Bottom Up (**Shift Reduce**) Parsing

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root
- A bottom-up parser tries to find the **right-most derivation** of the given input in the reverse order.

$S \Rightarrow \ldots \Rightarrow \omega$ (the right-most derivation of ω)

$\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)

Bottom Up Parsing

- LR Parsing
  - Also called “Shift-Reduce Parsing”

- Find a rightmost derivation
- Finds it in reverse order

- LR Grammars
  - Can be parsed with an LR Parser
- LR Languages
  - Can be described with LR Grammar
  - Can be parsed with an LR Parser

LR Parsing Techniques

- **LR Parsing**
  - Most General Approach
- **SLR**
  - Simpler algorithm, but not as general
- **LALR**
  - More complex, but saves space

LL vs. LR

- LR (shift reduce) is more powerful than LL (predictive parsing)

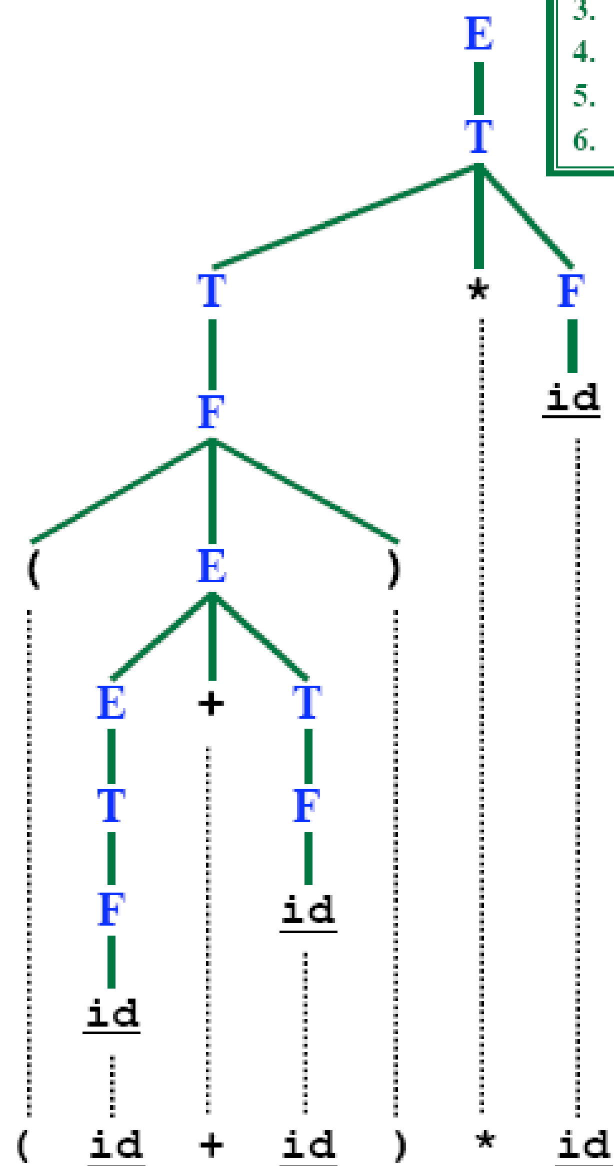
- Can detect a syntactic error as soon as possible.

- LR is difficult to do by hand (unlike LL) and
- LL accepts a much smaller set of grammars.

Rightmost Derivation

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

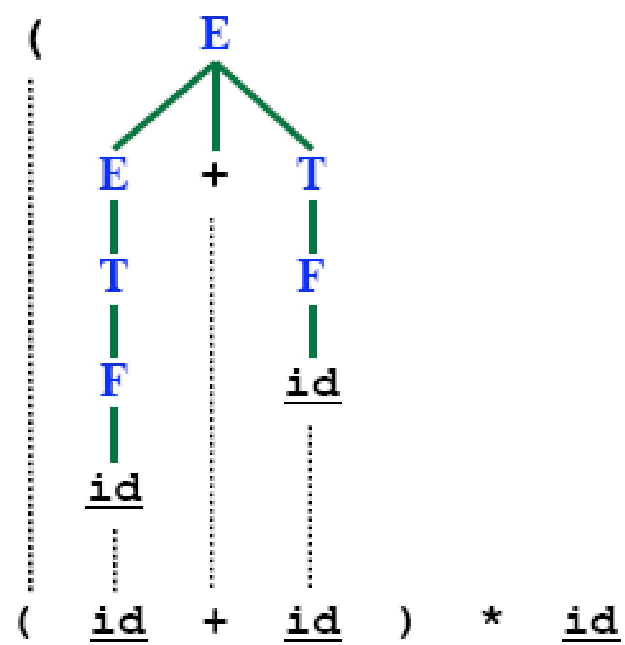
Rules Used:	*Right-Sentential Forms:*
	E
E → T	T
T → T * F	T * F
F → id	T * id
T → F	F * id
F → (E)	(E) * id
E → E + T	(E + T) * id
T → F	(E + F) * id
F → id	(E + id) * id
E → T	(T + id) * id
T → F	(F + id) * id
F → id	(id + id) * id



Rightmost Derivation In reverse

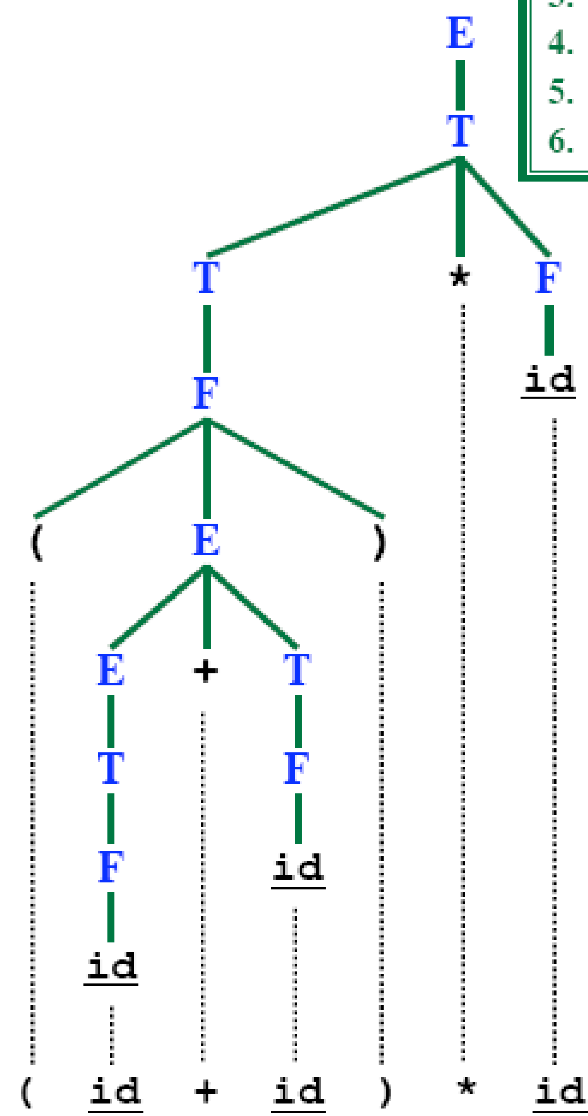
Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id
F → id	T * F
T → T * F	T
E → T	E



LR parsing corresponds to rightmost derivation in reverse

Reduction

- A reduction step replaces a specific substring (matching the body of a production)

(id + id) * id

(F + id) * id

(T + id) * id

(E + id) * id

(E + F) * id

(E + T) * id

(E) * id

F * id

T * id

T * F

T

E

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

- Reduction is the opposite of derivation
- Bottom up parsing is a process of reducing a string ω to the start symbol S of the grammar

Shift-Reduce Parsing

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
- data structures: input-string and stack
- Operations
  - 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.
  - Accept: Announce successful completion of parsing
  - Error: Discover a syntax error and call error recovery

Shift Reduce Parsing Example

S → a T R e
T → T b c | b
R → d

Remaining input: abbcde

Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ a **b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bbcde

→ Shift a

a

Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ a b b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bcde

→ Shift a, Shift b

a b

Rightmost derivation:

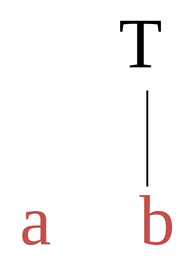
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ a b b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bcde

→ Shift a, Shift b
→ Reduce T → b



Rightmost derivation:

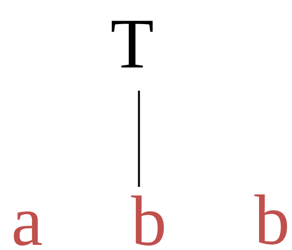
S → a T R e
→ a T **d** e
→ <u>a **T**</u> **b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: cde

→ Shift a, Shift b
→ Reduce T → b
→ Shift b



Rightmost derivation:

S → a T R e
→ a T **d** e
→ <u>a **T** **b** </u>c d e
→ **a** **b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: de

→ Shift a, Shift b
→ Reduce T → b
→ Shift b, Shift c

T

a b b c

Rightmost derivation:

S → a T R e
→ a T **d** e
→ <u>a **T** **b** **c**</u> d e
→ **a** **b** b c d e

Shift Reduce Parsing

S → a T R e

T → T b c | b

R → d

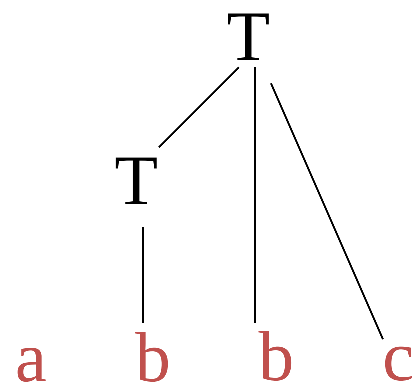
Remaining input: de

→ Shift a, Shift b

→ Reduce T → b

→ Shift b, Shift c

→ Reduce T → T b c



Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

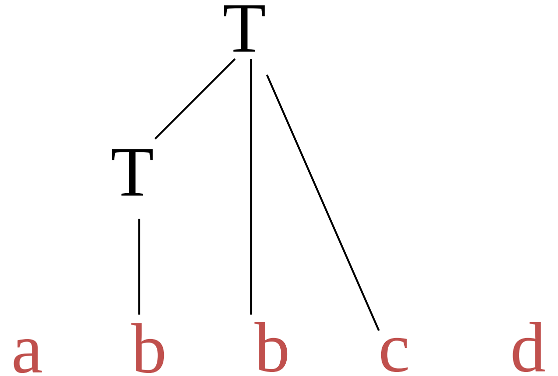
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: e

- → Shift a, Shift b
- → Reduce T → b
- → Shift b, Shift c
- → Reduce T → T b c
- → Shift d



Rightmost derivation:

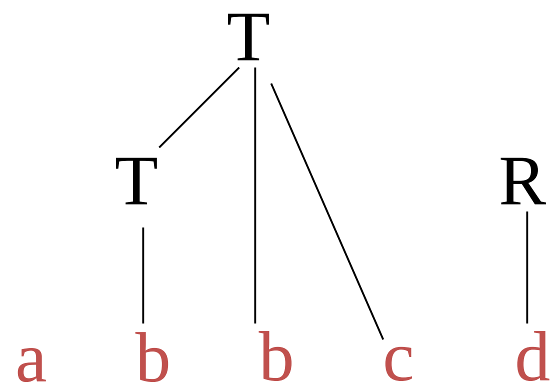
S → a T R e
→ <u>a T **d**</u> e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: e

→ Shift a, Shift b
→ Reduce T → b
→ Shift b, Shift c
→ Reduce T → T b c
→ Shift d
→ Reduce R → d



Rightmost derivation:

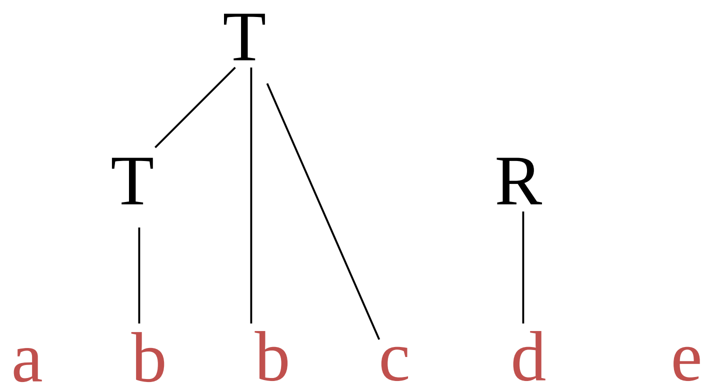
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input:

→ Shift a, Shift b
→ Reduce T → b
→ Shift b, Shift c
→ Reduce T → T b c
→ Shift d
→ Reduce R → d
→ Shift e



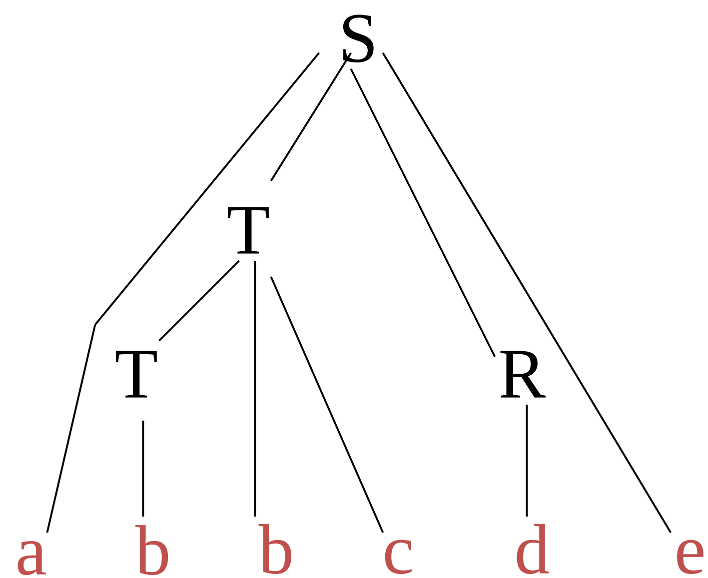
Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input:



Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

- → Shift a, Shift b
- → Reduce T → b
- → Shift b, Shift c
- → Reduce T → T b c
- → Shift d
- → Reduce R → d
- → Shift e
- → Reduce S → a T R e

Example Shift-Reduce Parsing

Consider the grammar:

Stack	Input	Action
$\$$	$id_1 + id_2\$$	shift
$\$id_1$	$+ id_2\$$	reduce 6
$\$F$	$+ id_2\$$	reduce 4
$\$T$	$+ id_2\$$	reduce 2
$\$E$	$+ id_2\$$	shift
$\$E +$	$id_2\$$	shift
$\$E + id_2$		reduce 6
$\$E + F$		reduce 4
$\$E + T$		reduce 1
$\$E$		accept

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \underline{id}$

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

left to right scanning   right-most derivation   k lookhead

- An ambiguous grammar can never be a LR grammar.

