Lecture 7

CIS 341: COMPILERS

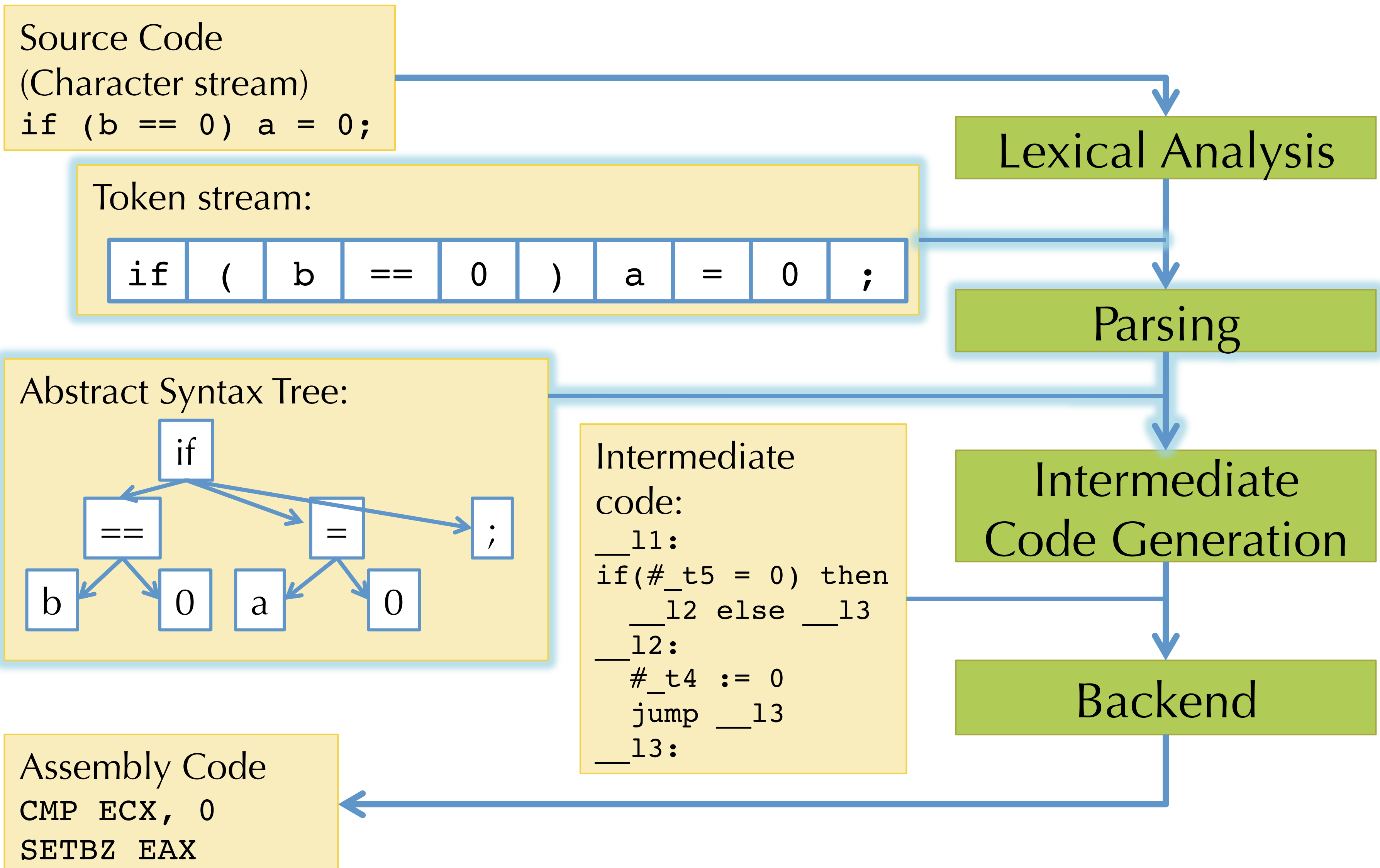
Announcements

- Project 2: Parsing and Compiling Expressions
  - Due: Tuesday, Feb 12th at 11:59:59pm

Searching for derivations.

LL & LR PARSING

Today: Parsing II



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:

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

- How many terminals? How many nonterminals? Productions?

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
$\underline{\mathbf{S}}$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto \underline{\mathbf{E}} + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{S}}) + S$	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{E}} + S) + S$	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{S}}) + S$	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$	3	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$	3	(1 + 2 + (3 + 4)) + 5
$\longmapsto \ldots$		

There is a problem

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

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

(1) $\boxed{S \longmapsto E} \longmapsto (S) \longmapsto (E) \longmapsto (1)$

vs.

