Introduction to LR Parsing

Dr. William Harrison
harrisonwl@missouri.edu
CS 4430 Compilers I

Announcements

- Midterm, Wednesday March 1.
- HW1 deadline extension until 2/22 @ 11:59pm.

Today's Class

- Starting Section 3.4 in Wilhelm text
- More About Parsing – recognizing languages
- LR parsers, LR parsing engine.
- Next Class
  - More on LR parsers: LR(1), LALR(1).
  - Tools for constructing parsers.

Shortcomings of LL Parsers

- Recursive decent renders a readable parser.
- But consider implementing this grammar

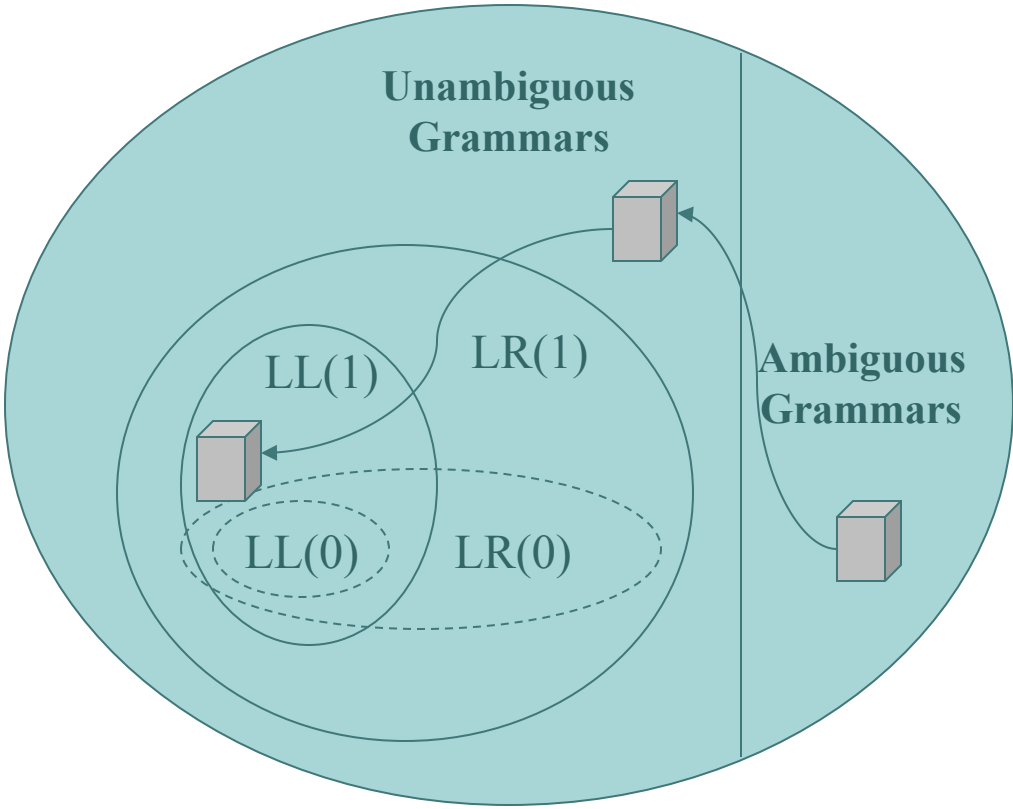
E → E + T
E → T
T → id

```
void E(){switch(tok) {
  case ?: E(); eat(TIMES); T(); ← no way of choosing production
  case ?: T();
  …
}
void T(){eat(ID);}
```

Predictive parsing depends on the first terminal symbol of each sub-expression providing enough information to choose which production to use.

Grammar Hierarchies (1)

- Grammars characterize languages.
- Grammars can be
  - Ambiguous
  - Unambiguous



- Compiler engineers rework grammars
  - From ambiguous to unambiguous
  - Into the chosen grammar class.

Two Styles of Derivation

- Leftmost derivation
- Always expand the leftmost non-terminal

S	
S ; S	*S → S ; S*
id := E ; S	*S → id := E*
id := num ; S	*E → num*
id := num ; id := E	*S → id := E*
…	

Rightmost derivation

- Always expand the rightmost non-terminal

S	
S ; S	*S → S ; S*
S ; id := E	*S → id := E*
S ; id := E + E	*E → E + E*
E)	*S → (S ,*
…	
id := num ; id := E + (S , E)	
…	

LL(k) vs. LR(k)

- **L**eft to Right parse
- **L**eftmost derivation
- ***k***-token look ahead

➔ LL(k)

- Needs to predict what production to use after seeing only *k* tokens from the right hand side.
- Both hand-written (recursive descent) and built by tools.
- Considered faster, but might use backtracking.
- Recently seen a renaissance.
  - ANTLR and javacc for Java
- Need to tweak the original grammar

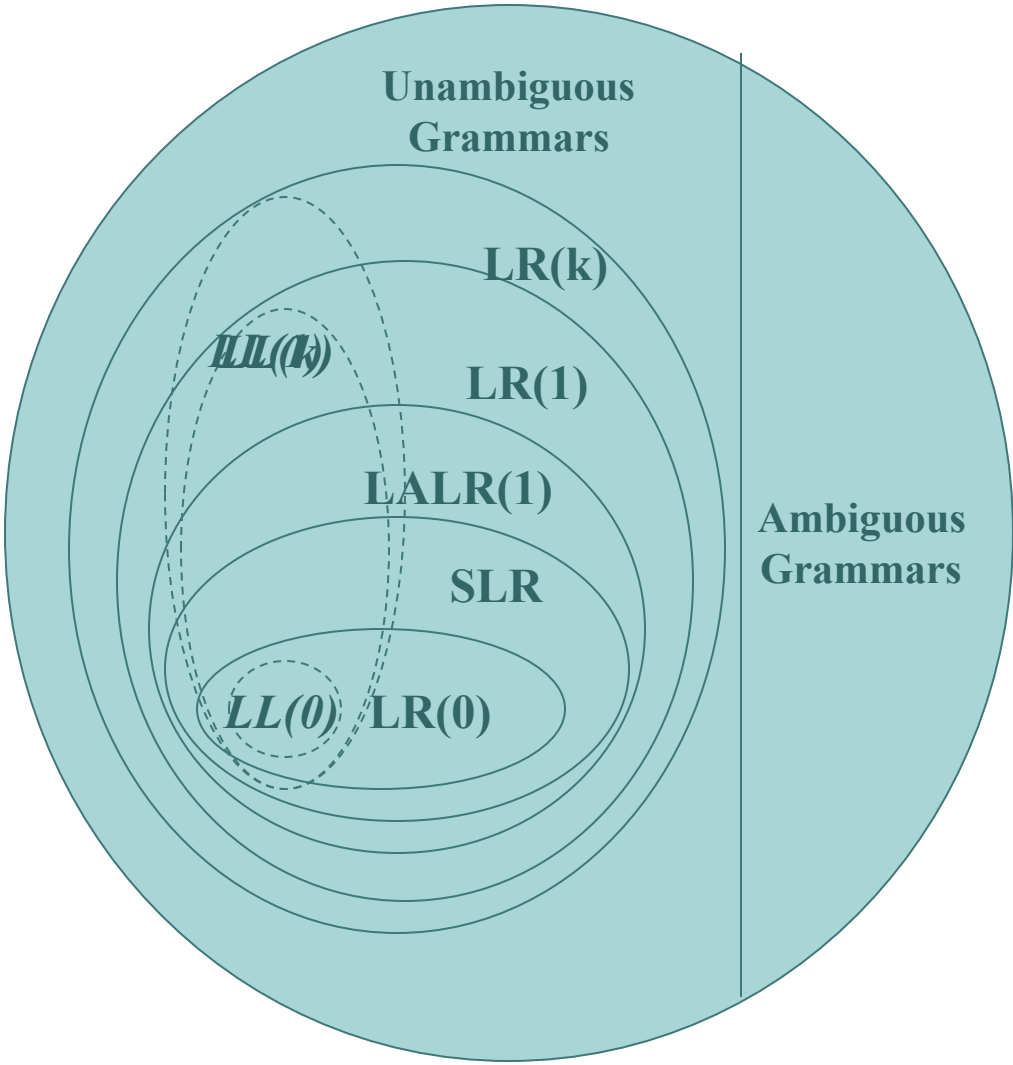
- **L**eft to Right parse
- **R**ightmost derivation
- ***k***-token look ahead

