



# **EECS 483: Compiler Construction**

**Lecture 24:**

**LR Grammars, Bottom-Up Parsing**

**April 13**

**Winter Semester 2026**

# Announcements

- Reminder: Assignment 5 Due Sunday
- Midterm regrades done. Grades updated on Canvas
- Teaching Evaluations are now open. Please fill them out!

# Final Exam Info

- Final Exam on Wednesday 04/29 10:30-12:30pm
- Location:  
LCSIB 1365 (that's the new building, Leinweber)
- Possible Topics: Assignments 4 and 5, lecture after spring break.
- Exam Review on Monday, 04/20
- 1 page of notes, double sided ok, printed or written ok.
- Practice material:  
Last year's final exam on canvas  
Appel book, Dragon book linked on webpage have relevant material and exercises.



# LR GRAMMARS

# The Parsing Problem

- The Parsing Problem:
  - Input: a context-free grammar  $G$
  - Output: a **parser** that takes in a string and outputs a parse tree of that string in  $G$  or raises an exception if there is no parse tree.
  - Notice that an ambiguous grammar may be parsed in multiple ways
- In practice: fuse the generation of the parse tree with *semantic actions* that construct the abstract syntax tree
  - The parse tree is usually never “materialized” in memory
- Another “mini-compiler” for a DSL
- Bad news: best algorithms are  $O(n^3)$ 
  - CYK, Earley, GLR algorithms
- Compromise: find restrictions on CFGs that allow for  $O(n)$  parsing
  - Intuition: parsing is a **search problem**, find restrictions that limit the amount of backtracking needed.
  - Cost: more burden on the programmer (i.e., **you**) to adapt their grammar to fit the restriction

# LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar
  - ⇒ LL(1) grammar (manual rewrite)
  - ⇒ prediction table (intermediate representation)
  - ⇒ recursive-descent parser (code generation)
- Problems:
  - Grammar must be LL(1)
  - Can extend to LL(k) (it just makes the table bigger)
  - Grammar cannot be left recursive (parser functions will loop!)
- Is there a better way?

# Bottom-up Parsing (LR Parsers)

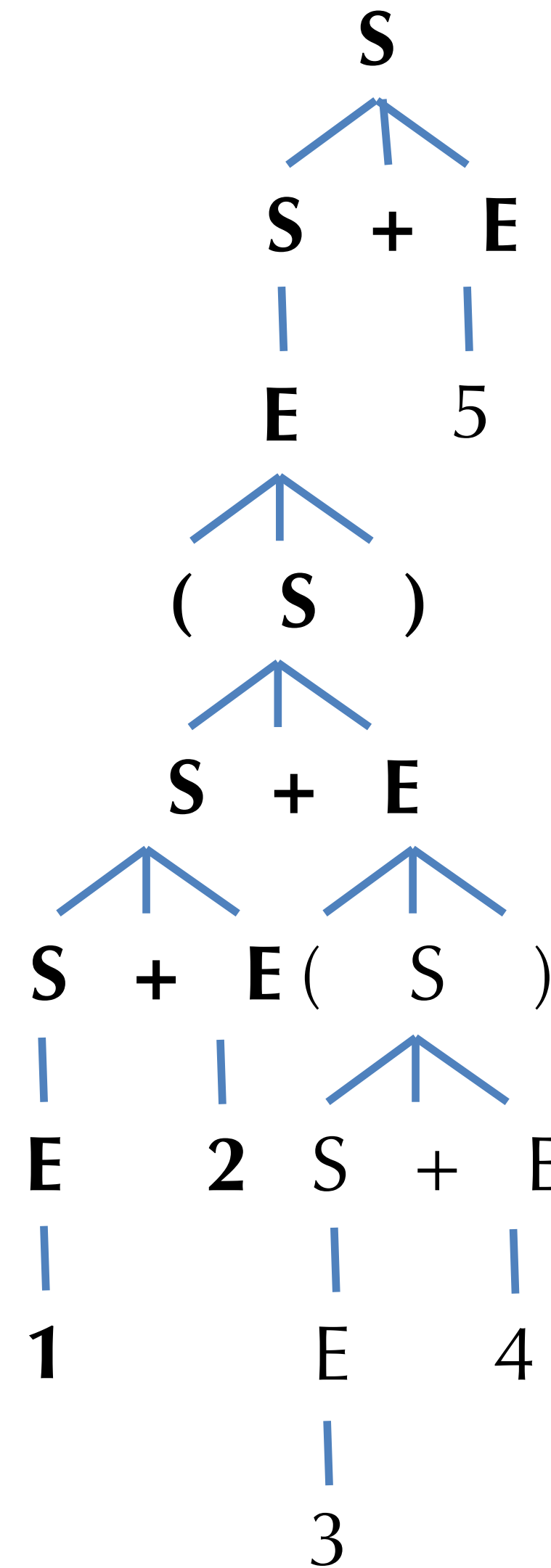
- LR(k) parser:
  - Left-to-right scanning
  - Rightmost derivation
  - k lookahead symbols
- LR grammars are more expressive than LL
  - Can handle left-recursive (and right recursive) grammars; virtually all programming languages
  - Easier to express programming language syntax (no left factoring)
- Technique: “Shift-Reduce” parsers
  - Work bottom up instead of top down
  - Construct right-most derivation of a program in the grammar
  - Used by many parser generators (e.g. yacc, ocaml yacc, lalrpop, etc.)
  - Better error detection/recovery

