Bottom Up (**Shift Reduce)** Parsing

LR(0) and SLR(1) Parser

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.

$S \Rightarrow ... \Rightarrow \omega$ (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

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.

LR Parsing Techniques

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

LR Parsing Types

There are three types of LR parsers:

- *LR(k), simple LR(k),* and *lookahead LR(k)* (abbreviated to LR(k), SLR(k), LALR(k))).
- We will usually only concern ourselves with 0 or 1 **tokens of lookahead**, but the techniques do generalize to **k > 1**.
- The different classes of parsers all operate the same way (Driven by their action and goto tables), but they differ in **how their action and goto tables** are constructed, and the **size of those tables**.

LR(0) **vs.** LR(1)

LR(0) parsing:

- The simplest and the weakest of all the LR parsing methods
- Not used much in practice because of its limitations
- LR(0) parses without using any look-ahead at all

LR(1) parsing:

- Adding just one token of look-ahead to get LR(1) vastly increases the parsing power

LR(0) **vs.** LR(1)

- Very few grammars can be parsed with LR(0), but most unambiguous CFGs can be parsed with LR(1)

- The **drawback of adding the look-ahead** is that the algorithm becomes somewhat <u>more complex</u> and the parsing table gets much, <u>much bigger</u>

- The full LR(1) parsing table for a typical programming language has many <u>thousands of states</u> compared to the <u>few hundred needed for</u> LR(0)

LR(k) vs. SLR(k) vs. LALR(k)

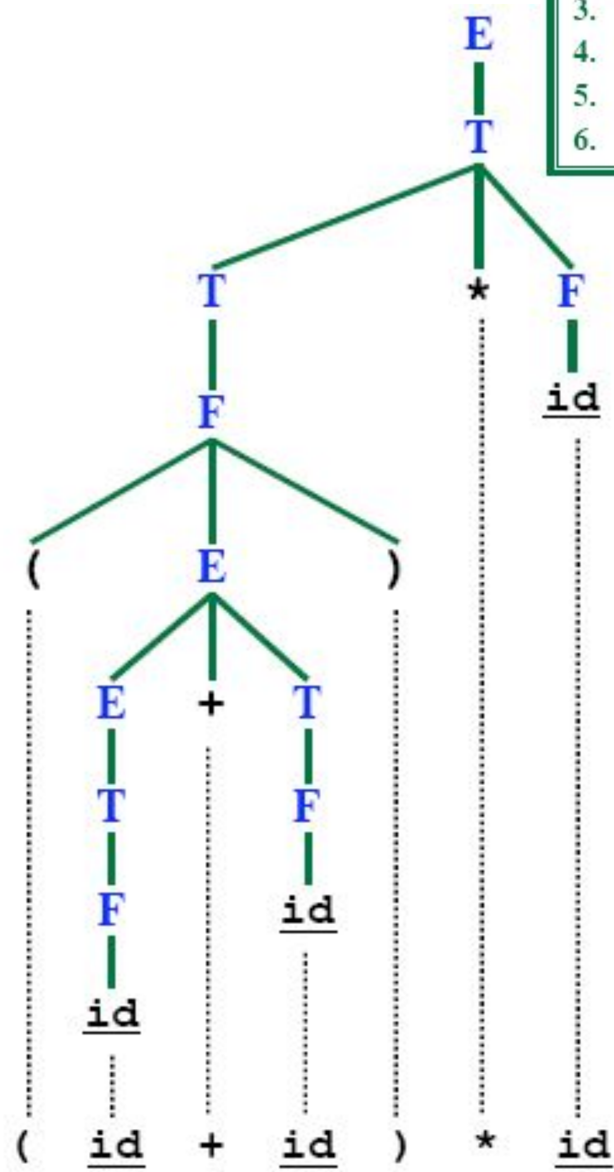
- A compromise in the middle is found in the **two variants**: **SLR(1) and LALR(1)** which also use:
  - One token of lookahead but employ techniques to keep the table as small as LR(0)

- SLR(k) is an improvement over LR(0) but much weaker than full LR(k) in terms of the number of grammars for which it is applicable.
- LALR(k) parses a larger set of languages than SLR(k) but not quite as many as LR(k).
- LALR(1) is the method used by the **yacc** parser generator

Rightmost Derivation

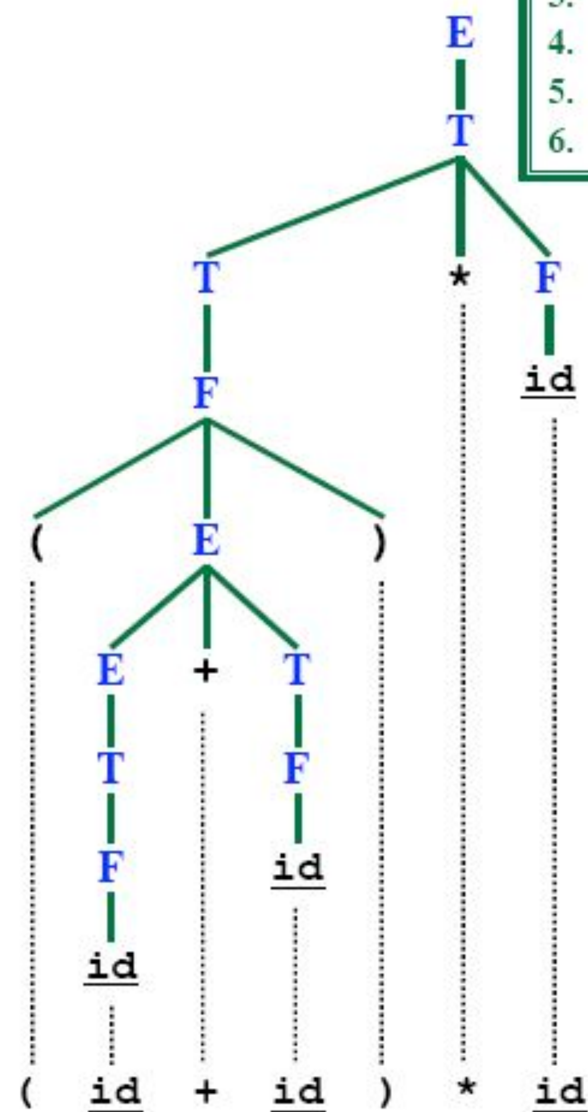
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

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



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	(E) * id
(F + id) * id	F * id
(T + id) * id	T * id
(E + id) * id	T * F
(E + F) * id	T
(E + T) * id	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
□ <u>a</u> 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

T

a 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

T

a b 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:

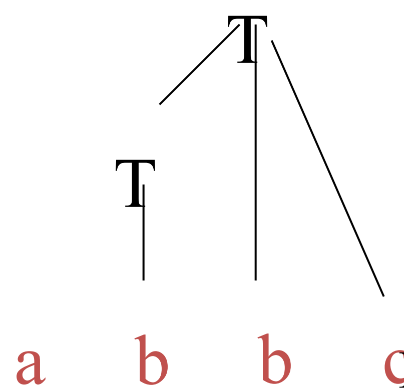
S □ a T R e
□ a T **d** e
□ <u>a **T b c**</u> d e
□ **a b** b c d e

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:

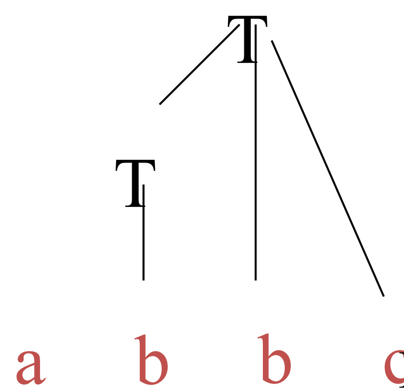
S □ a T R e
□ a T d e
□ a **T b c** d e
□ **a b** b c d e

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:

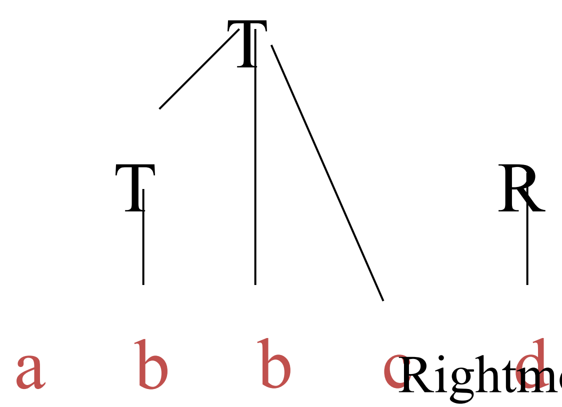
S □ a T R e
□ <u>a T **d**</u> e
□ a **T b c** d e
□ **a b** b c d e

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:

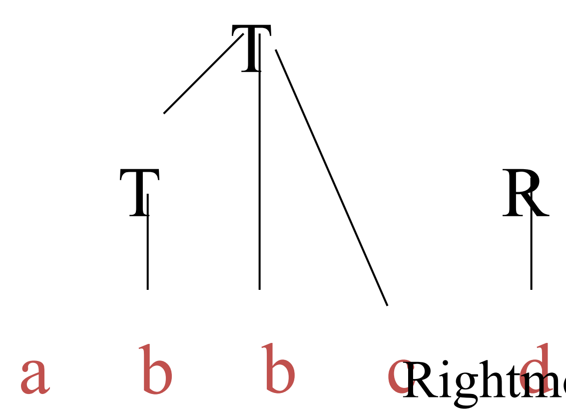
S □ <u>a T R</u> e
□ a T **d** e
□ a **T b c** d e
□ **a b** b c d e

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



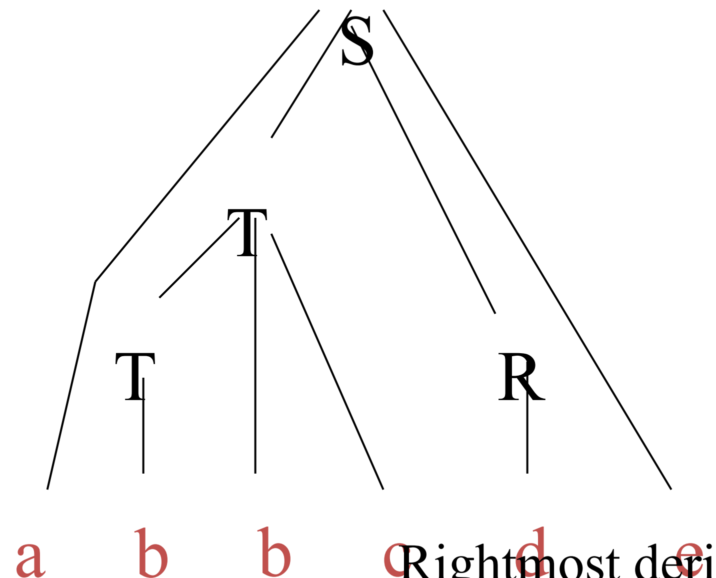
Rightmost derivation:

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:



- □ 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

Example Shift-Reduce Parsing

Consider the grammar:

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

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

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 → **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*

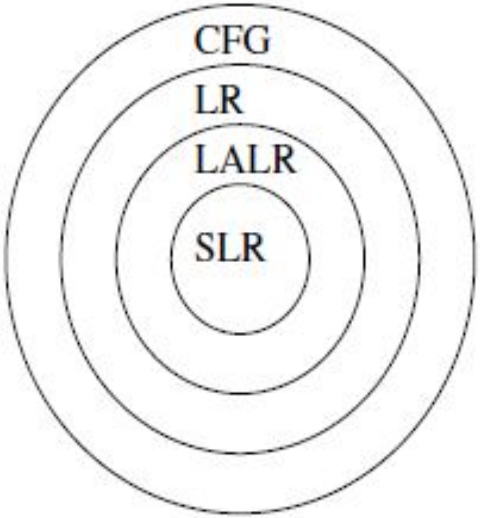
STACK	**INPUT**
….**id** (**id**	, **id**) ….$

- 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.



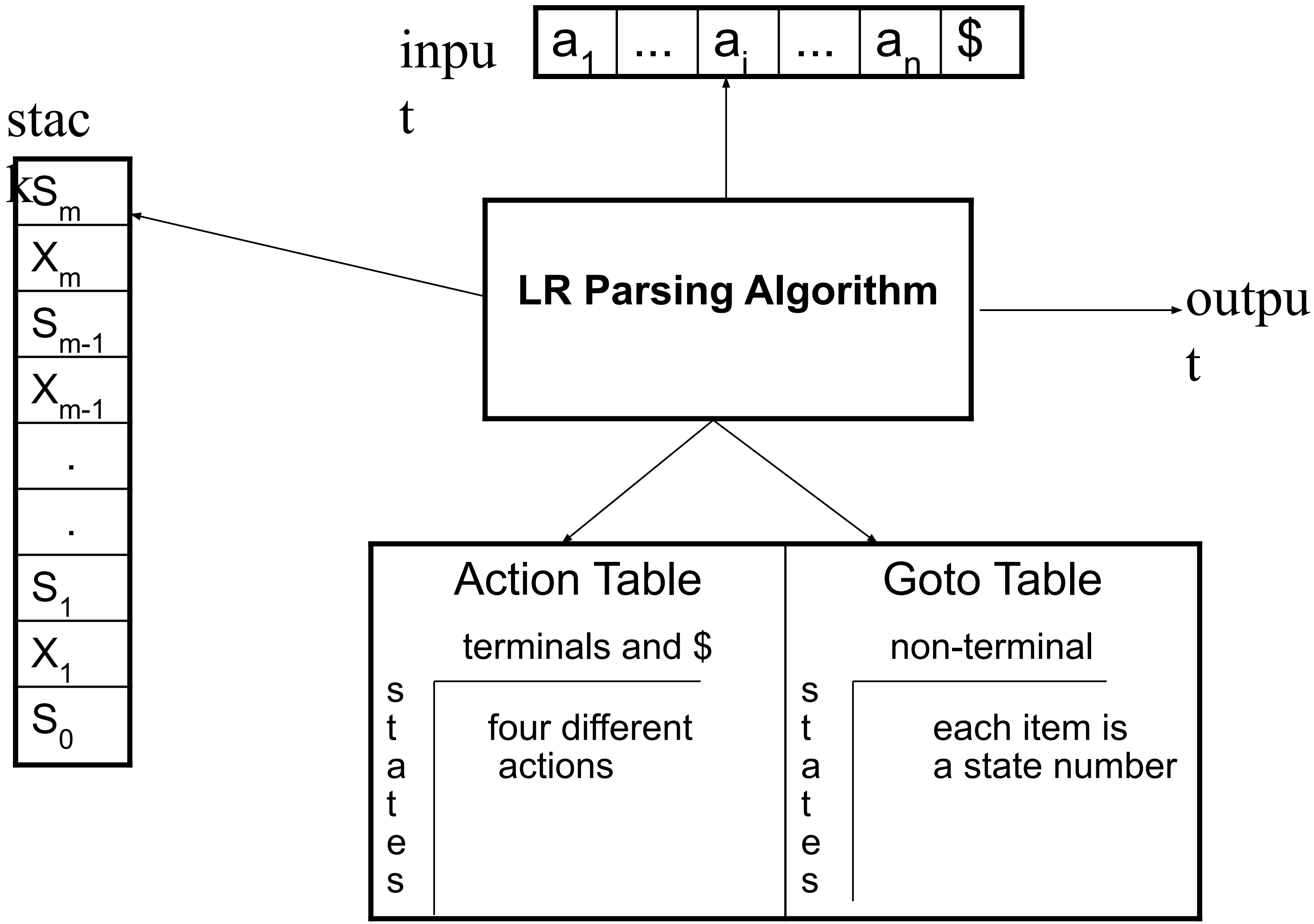
2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - LR – most general LR parser
     - LALR – intermediate LR parser (lookhead LR parser)
   - SLR, LR and LALR work same, only their parsing tables are different.

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



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(\underbrace{S_o\ X_1\ S_1\ \ldots\ X_m\ S_m}_{\text{Stack}},\ a_i\ \underbrace{a_{i+1}\ \ldots\ a_n\ \$}_{\text{Rest of Input}})$$

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1\ \ldots\ X_m\ a_i\ a_{i+1}\ \ldots\ a_n\ \$$$

Actions of A LR-Parser

1. **shift**

   If action[S_m, a_i] = shift s, the parser executes a shift move, entering the configuration

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) □ (S_o X_1 S_1 ... X_m S_m a_i s, a_{i+1} ... a_n \$)

   Here the parser has shifted both the current input symbol a_i and the next state s, which is given in action[S_m, a_i], onto the stack; a_{i+1} becomes the current input symbol.

2. **reduce**

   If action[S_m, a_i] = **A→β** then the parser executes a reduce move, entering the configuration

   (S_o X_1 S_1 ... X_m S_m, a_i a_{i+1} ... a_n \$) □ (S_o X_1 S_1 ... X_{m-r} S_{m-r} A s, a_i ... a_n \$)

   where s = goto[S_{m-r}, A] and r is the length of **β**, the right side of the production.

   – The parser first popped 2r symbols off the stack, exposing state S_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for goto [s_{m-r}, A], onto the stack.
   – Output is the reducing production reduce A→β

Actions of A LR-Parser (continued)

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

LR Parsing Algorithm

Input:

- String to parse, w
- Precomputed ACTION and GOTO tables for grammar G

Output:

- Success, if $w \in L(G)$ plus a trace of rules used
- Failure, if syntax error

```
push state 0 onto the stack
loop
  s = state on top of stack
  c = next input symbol
  if ACTION[s,c] = "Shift N" then
    push c onto the stack
    advance input
    push state N onto stack
  elseif ACTION[s,c] = "Reduce R"
   then
    let rule R be A → β
    pop 2*|β| items off the stack
    s' = state now on stack top
    push A onto stack
    push GOTO[s',A] onto stack
    print "A → β"
  elseif ACTION[s,c] = "Accept"
   then
    return success
  else
    print "Syntax error"
    return
  endIf
endLoop
```

Bottom-Up Parsing: LR(0) Table Construction

Constructing LR 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 LR 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 LR Parsing Tables – LR(0) Item

- 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' → S

"Goal"

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

```
0. S' → E
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → ( E )
6. F → id
```

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

S' → •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

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T^*F$

$T \to F$

$F \to (E)$

$F \to id$

closure($\{E' \to \bullet E\}$) =

$\{ \; E' \to \bullet E$ ← kernel items

$E \to \bullet E+T$

$E \to \bullet T$

$T \to \bullet T^*F$

$T \to \bullet F$

$F \to \bullet (E)$

$F \to \bullet id \; \}$

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:

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

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

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

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

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

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

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**.

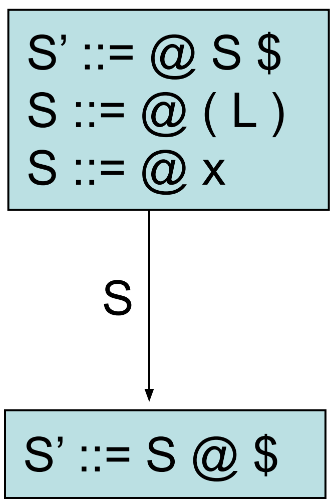
Grammar:

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

S’ ::= @ S $
S ::= @ (L)
S ::= @ x

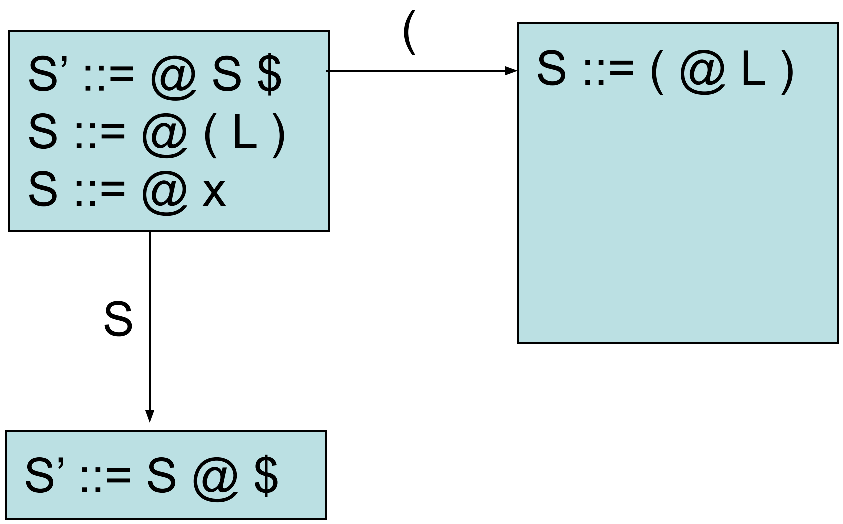
Grammar:

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



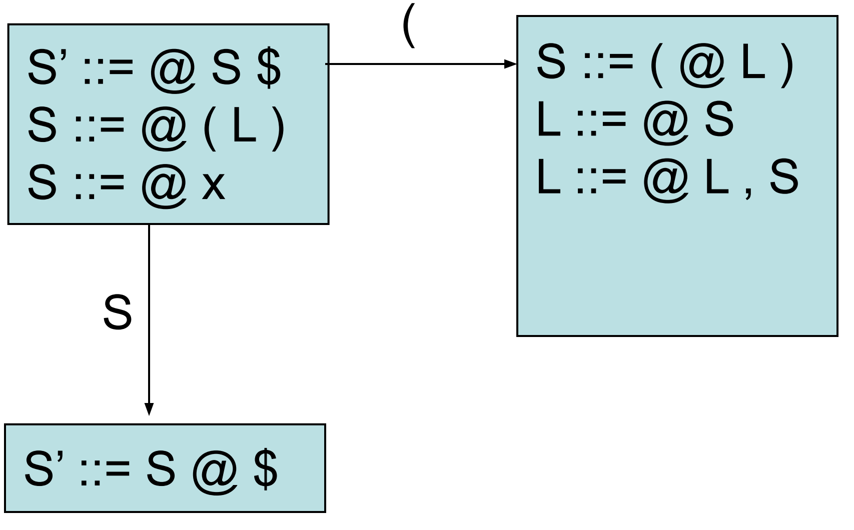
Grammar:

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



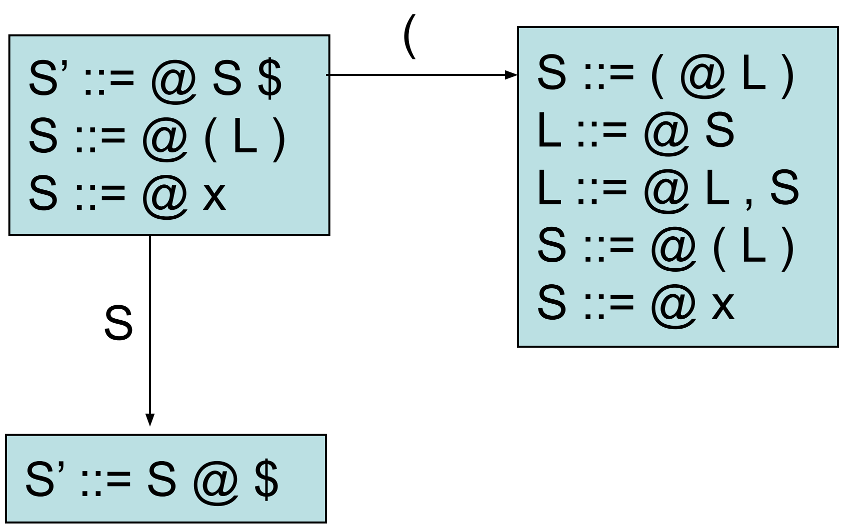
Grammar:

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



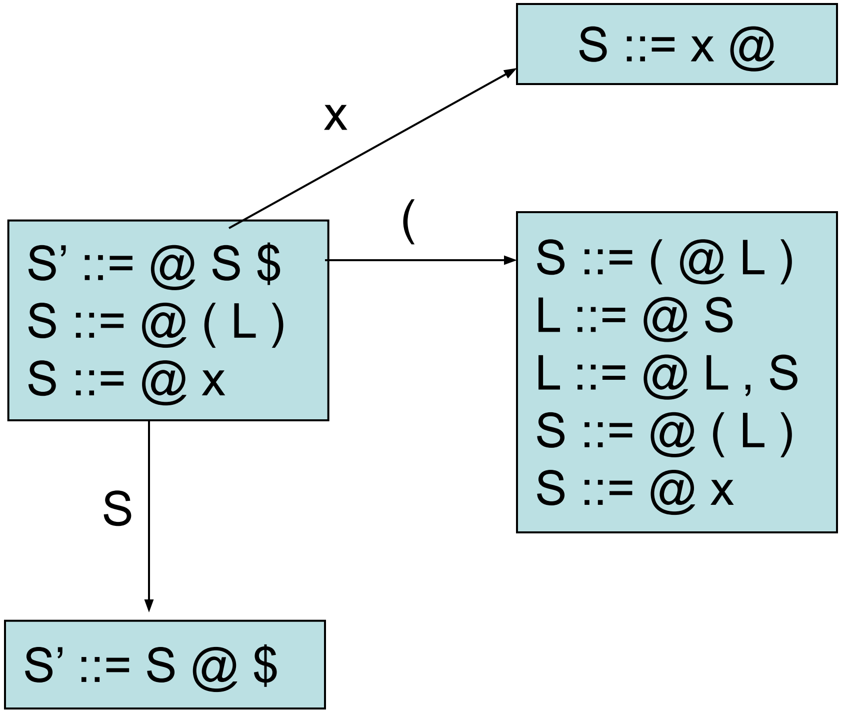
Grammar:

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



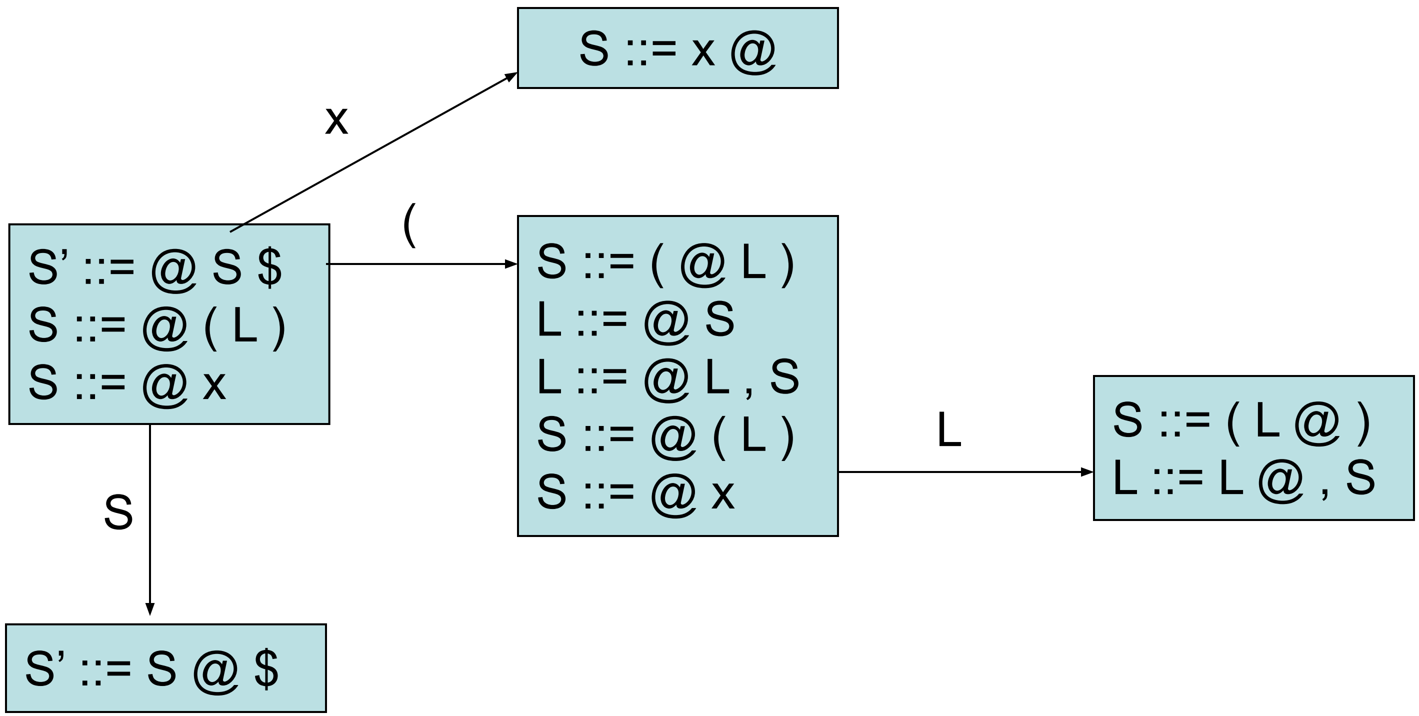
Grammar:

