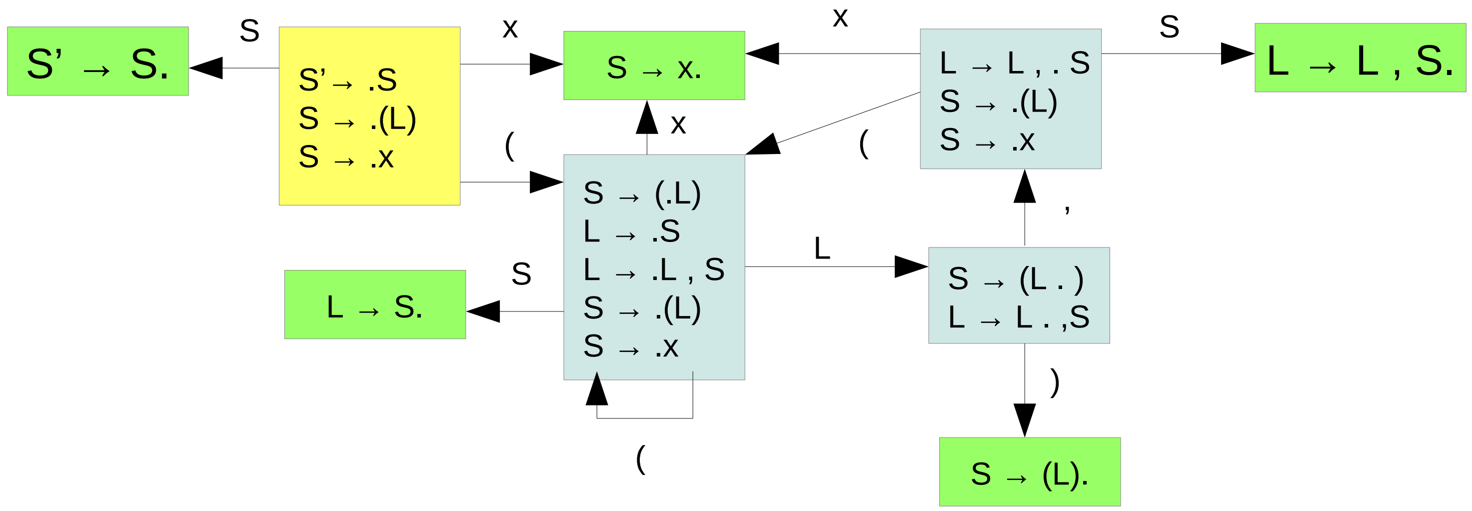
NTNU – Trondheim
Norwegian University of
Science and Technology

LR(0) parsing tables (and their application)

Where we are

- Last time, we looked at how stack machines remember the history of CFG productions they have taken, either
  - implicitly (via the function call stack), or
  - explicitly (automata with internal stacks)
- We constructed a pseudo-code LL(1) parser, based on its parsing table
  - Nice, because it is simple to do by hand
- We constructed an LR(0) automaton from a simple grammar
  - Nice to know (roughly) how parser generator output works

This is the LR(0) automaton we got out

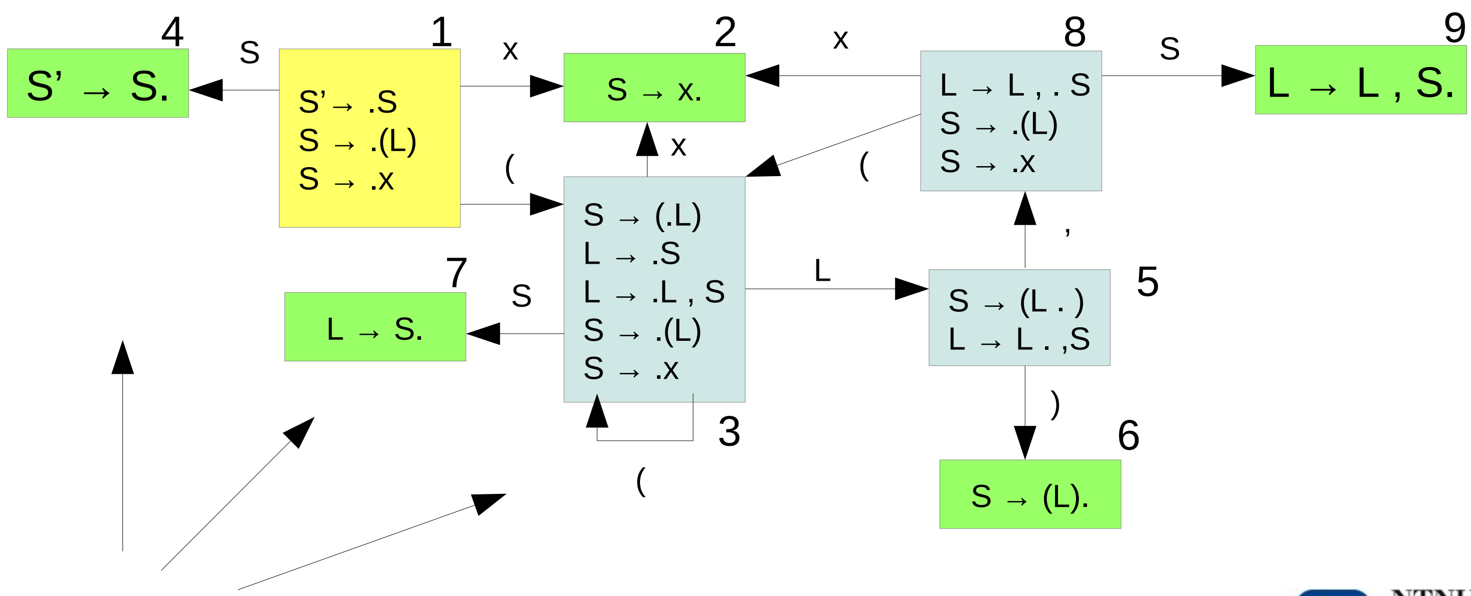


(Number the productions) →

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

Number Everything

- Since we want a table, it must have some indices



(Number the states)

Tabulate the transitions

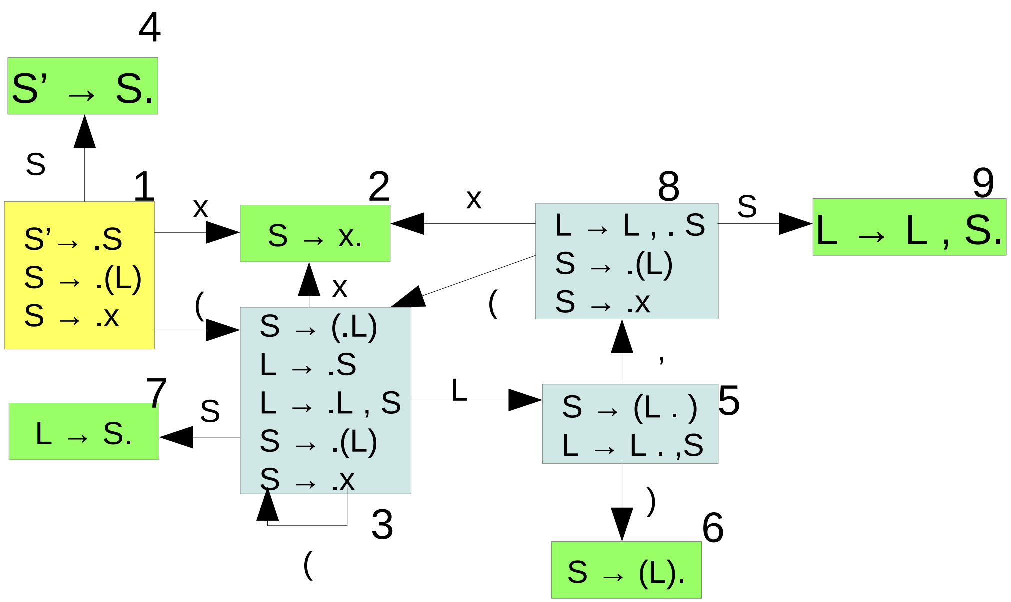
- The rows are our state indices
- The symbols we're looking at are at the top of the stack, they can be terminals or nonterminals
  - Terminals appear when you shift them there from the input
  - Non-terminals appear when some production is reduced
- Each pair of (state,symbol) identifies an action
  - Those are the table entries
- We've got three types of actions
  - Shift symbol and change to state (written as "s#", where # is the state)
  - Go to state (written as "g#", where # is the state)
  - Accept (written as "a")

