EECS 483: Compiler Construction

Lecture 22:

Parsing I: Context Free Grammars

April 7
Winter Semester 2025

Announcements

- Assignment 5 starter code and specification released.

  Delay on autograder release, will be out by Tuesday or Wednesday.

  Covers register allocation and assertion removal optimizations.

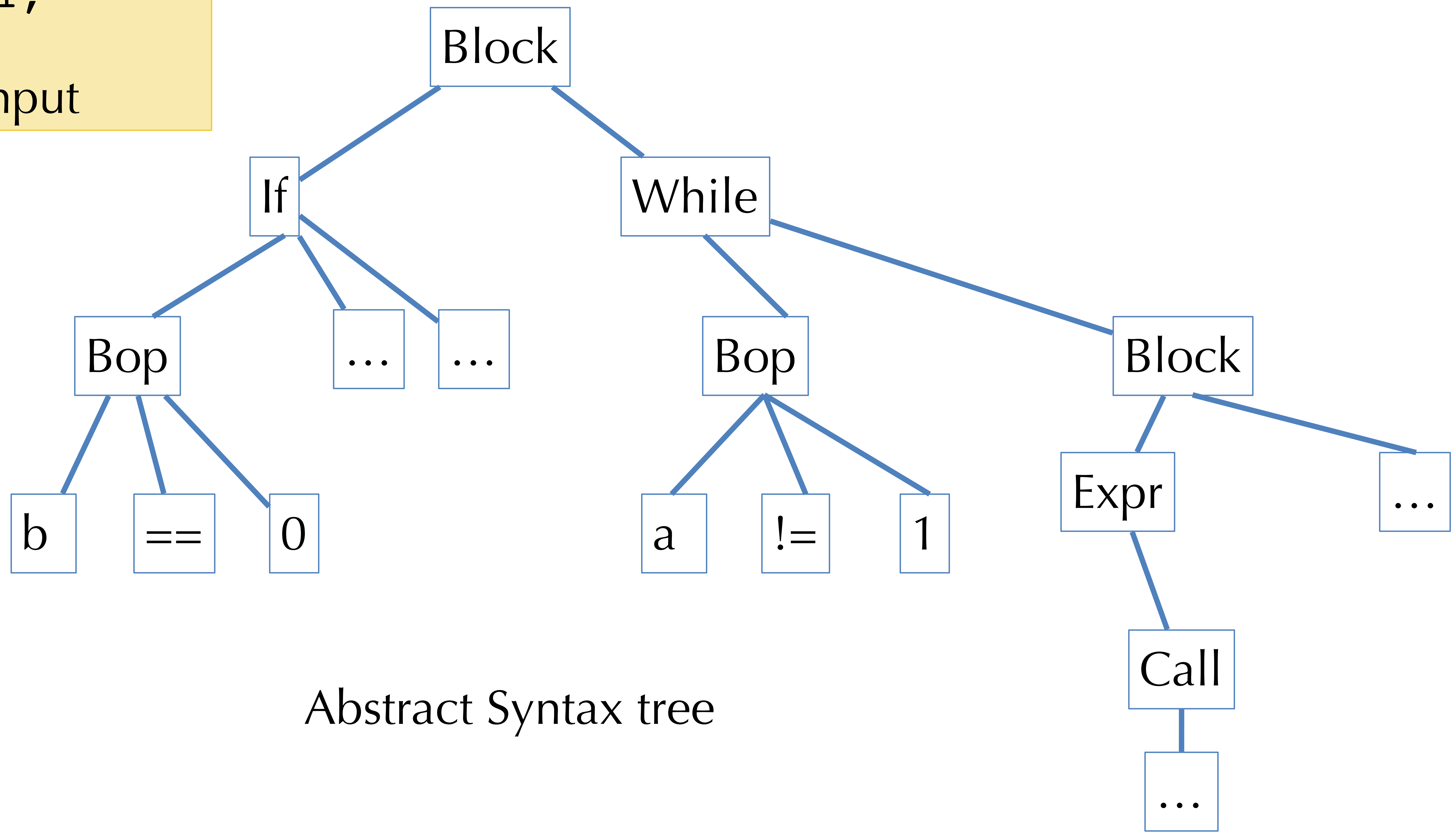
  Due on **Sunday, April 20**, with usual 2 late days.

