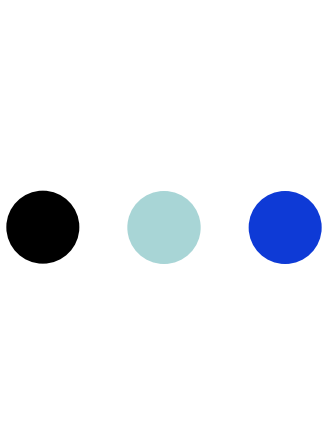


Parsing 2

LR parser construction.

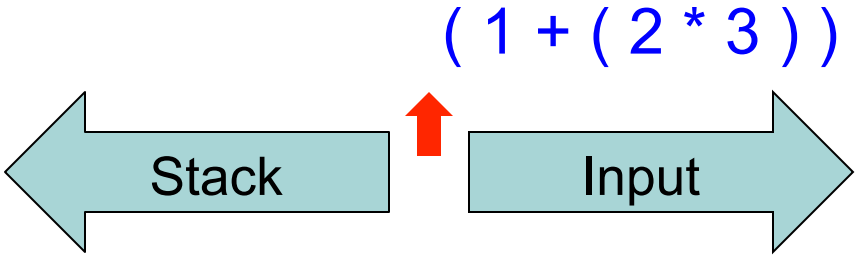
Slides by Rohit Chadha, Bill Harrison, Chris Hathhorn,
& some stolen wholesale from David Walker at Princeton.

So far…

- Previously: top-down, recursive decent parsing.
- Today: bottom-up, shift-reduce parsing.
  - LR(0).
  - Also mention: SLR, LR(1), LALR.

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



Actions:

- **Shift**: shift the next input symbol onto the stack.
- **Reduce**: replace a string of symbols on top of the stack with a corresponding non-terminal from the grammar.
- **Accept**: announce successful completion of parsing.
- **Error**: indicates a syntax error.

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (2 * 3))

Reduce!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (2 * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (2 * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (2 * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (2 * 3))

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * 3))

Reduce!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * 3))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * 3))

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * *Exp*))

Reduce!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * *Exp*))

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + (*Exp* * *Exp*))

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + *Exp*)

Reduce!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + *Exp*)

Shift!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(*Exp* + *Exp*)

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

Exp

Reduce!

Shift-reduce parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

Exp

Accept!

Shift-reduce parsing

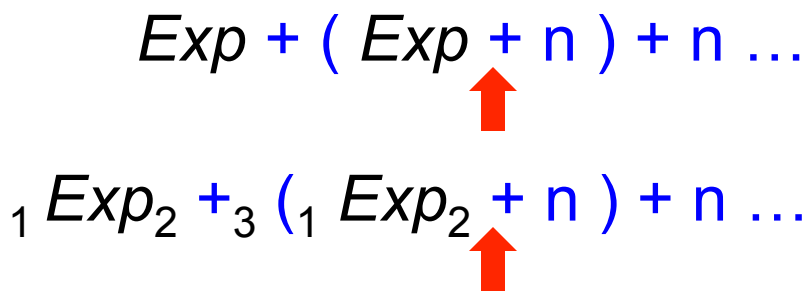
- Shift-reduce parsing is general technique for bottom-up parsing.
- Many practical parsing algorithms are based on this technique:
  - LR, SLR, GLR, LALR, …
- The basic problem: deciding when to shift and when to reduce.
  - Solution: use DFAs!

The magic: constructing an LR parser table

Exp + (*Exp* + n) + n …

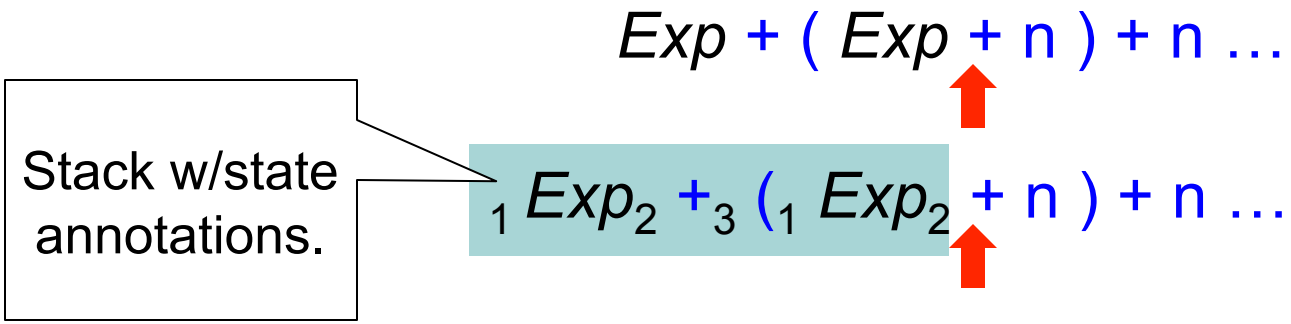
- At every point in the parse, the LR parser table tells us what to do next.
  - shift, reduce, error or accept
- To do so, the LR parser keeps track of the parse "state" on the stack (i.e., the state of the finite automaton).

The magic: constructing an LR parser table



- At every point in the parse, the LR parser table tells us what to do next.
  - shift, reduce, error or accept
- To do so, the LR parser keeps track of the parse "state" on the stack (i.e., the state of the finite automaton).

The magic: constructing an LR parser table



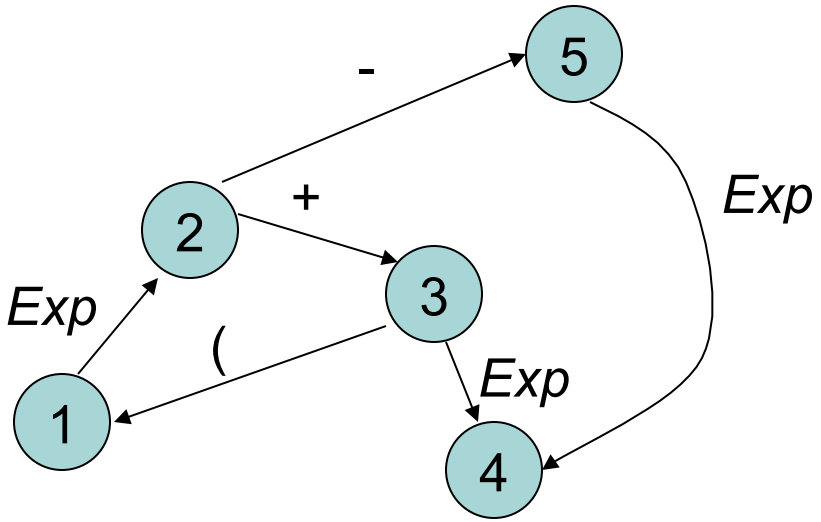
- At every point in the parse, the LR parser table tells us what to do next.
  - shift, reduce, error or accept
- To do so, the LR parser keeps track of the parse "state" on the stack (i.e., the state of the finite automaton).

The magic: constructing an LR parser table

This state & input tell us what to do next.

$_1 Exp_2 +_3 (_1 Exp_2 + n) + n \ldots$

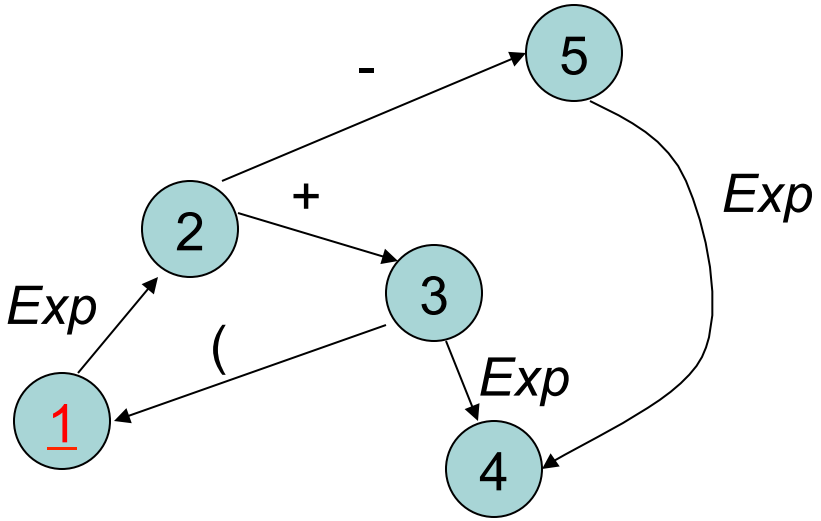
finite automaton; terminals and non terminals label edges



The magic: constructing an LR parser table

$_{1}\,Exp_{2}\ +_{3}\ (_{1}\ Exp_{2}\ +\ n\ )\ +\ n\ \ldots$

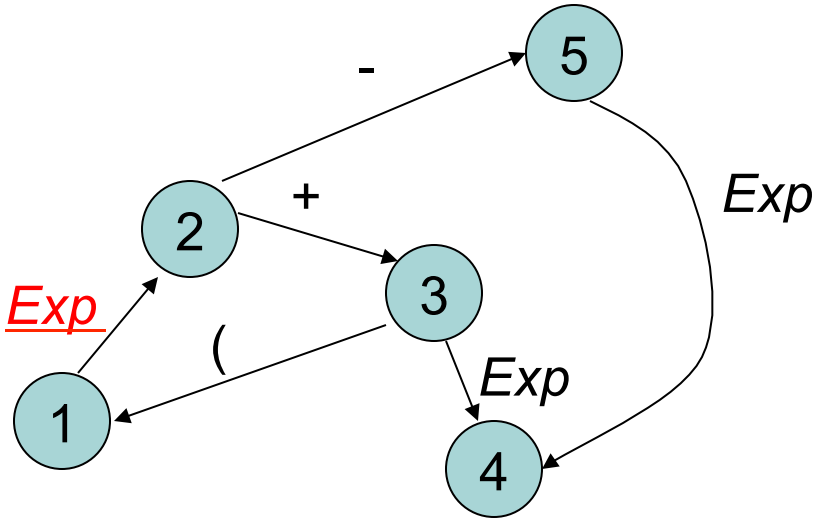
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1 Exp_2 +_3 (_1 Exp_2 + n) + n \ldots$

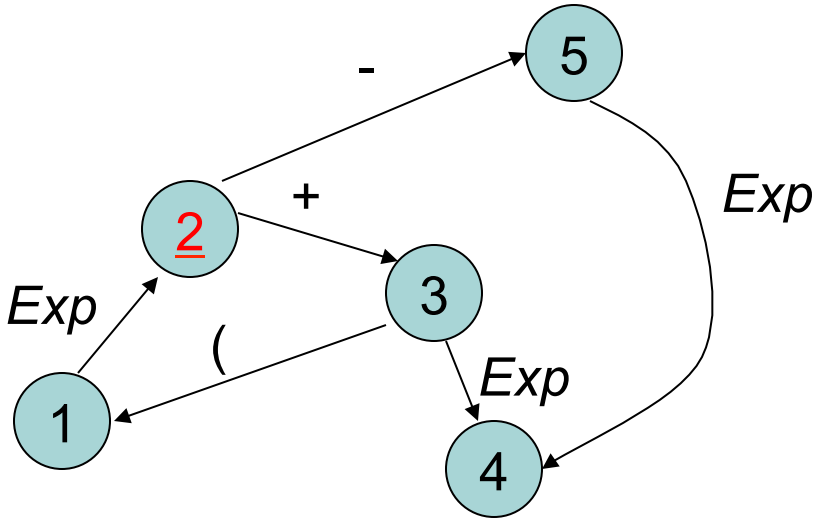
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1 Exp_{\underline{2}} +_3 (_1 Exp_2 + n) + n \ldots$

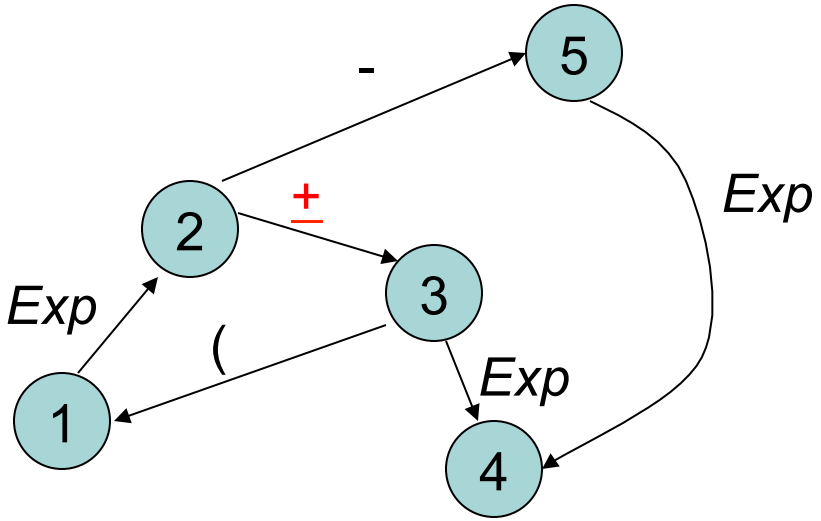
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1\,Exp_2\ \pm_3\ (_1\,Exp_2\ +\ n\ )\ +\ n\ \ldots$

