Compiler

Lecture 6

Syntax analysis Part 3

BOTTOM UP Parsing

Parsing Techniques

- *Top-down parsers* *(LL(1), recursive descent)*
  - ❑ Start at the root of the parse tree from the start symbol and grow toward leaves.
  - ❑ Pick a production and try to match the input
  - ❑ Bad "pick" ⇒ may need to backtrack
  - ❑ Some grammars are backtrack-free *(predictive parsing)*

- *Bottom-up parsers* *(LR(1), operator precedence)*
  - ❑ Start at the leaves and grow toward root
  - ❑ We can think of the process as reducing the input string to the start symbol
  - ❑ At each reduction step a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
  - ❑ Bottom-up parsers handle a large class of grammars

Bottom-Up Parsing

- Bottom-up parsing is also known as ***shift-reduce parsing*** because its two main actions are **shift and reduce.**
    - ❑ At each shift action, the current symbol in the input string is pushed to a stack.
    - ❑ At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
- If the substring is chosen correctly, **the right most derivation** of that string is created in the **reverse order.**

Shift-Reduce Parsing

- "Shift-Reduce" Parsing
- Reduce a string to the start symbol of the grammar.
- At every step a particular sub-string is matched (in left-to-right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

abbcde
aAbcde
aAde
aABe
S

Reverse order

Rightmost Derivation:
$S \Rightarrow \mathbf{aABe} \Rightarrow \mathbf{aAde} \Rightarrow \mathbf{aAbcde} \Rightarrow \mathbf{abbcde}$

Handles

- Handle of a string: Substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.

Example: Handle

Grammar

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

Handle

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ A\ \mathbf{e}$

… ?

NOT a handle, because
further reductions will fail
(result is not a sentential form)

A Stack Implementation of A Shift-Reduce Parser

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

  Stack Input

  $ w$
- Acceptance configuration

  Stack Input

  $S $
- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error

A Stack Implementation of A Shift-Reduce Parser

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$

Stack	Input	Action
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$\$\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$T	$* \mathbf{id}_2$ $	shift
$T *	$\mathbf{id}_2$ $	shift
$\$T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$T * F	$	reduce by $T \rightarrow T * F$
$T	$	reduce by $E \rightarrow T$
$E	$	accept

Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.

2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - LR – most general LR parser
     - LALR – intermediate LR parser (lookhead LR parser)
   - SLR, LR and LALR work same, only their parsing tables are different.

LR Parsers

- The most powerful shift-reduce parsing is:

LR(k) parsing.

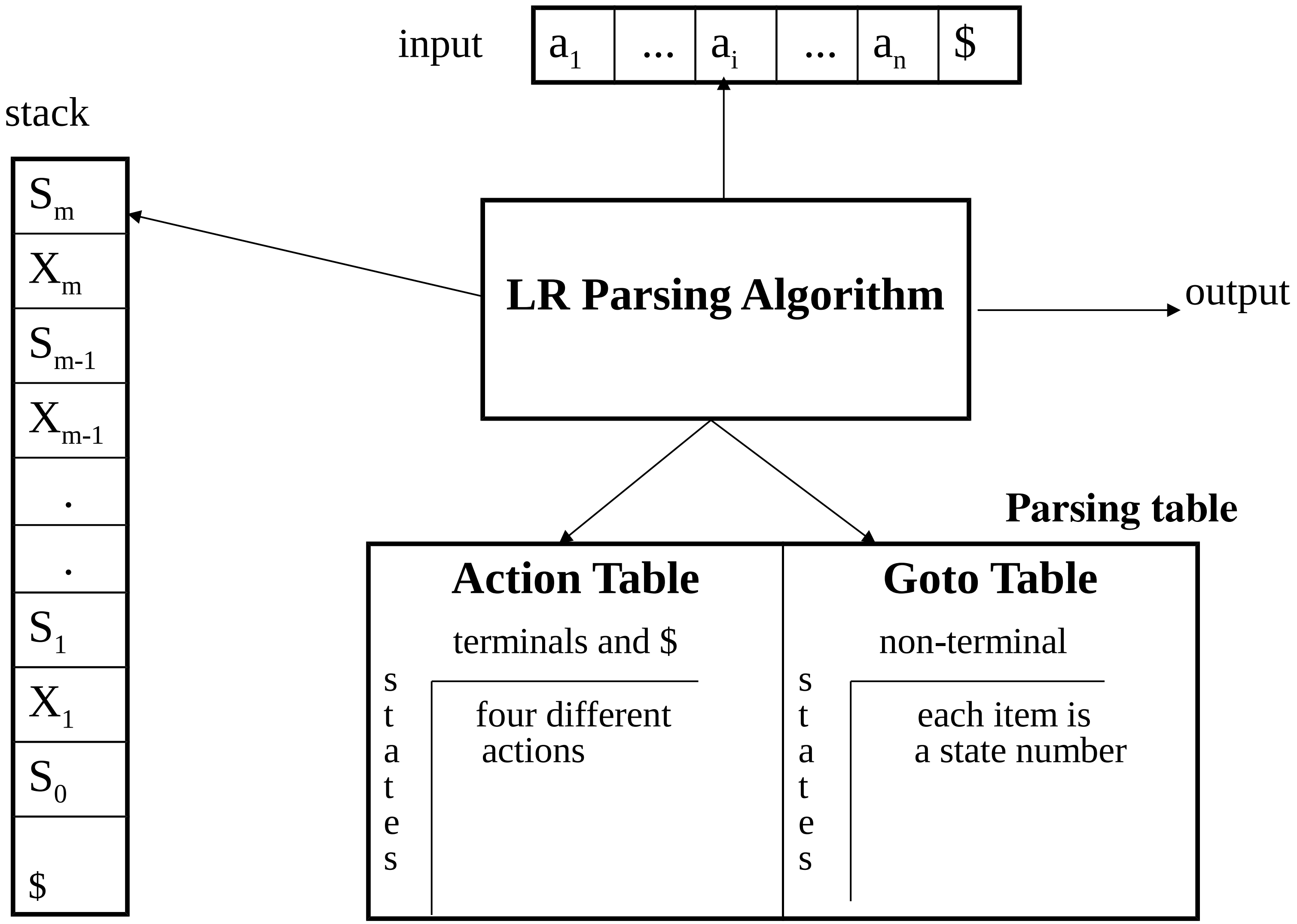
left to right scanning — right-most derivation — k lookhead (k is omitted □ it is 1)

- **LR parsing is attractive because:**
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers. LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

Potential Problems

- How do we know which action to take: whether to shift or reduce, and which production to apply
- Issues
  - Sometimes can reduce but should not
  - Sometimes can reduce in different ways

LR Parsing Algorithm



Constructing SLR Parsing Tables – LR(0) Items

- An LR parser makes shift-reduce decisions by maintaining **states to keep track of where we are in a parse.**
- States represent sets of **items.**
- An item of a grammar G is a production of G with a dot at some position of the body.
  - Example

$$A \rightarrow X\,Y\,Z$$

$$\text{items}\left\{\begin{array}{l} A \rightarrow \;.\; X\,Y\,Z \\ A \rightarrow X \;.\; Y\,Z \\ A \rightarrow X\,Y \;.\; Z \end{array}\right.$$

$$A \rightarrow X\,Y\,Z\;.$$

$$A \rightarrow \underbrace{X}_{\text{stack}} \;.\; \underbrace{Y\,Z}_{\text{next derivations with input strings}}$$

Note that production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

LR(0) Items

- Augmented the grammar
  - If G is a grammar with start symbol S, then G' is the augmented grammar for G with new start symbol S' and new production S' $\rightarrow$ S\$.
  - The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.
- Closure function
- Goto function

Closure function

- To construct states, we begin with a particular LR(0) item and construct its **closure**
  - the closure adds more items to a set when the "." appears to the left of a non-terminal
  - if the state includes **X → s . Y s'** and **Y → t** is a rule then the state also includes **Y → . t**

S → (L) | x
L → S | L , S

Augmented Grammar:

0. S' --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S

First state

$$\left.\begin{array}{l} S' \text{ --> } . S \$ \\ S \text{ --> } . (L) \\ S\square.x \end{array}\right\}$$

Full Closure

Goto Function

- Function *Goto*(*I*, *X*)
  - I is a set of items
  - X is a grammar symbol
  - *Goto*(*I*, *X*) is defined to be the closure of the set of all items [$A \to \alpha\ X \cdot \beta$] such that [$A \to \alpha \cdot X\beta$] is in *I*.
  - Goto function is used to define the transitions in the LR(0) automation for a grammar.

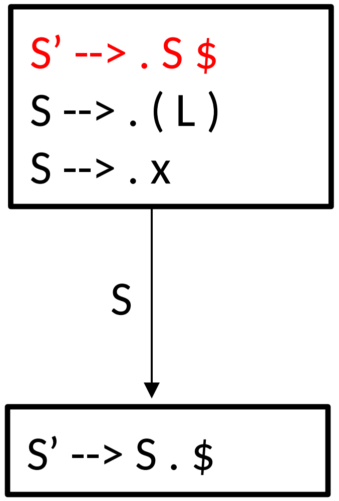
Augmented Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S

S’ --> .S $
S --> . (L)
S --> . x

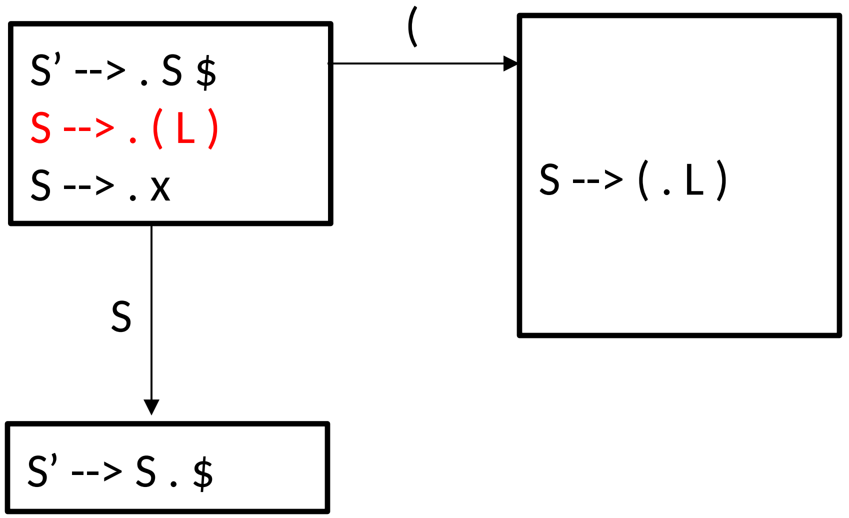
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