➔ LR(k)

- Can see the input corresponding to a specific non-terminal (and *k* tokens after) before needing to choose which production to use.
- Typically built by tools.
- Most popular for real parsing
  - YACC, CUP for Java, sablecc
- Also need to tweak the original grammar

LR Parsing

- LR grammars are strictly stronger than LL grammars.
  - LR(0) is of academic interest only.
- Simple LR (SLR) can parse some interesting languages.
- Most "real computer languages" can be expressed as LALR(1) grammars.
  - Many parser generators use this class of grammars.
- LR(1) is a very powerful parsing technology.
  - The implementation of LR(1) can get unwieldy.
  - Engineers try rewrite their grammars into LALR(1).
- LR parsers uses a LR parsing engine

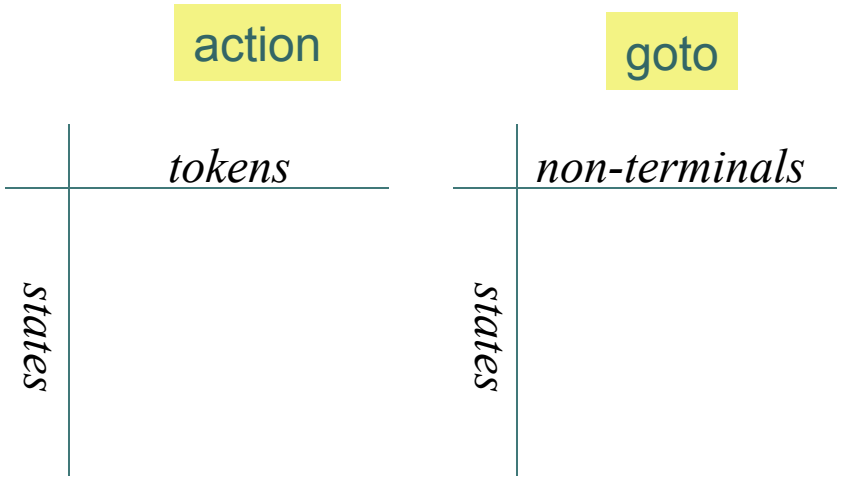


The LR Parsing Engine

- Four actions: shift(p), reduce(p), accept, error
  - "p" stands for "production number"
- Engine structure:

Input: $t_0\ t_1\ t_2\ t_3 \ldots$

Stack: … S n



N.b., I'm describing the LR(0) engine here

shift(p)

Input: $t_0\ t_1\ t_2\ t_3 \ldots$

Stack: … S n

	action	goto
	t_0	*non-terminals*
n	shift(p)	

states

after

Input: $t_1\ t_2\ t_3 \ldots$

Stack: … S n t_0 p

reduce(p)

Input: $t_0\ t_1\ t_2\ t_3 \ldots$

Stack: $k\ x_1\ \ldots\ x_i\ n$

Prod. p: $X \rightarrow Y_1\ \ldots\ Y_i$

action		goto	
	t_0		X
n	reduce(p)		
		k	m

after

Input: $t_1\ t_2\ t_3 \ldots$

Stack: $\ldots\ k\ X\ m$

Example LR engine tables

(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

LR parsing a small grammar (1)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

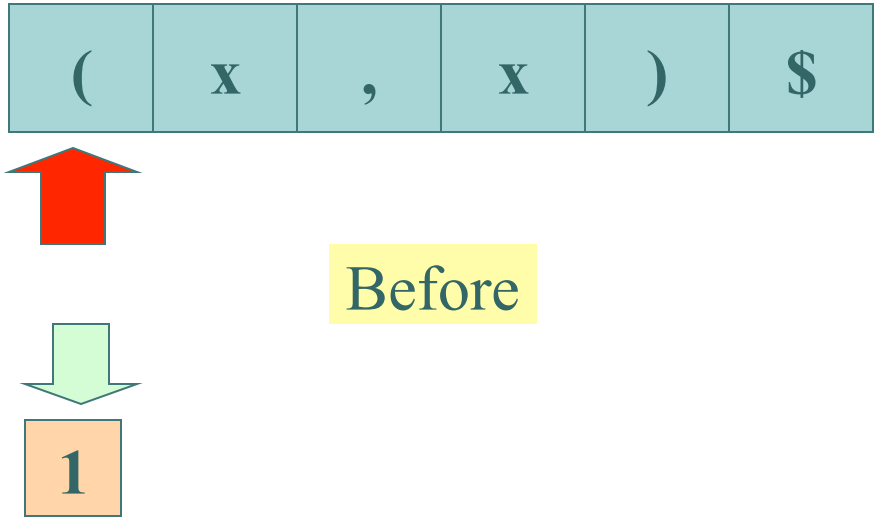
This grammar is LR(0)
(It is not LL(1))

Let us parse (x , x) $

- 4 instructions in LR engine
  - Shift(*n*)
    - Advance input
    - push *n* on stack
  - Reduce(*k*)
    - Pop things from stack
    - Lookup new state number
    - Apply reduction rule *k*
  - Accept
  - Error

LR parsing a small grammar (2)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



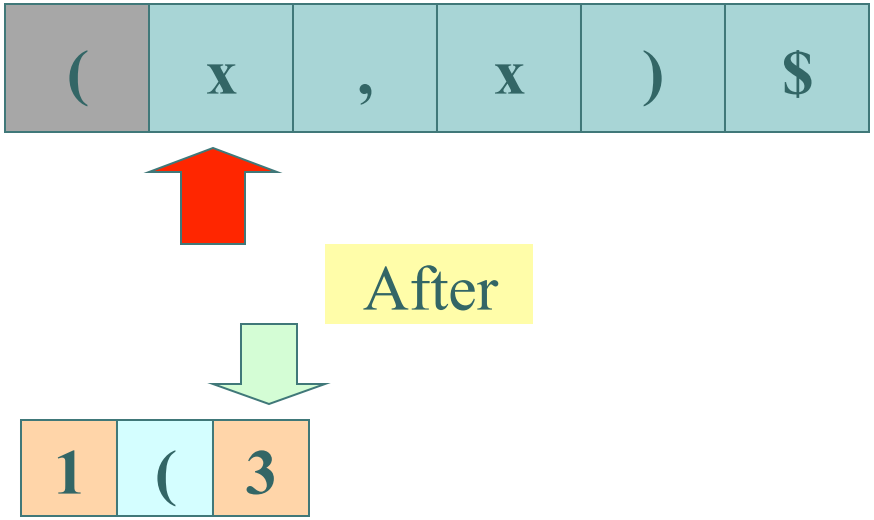
Always the number at top of stack

action[1][(] ➔ Shift(3)

- Advance input one token
- Push (
- Push 3

LR parsing a small grammar (3)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

