Summary: Top-Down Parsing

- **Top-Down Parsing**
  - **Backtracking Parsing** ("brute force")
  - **Predictive Parsing** (lookahead)
    - **Recursive Descent Parsing**
      - a func for a nonterminal
      - leverage call stack
    - **LL(1) Parsing**
      - use explicit stack
      - First and Follow sets
      - parsing table-driven

Common Issues:

- left recursion
- common prefix

Example

S → A | B C
A → a A | ε
B → b B | ε
C → c C | d C | ε

	FIRST	FOLLOW
S	a, b,c,d,ε	$
A	a,ε	$
B	b,ε	c,d,$
C	c,d,ε	$

	a	b	c	d	$
S	S → A	S → B C	S → B C	S → B C	S → A S → B C
A	A → a A				A → ε
B		B → b B	B → ε	B → ε	B → ε
C			C → c C	C → d C	C → ε

Bottom-up Parsing

Basic Idea :

- Scan the input string from left to right.
- Try to construct a parse tree starting at the bottom (i.e., the leaves) and working towards the root.

Shift-reduce parsing :

Basic Idea : Apply a sequence of "*reductions*" to transform the input string to the start symbol of the grammar.

reduction: replace a substring matching the RHS of a production by the LHS.

Example : Consider the grammar

$$S \longrightarrow \mathbf{a}AB\mathbf{e}$$
$$A \longrightarrow A\mathbf{bc}$$
$$A \longrightarrow \mathbf{b}$$
$$B \longrightarrow \mathbf{d}$$

Input: $\mathbf{a}\underline{\mathbf{b}}\mathbf{bcde}$

$\rightsquigarrow \mathbf{a}\underline{A\mathbf{bc}}\mathbf{de}$

$\rightsquigarrow \mathbf{a}A\underline{\mathbf{d}}\mathbf{e}$

$\rightsquigarrow \underline{\mathbf{a}AB\mathbf{e}}$

$\rightsquigarrow S$

Handles

Intuition : A *handle* of a string s is a substring α s.t.:

1. α matches the RHS of a production $A \longrightarrow \alpha$; and

2. replacing α by the LHS A represents a step in the *reverse* of a *rightmost derivation* of s.

Example : Consider the grammar

$$S \longrightarrow \mathrm{a}AB\mathrm{e}$$
$$A \longrightarrow A\mathrm{bc} \mid \mathbf{b}$$
$$B \longrightarrow \mathrm{d}$$

The rightmost derivation for the input **abbcde** is:

$$\underline{S} \Rightarrow \mathbf{a}A\underline{B}\mathbf{e} \Rightarrow \mathbf{a}\underline{A}\mathbf{de} \Rightarrow \mathbf{a}\underline{A}\mathbf{bcde}$$
$$\Rightarrow \mathbf{abbcde}.$$

The string **a**A**bcde** can be reduced in two ways:

1. $\mathbf{a}\underline{A\mathbf{bc}}\mathbf{de} \leadsto \mathbf{a}A\mathbf{de}$; and

2. $\mathbf{a}A\mathbf{bc}\underline{\mathbf{d}}\mathbf{e} \leadsto \mathbf{a}A\mathbf{bc}B\mathbf{e}$.

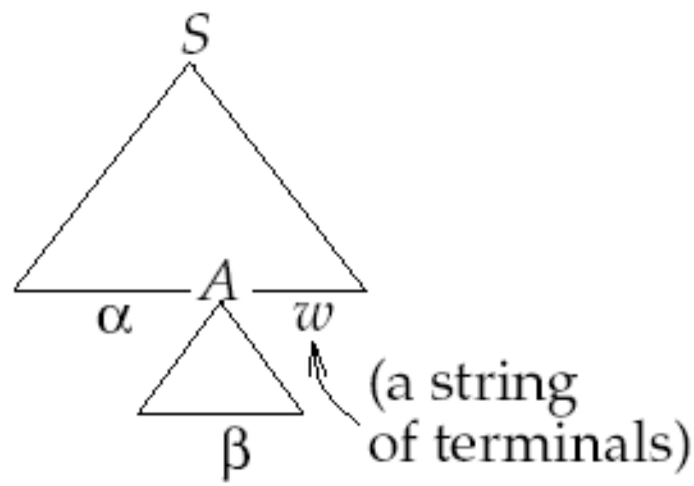
But (2) is not in a *rightmost* derivation, so A**bc** is the only handle.

Handles : cont'd

Definition : A *handle* of a right-sentential form γ is

1. a production $A \longrightarrow \beta$, and

2. a position in γ where β may be found and replaced by A to produce the *previous* sentential form in a rightmost derivation of γ.



The handle $A \longrightarrow \beta$ in $\alpha\beta\omega$

Shift Reduce Parsing

Stack	Tokens	
$	abc$	
$abc	$	Shifts
$S	$	Reduce

S → a b c

Stack Implementation of Shift-Reduce Parsing:

Data Structures :

- *the stack*, its bottom marked by \$, initially empty.
- *the input string*, its right end marked by \$, initially w.

Action :

repeat

1. *Shift* zero or more input symbols onto the stack, until a handle β is on the top of the stack.
2. *Reduce* β to the LHS of the appropriate production.

until ready to accept.

Acceptance : When the stack contains the start symbol and the input is empty.

Example : Consider the grammar

$$S \longrightarrow \mathrm{a}AB\mathrm{e}$$
$$A \longrightarrow A\mathrm{bc}$$
$$A \longrightarrow \mathrm{b}$$
$$B \longrightarrow \mathrm{d}$$

Input: abbcde
$\rightsquigarrow$ aAbcde
$\rightsquigarrow$ aAde
$\rightsquigarrow$ aABe
$\rightsquigarrow$ S

	c		e	
	b	d	B	
b	A	A	A	
a	a	a	a	S
$	$	$	$	$

Example :

Grammar: $S \longrightarrow \mathrm{a}AB\mathrm{e}$
$A \longrightarrow A\mathrm{bc} \mid \mathrm{b}$
$B \longrightarrow \mathrm{d}$

Input string : **abbcde**

Stack ($\rightarrow$)	*Input*	*Action*
$	**abbcde$**	shift
$**a**	**bbcde$**	shift
$**ab**	**bcde$**	reduce by $A \longrightarrow \mathrm{b}$
$**a**$A$	**bcde$**	shift
$**a**$A$**b**	**cde$**	shift
$**a**$A$**bc**	**de$**	reduce by $A \longrightarrow A\mathrm{bc}$
$**a**$A$	**de$**	shift
$**a**$A$**d**	**e$**	reduce by $B \longrightarrow \mathrm{d}$
$**a**$AB$	**e$**	shift
$**a**$AB$**e**	$	reduce by $S \longrightarrow \mathbf{a}AB\mathbf{e}$
$$S$	$	accept

Conflicts during Shift-Reduce Parsing :

1. Can't decide whether to shift or to reduce ("*shift-reduce conflict*").

   Example : "dangling else":

   $$\begin{aligned} Stmt \longrightarrow\ & \textbf{if}\ Expr\ \textbf{then}\ Stmt\ | \\ & \textbf{if}\ Expr\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt\ |\ \cdots \end{aligned}$$

2. Can't decide which of several possible reductions to make ("*reduce-reduce conflict*").

   Example :

   $$\begin{aligned} Stmt &\longrightarrow \textbf{id}\ (\ params\ )\ |\ Expr\ \textbf{:=}\ Expr\ |\ \cdots \\ Expr &\longrightarrow \textbf{id}\ (\ params\ ) \end{aligned}$$

   Given the input `A(I, J)` the parser doesn't know whether it's a procedure call or an array reference.

LR Parsing

- Bottom-up.
- LR(k) parser:
  - Scans the input **L**-to-R.
  - Produces a **R**ightmost derivation.
  - Uses $\underline{k}$-symbol lookahead.

