EECS 483: Compiler Construction

Lecture 24:
LR Grammars, Bottom-Up Parsing

April 14
Winter Semester 2025

Announcements

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

Final Exam Info

- Final Exam on Wednesday 04/30 4-6pm
  - Location:
    DOW 1010 (uniqname starts with A-L)
    DOW 1014 (uniqname starts with M-Z)
  - Topics: Assignments 4 and 5, lecture material after spring break.
  - Exam Review on Monday, 04/21
  - 1 page of notes, double sided ok, printed or written ok.
  - Practice material:
    https://maxsnew.com/teaching/eecs-483-wn24/syllabus.html
    questions about lexing/parsing/analysis/optimization
    Appel book, Dragon book linked on webpage have exercises as well.

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, ocamlyacc, lalrpop, etc.)
  - Better error detection/recovery

Top-down vs. Bottom up

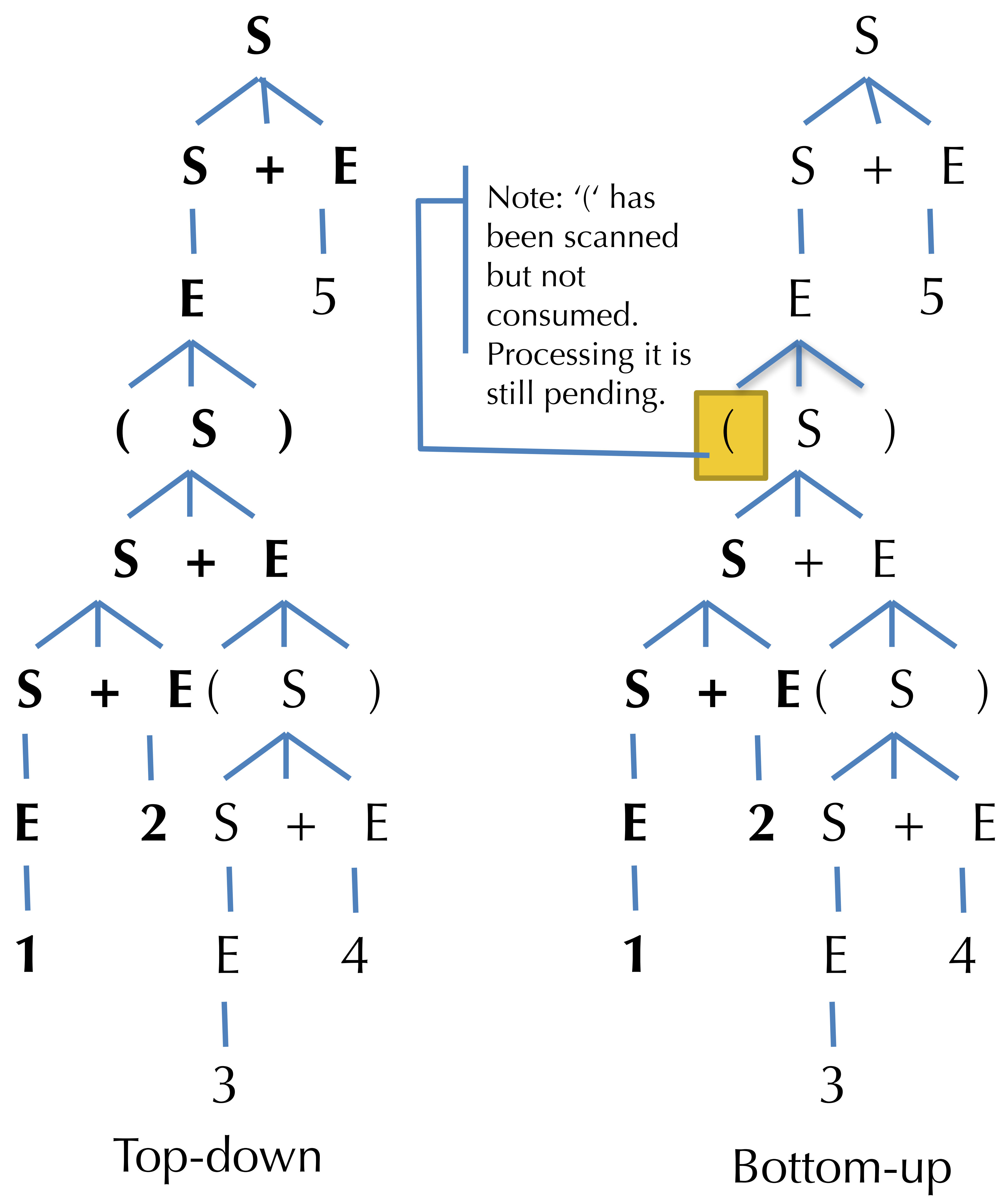
- Consider the left-recursive grammar:

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

- (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…



Progress of Bottom-up Parsing

Rightmost derivation

Reductions	Scanned	Input Remaining
(1 + 2 + (3 + 4)) + 5 $\longleftarrow$		(1 + 2 + (3 + 4)) + 5
(**E** + 2 + (3 + 4)) + 5 $\longleftarrow$	(	1 + 2 + (3 + 4)) + 5
(**S** + 2 + (3 + 4)) + 5 $\longleftarrow$	(1	+ 2 + (3 + 4)) + 5
(S + **E** + (3 + 4)) + 5 $\longleftarrow$	(1 + 2	+ (3 + 4)) + 5
(**S** + (3 + 4)) + 5 $\longleftarrow$	(1 + 2	+ (3 + 4)) + 5
(S + (**E** + 4)) + 5 $\longleftarrow$	(1 + 2 + (3	+ 4)) + 5
(S + (**S** + 4)) + 5 $\longleftarrow$	(1 + 2 + (3	+ 4)) + 5
(S + (S + **E**)) + 5 $\longleftarrow$	(1 + 2 + (3 + 4	)) + 5
(S + (**S**)) + 5 $\longleftarrow$	(1 + 2 + (3 + 4	)) + 5
(S + **E**) + 5 $\longleftarrow$	(1 + 2 + (3 + 4)	) + 5
(**S**) + 5 $\longleftarrow$	(1 + 2 + (3 + 4)	) + 5
E + 5 $\longleftarrow$	(1 + 2 + (3 + 4))	+ 5
S + 5 $\longleftarrow$	(1 + 2 + (3 + 4))	+ 5
S + **E** $\longleftarrow$	(1 + 2 + (3 + 4)) + 5	
S		

S $\longmapsto$ S + E | E
E $\longmapsto$ number | (S)

Shift/Reduce Parsing

$S \longmapsto S + E \ | \ E$
$E \longmapsto \text{number} \ | \ (\ S \ )$

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

Stack	Input	Action
	(1 + 2 + (3 + 4)) + 5	shift (
(	1 + 2 + (3 + 4)) + 5	shift 1
(1	+ 2 + (3 + 4)) + 5	reduce: $E \longmapsto$ number
(E	+ 2 + (3 + 4)) + 5	reduce: $S \longmapsto E$
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: $E \longmapsto$ number
(S + E	+ (3 + 4)) + 5	reduce: $S \longmapsto S + E$
(S	+ (3 + 4)) + 5	shift +

Shift/Reduce Parsing

$S \longmapsto S + E \ | \ E$
$E \longmapsto \text{number} \ | \ (S)$

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

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	(E + 2 + (3 + 4)) + 5
(S	+ 2 + (3 + 4)) + 5	(S + 2 + (3 + 4)) + 5
(S +	2 + (3 + 4)) + 5	
(S + 2	+ (3 + 4)) + 5	
(S + E	+ (3 + 4)) + 5	(S + E + (3 + 4)) + 5
(S	+ (3 + 4)) + 5	(S + (3 + 4)) + 5

Rightmost derivation

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 α as a finite parser state.
  - Parser state is computed by a DFA that reads the stack σ.
  - Accept states of the DFA correspond to unique reductions that apply.

- Example: LR(0) parsing
  - **L**eft-to-right scanning, **R**ight-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.

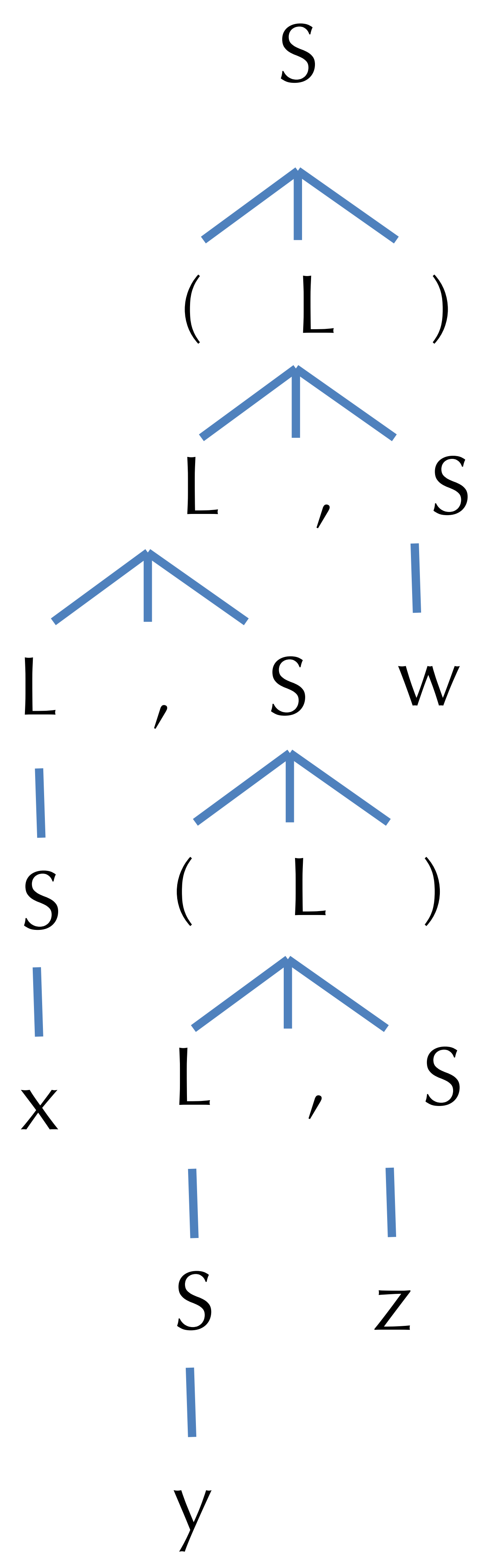
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

