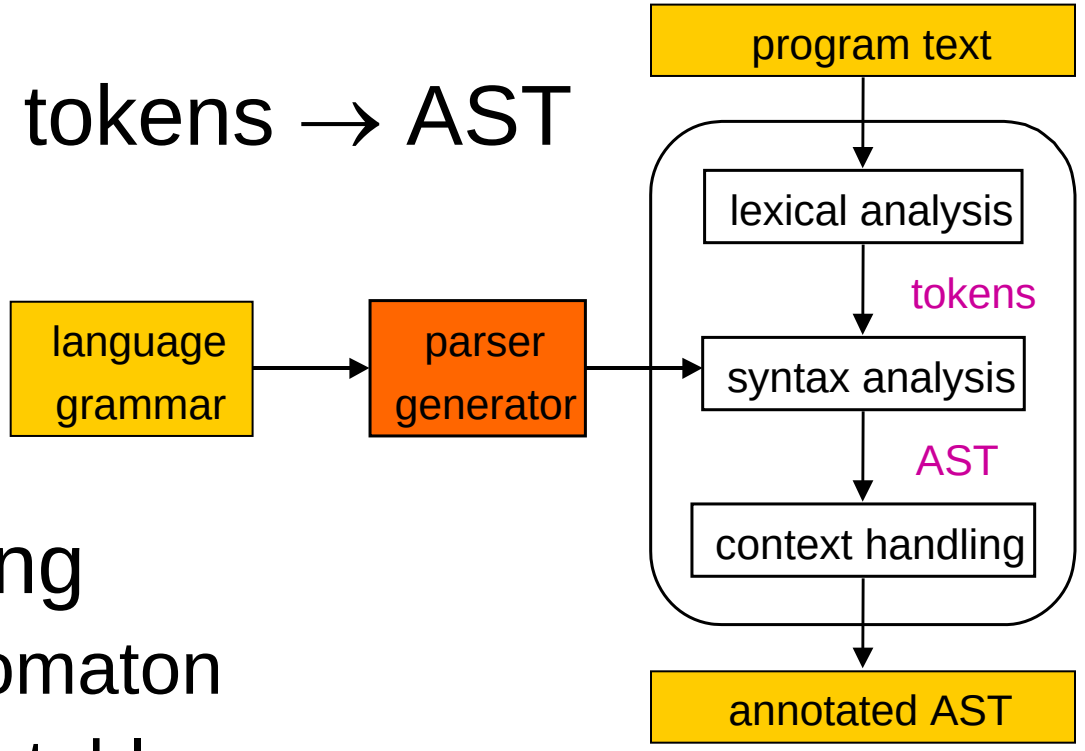
Compiler construction in4303 – lecture 4

Bottom-up parsing

Chapter 2.2.5

Overview

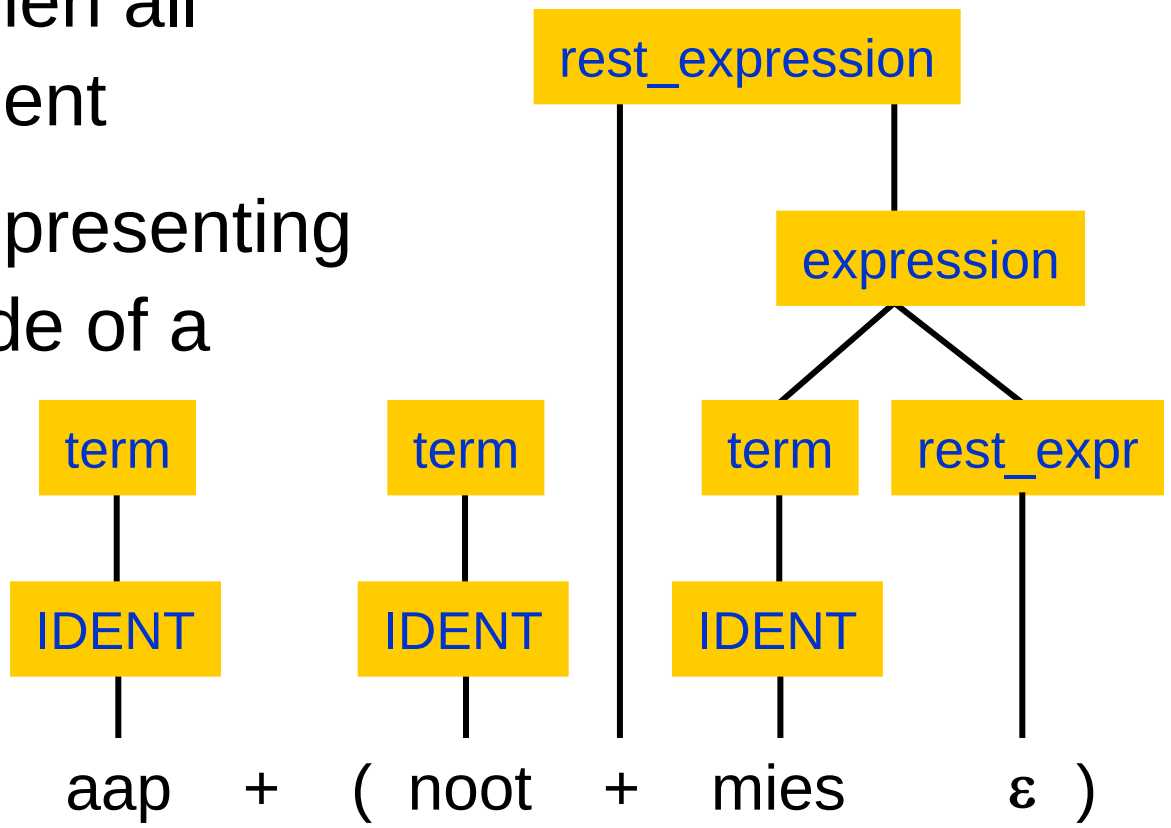
- syntax analysis: tokens → AST



- bottom-up parsing
  - push-down automaton
  - ACTION/GOTO tables
  - LR(0), SLR(1), LR(1), LALR(1)

