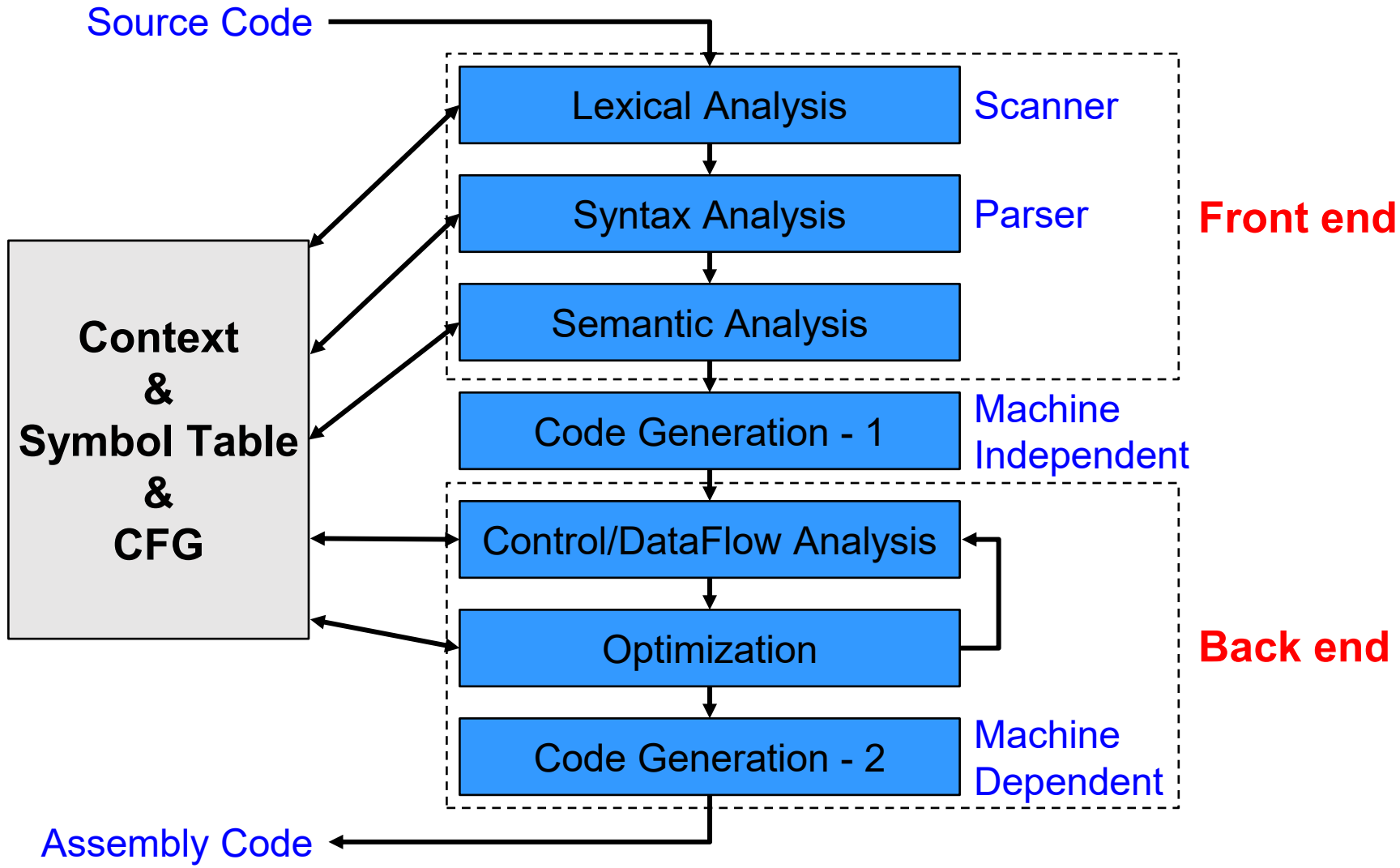
4. Syntax Analysis - 2

2025 Fall

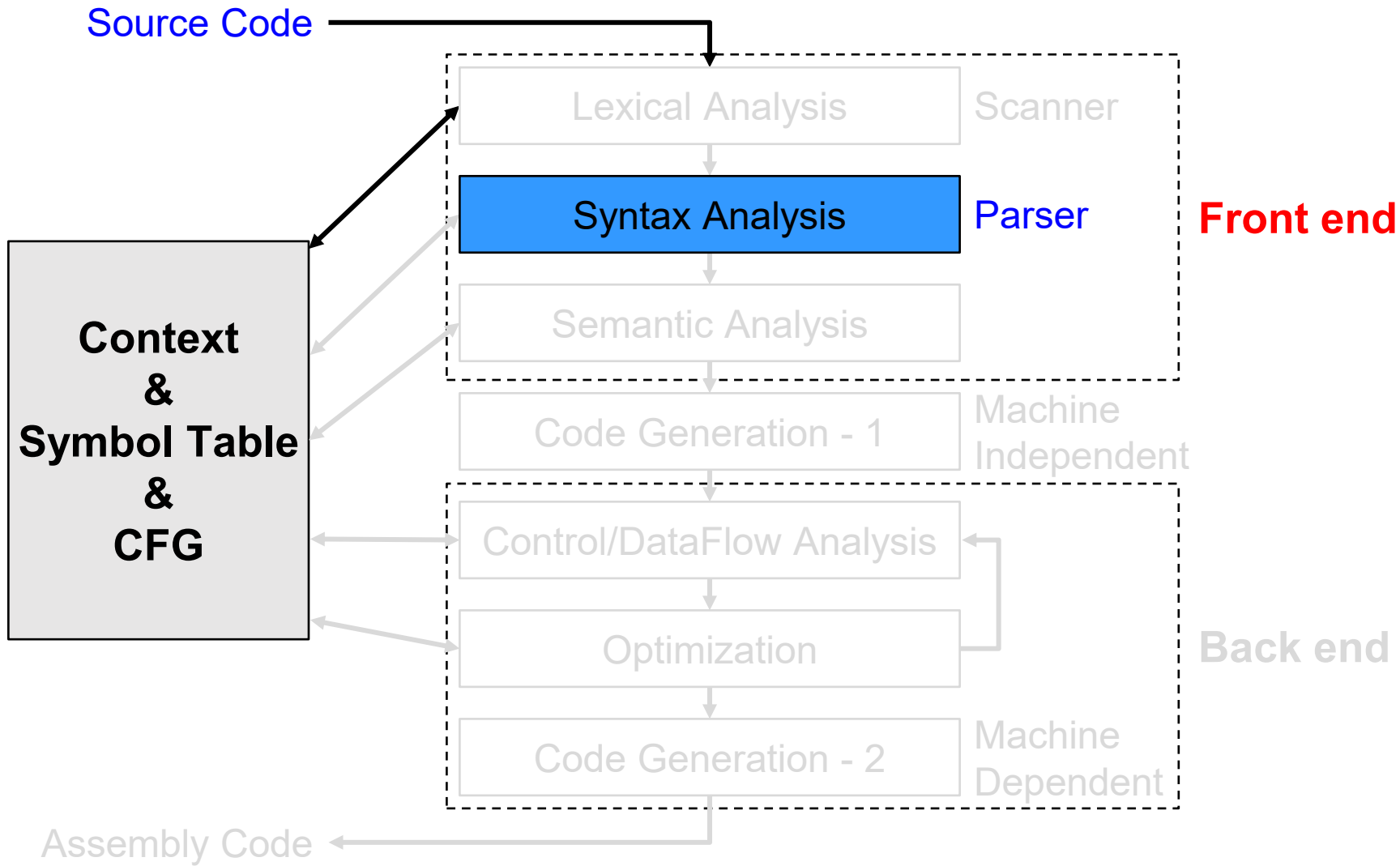
Hunjun Lee

Hanyang University

Syntax Analysis



Syntax Analysis



Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid strings of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

Review: Context-Free Grammars (CFG)

- **Terminals are the tokens: keywords, symbols, ID, NUM …**
- **Non-terminals**
  - program
  - var / expression (arithmetic, logical operations)
  - return / iteration / if statement
- **Start Symbol (S): “program” in compiler**
- **Productions**
  - type-specifier ➔ int | float | bool …
  - var-declaration ➔ type-specifier ID ; | type-specifier ID [NUM] ;
  - function-declaration ➔ type-specifier ID (params) compound-stmt

Review: More Specifically …
➔ To Be Covered in the Project

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* **ID ;** | *type-specifier* **ID [NUM] ;**
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier* **ID (** *params* **)** *compound-stmt*
7. *params* → *param-list* | **void**
8. *param-list* → *param-list* **,** *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID [ ]**
10. *compound-stmt* → **{** *local-declarations statement-list* **}**
11. *local-declarations* → *local-declarations var-declarations* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* **;** | **;**

Review: More Specifically …
➔ To Be Covered in the Project

15. *selection-stmt* → **if** **(** *expression* **)** *statement* | **if** **(** *expression* **)** *statement* **else** *statement*
16. *iteration-stmt* → **while** **(** *expression* **)** *statement*
17. *return-stmt* → **return** **;** | **return** *expression* **;**
18. *expression* → *var* **=** *expression* | *simple-expression*
19. *var* → **ID** | **ID** **[** *expression* **]**
20. *simple-expression* → *additive-expression relop additive-expression* | *additive-expression*
21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → **+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* → ***** | **/**
26. *factor* → **(** *expression* **)** | *var* | *call* | **NUM**
27. *call* → **ID** **(** *args* **)**
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* **,** *expression* | *expression*

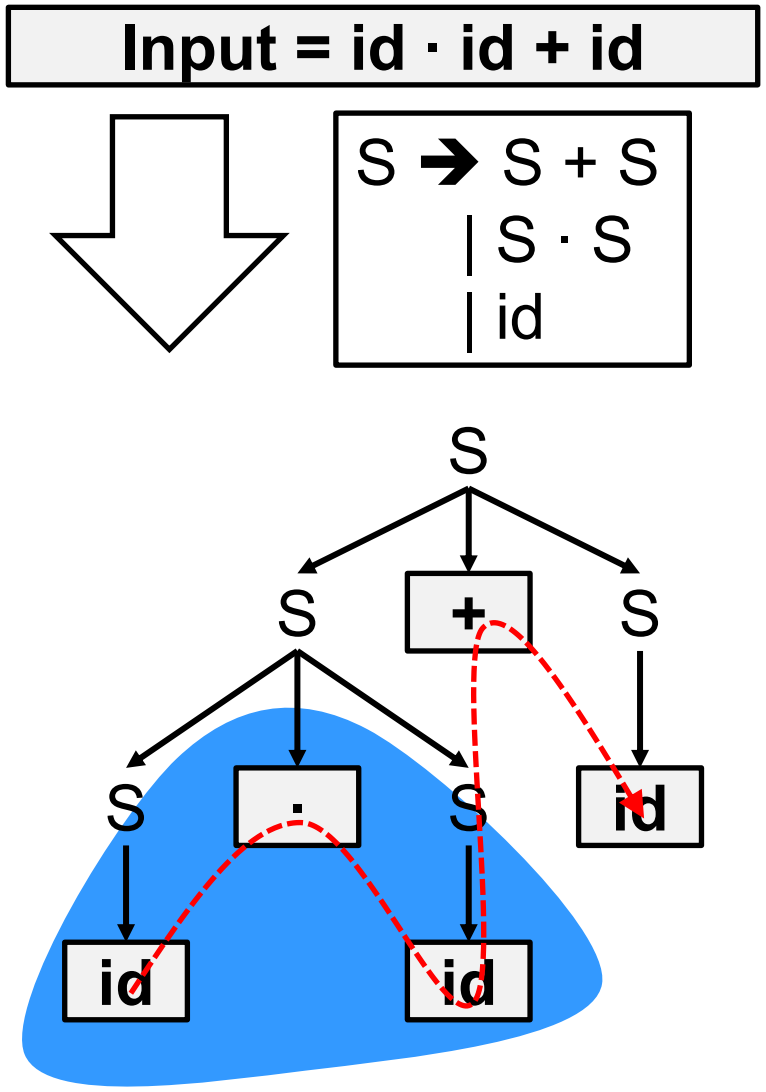
Specification, Recognition, and Automation in Parser

- Specification: how to specify valid sequence of tokens?
  - Context-free grammars (CFG)

- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)

- Automation: how to generate parse tree from CFG?
  - Top-down and bottom-up parsing

Review: Parse Tree

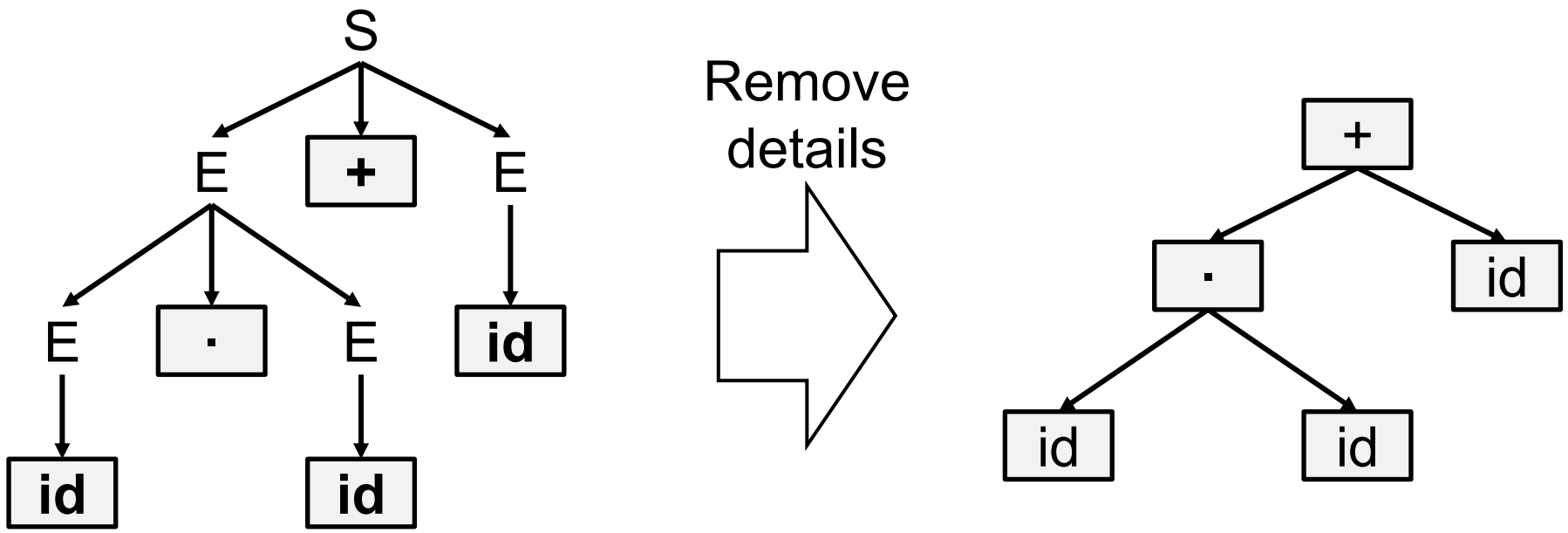


Binds more tightly (· > +)

- **Tree representation of the derivation**
- **A parse tree has**
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- **An in-order traversal of the leaves is the original input**
- **Shows the order of operations**
  - Ex) Operation order, association …

Review: Abstract Syntax Tree (AST)

- **AST discards unneeded information for syntax analysis (Removes non-terminals)**



Parse Tree **Abstract Syntax Tree**

Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid strings of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

Review: Top-Down Parsing

- **Construct a leftmost derivation of string while reading the token stream**

S ➔ E + S | E
E ➔ num | (S)

CFG

Partly-derived String	parsed part unparsed part
➔ E + S	(1+2+(3+4))+5
➔ (S) + S	(1+2+(3+4))+5
➔ (E+S)+S	(1+2+(3+4))+5
➔ (1+S)+S	(1+2+(3+4))+5
➔ (1+E+S)+S	(1+2+(3+4))+5
➔ (1+2+S)+S	(1+2+(3+4))+5
➔ (1+2+E)+S	(1+2+(3+4))+5
➔ (1+2+(S))+S	(1+2+(3+4))+5
➔ ...	