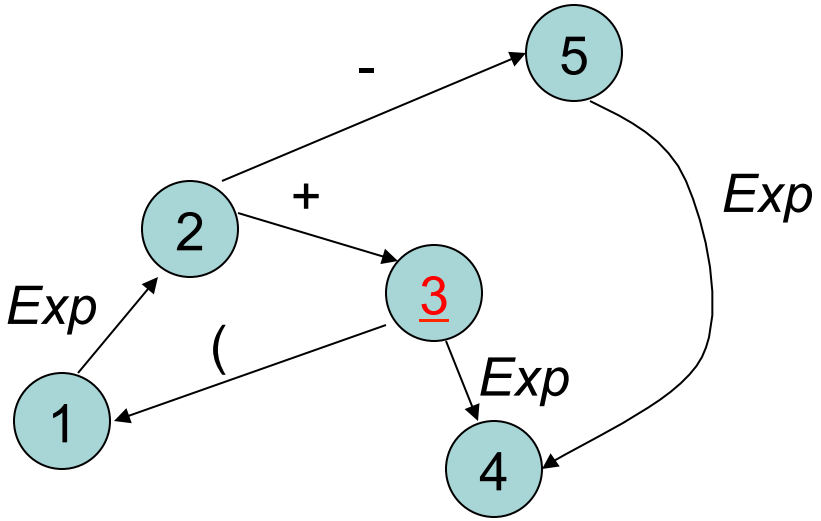
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1 Exp_2 +_{\underline{3}} (_1 Exp_2 + n) + n \ldots$

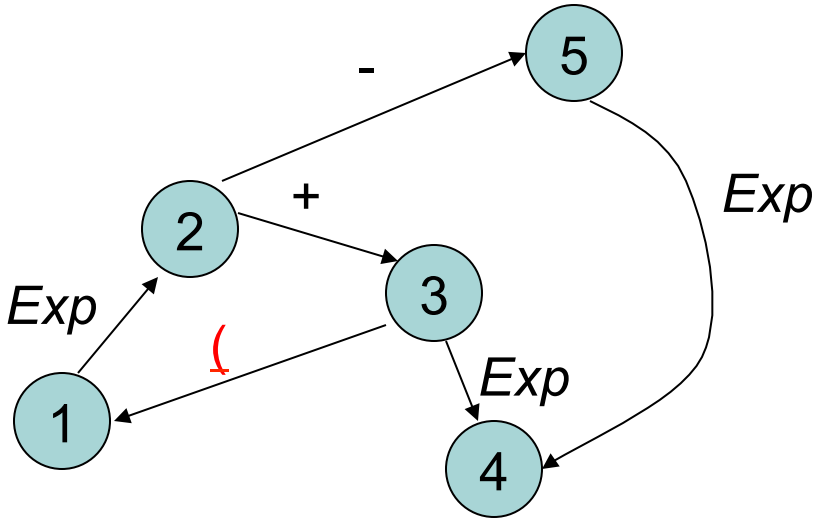
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1\,Exp_2\ +_3\ (_1\ Exp_2\ +\ n\ )\ +\ n\ \ldots$

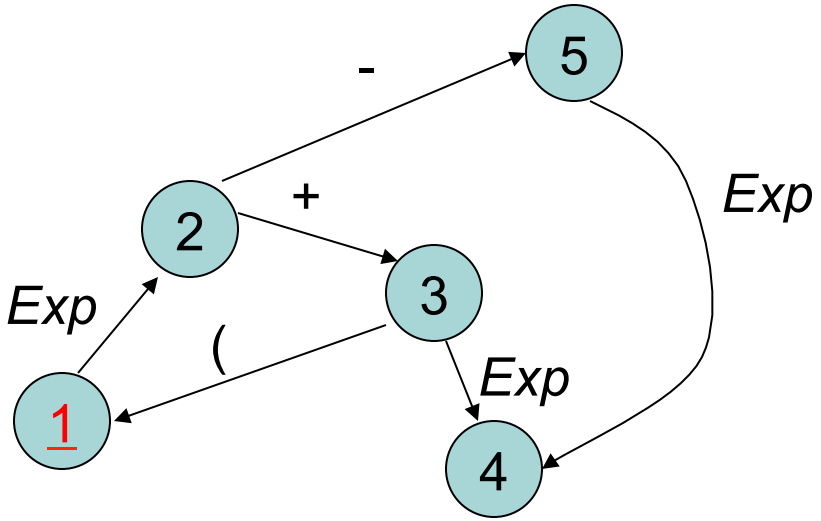
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1 Exp_2 +_3 (_1 Exp_2 + n) + n \ldots$

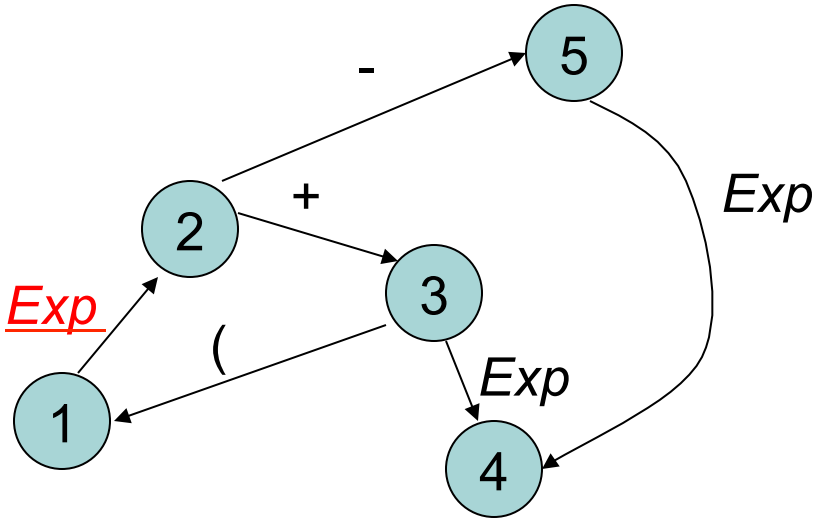
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1 Exp_2 +_3 (_1 \underline{Exp}_2 + n) + n \ldots$

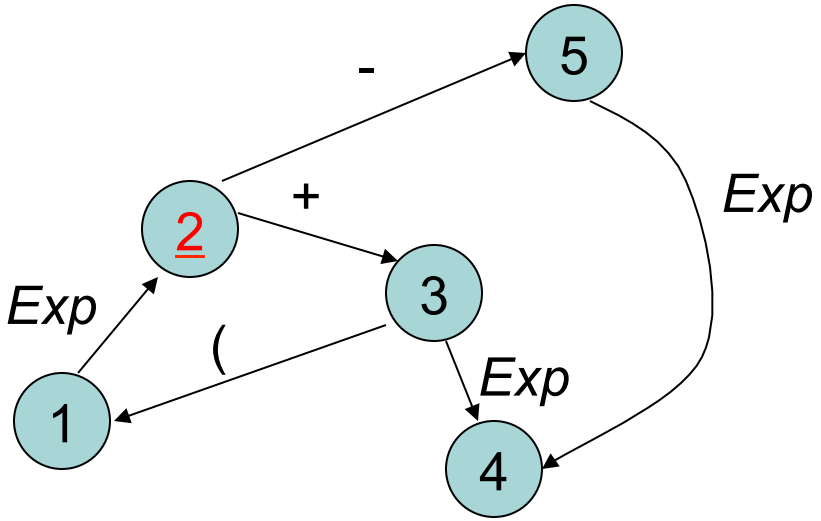
finite automaton;
terminals and
non terminals
label edges



The magic: constructing an LR parser table

$_1\,Exp_2\ +_3\ (_1\,Exp_2\ +\ n\ )\ +\ n\ \ldots$

finite automaton;
terminals and
non terminals
label edges



The parse table

At every point in the parse, the LR parser table tells us what to do next according to the automaton state at the top of the stack:

- shift, reduce, error or accept.

states	Terminal seen next ID, NUM, := ...
1	
2	s*n* = shift & goto state *n*
3	r*k* = reduce by rule *k*
...	a = accept
n	= error

The parse table

Reducing by rule k is broken into two steps:

- Current stack:

  $A_8\ B_3\ C_2\ \ .....{}_7\ RHS_{12}$

- Rewrite the stack according to X → RHS:

  $A_8\ B_3\ C_2\ \ .....{}_7\ X$

- Figure out new state based on state on top (i.e.: goto 13 from 7):

  $A_8\ B_3\ C_2\ \ .....{}_7\ X_{13}$

states	Terminal seen next ID, NUM, := ...	Non-terminals X,Y,Z ...
1		
2	s*n* = shift & goto state *n*	g*n* = goto state *n*
3	r*k* = reduce by rule *k*	
...	a = accept	
n	= error	

The parse table

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

Terminals

Non-terminals

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

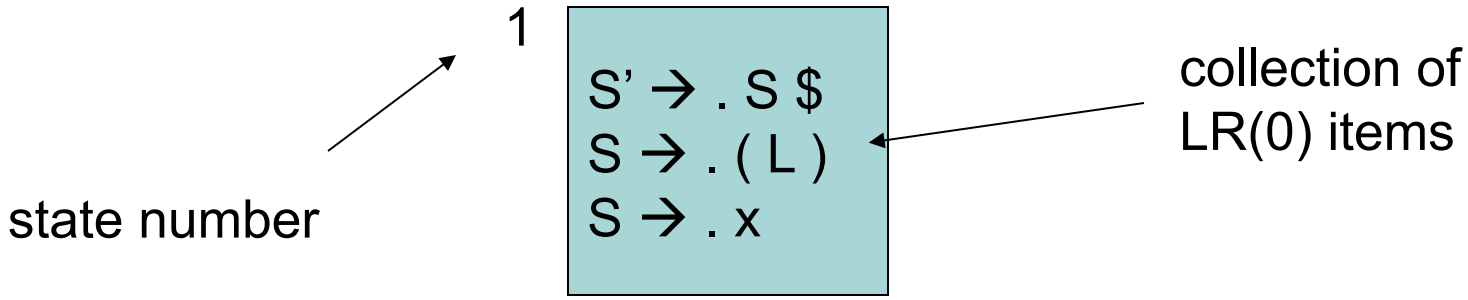
States go here

action

goto

LR(0) parsing

- Each state in the automaton represents a collection of LR(0) items:
  - an item is a rule from the grammar combined with "." to indicate where the parser currently is in the input
    - e.g.: S' → . S $ indicates that the parser is just beginning to parse this rule and it expects to be able to parse S then $ next
- A whole automaton state looks like this:



LR(0) parsing

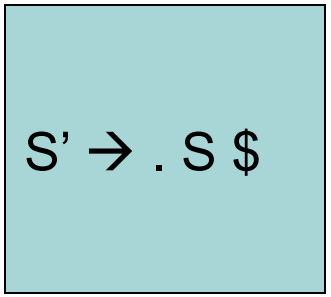
To construct states, we begin with a particular LR(0) item and construct its closure:

- 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.

Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S

1



LR(0) parsing

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.

Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S

1

S' → . S $
S → . (L)

LR(0) parsing

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.

Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S

1

S' → . S $
S → . (L)
S → . x

Full Closure

LR(0) parsing

To construct an LR(0) automaton:

- Start with start rule & compute initial state with closure.
- Pick one of the items from the state and move “.” to the right one symbol (as if you have just parsed the symbol)
  - this creates a new item ...
  - ... and a new state when you compute the closure of the new item
  - mark the edge between the two states with:
    - a terminal T, if you moved “.” over T
    - a non-terminal X, if you moved “.” over X
- Continue until there are no further ways to move “.” across items and generate new states or new edges in the automaton.

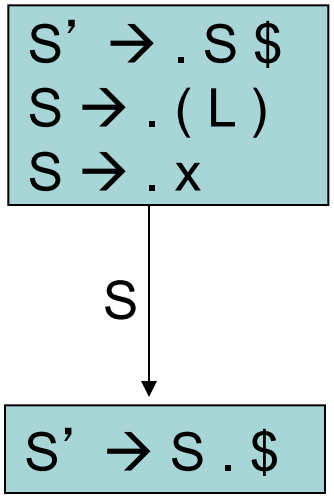
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S

S' → . S $
S → . (L)
S → . x

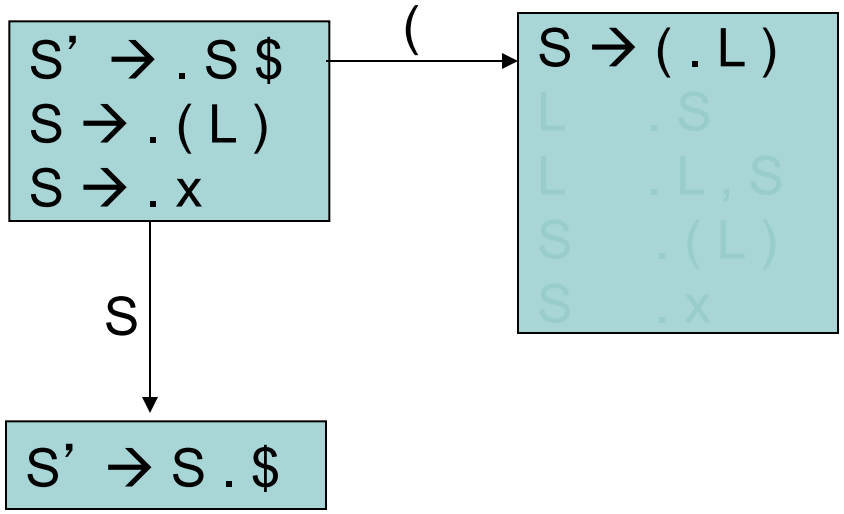
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