Shift-Reduce Conflict in Ambiguous Grammar

$stmt \rightarrow$ **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

STACK	**INPUT**
....**if** *expr* **then** *stmt*	**else** ....$

- We can't decide whether to shift or reduce?

Reduce-Reduce Conflict in Ambiguous Grammar

1. *stmt* → **id**(*parameter_list*)
2. *stmt* → *expr:=expr*
3. *parameter_list* → *parameter_list, parameter*
4. *parameter_list* → *parameter*
5. *parameter_list* → **id**
6. *expr* → **id**(*expr_list)*
7. *expr* → **id**
8. *expr_list* → *expr_list, expr*
9. *expr_list* → *expr*

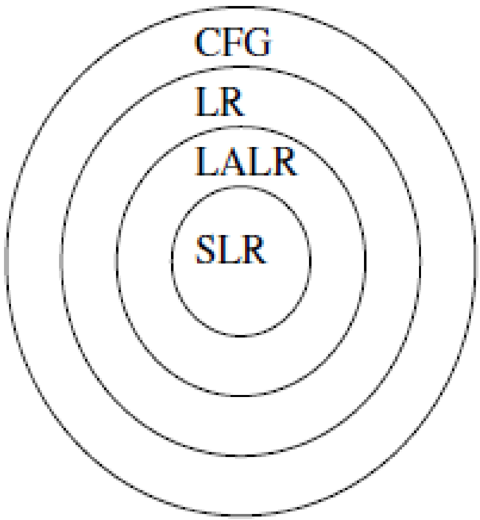
STACK	**INPUT**
….id (id	**, id) …$**

- We can't decide which production will be used to reduce **id**?

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

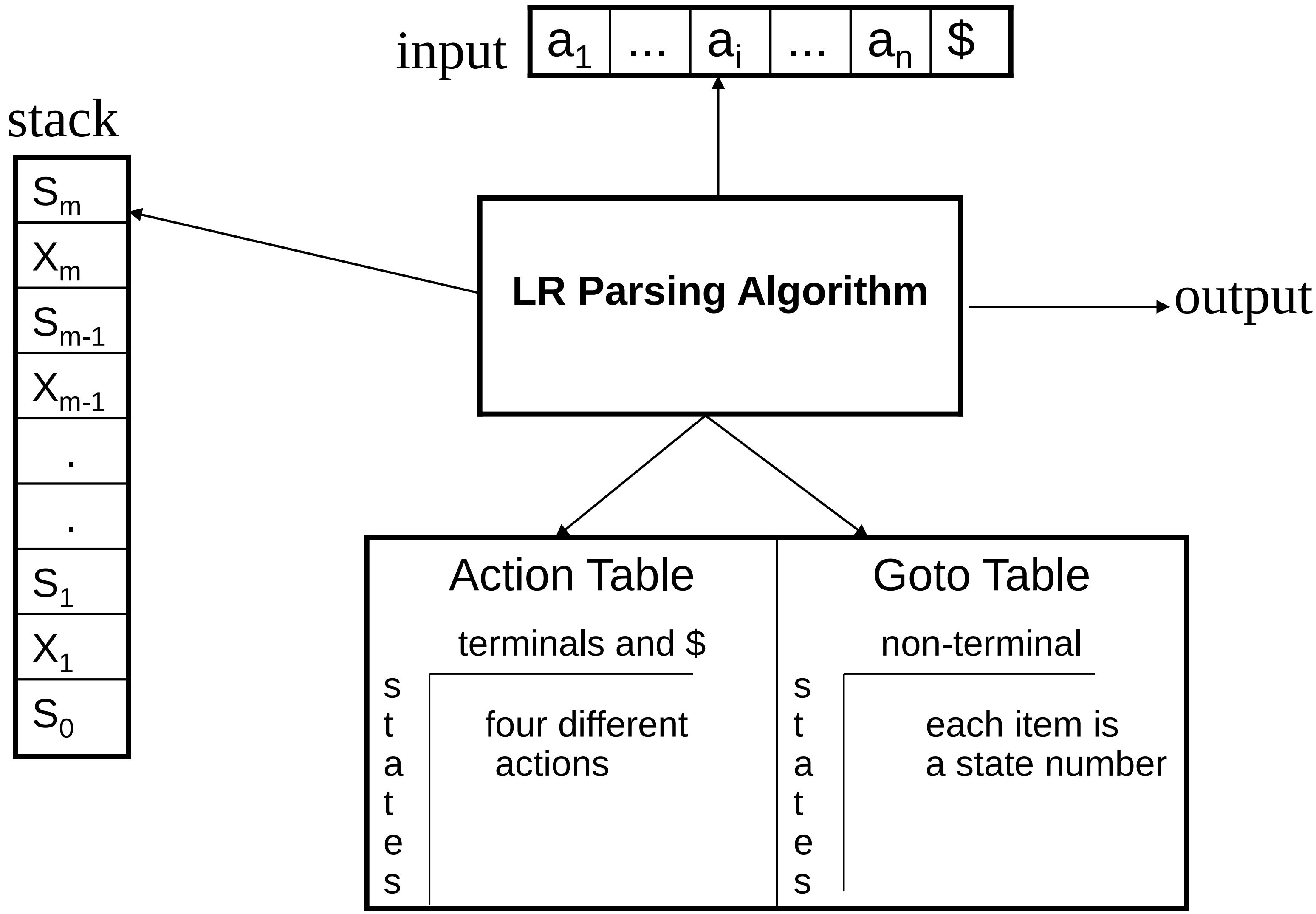
LR parsing is attractive because:

- 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.
- LR parsers can be constructed to recognize virtually all programming language constructs for which CFG grammars can be written

Drawback of LR method:

- Too much work to construct LR parser by hand
  - Fortunately tools (LR parsers generators) are available

LR Parsing Algorithm



Bottom-Up Parsing: LR(0) Table Construction

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: (four different possibility)

$$A \rightarrow \bullet aBb$$
$$A \rightarrow a \bullet Bb$$
$$A \rightarrow aB \bullet b$$
$$A \rightarrow aBb \bullet$$

- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
  - States represent sets of “items”
- LR parser makes shift-reduce decision by maintaining states to keep track of where we are in a parsing process

Constructing SLR Parsing Tables – LR(0) Item

- An item indicates how much of a production we have seen at a given point in the parsing process
- For Example the item $A \rightarrow X \bullet YZ$
  - We have already seen on the input a string derivable from X
  - We hope to see a string derivable from YZ
- For Example the item $A \rightarrow \bullet XYZ$
  - We hope to see a string derivable from XYZ
- For Example the item $A \rightarrow XYZ \bullet$
  - We have already seen on the input a string derivable from XYZ
  - It is possibly time to reduce XYZ to A

- Special Case:

  Rule: $A \rightarrow \varepsilon$ yields only one item

  $A \rightarrow \bullet$

Constructing SLR Parsing Tables

- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- Canonical LR(0) collection provides the basis of constructing a DFA called **LR(0) automaton**
  - This DFA is used to make parsing decisions
- Each state of LR(0) automaton represents a set of items in the canonical LR(0) collection
- To construct the canonical LR(0) collection for a grammar
  - Augmented Grammar
  - CLOSURE function
  - GOTO function

Grammar Augmentation

Augment the grammar by adding...

- A new start symbol, S'
- A new rule $S' \rightarrow S$

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ id

"Goal"

0. $S' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ id

Our goal is to find an S', followed by \$.

$$S' \rightarrow \bullet E , \$$$

Whenever we are about to reduce using rule 0...

Accept! Parse is finished!

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from ***I*** by the two rules:

1. Initially, every LR(0) item in ***I*** is added to ***closure(I)***.
2. If $A \rightarrow \alpha.B\beta$ is in ***closure(I)*** and $B \rightarrow \gamma$ is a production rule of G;

   then $B \rightarrow .\gamma$ will be in the ***closure(I)***.

   We will apply this rule until no more new LR(0) items can be added to ***closure(I)***.

The Closure Operation -- Example

$E' \to E$
$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id$

$$
\text{closure}(\{E' \to \bullet E\}) = \{\; E' \to \bullet E,\;\; E \to \bullet E+T,\;\; E \to \bullet T,\;\; T \to \bullet T*F,\;\; T \to \bullet F,\;\; F \to \bullet(E),\;\; F \to \bullet id \;\}
$$

kernel items

GOTO Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then GOTO(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I then every item in **closure({$A \rightarrow \alpha X \bullet \beta$})** will be in GOTO(I,X).

Example:

I ={ $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$,
$T \rightarrow \bullet T^*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

GOTO(I,E) = { $E' \rightarrow E \bullet$, $E \rightarrow E \bullet +T$ }

GOTO(I,T) = { $E \rightarrow T \bullet$, $T \rightarrow T \bullet {}^*F$ }

GOTO(I,F) = { $T \rightarrow F \bullet$ }

GOTO(I,() = { $F \rightarrow (\bullet E)$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T^*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

GOTO(I,id) = { $F \rightarrow id \bullet$ }

LR(0) Automation

- Start with **start rule** & compute **initial state with closure**
- Pick one of the items from the states and **move "." to the right one symbol** (as if you 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 the new states or new edges in the automation**.

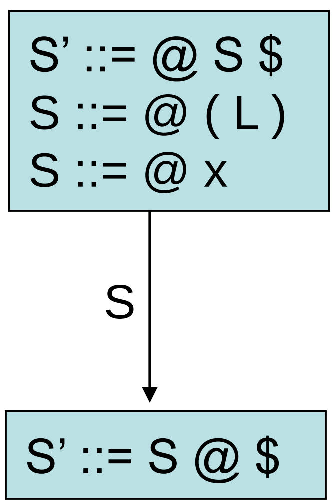
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

S’ ::= @ S $
S ::= @ (L)
S ::= @ x

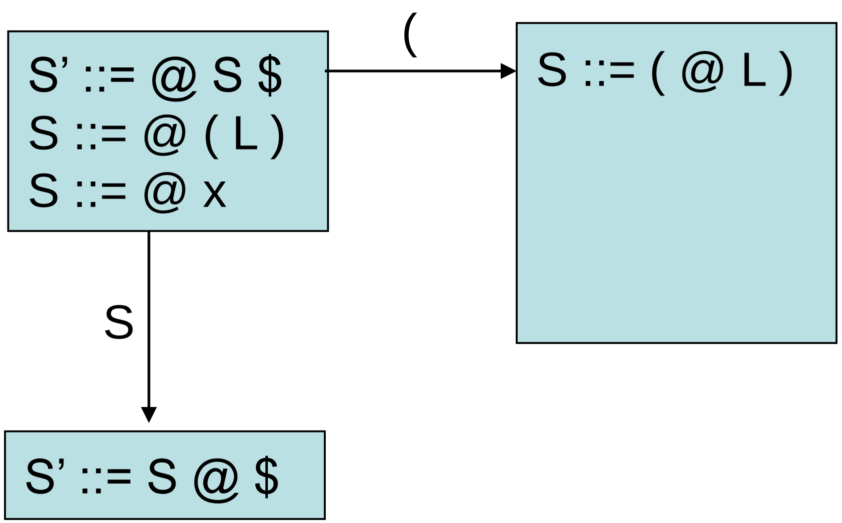
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