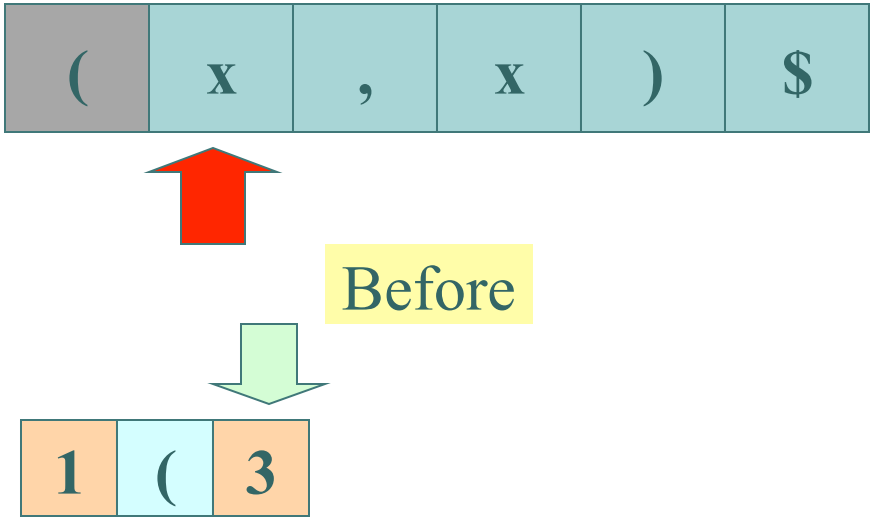


action[1][(] ➔ Shift(3)

- Advance input one token
- Push (
- Push 3

LR parsing a small grammar (4)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

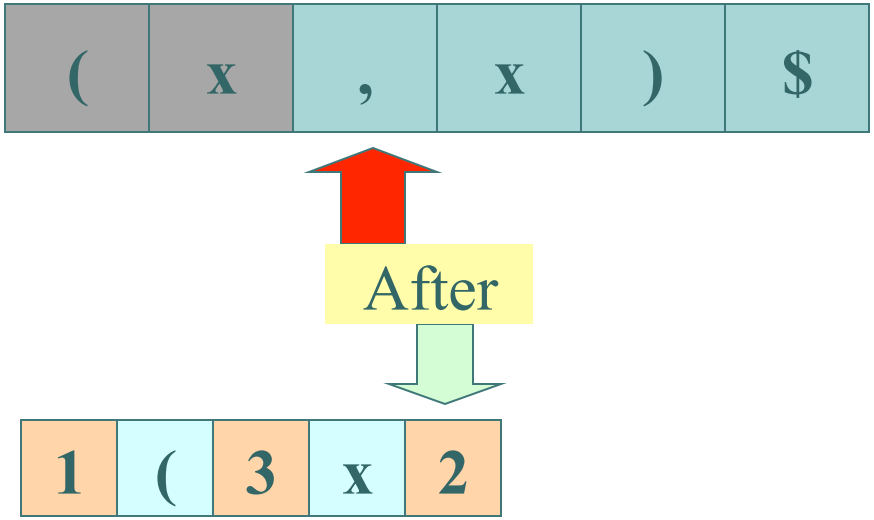


action[3][x] ➔ Shift(2)

- Advance input one token
- Push x
- Push 2

LR parsing a small grammar (5)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

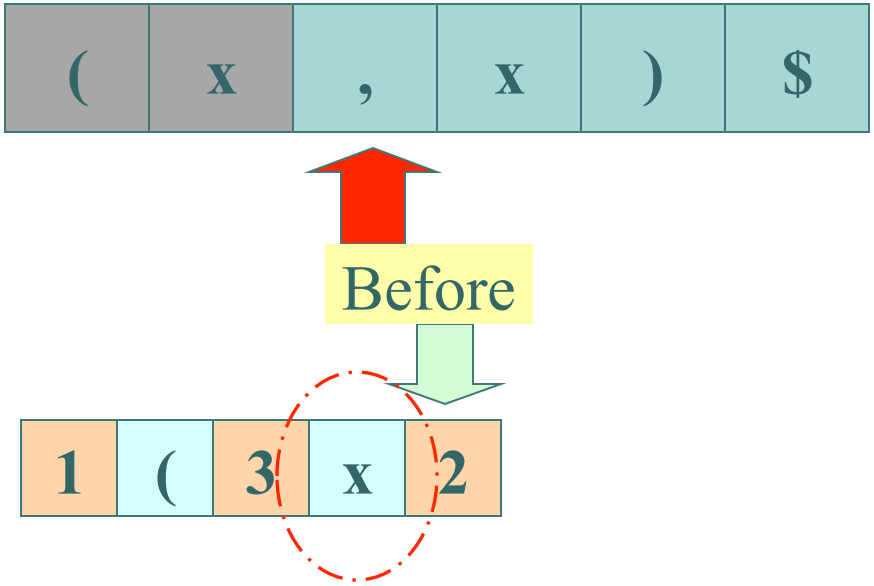


action[3][x] ➔ Shift(2)

- Advance input one token
- Push x
- Push 2

LR parsing a small grammar (6)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



action[2][,] ➔ Reduce(2)
- Pop 2 and x
- Reduce using production (2)
  - S → x

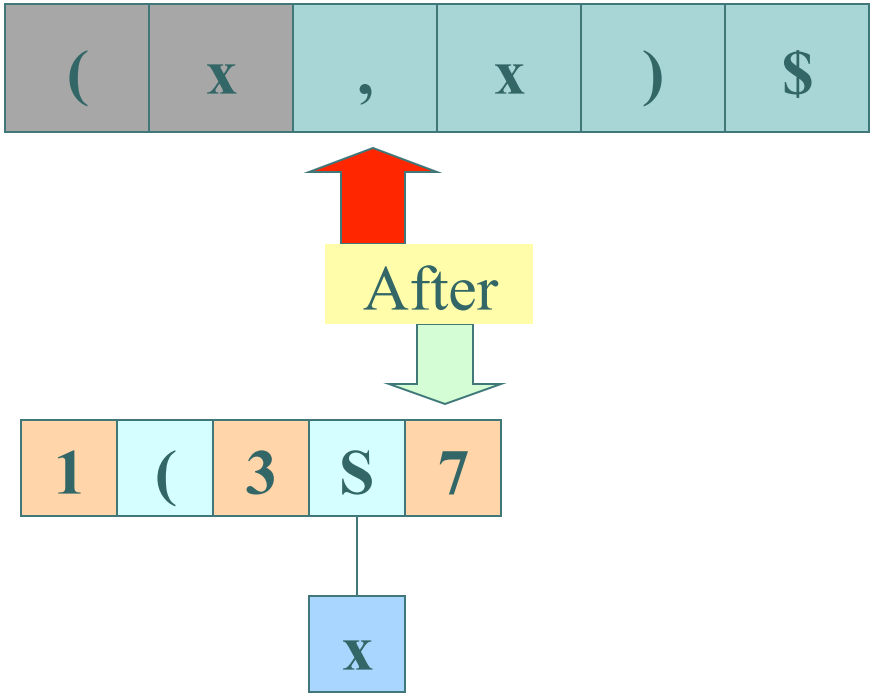
goto[3][S] ➔Goto(7)
- Push S onto stack
- Push 7 onto stack

LR parsing a small grammar (7)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

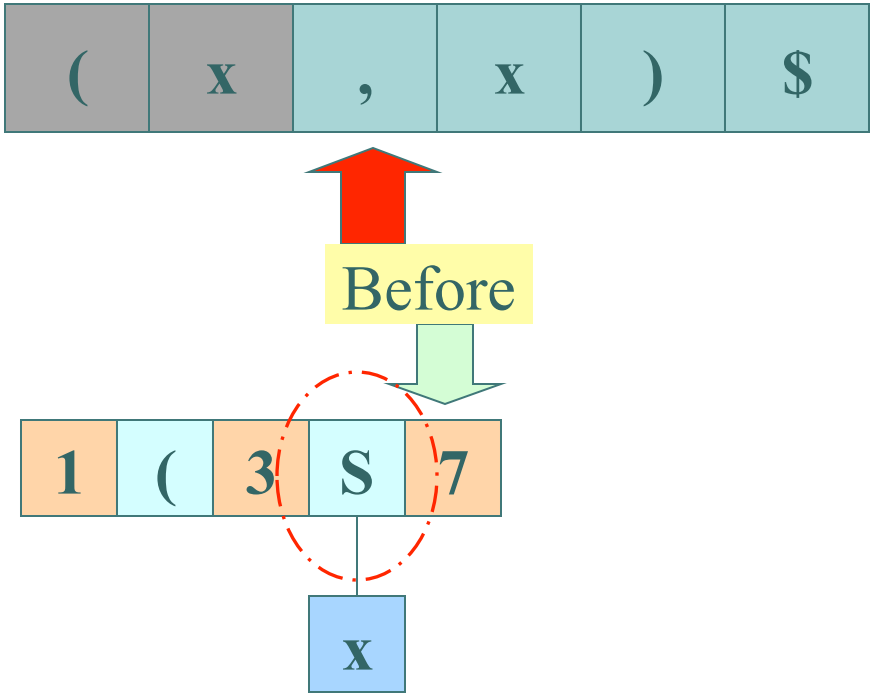
action[2][,] ➔ Reduce(2)
- Pop 2 and x
- Reduce using production (2)
  - S → x