- Last Midterm regrade requests due today. Midterm scores will be finalized on canvas by the end of the week.

Parsing: Finding Syntactic Structure

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

Source input



Abstract Syntax tree

Syntactic Analysis aka Parsing

The task of the syntactic analysis is to produce an abstract syntax tree from a token stream (the output from lexing), rejecting the input if it is not well-formed.

Similarities to lexing:

1. Input is a string (of tokens rather than characters)

2. Need to identify *if* the input is well-formed (language recognition)

3. Need to decide which of possibly multiple outputs to produce (ambiguity)

Differences:

1. Output is a **tree** rather than a string

2. Formalisms for lexing are too weak (regular expressions and finite automata)

3. Reason about ambiguity more directly in our grammar formalism (rather than only relying on tie-break rules, longest match conventions)

Parser Generators as Compilers

Writing parsers by hand is difficult, so we often use **parser generators,** similar to our lexer generators.

Adapt our formula for parser generators:

1. Design a source **language** for parsers: **context-free grammars**
2. Describe its **semantics**: **CFG**s define the set of **parse trees** for every string (not just a formal language)
3. Transform into an intermediate representation: **stack-based automata**
4. **Generate** code from the IR

Parser Generators as Compilers

Writing parsers by hand is difficult, so we often use **parser generators,** similar to our lexer generators.

The theory for parser generators is not as clean as lexer generators. Because parser generators are more powerful, they are also more computationally expensive to analyze and compile efficiently.

In practice we work with restricted versions of CFGs that support efficient algorithms and are flexible in practice.

Limitations of Regular Languages

Regular expressions and finite automata are not powerful enough to identify if an input string is well-formed for realistic programming languages.

Example:

Languages include delimited expressions like

parentheses "(1 + 2 + (3 + 4)) + 5"

braces "fn foo() { 1 }"

And reject unbalanced examples

"(1 + 2 + (3 + 4) + 5" is not valid, unmatched left paren.

Limitations of Regular Languages

Regular expressions and finite automata are not powerful enough to identify if an input string is well-formed for realistic programming languages.

Example:

Dyck language over just two character alphabet '(' and ')' consists of those words where the parentheses are balanced.

E.g., "(()()(((()))))" but not "(())(()"

Parsing a real language is at least as hard as parsing the Dyck language.

Theorem: The Dyck language is not regular, i.e., there is no regular expression expressing it.

Limitations of Regular Languages

Theorem: The Dyck language is not regular, i.e., there is no regular expression expressing it.

Proof:

Idea: Regular expressions are equivalent to finite automata, which have a finite number of states. Finite states means you can only count so high.

Consider a very long string of open parentheses ((((((...n times where n is > the number of states of the automaton. Since n is > the number of states, there must be some smaller prefix (((((...m where m < n and the automaton reaches the same state.

If the automaton decides the Dyck language, then (((...m)))...m is accepted, but therefore also (((...n)))...m is also accepted, which is incorrect.

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 $\longmapsto$ 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?

Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

S	$\longmapsto$	E + S	|	E
E	$\longmapsto$	number	|	(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 $\longmapsto$ 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
- **<u>S</u>** ⟼ **<u>E</u>** + S

⟼ (**<u>S</u>**) + S

⟼ (**<u>E</u>** + S) + S

⟼ (1 + **<u>S</u>**) + S

⟼ (1 + **<u>E</u>** + S) + S

⟼ (1 + 2 + **<u>S</u>**) + S

⟼ (1 + 2 + **<u>E</u>**) + S

⟼ (1 + 2 + (**<u>S</u>**)) + S

⟼ (1 + 2 + (**<u>E</u>** + S)) + S

⟼ (1 + 2 + (3 + **<u>S</u>**)) + S

⟼ (1 + 2 + (3 + **<u>E</u>**)) + S

⟼ (1 + 2 + (3 + 4)) + **<u>S</u>**

⟼ (1 + 2 + (3 + 4)) + **<u>E</u>**

⟼ (1 + 2 + (3 + 4)) + 5

S ⟼ E + S | E
E ⟼ number | (S)