# Top-down vs. Bottom up

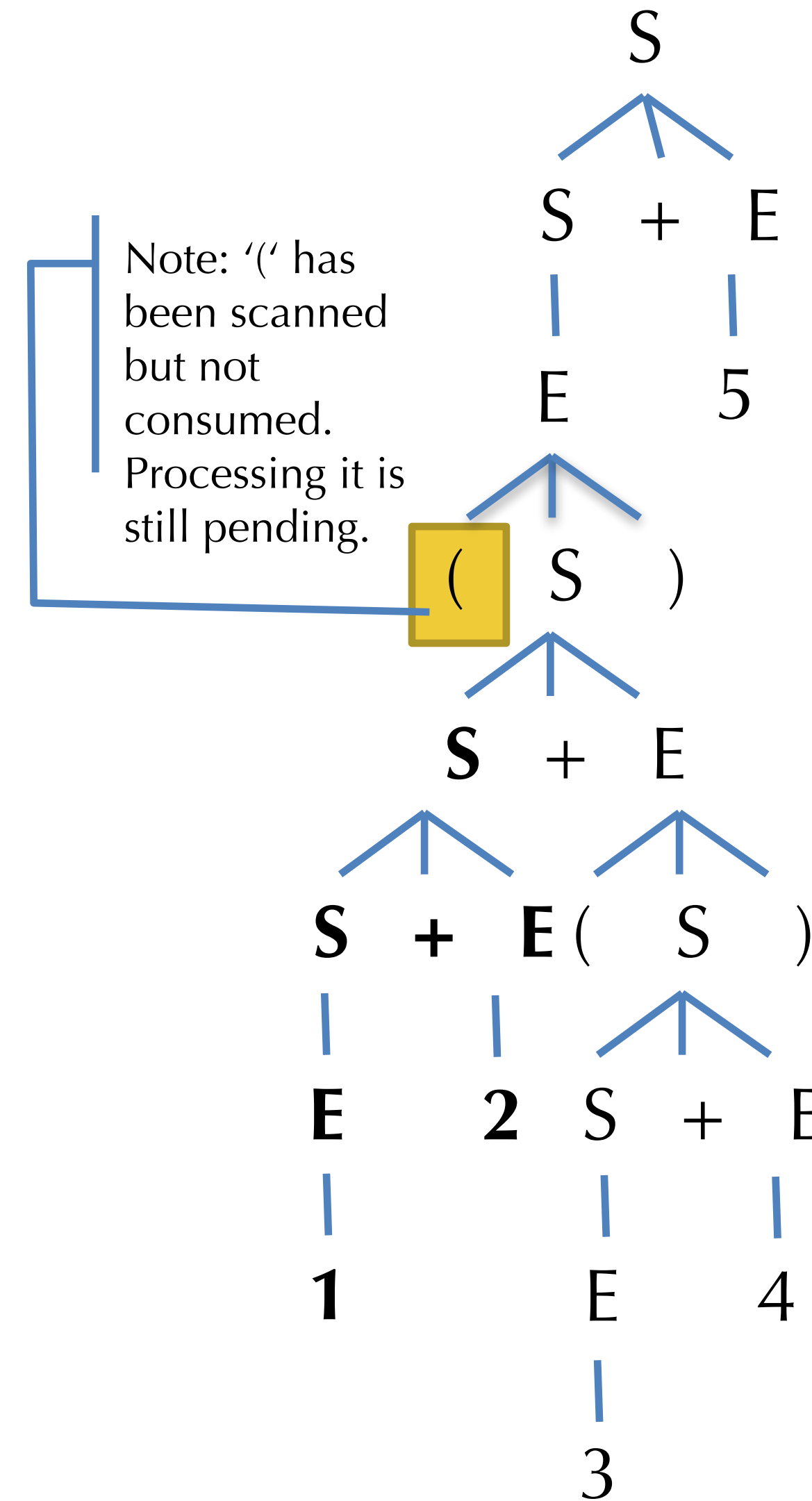
- Consider the left-recursive grammar:

$S \mapsto S + E \mid E$   
 $E \mapsto \text{number} \mid ( S )$

- $(1 + 2 + (3 + 4)) + 5$
- What part of the tree must we know after scanning just  $"(1 + 2"$  ?
- In top-down, must be able to guess which productions to use...




Top-down



Bottom-up

Note: '(' has been scanned but not consumed. Processing it is still pending.

# Progress of Bottom-up Parsing

|  | Reductions  | Scanned               | Input Remaining         |
|--|---|-----------------------|-------------------------|
| <br>Rightmost derivation | $(1 + 2 + (3 + 4)) + 5 \leftarrow$                                    |                       | $(1 + 2 + (3 + 4)) + 5$ |
|  | $(\underline{\mathbf{E}} + 2 + (3 + 4)) + 5 \leftarrow$               | (                     | $1 + 2 + (3 + 4)) + 5$  |
|  | $(\underline{\mathbf{S}} + 2 + (3 + 4)) + 5 \leftarrow$               | (1                    | $+ 2 + (3 + 4)) + 5$    |
|  | $(\mathbf{S} + \underline{\mathbf{E}} + (3 + 4)) + 5 \leftarrow$      | (1 + 2                | $+ (3 + 4)) + 5$        |
|  | $(\underline{\mathbf{S}} + (3 + 4)) + 5 \leftarrow$                   | (1 + 2                | $+ (3 + 4)) + 5$        |
|  | $(\mathbf{S} + (\underline{\mathbf{E}} + 4)) + 5 \leftarrow$          | (1 + 2 + (3           | $+ 4)) + 5$             |
|  | $(\mathbf{S} + (\underline{\mathbf{S}} + 4)) + 5 \leftarrow$          | (1 + 2 + (3           | $+ 4)) + 5$             |
|  | $(\mathbf{S} + (\mathbf{S} + \underline{\mathbf{E}})) + 5 \leftarrow$ | (1 + 2 + (3 + 4       | $) + 5$                 |
|  | $(\mathbf{S} + (\underline{\mathbf{S}})) + 5 \leftarrow$              | (1 + 2 + (3 + 4       | $) + 5$                 |
|  | $(\mathbf{S} + \underline{\mathbf{E}}) + 5 \leftarrow$                | (1 + 2 + (3 + 4)      | $) + 5$                 |
|  | $(\underline{\mathbf{S}}) + 5 \leftarrow$                             | (1 + 2 + (3 + 4)      | $) + 5$                 |
|  | $\underline{\mathbf{E}} + 5 \leftarrow$                               | (1 + 2 + (3 + 4))     | $+ 5$                   |
|  | $\underline{\mathbf{S}} + 5 \leftarrow$                               | (1 + 2 + (3 + 4))     | $+ 5$                   |
|  | $\mathbf{S} + \underline{\mathbf{E}} \leftarrow$                      | (1 + 2 + (3 + 4)) + 5 |                         |
|  | $\mathbf{S}$  |                       |                         |

$S \mapsto S + E \mid E$   
 $E \mapsto \text{number} \mid ( S )$

# Shift/Reduce Parsing

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is  $\text{stack} + \text{input}$
- Parsing is a sequence of *shift* and *reduce* operations:
- **Shift**: move look-ahead token to the stack
- **Reduce**: Replace symbols  $\gamma$  at top of stack with nonterminal  $X$  such that  $X \mapsto \gamma$  is a production. (pop  $\gamma$ , push  $X$ )

$S \mapsto S + E \mid E$   
 $E \mapsto \text{number} \mid ( S )$


| Stack  | Input                     | Action                            |
|--------|---------------------------|-----------------------------------|
|        | ( 1 + 2 + ( 3 + 4 ) ) + 5 | shift (                           |
| (      | 1 + 2 + ( 3 + 4 ) + 5     | shift 1                           |
| (1     | + 2 + ( 3 + 4 ) + 5       | reduce: $E \mapsto \text{number}$ |
| (E     | + 2 + ( 3 + 4 ) + 5       | reduce: $S \mapsto E$             |
| (S     | + 2 + ( 3 + 4 ) + 5       | shift +                           |
| (S +   | 2 + ( 3 + 4 ) + 5         | shift 2                           |
| (S + 2 | + ( 3 + 4 ) + 5           | reduce: $E \mapsto \text{number}$ |
| (S + E | + ( 3 + 4 ) + 5           | reduce: $S \mapsto S + E$         |
| (S     | + ( 3 + 4 ) + 5           | shift +                           |