Advantages :

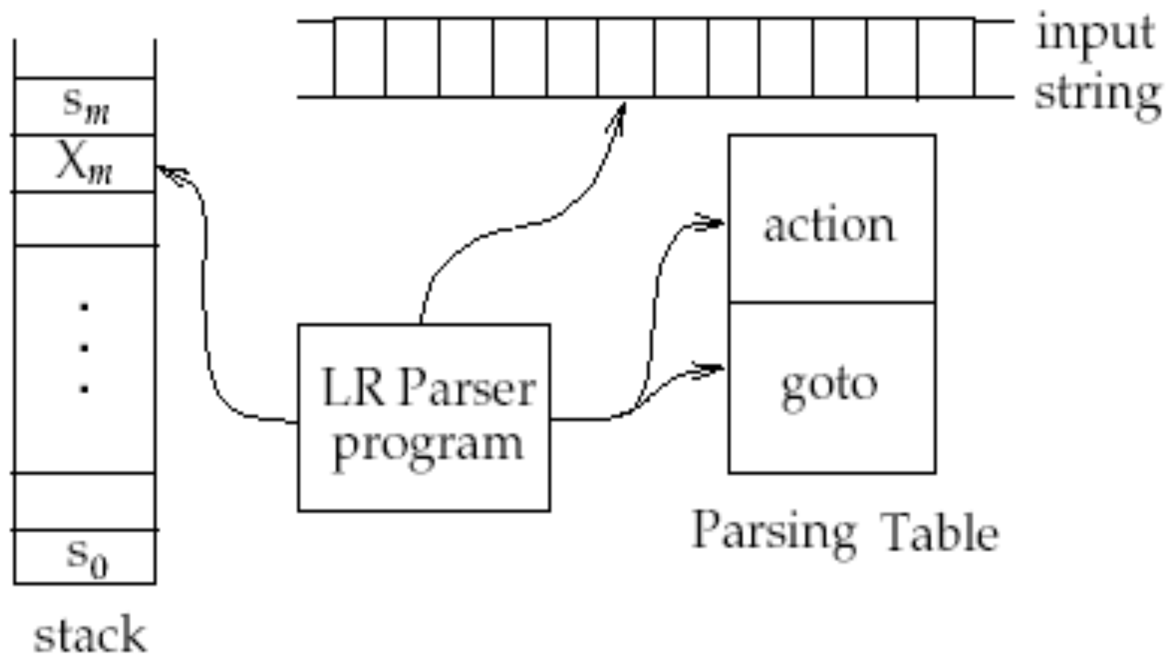
- Very general and flexible.
- Efficiently implemented.
- Parses a large class of grammars.

Disadvantages :

- Difficult to implement by hand for typical programming language grammars.

  (Use tools such as **yacc** or **bison.**)

Schematic of an LR Parser :



- The driver program is the same for all LR parsers (SLR(1), LALR(1), LR(1), ...) : only the parsing table changes.

- The stack holds strings of the form

$$s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m$$

where s_m is on top, the s_i are "states", and X_i are grammar symbols.

- The *configuration* of an LR parser is given by a pair ⟨stack contents, unexpended input⟩.

A configuration $\langle s_0 X_1 s_1 \cdots X_m s_m, \quad \mathrm{a}_i \mathrm{a}_{i+1} \cdots \mathrm{a}_n \rangle$ represents the right-sentential form

$$X_1 \cdots X_m \mathrm{a}_i \mathrm{a}_{i+1} \cdots \mathrm{a}_n$$

The sequence of symbols $X_1 \cdots X_m$ on the parser stack is called a *viable prefix* of the right sentential form.

LR Parse Tables

- The parsing table consists of two parts: a parsing **action** function, and a **goto** function.

- For a given configuration of the parser, the next move is determined by the parse table entry

  $\texttt{action}[s_m, \mathrm{a}_i]$.

  where s_m is the topmost state on the stack, and a_i is the next input symbol.

- An `action` table entry can be of four types:

  1. **shift** s, where s is a state.
  2. **reduce** by a grammar production $A \longrightarrow \beta$.
  3. **accept**
  4. **error**

UCR

LR Parsing : cont'd

Suppose the parser configuration is

$$\langle s_0 X_1 s_1 \cdots X_m s_m, \quad \mathbf{a}_i \cdots \mathbf{a}_n \$ \rangle.$$

- if $\mathtt{action}[s_m, \mathbf{a}_i] = \mathit{shift}\ s$ then the parser executes a *shift* move. The new configuration is

$$\langle s_0 X_1 s_1 \cdots X_m s_m \underbrace{\mathbf{a}_i\ s}_{pushed}, \quad \mathbf{a}_{i+1} \cdots \mathbf{a}_n \$ \rangle.$$

- if $\texttt{action}[s_m, \mathrm{a}_i] = $ *reduce* $A \longrightarrow \beta$ then the parser does a *reduce* move. The new configuration is

$$\langle s_0 X_1 s_1 \cdots X_{m-r} s_{m-r} \underbrace{A\ s,}_{new} \quad \mathrm{a}_i, \cdots \mathrm{a}_n \$ \rangle.$$

  where

  - $r =$ length of β; and
  - $s = \texttt{goto}[s_{m-r}, A]$.

- if $\texttt{action}[s_m, \mathrm{a}_i] = $ *accept* then parsing is done.

- if $\texttt{action}[s_m, \mathrm{a}_i] = $ *error* the parser calls an error recovery routine.

5.2. Finite Automata to recognize Viable Prefixes

Definition : An *LR(0) item* of a grammar G is is a production of G with a dot '$\bullet$' added at some position in the RHS.

Example : The production $A \longrightarrow \mathrm{a}A\mathrm{b}$ gives the items

$$A \longrightarrow \bullet \mathrm{a}A\mathrm{b}$$
$$A \longrightarrow \mathrm{a}\bullet A\mathrm{b}$$
$$A \longrightarrow \mathrm{a}A\bullet\mathrm{b}$$
$$A \longrightarrow \mathrm{a}A\mathrm{b}\bullet$$

Intuition : An item $A \longrightarrow \alpha\bullet\beta$ denotes:

– we have seen a string derivable from α; and

– we hope to see a string derivable from β.

<u>Overall Goal</u> : Given a grammar with start symbol S,

- Construct an *<u>augmented grammar</u>* by adding a new start symbol S' and production $S' \to S$;

- Starting with the item $S' \to .S$, recognize the viable prefix $S' \to S.$.

Viable Prefix DFA

1. *closure* :

Definition : If I is a set of items for a grammar G, then $closure(I)$ is the set of items constructed as follows:

repeat

1. add every item in I to $closure(I)$;
2. if $A \longrightarrow \alpha \bullet B\beta$ is in $closure(I)$ and $B \longrightarrow \gamma$ is a production of G, then add $B \longrightarrow \bullet\gamma$ to $closure(I)$.

until no new item can be added to $closure(I)$.

Intuition : If $A \longrightarrow \alpha \bullet B\beta$ is in $closure(I)$ then we hope to see a string derivable from B in the input. So if $B \longrightarrow \gamma$ is a production of G, then we should hope to see a string derivable from γ in the input. Hence, $B \longrightarrow \bullet\gamma$ is in $closure(I)$.

Viable Prefix DFA – cont'd:

2. *goto* :

Definition : If I is a set of items for a grammar G and X a grammar symbol, then $goto(I, X)$ is the set of items

$$closure(\{A \longrightarrow \alpha X \bullet \beta \mid A \longrightarrow \alpha \bullet X\beta \in I\}).$$

Intuition :

- A set of items I corresponds to a state.
- If $A \longrightarrow \alpha \bullet X\beta \in I$ then
  - we've seen a string derivable from α; and
  - we hope to see a string derivable from $X\beta$;

- now suppose we see a string derivable from X : the resulting state should be one in which:
  - we've seen a string derivable from αX; and
  - we hope to see a string derivable from β;
- The item corresponding to this is $A \longrightarrow \alpha X \bullet \beta$.

Constructing the Viable Prefix DFA for LR(0) Items

- Given a grammar G with start symbol S, construct the *augmented grammar* by adding a special production

  $$S' \longrightarrow S$$

  where S' does not appear in G.

- Algorithm for constructing the canonical collection of LR(0) items for an augmented grammar G':

begin
    $C := \{closure(\{S' \longrightarrow \cdot S\})\}$;
    repeat
        for each set of items $I \in C$ **do**
            for each grammar symbol X **do**
                if $goto(I, X) \neq \emptyset$ **then**
                    add $goto(I, X)$ to C;
                fi
    until no new set of items can be added to C;
    return C;
end