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

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

(*substitute* β for an occurrence of A)

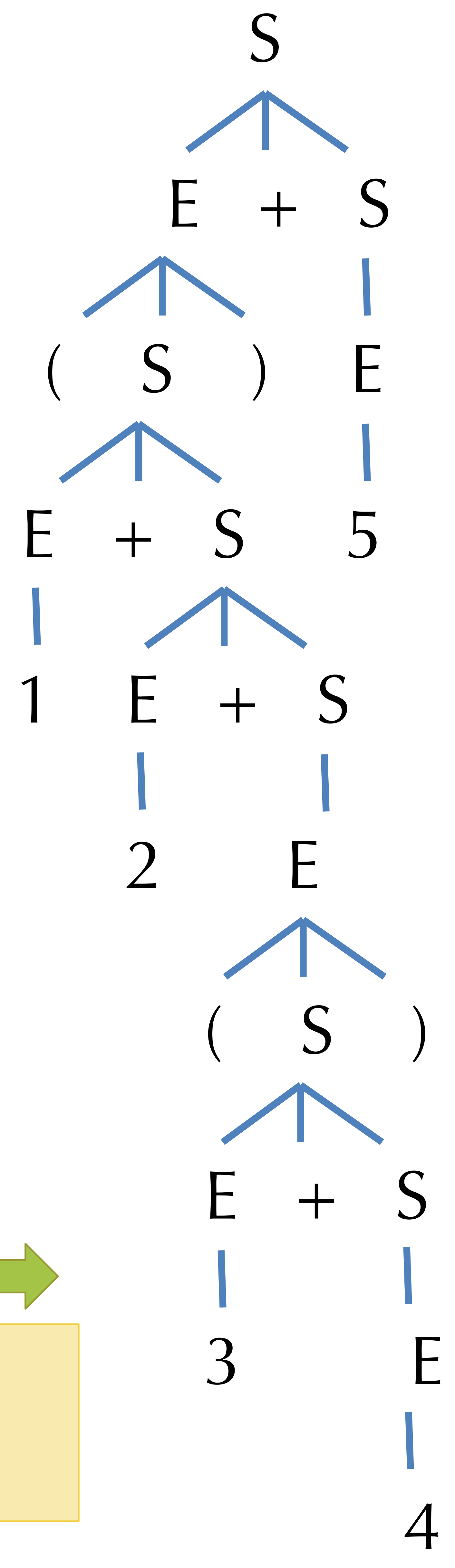
In general, there are many possible derivations for a given string

Note: Underline indicates symbol being expanded.

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



Parse Tree

S ⟼ E + S | E
E ⟼ number | (S)