# Shift/Reduce Parsing

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is  $\text{stack} + \text{input}$
- Invariant: Stack plus input is a step in building the Rightmost derivation in reverse

$$S \mapsto S + E \mid E$$

$$E \mapsto \text{number} \mid ( S )$$

| Stack   | Input                     | Derivation steps                 |  |
|---------|---------------------------|----------------------------------|--|
|         | ( 1 + 2 + ( 3 + 4 ) ) + 5 | ( 1 + 2 + ( 3 + 4 ) ) + 5        |  |
| (       | 1 + 2 + ( 3 + 4 ) ) + 5   |                                  |  |
| ( 1     | + 2 + ( 3 + 4 ) ) + 5     |                                  |  |
| ( E     | + 2 + ( 3 + 4 ) ) + 5     | ( <u>E</u> + 2 + ( 3 + 4 ) ) + 5 |  |
| ( S     | + 2 + ( 3 + 4 ) ) + 5     | ( <u>S</u> + 2 + ( 3 + 4 ) ) + 5 |  |
| ( S +   | 2 + ( 3 + 4 ) ) + 5       |                                  |  |
| ( S + 2 | + ( 3 + 4 ) ) + 5         |                                  |  |
| ( S + E | + ( 3 + 4 ) ) + 5         | ( S + <u>E</u> + ( 3 + 4 ) ) + 5 |  |
| ( S     | + ( 3 + 4 ) ) + 5         | ( <u>S</u> + ( 3 + 4 ) ) + 5     |  |

Simple LR parsing with no look ahead.

# LR(0) GRAMMARS

# LR Parser States

- Goal: know what set of reductions are legal at any given point.
- Idea: Summarize all possible stack prefixes  $\alpha$  as a finite parser state.
  - Parser state is computed by a DFA that reads the stack  $\sigma$ .
  - Accept states of the DFA correspond to unique reductions that apply.
- Example: LR(0) parsing
  - Left-to-right scanning, Right-most derivation, zero look-ahead tokens
  - Too weak to handle many language grammars (e.g. the “sum” grammar)
  - But, helpful for understanding how the shift-reduce parser works.

# Example LR(0) Grammar: Tuples

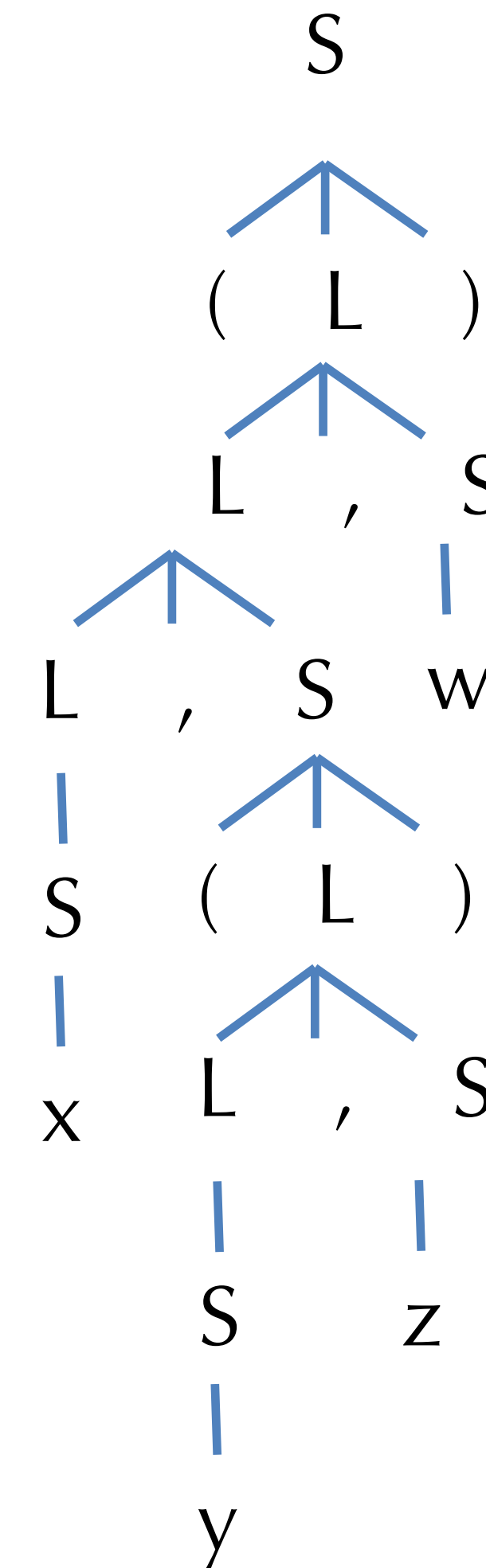
- Example grammar for non-empty tuples and identifiers:

$$\begin{aligned} S &\mapsto ( L ) \mid \text{id} \\ L &\mapsto S \mid L , S \end{aligned}$$

- Example strings:

- x
- (x,y)
- (((x)))
- (x, (y, z), w)
- (x, (y, (z, w)))

Parse tree for:  
(x, (y, z), w)



# Shift/Reduce Parsing

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is stack + input
- Parsing is a sequence of *shift* and *reduce* operations:
- **Shift**: move look-ahead token to the stack: e.g.

$$S \mapsto ( L ) \mid id$$

$$L \mapsto S \mid L , S$$

| Stack | Input          | Action  |
|-------|----------------|---------|
|       | (x, (y, z), w) | shift ( |
| (     | x, (y, z), w)  | shift x |

- **Reduce**: Replace symbols  $\gamma$  at top of stack with nonterminal  $X$  such that  $X \mapsto \gamma$  is a production. (pop  $\gamma$ , push  $X$ ): e.g.

| Stack | Input        | Action                |
|-------|--------------|-----------------------|
| (x    | , (y, z), w) | reduce $S \mapsto id$ |
| (S    | , (y, z), w) | reduce $L \mapsto S$  |

# Example Run

| Stack     | Input          | Action                   |
|-----------|----------------|--------------------------|
|           | (x, (y, z), w) | shift (                  |
| (         | x, (y, z), w)  | shift x                  |
| (x        | , (y, z), w)   | reduce $S \mapsto id$    |
| (S        | , (y, z), w)   | reduce $L \mapsto S$     |
| (L        | , (y, z), w)   | shift ,                  |
| (L,       | (y, z), w)     | shift (                  |
| (L, (     | y, z), w)      | shift y                  |
| (L, (y    | , z), w)       | reduce $S \mapsto id$    |
| (L, (S    | , z), w)       | reduce $L \mapsto S$     |
| (L, (L    | , z), w)       | shift ,                  |
| (L, (L,   | z), w)         | shift z                  |
| (L, (L, z | ), w)          | reduce $S \mapsto id$    |
| (L, (L, S | ), w)          | reduce $L \mapsto L, S$  |
| (L, (L    | ), w)          | shift )                  |
| (L, (L)   | , w)           | reduce $S \mapsto ( L )$ |
| (L, S     | , w)           | reduce $L \mapsto L, S$  |
| (L        | , w)           | shift ,                  |
| (L,       | w)             | shift w                  |
| (L, w     | )              | reduce $S \mapsto id$    |
| (L, S     | )              | reduce $L \mapsto L, S$  |
| (L        | )              | shift )                  |
| (L)       |                | reduce $S \mapsto ( L )$ |
| S         |                |                          |

