Bottom-up Parsing

Kirill Yukhin, Intel Lab, 14.11.2018

Top-down versus Bottom-up Parsing

- Top down:
    - Recursive descent parsing
    - LL(k) parsing
  - Top to down and leftmost derivation
    - Expanding from starting symbol (top) to gradually derive the input string
  - Can use a parsing table to decide which production to use next
  - The power is limited
    - Many grammars are not LL(k)
    - Left recursion elimination and left factoring can help make some grammars LL(k), but after rewriting, the grammar can be very hard to comprehend
  - Space efficient
  - Easy to build the parse tree

Top-down versus Bottom-up Parsing

- Bottom up:
  - Also known as shift-reduce parsing
    - LR family
    - Precedence parsing
  - Shift: allow shifting input characters to the stack, waiting till a matching production can be determined
  - Reduce: once a matching production is determined, reduce
  - Follow the rightmost derivation, in a reversed way
    - Parse from bottom (the leaves of the parse tree) and work up to the starting symbol
  - Due to the added "shift"

    $\Rightarrow$ More powerful

    - Can handle left recursive grammars and grammars with left factors

    $\Rightarrow$ Less space efficient

Basic Concepts

- How to build a predictive bottom-up parser?

- Sentential form
  - For a grammar G with start symbol S

    A string α is a sentential form of G if $S \Rightarrow^* \alpha$
    - α may contain terminals and nonterminals
    - If α is in T^*, then α is a sentence of L(G)
  - Left sentential form: A sentential form that occurs in the leftmost derivation of some sentence
  - Right sentential form: A sentential form that occurs in the rightmost derivation of some sentence

Basic Concepts

- Example of the sentential form
  - $E \to E * E \mid E + E \mid (E) \mid id$
  - Leftmost derivation:

    $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow$
    $id * id + E * E \Rightarrow id * id + id * E \Rightarrow id * id + id * id$

    - All the derived strings are of the left sentential form
  - Rightmost derivation

    $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow$
    $E * E + id * id \Rightarrow E * id + id * id \Rightarrow id * id + id * id$

    - All the derived strings are of the right sentential form
- Another example
  - $S \to AB$, $A \to CD$, $B \to EF$
  - $S \Rightarrow AB \Rightarrow CDB$
  - $S \Rightarrow AB \Rightarrow AEF$

Basic Concepts

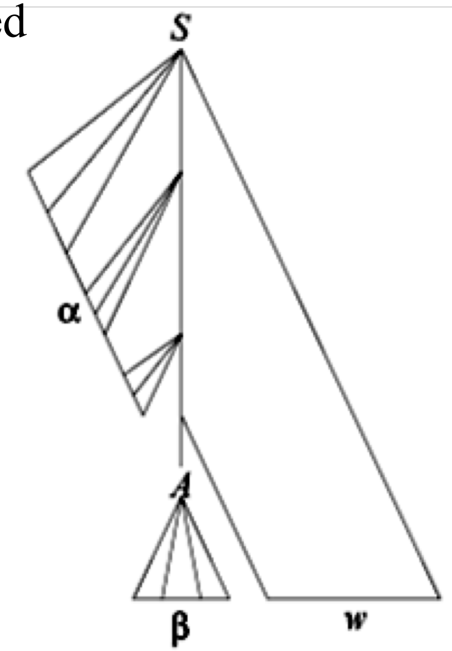
❖ Handle

- Given a rightmost derivation

  $S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_k\ (\alpha A w) \Rightarrow \gamma_{k+1}\ (\alpha\beta w) \Rightarrow \dots \Rightarrow \gamma_n$

  - γ_i, for all i, are the right sentential forms
  - From γ_k to γ_{k+1}, production $A \rightarrow \beta$ is used

- A handle of γ_{k+1} (= $\alpha\beta w$) is
  - the production $A \rightarrow \beta$ and the position of β in γ_{k+1}
  - Informally, β is the handle



The handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

Basic Concepts

- Theorem
  - If G is unambiguous, then every right-sentential form has a unique handle
- Proof
  - G is unambiguous
    - $\Rightarrow$ rightmost derivation is unique
  - Consider a right-sentential form γ_{k+1}
    - $\Rightarrow$ A unique production $A \rightarrow \beta$ is applied to γ_k, and applied at a unique position
    - $\Rightarrow$ A unique handle in γ_{k+1}
- But
  - During the derivation, the production rule is unique
  - During the reduction, can we uniquely determine the production that was used during the derivation?

Basic Concepts

- Viable prefix
  - Prefix of a right-sentential form, do not pass the end of the handle
  - E.g., αβ
    - Or the prefix of αβ
- Example: $E \rightarrow E * E \mid E + E \mid (E) \mid id$

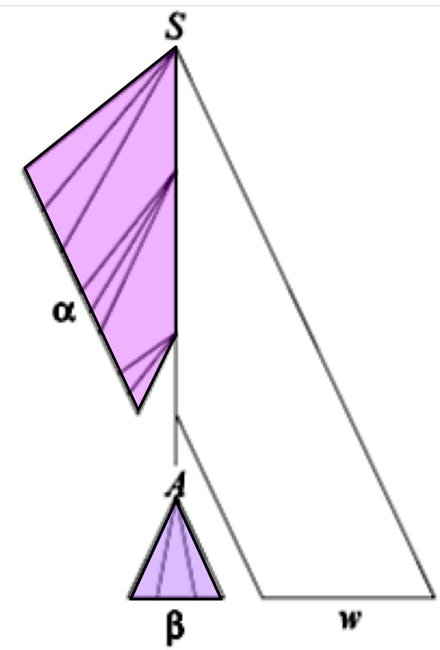
E
⇒ E + E
⇒ E + E * E
⇒ E + E * id
⇒ E + id * id
⇒ E * E + id * id
⇒ E * id + id * id
⇒ id * id + id * id

α β w

Parsing (reduction)

Handles

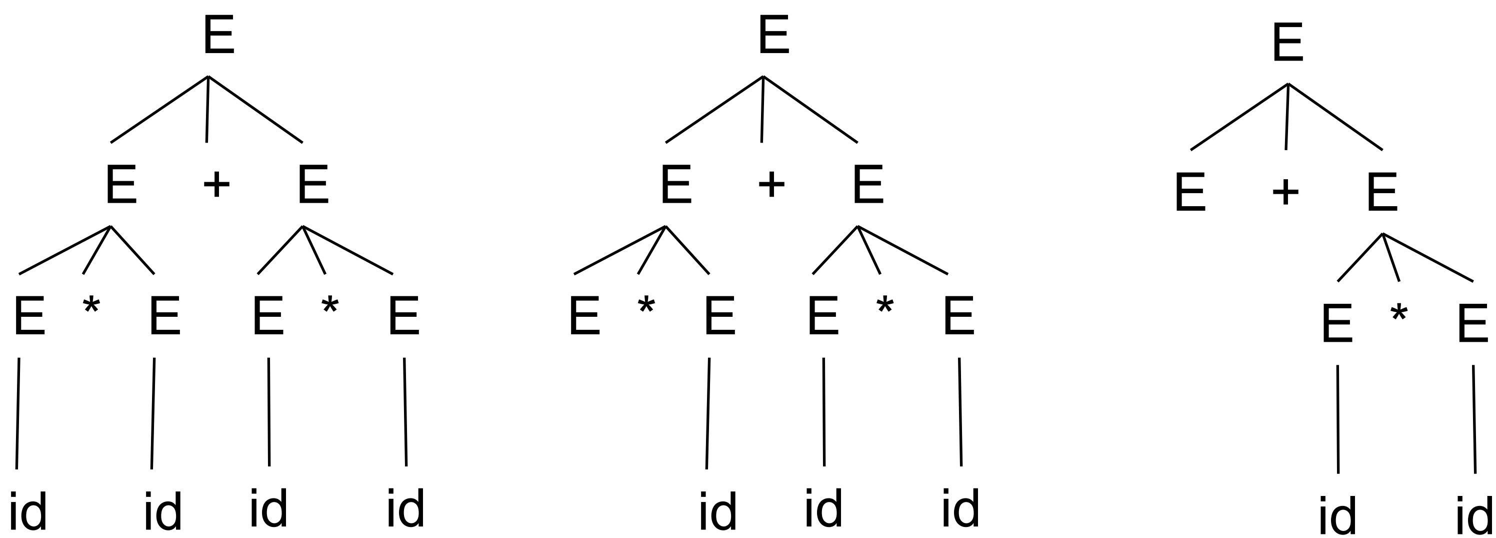
Viable prefix



The handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

Meaning of LR

- L: Process input from left to right
- R: Use rightmost derivation, but in reversed order
- $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id$
  $\Rightarrow E * E + id * id \Rightarrow E * id + id * id \Rightarrow id * id + id * id$

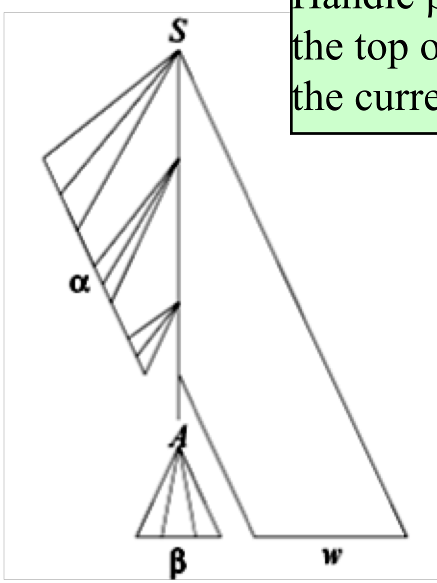


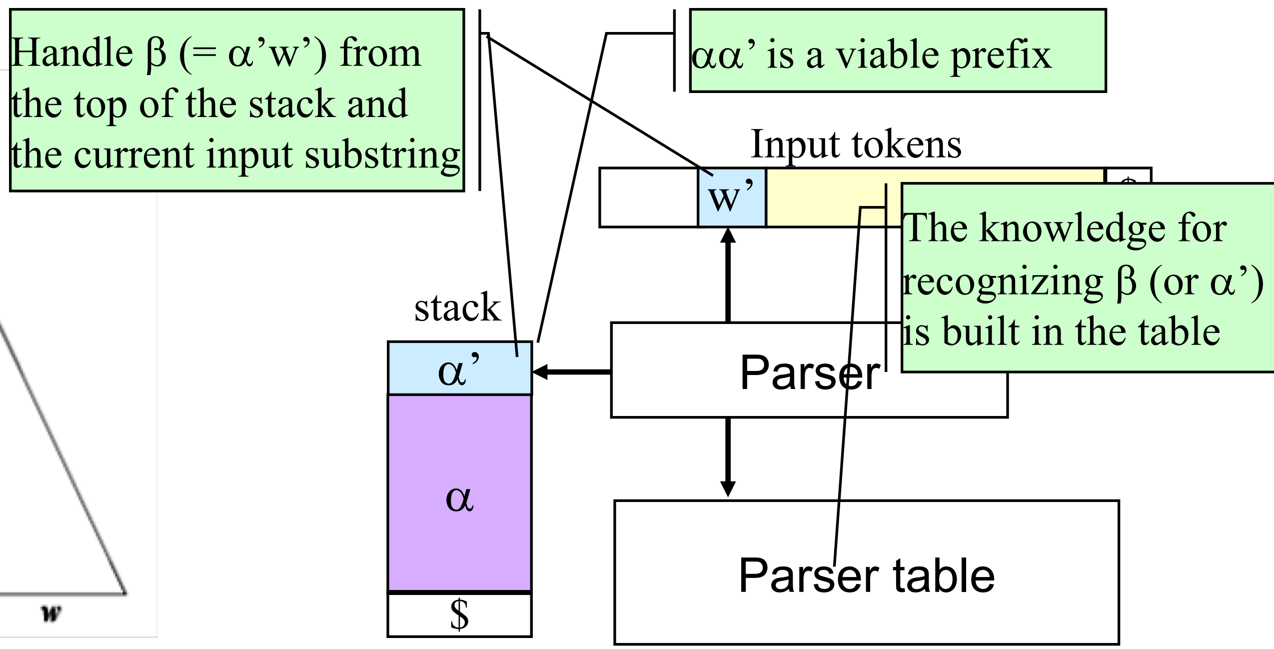
Bottom-up Parsing

- Traverse rightmost derivation backwards
  - If reduction is done arbitrarily
    - It may not reduce to the starting symbol
    - Need backtracking
  - By follow the path of rightmost derivation
    - All the reductions are guaranteed to be “correct”
    - Guaranteed to lead to the starting symbol without backtracking
  - That is: If it is always possible to correctly find the handle
- How to find the handle for reduction for each right sentential form
  - Use a stack to keep track of the viable prefix
  - The prefix of the handle will always be at the top of the stack

