Lecture 11

CIS 341: COMPILERS

Announcements

- HW3: LLVM lite
  - Available on the course web pages.
  - Due: Weds., February 23rd at 11:59:59pm

it is officially too late to
START EARLY!!

- Midterm: March 3rd
  - In class
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - Coverage: interpreters / program transformers / x86 / calling conventions / IRs / LLVM / Lexing / Parsing
  - See examples of previous exams on the web pages

Parsing

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

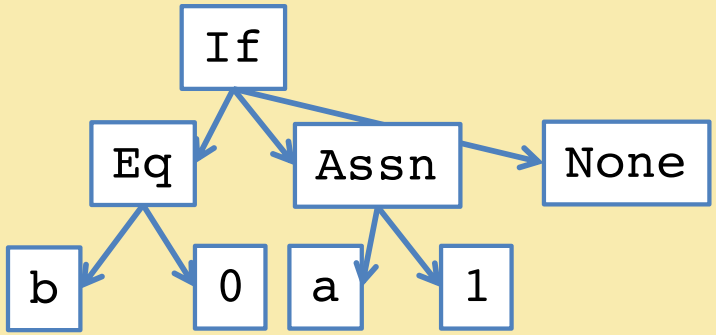
Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	0	;	}

Parsing

Abstract Syntax Tree:



Intermediate code:

```
l1:
  %cnd = icmp eq i64 %b,
0
  br i1 %cnd, label %l2,
label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Analysis & Transformation

Backend

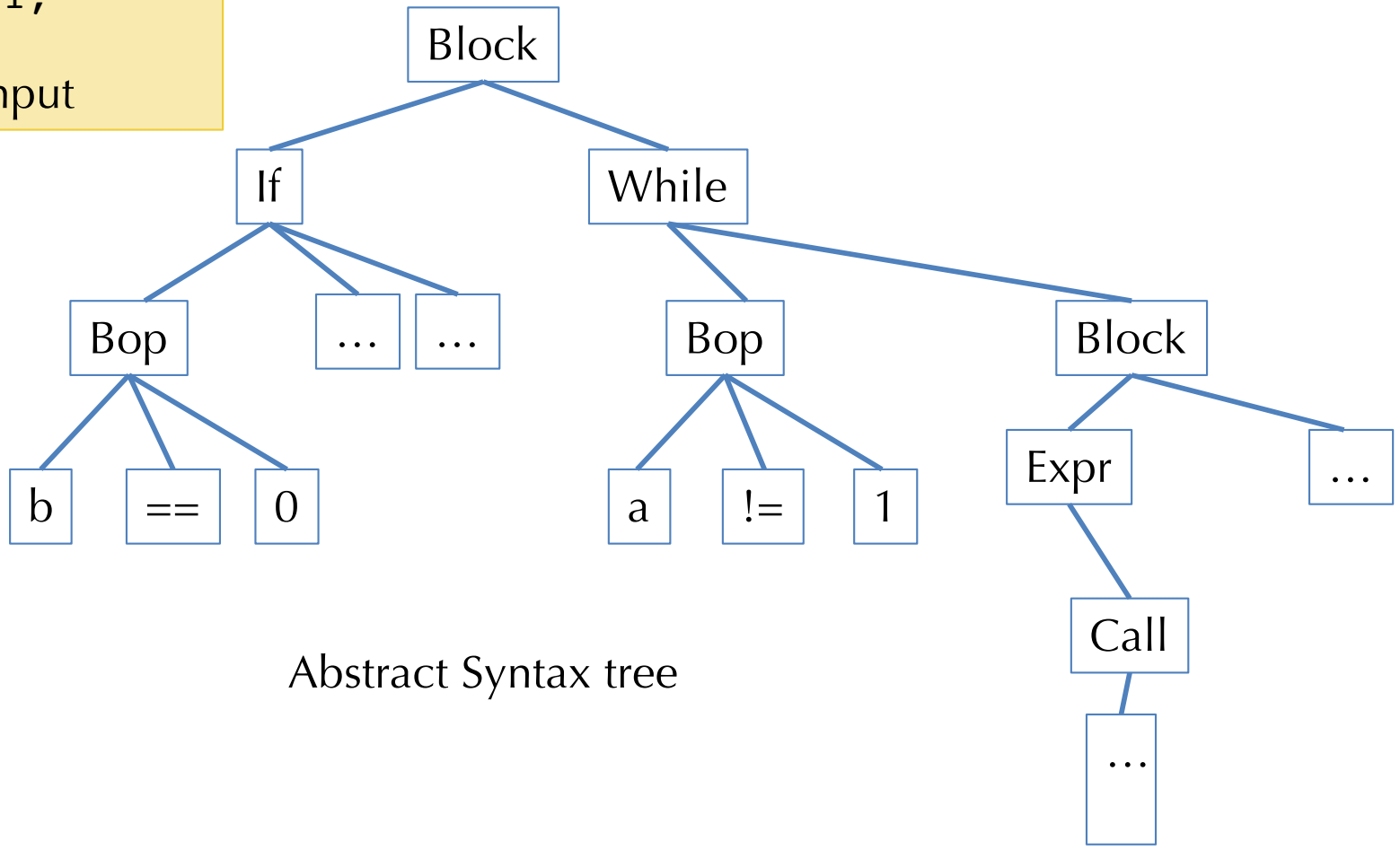
Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

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 "⟼") 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 ⟼ 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?

Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

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

e.g.: (1 + 2 + (3 + 4)) + 5

- Note the vertical bar '|' is shorthand for multiple productions:

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

4 productions
2 nonterminals: S, E
4 terminals: (,), +, number
Start symbol: S

Derivations in CFGs

- Example: derive (1 + 2 + (3 + 4)) + 5
- **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 ⟼ 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

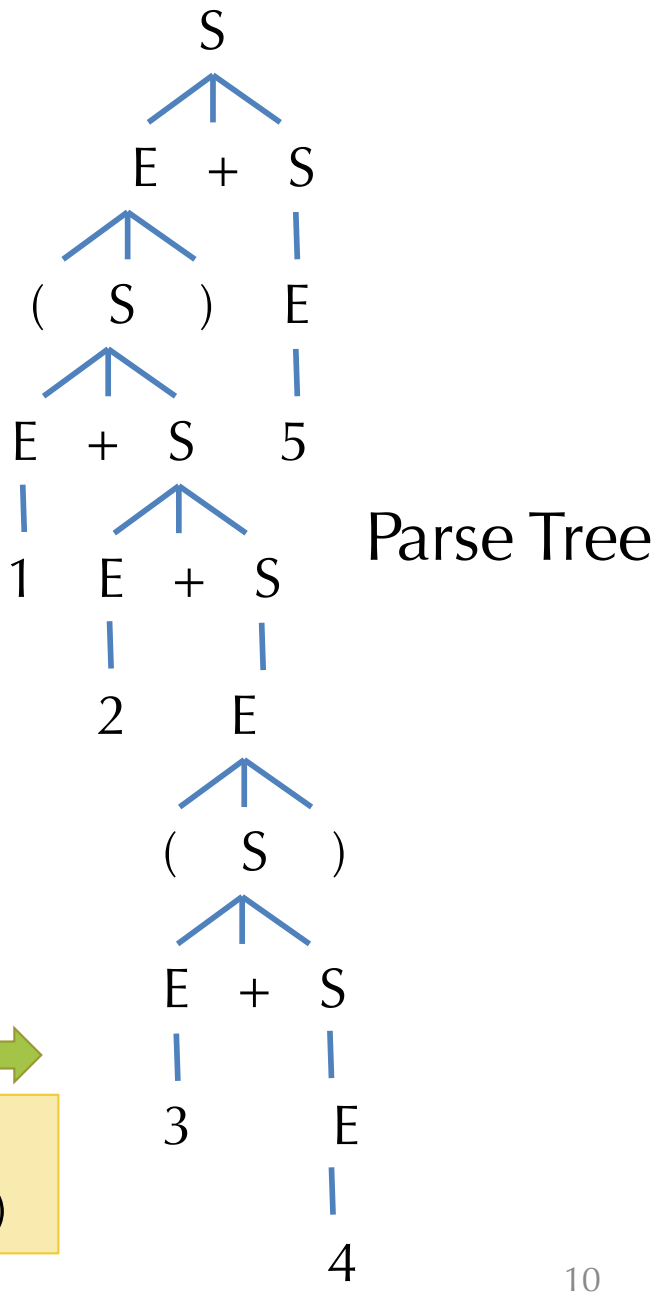
Note: Underline indicates symbol being expanded.

From Derivations to Parse Trees

- Tree representation of the derivation
- Leaves of the tree are terminals
  - In-order traversal yields the input sequence of tokens
- Internal nodes: nonterminals
- No information about the *order* of the derivation steps

- (1 + 2 + (3 + 4)) + 5

S ⟼ E + S | E
E ⟼ number | (S)

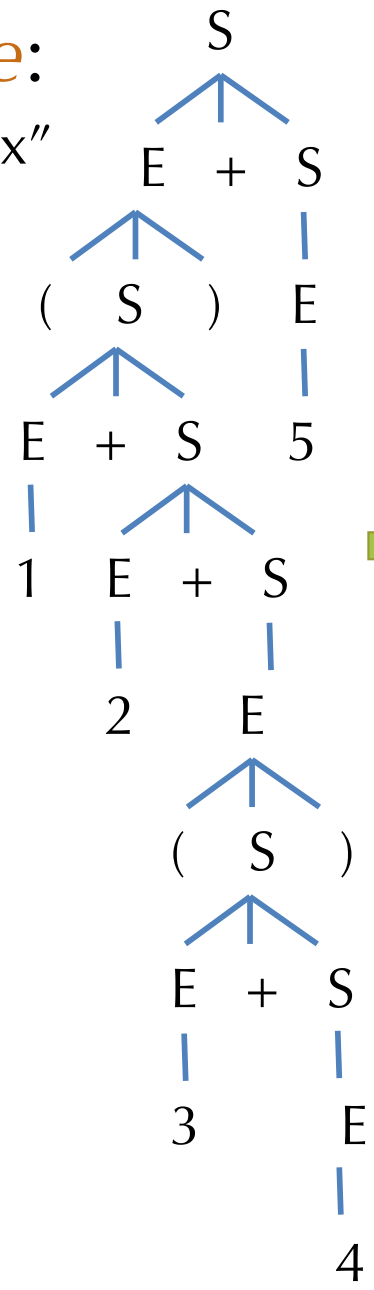


Parse Tree

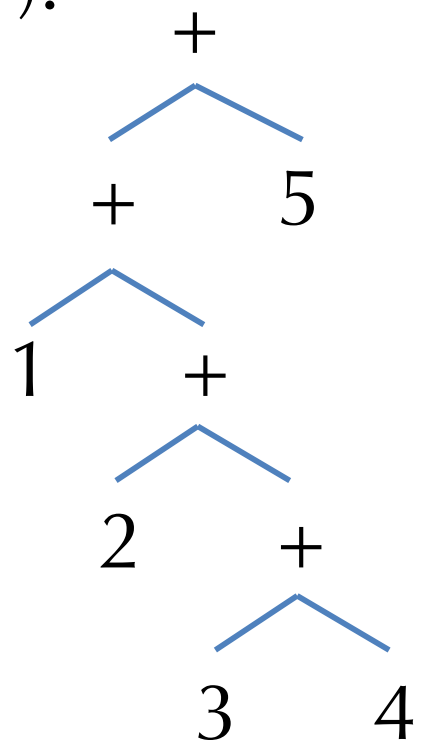
From Parse Trees to Abstract Syntax

- *Parse tree*:

"concrete syntax"



- *Abstract syntax tree* (AST):



- Hides, or *abstracts*, unneeded information.

