Parsing & Error Recovery

David Walker
COS 320

Error Recovery

- What should happen when your parser finds an error in the user's input?
  - stop immediately and signal an error
  - record the error but try to continue
- In the first case, the user must recompile from scratch after possibly a trivial fix
- In the second case, the user might be overwhelmed by a whole series of error messages, all caused by essentially the same problem
- We will talk about how to do error recovery in a principled way

Error Recovery

- Error recovery:
  - process of adjusting input stream so that the parser can continue after unexpected input
- Possible adjustments:
  - delete tokens
  - insert tokens
  - substitute tokens
- Classes of recovery:
  - local recovery: adjust input at the point where error was detected (and also possibly immediately after)
  - global recovery: adjust input before point where error was detected.
- Error recovery is possible in both top-down and bottom-up parsers

Local Bottom-up Error Recovery

```
exp : NUM                  ()
    | exp PLUS exp         ()
    | LPAR exp RPAR        ()

exps : exp                 ()
     | exps ; exp          ()
```

- general strategy for both bottom-up and top-down:
  - look for a synchronizing token

Local Bottom-up Error Recovery

```
exp : NUM                    ()
    | exp PLUS exp           ()
    | LPAR exp RPAR          ()

exps : exp                   ()
     | exps ; exp            ()
```

- general strategy for both bottom-up and top-down:
  - look for a synchronizing token
- accomplished in bottom-up parsers by adding error rules to grammar:

```
exp : LPAR error RPAR        ()

exps : exp                   ()
     | error ; exp           ()
```

Local Bottom-up Error Recovery

```
exp : NUM                    ()        exps : exp              ()
    | exp PLUS exp           ()             | exps ; exp       ()
    | LPAR exp RPAR          ()
```

- general strategy for both bottom-up and top-down:
  - look for a synchronizing token
- accomplished in bottom-up parsers by adding error rules to grammar:

```
exp : LPAR error RPAR      ()

exps : exp                 ()
     | error ; exp         ()
```

- in general, follow error with a synchronizing token. Recovery steps:
  - Pop stack (if necessary) until a state is reached in which the action for the error token is shift
  - Shift the error token
  - Discard input symbols (if necessary) until a state is reached that has a non-error action
  - Resume normal parsing

Local Bottom-up Error Recovery

```
exp : NUM                 ()        exps : exp             ()
    | exp PLUS exp        ()             | exps ; exp      ()
    | ( exp )             ()
```

```
exp : ( error )           ()

exps : exp                ()
     | error ; exp        ()
```

yet to read

input: NUM PLUS (NUM PLUS @#$ PLUS NUM) PLUS NUM

stack: exp PLUS (exp PLUS

@#$ is an unexpected token!

Local Bottom-up Error Recovery

```
exp : NUM                    ()
    | exp PLUS exp           ()
    | ( exp )                ()

exps : exp                   ()
     | exps ; exp            ()
```

```
exp : ( error )              ()

exps : exp                   ()
     | error ; exp           ()
```

yet to read

input: NUM PLUS (NUM PLUS $\underbrace{\text{@\#\$ PLUS NUM) PLUS NUM}}$

stack: exp PLUS (

pop stack until shifting “error” can result in correct parse

Local Bottom-up Error Recovery

```
exp : NUM                  ()        exps : exp          ()
    | exp PLUS exp         ()            | exps ; exp    ()
    | ( exp )              ()
```

```
exp : ( error )            ()

exps : exp                 ()
     | error ; exp         ()
```

yet to read

input: NUM PLUS (NUM PLUS @#$ PLUS NUM) PLUS NUM

stack: exp PLUS (error

shift “error”

Local Bottom-up Error Recovery

```
exp : NUM                  ()          exps : exp           ()
    | exp PLUS exp         ()              | exps ; exp     ()
    | ( exp )              ()
```

```
exp : ( error )            ()

exps : exp                 ()
     | error ; exp         ()
```

yet to read

input: NUM PLUS (NUM PLUS @#$ PLUS NUM) PLUS NUM

stack: exp PLUS (error

discard input until we can legally shift or reduce

Local Bottom-up Error Recovery

```
exp : NUM                 ()        exps : exp           ()
    | exp PLUS exp        ()             | exps ; exp    ()
    | ( exp )             ()
```

```
exp : ( error )           ()

exps : exp                ()
     | error ; exp        ()
```

yet to read

input: NUM PLUS (NUM PLUS @#$ PLUS NUM) $\overbrace{\text{PLUS NUM}}$

stack: exp PLUS (error)

SHIFT)

Local Bottom-up Error Recovery

```
exp : NUM                  ()        exps : exp              ()
    | exp PLUS exp         ()             | exps ; exp       ()
    | ( exp )              ()
```

```
exp : ( error )            ()

exps : exp                 ()
     | error ; exp         ()
```

yet to read

input: NUM PLUS (NUM PLUS @#$ PLUS NUM) PLUS NUM

stack: exp PLUS exp

REDUCE using exp ::= (error)

Local Bottom-up Error Recovery

```
exp : NUM                  ()        exps : exp              ()
    | exp PLUS exp         ()             | exps ; exp       ()
    | ( exp )              ()
```

```
exp : ( error )            ()

exps : exp                 ()
     | error ; exp         ()
```

yet to read

input: NUM PLUS (NUM PLUS @#$ PLUS NUM) $\underbrace{\text{PLUS NUM}}_{}$

stack: exp PLUS exp

continue parsing...

Top-down Local Error Recovery

- also possible to use synchronizing tokens
- here, a synchronizing token for non terminal X is a member of Follow(X)
  - when parsing X and an error is found; eat input stream until you get to a member of Follow(X)

non-terminals: S, E, L
terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =
rules:

1. S ::= IF E THEN S ELSE S
2. | BEGIN S L
3. | PRINT E
4. L ::= END
5. | ; S L
6. E ::= NUM = NUM

```
val tok = ref (getToken ())
fun advance () = tok := getToken ()
fun eat t = if (! tok = t) then advance () else error ()
```

```
fun skipto toks =
  if member(!tok, toks) then ()
  else
     eat(!tok); skipto toks
```

```
fun S () = case !tok of
              IF => ... | BEGIN => ... | PRINT => ...


and L () = case !tok of
             END    =>   eat END
            | SEMI    =>  eat SEMI; S (); L ()


and E () = case !tok of
              NUM => eat NUM; eat EQ; eat NUM
```

non-terminals: S, E, L

terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =

rules:

1. S ::= IF E THEN S ELSE S	4. L ::= END
2. | BEGIN S L	5. | ; S L
3. | PRINT E	6. E ::= NUM = NUM

```
val tok = ref (getToken ())
fun advance () = tok := getToken ()
fun eat t = if (! tok = t) then advance () else error ()
```

```
fun skipto toks =
  if member(!tok, toks) then ()
  else
    eat(!tok); skipto toks
```

```
fun S () = case !tok of
            IF => ... | BEGIN => ... | PRINT => ...
          | _ => skipto [ELSE,END,SEMI]

and L () = case !tok of
            END    =>  eat END
          | SEMI   =>  eat SEMI; S (); L ()
          | _      =>

and E () = case !tok of
            NUM => eat NUM; eat EQ; eat NUM
          | _     =>
```

non-terminals: S, E, L
terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =
rules:

```
1.  S ::= IF E THEN S ELSE S        4.  L ::= END
2.        | BEGIN S L               5.        | ; S L
3.        | PRINT E                 6.  E ::= NUM = NUM
```

```
val tok = ref (getToken ())
fun advance () = tok := getToken ()
fun eat t = if (! tok = t) then advance () else error ()
```

```
fun skipto toks =
  if member(!tok, toks) then ()
  else
     eat(!tok); skipto toks
```

```
fun S () = case !tok of
              IF => ... | BEGIN => ... | PRINT => ...
            | _ => skipto [ELSE,END,SEMI]

and L () = case !tok of
              END    =>   eat END
            | SEMI    =>  eat SEMI; S (); L ()
            | _        =>   skipto [ELSE, END,SEMI]

and E () = case !tok of
              NUM => eat NUM; eat EQ; eat NUM
            | _      =>
```

non-terminals: S, E, L
terminals: NUM, IF, THEN, ELSE, BEGIN, END, PRINT, ;, =
rules:

1. S ::= IF E THEN S ELSE S
2. | BEGIN S L
3. | PRINT E
4. L ::= END
5. | ; S L
6. E ::= NUM = NUM

```
val tok = ref (getToken ())
fun advance () = tok := getToken ()
fun eat t = if (! tok = t) then advance () else error ()
```

```
fun skipto toks =
  if member(!tok, toks) then ()
  else
     eat(!tok); skipto toks
```

```
fun S () = case !tok of
             IF => ... | BEGIN => ... | PRINT => ...
           | _ => skipto [ELSE,END,SEMI]

and L () = case !tok of
             END    =>  eat END
           | SEMI   =>  eat SEMI; S (); L ()
           | _      =>  skipto [ELSE, END,SEMI]

and E () = case !tok of
             NUM => eat NUM; eat EQ; eat NUM
           | _    => skipto [THEN,ELSE,END,SEMI]
```

global error recovery

- global error recovery determines the smallest set of insertions, deletions or replacements that will allow a correct parse, even if those insertions, etc. occur before an error would have been detected
- ML-Yacc uses Burke-Fisher error repair
  - tries every possible single-token insertion, deletion or replacement at every point in the input up to K tokens before the error is detected
    - eg: K = 20; parser gets stuck at token 500; all possible repairs between token 480-500 tried
    - best repair = longest successful parse

global error recovery

- Consider Burke-Fisher with
  - K-token window
  - N different token types
- Total number of repairs = K + 2K*N
    - deletions (K) +
    - insertions (K + 1)*N +
    - replacements (K)(N-1)
- Affordable in the uncommon case when there is an error

global error recovery

- Parser must be able to back up K tokens and reparse
- Mechanics:
  - parser maintains old stack and new stack

K-token window maintained in queue by parser

K-token window | yet to read

input: ID := NUM ; ID := ID + (ID := NUM + ...

new stack: S ; ID := E + (

old stack: ID := NUM

global error recovery

- Parser must be able to back up K tokens and reparse
- Mechanics:
  - parser maintains old stack and new stack

K-token window maintained in queue by parser

K-token window | yet to read

input: ID := NUM ; ID := ID + (ID := NUM + ...

new stack: S ; ID := E + (

old stack: ID := NUM

old stack lags the new stack by K=6 tokens

Reductions (E ::= NUM) and (S ::= ID := NUM) applied to old stack in turn

global error recovery

- Parser must be able to back up K tokens and reparse
- Mechanics:
  - parser maintains old stack and new stack

K-token window
maintained in queue
by parser

K-token window

yet to read

input: ID := NUM ; ID := ID + (ID := NUM + ...

new stack: S ; ID := E + (

old stack: ID := NUM

semantic actions performed once when reduction is “committed” on the old stack

Burke-Fisher in ML-Yacc

- ML-Yacc provides additional support for Burke-Fisher
  - to continue parsing, we need semantics values for inserted tokens

```
%value ID {make_id "bogus"}
%value INT {0}
%value STRING {""}
```

  - some multiple-token insertions & deletions are common, but it is too expensive for ML-Yacc to try every 2,3,4- token insertion, deletion

```
%change EQ -> ASSIGN
        | SEMI ELSE -> ELSE
        |      -> IN INT END
```

ML-Yacc would do this anyway but by specifying, it tries it first

finally the magic: how to construct an LR parser table

$$\text{input:}\quad \text{NUM PLUS (NUM PLUS } \overbrace{\text{NUM) PLUS NUM}}^{\text{yet to read}}$$

stack: exp PLUS (exp PLUS

- 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" ==> a state in a finite automaton