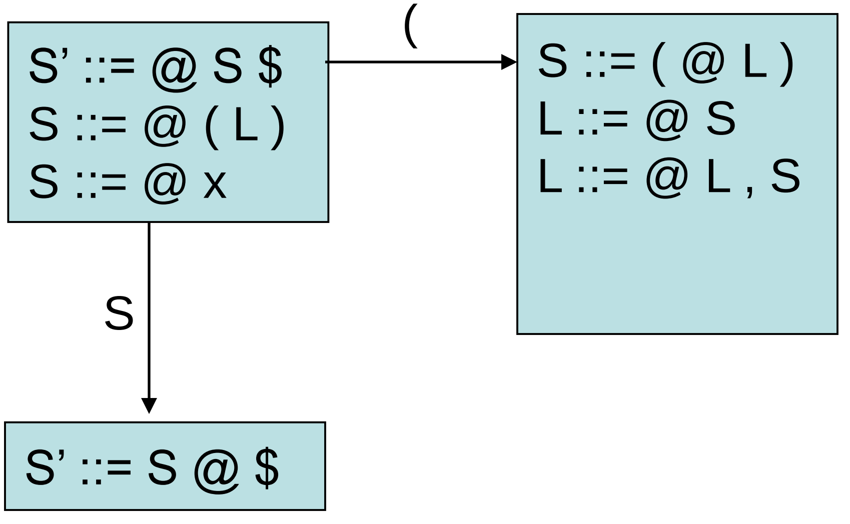
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



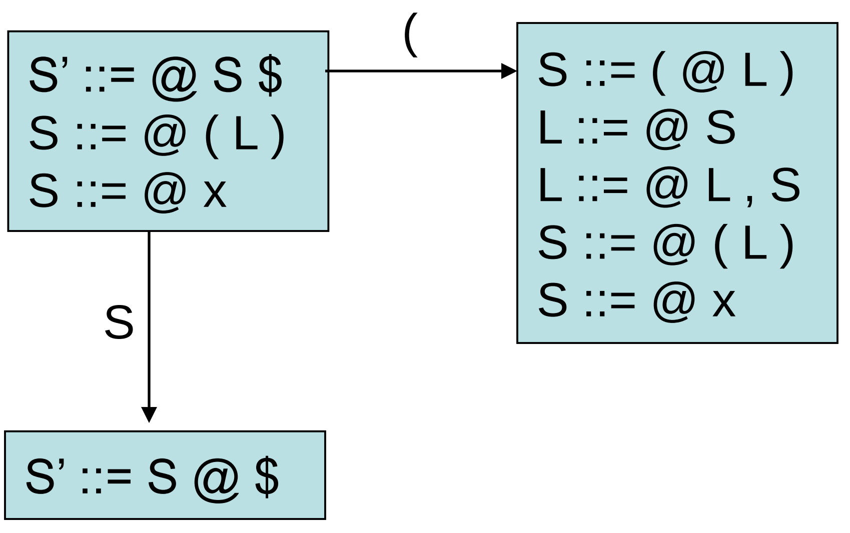
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



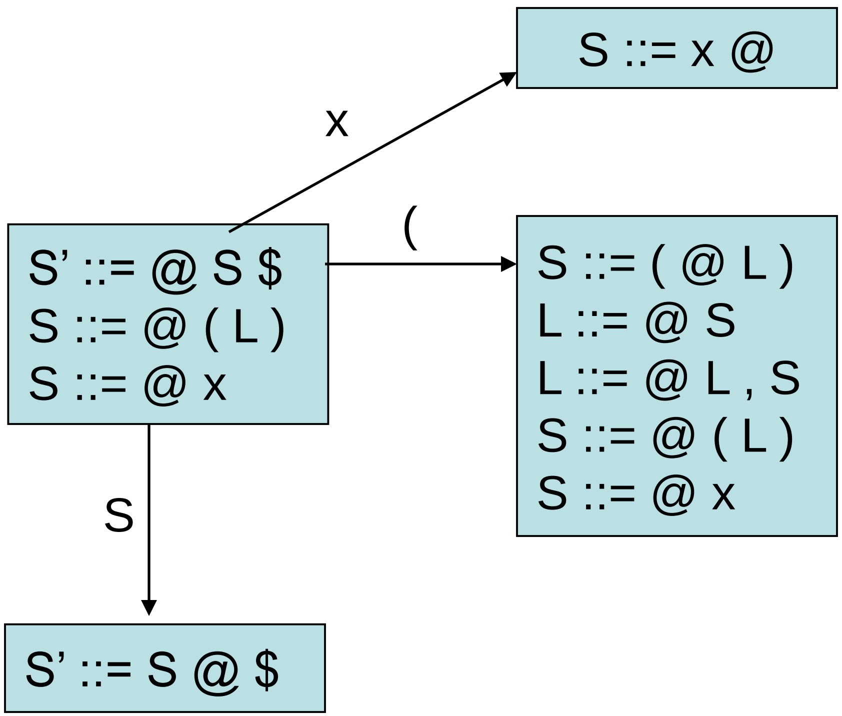
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



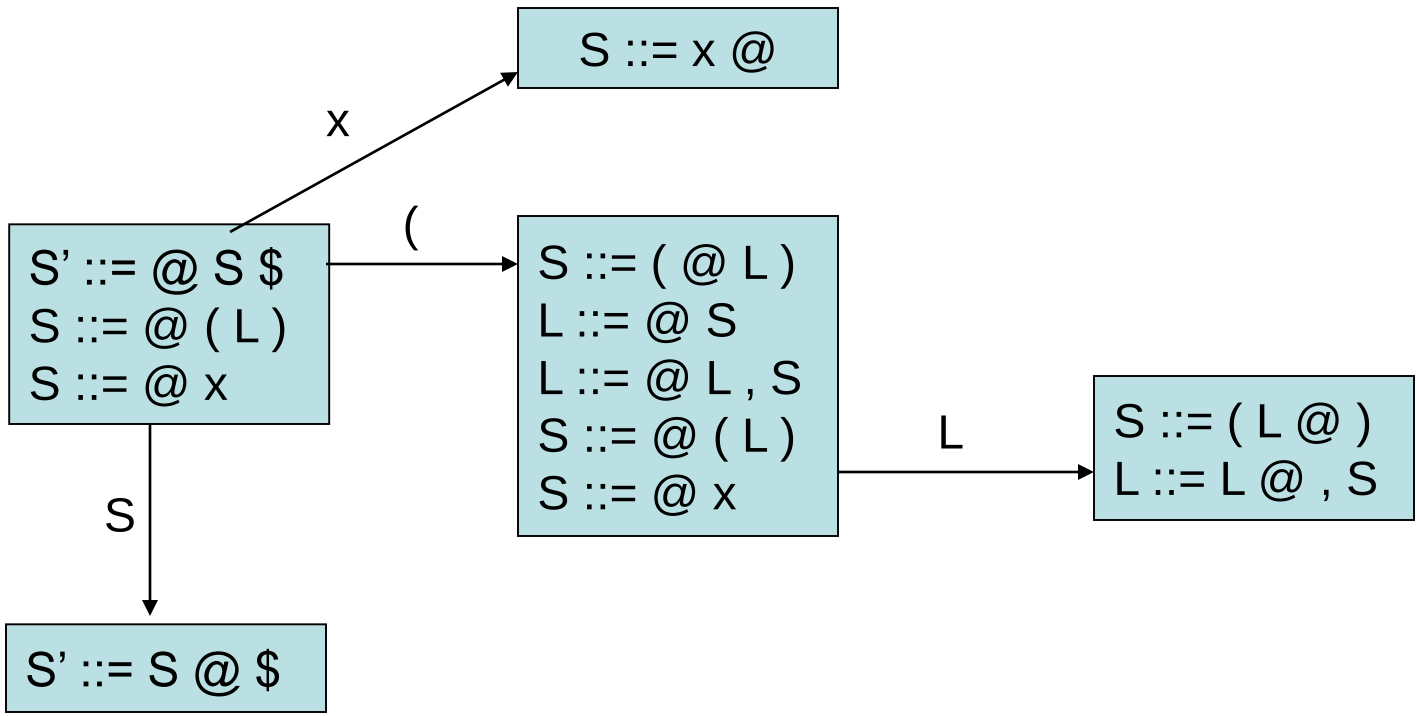
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



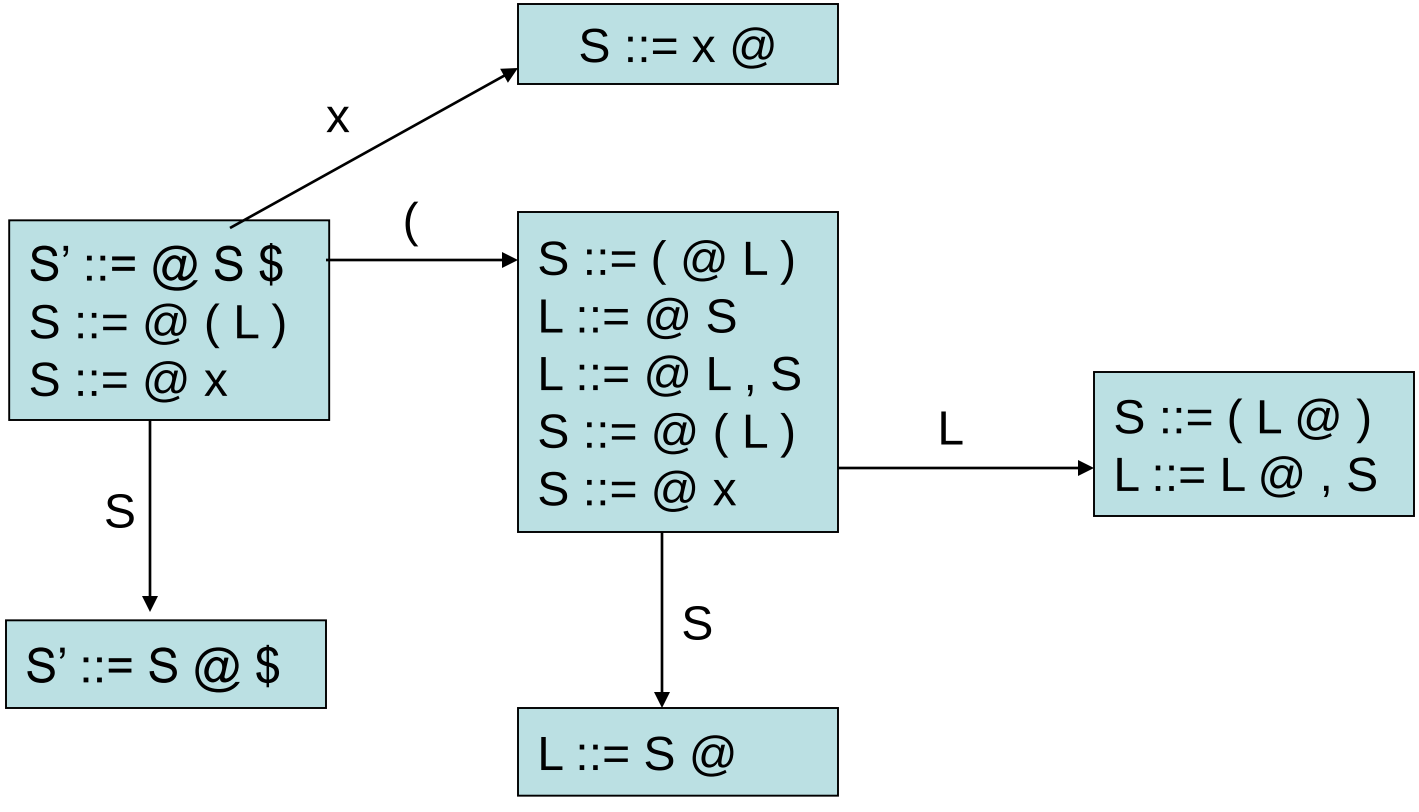
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