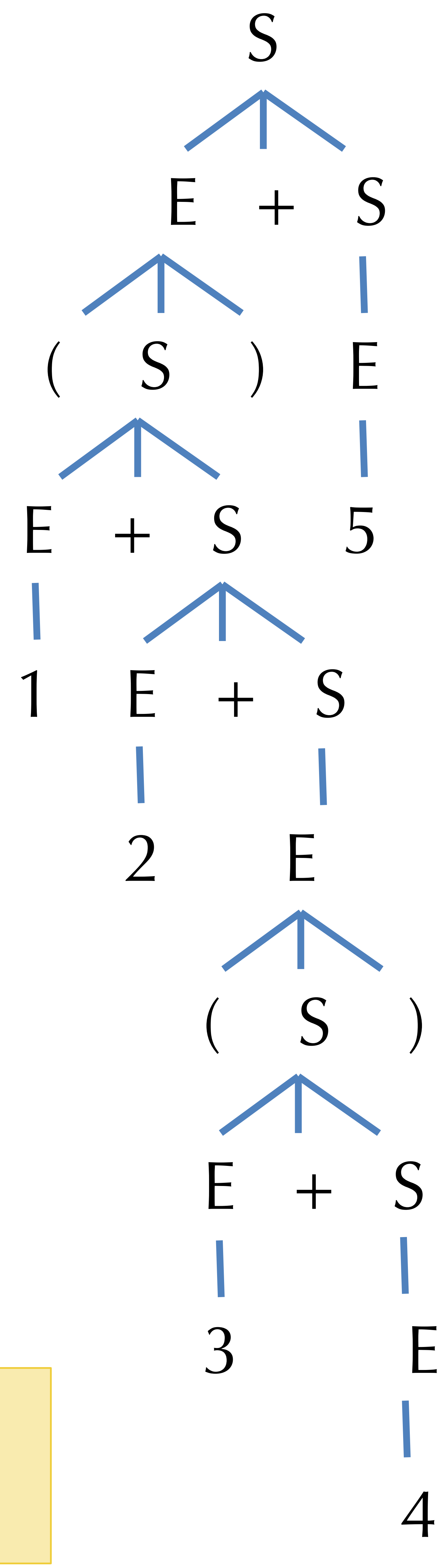
From Derivations to Parse Trees

- Idea: Think of the non-terminals of the CFG as mutually-recursive **enum** types in Rust

```
enum S {
  Plus(E,PlusSign,S),
  Exp(E)
}
enum E {
  Num(number),
  Paren(LP,S,RP)
}
```

- `Then the parse trees are the values of the enum type`

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



Parse Tree

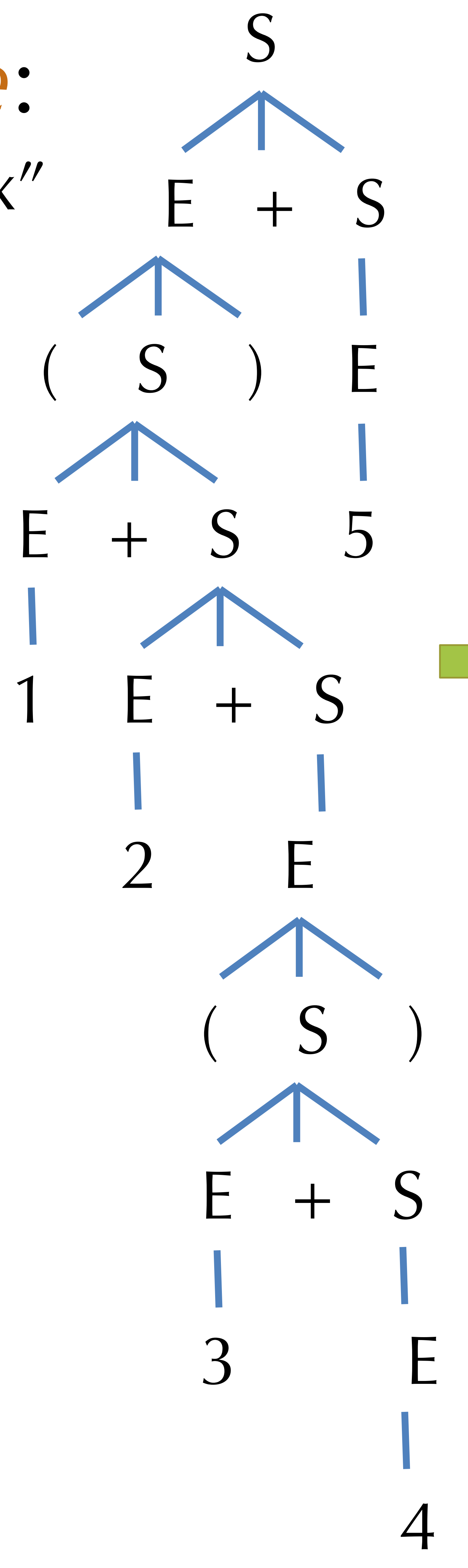
From Parse Trees to Abstract Syntax

- *Parse tree*:

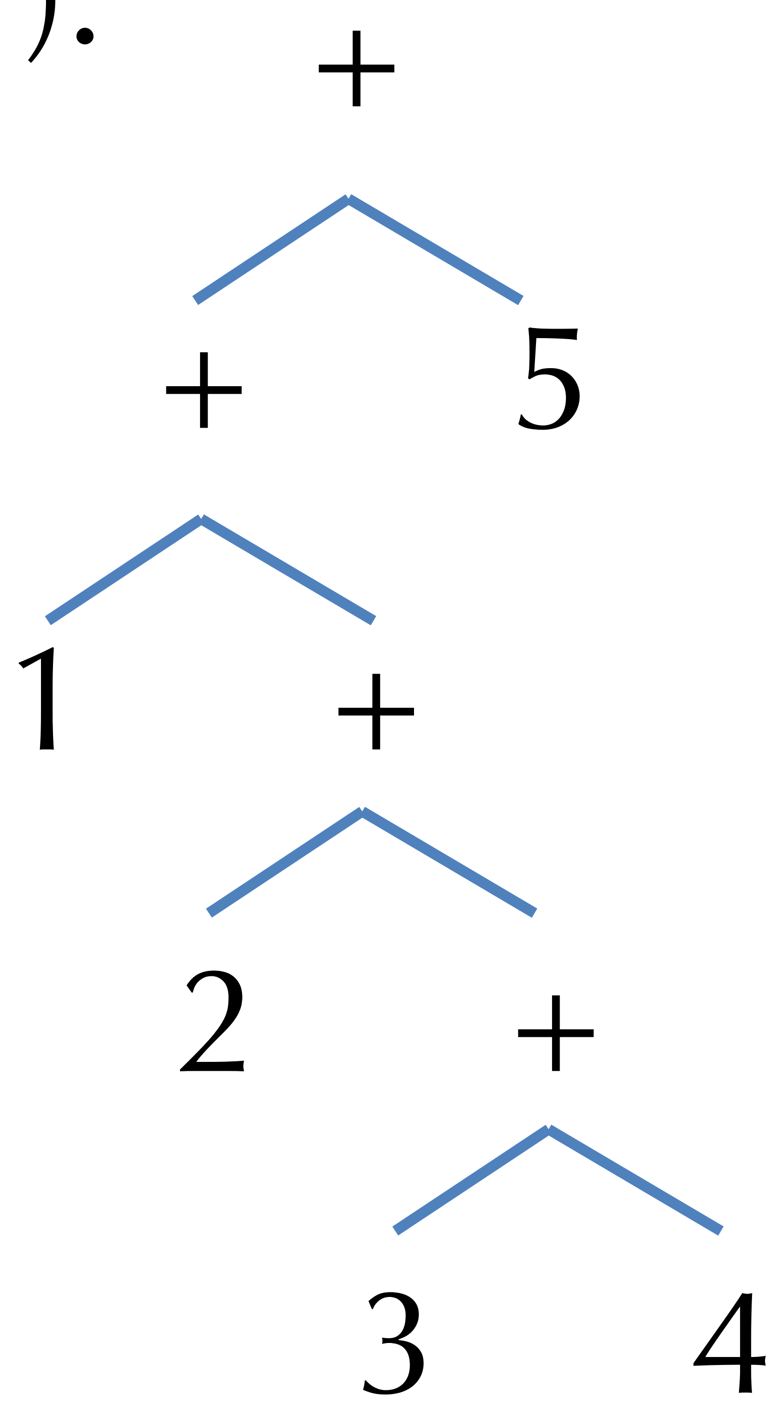
"concrete syntax"

- Parse Trees

```
enum S {
   Plus(E,PlusSign,S),
   Exp(E)
}
enum E {
   Num(number),
   Paren(LP,S,RP)
}
```



- *Abstract syntax tree* (AST):



- AST

```
enum Exp {
   Plus(Exp,Exp),
   Num(number)
}
```

- Hides, or *abstracts*, unneeded information.

$S \longmapsto E + S \mid E$

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

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.

- Idea: These are ***search strategies*** for finding a parse tree
  - Both strategies (and any other) yield the same parse tree!
  - Parse tree doesn’t contain the information about what order the productions were applied.
  - Just like an enum value doesn't tell you an order in which its subtrees were constructed.

Example: Left- and rightmost derivations

$S \longmapsto E + S \mid E$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Rightmost derivation:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

Regular Expressions to CFGs

Theorem: every Regex can be expressed as a CFG, i.e. there is a CFG that generates exactly the strings in the Regex

- ‘a’ — S -> a
- ∅ — S with no productions
- ε — S -> ε
- $R_1 \mid R_2$ — $S \to S_1$
  $S \to S_2$
  where $S_1 \to \ldots$ and $S_2 \to \ldots$ are CFGs for R_1, R_2
- R_1R_2 — $S \to S_1\ S_2$
  where S_1, S_2 are CFGs for R_1, R_2
- R* — $S \to \varepsilon$
  $S \to S_R\ S$
  where S_R are CFGs for R

Associativity, ambiguity, and precedence.