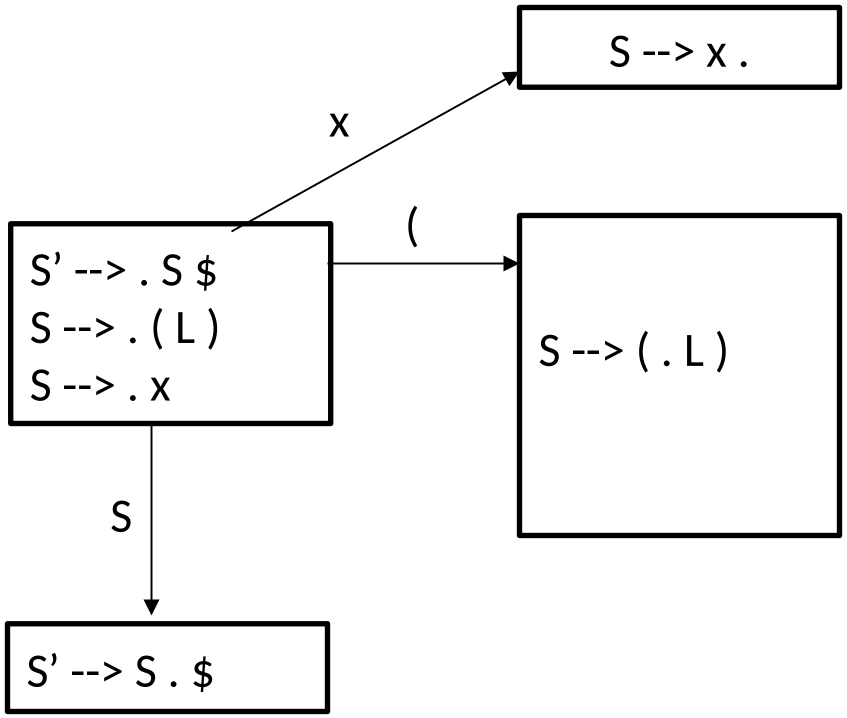
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



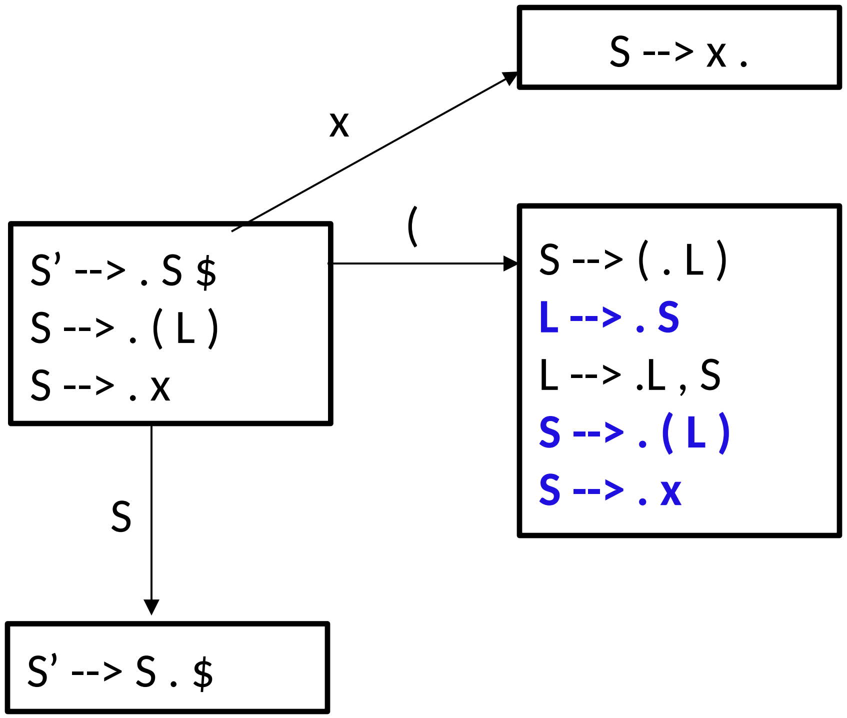
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



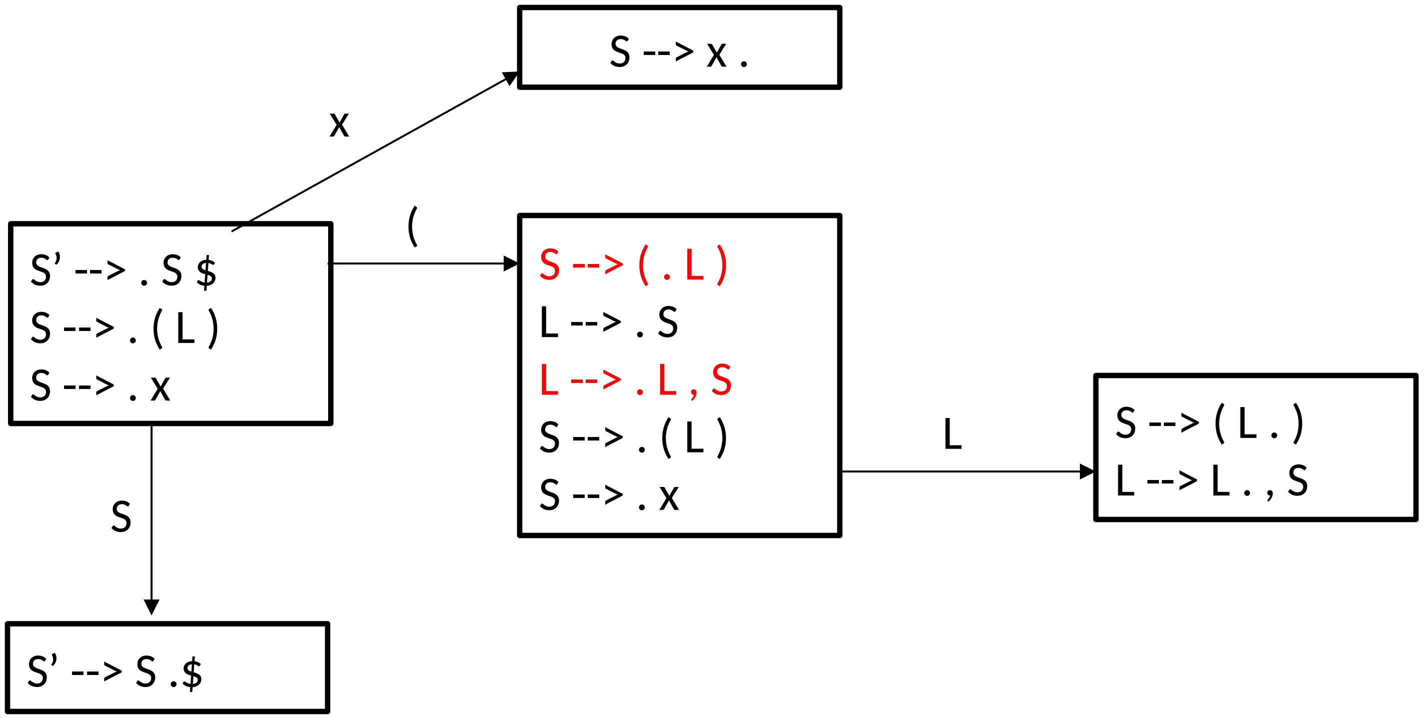
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



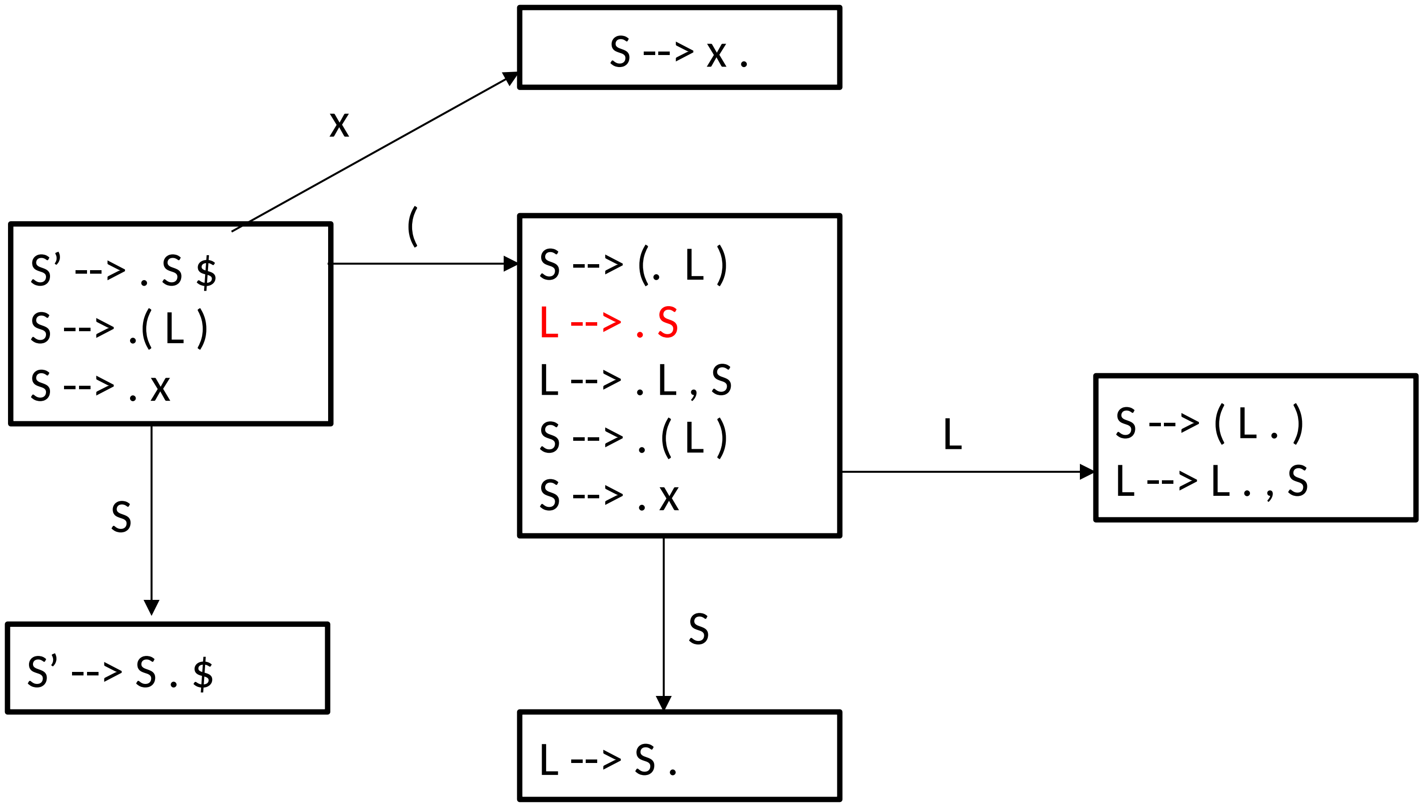
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