goto[3][S] ➔Goto(7)
- Push S onto stack
- Push 7 onto stack



LR parsing a small grammar (8)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



action[7][,] ➔ Reduce(3)
- Pop 7 and S
- Reduce using production (3)
  - L → S

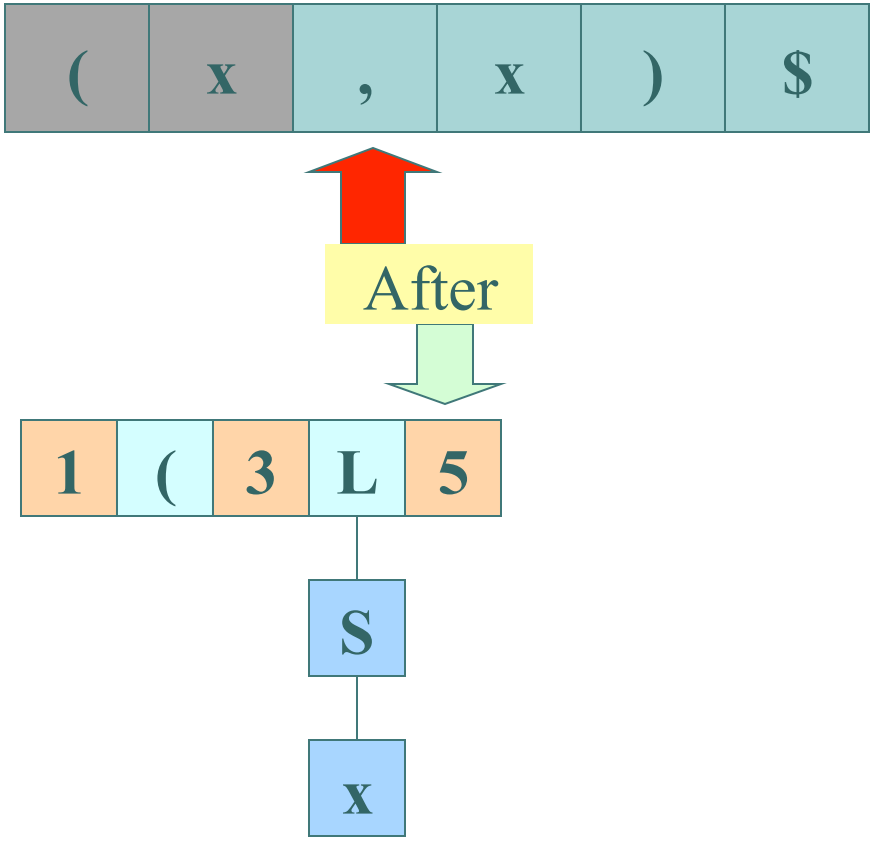
goto[3][L] ➔Goto(5)
- Push L onto stack
- Push 5 onto stack

LR parsing a small grammar (9)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[7][,] ➔ Reduce(3)
- Pop 7 and S
- Reduce using production (3)
  - L → S

goto[3][L] ➔Goto(5)
- Push L onto stack
- Push 5 onto stack

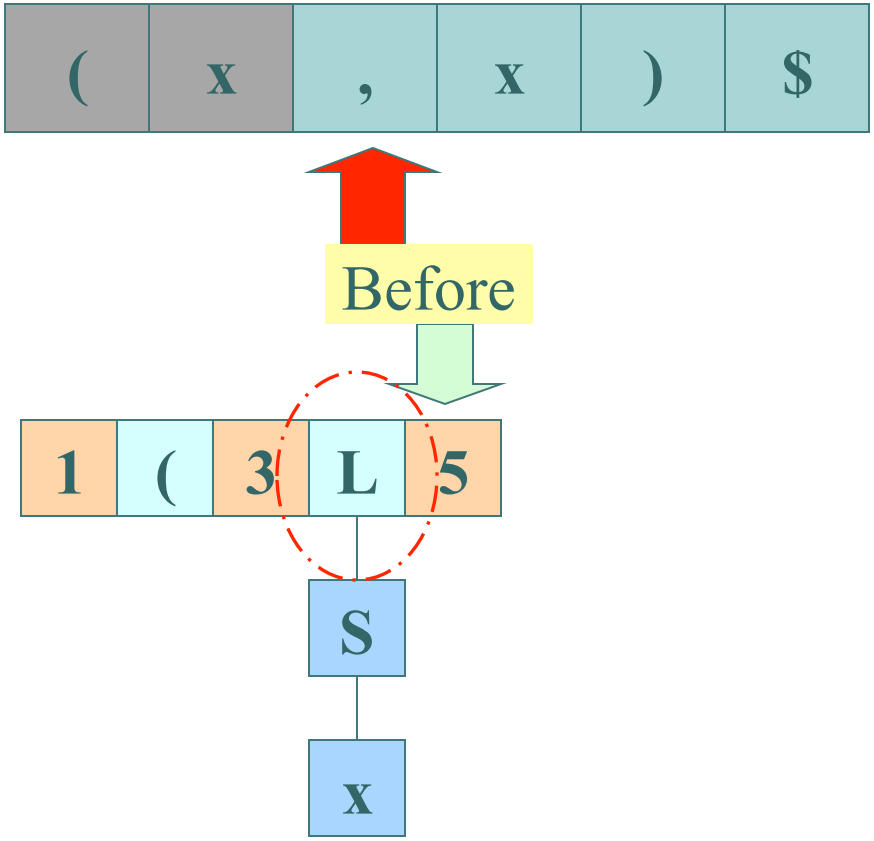


LR parsing a small grammar (10)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[5][,] ➔ Shift(8)

- Advance input one token
- Push ,
- Push 8

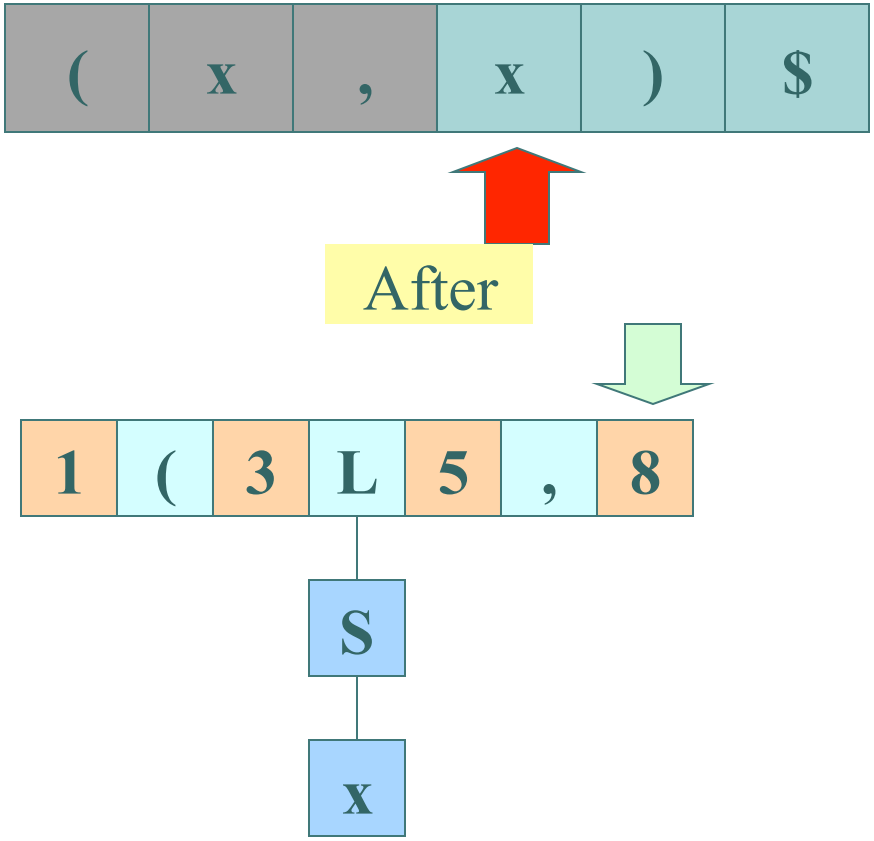


LR parsing a small grammar (11)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[5][,] ➔ Shift(8)