GRAMMARS FOR PROGRAMMING LANGUAGES

Associativity

S ↦ E + S | E
E ↦ number | (S)

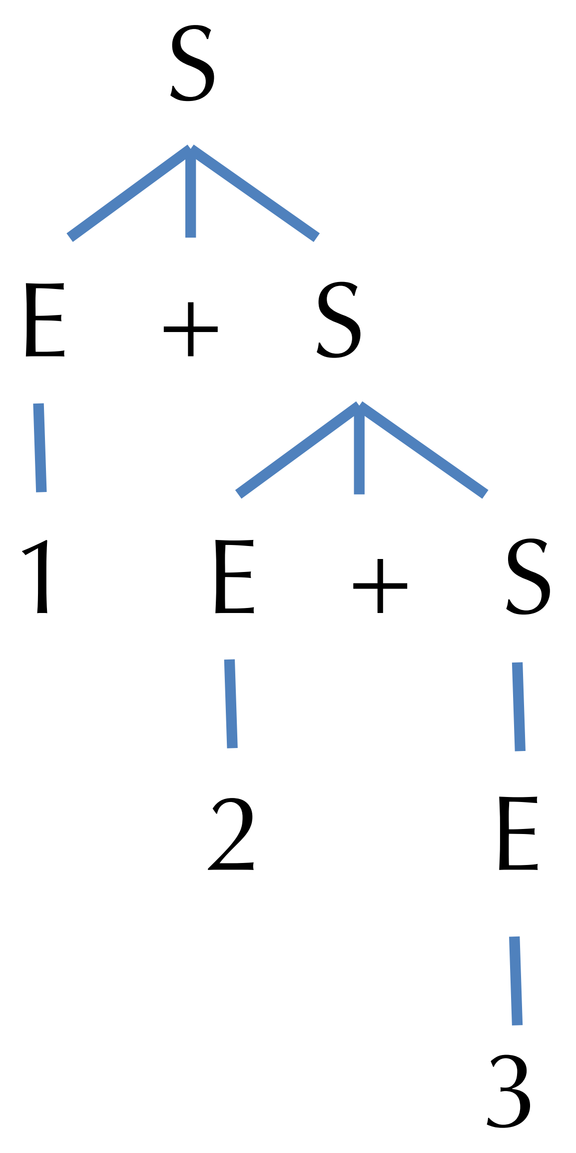
Consider the input: 1 + 2 + 3

Leftmost derivation:

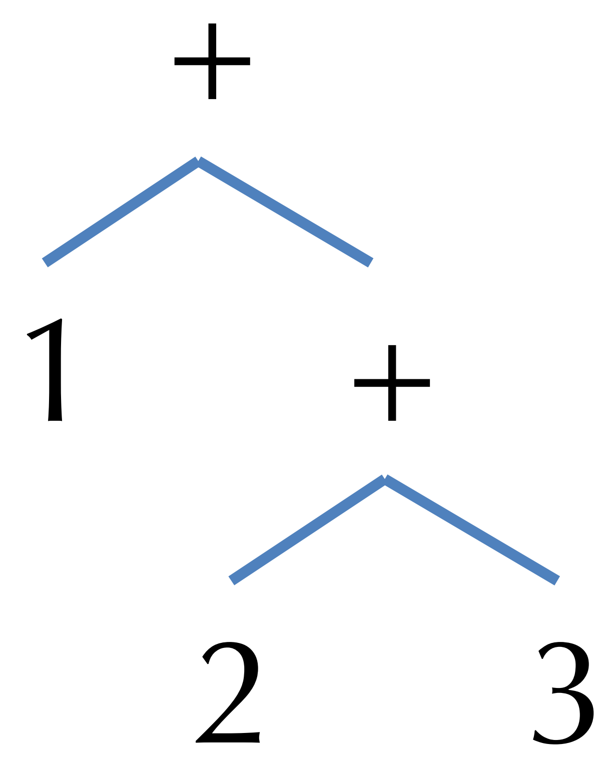
S ↦ **E** + S
↦ 1 + **S**
↦ 1 + **E** + S
↦ 1 + 2 + **S**
↦ 1 + 2 + **E**
↦ 1 + 2 + 3

