EECS 483: Compiler Construction

Lecture 23:
LL(1) Grammars, Top-Down Parsing, Bottom-up Parsing I

April 8
Winter Semester 2026

Learning Objectives

What is top-down/predictive parsing?

How can we determine when a grammar is LL(1)?

How can we rewrite grammars to fit the LL(1) restriction?

How can we implement a top-down parser for an LL(1) grammar?

What is bottom-up/shift-reduce parsing?

Why is bottom-up parsing more powerful than top-down?

Basics of LR(0) (time-permitting)

Searching for derivations.

LL & LR PARSING

CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a token or ε)
  - A set of *nonterminals* (e.g., S and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of productions: LHS $\longmapsto$ RHS
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals

- Example: The balanced parentheses language:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

Derivations in CFGs

- Example: derive (1 + 2 + (3 + 4)) + 5
- <u>**S**</u> ⟼ <u>**E**</u> + S

⟼ (<u>**S**</u>) + S

⟼ (<u>**E**</u> + S) + S

⟼ (1 + <u>**S**</u>) + S

⟼ (1 + <u>**E**</u> + S) + S

⟼ (1 + 2 + <u>**S**</u>) + S

⟼ (1 + 2 + <u>**E**</u>) + S

⟼ (1 + 2 + (<u>**S**</u>)) + S

⟼ (1 + 2 + (<u>**E**</u> + S)) + S

⟼ (1 + 2 + (3 + <u>**S**</u>)) + S

⟼ (1 + 2 + (3 + <u>**E**</u>)) + S

⟼ (1 + 2 + (3 + 4)) + <u>**S**</u>

⟼ (1 + 2 + (3 + 4)) + <u>**E**</u>

⟼ (1 + 2 + (3 + 4)) + 5

S ⟼ E + S | E
E ⟼ number | (S)

For arbitrary strings α, β, γ and production rule A ⟼ β
a single step of the derivation is:

$$\alpha A \gamma \longmapsto \alpha\beta\gamma$$

(*substitute* β for an occurrence of A)

In general, there are many possible derivations for a given string

Note: Underline indicates symbol being expanded.

Example: Left- and rightmost derivations

$S \longmapsto E + S \mid E$

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

- Leftmost derivation:
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$

$\longmapsto (\underline{\mathbf{S}}) + S$

$\longmapsto (\underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$

$\longmapsto (1 + 2 + (3 + 4)) + 5$

Rightmost derivation:

$\underline{\mathbf{S}} \longmapsto E + \underline{\mathbf{S}}$

$\longmapsto E + \underline{\mathbf{E}}$

$\longmapsto \underline{\mathbf{E}} + 5$

$\longmapsto (\underline{\mathbf{S}}) + 5$

$\longmapsto (E + \underline{\mathbf{S}}) + 5$

$\longmapsto (E + E + \underline{\mathbf{S}}) + 5$

$\longmapsto (E + E + \underline{\mathbf{E}}) + 5$

$\longmapsto (E + E + (\underline{\mathbf{S}})) + 5$

$\longmapsto (E + E + (E + \underline{\mathbf{S}})) + 5$

$\longmapsto (E + E + (E + \underline{\mathbf{E}})) + 5$

$\longmapsto (E + E + (\underline{\mathbf{E}} + 4)) + 5$

$\longmapsto (E + \underline{\mathbf{E}} + (3 + 4)) + 5$

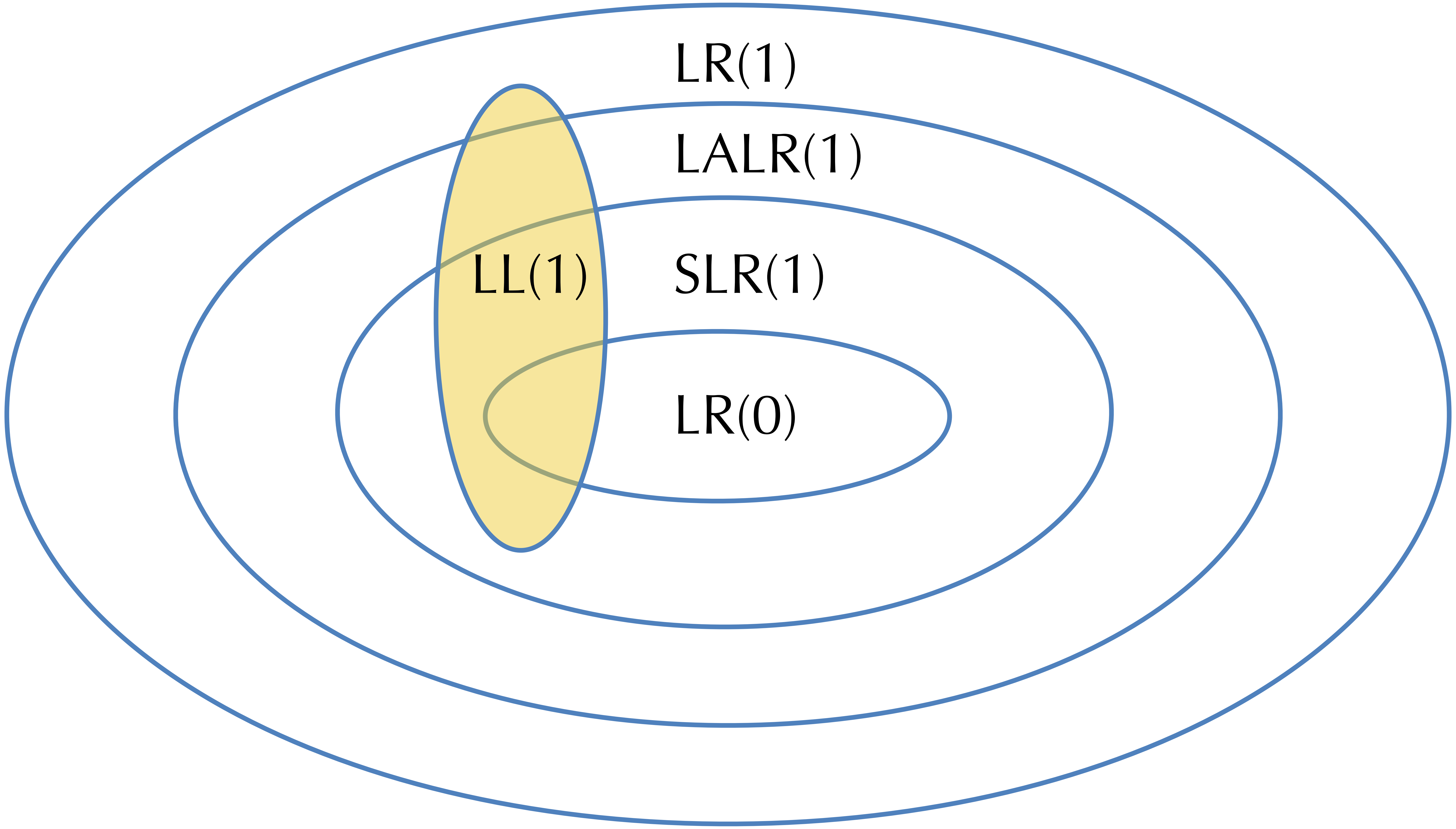
$\longmapsto (\underline{\mathbf{E}} + 2 + (3 + 4)) + 5$

$\longmapsto (1 + 2 + (3 + 4)) + 5$

CFGs In Practice

- Context-free Grammars are elegant, *declarative* specifications, generalizing regular expressions
- A parser for a CFG amounts to a *search procedure* for derivations
- Unlike regular expressions, which are easily compiled to linear time recognizers, practical algorithms for parsing *general* CFGs are O(n^3) in input string length
  - Compromise: add restrictions to the CFGs
    - Benefit: Linear time
    - Drawback: have to rewrite the grammar to make it fit the restrictions

Classification of Grammars



LL(1) GRAMMARS

Consider finding left-most derivations

- Look at only one input symbol at a time.

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

Partly-derived String	Look-ahead	Parsed/Unparsed Input
<u>S</u>	(	(1 + 2 + (3 + 4)) + 5
⟼ **<u>E</u>** + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (**<u>S</u>**) + S	1	(1 + 2 + (3 + 4)) + 5
⟼ (**<u>E</u>** + S) + S	1	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **<u>S</u>**) + S	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **<u>E</u>** + S) + S	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **<u>S</u>**) + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **<u>E</u>**) + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**<u>S</u>**)) + S	3	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**<u>E</u>** + S)) + S	3	(1 + 2 + (3 + 4)) + 5
⟼ …		

There is a problem

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

- We want to decide which production to apply based on the look-ahead symbol.
- But, there is a choice:

(1) $\quad S \longmapsto E \longmapsto (S) \longmapsto (E) \longmapsto (1)$

vs.

(1) + 2 $\quad S \longmapsto E + S \longmapsto (S) + S \longmapsto (E) + S \longmapsto (1) + S \longmapsto (1) + E \longmapsto (1) + 2$

- Given the look-ahead symbol: '(' it isn't clear whether to pick $S \longmapsto E$ or $S \longmapsto E + S$ first.