Bottom-up (LR) parsing

- **L**eft-to-right parse, **R**ightmost-derivation
- create a node when all children are present
- handle: nodes representing the right-hand side of a production



LR(0) parsing

- running example: expression grammar

  input → expression EOF

  expression → expression ‘+’ term | term

  term → IDENTIFIER | ‘(’ expression ‘)’

- short-hand notation

  Z → E $

  E → E ‘+’ T

  E → T

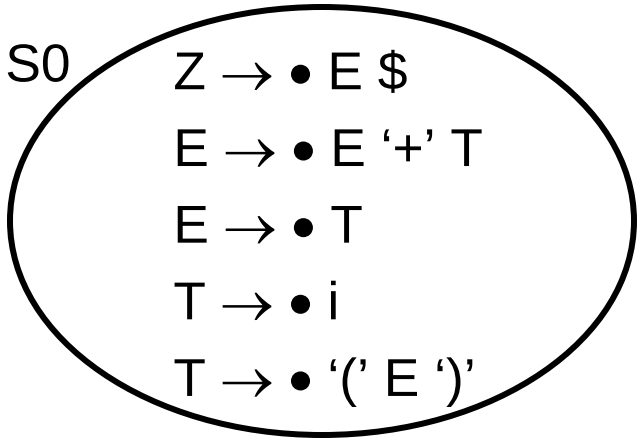
  T → i

  T → ‘(’ E ‘)’

LR(0) parsing

- keep track of progress inside potential handles when consuming input tokens
- LR items: $N \rightarrow \alpha \bullet \beta$
- initial set

$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$



- ε-closure: expand dots in front of non-terminals

LR(0) parsing

stack

S0

input

i + i $

$Z \rightarrow E\ \$$

$E \rightarrow E\ \text{‘+’}\ T$

$E \rightarrow T$

$T \rightarrow i$

$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- shift input token (i) onto the stack
- compute new state

LR(0) parsing

stack

S0 i S1

input

+ i $

$Z \rightarrow E\ \$$

$E \rightarrow E\ \text{‘+’}\ T$

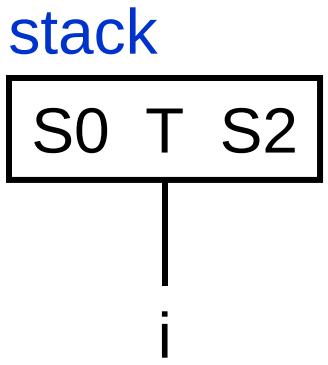
$E \rightarrow T$

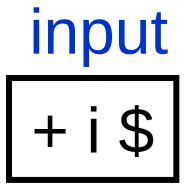
$T \rightarrow i$

$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- reduce handle on top of the stack
- compute new state

LR(0) parsing

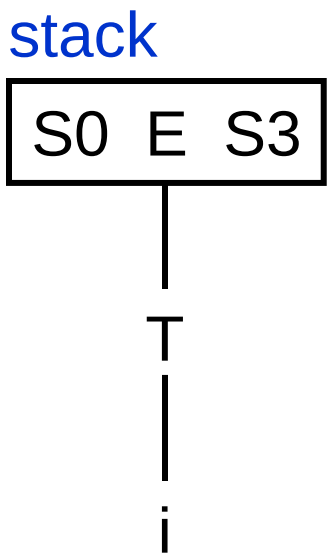


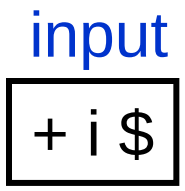


$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- reduce handle on top of the stack
- compute new state

LR(0) parsing

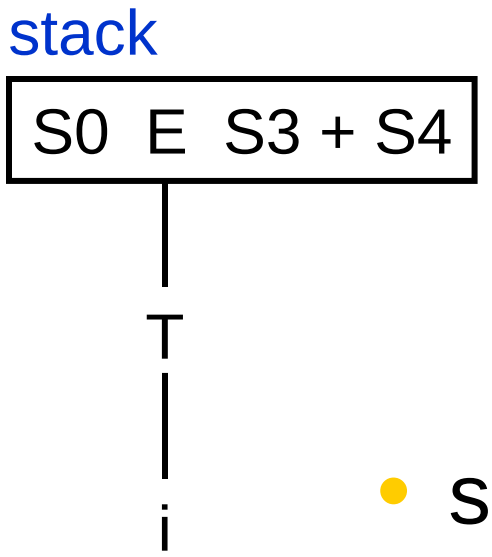


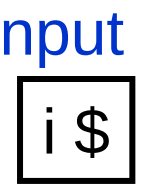


$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- shift input token on top of the stack
- compute new state

LR(0) parsing

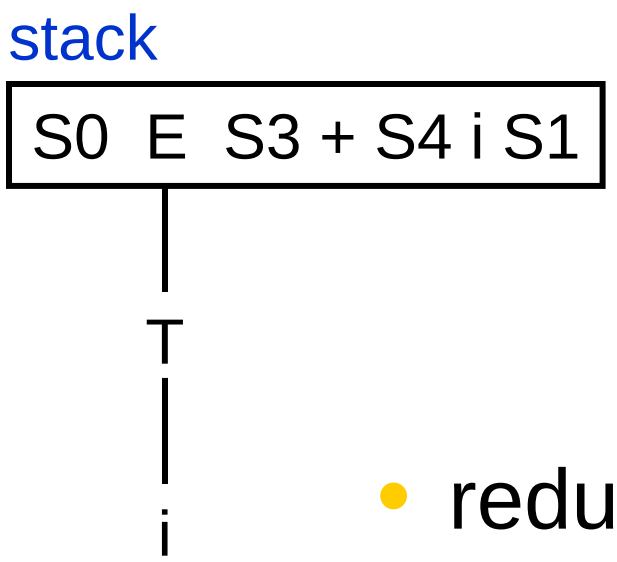


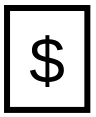


$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- shift input token on top of the stack
- compute new state