Bottom-up Parsing

- Consider a right-sentential form αβw
  - Where A → β and β is a handle (let β = α’w’)
  - Right to β is always a subsentence (T*)





Bottom-up Parsing

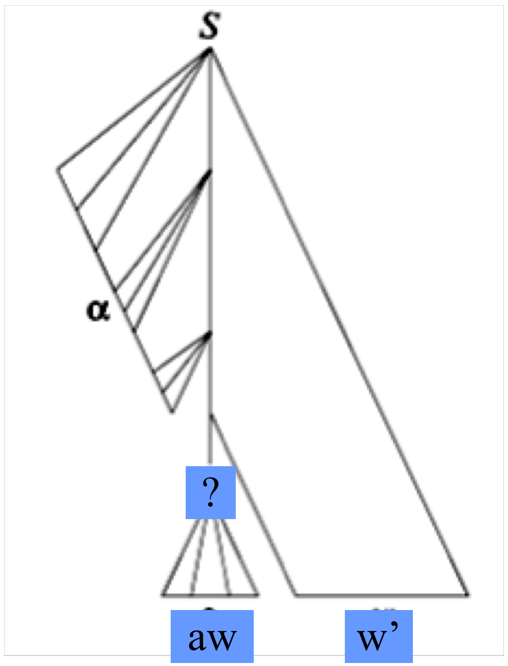
- Example grammar

  S → …

  X → aAB | …

  Y → aAC | …

- Cannot know what aw should be reduced to
  - ⇒ shift a to stack,
    reduce some part of w to A,
    shift A to stack, …
    till something is clear
  - Shift adds power to parsing
  - How to systematically do this?



Bottom-up Parsing

- Shift-reduce operations in bottom-up parsing
  - Shift the input into the stack
    - Wait for the current handle to complete or to appear
    - Or wait for a handle that may complete later
  - Reduce
    - Once the handle is completely in the stack, then reduce
  - The operations are determined by the parsing table
- Parsing table includes
  - Action table
    - Determine the action of shift or reduce
    - To shift (current handle is not completely or not yet in stack)
    - To reduce (current handle is completely in stack)
  - Goto table
    - Determine which state to go to next

Parsing Table

- Idea
  - Build a finite automata based on the grammar
  - Follow the automata to construct the parsing tables
- Characteristic finite state automata (CFSA)
  - Is the basis for building the parsing table
    - But the automata is not a part of the parsing table
  - States of the automata
    - Each state is represented by a set of LR(0) items
      - To keep track of what has already been seen (already in the stack)
        - In other words, keep track of the viable prefix
      - To track the possible productions that may be used for reduction
  - State transitions
    - Fired by grammar symbols (terminals or nonterminals)

Build the Automata

- LR(0) Item of a grammar G
  - Is a production of G with a distinguished position
  - Position is used to indicate how much of the handle has already been seen (in the stack)
    - For production $S \rightarrow a\ B\ S$, items for it include

      $S \rightarrow \bullet\ a\ B\ S$

      $S \rightarrow a \bullet B\ S$

      $S \rightarrow a\ B \bullet S$

      $S \rightarrow a\ B\ S \bullet$

      - Left of $\bullet$ are the parts of the handle that has already been seen
      - When $\bullet$ reaches the end of the handle $\Rightarrow$ reduction
    - For production $S \rightarrow \varepsilon$, the single item is

      $S \rightarrow \bullet$

Building the Automata

- Closure function Closure(I)
  - I is a set of items for a grammar G
  - Every item in I is in Closure(I)
  - If $A \rightarrow \alpha \bullet B\ \beta$ is in Closure(I) and $B \rightarrow \gamma$ is a production in G
    Then add $B \rightarrow \bullet\ \gamma$ to Closure(I)
    - If it is not already there
    - Meaning
      - When α is in the stack and B is expected next
      - One of the B-production rules may be used to reduce the input to B
        - May not be one-step reduction though
  - Apply the rule until no more new items can be added

Building the Automata

- Goto function Goto(I,X)
  - X is a grammar symbol
  - If $A \rightarrow \alpha \bullet X \beta$ is in I then $A \rightarrow \alpha X \bullet \beta$ is in Goto(I, X)
    - Let J denote the set constructed by this step
  - All items in Closure(J) are in Goto(I, X)
  - Meaning
    - If I is the set of valid items for some viable prefix γ
    - Then goto(I, X) is the set of valid items for the viable prefix γX

Building the Automata

- Augmented grammar
  - G is the grammar and S is the staring symbol
  - Construct G' by adding production S' $\rightarrow$ S into G
    - S' is the new starting symbol
    - E.g.: G: $S \rightarrow \alpha \mid \beta$ $\Rightarrow$ G': $S' \rightarrow S$, $S \rightarrow \alpha \mid \beta$
  - Meaning
    - The starting symbol may have several production rules and may be used in other non-terminal's production rules
    - Add S' $\rightarrow$ S to force the starting symbol to have a single production
    - When S' $\rightarrow$ S $\bullet$ is seen, it is clear that parsing is done

Building the Automata

- Given a grammar G
  - Step 1: augment G
  - Step 2: initial state
    - Construct the valid item set "I" of State 0 (the initial state)
    - Add S' $\rightarrow \bullet$ S into I
      - All expansions have to start from here
    - Compute Closure(I) as the complete valid item set of state 0
      - All possible expansions S can lead into
  - Step 3:
    - From state I, for all grammar symbol X

      Construct J = Goto(I, X)

      Compute Closure(J)
    - Create the new state with the corresponding Goto transition
      - Only if the valid item set is non-empty and does not exist yet
  - Repeat Step 3 till no new states can be derived

Building the Automata -- Example

- Grammar G:

  $S \rightarrow E$

  $E \rightarrow E + T \mid T$

  $T \rightarrow id \mid (E)$

  - Step 1: Augment G

    $S' \rightarrow S \qquad S \rightarrow E \qquad E \rightarrow E + T \mid T \qquad T \rightarrow id \mid (E)$

  - Step 2:
    - Construct Closure(I_0) for State 0
    - First add into I_0: $S' \rightarrow \bullet\, S$
    - Compute Closure(I_0)

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

      $E \rightarrow \bullet\, E + T \qquad E \rightarrow \bullet\, T$

      $T \rightarrow \bullet\, id \qquad T \rightarrow \bullet\, (E)$

Expect to see S next

S won't just appear
May have to see E first and
reduce it to S using this rule

Building the Automata -- Example

- Step 3
  - I_1
    - Add into I_1: Goto(I_0, S) = $S' \rightarrow S \bullet$
    - No new items to be added to Closure (I_1)
  - I_2
    - Add into I_2: Goto(I_0, E) = $S \rightarrow E \bullet$ $\quad E \rightarrow E \bullet + T$
    - No new items to be added to Cl… I_2)
  - I_3
    - Add into I_3: Goto(I_0
    - No new items to be
  - I_4
    - Add into I_4: Goto(I_0, id) = $T \rightarrow id \bullet$
    - No new items to be added to Closure (I_4)

I_0:

$S' \rightarrow \bullet S \quad S \rightarrow \bullet E$

$E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

$T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$

When E is moved to the stack (after a reduction), these two are the possible handles

$S \rightarrow E \bullet$ implies a reduction is to be done

- o should be done if seeing Follow(S)

$E \rightarrow E \bullet + T$ implies + is expected to be the next input

Building the Automata -- Example

- Step 3
  - I_5
    - Add into I_5: Goto(I_0, "(") = $T \rightarrow (\bullet E)$
    - Closure(I_5)

      $E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

      $T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$
  - No more moves from I_0
  - No possible moves from I_1
  - I_6
    - Add into I_6: Goto(I_2, +) = $E \rightarrow E + \bullet T$
    - Closure(I_5)

      $T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$
  - No possible moves from I_3 and I_4

I_0:

$S' \rightarrow \bullet S \quad S \rightarrow \bullet E$

$E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

$T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$

After seeing (, we expect E next
E could be reduced from other
E-production rules
So, put E-productions in the set

Building the Automata -- Example

- Step 3
  - I_7
    - Add into I_7: Goto(I_5, E) =

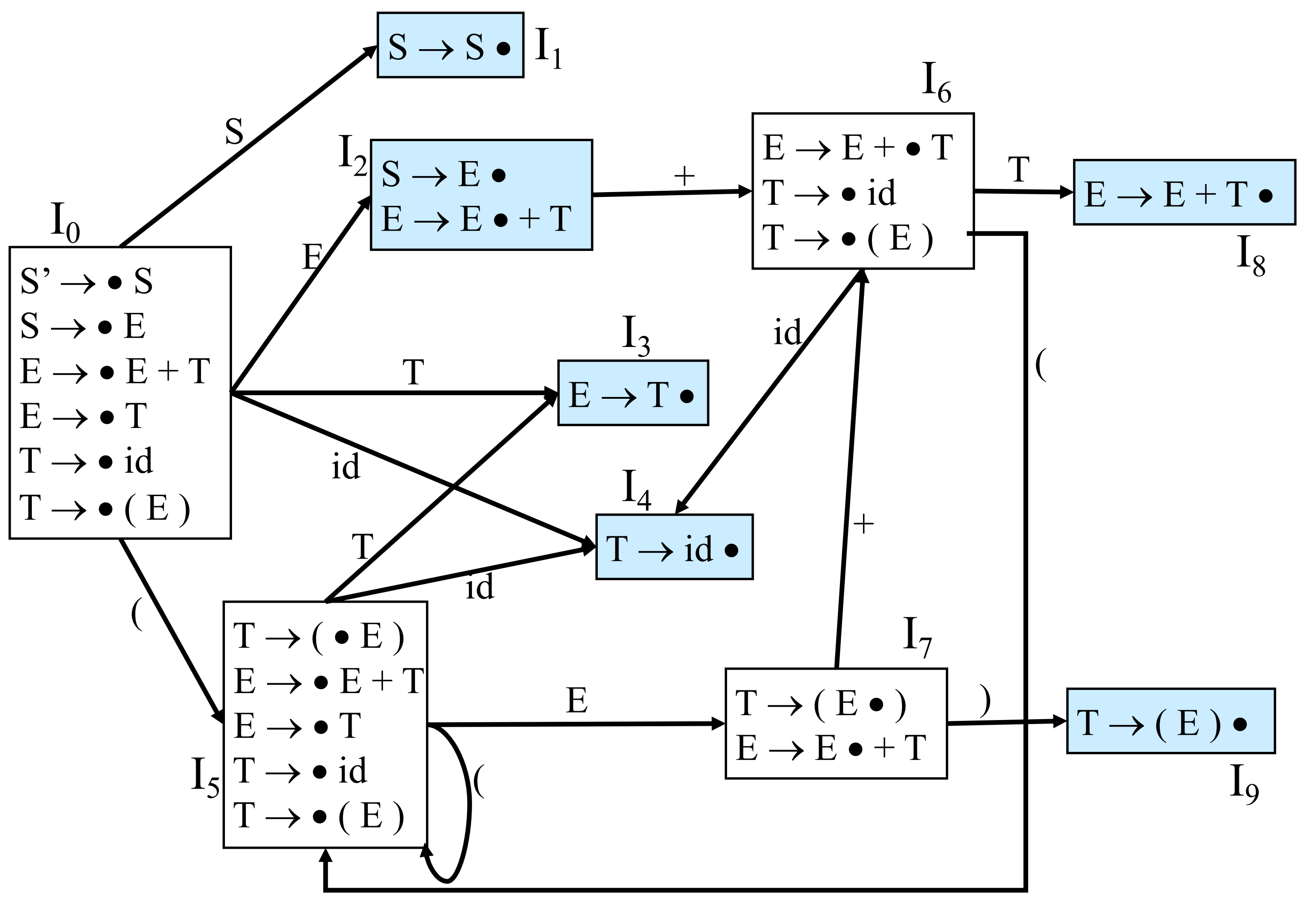
      $T \rightarrow (E \bullet)$ $E \rightarrow E \bullet + T$
    - No new items to be added to Closure (I_7)
  - Goto(I_5, T) = I_3
  - Goto(I_5, id) = I_4
  - Goto(I_5, "(") = I_5
  - No more moves from I_5
  - I_8
    - Add into I_8: Goto(I_6, T) = $E \rightarrow E + T \bullet$
    - No new items to be added to Closure (I_8)
  - Goto(I_6, id) = I_4
  - Goto(I_6, "(") = I_5