Derivation Order

Review: Making a Grammar LL(1)

- **LL(1) eliminates the multiple matches in top-down parsing**
  - Left-to-right scanning, Leftmost derivation, and 1 look-ahead symbol
    - We have to read one token to determine which production to apply

- **Left factoring: factor common prefix and add a non-terminal S'**
  - S ➔ E + S | E is converted as follows
  - S ➔ ES' and S' ➔ +S | ε

- **Convert left recursion to right recursion**
  - Example left recursion: S ➔ Sa (infinite derivation)
  - We use the term S ➔* Sa (for some a) to indicate the left recursion

Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid strings of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

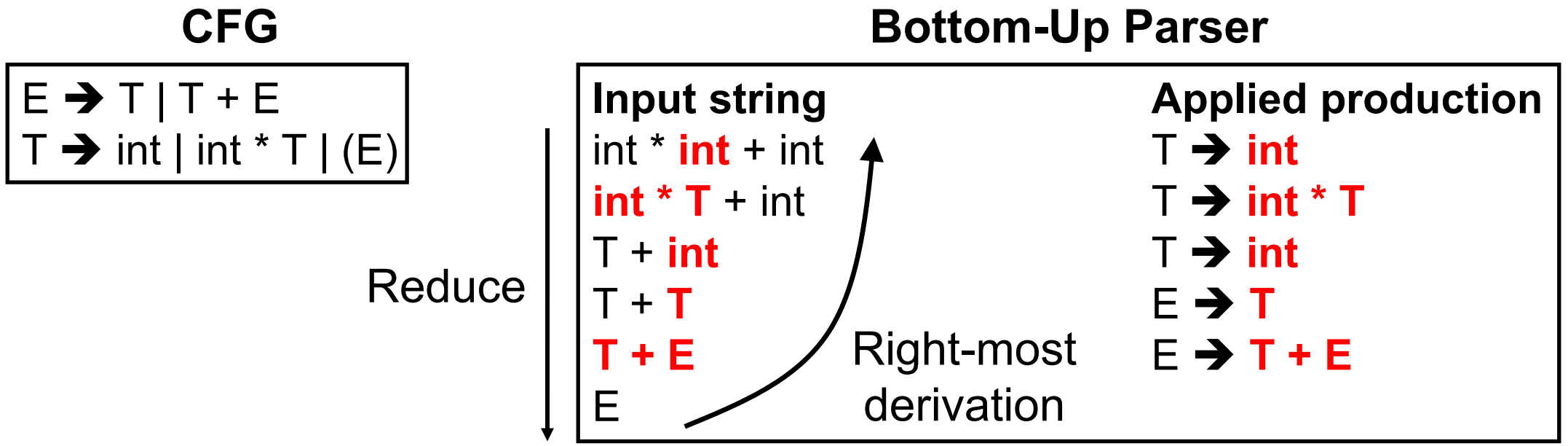
Bottom-Up Parsing - 1

- **Bottom-up parsing is more efficient than Top-down parsing using LR grammars**
  - Construct right-most derivation of program
  - **L**eft-recursive grammars, and **R**ight-most derivation

- **Rely on shift-reduce parsers**
  - The parser either shifts to the next input token or reduce the symbols (we'll get to this later on)

Bottom-Up Parsing - 2

- **Right-most derivation requires reducing symbols to the start symbol by "inverting" productions**
  - Match RHS of production and replace it by LHS



Shift-Reduce Parsing

- **Bottom-up parser traces a rightmost derivation in reverse**
  - We should scan the input terminals in a left-to-right manner

- **We can split a string into two parts: sequence of symbols to reduce (stack) + remaining tokens (input)**

- **Shift reduce parsing requires two actions**
  - Reduce using symbols in the stack **(Reduce)**
  - Scan the left-most token in the input and move it to the stack **(Shift)**

Shift-Reduce Parsing Example

CFG

E ➔ T | T + E
T ➔ int | int * T | (E)

Shift-Reduce Parser

Stack		Input string	Applied production
	|	int * int + int	shift
int	|	* int + int	shift
int *	|	int + int	shift
int * int	|	+ int	reduce T ➔ int
int * T	|	+ int	reduce T ➔ int * T
T	|	+ int	shift
T +	|	int	shift
T + int	|		reduce T ➔ int
T + T	|		reduce E ➔ T
T + E	|		reduce E ➔ T + E
E	|		

Key Challenge: Action Selection Problem

- **Which action should we take?**
  - Should we shift?
  - Should we reduce? If so, which production should we applied?

- **There are conflicts**
  - For a given state (stack + input), there can be multiple possible actions
    - Shift-reduce conflict: can both shift and reduce
    - Reduce-reduce conflict: can apply two different productions

E ➔ T | T + E T ➔ int | int * T | (E)

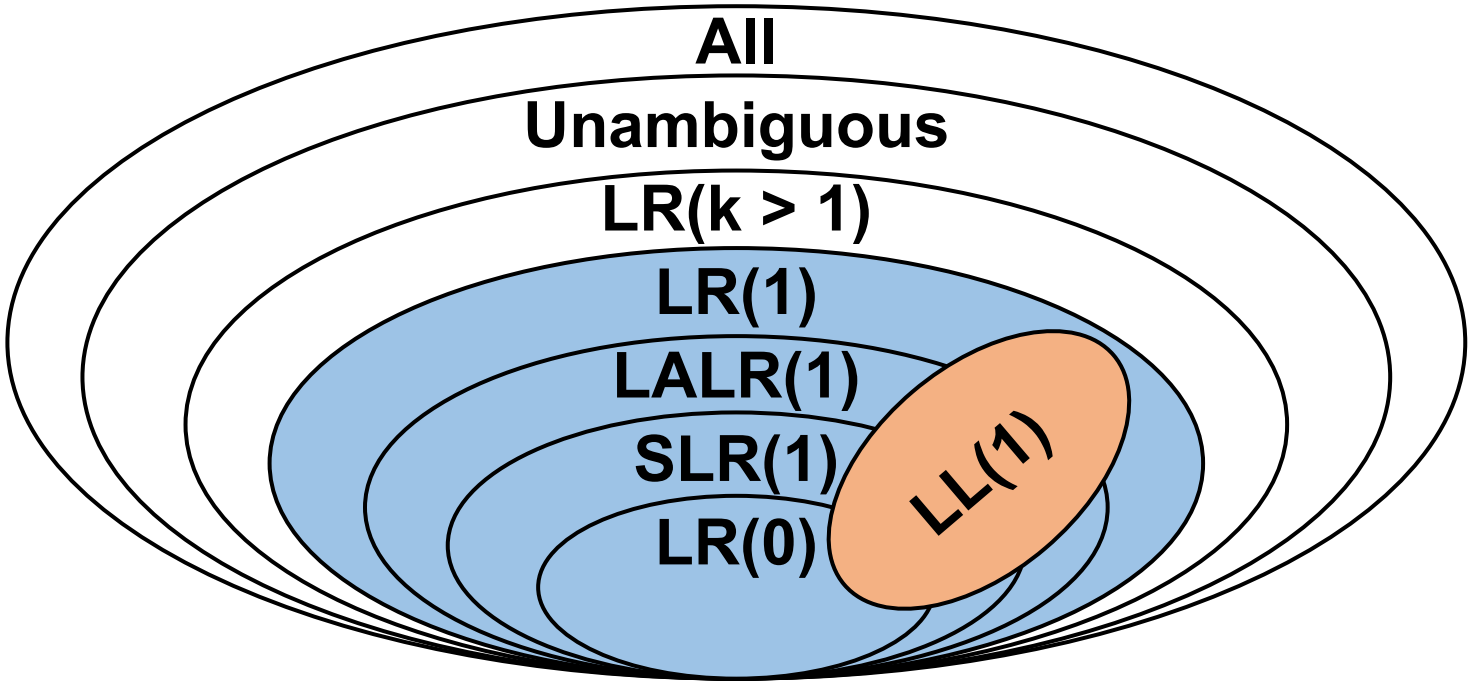
Stack	**Input String**
int |	* int + int

Lookahead?

Shift vs. Reduce

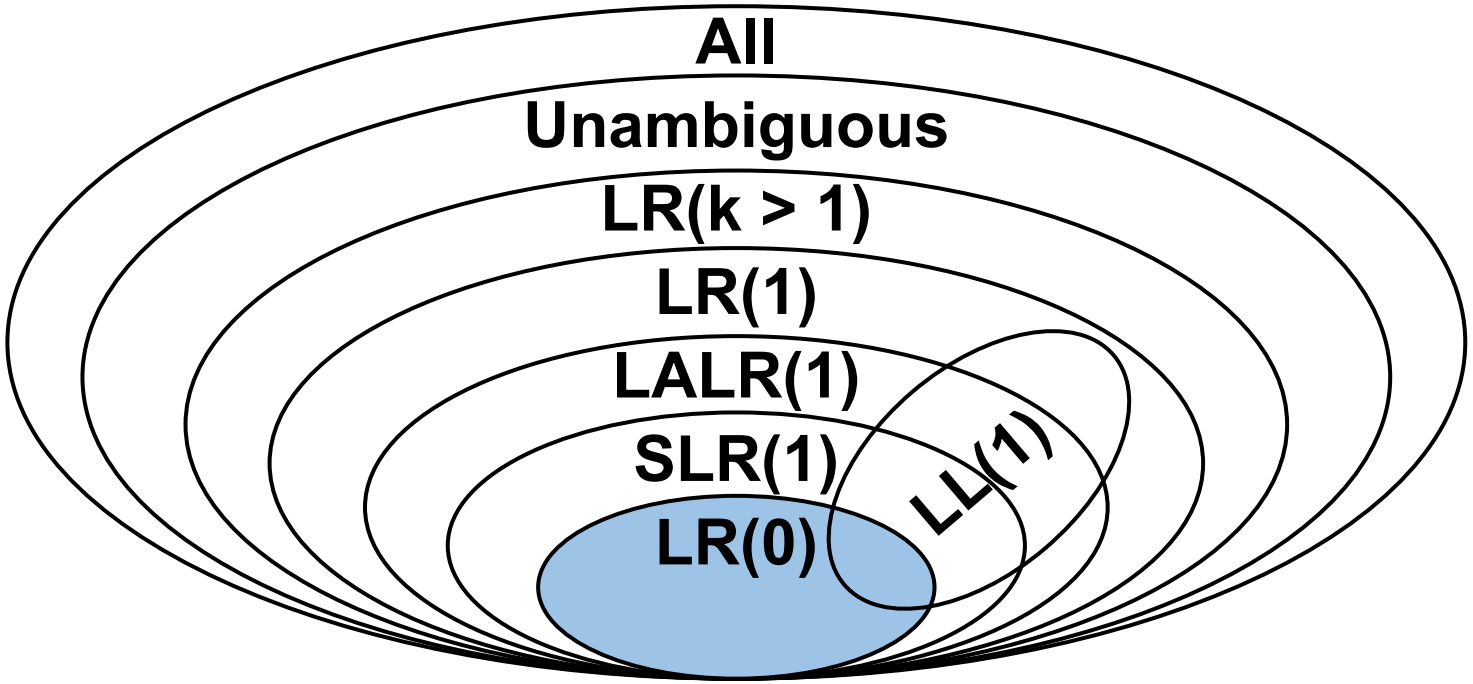
LR-Style Grammars

- LR(k): left-to-right scanning, right-most derivation, and k symbol lookahead
- There are variations of the LR(k) (i.e., LALR, SLR)



LR(0) Grammars

- **LR(0) indicates grammars that can determine actions w/o any lookahead**
  - There are no reduce-reduce and shift-reduce conflicts when using only the symbols in the stack