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

- Example strings:
  - x
  - (x,y)
  - ((((x))))
  - (x, (y, z), w)
  - (x, (y, (z, w)))

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



Shift/Reduce Parsing

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

$L \longmapsto S\ \mid\ L\ ,\ S$

- 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.

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

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

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

Example Run

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

Stack	Input	Action
	(x, (y, z), w)	shift (
(	x, (y, z), w)	shift x
(x	, (y, z), w)	reduce S ⟼ id
(S	, (y, z), w)	reduce L ⟼ S
(L	, (y, z), w)	shift ,
(L,	(y, z), w)	shift (
(L, (	y, z), w)	shift y
(L, (y	, z), w)	reduce S ⟼ id
(L, (S	, z), w)	reduce L ⟼ S
(L, (L	, z), w)	shift ,
(L, (L,	z), w)	shift z
(L, (L, z	), w)	reduce S ⟼ id
(L, (L, S	), w)	reduce L ⟼ L, S
(L, (L	), w)	shift)
(L, (L)	, w)	reduce S ⟼ (L)
(L, S	, w)	reduce L ⟼ L, S
(L	, w)	shift ,
(L,	w)	shift w
(L, w	)	reduce S ⟼ id
(L, S	)	reduce L ⟼ L, S
(L	)	shift)
(L)		reduce S ⟼ (L)
S		

Action Selection Problem

- Given a stack σ and a look-ahead symbol b, should the parser:
  - Shift b onto the stack (new stack is σb)
  - Reduce a production $X \longmapsto \gamma$, assuming that $\sigma = \alpha\gamma$ (new stack is αX)?

- Sometimes the parser can reduce but shouldn't
  - For example, $X \longmapsto \varepsilon$ can *always* be reduced
- Sometimes the stack can be reduced in different ways

- Main idea: decide what to do based on a *prefix* α of the stack plus the look-ahead symbol.
  - The prefix α is different for different possible reductions since in productions $X \longmapsto \gamma$ and $Y \longmapsto \beta$, γ and β 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

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

- 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 γ's to be reduced)
  - Stuff after the '.' is what might be seen next
  - The prefixes α are represented by the state itself

Constructing the DFA: Start state & Closure

$$S' \longmapsto S\$$$
$$S \longmapsto (\ L\ )\ \mid\ \mathrm{id}$$
$$L \longmapsto S\ \mid\ L\ ,\ S$$

- First step: Add a new production $S' \longmapsto S\$$ to the grammar
- Start state of the DFA = empty stack, so it contains the item:
  $S' \longmapsto .S\$$
- 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: CLOSURE($\{S' \longmapsto .S\$\}$) = $\{S' \longmapsto .S\$,\ S \longmapsto .(L),\ S \longmapsto .\mathrm{id}\}$
- Resulting "closed state" contains the set of all possible productions that might be reduced next.

