CS4212: Compiler Design

Week 7: Parsing

Ilya Sergey

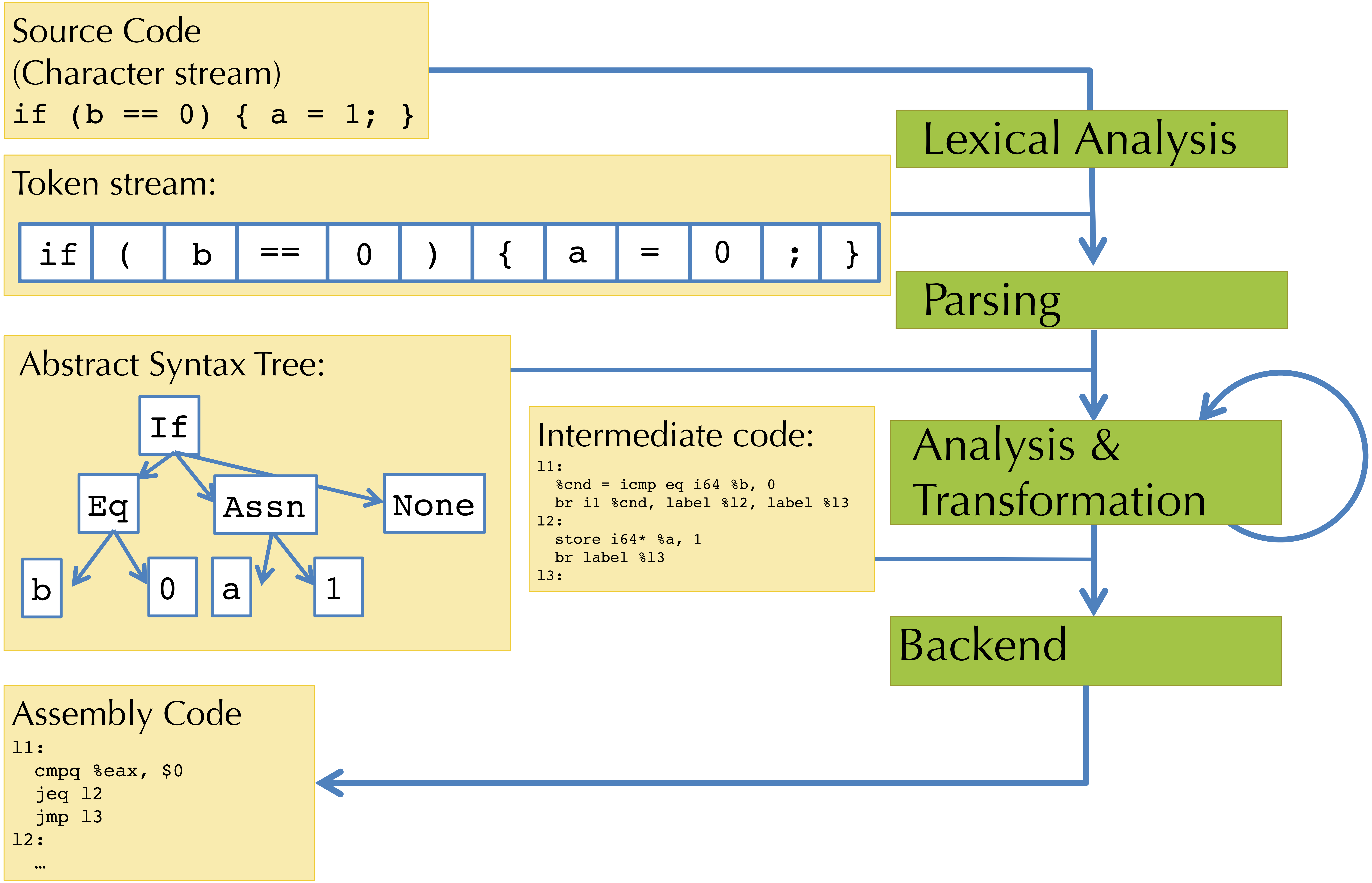
ilya@nus.edu.sg

ilyasergey.net/CS4212/

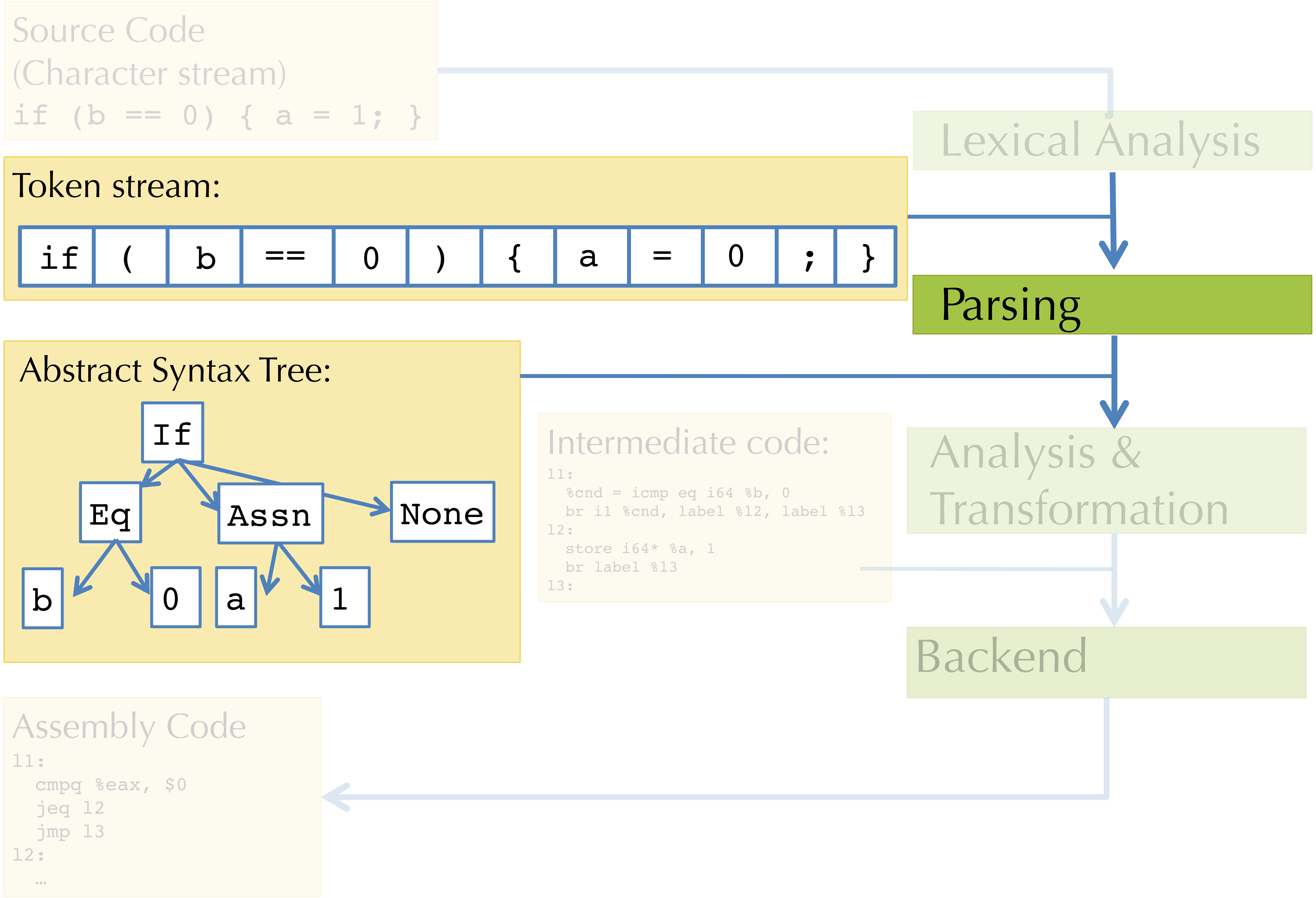
Where we are

- Before in the Course:
  - basics of x86
  - LLVM
- Last week:
  - Lexical Analysis
- This week:
  - Algorithms for Parsing
  - Parser Generation
- Next week:
  - Types and Type Systems

Compilation in a Nutshell



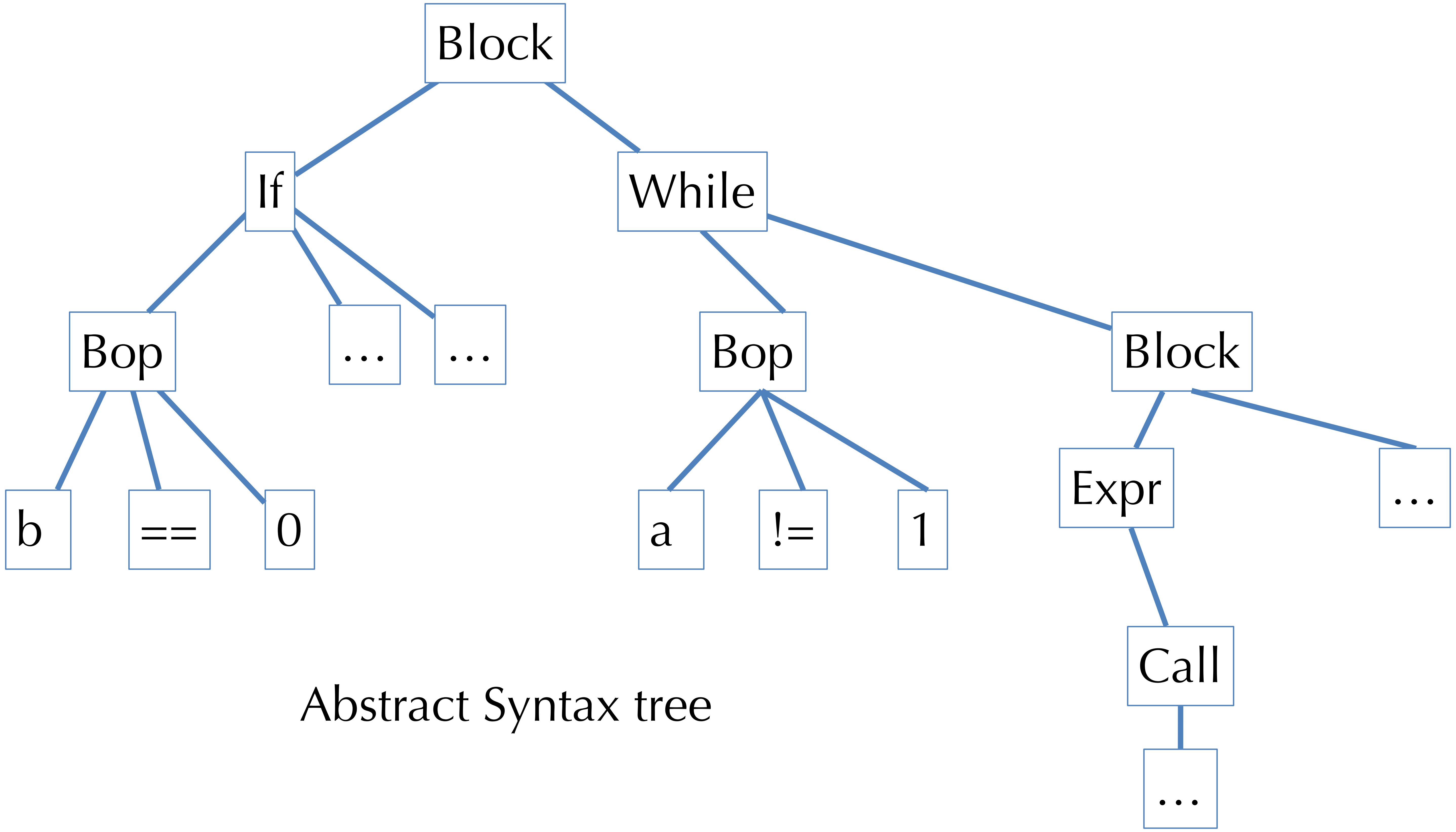
This week: Parsing



Parsing: Finding Syntactic Structure

```
{
  if (b == 0) a = b;
  while (a != 1) {
    print_int(a);
    a = a – 1;
  }
}
```

