Lecture 12

CIS 4521/5521: COMPILERS

Announcements

- HW3: LLVM Backend
  - Available on the course web pages.
  - Due: Weds., February 26th at 10:00PM
  - Note: test cases should be submitted **TONIGHT at 10pm**

- Midterm: March 6th
  - In class
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - Coverage: interpreters, x86, IRs, LLVM IR, calling conventions, lexing, parsing
  - See Ed post for previous exams

Creating an abstract representation of program syntax.

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 ⟼ RHS
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals

S ⟼ S + E | E

E ⟼ number | (S)

Example: Left- and rightmost derivations

- Leftmost Derivation
- **S** ⟼ **E** + S
  ⟼ (**S**) + S
  ⟼ (**E** + S) + S
  ⟼ (1 + **S**) + S
  ⟼ (1 + **E** + S) + S
  ⟼ (1 + 2 + **S**) + S
  ⟼ (1 + 2 + **E**) + S
  ⟼ (1 + 2 + (**S**)) + S
  ⟼ (1 + 2 + (**E** + S)) + S
  ⟼ (1 + 2 + (3 + **S**)) + S
  ⟼ (1 + 2 + (3 + **E**)) + S
  ⟼ (1 + 2 + (3 + 4)) + **S**
  ⟼ (1 + 2 + (3 + 4)) + **E**
  ⟼ (1 + 2 + (3 + 4)) + 5

S ⟼ S + E | E
E ⟼ number | (S)

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

$$
\begin{aligned}
T &\mapsto S\$ \\
S &\mapsto ES' \\
S' &\mapsto \varepsilon \\
S' &\mapsto + S \\
E &\mapsto \text{number} \mid (S)
\end{aligned}
$$

	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 $.

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, CUP, ocamlyacc, merlin, etc.)
  - Better error detection/recovery
  - But… harder to implement by hand

Top-down vs. Bottom up

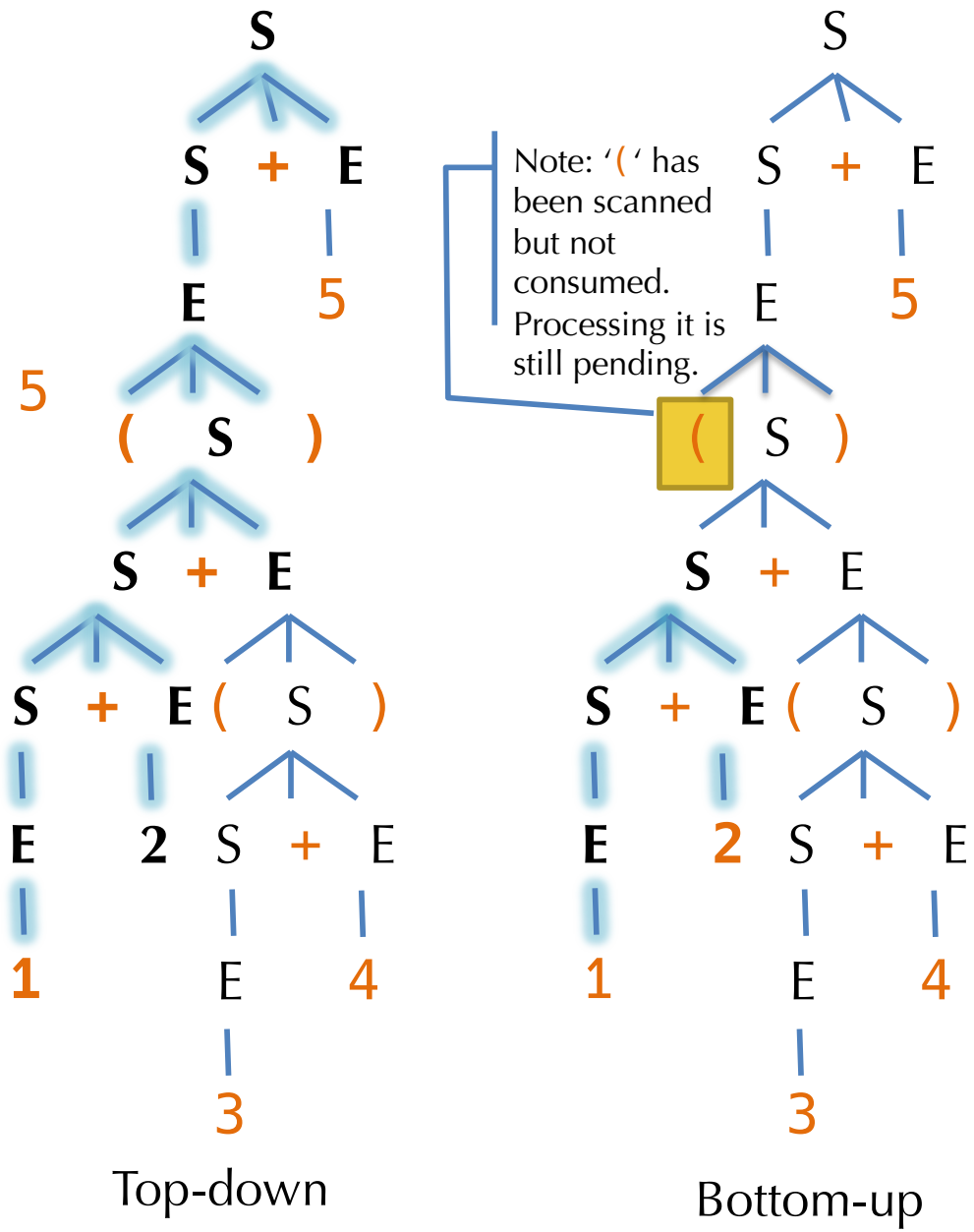
- Consider the left-recursive grammar:

  S ⟼ S + E | E
  E ⟼ number | (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…



Example: Left- and rightmost derivations

- Leftmost Derivation
- $\underline{\mathbf{S}} \mapsto \underline{\mathbf{E}} + S$
  $\mapsto (\underline{\mathbf{S}}) + S$
  $\mapsto (\underline{\mathbf{E}} + S) + S$
  $\mapsto (1 + \underline{\mathbf{S}}) + S$
  $\mapsto (1 + \underline{\mathbf{E}} + S) + S$
  $\mapsto (1 + 2 + \underline{\mathbf{S}}) + S$
  $\mapsto (1 + 2 + \underline{\mathbf{E}}) + S$
  $\mapsto (1 + 2 + (\underline{\mathbf{S}})) + S$
  $\mapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$
  $\mapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$
  $\mapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$
  $\mapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$
  $\mapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$
  $\mapsto (1 + 2 + (3 + 4)) + 5$

- Rightmost derivation:
- $\underline{\mathbf{S}} \mapsto E + \underline{\mathbf{S}}$
  $\mapsto E + \underline{\mathbf{E}}$
  $\mapsto \underline{\mathbf{E}} + 5$
  $\mapsto (\underline{\mathbf{S}}) + 5$
  $\mapsto (E + \underline{\mathbf{S}}) + 5$
  $\mapsto (E + E + \underline{\mathbf{S}}) + 5$
  $\mapsto (E + E + \underline{\mathbf{E}}) + 5$
  $\mapsto (E + E + (\underline{\mathbf{S}})) + 5$
  $\mapsto (E + E + (E + \underline{\mathbf{S}})) + 5$
  $\mapsto (E + E + (E + \underline{\mathbf{E}})) + 5$
  $\mapsto (E + E + (\underline{\mathbf{E}} + 4)) + 5$
  $\mapsto (E + \underline{\mathbf{E}} + (3 + 4)) + 5$
  $\mapsto (\underline{\mathbf{E}} + 2 + (3 + 4)) + 5$
  $\mapsto (1 + 2 + (3 + 4)) + 5$

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

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 ⟼ S + E | E
E ⟼ 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 ⟼ 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

parser.mly, lexer.mll, range.ml, ast.ml, main.ml

DEMO: BOOLEAN LOGIC

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

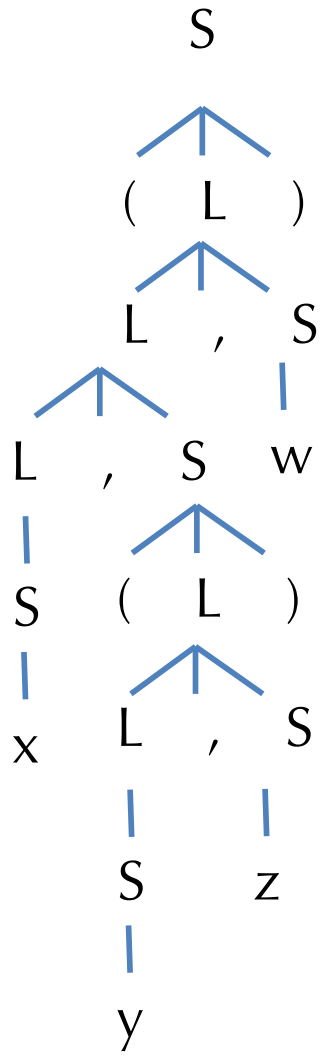
  (x,y)

  ((((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

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 ,

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 $X \mapsto \gamma$, assuming that $\sigma = \alpha\gamma$ (new stack is αX)?