NTNU – Trondheim
Norwegian University of Science and Technology

0) S’ → S
1) S → (L)
2) S → x
3) L → S
4) L → L , S

Structure of the table

- Here’s the automaton, and its empty parsing table:



	(Terminals)					(Non-terms)	
	(	)	x	,	$	S	L
1							
2							
3							
4							
5							
6							
7							
8							
9							

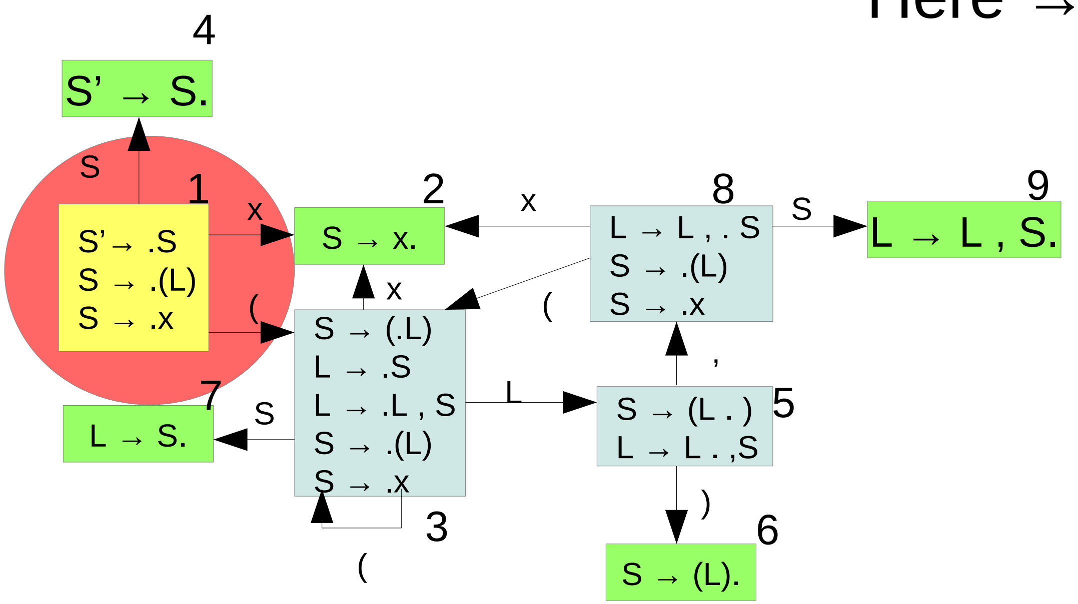
Filling it in

- Going through all the states that aren't accepting or reducing, look at the transitions
  - Transitions on terminals get a shift-and-goto action
  - Transitions on nonterminals just get the goto part

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

State 1

- There is S, x, and (



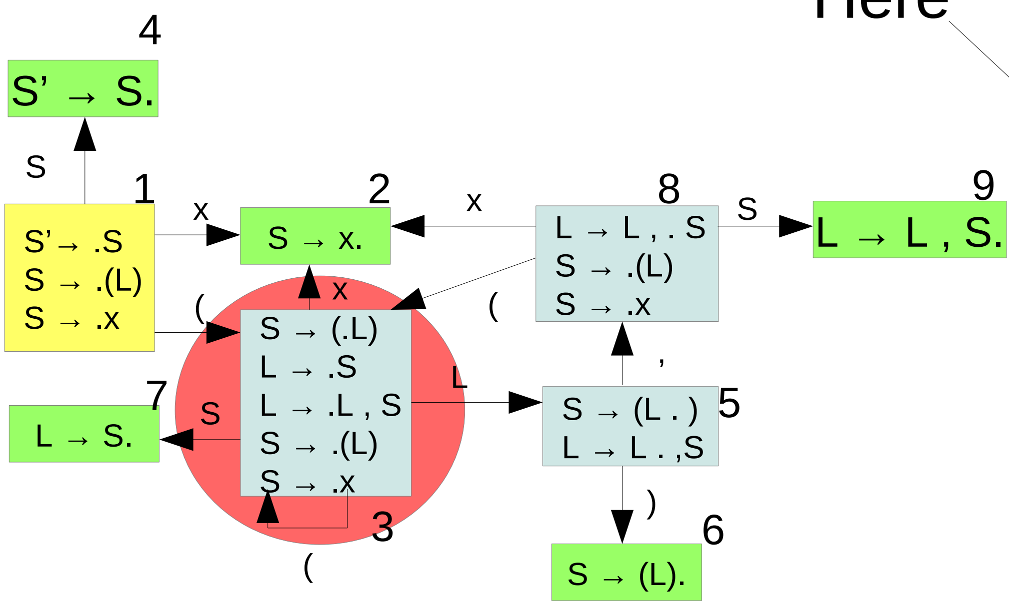
Here →

	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3							
4							
5							
6							
7							
8							
9							

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

State 3

- There is S, x, (, and L



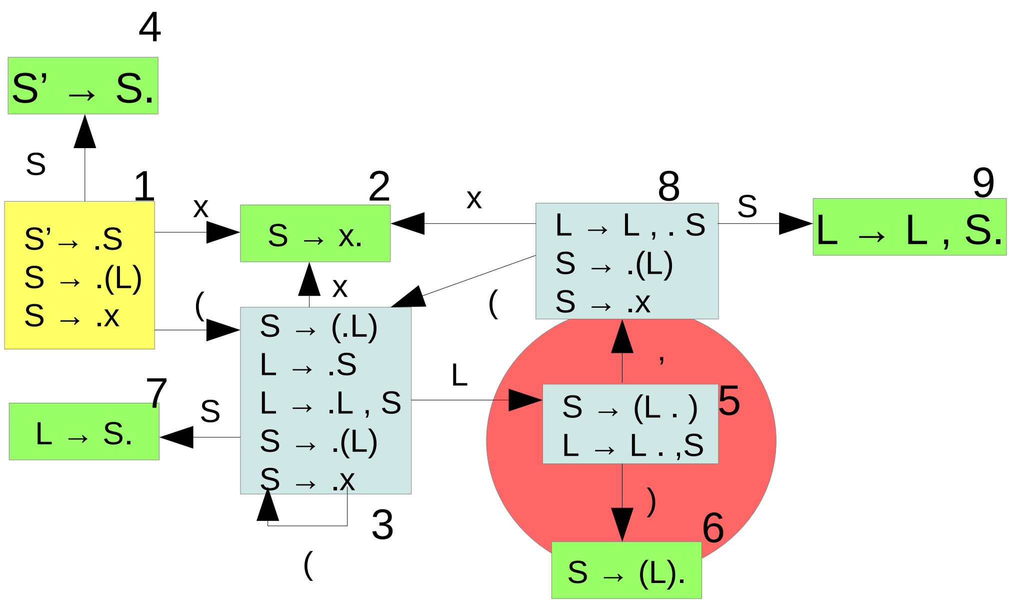
Here

	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3	s3		s2			g7	g5
4							
5							
6							
7							
8							
9							