Building the Automata -- Example

- Step 3
  - I_9
    - Add into I_9: Goto(I_7, ")") =

      $T \rightarrow (E) \bullet$
    - No new items to be added to Closure (I_9)
  - Goto(I_7, +) = I_6
  - No possible moves from I_8 and I_9

Building the Automata -- Example

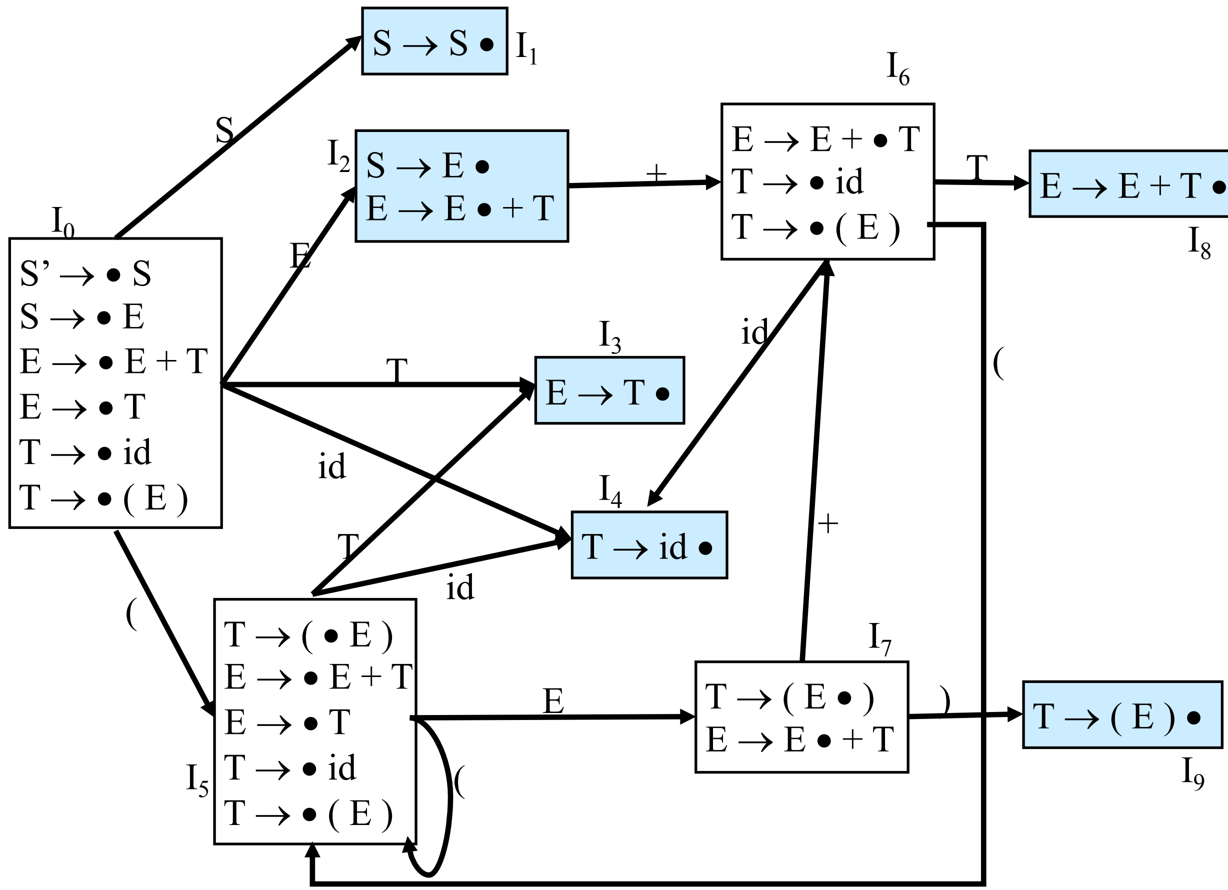


Building the Automata -- Example

Stack	Input	Action
0	id + id $	S4
0 id 4	+ id $	T→id, Goto[0,T]=3
0 T 3	+ id $	E→T, Goto[0,E]=2
0 E 2	+ id $	s6
0 E 2 + 6	id $	S4
0 E 2 + 6 id 4	$	T→id, Goto[6,T]=8
0 E 2 + 6 T 8	$	E→E+T, Goto[0,E]=2
0 E 2	$	S→E, Goto[0,S]=1
0 S 1	$	accept

See how parsing works directly on the automata

Follow(S) = {$}
Follow(E) = {+,), $}
Follow(T) = {+,), $}



Building the Parsing Table

- Action [M, N]
  - M states
  - N tokens
  - Actions =
    - Shift i: shift the input token into the stack and go to state i
    - Reduce i: reduce by the i-th production $\alpha \rightarrow \beta$
    - Accept
    - Error
- Goto [M, L]
  - M states
  - L non-terminals
  - Goto[i, j] = x
    - Move to state S_x

Building the Action Table

- If state I_i has item $A \rightarrow \alpha \bullet a\ \beta$, and
  - Goto(I_i, a) = I_j
  - Next symbol in the input is a
- Then Action[I_i, a] = I_j
  - Meaning: Shift "a" to the stack and move to state I_j
    - Need to wait for the handle to appear or to complete
- If State I_i has item $A \rightarrow \alpha \bullet$
- Then Action[S, b] = reduce using $A \rightarrow \alpha$
  - For all b in Follow(A)
  - Meaning: The entire handle α is in the stack, need to reduce
  - Need to wait to see Follow(A) to know that the handle is ready
    - E.g. $S \rightarrow E \bullet$ $\quad$ $E \rightarrow E \bullet + T$
    - Current input can be either Follow(S) or +

Building the Action Table

- If state has $S' \rightarrow S_0 \bullet$
- Then Action[S, $] = accept

- Current state
  - The action to be taken depends on the current state
    - In LL, it depends on the current non-terminal on the top of the stack
    - In LR, non-terminal is not known till reduction is done
  - Who is keeping track of current state?
  - The stack
    - Need to push the state also into the stack
    - The stack includes the viable prefix and the corresponding state for each symbol in the viable prefix

Building the Goto Table

- If $Goto(I_i, A) = I_j$
- Then $Goto[i, A] = j$
- Meaning
  - When a reduction $X \rightarrow \alpha$ taken place
  - The non-terminal X is added to the stack replacing α
  - What should the state be after adding X
  - This information is kept in Goto table

Building the Parsing Table -- Example

Follow(S) = {$}
Follow(E) = {+,), $}
Follow(T) = {+,), $}

	+	id	(	)	$	S	E	T
0		4	5			1	2	3
1					Acc			
2	6				S→E			
3	E→T			E→T	E→T			
4	T→id			T→id	T→id			
5		4	5				7	3
6		4	5					8
7	6			9				
8	E→E+T			E→E+T	E→E+T			
9	T→(E)			T→(E)	T→(E)			

Action Table

Goto Table

LR Parsing Algorithm

- Elements
  - Parser, parsing tables, stack, input
- Initialization
  - Append the $ at the end of the input
  - Push state 0 into the stack
    - On the top of the stack, it is always a state
    - It is the current state of parsing

LR Parsing Algorithm

- Steps
  - If Action[x, a] = y
    - x is the current state, on the top of the stack
    - a is the input token
  - Then shift a into the stack and put y on top of the stack
  - If Action[x, a] = $A \rightarrow \alpha$
    - Note that a is in Follow(A)
  - Then
    - x is the current state, on the top of the stack
    - Pop the handle α and all the state corresponding to α out of the stack
    - y is the state on the top of the stack after popping
    - Check Goto table, if Goto[y, A] = z
    - Push A and then z into the stack

LR Parsing - Example

	+	id	(	)	$	S	E	T
0		4	5			1	2	3
1								
2								
3								
4	T→id			T→id	T→id			
5		4	5				7	3
6		4	5					8
7	6			9				
8	E→E+T			E→E+T	E→E+T			
9	T→(E)			T→(E)	T→(E)			

Rightmost derivation:

$S \Rightarrow E \Rightarrow E + T \Rightarrow E + id \Rightarrow T + id \Rightarrow id + id$

Reverse trace back:

Reduce left most input first.

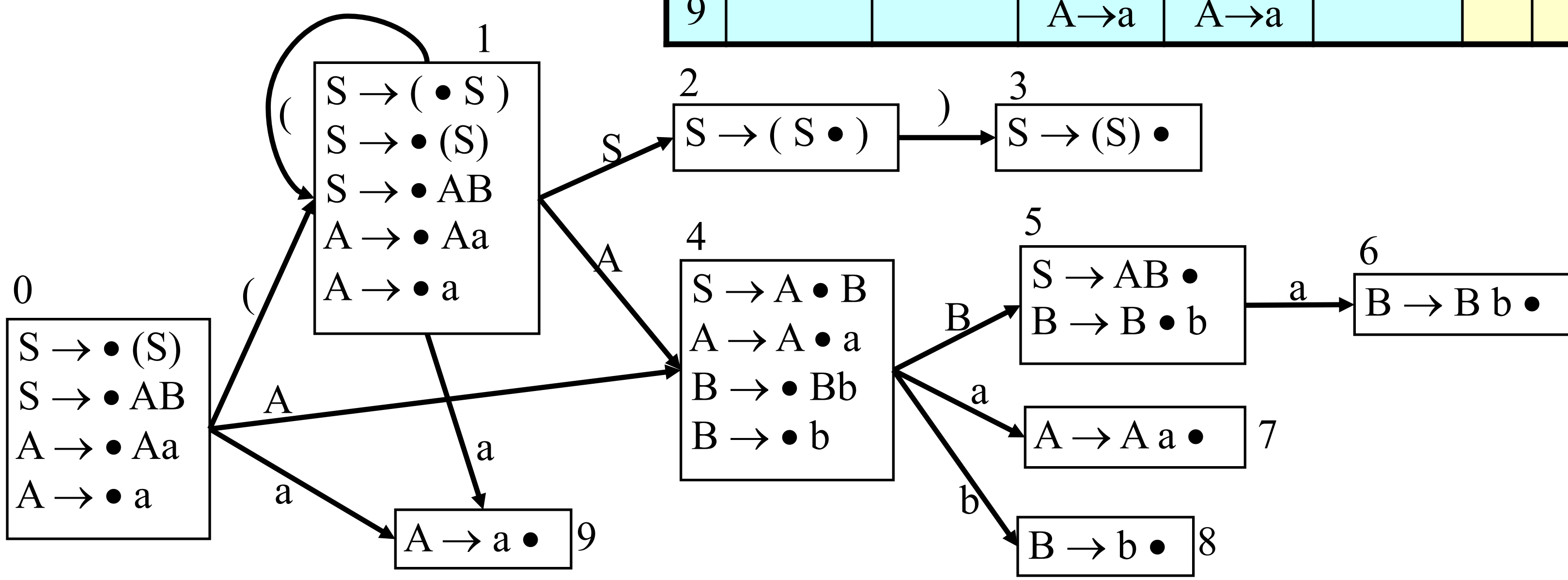
Stack	Input	Action
0	id + id $	S4
0 id 4	+ id $	T→id, Goto[0,T]=3
0 T 3	+ id $	E→T, Goto[0,E]=2
0 E 2	+ id $	s6
0 E 2 + 6	id $	S4
0 E 2 + 6 id 4	$	T→id, Goto[6,T]=8
0 E 2 + 6 T 8	$	E→E+T, Goto[0,E]=2
0 E 2	$	S→E, Goto[0,S]=1
0 S 1	$	accept