Derivation Orders

- Productions of the grammar can be applied in any order.
- There are two standard orders:
  - *Leftmost derivation*: Find the left-most nonterminal and apply a production to it.
  - *Rightmost derivation*: Find the right-most nonterminal and apply a production there.

- Note that both strategies (and any other) yield the same parse tree!
  - Parse tree doesn't contain the information about what order the productions were applied.

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

Rightmost derivation:

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

S ⟼ E + S | E
E ⟼ number | (S)

Loops and Termination

- Some care is needed when defining CFGs
- Consider:

$$S \longmapsto E$$
$$E \longmapsto S$$

  - This grammar has nonterminal definitions that are "nonproductive". (i.e. they don't mention any terminal symbols)
  - There is no finite derivation starting from S, so the language is empty.

- Consider: $S \longmapsto (S)$

  - This grammar is productive, but again there is no finite derivation starting from S, so the language is empty

- Easily generalize these examples to a "cycle" of many nonterminals, which can be harder to find in a large grammar

- Upshot: be aware of "vacuously empty" CFG grammars.
  - Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols.

Associativity, ambiguity, and precedence.

GRAMMARS FOR PROGRAMMING LANGUAGES

Associativity

Consider the input: 1 + 2 + 3

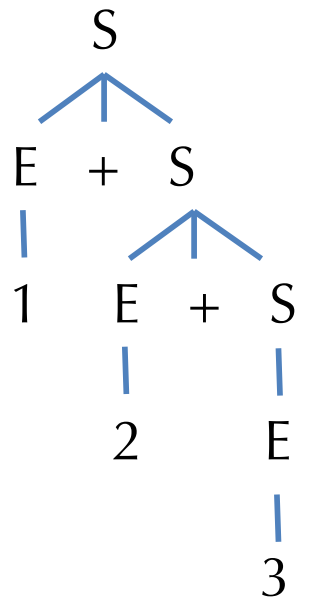
S ⟼ E + S | E
E ⟼ number | (S)

Leftmost derivation:

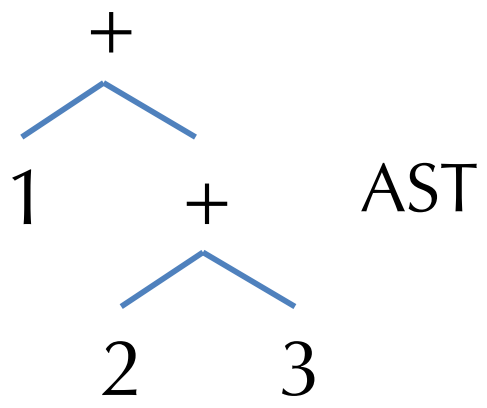
S ⟼ **E** + S
⟼ 1 + **S**
⟼ 1 + **E** + S
⟼ 1 + 2 + **S**
⟼ 1 + 2 + **E**
⟼ 1 + 2 + 3

Rightmost derivation:

S ⟼ E + **S**
⟼ E + E + **S**
⟼ E + E + **E**
⟼ E + **E** + 3
⟼ **E** + 2 + 3
⟼ 1 + 2 + 3



Parse Tree



AST

Associativity

- This grammar makes '+' *right associative*…
  - i.e., the abstract syntax tree is the same for both 1 + 2 + 3 and 1 + (2 + 3)
- Note that the grammar is *right recursive*…

S ⟼ E + S | E
E ⟼ number | (S)

S refers to itself on the right of +

- How would you make '+' left associative?
- What are the trees for "1 + 2 + 3"?