Grammar is the problem

- Not all grammars can be parsed "top-down" with only a single lookahead symbol.
- *Top-down*: starting from the start symbol (root of the parse tree) and going down

- LL(1) means
  - **L**eft-to-right scanning
  - **L**eft-most derivation,
  - **1** lookahead symbol

- This language isn't "LL(1)"
- Is it LL(k) for some k?

$S \longmapsto E + S \mid E$

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

- What can we do?

Making a grammar LL(1)

- *Problem:* We can't decide which S production to apply until we see the symbol after the first expression.
- *Solution:* "Left-factor" the grammar. There is a common S prefix for each choice, so add a new non-terminal S′ at the decision point:

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

→

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

- Also need to eliminate left-recursion. Why?
- Consider:

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

LL(1) Parse of the input string

- Look at only one input symbol at a time.

$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto$ number | (S)

Partly-derived String	Look-ahead	Parsed/Unparsed Input
S	(	(1 + 2 + (3 + 4)) + 5
⟼ **E** S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (**S**) S′	1	(1 + 2 + (3 + 4)) + 5
⟼ (**E** S′) S′	1	(1 + 2 + (3 + 4)) + 5
⟼ (1 **S′**) S′	+	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **S**) S′	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + **E** S′) S′	2	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 **S′**) S′	+	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **S**) S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + **E** S′) S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**S**)S′) S′	3	(1 + 2 + (3 + 4)) + 5

Predictive Parsing

- Given an LL(1) grammar:
  - For a given nonterminal, the lookahead symbol uniquely determines the production to apply.
  - Top-down parsing = predictive parsing
  - Driven by a predictive parsing table:
    nonterminal * input token → production

T ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num		⟼ (S)		

- Note: it is convenient to add a special *end-of-file* token $ and a start symbol T (top-level) that requires $.

How do we construct the parse table?

- Consider a given production: A → γ
- Construct the set of all input tokens that may appear *first* in strings that can be derived from γ
  - Add the production → γ to the entry (A,token) for each such token.
- If γ can derive ε (the empty string), then we construct the set of all input tokens that may *follow* the nonterminal A in the grammar.
  - Add the production → γ to the entry (A, token) for each such token.

- Note: The grammar is LL(1) *if and only if* all entries have at most one production

Example

- First(T) = First(S)
- First(S) = First(E)
- First(S′) = { + }
- First(E) = { number, '(' }

- Follow(S′) = Follow(S)
- Follow(S) = { $, ')' } ∪ Follow(S′)

Note: we want the *least* solution to this system of set equations… a *fixpoint* computation. Just like in program analysis!

T ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (S)		

Converting the table to code

- Define n mutually recursive functions
  - one for each nonterminal A: parse_A
  - The type of parse_A is `() -> Result<AST,Error>` if A is *not* an auxiliary nonterminal
  - Parse functions for auxiliary nonterminals (e.g., S′) take extra ast's as inputs, one for each nonterminal in the "factored" prefix.

- Each function "peeks" at the lookahead token and then follows the production rule in the corresponding entry.
  - Consume terminal tokens from the input stream
  - Call parse_X to create sub-tree for nonterminal X
  - If the rule ends in an auxiliary nonterminal, call it with appropriate ast's. (The auxiliary rule is responsible for creating the ast after looking at more input.)
  - Otherwise, this function builds the ast tree itself and returns it.

	number	+	(	)	$ (EOF)
T	$\longmapsto$ S$		$\longmapsto$S$		
S	$\longmapsto$ E S′		$\longmapsto$E S′		
S′		$\longmapsto$ + S		$\longmapsto$ ε	$\longmapsto$ ε
E	$\longmapsto$ num.		$\longmapsto$ (S)		

Hand-generated LL(1) code for the table above.

DEMO: HANDPARSER.RS

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?

