Compilers

Topic: Bottom-Up Parsing - LR(1) Parsers

Monsoon 2011, IIIT-H, Suresh Purini

ACK: Some slides are based on Keith Cooper's CS412 at Rice University

LR(1) Grammars

LR(1) – Reading the input string from left to right.

LR(1) – Deriving a right-most derivation for the string.

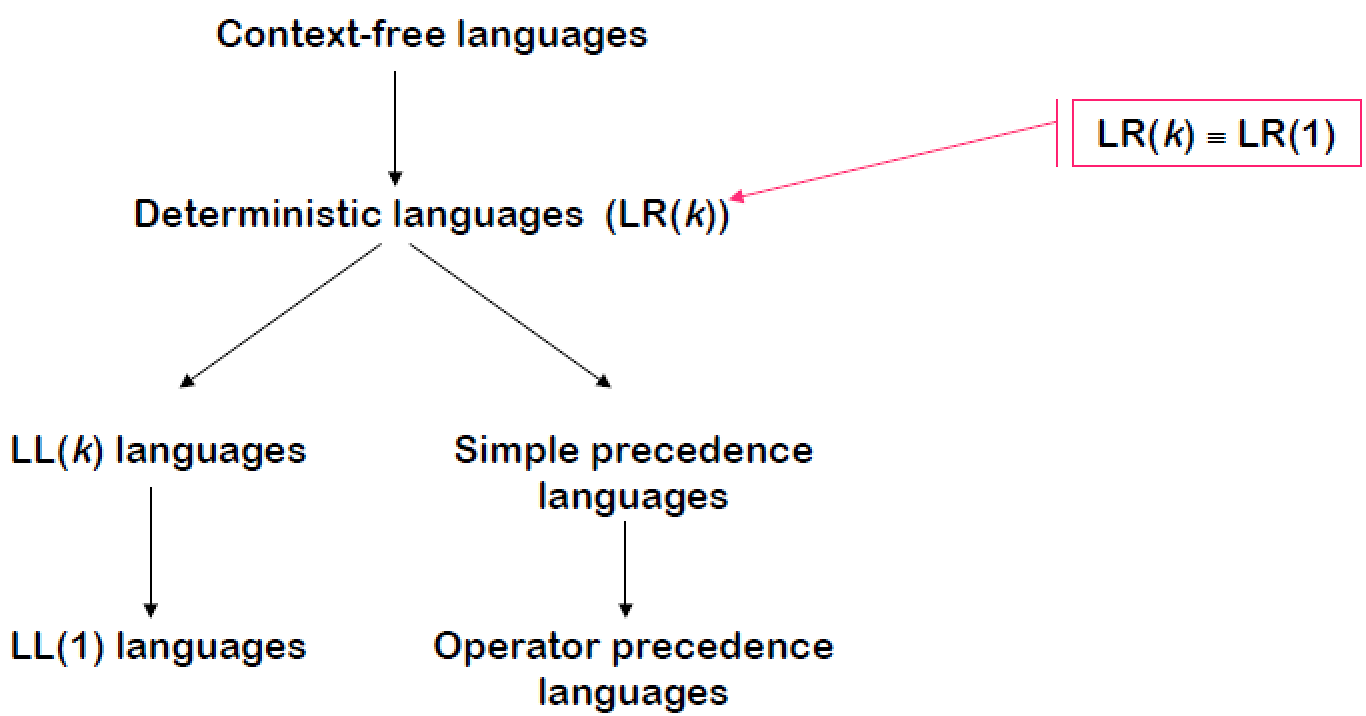
LR(1) – one token look-ahead

- Intuitively: For a grammar to be LR(1) it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential form when they appear on the stack.
- Remark: LR(k) and LR(1) grammars have the same expressive power. However, we could probably write a simpler LR(k) grammar for a language than an LR(1)