0) S’ → S
1) S → (L)
2) S → x
3) L → S
4) L → L , S

State 5

- There is) and ,

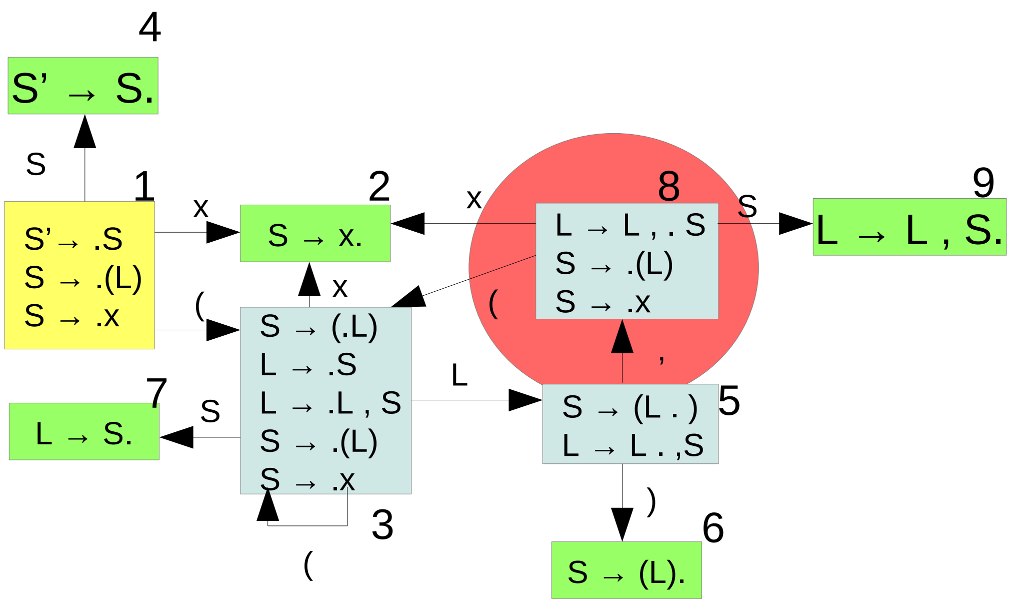


	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3	s3		s2			g7	g5
4							
5		s6		s5			
6							
7							
8							
9							

0) S’ → S
1) S → (L)
2) S → x
3) L → S
4) L → L , S

State 8

- There is x, (, and S



	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3	s3		s2			g7	g5
4							
5		s6		s5			
6							
7							
8	s3		s2			g9	
9							

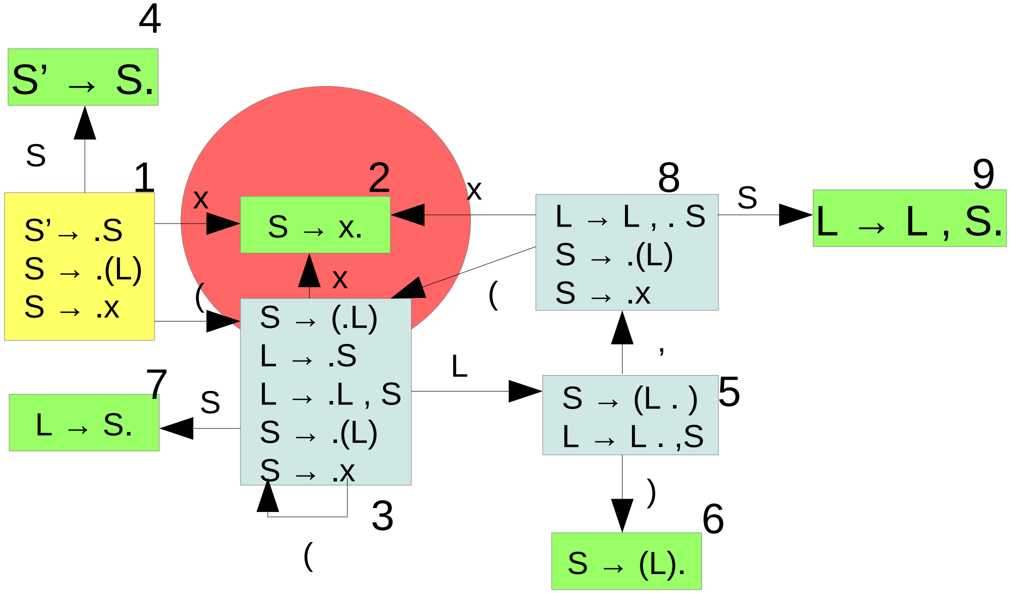
Halfway there

- Those were the ‘ordinary’ states, we still need to do something with reducing states and accept
- For LR(0), a reducing state has no need to know anything about the top of the stack
  - it’s determined because building a particular sequence at the top of the stack is what brought us to the reducing state in the first place
- Thus, reduce actions go in every terminal column for the reducing state
  - We can write them as “r#” where # is the grammar production being reduced

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

State 2

- This reduces rule #2, S → x

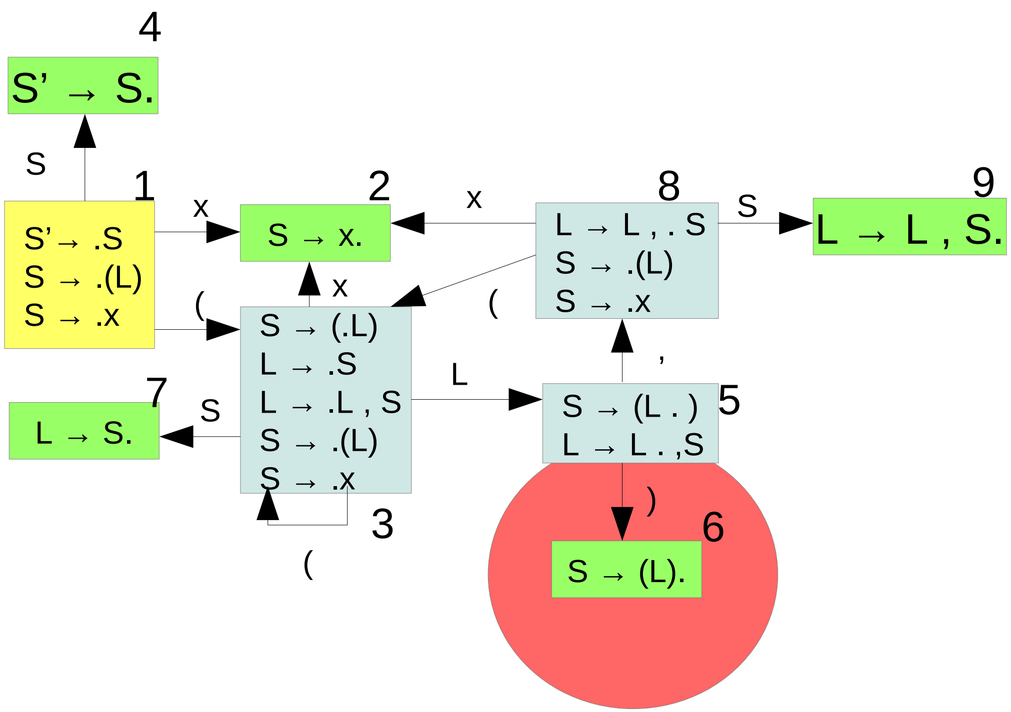


	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4							
5		s6		s5			
6							
7							
8	s3		s2			g9	
9							