Example: Constructing the DFA

$S' \longmapsto .S\$$

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

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

Example: Constructing the DFA

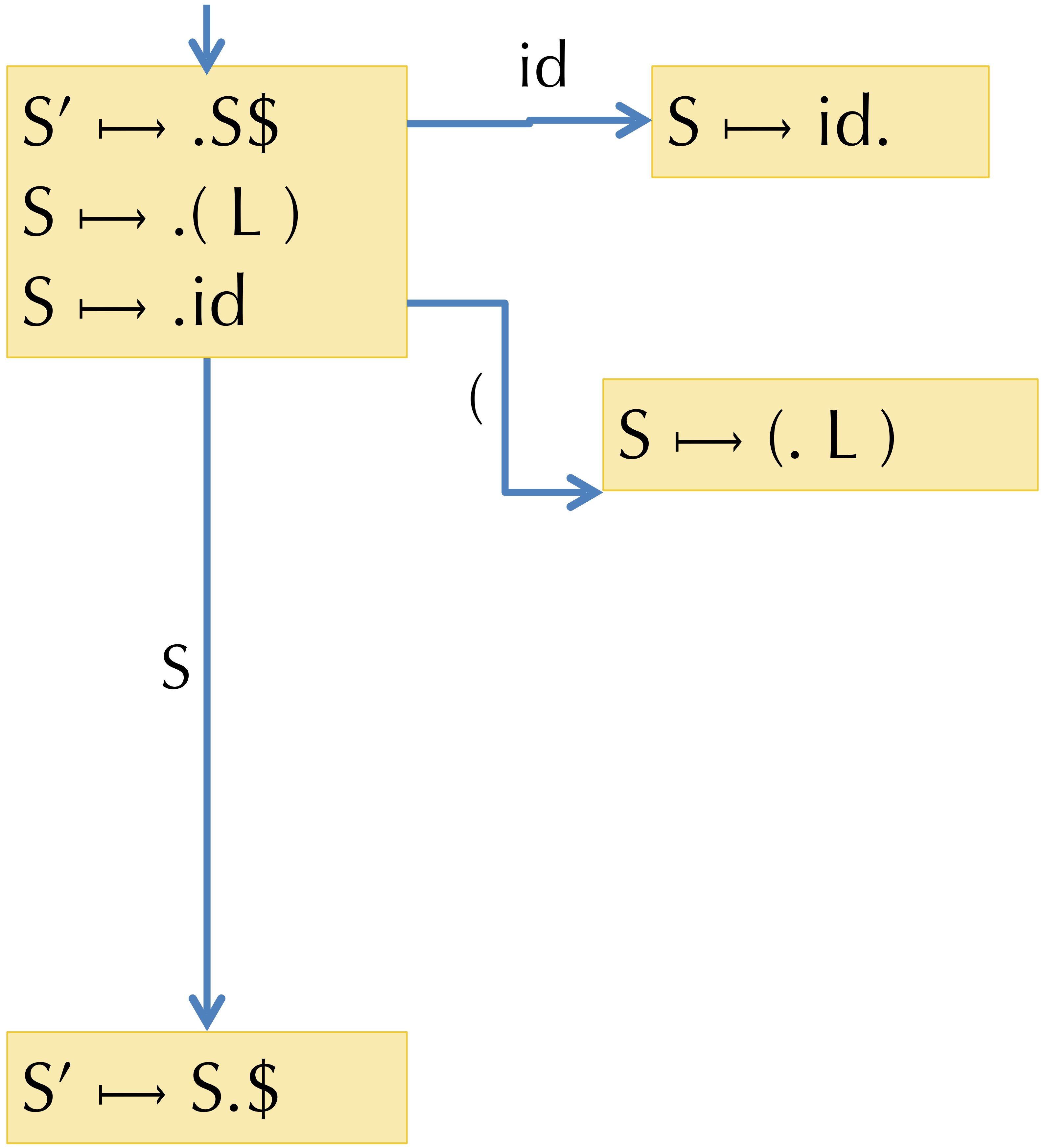
$S' \longmapsto .S\$$
$S \longmapsto .(\ L\ )$
$S \longmapsto .id$

$S' \longmapsto S\$$
$S \longmapsto (\ L\ )\ |\ id$
$L \longmapsto S\ |\ L\ ,\ S$

- Next, we take the closure of that state:
  CLOSURE($\{S' \longmapsto .S\$\}$) = $\{S' \longmapsto .S\$,\ S \longmapsto .(\ L\ ),\ S \longmapsto .id\}$

- 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′ ⟼ S$
S ⟼ (L) | id
L ⟼ S | L , S