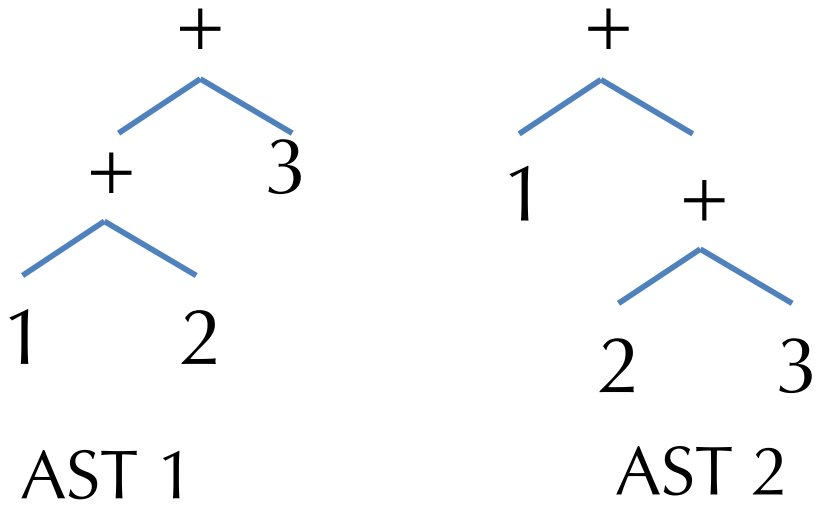
Ambiguity

- Consider this grammar:

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

- Claim: it accepts the *same* set of strings as the previous one.
- What's the difference?
- Consider these *two* leftmost derivations:
  - $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto 1 + \underline{\mathbf{S}} \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$
  - $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto \underline{\mathbf{S}} + S + S \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$

- One derivation gives left associativity, the other gives right associativity to '+'
  - Which is which?



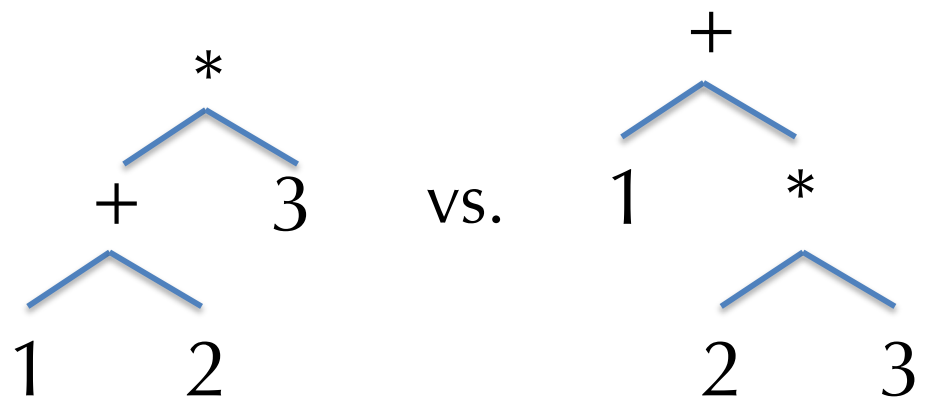
Why do we care about ambiguity?

- The '+' operation is associative, so it doesn't matter which tree we pick. Mathematically, $x + (y + z) = (x + y) + z$
  - But, some operations aren't associative. Examples?
  - Some operations are only left (or right) associative. Examples?

- Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*
- Consider:

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

- Input: 1 + 2 * 3
  - One parse = (1 + 2) * 3 = 9
  - The other = 1 + (2 * 3) = 7



Eliminating Ambiguity

- We can often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right) .
- Higher-precedence operators go *farther* from the start symbol.
- Example:

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

- To disambiguate:
  - Decide (following math) to make '*' higher precedence than '+'
  - Make '+' left associative
  - Make '*' right associative
- Note:
  - S_2 corresponds to 'atomic' expressions

$$S_0 \longmapsto S_0 + S_1 \mid S_1$$
$$S_1 \longmapsto S_2 * S_1 \mid S_2$$
$$S_2 \longmapsto \text{number} \mid (S_0)$$

Context Free Grammars: Summary

- Context-free grammars allow concise specifications of programming languages.
  - An unambiguous CFG specifies how to parse: convert a token stream to a (parse tree)
  - Ambiguity can (often) be removed by encoding precedence and associativity in the grammar.

- Even with an unambiguous CFG, there may be more than one derivation
  - Though all derivations correspond to the same abstract syntax tree.

- Still to come: finding a derivation
  - But first: menhir

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

DEMO: BOOLEAN LOGIC

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 ⟼ 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 ⟼ E + S | E
E ⟼ 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	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

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

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

(1) $S \mapsto E \mapsto (S) \mapsto (E) \mapsto (1)$

vs.

(1) + 2 $S \mapsto E + S \mapsto (S) + S \mapsto (E) + S \mapsto (1) + S \mapsto (1) + E$
$\mapsto (1) + 2$

- Given the look-ahead symbol: '(' it isn't clear whether to pick $S \mapsto E$ or $S \mapsto E + S$ 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 ⟼ E + S | E
E ⟼ 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:

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

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

- Also need to eliminate left-recursion somehow. 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.

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

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

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

	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 $.

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′)

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

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	↦ 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 `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	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (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, merlin, etc.)