- Advance input one token
- Push ,
- Push 8

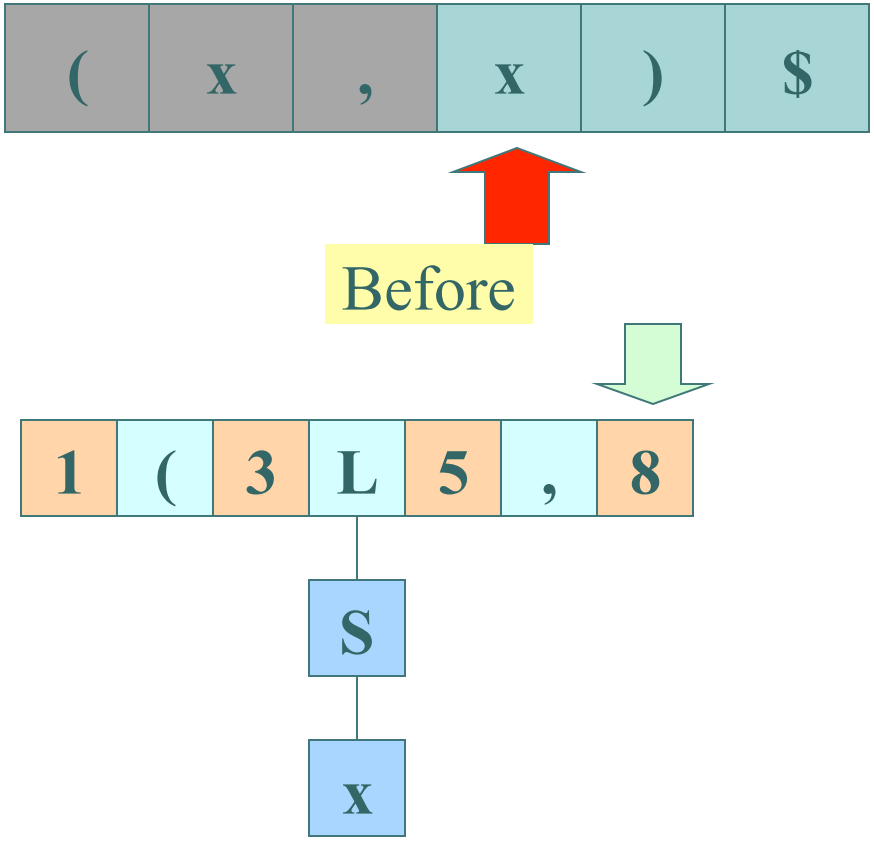


LR parsing a small grammar (12)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[8][x] ➔ Shift(2)

- Advance input one token
- Push x
- Push 2

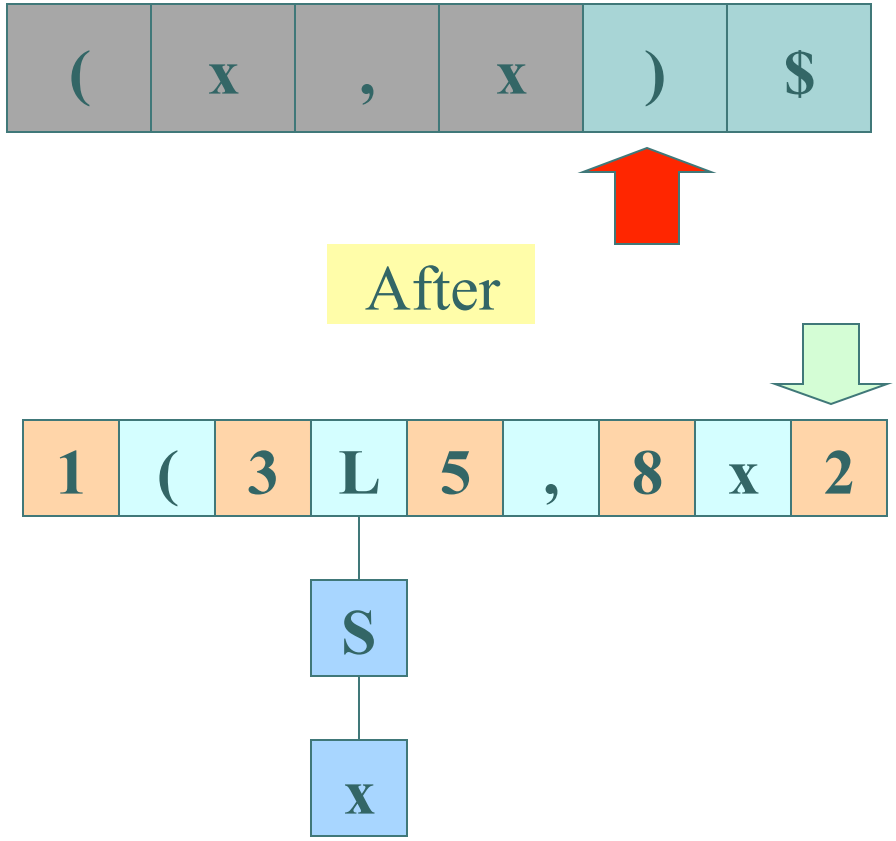


LR parsing a small grammar (13)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

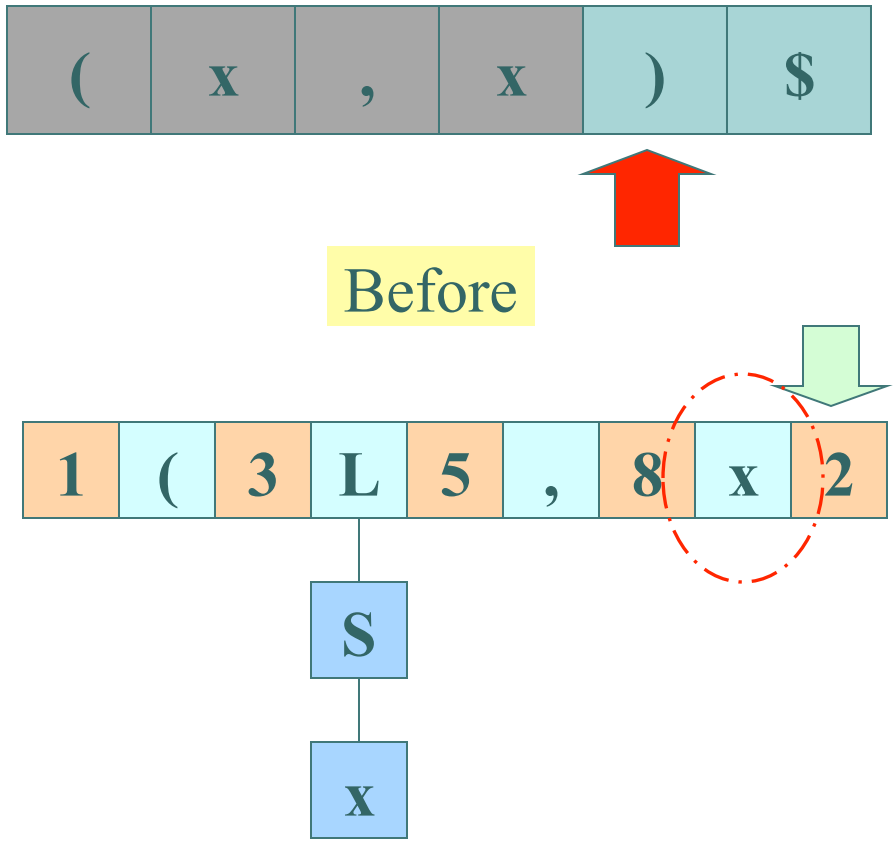
action[8][x] ➔ Shift(2)