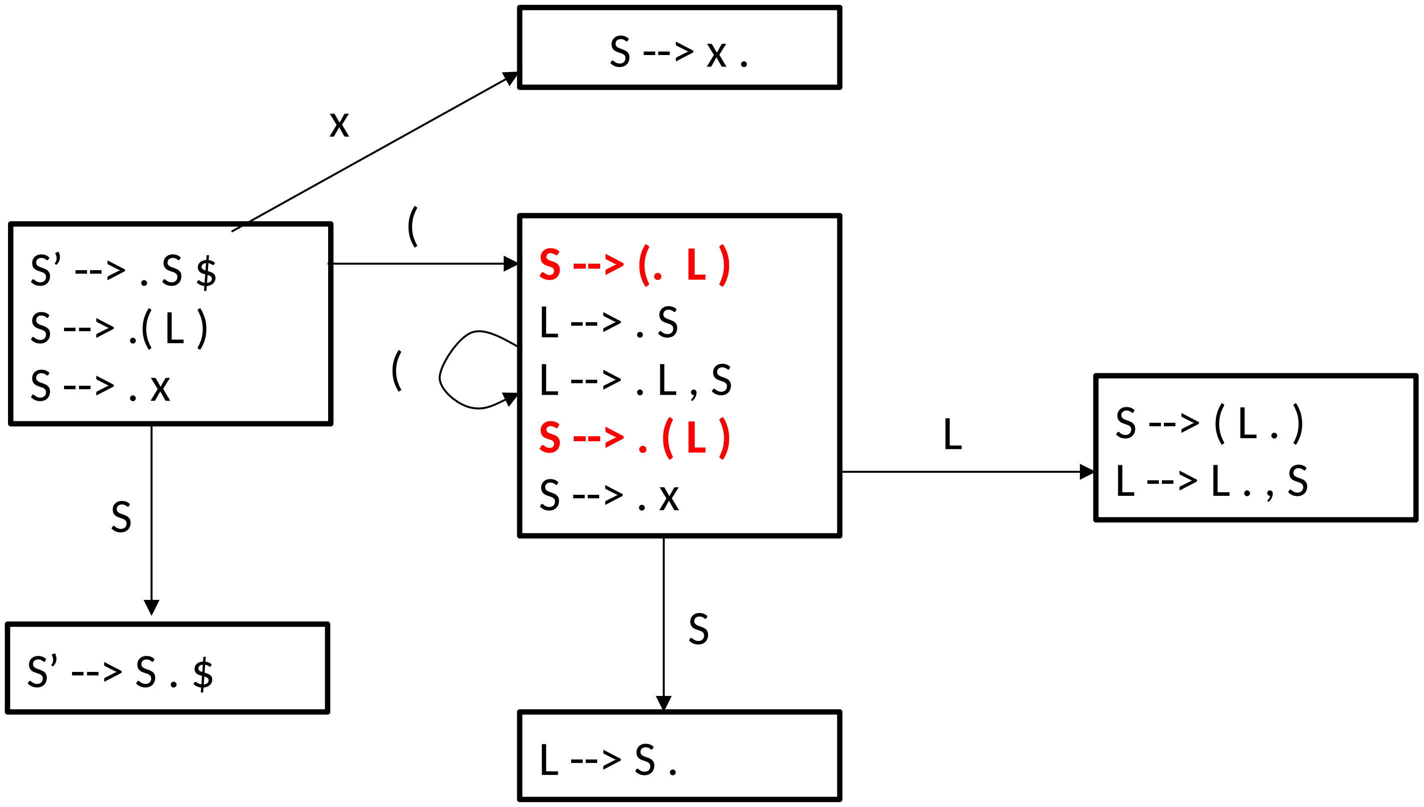
Grammar: G'

0. S' --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



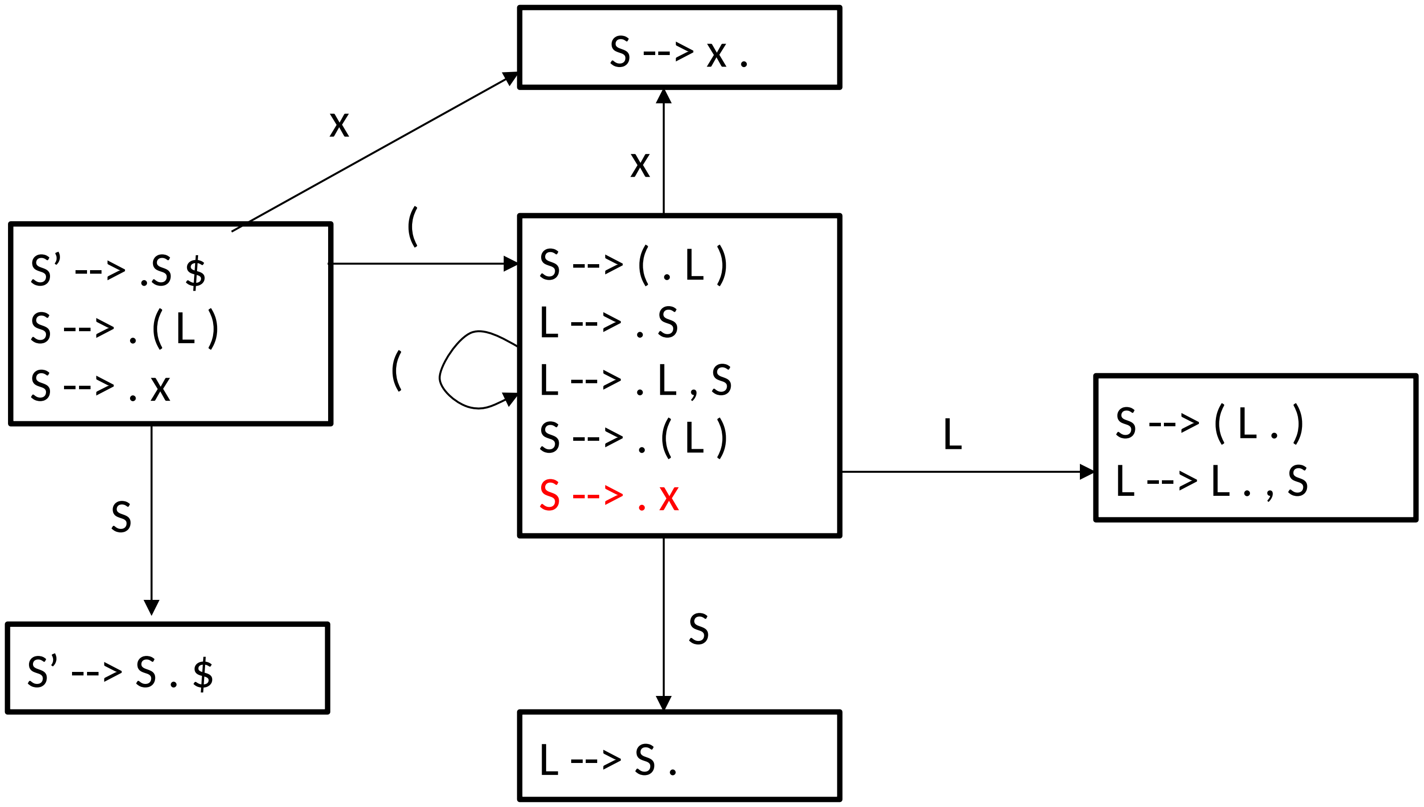
Grammar: G'

0. S' --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



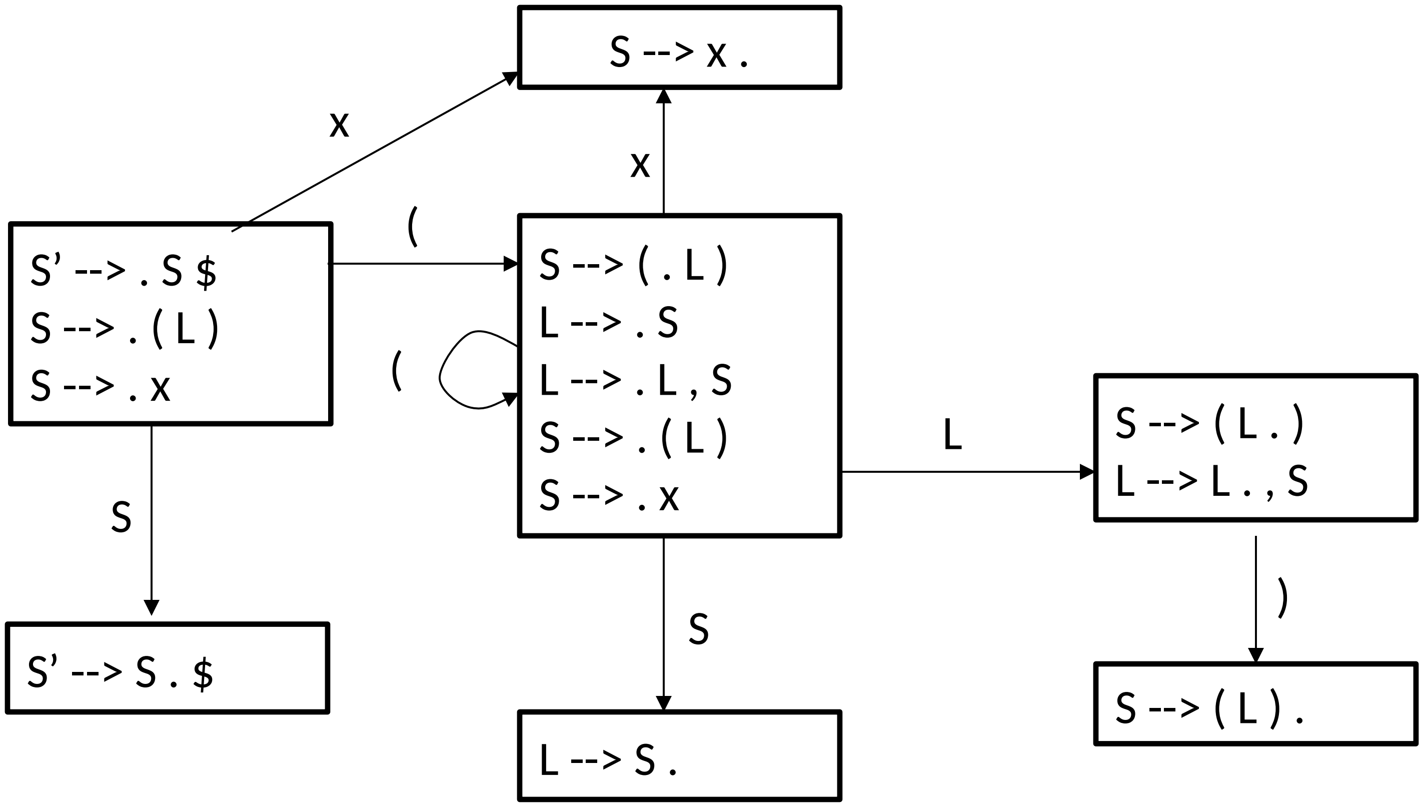
Grammar: G'