Example

Original Grammar

E → E + T | T

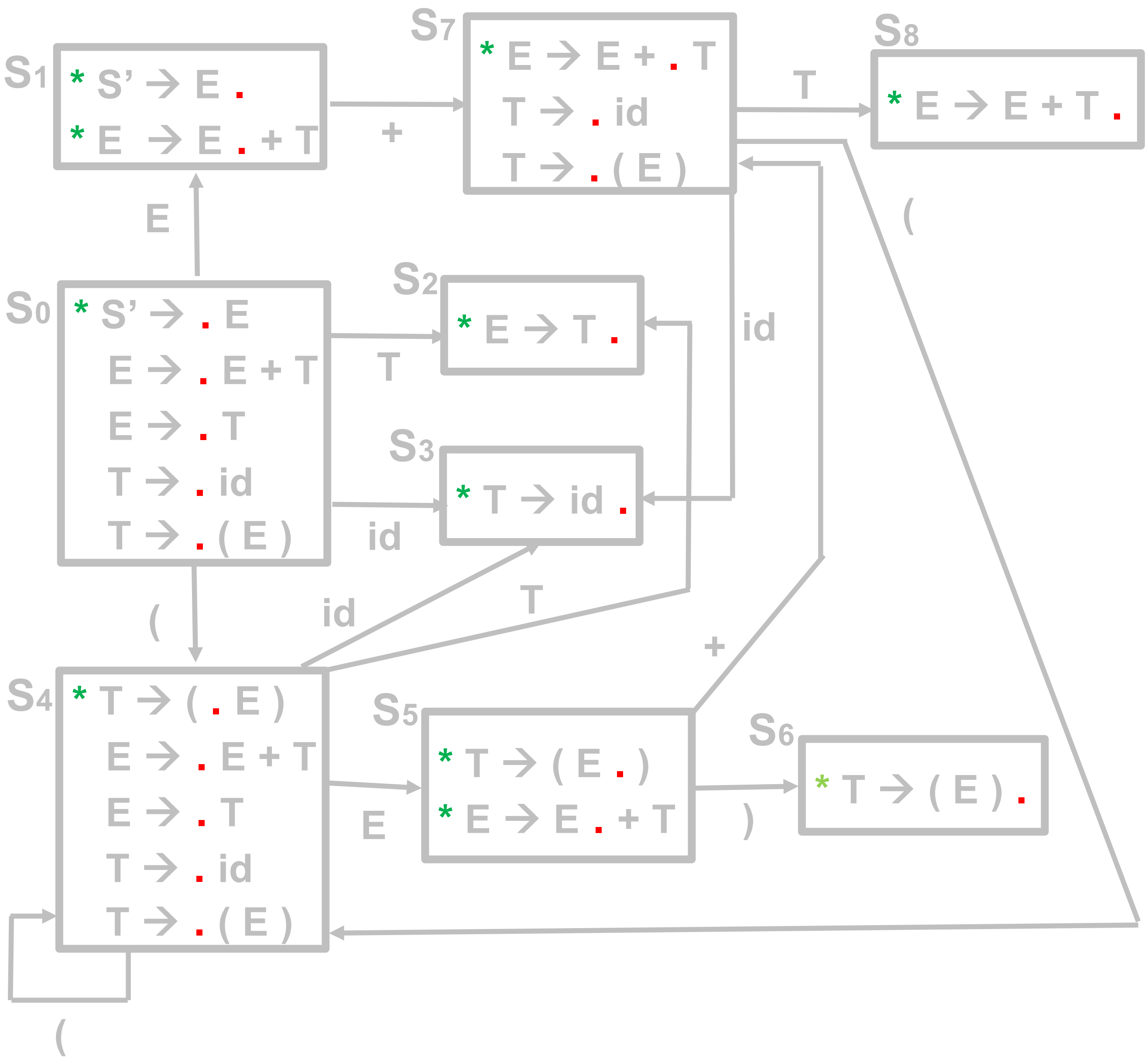
T → id | (E)

Augmented Grammar

S' → E

E → E + T | T

T → id | (E)



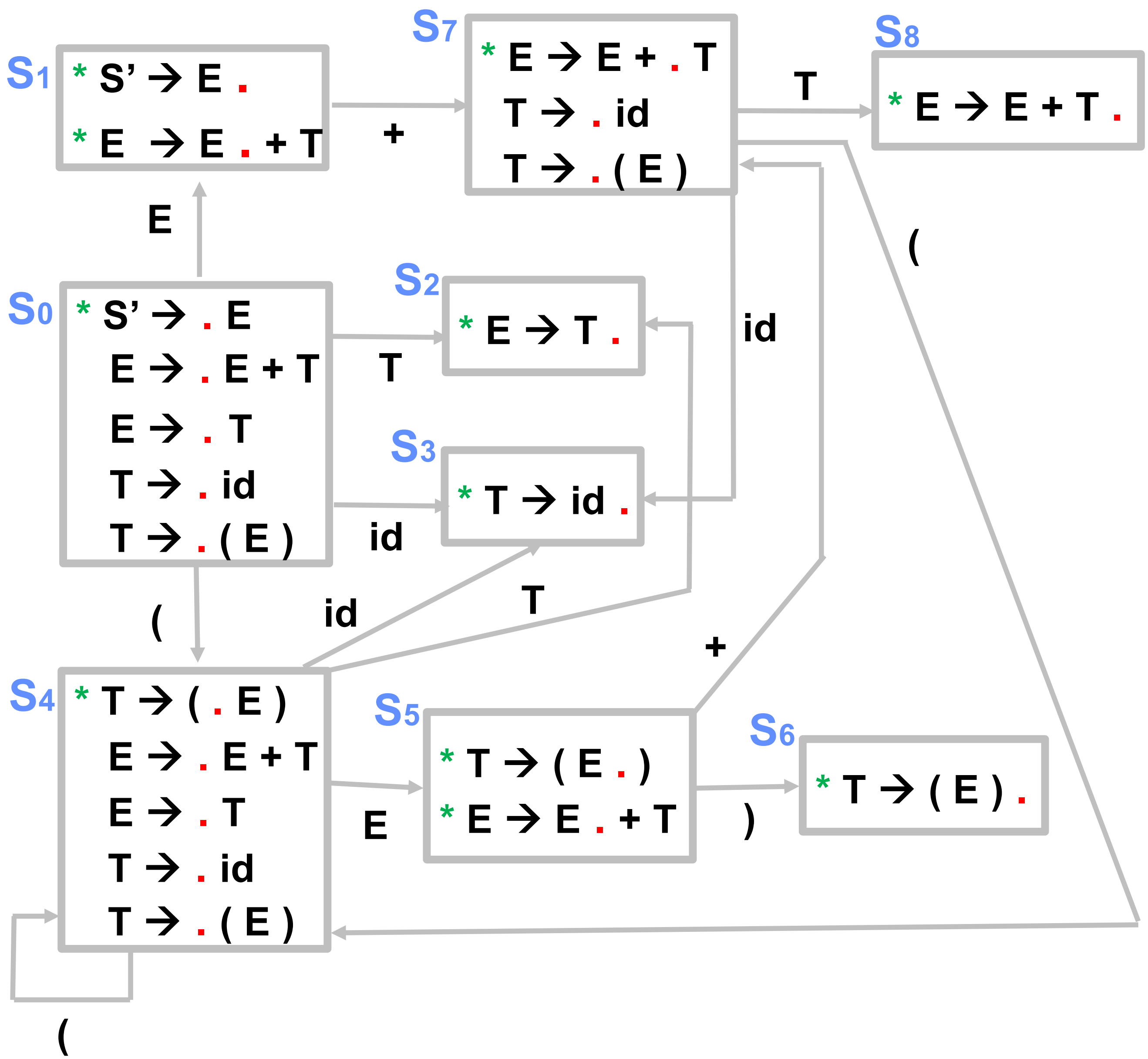
Augmented Grammar

S' → E
E → E + T | T
T → id | (E)

Kernel items are Marked with *

Rest of the items added by closure

. Tells where we are in the production



Augmented Grammar

S' → E

E → E + T | T

T → id | (E)

Kernel items are Marked with *

Rest of the items added by closure

. Tells where we are in the production

5.3. Constructing an SLR(1) Parse Table

1. Given a grammar G, construct the augmented grammar G' by adding the production $S' \longrightarrow S$.

2. Construct $C = \{I_0, \ldots, I_n\}$, the set of states of the viable prefix DFA for G'.

3. State i is constructed from I_i, with parsing action determined as follows:

   (a) $A \longrightarrow \alpha \bullet \mathrm{a}\beta \in I_i$, **a** a terminal, $goto(I_i, \mathrm{a}) = I_j$: set $action[i, \mathrm{a}] =$ shift j.

   (b) $A \longrightarrow \alpha \bullet \in I_i, A \neq S'$: for each a $\in$ FOLLOW(A), set $action[i, \mathrm{a}] =$ reduce $A \longrightarrow \alpha$.

   (c) $S' \longrightarrow S \bullet \in I_i$: set $action[i, \$] =$ accept.

4. goto transitions are constructed as follows: for each nonterminal A, if $goto(I_i, A) = I_j$ then $goto[i, A] = j$.

5. All entries not defined by the above steps are made *error*.

   If there are any multiply defined entries, then G is not SLR.

6. Initial state of the parser: that constructed from $I_0 \sim S' \longrightarrow \bullet S$.

	ACTION					GOTO		
	+	Id	(	)	$	E	T	S
S0		S,S3	S,S4			S1	S2	
S1	S,S7				accept			
S2	R,#3			R,#3	R,#3			
S3	R,#4			R,#4	R,#4			
S4		S,S3	S,S4			S5	S2	
S5	S,S7			S,S6				
S6	R,#5			R,#5	R,#5			
S7		S,S3	S,S4				S8	
S8	R,#2			R,#2	R,#2			

#1 S’ → E

#2 E → E + T

#3 E → T

#4 T → id

#5 T → (E)

Follow(S’) → { $ }

Follow(E) → { +,), $ }

Follow(T) → { +,), $ }

S - SHIFT R - REDUCE

S# - Next State

#n - Production Rule Number

The LR Parsing Algorithm

begin
set ip to point to the first symbol of the input $w\$$;

while TRUE **do**
let s be the state on top of the stack,
a the symbol pointed at by ip;

if $\texttt{action}[s, a]$ = shift s' **then**
push a then s' on top of the stack;
advance ip to the next input symbol;

else if $\texttt{action}[s, a]$ = reduce $A \longrightarrow \beta$ **then**
pop $2* \mid \beta \mid$ symbols off the stack;
let s' be the state now on top of the stack;
push A then $\texttt{goto}[A, s']$ on top of the stack;

else if $\texttt{action}[s, a]$ = accept **then return**;

else *error()*;
fi
od
end

Stack	Input	Action
$ S0	id + id $	shift S3
$ S0 id S3	+ id $	red. T→id GOTO[S0,T]=S2
$ S0 T S2	+ id $	red E→T GOTO[S0,E]=S1
$ S0 E S1	+ id $	shift S7
$ S0 E S1 + S7	id $	shift S3
$ S0 E S1 + S7 id S3	$	red T→id GOTO[S7,T]=S8
$ S0 E S1 + S7 T S8	$	red E→E+T GOTO[S0,E]=S1
$ S0 E S1	$	accept

Limitations of SLR Parsing

Cannot handle many "reasonable" grammars, e.g.:

$$S \longrightarrow R \mid L{=}R$$
$$L \longrightarrow * \; R \mid \mathrm{id}$$
$$R \longrightarrow L$$

The SLR parse table contains a state

$$I = \{S \longrightarrow L{\bullet}{=}R, R \longrightarrow L{\bullet}\}$$

which causes a shift/reduce conflict on '=', since '=' is in FOLLOW(L).