LR GRAMMARS

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

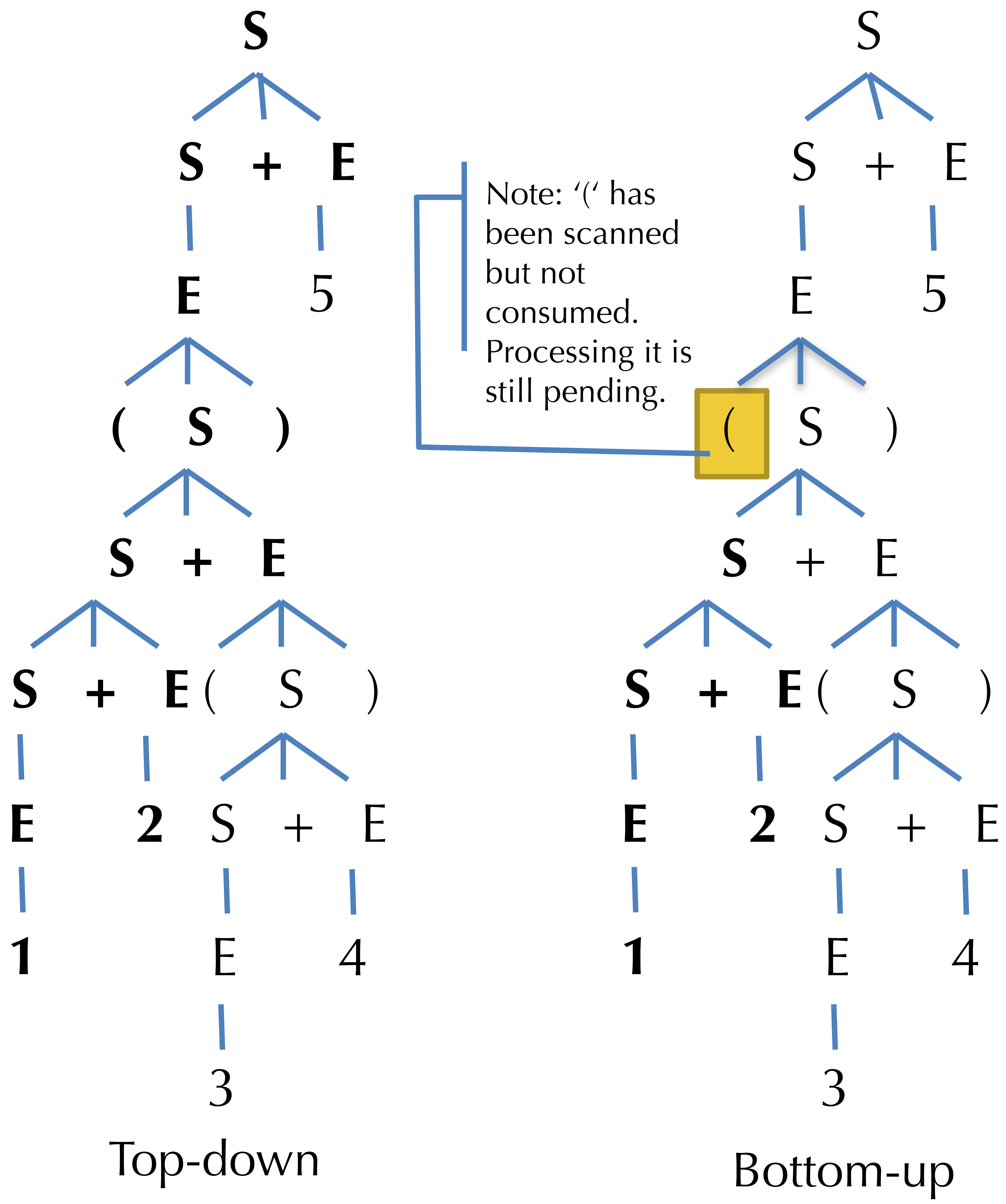
- Consider the left-recursive grammar:

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

- (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 ⟵		(1 + 2 + (3 + 4)) + 5
(**E** + 2 + (3 + 4)) + 5 ⟵	(	1 + 2 + (3 + 4)) + 5
(**S** + 2 + (3 + 4)) + 5 ⟵	(1	+ 2 + (3 + 4)) + 5
(S + **E** + (3 + 4)) + 5 ⟵	(1 + 2	+ (3 + 4)) + 5
(**S** + (3 + 4)) + 5 ⟵	(1 + 2	+ (3 + 4)) + 5
(S + (**E** + 4)) + 5 ⟵	(1 + 2 + (3	+ 4)) + 5
(S + (**S** + 4)) + 5 ⟵	(1 + 2 + (3	+ 4)) + 5
(S + (S + **E**)) + 5 ⟵	(1 + 2 + (3 + 4	)) + 5
(S + (**S**)) + 5 ⟵	(1 + 2 + (3 + 4	)) + 5
(S + **E**) + 5 ⟵	(1 + 2 + (3 + 4)	) + 5
(**S**) + 5 ⟵	(1 + 2 + (3 + 4)	) + 5
E + 5 ⟵	(1 + 2 + (3 + 4))	+ 5
S + 5 ⟵	(1 + 2 + (3 + 4))	+ 5
S + **E** ⟵	(1 + 2 + (3 + 4)) + 5	
S		

S ⟼ S + E | E
E ⟼ number | (S)