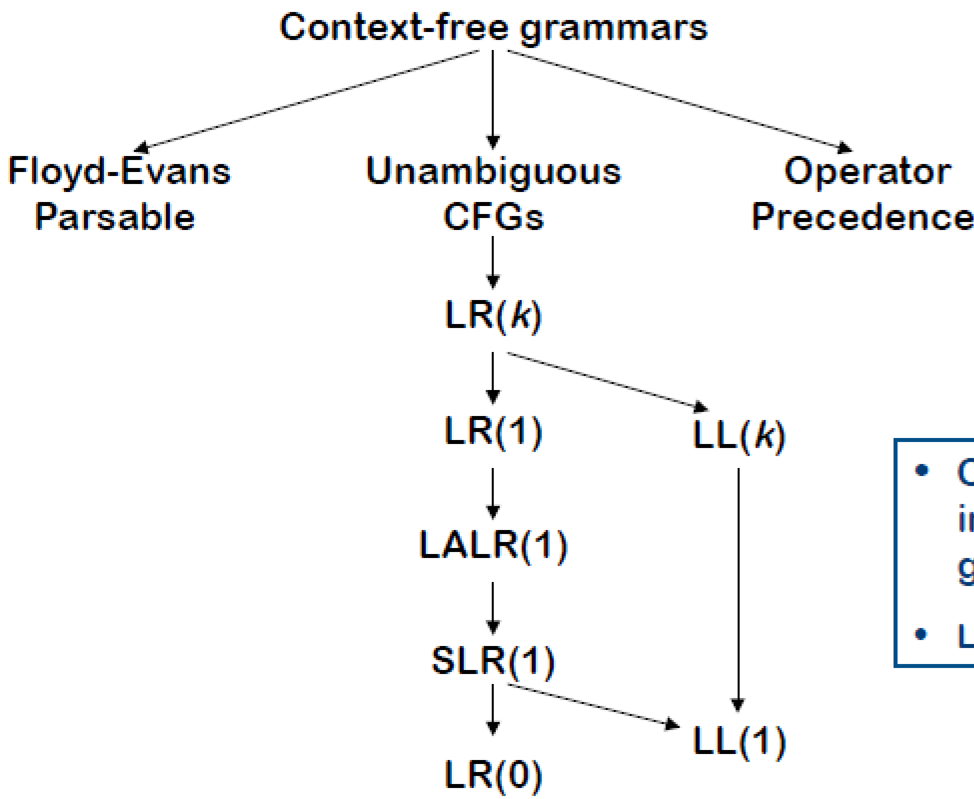
LR(1) Grammars

- Almost all programming language constructs are LR(1) Parsable
- LR-parsing method is the most general non-backtracking shift-reduce parsing method known.
- LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input
- LR(1) Grammars are strictly more powerful than LL(1)

Hierarchy of Context-Free Languages

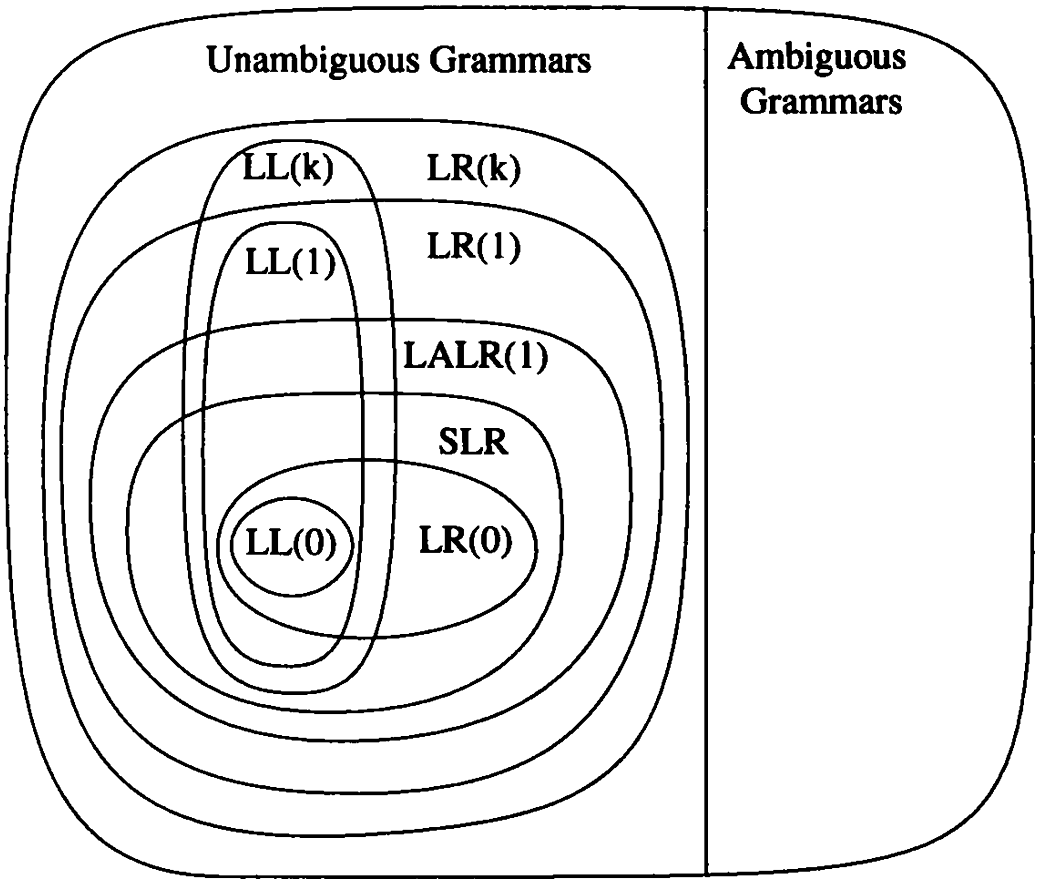


Hierarchy of Context-Free Languages



- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)

Hierarchy of Context-Free Languages



LR(k) versus LL(k)

Finding Reductions

- LR(k) – Each reduction in the parse is detectable with
  - the complete left context,
  - the reducible phrase, itself, and
  - the k terminal symbols to its right
- LL(k) – Parser must select the reduction based on
  - The complete left context
  - The next k terminals
  - Thus, LR(k) examines more context

LR(1) versus LL(1)

The following LR(1) grammar has no LL(1) counterpart

0	*Goal*	$\rightarrow$	*S*
1	*S*	$\rightarrow$	*A*
2		|	*B*
3	*A*	$\rightarrow$	(*A*)
4		|	<u>a</u>
5	*B*	$\rightarrow$	(*B* >
6		|	<u>b</u>

- It requires an arbitrary lookahead to choose between *A* & *B*
- An LR(1) parser can carry the left context (the '(' s) until it sees <u>a</u> or <u>b</u>
- The table construction will handle it
- In contrast, an LL(1) parser cannot decide whether to expand *Goal* by *A* or *B*
  - → *No amount of massaging the grammar will resolve this problem*

LR(1) Item

Def: An LR(1) item is of the form [A⟶α.β, a], where A⟶αβ is a production and a is either a terminal symbol or $.