Problem : For an input

***id = id**

we want to remember enough "left context" after seeing * to make the right shift/reduce decision. SLR cannot do this adequately.

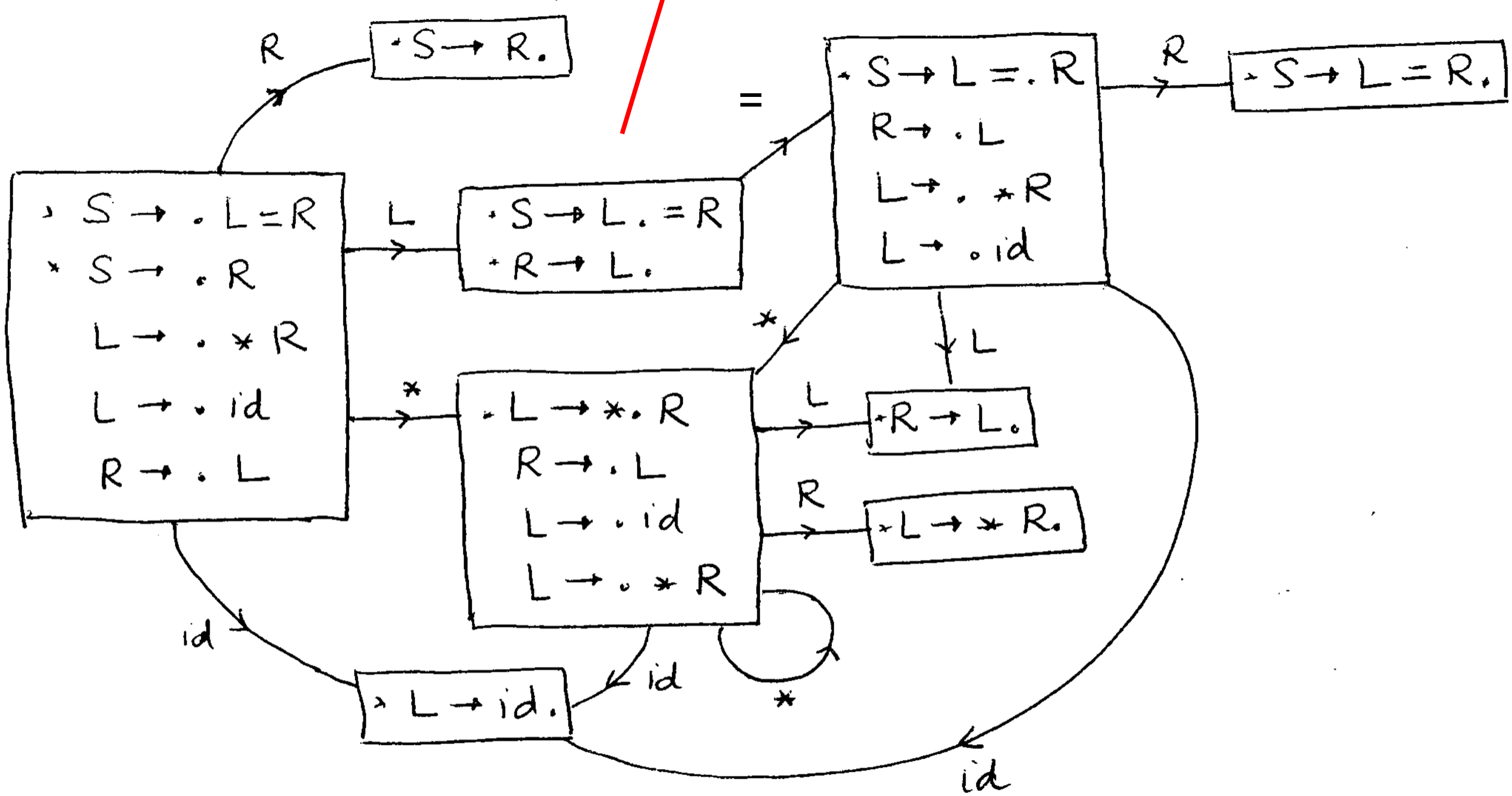
S → L = R | R

R → L

L → * R | id

S → L . = R → Shift on =

R → L . → Since = is in Follow(R) so Reduce on =



5.4. LR(1) Parsing

Idea : Extend SLR parsing to incorporate lookahead.

LR(1) Item :

- Of the form $[A \longrightarrow \alpha \cdot \beta, \mathrm{a}]$, where **a** is a terminal or is the endmarker $.
- The lookahead has no effect on items of the form $[A \longrightarrow \alpha \cdot \beta, \mathrm{a}]$, where $\beta \neq \varepsilon$.
- For items of the form $[A \longrightarrow \alpha \cdot, \mathrm{a}]$, reduce only if the next symbol is **a**.

Note: For an item of the form $[A \longrightarrow \alpha \cdot \beta, \mathrm{a}]$, $\mathrm{a} \in$ FOLLOW(A). But there may be $\mathrm{b} \in$ FOLLOW(A) for which there is no item $[A \longrightarrow \alpha \cdot \beta, \mathrm{b}]$.

LR(1) Parsing: *closure* and *goto* Functions

1. *closure*(I) :

begin
$S := I$;
  repeat
    for(each item $[A \longrightarrow \alpha \cdot B\beta, \mathrm{a}] \in I$,
        each production $B \longrightarrow \gamma$,
        each terminal $\mathrm{b} \in$ FIRST$(\beta\mathrm{a})$) **do**
      add $[B \longrightarrow \cdot\gamma, \mathrm{b}]$ to S;
  until no new item can be added to S;
  return S;
end

2. $goto(I, X)$:

begin
  let $J = \{[A \longrightarrow \alpha X \cdot \beta, \mathrm{a}] \mid [A \longrightarrow \alpha \cdot X\beta, \mathrm{a}] \in I\}$;
  return $closure(J)$;
end

Constructing the Viable Prefix DFA for LR(1) Items

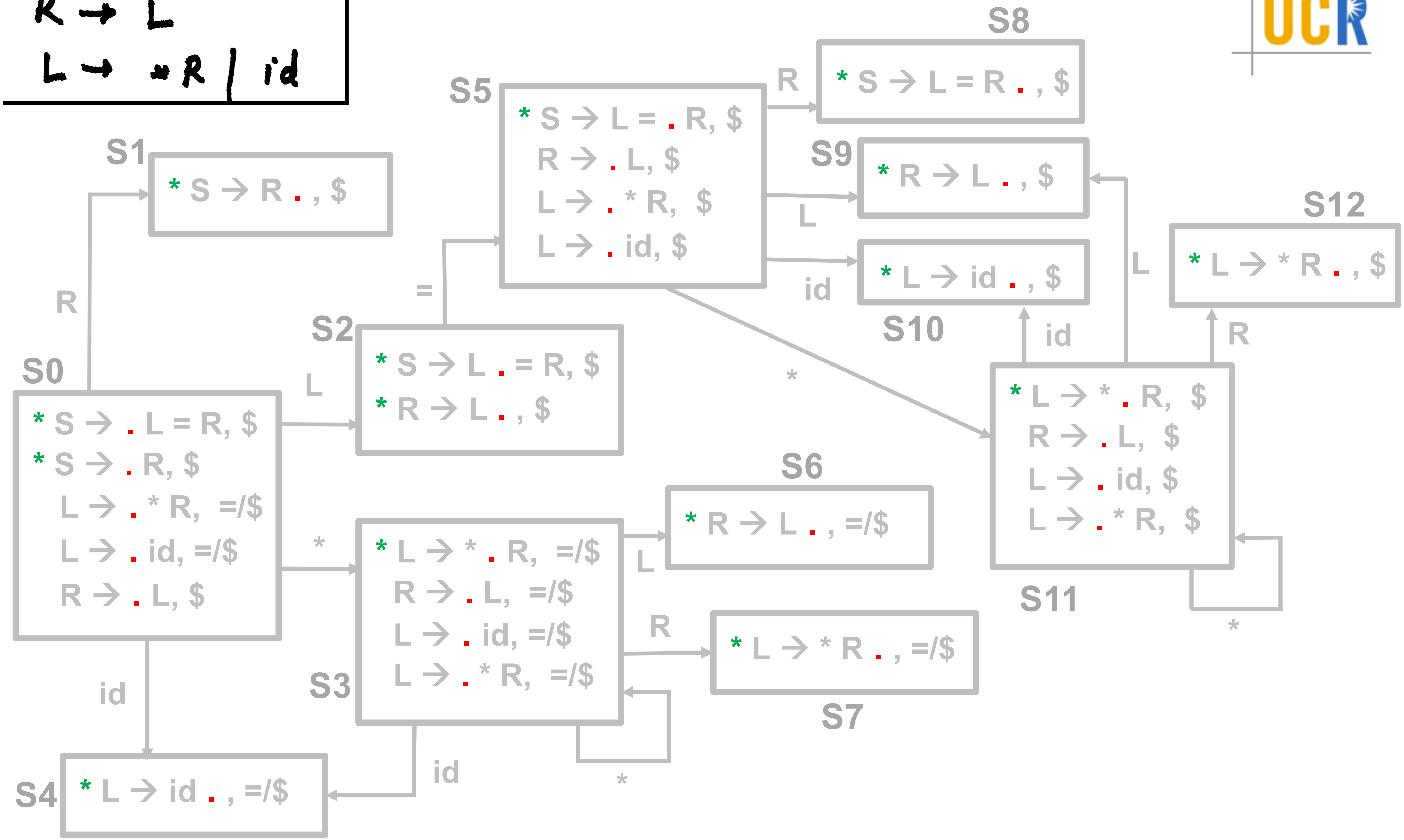
- *Given* : An augmented grammar G'.