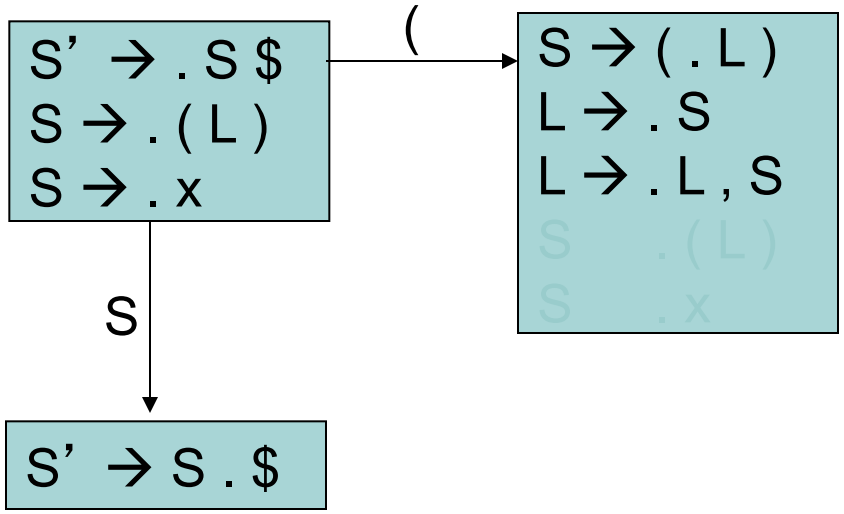
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



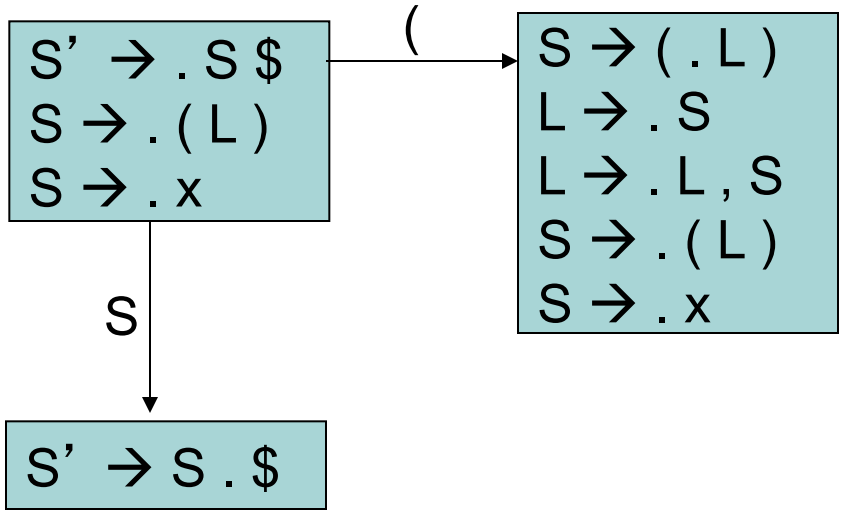
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



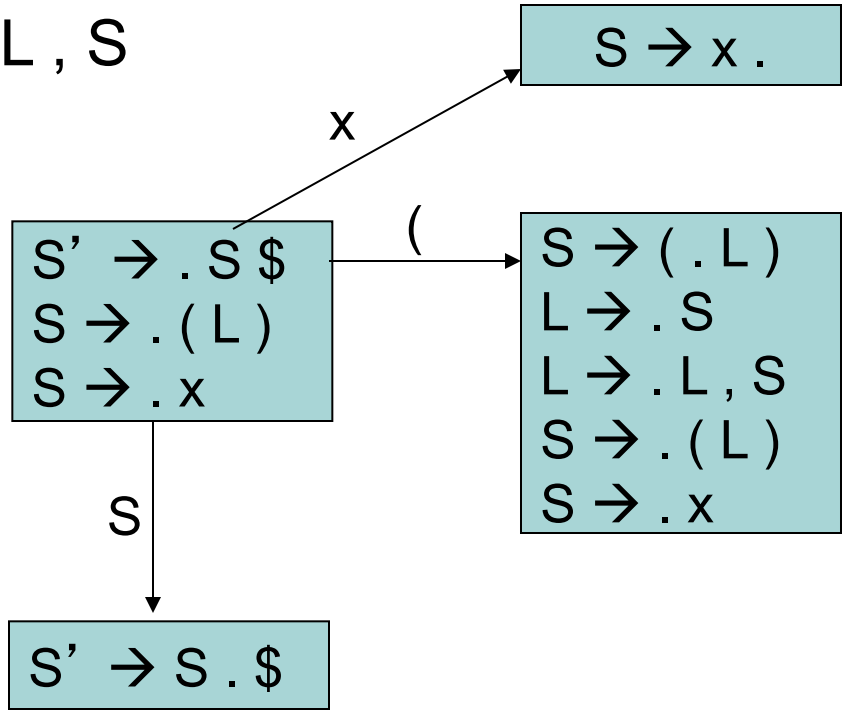
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



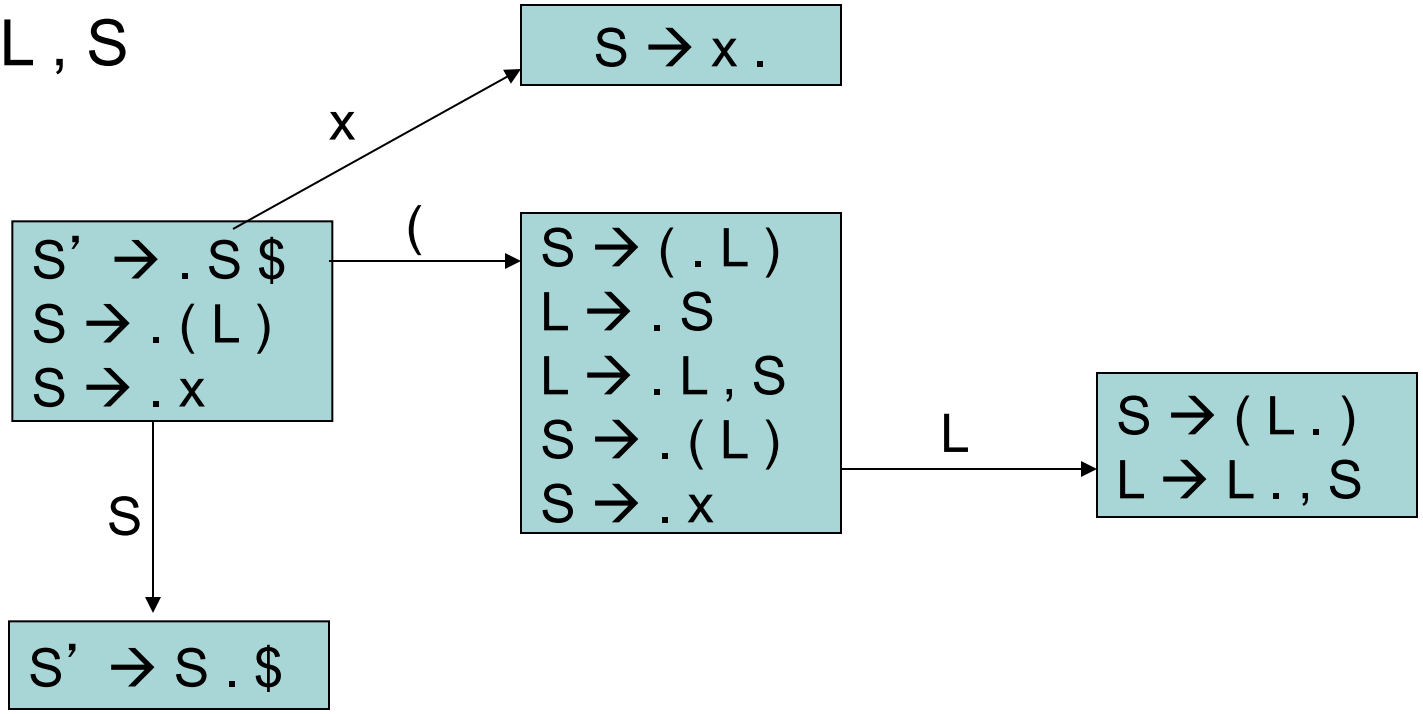
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



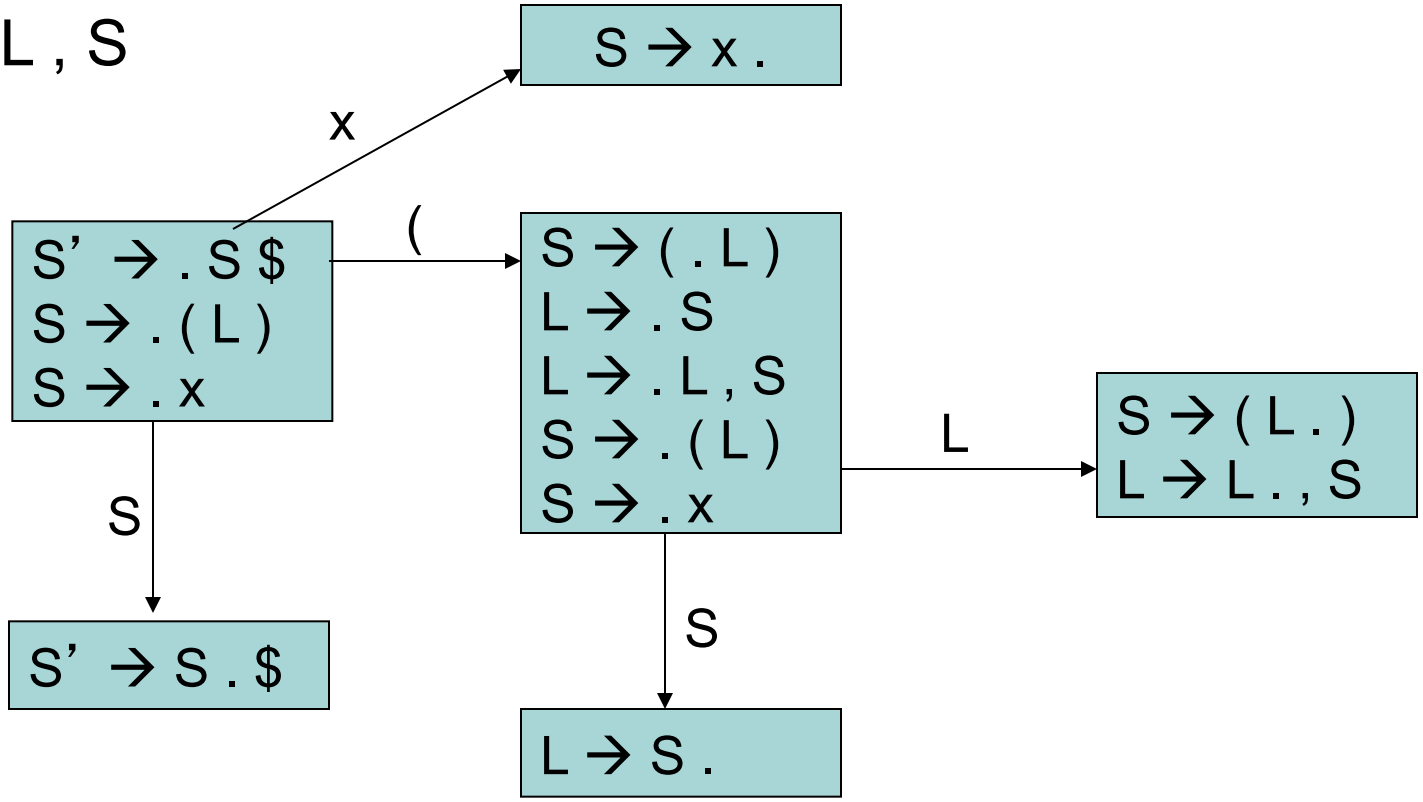
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