- 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* α of the stack plus the look-ahead symbol.
  - The prefix α is different for different possible reductions since in productions $X \mapsto \gamma$ and $Y \mapsto \beta$, 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

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

- 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

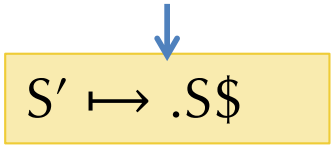
Constructing the DFA: Start state & Closure

- First step: Add a new production
  $S' \mapsto S\$$ 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: CLOSURE({$S' \mapsto .S\$$}) = {$S' \mapsto .S\$$, $S \mapsto .(L)$, $S \mapsto .id$}

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

- Resulting "closed state" contains the set of all possible productions that might be reduced next.

Example: Constructing the DFA

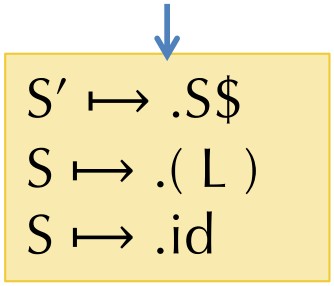


S′ ⟼ S$
S ⟼ (L) | id
L ⟼ S | L , S

- First, we construct a state with the initial item S′ ⟼ .S$

Example: Constructing the DFA

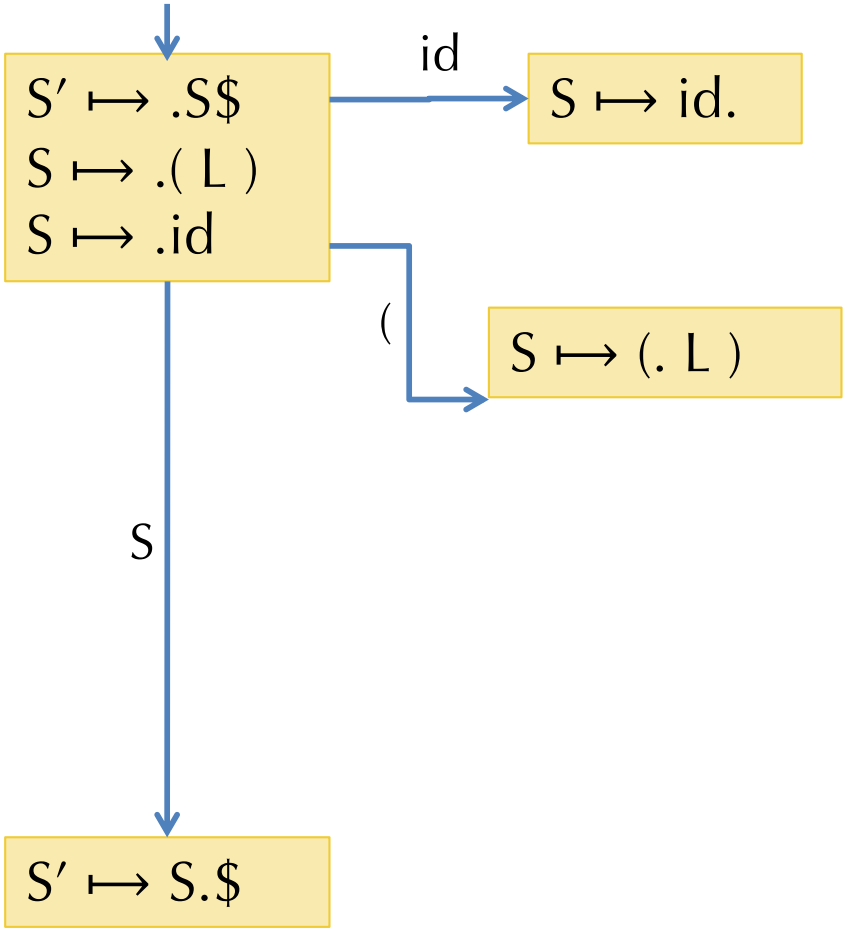
$S' \mapsto S\$$
$S \mapsto (L) \mid id$
$L \mapsto S \mid L , S$