0. S' --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



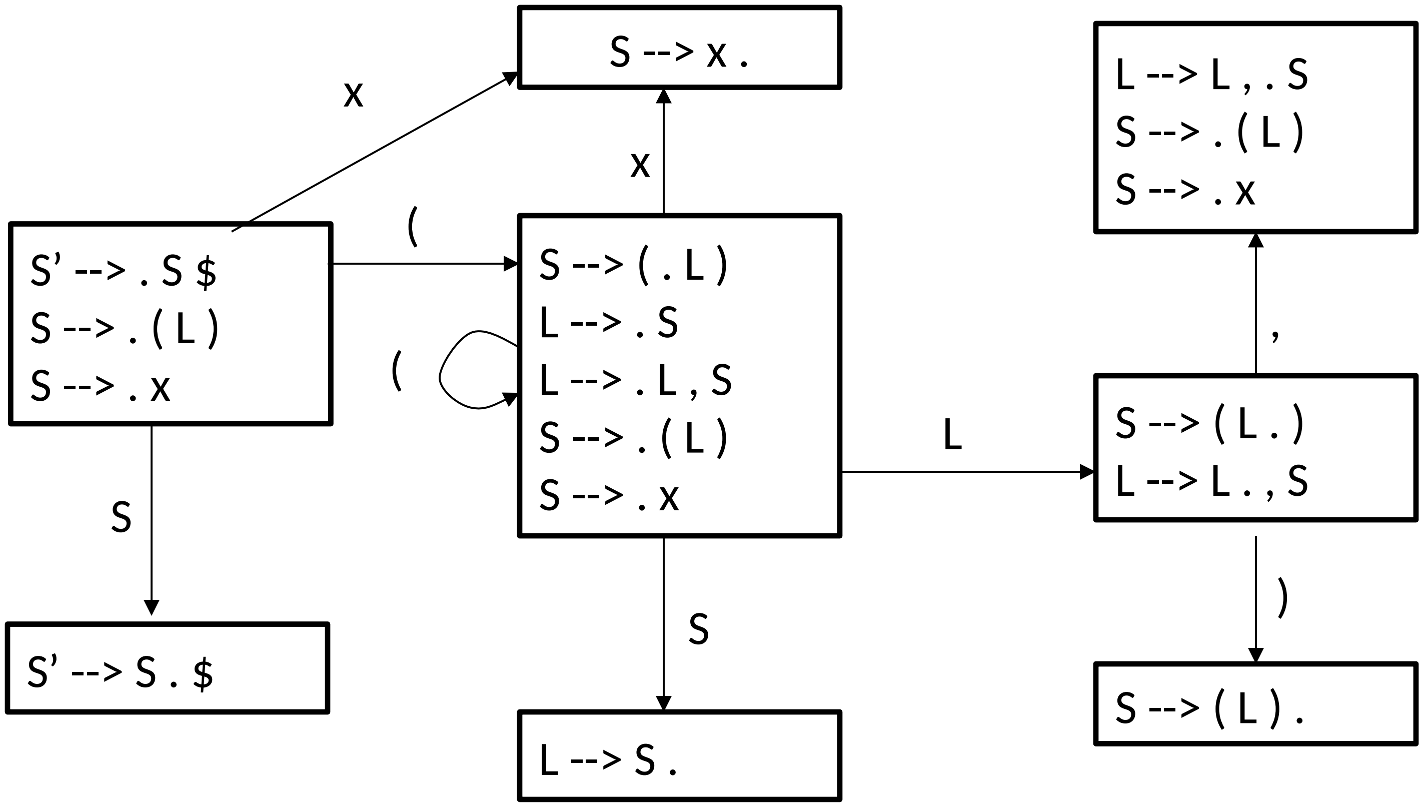
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



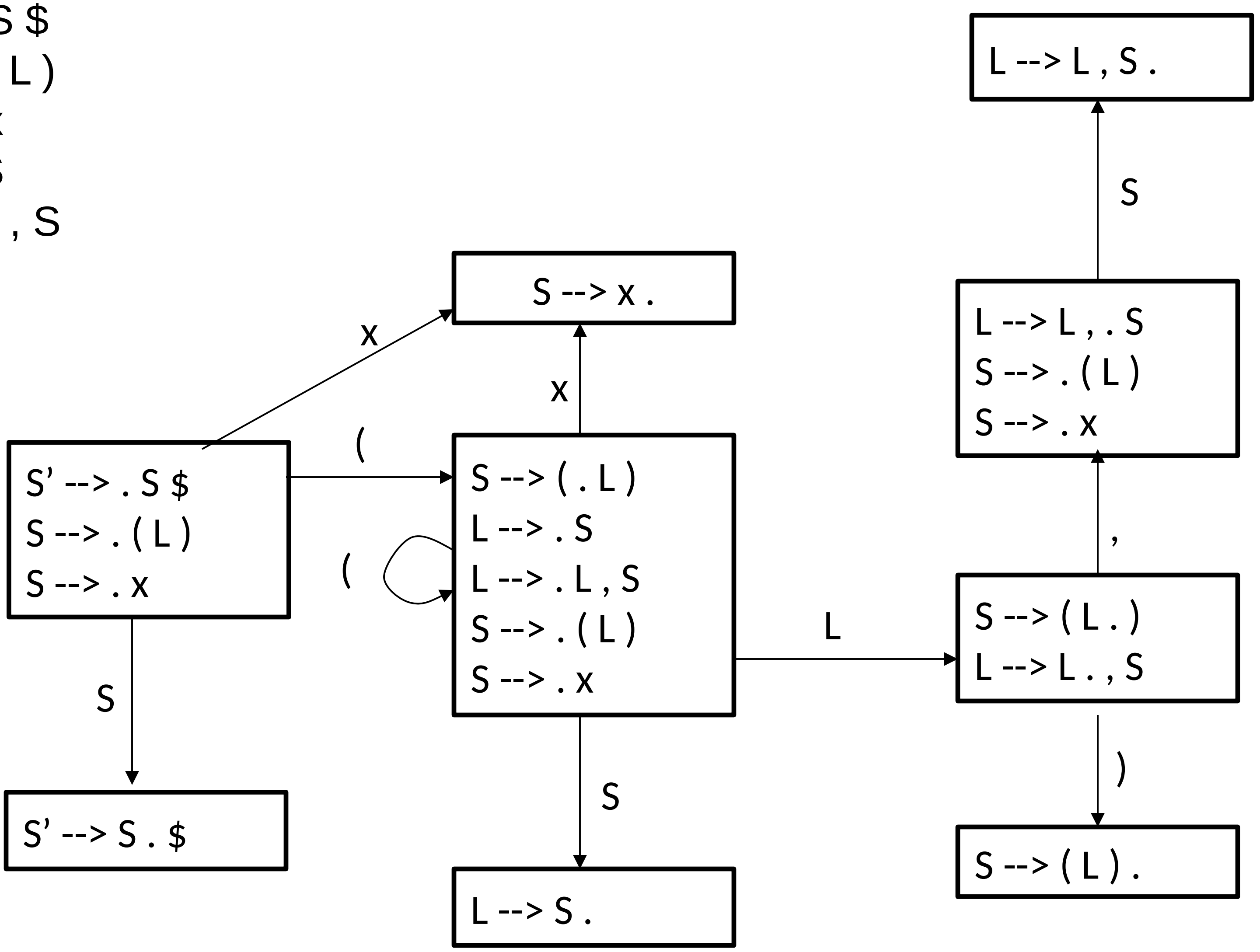
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



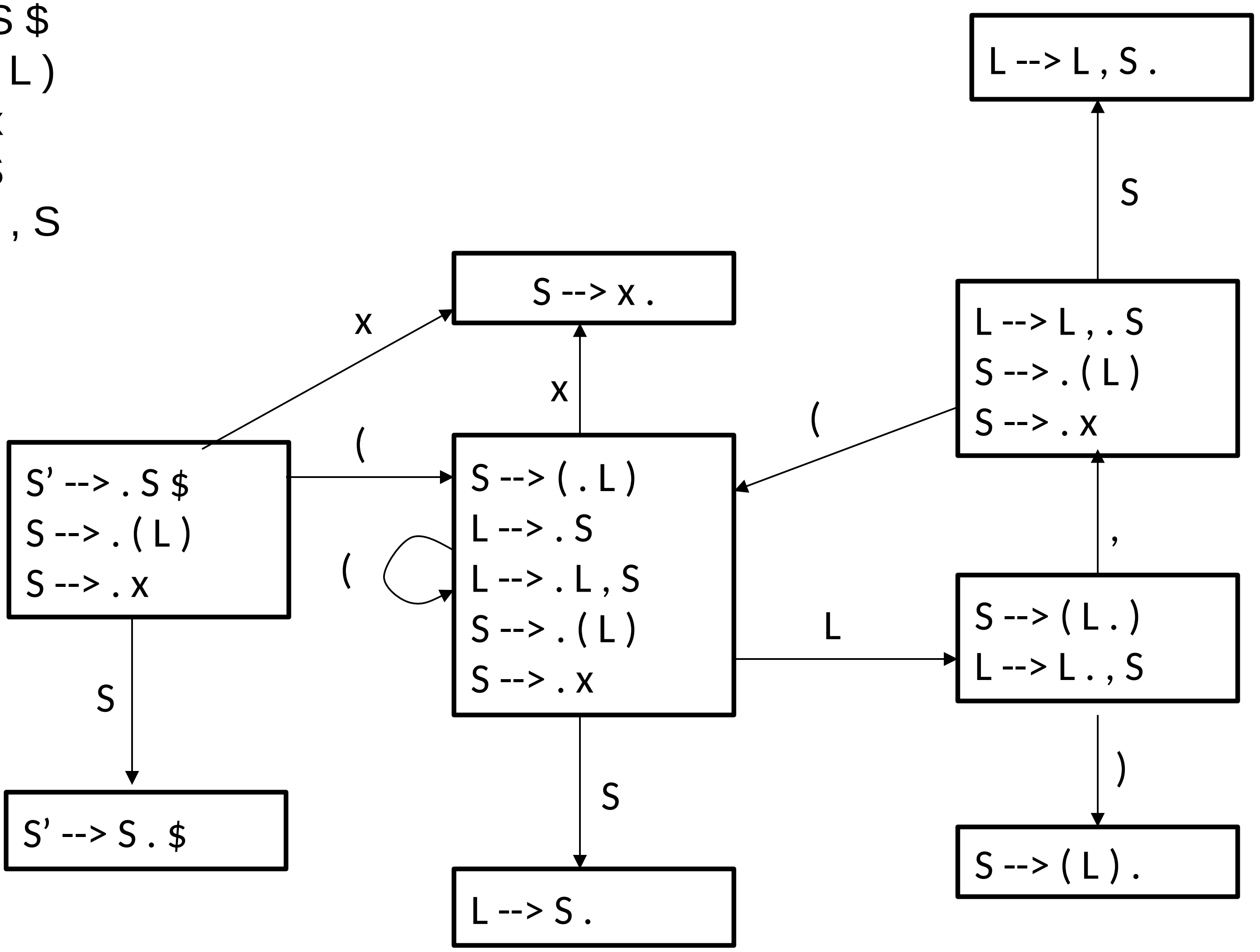
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