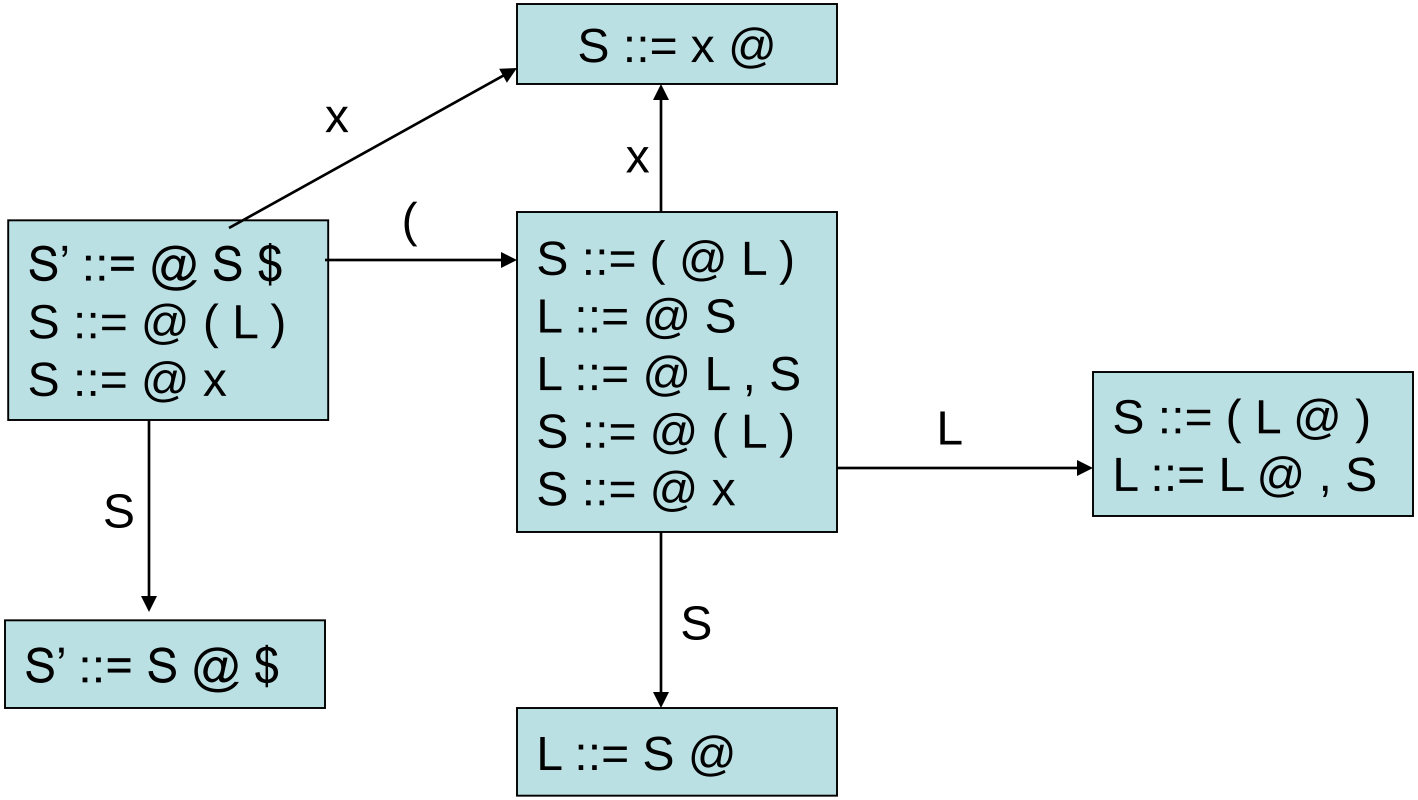
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



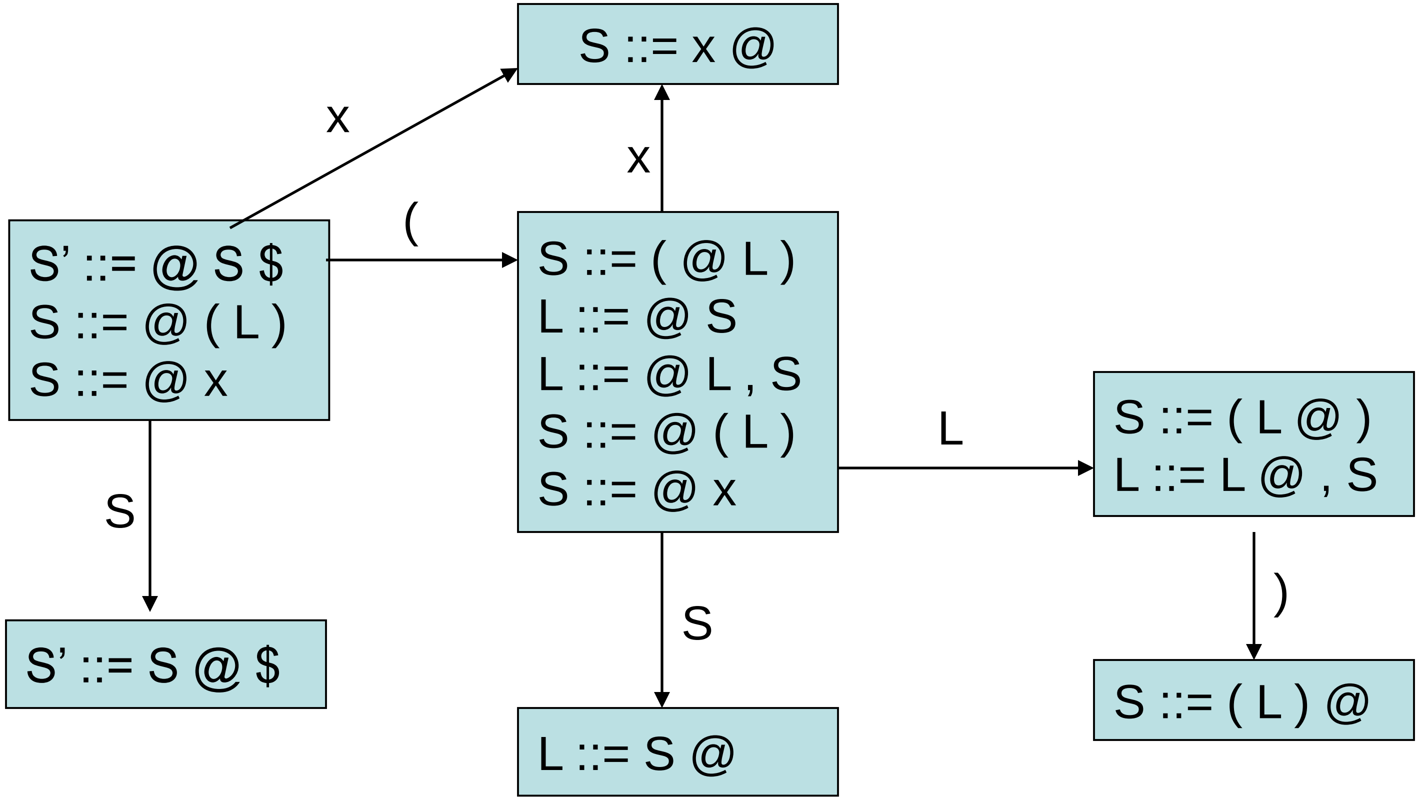
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



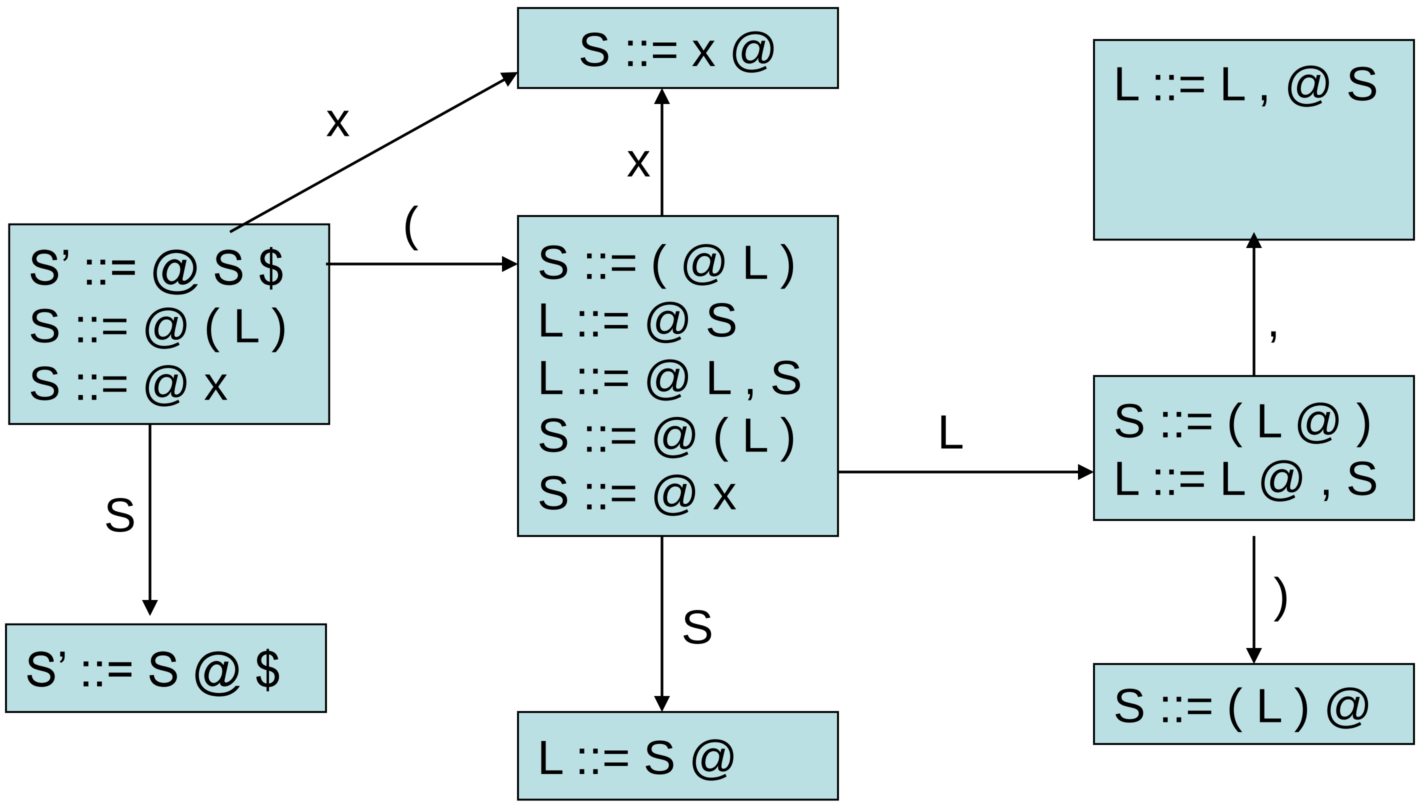
Grammar:

0. S' ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



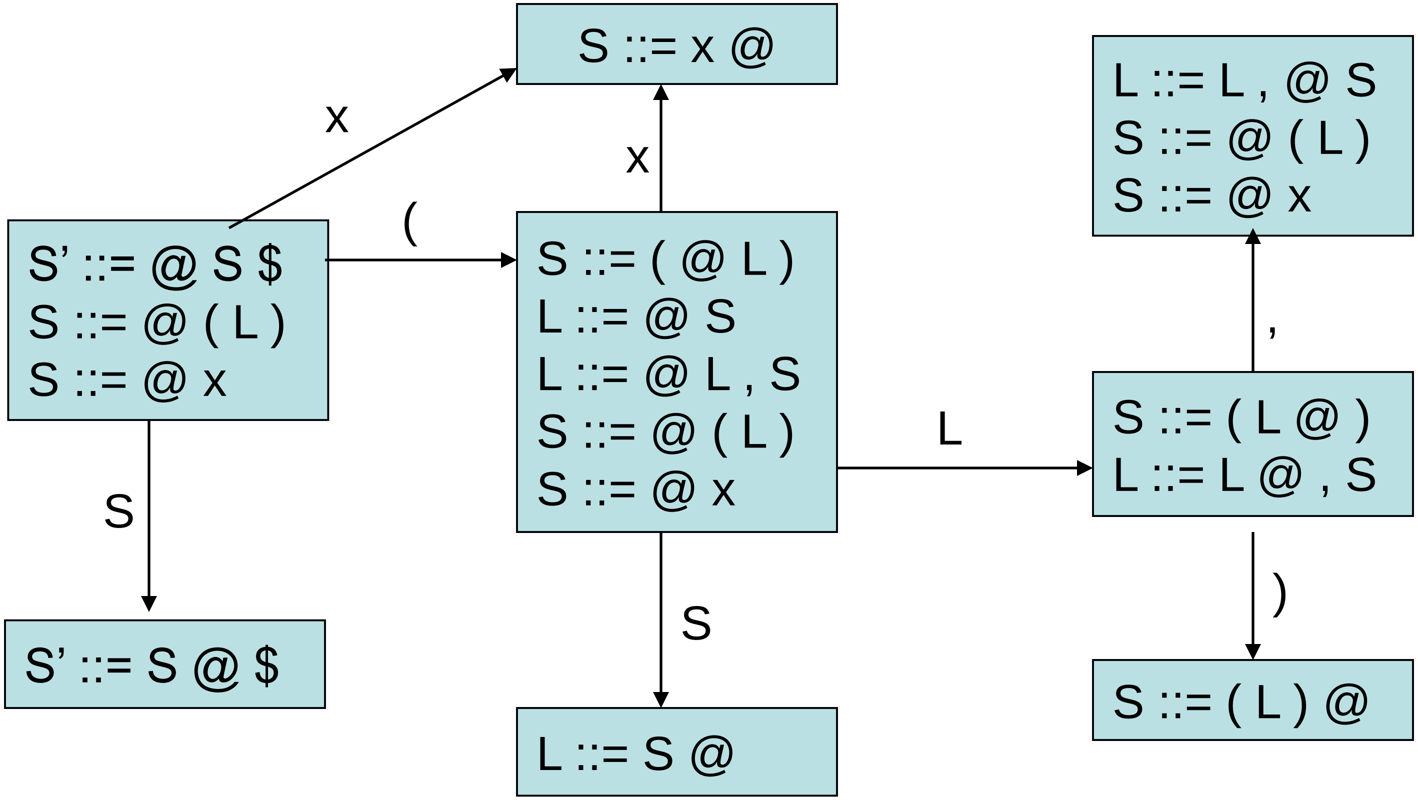
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