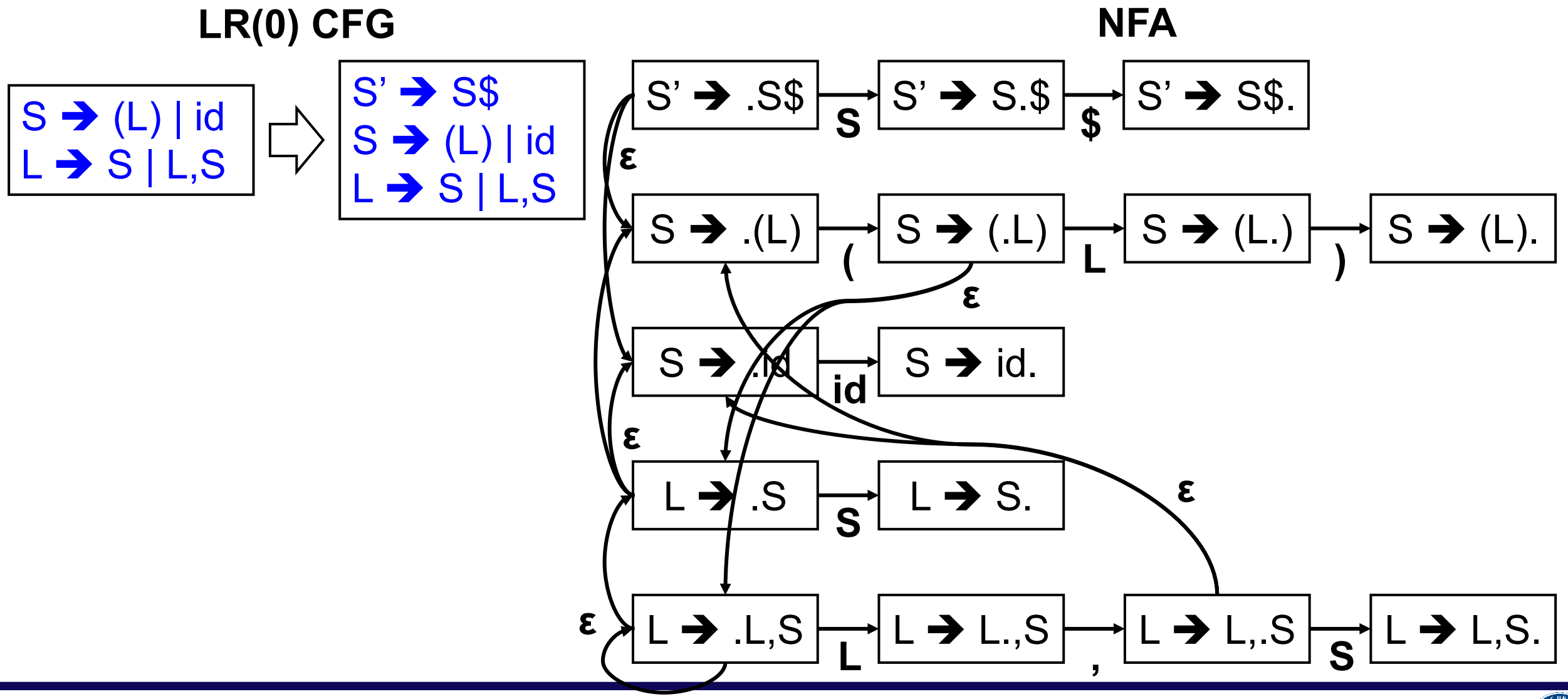


NFA Representation

- **We can represent shift-reduce parsing using an NFA**
  - The states are items, which are productions with a separator "." on RHS
    - Ex) T ➔ (E) has four items: **T ➔ .(E)**, **T ➔ (.E)**, **T ➔ (E.)**, and **T ➔ (E).**
    - T ➔ (.E) indicates that "(" is on the stack and we expect to see "E" in the upcoming input
  - We have an additional dummy production S' ➔ S$ for a start and end state

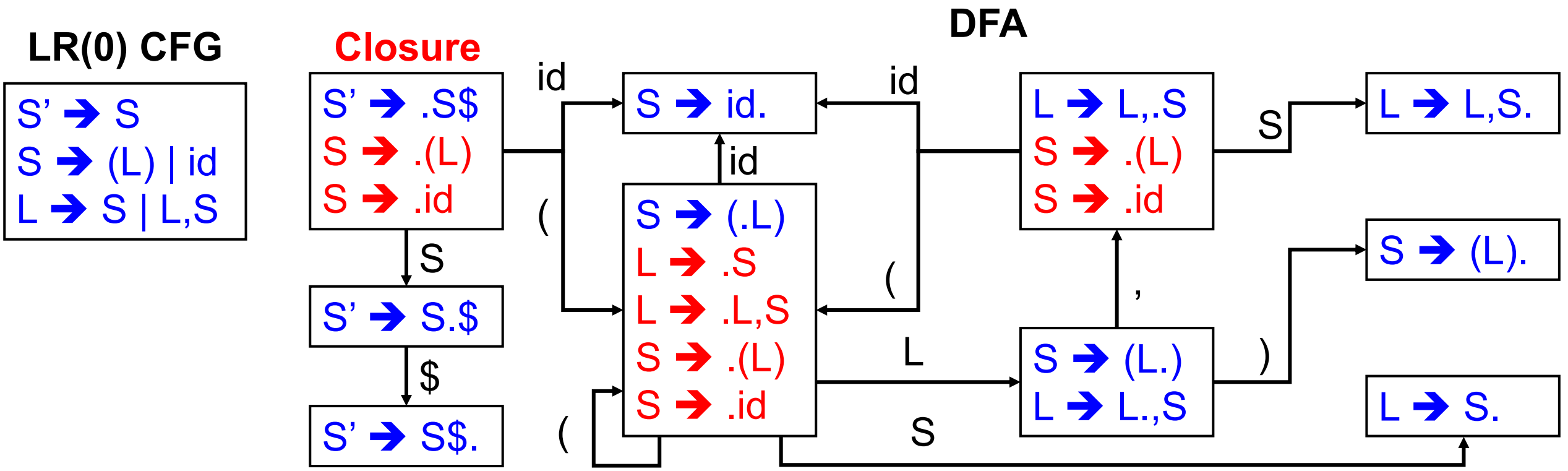
- **There are two types of transitions between the states**
  - Shift transition: transition by the shift actions
    - There is a transition between State: {T ➔ (.E)} and State: {T ➔ (E.)} if the left most symbol of the input string is "E"
  - ε-transition: we see another production that stems from the expected symbol
    - State: {T ➔ (.E)} becomes State: {E ➔ .T} and State: {T ➔ .T + E}

LR(0) NFA Representation Example



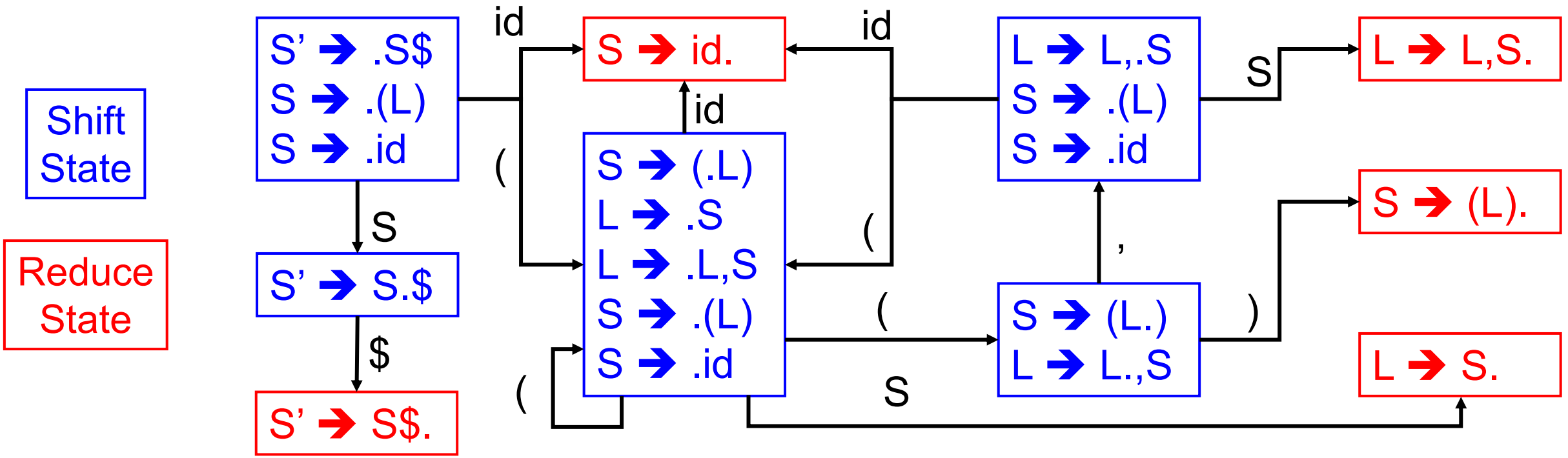
LR(0) DFA Representation

- **Recall NFA to DFA: Group NFA states with ε-closure (closure)!**
  - For Closure(s) which keeps {A ➔ α.Bβ}, and {B ➔ γ} is a production, add {B ➔ .γ} to the production



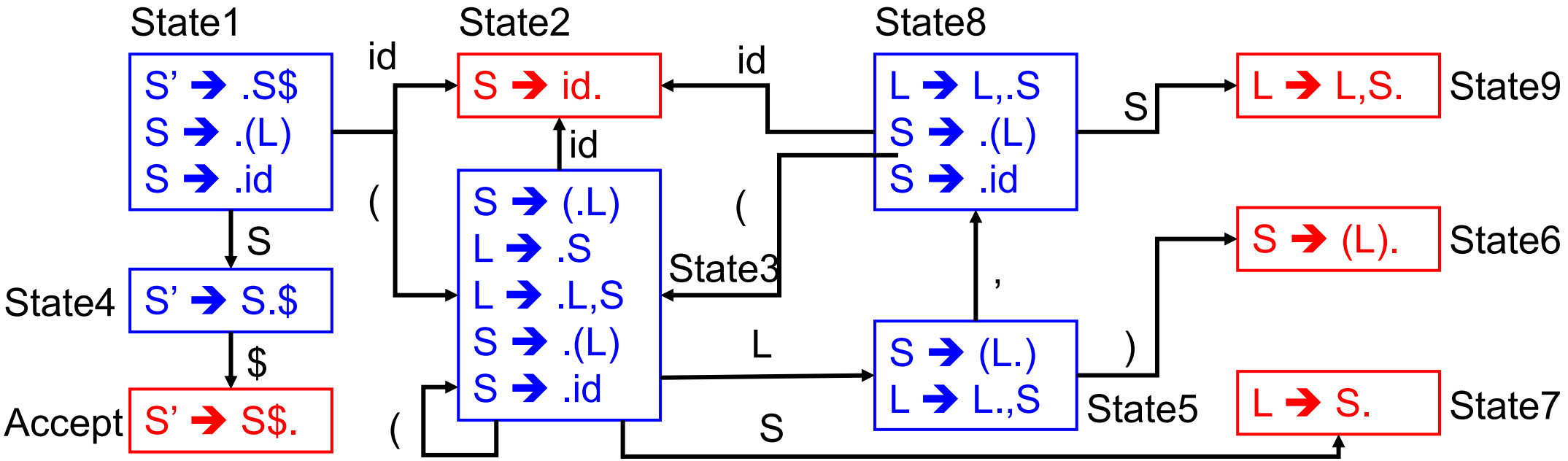
LR(0) Parsing Using DFA

- **Using the DFA, we determine whether to shift or reduce**
  - We use symbols in the stack to traverse the state
  - Use the destination state to determine whether to shift or reduce



LR(0) Parsing Using DFA Example

stack	input	action
	((a),b)	
(	(a),b)	
((	a),b)	
((a	),b)	
((S	),b)	

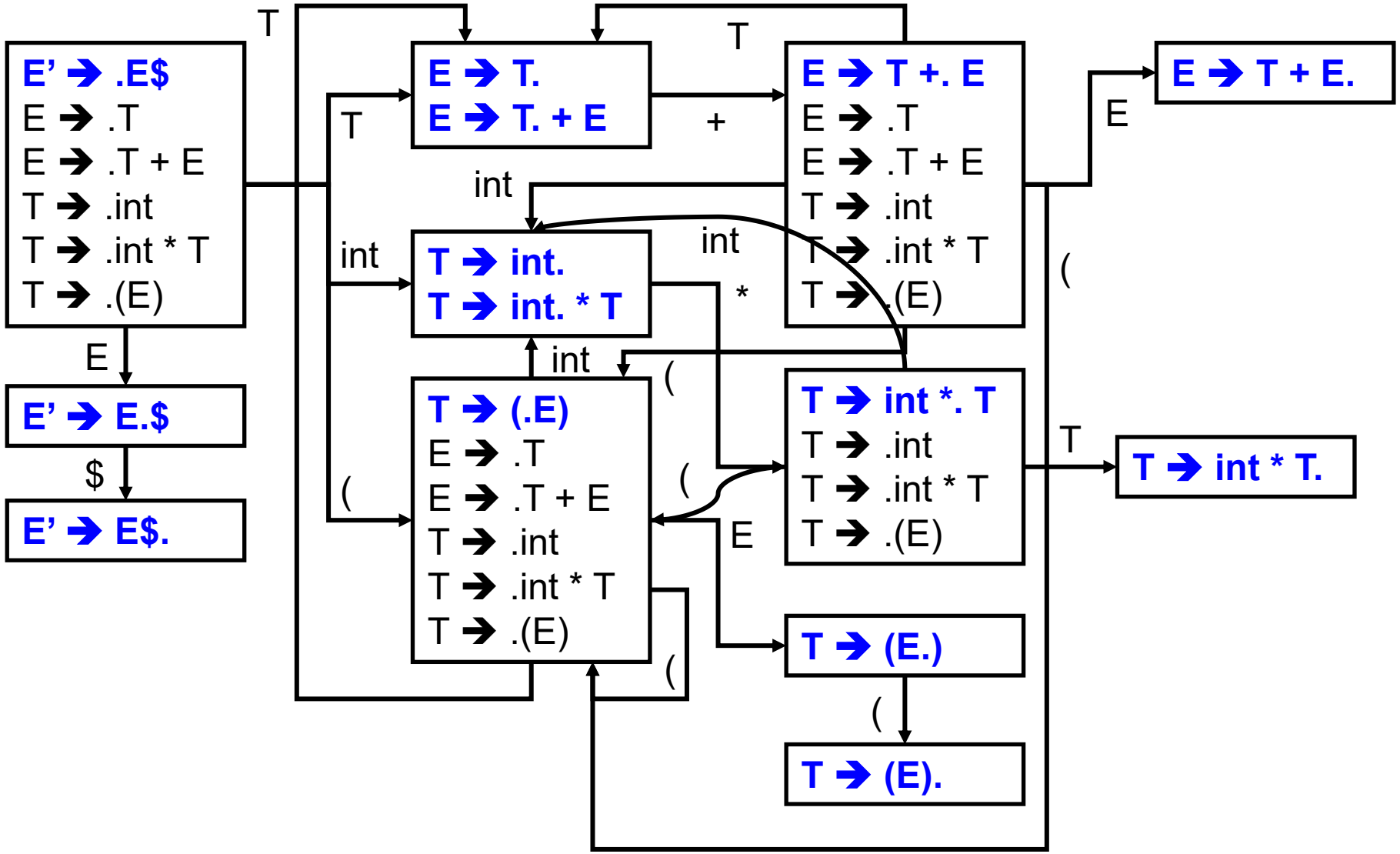


Class Exercise

- **Generate the DFA for the following grammar**
  - You will need a large space!