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

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

LL(1) GRAMMARS

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?

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

- 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:

$$
\begin{aligned}
S &\longmapsto E + S \mid E \\
E &\longmapsto \text{number} \mid (\ S\ )
\end{aligned}
\quad\Longrightarrow\quad
\begin{aligned}
S &\longmapsto ES' \\
S' &\longmapsto \varepsilon \\
S' &\longmapsto + S \\
E &\longmapsto \text{number} \mid (\ S\ )
\end{aligned}
$$

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

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

LL(1) Parse of the input string

- Look at only one input symbol at a time.

$$
\begin{aligned}
S &\longmapsto ES' \\
S' &\longmapsto \varepsilon \\
S' &\longmapsto +\, S \\
E &\longmapsto \text{number} \mid (\, S\,)
\end{aligned}
$$

Partly-derived String	Look-ahead	Parsed/Unparsed Input
$\underline{\mathbf{S}}$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto \underline{\mathbf{E}}\ S'$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{S}})\ S'$	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{E}}\ S')\ S'$	1	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1\ \underline{\mathbf{S'}})\ S'$	+	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{S}})\ S'$	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{E}}\ S')\ S'$	2	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2\ \underline{\mathbf{S'}})\ S'$	+	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{S}})\ S'$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{E}}\ S')\ S'$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{S}})S')\ S'$	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

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

	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)		

- 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: if there are two different productions for a given entry, the grammar is not LL(1)

Example

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

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

$T \longmapsto S\$$
$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto$ number | (S)

	number	+	(	)	\$ (EOF)
T	$\longmapsto S\$$		$\longmapsto S\$$		
S	$\longmapsto E\ S'$		$\longmapsto E\ S'$		
S′		$\longmapsto + S$		$\longmapsto \varepsilon$	$\longmapsto \varepsilon$
E	$\longmapsto$ num.		$\longmapsto$ (S)		

Converting the table to code

- Define n mutually recursive functions
  - one for each nonterminal A: parse_A
  - The type of parse_A is `unit -> ast` 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: PARSER.ML

LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser

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

Top-down vs. Bottom up

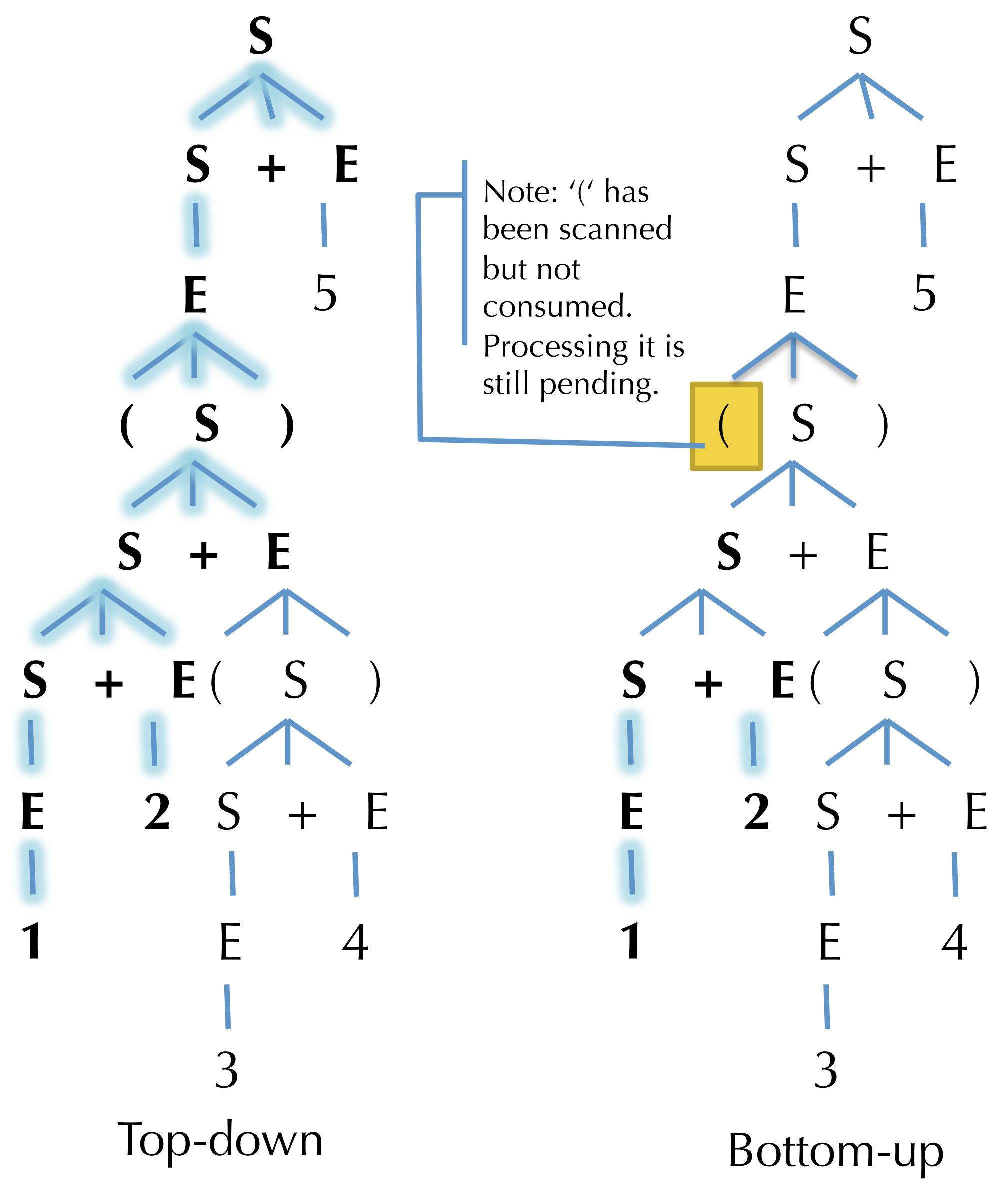
- Consider the left-recursive grammar:

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

- (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 ⟻		(1 + 2 + (3 + 4)) + 5
(**E** + 2 + (3 + 4)) + 5 ⟻	(	+ 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		

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

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 $X \longmapsto \gamma$ 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

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.

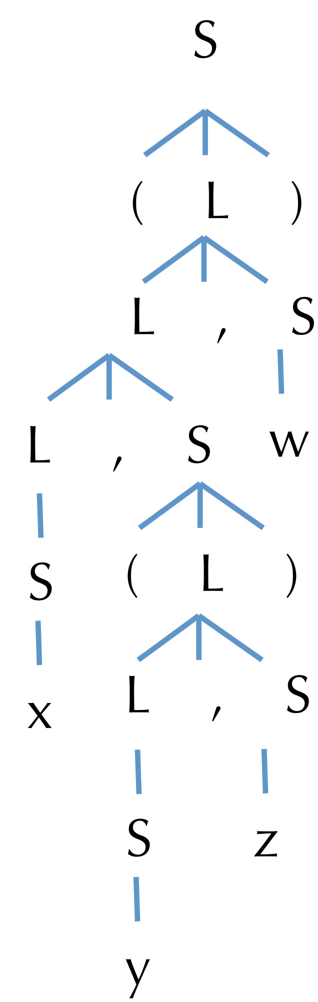
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

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

- 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 $\longmapsto$ id
(S	, (y, z), w)	reduce L $\longmapsto$ S
(L	, (y, z), w)	shift ,
(L,	(y, z), w)	shift (
(L, (	y, z), w)	shift y
(L, (y	, z), w)	reduce S $\longmapsto$ id
(L, (S	, z), w)	reduce L $\longmapsto$ S
(L, (L	, z), w)	shift ,
(L, (L,	z), w)	shift z
(L, (L, z	), w)	reduce S $\longmapsto$ id
(L, (L, S	), w)	reduce L $\longmapsto$ L, S
(L, (L	), w)	shift)
(L, (L)	, w)	reduce S $\longmapsto$ (L)
(L, S	, w)	reduce L $\longmapsto$ L, S
(L	, w)	shift ,
(L,	w)	shift w
(L, w	)	reduce S $\longmapsto$ id

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 a is different for different possible reductions since in productions $X \longmapsto g$ and $Y \longmapsto b$, g and b 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
- Start state of the DFA = empty stack, so it contains the item:
  $S' \longmapsto .S\$$

$$
\begin{aligned}
S' &\longmapsto S\$ \\
S &\longmapsto (\ L\ ) \ \mid\ id \\
L &\longmapsto S \ \mid\ L\ ,\ S
\end{aligned}
$$

- 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: $\text{CLOSURE}(\{S' \longmapsto .S\$\}) = \{S' \longmapsto .S\$,\ S \longmapsto .(L),\ S \longmapsto .id\}$