- Next, we take the closure of that state:
  CLOSURE({S′ ⟼ .S$}) = {S′ ⟼ .S$, S ⟼ .(L), S ⟼ .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

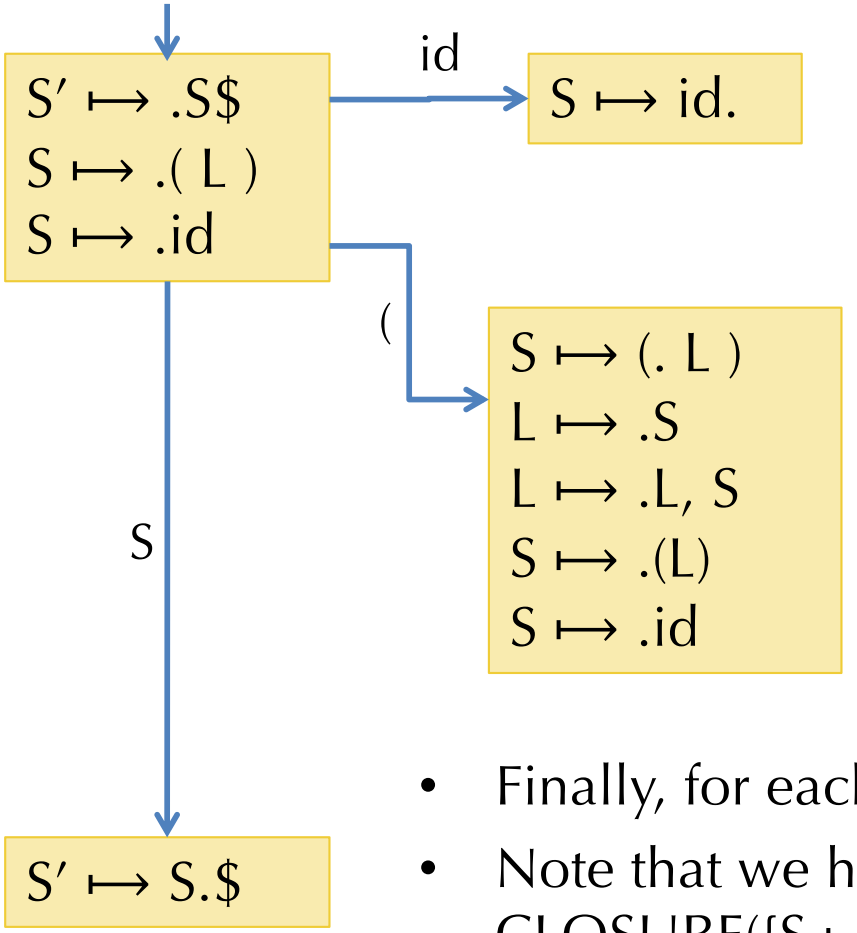


S′ ⟼ S$
S ⟼ (L) | id
L ⟼ S | L , S

- 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)