$S \mapsto ( L ) \mid id$   
 $L \mapsto S \mid L , S$

# Action Selection Problem

- Given a stack  $\sigma$  and a look-ahead symbol  $b$ , should the parser:
  - Shift  $b$  onto the stack (new stack is  $\sigma b$ )
  - Reduce a production  $X \mapsto \gamma$ , assuming that  $\sigma = \alpha\gamma$  (new stack is  $\alpha X$ )?
- Sometimes the parser can reduce but shouldn't
  - For example,  $X \mapsto \varepsilon$  can *always* be reduced
- Sometimes the stack can be reduced in different ways
- Main idea: decide what to do based on a *prefix*  $\alpha$  of the stack plus the look-ahead symbol.
  - The prefix  $\alpha$  is different for different possible reductions since in productions  $X \mapsto \gamma$  and  $Y \mapsto \beta$ ,  $\gamma$  and  $\beta$  might have different lengths.
- Main goal: know what set of reductions are legal at any point.
  - How do we keep track?

# LR(0) States

- An LR(0) *state* is a set of *items* keeping track of progress on possible upcoming reductions.
- An LR(0) *item* is a production from the language with an extra separator “.” somewhere in the right-hand-side

$$\begin{array}{l} S \mapsto ( L ) \mid id \\ L \mapsto S \mid L , S \end{array}$$

- Example items:  $S \mapsto .( L )$  or  $S \mapsto (. L)$  or  $L \mapsto S.$
- Intuition:
  - Stuff before the ‘.’ is already on the stack (beginnings of possible  $\gamma$ 's to be reduced)
  - Stuff after the ‘.’ is what might be seen next
  - The prefixes  $\alpha$  are represented by the state itself

# Constructing the DFA: Start state & Closure

- First step: Add a new production  $S' \mapsto S\$$  to the grammar
- Start state of the DFA = empty stack, so it contains the item:  
 $S' \mapsto .S\$$
- Closure of a state:
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the  $'.'$
  - The added items have the  $'.'$  located at the beginning (no symbols for those items have been added to the stack yet)
  - Note that newly added items may cause yet more items to be added to the state... keep iterating until a *fixed point* is reached.
- Example:  $\text{CLOSURE}(\{S' \mapsto .S\$\}) = \{S' \mapsto .S\$, S \mapsto .(L), S \mapsto .id\}$
- Resulting “closed state” contains the set of all possible productions that might be reduced next.

$$\begin{array}{l} S' \mapsto S\$ \\ S \mapsto ( L ) \mid id \\ L \mapsto S \mid L , S \end{array}$$

# Example: Constructing the DFA

$S' \mapsto .S\$$

$S' \mapsto S\$$   
 $S \mapsto ( L ) \mid id$   
 $L \mapsto S \mid L , S$

- First, we construct a state with the initial item  $S' \mapsto .S\$$

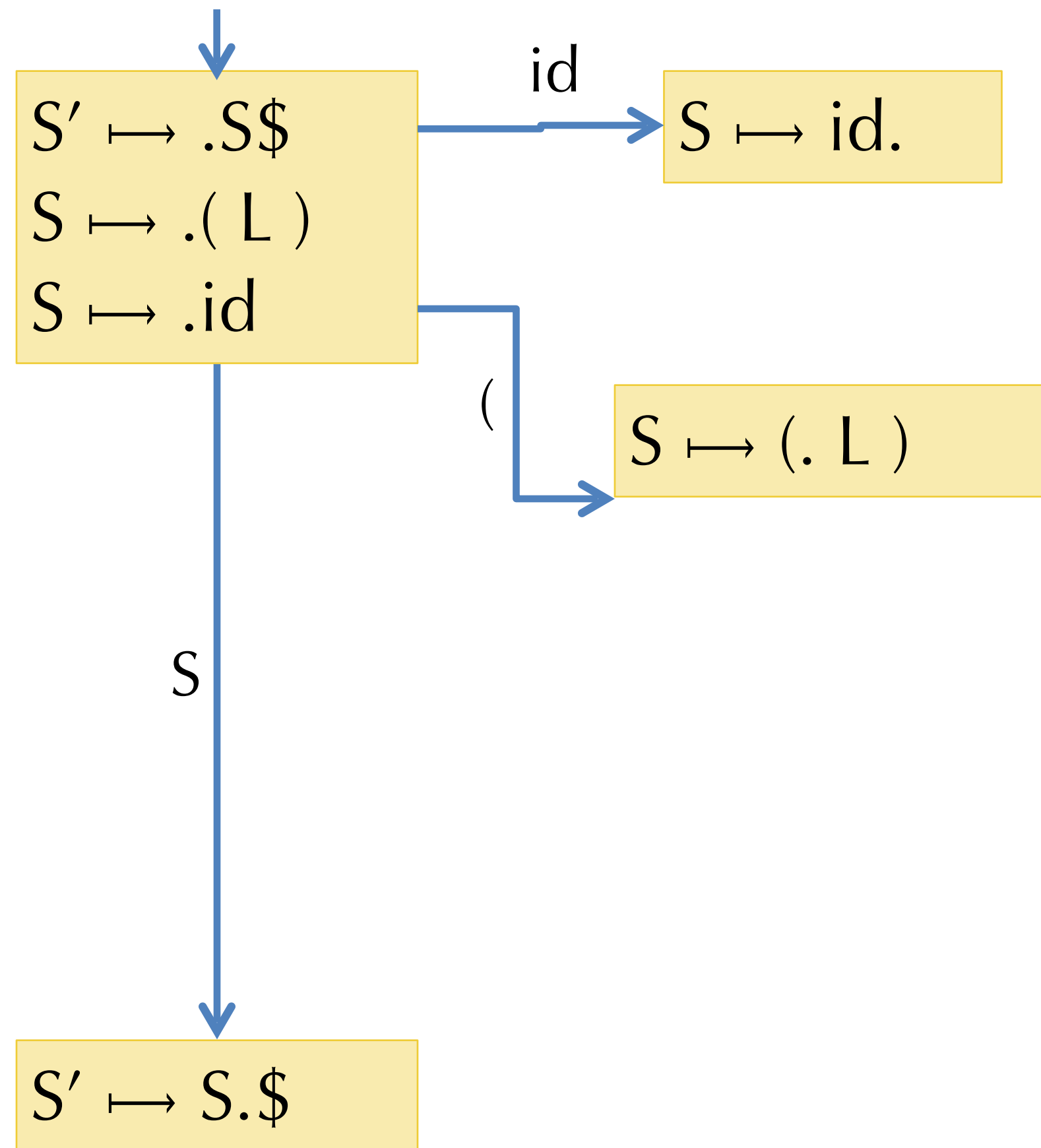
# Example: Constructing the DFA

↓  
 $S' \mapsto .S\$$   
 $S \mapsto .(L)$   
 $S \mapsto .id$

$S' \mapsto S\$$   
 $S \mapsto (L) \mid id$   
 $L \mapsto S \mid L, S$

- Next, we take the closure of that state:  
 $\text{CLOSURE}(\{S' \mapsto .S\}) = \{S' \mapsto .S\}, S \mapsto .(L), S \mapsto .id\}$
- In the set of items, the nonterminal S appears after the '.'
- So we add items for each S production in the grammar