- *Algorithm* :

begin
  $C := \{closure(\{[S' \longrightarrow \bullet S, \$]\})\}$;
  repeat
    for each set of items $I \in C$ **do**
      for each grammar symbol X **do**
        if $goto(I, X) \neq \emptyset$ **then**
          add $goto(I, X)$ to C;
  until no new set of items can be added to C;
  return C;
end

- **Note** : The set of items construction is essentially the same as for the SLR(1) case.

$S \rightarrow L = R \mid R$
$R \rightarrow L$
$L \rightarrow * R \mid id$



	ACTION				GOTO		
	*	=	id	$	S	R	L
S0	S,S3		S,S4			S1	S2
S1				accept			
S2		S,S5		R,R→L			
S3	S,S3		S,S4			S7	S6
S4		R,L→id		R,L→id			
S5	S,S11		S,S10			S8	S9
S6		R,R→L		R,R→L			
S7		R,L→*R		R,L→*R			
S8				accept			
S9				R,R→L			
S10				R,L→id			
S11	S,S11		S,S10			S12	S9
S12				R,L→*R			

S - SHIFT R - REDUCE S# - Next State #n – Production Number

$ S0	* id = id $
$ S0 * S3	id = id $
$ S0 * S3 id S4	= id $
$ S0 * S3 L S6	= id $
$ S0 * S3 R S7	= id $
$ S0 L S2	= id $
$ S0 L S2 = S5	id $
$ S0 L S2 = S5 id S10	$
$ S0 L S2 = S5 L S9	$
$ S0 L S2 = S5 R S8	$

accept

Constructing an LR(1) Parse Table

1. Given a grammar G, construct the augmented grammar G'' by adding the production $S' \longrightarrow S$.

2. Construct $C = \{I_0, \ldots, I_n\}$, the viable prefix DFA for G''.

3. State i is constructed from I_i, with parsing action determined as follows:

   (a) $[A \longrightarrow \alpha \bullet \mathrm{a}\beta, \mathrm{b}] \in I_i$, a a terminal, $goto(I_i, \mathrm{a}) = I_j$: set $action[i, \mathrm{a}]$ = shift j.

   (b) $[A \longrightarrow \alpha \bullet, \mathrm{a}] \in I_i, A \neq S'$: set $action[i, \mathrm{a}]$ = reduce $A \longrightarrow \alpha$.

   (c) $[S' \longrightarrow S \bullet, \$] \in I_i$: set $action[i, \$]$ = accept.

4. goto transitions are constructed as follows: for each nonterminal A, if $goto(I_i, A) = I_j$ then $goto[i, A] = j$.

5. All entries not defined by the above steps are made *error*.

   If there are any multiply defined entries, then G is not LR(1).

6. Initial state of the parser: that constructed from $I_0 \sim [S' \longrightarrow \cdot S, \$]$.

LR(1) vs. SLR(1) :

- LR(1) more powerful, can handle a strictly larger class of grammars than SLR(1).
- The parse tables for LR(1) become very large — may be impractical for realistic grammars.
- A compromise between parsing power and table size that is commonly used is seen in LALR parsers.

  An LALR parser can be thought of as an LR(1) parser, some of whose states have been merged into a single state. This can be done in many (but not all) cases without causing problems.

  The parsers generated by tools such as yacc and bison are LALR.

5.4.3. LALR(1) Parsing

Observation : Every SLR grammar is an LR(1) grammar, but the LR(1) parser usually has many more states than the SLR parser.

Many of these states differ only on the lookahead token. But the lookahead token does not play any role except on reductions.

Definition : The *core* of a set of LR(1) items I is

$$core(I) = \{J \mid [J, \mathbf{a}] \in I \text{ for some } \mathbf{a}\}$$

I.e., $core(I)$ is the set of first components of I.

Example : Suppose

$$I = \{[A \longrightarrow \mathbf{c}_\bullet, \mathrm{a}], [A \longrightarrow \mathbf{c}_\bullet, \mathrm{b}], [B \longrightarrow \mathbf{c}_\bullet, \mathrm{c}]\}$$

Then,

$$core(I) = \{A \longrightarrow \mathbf{c}_\bullet, B \longrightarrow \mathbf{c}_\bullet\}$$

Merging sets of LR(1) Items

- If sets of items with the same core are merged, the parser behaves essentially as before.

  However, some redundant reductions might be done before an error is detected.

- $core(goto(I, X))$ depends only on $core(I)$, so *goto*'s of merged sets may themselves be merged.

- Suppose we take a set C_0 of sets of LR(1) items for a given grammar, and merge those sets of items that have the same core to get a set C_1 of sets of LR(1) items.

  LR(1) parse table construction using C_1 will not introduce any new shift/reduce conflicts compared to C_0.

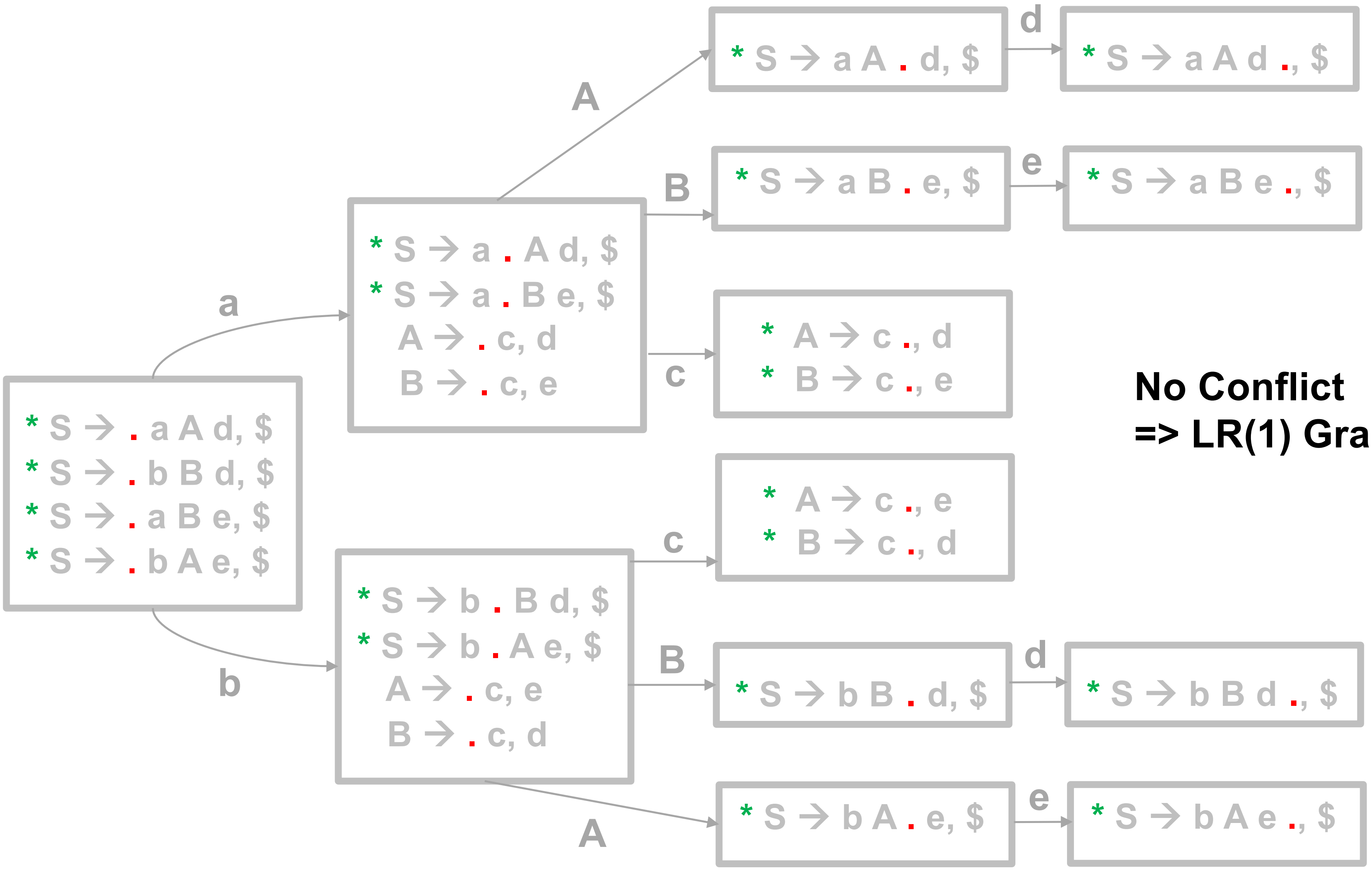
  However, this can introduce new reduce/reduce conflicts.