E ➔ T | T + E
T ➔ int | int * T | (E)

E ➔ T | T + E
T ➔ int | int * T | (E)

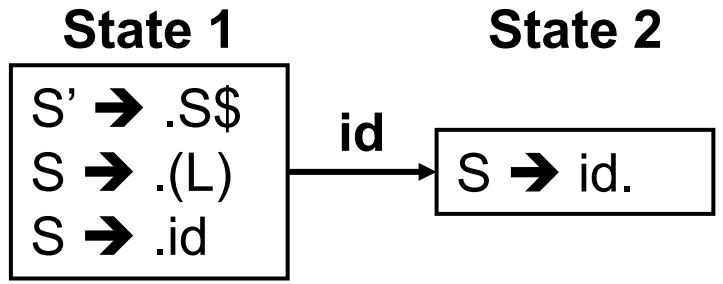


DFA Traversal Implementation

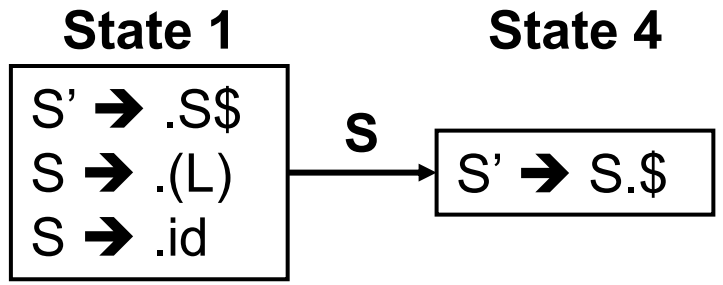
- **Store "the states along with the symbols" in the stack**
  - <symbol, state> and start using the top <symbol, state> on the stack
- **There are two different types of tables**
  - goto: determine the next state using the top state and an input non-terminal
  - action: determine the action using the top state and an input terminal
- **There are four different actions**
  - Shift x: push<a, x> on the stack (a is the current input and x is a state)
  - Reduce x ➔ α: pop α from the stack and push <x, goto[curr_state, x]>
  - Accept (S' ➔ S$.) / Error (no valid action available)

LR Parsing Table Generation

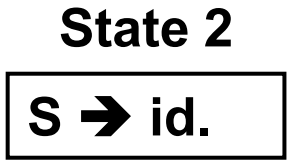
- **We can simplify the DFA using a parsing table**
- **DFA states are converted to index of each rows**
- **For transition S_{old} ➔ S_{new} on terminal C:**
  - Table[S_{old}, C] += Shift(S_{new})



- **For transition S_{old} ➔ S_{new} on non-terminal N:**
  - Table[S_{old}, N] += Goto(S_{new})



- **If S is a reduction state X ➔ β then:**
  - Table[S, *] += Reduce(X ➔ β)



LR Parsing Table

- **We can simplify the DFA using a parsing table**

	(	)	id	,	$	S	L
State 1	Shift 3		Shift 2			Goto 4	
State 2	S ➔ id						
State 3	Shift 3		Shift 2			Goto 7	Goto 5
State 4					Accept		
State 5		Shift 6		Shift 8			
State 6	S ➔ (L)						
State 7	L ➔ S						
State 8	Shift 3		Shift 2			Goto 9	
State 9	L ➔ L,S						

Action Table **Goto Table**

Recall: Action Selection Problem

- **Which action should we take?**
  - Should we shift?
  - Should we reduce? If so, which production should we applied?

- **There are conflicts**
  - For a given state (stack + input), there can be multiple possible actions
    - Shift-reduce conflict: can both shift and reduce
    - Reduce-reduce conflict: can apply two different productions

E ➔ T | T + E T ➔ int | int * T | (E)

Stack	**Input String**
int |	* int + int

Lookahead?

Shift vs. Reduce

Parsing Implementation

```
// I is an input stream
// j = 0
// DFA state 1 is the start state
stack = <dummy, 1>
repeat
      case action[top_state(stack), I[j]]
            shift X: push <I[j++], X>
            reduce X ➔ A:
                  pop |A| pairs
                  push <X, goto[top_state(stack), X]>
            accept: halt normally
            error: halt and report error
end
```

LR Parsing Table

- **We can simplify the DFA using a parsing table**

	(	)	id	,	$	S	L
State 1	Shift 3		Shift 2			Goto 4	
State 2	S ➔ id						
State 3	Shift 3		Shift 2			Goto 7	Goto 5
State 4					Accept		
State 5		Shift 6		Shift 8			
State 6	S ➔ (L)						
State 7	L ➔ S						
State 8	Shift 3		Shift 2			Goto 9	
State 9	L ➔ L,S						

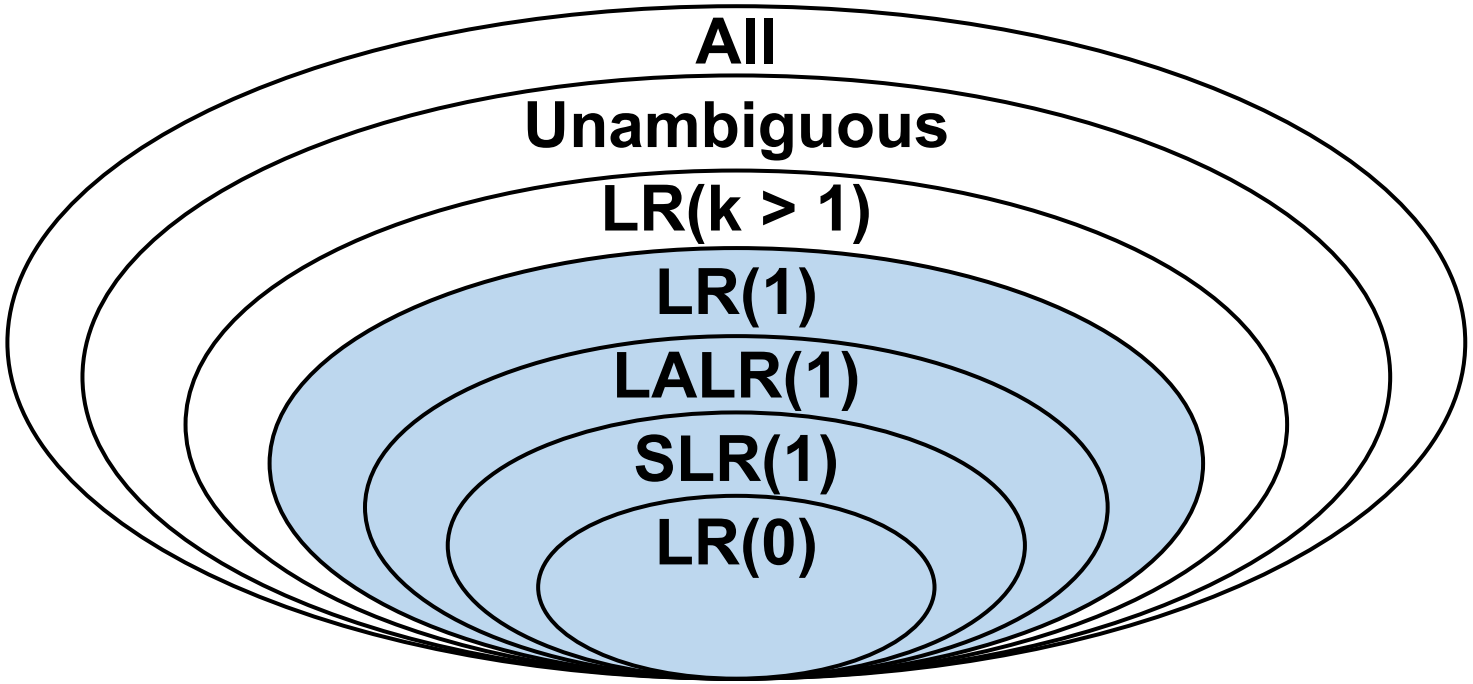
Action Table **Goto Table**

Parsing Using Table Example

stack	input	action
<dummy, **1**>	((a),b)$	
<dummy, 1><(, **3**>	(a),b)$	
<dummy, 1><(, 3><(, **3**>	a),b)$	
<dummy, 1><(, 3><(, 3><a, **2**>	),b)$	
<dummy, 1><(, 3><(, 3><S, **7**>	),b)$	
<dummy, 1><(, 3><(, 3><L, **5**>	),b)$	
<dummy, 1><(, 3><(, 3><L, 5><), **6**>	,b)$	
<dummy, 1><(, 3><S, **7**>	,b)$	
<dummy, 1><(, 3><L, **5**>	,b)$	
<dummy, 1><(, 3><L, 5><,, **8**>	b)$	
<dummy, 1><(, 3><L, 5><,, 8><b, **2**>	)$	
<dummy, 1><(, 3><L, 5><,, 8><S, **9**>	)$	
<dummy, 1><(, 3><L, **5**>	)$	
<dummy, 1><(, 3><L, 5><), **6**>	$	
<dummy, 1><S, **4**>	$	

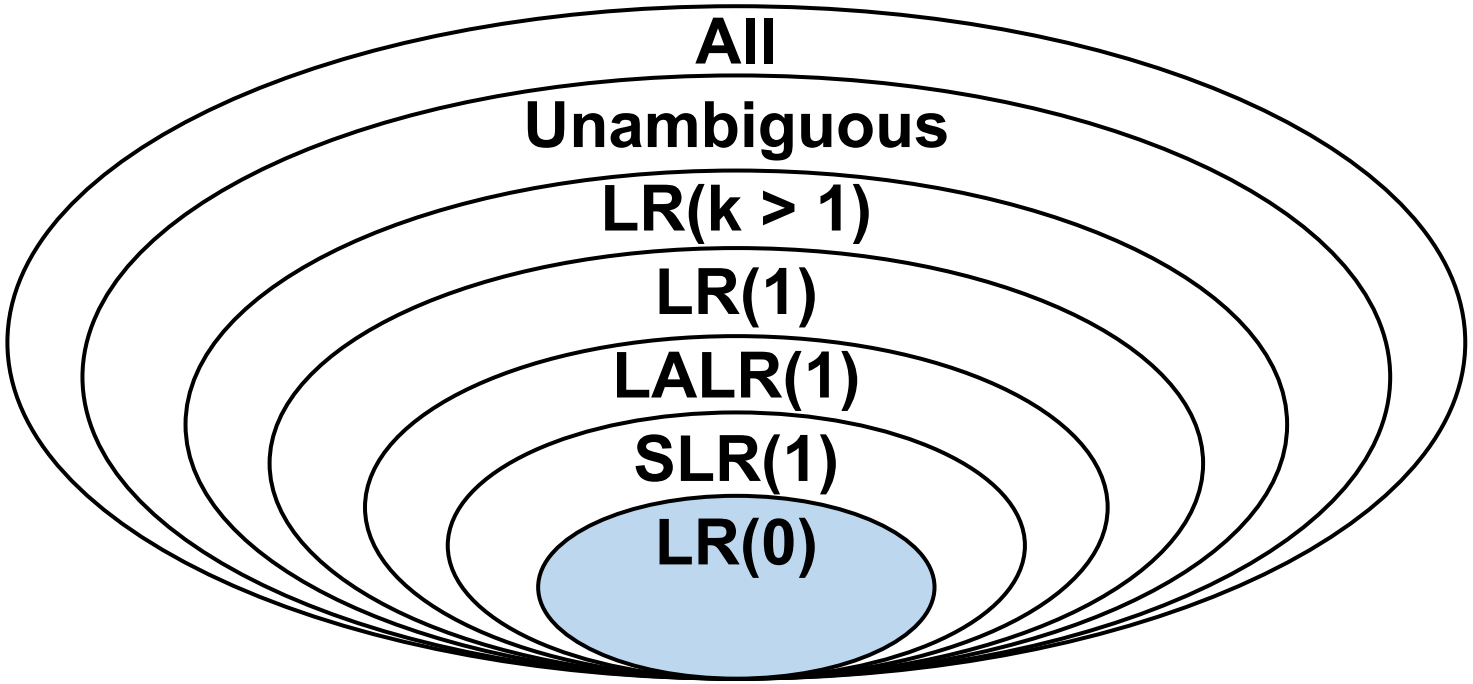
LR-Style Grammars

- LR(k): left-to-right scanning, right-most derivation, and k lookahead
- There are variations of the LR(k) (i.e., LALR, SLR)



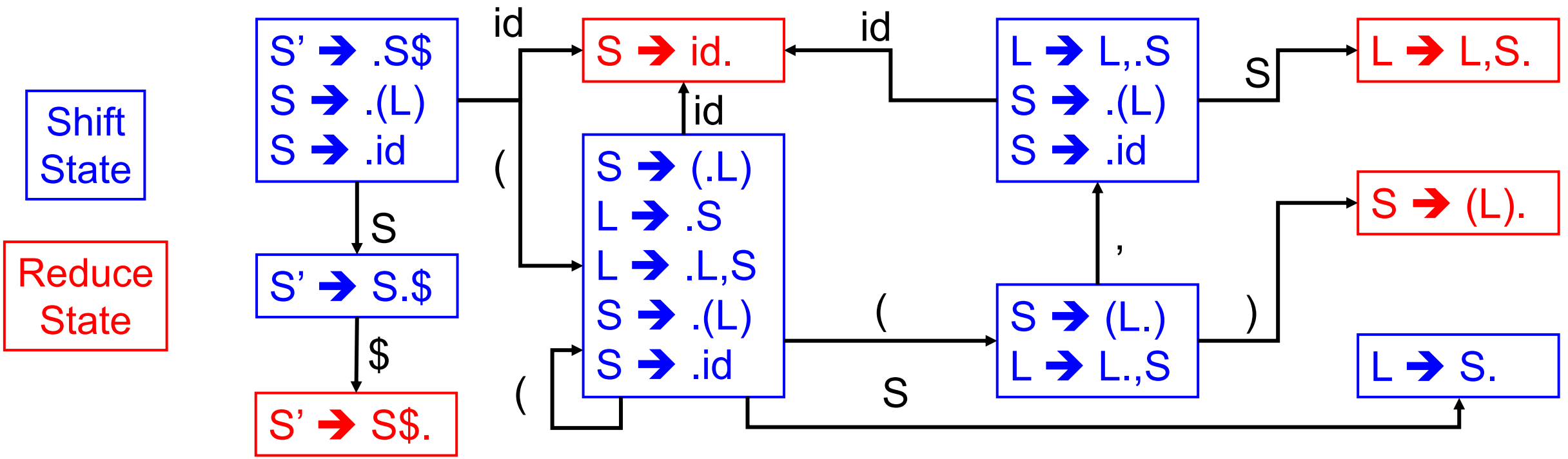
LR(0) Grammars

- **LR(0) assumes no shift-reduce and reduce-reduce conflicts even without lookahead**