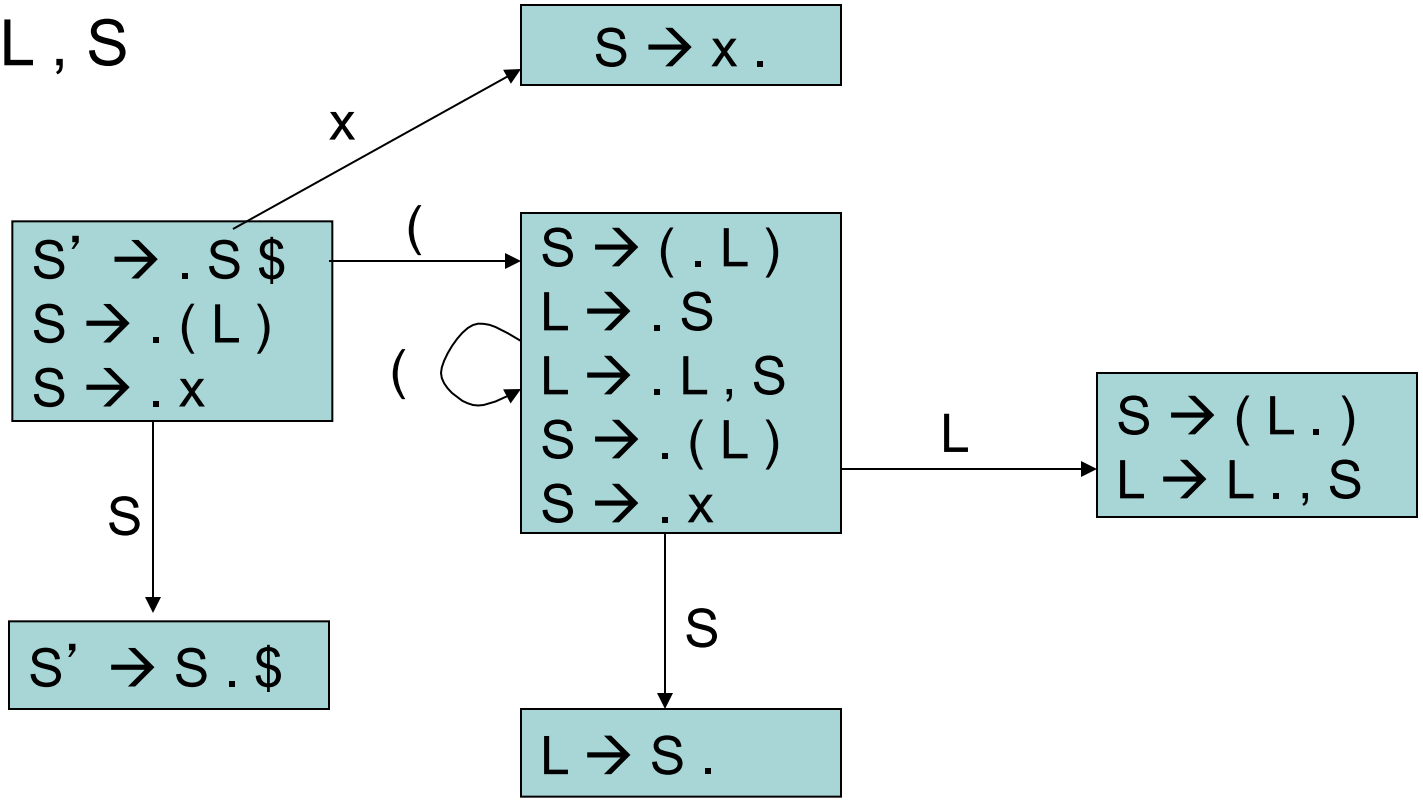
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



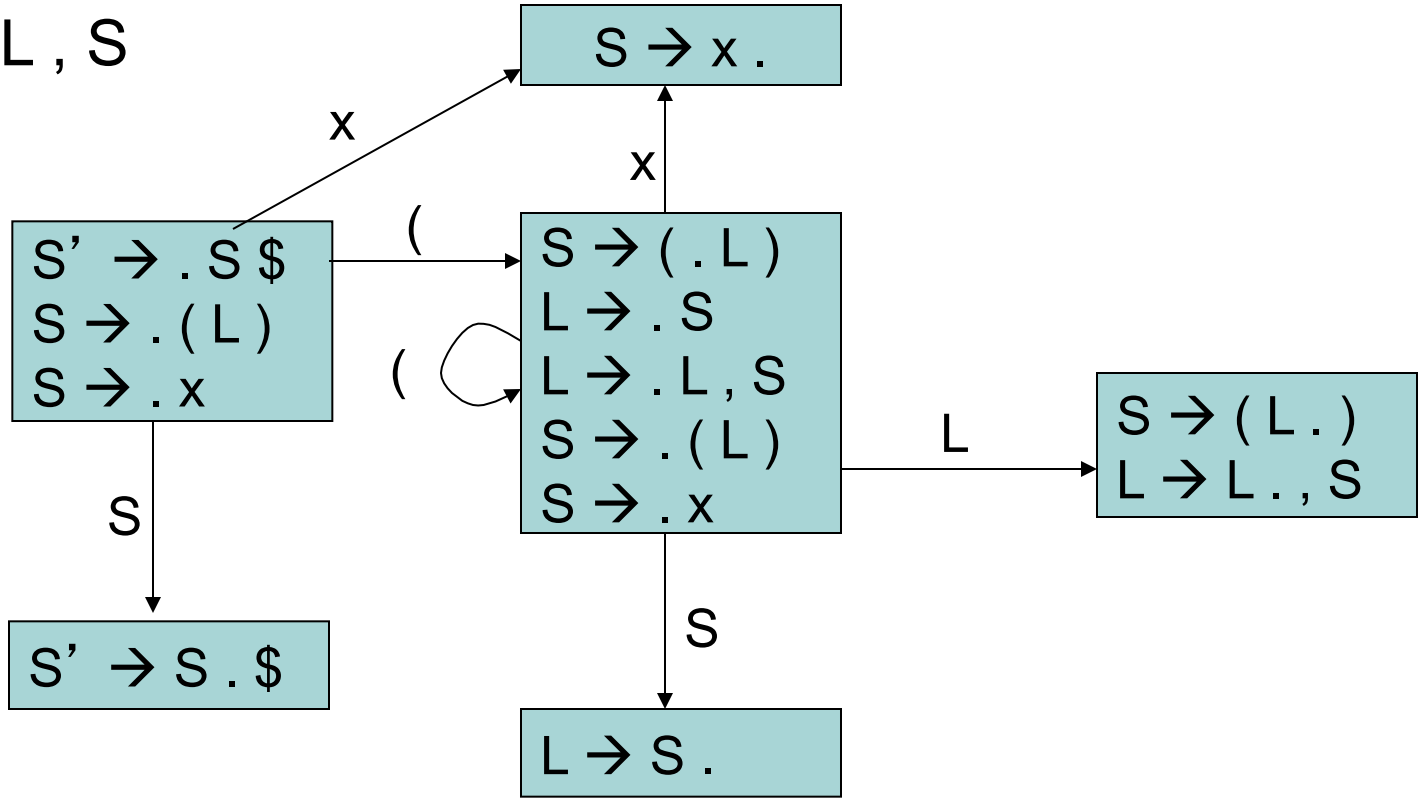
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



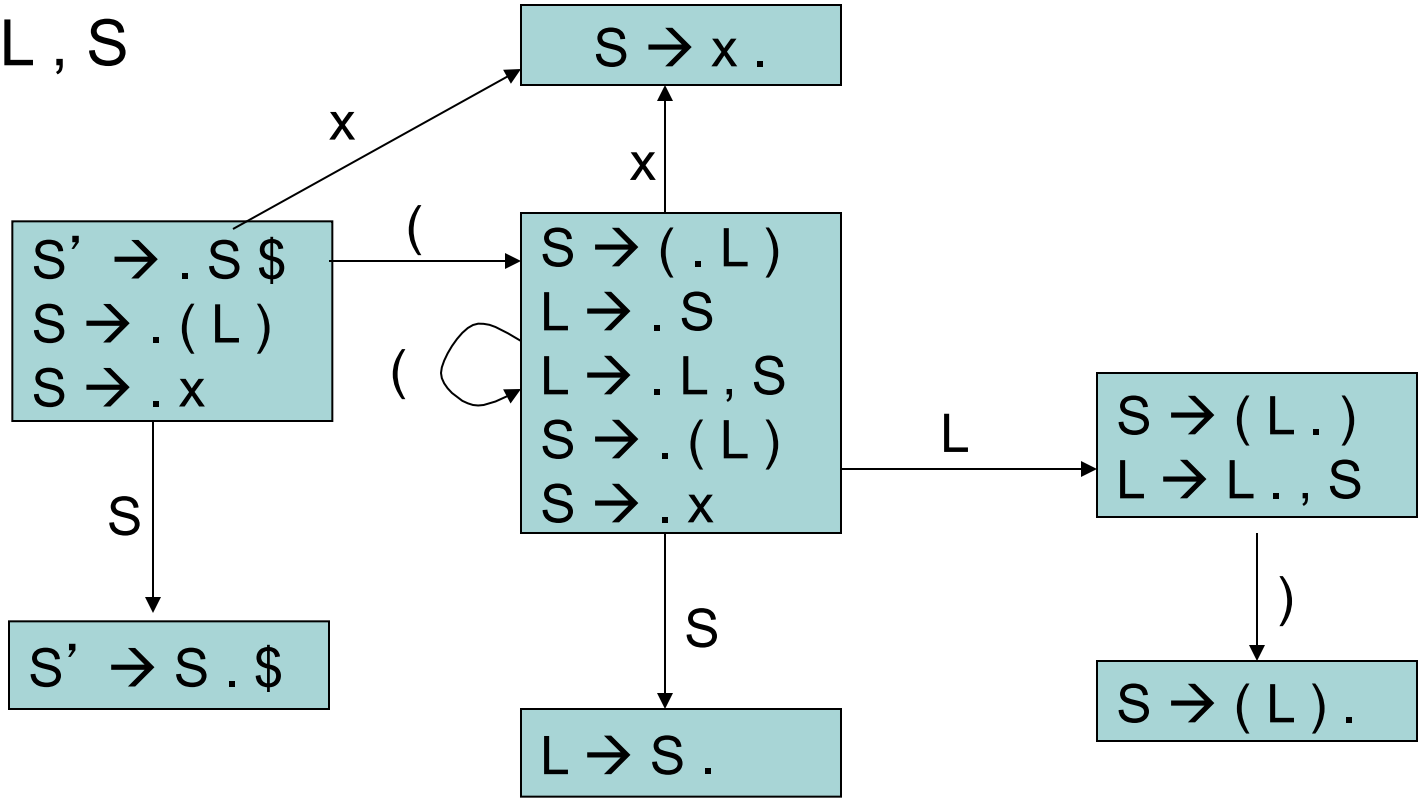
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



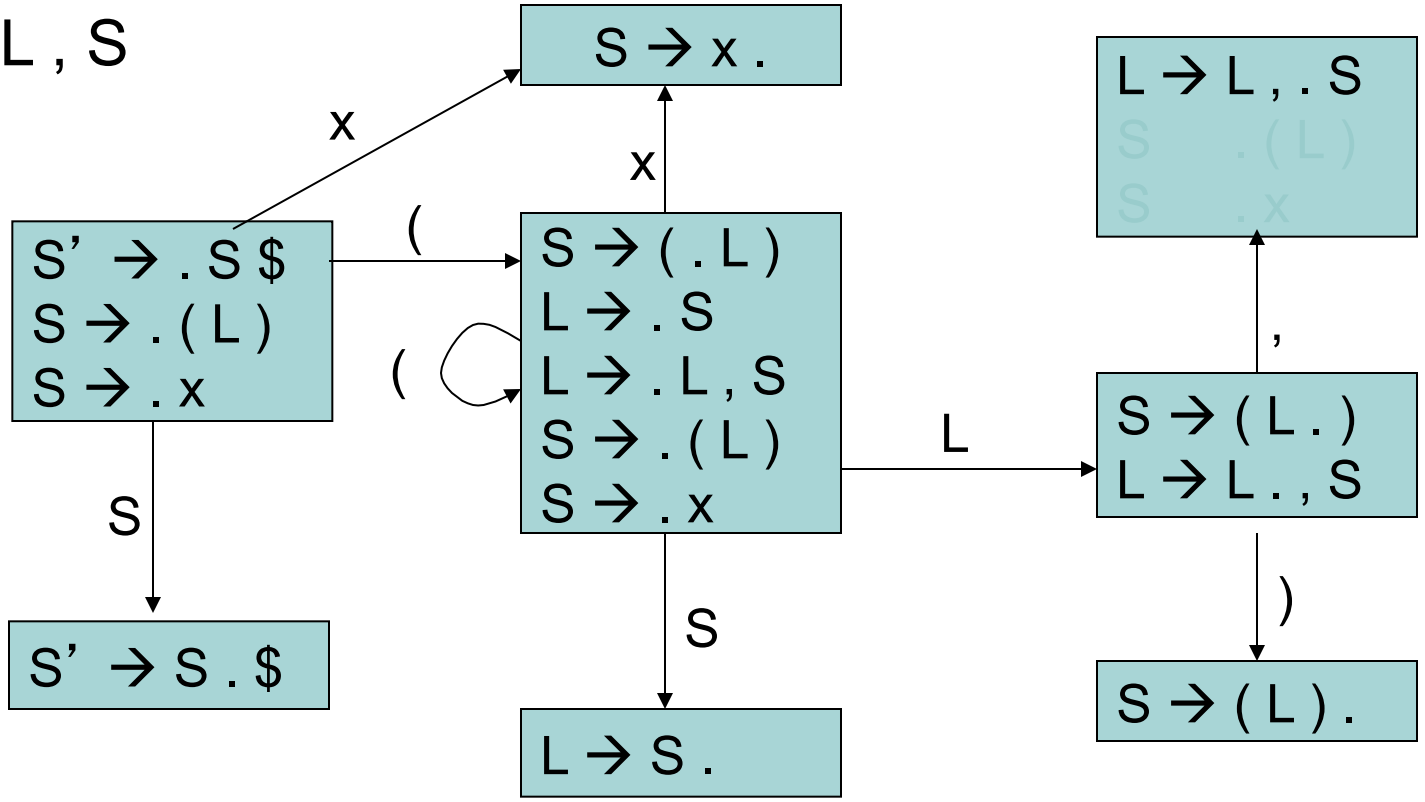
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