LR Parsing -- Anoth

$S \rightarrow (S) \mid AB$
$A \rightarrow Aa \mid a$
$B \rightarrow Bb \mid b$

Follow(S) = {\$,)}
Follow(A) = {a, b}
Follow(B) = {\$,), b}

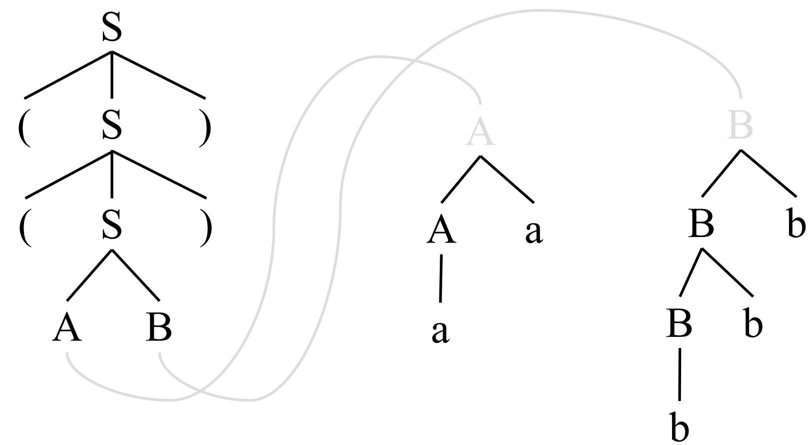
	(	)	a	b	\$	S	A	B
0	1		9			?	4	
1	1		9			2	4	
2		3						
3		$S \rightarrow (S)$			$S \rightarrow (S)$			
4			7	8				5
5		$S \rightarrow AB$		6	$S \rightarrow AB$			
6		$B \rightarrow Bb$		$B \rightarrow Bb$	$B \rightarrow Bb$			
7			$A \rightarrow Aa$	$A \rightarrow Aa$				
8		$B \rightarrow b$		$B \rightarrow b$	$B \rightarrow b$			
9			$A \rightarrow a$	$A \rightarrow a$				

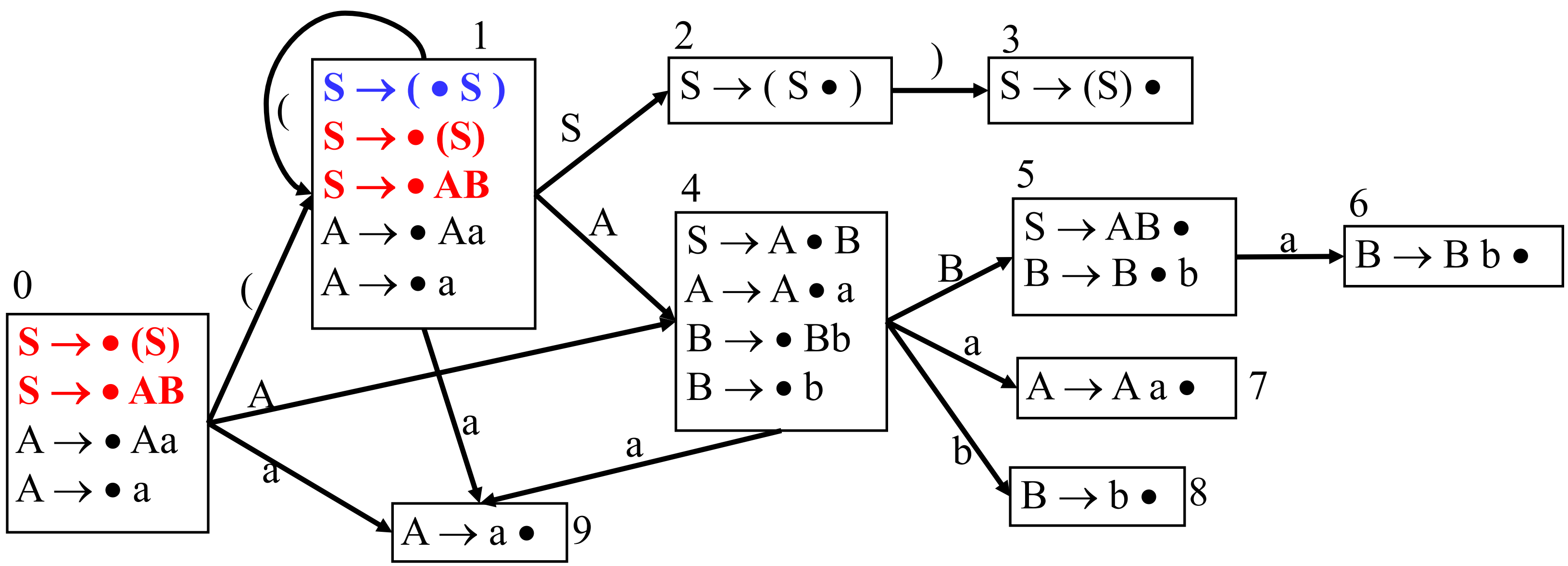


LR Parsing -- Looking into the Automata

Input: ((aabbb))$

S ⇒ **(S)** ⇒ (**(S)**) ⇒ ((**AB**))
⇒ ((A**Bb**)) ⇒ ((A**Bb**b)) ⇒ ((A**b**bb))
⇒ ((**Aa**bbb)) ⇒ ((**a**abbb))





LR Parsing -- The RM Deriv

	(	)	a	b	$	S	A	B
0	1		9			?	4	
1	1		9			2	4	
2		3						
3		S→(S)			S→(S)			
4			7	8				5
5		S→AB		6	S→AB			
6		B→Bb		B→Bb	B→Bb			
7			A→Aa	A→Aa				
8		B→b		B→b	B→b			
9			A→a	A→a				

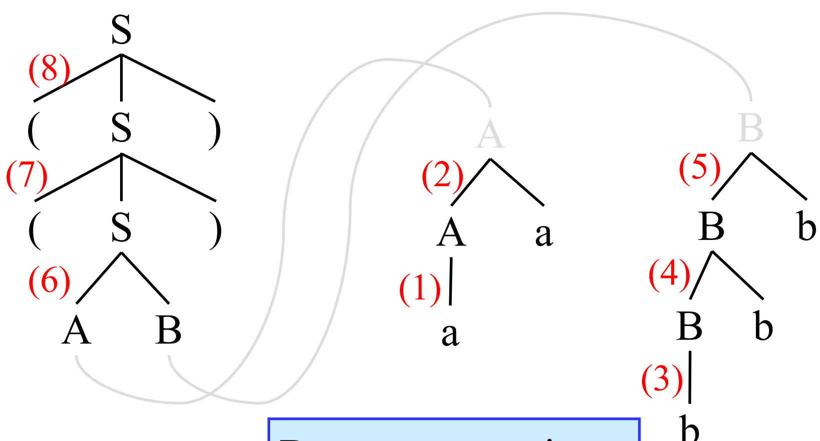
Input: ((abbb))$

Stack	Input	Action
0	((aabbb))$	S1
0(1	(aabbb))$	S1
0(1(1	aabbb))$	S9
0(1(1a9	abbb))$	A→a
0(1(1A4	abbb))$	S6
0(1(1A4a7	bbb))$	A→Aa
0(1(1A4	bbb))$	S8
0(1(1A4b8	bb))$	B→b
0(1(1A4B5	bb))$	S6
0(1(1A4B5b6	b))$	B→Bb
0(1(1A4B5	b))$	S6
0(1(1A4B5b6	))$	B→Bb
0(1(1A4B5	))$	S→AB
0(1(1S2	))$	S3
0(1(1S2)3	)$	S→(S)
0(1S2	)$	S3
0(1S2)3	$	S→(S)
0S	$	?accept

LR Parsing -- The RM De

S
(8) ⇒ (S)
(7) ⇒ ((S))
(6) ⇒ ((AB))
(5) ⇒ ((ABb))
(4) ⇒ ((ABbb))
(3) ⇒ ((Abbb))
(2) ⇒ ((Aabbb))
(1) ⇒ ((aabbb))

traverse the rightmost derivation backwards



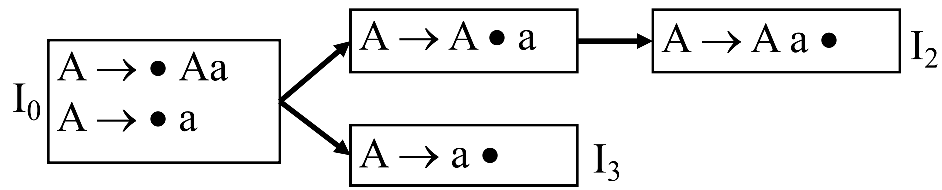
Bottom up parsing

Stack	Input	Action	Order
0	((aabbb))$	S1	
0(1	(aabbb))$	S1	
0(1(1	aabbb))$	S9	
0(1(1a9	abbb))$	A→a	(1)
0(1(1A4	abbb))$	S6	
0(1(1A4a7	bbb))$	A→Aa	(2)
0(1(1A4	bbb))$	S8	
0(1(1A4b8	bb))$	B→b	(3)
0(1(1A4B5	bb))$	S6	
0(1(1A4B5b6	b))$	B→Bb	(4)
0(1(1A4B5	b))$	S6	
0(1(1A4B5b6	))$	B→Bb	(5)
0(1(1A4B5	))$	S→AB	(6)
0(1(1S2	))$	S3	
0(1(1S2)3	)$	S→(S)	(7)
0(1S2	)$	S3	
0(1S2)3	$	S→(S)	(8)
0S	$	?accept	

SLR Parsing

- LR
  - L: input scanned from left
  - R: traverse the rightmost derivation path
- LR(0) = SLR(1)
  - The LR parser we discussed is LR(0)
    - 0 in LR: lookahead symbol with the item (will be clear later)
  - LR(0) is also called SLR(1)
    - Simple LR
    - 1 in SLR: lookahead symbol

SLR and LL



- Example:

  $A \to Aa \mid a$

  Follow(A) = {a, \$}

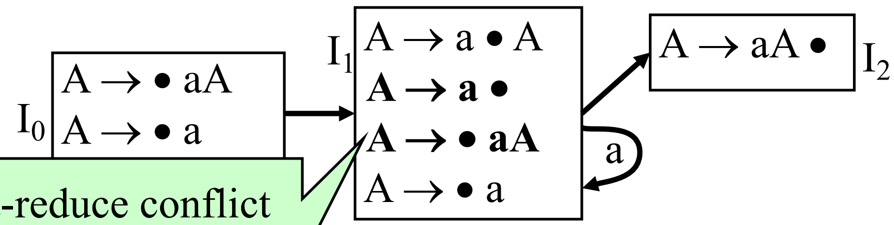
	a	\$	A
0	3		1
1	2		
2	A→Aa	A→Aa	
3	A→a	A→a	

Stack	Input	Action
0	aaa\$	S3
0a3	aa\$	A→a, Goto[0,A]=1
0A1	aa\$	S2
0A1a2	a\$	A→Aa Goto[0,A]=1
0A1	a\$	S2
0A1a2	\$	A→Aa Goto[0,A]=1
0A1	\$	

Unclear accepting state
Incorrect state transition

- Not LL
  - Left recursive grammar
- But is SLR(1)
  - First a got reduced to A
  - The remaining a's got reduced with the already generated A (Aa)
  - In LR, it is reduction based, when seeing 'a', '$A \to a$' is the only choice, after there is A, then reduce Aa by $A \to Aa$

SLR and LL



- Example:

  $A \rightarrow aA \mid a$

  Follow(A) = {

Potential shift-reduce conflict
shift: expect to see ‘a’
reduce: follow(A) only has $
⇒ no problem

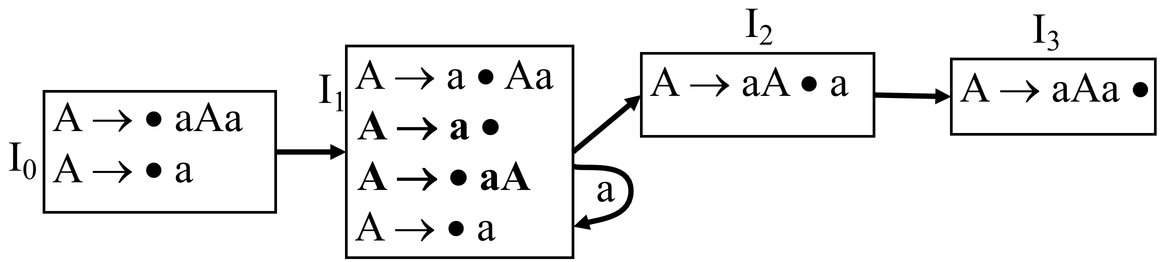
0	1		
1	1	A→a	2
2		A→aA	

Stack	Input	Action
0	aaa$	S1
0a1	aa$	S1
0a1a1	a$	S1
0a1a1a1	$	A→a Goto[1,A]=2
0a1a1A2	$	A→aA Goto[1,A]=2
0a1A2	$	same as above
0A?	$	