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

Example: Constructing the DFA

$S' \longmapsto .S\$$

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

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

Example: Constructing the DFA

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

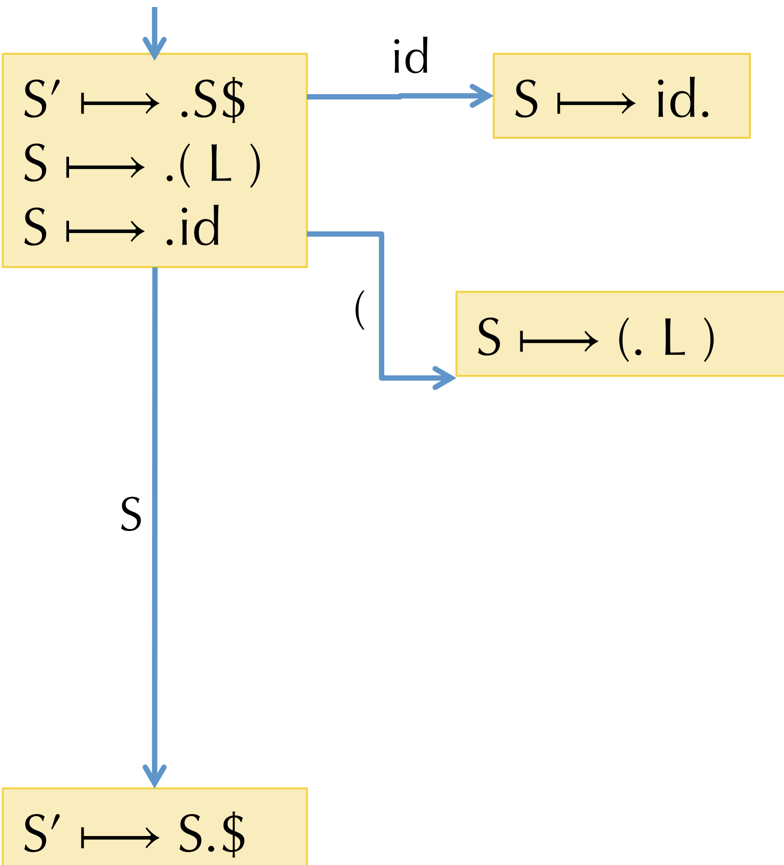
$S' \longmapsto S\$$
$S \longmapsto (\ L\ ) \ |\ id$
$L \longmapsto S \ |\ L\ ,\ S$

- 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

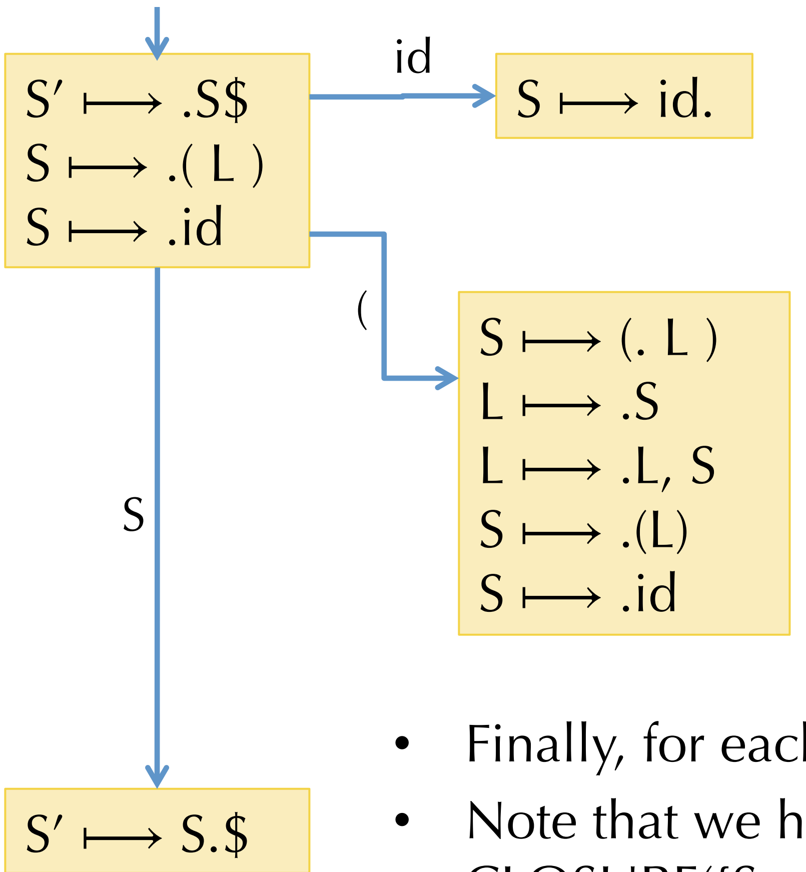
$$
\begin{aligned}
S' &\longmapsto S\$ \\
S &\longmapsto (\ L\ )\ \mid\ id \\
L &\longmapsto S\ \mid\ L\ ,\ S
\end{aligned}
$$