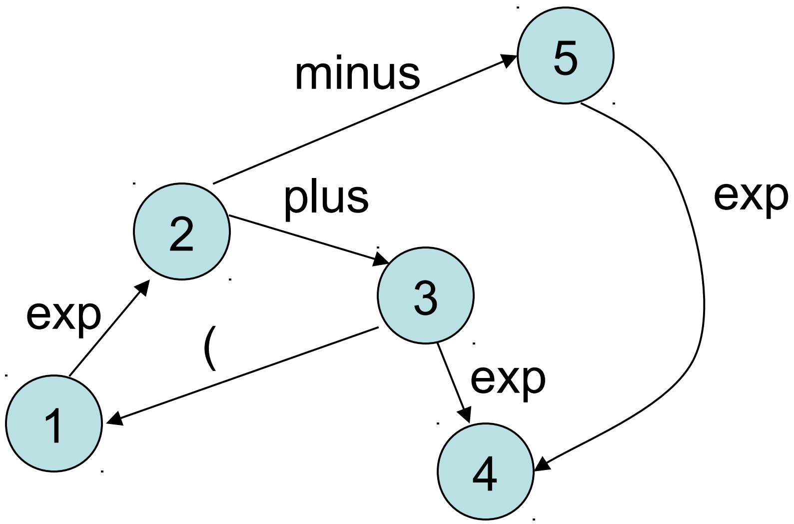
finally the magic: how to construct an LR parser table

$\overbrace{\hphantom{NUM) PLUS NUM}}^{\text{yet to read}}$

input: NUM PLUS (NUM PLUS NUM) PLUS NUM

stack: exp PLUS (exp PLUS

finite automaton; terminals and non terminals label edges



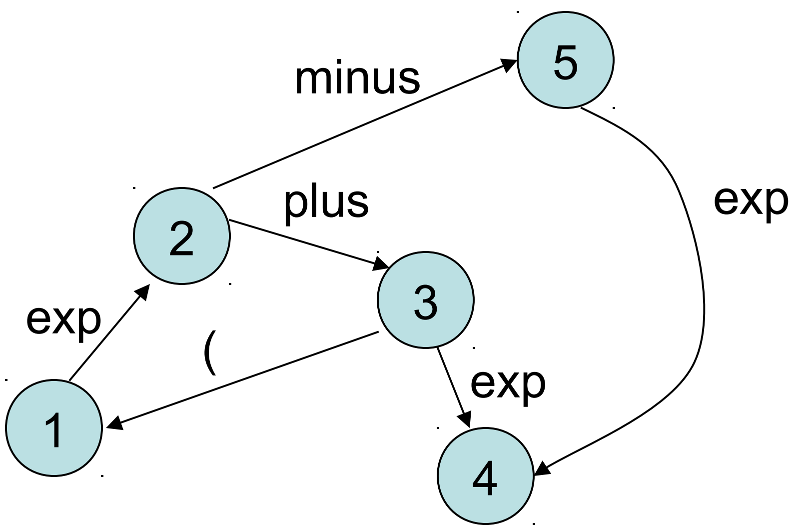
finally the magic: how to construct an LR parser table

yet to read

input: NUM PLUS (NUM PLUS NUM) PLUS NUM

stack: exp PLUS (exp PLUS

finite automaton; terminals and non terminals label edges



state-annotated stack: 1

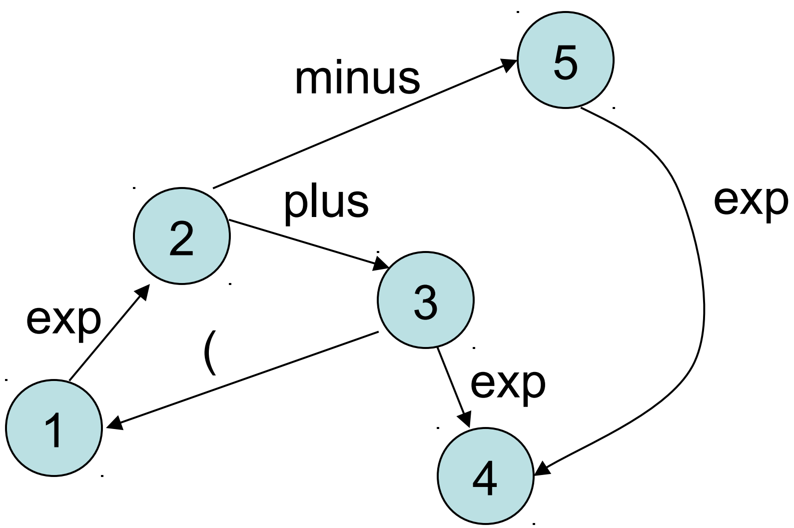
finally the magic: how to construct an LR parser table

yet to read

input: NUM PLUS (NUM PLUS NUM) PLUS NUM

stack: exp PLUS (exp PLUS

finite automaton; terminals and non terminals label edges



state-annotated stack: 1 exp 2

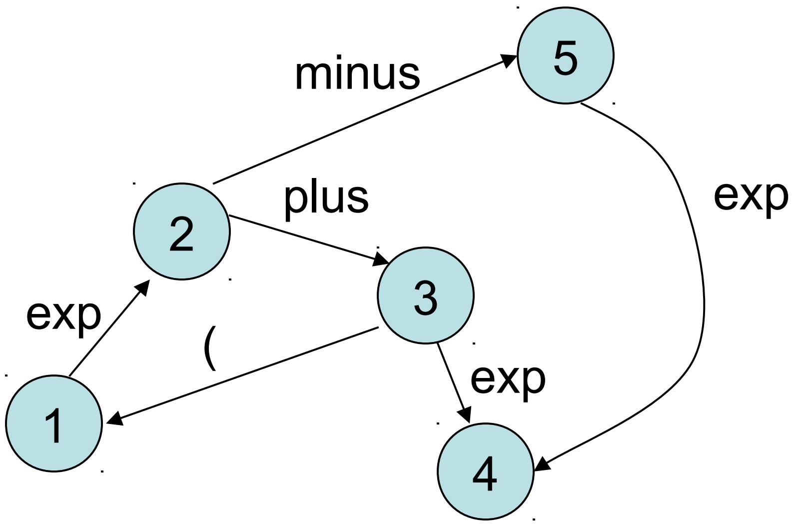
finally the magic: how to construct an LR parser table

yet to read

input: NUM PLUS (NUM PLUS NUM) PLUS NUM

stack: exp PLUS (exp PLUS

finite automaton; terminals and non terminals label edges



state-annotated stack: 1 exp 2 PLUS 3

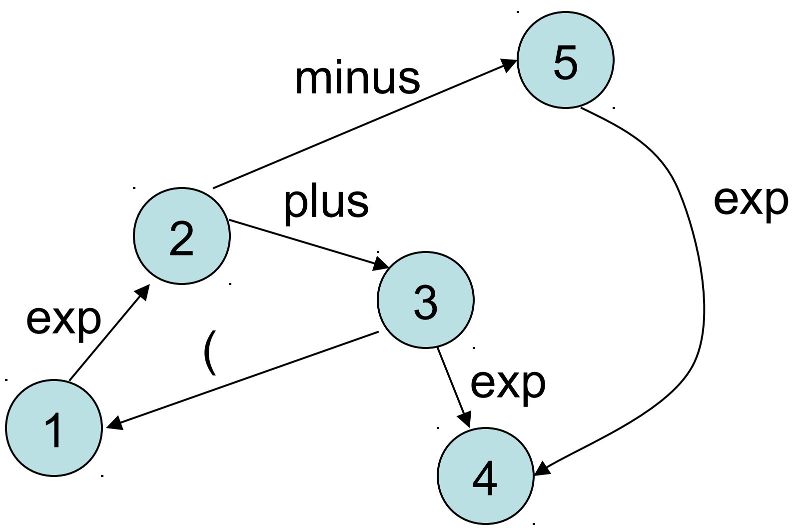
finally the magic: how to construct an LR parser table

yet to read

input: NUM PLUS (NUM PLUS NUM) PLUS NUM

stack: exp PLUS (exp PLUS

finite automaton; terminals and non terminals label edges



state-annotated stack: 1 exp 2 PLUS 3 (1 exp 2 PLUS 3

this state and input tell us what to do next

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	sn = shift & 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		
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		
2	sn = shift & goto state n	gn = goto state n
3	rk = reduce by rule k	
...	a = accept	
n	= error	

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
    - eg: 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:

1

S’ ::= @ S $
S ::= @ (L)
S ::= @ x

state number

collection of LR(0) items

- LR(1) states look very similar, it is just that the items contain some look-ahead info

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:

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

1

S’ ::= @ S $

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:

0. 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:

0. 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

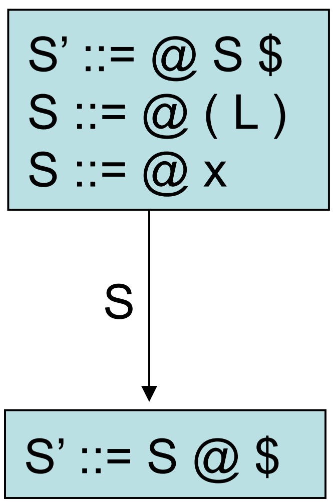
Grammar:

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

S’ ::= @ S $
S ::= @ (L)
S ::= @ x

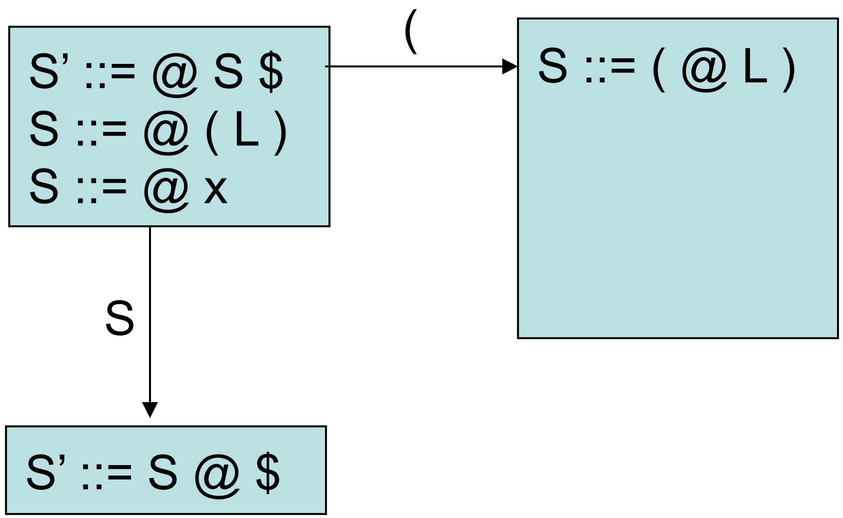
Grammar:

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



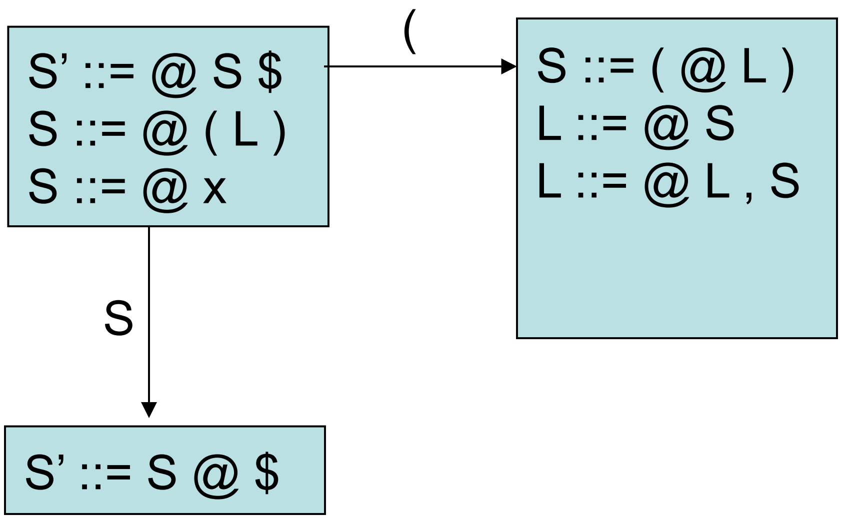
Grammar:

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



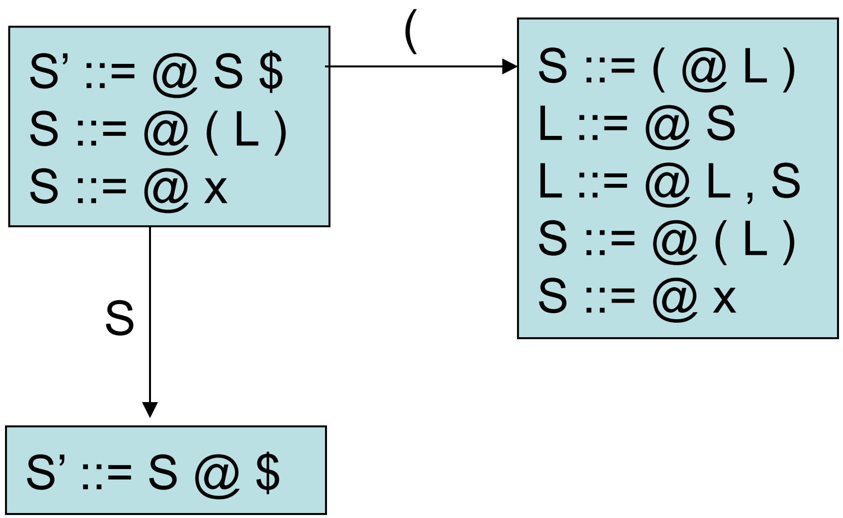
Grammar:

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



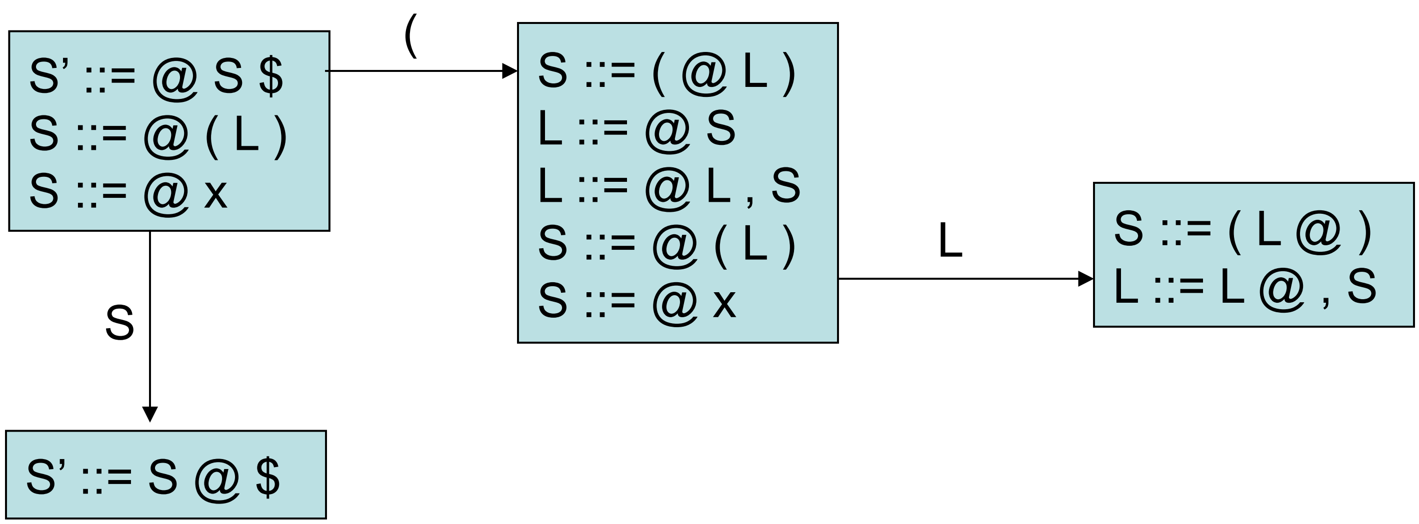
Grammar:

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



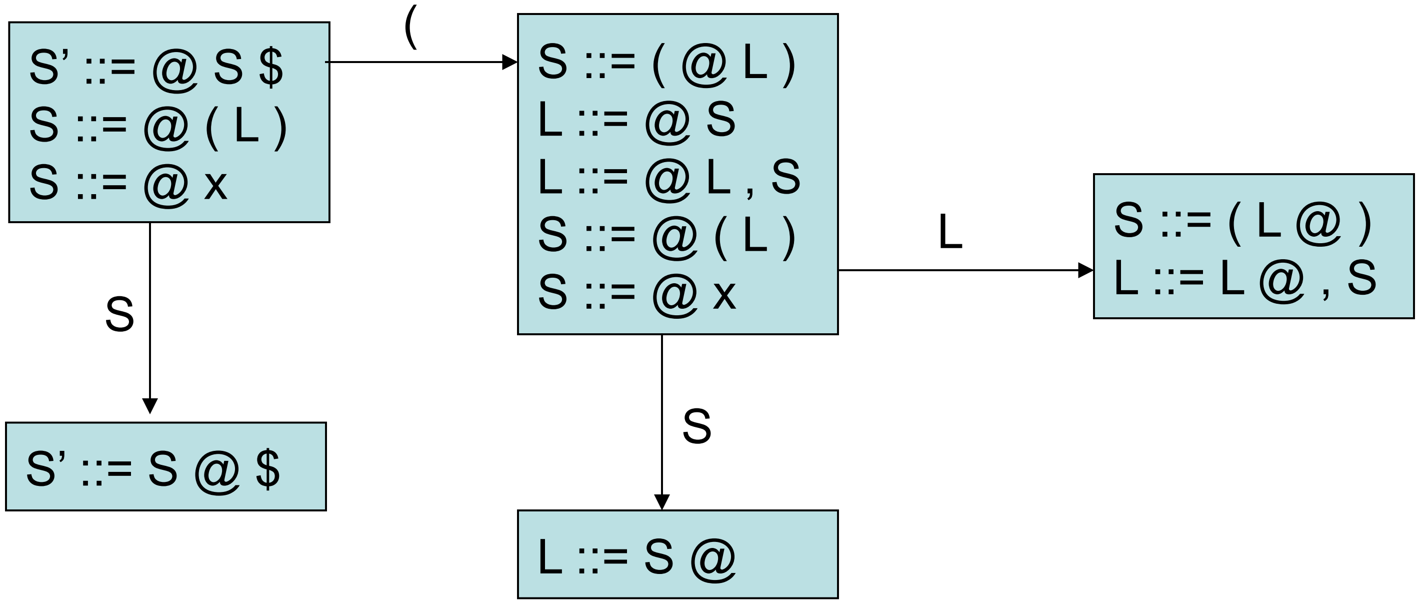
Grammar:

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



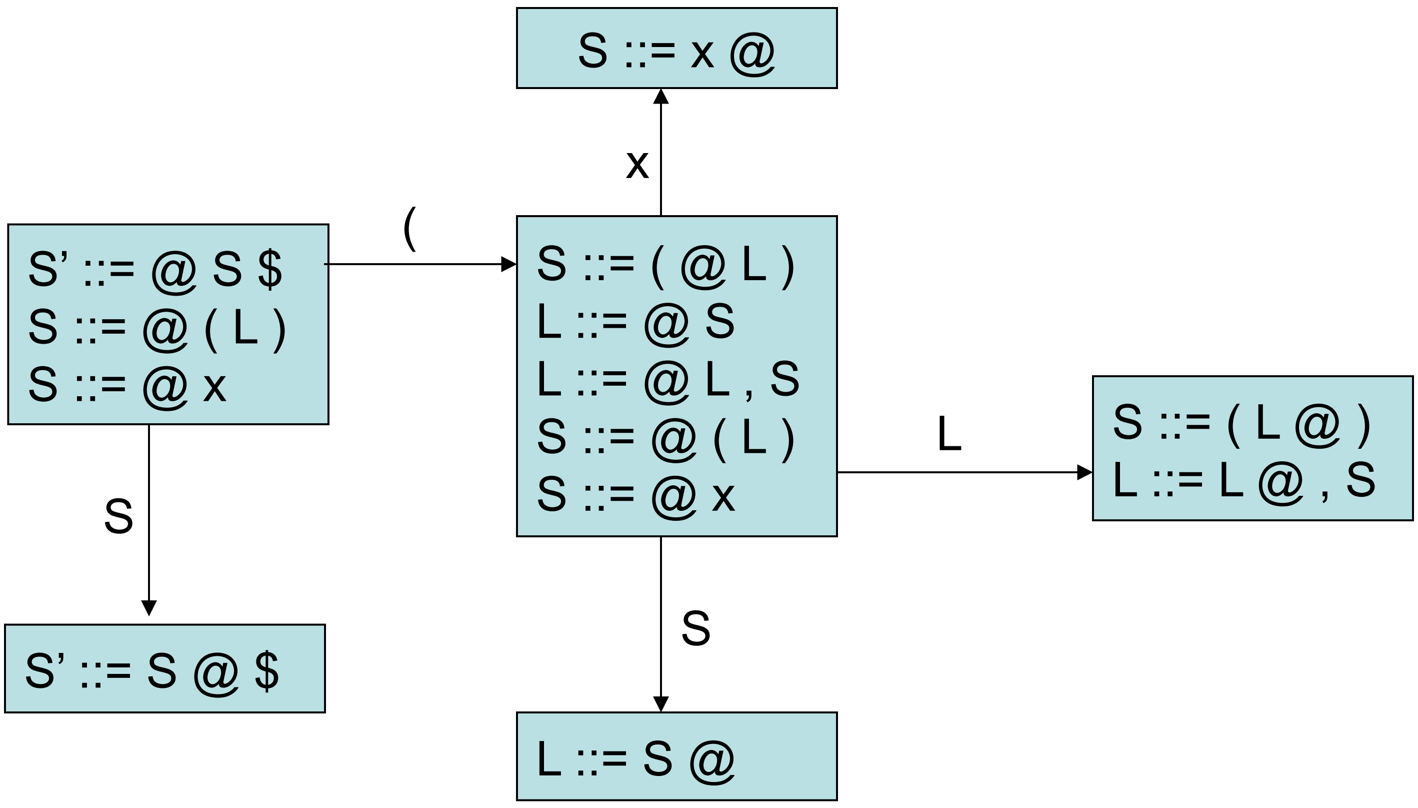
Grammar:

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



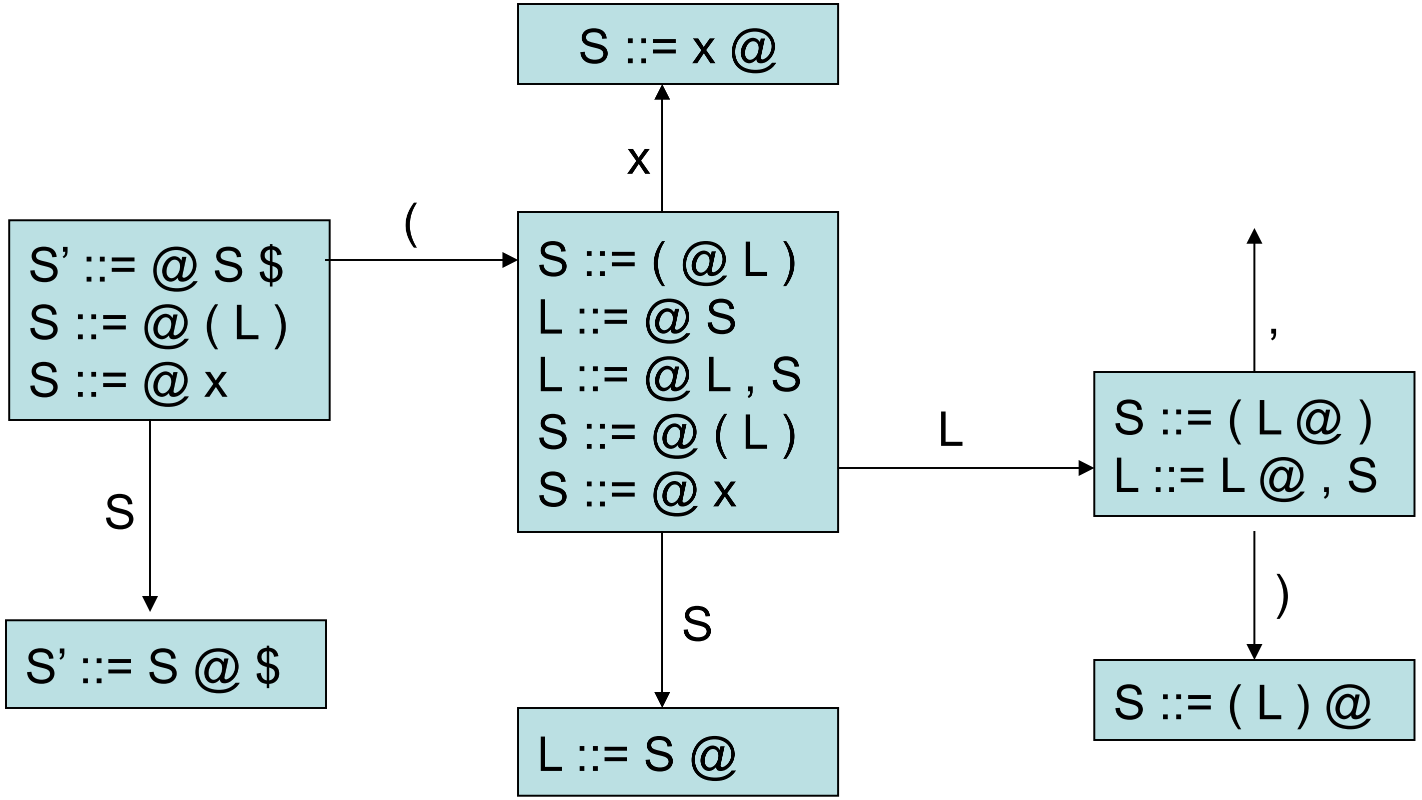
Grammar:

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



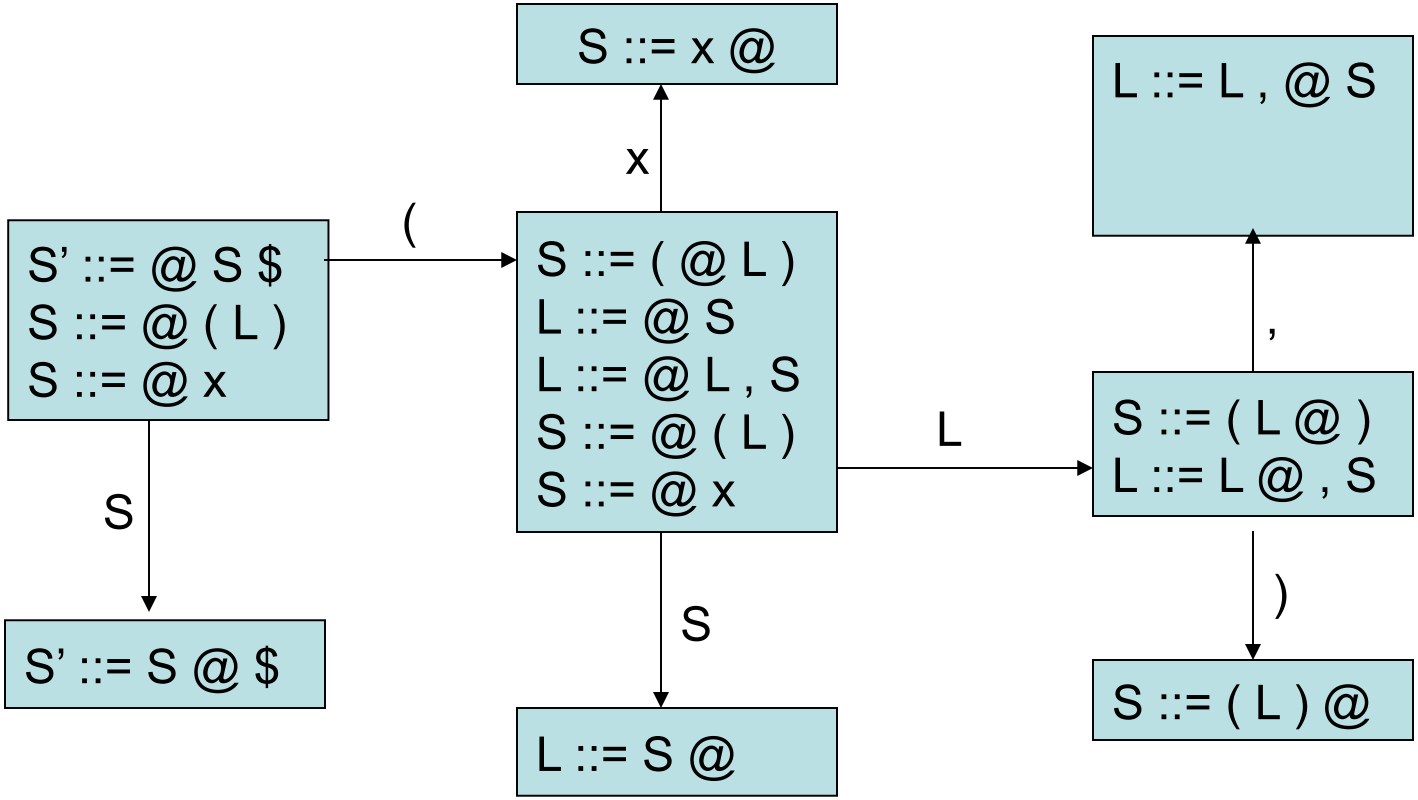
Grammar:

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



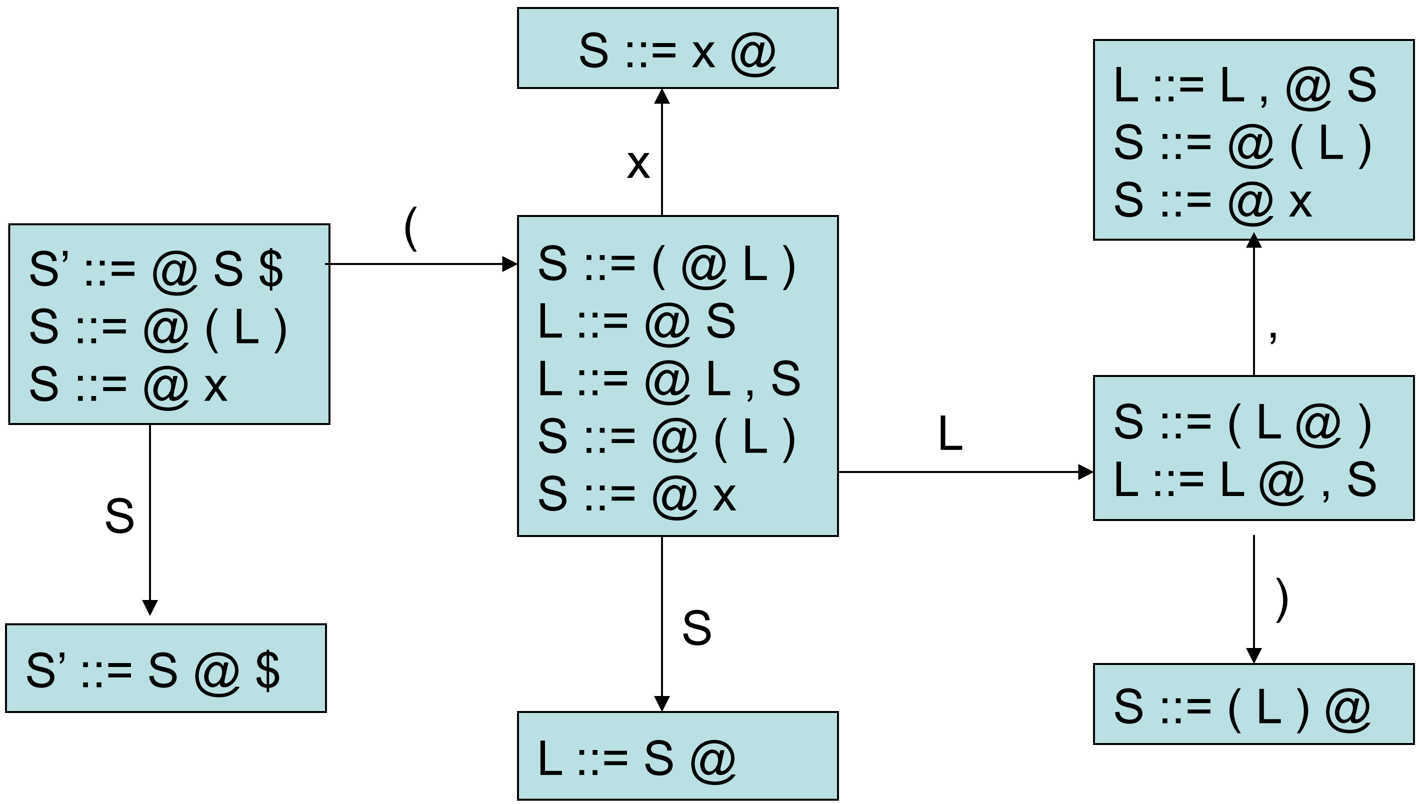
Grammar:

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



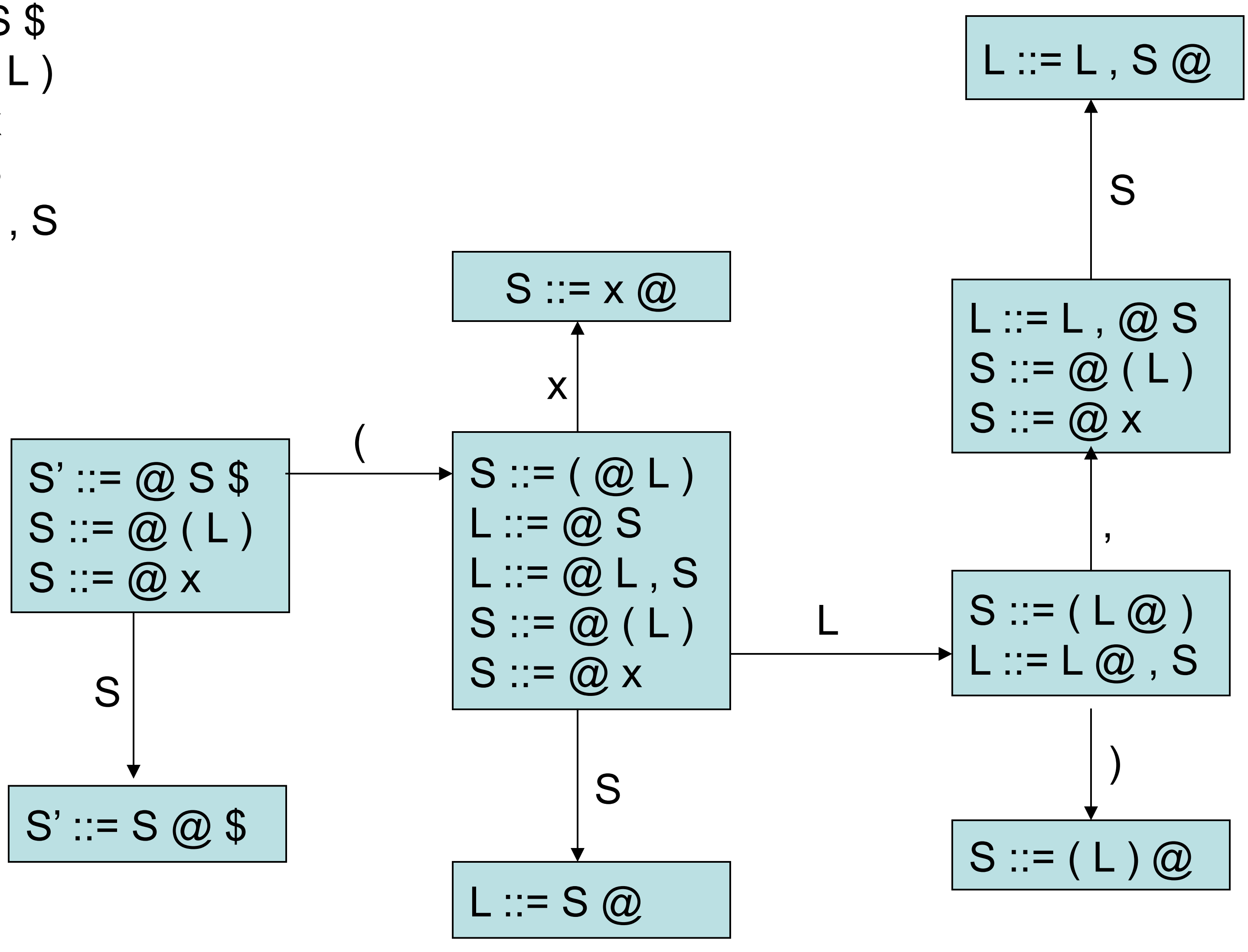
Grammar:

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



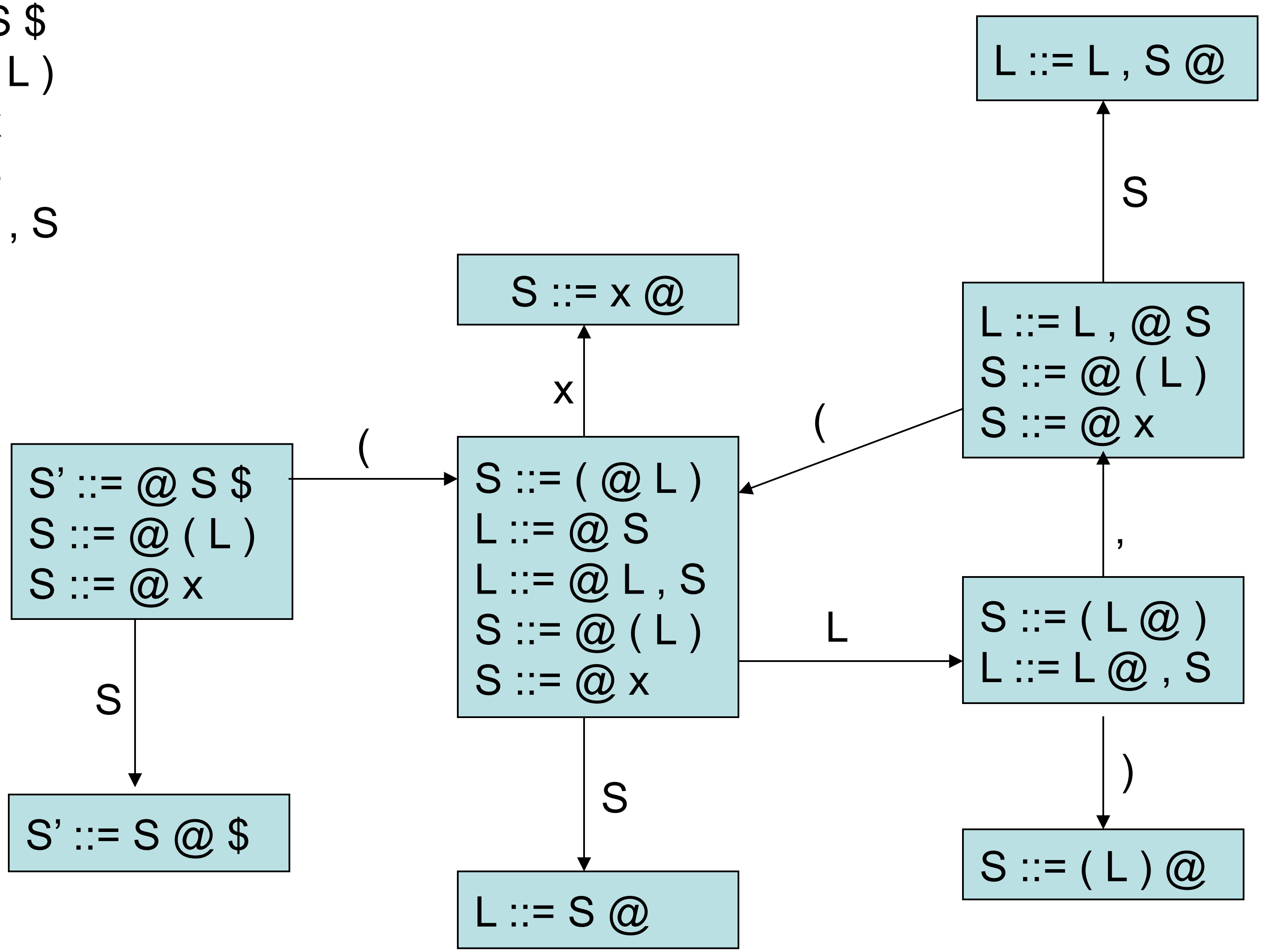
Grammar:

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



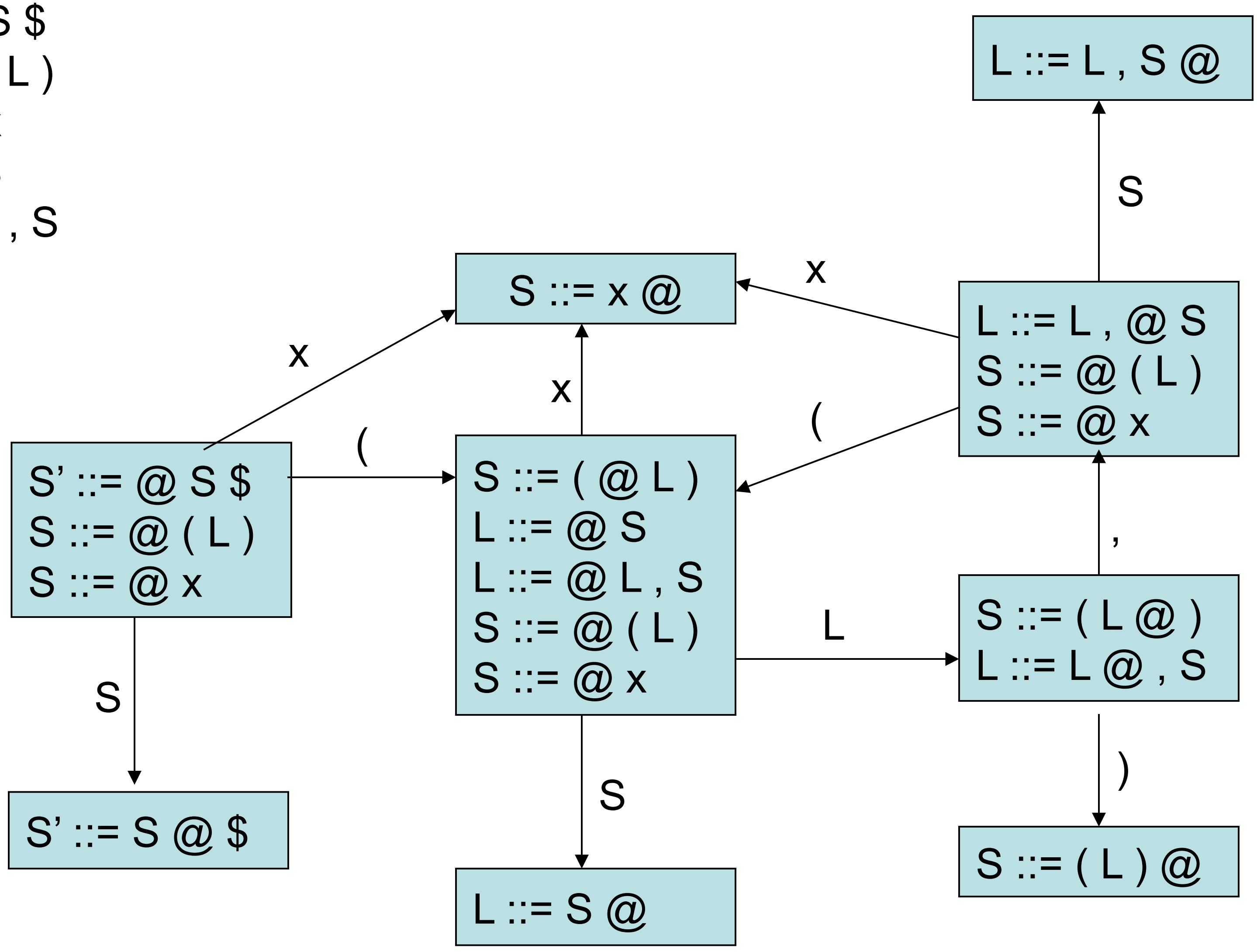
Grammar:

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



Grammar:

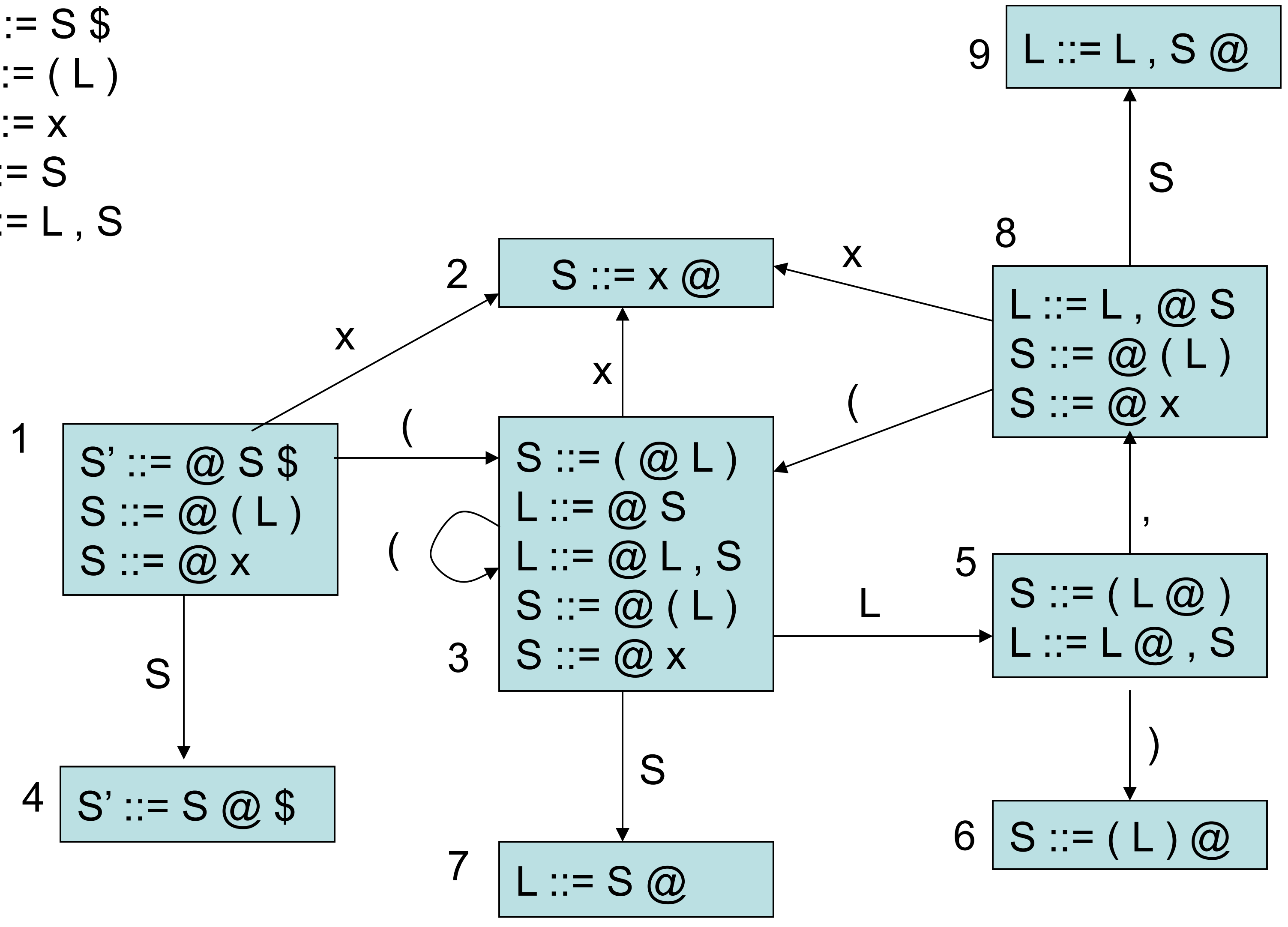
0. S’ ::= S $
- S ::= (L)
- S ::= x
- L ::= S
- L ::= L , S