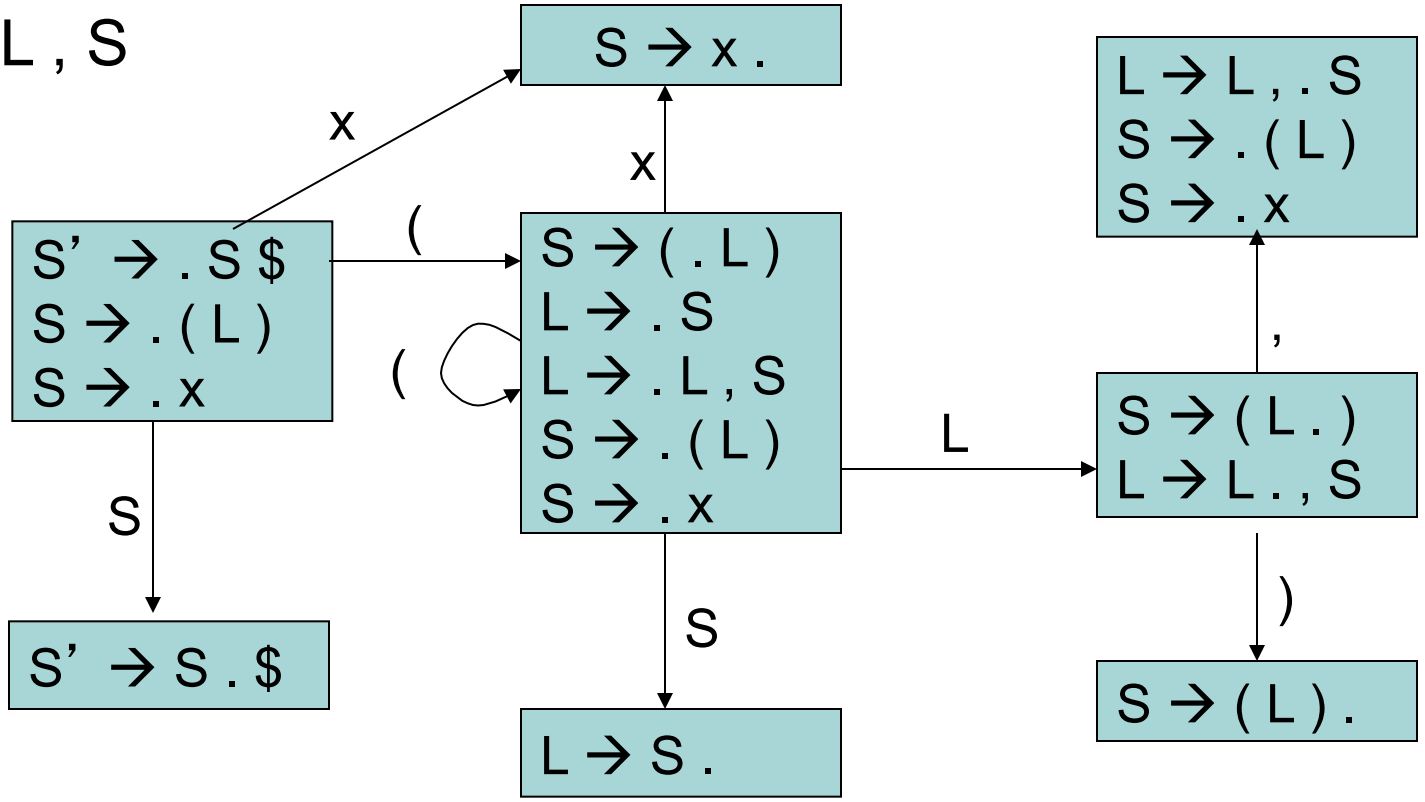
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



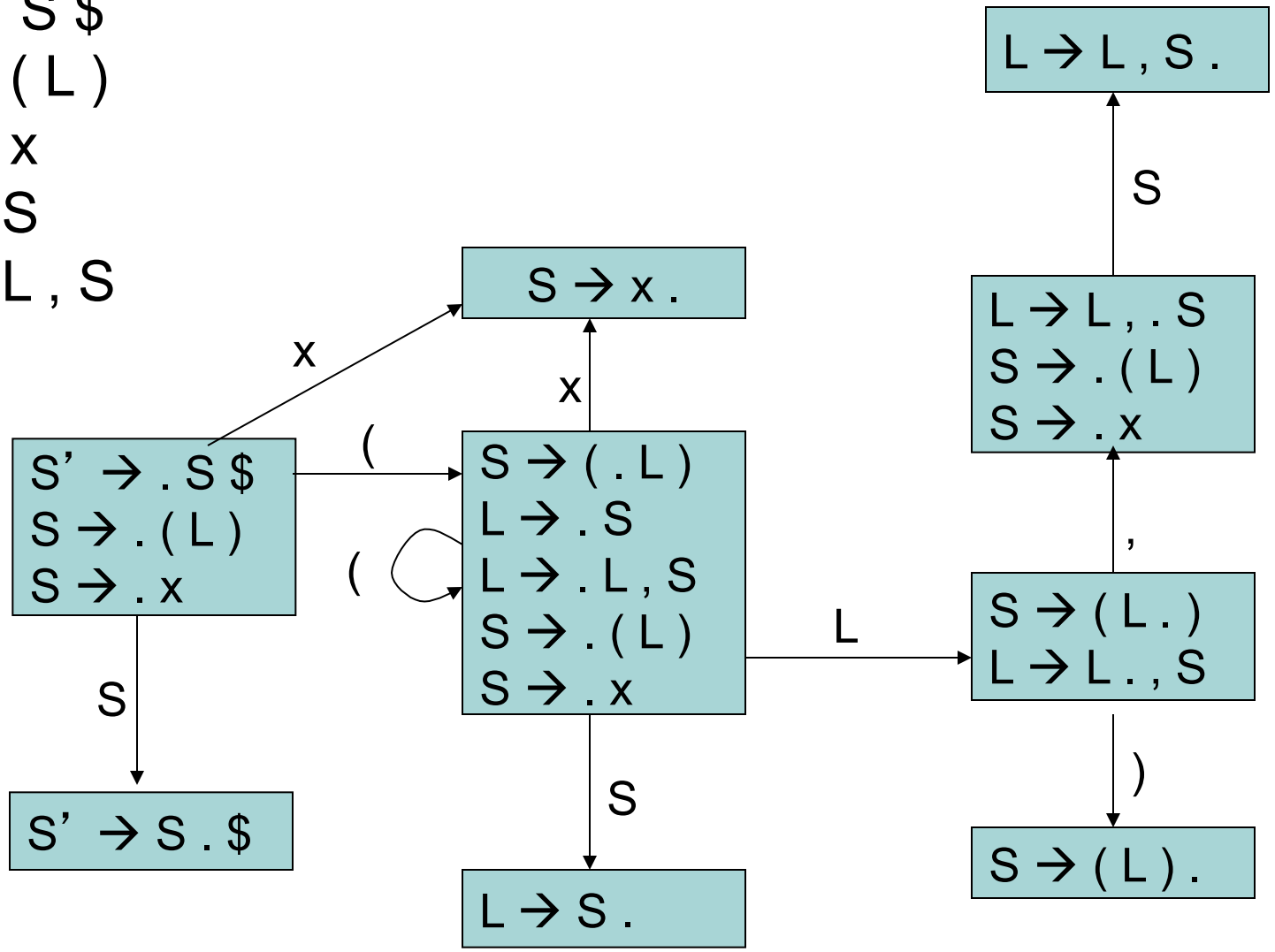
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



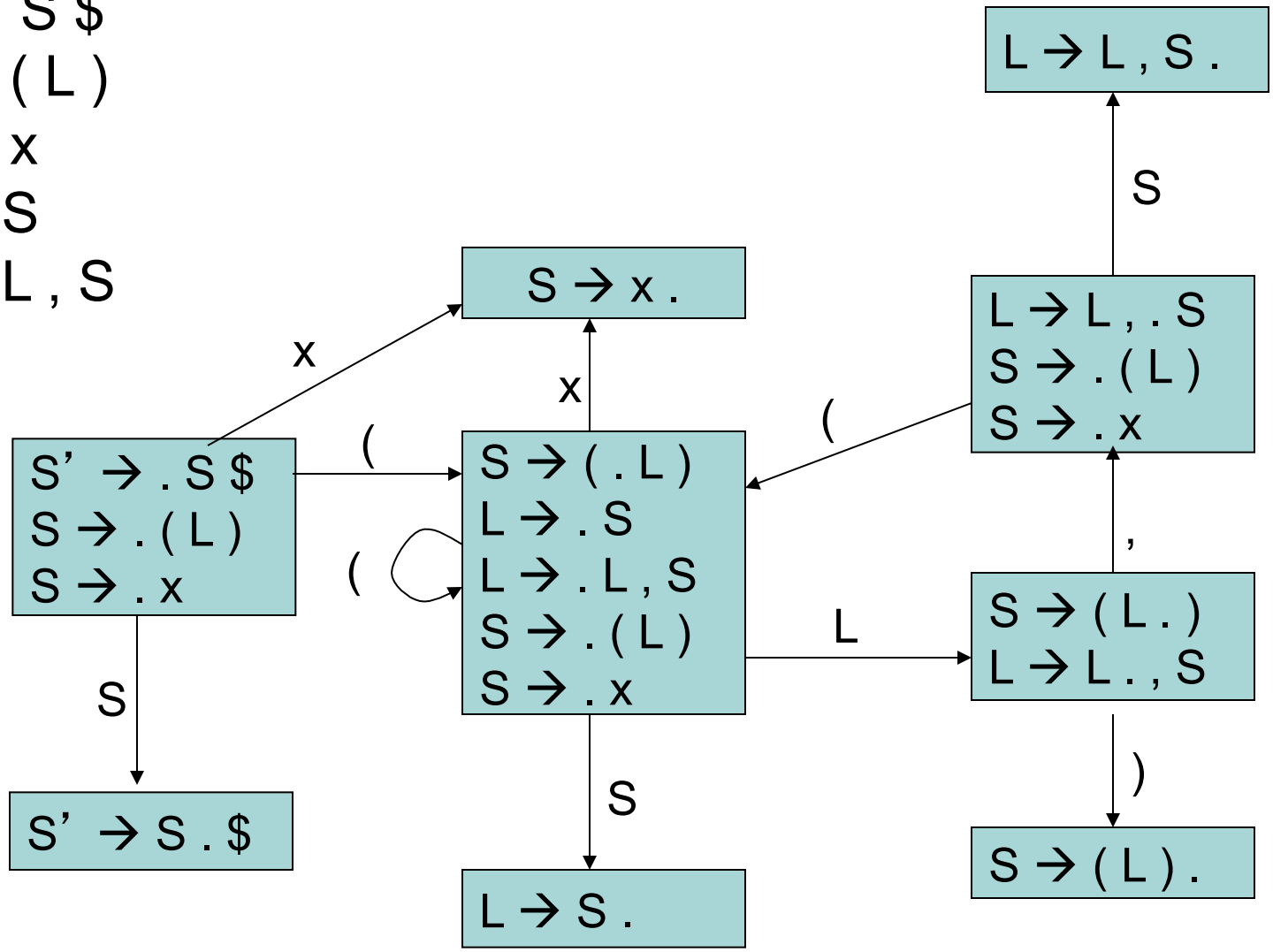
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



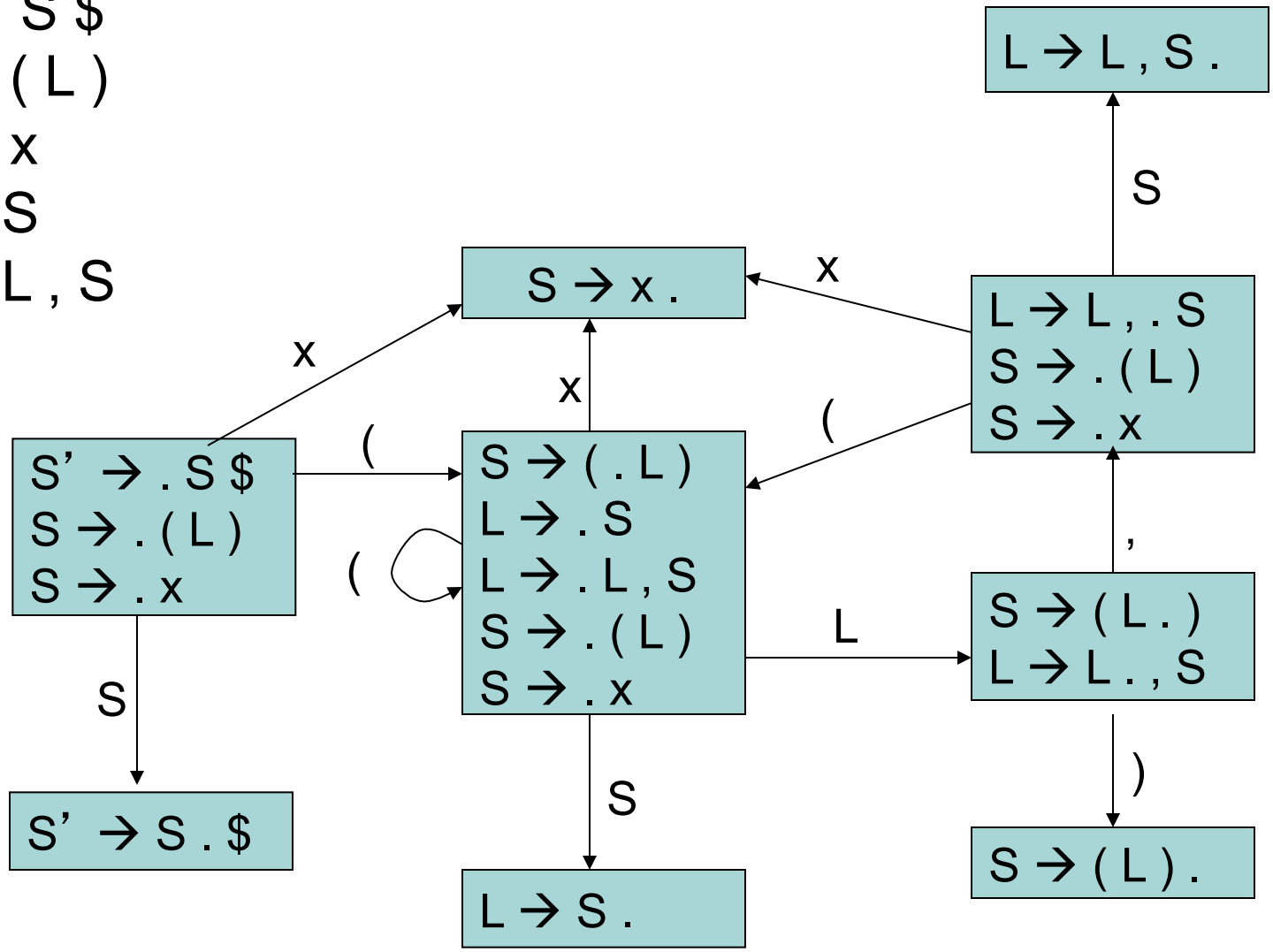
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