0) S’ → S
1) S → (L)
2) S → x
3) L → S
4) L → L , S

State 6

- This reduces rule #1, S → (L)

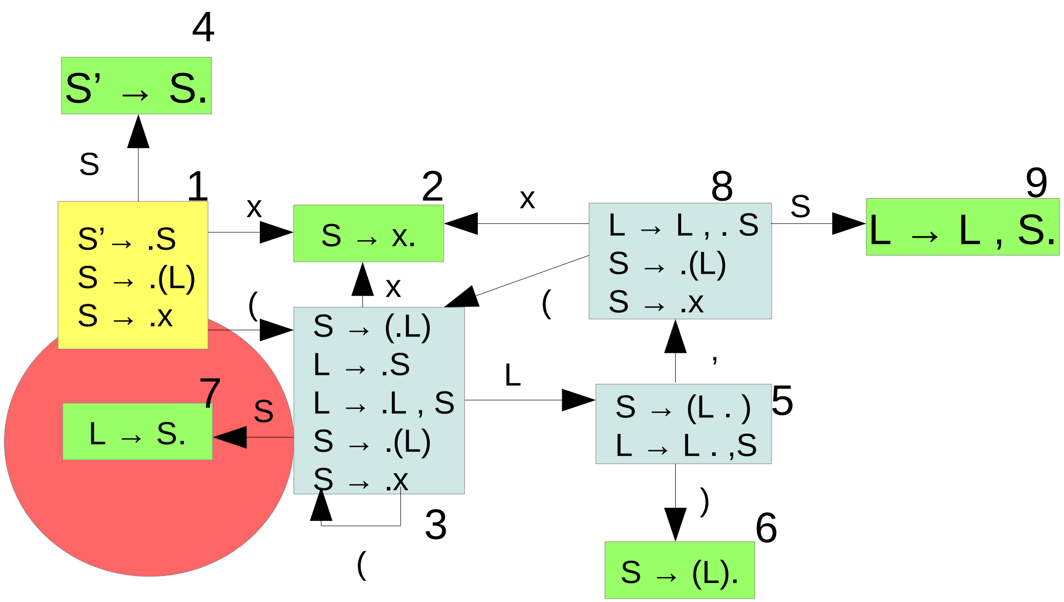


	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4							
5		s6		s5			
6	r1	r1	r1	r1	r1		
7							
8	s3		s2			g9	
9							

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

State 7

- This reduces rule #3, L → S

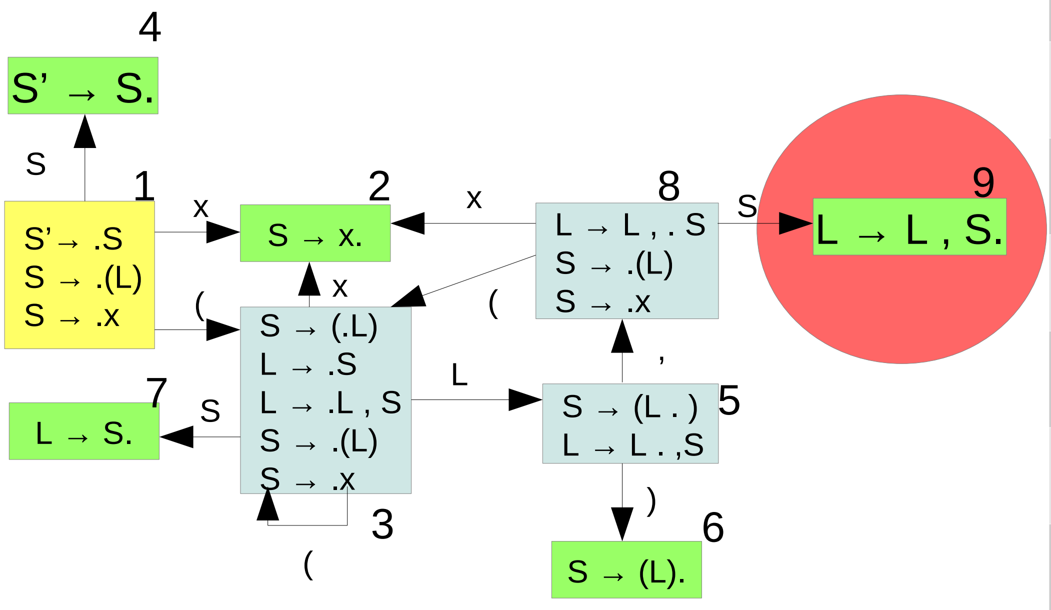


	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4							
5		s6		s5			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9							

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

State 9

- This reduces rule #4, L → L,S



	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4							
5		s6		s5			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

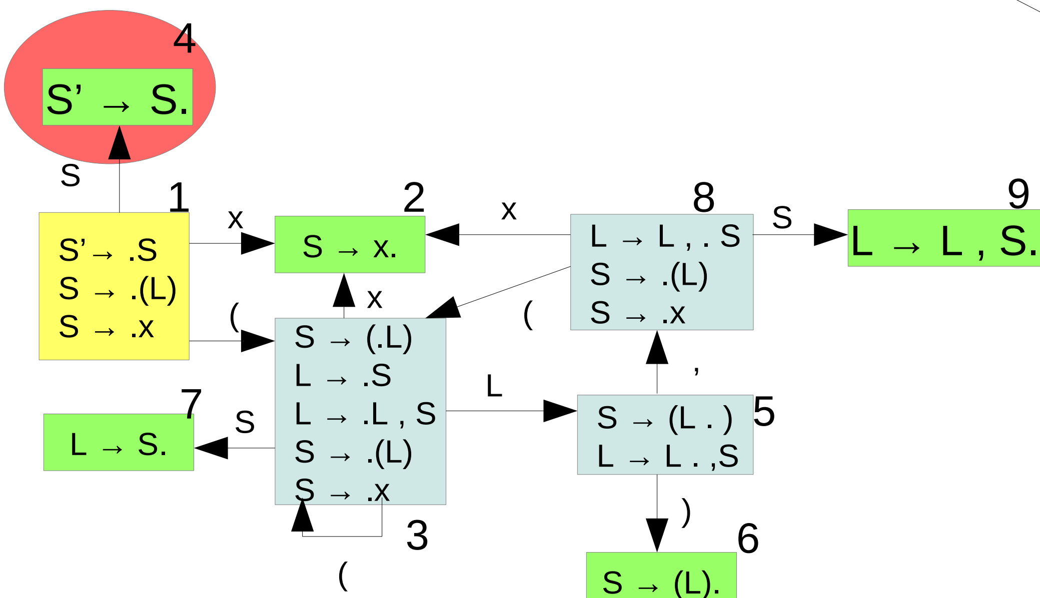
The accepting state

- Accepting states are extremely easy since we started by adding an extra grammar rule to represent this alone
  - That is, S’ → S
- If the input is correct, this reduces precisely when we are out of terminals
  - So: shift the end-of-input marker, and conclude parsing

0) S’ → S
1) S → (L)
2) S → x
3) L → S
4) L → L , S

State 4 accepts

- This reduces our whole syntax enchilada



	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s5			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

A bottom-up traversal

- Using the table we've constructed, we can see how it plays out when parsing a statement like (x,(x,x))

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s5			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

The procedure has 29 steps, so we'll have to do it in parts...