Example of reduce/reduce conflicts due to merging :

Consider the grammar given by

$$S \longrightarrow \mathbf{a}A\mathbf{d} \mid \mathbf{b}B\mathbf{d} \mid \mathbf{a}B\mathbf{e} \mid \mathbf{b}A\mathbf{e}$$
$$A \longrightarrow \mathbf{c}$$
$$B \longrightarrow \mathbf{c}$$

$$S \longrightarrow \mathrm{a}A\mathrm{d} \mid \mathrm{b}B\mathrm{d} \mid \mathrm{a}B\mathrm{e} \mid \mathrm{b}A\mathrm{e}$$
$$A \longrightarrow \mathrm{c}$$
$$B \longrightarrow \mathrm{c}$$



No Conflict
=> LR(1) Grammar

$$S \longrightarrow \mathrm{a}A\mathrm{d} \mid \mathrm{b}B\mathrm{d} \mid \mathrm{a}B\mathrm{e} \mid \mathrm{b}A\mathrm{e}$$
$$A \longrightarrow \mathrm{c}$$
$$B \longrightarrow \mathrm{c}$$

A → c . , d
B → c . , e

A → c . , e
B → c . , d

Merge

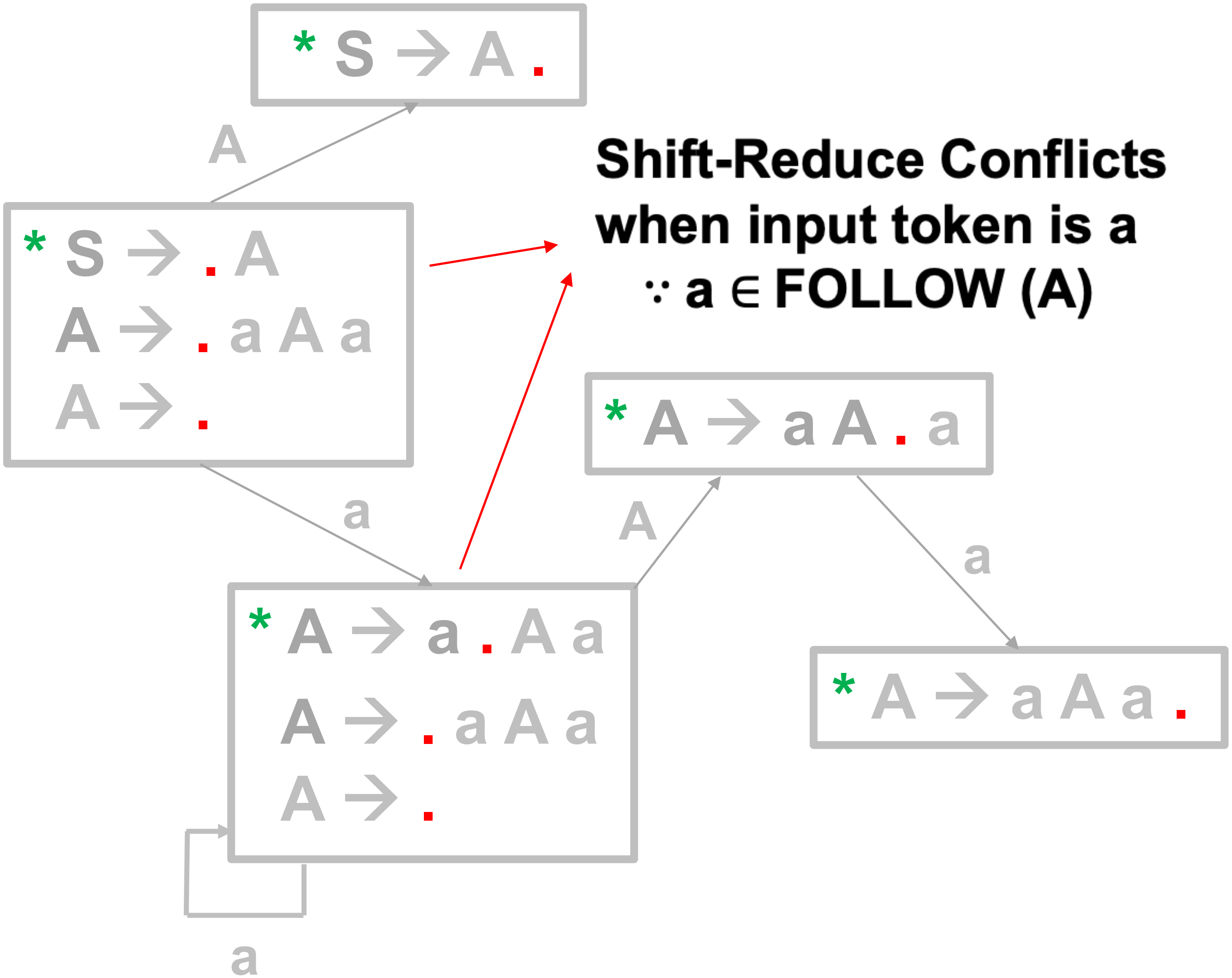
A → c . , d/e
B → c . , d/e

Contains reduce-reduce conflict

→ not LALR(1)