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



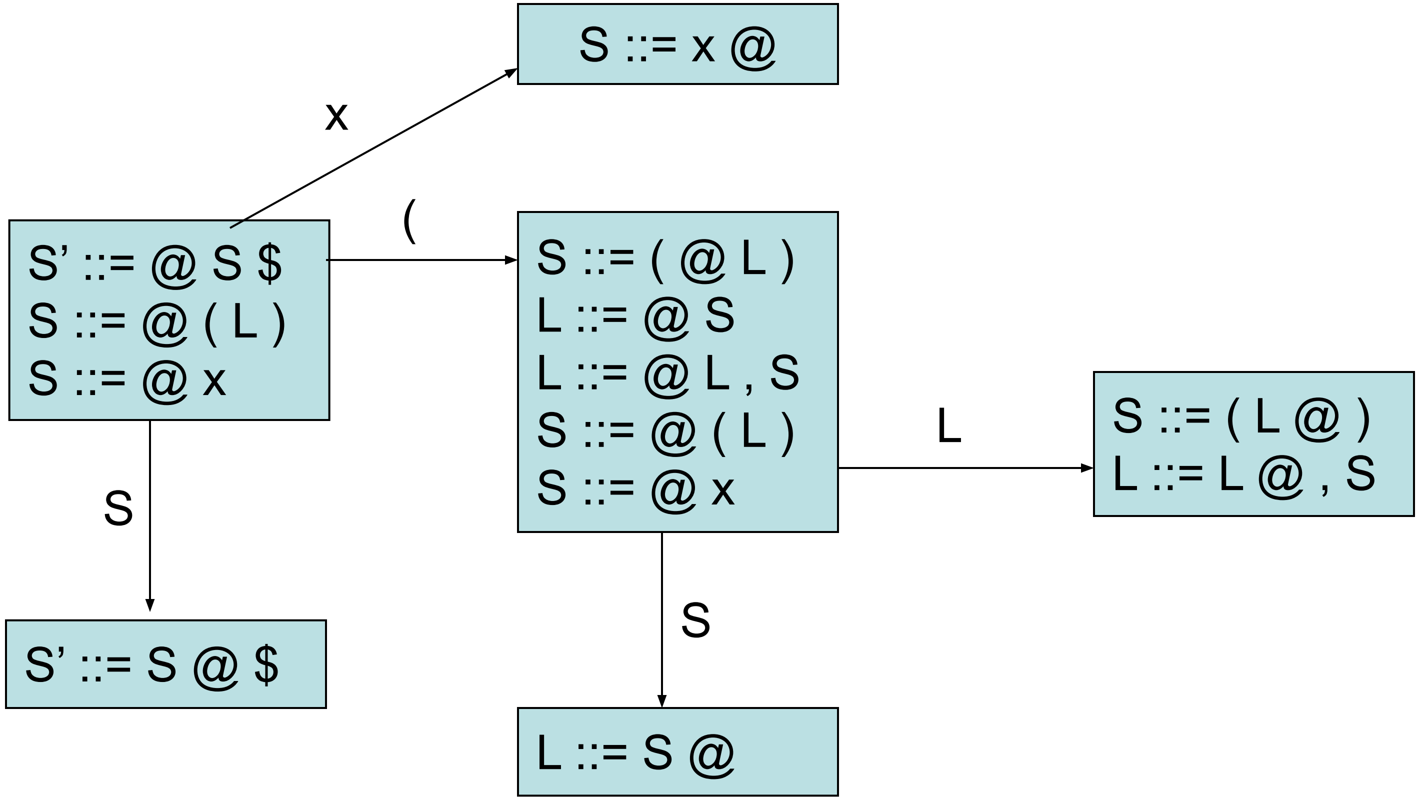
Grammar:

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



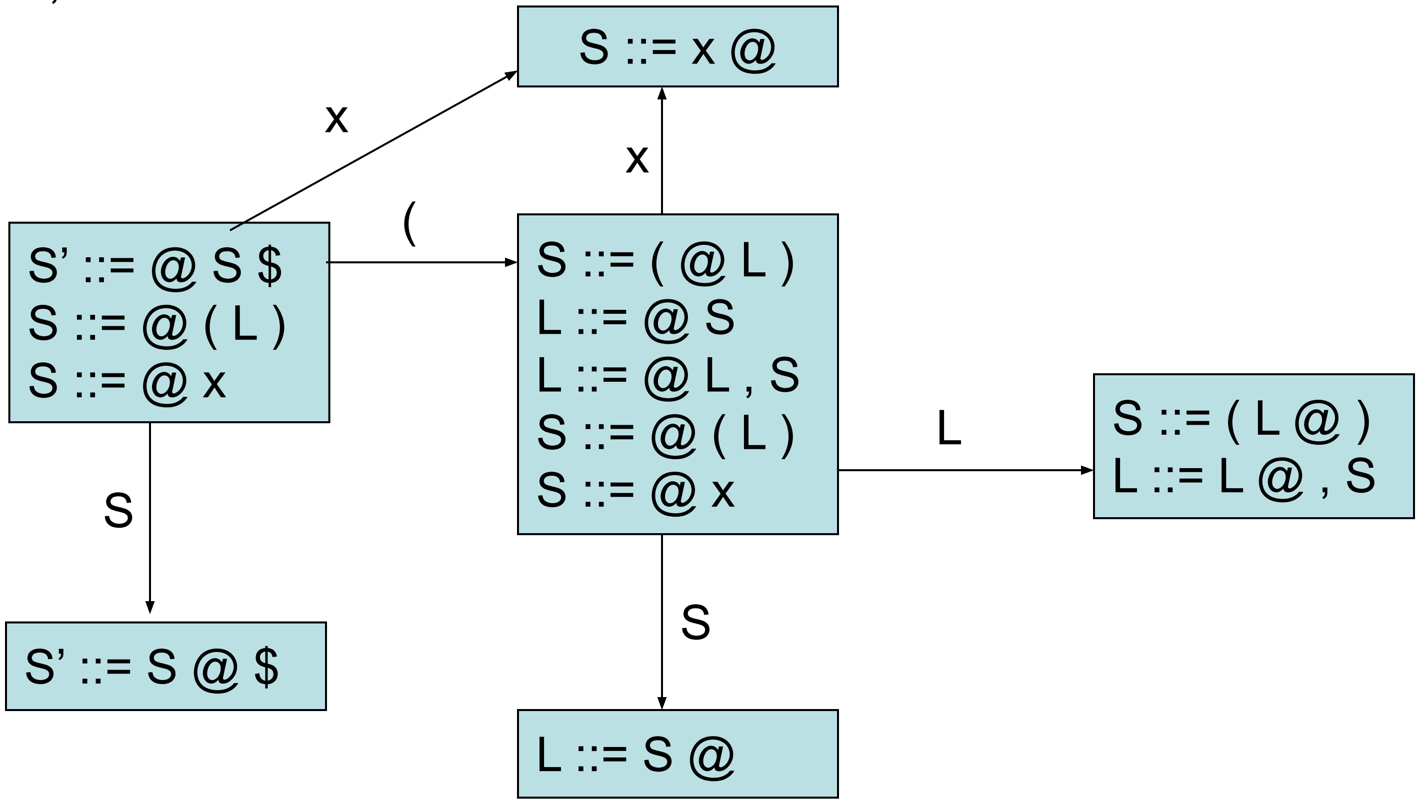
Grammar:

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



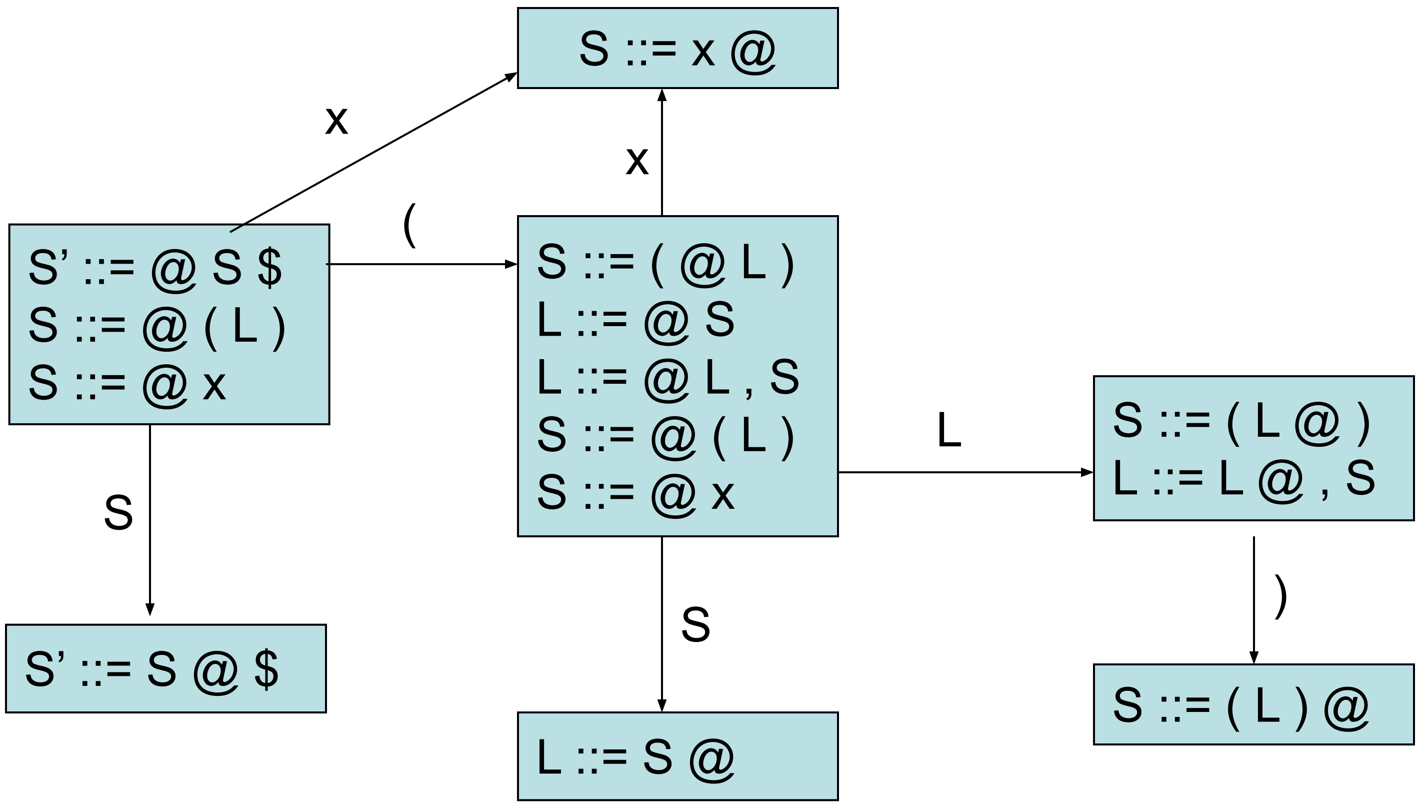
Grammar:

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



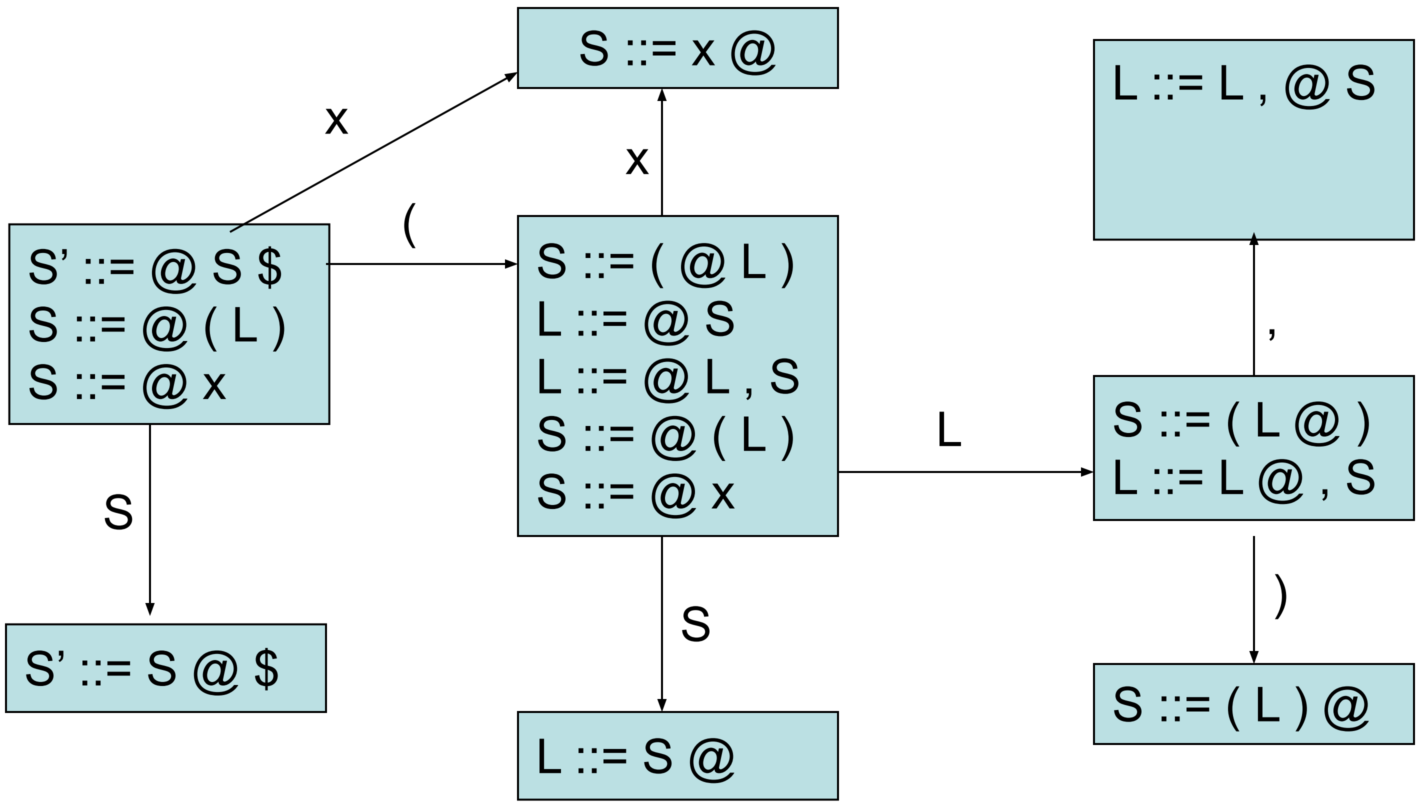
Grammar:

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



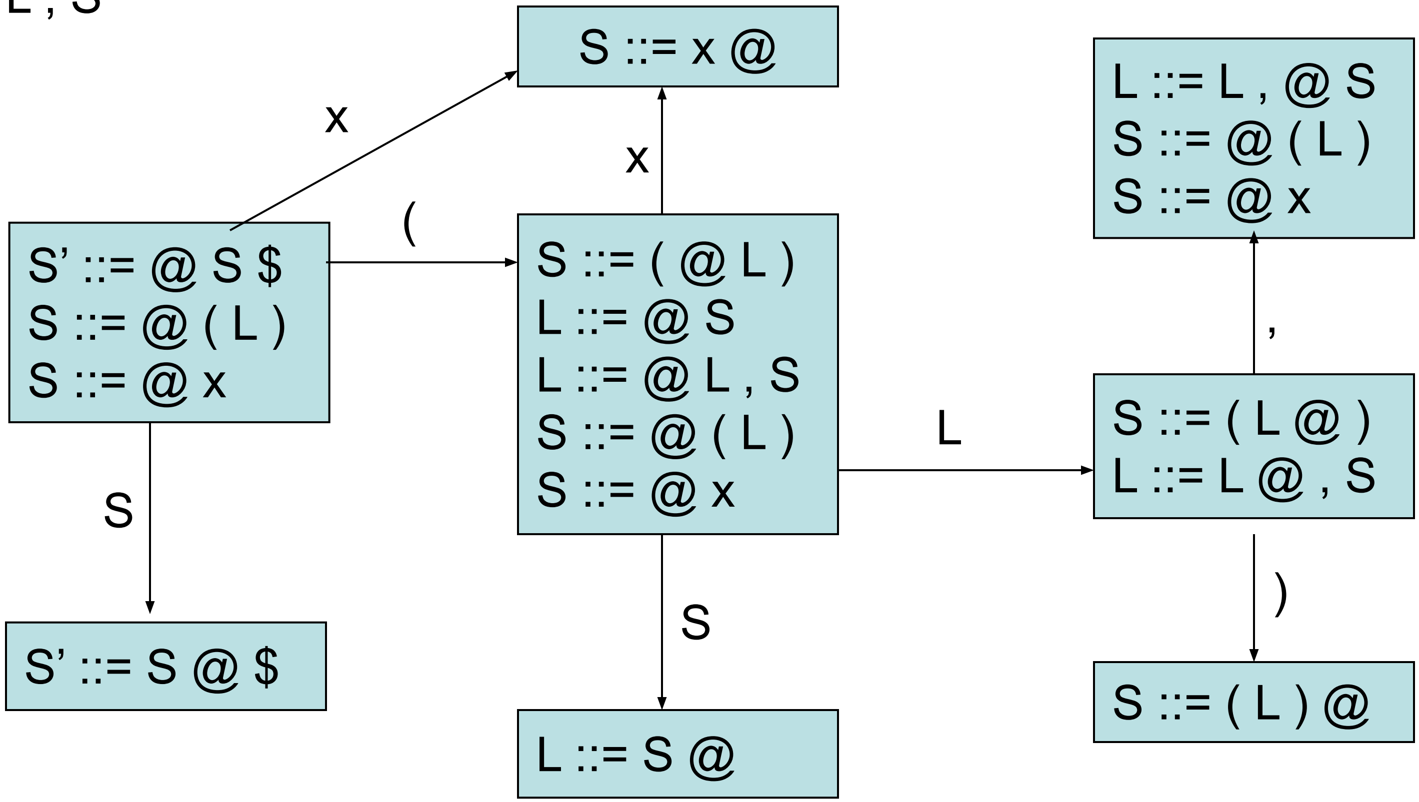
Grammar:

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



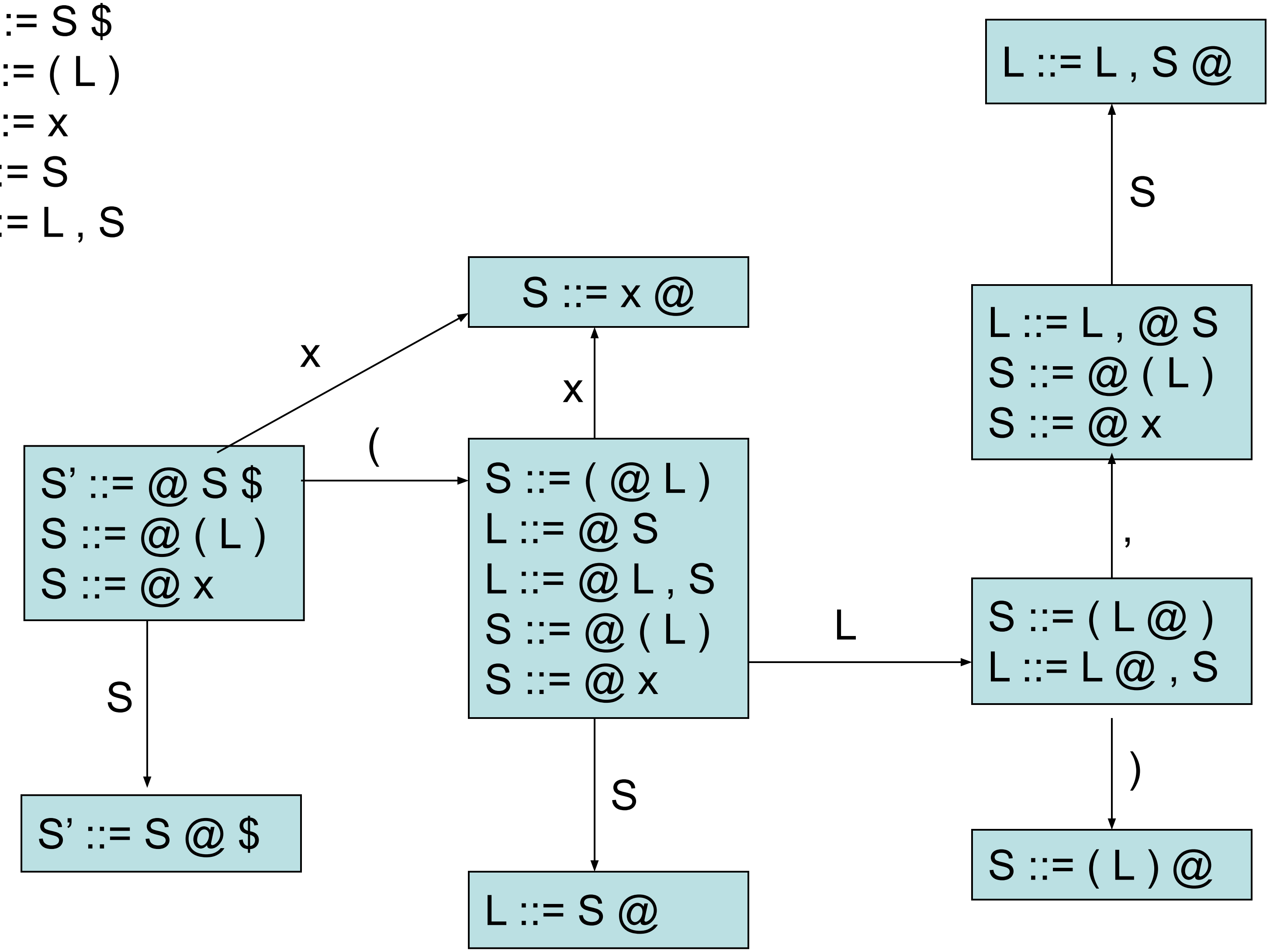
Grammar:

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



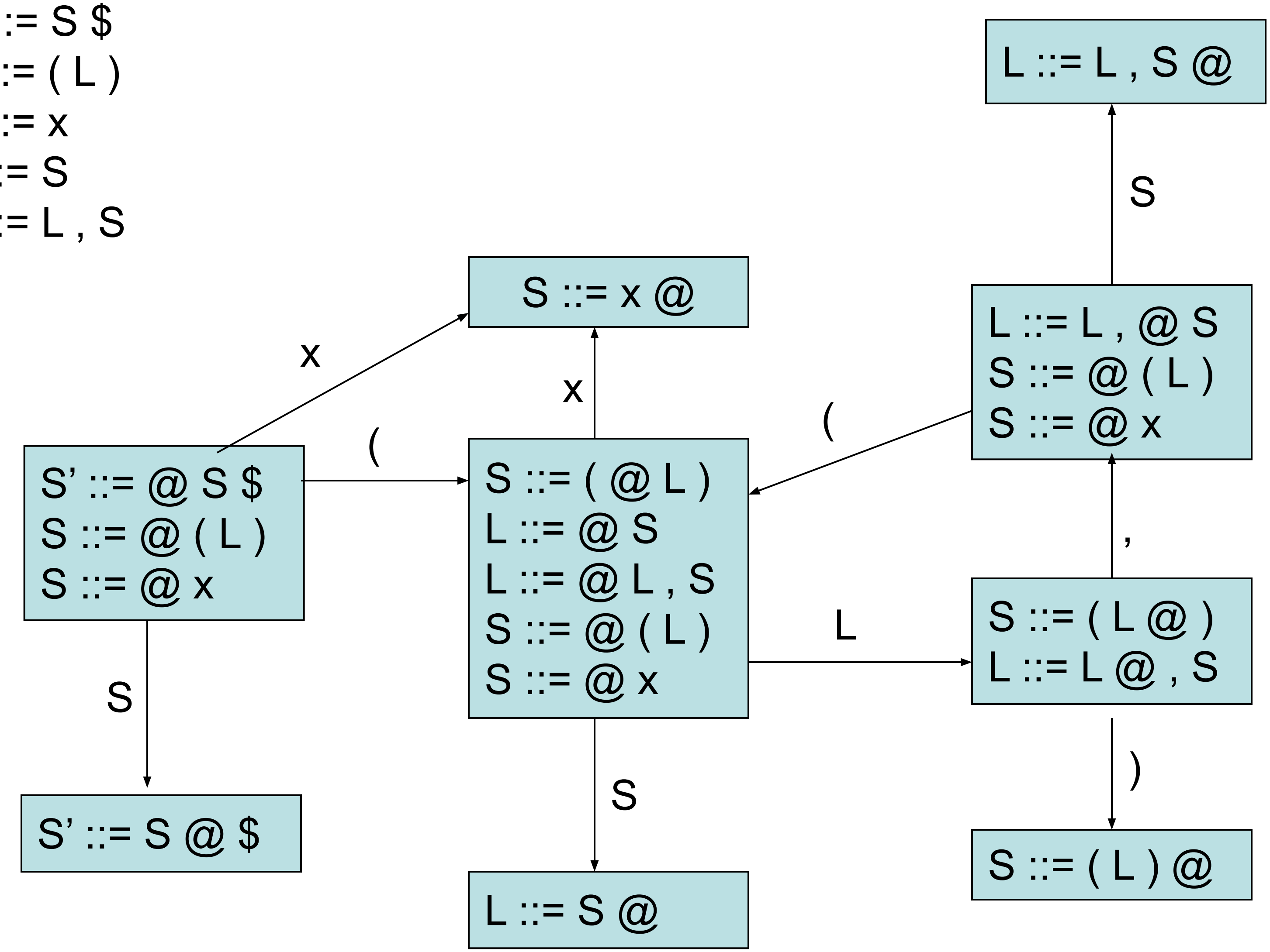
Grammar:

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



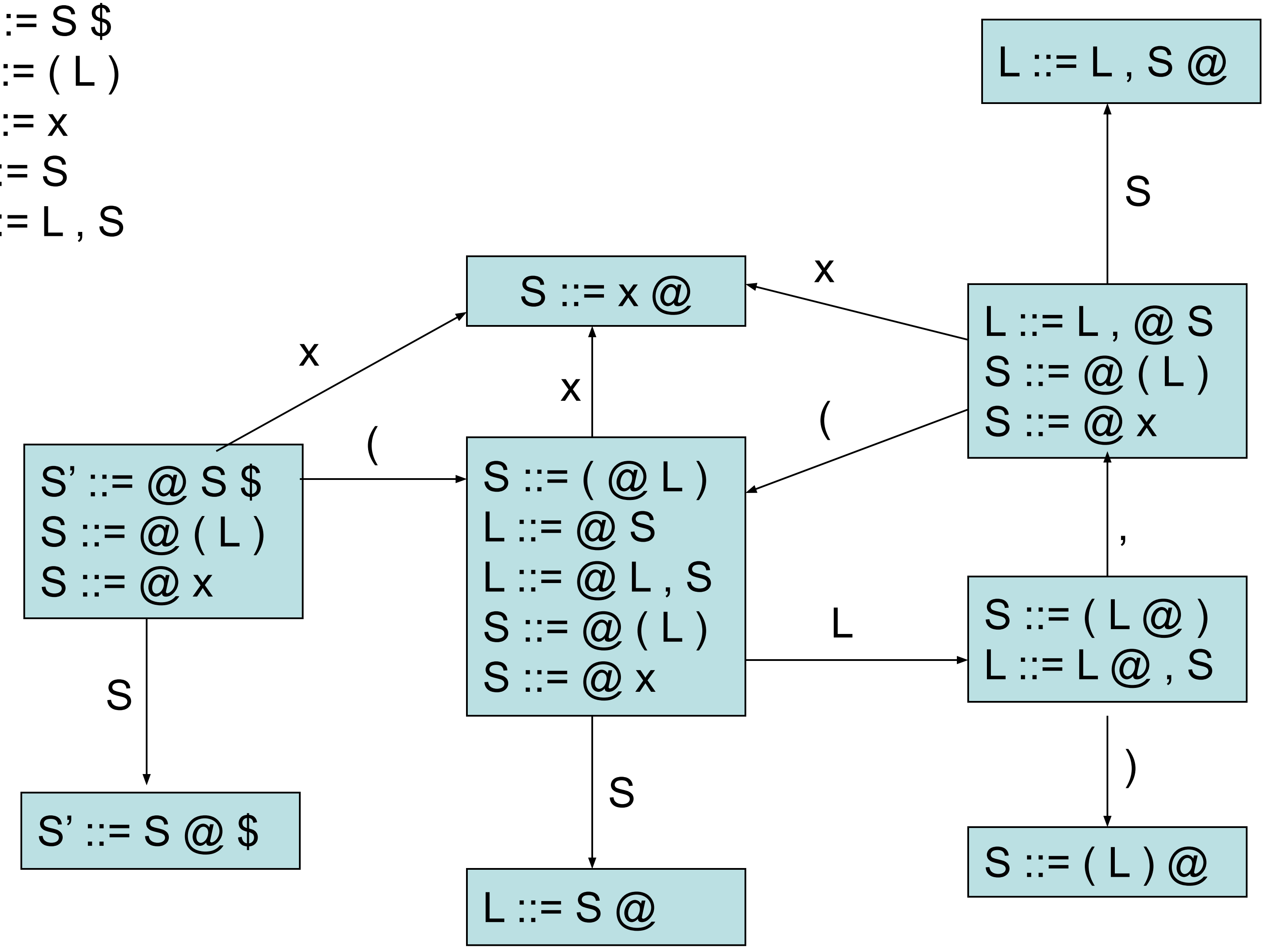
Grammar:

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



Grammar:

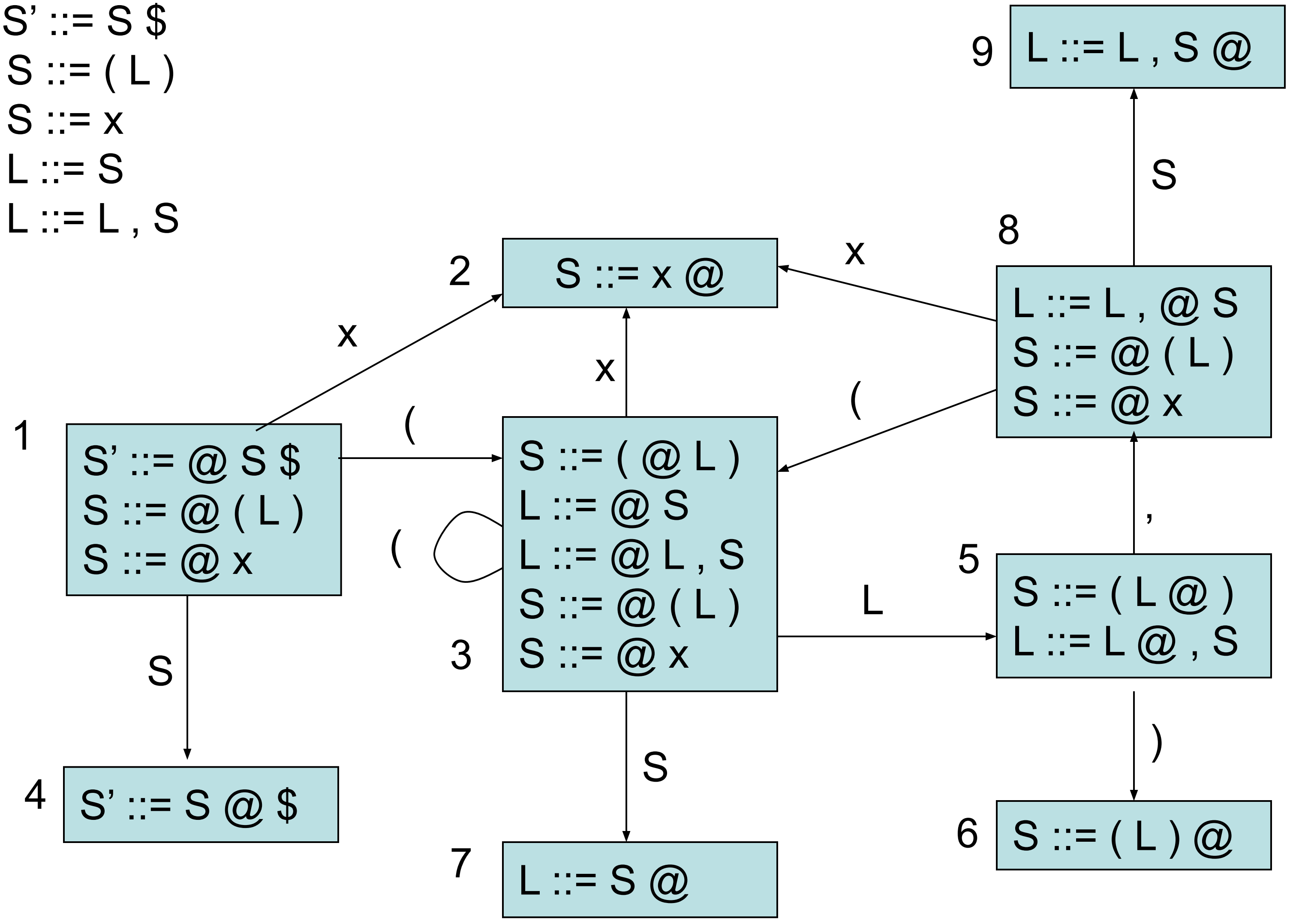
0. S’ ::= S $
1. S ::= (L)
2. S ::= x
3. L ::= S
4. L ::= L , S



Grammar:

Assigning numbers to states:

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



Computing LR(0) 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 LR(0) 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

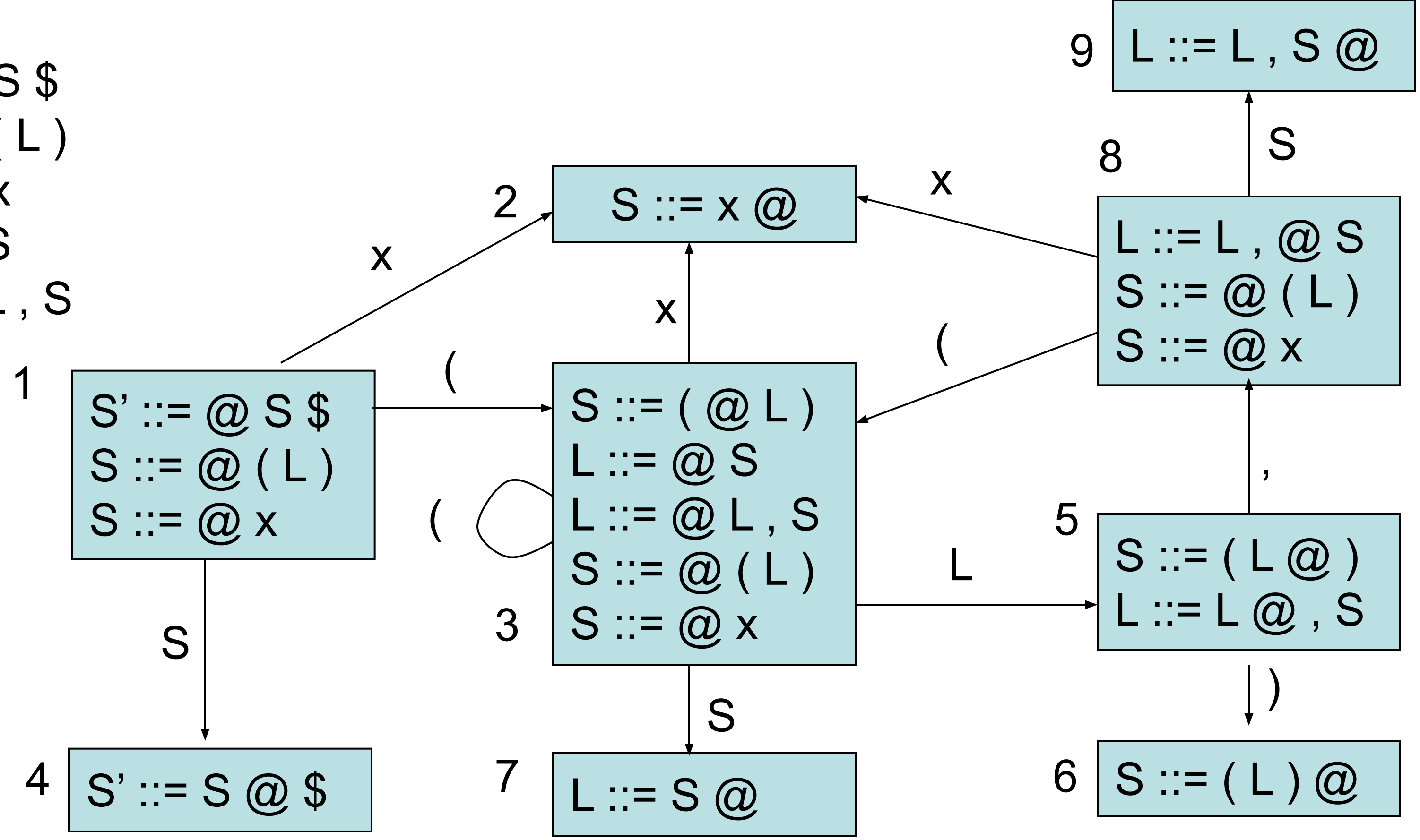
states	**Terminal** seen next ID, NUM, := ...	**Non-terminals** X,Y,Z ...
1		
2	**sn** = shift & goto state n	**gn** = goto state n
3	**rk** = reduce by rule k	
...	**a** = accept	
n	= error	

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

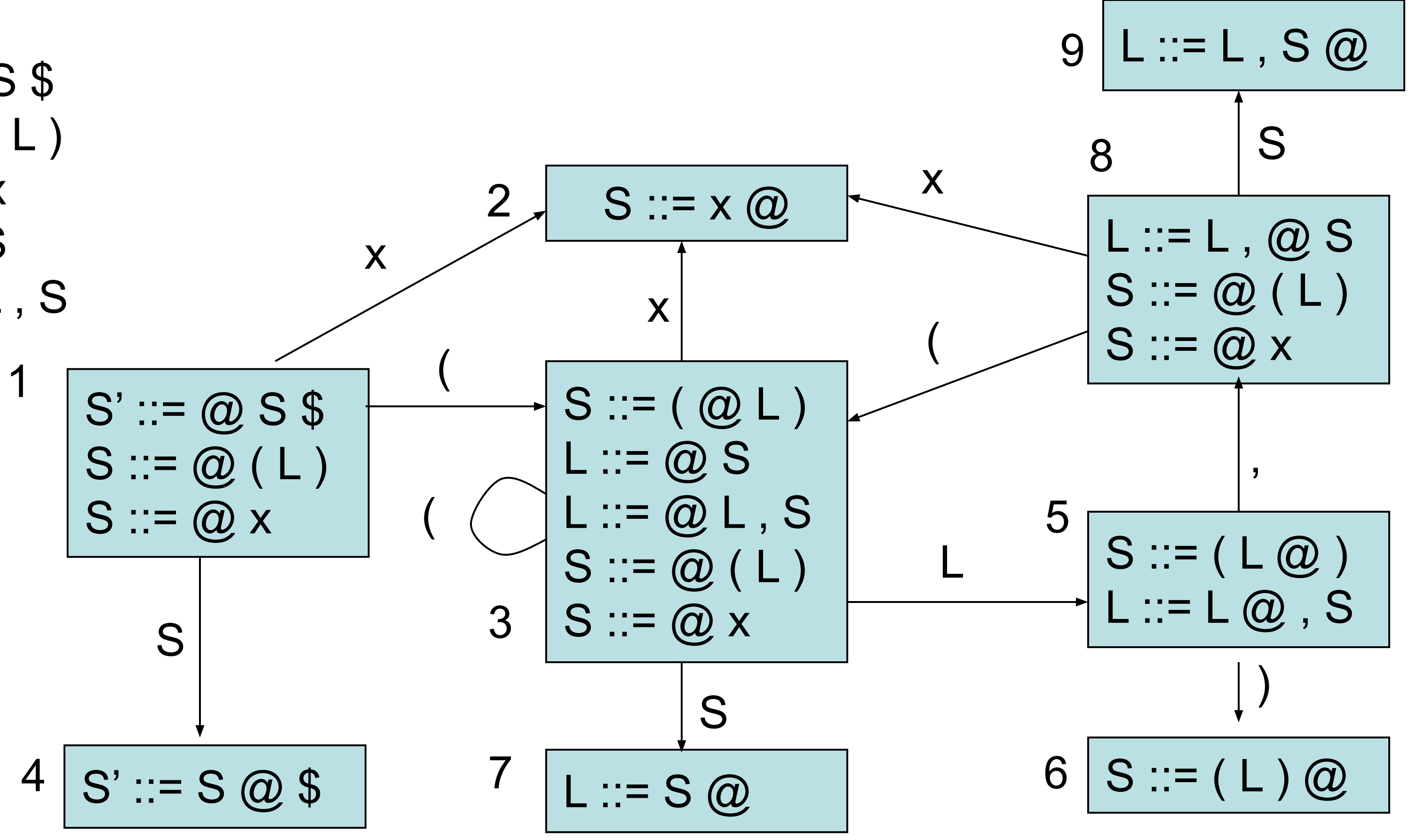
states	Terminal seen next ID, NUM, := ...	Non-terminals X,Y,Z ...
1		gn = goto state n
2	sn = shift & goto state n	
3	rk = reduce by rule k	
...	a = accept	
n	= error	

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



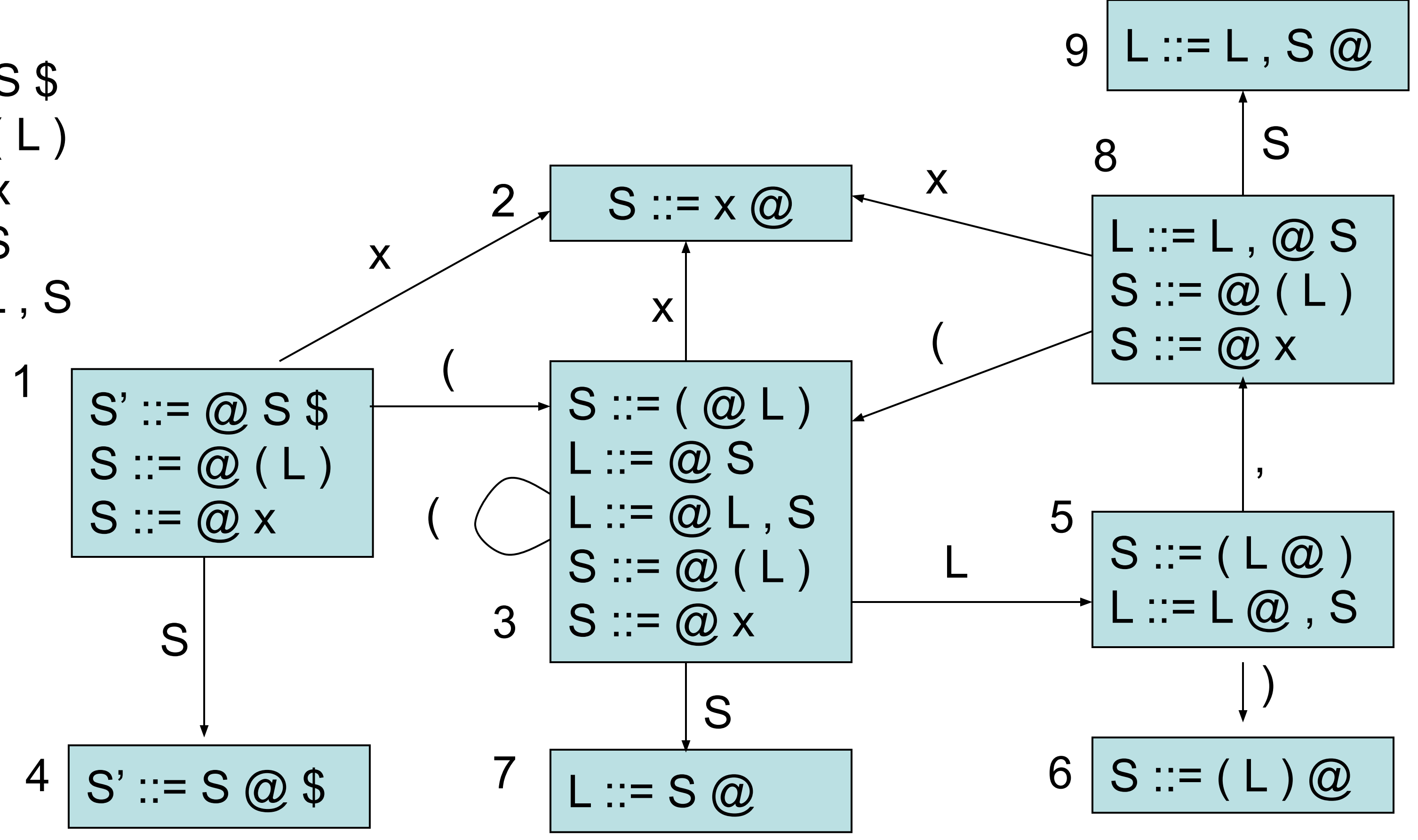
states	(	)	x	,	$	S	L
1							
2							
3							
4							
...							

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



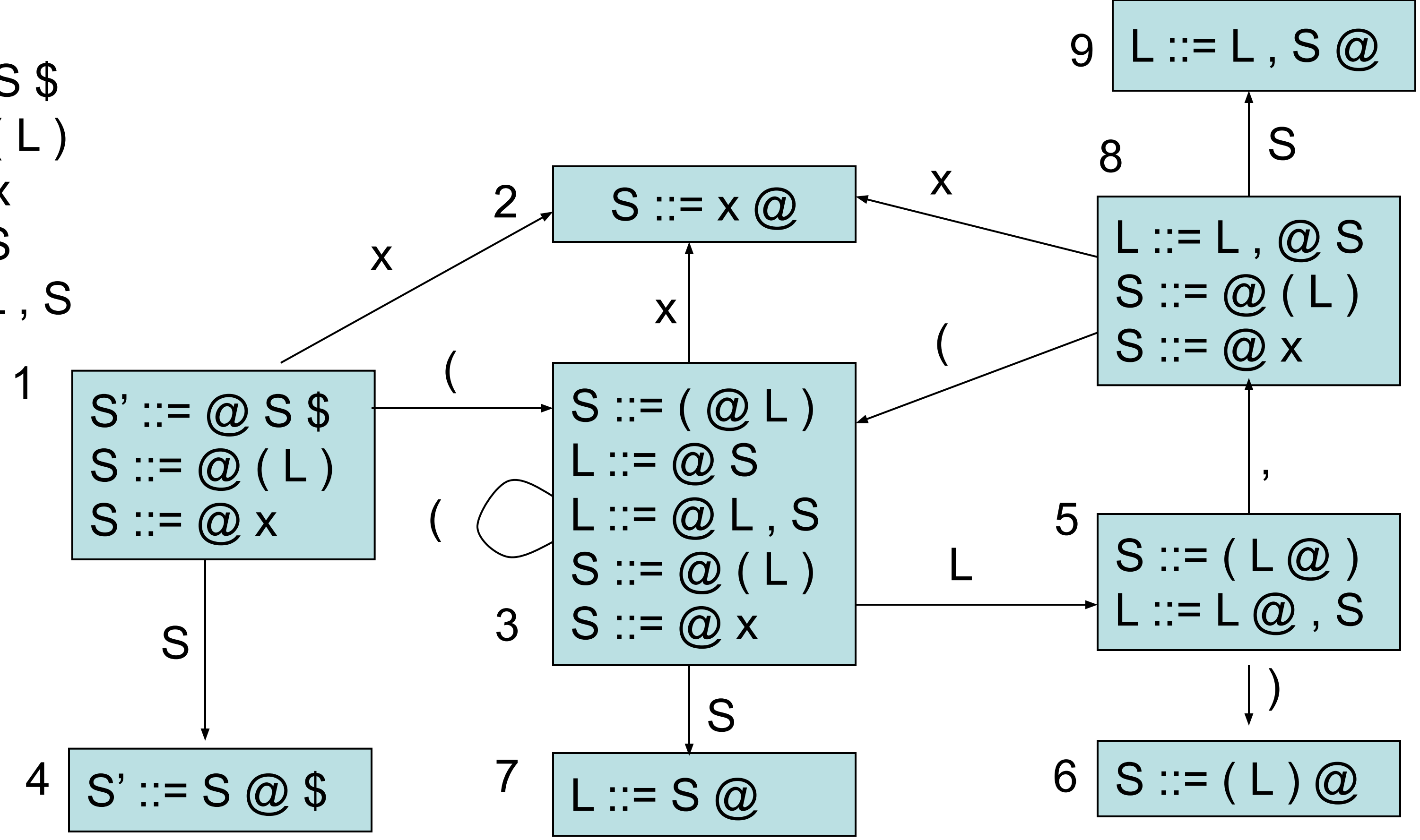
states	(	)	x	,	$	S	L
1	s3						
2							
3							
4							
...							

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



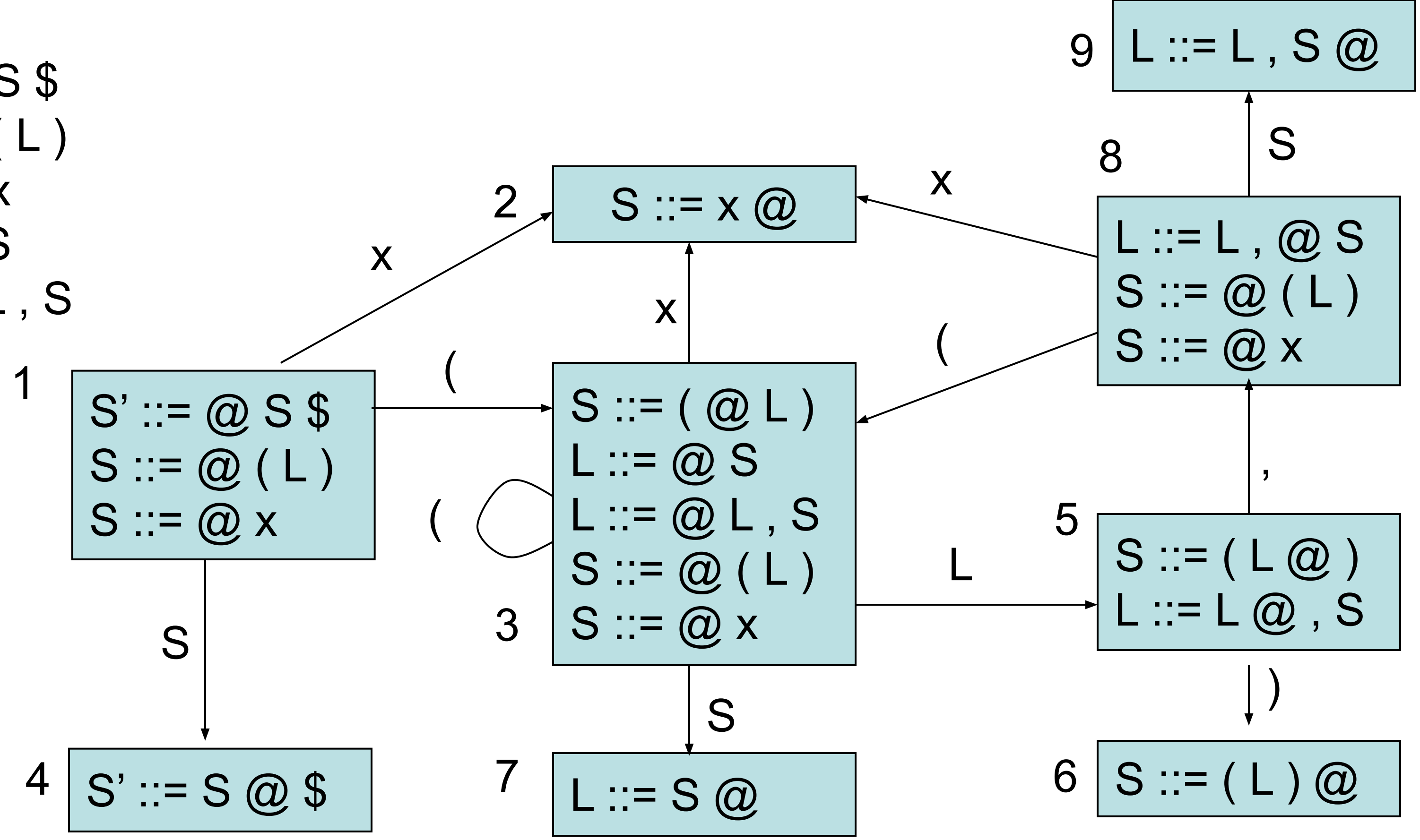
states	(	)	x	,	$	S	L
1	s3		s2				
2							
3							
4							
...							

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



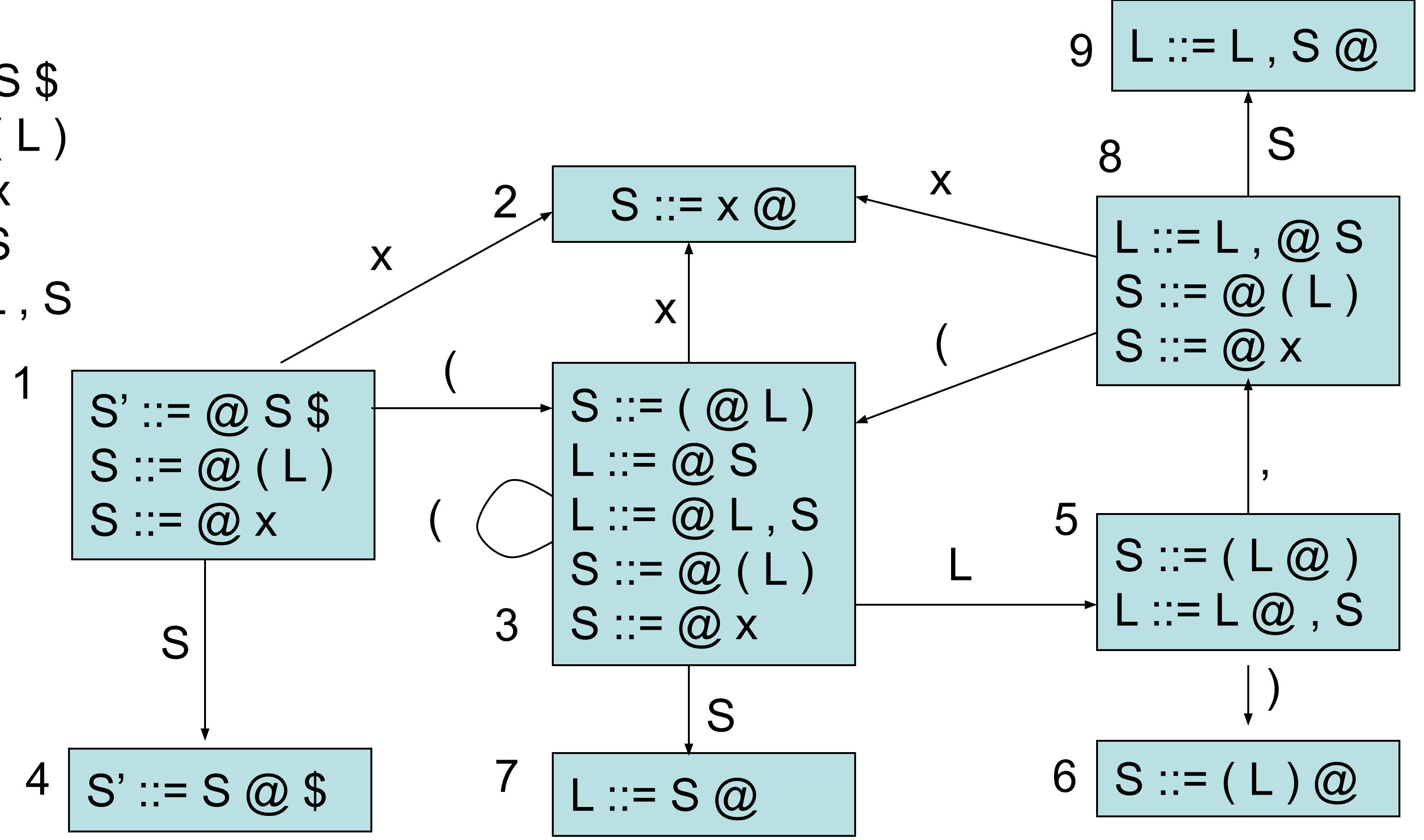
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2							
3							
4							
...							