Grammar:

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

Assigning numbers to states:

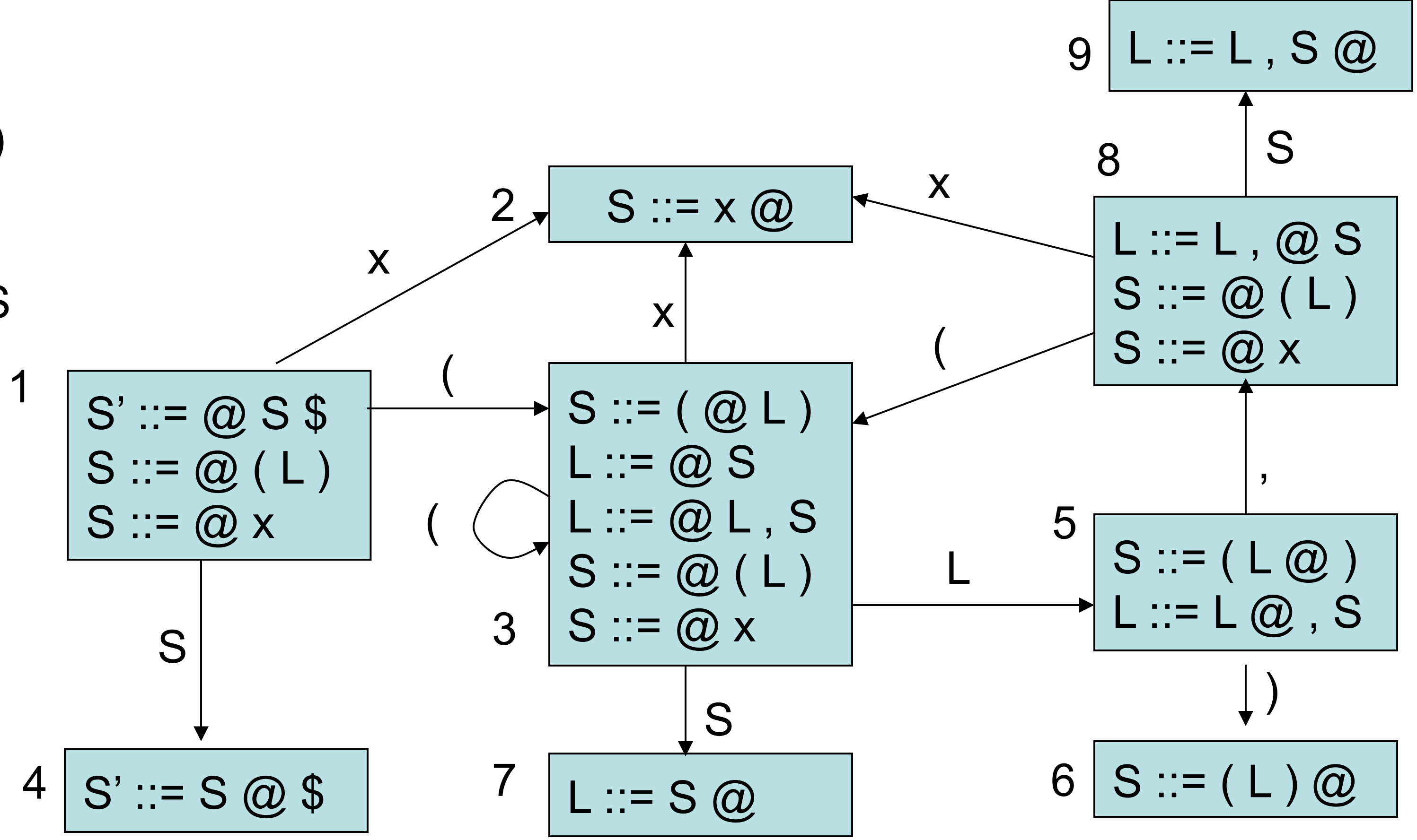


computing parse table

- 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

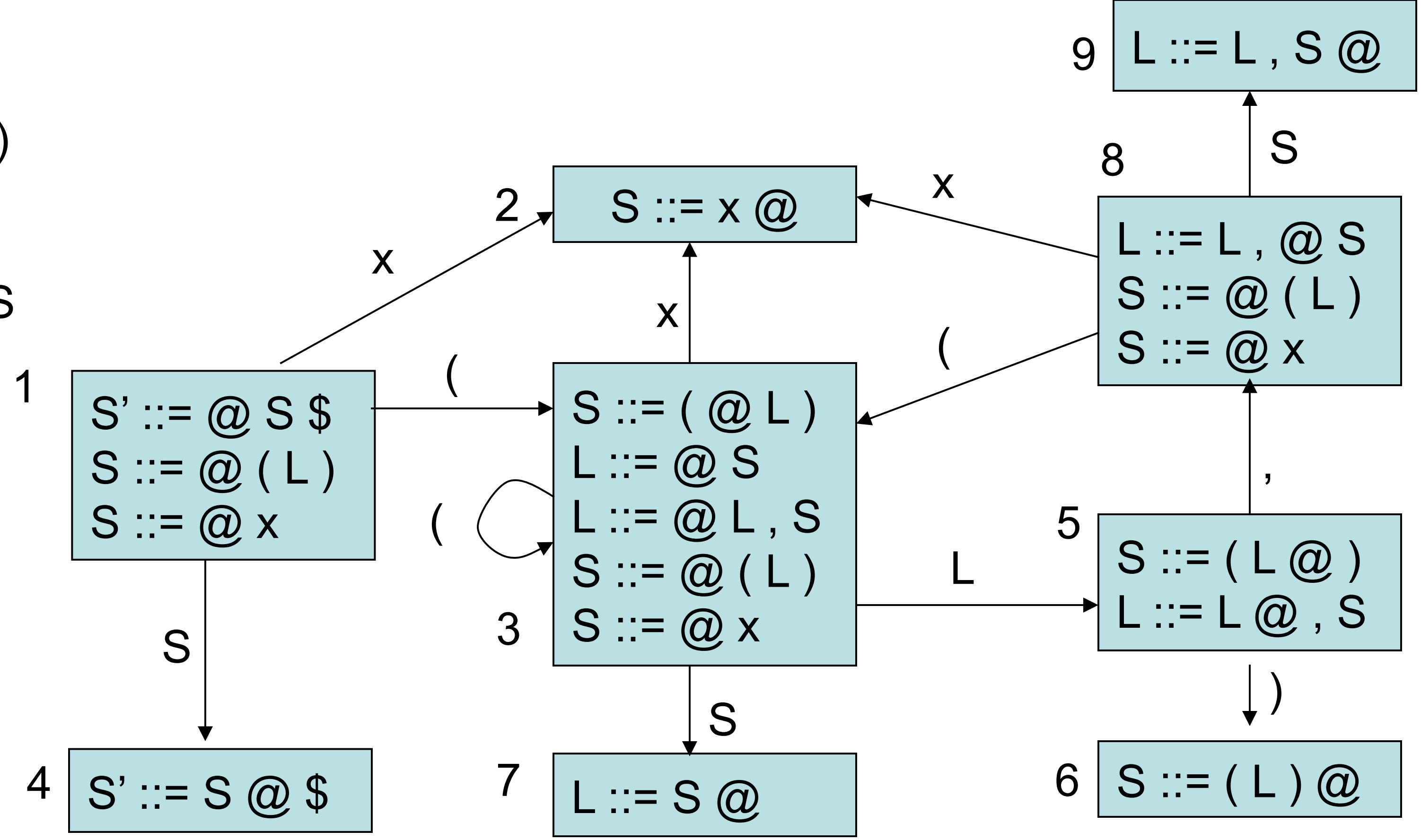
states	Terminal seen next ID, NUM, := ...	Non-terminals X,Y,Z ...
1		
2	sn = shift & goto state n	gn = goto state n
3	rk = reduce by rule k	
...	a = accept	
n	= error	

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



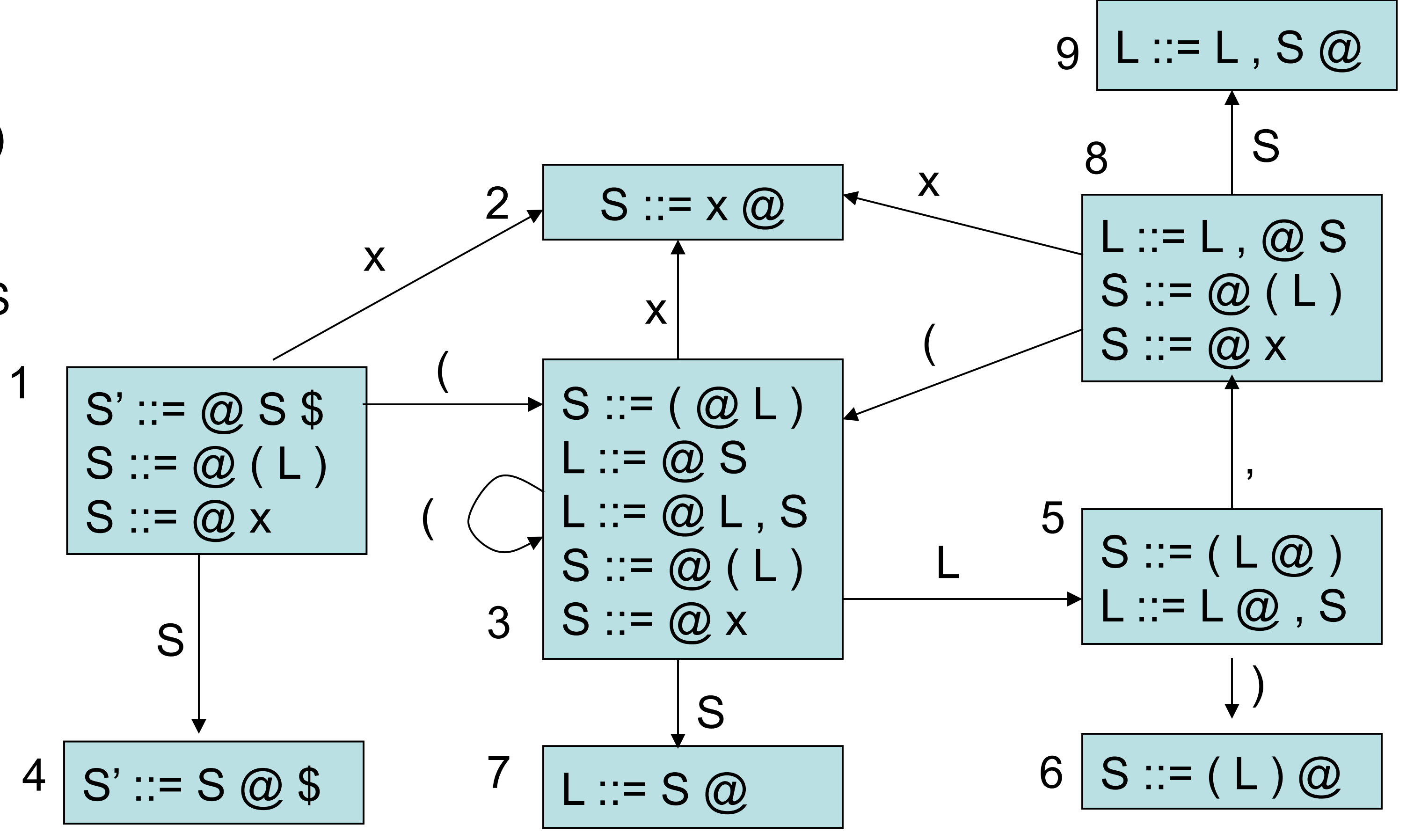
states	(	)	x	,	$	S	L
1							
2							
3							
4							
...							