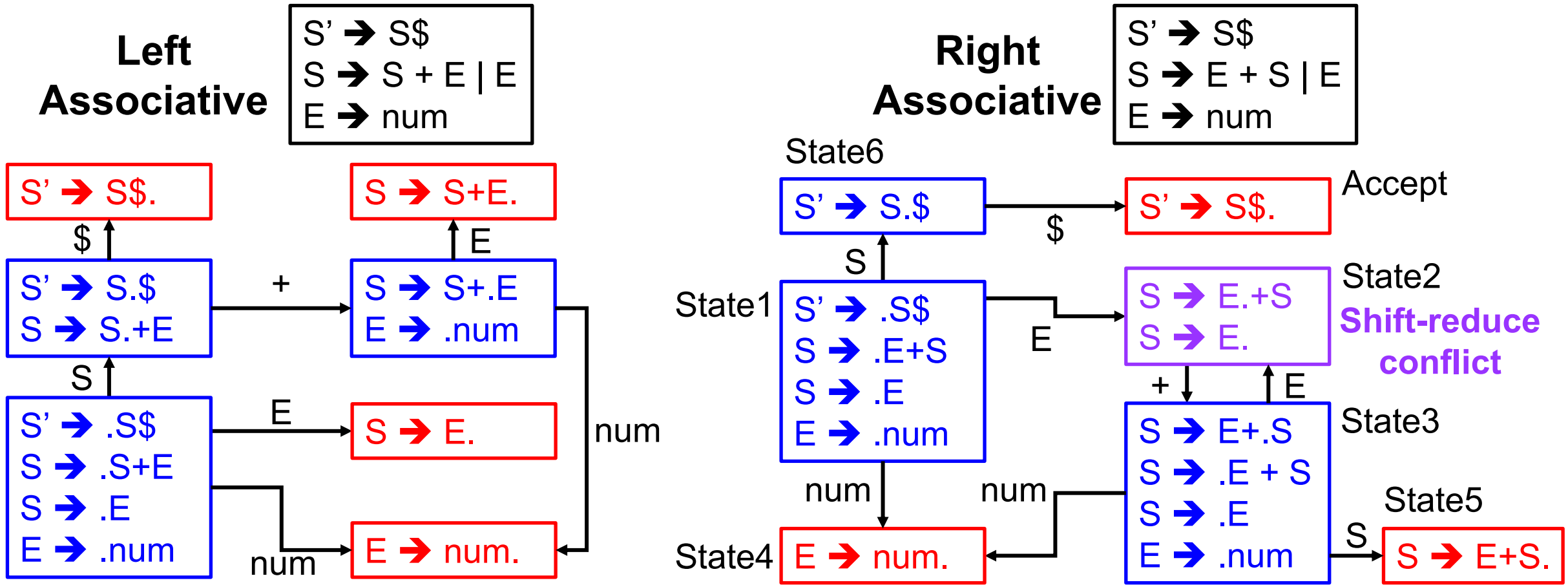
LR(0) Limitations

- **An LR(0) parser only works if states with reduce actions have a single reduction action (no other reduction or shift)**



A Non-LR(0) Grammar

- **Right-associative is not LR(0); there is a shift-reduce conflict!**



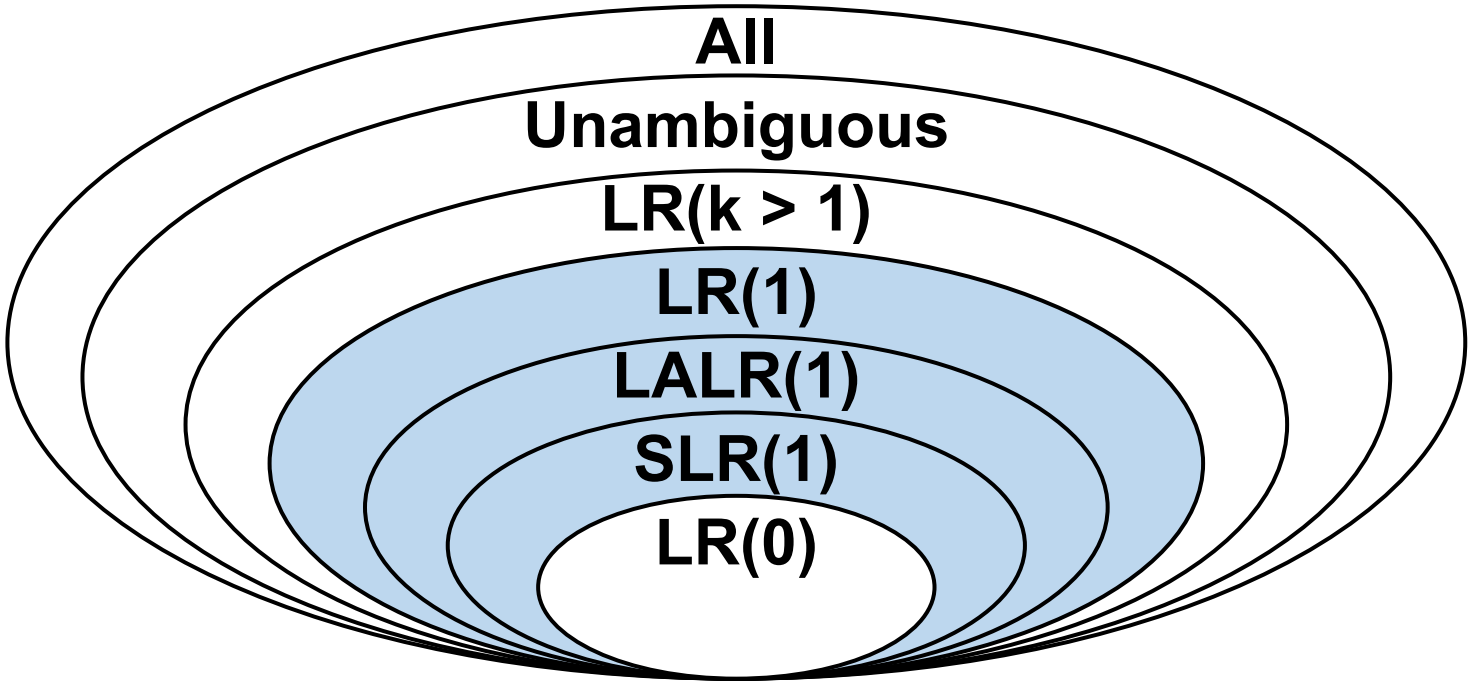
A Non-LR(0) Grammar

- **There are multiple options to fill the parsing table**

	num	+	$	E	S
State 1	Shift 4			Goto 2	Goto 6
State 2	S → E	S → E vs. Shift 3	S → E		
State 3	Shift 4			Goto 2	Goto 5
State 4	E → num	E → num	E → num		

Solve Conflict With Lookahead

- **There are three popular techniques for employing lookahead of a single symbol**
  - Each has a different mean of utilizing the lookahead!



SLR(1) Parsing

- **A simple extension of LR(0)**
  - For each reduction X ➔ β, look at the next symbol γ
  - Apply reduction only if γ is in Follow(X)
    - Ex) αβγ can become αXγ only if γ is in Follow(X)

- **SLR(1) (a.k.a., SLR) parsing table can eliminate conflict**

S' ➔ S$
S ➔ E + S | E
E ➔ num

⇩

Follow(S) = {$}

	num	+	$	E	S
State 1	Shift 4			Goto 2	Goto 6
State 2	~~S ➔ E~~	~~S ➔ E~~ Shift 3	S ➔ E		

SLR Parsing Table

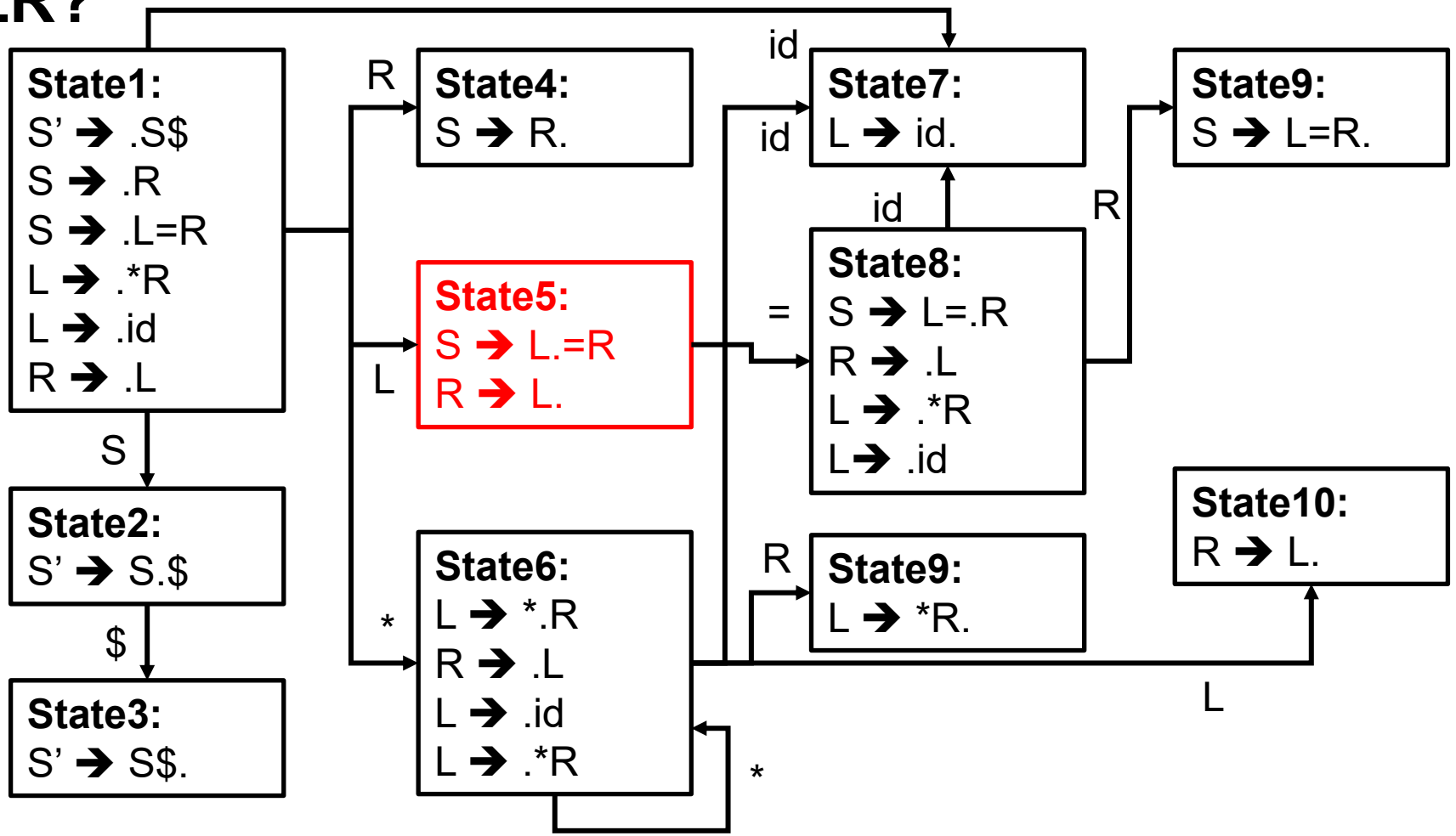
- **Do not fill the entire rows as before**

	num	+	$	E	S
State 1	Shift 4			Goto 2	Goto 6
State 2		Shift 3	S ➔ E		
State 3	Shift 4			Goto 2	Goto 5
State 4		E ➔ num	E ➔ num		
State 5			S ➔ E+S		
State 6			Shift 7		
State 7			accept		

Class Exercise

- **Is this grammar SLR?**
  - S → L = R
  - S → R
  - L → *R
  - L → id
  - R → L

Hint: Check if = follows R



LR(1) Parsing

- **We use more out of the lookahead symbol**
- **LR(1) determines whether to shift or reduce using a lookahead**
- **Uses a more complex state X ➔ α.β , γ**
  - Meaning: α already matched at top of the stack, next expect to see β, followed by γ
  - [X ➔ α.β, {x_1, x_2, …, x_n}] indicates a set of states:
    - X ➔ α.β, x_1
    - …
    - X ➔ α.β, x_n
- **We extend the ε-closure and goto operations**

LR(1) Closure

- **LR(1) closure identification:**
  - Start with Closure(S) = S
  - For each item in S [X ➔ α . B β , λ]
    - Add {B ➔ .γ, First(βλ)}
- **Initialize the state with [S' ➔ .S , $] and apply closure**

CFG

S' ➔ S$
S ➔ E + S | E
E ➔ num

Start State

S' ➔ .S $

Closure

S' ➔ .S	**$**
S ➔ .E+S	First($) → $
S ➔ .E	First($) → $
E ➔ .num	First(+S$) + First($) → +, $

LR(1) Goto

- **LR(1) goto operations:**
  - Given an item in the state I (X ➔ α . B β , λ),
  - Goto/Shift(I, B) = Closure([X ➔ α B . β , λ])

