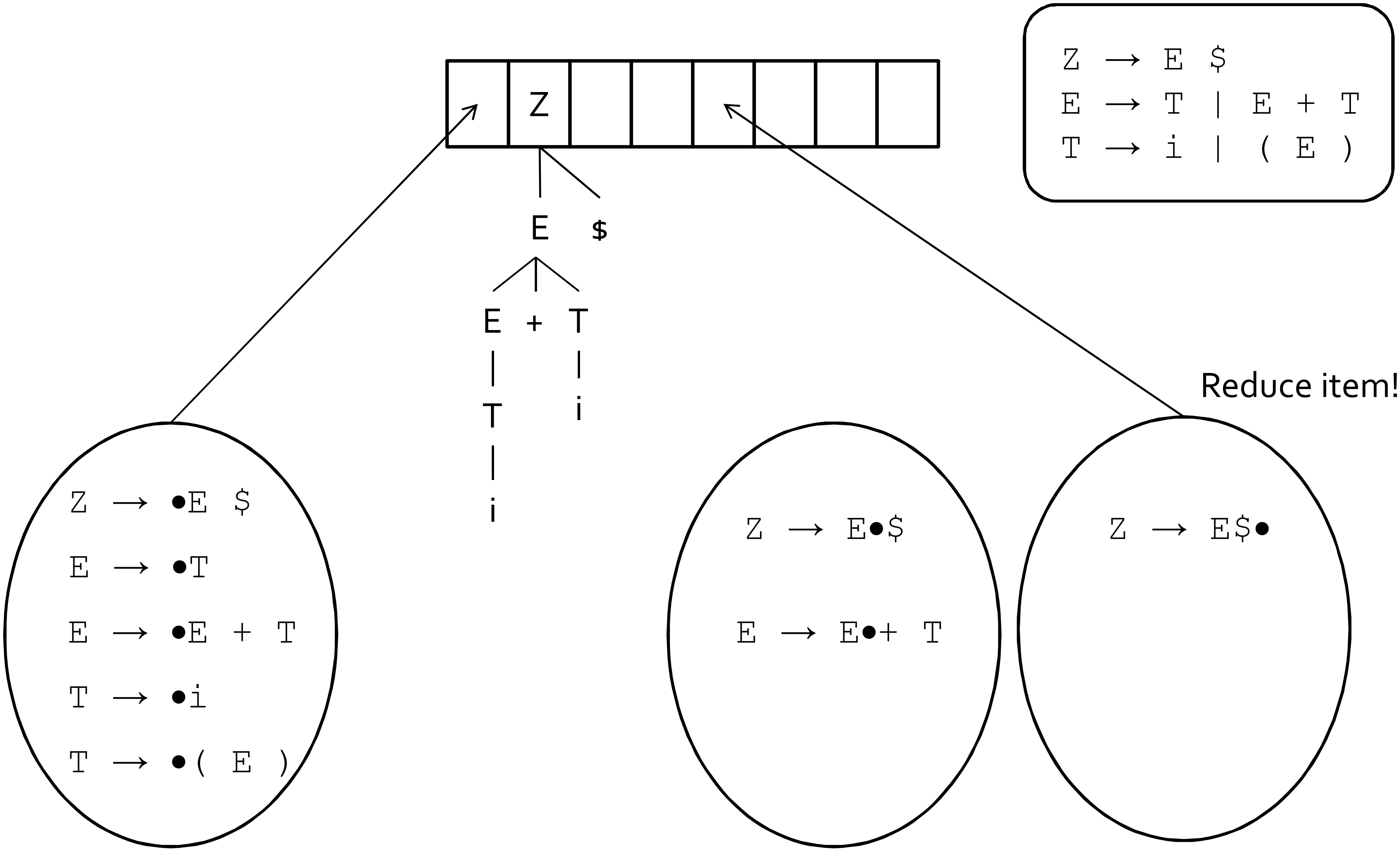
Z ⟶ E $
E ⟶ T | E + T
T ⟶ i | (E)