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



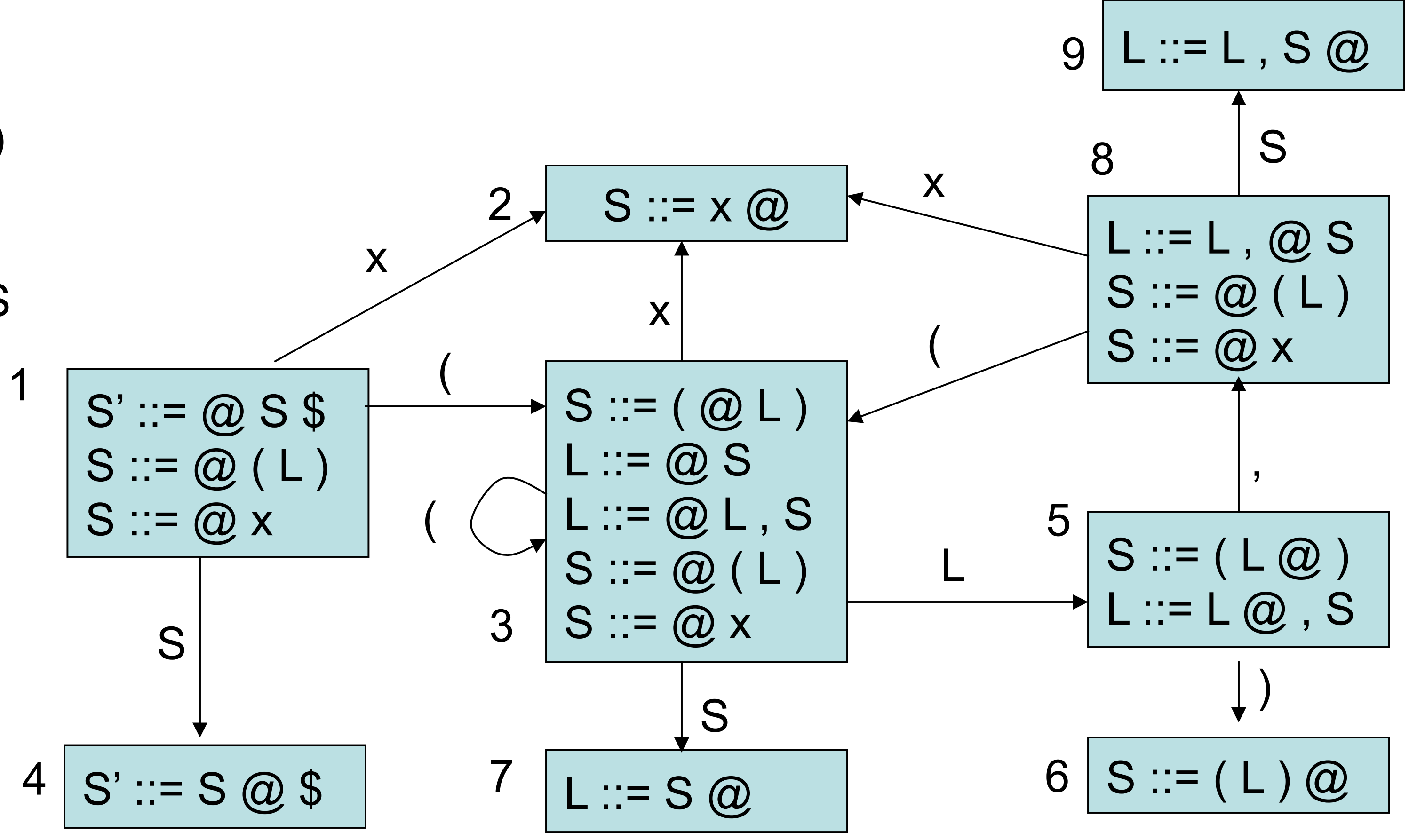
states	(	)	x	,	$	S	L
1	s3						
2							
3							
4							
...							

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



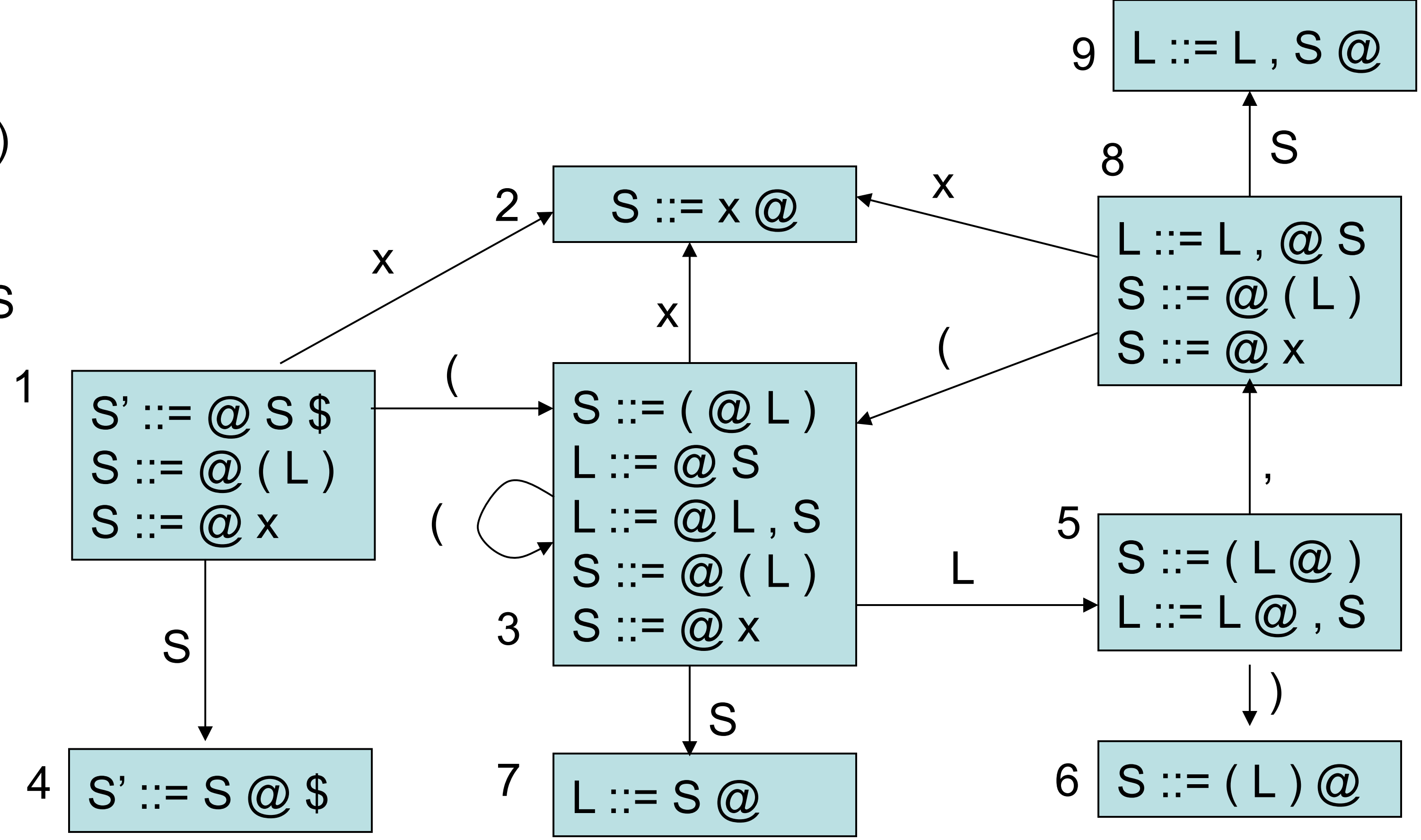
states	(	)	x	,	$	S	L
1	s3		s2				
2							
3							
4							
...							

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



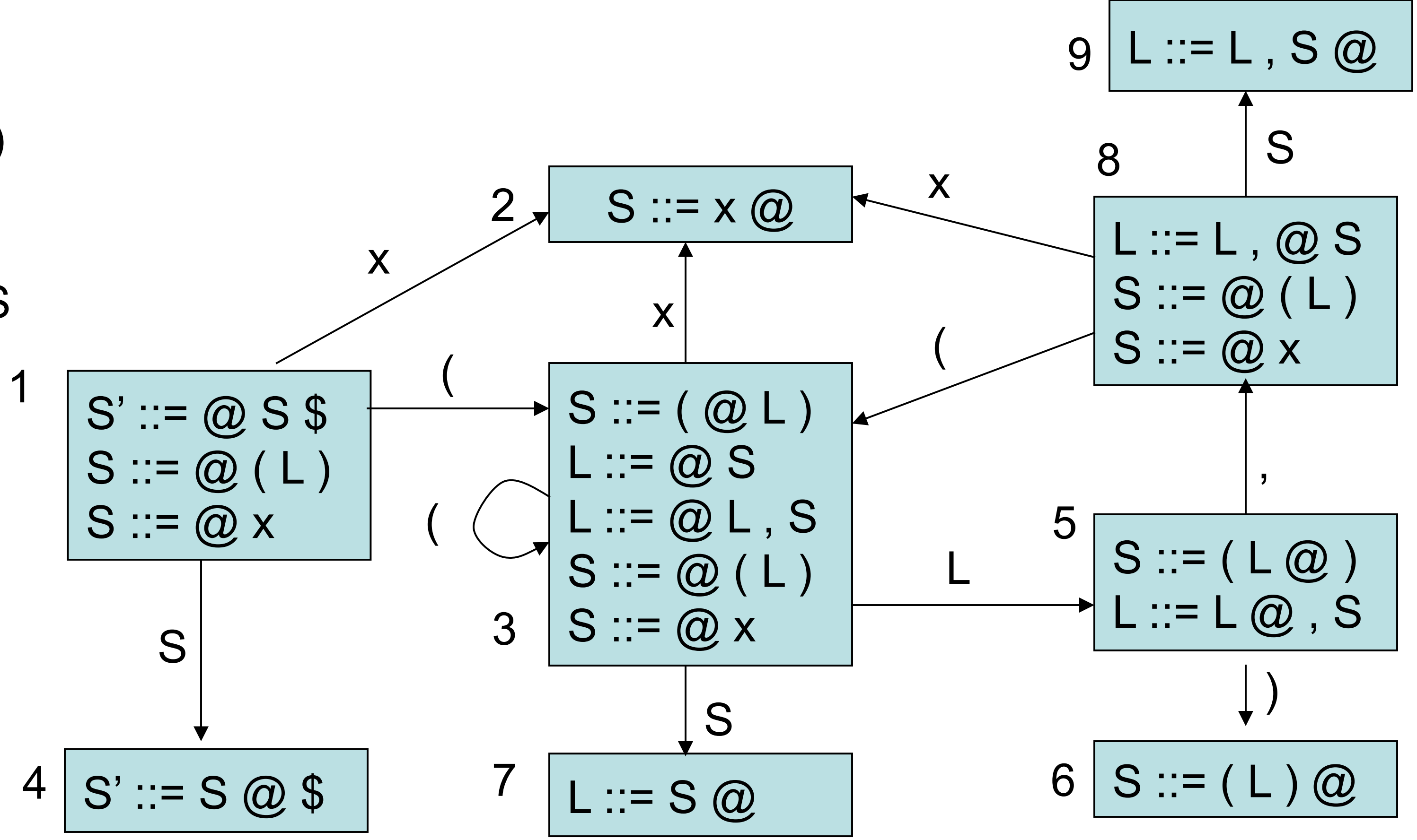
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3							
4							
...							