- Advance input one token
- Push x
- Push 2



LR parsing a small grammar (14)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



action[2][)] ➔ Reduce(2)
- Pop 2 and x
- Reduce using production (2)
  - S → x

goto[8][S] ➔Goto(9)
- Push S onto stack
- Push 9 onto stack

LR parsing a small grammar (15)

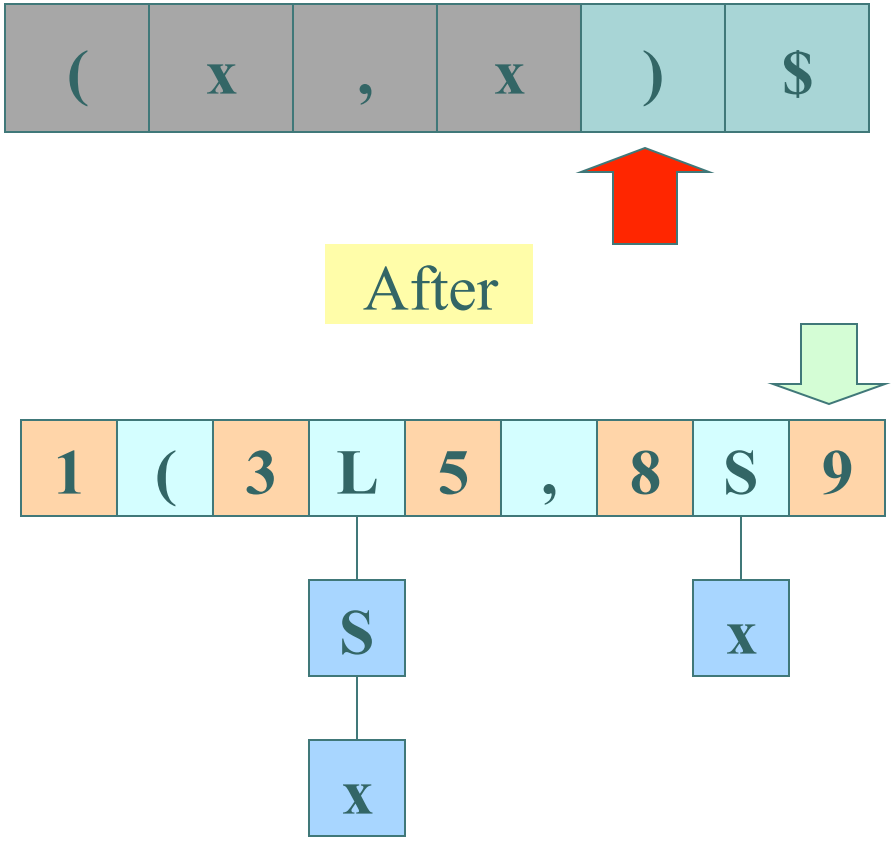
(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[2][)] ➔ Reduce(2)

- Pop 2 and x
- Reduce using production (2)
  - S → x

goto[8][S] ➔Goto(9)

- Push S onto stack
- Push 9 onto stack

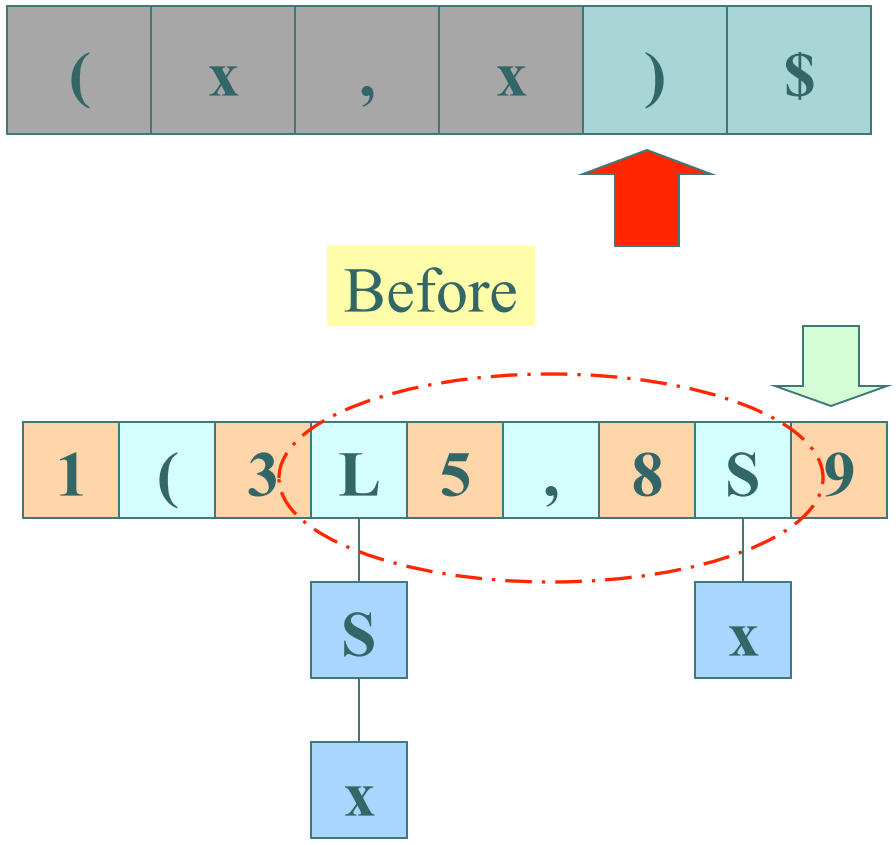


LR parsing a small grammar (16)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[9][)] ➔ Reduce(4)
- Pop 9 through to L
- Reduce using production (4)
  - L → L , S