(History)	State	Stack	Input	Action	(Backtrack)
	1	-	(x,(x,x))	s3	
1	3	(	x,(x,x))	s2	
1,3	2	(x	,(x,x))	r2	Throw 2, rev. to 3
1	3	(S	,(x,x))	g7	
1,3	7	(S	,(x,x))	r3	Throw 7, rev. to 3
1	3	(L	,(x,x))	g5	
1,3	5	(L	,(x,x))	s8	
1,3,5	8	(L,	(x,x))	s3	
1,3,5,8	3	(L,(	x,x))	s2	
1,3,5,8,3	2	(L,(x	,x))	r2	Throw 2, rev. to 3
1,3,5,8	3	(L,(S	,x))	g7	
1,3,5,8,3	7	(L,(S	,x))	r3	Throw 7, rev. to 3
1,3,5,8	3	(L,(L	,x))	g5	
1,3,5,8,3	5	(L,(L	,x))	s8	

(Replicate the last row, pick up where we were)

(History)	State	Stack	Input	Action	(Backtrack)
1,3,5,8,3	5	(L,(L	,x))	s8	
1,3,5,8,3,5	8	(L,(L,	x))	s2	
1,3,5,8,3,5,8	2	(L,(L,x	))	r2	Throw 2, rev. to 8
1,3,5,8,3,5	8	(L,(L,S	))	g9	
1,3,5,8,3,5,8	9	(L,(L,S	))	r4	Throw 9,8,5, rev. to 3
1,3,5,8	3	(L,(L	))	g5	
1,3,5,8,3	5	(L,(L	))	s6	
1,3,5,8,3,5	6	(L,(L)	)	r4	Throw 6,5,3, rev. to 8
1,3,5	8	(L,S	)	g9	
1,3,5,8	9	(L,S	)	r4	Throw 9,8,5, rev. to 3
1	3	(L	)	g5	
1,3	5	(L	)	s6	
1,3,5	6	(L)	$	r4	Throw 6,5,3, rev. to 1
-	1	S	$	g4	

In state 4...

(History)	State	Stack	Input	Action	(Backtrack)
-	4	S	$	accept	

...that's all she wrote.

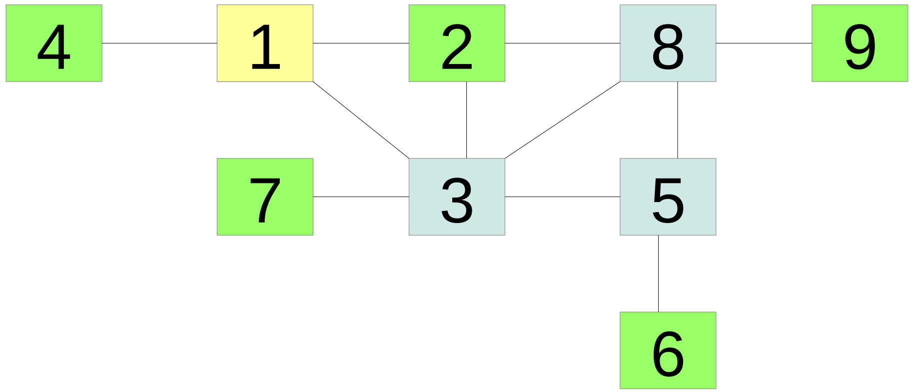
- We have read all the input, and gotten the start symbol + the end of input

The ‘0’ in LR(0)

- It can be slightly tricky to see how the machine operates
  - At least if you’re stuck in the LL(1) mind-set of making decisions based on what’s coming next on the input
- The ‘0’ is ‘0 lookahead symbols’
  - If there is no transition to take based on the top-of-stack, shift another token and *then* see where it takes you
  - The shift-and-goto maneuver could merit 2 rows of derivation steps, but then our walkthrough would be almost twice as long

A cleaner diagram

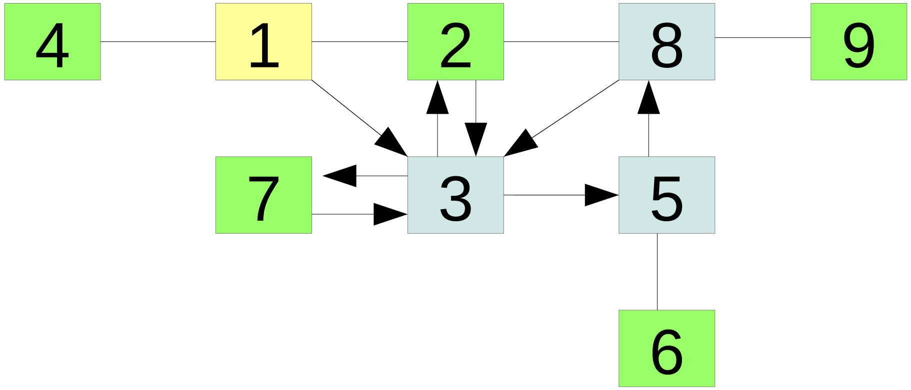
- If we simplify the machine a little, it looks like this:



The beginning of our traversal

- The first few steps went

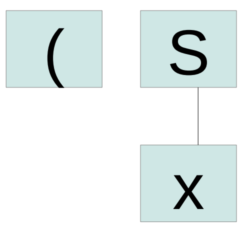
  1,3,2,3,7,3,5,8,3,2,...



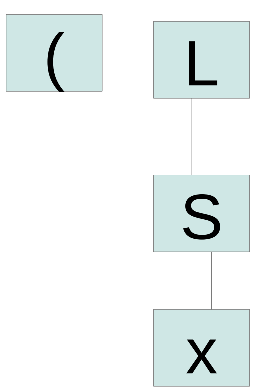
(Trace it out with your finger)