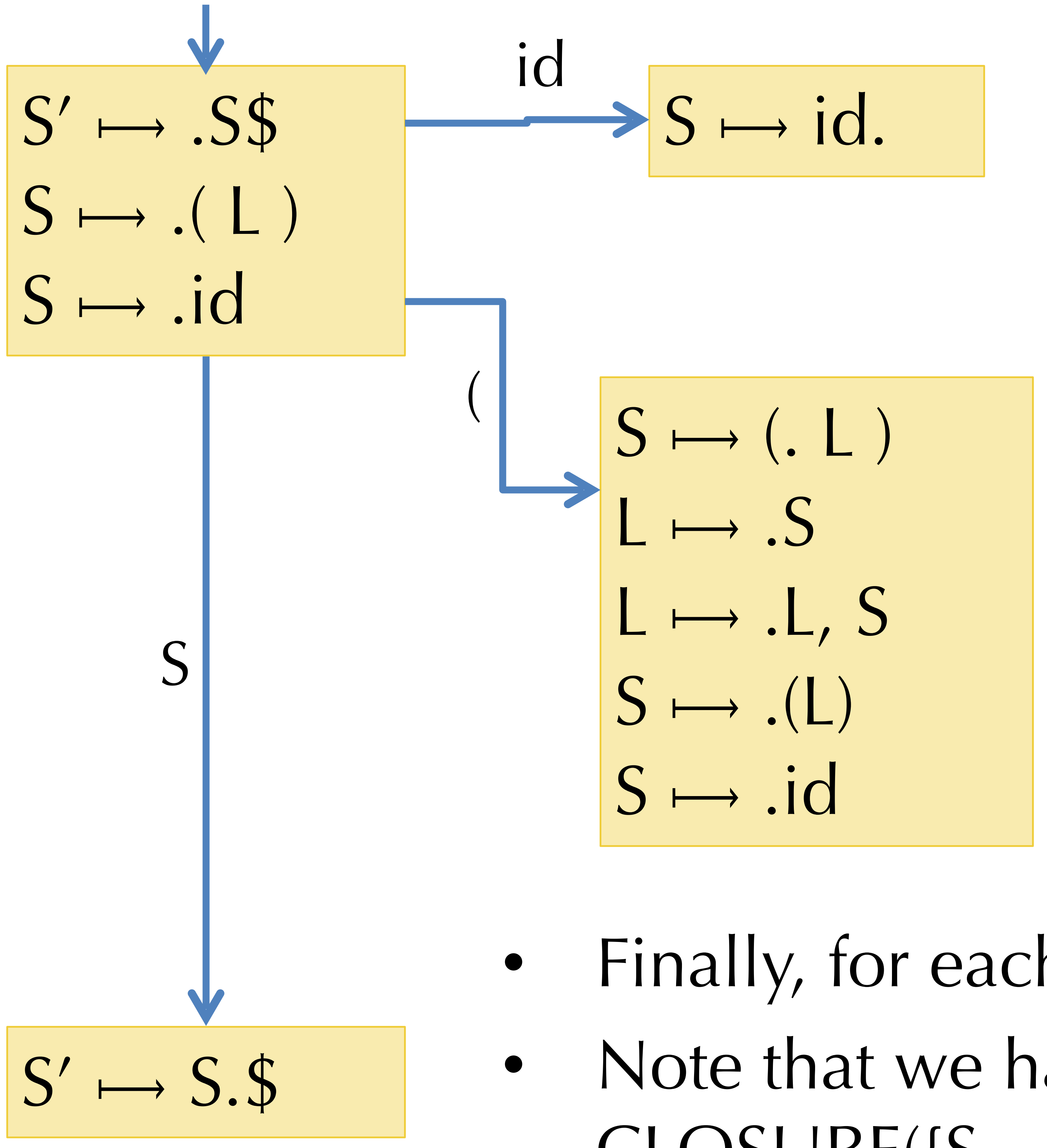
- Next we add the transitions:
- First, we see what terminals and nonterminals can appear after the '.' in the source state.
  - Outgoing edges have those label.
- The target state (initially) includes all items from the source state that have the edge-label symbol after the '.', but we advance the '.'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