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



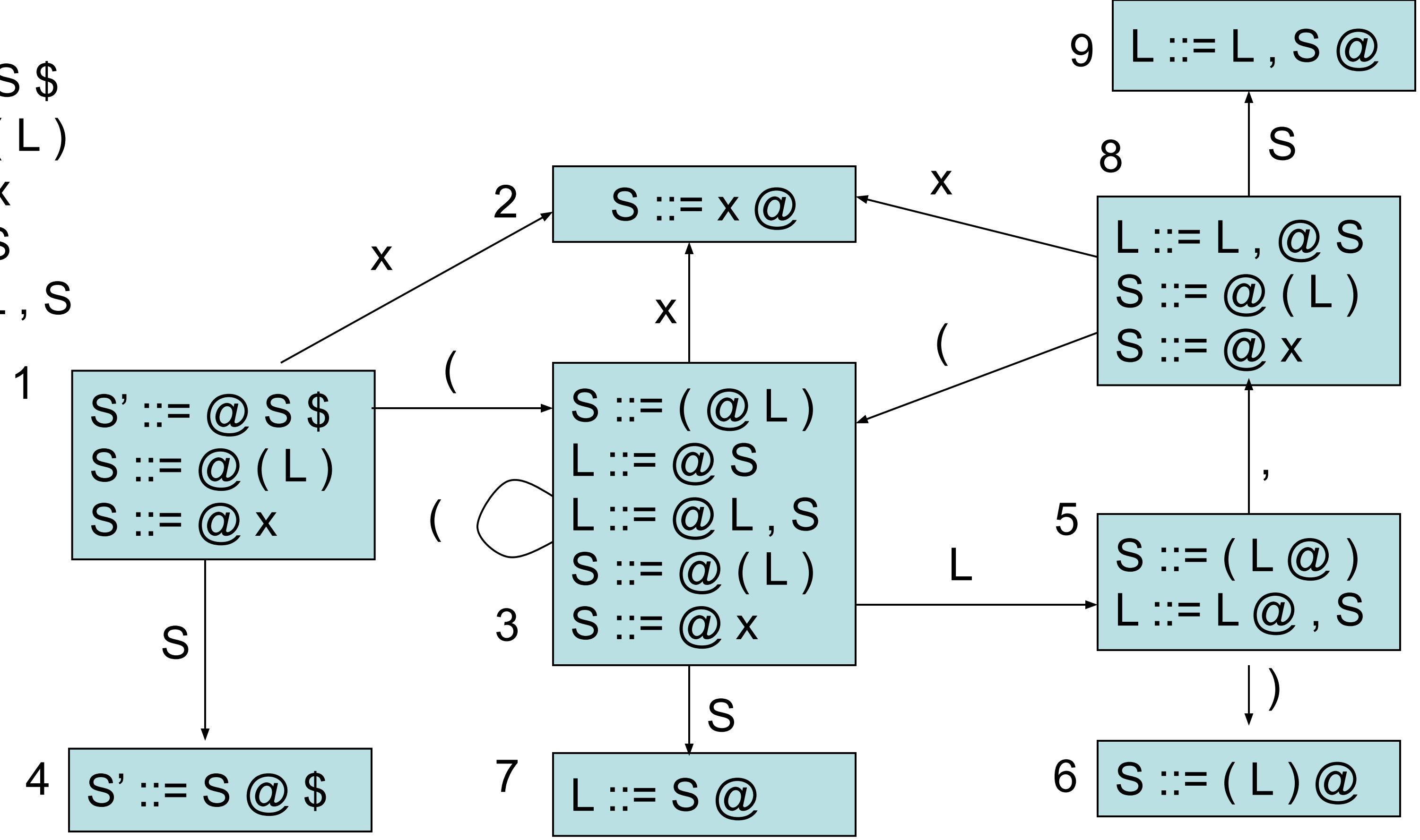
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3							
4							
...							

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



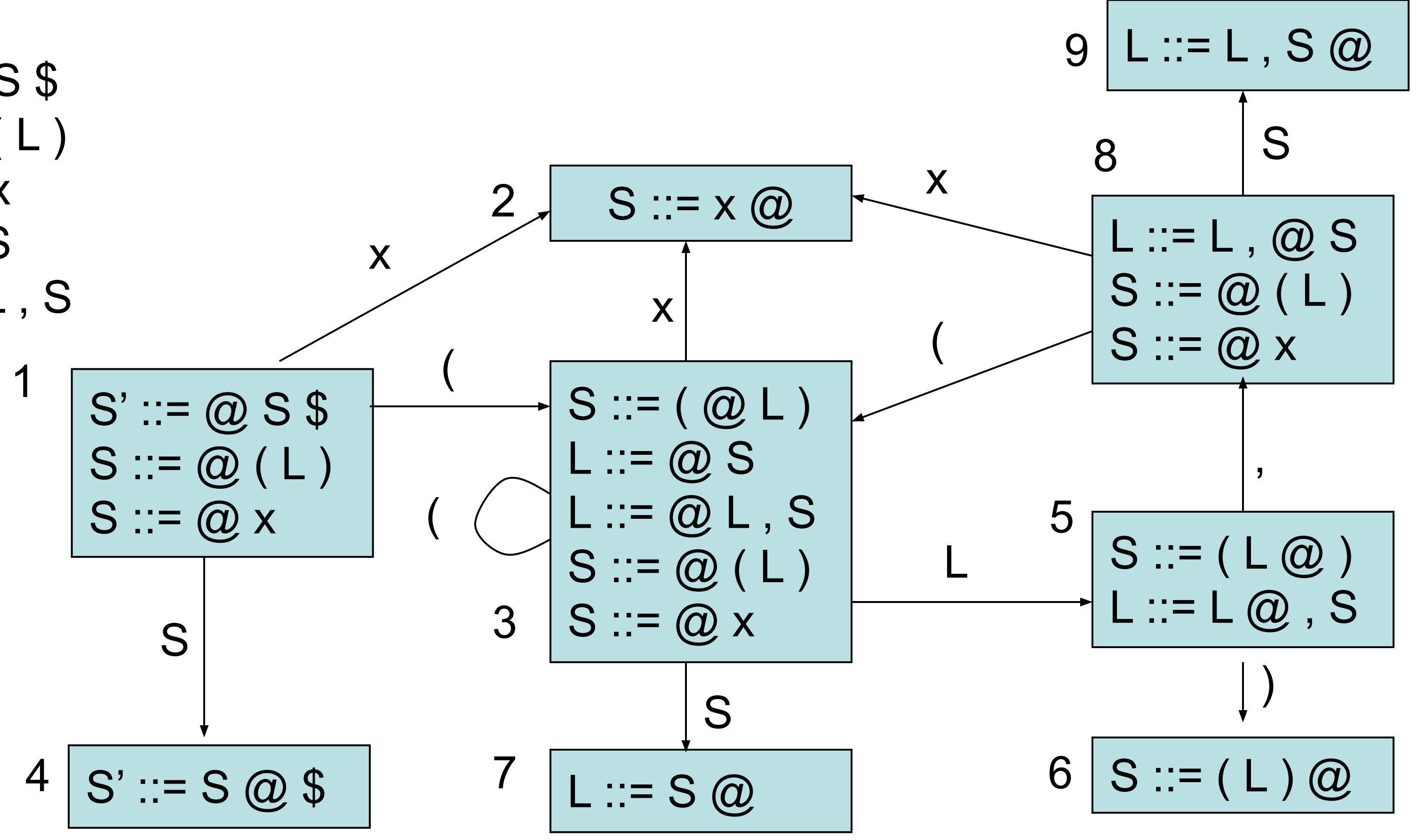
states	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2				
4							
...							

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



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

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



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

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

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

$\underbrace{\texttt{(x , x) \$}}_{}$ yet to read

input: (x , x) $

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

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

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		

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

input: $(\; x \; , \; x \; \underbrace{) \; \$}_{}$ — 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 $
1. S ::= (L)
2. S ::= x
3. L ::= S
4. L ::= L , S

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

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

input: (x , x) $ (yet to read: $)

stack: 1 (3 L 5) 6

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

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

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

input: (x , x) $

stack: 1 S

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

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

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

stack: 1 S 4

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

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

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

stack: Accept

LR(0) Limitations

- 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) Limitations

- 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) Limitations

- 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

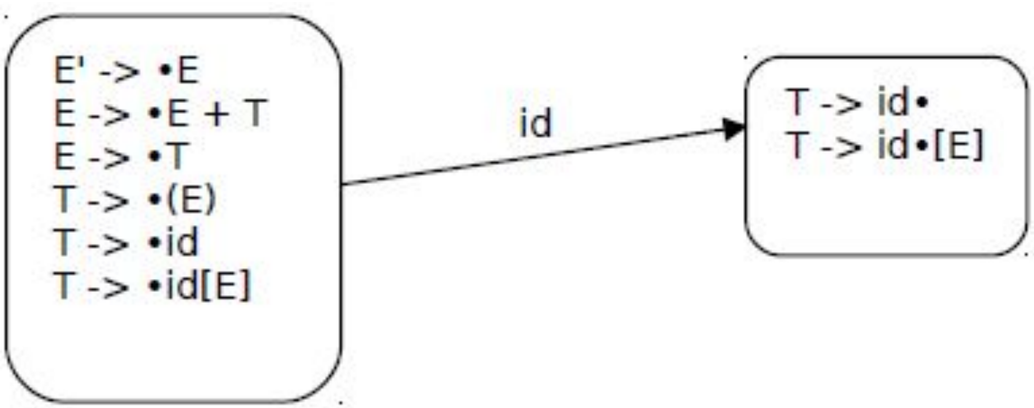
states	no look-ahead	S	L
1	shift, reduce 5	g4	
2	reduce 2, reduce 7		
3	shift	g7	g5

LR(0) Limitations: Example

Grammar:

E' -> E
E -> E + T | T
T -> (E) | id | id[E]

Here are the first two LR(0) configuration sets entered if **id** is the first token of the input.



In an LR(0) parser, the set on the **right** has a **shift reduce conflict.**

SLR(1) Parser

- 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

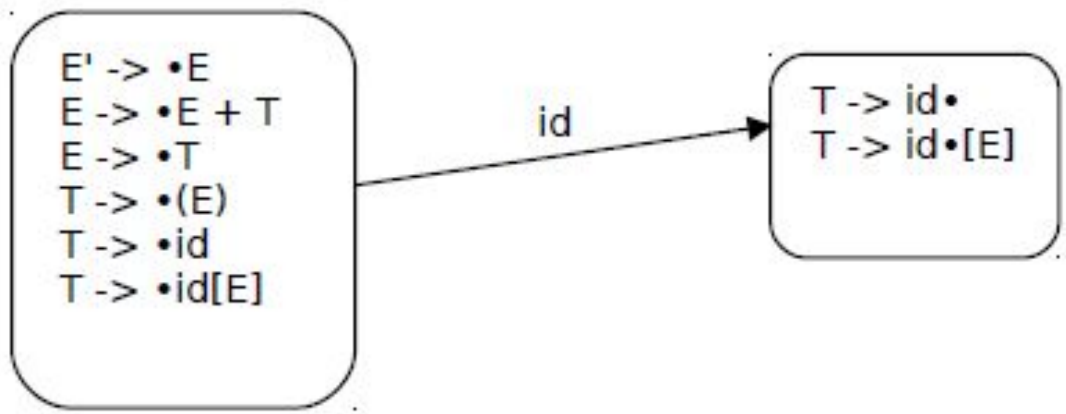
cuts down the number of rk slots & therefore cuts down conflicts

SLR(1) Parser

- An SLR(1) will compute Follow(T) = { +)] $ } and **only enter the reduce action on those tokens.** The
- input [will **shift** and there is no conflict. Thus this grammar is SLR(1) even though it is not LR(0).

Grammar:

E' -> E
E -> E + T | T
T -> (E) | id | id[E]



Construction of The Canonical LR(0) Collection (CC)

- To create the **SLR parsing tables** for a grammar G, we will create the **canonical LR(0) collection** of the grammar G'.

- ***Algorithm***:

  C is { closure({S'→ . S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

  for each *I* in ***C*** and each grammar symbol X

  if GOTO(I,X) is not empty and not in ***C***

  add GOTO(I,X) to ***C***

- GOTO function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

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

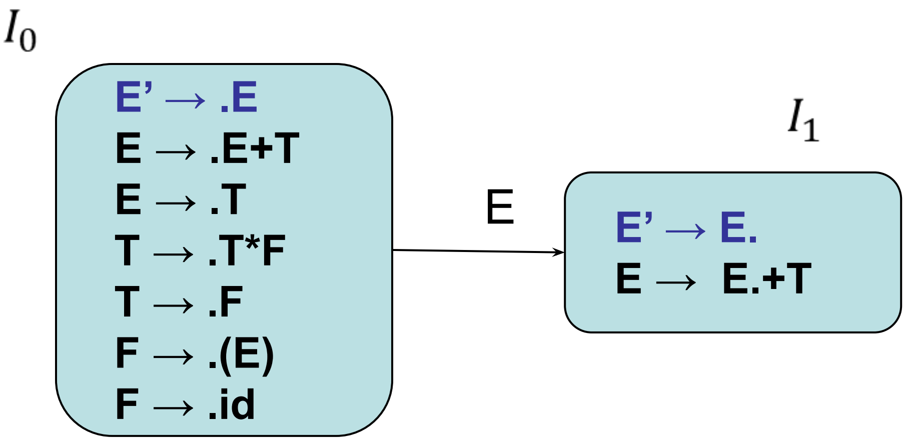
After Augmentation the Grammar will be

0. E’ → .E
1. E → .E+T
2. E → .T
3. T → .T*F
4. T → .F
5. F → .(E)
6. F → .id

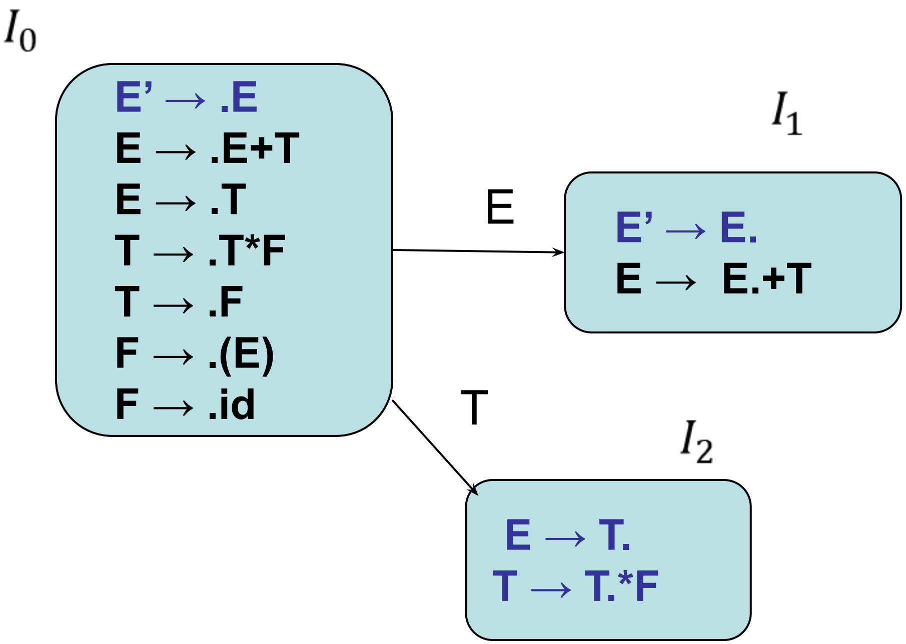
I_0

1. E’ → .E
2. E → .E+T
3. E → .T
4. T → .T*F
5. T → .F
6. F → .(E)
7. F → .id

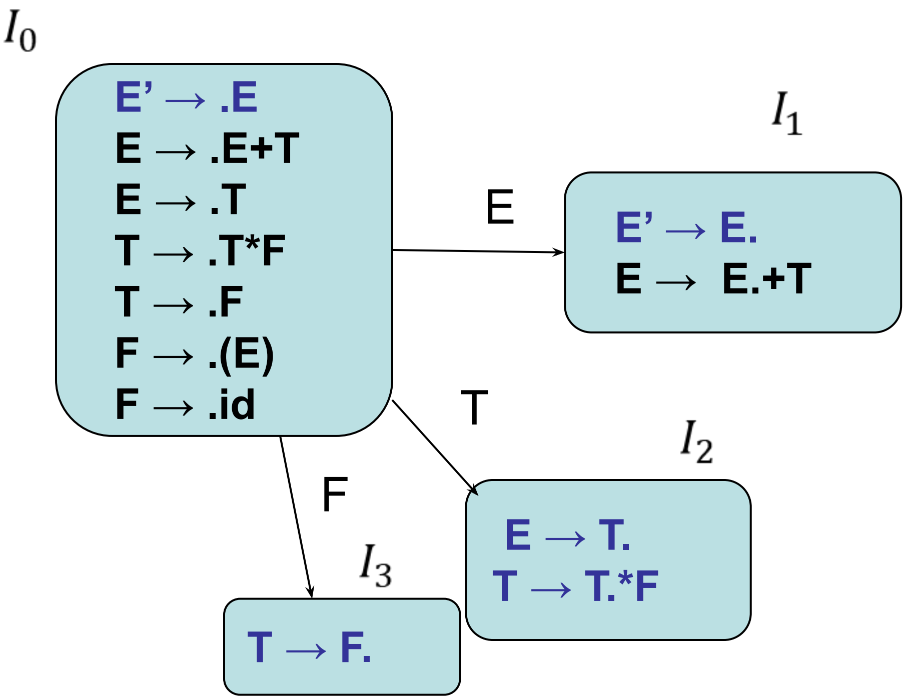
The Canonical LR(0) Collection -- Example



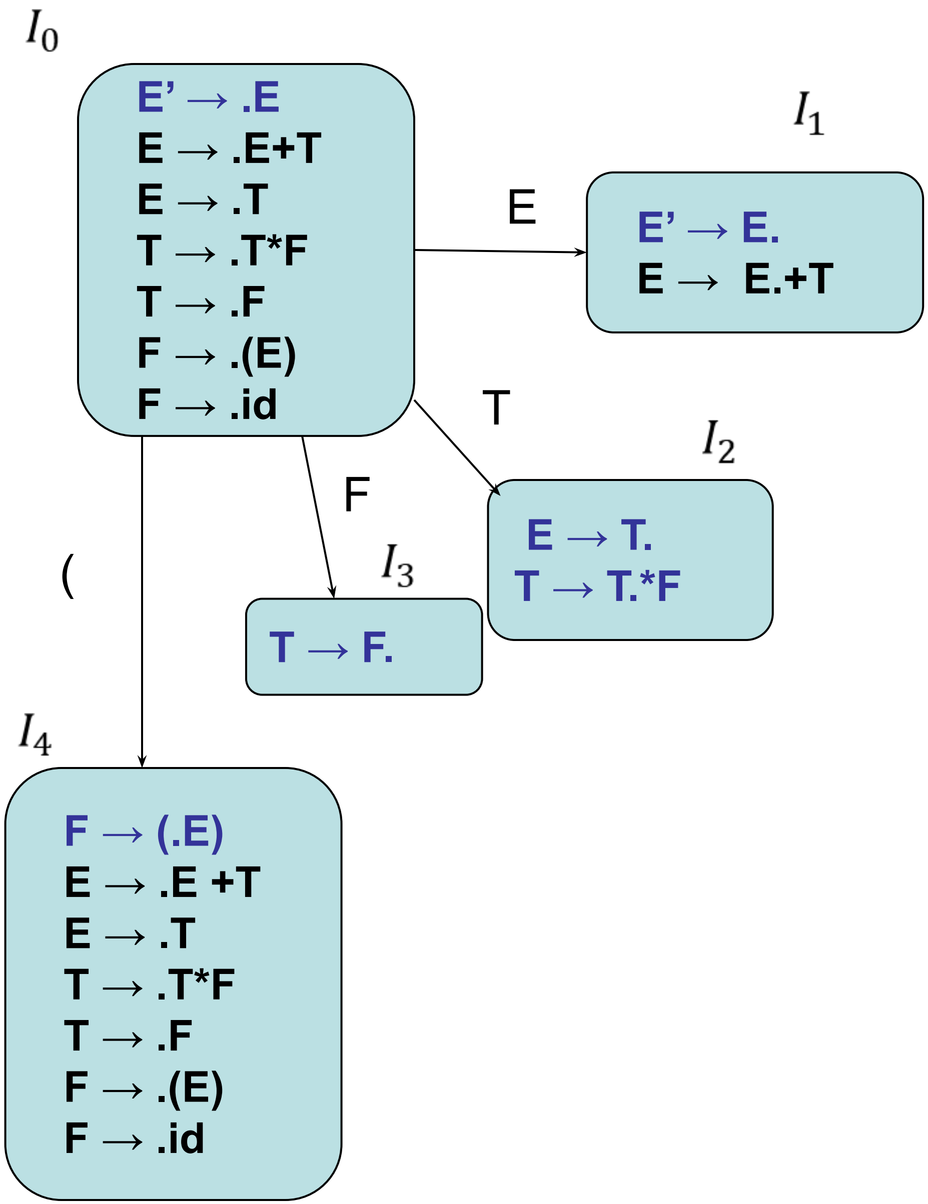
The Canonical LR(0) Collection -- Example



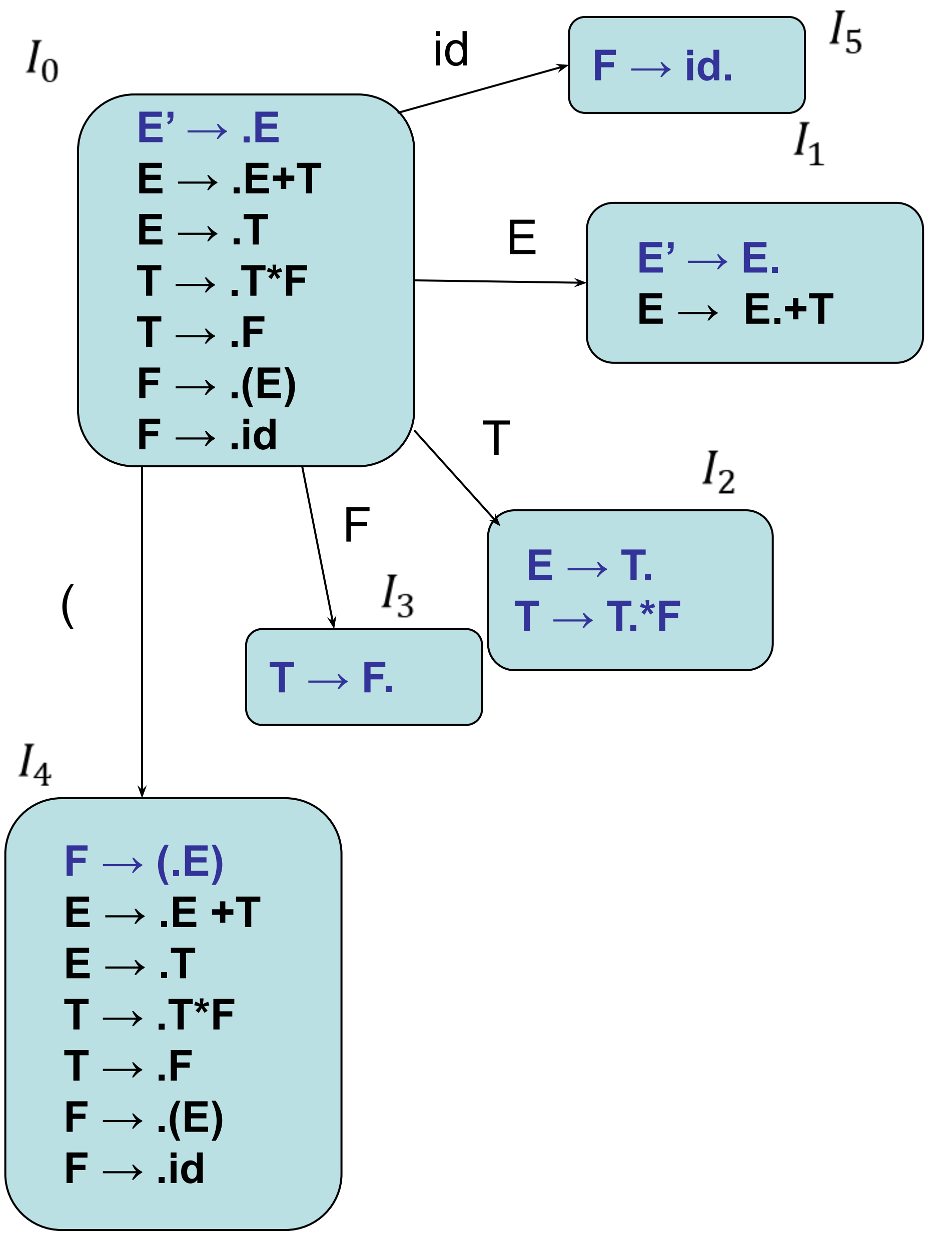
The Canonical LR(0) Collection -- Example



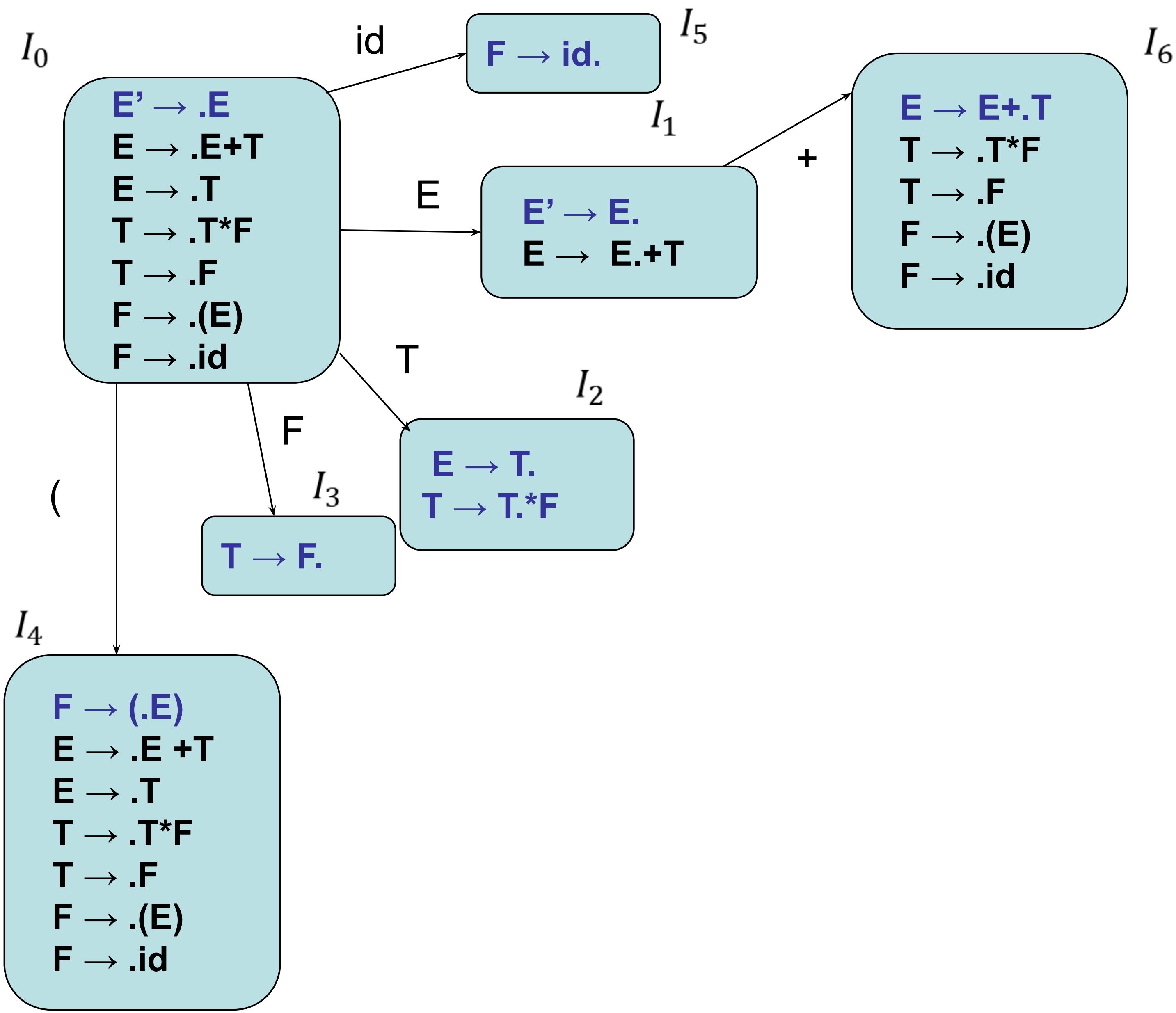
The Canonical LR(0) Collection -- Example