goto[3][L] ➔Goto(5)
- Push L onto stack
- Push 5 onto stack

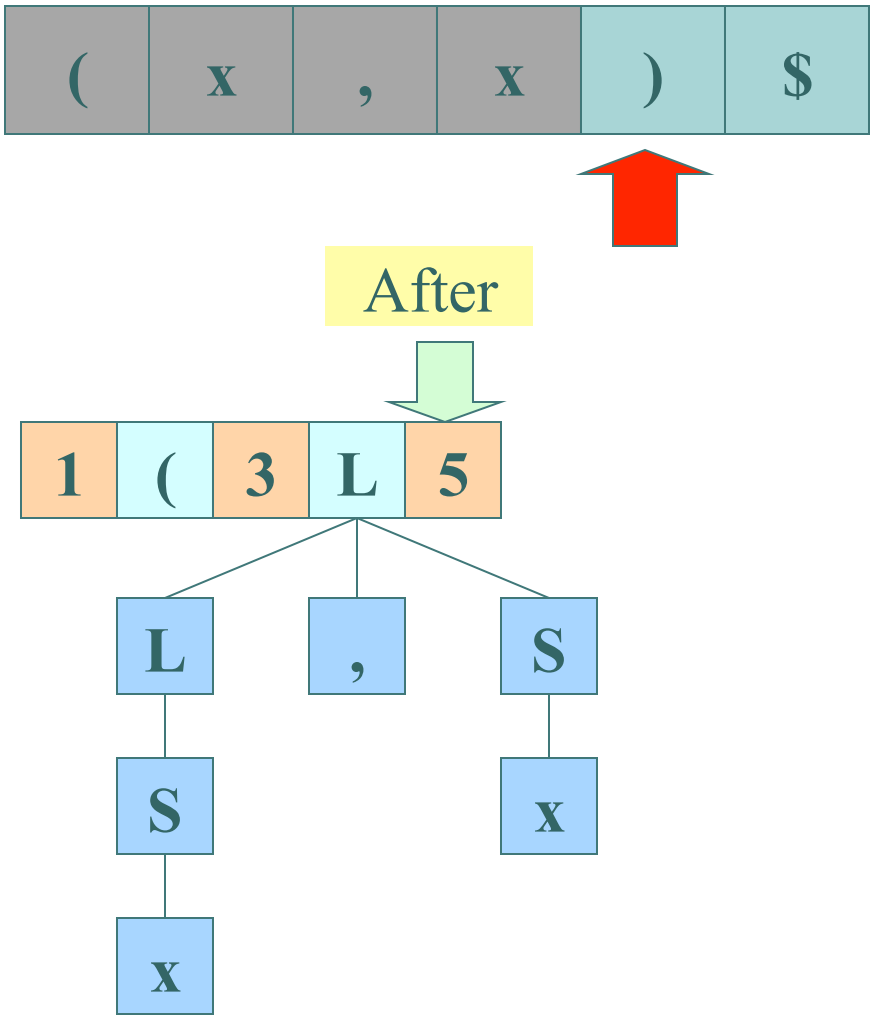


LR parsing a small grammar (17)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

action[9][)] ➔ Reduce(4)
- Pop 9 through to L
- Reduce using production (4)
  - L → L , S

goto[3][L] ➔Goto(5)
- Push L onto stack
- Push 5 onto stack

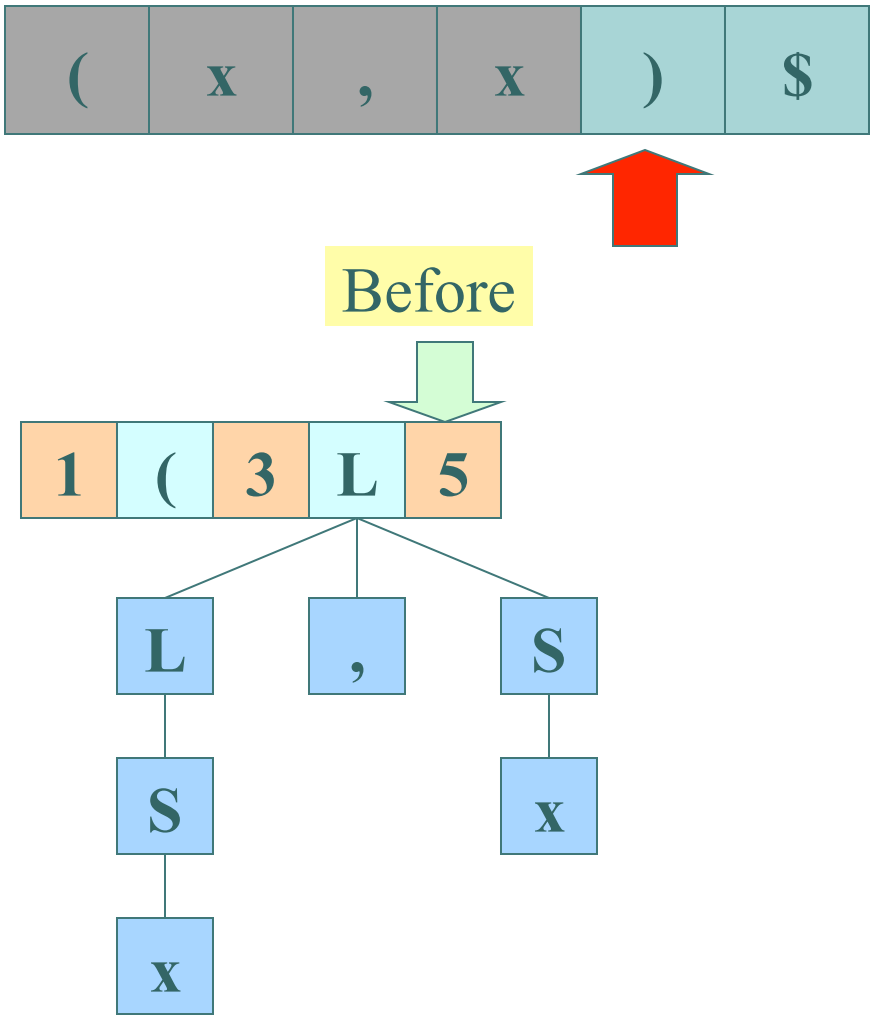


LR parsing a small grammar (18)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

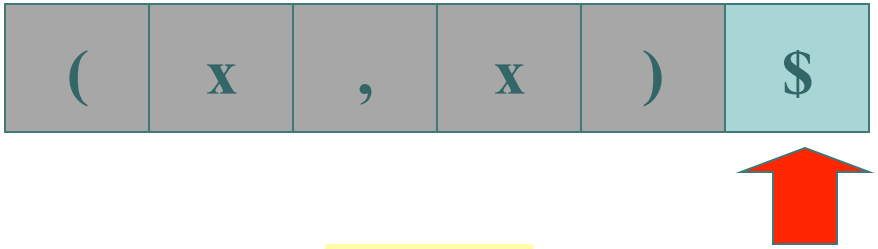
action[5][)] ➔ Shift(6)

- Advance input one token
- Push)
- Push 6



LR parsing a small grammar (19)

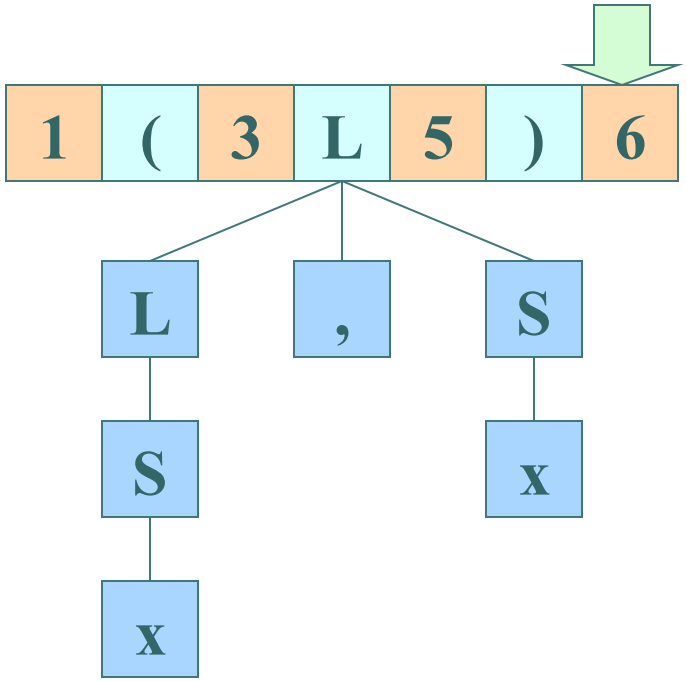
(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



action[5][)] ➔ Shift(6)

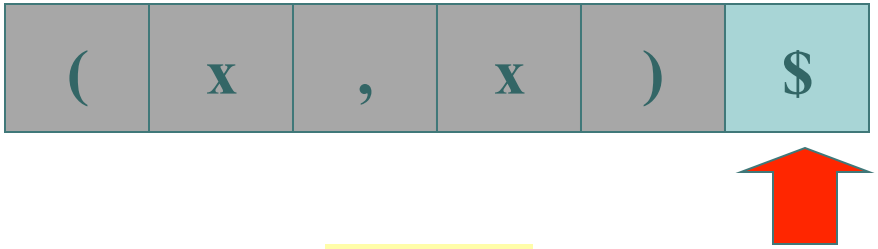
- Advance input one token
- Push)
- Push 6