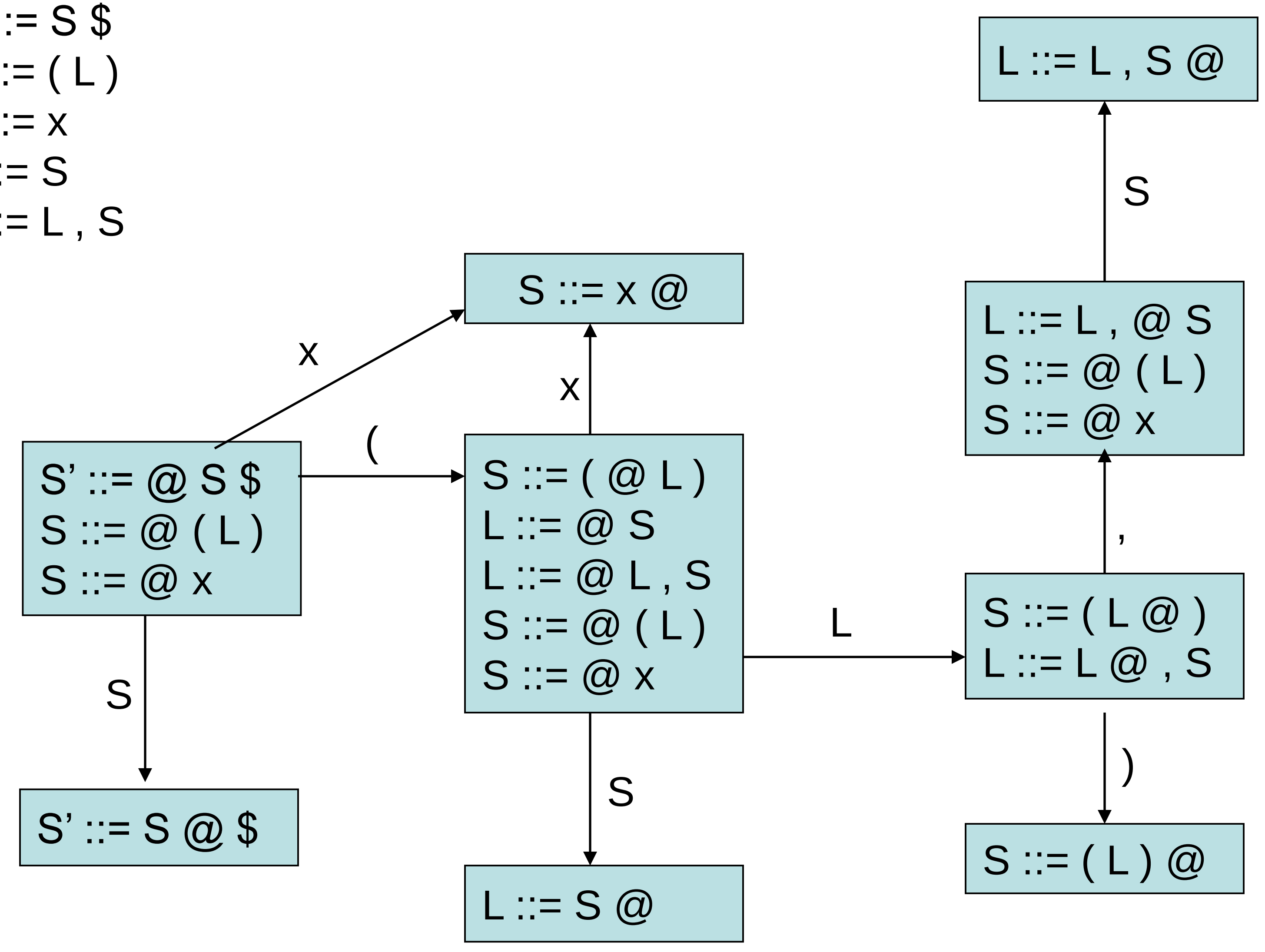
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



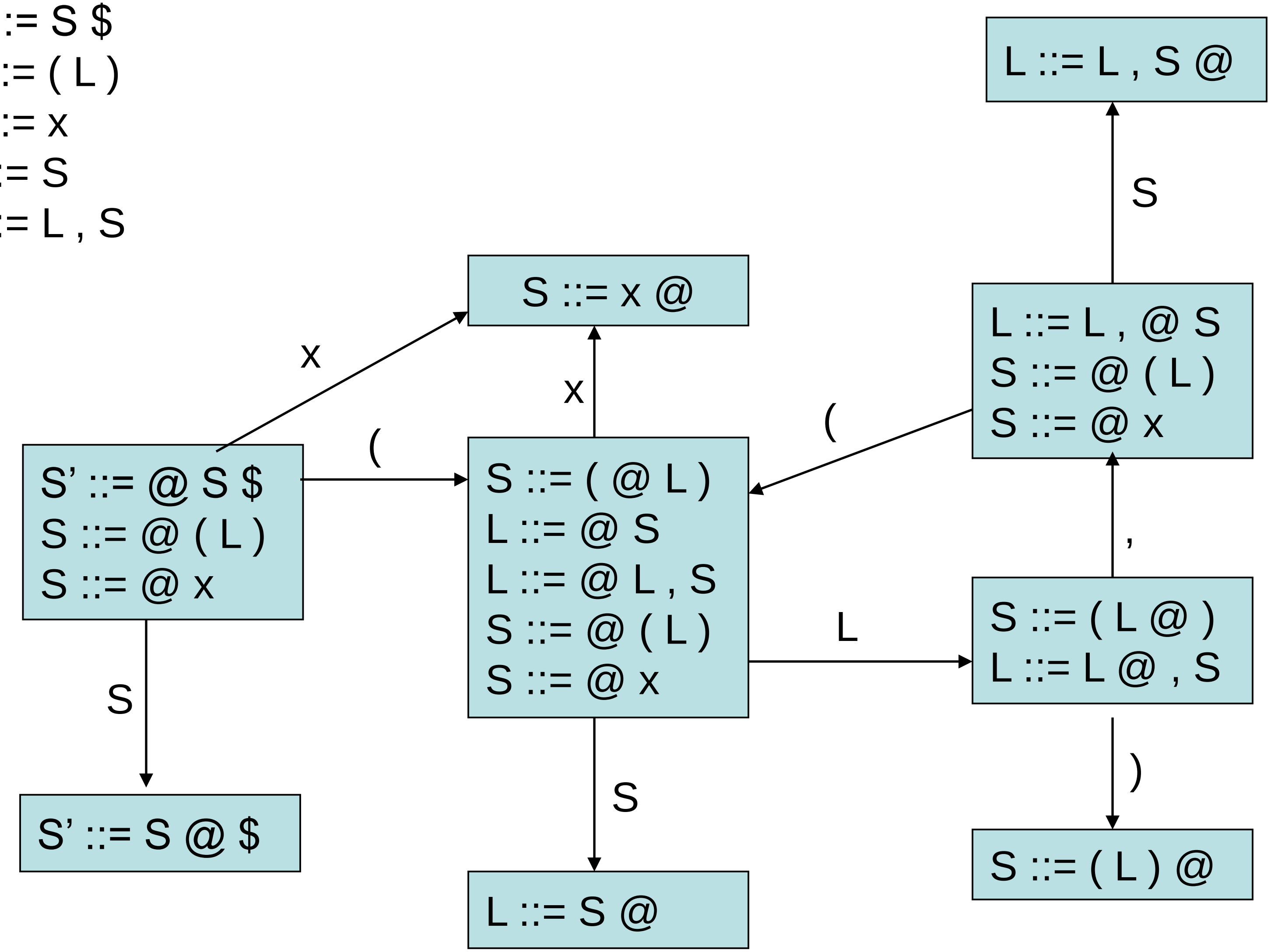
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



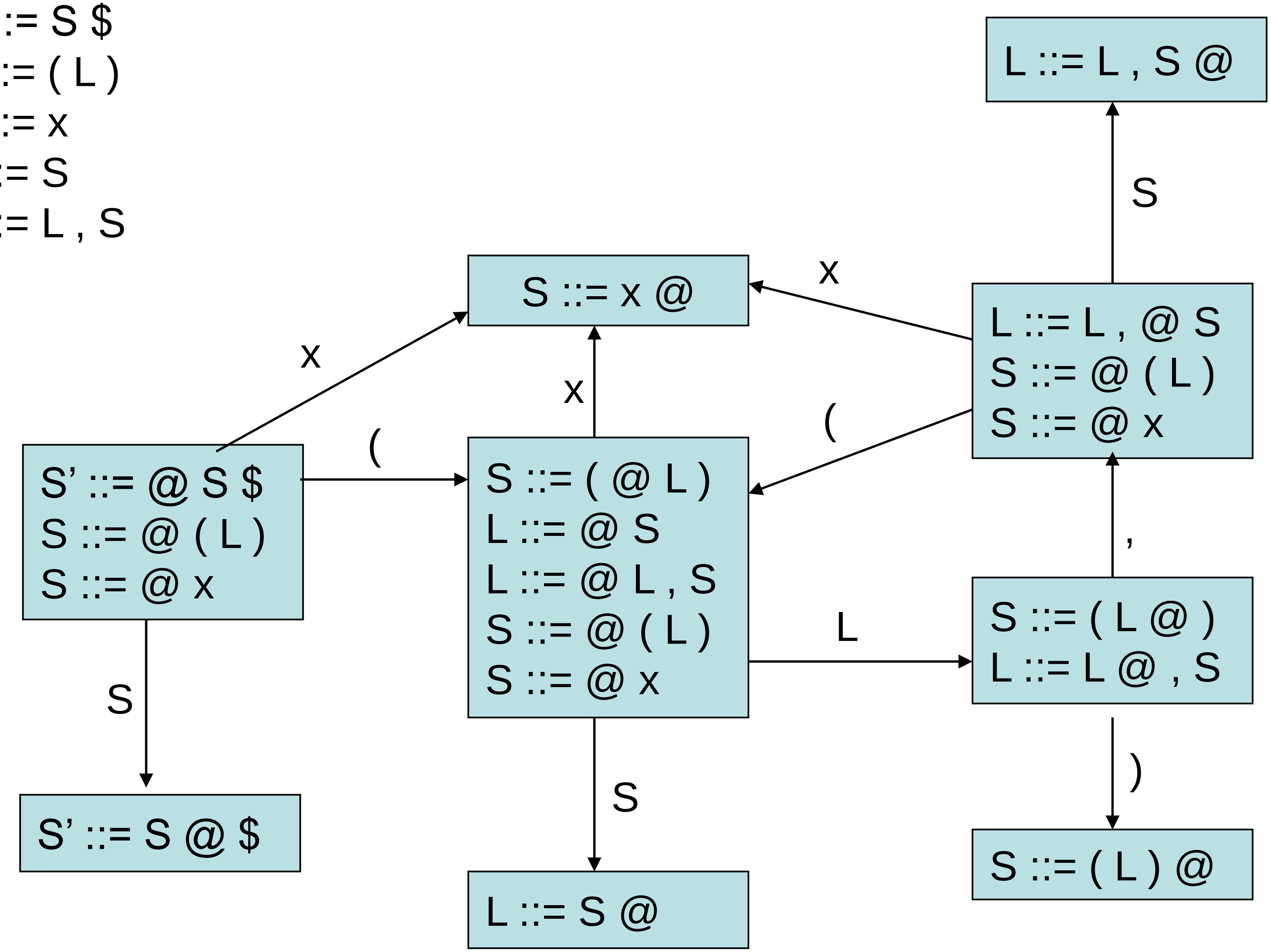
Grammar:

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



Grammar:

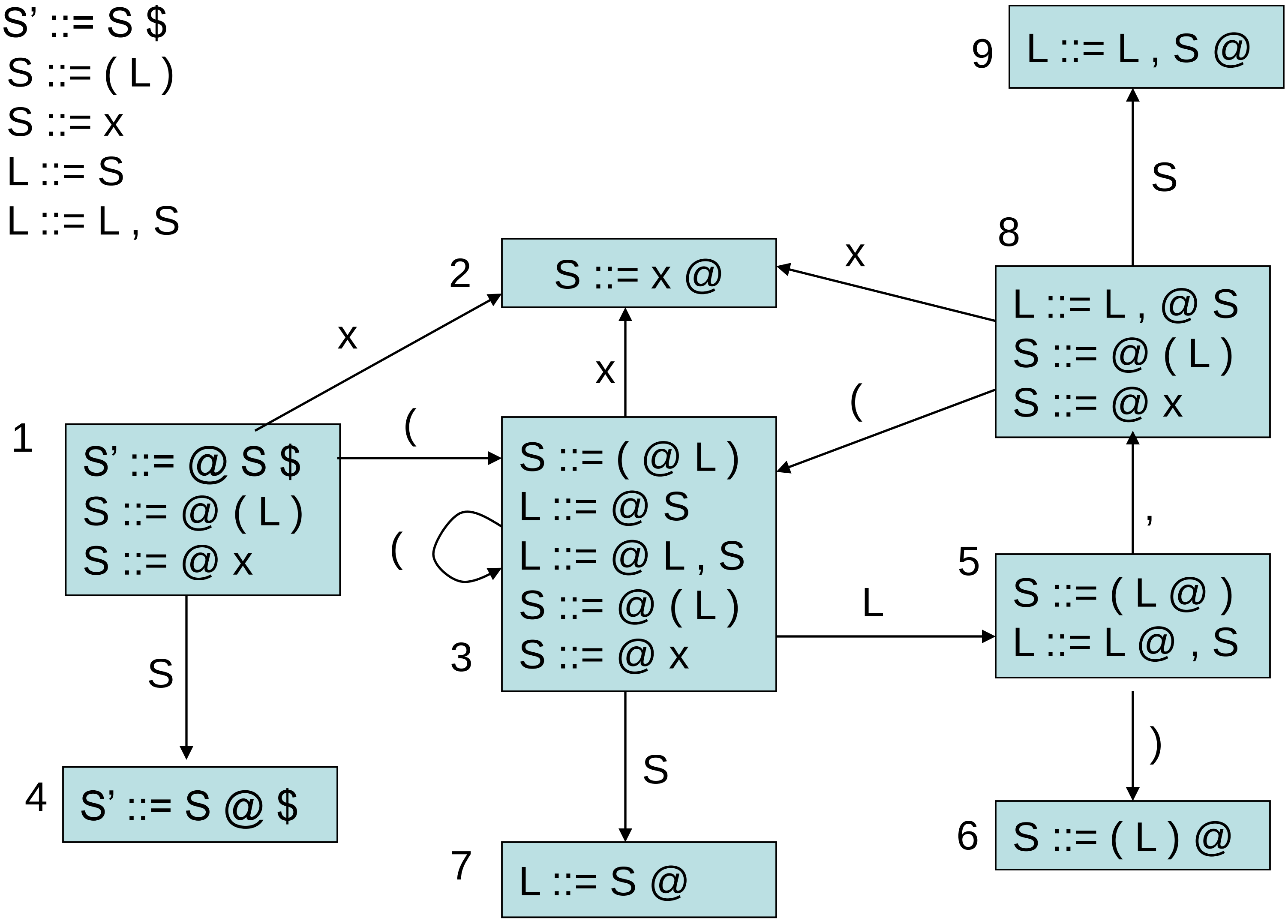
0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