Shift/Reduce Parsing

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

- 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 ⟼ γ 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 ⟼ number
(E	+ 2 + (3 + 4)) + 5	reduce: S ⟼ E
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: E ⟼ number
(S + E	+ (3 + 4)) + 5	reduce: S ⟼ 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 stack + input
- Invariant: Stack plus input is a step in building the Rightmost derivation in reverse

S ⟼ S + E | E
E ⟼ number | (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	($\underline{E}$ + 2 + (3 + 4)) + 5
(S	+ 2 + (3 + 4)) + 5	($\underline{S}$ + 2 + (3 + 4)) + 5
(S +	2 + (3 + 4)) + 5	
(S + 2	+ (3 + 4)) + 5	
(S + E	+ (3 + 4)) + 5	(S + $\underline{E}$ + (3 + 4)) + 5
(S	+ (3 + 4)) + 5	($\underline{S}$ + (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.

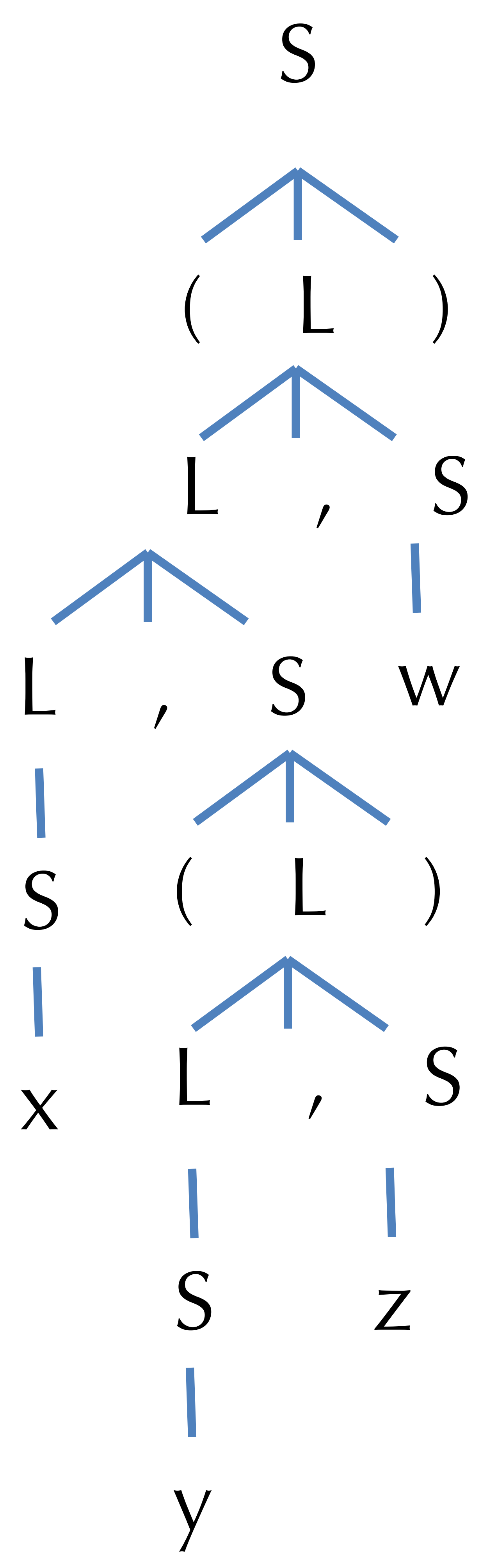
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

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

- 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

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		

S ⟼ (L) | id
L ⟼ 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

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

- Example items: $S \longmapsto .(\ L\ )$ or $S \longmapsto (.\ L)$ or $L \longmapsto S.$
- 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

- First step: Add a new production
  $S' \longmapsto S\$$ to the grammar

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

- 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($\{S' \longmapsto .S\$\}$) = $\{S' \longmapsto .S\$,\ S \longmapsto .(L),\ S \longmapsto .id\}$
- Resulting "closed state" contains the set of all possible productions that might be reduced next.