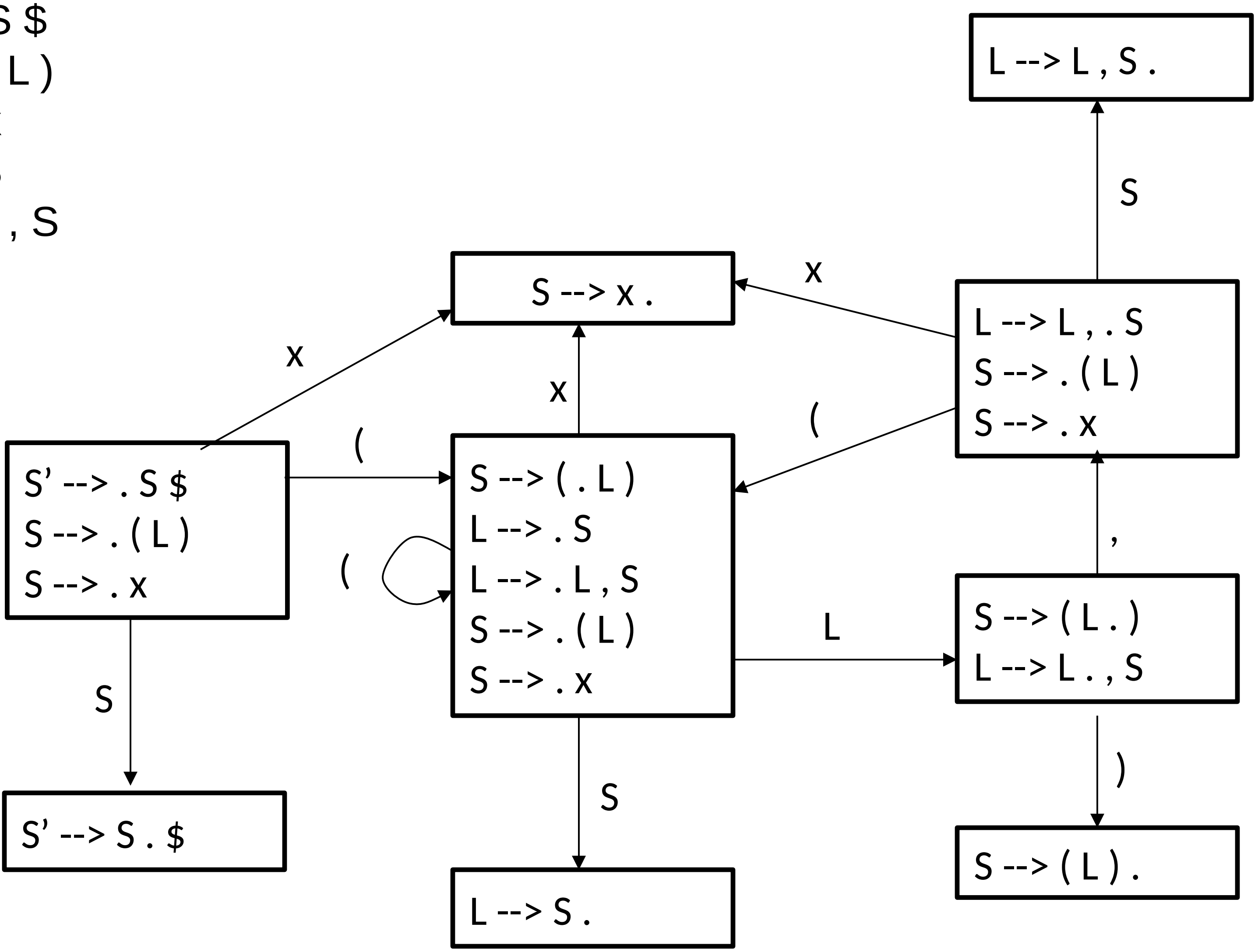
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



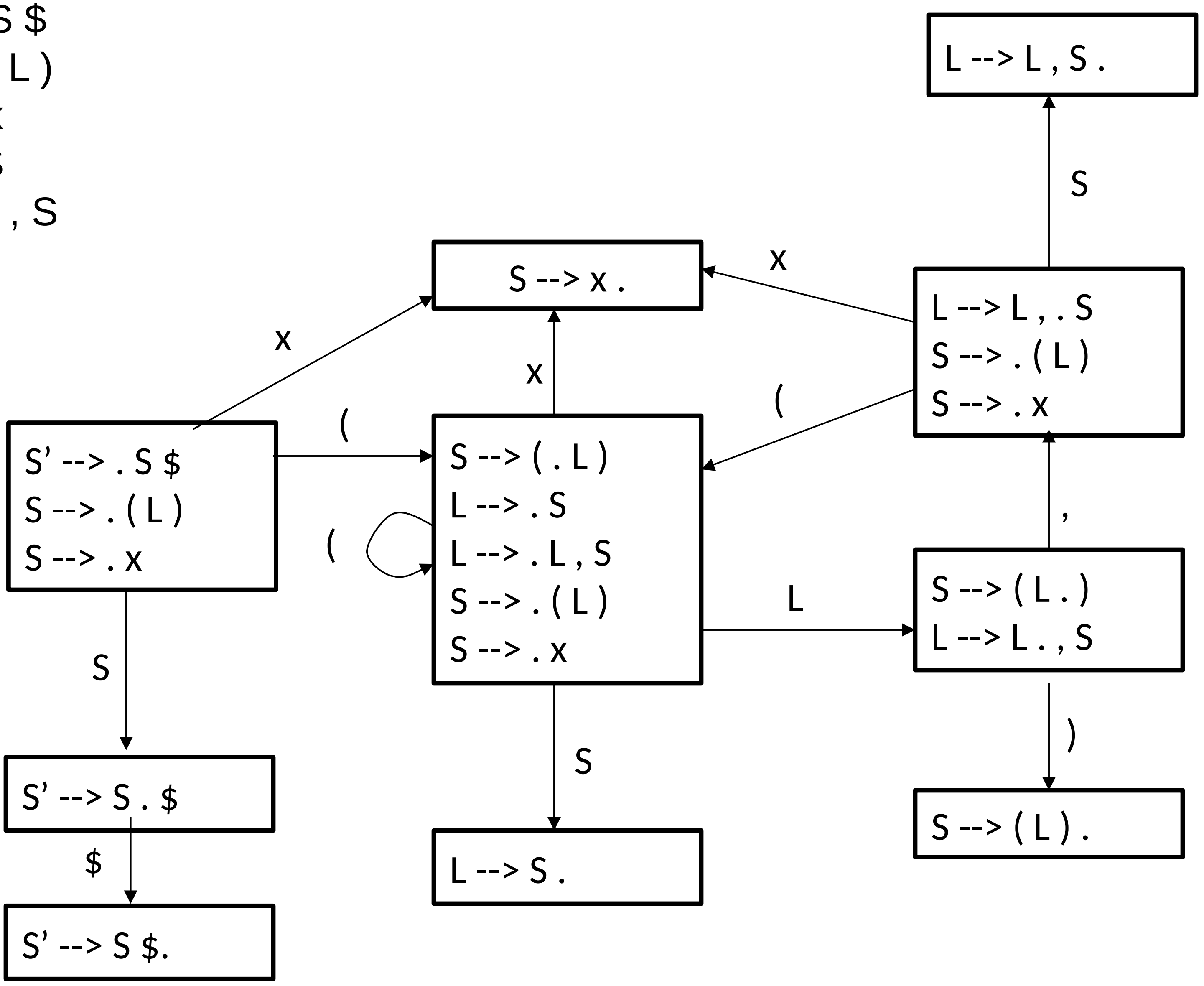
Grammar: G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



Grammar: G’

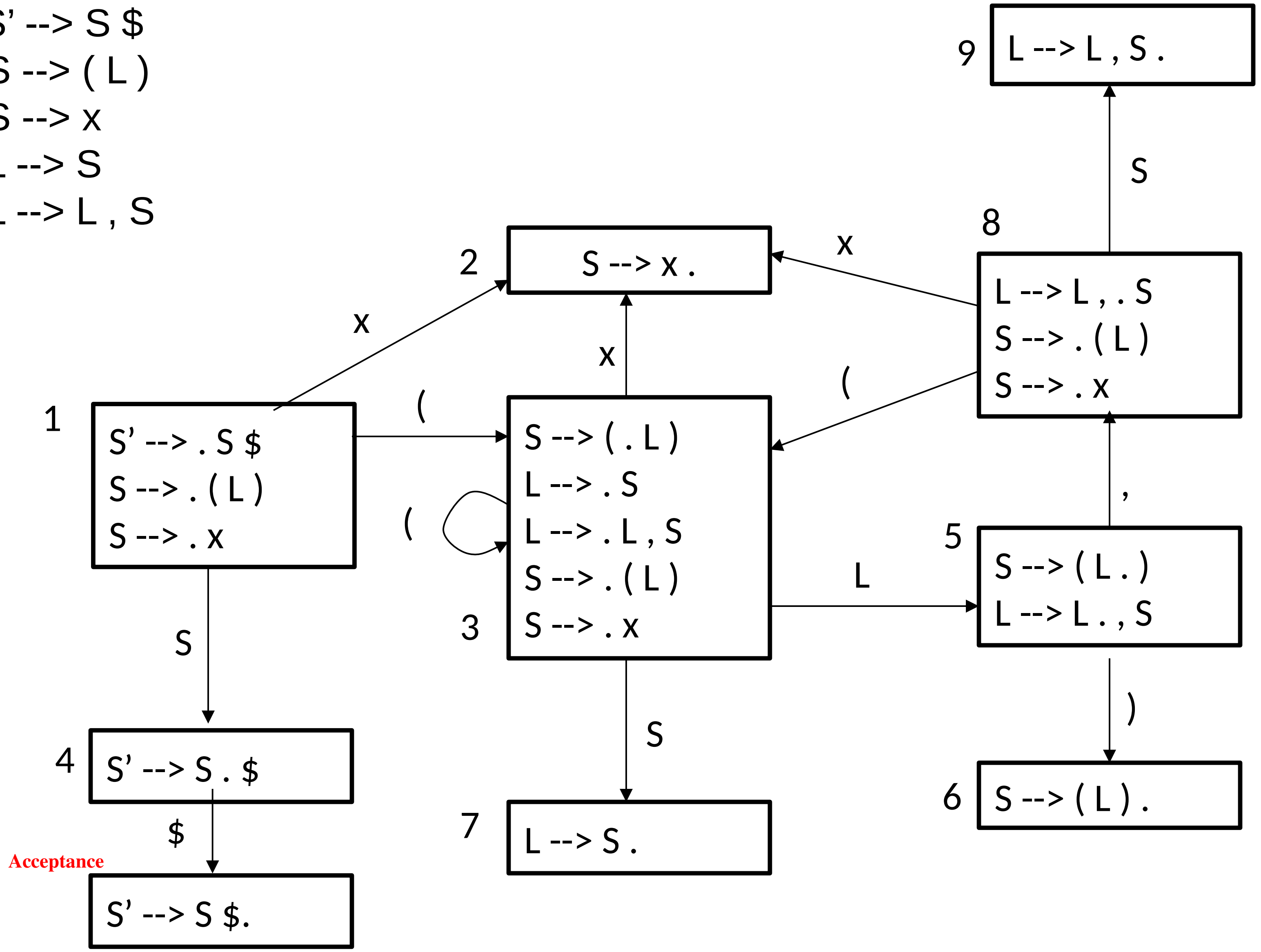
0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S