Source input



Abstract Syntax tree

Context-Free Grammars

Context-Free Grammars

- Here is a specification of the language of balanced parens:

$$S \longmapsto (S)S$$
$$S \longmapsto \varepsilon$$

Note: Once again we have to take care to distinguish meta-language elements (e.g. "S" and "$\longmapsto$") from object-language elements (e.g. "(").*

- The definition is *recursive* – S mentions itself.
- Idea: "derive" a string in the language by starting with S and *rewriting* according to the rules:
  - Example: $S \longmapsto (S)S \longmapsto ((S)S)S \longmapsto ((\varepsilon)S)S \longmapsto ((\varepsilon)S)\varepsilon \longmapsto ((\varepsilon)\varepsilon)\varepsilon = (())$
- You can replace the "nonterminal" S by one of its definitions anywhere
- A context-free grammar ***accepts*** a string iff there is a derivation from the start symbol

* And, since we're writing this description in English, we are careful distinguish the meta-meta-language (e.g. words) from the meta-language and object-language (e.g. symbols) by using quotes.

CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical 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?

LL & LR Parsing

Searching for derivations

Consider finding left-most derivations

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

- Look at only one input symbol at a time.

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	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 + **E**) + S	(	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**S**)) + S	3	(1 + 2 + (3 + 4)) + 5
⟼ (1 + 2 + (**E** + S)) + S	3	(1 + 2 + (3 + 4)) + 5
⟼ …		

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

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 *only one* 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)"

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

- Is it LL(k) for some k?
- 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.

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

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

Predictive Parsing

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

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼ S$		
S	⟼ E S’		⟼ E S’		
S’		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (S)		

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

- 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 → ε 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

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

- 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. More on these later in the course.

	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)		

Converting the table to code

- Define n mutually recursive functions
  - one for each nonterminal A: parse_A
  - Assuming the stream of tokens is globally available, 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.

Demo: LL(1) Parsing

- https://github.com/cs4212/week-06-parsing
- ll1_parser.ml
- Hand-generated LL(1) code for the table below.

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S’		⟼E S’		
S’		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (S)		

LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar ⇒ LL(1) grammar ⇒ prediction table ⇒ recursive-descent parser
- **Great** for simple hand-written implementation with *fine-tuned error control* (e.g., for editors)

- 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!)
  - There are CF grammars that cannot be transformed to LL(k)

- Is there a better way?

LR Grammars

Bottom-up Parsing (LR Parsers)

- LR(k) parser:
  - **L**eft-to-right scanning
  - **R**ightmost 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, menhir, etc.)
  - Better error detection/recovery

Top-down vs. Bottom up

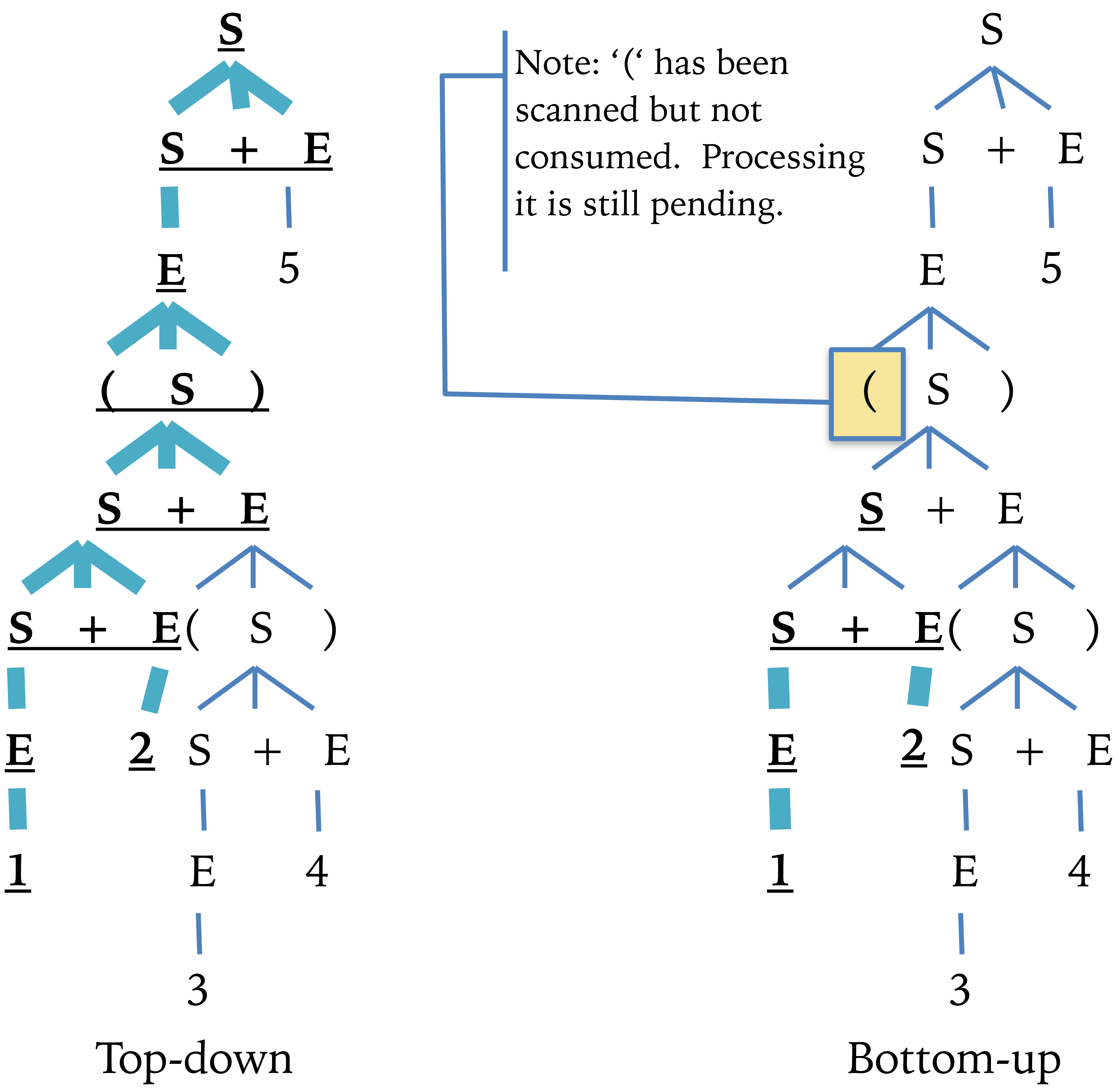
- Consider the left-recursive grammar:

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

- (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 ⟵	(	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

- 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

LR(0) Grammars

Simple LR parsing with no look-ahead.

LR Parser States

- Goal: know what set of reductions are legal at any given point.

- Idea: Summarise 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
  - **<u>L</u>**eft-to-right scanning, **<u>R</u>**ight-most derivation, **<u>zero</u>** 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.

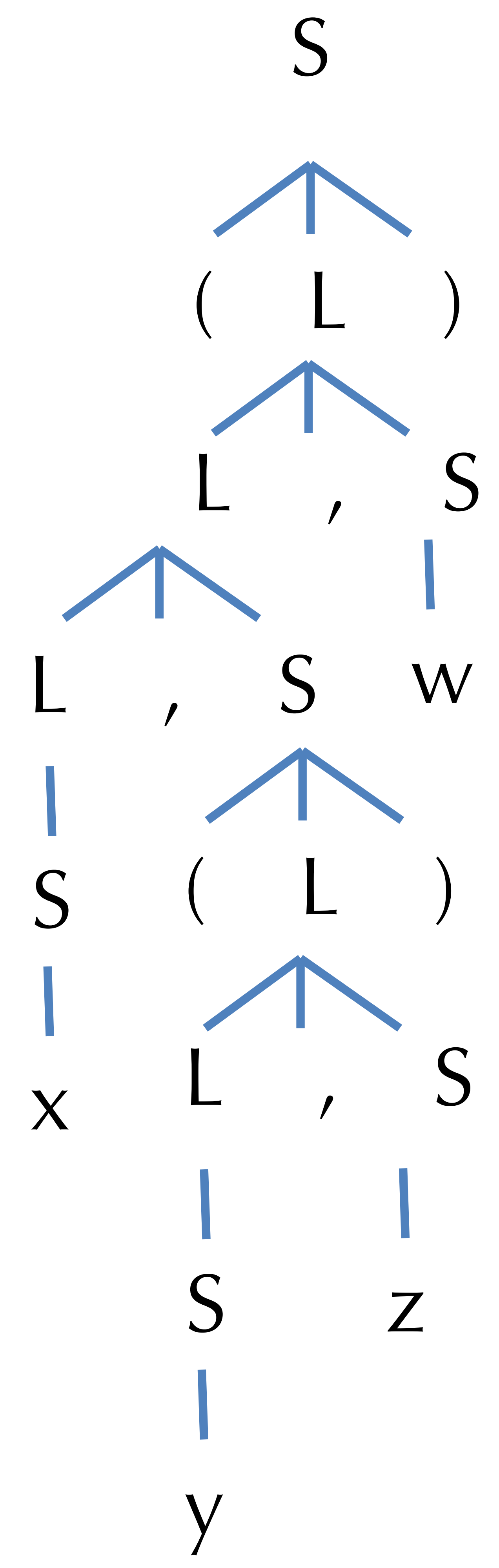
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

$$S \mapsto (\, L\,) \;|\; id$$
$$L \mapsto S \;|\; L\, ,\, S$$

- 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

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

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 (*reduce/reduce* conflict)

- 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

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

- 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

- Idea of the Closure: productions that can be applicable with the already observed stack
- 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($\{S' \longmapsto .S\$\}$) = $\{S' \longmapsto .S\$,\ S \longmapsto .(L),\ S \longmapsto .id\}$