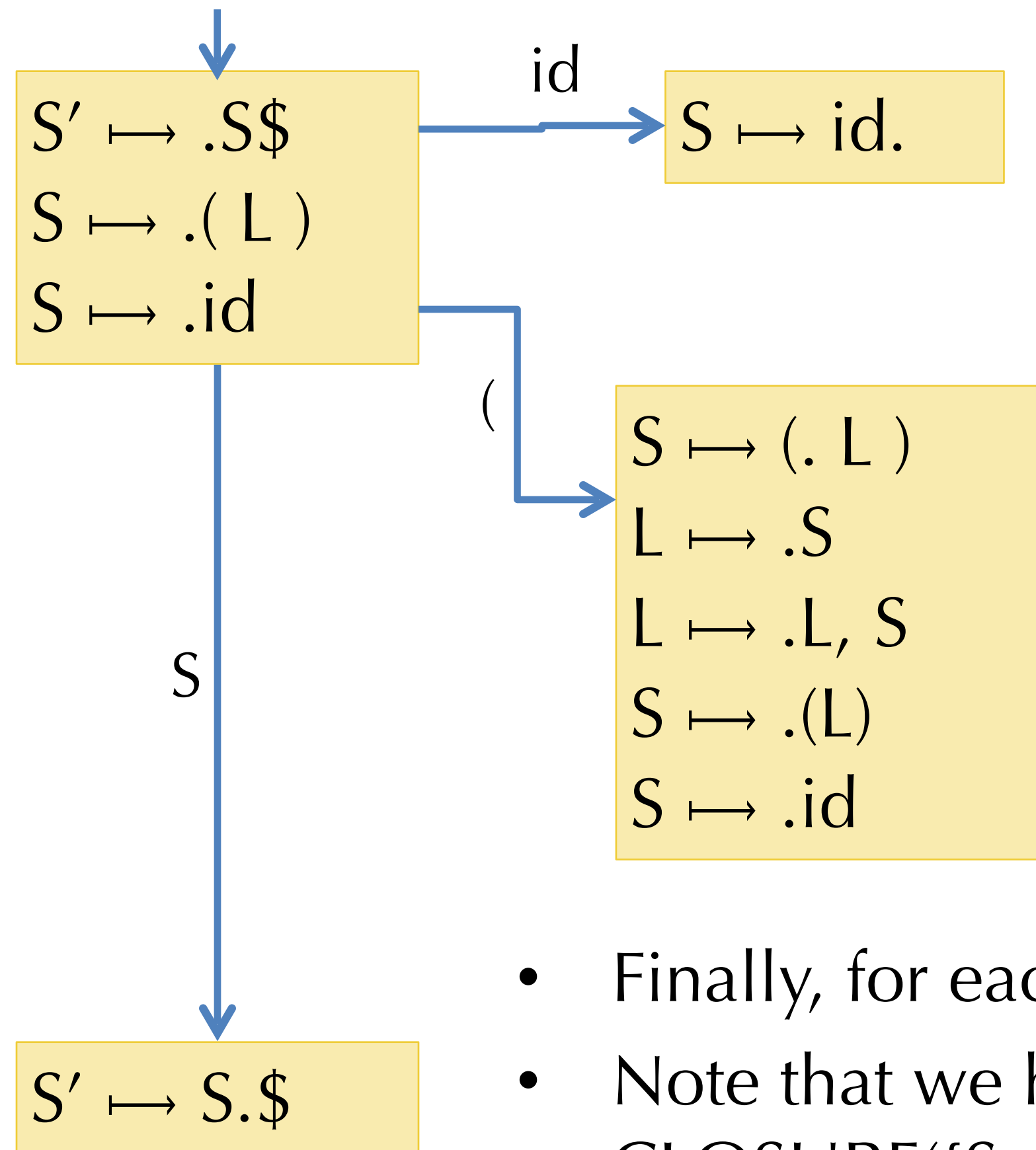
# Example: Constructing the DFA



$S' \mapsto S\$$   
 $S \mapsto (L) \mid id$   
 $L \mapsto S \mid L, S$

- Next we add the transitions:
- First, we see what terminals and nonterminals can appear after the '.' in the source state.
  - Outgoing edges have those label.
- The target state (initially) includes all items from the source state that have the edge-label symbol after the '.', but we advance the '.' (to simulate shifting the item onto the stack)