Example: Constructing the DFA

$S' \longmapsto .S\$$

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

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

Example: Constructing the DFA

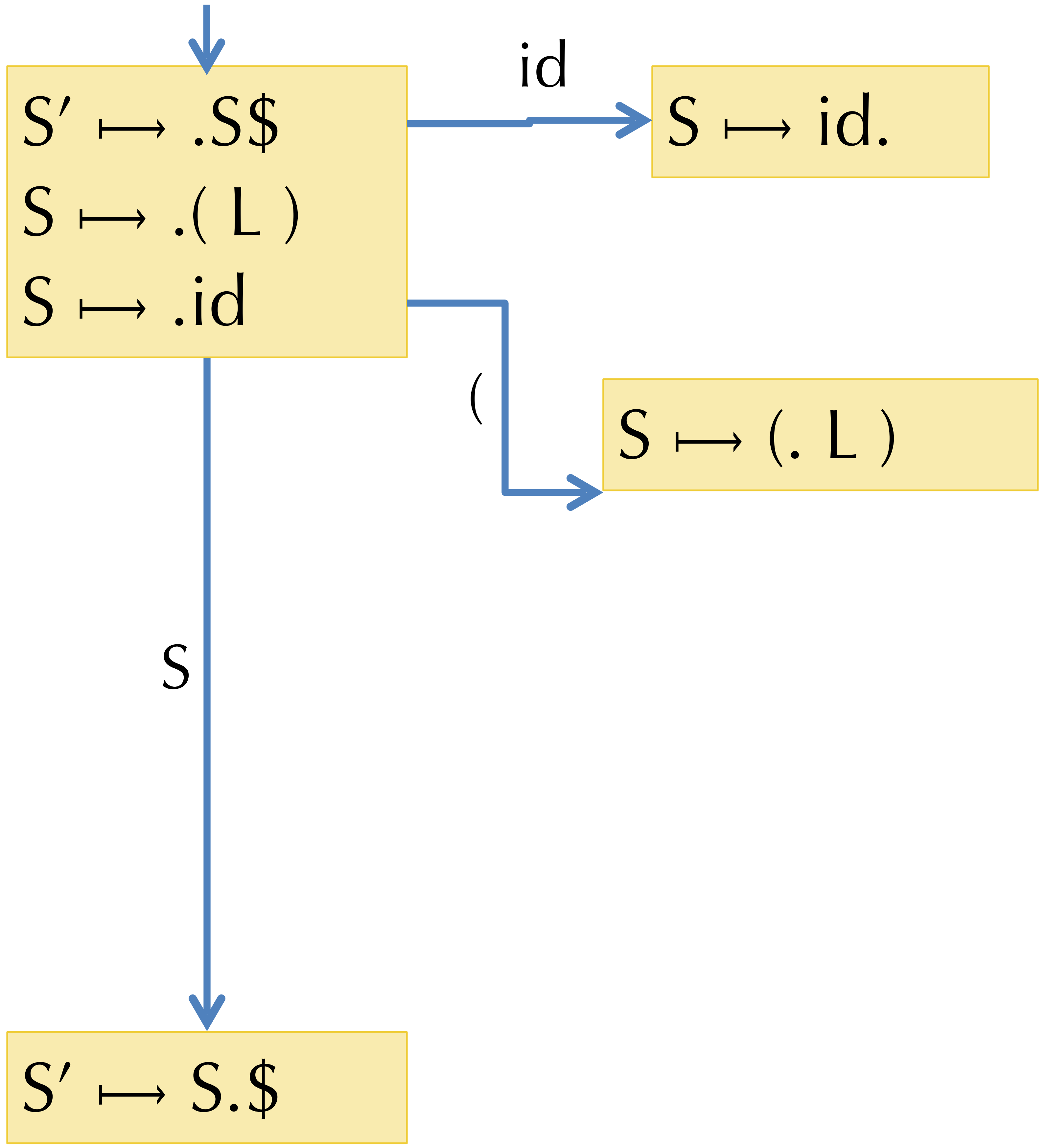
$S' \longmapsto .S\$$
$S \longmapsto .(\ L\ )$
$S \longmapsto .id$