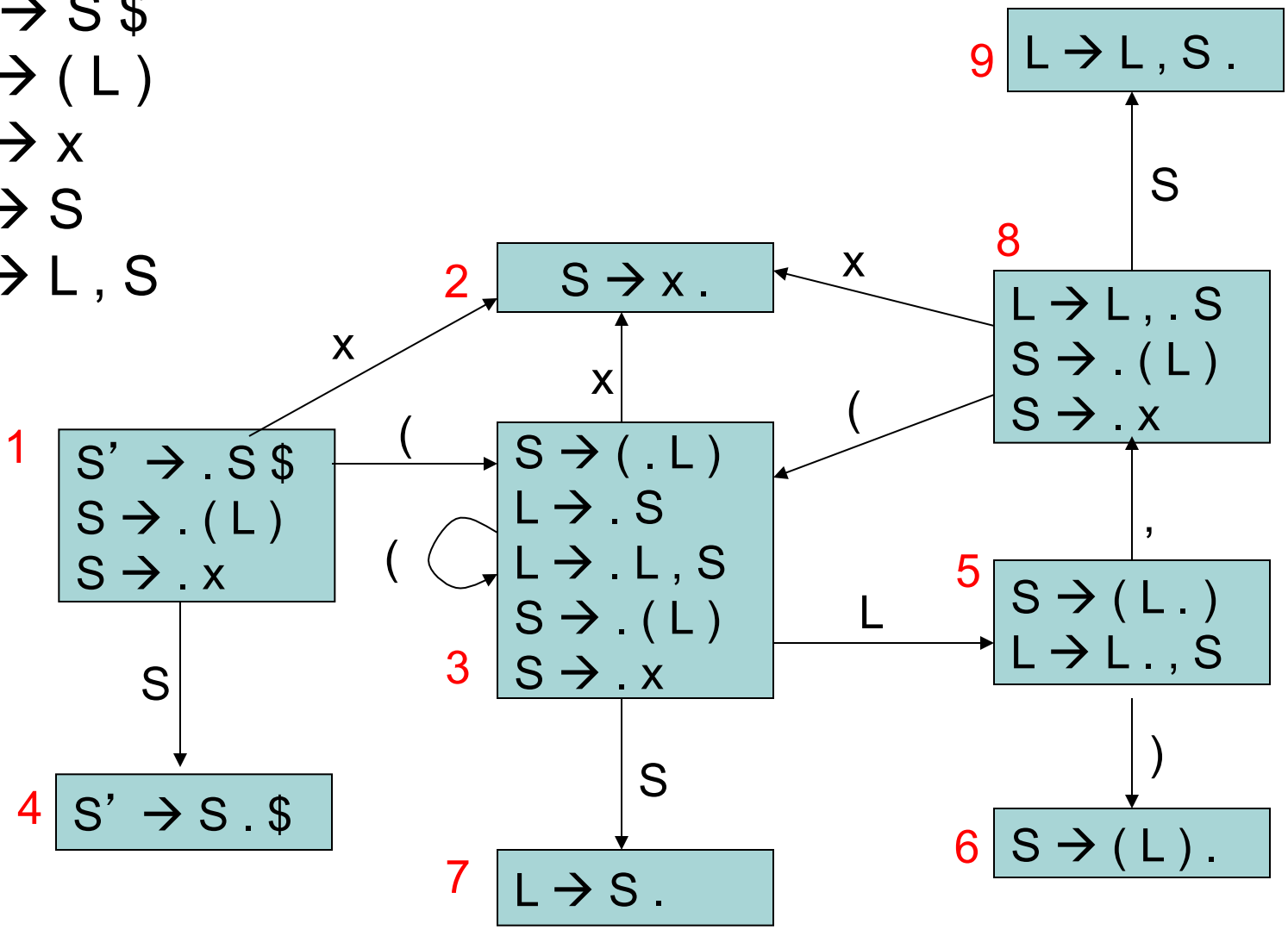
Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S



Grammar:

S' → S $
S → (L)
S → x
L → S
L → L , S

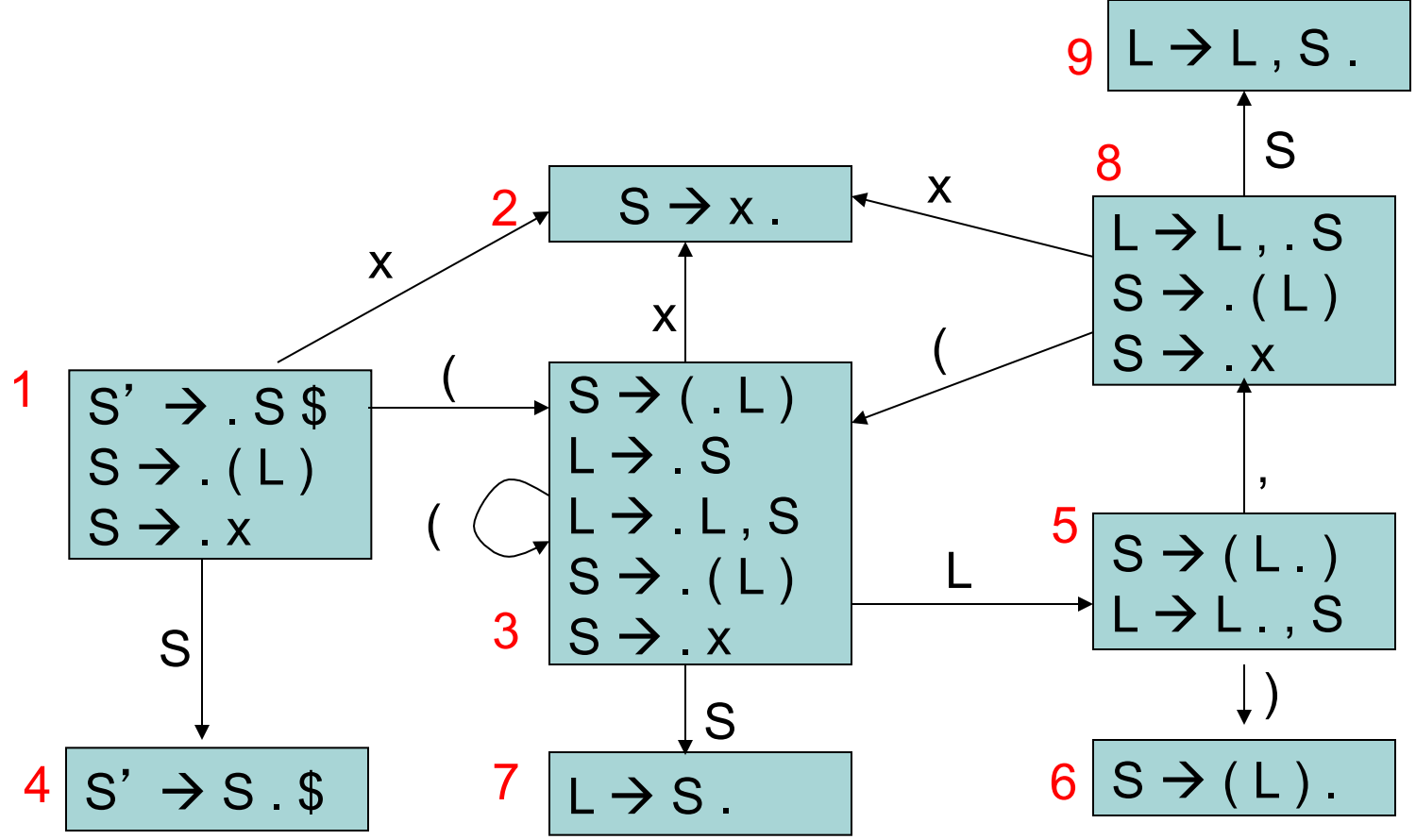


Finally, assign numbers to states.

Computing the parse table

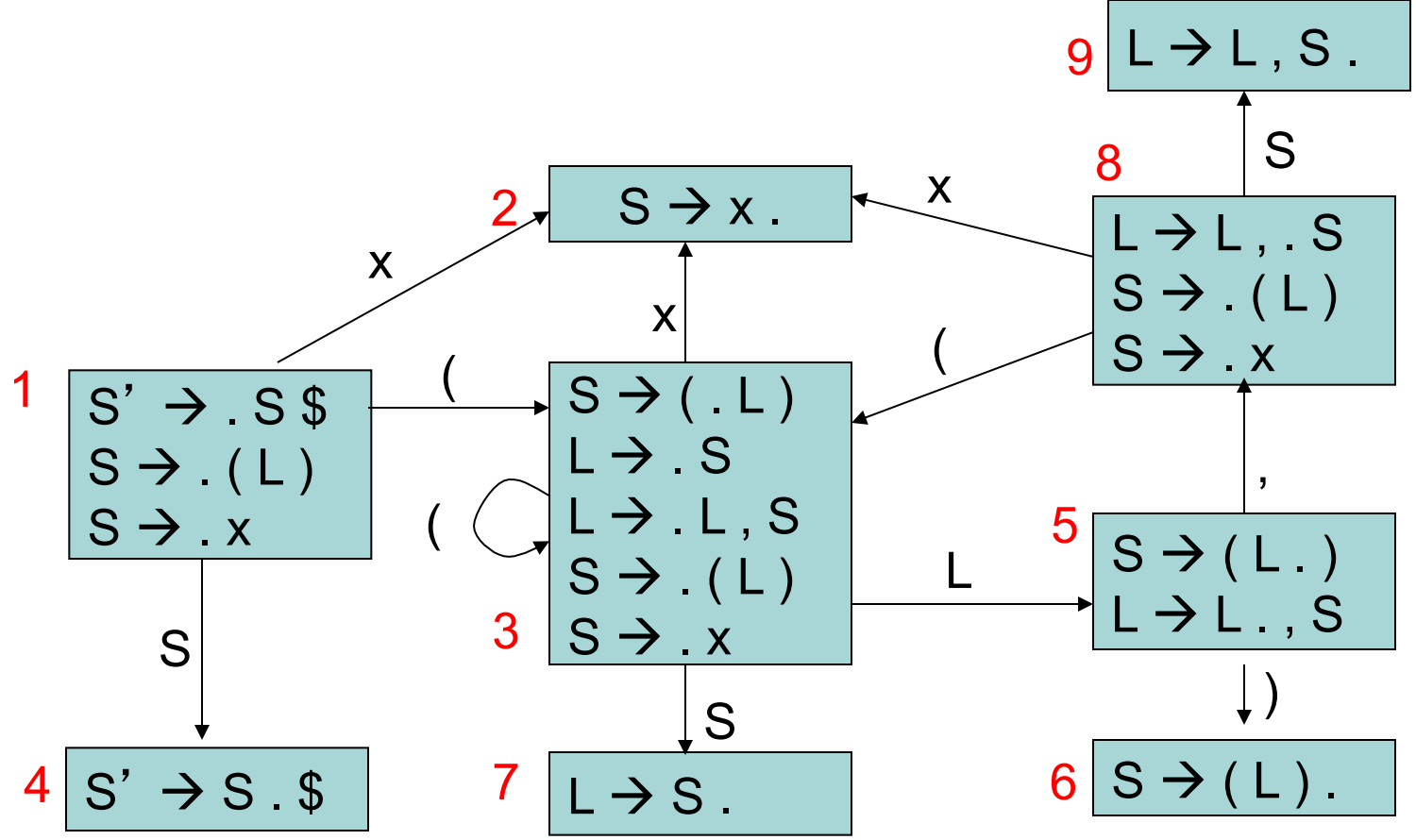
- If state *i* contains *X* → s . $, then action[*i*,$] = a.
- If state *i* contains rule *k*: *X* → s ., then table[*i*,*t*] = r*k*, for all terminals *t*.
- If there's a transition from *i* to *j* marked with *t*, then action[*i*,*t*] = s*j*.
- If there's a transition from *i* to *j* marked with *X*, then goto[*i*,*X*] = g*j*.