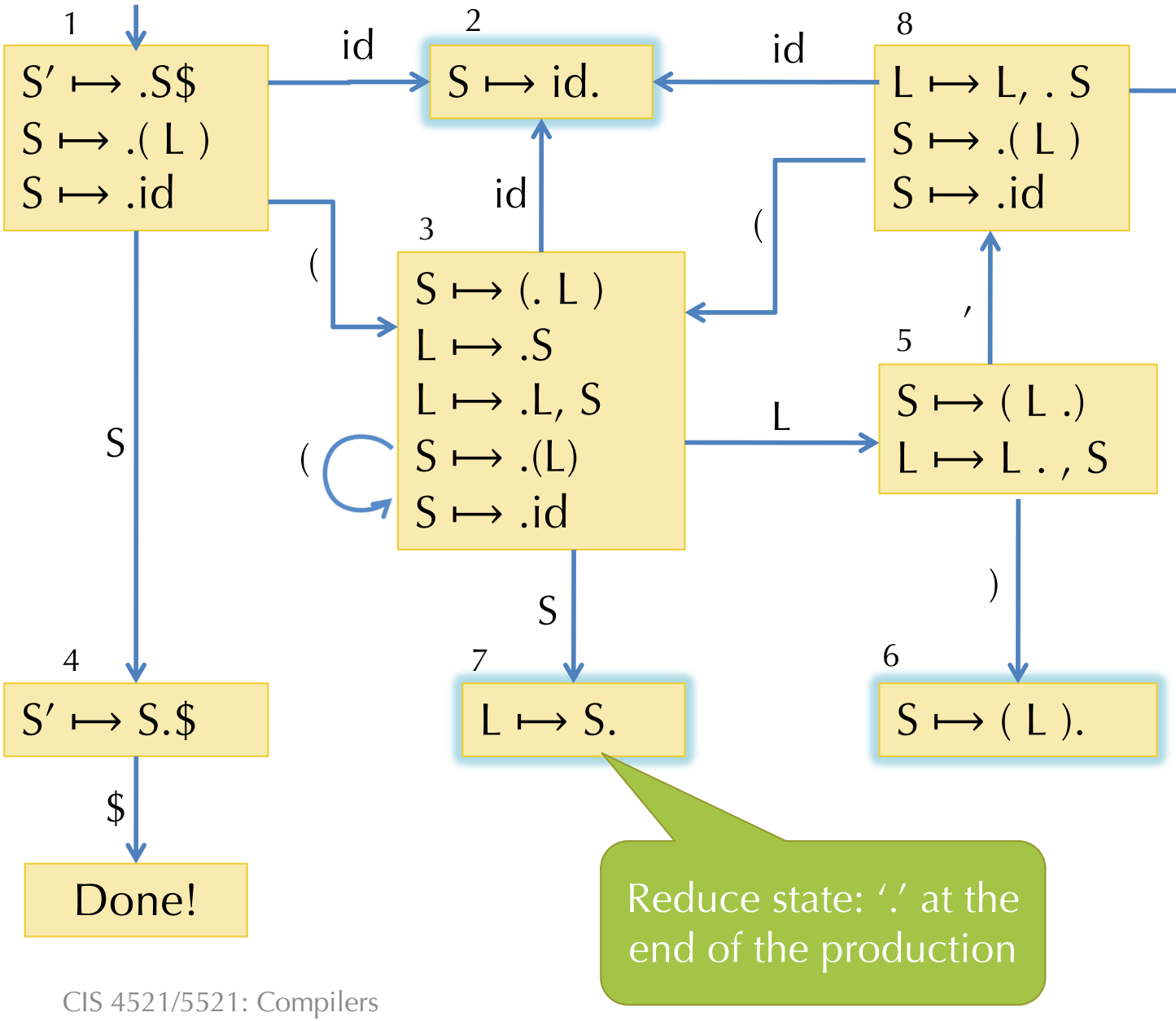
Example: Constructing the DFA

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



- Finally, for each new state, we take the closure.
- Note that we have to perform two iterations to compute CLOSURE({S ⟼ (. 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, $X \mapsto \gamma$ with stack $\alpha\gamma$, pop γ and push X.

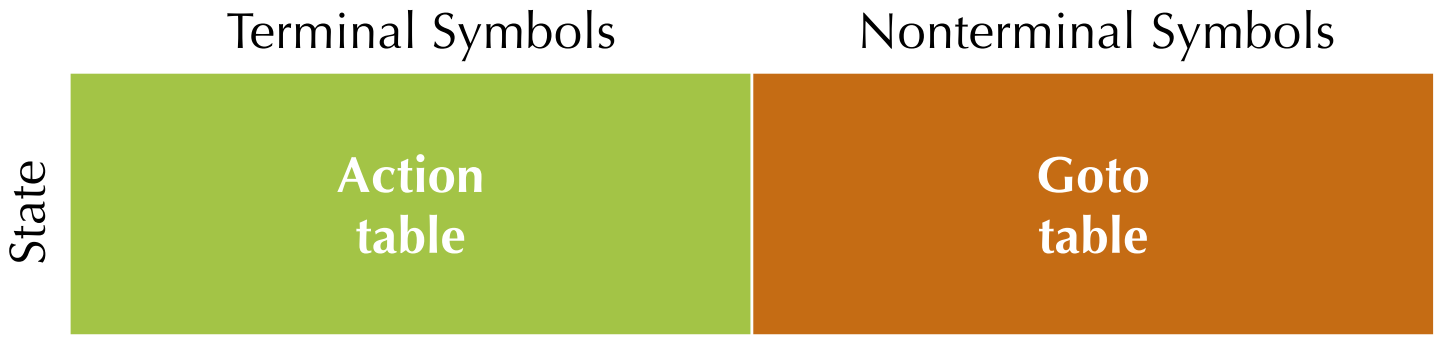
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. ${}_1({}_3({}_3L_5)_6$
  - On a reduction $X \mapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack ${}_1({}_3({}_3L_5)_6$ reduce $S \mapsto (\ L\ )$ to reach stack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

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