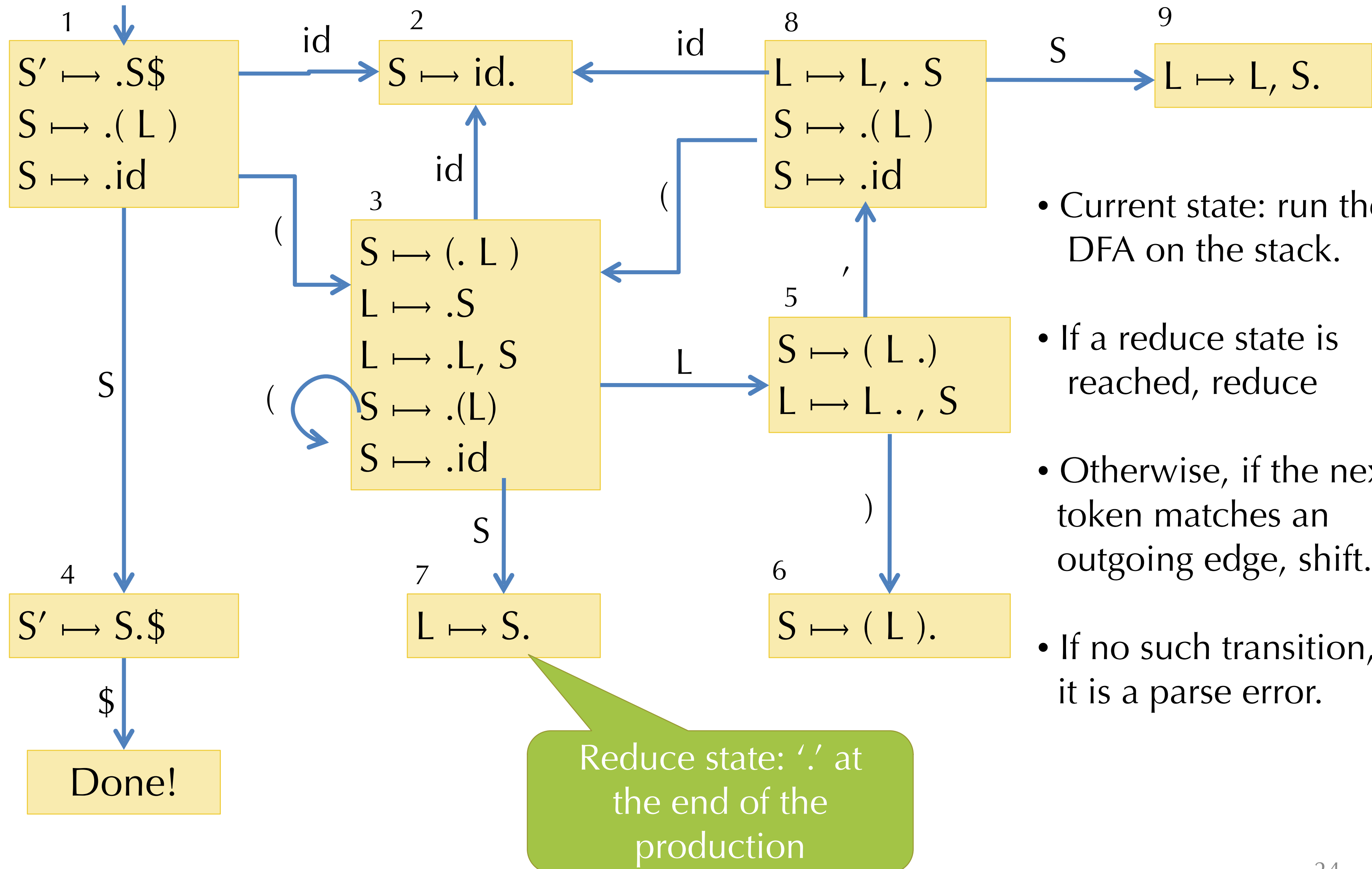
# Example: Constructing the DFA



$S' \mapsto S\$$   
 $S \mapsto (L) \mid id$   
 $L \mapsto S \mid L, S$

- Finally, for each new state, we take the closure.
- Note that we have to perform two iterations to compute  $CLOSURE(\{S \mapsto (.L)\})$ 
  - First iteration adds  $L \mapsto .S$  and  $L \mapsto .L, S$
  - Second iteration adds  $S \mapsto .(L)$  and  $S \mapsto .id$

# Full DFA for the Example



- Current state: run the DFA on the stack.
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift.
- If no such transition, it is a parse error.

# Using the DFA

- Run the parser stack through the DFA.
- The resulting state tells us which productions might be reduced next.
  - If not in a reduce state, then shift the next symbol and transition according to DFA.
  - If in a reduce state,  $X \mapsto \gamma$  with stack  $\alpha\gamma$ , pop  $\gamma$  and push  $X$ .
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g.  ${}_1({}_3({}_3L_5)_6$
  - On a reduction  $X \mapsto \gamma$ , pop stack to reveal the state too:  
e.g. From stack  ${}_1({}_3({}_3L_5)_6$  reduce  $S \mapsto ( L )$  to reach stack  ${}_1({}_3$
  - Next, push the reduction symbol: e.g. to reach stack  ${}_1({}_3S$
  - Then take just one step in the DFA to find next state:  ${}_1({}_3S_7$

# Implementing the Parsing Table

Represent the DFA as a table of shape:

state \* (terminals + nonterminals)

- Entries for the “action table” specify two kinds of actions:
  - Shift and goto state n
  - Reduce using reduction  $X \mapsto \gamma$ 
    - First pop  $\gamma$  off the stack to reveal the state
    - Look up X in the “goto table” and goto that state



# Example Parse Table

|   | (               | )               | id              | ,               | \$              | S  | L  |
|---|-----------------|-----------------|-----------------|-----------------|-----------------|----|----|
| 1 | s3              |                 | s2              |                 |                 | g4 |    |
| 2 | $S \mapsto id$  | $S \mapsto id$  | $S \mapsto id$  | $S \mapsto id$  | $S \mapsto id$  |    |    |
| 3 | s3              |                 | s2              |                 |                 | g7 | g5 |
| 4 |                 |                 |                 |                 | DONE            |    |    |
| 5 |                 | s6              |                 | s8              |                 |    |    |
| 6 | $S \mapsto (L)$ | $S \mapsto (L)$ | $S \mapsto (L)$ | $S \mapsto (L)$ | $S \mapsto (L)$ |    |    |
| 7 | $L \mapsto S$   | $L \mapsto S$   | $L \mapsto S$   | $L \mapsto S$   | $L \mapsto S$   |    |    |
| 8 | s3              |                 | s2              |                 |                 | g9 |    |
| 9 | $L \mapsto L,S$ | $L \mapsto L,S$ | $L \mapsto L,S$ | $L \mapsto L,S$ | $L \mapsto L,S$ |    |    |

sx = shift and goto state x

gx = goto state x (used when we reduce)

# Example

- Parse the token stream:  $(x, (y, z), w)\$$

| Stack                   | Stream             | Action (according to table) |
|-------------------------|--------------------|-----------------------------|
| $\epsilon_1$            | $(x, (y, z), w)\$$ | s3                          |
| $\epsilon_1(3$          | $x, (y, z), w)\$$  | s2                          |
| $\epsilon_1(3x_2$       | $, (y, z), w)\$$   | Reduce: $S \mapsto id$      |
| $\epsilon_1(3S$         | $, (y, z), w)\$$   | g7 (from state 3 follow S)  |
| $\epsilon_1(3S_7$       | $, (y, z), w)\$$   | Reduce: $L \mapsto S$       |
| $\epsilon_1(3L$         | $, (y, z), w)\$$   | g5 (from state 3 follow L)  |
| $\epsilon_1(3L_5$       | $, (y, z), w)\$$   | s8                          |
| $\epsilon_1(3L_{5,8}$   | $(y, z), w)\$$     | s3                          |
| $\epsilon_1(3L_{5,8}(3$ | $y, z), w)\$$      | s2                          |

# LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action.
  - In such states, the machine *always* reduces (ignoring lookahead)
- With more complex grammars, the DFA construction will yield states with shift/reduce and reduce/reduce conflicts:

OK

$S \mapsto ( L ).$

shift/reduce

$S \mapsto ( L ).$   
 $L \mapsto .L , S$

reduce/reduce

$S \mapsto L , S.$   
 $S \mapsto , S.$

- Such conflicts can often be resolved by using a look-ahead symbol: SLR(1) or LR(1)

# Examples

- Consider the left associative and right associative “sum” grammars:

left

$$\begin{aligned} S &\mapsto S + E \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

right

$$\begin{aligned} S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

- One is LR(0) the other isn't... which is which and why?
- What kind of conflict do you get? Shift/reduce or Reduce/reduce?
- Ambiguities in associativity/precedence usually lead to shift/reduce conflicts.