Example: Constructing the DFA

$S' \longmapsto S\$$

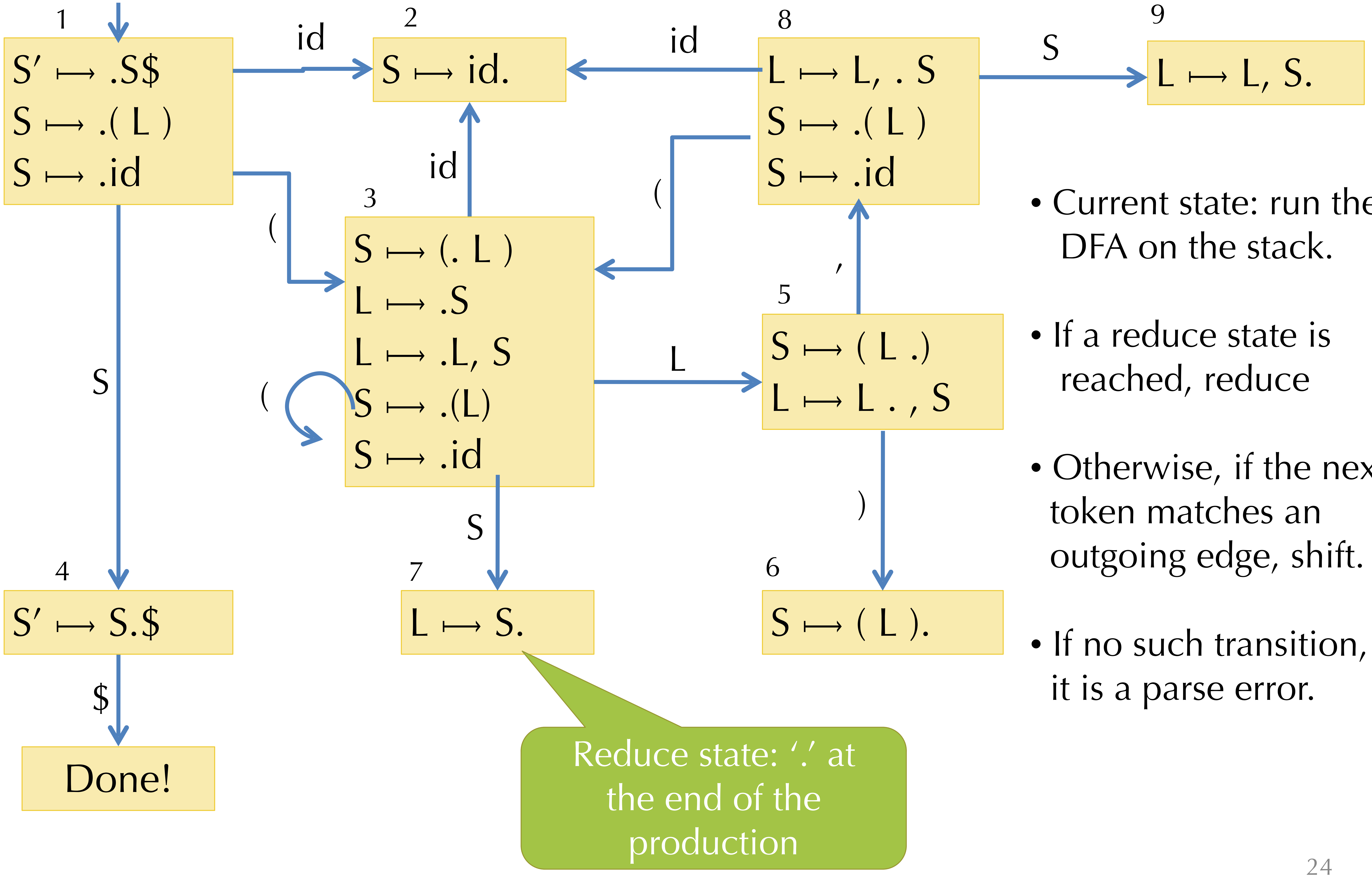
$S \longmapsto (\ L\ )\ \ |\ \ id$

$L \longmapsto S\ \ |\ \ L\ ,\ S$



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

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, $X \longmapsto \gamma$ with stack $\alpha\gamma$, pop γ 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 $X \longmapsto \gamma$, pop stack to reveal the state too: e.g. From stack ${}_1({}_3({}_3L_5)_6$ reduce $S \longmapsto (\ L\ )$ 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 γ off the stack to reveal the state
    - Look up X in the "goto table" and goto that state

Terminal Symbols

Nonterminal Symbols

State

Action table

Example Parse Table

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S⟼id	S⟼id	S⟼id	S⟼id	S⟼id		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	S ⟼ (L)	S ⟼ (L)	S ⟼ (L)	S ⟼ (L)	S ⟼ (L)		
7	L ⟼ S	L ⟼ S	L ⟼ S	L ⟼ S	L ⟼ S		
8	s3		s2			g9	
9	L ⟼ L,S	L ⟼ L,S	L ⟼ L,S	L ⟼ L,S	L ⟼ 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)
ε_1	(x, (y, z), w)$	s3
$\varepsilon_1(_3$	x, (y, z), w)$	s2
$\varepsilon_1(_3x_2$	, (y, z), w)$	Reduce: S⟼id
$\varepsilon_1(_3S$	, (y, z), w)$	g7 (from state 3 follow S)
$\varepsilon_1(_3S_7$	, (y, z), w)$	Reduce: L⟼S
$\varepsilon_1(_3L$	, (y, z), w)$	g5 (from state 3 follow L)
$\varepsilon_1(_3L_5$	, (y, z), w)$	s8
$\varepsilon_1(_3L_5,_8$	(y, z), w)$	s3
$\varepsilon_1(_3L_5,_8(_3$	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	shift/reduce	reduce/reduce
$S \longmapsto (L).$	$S \longmapsto (L).$ $L \longmapsto .L , S$	$S \longmapsto L ,S.$ $S \longmapsto ,S.$

- 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	right
$S \longmapsto S + E \mid E$ $E \longmapsto \text{number} \mid (S)$	$S \longmapsto E + S \mid E$ $E \longmapsto \text{number} \mid (S)$

- 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.