states	Terminal seen next ID, NUM, := ...	Non-terminals X,Y,Z ...
1		
2	s*n* = shift & goto state *n*	g*n* = goto state *n*
3	r*k* = reduce by rule *k*	
...	a = accept	
n	= error	



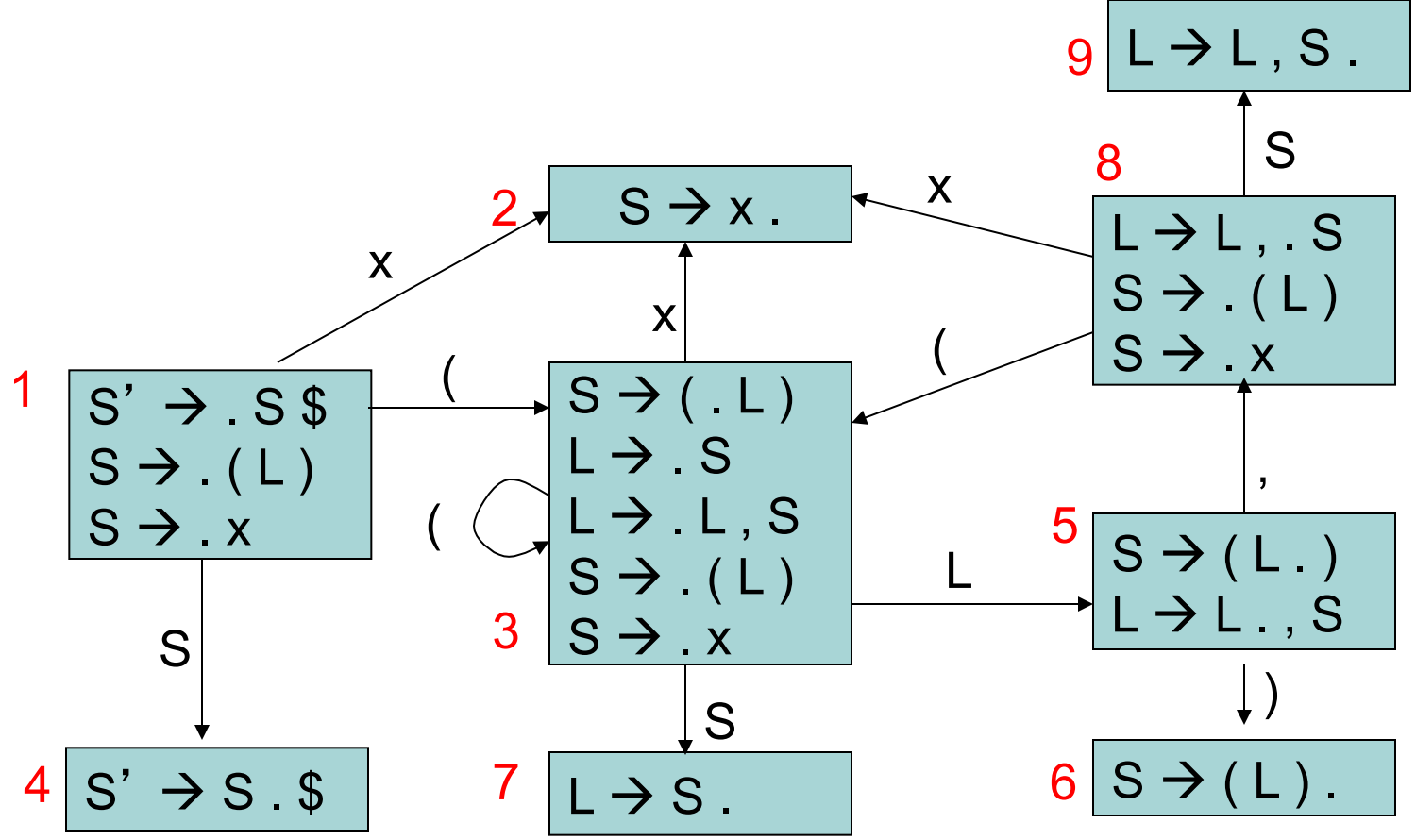
0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

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



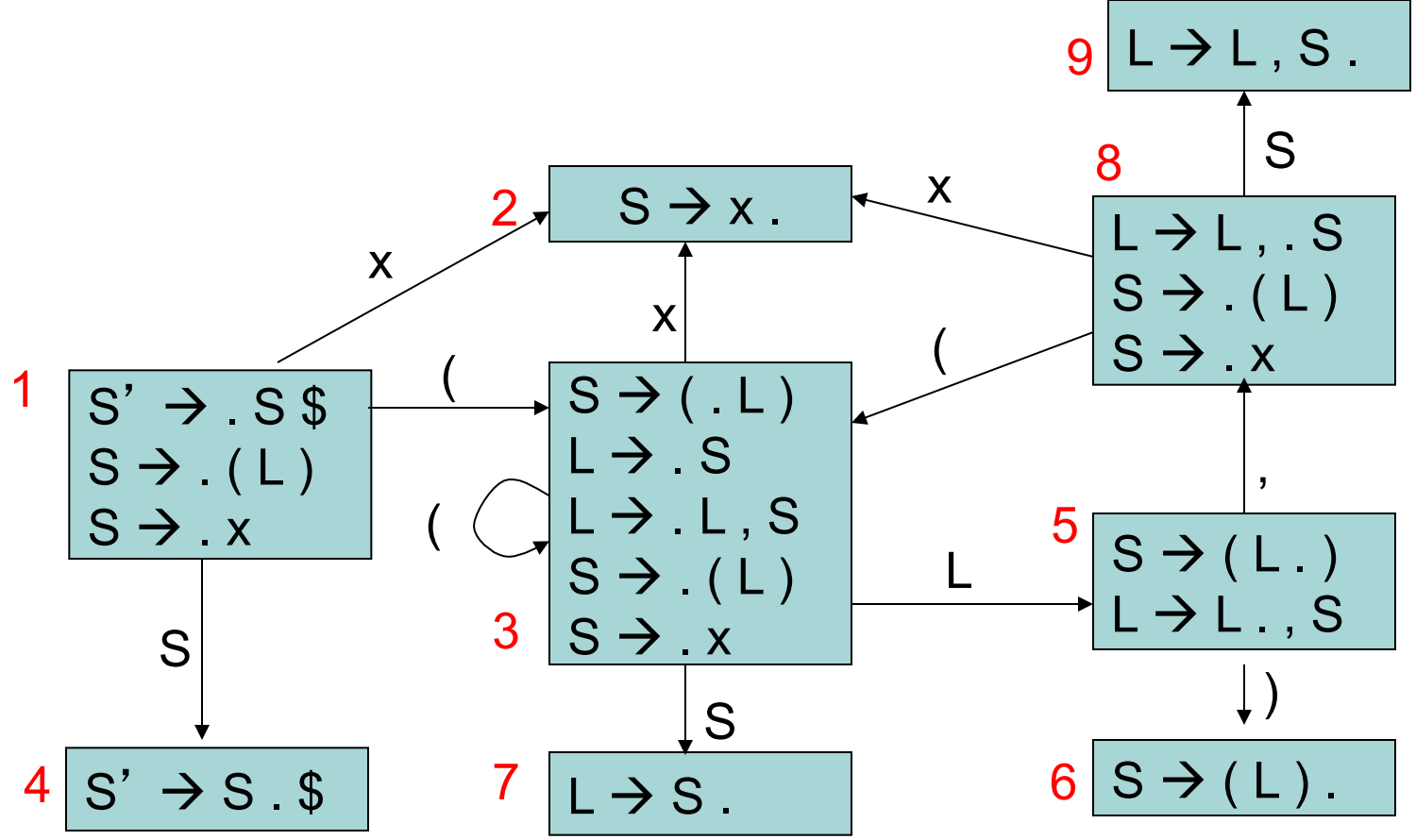
0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

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



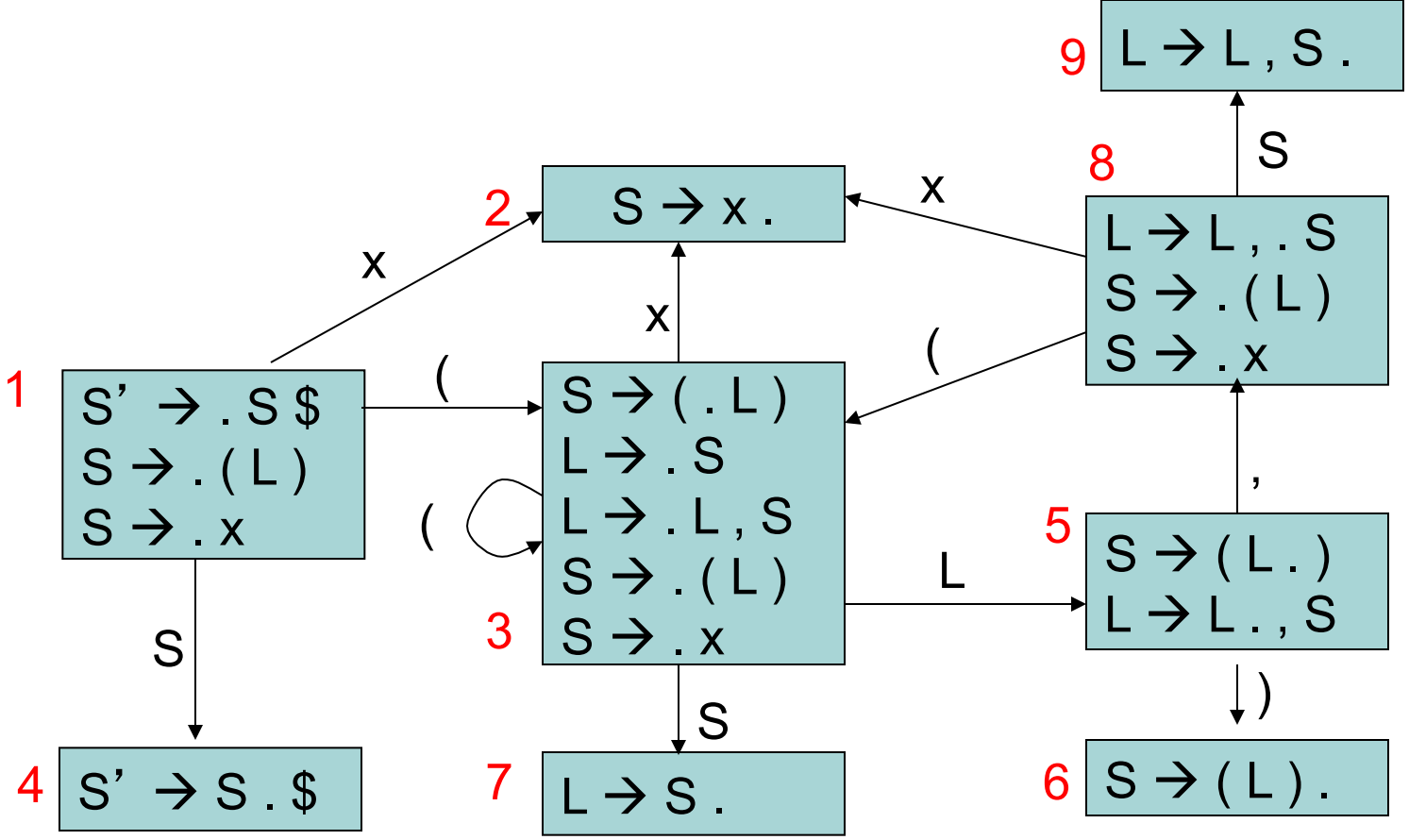
0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

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



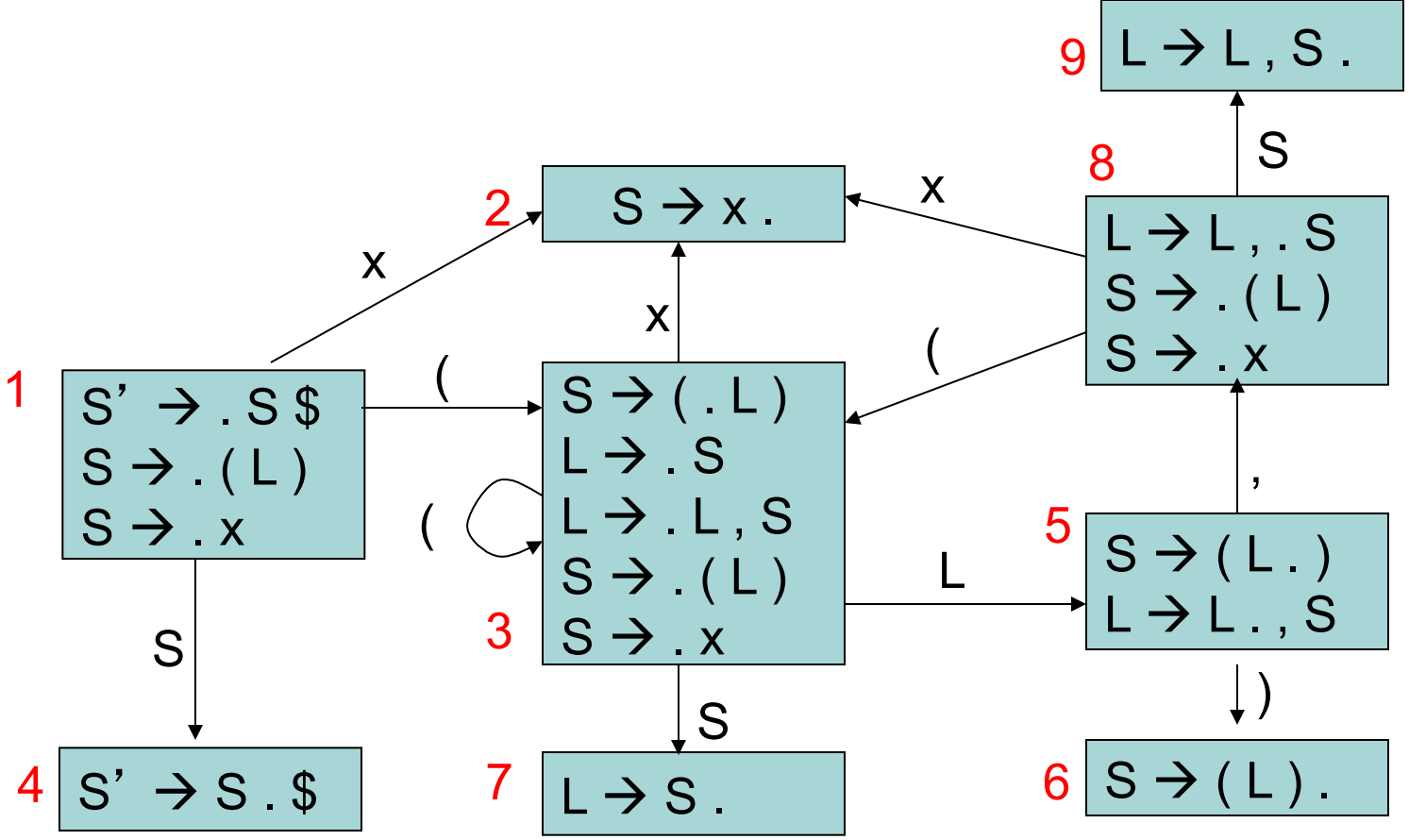
0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3							
4							
...							