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

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

Example: Constructing the DFA

$S' \longmapsto .S\$$

$S' \longmapsto S\$$

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

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

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

Example: Constructing the DFA

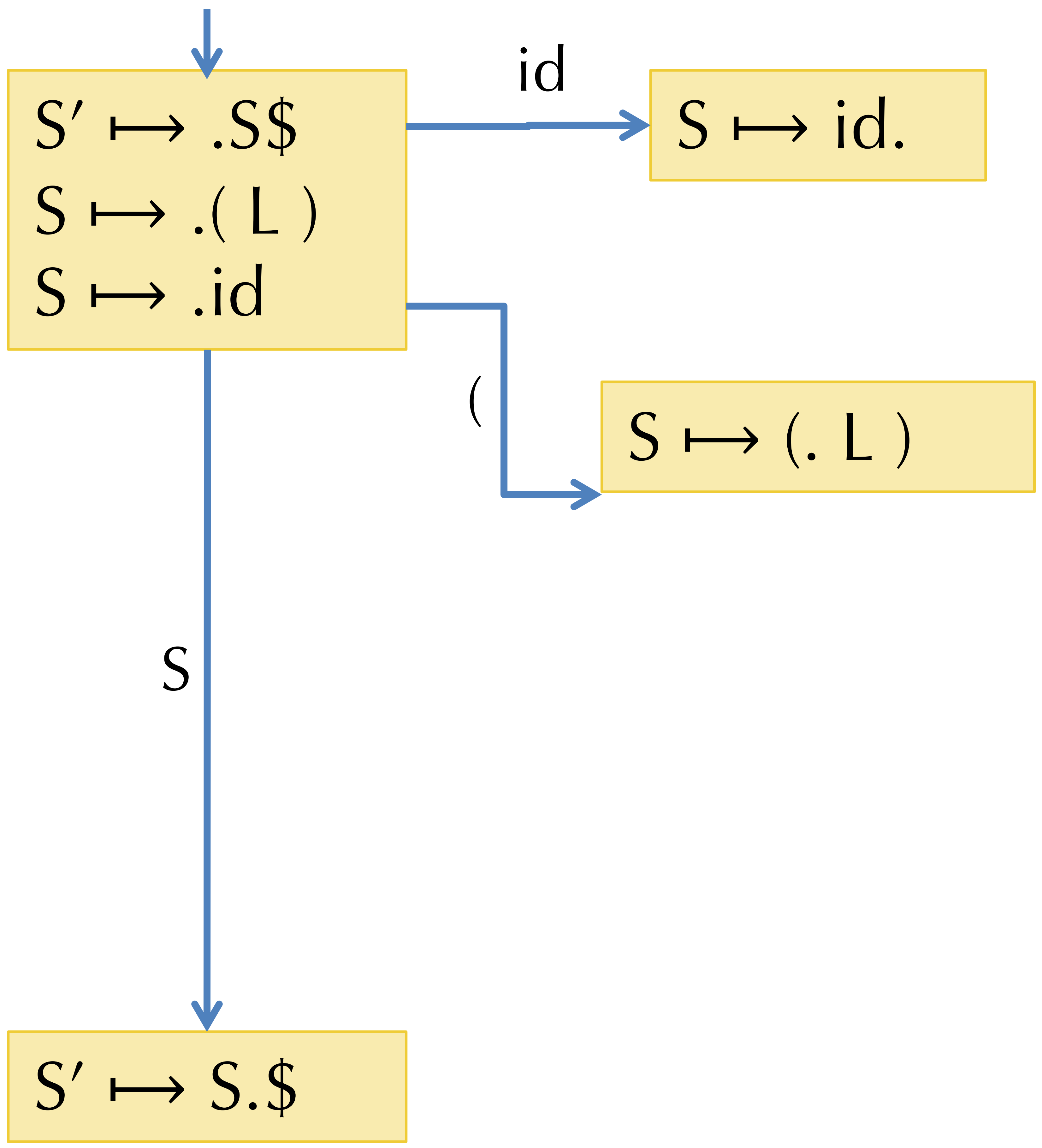
$S' \longmapsto .S\$$
$S \longmapsto .(\ L\ )$
$S \longmapsto .id$