Grammar	Example LR(1) Items
S → L = R | R L → * R | **id** R → L	[S → . L = R , id] [S → L . = R , id] [S → L = . R , id] [S → L = R . , id] [L → . * R , $] [L → * . R, $] [L → * R ., $]

LR(1) Sets of Items

Def: An LR(1) Set of Items I is a collection of LR(1) Items (nothing fancy here, just the usual set definition).

Examples:

1. $I_1 = \{\ [S \rightarrow .\ L = R\ ,\ id],\ \ [S \rightarrow L = .\ R\ ,\ id]\ \}$
2. $I_2 = \{\ [S \rightarrow L = .\ R\ ,\ id],\ [S \rightarrow L = R\ .\ ,\ id],\ [L \rightarrow *\ R\ .,\ \$]\ \}$

$S \rightarrow L = R \mid R$

$L \rightarrow *\ R \mid \mathbf{id}$

$R \rightarrow L$

Closure of LR(1) Items

Def: If I is the set of LR(1) items for a grammar G, then CLOSURE(I) consists of

- Every item of I
- If $[A \rightarrow \alpha.B\beta, a]$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then for each terminal b of FIRST(βa), $[B \rightarrow .\gamma, b]$ is also present in CLOSURE(I).

Compute

1) CLOSURE({ $[S \rightarrow L . = R, \$]$ })
2) CLOSURE({ $[L \rightarrow \ * . R, =]$, $[L \rightarrow \ * . R, id]$ })
3) CLOSURE({ $[S \rightarrow . L \ = R, \$]$ })

$S \rightarrow L = R \mid R$

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

$R \rightarrow L$

Algorithm for Computing CLOSURE(I)

SetOfItems $\text{CLOSURE}(I)$ {
  repeat
    for (each item $[A \rightarrow \alpha{\cdot}B\beta, a]$ in I)
      for (each production $B \rightarrow \gamma$ in G')
        for (each terminal b in $\text{FIRST}(\beta a)$)
          add $[B \rightarrow {\cdot}\gamma, b]$ to set I;
  until no more items are added to I;
  return I;
}

Computing GOTO(I,X)

- I is an LR(1) Set of Items and X is either a terminal, or a non-terminal, or \$ symbol i.e., (X ∈ N ∪ T ∪ { \$ }).

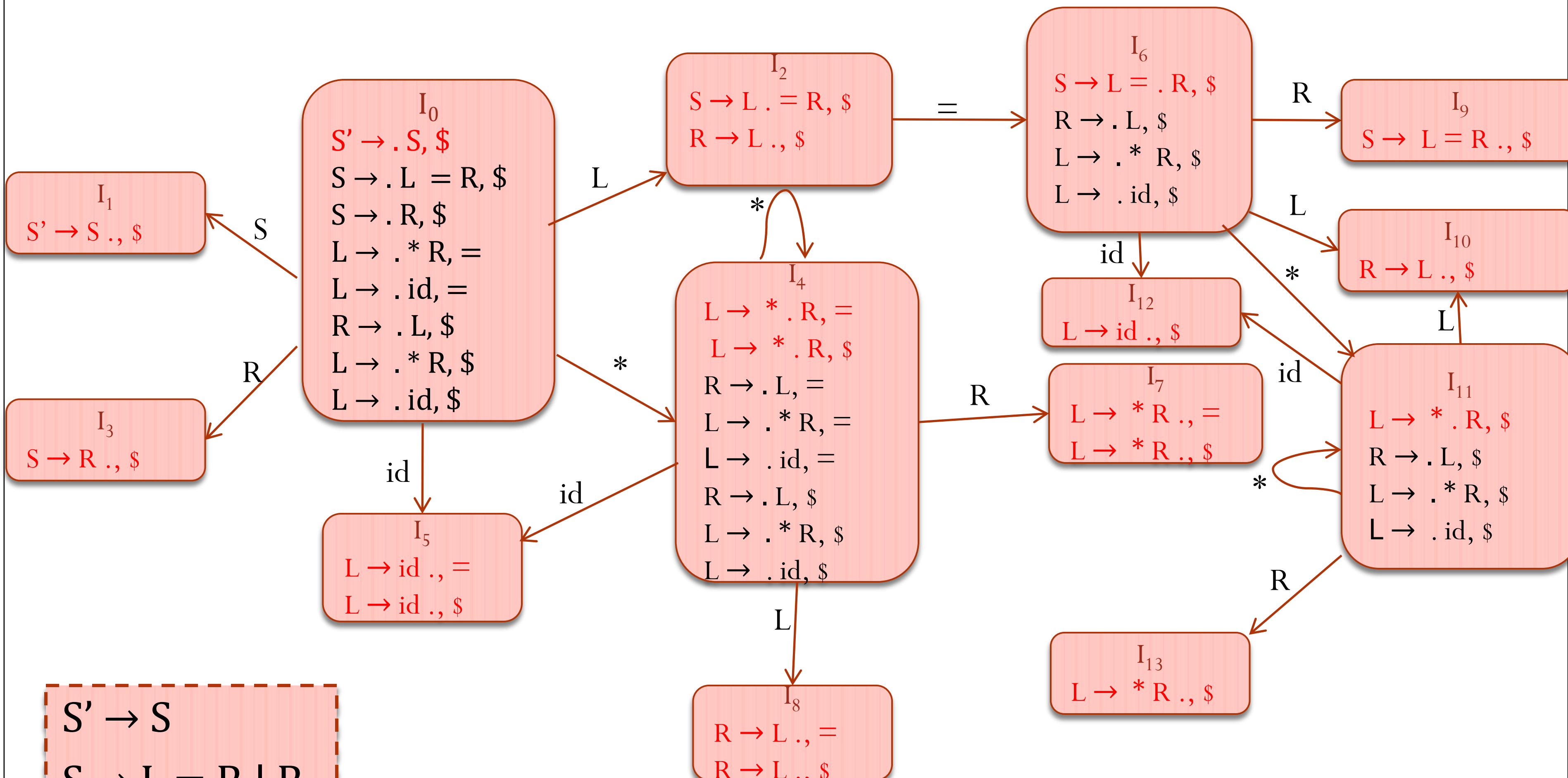
```
SetOfItems GOTO(I, X) {
        initialize J to be the empty set;
        for ( each item [A → α·Xβ, a] in I )
                add item [A → αX·β, a] to set J;
        return CLOSURE(J);
}
```