Example: Parsing with LR Items



Example: Parsing with LR Items

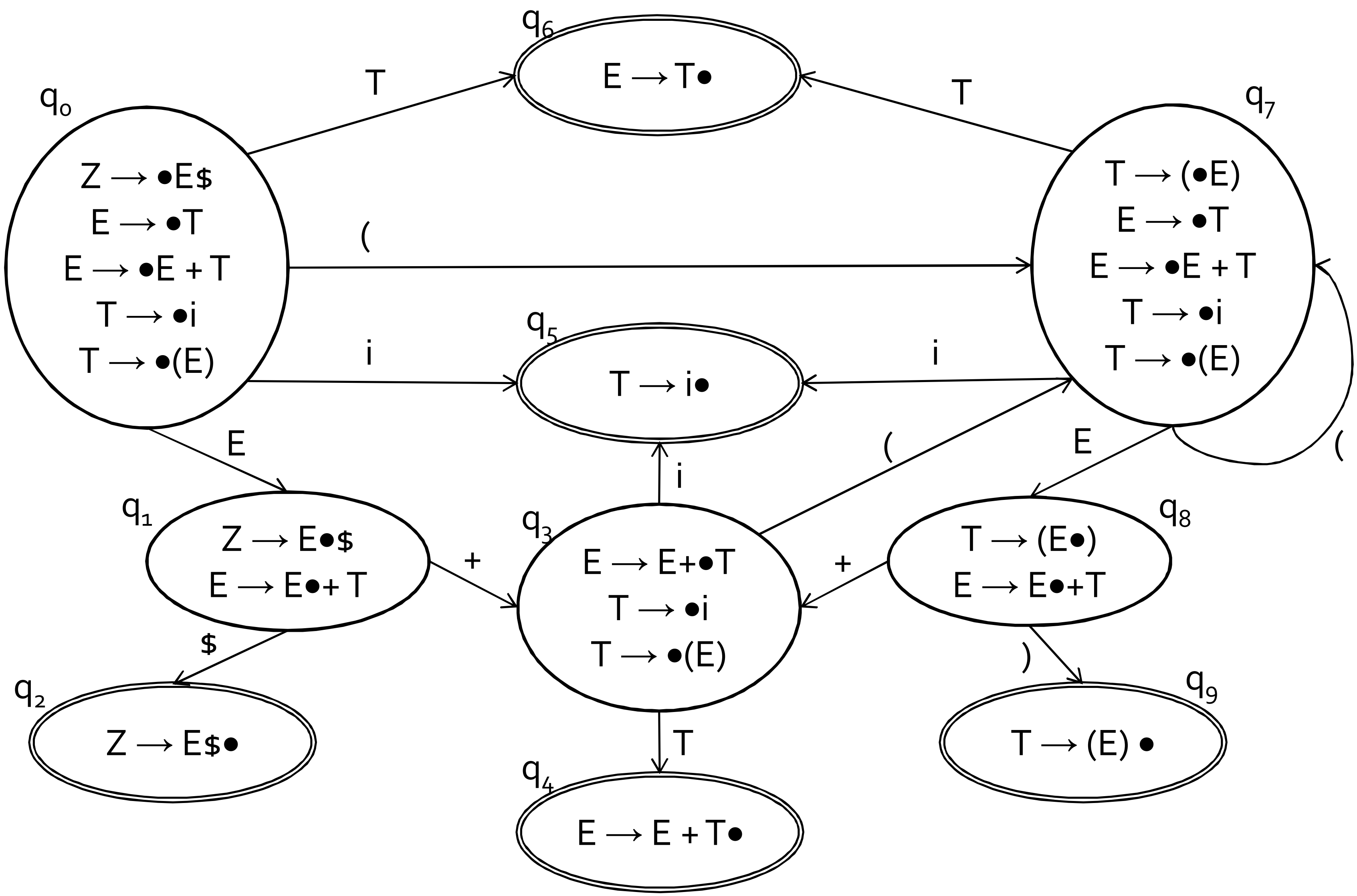


Computing Item Sets

- 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))