LR(0) parsing

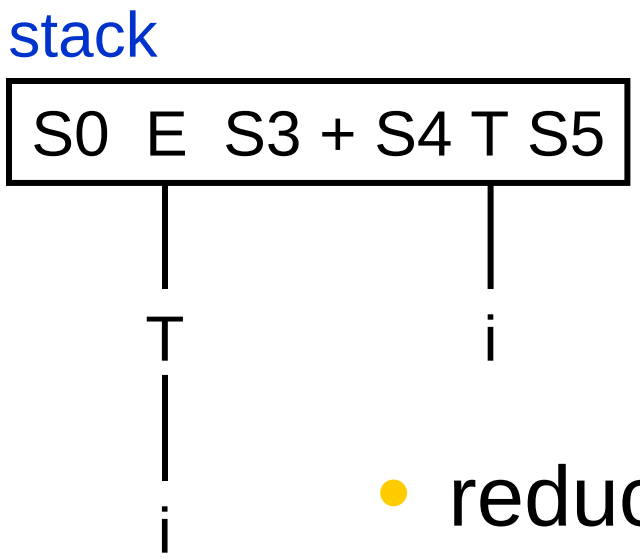




$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- reduce handle on top of the stack
- compute new state

LR(0) parsing



$Z \rightarrow E\ \$$

$E \rightarrow E\ \text{‘+’}\ T$

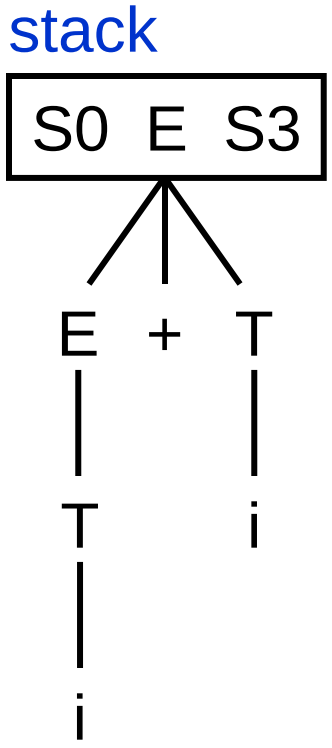
$E \rightarrow T$

$T \rightarrow i$

$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- reduce handle on top of the stack
- compute new state

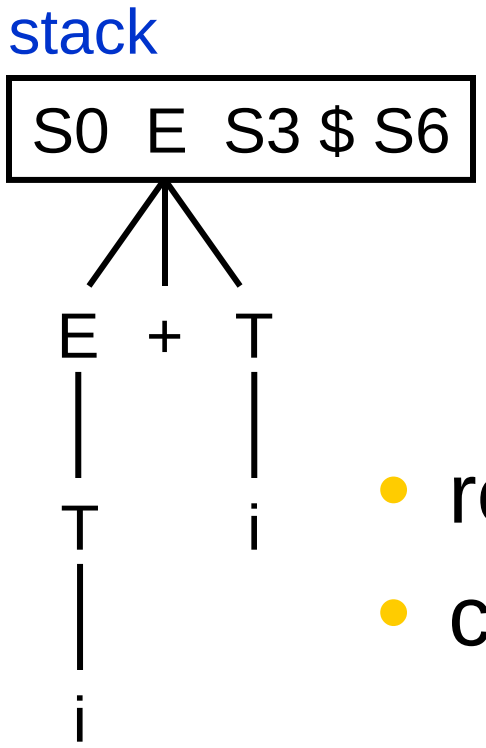
LR(0) parsing



$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- shift input token on top of the stack
- compute new state

LR(0) parsing



input

$Z \rightarrow E\ \$$

$E \rightarrow E\ \text{‘+’}\ T$

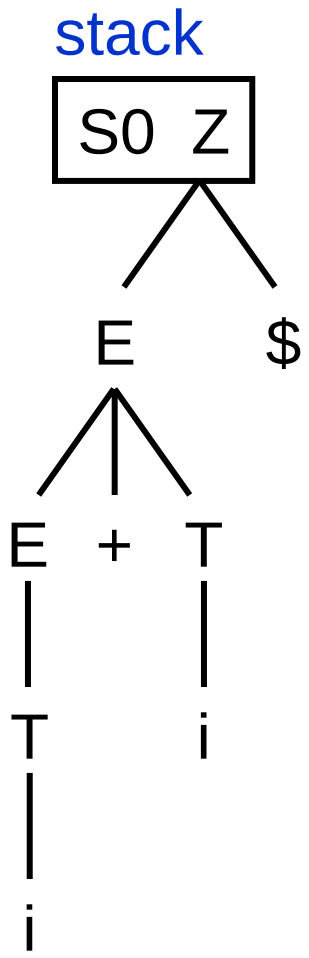
$E \rightarrow T$

$T \rightarrow i$

$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- reduce handle on top of the stack
- compute new state