Computing LR(1) Sets of Items in an LR(1) Automaton

```
void items(G') {
    initialize C to CLOSURE({[S' → ·S, $]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

Note: We have to augment the grammar with the production S' ⟶ S where S' is the new start symbol.

LR(1) Automaton



S' → S

S → L = R | R

L → * R | **id**

R → L

Note: LR(1) items marked in Red in each set are called kernel items. Non-kernels items can be derived from kernel items by applying a closure operation.

LR(1) Parsing Table Contruction

Rules for constructing the Action Part of the LR(1) Parsing Table

1. If state i contains [S' → S ., \$], set ACTION[i, \$] to acc.
2. If state i contains [A →α.aβ, b] and GOTO(i, a) = j,

   then set ACTION[i, a] = shift j.
3. If state i contains [A →α., a],

   then set ACTION[i, a] to "reduce A →α".

Rules for constructing the GOTO Part of the LR(1) Parsing Table

1. For state i and non-terminal A if GOTO(i, A) = j, set GOTO(i, A) = j.
2. All the blank entries at the end are made "error".
3. The state corresponding to the item [S' → . S] is the initial state.

LR(1) Parsing Table Construction

	Action Part				GOTO Part		
	*	=	id	$	S	L	R
0	S4		S5		1	2	3
1				acc			
2		S6		R6	4	5	
3				R3			
4	S4					8	7
5		R5		R5			
6	S11		S12			10	9
7		R4		R4	8		
8		R6		R6			
9				R2			
10				R6			
11						10	13
12				R5			
13				R4			

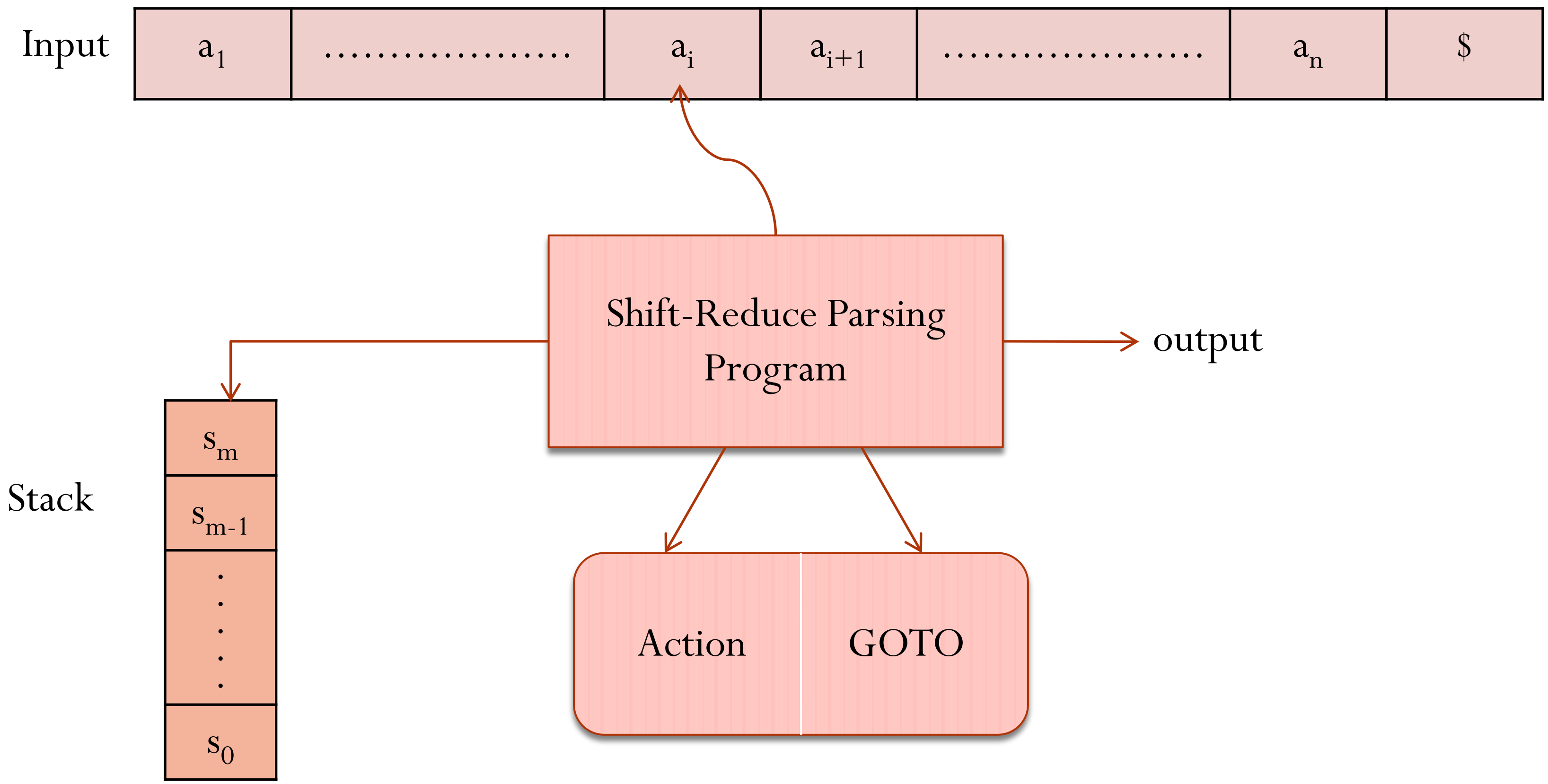
1. $S' \rightarrow S$
2. $S \rightarrow L = R$
3. $S \rightarrow R$
4. $L \rightarrow * R$
5. $L \rightarrow$ **id**
6. $R \rightarrow L$

LR(1) Parsing Example

Also try the algorithm on the string * id = id.

Stack	Input	Action
0	id = * id $	Shift-5 (Shift State 5, equivalently teminal id)
0 id 5	= * id $	Reduce-5 (Reduce using Rule-5, L → id). Pop id. Push L which is equivalent to pushing GOTO(0, L)
0 L 2	= * id $	Shift-6 (Shift =)
0 L 2 = 6	* id $	Shift-11 (Shift *)