Grammar:

Assigning numbers to states:

0. S' ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



Computing 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**
  - **accept**
  - **error**

Computing Parse table

- State i contains X ::= s @ $ ==> table[i,$] = a
- State i contains rule k: X ::= s @ ==> table[i,T] = rk for all terminals T
- Transition from i to j marked with T ==> table[i,T] = sj
- Transition from i to j marked with X ==> table[i,X] = gj for all nonterminals X

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

The Parse Table

- Reducing by rule k is broken into two steps:
  - current stack is:

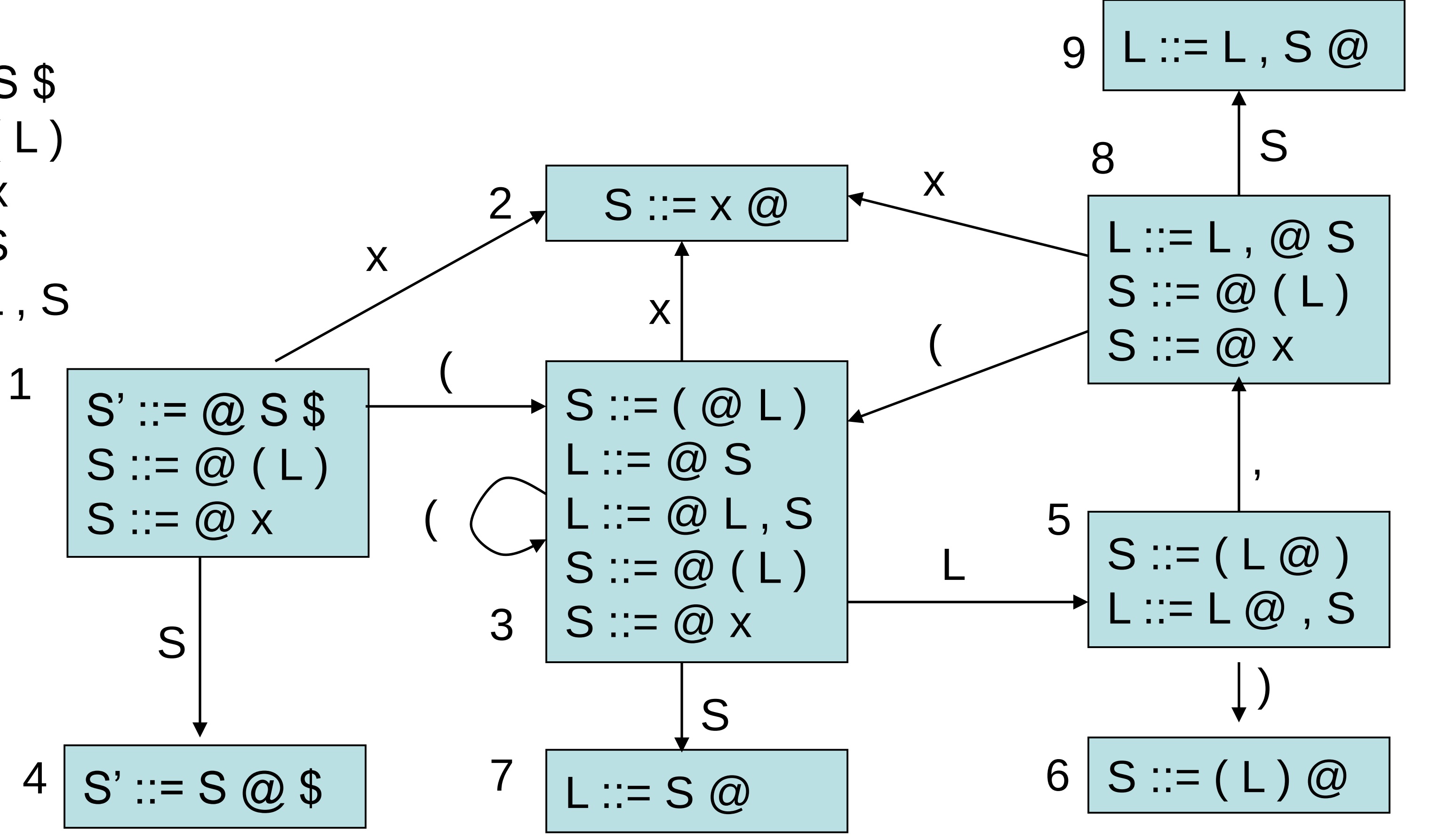
    A 8 B 3 C .... 7 RHS 12
  - rewrite the stack according to X ::= RHS:

    A 8 B 3 C .... 7 X
  - figure out state on top of stack (ie: goto 13)

    A 8 B 3 C .... 7 X 13

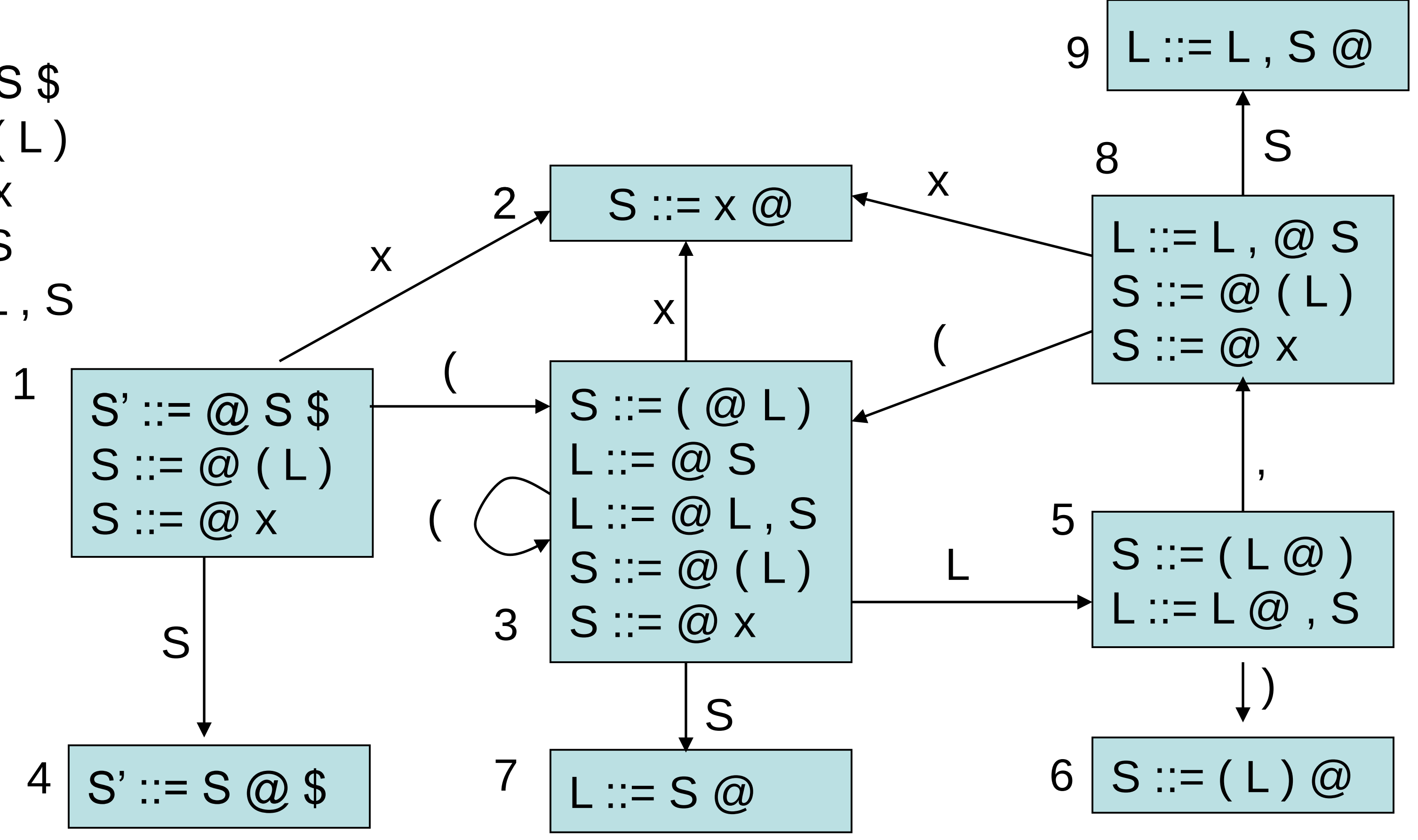
states	Terminal seen next ID, NUM, := ...	Non-terminals X,Y,Z ...
1		gn = goto state n
2	sn = shift & goto state n	
3	rk = reduce by rule k	
...	a = accept	
n	= error	

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



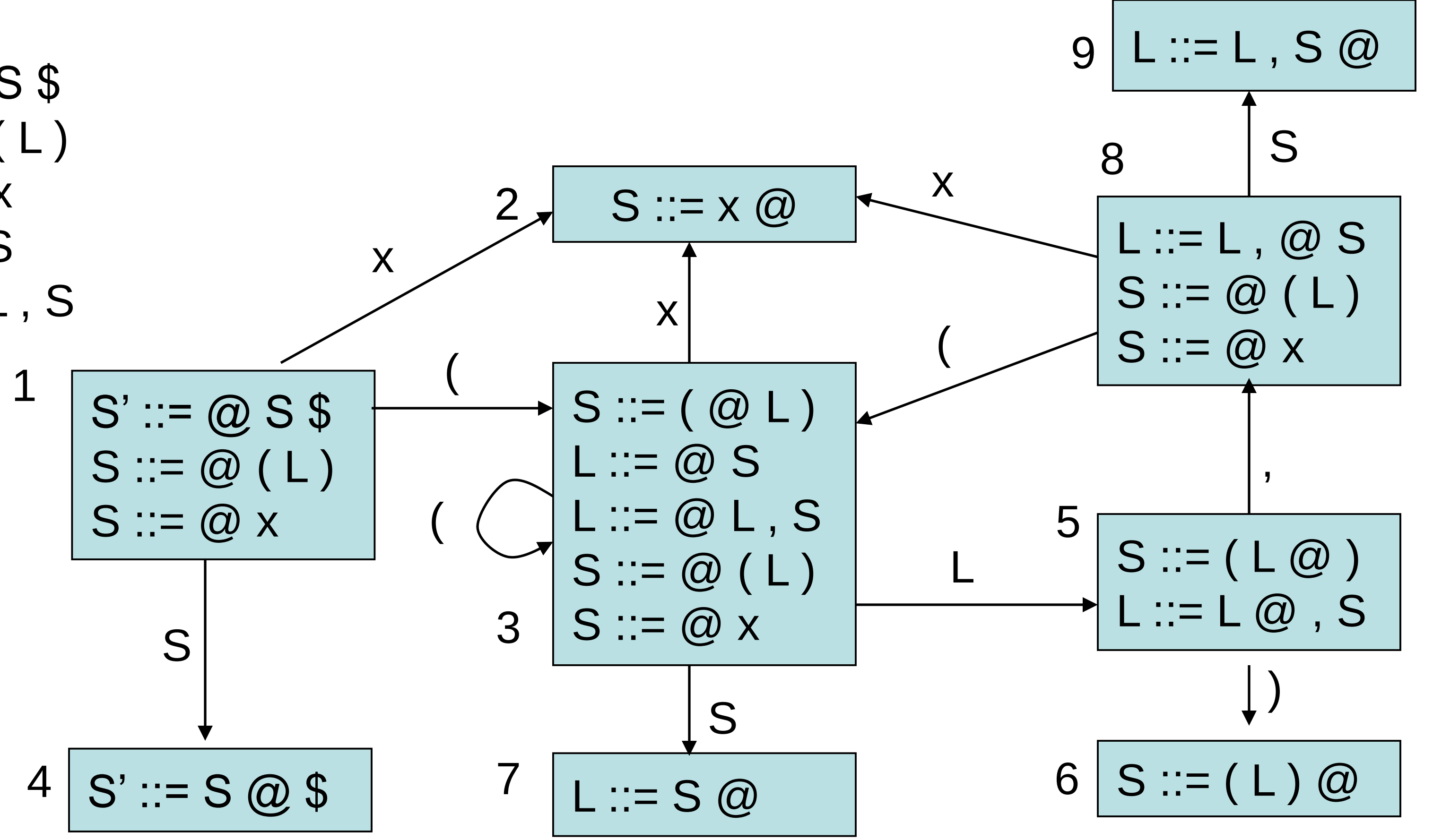
states	(	)	x	,	$	S	L
1							
2							
3							
4							
...							