Grammar: G’

Complete LR(0) states for G’

0. S’ --> S $
- S --> (L)
- S --> x
- L --> S
- L --> L , S

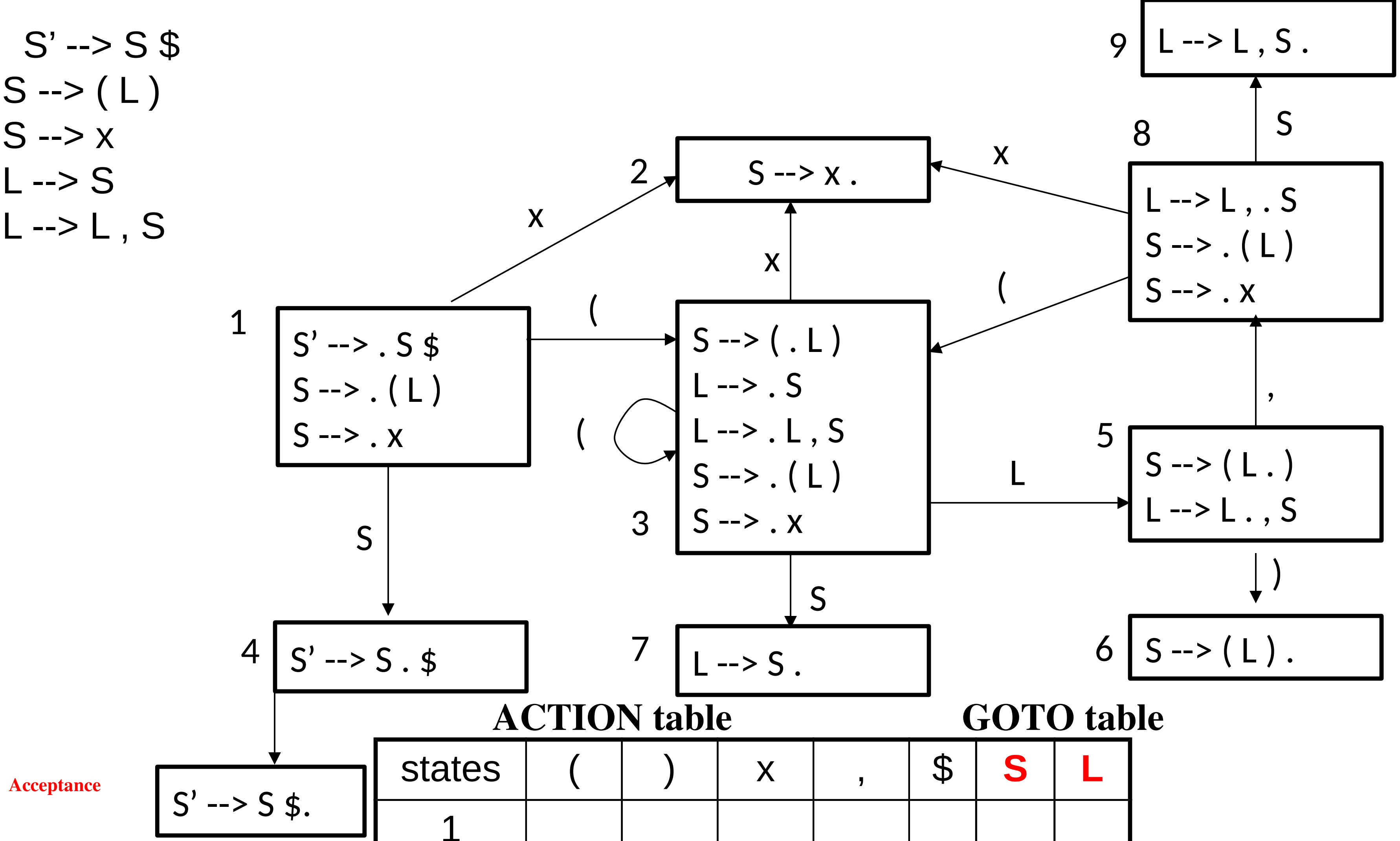


Constructing SLR Parsing Tables

- Input :Augment the grammar with $S' \rightarrow S$
- *Output the SLR parsing table*
- *Method :*

1. Construct the canonical collection of sets of LR(0) items for G'. $C \leftarrow \{I_0,\dots,I_n\}$
2. State *i* is constructed from *Ii*
   - If $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and $goto(I_i, a)=I_j$ then set $action[i, a]=$**shift** *j*. *Here a must be a terminal*
   - If $[A \rightarrow \alpha \bullet] \in I_i$ then set $action[i,a]=$reduce $A \rightarrow \alpha$ for all $a \in FOLLOW(A)$ (apply only if $A \neq S'$)
   - If $[S' \rightarrow S\bullet]$ is in I_i then set $action[i,\$]=$accept
3. If $goto(I_i, A)=I_j$ then set $goto[i, A]=j$ set **goto table**
4. All entries not defined by rules (2) and (3) are made "error".
5. The initial state *i* is the I_i holding item $[S' \rightarrow \bullet S]$

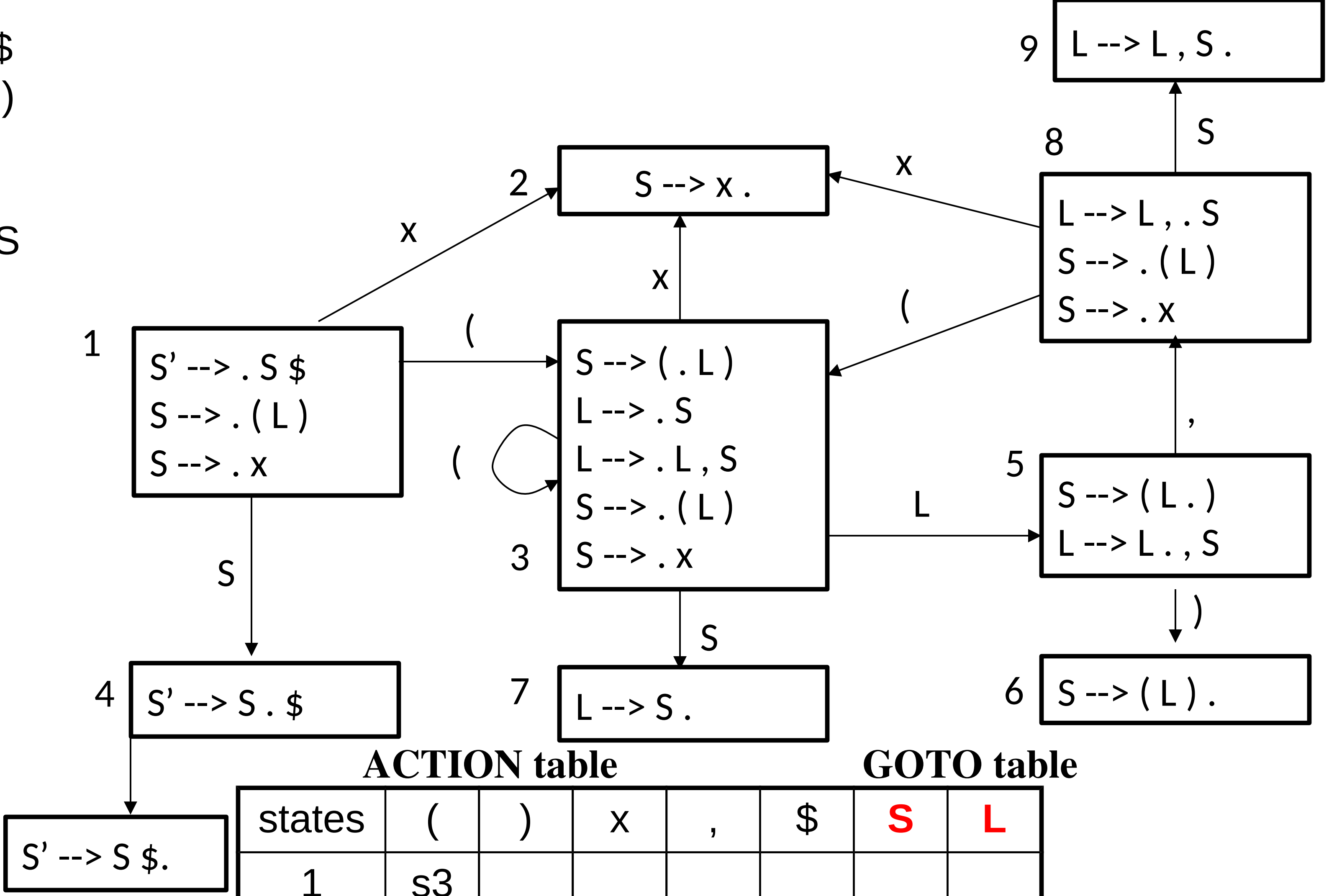
S' --> S $
S --> (L)
S --> x
L --> S
L --> L , S