0 L 2 = 6 * 11	id $	Shift-12 (Shift id)
0 L 2 = 6 * 11 id 12	$	Reduce-5 (Reduce using Rule 5, L → id). Pop id. Push L which is equivalent to pushing GOTO(11, L)
0 L 2 = 6 * 11 L 10	$	Reduce-6 (Reduce using Rule 6, R → L). Pop L. Push R which is equivalent to pushing GOTO(11, R)
0 L 2 = 6 * 11 R 13	$	Reduce-5 (Reduce using Rule 5, L → * R). Pop *R. Push L which is equivalent to pushing GOTO(6, L)
0 L 2 = 6 L 10	$	Reduce-6 (Reduce using Rule 6, R → L). Pop L. Push R which is equivalent to pushing GOTO(6, R)
0 L 2 = 6 R 9	$	Reduce-2 (Reduce using Rule, S → L = R Pop L = R. Push S which is equivalent to pushing GOTO(0, S)
0 S 1	$	Accept

LR Parsing Algorithm



- LR Parser Configuration: $(s_0 s_1 \ldots s_m,\ a_i a_{i+1} \ldots a_n \$)$
- This configuration represents the right-sentential form $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$ where X_i is the transition label from the state s_i-1 to s_i.

Behavior of the Parser

Current Configuration: $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n \$)$

1. If ACTION$[s_m, a_i]$ = shift s, then the next configuration is

$$(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n \$)$$

Remark: This is equivalent to shifting the terminal symbol a_i as the transition arc (s_m, s) is labeled a_i.

2. If ACTION$[s_m, a_i]$ = Reduce $A \rightarrow \beta$, then
   a) Pop $r = |\beta|$ states from the stack and enter the intermediate configuration $(s_0 s_1 \ldots s_{m-r}, a_i a_{i+1} \ldots a_n \$)$.
   b) Shift state s onto the stack where GOTO$(s_{r-m}, A) = s$

   Final Configuration: $(s_0 s_1 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n \$)$.

3. If ACTION$[s_m, a_i]$ = accept, accept the input string.
4. If ACTION$[s_m, a_i]$ = error, declare error and call an error recovery routine.

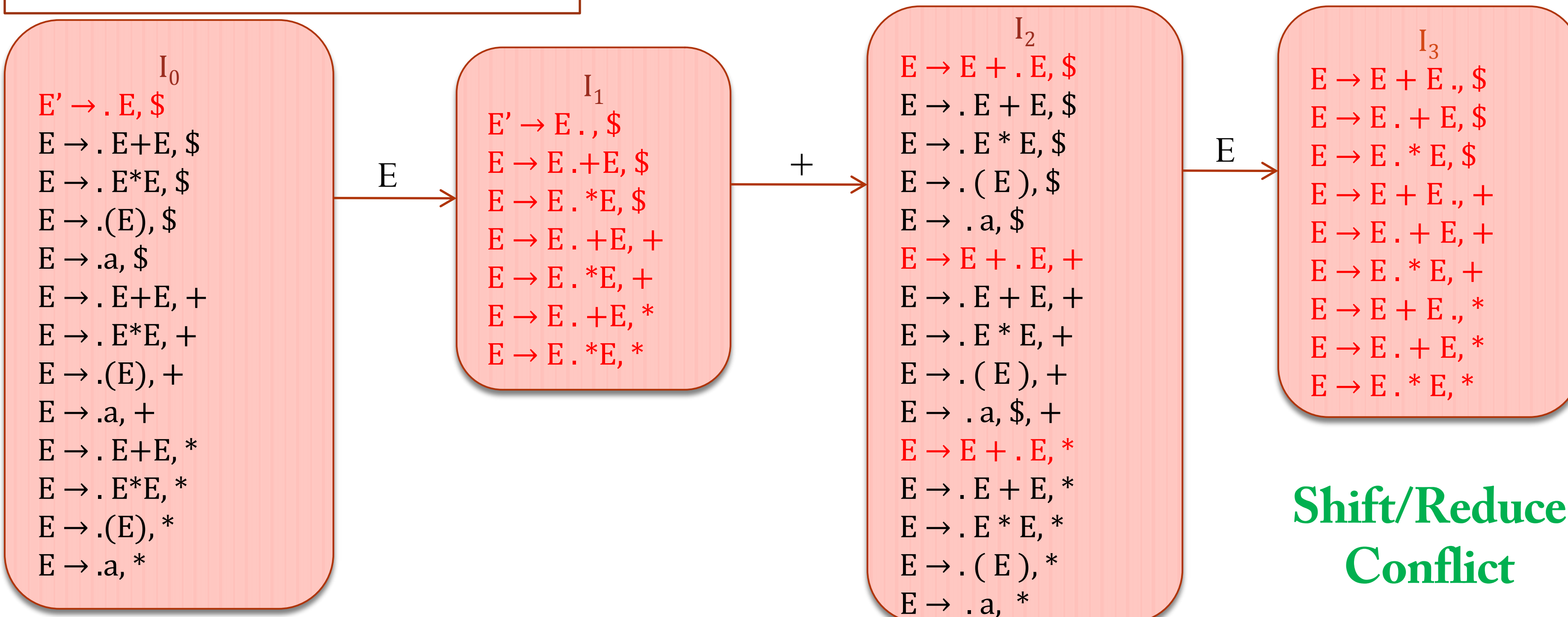
LR Parsing Algorithm