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

- 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

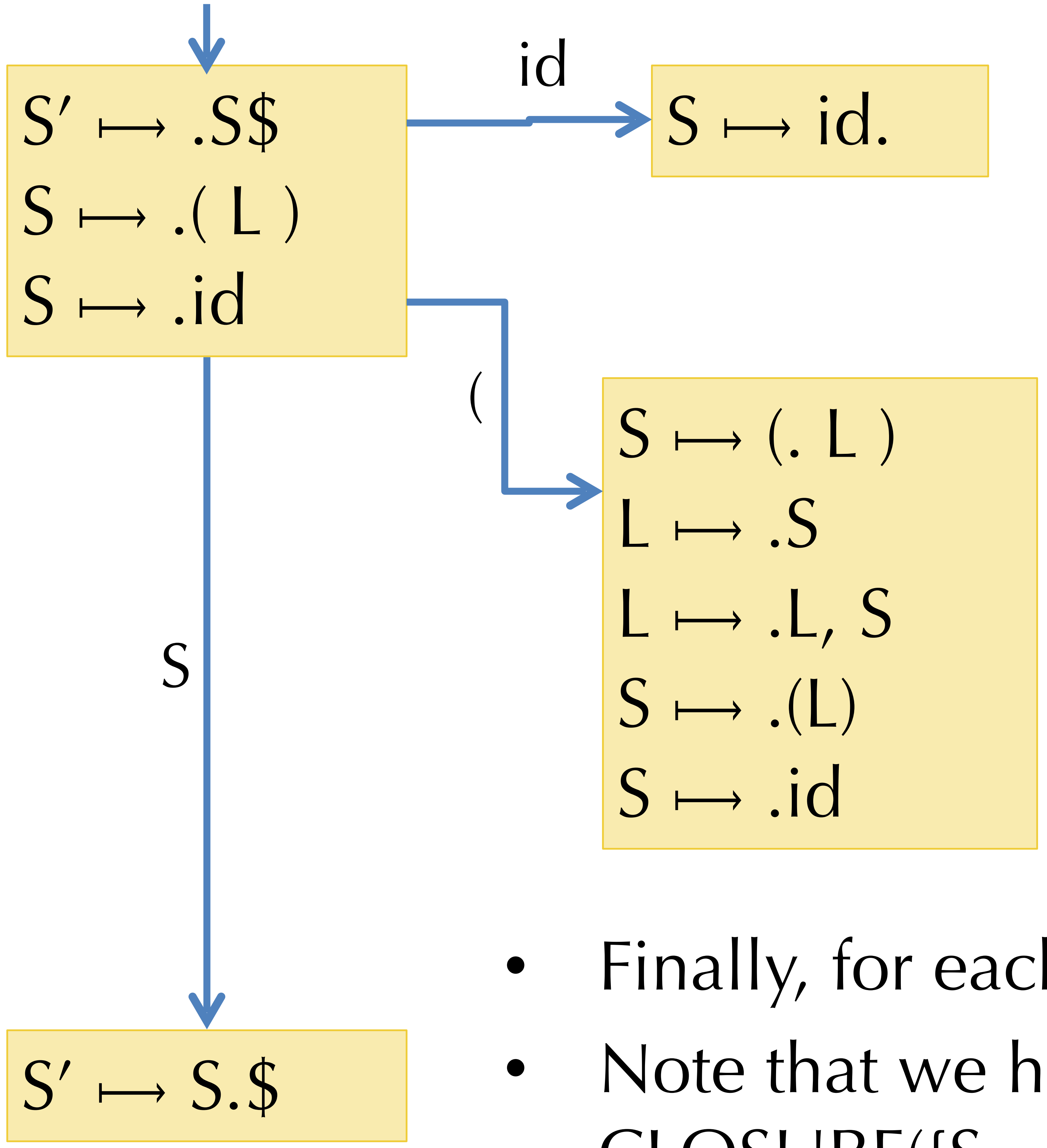
Example: Constructing the DFA



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)