LR(0) Automaton Example

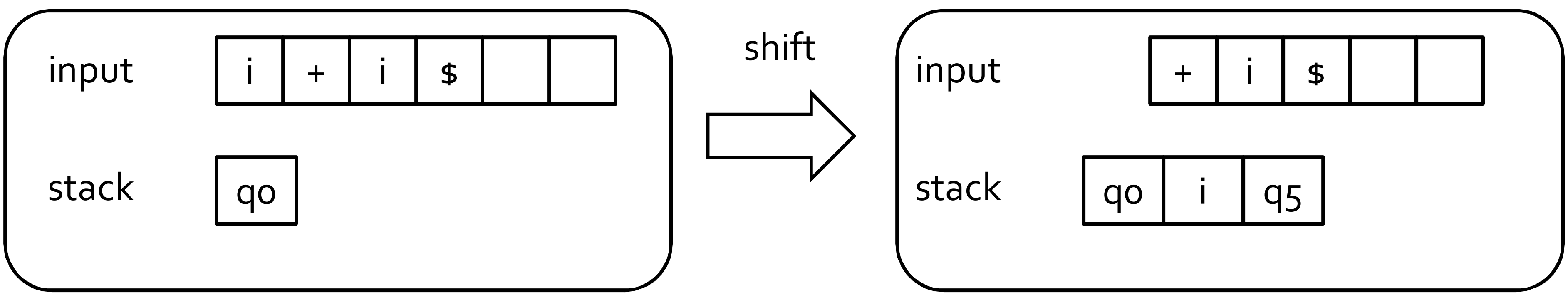


GOTO/ACTION Tables

	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 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



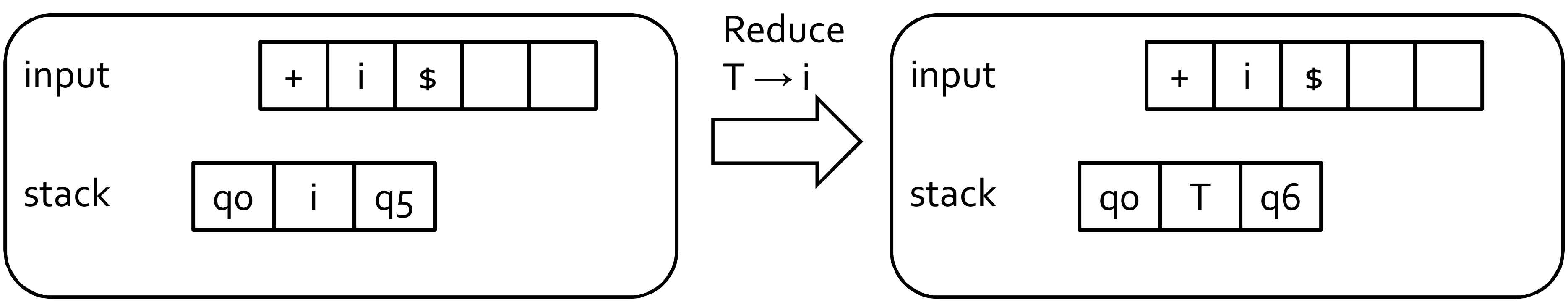
State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift

LR Pushdown Automaton

T → i.

V → i.

- Reduce move
  - Using a rule N $\rightarrow \alpha$
  - 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