```
Let a be the first symbol of w$
while(1) { /* repeat forever */
    Let s be the state on top of the stack;
    if ( ACTION[s, a] = shift t ) {
        a)  push t onto the stack;
        b)  let a be the next input symbol
    }
    else {
        if ( ACTION[s, a] = reduce A → β ) {
            a)  pop | β |symbols off the stack;
            b)  let state t now be on top of the stack;
            c)  push GOTO[t, A] onto the stack;
        }
        else {
            if ( ACTION[s, a] = accept ){ print ACCEPT; break; }
            else{ call error-recovery routine; }
        }
    }
}
```

Handling Precedence and Associativity Rules

$E \rightarrow E + E \mid E * E \mid (E) \mid a$

Partial LR(1) Automaton



Shift/Reduce Conflict

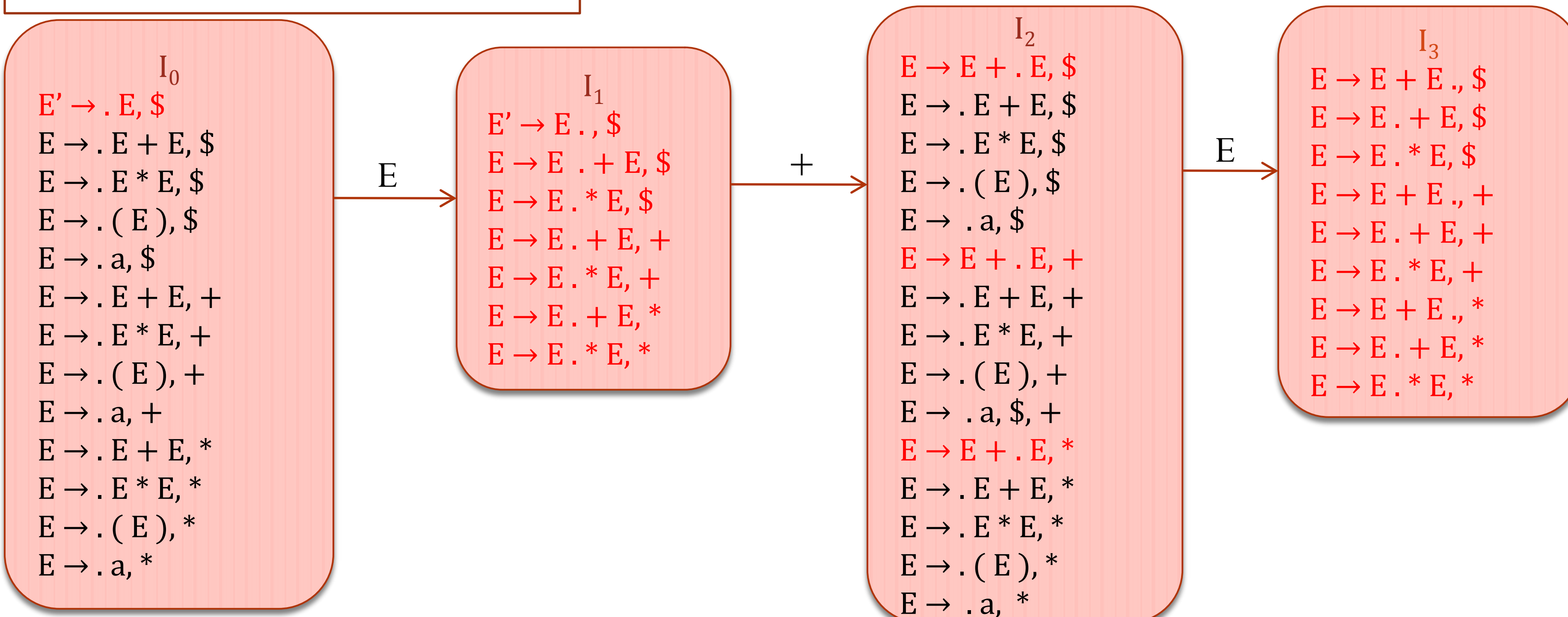
Q: If the state on the top of the parser stack is 3 and the next input symbol is +, should the parser perform a shift action or a reduce action?

- Perform reduce action if we want to make + left-associativity operation.
- Perform shift action if we want to make + right-associative.

Handling Precedence and Associativity Rules

E → E + E | E * E | (E) | a