LR(0) parsing

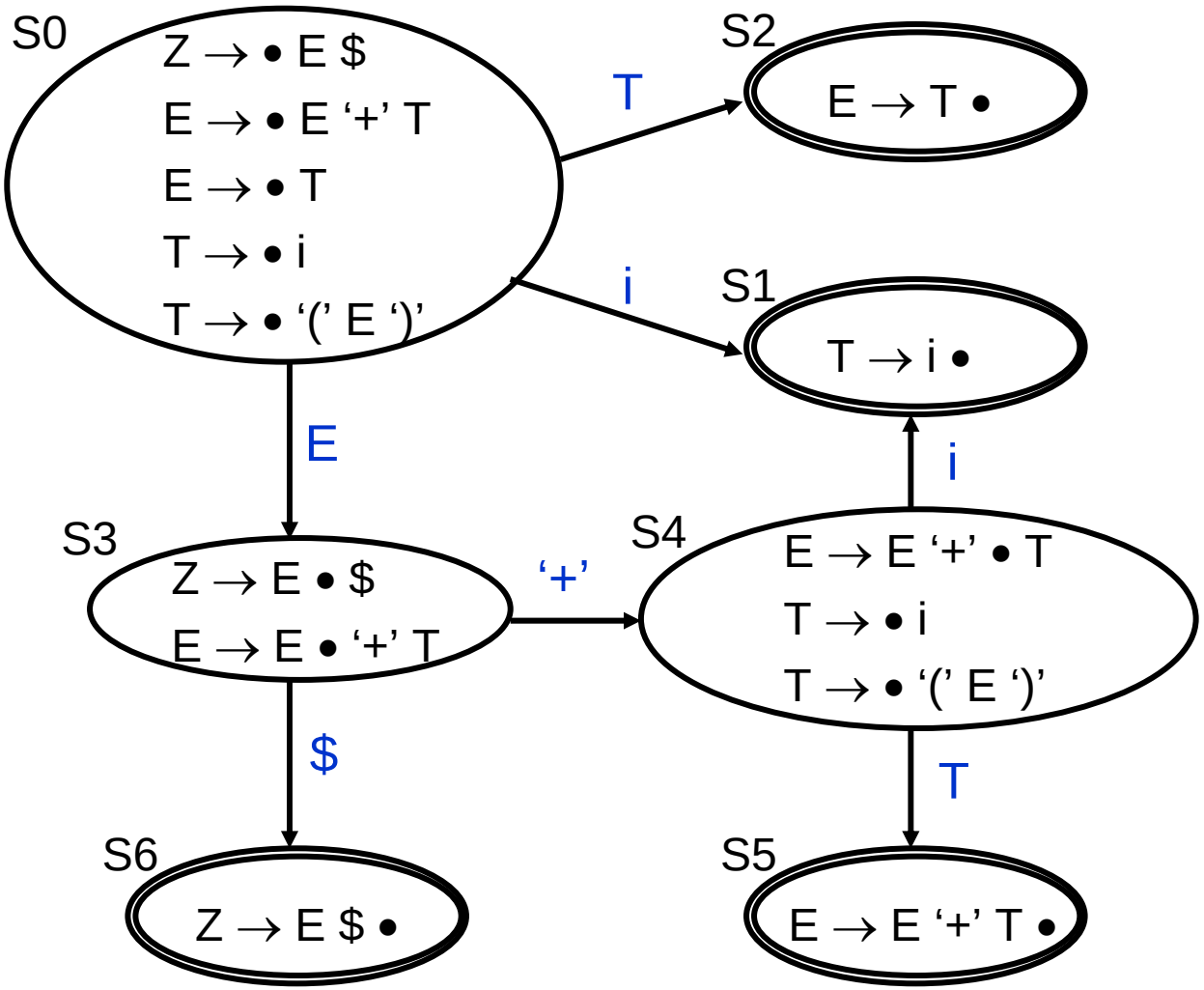


input

$Z \rightarrow E\ \$$
$E \rightarrow E\ \text{‘+’}\ T$
$E \rightarrow T$
$T \rightarrow i$
$T \rightarrow \text{‘(’}\ E\ \text{‘)’}$

- accept!

Transition diagram



Exercise (8 min.)

- complete the transition diagram for the LR(0) automaton

- can you think of a single input expression that causes all states to be used? If yes, give an example. If no, explain.

Answers

The LR tables

state	GOTO table						
	i	+	(	)	$	E	T
0	1		7			3	2
1							
2							
3		4			6		
4	1		7				5
5							
6							
7	1		7			8	2
8		4		9			
9							

state	ACTION table
0	shift
1	$T \rightarrow i$
2	$E \rightarrow T$
3	shift
4	shift
5	$E \rightarrow E + T$
6	$Z \rightarrow E \$$
7	shift
8	shift
9	$T \rightarrow (E)$

LR(0) parsing concise notation

stack	input	action
S0	i + i $	shift
S0 i S1	+ i $	reduce by T → i
S0 T S2	+ i $	reduce by E → T
S0 E S3	+ i $	shift
S0 E S3 + S4	i $	shift
S0 E S3 + S4 i S1	$	reduce by T → i
S0 E S3 + S4 T S5	$	reduce by E → E + T
S0 E S3	$	shift
S0 E S3 $ S6		reduce by Z → E $
S0 Z		accept

The LR push-down automaton

SWITCH action_table[top of stack]:

CASE "shift":

see book;

CASE ("reduce", $N \rightarrow \alpha$):

POP the symbols of α FROM the stack;

SET state TO top of stack;

PUSH N ON the stack;

SET new state TO goto_table[state,N];

PUSH new state ON the stack;

CASE empty:

ERROR;

LR(0) conflicts

Shift-reduce conflict

- array indexing: $T \rightarrow i\ [\ E\ ]$

$T \rightarrow i \bullet [\ E\ ]$ (shift)

$T \rightarrow i \bullet$ (reduce)

- ε-rule: $RestExpr \rightarrow \varepsilon$

$Expr \rightarrow Term \bullet RestExpr$ (shift)

$RestExpr \rightarrow \bullet$ (reduce)

LR(0) conflicts

Reduce-reduce conflict

- assignment statement: $Z \rightarrow V := E \$$

$V \rightarrow i \bullet$ (reduce)

$T \rightarrow i \bullet$ (reduce)

- typical LR(0) table contains many conflicts

Handling LR(0) conflicts

- solution: use a one-token look-ahead

  two-dimensional ACTION table [state,token]

- different construction of ACTION table
  - SLR(1) – Simple LR
  - LR(1)
  - LALR(1) – Look-Ahead LR

SLR(1) parsing

- solves (some) shift-reduce conflicts
- reduce $N \rightarrow \alpha$ iff token $\in$ FOLLOW(N)