- 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)

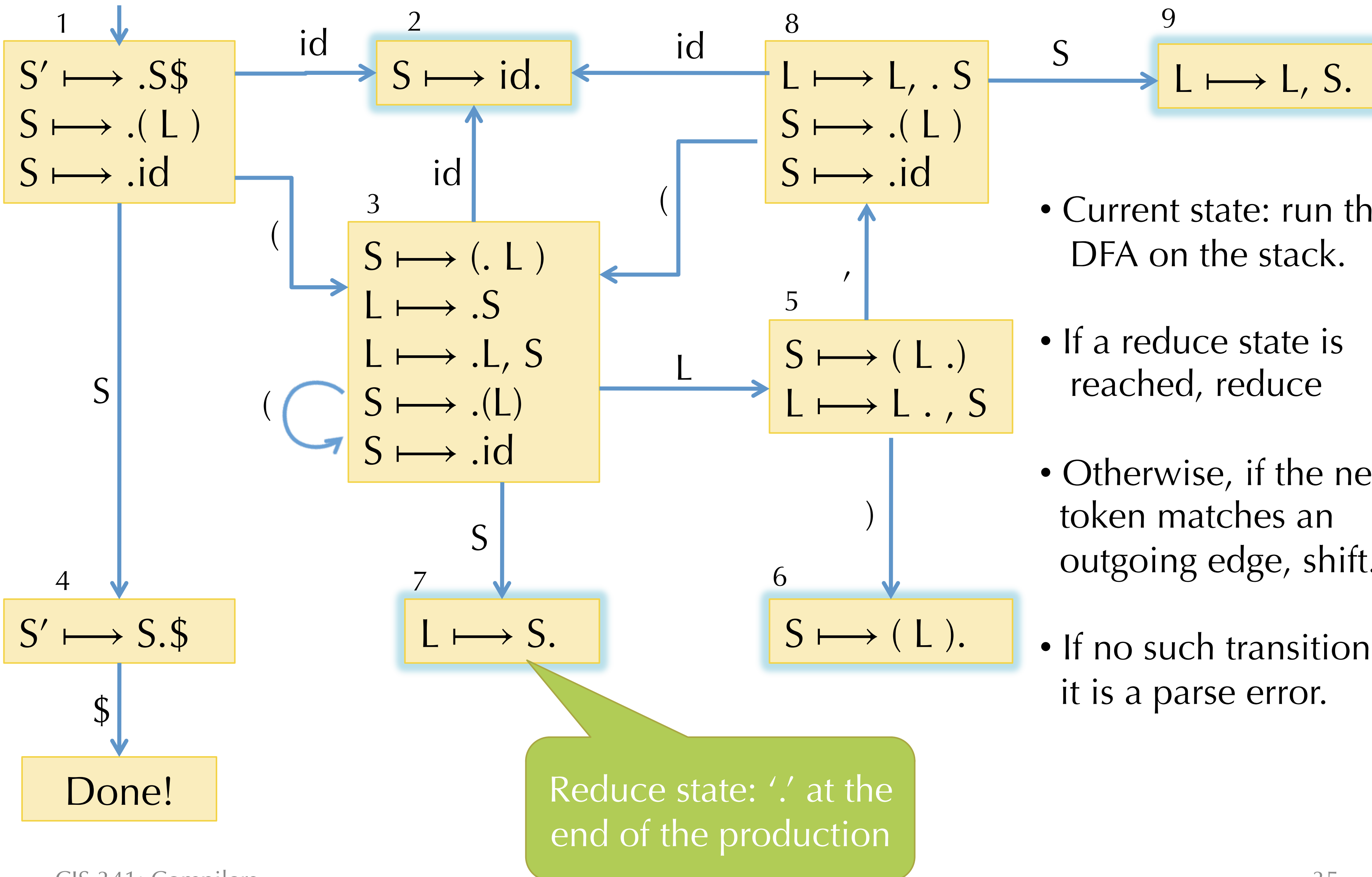
Example: Constructing the DFA

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



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