# Examples

- Consider the left associative and right associative “sum” grammars:

left

$$\begin{aligned} S &\mapsto S + E \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

right

$$\begin{aligned} S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

- One is LR(0) the other isn't... which is which and why?
- What kind of conflict do you get? Shift/reduce or Reduce/reduce?

If the stack is a single E, then the state is

$$\begin{aligned} S &\mapsto E . + S \\ S &\mapsto E . \end{aligned}$$

shift-reduce conflict: we can either shift the + or reduce the E to an S.  
LR(0) parser can't decide

# SLR(1) (“simple” LR) Parsers

- What conflicts are there in LR(0) parsing?
  - reduce/reduce conflict: an LR(0) state has two reduce actions
  - shift/reduce conflict: an LR(0) state mixes reduce and shift actions
- Can we use lookahead to disambiguate?
- SLR(1) – uses the same DFA construction as LR(0)
  - modifies the actions based on lookahead. More powerful
- Suppose reducing an A nonterminal is possible in some state:
  - compute Follow(A) for the given grammar
  - if the lookahead symbol is in Follow(A), then reduce, otherwise shift
  - can disambiguate between reduce/reduce conflicts if the follow sets are disjoint

# LR(1) Parsing

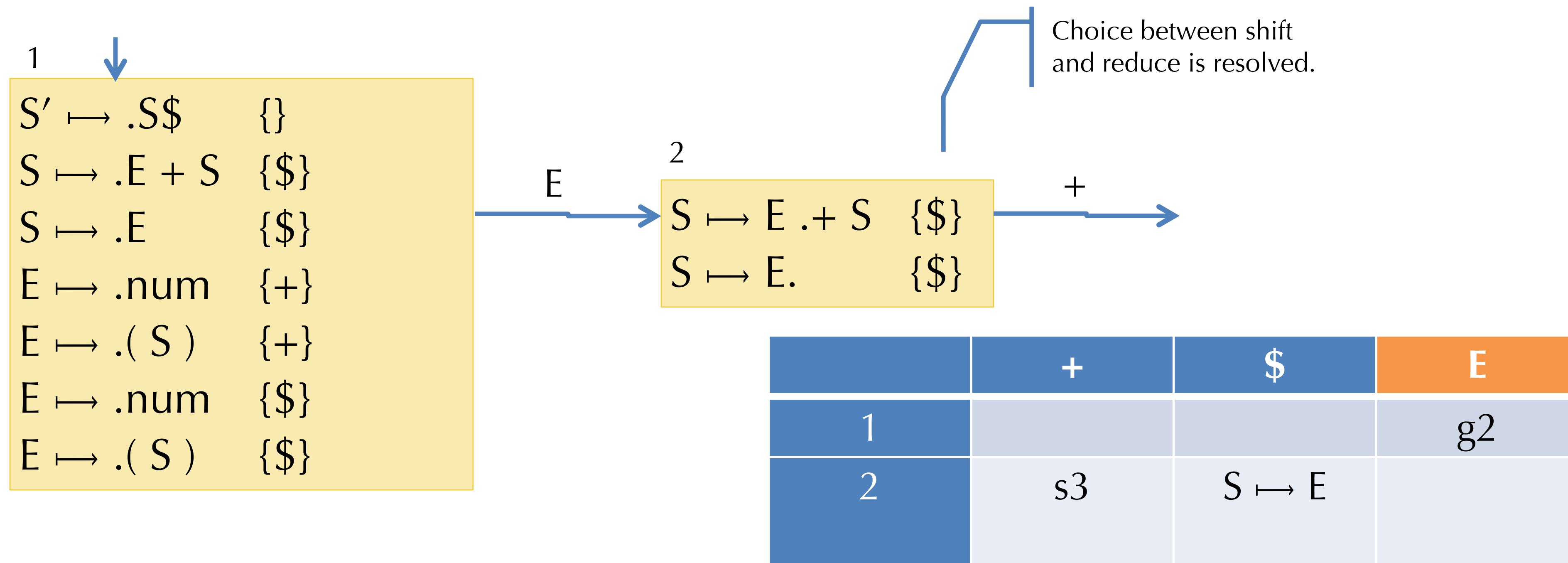
- Yet more powerful than SLR(1)
- Algorithm is similar to LR(0) DFA construction:
  - LR(1) state = set of LR(1) items
  - An LR(1) item is an LR(0) item + a set of look-ahead symbols:
$$A \mapsto \alpha.\beta, \mathcal{L}$$
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item  $C \mapsto .\gamma$  is added because  $A \mapsto \beta.C\delta, \mathcal{L}$  is already in the set, we need to compute its look-ahead set  $\mathcal{M}$ :
  1. The look-ahead set  $\mathcal{M}$  includes  $\text{FIRST}(\delta)$   
(the set of terminals that may start strings derived from  $\delta$ )
  2. If  $\delta$  is itself  $\epsilon$  or can derive  $\epsilon$  (i.e. it is nullable), then the look-ahead  $\mathcal{M}$  also contains  $\mathcal{L}$

# Example Closure

$$\begin{aligned} S' &\mapsto S\$ \\ S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

- Start item:  $S' \mapsto .S\$$  ,  $\{\}$
- Since  $S$  is to the right of a '.', add:  
 $S \mapsto .E + S$  ,  $\{\$\}$       Note:  $\{\$\}$  is  $\text{FIRST}(\$)$   
 $S \mapsto .E$  ,  $\{\$\}$
- Need to keep closing, since  $E$  appears to the right of a '.' in ' $.E + S$ ':  
 $E \mapsto .\text{number}$  ,  $\{+\}$       Note:  $+$  added for reason 1  
 $E \mapsto .( S )$  ,  $\{+\}$        $\text{FIRST}(+ S) = \{+\}$
- Because  $E$  also appears to the right of '.' in ' $.E$ ' we get:  
 $E \mapsto .\text{number}$  ,  $\{\$\}$       Note:  $\$\}$  added for reason 2  
 $E \mapsto .( S )$  ,  $\{\$\}$        $\delta$  is  $\epsilon$
- All items are distinct, so we're done