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

- 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

- 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:

    $A \longmapsto \alpha.\beta \ , \ \mathcal{L}$

- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item $C \longmapsto .\gamma$ is added because $A \longmapsto \beta.C\delta \ , \ \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$:

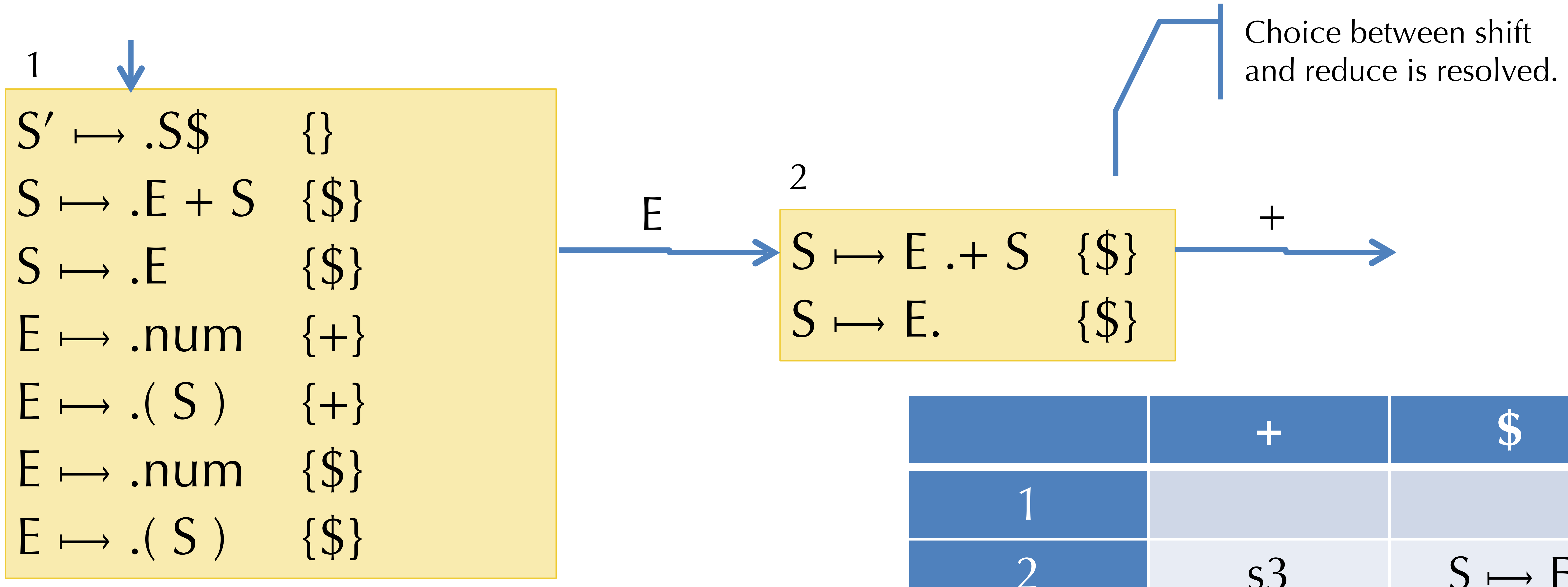
1. The look-ahead set $\mathcal{M}$ includes FIRST(δ)
   (the set of terminals that may start strings derived from δ)
2. If δ is itself ε or can derive ε (i.e. it is nullable), then the look-ahead $\mathcal{M}$ also contains $\mathcal{L}$

Example Closure

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

- Start item: $S' \longmapsto .S\$ \quad , \quad \{\}$
- Since S is to the right of a '.', add:
  - $S \longmapsto .E + S \quad , \quad \{\$\}$ — Note: $\{\$\}$ is FIRST($)
  - $S \longmapsto .E \quad , \quad \{\$\}$
- Need to keep closing, since E appears to the right of a '.' in '.E + S':
  - $E \longmapsto .\text{number} \quad , \quad \{+\}$ — Note: + added for reason 1
  - $E \longmapsto .(S) \quad , \quad \{+\}$ — FIRST(+ S) = {+}
- Because E also appears to the right of '.' in '.E' we get:
  - $E \longmapsto .\text{number} \quad , \quad \{\$\}$ — Note: $ added for reason 2
  - $E \longmapsto .(S) \quad , \quad \{\$\}$ — δ is ε
- 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.
  - Modern implementations (*e.g.*, menhir) directly generate code