FOLLOW(T) = { '+', ')', $ }
FOLLOW(E) = { '+', ')', $ }
FOLLOW(Z) = { $ }

SLR(1) ACTION table

state	look-ahead token				
	i	+	(	)	$
0	shift		shift		
1		T → i		T → i	T → i
2		E → T		E → T	E → T
3		shift			shift
4	shift		shift		
5		E → E + T		E → E + T	E → E + T
6					Z → E $
7	shift		shift		
8		shift		shift	
9		T → (E)		T → (E)	T → (E)

SLR(1) ACTION/GOTO table

state	stack symbol / look-ahead token								
	i	+	(	)	[	]	$	E	T
0	s1		s7					s3	s2
1		r4		r4	s10	r4	r4		
2		r3		r3		r3	r3		
3		s4					s6		
4	s1		s7						s5
5		r2		r2		r2	r2		
6							r1		
7	s1		s7					s8	s2
8		s4		s9					
9		r5		r5		r5	r5		

1: Z → E $
2: E → E ‘+’ T
3: E → T
4: T → i
5: T → ‘(’ E ‘)’
6: T → i ‘[‘ E ‘]’

s*n* – shift to state *n*
r*n* – reduce rule *n*

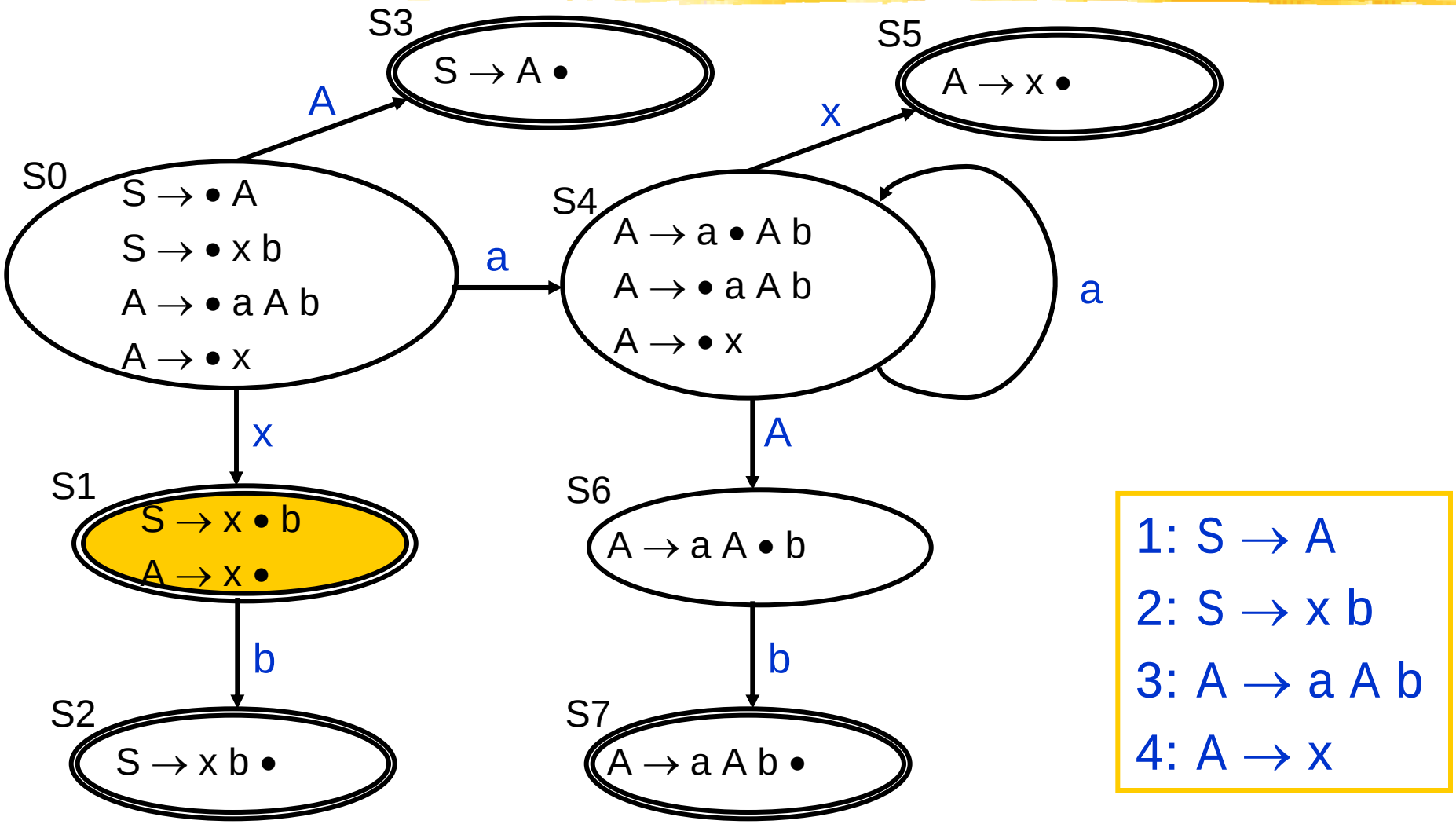
Unfortunately ...

- SLR(1) leaves many shift-reduce conflicts unsolved
- problem: FOLLOW(*N*) set is a union of all contexts in which *N* may occur
- example

  $S \rightarrow A \mid x\ b$

  $A \rightarrow a\ A\ b \mid x$

LR(0) automaton



1: S → A

2: S → x b

3: A → a A b

4: A → x

Exercise (6 min.)