Partial LR(1) Automaton



Q: If the state on the top of the parser stack is 3 and the next input symbol is *, should the parser perform a shift action or a reduce action?

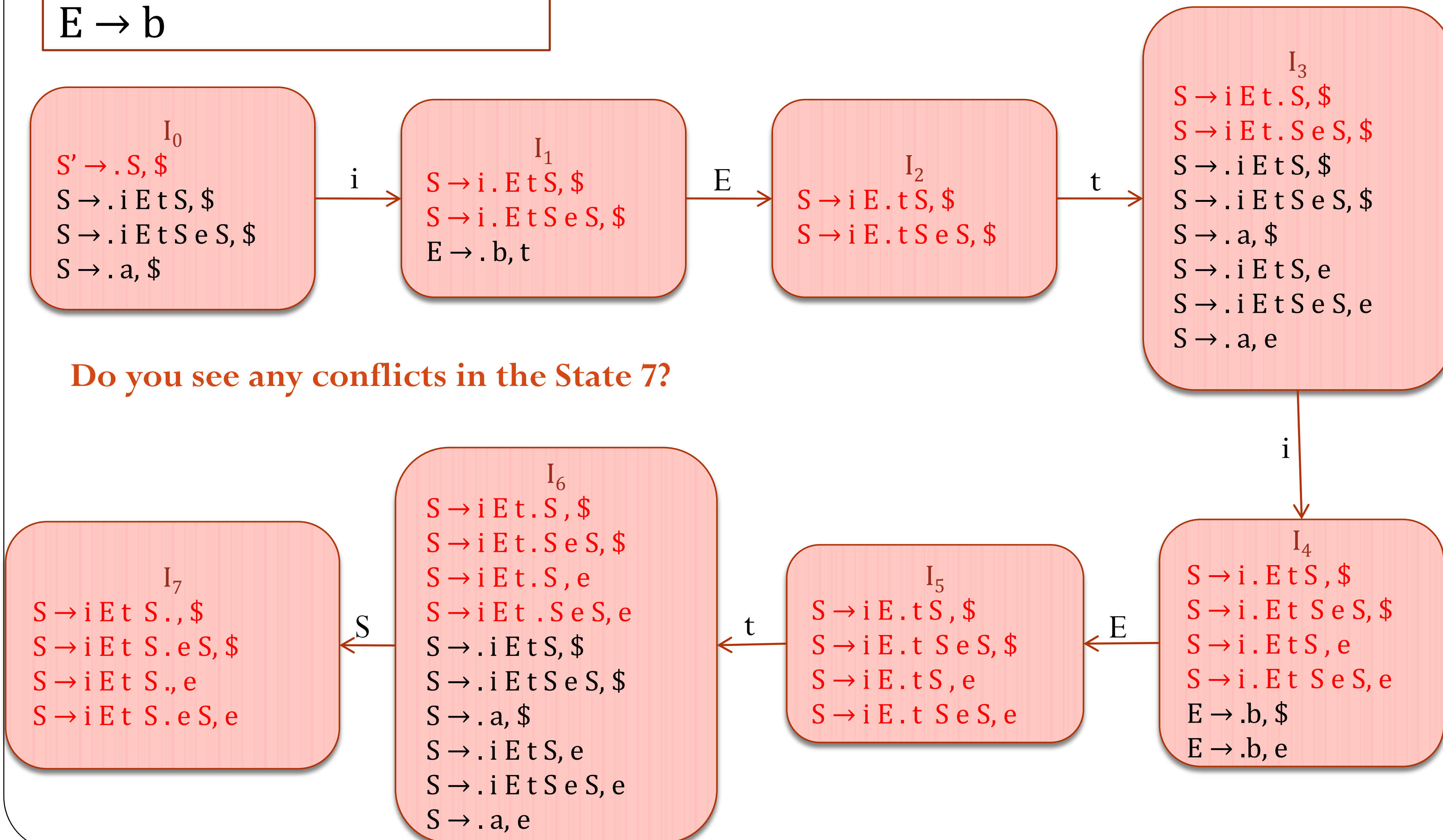
- Perform reduce action if we want to make Prec(+) > Prec(*)
- Perform shift action if we want to make Prec(+) < Prec(*)

Handling if-then-else ambiguity

$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$

$E \rightarrow b$

Partial LR(1) Automaton



Do you see any conflicts in the State 7?

LALR(1) Grammars

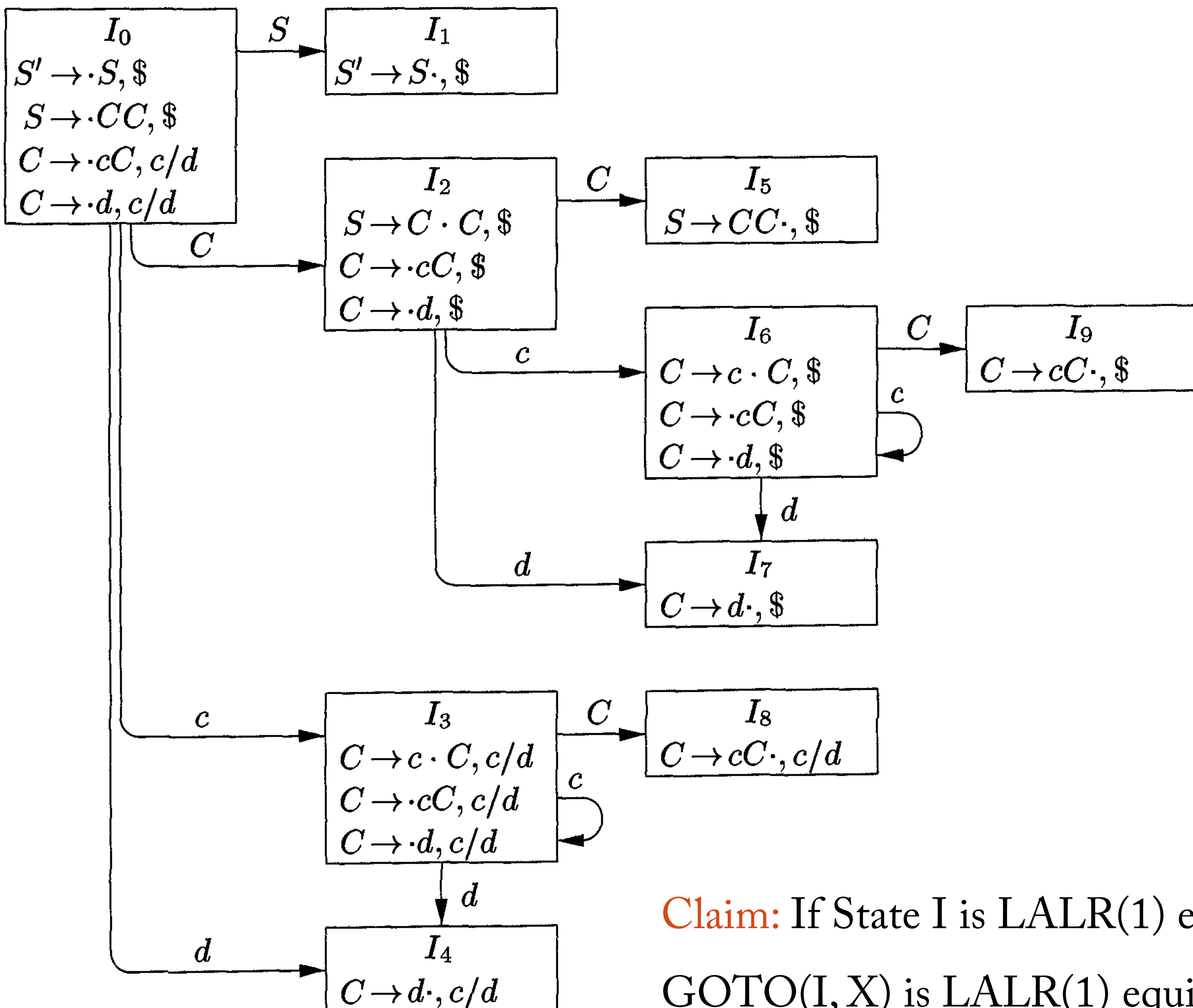
LALR(1) Grammars

- LR(1) automaton for a C-like programming language could have thousands of states.
  - Huge parsing table.
- LALR(1) Approach: If two states have kernels whose first elements are same merge them.

$$\boxed{\begin{array}{l}[\,S \rightarrow id, +\,] \\ [\,S \rightarrow E., \$\,]\end{array}} + \boxed{\begin{array}{l}[\,S \rightarrow id, \$\,] \\ [\,S \rightarrow E., +\,]\end{array}} = \boxed{\begin{array}{l}[\,S \rightarrow id, +\,] \\ [\,S \rightarrow id, \$\,] \\ [\,S \rightarrow E., +\,] \\ [\,S \rightarrow E., \$\,]\end{array}}$$