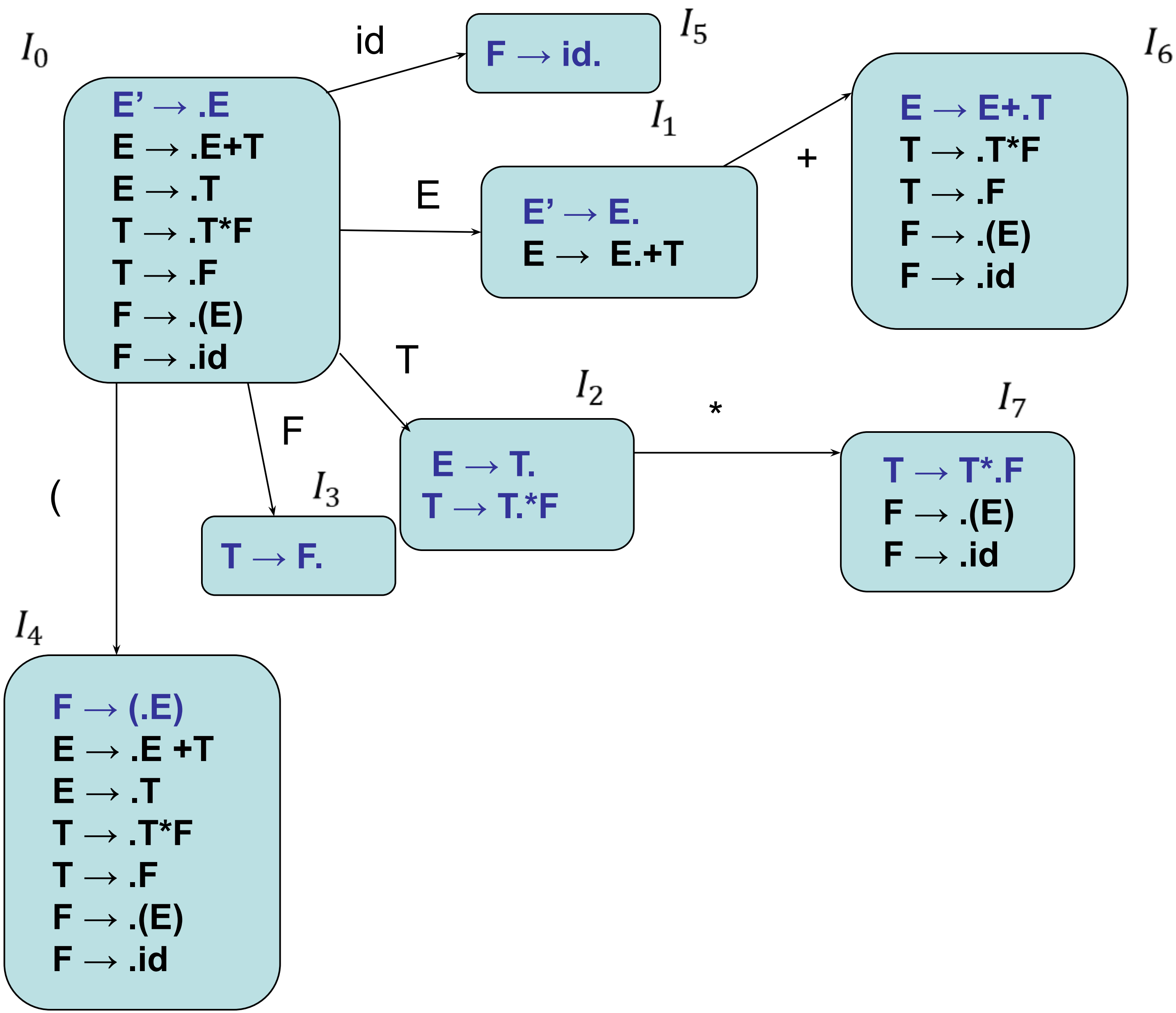
The Canonical LR(0) Collection -- Example



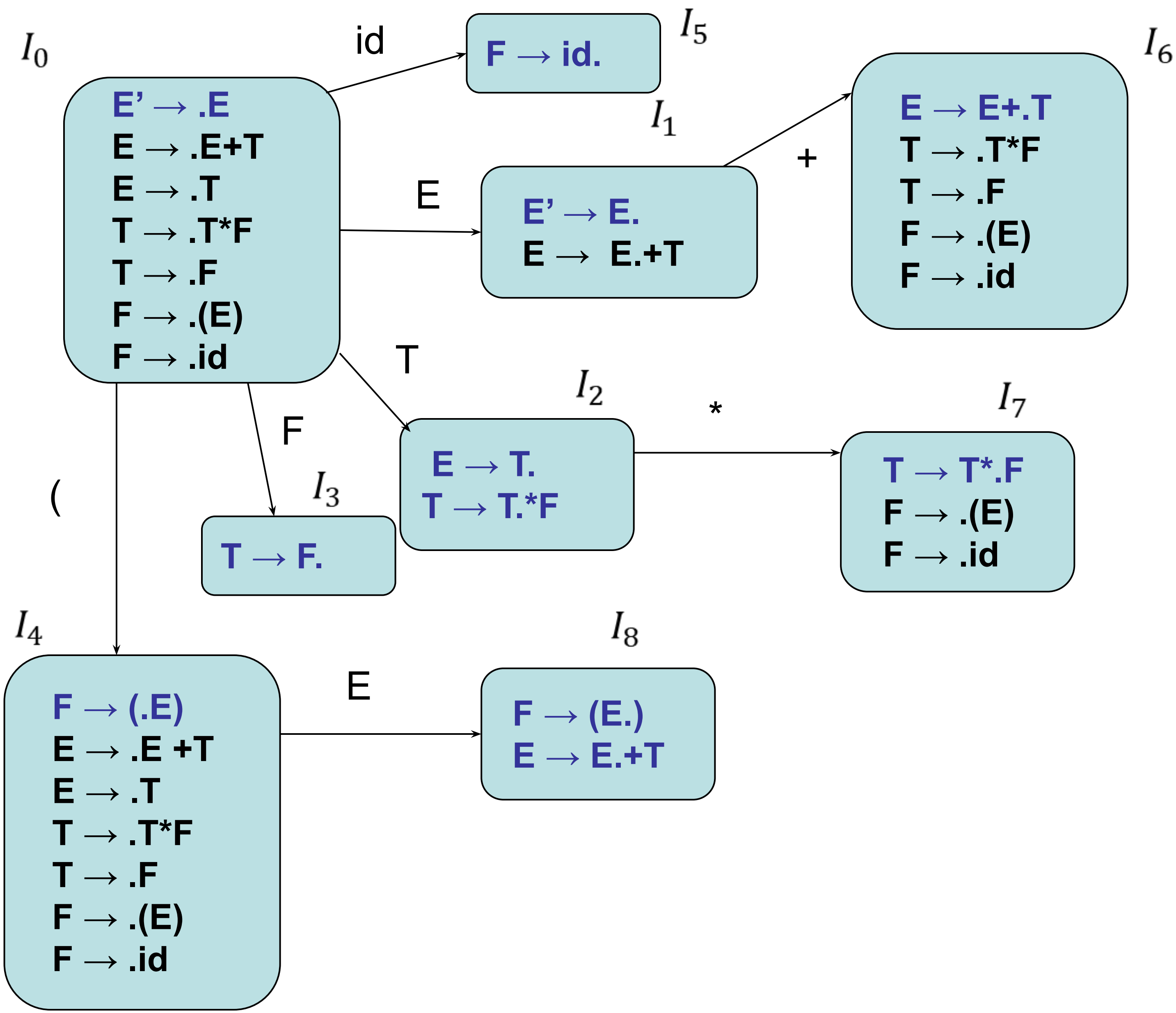
The Canonical LR(0) Collection -- Example



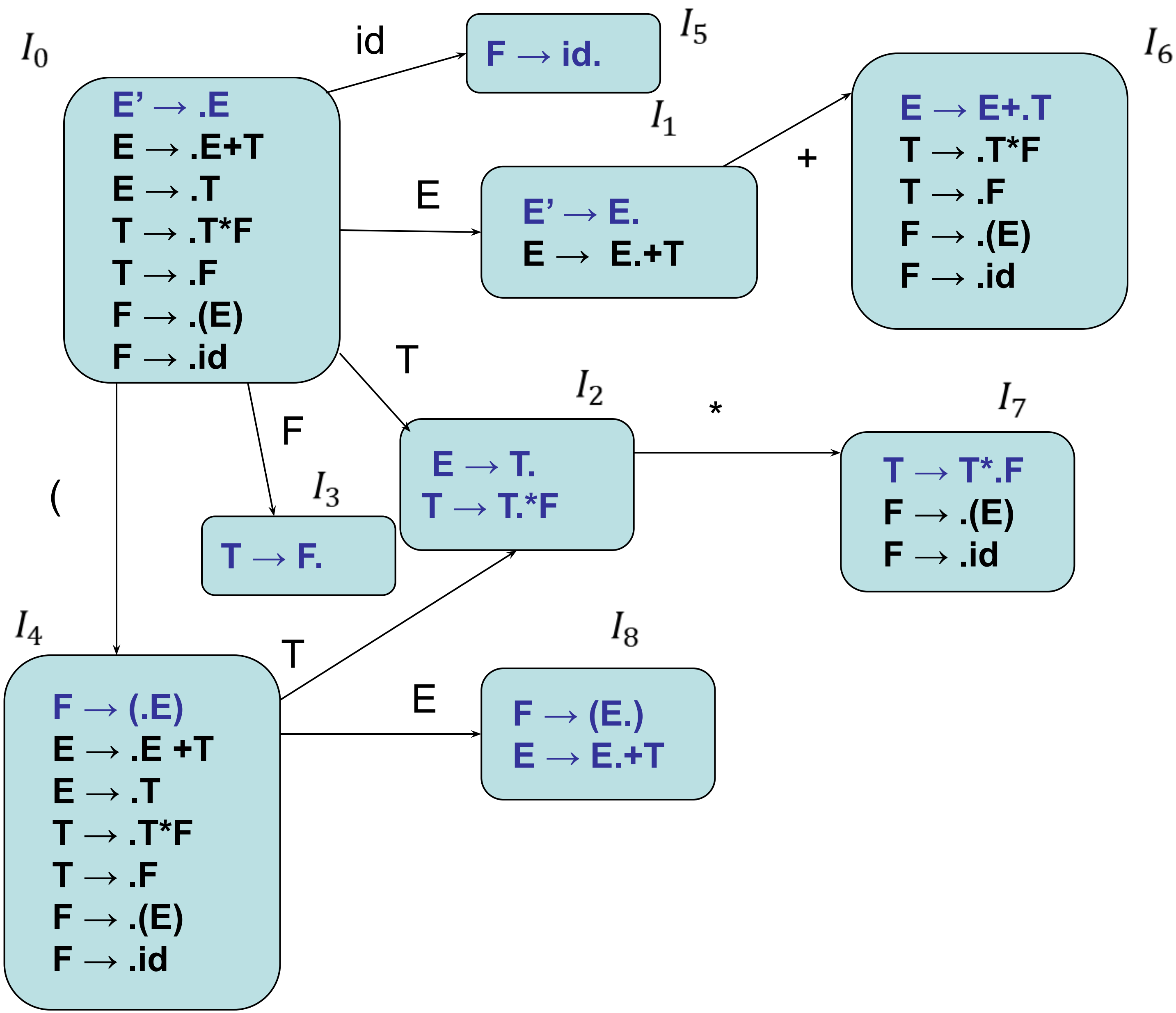
The Canonical LR(0) Collection -- Example



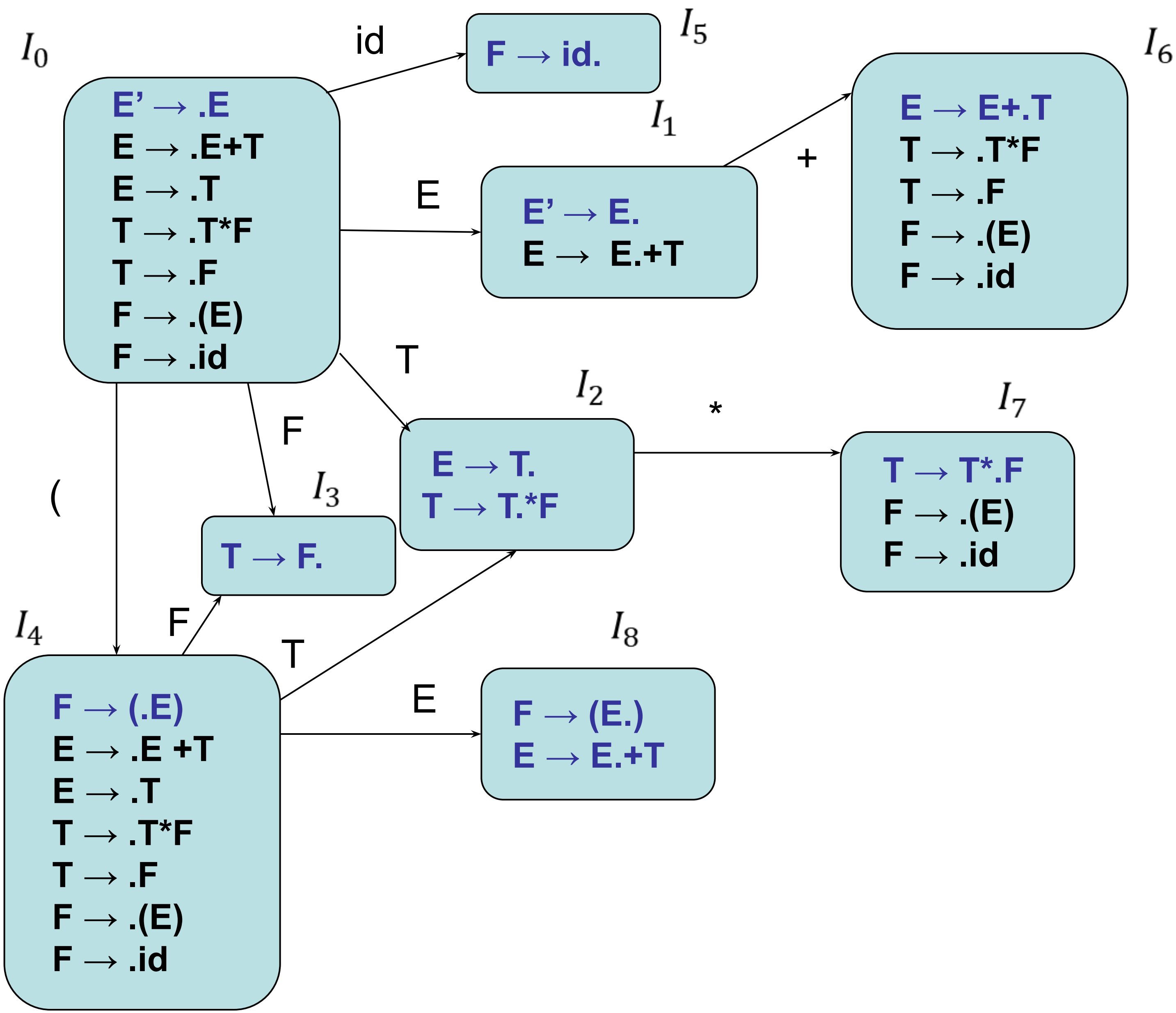
The Canonical LR(0) Collection -- Example



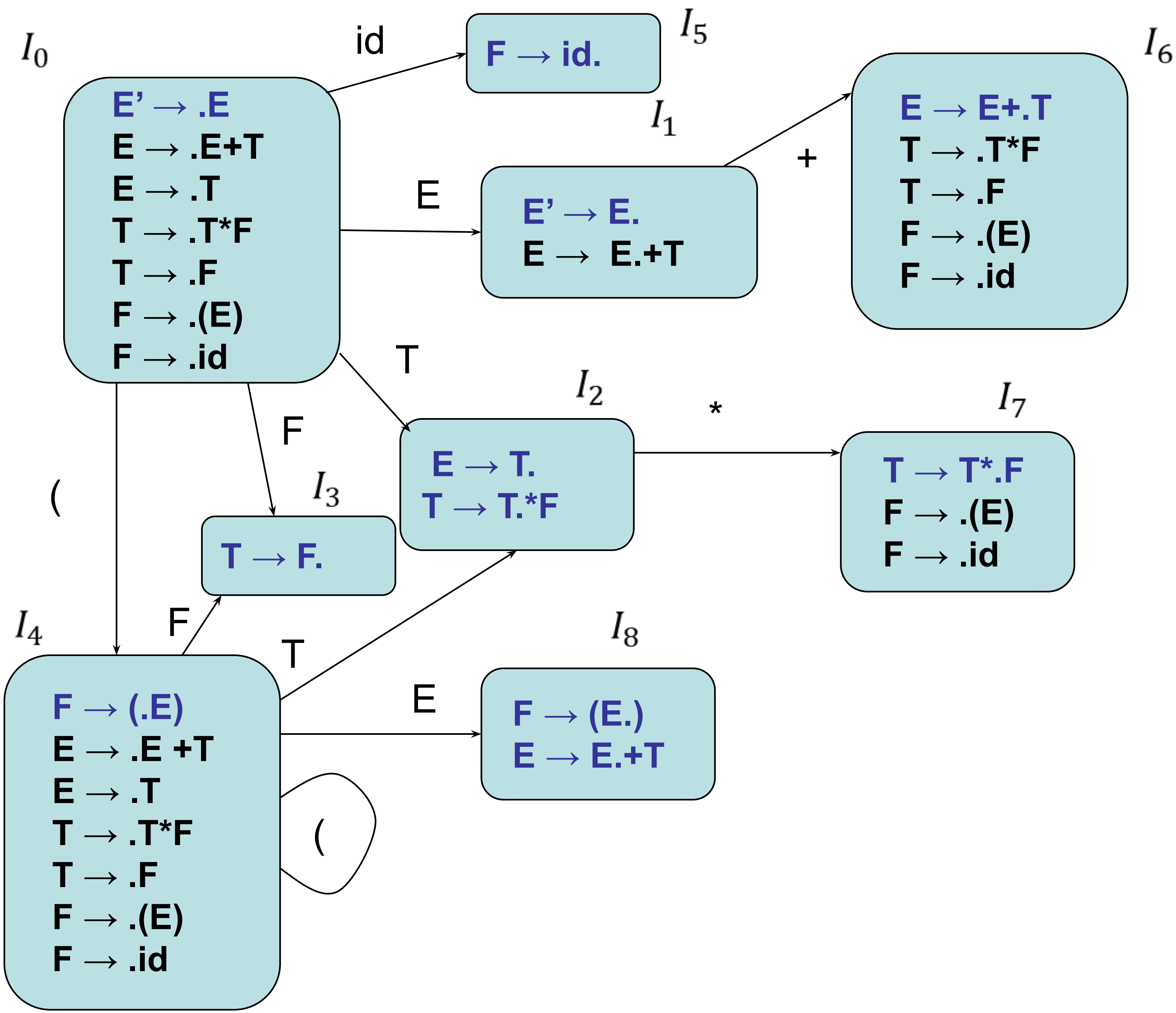
The Canonical LR(0) Collection -- Example



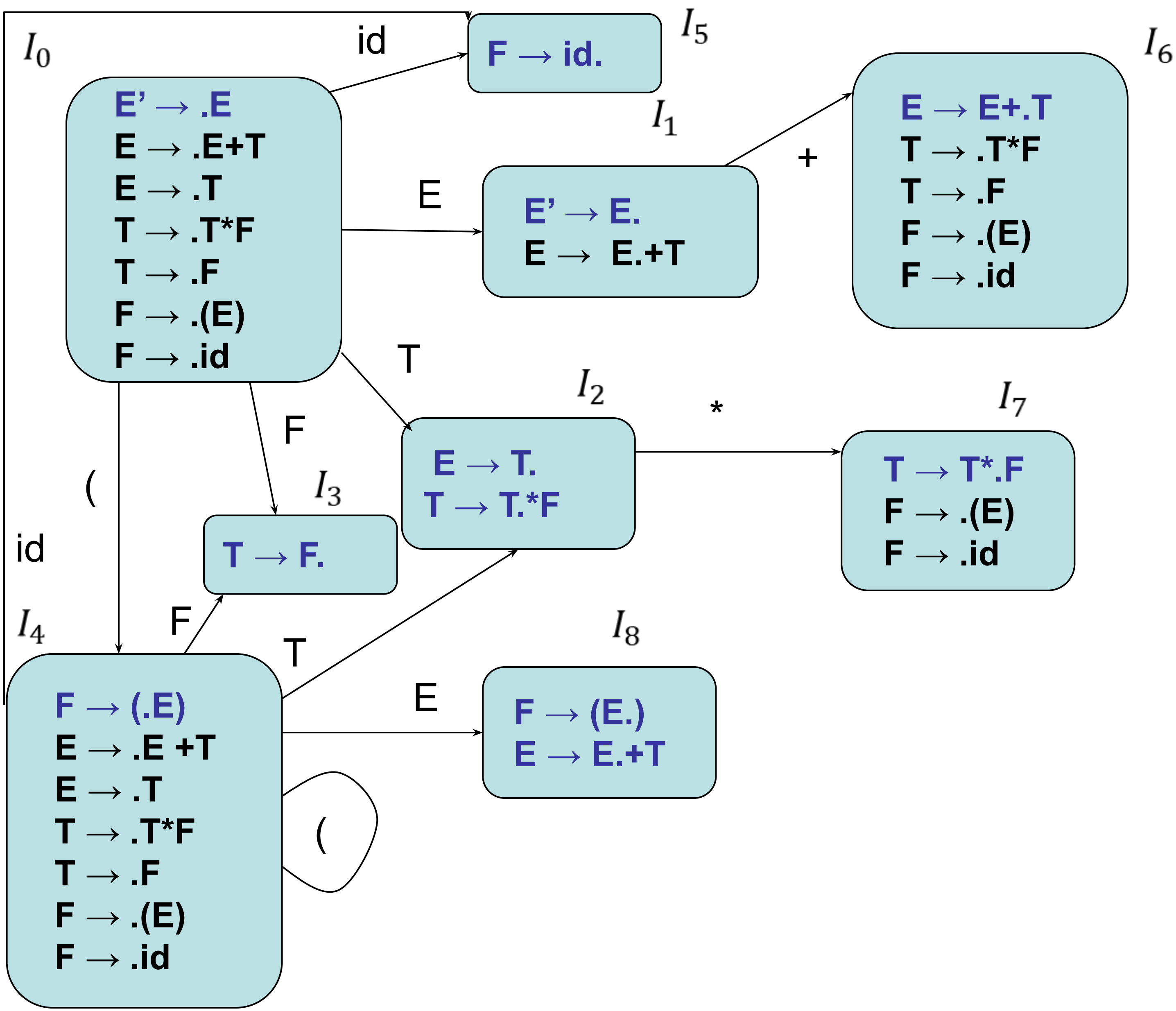
The Canonical LR(0) Collection -- Example



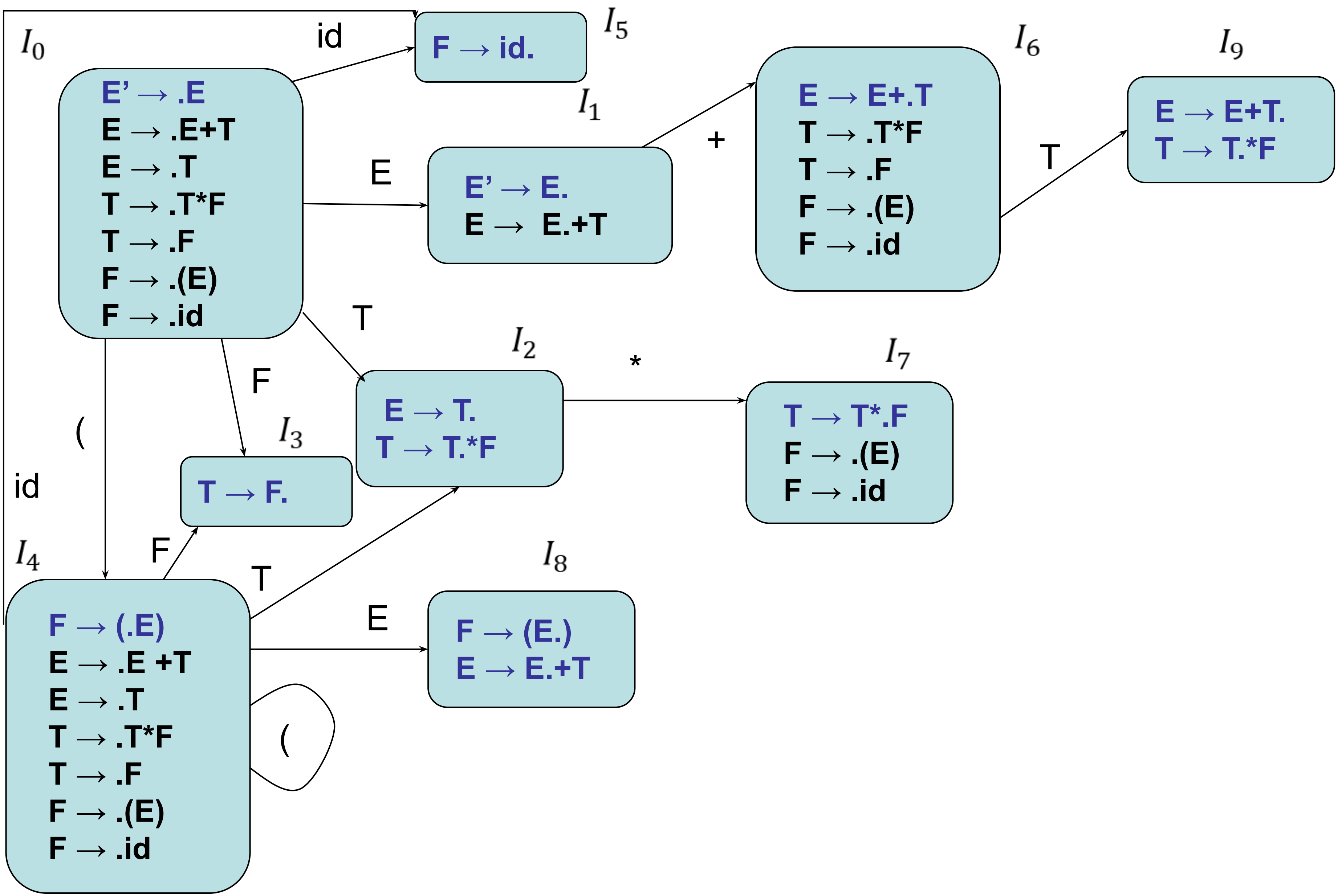
The Canonical LR(0) Collection -- Example



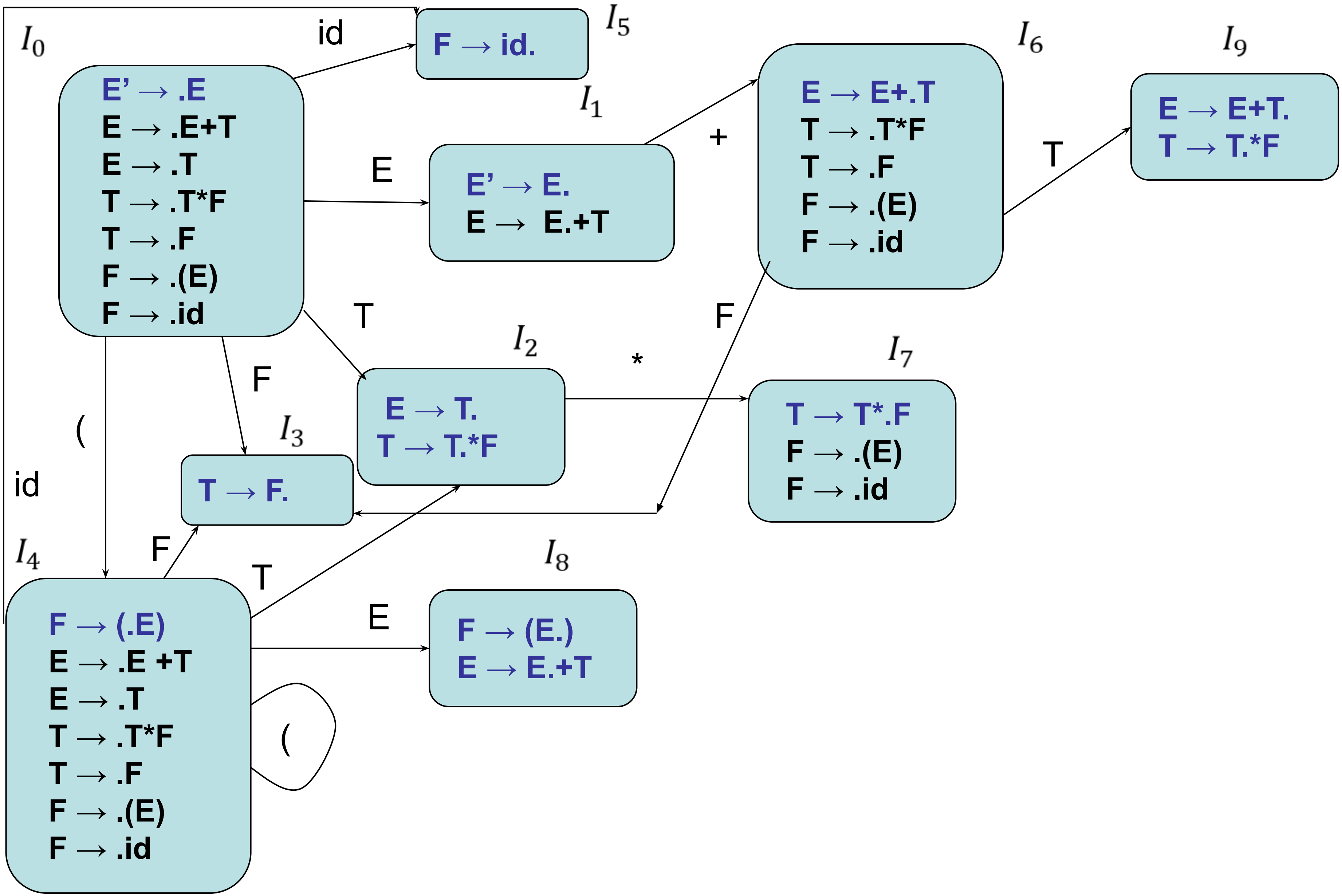
The Canonical LR(0) Collection -- Example