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



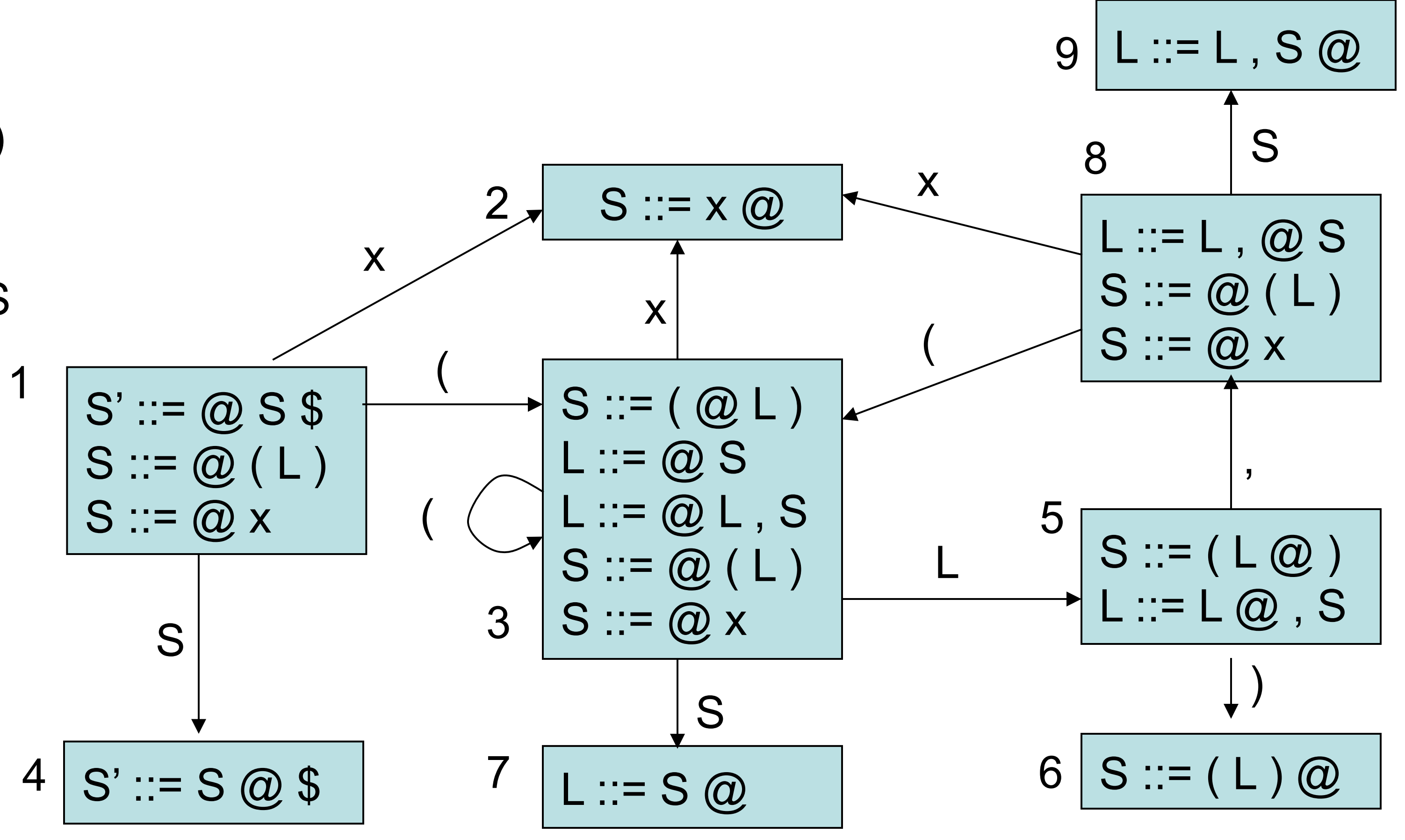
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3							
4							
...							

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



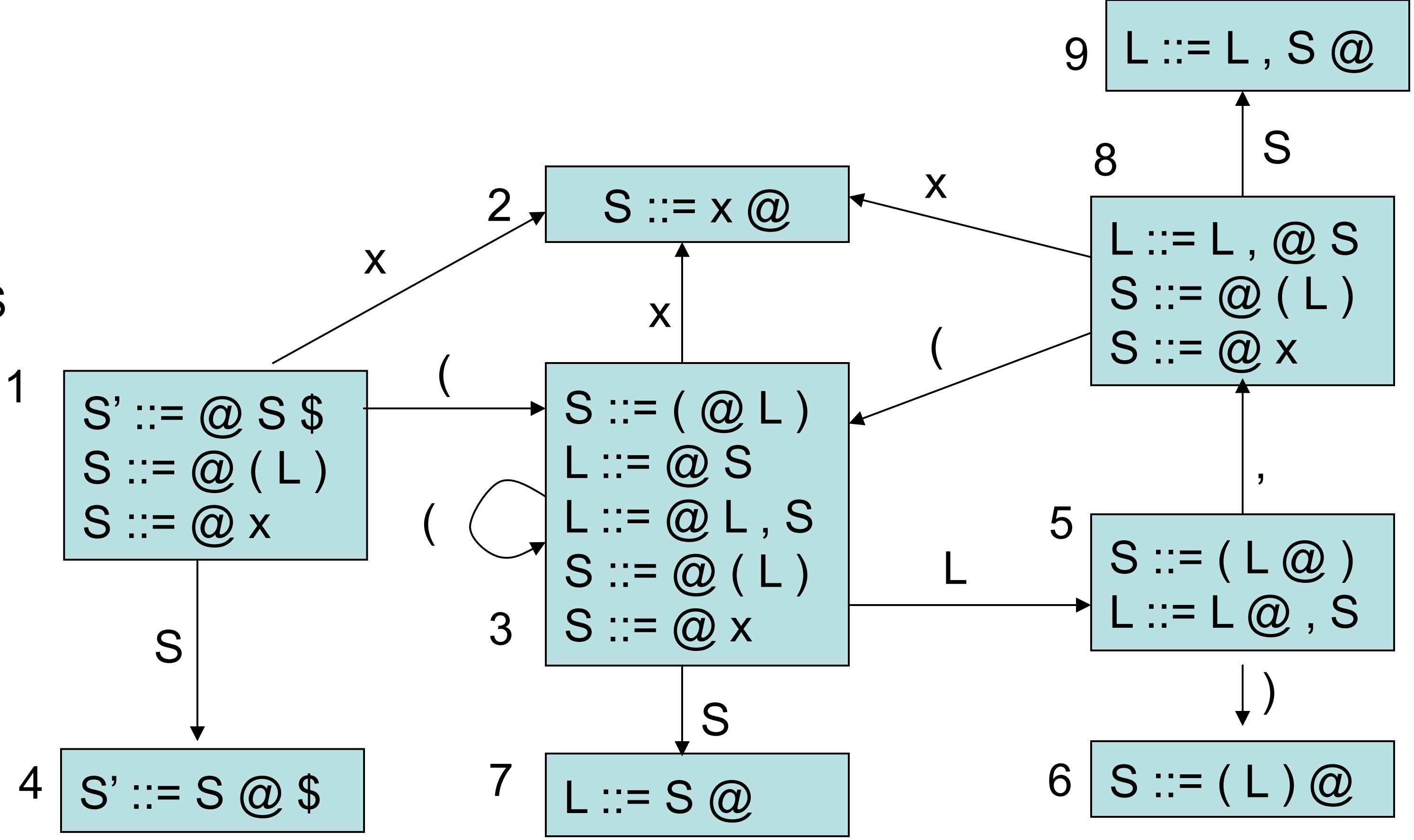
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2				
4							
...							

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

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

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

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		

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

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

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

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

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

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

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

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

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

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

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

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

$\underbrace{\text{) \$}}_{}$ 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

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

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

$\overbrace{\quad}^{\text{yet to read}}$

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

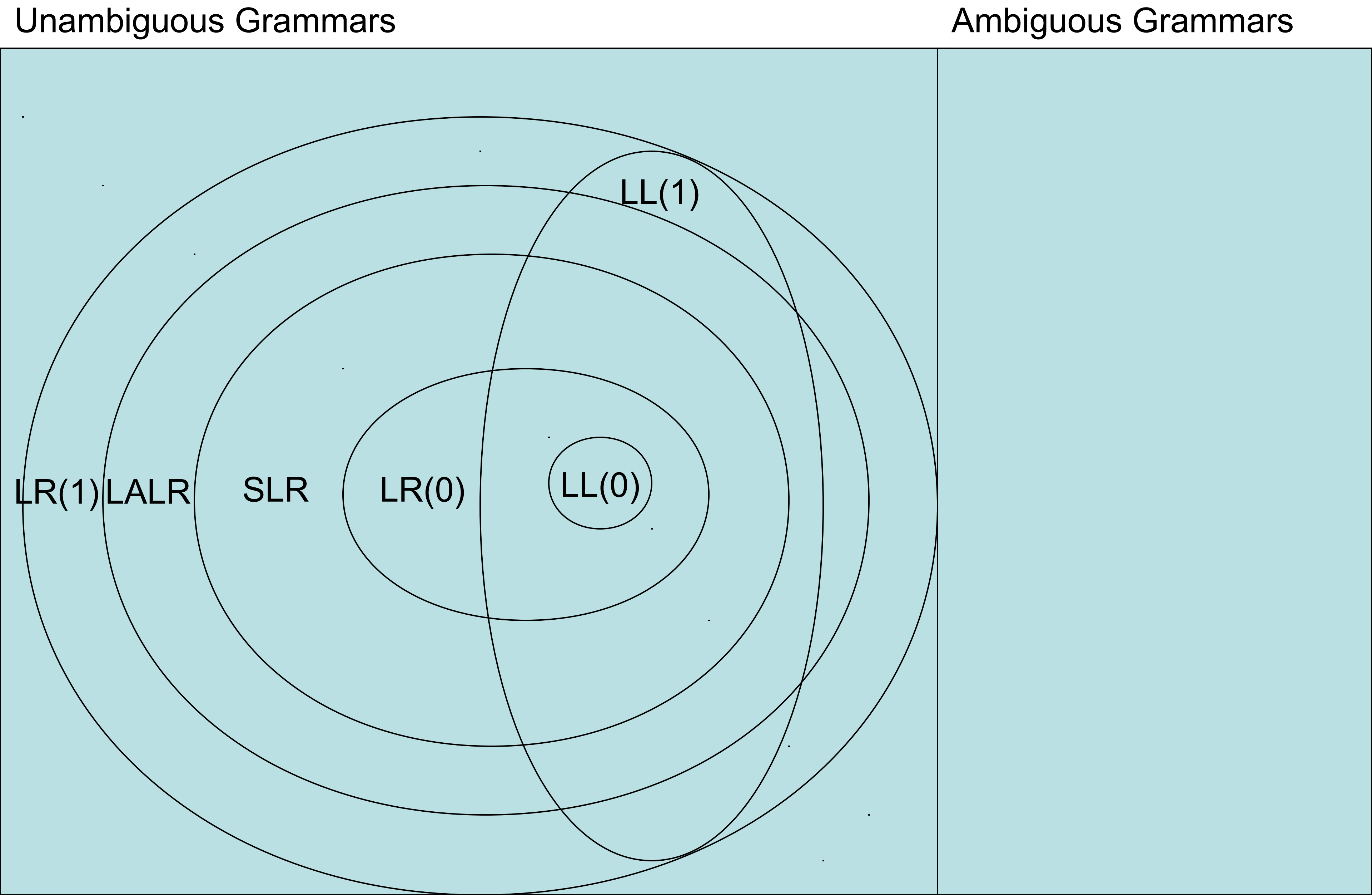
- LR(1) items

  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)