  - Better error detection/recovery

Top-down vs. Bottom up

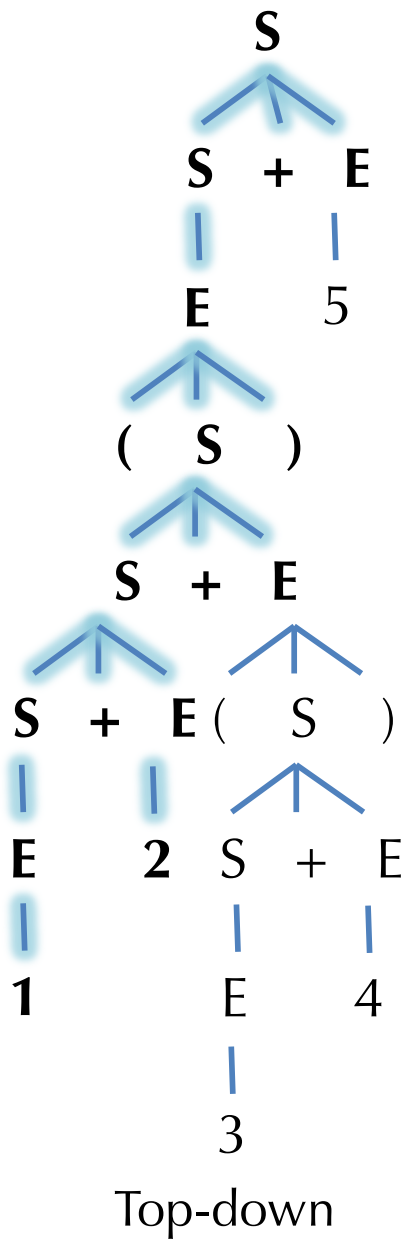
- Consider the left-recursive grammar:

  S ⟼ S + E | E
  E ⟼ number | (S)

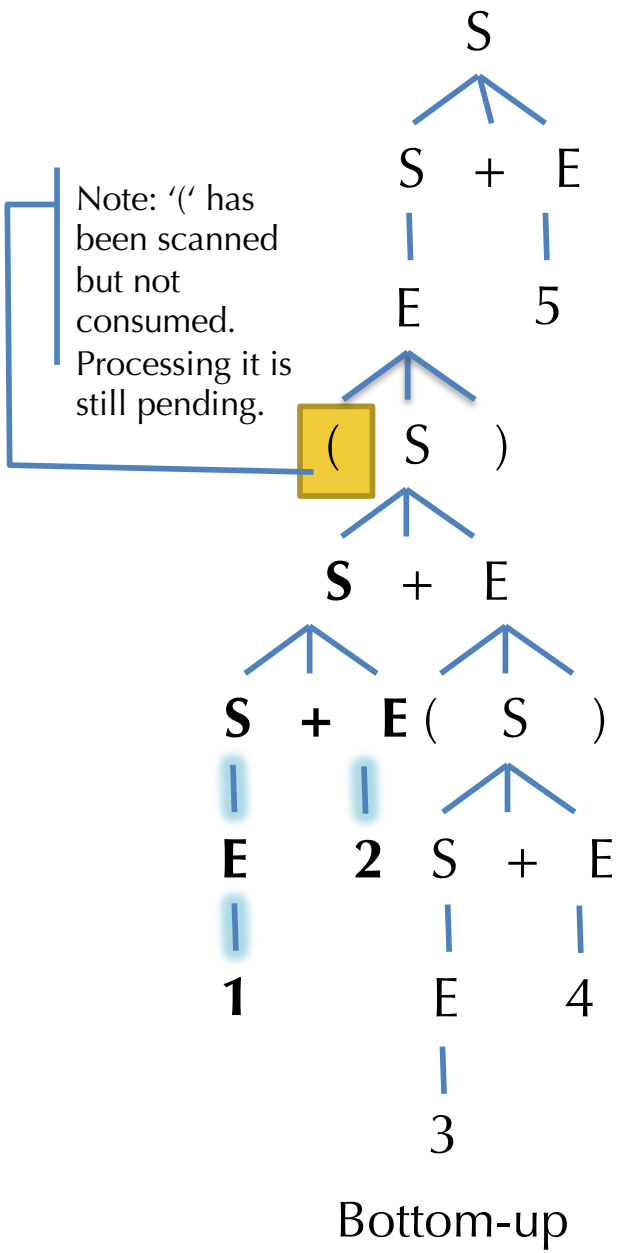
- (1 + 2 + (3 + 4)) + 5

- 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…



Top-down



Bottom-up

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 $X \longmapsto \gamma$ 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 ⟼ 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

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

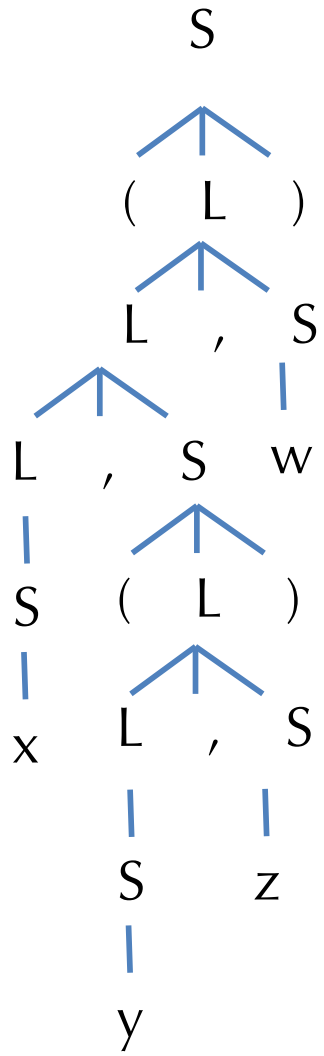
  (x,y)

  ((((x))))

  (x, (y, z), w)

  (x, (y, (z, w)))

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



Shift/Reduce Parsing

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

- 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 $X \mapsto \gamma$ is a production. (pop γ, push X): e.g.