CFG

S' ➔ S$
S ➔ E + S | E
E ➔ num

State I

S ➔ E.+S	$
S ➔ E.	$

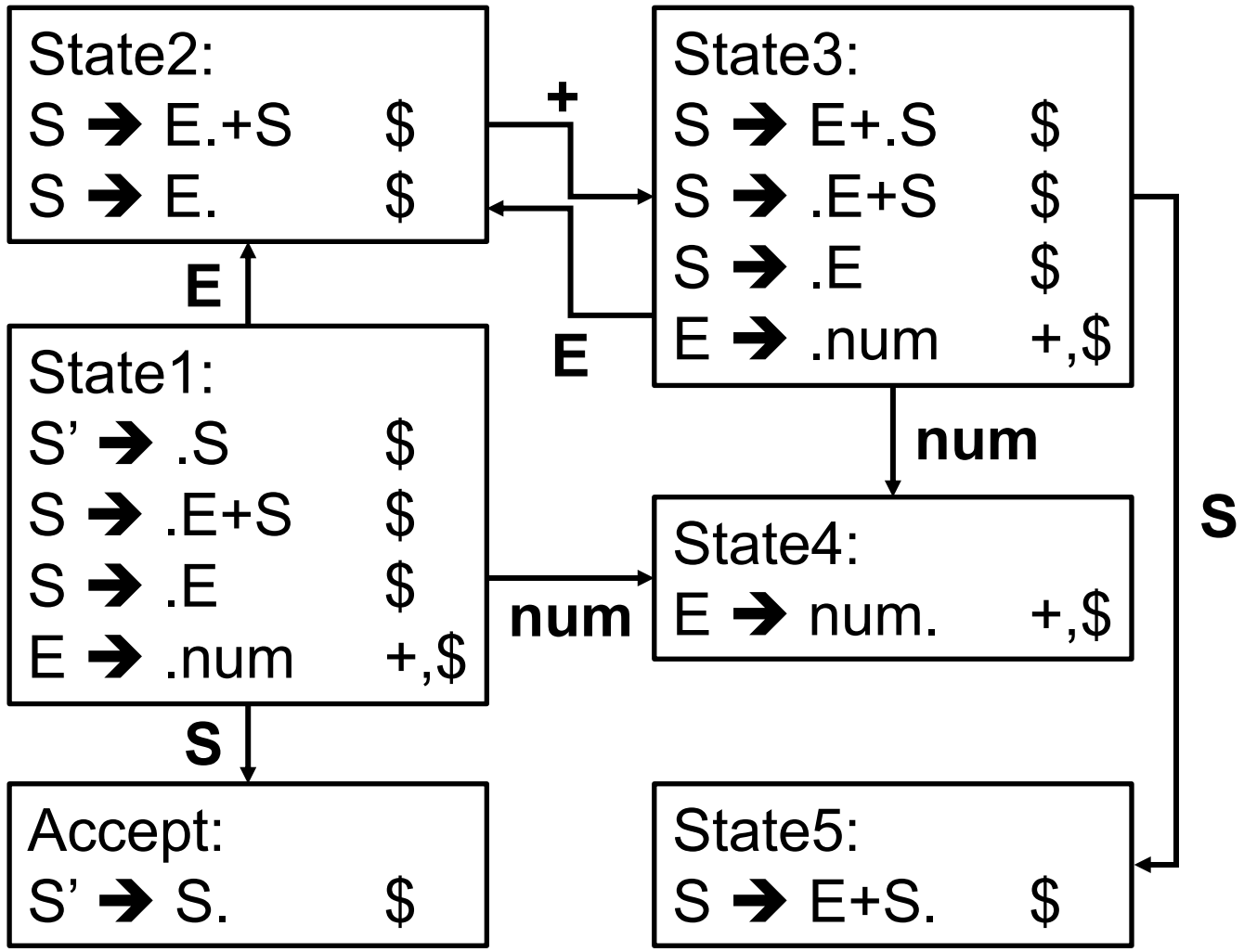
Goto/Shift (I, +)

Closure([S ➔ E + .S, $])

S ➔ E+.S	**$**
S ➔ .E+S	$
S ➔ .E	$
E ➔ .num	+,$

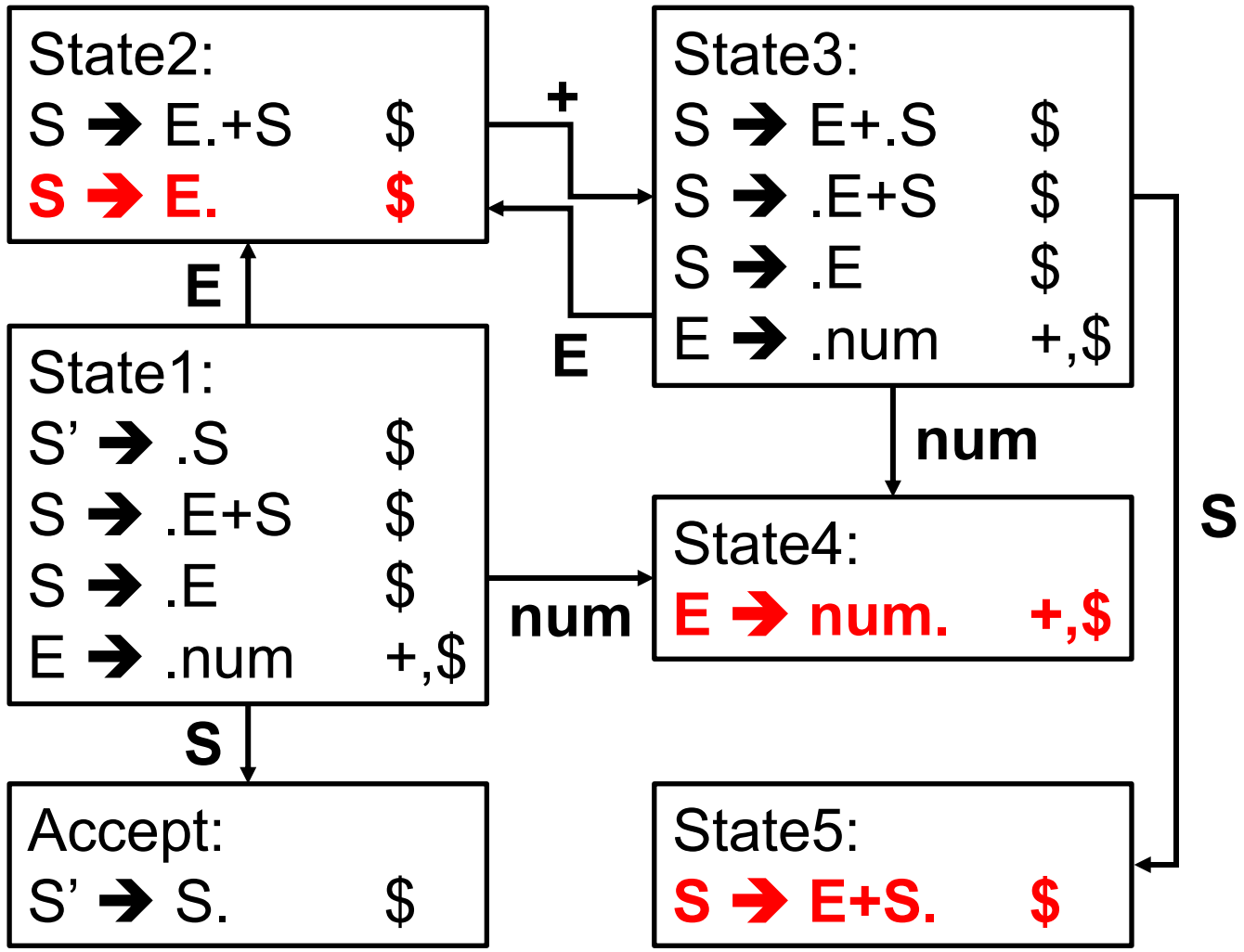
LR(1) DFA Example

S' ➔ S$
S ➔ E + S | E
E ➔ num



LR(1) Reduction

S' ➔ S$
S ➔ E + S | E
E ➔ num



LR(1) Parsing Table

- **Same as LR(0) parsing table construction, except for reductions**
  - For a transition S ➔ S' on terminal x:
    - Table[S, x] = Shift(S')
  - For a transition S ➔ S' on non-terminal N:
    - Table[S, N] = Goto(S')
  - If I contains [X ➔ α, β] then:
    - Table[I, β] = Reduce(X ➔ α)

LR(1)

S ➔ E.+S	$
S ➔ E.	**$**

Reduce if lookahead is $

SLR

S ➔ E.+S
S ➔ E.

Reduce if lookahead is Follow(S) = {$}

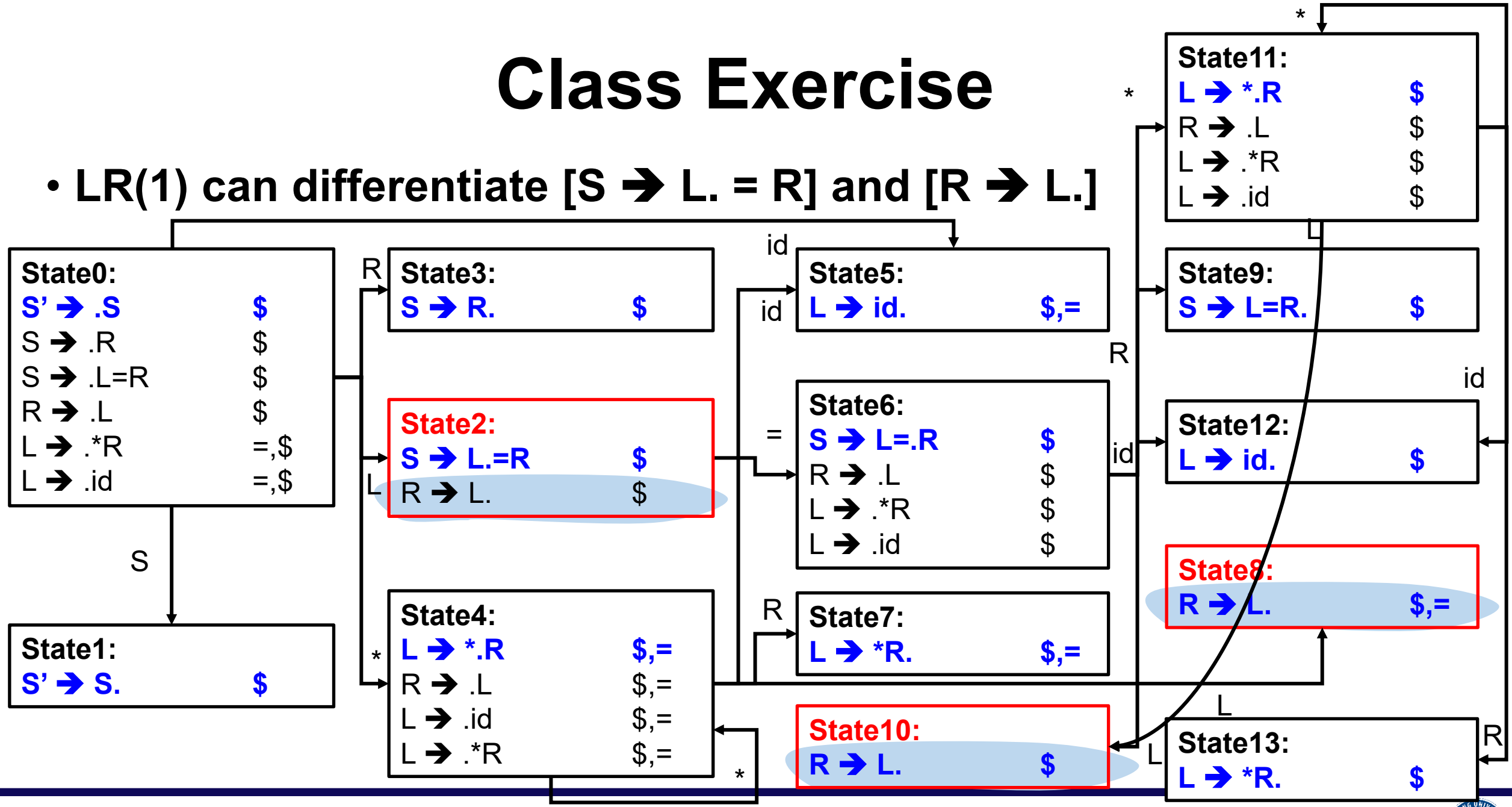
Class Exercise

- **Draw RL(1) DFA for the following grammar**
  - S ➔ L = R
  - S ➔ R
  - L ➔ *R
  - L ➔ id
  - R ➔ L

Hint:
First(S) = {*, id}
First(R) = {*, id}
First(L) = {*, id}

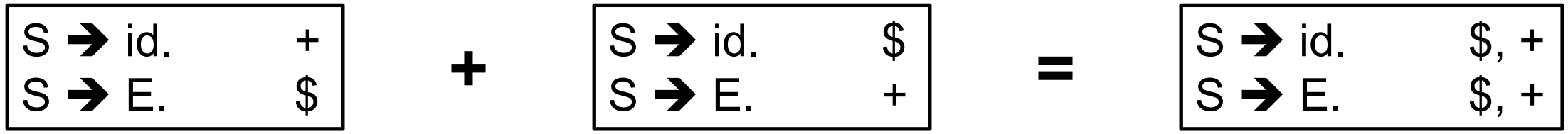
Class Exercise

- LR(1) can differentiate [S → L. = R] and [R → L.]



LALR(1) Parsing

- **LR(1) has too many states**
- **LALR(1) parsing is a lookahead LR**
  - Constructs LR(1) DFA and merges any two LR(1) states whose items have the same production rule, but different lookahead
  - Reduces the number of parser table entries
  - Theoretically less powerful than LR(1)



LALR(1) Parsing

- **LALR(1) generally has the same number of states as SLR (much less than LR(1))**
  - For Pascal programming language:
    - SLR requires several hundred states
    - LR(1) requires several thousand states

- **LALR(1) has the same lookahead capability of LR(1) and much better than SLR**

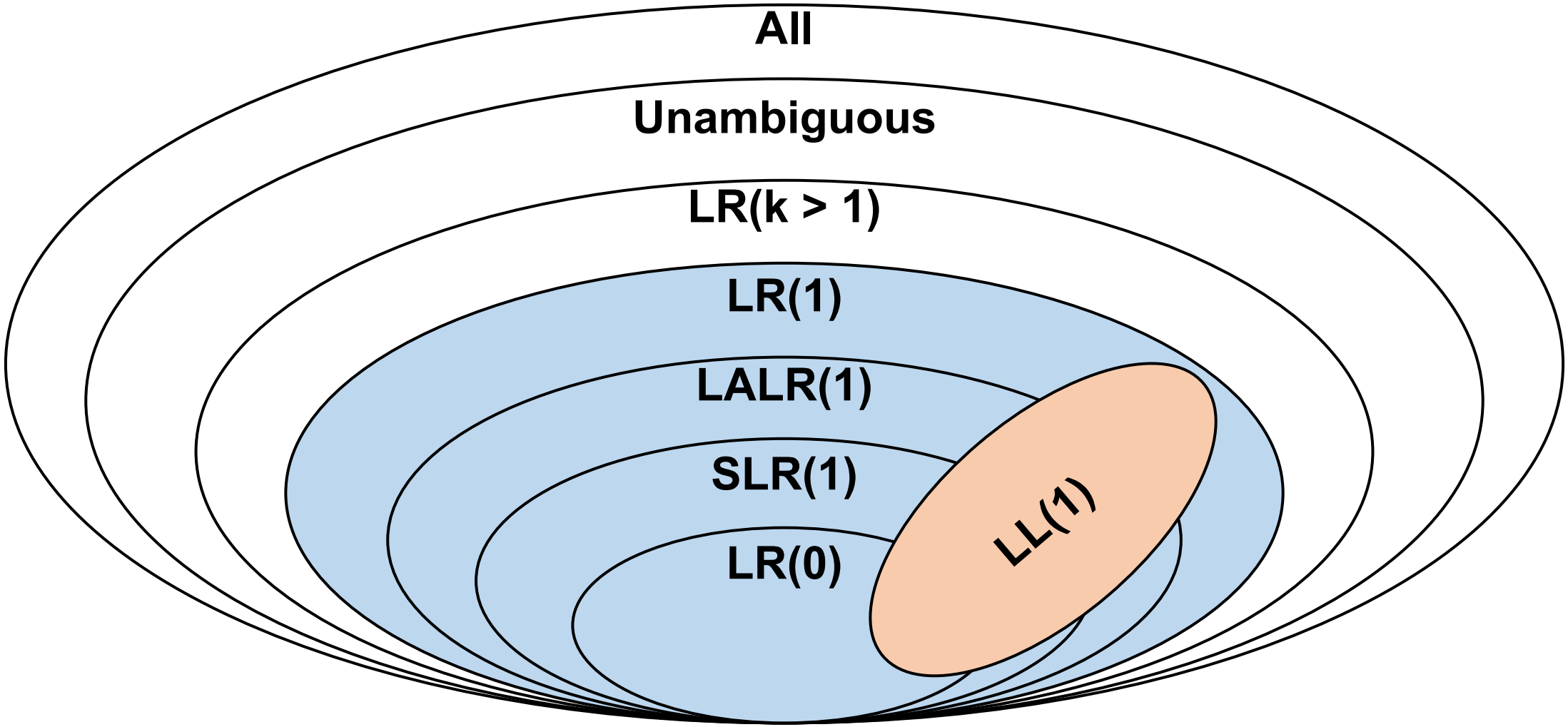
We will not dive into the details of LALR(1)