After



LR parsing a small grammar (20)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

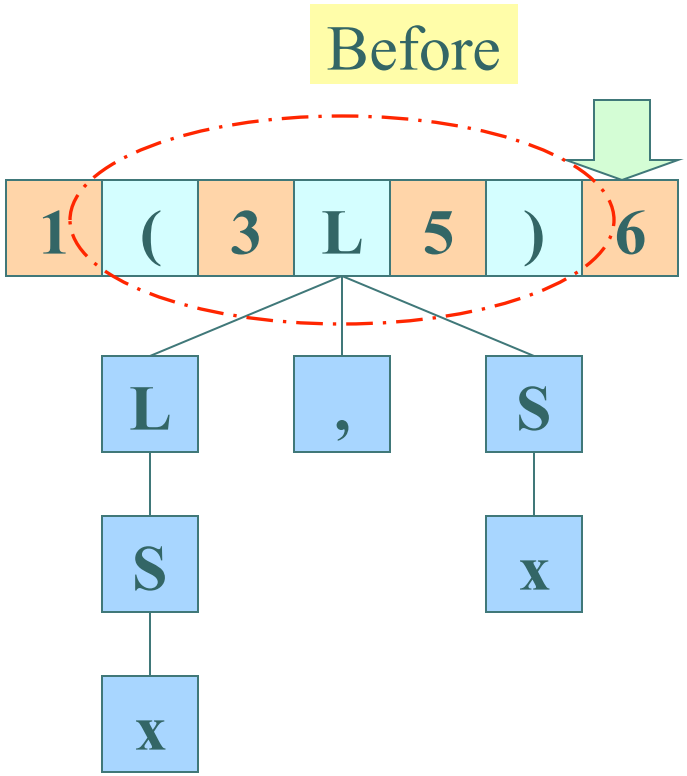


action[6][$] ➔ Reduce(1)

- Pop 6 through to (
- Reduce using production (1)
  - S → (L)

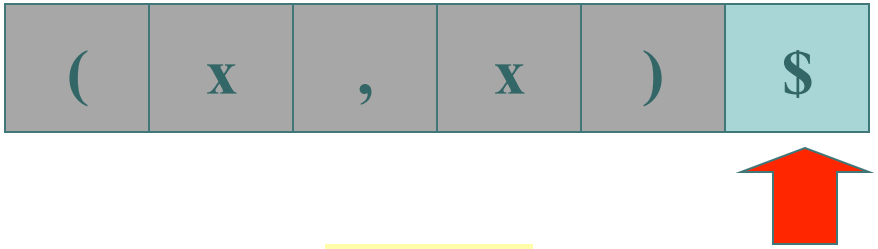
goto[1][S] ➔Goto(4)

- Push S onto stack
- Push 4 onto stack



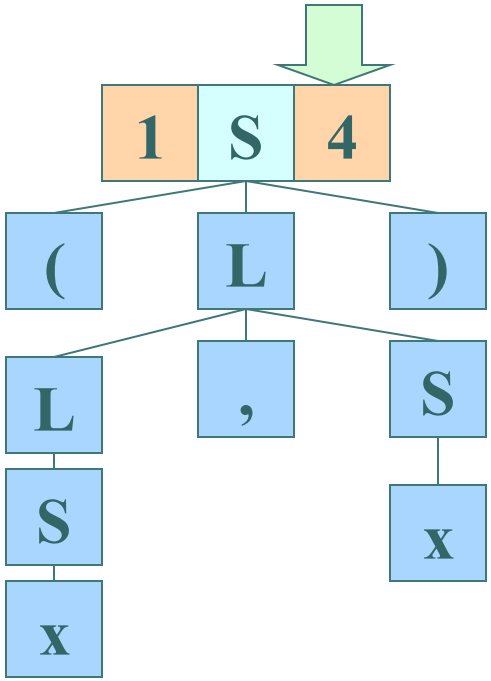
LR parsing a small grammar (21)

(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



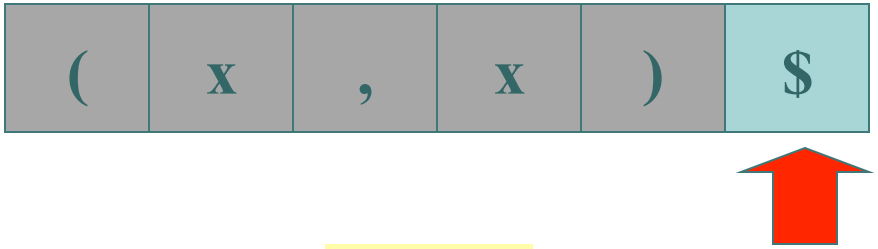
action[6][$] ➔ Reduce(1)
- Pop 6 through to (
- Reduce using production (1)
  - S → (L)

goto[1][S] ➔Goto(4)
- Push S onto stack
- Push 4 onto stack



LR parsing a small grammar (22)

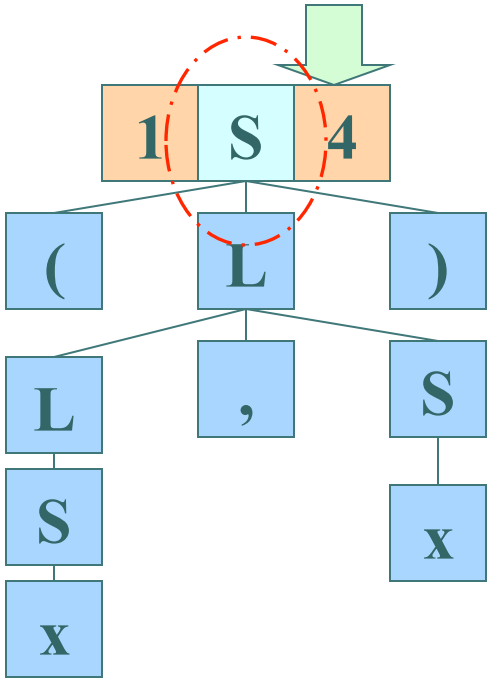
(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S



action[4][$] ➔ Accept

- S is what we were looking for!

Before



LR parsing a small grammar (23)

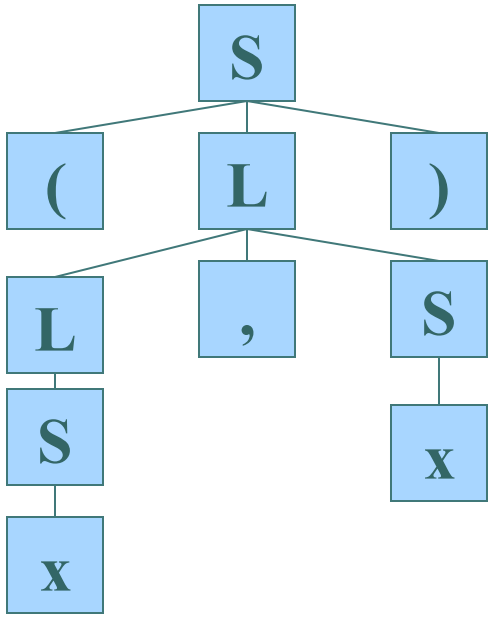
(0) S' → S $ -- $ is EOF
(1) S → (L)
(2) S → x
(3) L → S
(4) L → L , S

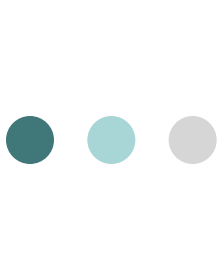
action[4][$] ➔ Accept

- This is what we were looking for!

The magic is in the LR engine tables

After





Next Time

- Generating LR parser tables