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



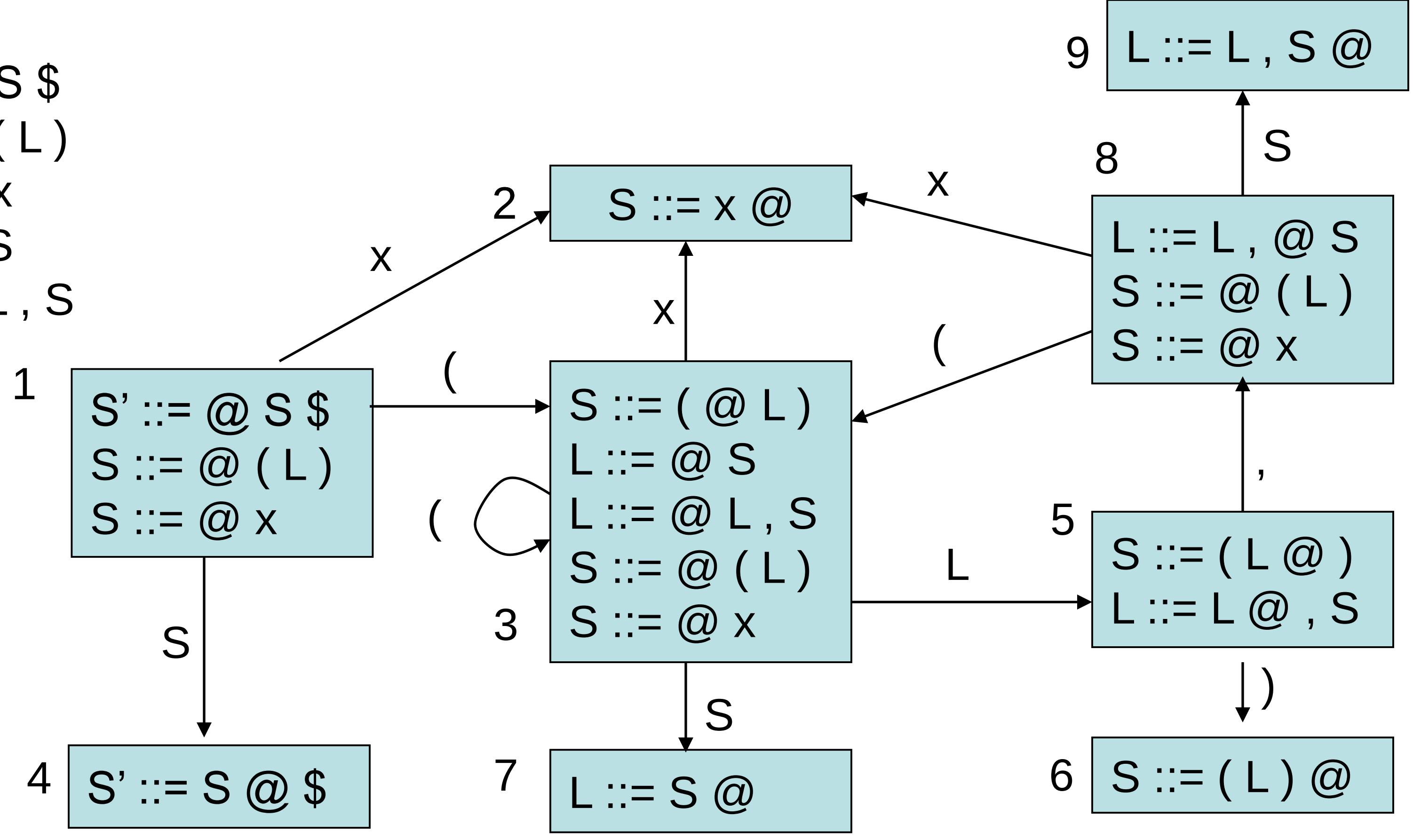
states	(	)	x	,	$	S	L
1	s3						
2							
3							
4							
...							

0. S' ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



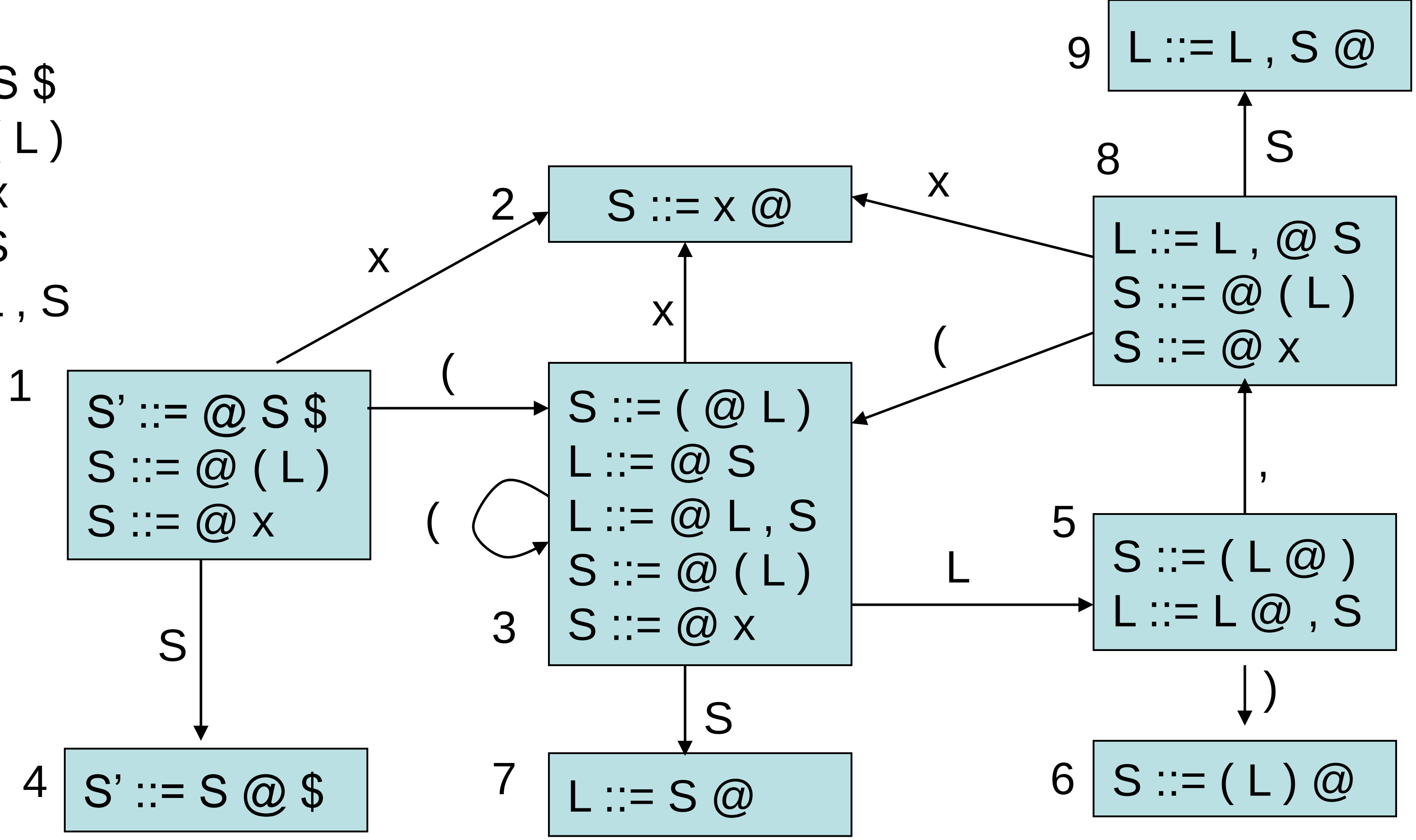
states	(	)	x	,	$	S	L
1	s3		s2				
2							
3							
4							
...							

0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



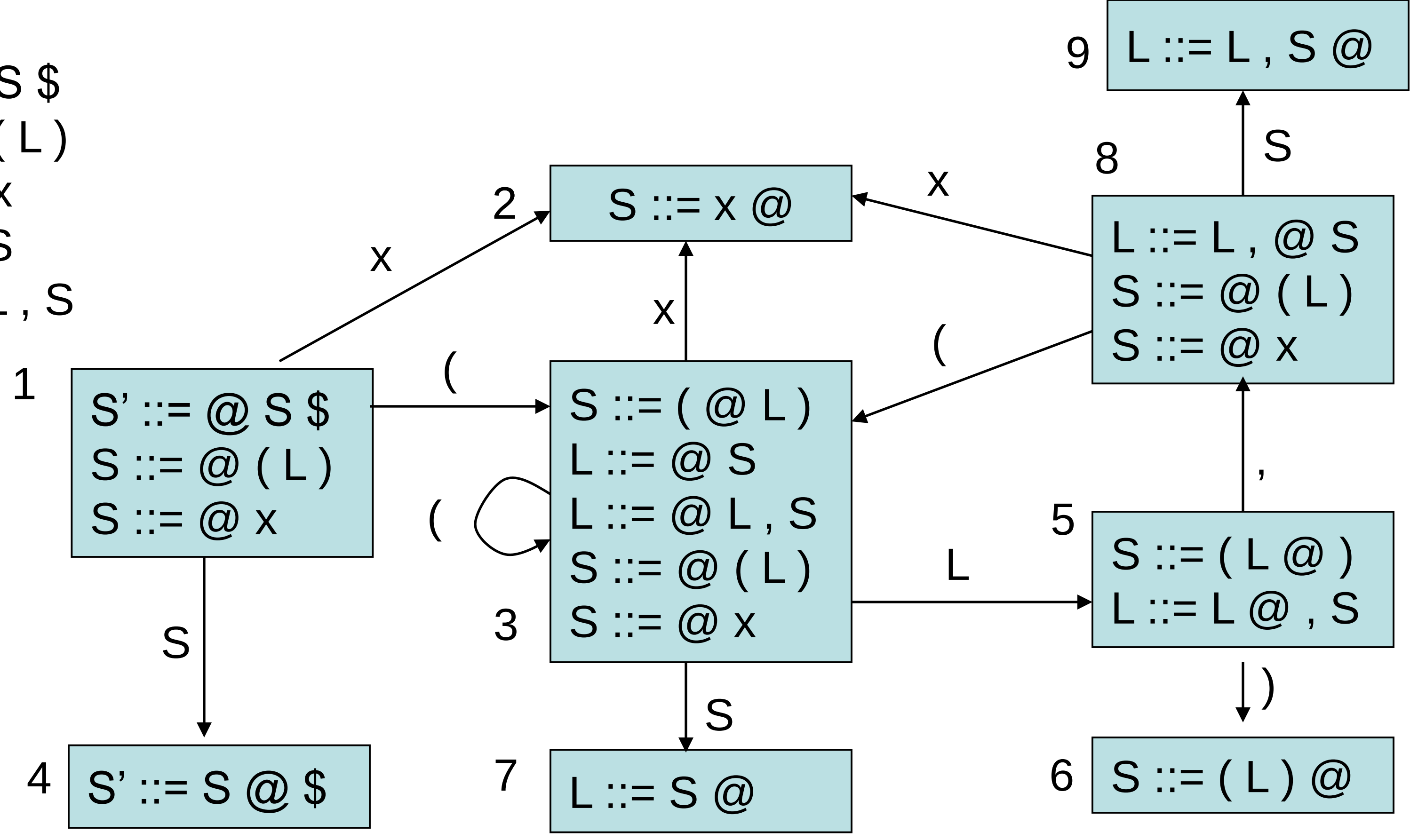
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3							
4							
...							