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

- 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

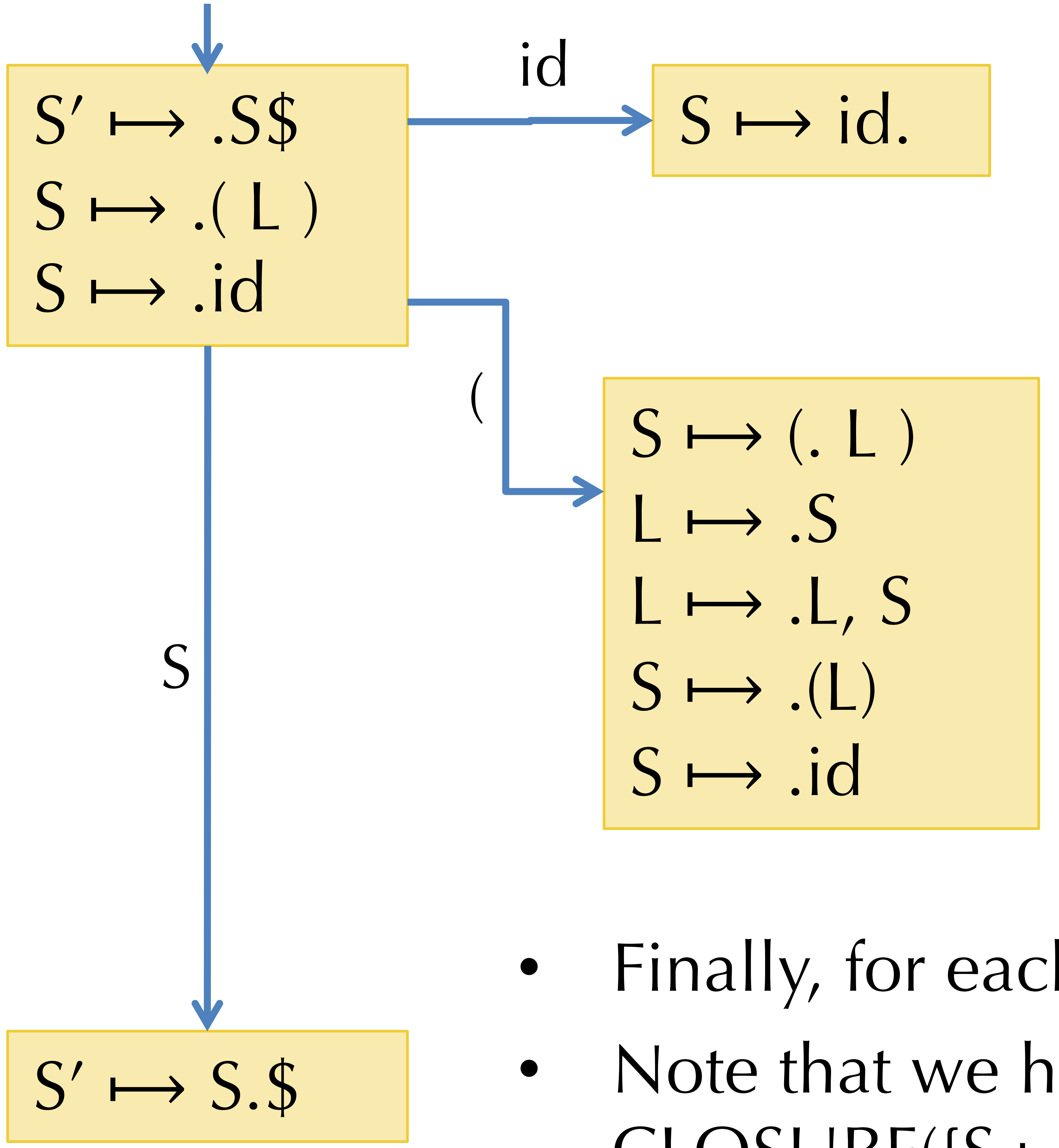
Example: Constructing the DFA



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)

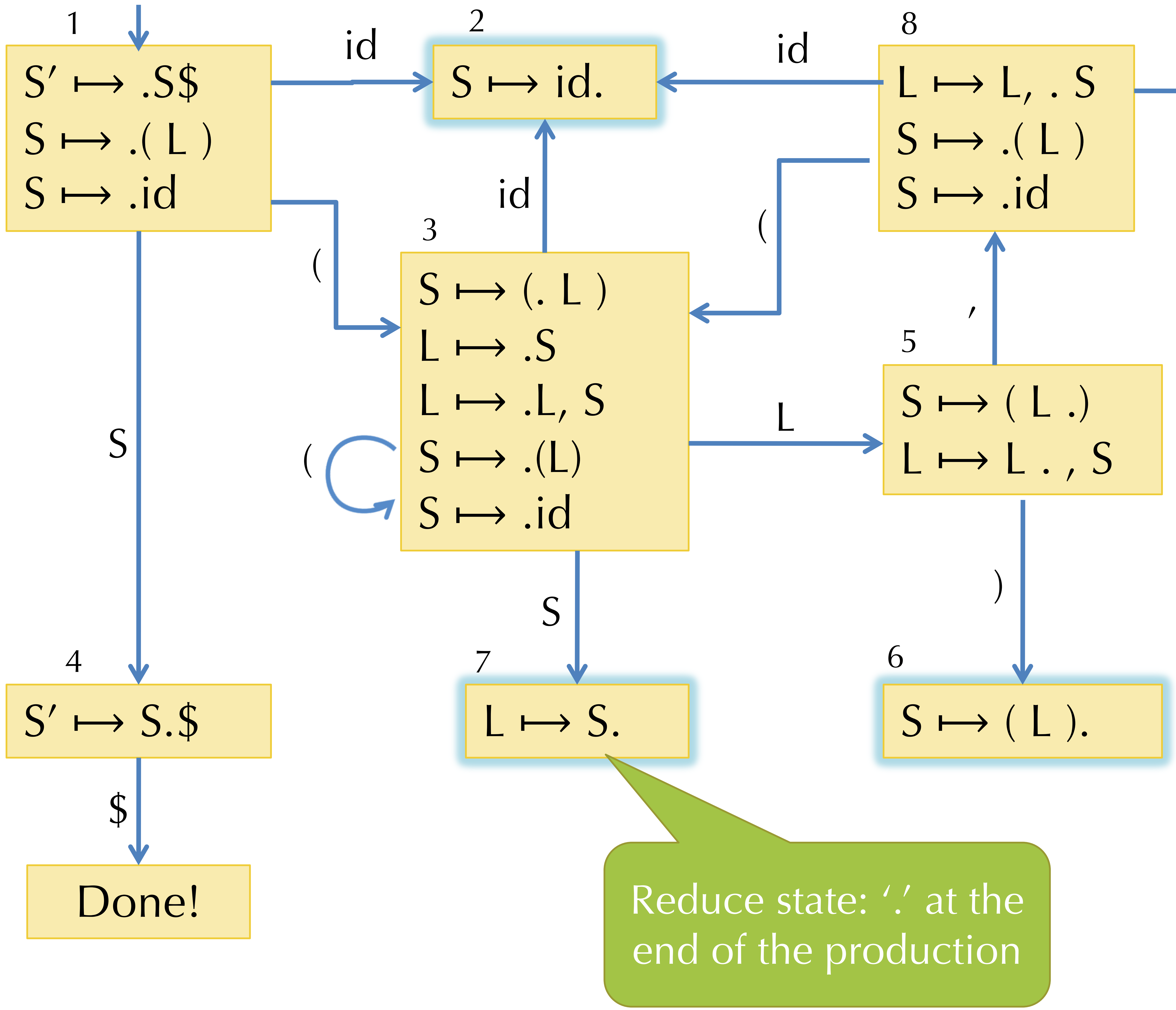
Example: Constructing the DFA



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

Example: Constructing the DFA



- 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.