LL/LR Grammars - 1

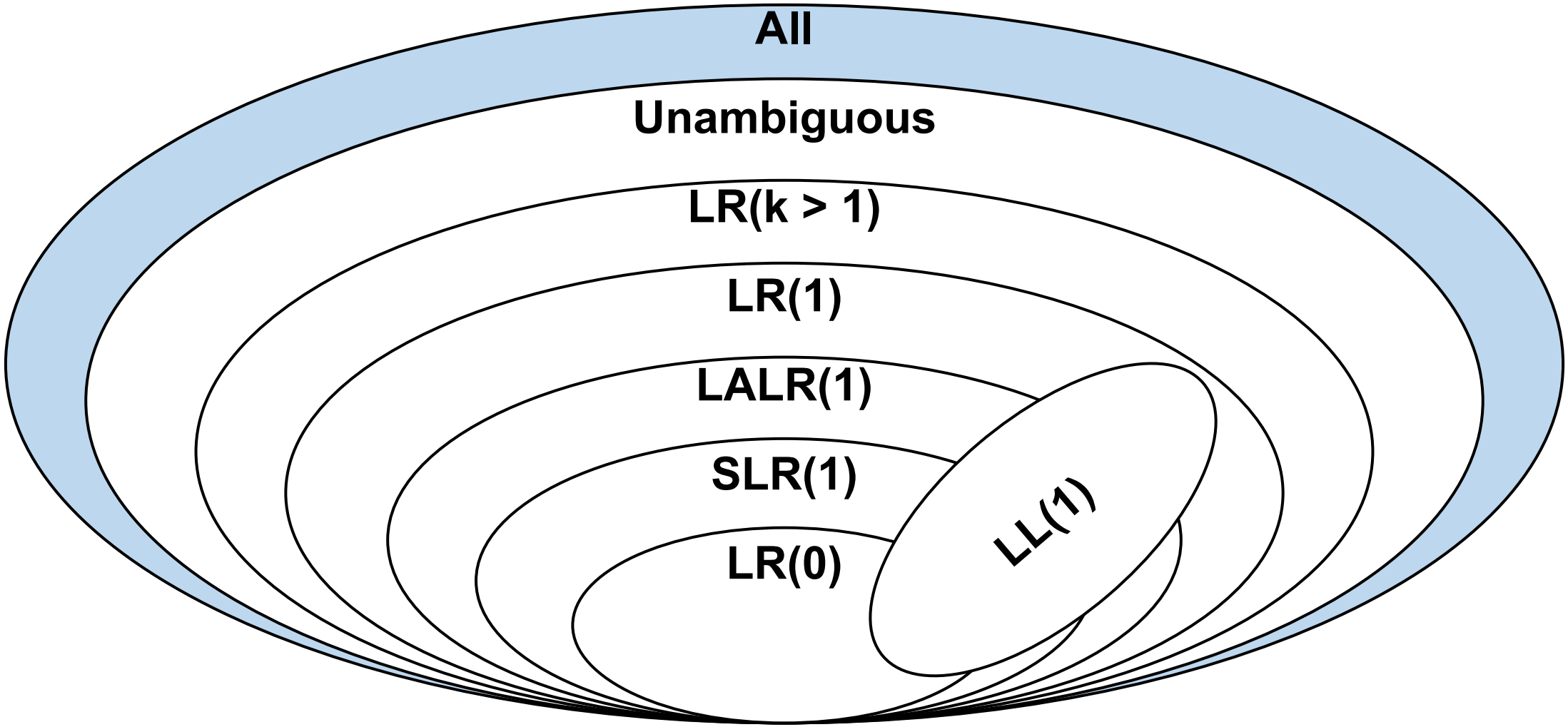
- **LL parsing tables**
  - Table[non-terminal, terminal] = Production to apply
  - Compute using First and Follow

- **LR parsing tables**
  - Table[LR state, terminal] = {shift / reduce / error / accept}
  - Table[LR state, non-terminal] = {goto / error}
  - Compute using closure & goto operations on LR states

LL/LR Grammars - 2

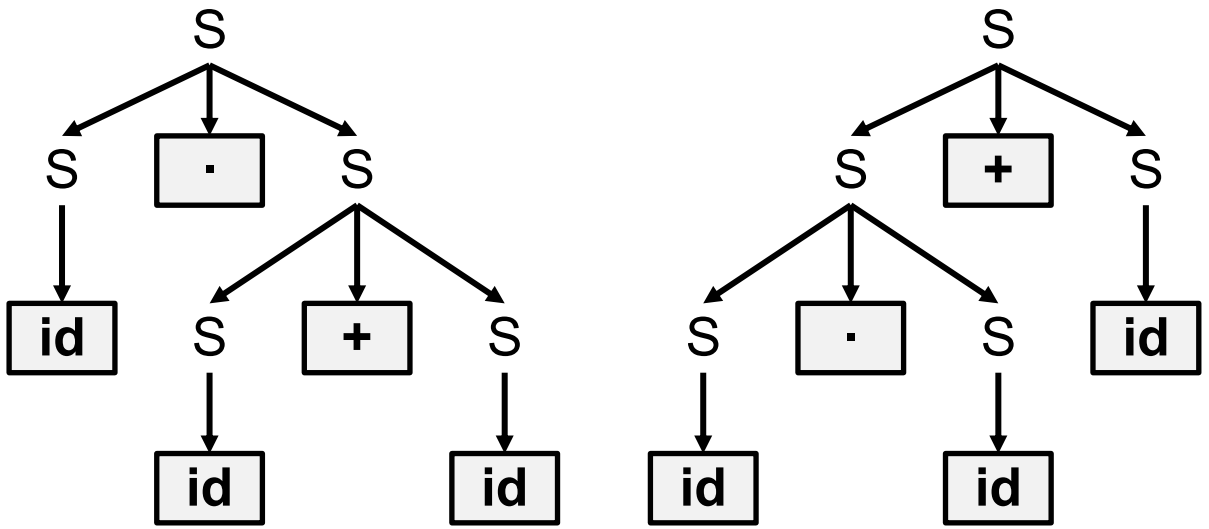


Ambiguous Grammar



Recall: Removing Ambiguity - 1

- **Precedence:**
  - The production at higher levels (closer to the root) will have operators with lower priorities (and vice versa)
  - We can insert non-terminals to enforce precedence



S ➔ S + S
| S · S | id

Ambiguous

S ➔ S + T | T
T ➔ T · id | id

Unambiguous

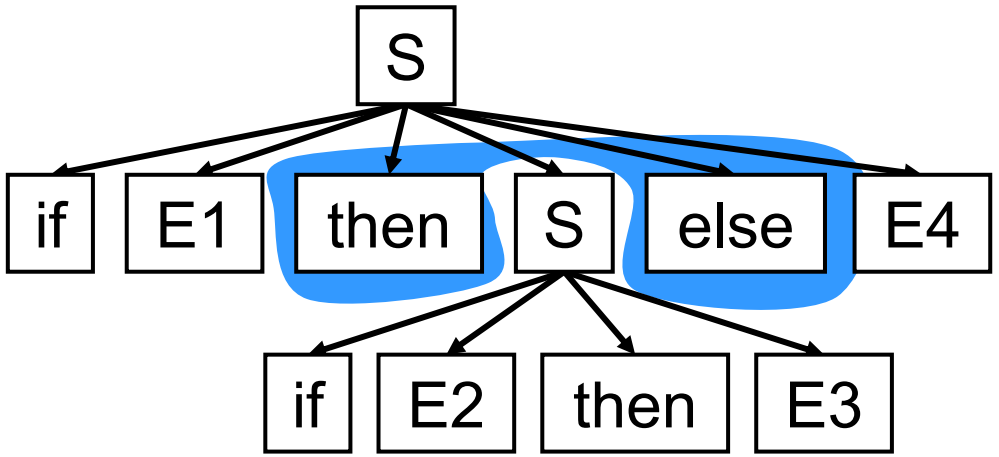
Recall: Removing Ambiguity - 2

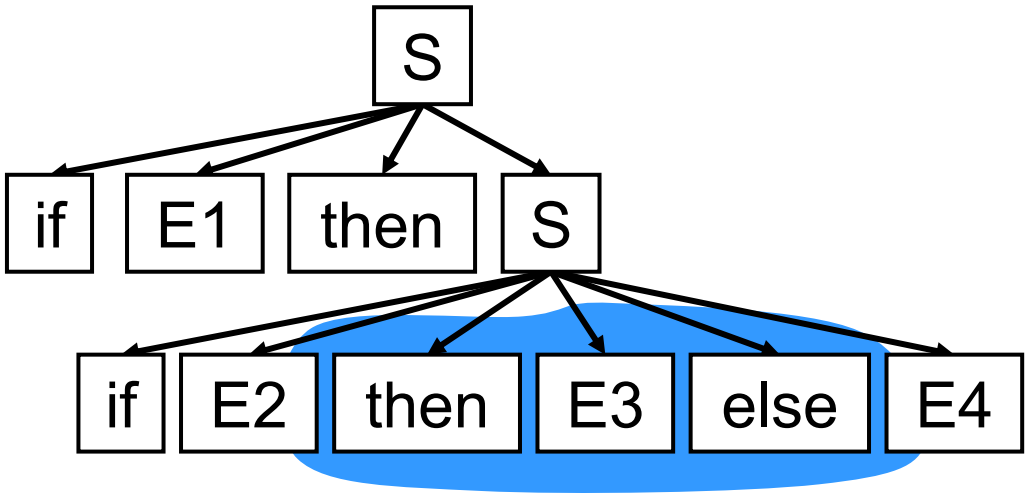
- **Associativity:**
  - We should determine where to place recursion depending on the associativity
    - Left associative: place the recursion on the left
    - Right associative: place the recursion on the right
    - Non associative: do not use recursion

S → S - T | T T → id	S → T ^ S | T T → id	S → A < A | A A → id
Left	**Right**	**Non**

Recall: Removing Ambiguity - 3

- **If/Then/Else**
  - We need to find the correct closure if there are both if/then/else and if/then
  - Consider “if E1 then if E2 then E3 else E4”
    - S ➔ if E then S | if E then S else S | other
    - We should match else to the closest then





Recall: Removing Ambiguity - 4

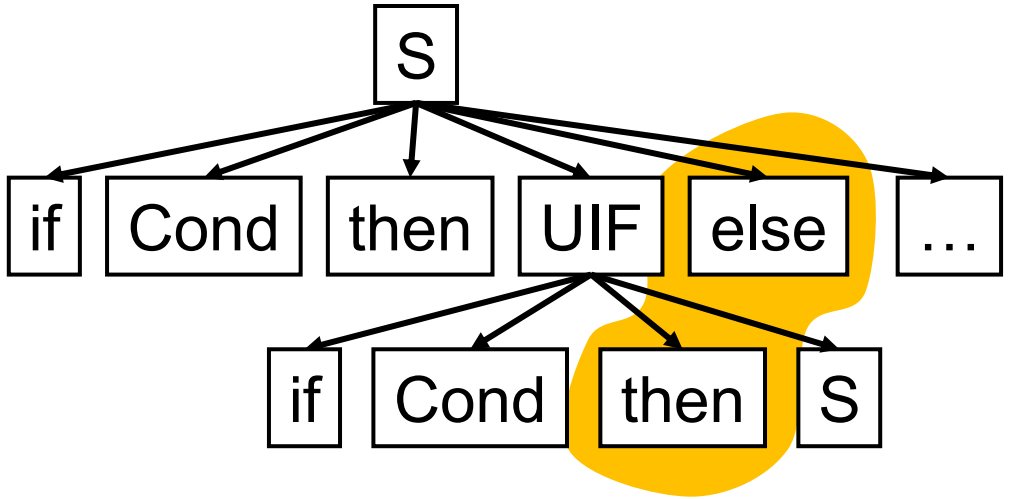
- **We should enforce "else" to match the closest then**

```
S ➔ MIF      // All "then"s are matched
  | UIF      // Some "then"s are unmatched

MIF ➔ if Cond then MIF else MIF
    | Others

UIF ➔ if Cond then S
    | if Cond then MIF else UIF

// Consider if Cond then UIF else S
```

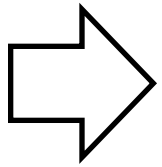


Automatic Disambiguation - 1

- **It is highly complex to propose unambiguous grammars**
  - Precedence: need to consider precedence between different operators
  - Associativity: need to consider associativity of the given operator
- **We can define precedence to use ambiguous grammars w/o shift-reduce conflicts**
  - Define precedence between different types of terminals
  - Define precedence between terminals on the stack vs. terminals on the input