0. S' ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



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

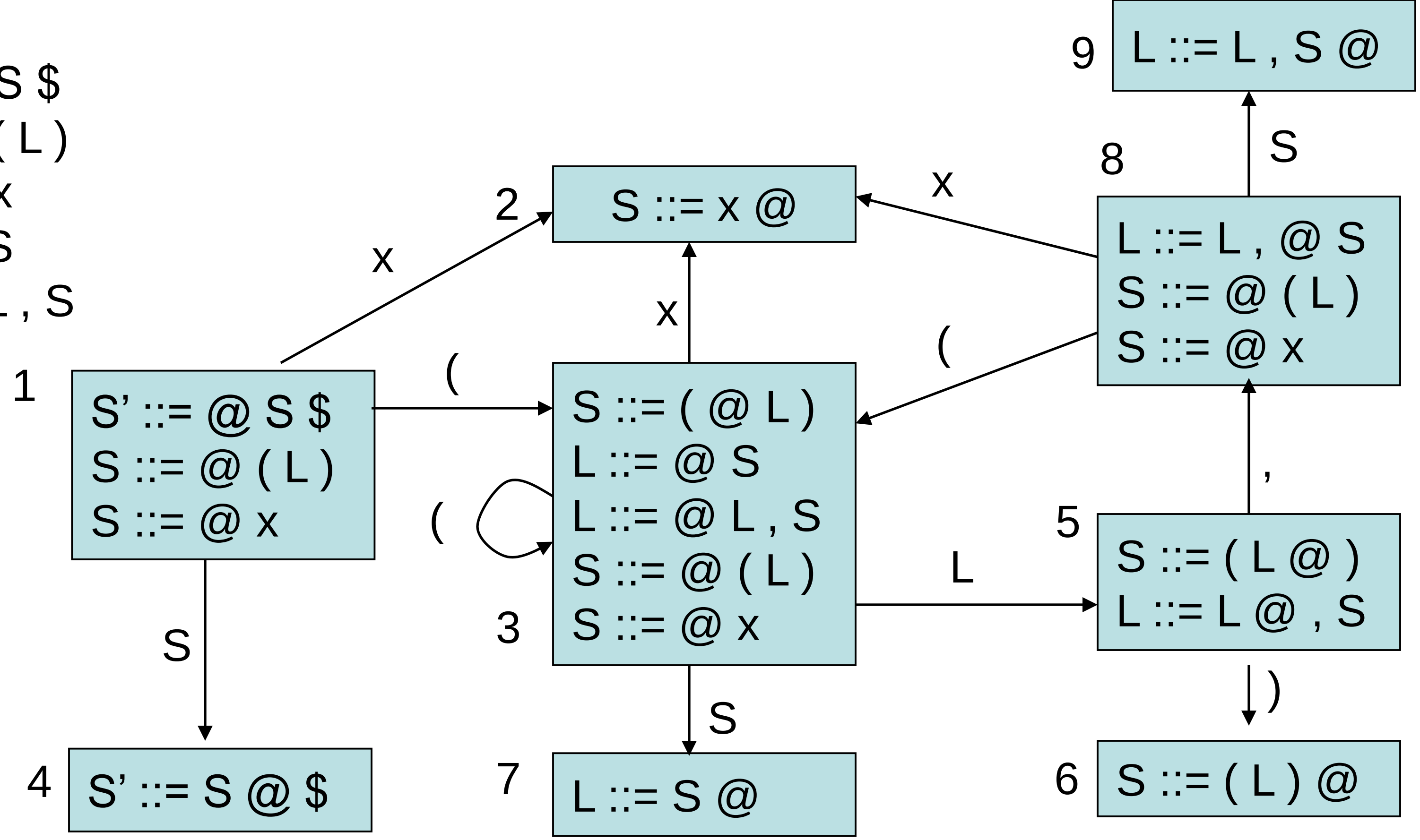
0. S' ::= S $
1. S ::= (L)
2. S ::= x
3. L ::= S
4. L ::= L , S



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

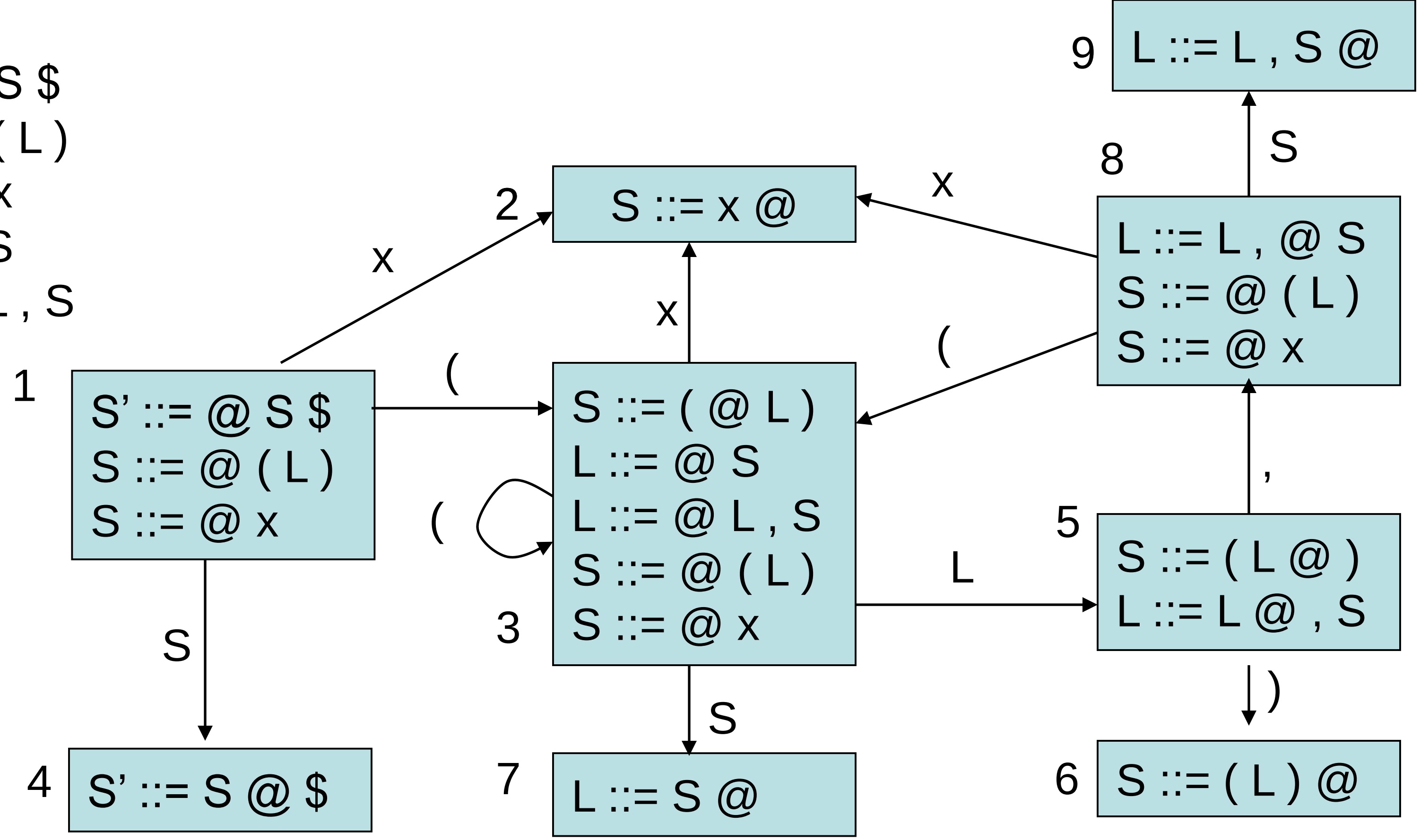
0. S’ ::= S $

- S ::= (L)
- S ::= x
- L ::= S
- 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



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

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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$$\text{input:}\quad \overbrace{(\ x\ ,\ x\ )\ \$}^{\text{yet to read}}$$

stack: 1

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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

input: $($ $\underbrace{\text{x , x) \$}}_{}$

yet to read

stack: 1 (3

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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

input: $(\; x \; \underbrace{, \; x \;) \; \$}_{}$ — yet to read: , x) $

stack: 1 (3 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$$\text{input:} \quad (\; x \; \overbrace{, \; x \;) \; \$}^{\text{yet to read}}$$

stack: 1 (3 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

yet to read

input: (x , x) $

stack: 1 (3 S 7

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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$$\text{input:} \quad (\ x \ \underbrace{, \ x \ ) \ \$}_{\text{yet to read}}$$

stack: 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$$\text{input:}\quad (\ x\ \underbrace{,\ x\ )\ \$}_{\text{yet to read}}$$

stack: 1 (3 L 5

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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

input: (x , $\underbrace{\text{x) \$}}_{}$ yet to read

stack: 1 (3 L 5 , 8

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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$$\text{input:} \quad (\ x\ ,\ x\ \underbrace{)\ \$}_{}$$

yet to read

input: (x , x) $

stack: 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

input: $\underbrace{\texttt{(\ x\ ,\ x\ )\ \$}}$ — $\overbrace{\texttt{, x) \$}}^{\text{yet to read}}$

stack: 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$\underbrace{\text{x) \$}}_{}$ yet to read

input: (x , x) $

stack: 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

$$\text{input:} \quad (\; x \; , \; x \; \underbrace{) \; \$}_{\text{yet to read}}$$

stack: 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 $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S

yet to read

input: (x , x) $

stack: 1 (3 L 5

etc ......

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

ignore next automaton state

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 ==> 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 look ahead 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 rk slots & therefore cuts down conflicts

LR(1) & LALR

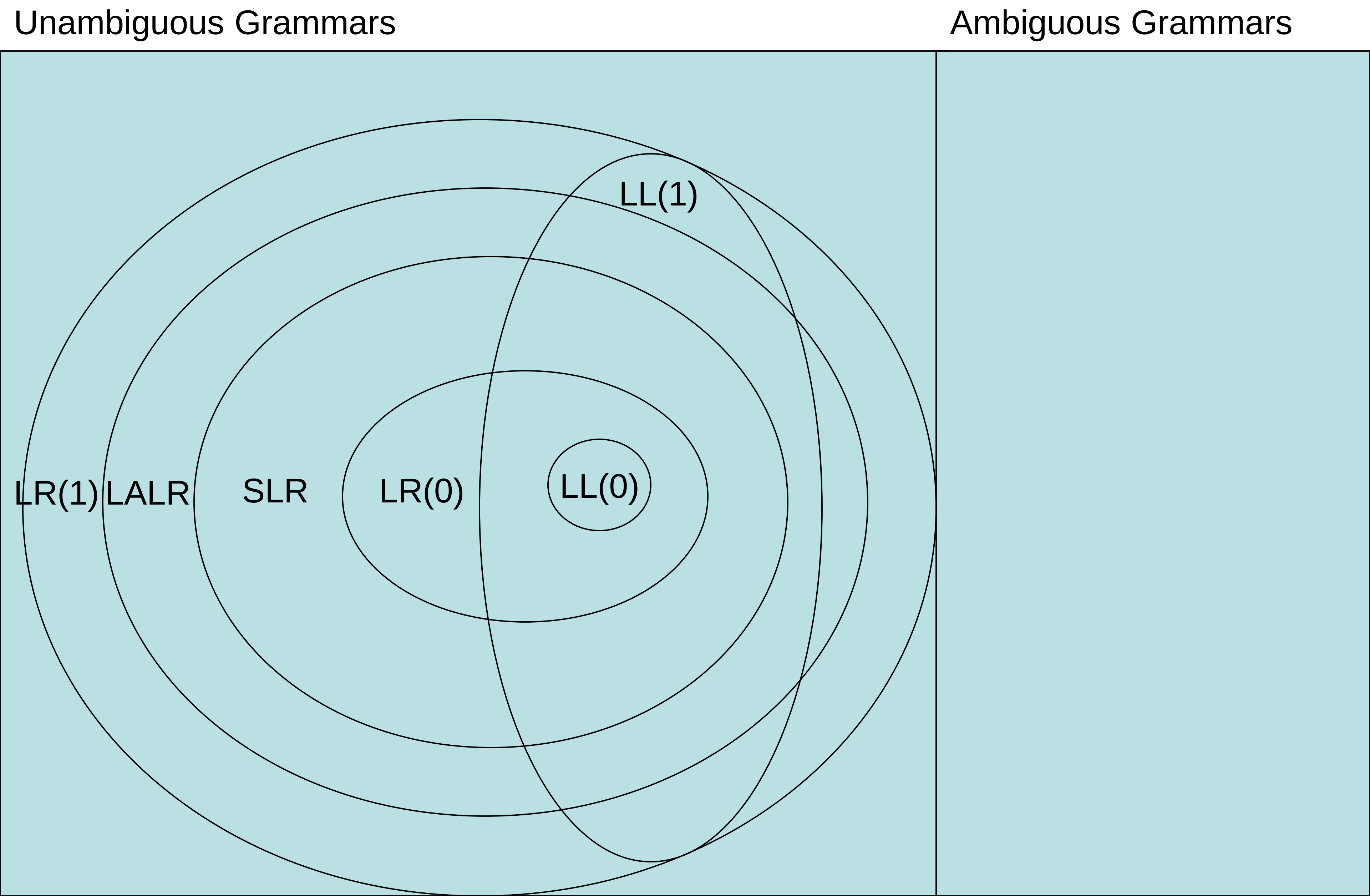
- LR(1) automata are identical to LR(0) except for the “items” that make up the states
- LR(0) items:

  X ::= s1 . s2
- LR(1) items

  X ::= s1 . s2, T

  look-ahead 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
  - ML-Yacc & most parser generators use LALR
- READ: Appel 3.3 (and also all of the rest of chapter 3)

Grammar Relationships



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
- ML-Yacc uses LALR, a compact variant of LR(1)