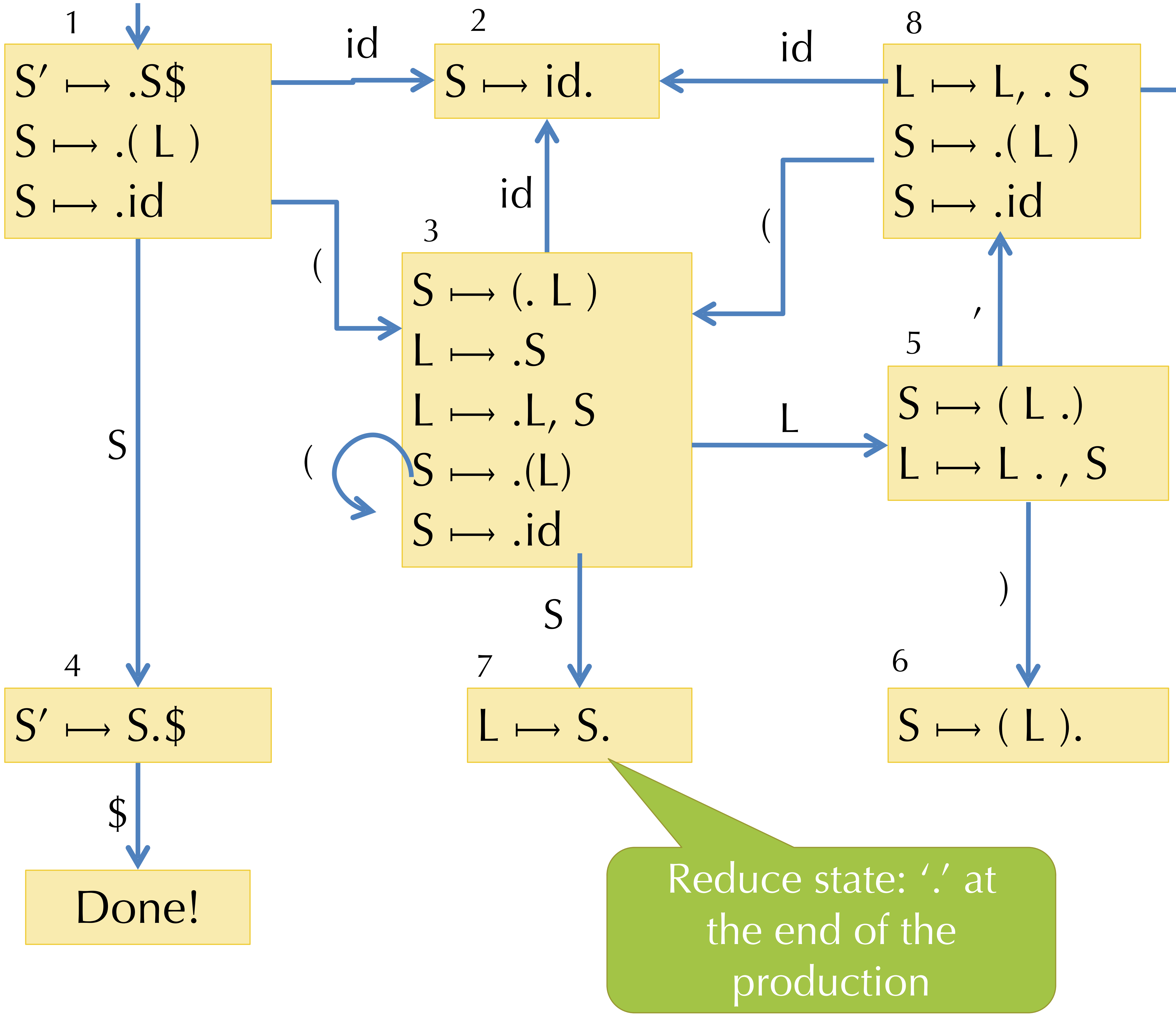
Example: Constructing the DFA



S′ ⟼ S$
S ⟼ (L) | id
L ⟼ S | L , S

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

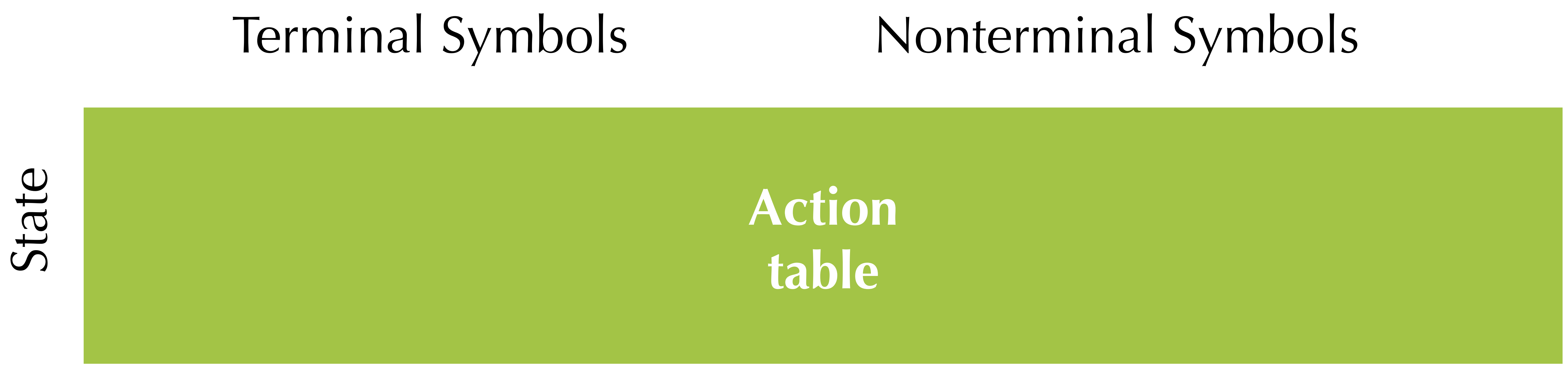
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. ${}_1({}_3({}_3L_5)_6$
  - On a reduction $X \longmapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack ${}_1({}_3({}_3L_5)_6$ reduce $S \longmapsto (L)$ to reach 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



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.

Examples

- Consider the left associative and right associative "sum" grammars:

left	right
$S \longmapsto S + E \mid E$ $E \longmapsto number \mid (S)$	$S \longmapsto E + S \mid E$ $E \longmapsto number \mid (S)$

- 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?

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

$$S \longmapsto E\ .+ S$$
$$S \longmapsto E.$$

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