S ➔ S + E | E
E ➔ E * T | T
T ➔ F ^ T | id

Unambiguous



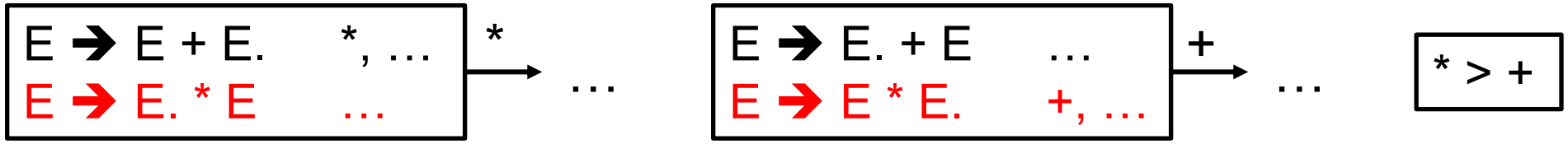
E ➔ E + E
| E * E
| E ^ E | id

Ambiguous

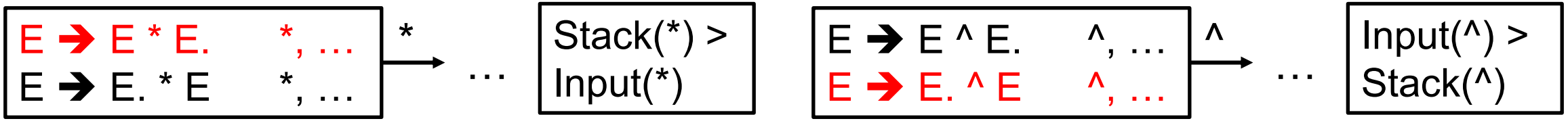
Input(^) > Stack(^) >
Stack(*) > Input(*) >
Stack(+) > Input(+)

Automatic Disambiguation - 2

- **Precedence:**
  - If precedence of the input token is greater than the last terminal on the stack, favor shift over reduce (vice versa)

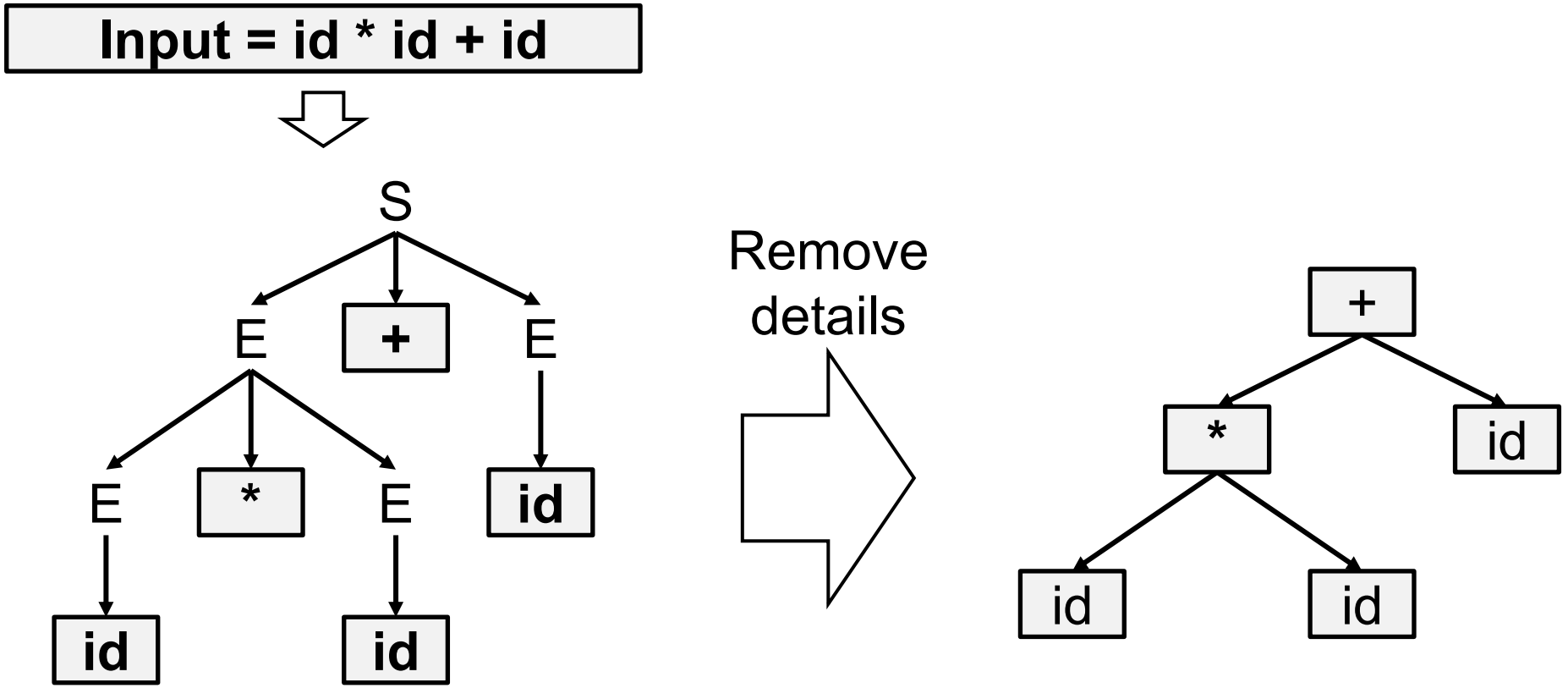


- **Associativity**
  - If an operator is left-associative, assign higher precedence to the operator on the stack (than the one in the input stream)



Recall: AST

- **AST discards unneeded information for syntax analysis**

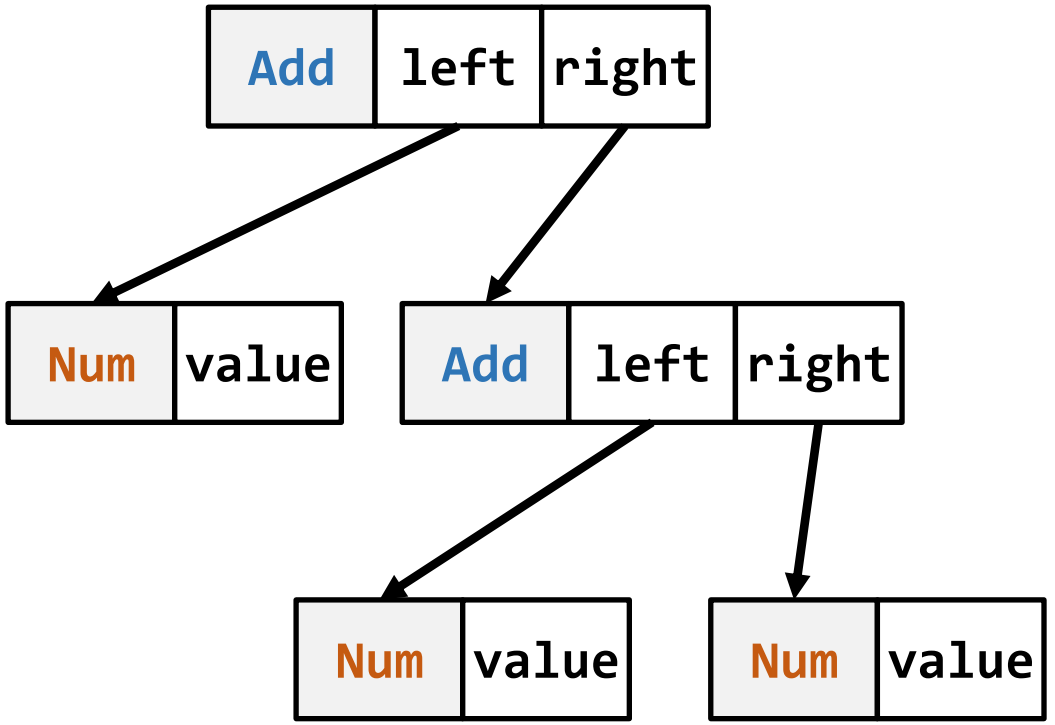


AST Data Structure

```
abstract class expr {}

class Add extends expr {
        expr left, right;
        Add (expr L, expr R) {
                left = L; right = R;
        }
}

class Num extends expr {
        int value;
        Num (int v) {value = v;}
}
```



AST Data Structure

Implicit AST Construction

- **LL/LR parsing techniques implicitly build AST**
- **The parse tree is captured in the derivation**
  - LL parsing: AST represented by the productions
  - LR parsing: AST represented by the reductions
- **We should implicitly construct the AST during the parsing phase**

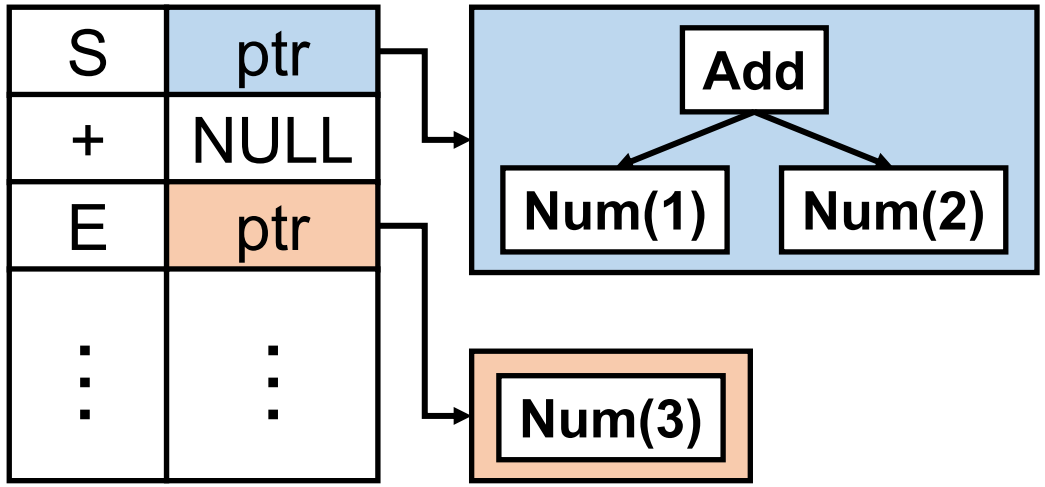
AST Construction: LL

	num	+	(	)	$ (EOF)
S	ES’		ES’		
S’		+S		ε	ε
E	num		(S)		

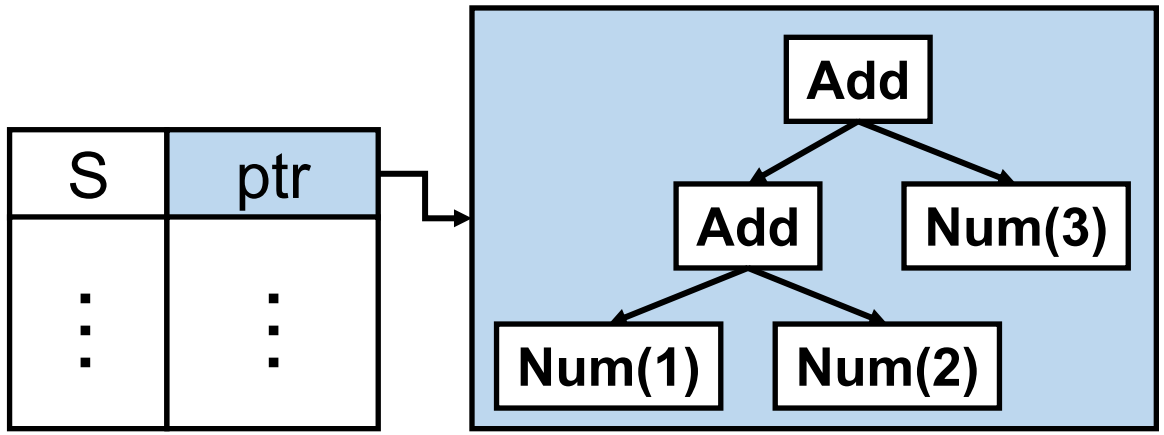
```
expr parse_S() {
    switch (token) {
       case num:
       case ‘(‘:
              expr child1 = parse_E();
              expr child2 = parse_S’();
              return new S(child1, child2);
       default: ParseError(); }}
```

AST Construction: LR

- **AST construction mechanism**
  - Store parts of the tree on the stack
  - For each nonterminal X on the stack, store the sub-tree for X on the stack
  - After reduce operation for a production X ➔ α, create an AST node for X



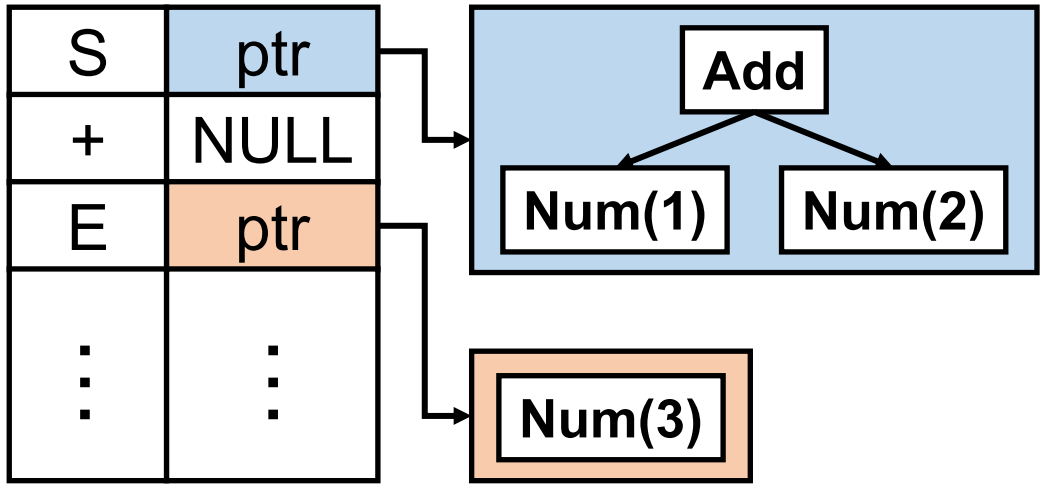
Before S ➔ E + S Reduction



After S ➔ E + S Reduction

AST Construction: LR

- **AST construction mechanism**
  - Store parts of the tree on the stack
  - For each nonterminal X on the stack, store the sub-tree for X on the stack
  - After reduce operation for a production X ➔ α, create an AST node for X



Before S ➔ E + S Reduction

```
// For the reduce action
$3 = pop_stack(); // S
$2 = pop_stack(); // +
$1 = pop_stack(); // E
$$ = new S($1, $3);
push_stack($$);
```

After S ➔ E + S Reduction

Specification, Recognition, and Automation in Parser

- Specification: how to specify valid strings of tokens?
  - Context-free grammars (CFG)
- Recognition: how to recognize the specified patterns?
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (yacc/bison) → Will be covered in the project