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

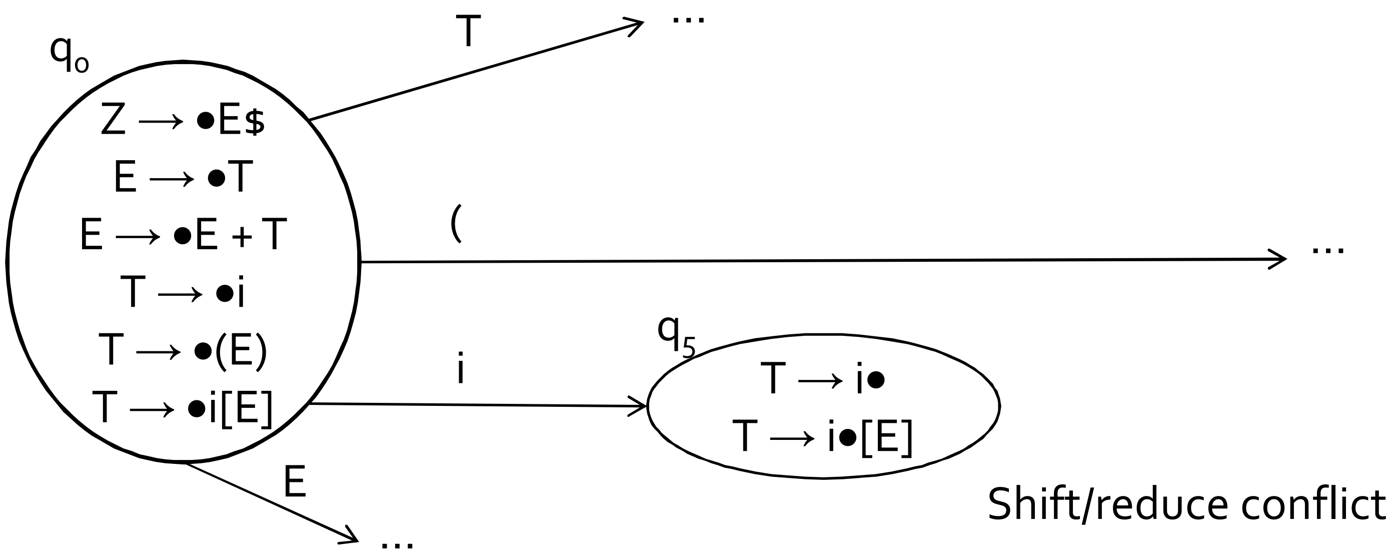
Stack	Input	Action
q0	i + i $	s5
q0 i q5	+ i $	r4
q0 T q6	+ i $	r2
q0 E q1	+ i $	s3
q0 E q1 + q3	i $	s5
q0 E q1 + q3 i q5	$	r4
q0 E q1 + q3 T q4	$	r3
q0 E q1	$	s2
q0 E q1 $ q2		r1
q0 Z		