The matching syntax (sub-)trees

- 1,3,2 walks through

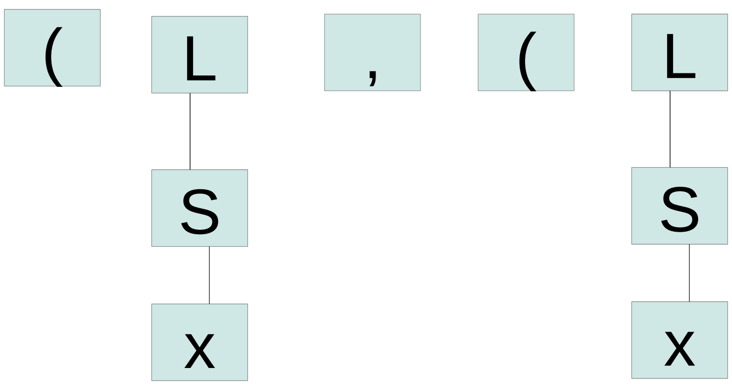


- 3,7 extends what we've seen (and remember) to



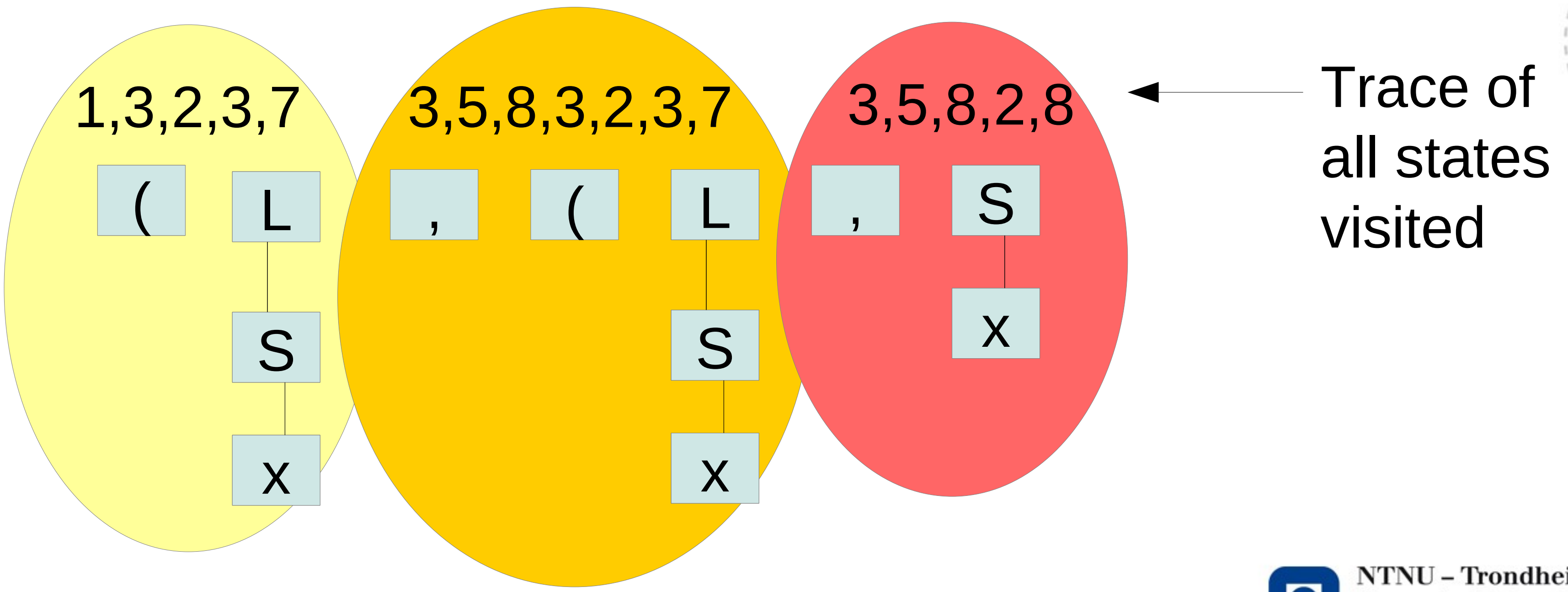
The matching syntax (sub-)trees

- 3,5,8,3<u>,2,3,7</u> passes a ‘,’ $5 \rightarrow 8$, and a ‘(‘ $8 \rightarrow 3$, and does <u>the same thing over again</u>



The matching syntax (sub-)trees

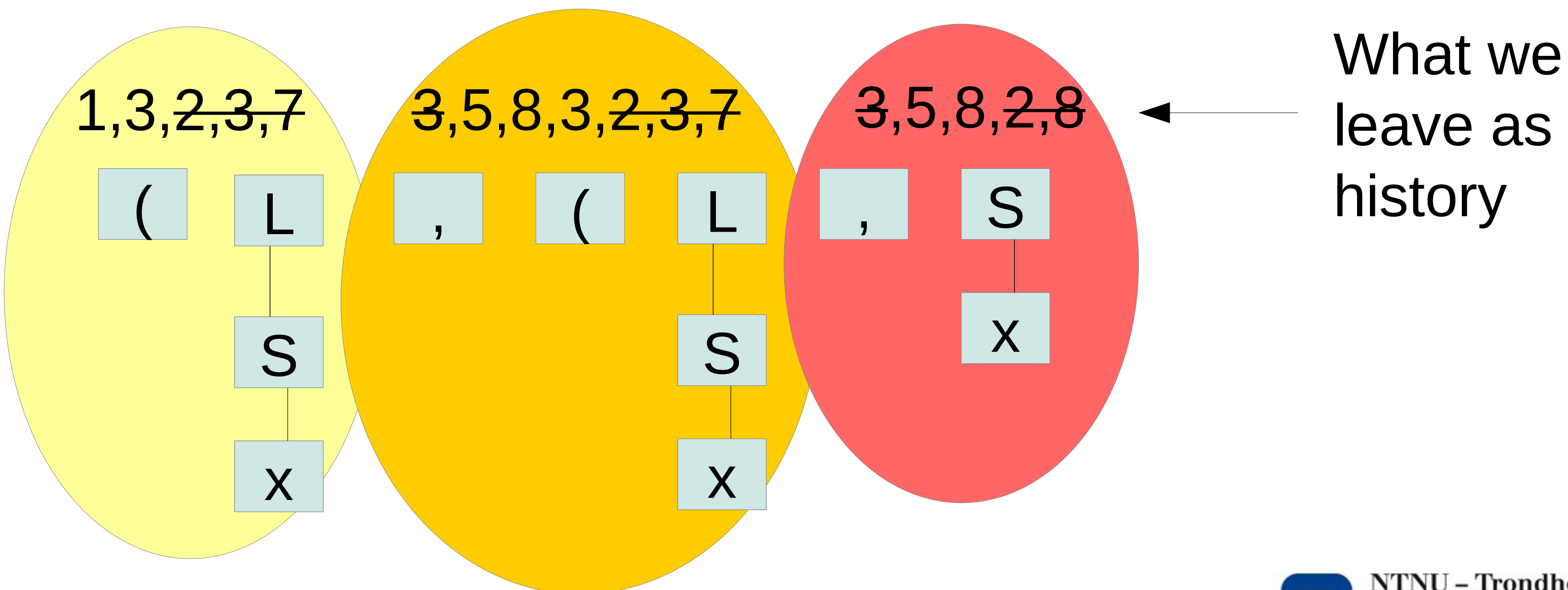
- 3,5,8,2,8 passes ',' 5->8, reduces S (8→2 and back)...



NTNU – Trondheim
Norwegian University of Science and Technology

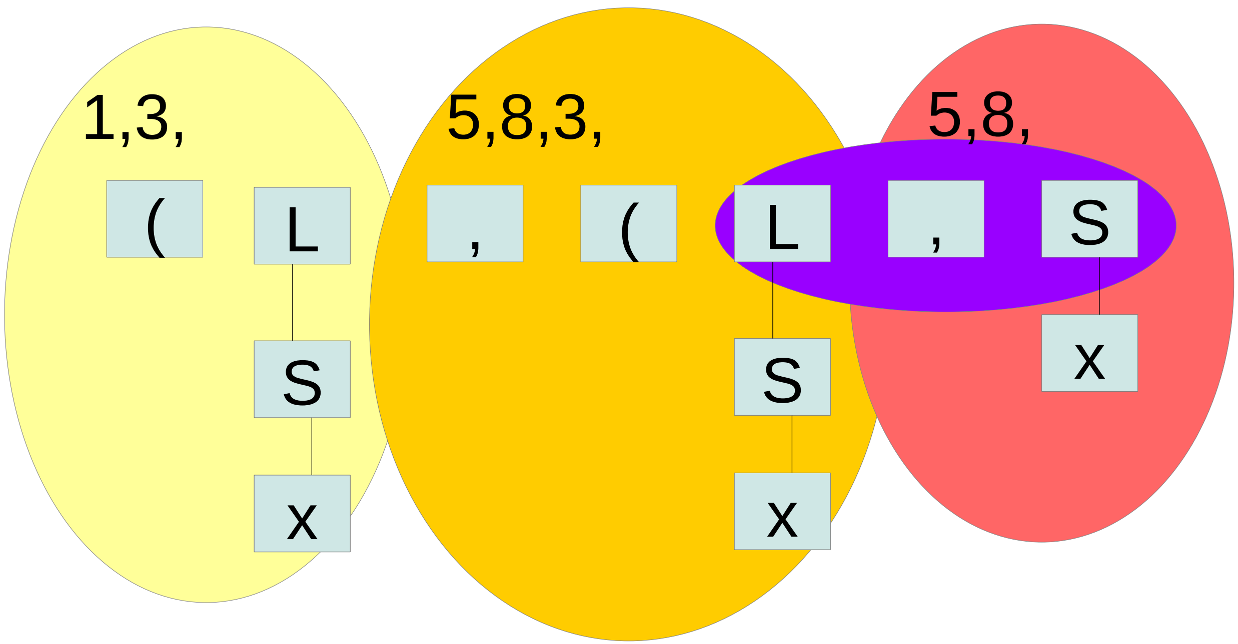
The matching syntax (sub-)trees

- If we strike out the detours/backtracking, (1,3,5,8,3,5,8) is where we were before reaching 9