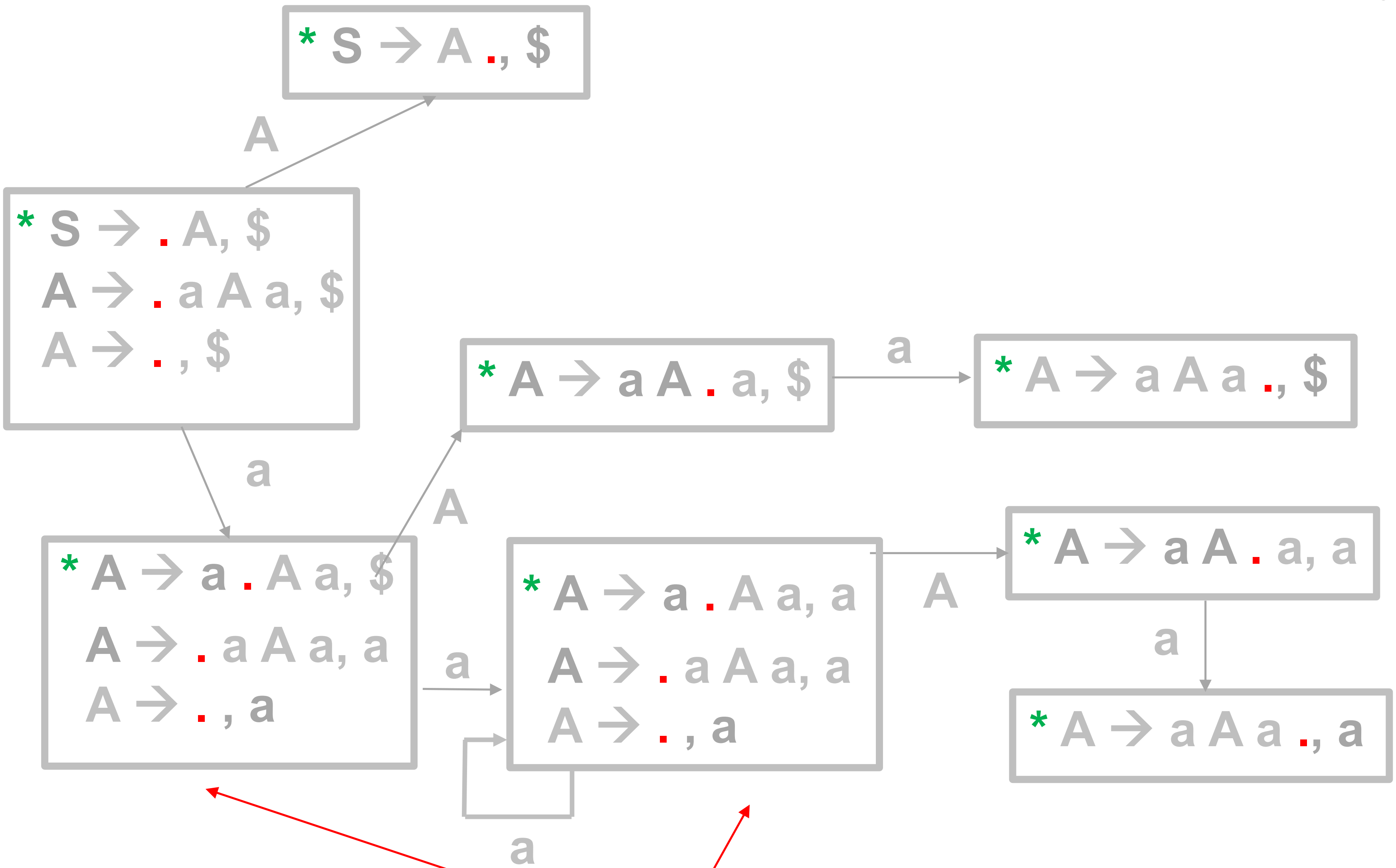
SAMPLE PROBLEMS

S → A
A → a A a | ϵ



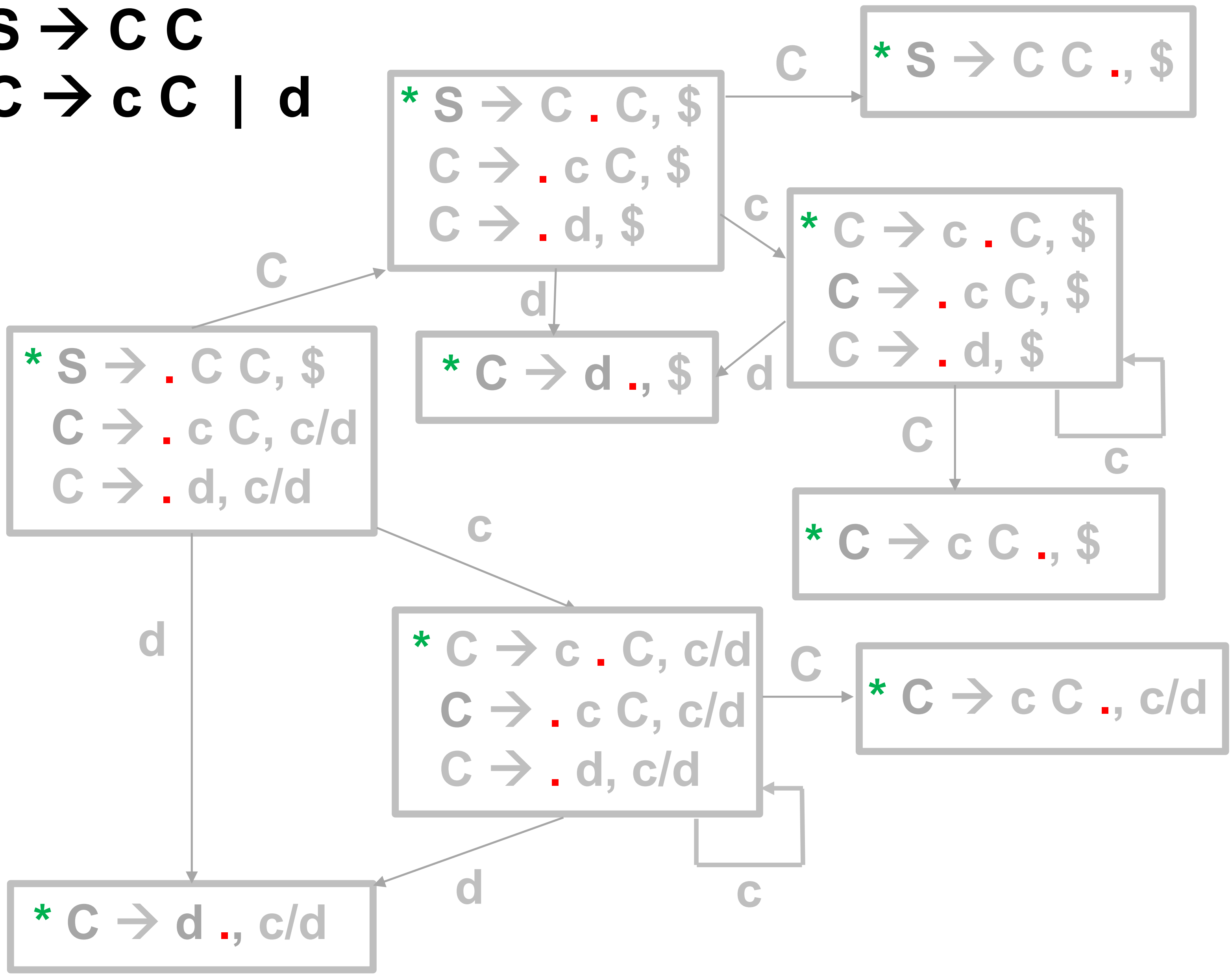
Shift-Reduce Conflicts when input token is a
∵ a ∈ FOLLOW (A)