- LALR(1) Table Compaction Approach can reduce the number of LR(1) states from thousands to hundreds.

LALR(1) Grammars



$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & C\ C \\
C & \rightarrow & c\ C \mid d
\end{array}
$$

Merge

1. **States 4 and 7.**
2. **States 3 and 6.**
3. **States 8 and 9.**

Claim: If State I is LALR(1) equivalent Sate J, then GOTO(I, X) is LALR(1) equivalent to GOTO(J, X).

LALR(1) Grammars

LR(1) Parsing Table

STATE	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LALR(1) Grammars

LALR(1) Parsing Table

STATE	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

LALR(1) Grammars

- A Grammar is said to be LALR(1) if after merging states in LR(1) automaton no conflicts arise.
- When a LALR(1) automaton is constructed from LR(1) automaton, we can only expect to see Reduce-Reduce conflicts but not Shift-Reduce conflicts. Why?

$$\begin{array}{lcl} S' & \to & S \\ S & \to & a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e \\ A & \to & c \\ B & \to & c \end{array}$$

$$\{[A \to c\cdot,\ d],\ [B \to c\cdot,\ e]\}$$

$$\{[A \to c\cdot, e],\ [B \to c\cdot,\ d]\}$$

- Final Remark: In Practice LALR(1) suffice for our purposes.