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

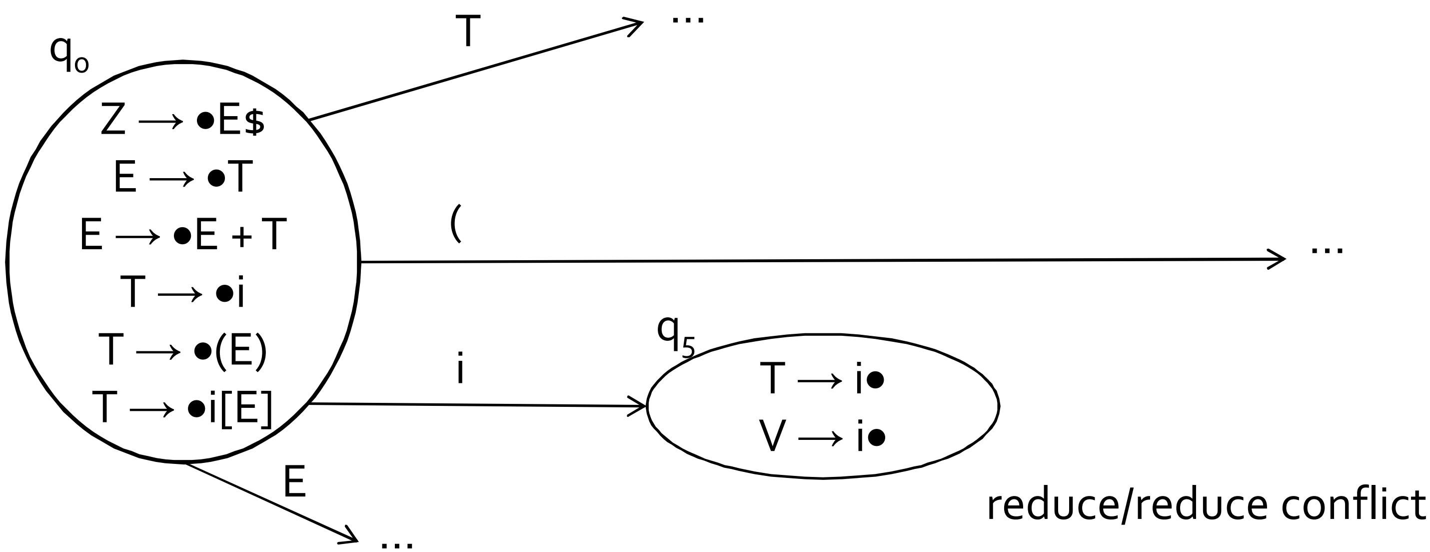
LR(0) Conflicts



Shift/reduce conflict

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

LR(0) Conflicts



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→ ε• - reduce item
  - P →α•Aβ – shift item
  - A→ ε• can always be predicted from P →α•Aβ

Back to the GOTO/ACTIONS tables

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 only by transition diagram, ignores input

SRL Grammars

- A handle should not be reduced to a non-terminal N if the look-ahead is a token that cannot follow N
- A reduce item $N \longrightarrow \alpha\bullet$ is applicable only when the look-ahead is in FOLLOW(N)

- Differs from LR(0) only on the ACTION table

SLR ACTION Table

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

Look-ahead token from the input

Remember:
In contrast, GOTO table is indexed by state and a grammar symbol from the stack

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

SLR ACTION Table

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

SLR – use 1 token look-ahead

vs.

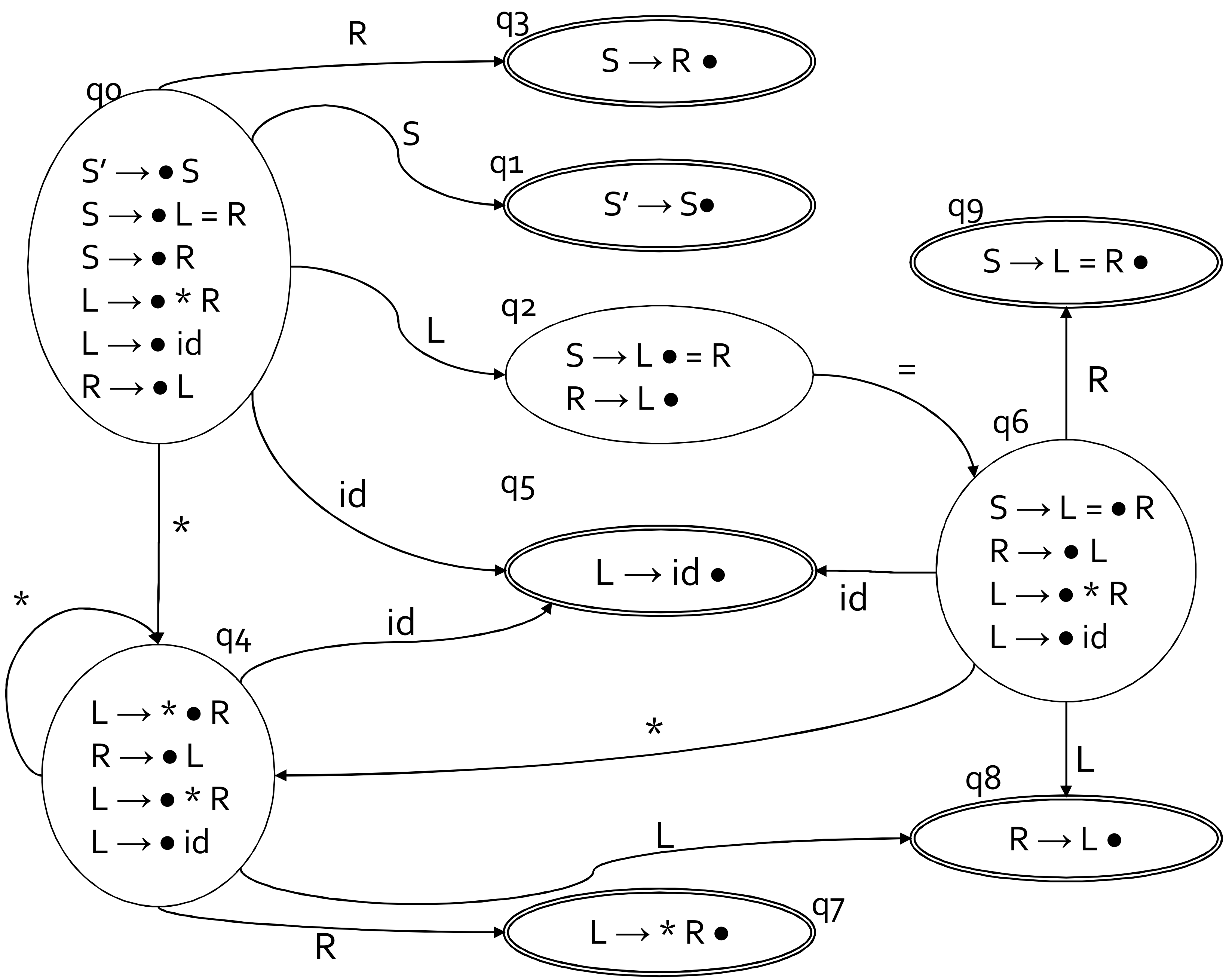
state	action
q0	shift
q1	shift
q2	Z⟶E$
q3	Shift
q4	E⟶E+T
q5	T⟶i
q6	E⟶T
q7	shift
q8	shift
q9	T⟶E