The matching syntax (sub-)trees

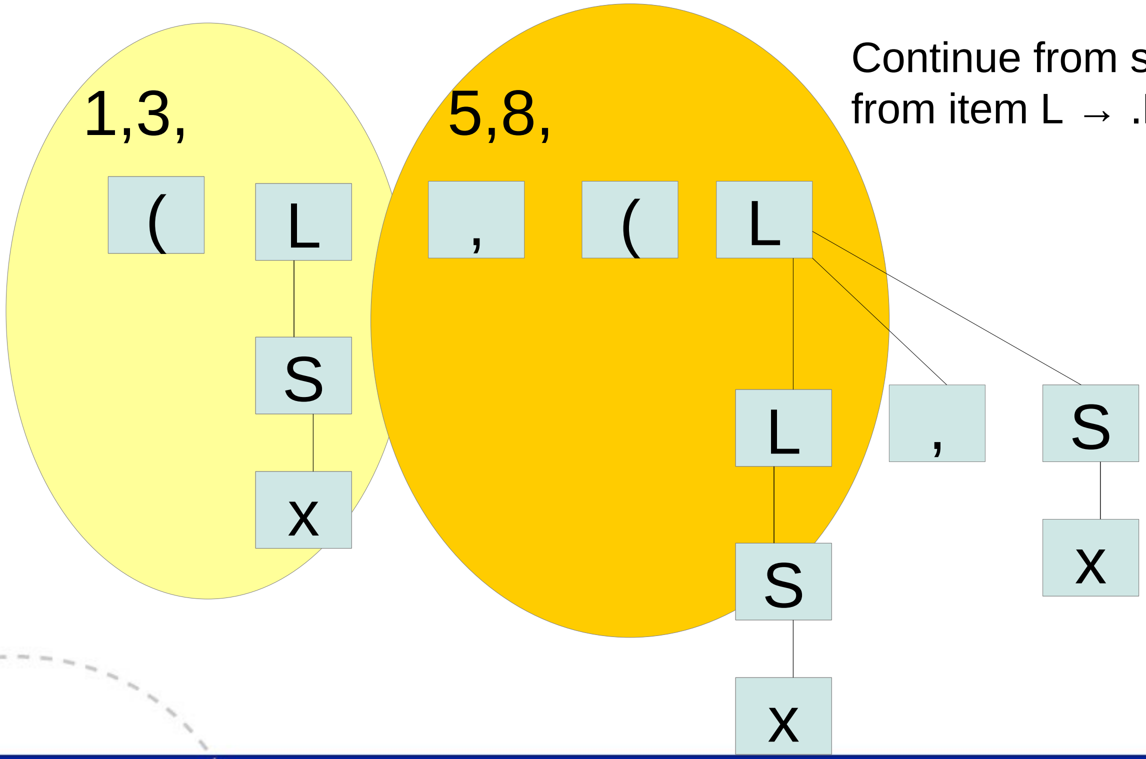
- We're beginning to get right-hand sides which are not just trivial 1-symbol reductions



State 9, Eureka!

The matching syntax (sub-)trees

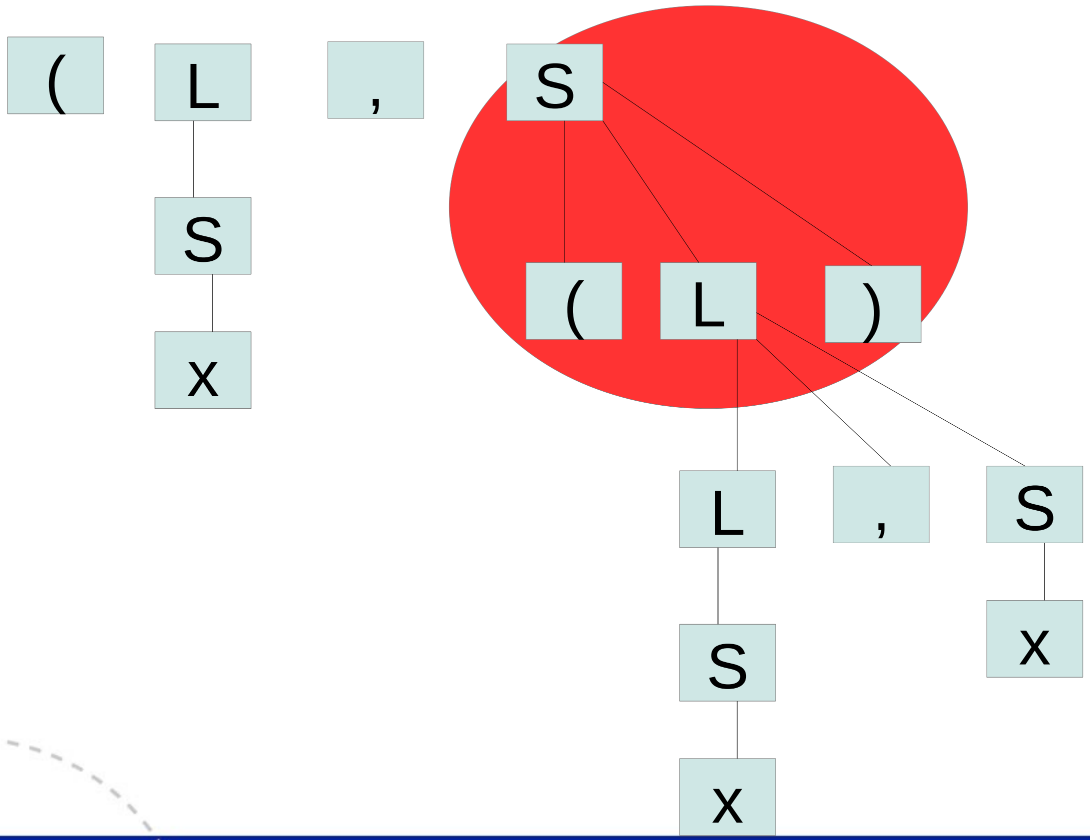
- State 9 reduces a right-hand side with multiple non-terminals, and must revert by 3 stages because it concludes 3 choices of direction: the L, the comma, and the S.



Continue from state 3, it's where we began from item L → .L,S to reach item L → L,S.