# Using the DFA



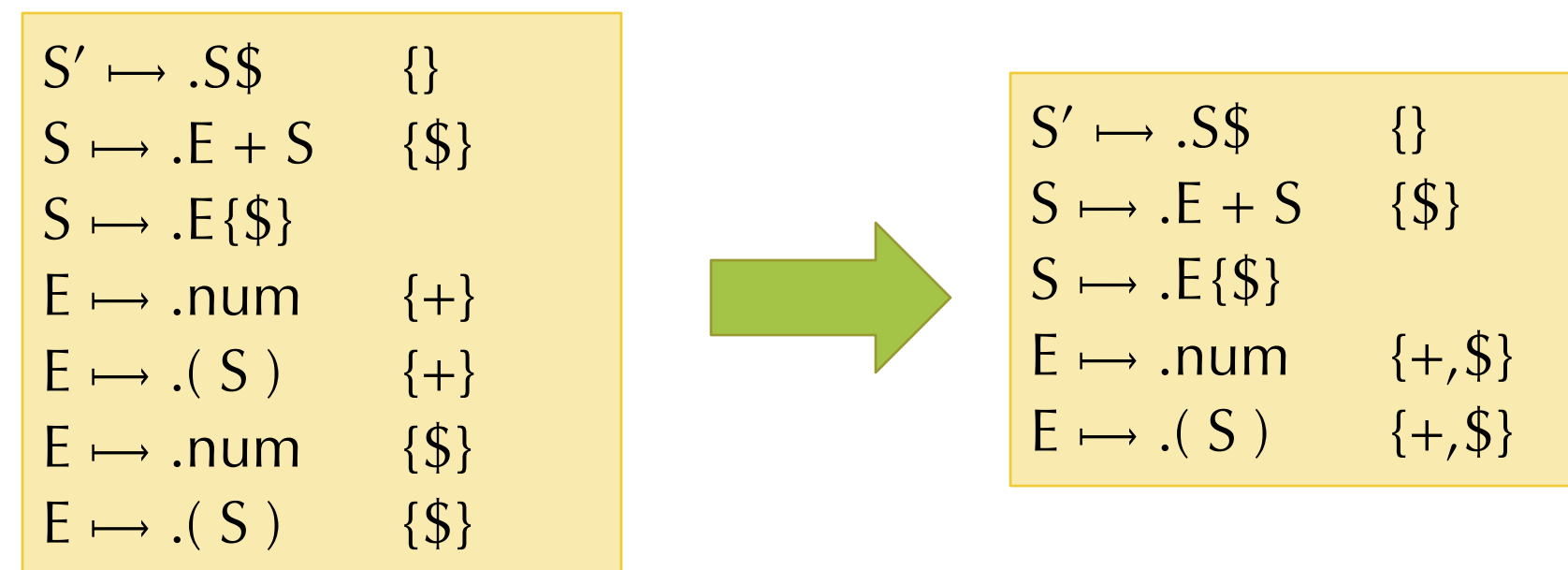
|   | +  | \$    | E  |
|---|----|-------|----|
| 1 |    |       | g2 |
| 2 | s3 | S → E |    |

Fragment of the Action & Goto tables

- The behavior is determined if:
  - There is no overlap among the look-ahead sets for each reduce item, and
  - None of the look-ahead symbols appear to the right of a ‘.’

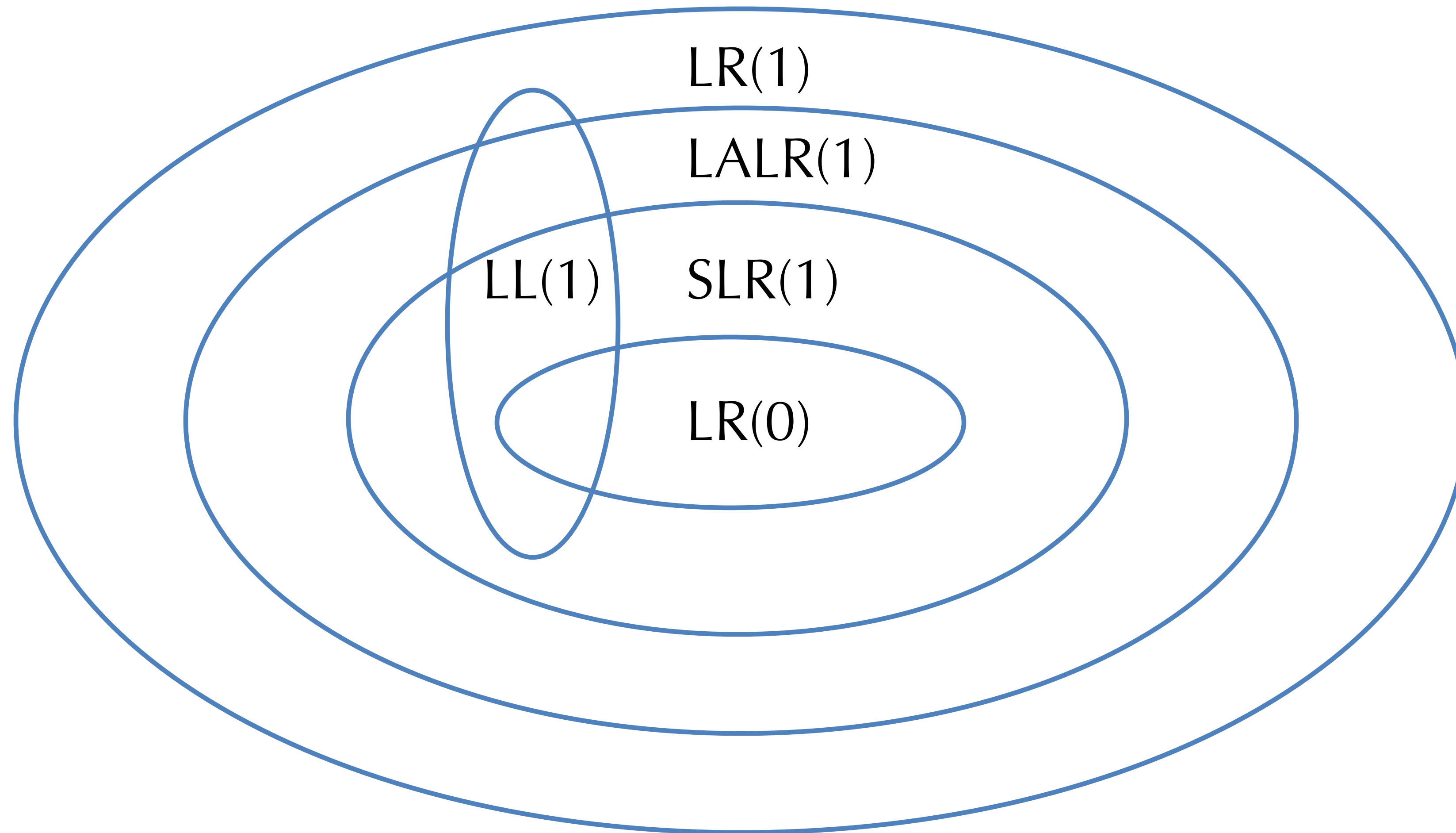
# LR variants


- LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  - DFA + stack is a push-down automaton
- In practice, LR(1) tables are big.
  - Some modern implementations (e.g., menhir) directly generate code rather than use tables
- LALR(1) = “Look-ahead LR”
  - Merge any two LR(1) states whose items are identical except for the look-ahead sets:



- Such merging can lead to nondeterminism (e.g., reduce/reduce conflicts), but
- Results in a much smaller parse table and works well in practice
- This is the usual technology for automatic parser generators: yacc, ocamllyacc
- GLR = “Generalized LR” parsing
  - Efficiently compute the set of *all* parses for a given input
  - Later passes should disambiguate based on other context

# Classification of Grammars





Debugging parser conflicts.  
Disambiguating grammars.

# LALRPOP DEMO