LR(0) – no look-ahead

```
… as before…
T ⟶ i
T ⟶ i[E]
```

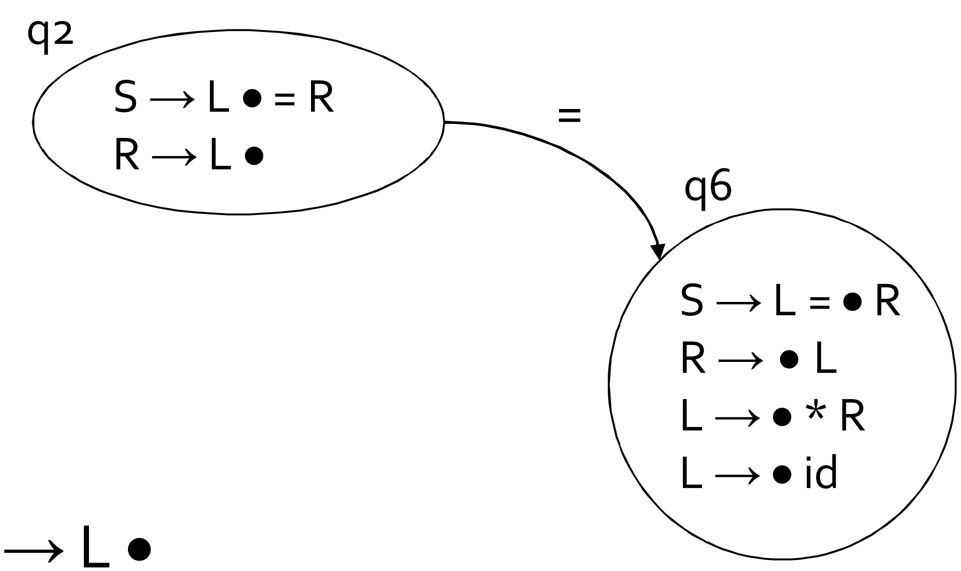
Are we done?

(0) $S' \longrightarrow S$
(1) $S \longrightarrow L = R$
(2) $S \longrightarrow R$
(3) $L \longrightarrow * R$
(4) $L \longrightarrow id$
(5) $R \longrightarrow L$


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

Shift/reduce conflict

(0) $S' \rightarrow S$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow * R$
(4) $L \rightarrow id$
(5) $R \rightarrow L$



- $S \rightarrow L \bullet = R$ vs. $R \rightarrow L \bullet$
- FOLLOW(R) contains =
  - $S \Rightarrow L = R \Rightarrow * R = R$
- SLR cannot resolve the conflict either

LR(1) Grammars

- In SLR: a reduce item $N \rightarrow \alpha\bullet$ is applicable only when the look-ahead is in FOLLOW(N)
- But FOLLOW(N) merges look-ahead for all alternatives for N