- derive the SLR(1) ACTION/GOTO table (with shift-reduce conflict) for the grammar:

$S \rightarrow A \mid x\ b$

$A \rightarrow a\ A\ b \mid x$

Answers

LR(1) parsing

- maintain follow set per item

  LR(1) item: $N \rightarrow \alpha \bullet \beta \ \{\sigma\}$

- ε - closure for LR(1) item sets:

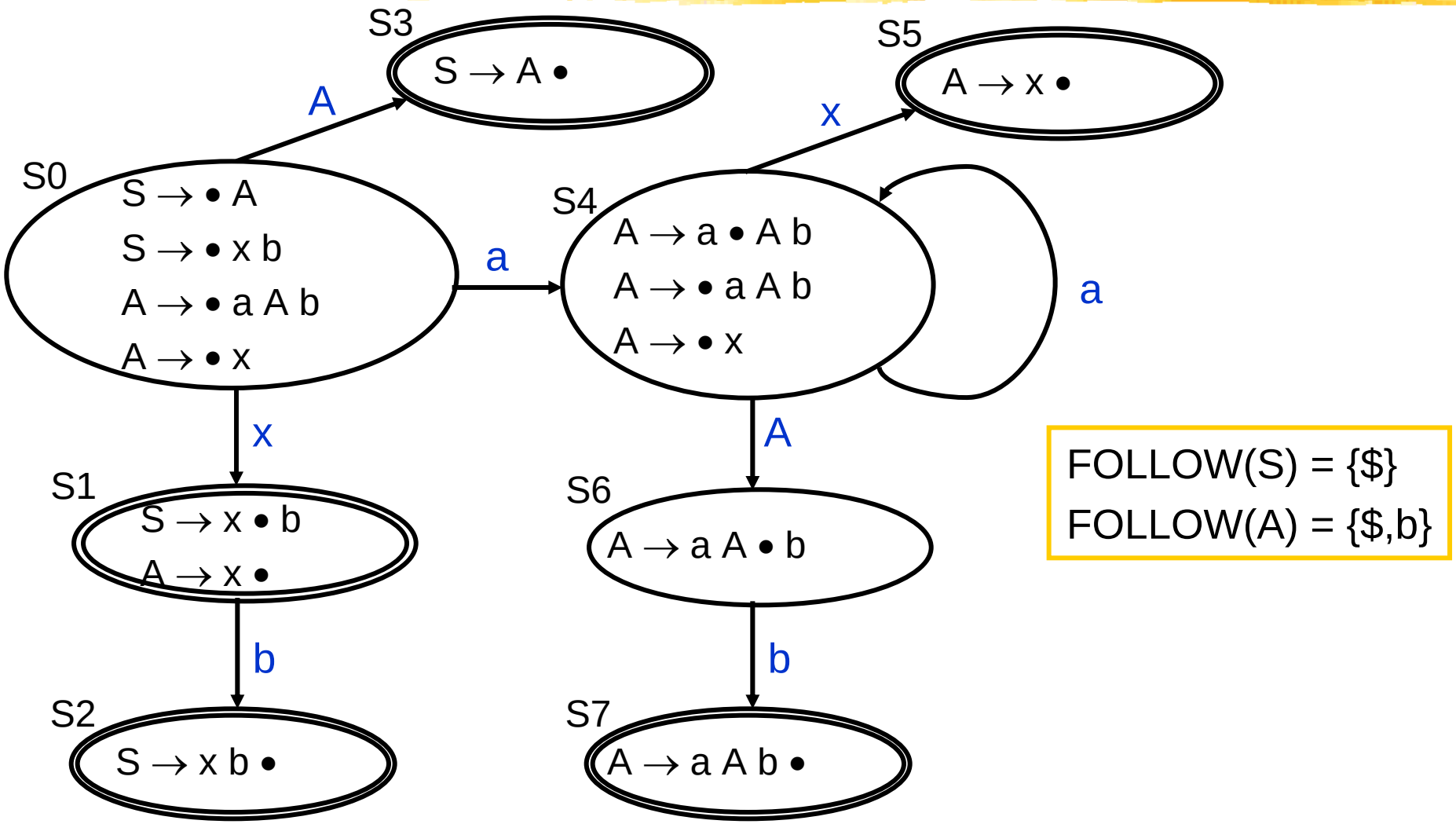
  if set S contains an item $P \rightarrow \alpha \bullet N \beta \ \{\sigma\}$ then