Unclear accepting state
The input string is actually acceptable
If [0,$] is *accept*, will accept ε

- Not LL(1)
  - Productions for A have left factors
- But is SLR(1)
  - All ‘a’s got shifted to stack
  - Final ‘a’, seeing $, got reduced to ‘A’
  - All ‘a’s in stack got reduced with newly generated ‘A’s

SLR and LL



- Example:

  $A \to aAa \mid a$

  Follow(A) = {\$, a}

Shift-reduce conflict
reduce: follow(A) has \$, a
$\Rightarrow$ conflict

	a	\$	A
0	1		
1	1 A→a	A→a	2
2		A→aA	

Stack	Input	Action
0	aaa\$	S1
0a1	aa\$	A→a
0A?	aa\$	

- Not LL(1)
  - Productions for A have left factors
- Not SLR(1)
  - Has shift-reduce conflict

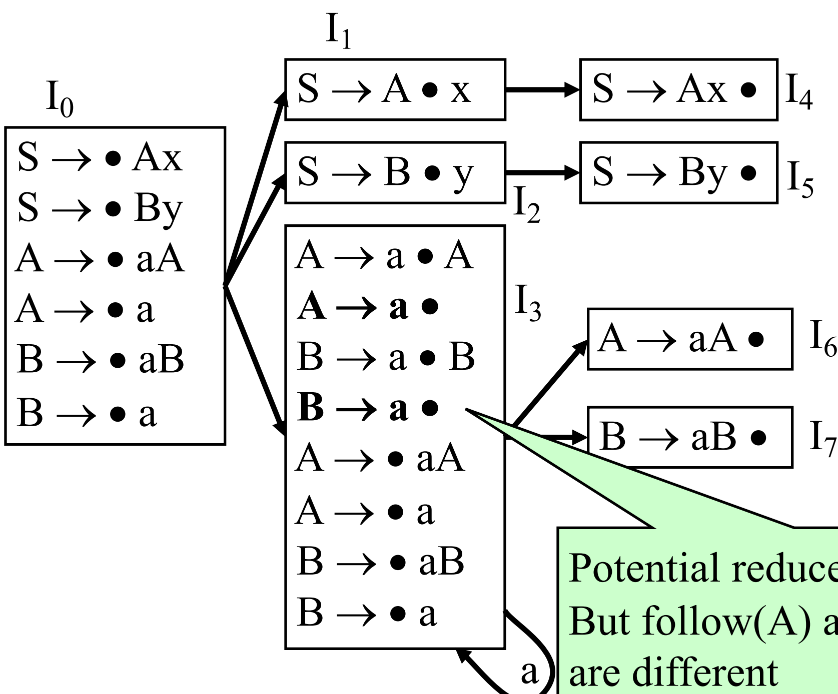
SLR and LL

- Example:

  $S \rightarrow Ax \mid By$ 　 Follow(S) = {$}

  $A \rightarrow aA \mid a$ 　 Follow(A) = {x}

  $B \rightarrow aB \mid a$ 　 Follow(B) = {y}



Potential reduce-reduce conflict
But follow(A) and follow(B)
are different

Stack	Input	Action
0	aaax$	S3
0a3	aax$	S3
0a3a3	ax$	S1
0a3a3a3	x$	A→a Goto[3,A]=6
0a3a3A6	x$	A→aA Goto[3,A]=6
0a3A6	x$	same as above
0A1	x$	S4
0A1x4	$	S→Ax
0S	$	

Unclear accepting state
S does not appear at
the right hand side
So, no Goto info

SLR and LL

- Continue with the example:

  $S \to Ax \mid By$

  $A \to aA \mid a$

  $B \to aB \mid a$

  - Not LL(k)
    - $S \to Ax$ and $S \to By$, First(Ax) and First(By) are 'a'
    - Even with large k, $First_k$ of both will have "aa…a"
  - Is SLR(1)
    - No problem with $A \to aA$ and $A \to a$, they lead to different states
    - No problem with $A \to a$ and $B \to a$, just go back to the same state
      - o $\Rightarrow$ During parsing, 'a' continuously got shifted into the stack
      - o When x or y appears, reduce
        - By that time, it is clear which rule to use for reduction
        - Follow(A) = {x}, if seeing x, reduce with $A \to a$
        - Follow(B) = {y}, if seeing y, reduce with $B \to a$

SLR and LL

- Example:

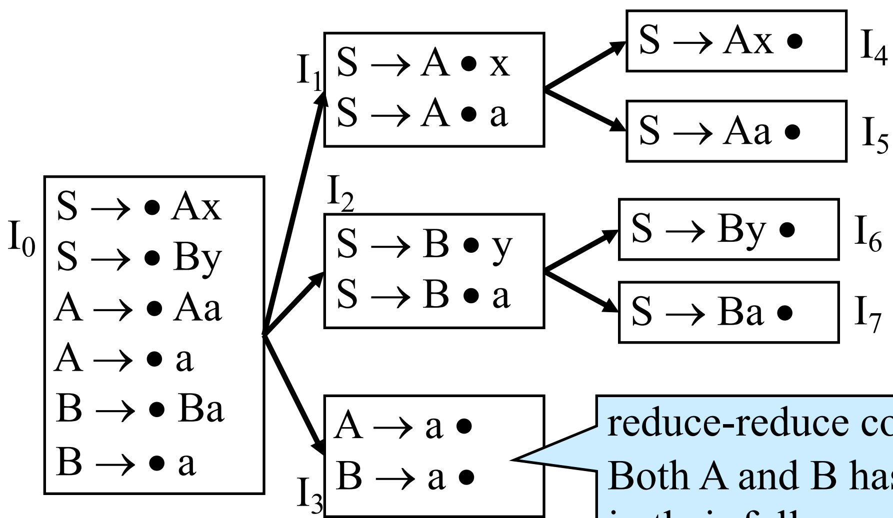
  $S \rightarrow Ax \mid By$

  $A \rightarrow Aa \mid a$

  $B \rightarrow Ba \mid a$

Stack	Input	Action
0	aaax$	S3
0a3	aax$	Reduction Multiple productions

Have to make decision too soon, right at the first ‘a’



reduce-reduce conflict
Both A and B has ‘a’ in their follow sets

Follow(S) = {$}
Follow(A) = {x, a}
Follow(B) = {y, a}

SLR and LL

- Continue with the example:

  $S \to Ax \mid By$

  $A \to Aa \mid a$

  $B \to Ba \mid a$

  - Not LL
    - $S \to Ax$ and $S \to By$, First(Ax) and First(By) are ‘a’
    - Even with large k, $First_k$ of both A and B will have “aa…a” (A and B are both in S’s productions)
  - Not SLR either
    - Not SLR(k), for any k
    - Even with large k, $Follow_k$ of both A and B will have “aa…a”

SLR and LL

- Example:

  $S \to (X \mid [Y$

  $X \to A) \mid B]$

  $Y \to A] \mid B)$

  $A \to \varepsilon$

  $B \to \varepsilon$

  - Not SLR(1)
  - Is LL(1)

$S \to \bullet (X$
$S \to \bullet [Y$

$S \to (\bullet X$
$X \to \bullet A)$
$X \to \bullet B]$
$A \to \bullet$
$B \to \bullet$

$S \to [\bullet Y$
$Y \to \bullet A]$
$Y \to \bullet B)$
$A \to \bullet$
$B \to \bullet$

reduce-reduce conflict
Both A and B has]/) in their follow sets

Follow(S) = {$}
Follow(X) = {$}
Follow(Y) = {$}
Follow(A) = {],)}
Follow(B) = {],)}

The rules of each nonterminal have different first symbols
$A \to \varepsilon$ and $B \to \varepsilon$ are from different nonterminals

First(A) = $\{ \varepsilon \}$
First(B) = $\{ \varepsilon \}$
First(X) = $\{ \varepsilon,),] \}$
First(Y) = $\{ \varepsilon,),] \}$
First(S) = $\{ (, [\}$

	(	[	)	]	$
S	$S \to (X$	$S \to [Y$			
X			$X \to A)$	$X \to B]$	
Y			$Y \to B)$	$Y \to A]$	
A			$A \to \varepsilon$	$A \to \varepsilon$	
B			$B \to \varepsilon$	$B \to \varepsilon$	

SLR Parser Family

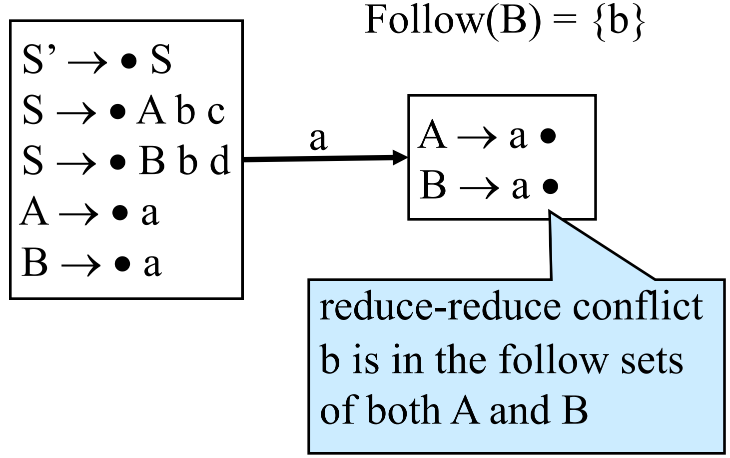
- Consider grammar G

  $S \to A\ b\ c \mid B\ b\ d$

  $A \to a$

  $B \to a$

Follow(S) = {$}
Follow(A) = {b}
Follow(B) = {b}



- G is SLR(2)
  - Lookahead two characters will resolve the conflict
  - $Follow_2(A) = \{bc\}$, $Follow_2(B) = \{bd\}$
  - $Action[4, bc] = A \to a$
  - $Action[4, bd] = B \to a$

SLR Parser Family

- Consider grammar G

  $S \to A\ b^{k-1}c \mid B\ b^{k-1}d$

  $A \to a$

  $B \to a$

  - G is SLR(k) not SLR(k-1)
    - Need to lookahead k characters in the Follow set
    - $Follow_{k-1}(A) = \{b^{k-1}\}$, $Follow_{k-1}(B) = \{b^{k-1}\}$
    - $Follow_k(A) = \{b^{k-1}c\}$, $Follow_k(B) = \{b^{k-1}d\}$