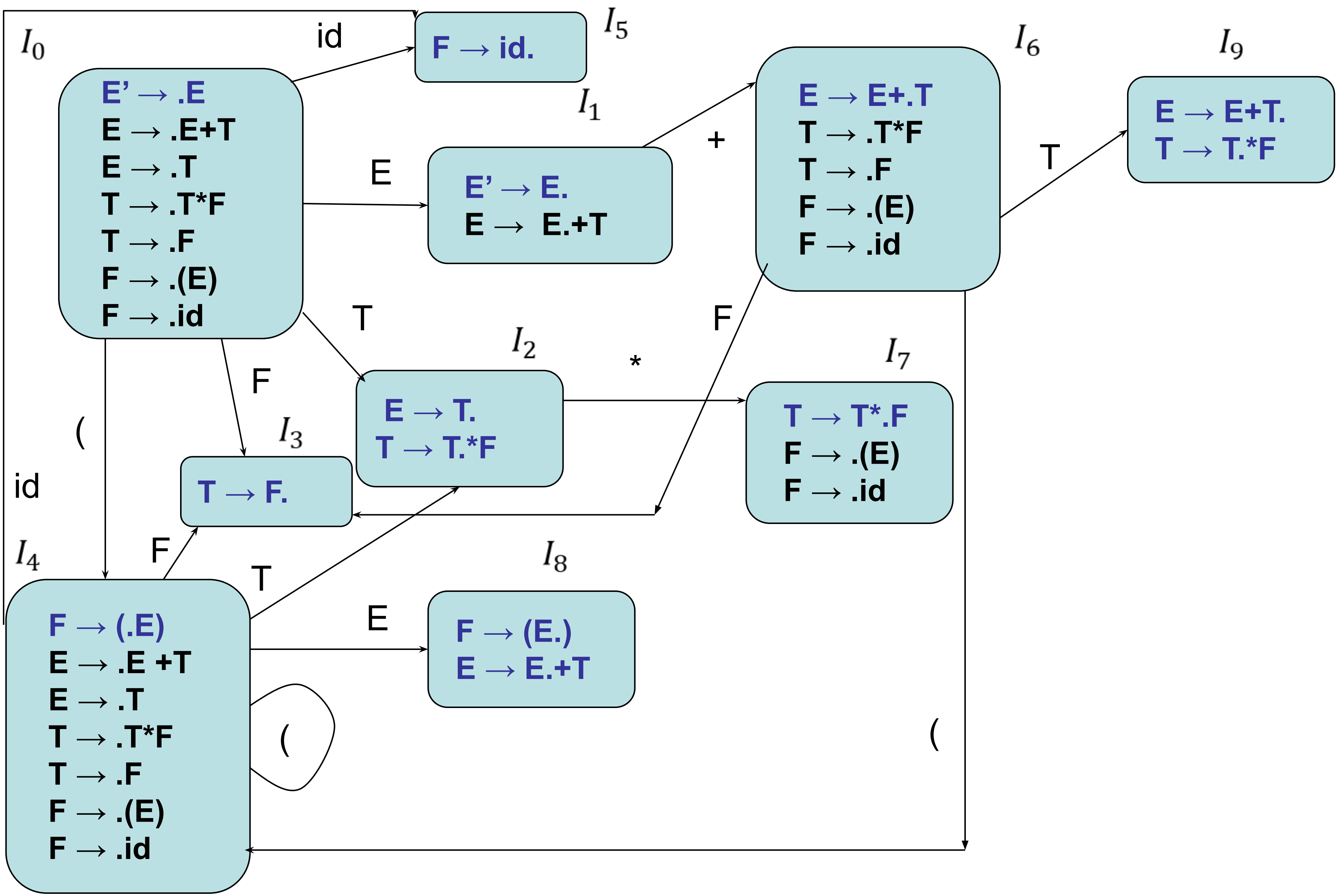
The Canonical LR(0) Collection -- Example



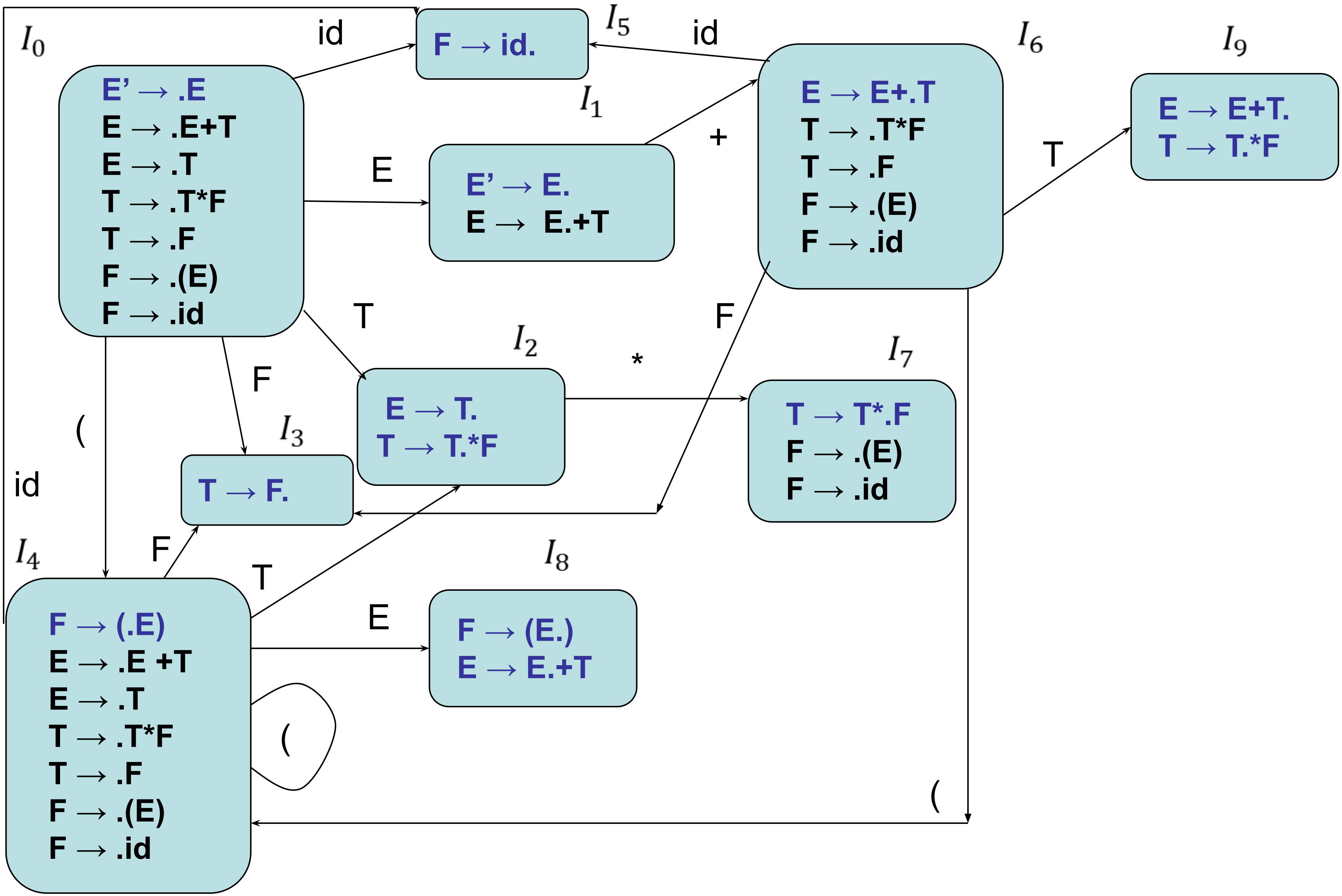
The Canonical LR(0) Collection -- Example



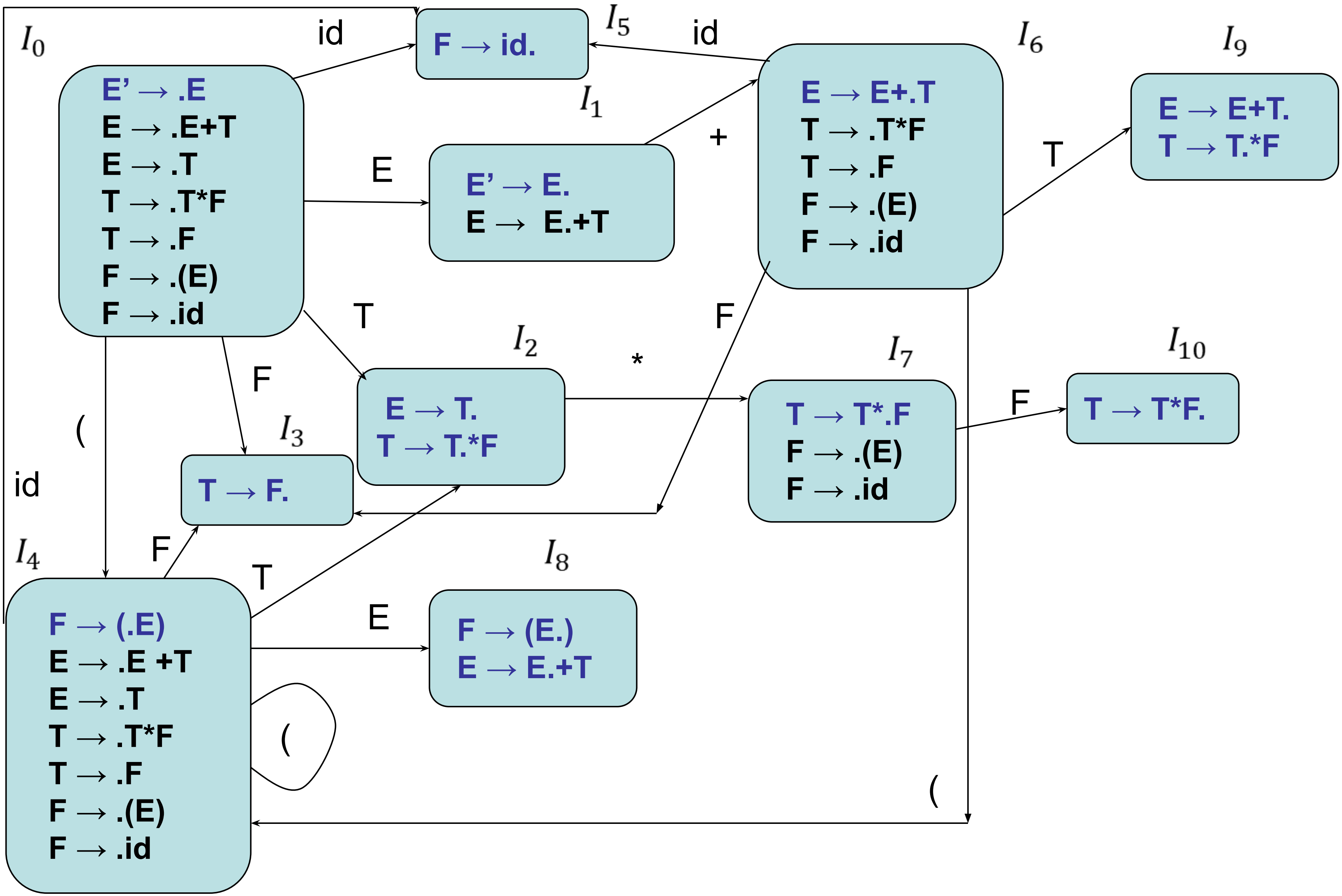
The Canonical LR(0) Collection -- Example



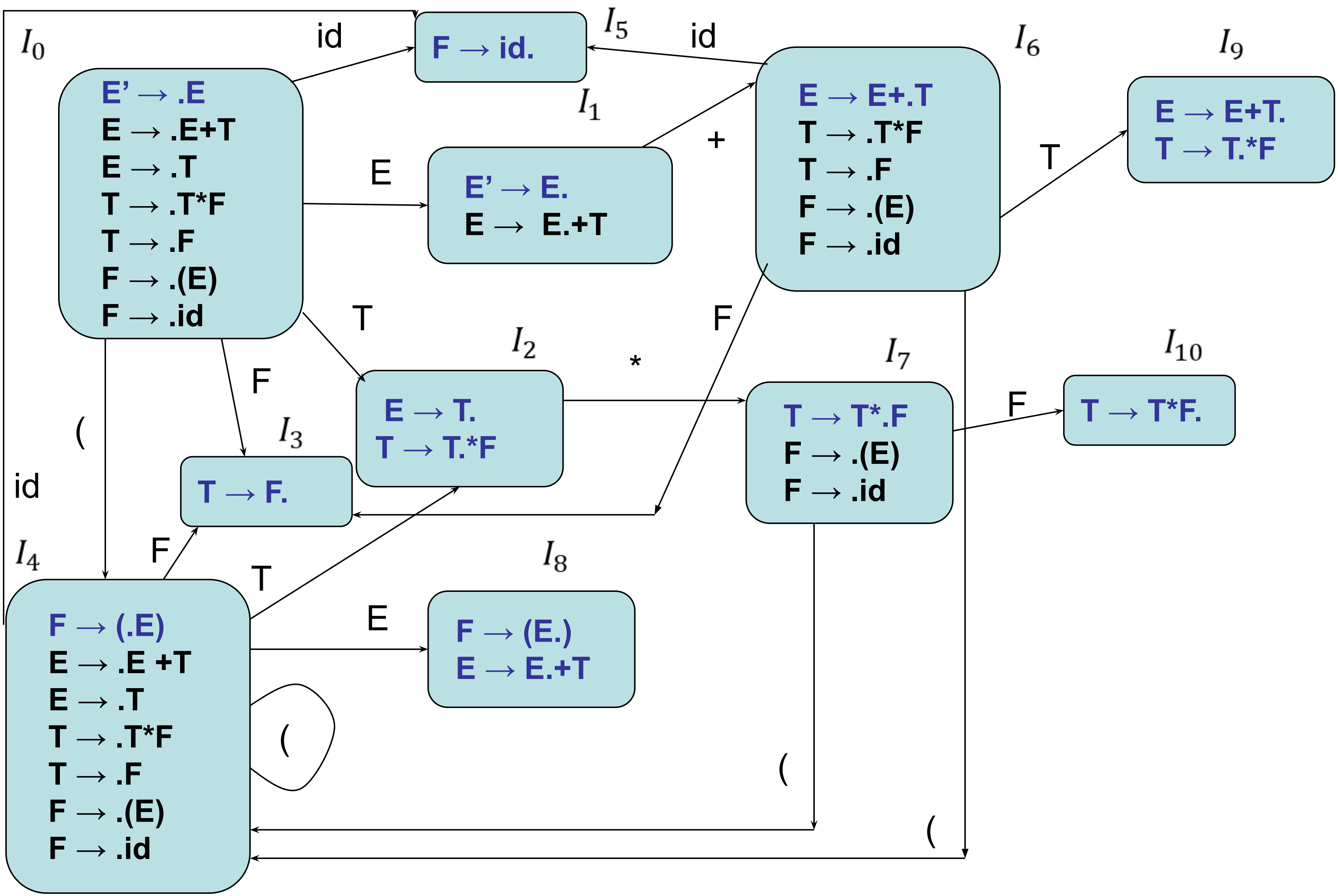
The Canonical LR(0) Collection -- Example



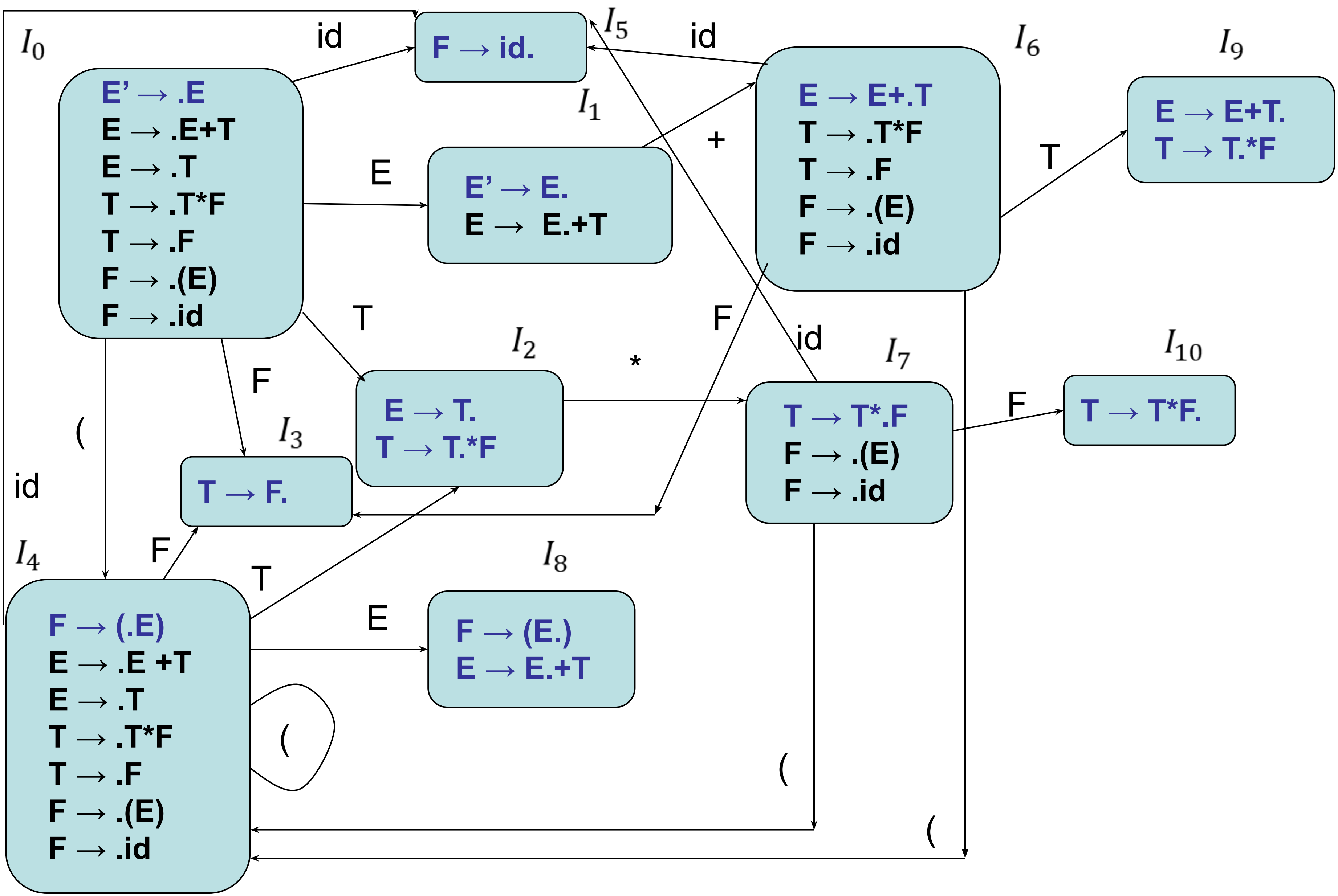
The Canonical LR(0) Collection -- Example



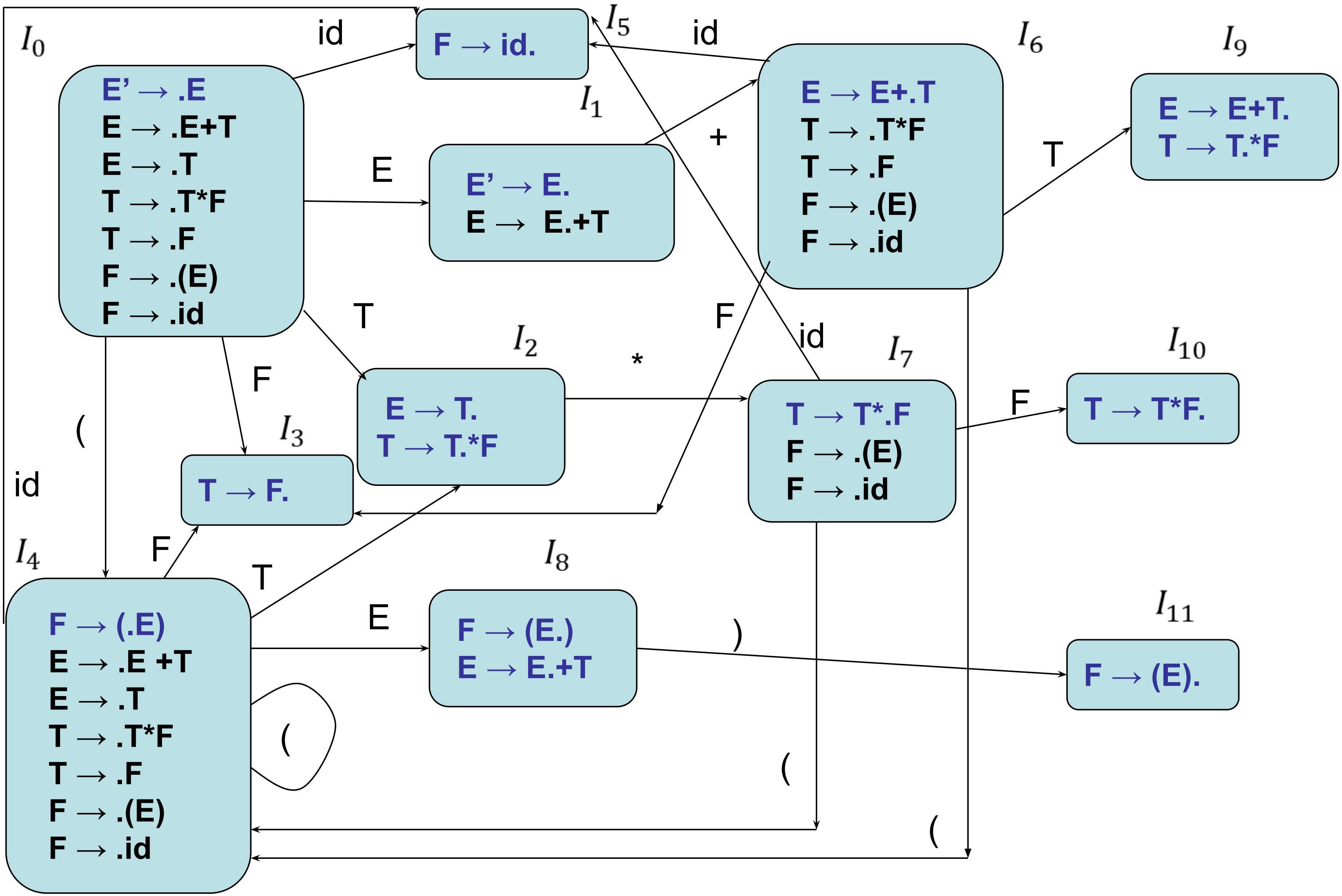
The Canonical LR(0) Collection -- Example



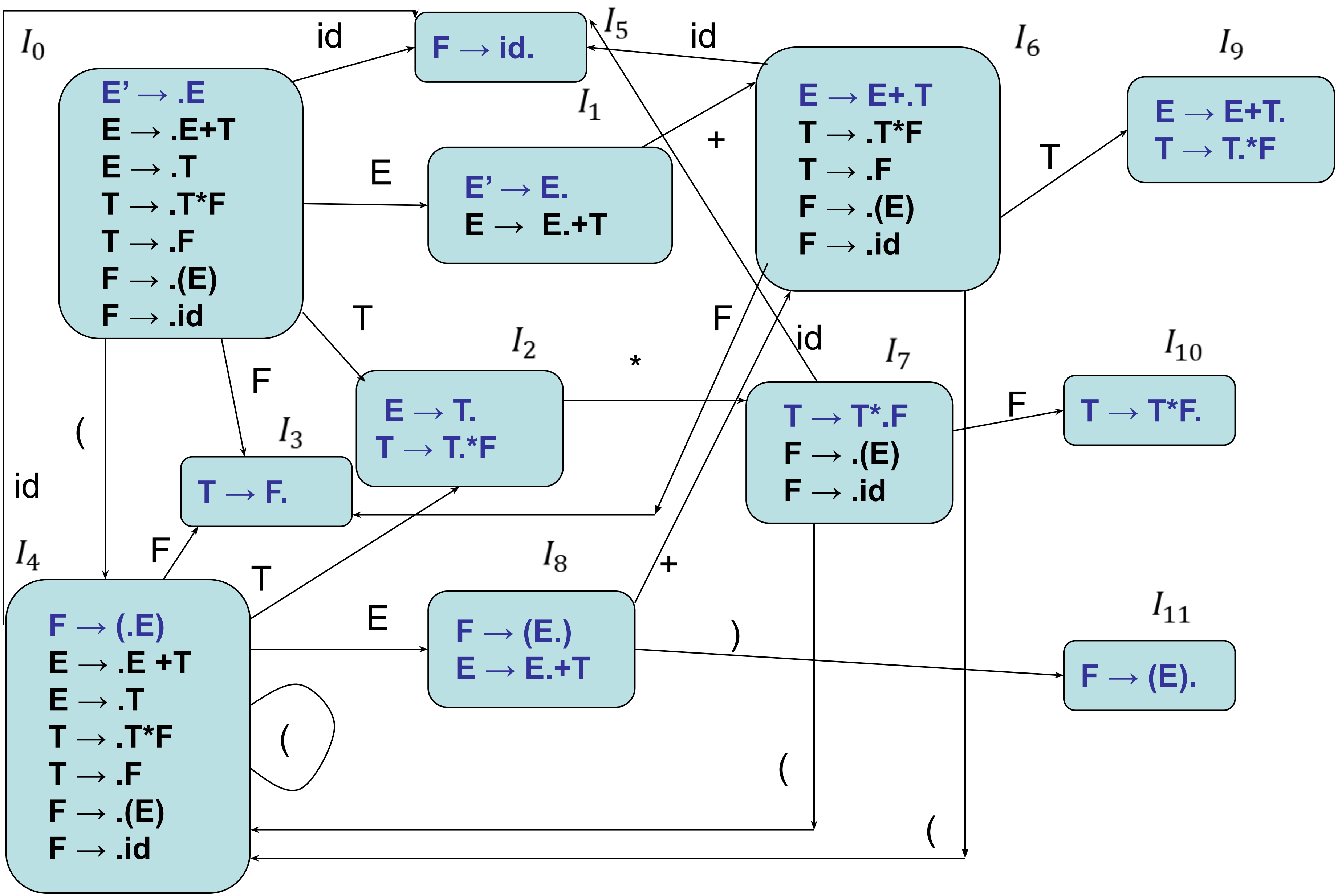
The Canonical LR(0) Collection -- Example