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

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

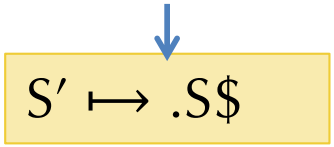
- First step: Add a new production
  $S' \mapsto S\$$ to the grammar
- Start state of the DFA = empty stack,
  so it contains the item:
  $S' \mapsto .S\$$
- Closure of a state:
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the '.'
  - The added items 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

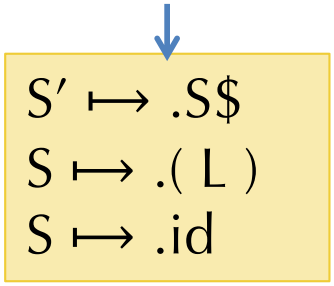
$S' \longmapsto S\$$
$S \longmapsto (L) \mid id$
$L \longmapsto S \mid L , S$



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

Example: Constructing the DFA

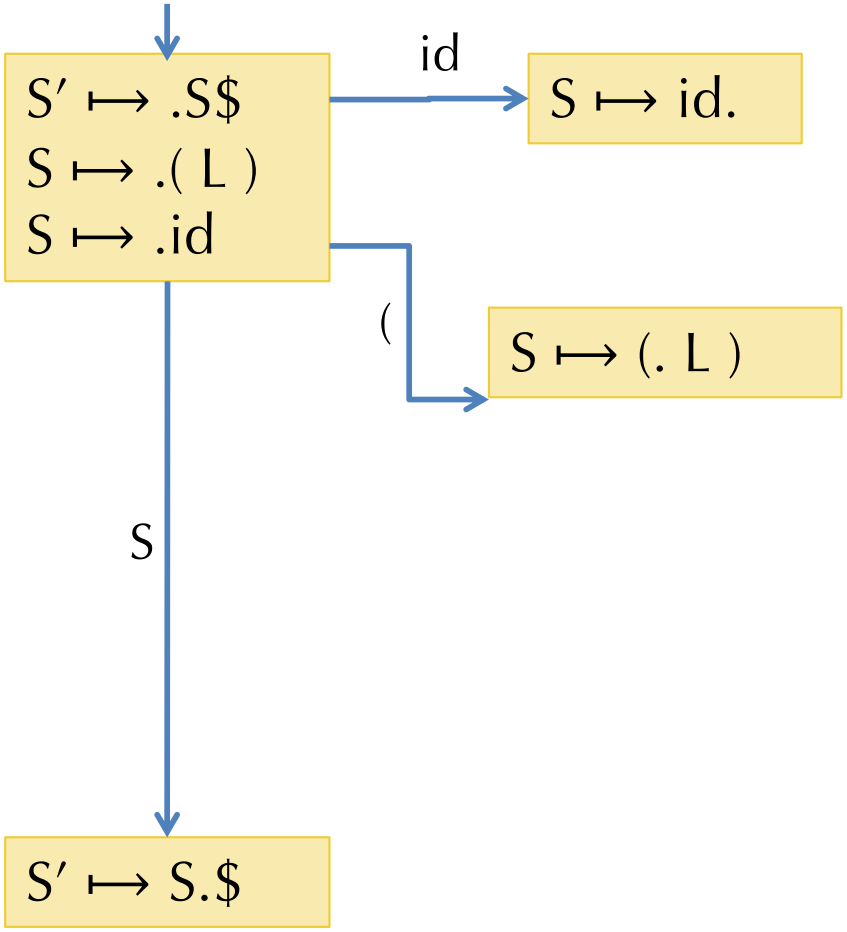
$S' \mapsto S\$$
$S \mapsto (L) \mid id$
$L \mapsto S \mid L , S$



- 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

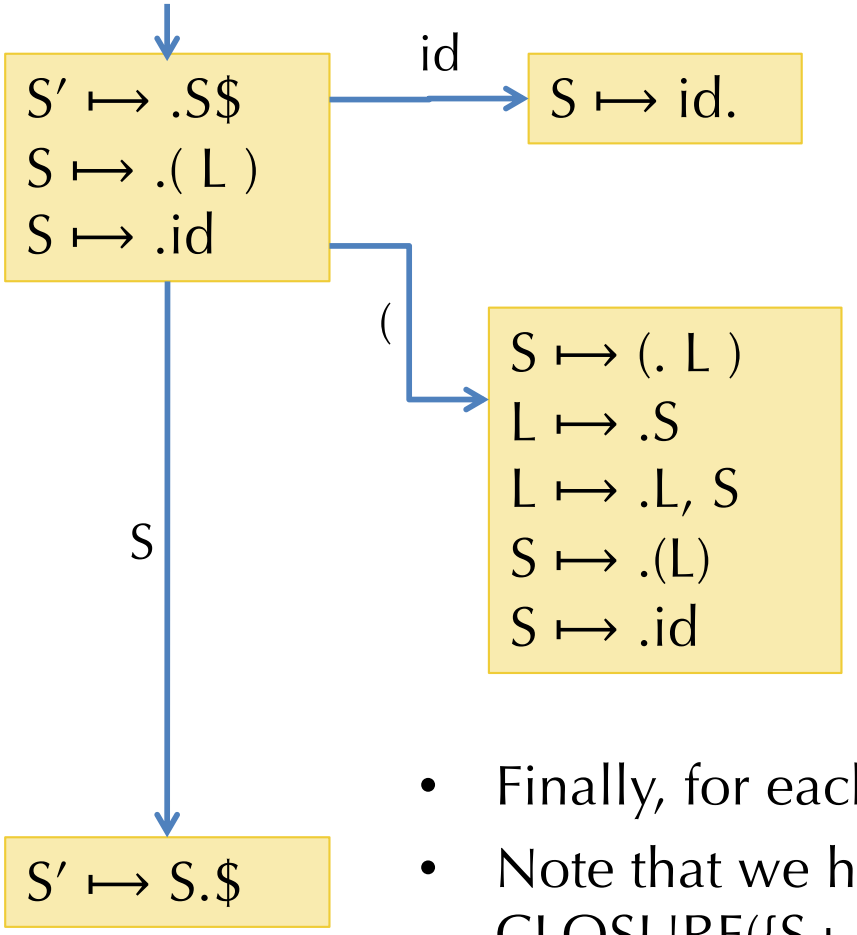


S′ ⟼ S$
S ⟼ (L) | id
L ⟼ S | L , S

- 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)

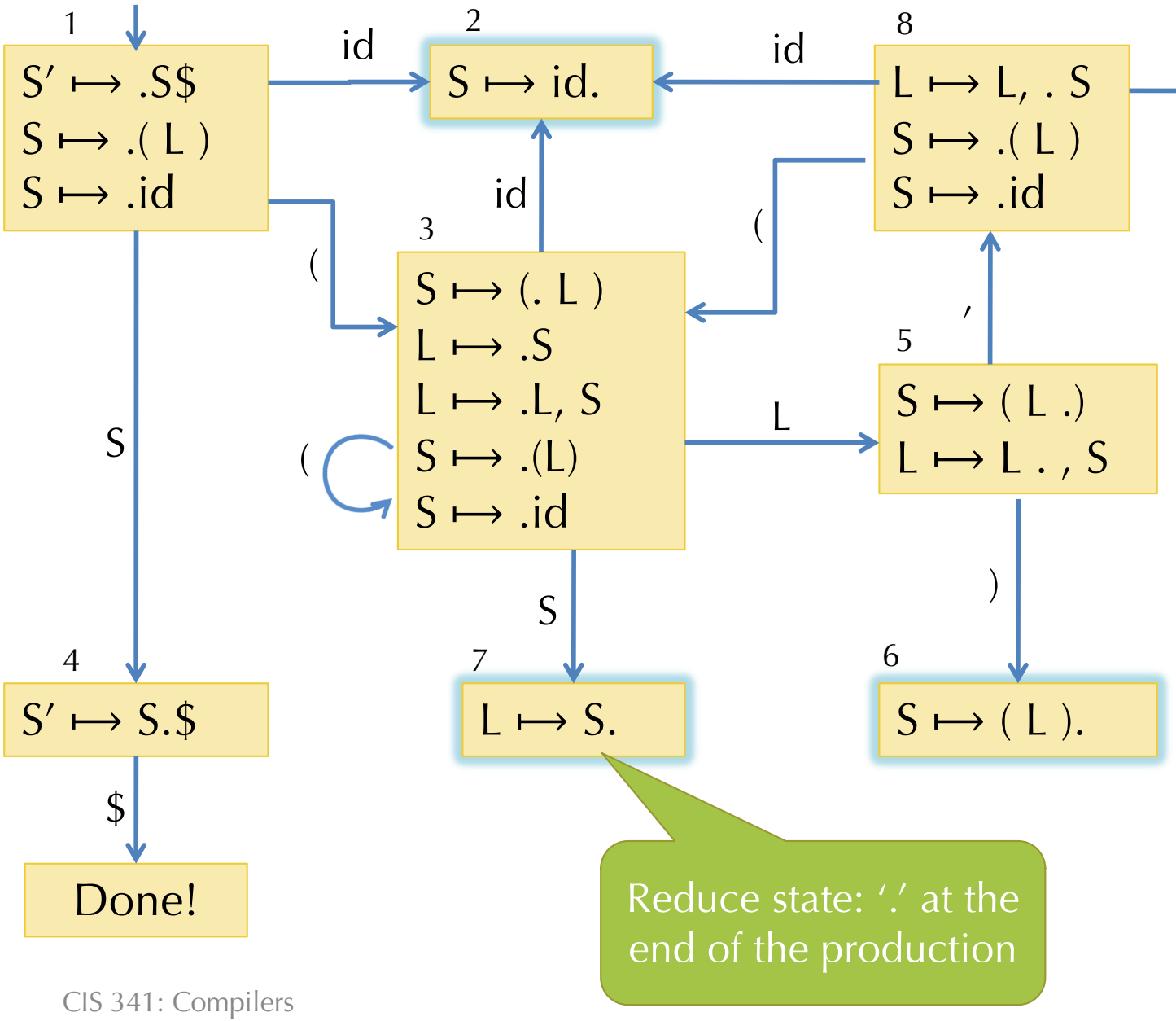
Example: Constructing the DFA

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



- 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.

Reduce state: '.' at the end of the production

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.