  foreach production rule $N \rightarrow \gamma$

  S must contain the item $N \rightarrow \bullet \gamma \ \{\tau\}$

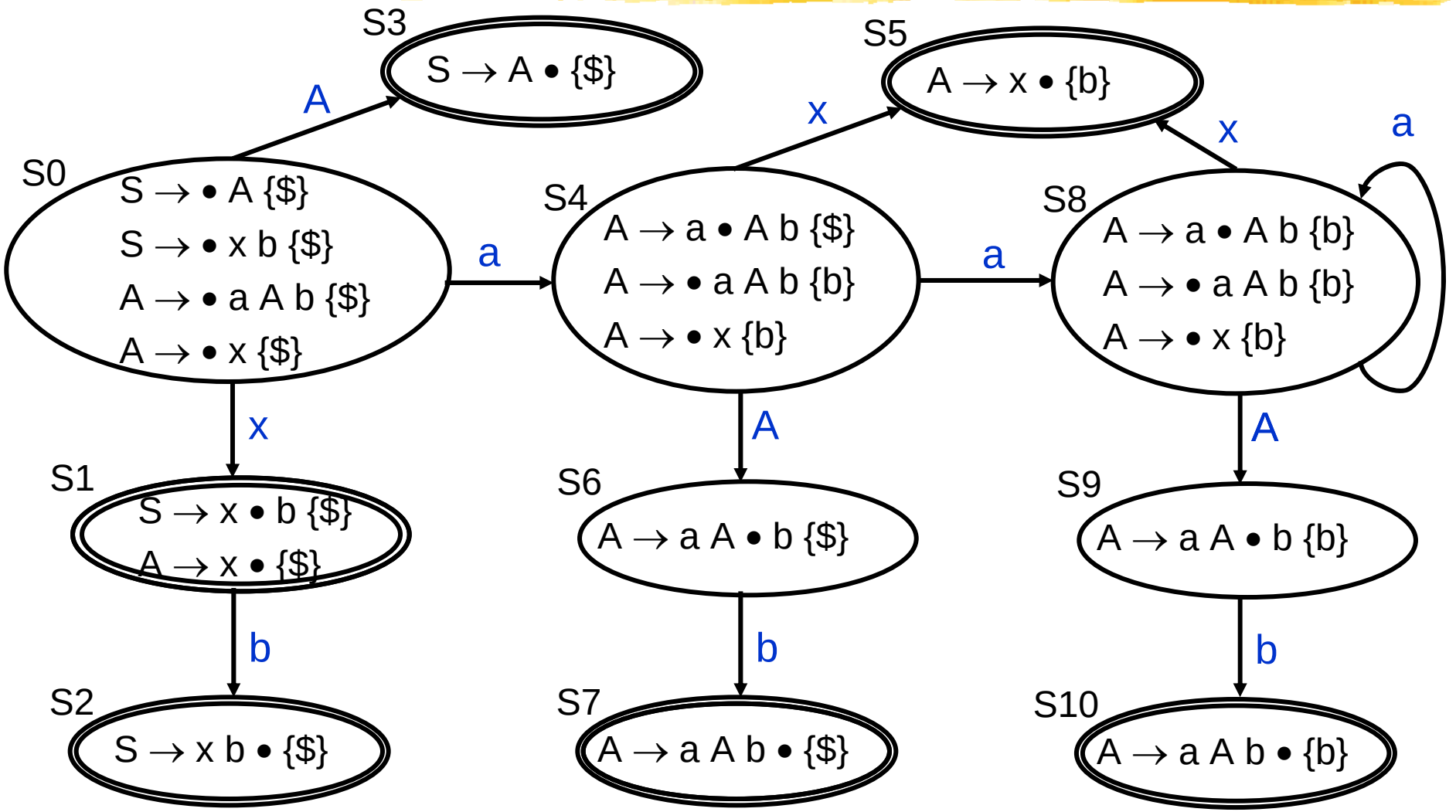
  where τ = FIRST($\beta \ \{\sigma\}$)