S → A
A → a A a | ϵ

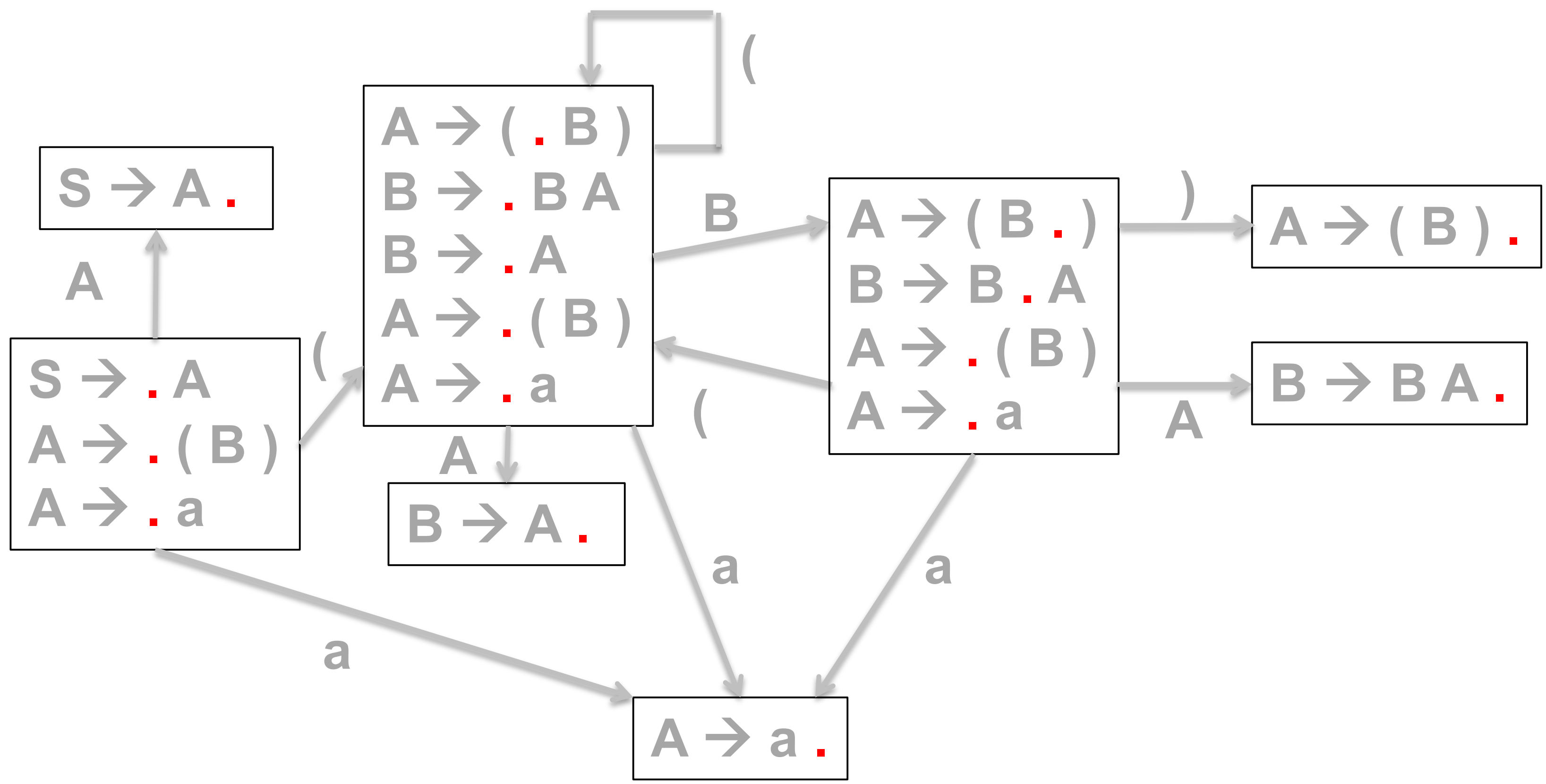


Shift-Reduce Conflicts when input token is a

S → C C
C → c C | d

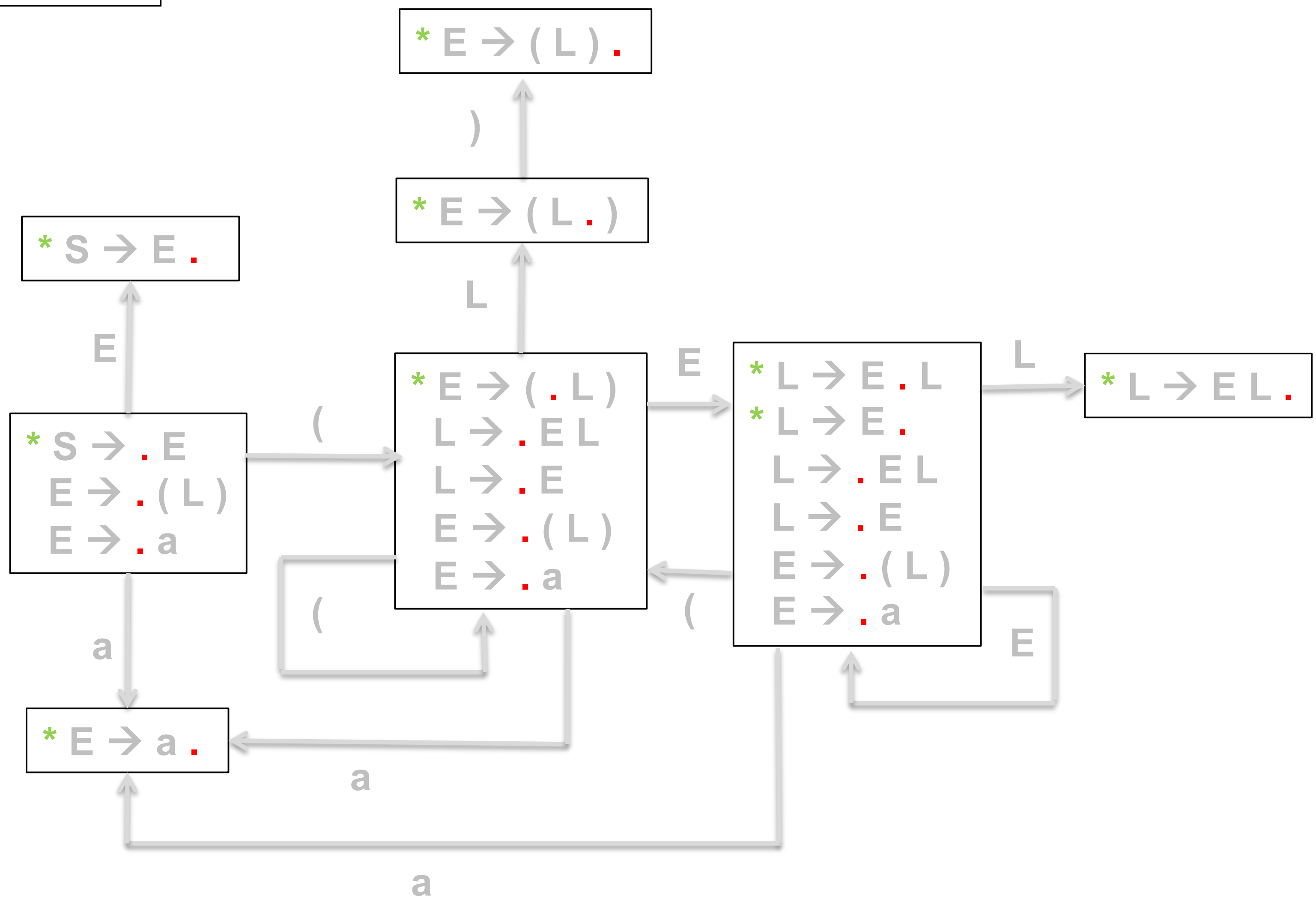

* S → C . C, $
C → . c C, $
C → . d, $
C
* S → C C ., $
c
* C → c . C, $
C → . c C, $
C → . d, $
C
d
* S → . C C, $
C → . c C, c/d
C → . d, c/d
* C → d ., $
d
C
c
* C → c C ., $
c
d
* C → c . C, c/d
C → . c C, c/d
C → . d, c/d
C
* C → c C ., c/d
d
c
* C → d ., c/d

S → A
A → (B) | a
B → B A | A

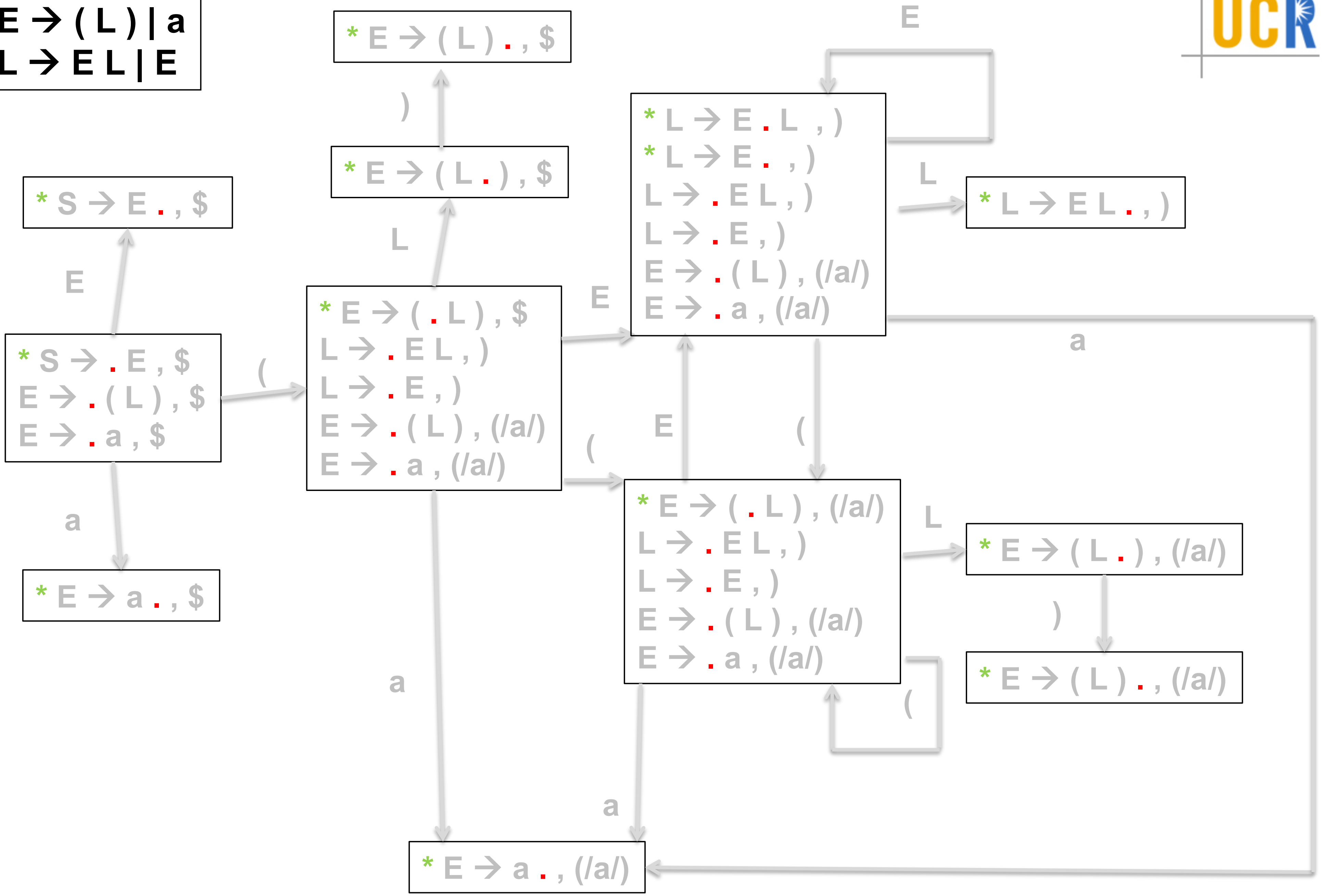


No Conflicts!

S → E
E → (L) | a
L → E L | E


* E → (L) .
)
* E → (L .)
L
* S → E .
E
* E → (. L)
L → . E L
L → . E
E → . (L)
E → . a
E
* L → E . L
* L → E .
L → . E L
L → . E
E → . (L)
E → . a
L
* L → E L .
* S → . E
E → . (L)
E → . a
(
(
(
E
a
* E → a .
a
a

S → E
E → (L) | a
L → E L | E


* S → E . , $
* S → . E , $
E → . (L) , $
E → . a , $
* E → a . , $
* E → (. L) , $
L → . E L ,)
L → . E ,)
E → . (L) , (/a/)
E → . a , (/a/)
* E → (L .) , $
* E → (L) . , $
* L → E . L ,)
* L → E . ,)
L → . E L ,)
L → . E ,)
E → . (L) , (/a/)
E → . a , (/a/)
* L → E L . ,)
* E → (. L) , (/a/)
L → . E L ,)
L → . E ,)
E → . (L) , (/a/)
E → . a , (/a/)
* E → (L .) , (/a/)
* E → (L) . , (/a/)
* E → a . , (/a/)
E
(
a
L
)
E
L
E
(
a
E
(
L
)
(
a
a