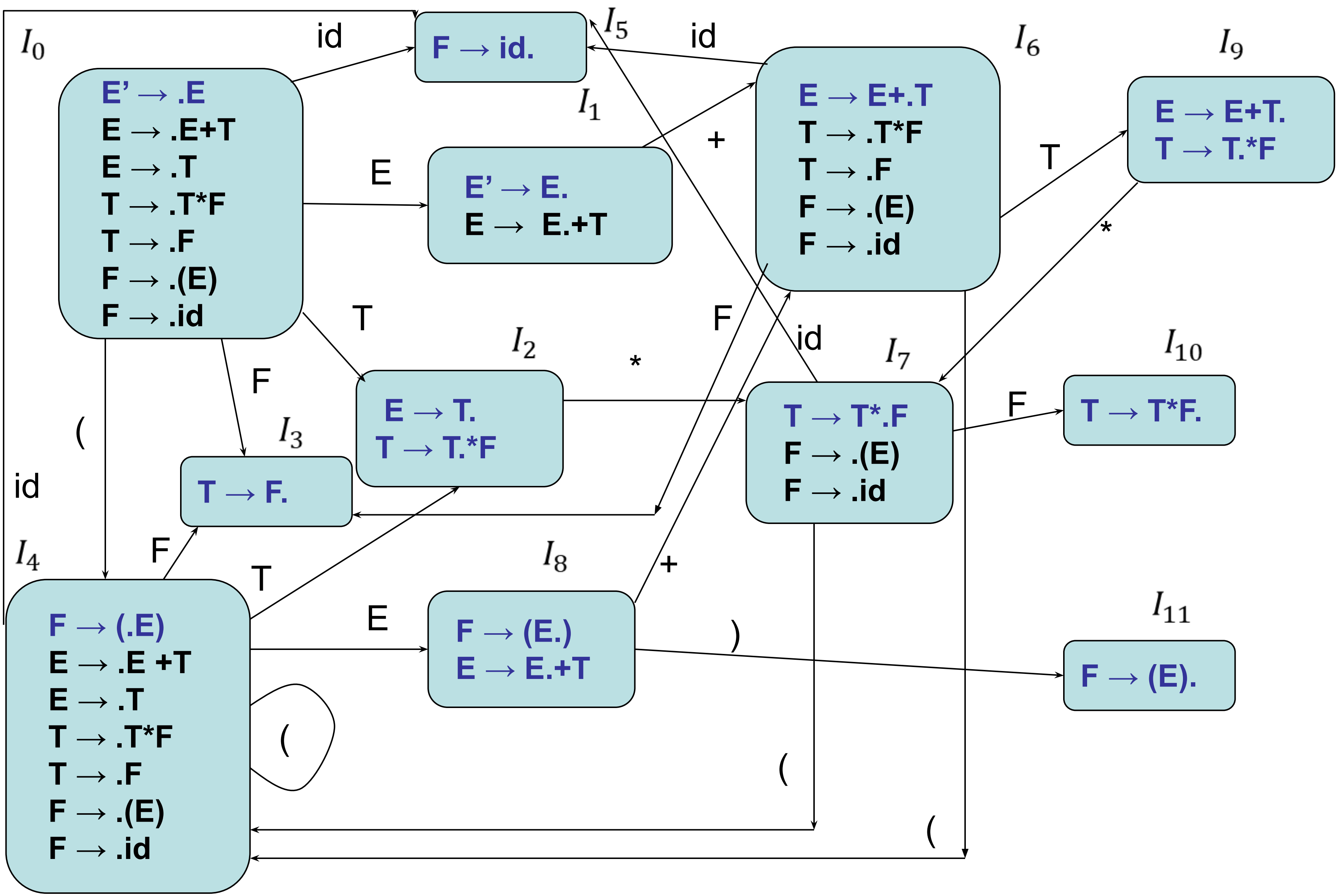
The Canonical LR(0) Collection -- Example



The Canonical LR(0) Collection -- Example



The Canonical LR(0) Collection -- Example



LR(0) Parsing Tables of Expression Grammar

	Action Table							Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2	r2	r2	s7/r2	r2	r2	r2				
3	r4	r4	r4	r4	r4	r4				
4	s5			s4				8	2	3
5	r6	r6	r6	r6	r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9	r1	r1	s7	r1	r1	r1				
10	r3	r3	r3	r3	r3	r3				
11	r5	r5	r5	r5	r5	r5				

The Canonical LR(0) Collection -- Example

I_0: $E' \rightarrow .E$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_1: $E' \rightarrow E.$
$E \rightarrow E.+T$

I_2: $E \rightarrow T.$
$T \rightarrow T.*F$

I_3: $T \rightarrow F.$

I_4: $F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_5: $F \rightarrow id.$

I_6: $E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_7: $T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_8: $F \rightarrow (E.)$
$E \rightarrow E.+T$

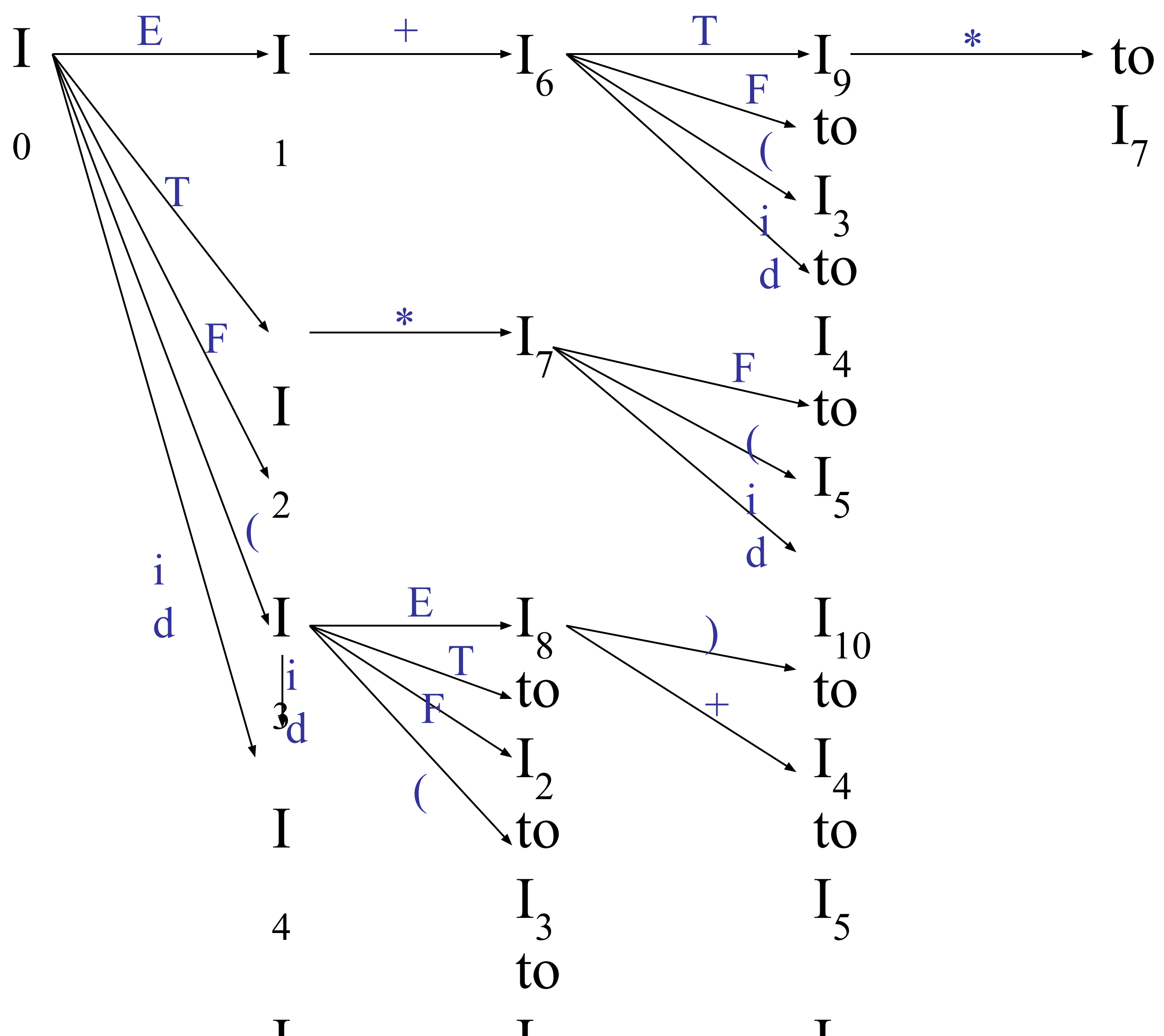
I_9: $E \rightarrow E+T.$
$T \rightarrow T.*F$

I_{10}: $T \rightarrow T*F.$

I_{11}: $F \rightarrow (E).$

FOLLOW(E) = {$,), +}
FOLLOW(T) = {+, *, $,)}
FOLLOW(F) = {+, *, $,)}

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0, \ldots, I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$

SLR Parsing Tables of Expression Grammar

	Action Table							Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-ahead LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

SLR(1) Parser Limitations

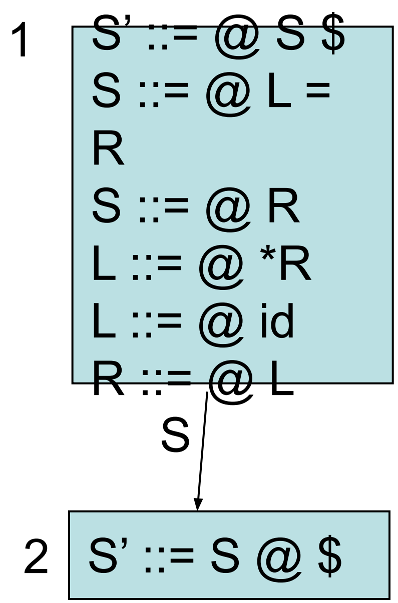
0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L

1

S’ ::= @ S $
S ::= @ L = R
S ::= @ R
L ::= @ *R
L ::= @ id
R ::= @ L

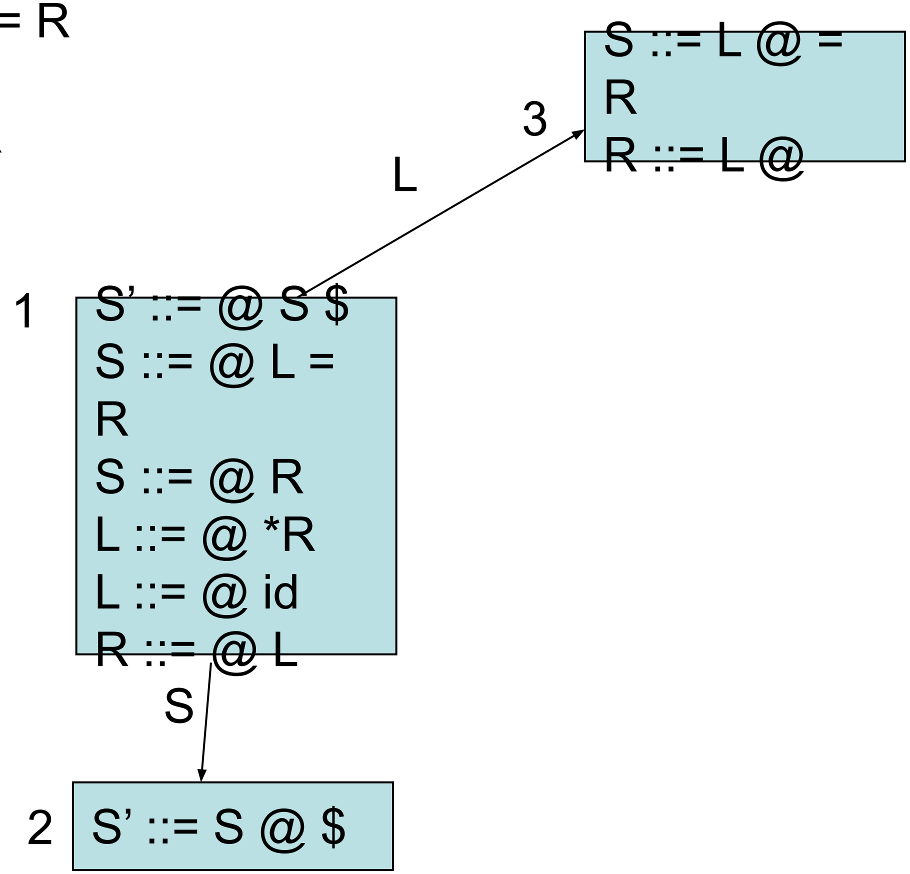
SLR(1) Parser Limitations

0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



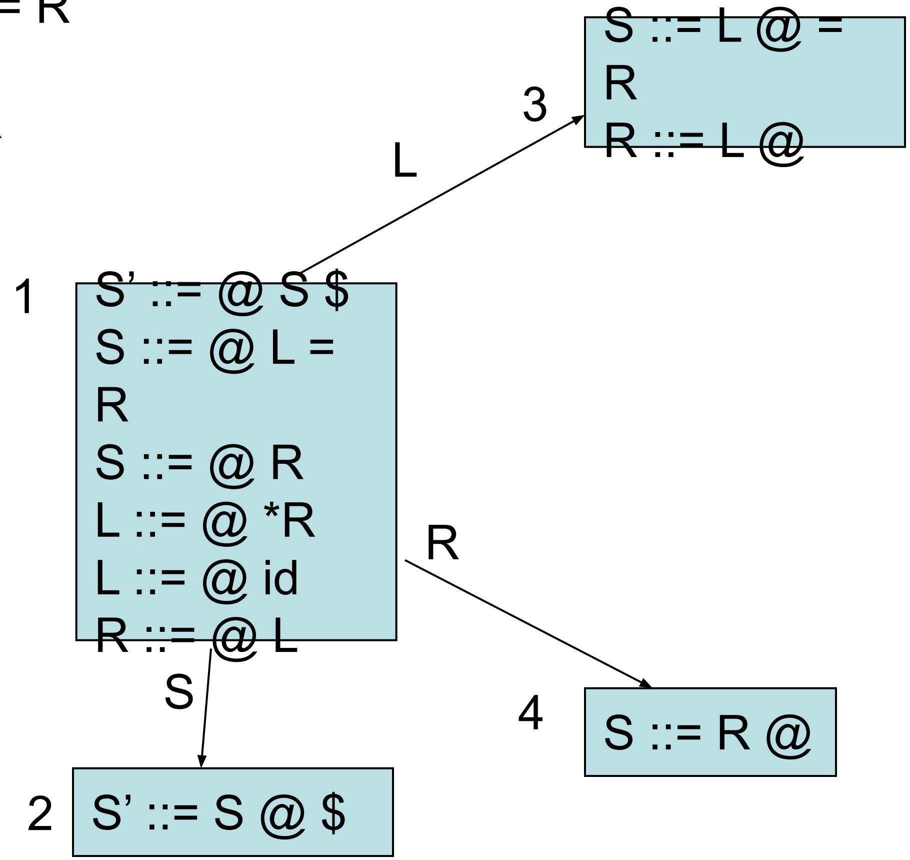
SLR(1) Parser Limitations

0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



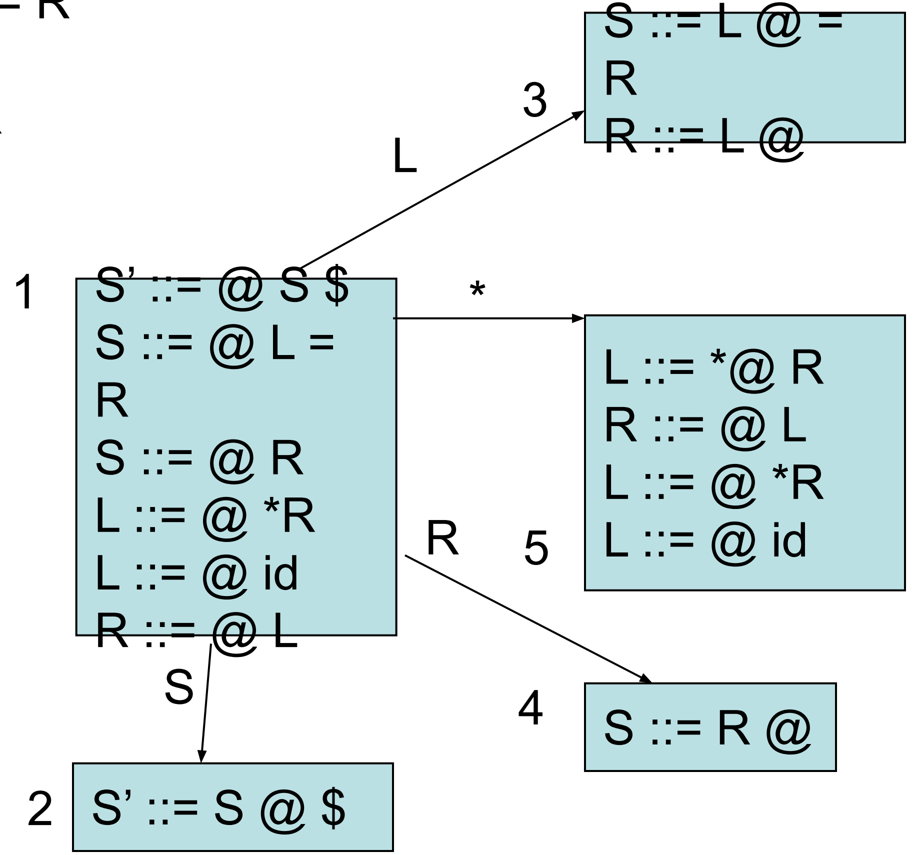
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



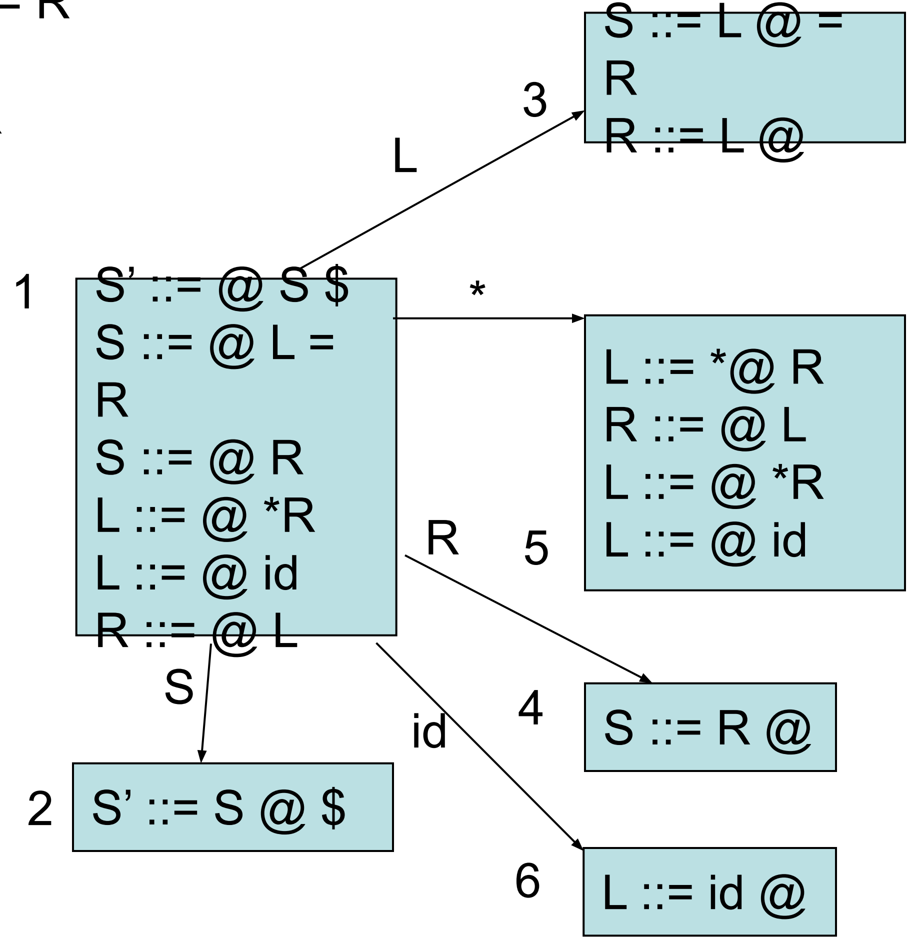
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



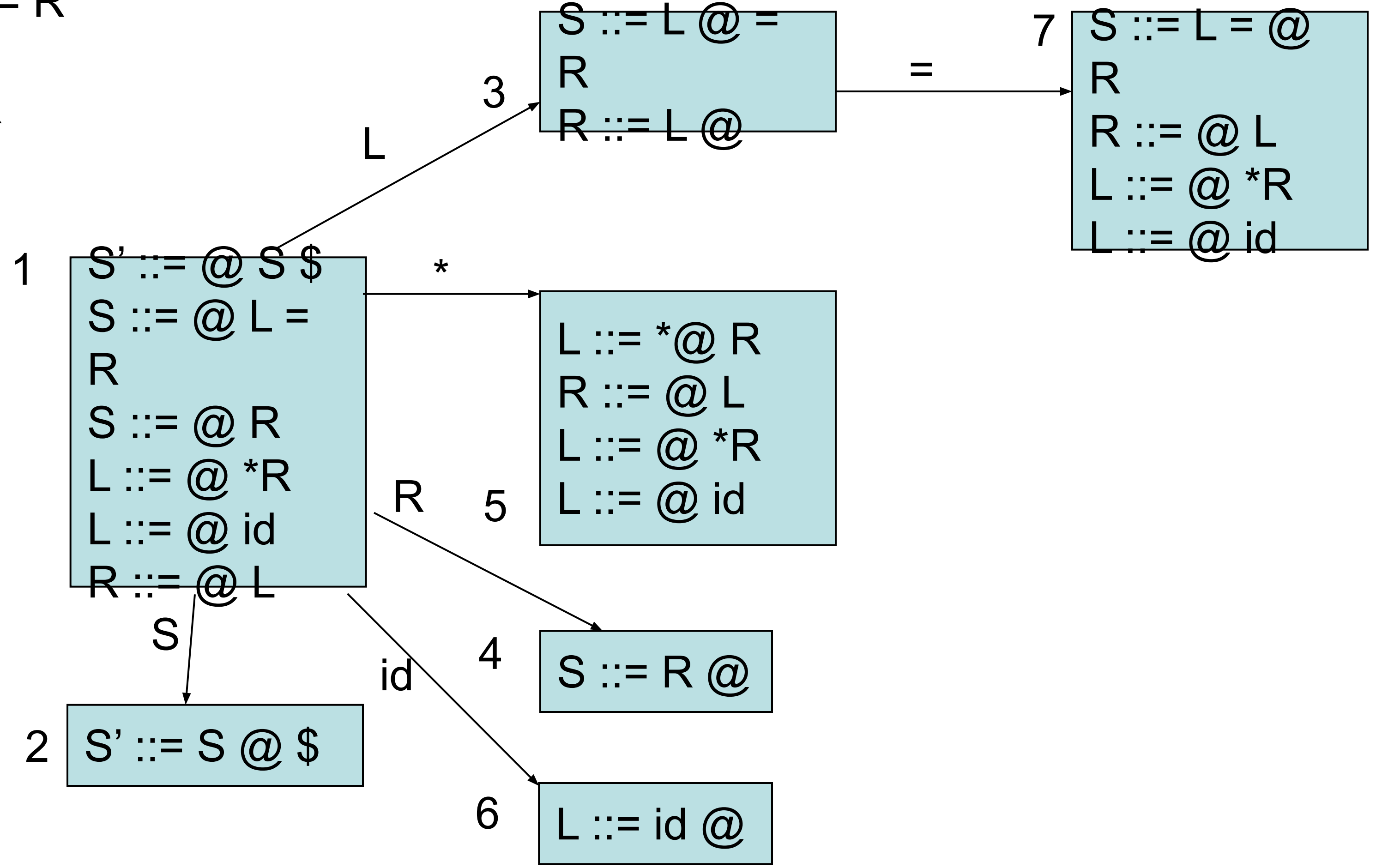
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



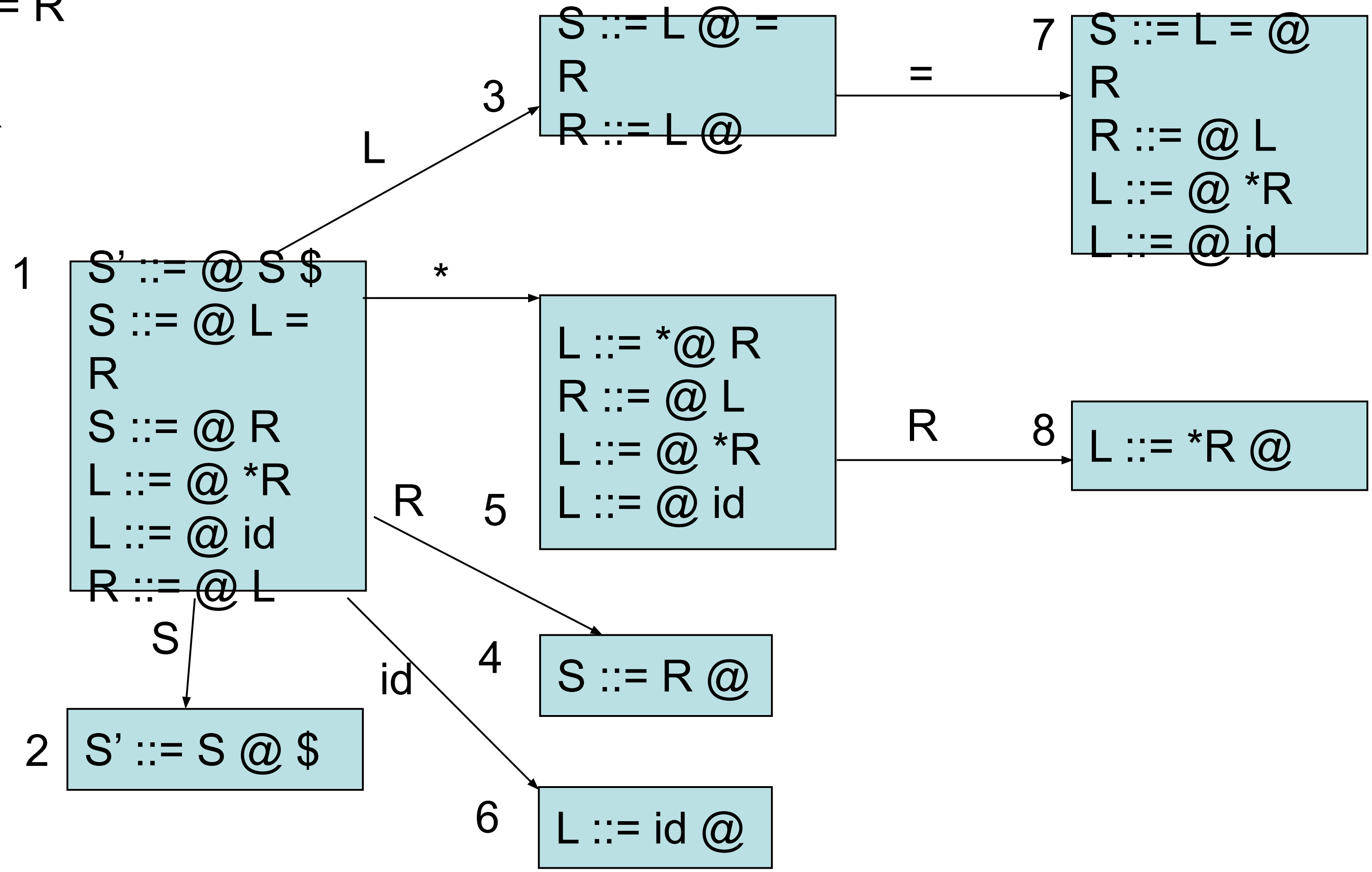
SLR(1) Parser Limitations

0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



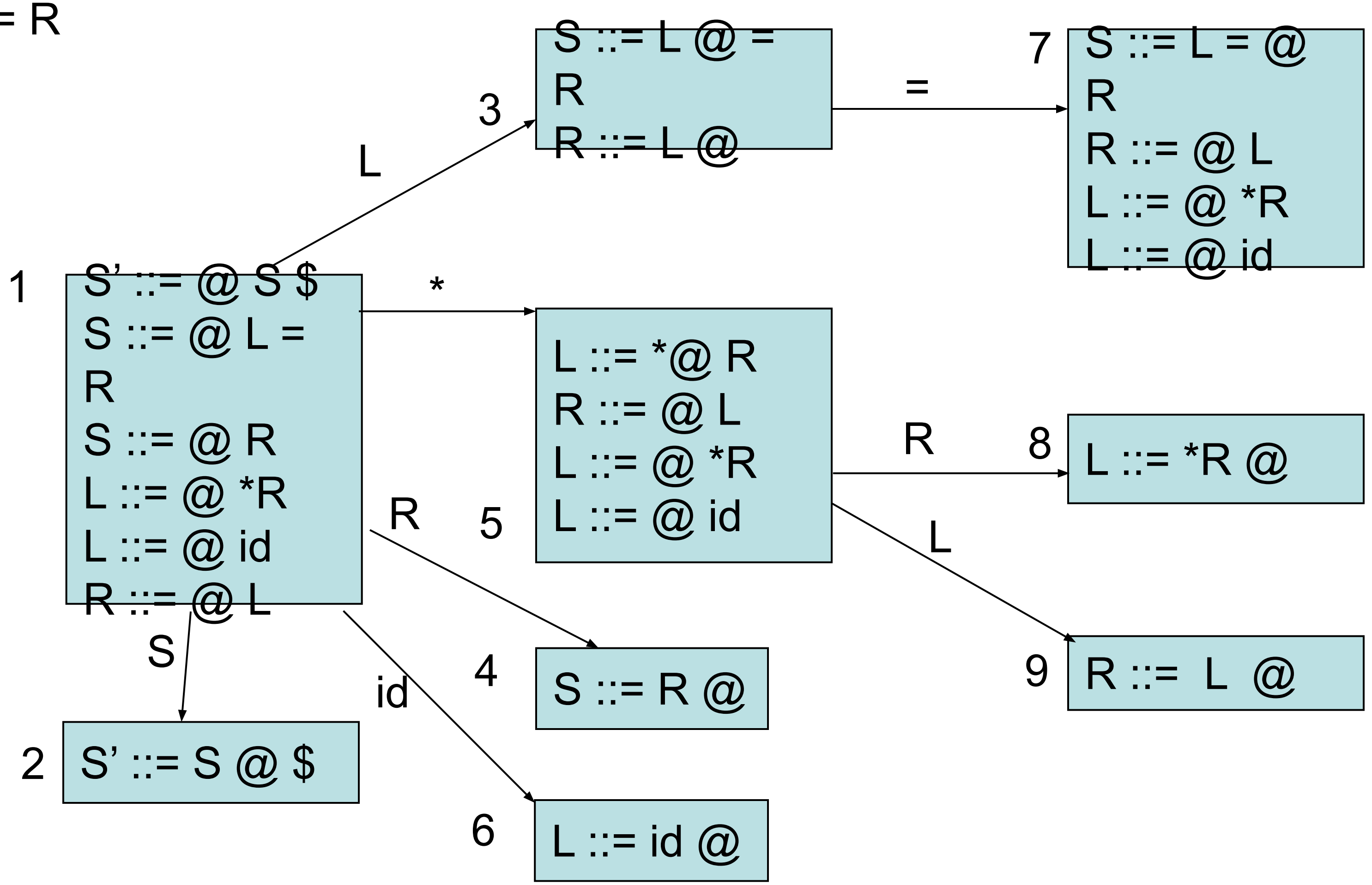
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