ACTION table **GOTO table**

states	(	)	x	,	$	S	L
1							
2							
3							
4							

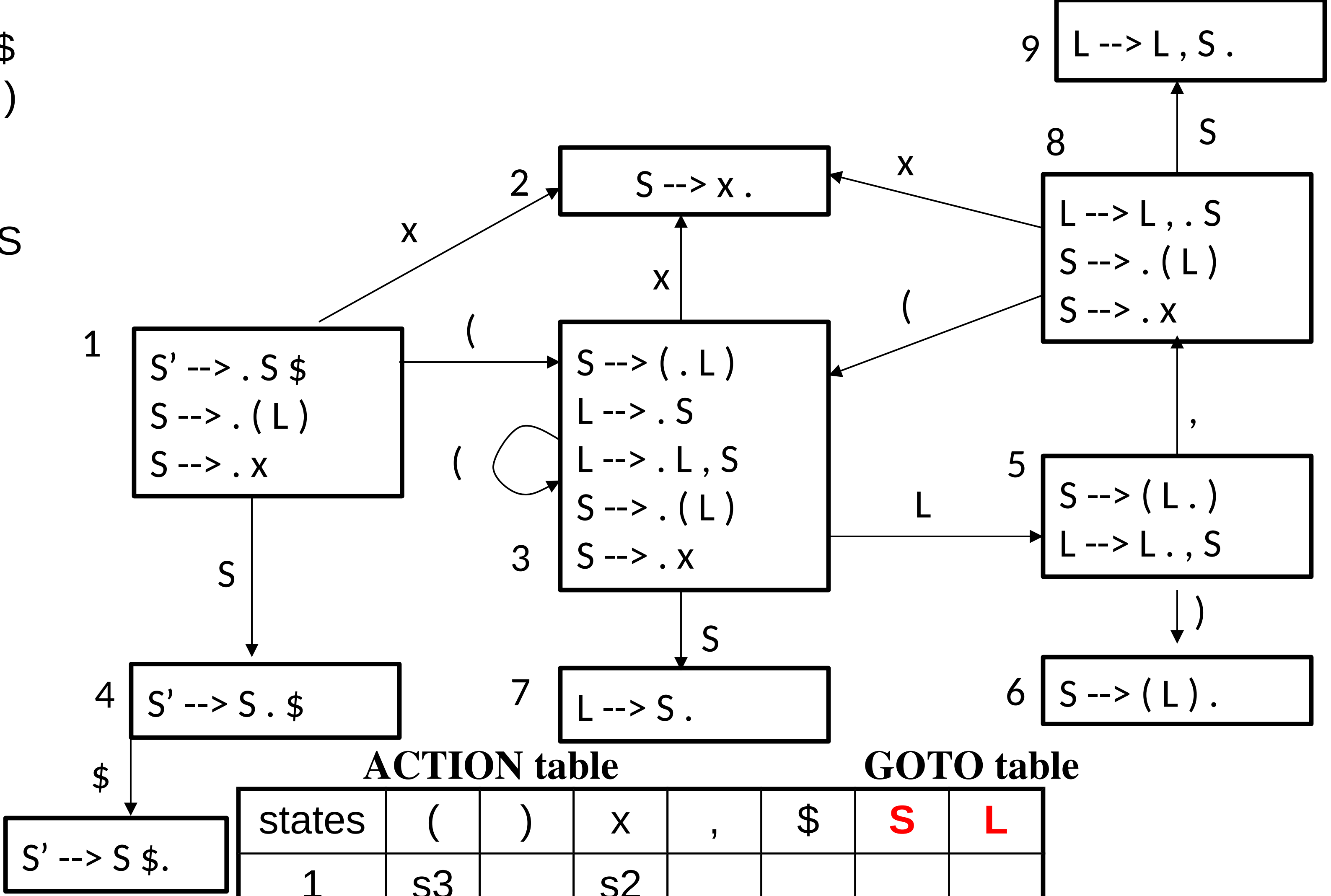
S' --> S $
S --> (L)
S --> x
L --> S
L --> L , S



ACTION table **GOTO table**

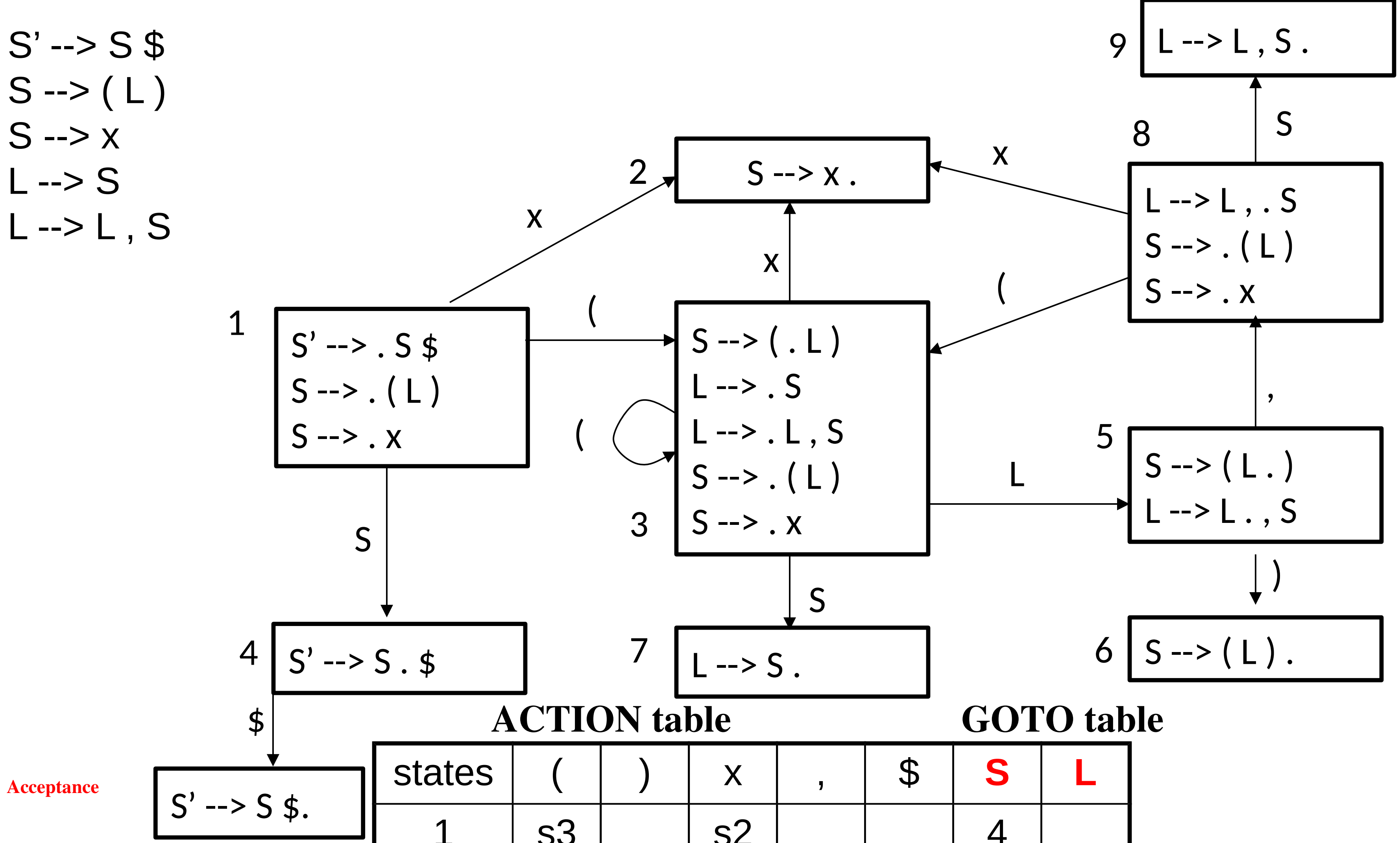
states	(	)	x	,	$	S	L
1	s3						
2							
3							
4							

S’ --> S $
S --> (L)
S --> x
L --> S
L --> L , S



	ACTION table					GOTO table	
states	(	)	x	,	$	S	L
1	s3		s2				
2							
3							
4							