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 refers to itself on the right of +

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

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

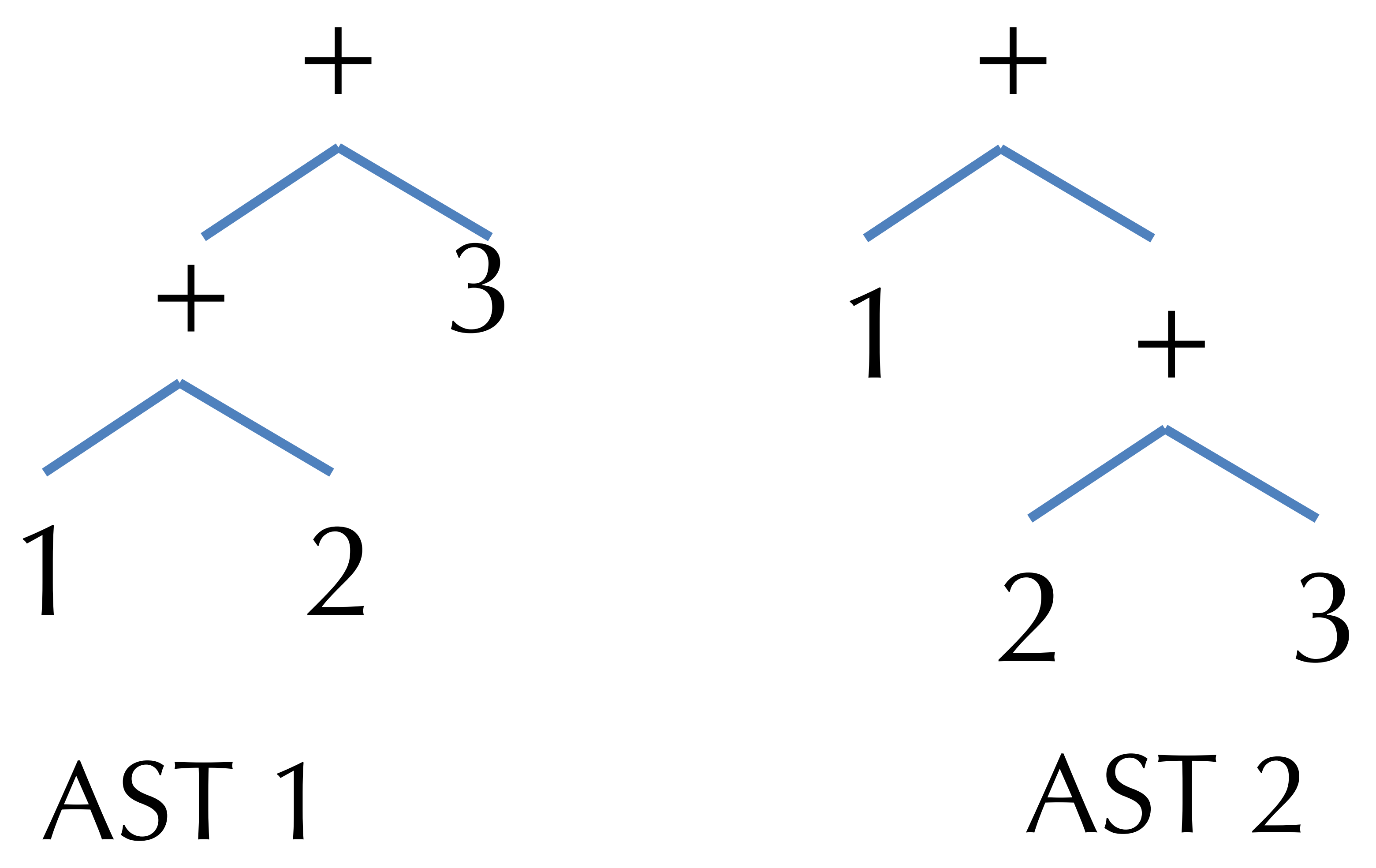
Ambiguity

- Consider this grammar:

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

- 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?