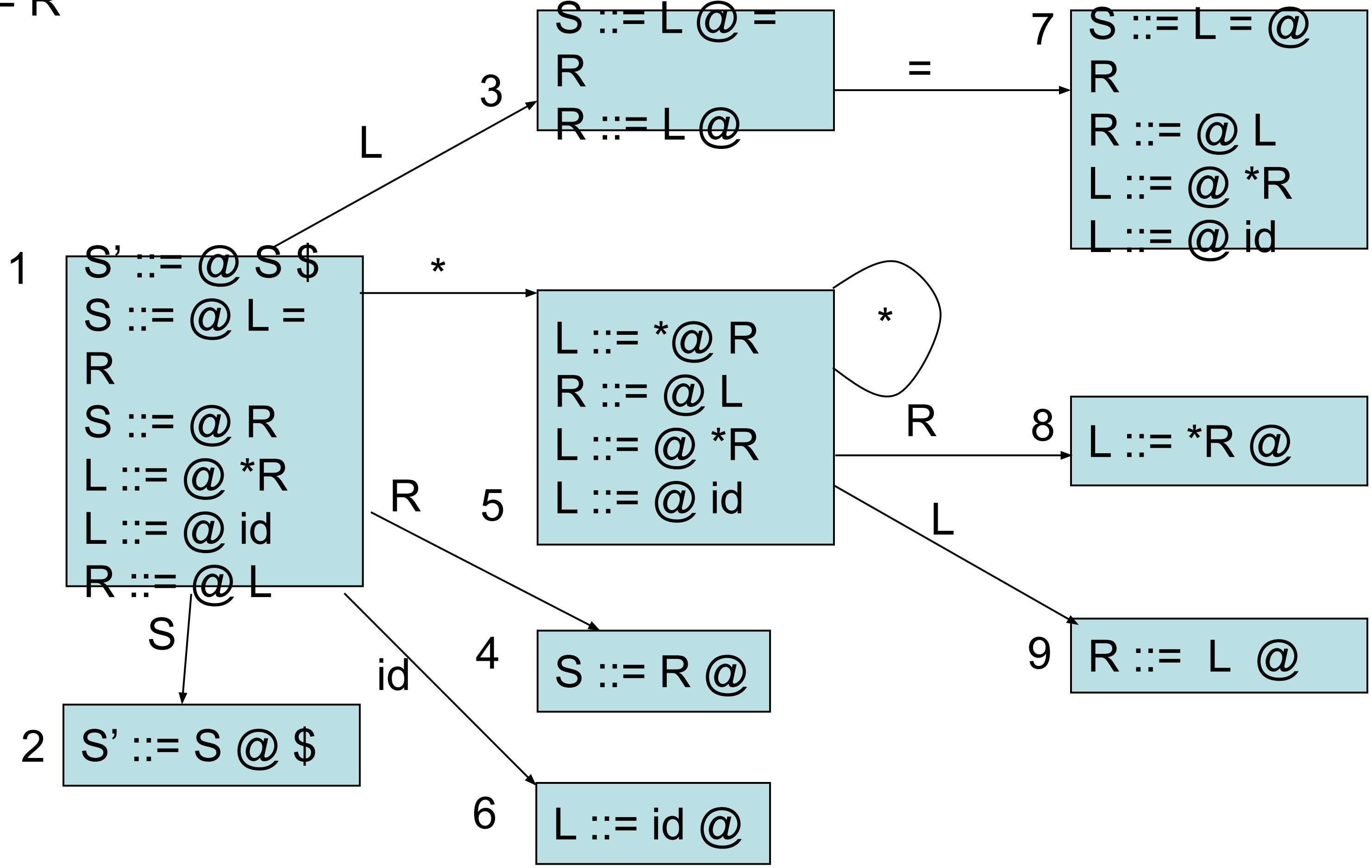
SLR(1) Parser Limitations

0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



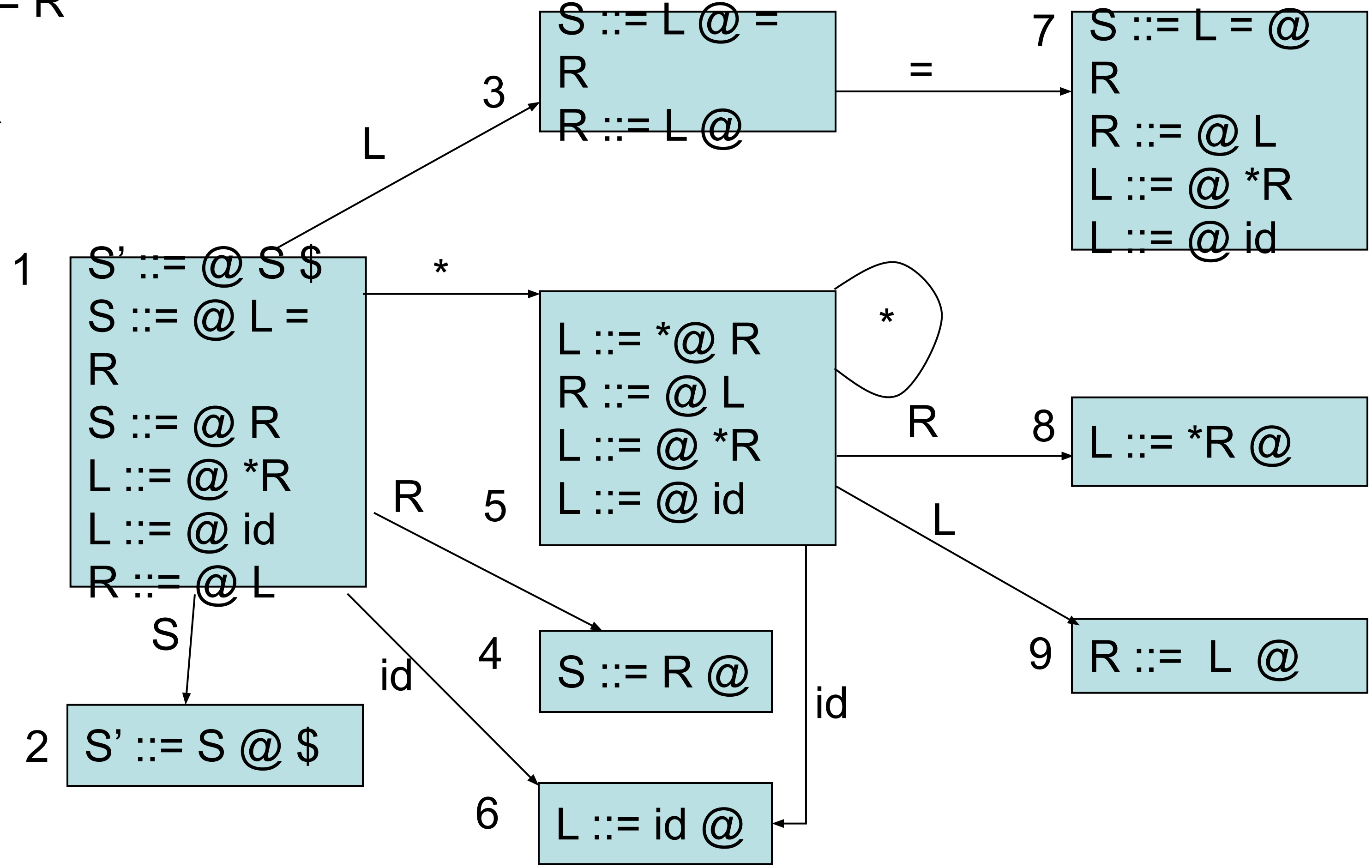
SLR(1) Parser Limitations

0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



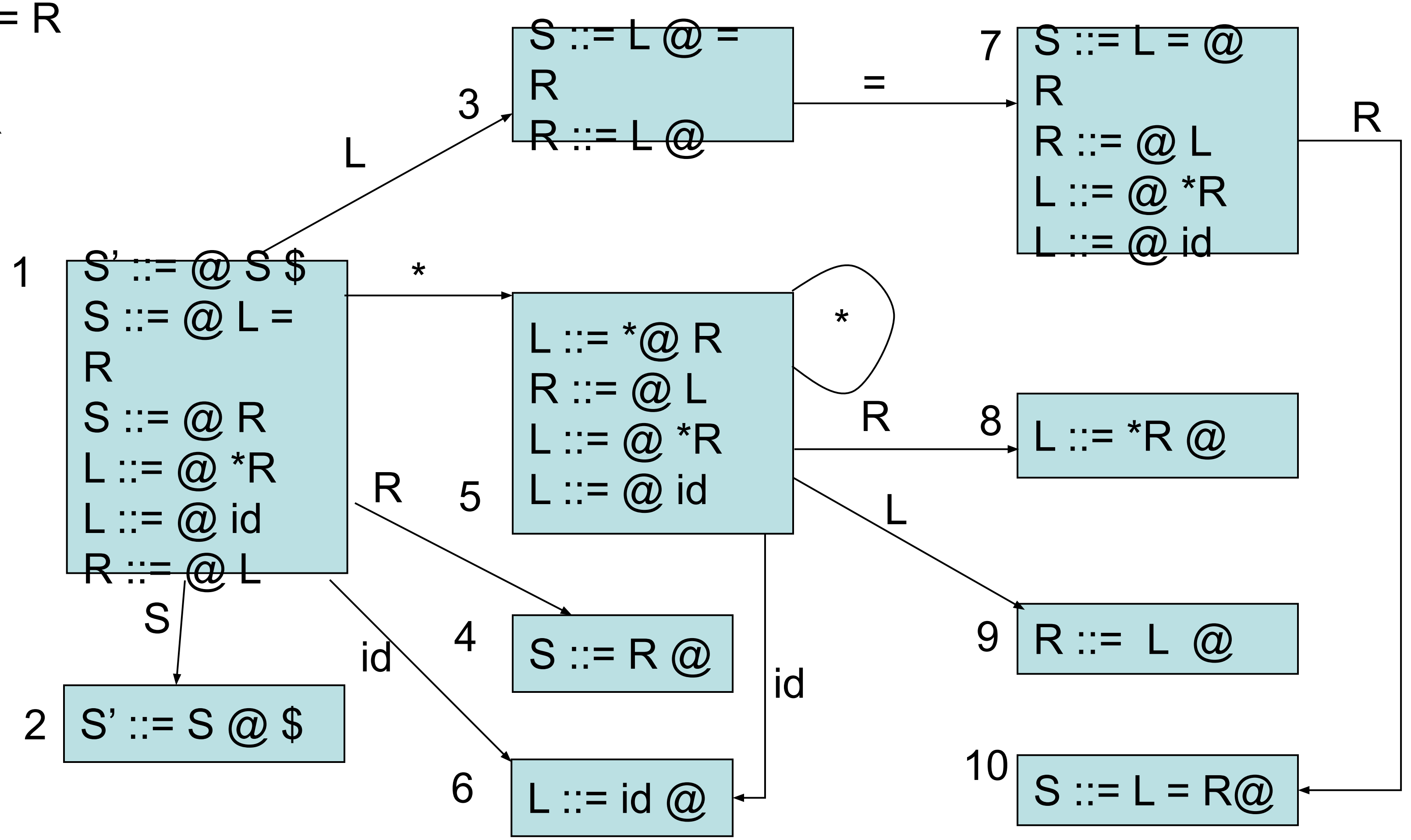
SLR(1) Parser Limitations

0. S’ ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



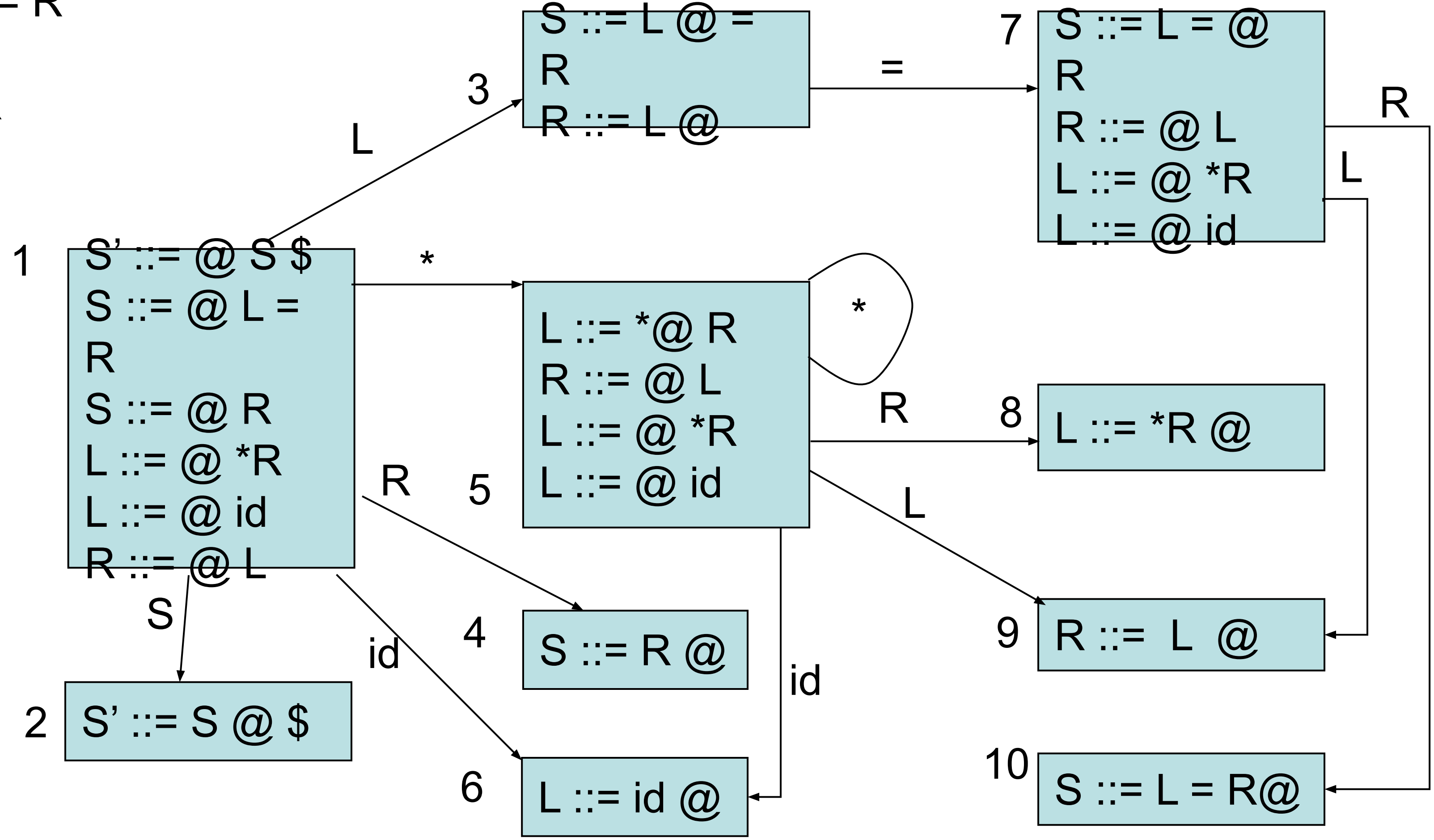
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



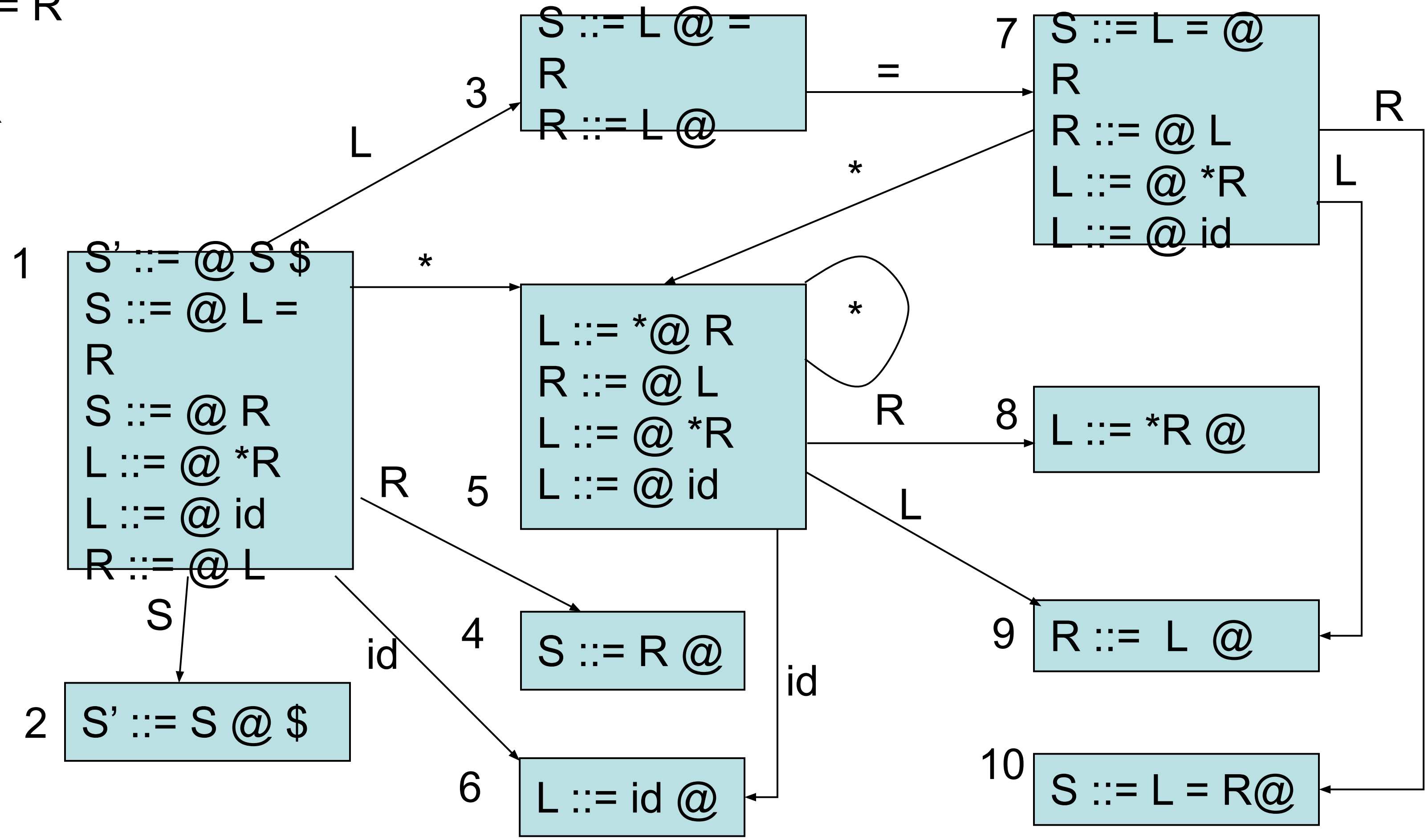
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



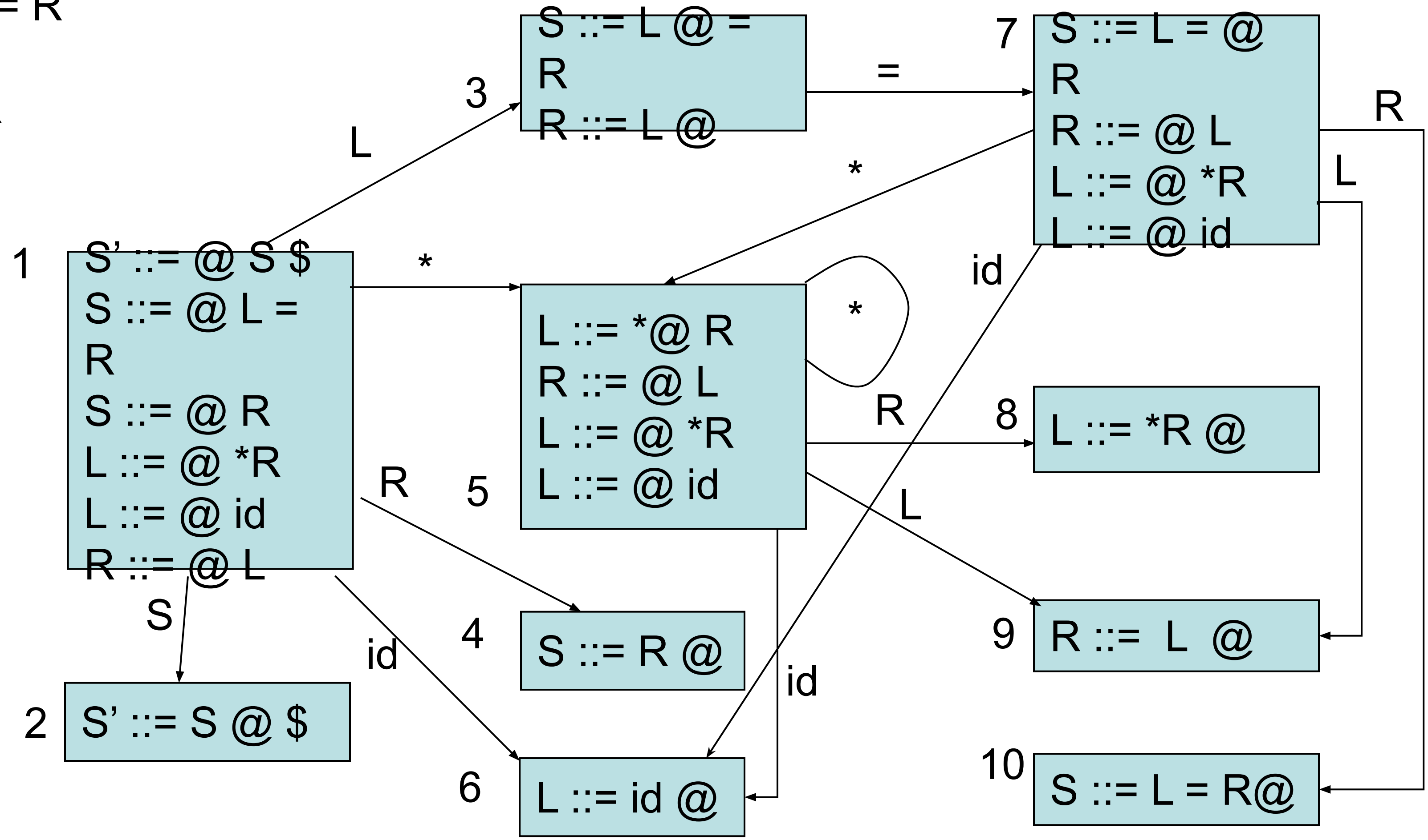
SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L



SLR(1) Parser Limitations

0. S' ::= S $
1. S ::= L = R
2. S ::= R
3. L ::= *R
4. L ::= id
5. R ::= L

Follow set of this Grammar

Follow(S) = {$}

Follow(L) = {=, $}

Follow(R) = {$, =}

state	=	*	id	$		S	L	R
1		s5	s6			2	3	4
2				acc				
3	s7/r5			r5				
4				r2				
5		s5	s6				9	8
6	r4			r4				
7		s5	s6				9	10
8	r3			r3				
9	r5			r5				
10				r1				

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

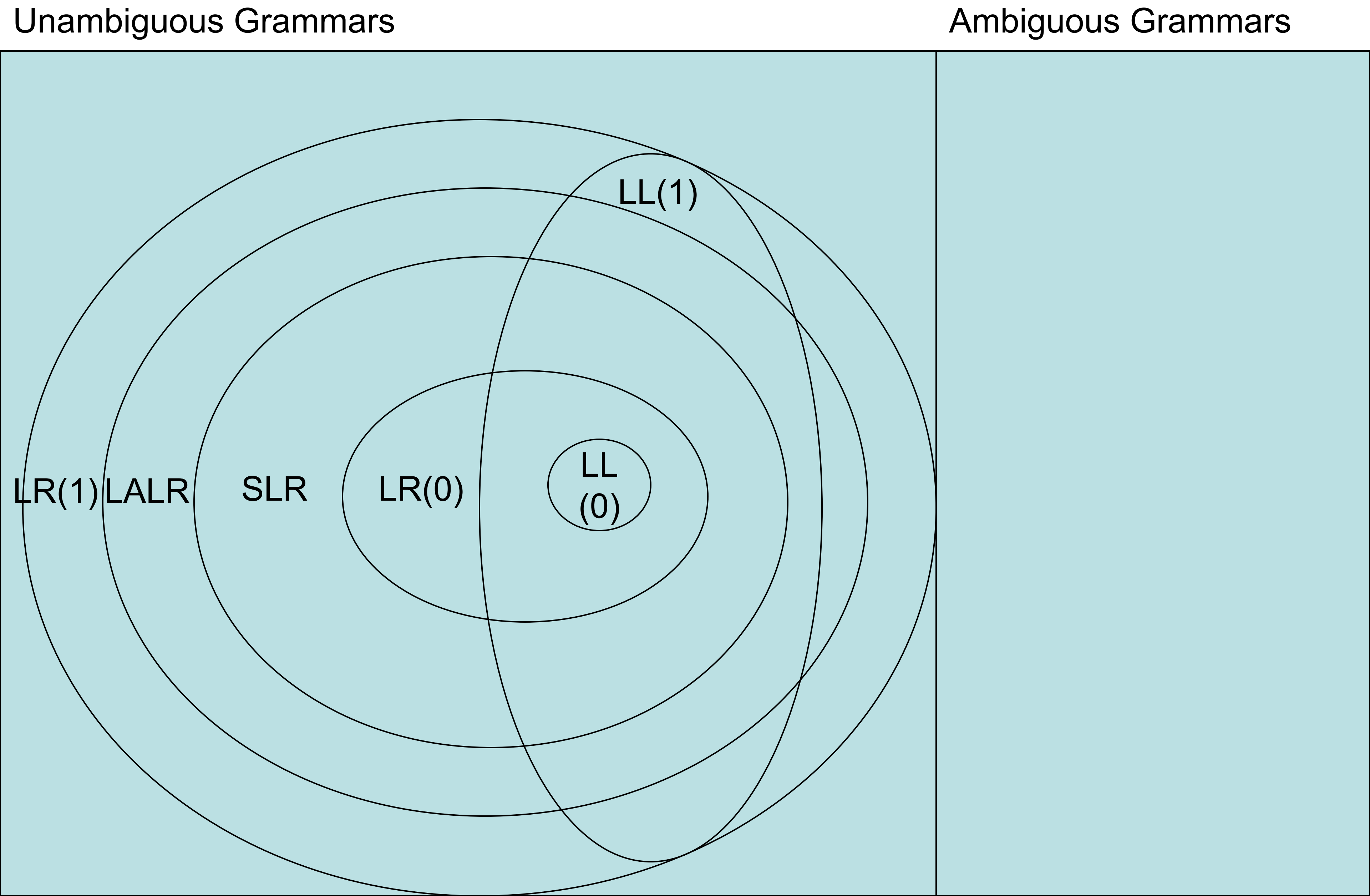
- LR(1) items

  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
- tTo 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)