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

LR(1) Item

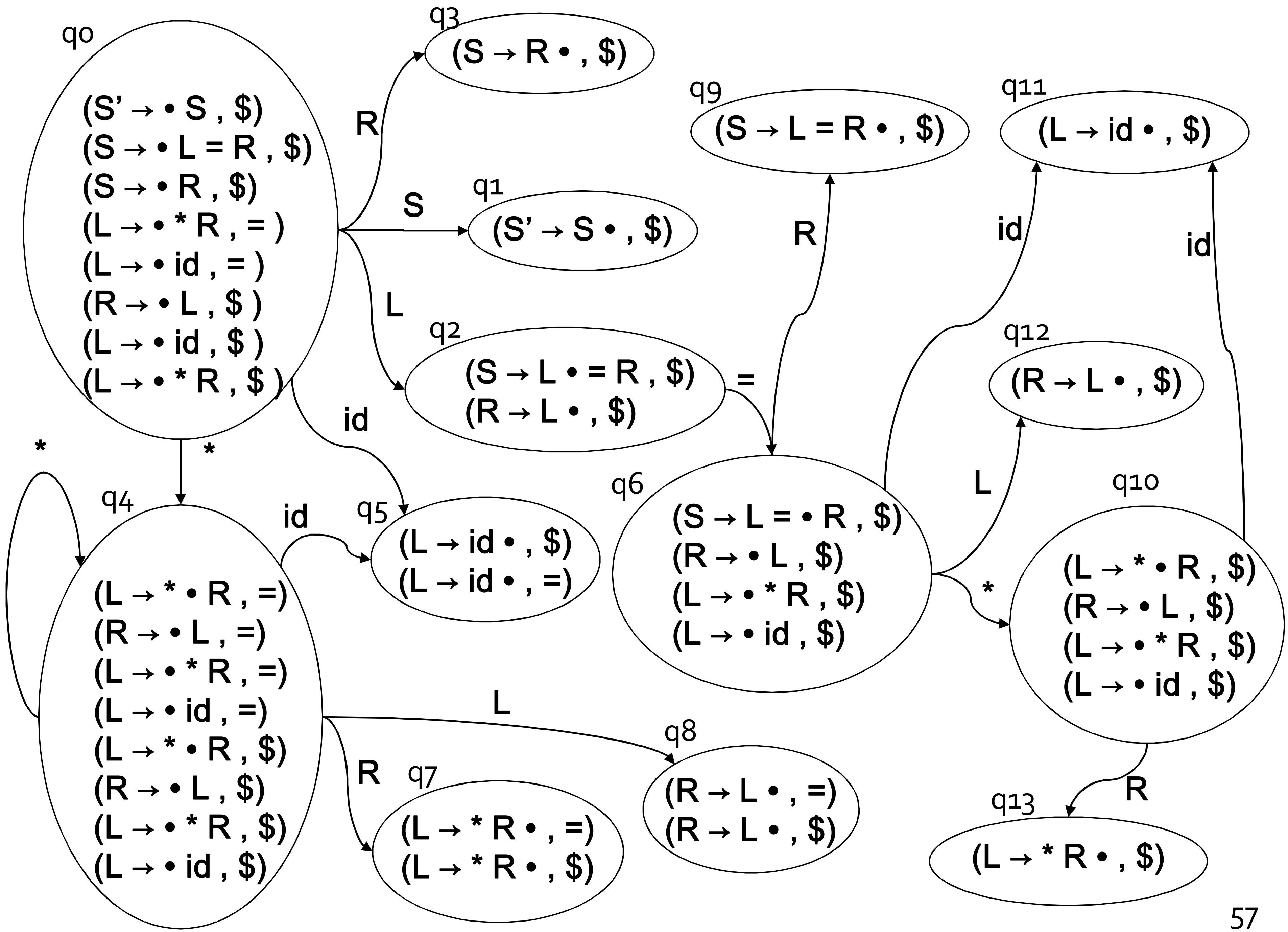
- LR(1) item is a pair
  - LR(0) item
  - Look-ahead token
- Meaning
  - We matched the part left of the dot, looking to match the part on the right of the dot, followed by the look-ahead token.
- Example
  - The production L ⟶ id yields the following LR(1) items