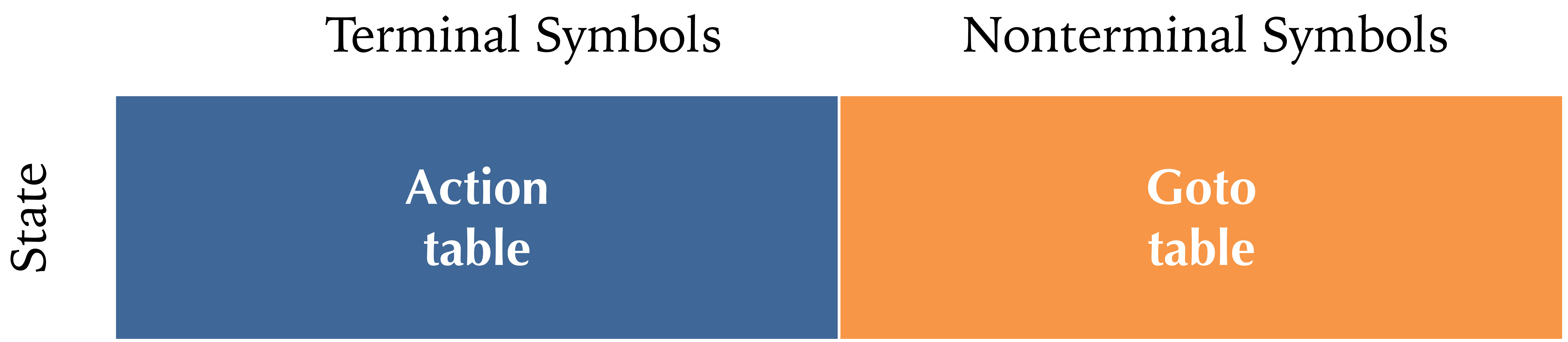
- *Optimisation*: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. $_1(_3(_3L_5)_6$
  - On a reduction $X \mapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack $_1(_3(_3L_5)_6$ reduce $S \mapsto (L)$ to reach stack $_1(_3$
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

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

	(	)	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		

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: LR(1)

Examples

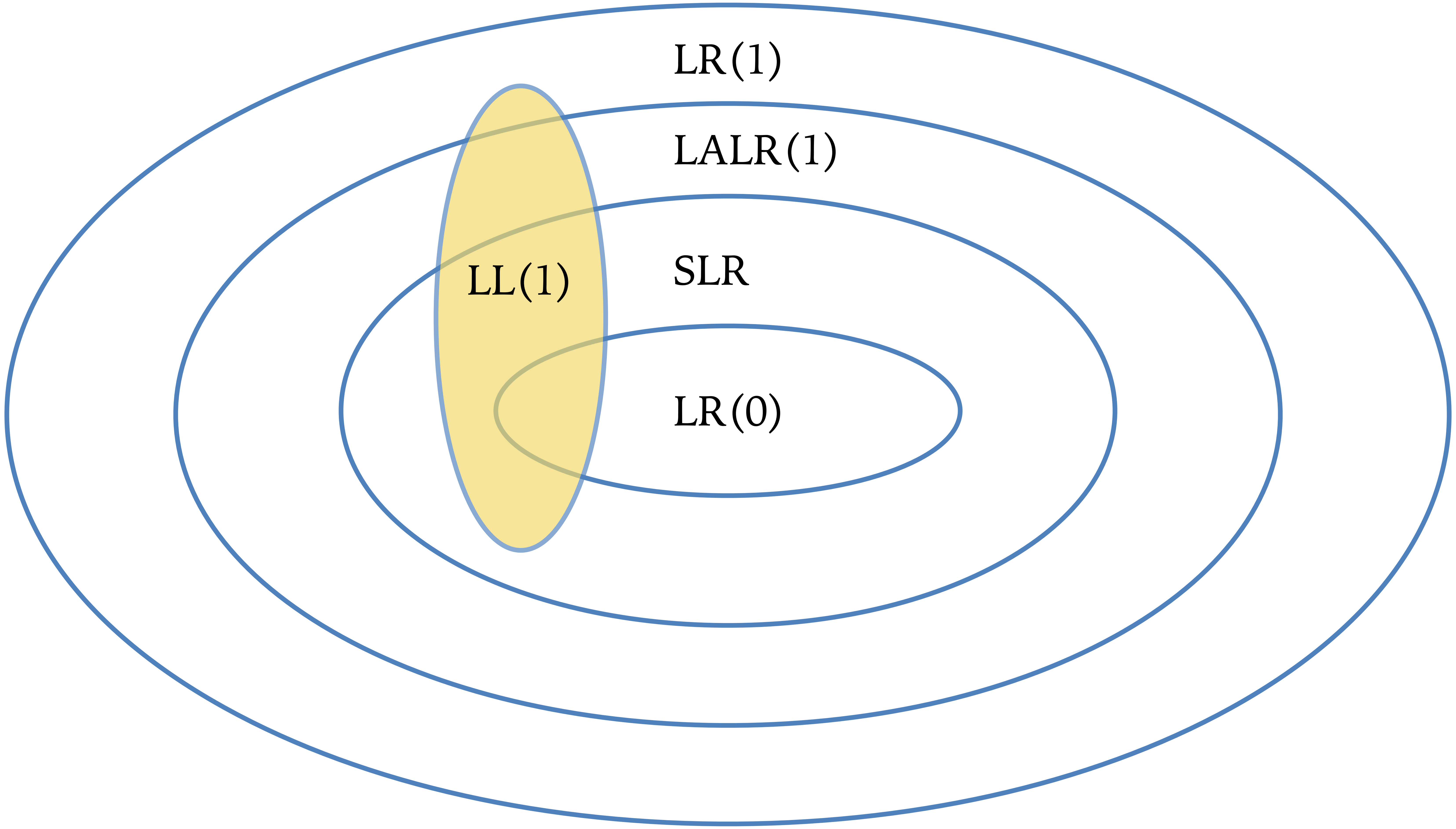
- Consider the left associative and right associative "sum" grammars:

left	right
S ⟼ S + E | E E ⟼ number | (S)	S ⟼ E + S | E E ⟼ number | (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.