SLR and LR

- Consider grammar G

  $S \rightarrow L = R$

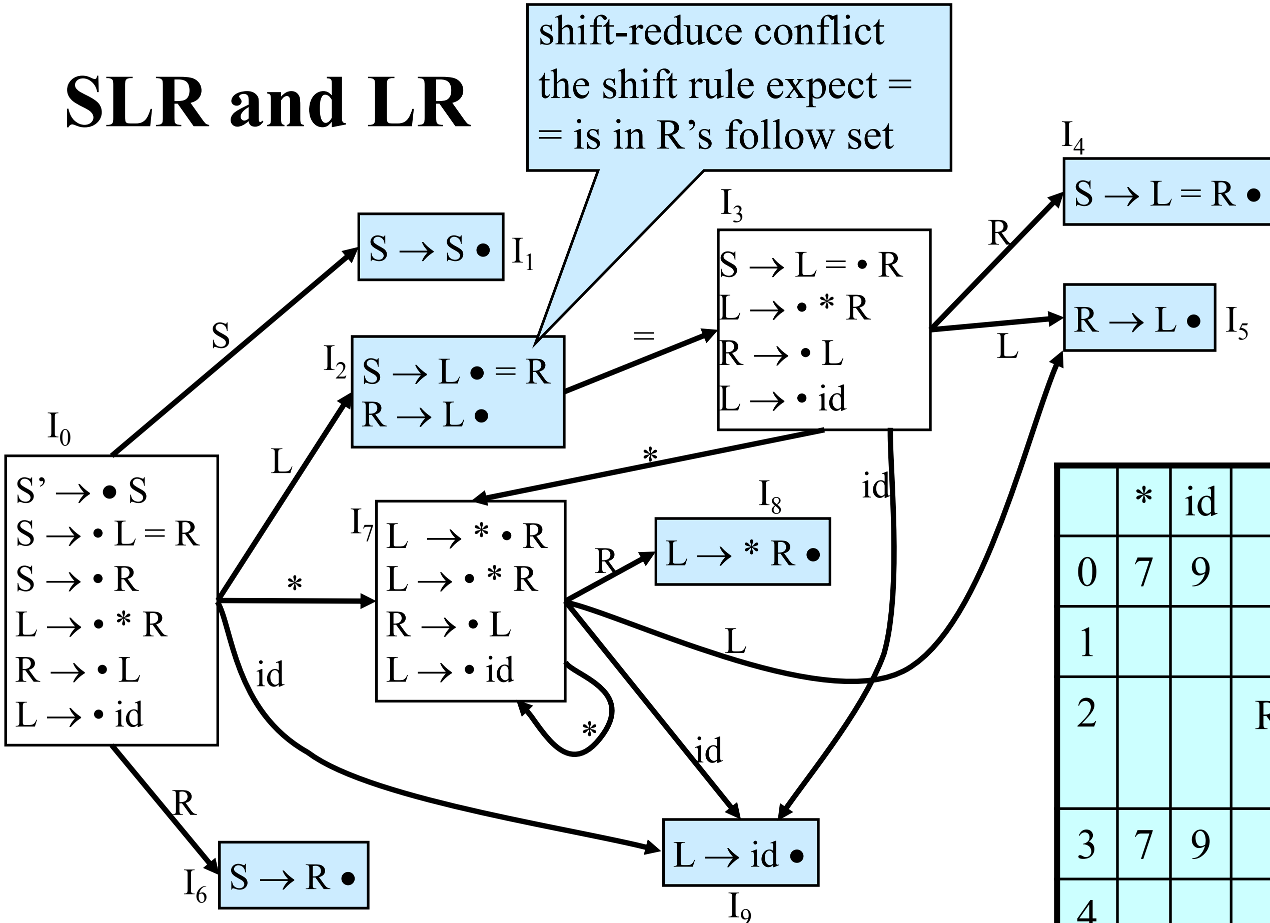
  $S \rightarrow R$

  $R \rightarrow L$

  $L \rightarrow * R$

  $L \rightarrow id$

SLR and LR



S → L = R
S → R
R → L
L → * R
L → id

Follow(S) = {$}
Follow(L) = {=, $}
Follow(R) = {=, $}

	*	id	=	$	S	L	R
0	7	9			1	2	6
1				Acc			
2			R→L 3	R→L			
3	7	9				5	4
4				S→L=R			
5			R→L	R→L			
6				S→R			
7	7	9				5	8
8			R→*L	R→*L			
9			L→id	L→id			

SLR and LR

- Grammar G has shift-reduce conflict
  - Not helpful by looking further ahead the Follow set
    - $Follow_k(L)$ = {$, =id$, =*id$, =**id$, …, =*…*id$, =*…*id, =*…*}
    - $Follow_k(R) = Follow_k(L)$

    ⇒ This is not SLR(k)
      - Further lookahead will not help with distinguishing $Follow_k(R)$ from $Follow_k(L)$

SLR and LR

- What is the problem?
  - Lookahead information is too crude
  - Need to distinguish
    - If $L \rightarrow * R$ is from $S \Rightarrow L = R \Rightarrow *R = R$, then Follow(R) = {=, $}
    - If $L \rightarrow * R$ is from $S \Rightarrow R \Rightarrow L \Rightarrow *R$, then Follow(R) = {$}
- Solution:
  - Carry the specific lookahead information with the LR(0) item
  - The item becomes LR(1) item
  - Use the lookahead symbol(s) with the item to identify the correct reduction rule to apply
- Canonical LR Parsing
  - The parsing scheme based on LR(1) item

LR(1) Item

- LR(1) Item of a grammar G
  - $[A \to \alpha \bullet \beta, a]$
  - $A \to \alpha \bullet \beta$ is an LR(0) item
  - a is the lookahead symbol (a terminal in Follow(A))
  - $[A \to \alpha \bullet, a]$ implies
    - $S \Rightarrow^* \delta A \gamma \Rightarrow \delta \alpha \gamma$
    - a is in First(γ\$)
    - I.e., "a" follows A in a right sentential form
- When $[A \to \alpha \bullet, a]$ is in the state
  - $\Rightarrow$ Reduction (same as SLR)
  - But only if "a" is seen in the input string
- Next, need to define Closure and Goto functions for LR(1) items

Building the Automata

- Changes to Closure(I)
  - If $A \rightarrow \alpha \bullet B\ \beta$ is in Closure(I) and $B \rightarrow \gamma$ is a production in G

    Then add $B \rightarrow \bullet\ \gamma$ to Closure(I)

  $\Rightarrow$

  - If $[A \rightarrow \alpha \bullet B\ \beta, a]$ is in Closure(I) and $B \rightarrow \gamma$ is a production in G

    Then add $[B \rightarrow \bullet\ \gamma, c]$ to Closure(I)

    - For all c, $c \in First(\beta a)$
- Changes to Goto(I,X)
  - If $A \rightarrow \alpha \bullet X\ \beta$ is in I then $A \rightarrow \alpha\ X \bullet \beta$ is in Goto(I, X)

  $\Rightarrow$

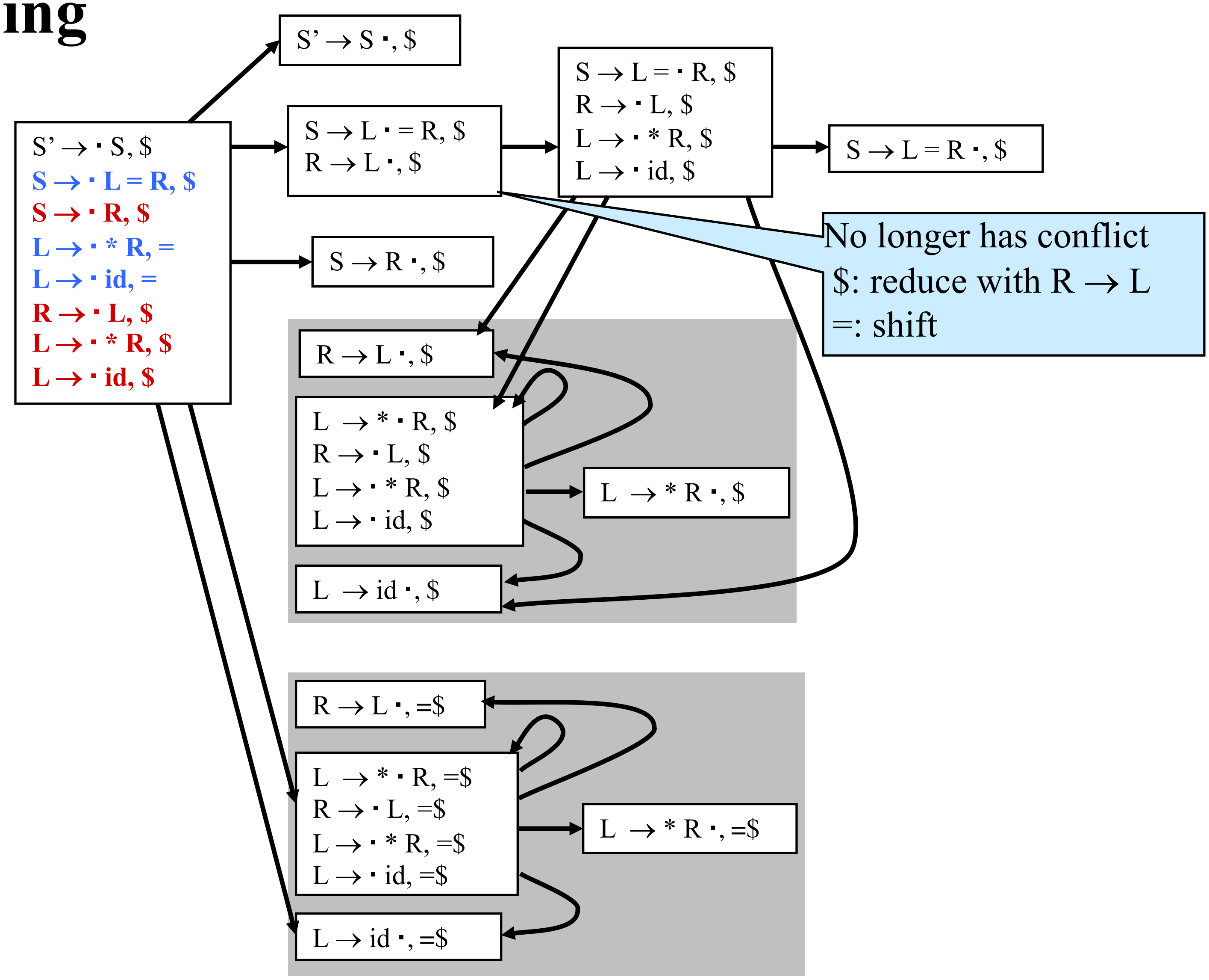
  - If $[A \rightarrow \alpha \bullet X\ \beta, a]$ is in I then $[A \rightarrow \alpha\ X \bullet \beta, a]$ is in Goto(I, X)
    - Simply carry the lookahead symbol over

Building the Action Table

- If state has item $[A \rightarrow \alpha \bullet a\ \beta, b]$
  - Add the shift action to the Action table (same as before)
- If state has $[S' \rightarrow S_0 \bullet, \$]$
  - Add accept to Action table (same as before)
- If State I_i has item $[A \rightarrow \alpha \bullet, b]$
  - $Action[S, b]$ = reduce using $A \rightarrow \alpha$
    - Not for all terminals in Follow(A)
    - Only for all terminals in the lookahead part of the item

- Goto table construction is the same as before

LR Parsing



S’ → · S, $
S → · L = R, $
S → · R, $
L → · * R, =
L → · id, =
R → · L, $
L → · * R, $
L → · id, $
S’ → S ·, $
S → L · = R, $
R → L ·, $
S → R ·, $
S → L = · R, $
R → · L, $
L → · * R, $
L → · id, $
S → L = R ·, $
No longer has conflict
$: reduce with R → L
=: shift
R → L ·, $
L → * · R, $
R → · L, $
L → · * R, $
L → · id, $
L → * R ·, $
L → id ·, $
R → L ·, =$
L → * · R, =$
R → · L, =$
L → · * R, =$
L → · id, =$
L → * R ·, =$
L → id ·, =$

LR Parsing

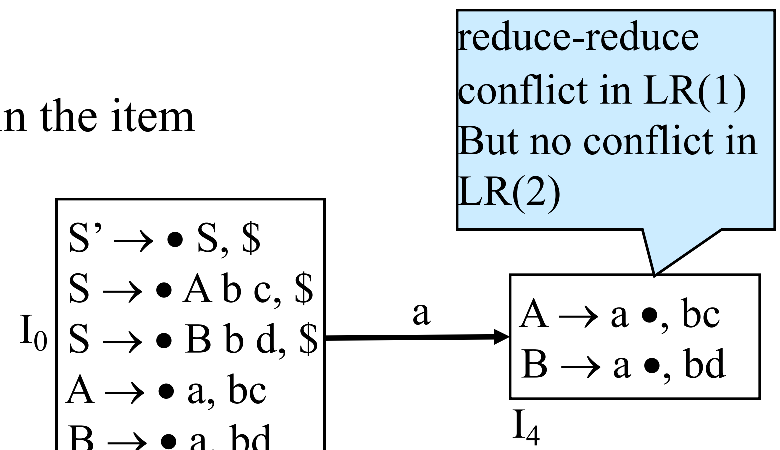
- The parsing algorithm is the same for the LR family
  - Only the table is different
- LR is more powerful
  - An SLR(1) grammar is always an LR(1), but not vice versa
  - LR(1)
    - Use one lookahead symbol in the item
  - LR(k)
    - Use k lookahead symbols in the item
  - LR(2) grammar