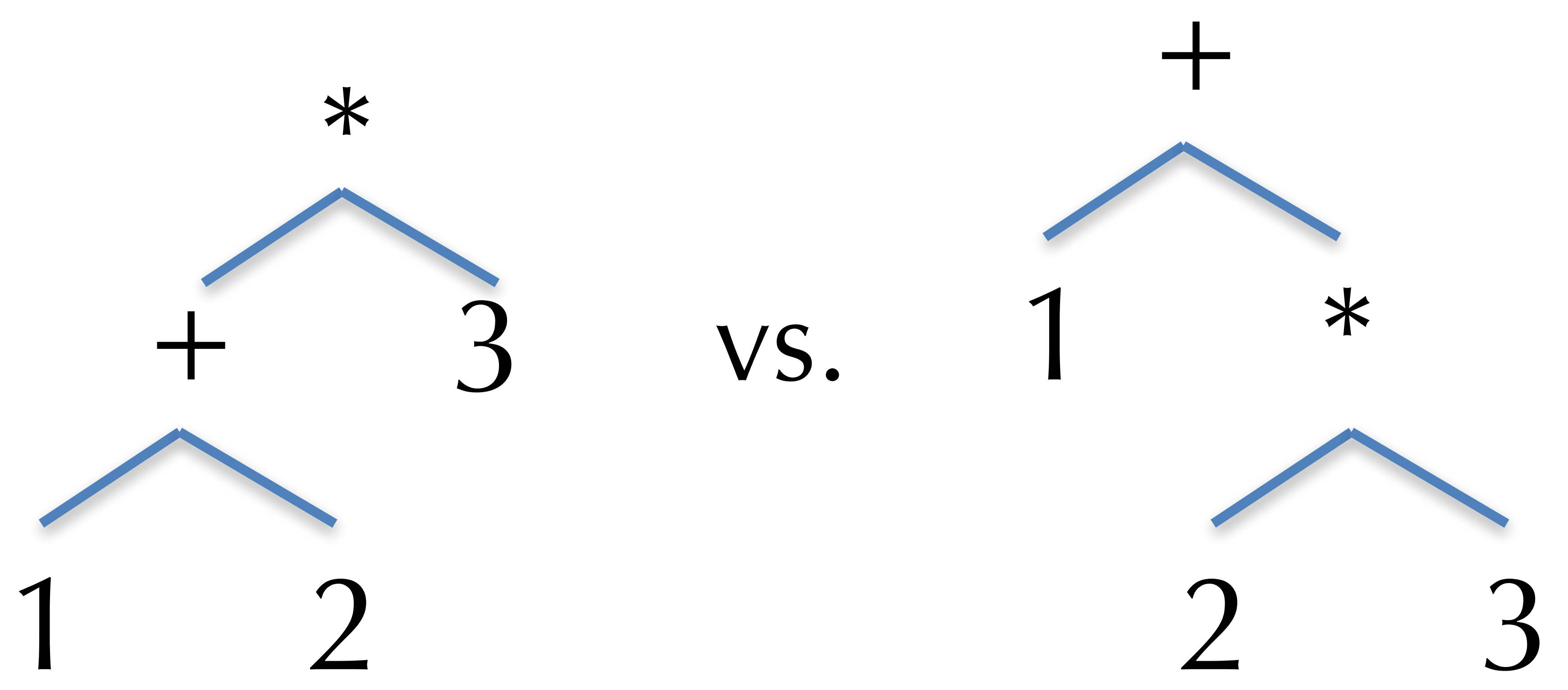
Why do we care about ambiguity?

- 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 ⟼ S + S | S * S | (S) | 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:

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

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

$$S_0 \longmapsto S_0 + S_1 \;|\; S_1$$

$$S_1 \longmapsto S_2 * S_1 \;|\; S_2$$

$$S_2 \longmapsto \text{number} \;|\; (S_0)$$

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
    - Parse trees describe what productions were used to construct the string
    - Derivations describe a trace of what order the productions were used in
  - 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 parse tree.

- Next time: parsing CFGs