LR(0) automaton



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

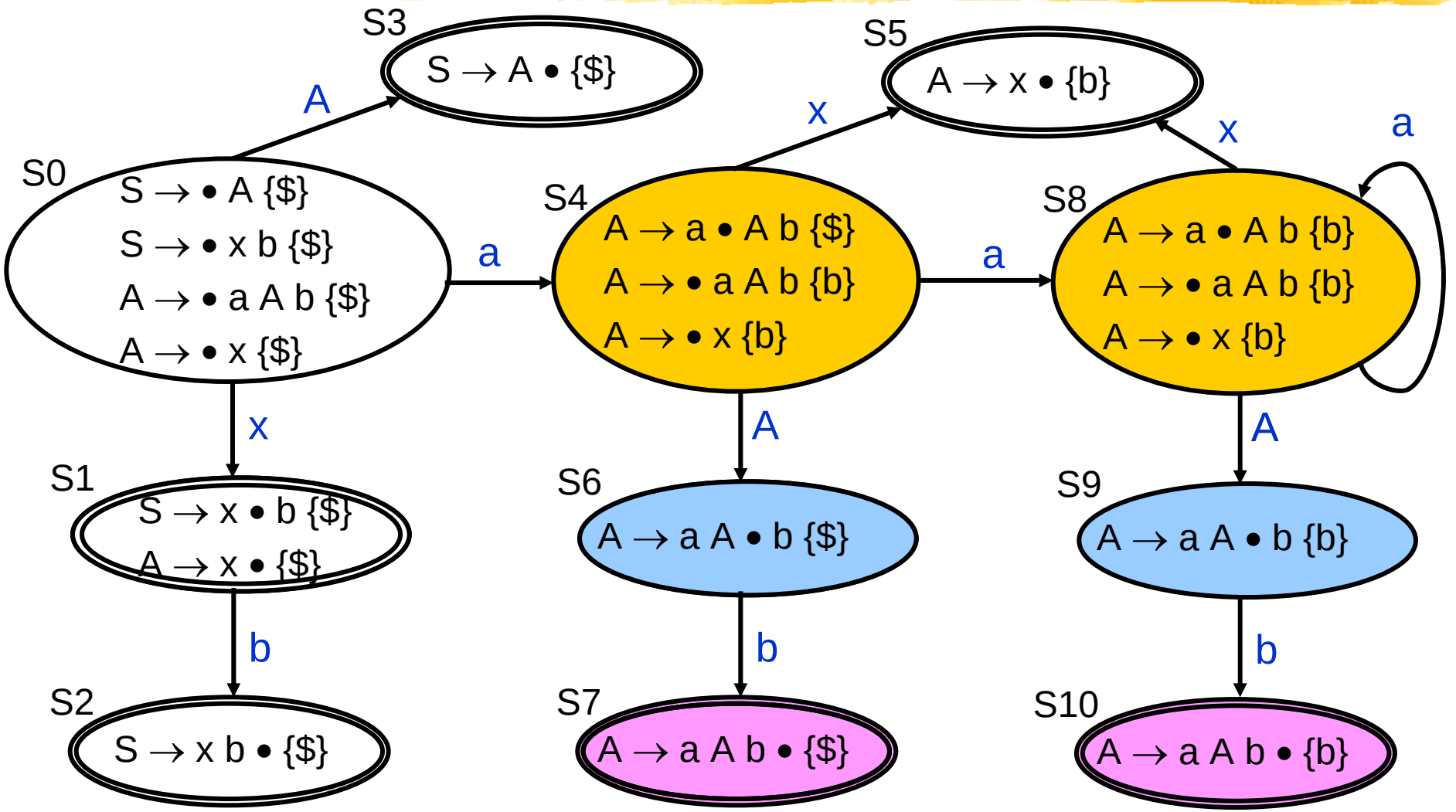
LR(1) automaton



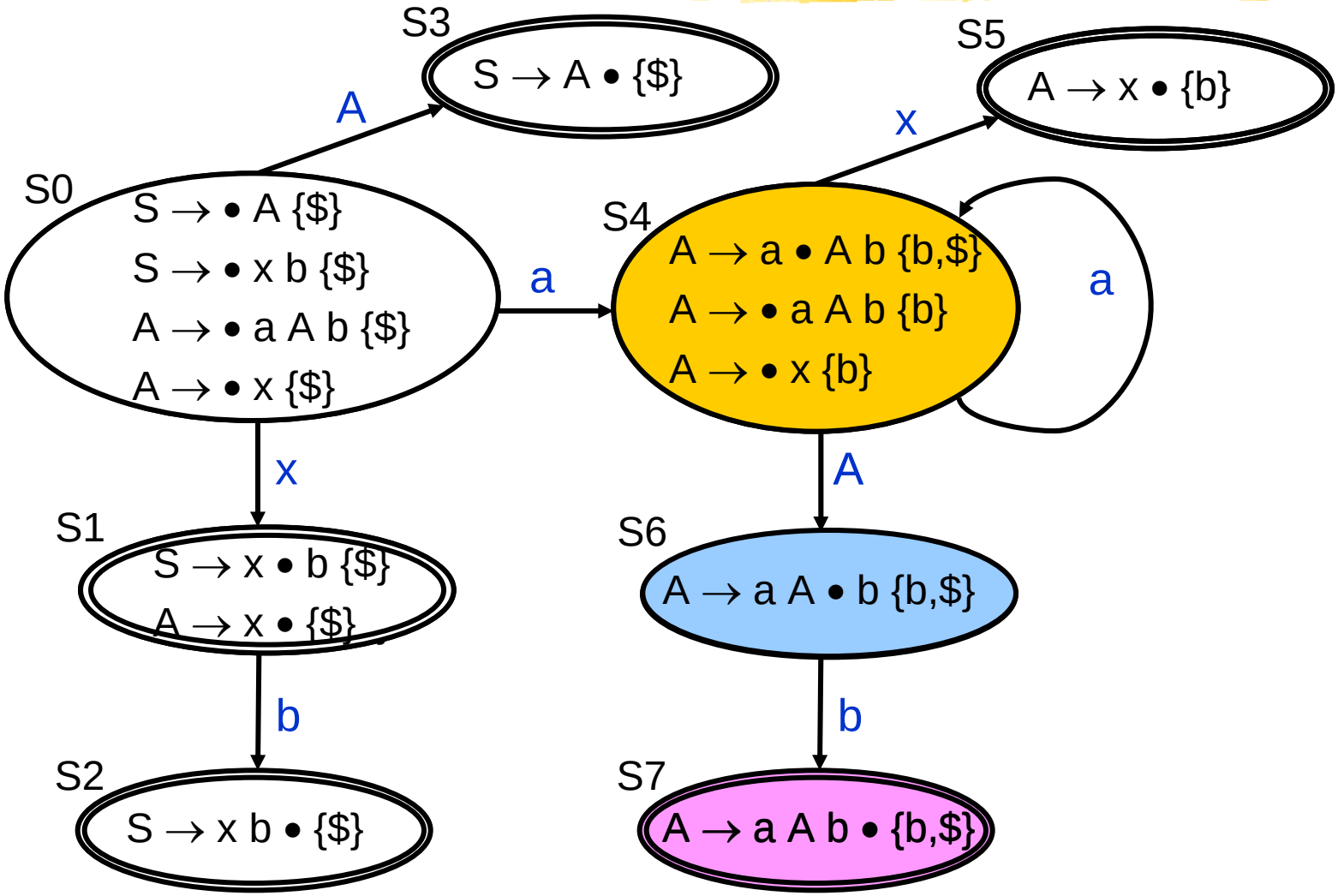
LALR(1) parsing

- LR tables are big
- combine “equal” sets by merging look-ahead sets

LR(1) automaton



LALR(1) automaton



LALR(1) ACTION/GOTO table

state	stack symbol / look-ahead token				
	a	b	x	$	A
0	s4		s1		s3
1		s2		r4	
2				r2	
3				r1	
4	s4		s5		s6
5		r4			
6		s7			
7		r3		r3	

1: S → A

2: S → x b

3: A → a A b

4: A → x

Making grammars LR(1) – or not?

- grammars are often ambiguous

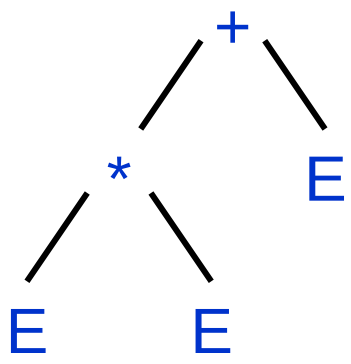
  $E \rightarrow E$ ‘+’ $E \mid E$ ‘*’ $E \mid i$

- handle shift-reduce conflicts
  - (default) longest match: shift
  - precedence directives

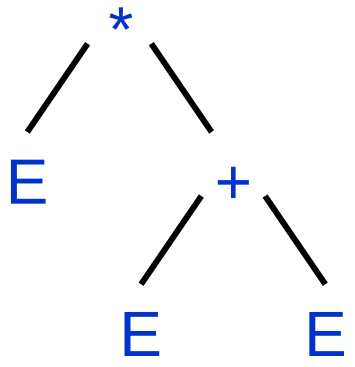
  input: i * i + i

  $E \rightarrow E \bullet$ ‘+’ E

  $E \rightarrow E$ ‘*’ $E \bullet$



reduce



shift

Summary

- syntax analysis: tokens → AST
- bottom-up parsing
  - push-down automaton
  - ACTION/GOTO tables
    - LR(0) NO look-ahead
    - SLR(1) one-token look-ahead, FOLLOW sets to solve shift-reduce conflicts
    - LR(1) SLR(1), but FOLLOW set per item
    - LALR(1) LR(1), but “equal” states are merged