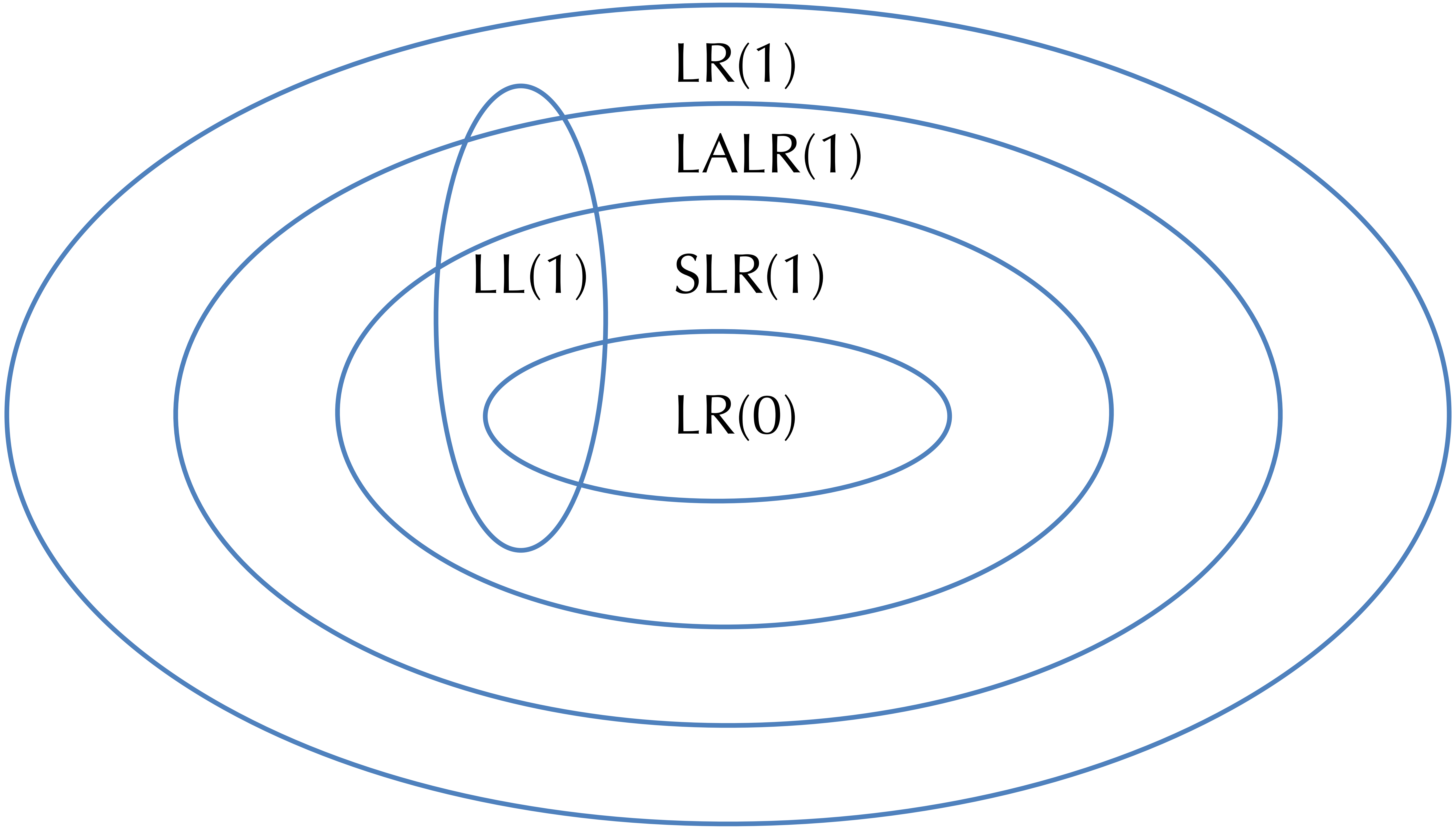
- LALR(1) = "Look-ahead LR"
  - Merge any two LR(1) states whose items are identical except for the look-ahead sets:

S′ ⟼ .S$	{}
S ⟼ .E + S	{$}
S ⟼ .E{$}	
E ⟼ .num	{+}
E ⟼ .(S)	{+}
E ⟼ .num	{$}
E ⟼ .(S)	{$}

S′ ⟼ .S$	{}
S ⟼ .E + S	{$}
S ⟼ .E{$}	
E ⟼ .num	{+,$}
E ⟼ .(S)	{+,$}

  - 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, ocamlyacc
- 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