S’ --> S $
S --> (L)
S --> x
L --> S
L --> L , S

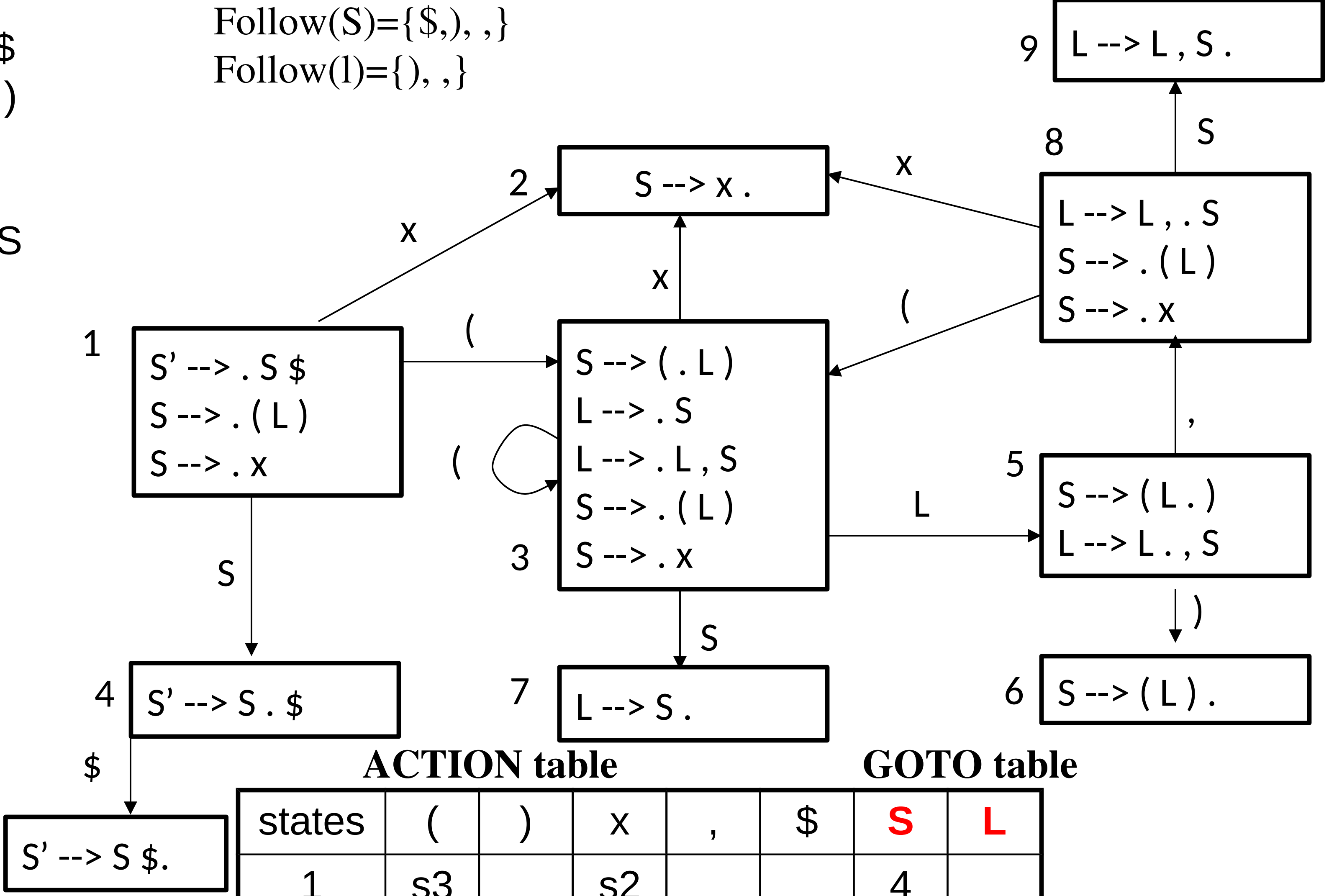


ACTION table **GOTO table**

states	(	)	x	,	$	S	L
1	s3		s2			4	
2							
3							
4							

S' --> S $
S --> (L)
S --> x
L --> S
L --> L , S

Follow(S)={$,), ,}
Follow(l)={), ,}

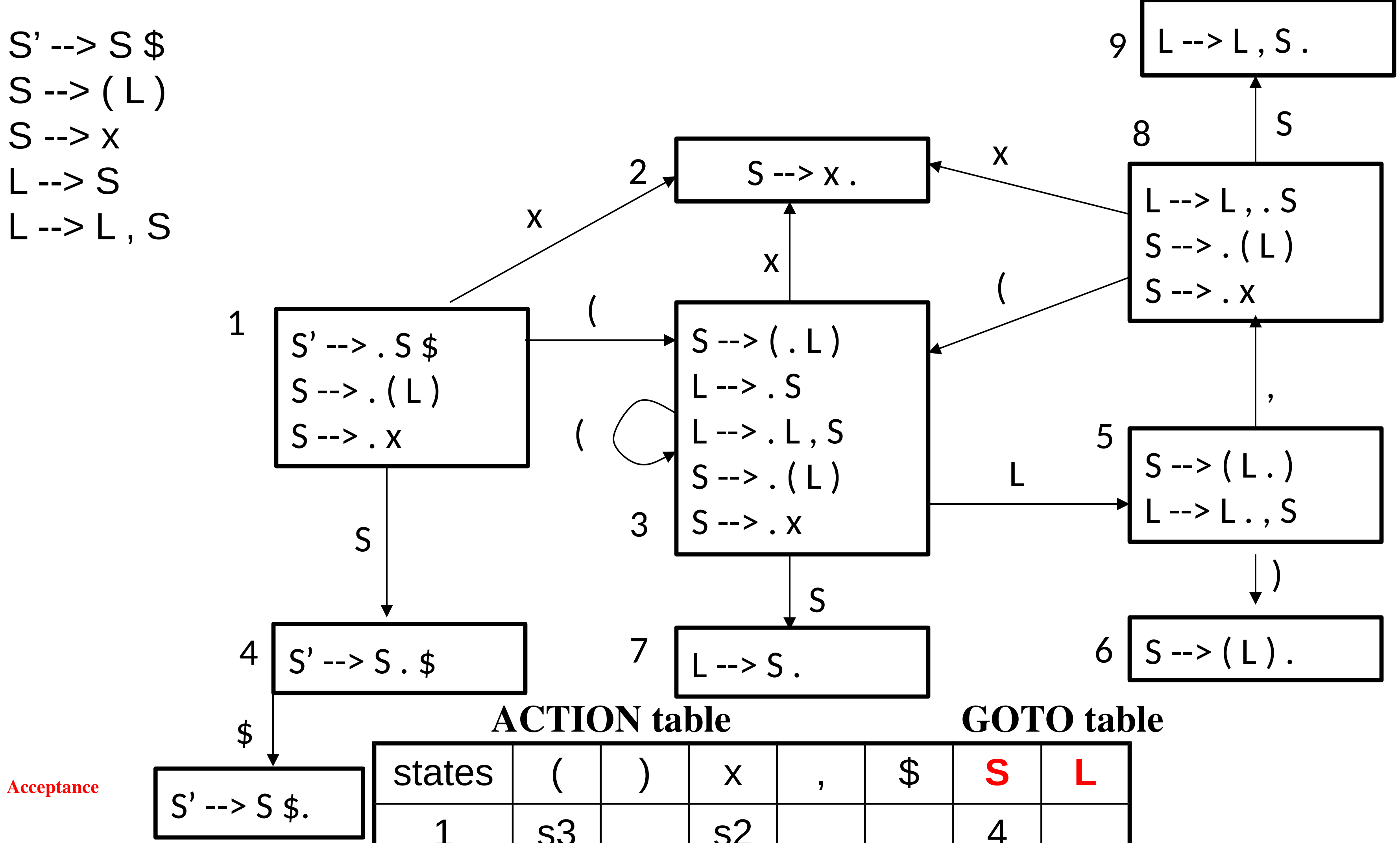


Acceptance

ACTION table **GOTO table**

states	(	)	x	,	$	S	L
1	s3		s2			4	
2		r2		r2	r2		
3	s3		s2				
4							

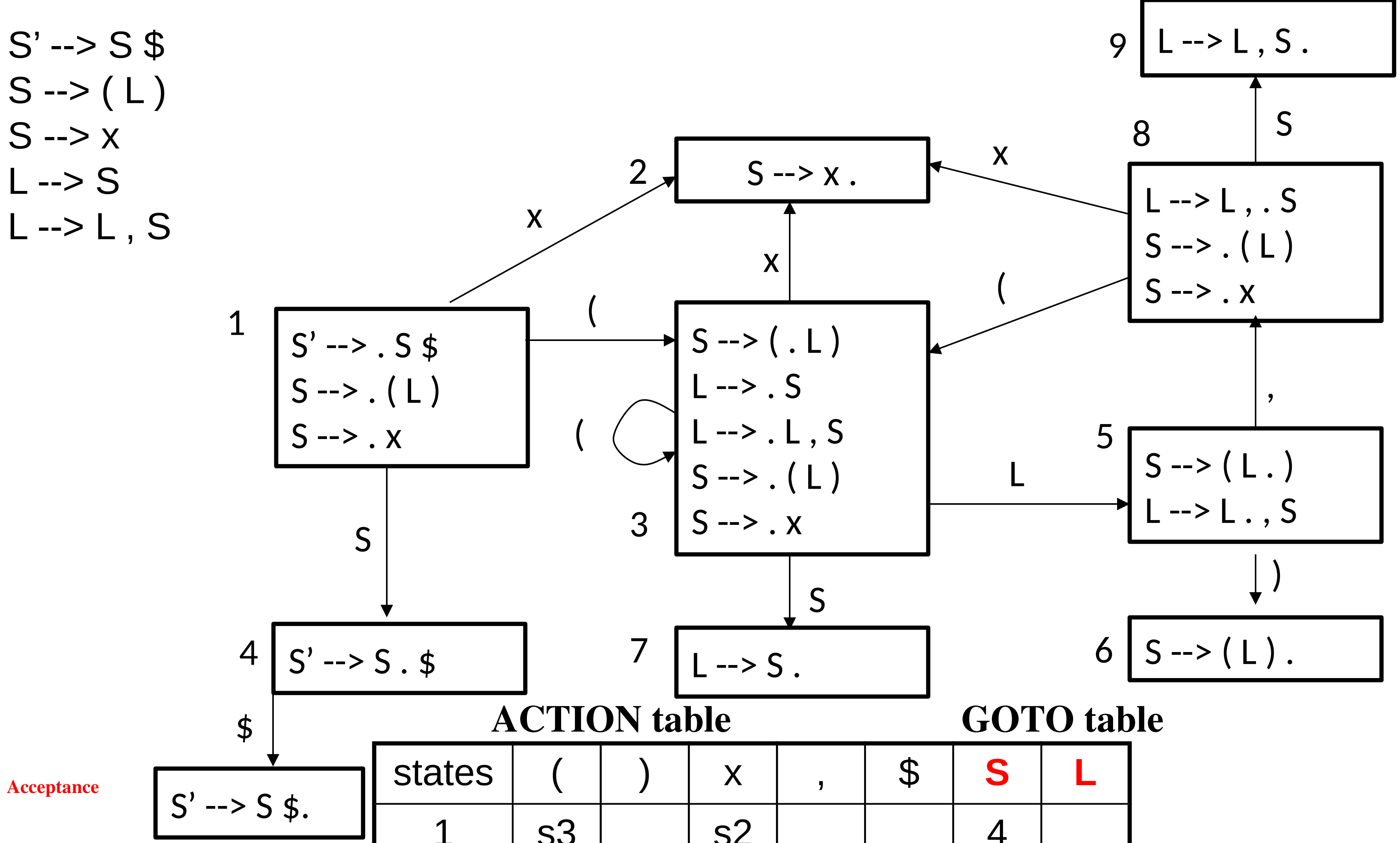
S’ --> S $
S --> (L)
S --> x
L --> S
L --> L , S



ACTION table GOTO table

states	(	)	x	,	$	S	L
1	s3		s2			4	
2		r2		r2	r2		
3	s3		s2			7	5
4							

S’ --> S $
S --> (L)
S --> x
L --> S
L --> L , S



	ACTION table					GOTO table	
states	(	)	x	,	$	S	L
1	s3		s2			4	
2		r2		r2	r2		
3	s3		s2			7	5
4					accept		

SLR parsing table for G4’

states	(	)	x	,	$	S	L
1	s3		s2			4	
2		r2		r2	r2		
3	s3		s2			7	5
4					accept		
5		s6		s8			
6		r1		r1	r1		
7		r3		r3			
8	s3		s2			9	
9		r4		r4			

r0 S’ --> S $
r1 S --> (L)
r2 S --> x
r3 L --> S
r4 L --> L , S

Follow(S)={$,), ,}
Follow(L)={), ,}

Example: Moves of LR parser on (x,x) input.

Stack	Input	Action
1	(x,x)$	shift
1(3	x,x)$	shift
1(3x2	,x)$	Reduce S□x
1(3S7	,x)$	Reduce L□S
1(3L5	,x)$	shift
1(3L5,8	x)$	shift
1(3L5,8x2	)$	Reduce S□x
1(3L5,8S9	)$	Reduce L□L,S
1(3L5	)$	shift
1(3L)6	$	Reduce S□(L)
1S4	$	accept

Task

- Task Given the following CFG grammar G with P:

$$
\begin{aligned}
S &\rightarrow aABe \\
A &\rightarrow Abc \mid b \\
B &\rightarrow d
\end{aligned}
$$

a) Construct the corresponding parsing table using SLR parsing algorithm.

b) Show the stack contents, the input and the action used during parsing for the input string w = abbcde

Exercise

- Task Given the following CFG grammars G1 & G2 with P:

G1:
E→E+T | T
T →T*F | F
F□(E) | id

G2:
S → L=R
S → R
L → *R
L → id
R → L

a) Construct the corresponding parsing tables for G1 & G2 using SLR parsing algorithm.

b) Show the stack contents, the input and the action used during parsing for the input string w = id+id*id using parsing table of G1 .