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

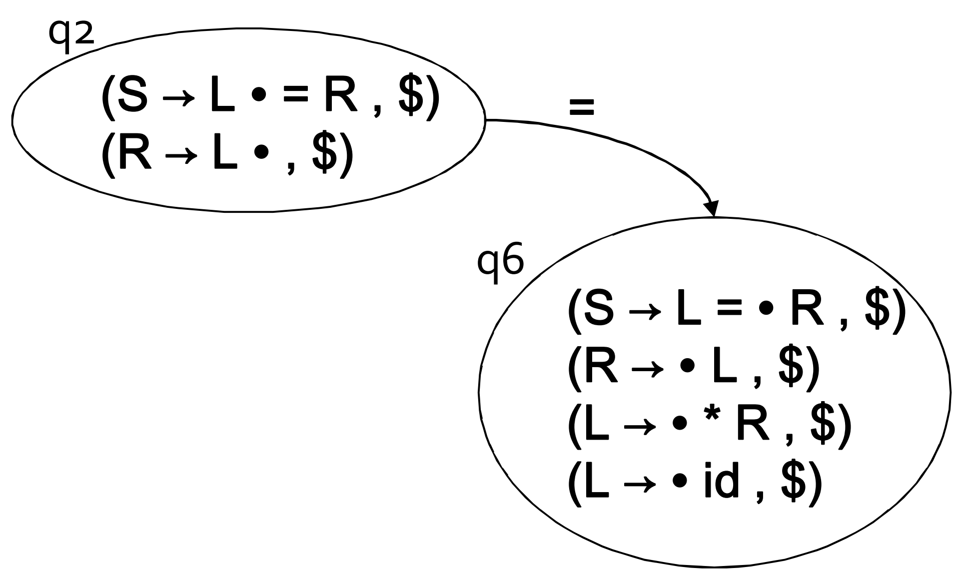
[L ⟶ ● id, *]
[L ⟶ ● id, =]
[L ⟶ ● id, id]
[L ⟶ ● id, $]
[L ⟶ id ●, *]
[L ⟶ id ●, =]
[L ⟶ id ●, id]
[L ⟶ id ●, $]

ε-closure for LR(1)

- For every $[A \rightarrow \alpha \bullet B\beta , c]$ in S
    - for every production $B \rightarrow \delta$ and every token b in the grammar such that $b \in FIRST(\beta c)$
    - Add $[B \rightarrow \bullet \delta , b]$ to S


q0
(S' → • S , $)
(S → • L = R , $)
(S → • R , $)
(L → • * R , =)
(L → • id , =)
(R → • L , $)
(L → • id , $)
(L → • * R , $)
q3
(S → R • , $)
q1
(S' → S • , $)
q2
(S → L • = R , $)
(R → L • , $)
q9
(S → L = R • , $)
q11
(L → id • , $)
q12
(R → L • , $)
q6
(S → L = • R , $)
(R → • L , $)
(L → • * R , $)
(L → • id , $)
q10
(L → * • R , $)
(R → • L , $)
(L → • * R , $)
(L → • id , $)
q13
(L → * R • , $)
q4
(L → * • R , =)
(R → • L , =)
(L → • * R , =)
(L → • id , =)
(L → * • R , $)
(R → • L , $)
(L → • * R , $)
(L → • id , $)
q5
(L → id • , $)
(L → id • , =)
q7
(L → * R • , =)
(L → * R • , $)
q8
(R → L • , =)
(R → L • , $)
R
S
L
id
*
*
id
L
R
=
R
id
id
L
*
R

Back to the conflict



- Is there a conflict now?

LALR

- LR tables have large number of entries
- Often don't need such refined observation (and cost)
- LALR idea: find states with the same LR(0) component and merge their look-ahead component as long as there are no conflicts
- LALR not as powerful as LR(1)

Summary

- Bottom up
  - LR Items
  - LR parsing with pushdown automata
  - LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm

Next time

- Semantic analysis

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