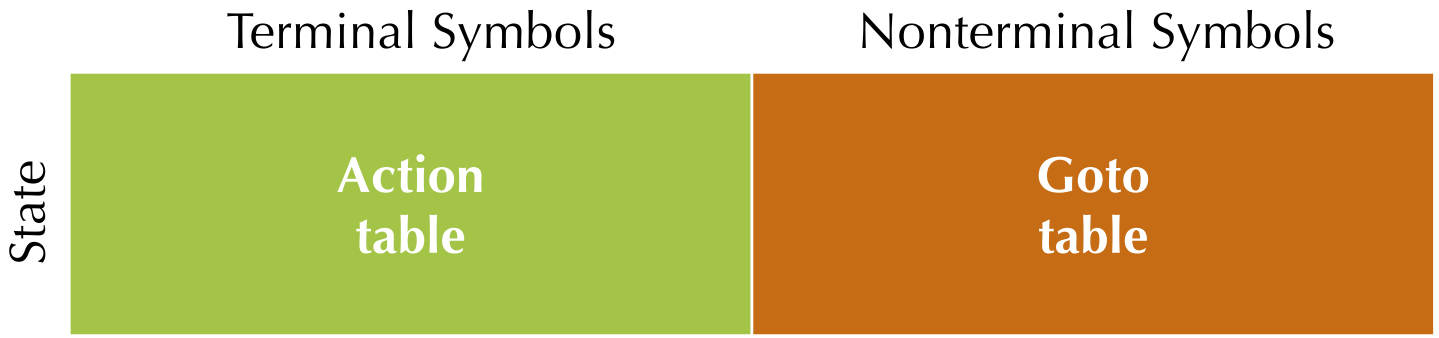
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. ${}_1({}_3({}_3L_5)_6$
  - On a reduction $X \mapsto \gamma$, pop stack to reveal the state too:
    e.g. From stack ${}_1({}_3({}_3L_5)_6$ reduce $S \mapsto (\ L\ )$ to reach stack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

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 ⟼ (L).	S ⟼ (L). L ⟼ .L , S	S ⟼ L ,S. S ⟼ ,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

S ⟼ S + E | E
E ⟼ number | (S)

right

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.

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

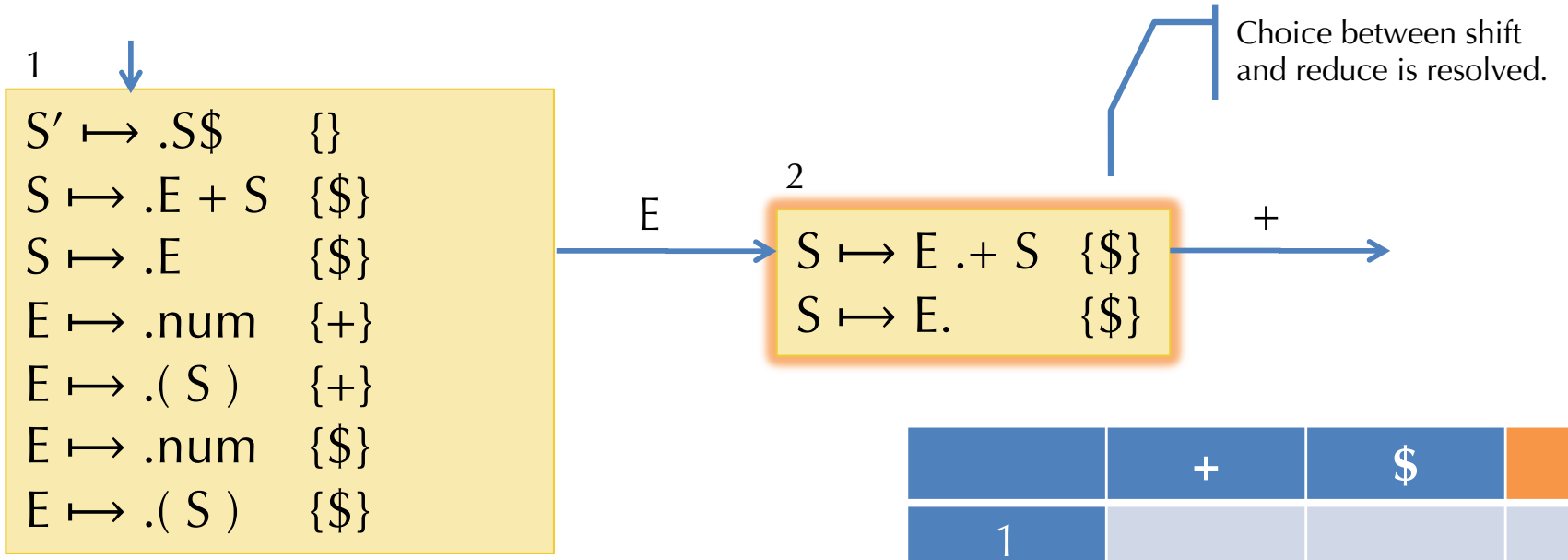
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item $C \longmapsto .\gamma$ is added because $A \longmapsto \beta.C\delta \ , \ \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$:
  1. The look-ahead set $\mathcal{M}$ includes FIRST(δ)
     (the set of terminals that may start strings derived from δ)
  2. If δ can derive ε (it is nullable), then the look-ahead $\mathcal{M}$ also contains $\mathcal{L}$

Example Closure

S′ ⟼ S$
S ⟼ E + S | E
E ⟼ number | (S)

- Start item: S′ ⟼ .S$, {}
- Since S is to the right of a '.', add:
  S ⟼ .E + S , {$} Note: {$} is FIRST($)
  S ⟼ .E , {$}
- Need to keep closing, since E appears to the right of a '.' in '.E + S':
  E ⟼ .number , {+} Note: + added for reason 1
  E ⟼ .(S) , {+}
- Because E also appears to the right of '.' in '.E' we get:
  E ⟼ .number , {$} Note: $ added for reason 2
  E ⟼ .(S) , {$}
- 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 (recall 262)
- 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