    $S \rightarrow A\ b\ c \mid B\ b\ d$

    $A \rightarrow a$

    $B \rightarrow a$

    - SLR(2) also



SLR and LR

❖ Example:

$S \to (X \mid [Y$

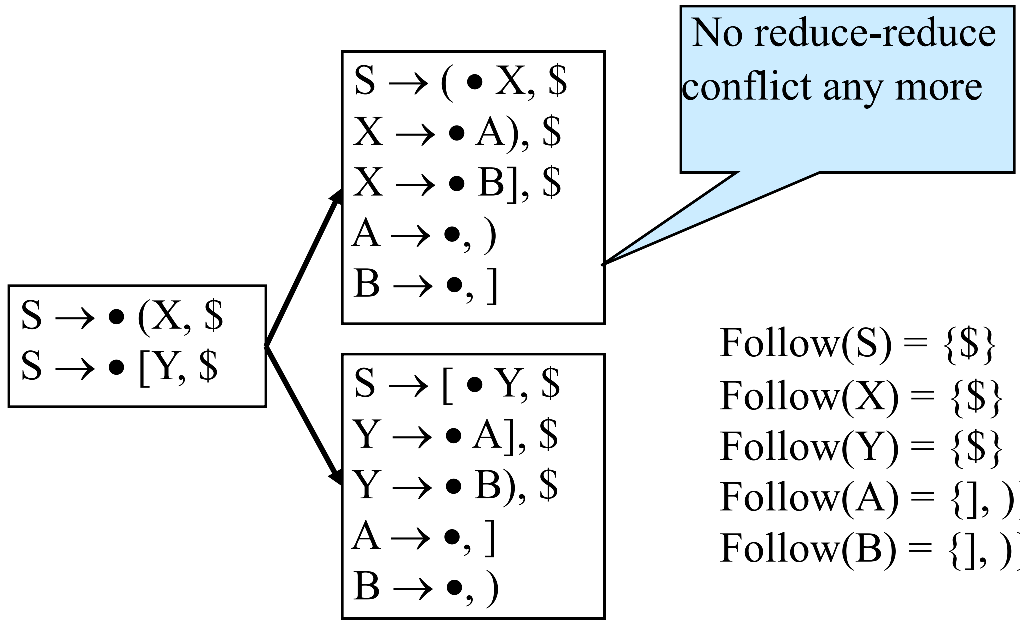
$X \to A) \mid B]$

$Y \to A] \mid B)$

$A \to \varepsilon$

$B \to \varepsilon$

- ❑ Not SLR(1)
- ❑ Is LR(1)

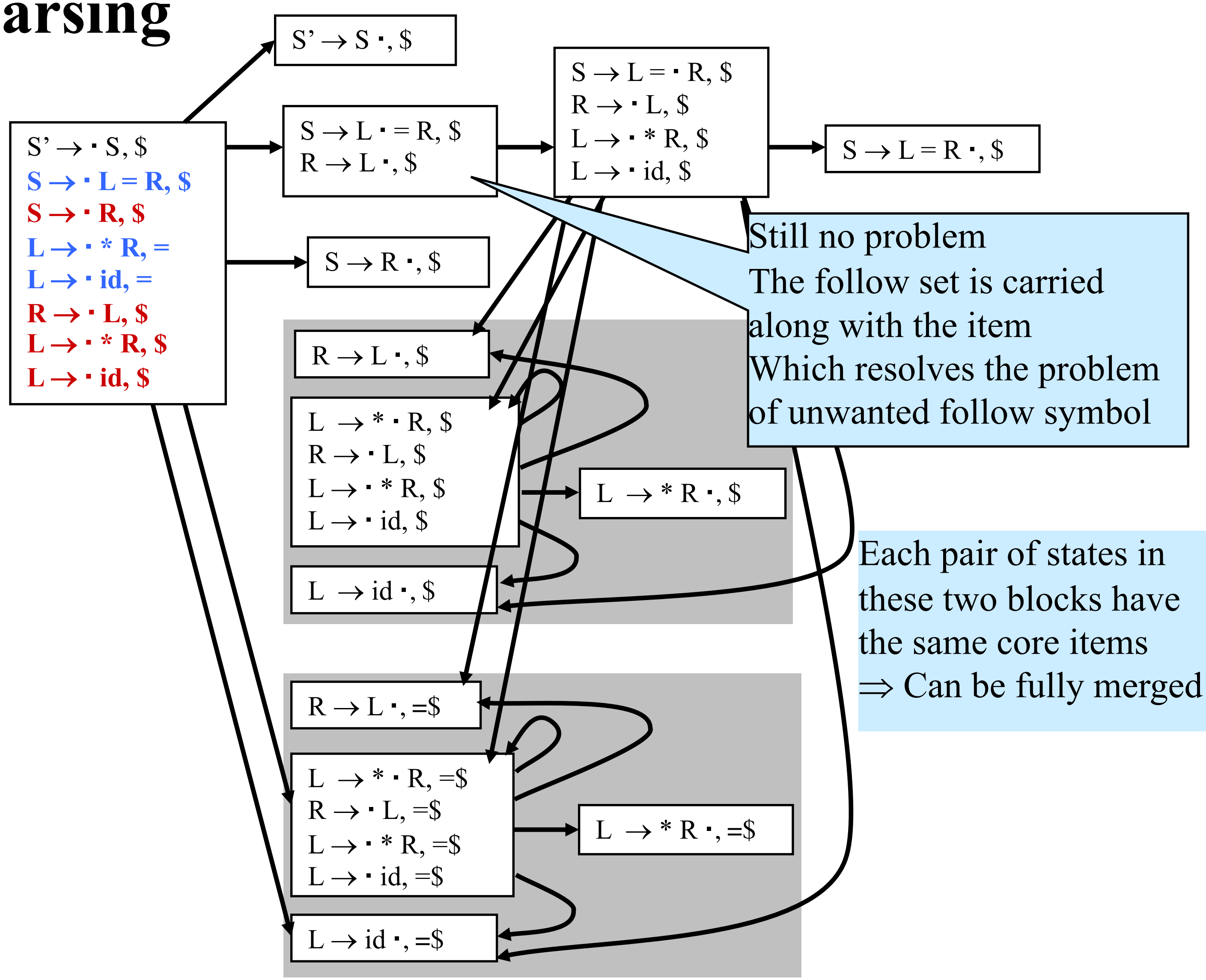


Follow(S) = {$}
Follow(X) = {$}
Follow(Y) = {$}
Follow(A) = {],)}
Follow(B) = {],)}

LR Parsing

- LR is more powerful than SLR
- But LR has a larger number of states
  - Higher space consuming
    - Common programming language has hundreds of states and hundreds of terminals
    - Approximately 100 X 100 table size
  - Can the number of states in LR be reduced?
    - Some states in LR are duplicated and can be merged
- LALR
  - LookAhead LR
  - Try to merge states in LR(1) automata
  - When the core items in two LR(1) states are the same
    $\Rightarrow$ merge them

LALR Parsing



LALR Parsing

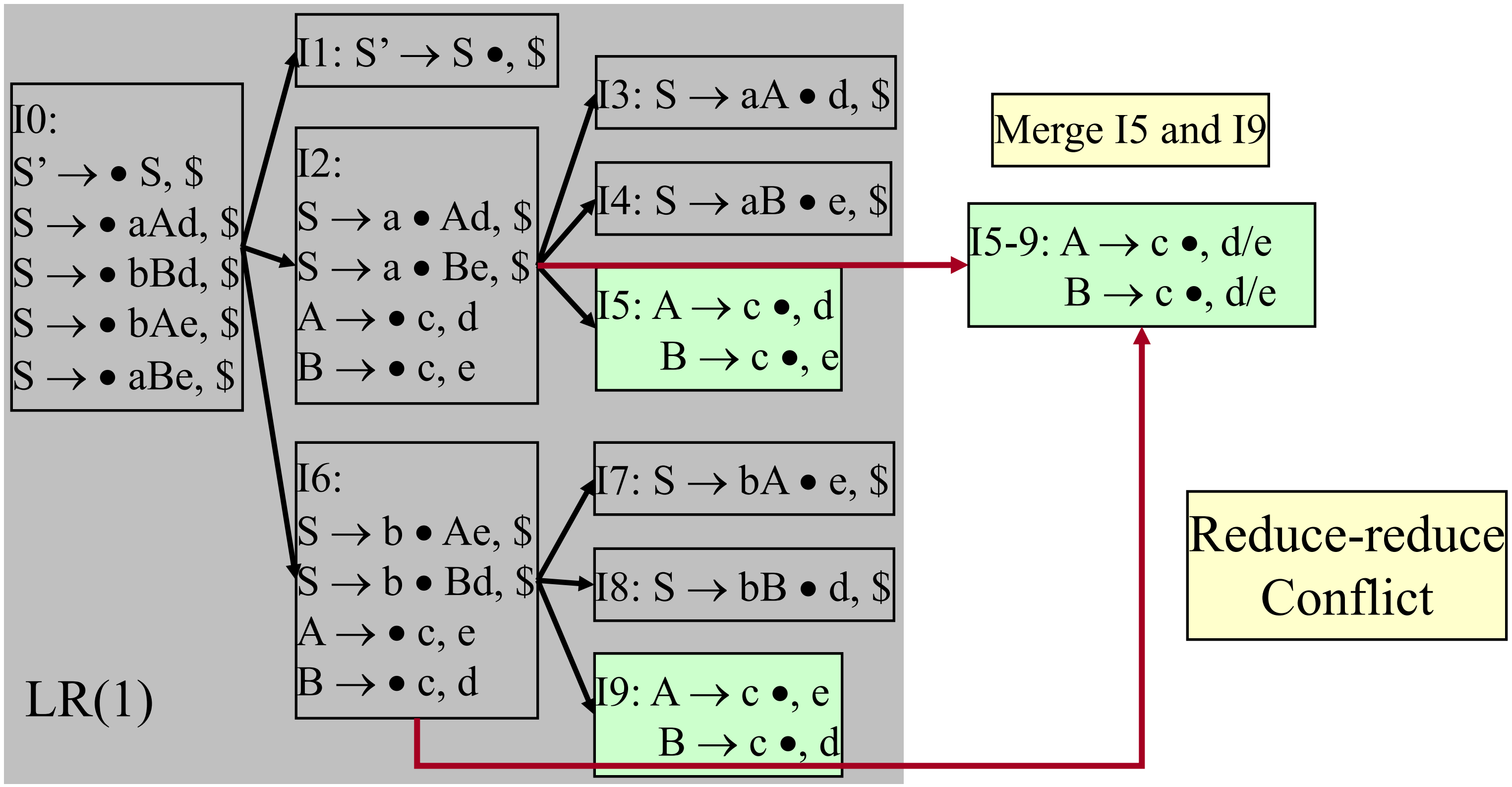
- Can merging states introduce conflicts?
  - Cannot introduce shift-reduce conflict
  - May introduce reduce-reduce conflict
- Cannot introduce shift-reduce conflict?
  - Assume: two LR states I1, I2 are merged into an LALR state I
  - If conflict, I must have items
    - $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \beta \bullet a\delta, b]$
      - In fact, α and β have to be the same, otherwise, they won't come to the same state
    - If they are from different states, they are different core items, cannot be merged into I
    - If I1 has $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \alpha \bullet b\delta, c]$ and I2 has $[A \rightarrow \alpha \bullet, d]$ and $[B \rightarrow \alpha \bullet b\delta, e]$
      - To have a conflict, we should have b = d or b = a, shift-reduce conflicts were there in I1 and I2 already!

LALR Parsing

- Introducing reduce-reduce conflict?