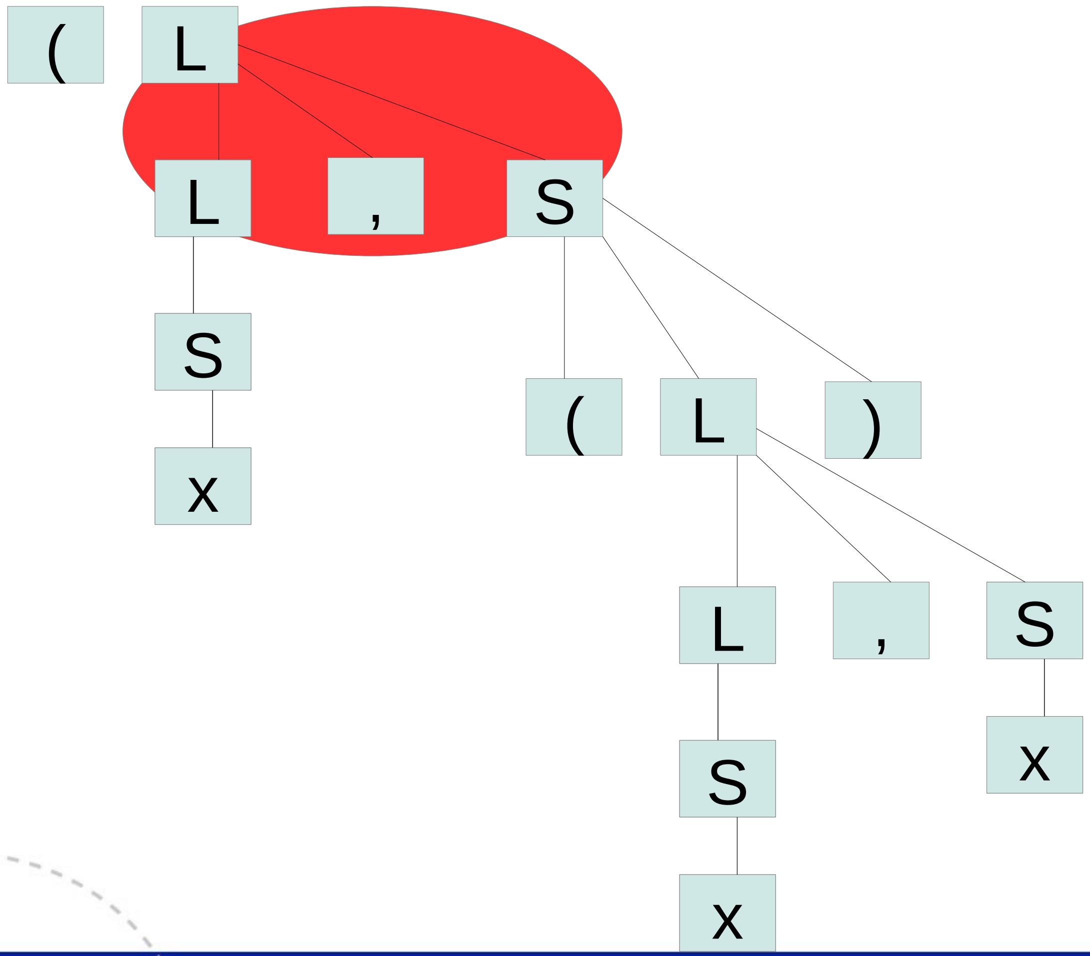
...and so it proceeds...

...shifting), and passing by the reduction in state 6...

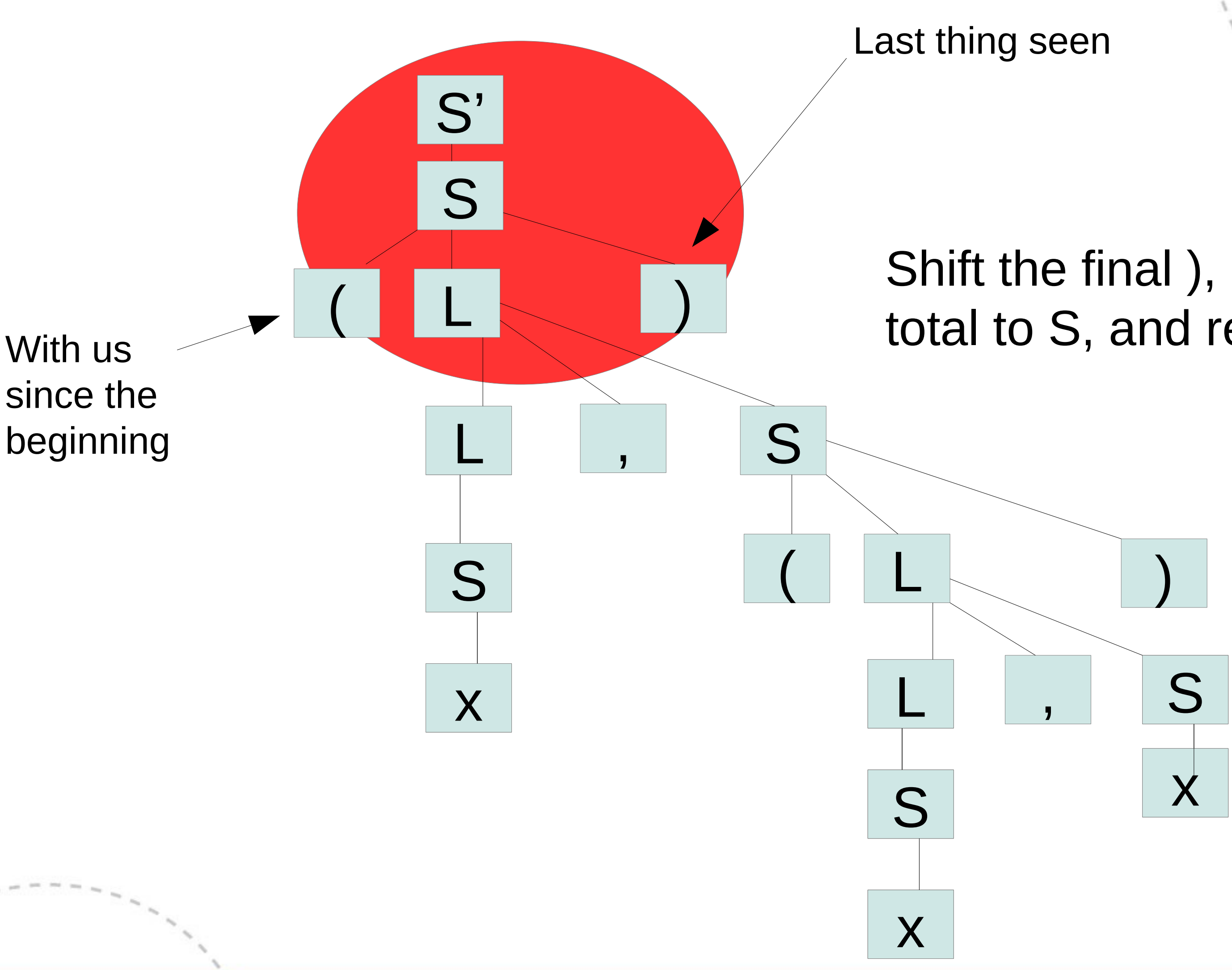


...and proceeds...

...visiting state 9 again, to reduce another L...



...until the end.



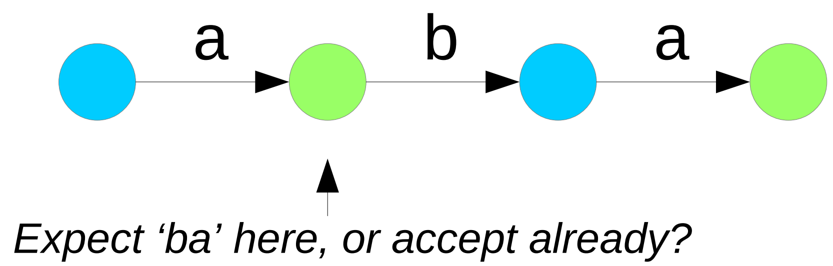
Shift the final), reduce the total to S, and reduce S to S'

As you can see

- Top-down parsing creates **leftmost** derivations, by taking the leftmost nonterminal and predicting the input yet to come
- Bottom-up parsing creates **rightmost** derivations, by working ahead in the input, and stacking up all the nonterminals it passed on the way, until they are completed

What's ahead

- We already know of DFA that they can give conflicting decisions:



- Regular expression matchers commonly buffer, and accept the longest match in the end
- LR parsers see these situations as well, they're called *shift/reduce* conflicts in such a context
- LR(0) isn't very flexible when it comes to these, so next, we'll extend it with different ways to see what's coming.