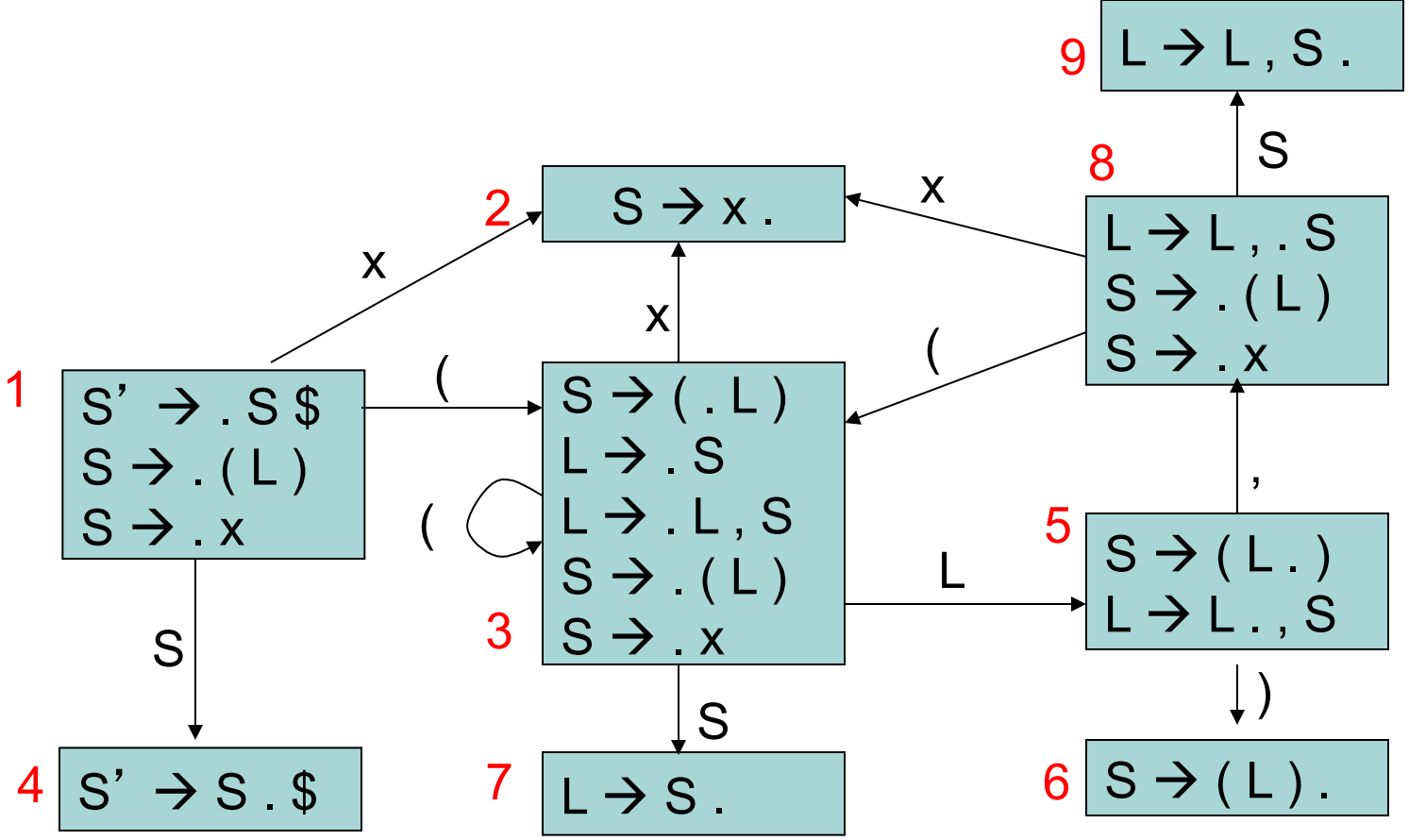
0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3							
4							
...							



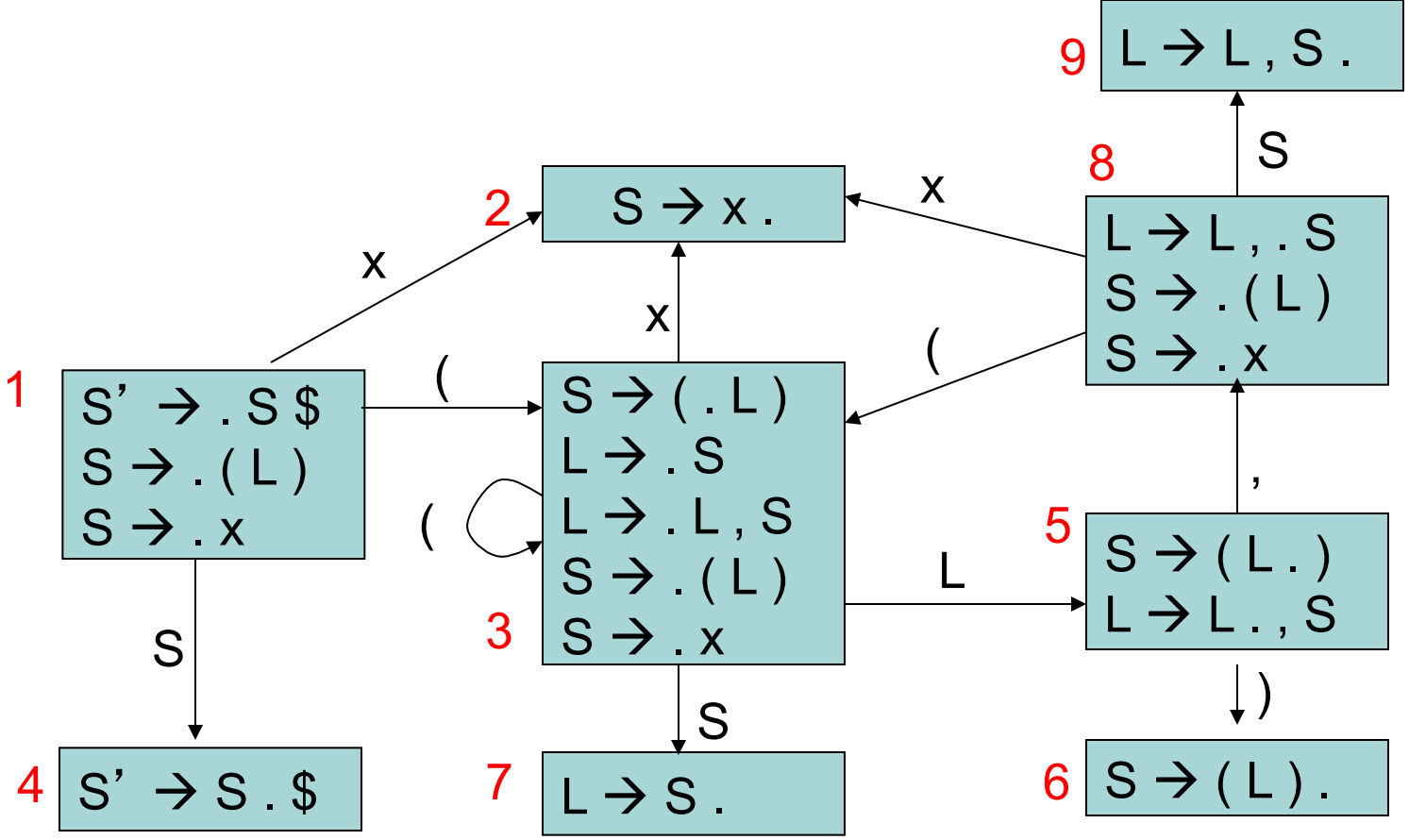
0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2				
4							
...							



0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4							
...							



0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
...							

The parse table

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

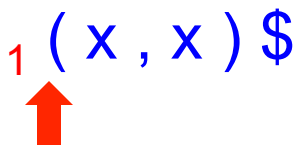
	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

action

goto

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S



states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1(_3$ x , x) $

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 x_2 , x) \$$

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3$ S , x) $

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 S_7 , x) \$$

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 L , x) \$$

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1$ ($_3$ L_5 , x) $

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1$ ($_3$ L_5 ,$_8$ x) $

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 \; L_5 \; ,_8 \; x_2 \;) \; \$$

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 \; L_5 \; ,_8 \; S \;) \; \$$

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 \; L_5 \; ,_8 \; S_9 \;) \; \$$

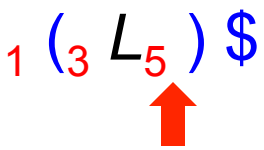
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 \; L \;) \; \$$

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S



states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

$_1 (_3 \; L_5 \;)_6 \; \$$

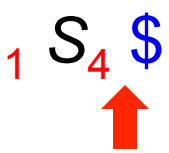
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S

1 S $

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

0. S' → S $
1. S → (L)
2. S → x
3. L → S
4. L → L , S



LR(0)

- Even though we are doing LR(0) parsing we are using some look ahead (there is a column for each non-terminal).
- However, we only use the terminal to figure out which state to go to next, not to decide whether to shift or reduce.

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5

LR(0)

- Even though we are doing LR(0) parsing we are using some look ahead (there is a column for each non-terminal).
- However, we only use the terminal to figure out which state to go to next, not to decide whether to shift or reduce.

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5

states	no look-ahead	S	L
1	shift	g4	
2	reduce 2		
3	shift	g7	g5

LR(0)

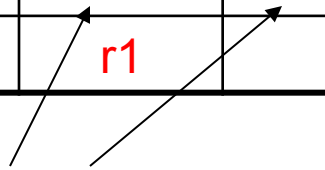
- Even though we are doing LR(0) parsing we are using some look ahead (there is a column for each non-terminal).
- However, we only use the terminal to figure out which state to go to next, not to decide whether to shift or reduce.
- If the same row contains both shift and reduce, we will have a conflict – i.e., the grammar is not LR(0).
- Likewise, if the same row contains reduce by two different rules.

states	no look-ahead	S	L
1	shift, reduce 5	g4	
2	reduce 2, reduce 7		
3	shift	g7	g5

SLR

- SLR (simple LR) is a variant of LR(0) that reduces the number of conflicts in LR(0) tables by using a tiny bit of look ahead.
- To determine when to reduce, 1 symbol of lookahead is used.
- Only put reduce by rule (X → RHS) in column T if T is in FOLLOW(X).

states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	s5	r2				
3	r1		r1	r5	r5	g7	g5



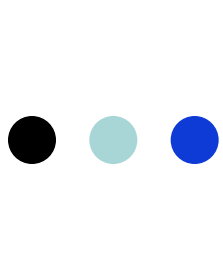
cuts down the number of r*k* slots & therefore cuts down conflicts

LR(1) & LALR

- LR(1) automata are identical to LR(0) except for the "items" that make up the states:
  - LR(0) items: X → s1 . s2
  - LR(1) items: X → s1 . s2, T

Lookahead symbol added.

- Idea: sequence s1 is on stack; input stream is s2 T.
- Find closure with respect to X → s1 . Y s2, T by adding all items Y → s3, U when Y → s3 is a rule and U is in FIRST(s2 T).
- Two states are different if they contain the same rules but the rules have different look-ahead symbols
  - Leads to many states.
  - LALR(1) = LR(1) where states that are identical aside from look-ahead symbols have been merged.
  - YACC/Bison & most parser generators use LALR by default.



Summary

- LR parsing is more powerful than LL parsing, given the same look ahead.
- To construct an LR parser, it is necessary to compute an LR parser table.
- The LR parser table represents a finite automaton that walks over the parser stack.
- YACC and Bison use LALR, a compact variant of LR(1).