Classification of Grammars



Parsing in OCaml via Menhir

Practical Issues

- https://github.com/cs4212/week-07-more-parsing

- Dealing with source file location information
  - In the lexer and parser
  - In the abstract syntax

  - See range.ml, ast.ml
  - Check the parse tree (printing via driver.ml)

- Lexing comments / strings

Menhir output

- You can get verbose parser debugging information by doing:
  - `menhir --explain …`
  - or, if using ocamlbuild:
    `ocamlbuild –use-menhir -yaccflag -–explain …`
- The result is a <parsername>.conflicts file that contains a description of the error
  - The parser items of each state use the '.' just as described above
- The flag --dump generates a full description of the automaton
- Example: see start_parser.mly

Shift/Reduce conflicts

- Conflict 1:
  - Operator precedence (State 13)

- Conflict 2:
  - Parsing if-then-else statements

Shift/Reduce conflicts

- Conflict 1:
  - Operator precedence (State 13)
  - Resolving by changing the grammar (see good_parser.ml)

- Conflict 2:
  - Parsing if-then-else statements

5.3 Inlining

It is well-known that the following grammar of arithmetic expressions does not work as expected: that is, in spite of the priority declarations, it has shift/reduce conflicts.

```
%token < int > INT
%token PLUS TIMES
%left PLUS
%left TIMES

%%

expression:
      | i = INT { i }
      | e = expression; o = op; f = expression { o e f }
op:
      | PLUS { ( + ) }
      | TIMES { ( * ) }
```

The trouble is, the precedence level of the production *expression* $\rightarrow$ *expression op expression* is undefined, and there is no sensible way of defining it via a **%prec** declaration, since the desired level really depends upon the symbol that was recognized by *op*: was it *PLUS* or *TIMES*?

From Menhir Manual

http://gallium.inria.fr/~fpottier/menhir/manual.pdf

The standard workaround is to abandon the definition of *op* as a separate nonterminal symbol, and to inline its definition into the definition of *expression*, like this:

expression:
| $i = INT$ { i }
| $e = expression$; $PLUS$; $f = expression$ { $e + f$ }
| $e = expression$; $TIMES$; $f = expression$ { $e * f$ }

This avoids the shift/reduce conflict, but gives up some of the original specification's structure, which, in realistic situations, can be damageable. Fortunately, Menhir offers a way of avoiding the conflict without manually transforming the grammar, by declaring that the nonterminal symbol *op* should be inlined:

expression:
| $i = INT$ { i }
| $e = expression$; $o = op$; $f = expression$ { $o\ e\ f$ }
%inline *op*:
| $PLUS$ { (+) }
| $TIMES$ { (*) }

The **%inline** keyword causes all references to *op* to be replaced with its definition. In this example, the definition of *op* involves two productions, one that develops to *PLUS* and one that expands to *TIMES*, so every production that refers to *op* is effectively turned into two productions, one that refers to *PLUS* and one that refers to *TIMES*. After inlining, *op* disappears and *expression* has three productions: that is, the result of inlining is exactly the manual workaround shown above.

Precedence and Associativity Declarations

- Parser generators, like menhir often support precedence and associativity declarations.
  - Hints to the parser about how to resolve conflicts.
  - See: good-parser.mly

- Pros:
  - Avoids having to manually resolve those ambiguities by manually introducing extra nonterminals (see parser.mly)
  - Easier to maintain the grammar

- Cons:
  - Can't as easily re-use the same terminal (if associativity differs)
  - Introduces another level of debugging

- Limits:
  - Not always easy to disambiguate the grammar based on just precedence and associativity.

Conflict 2: Ambiguity in Real Languages

- Consider this grammar:

  S ⟼ `if (E) S`
  S ⟼ `if (E)` S `else` S
  S ⟼ X = E
  E ⟼ …

- Is this grammar OK?

- Consider how to parse:

  `if (`E_1`) if (`E_2`)` S_1 `else` S_2

- This is known as the "dangling else" problem.

- What should the "right" answer be?

- How do we change the grammar?

How to Disambiguate if-then-else

- Want to rule out:

$$\texttt{if (}E_1\texttt{)} \left\{ \texttt{if (}E_2\texttt{) } S_1 \right\} \texttt{else } S_2$$

- Observation: An un-matched ‘`if`’ should not appear as the ‘`then`’ clause of a containing ‘`if`’.

$S \longmapsto M \mid U$ // M = “matched”, U = “unmatched”

$U \longmapsto$ `if (`E`)` S // Unmatched ‘if’

$U \longmapsto$ `if (`E`)` M `else` U // Nested if is matched

$M \longmapsto$ `if (`E`)` M `else` M // Matched ‘if’

$M \longmapsto X = E$ // Other statements

- See: else-resolved-parser.mly

Alternative: Use { }

- Ambiguity arises because the 'then' branch is not well bracketed:

  `if (`E_1`) { if (`E_2`) { `S_1` } } else `S_2 // unambiguous

  `if (`E_1`) { if (`E_2`) { `S_1` } else `S_2` }` // unambiguous

- So: could just require brackets
  – But requiring them for the else clause too leads to ugly code for chained if-statements:

```
if (c1) {
  …
} else {
  if (c2) {

  } else {
    if (c3) {

    } else {

    }
  }
}
```

How about a compromise? Allow unbracketed else block only if the body is 'if':

```
if (c1) {

} else if (c2) {

} else if (c3) {

} else {

}
```

Benefits:

- Less ambiguous
- Easy to parse
- Enforces good style

HW4: Oat v.1

Oat

- Simple C-like Imperative Language
  - supports 64-bit integers, arrays, strings
  - top-level, mutually recursive procedures
  - scoped local, imperative variables

- See examples in *hw4programs* folder

- How to design/specify such a language?

Oat v.1 Language Specification

1 Grammar

The following grammar defines the Oat syntax. All binary operations are *left associative* with precedence levels indicated numerically. Higher precedence operators bind tighter than lower precedence ones.

prog	::=		prog
	|	$decl_1 \,..\, decl_i$	
decl	::=		global declarations
	|	*gdecl*	
	|	*fdecl*	
gdecl	::=		global variable declarations
	|	`global` *id* = *gexp*;	