  $S \rightarrow aAd \mid bBd \mid bAe \mid aBe$

  $A \rightarrow c$ $\qquad\qquad B \rightarrow c$



LALR Parsing

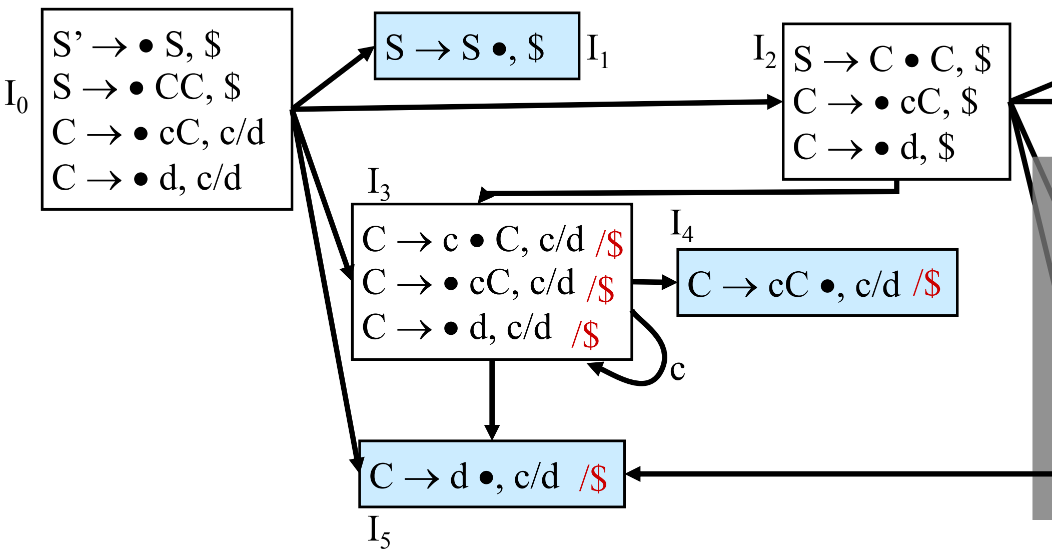
❖ Another LALR example

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

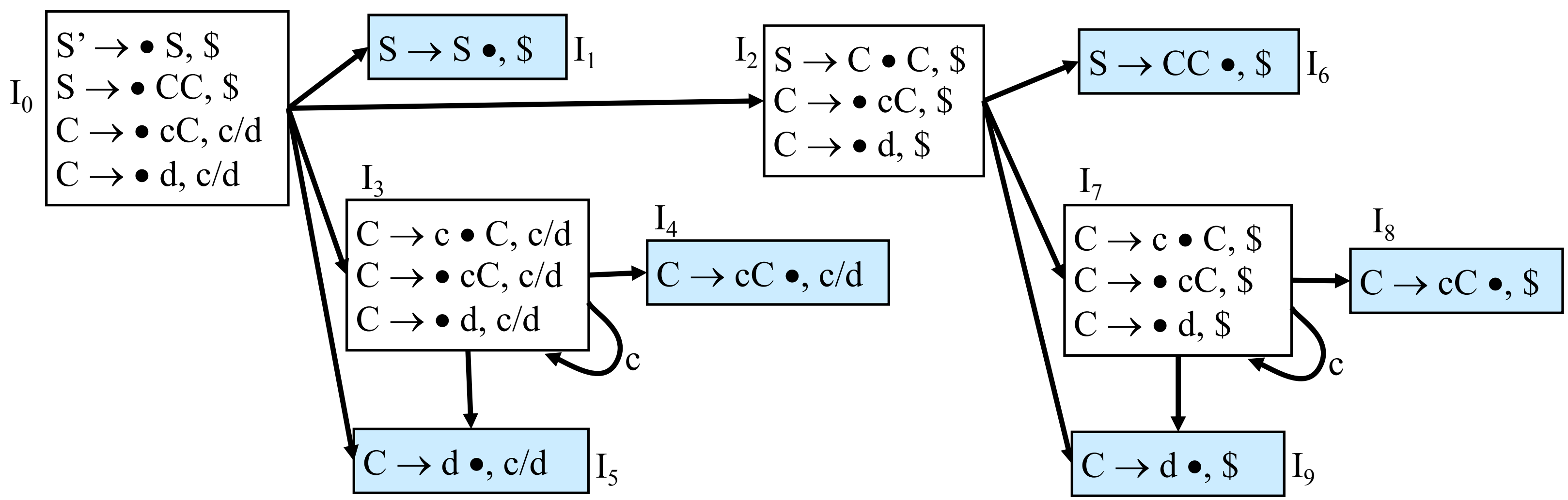
First(C) = {c, d}
First(S) = {c, d}
Follow(S) = {$}
Follow(C) = {c,d,$}



	c	d	$	S	C
0	3	5		1	2
1			Acc		
2	3	5			6
3	3	5			4
4	$C \rightarrow cC$	$C \rightarrow cC$	$C \rightarrow cC$		
5	$C \rightarrow d$	$C \rightarrow d$	$C \rightarrow d$		
6			$S \rightarrow CC$		

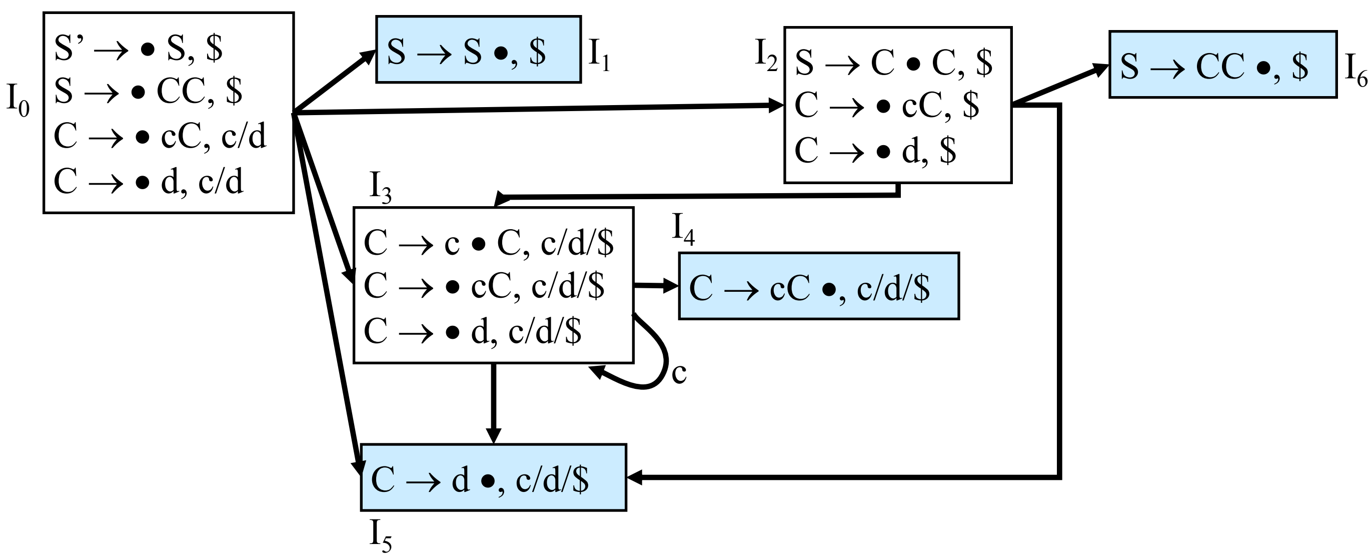
LALR Parsing

- Delay error detection?
  - $S \rightarrow CC$, $C \rightarrow cC$, $C \rightarrow d$
  - Parse string ccd$
- LR stack
  - 0c3c3d5, seeing \$ $\Rightarrow$ reduce using $C \rightarrow d$ only if seeing {c, d}, not \$ $\Rightarrow$ error



LALR Parsing

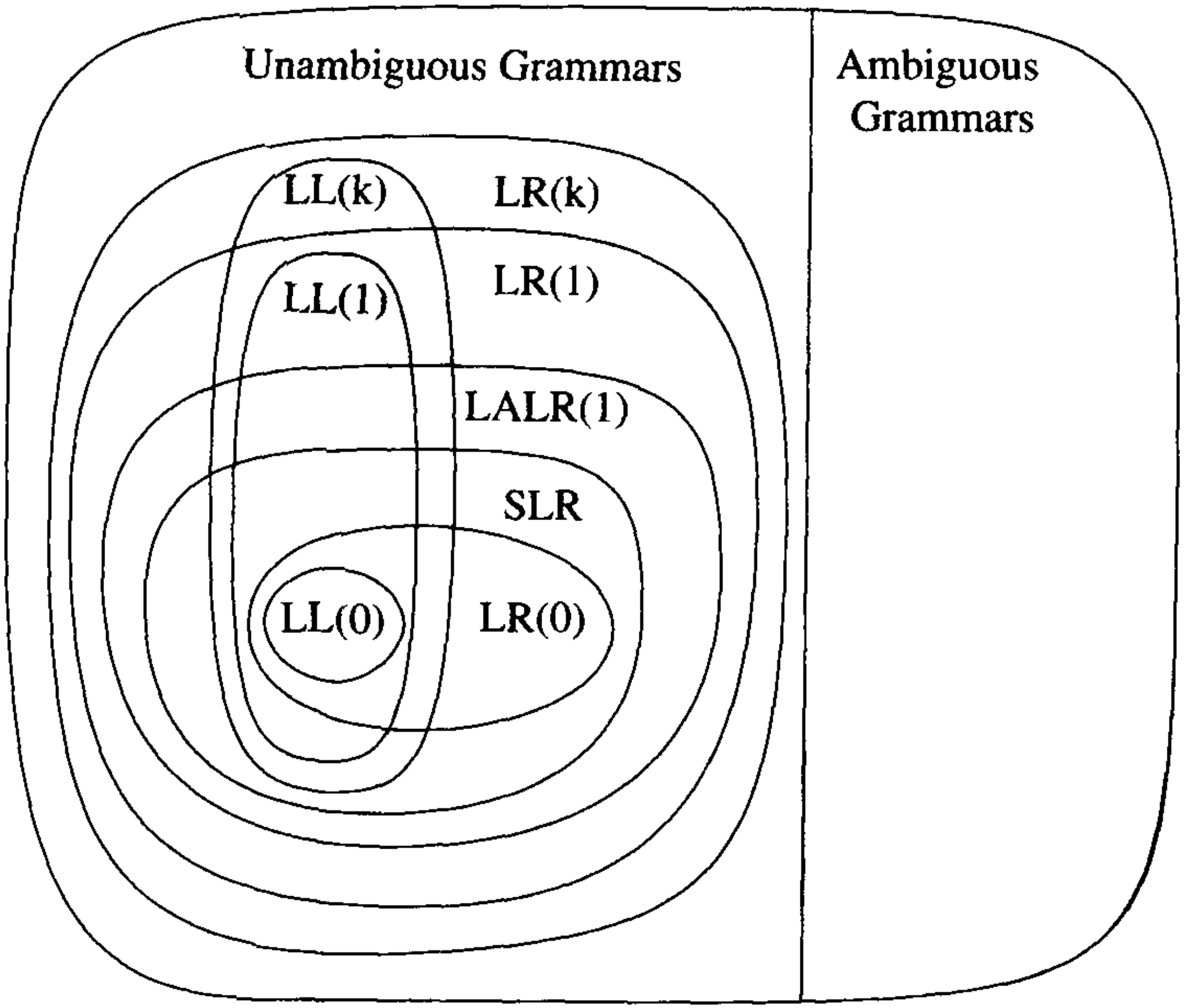
- Delay error detection?
  - LALR stack
    - 0c3c3d5, seeing \$ ⇒ reduce using C → d, goto 4 (0c3c3C4)
    - 0c3c3C4, seeing \$ ⇒ Reduce by C → cC, goto 4 (0c3C4)
    - 0c3C4, seeing \$ ⇒ Reduce by C → cC, goto 2 (0C2)
    - 0C2, seeing \$ ⇒ error, only allow seeing c, d, C



LALR Parsing

- LALR
  - Can also be constructed using SLR procedure
  - But add lookahead symbols
- SLR, LR, LALR
  - LR is most powerful and SLR is least powerful
  - LALR(1) is most commonly used
    - All reasonable languages are LALR(1)
    - Has the same number of states as SLR(1)

Grammar Class Hierarchy



Bottom-up Parsing -- Summary

- Read textbook Sections 4.5-4.6
- Bottom-up Parsing
  - Handle and viable prefix
  - SLR parsing
    - SLR(1) = LR(0)
    - SLR(k)
  - Canonical LR Parsing
    - LR(1)
    - LR(k)
  - LALR