

Lecture 10

# CIS 341: COMPILERS

# Announcements

- **Homework 3:** Compiling LLVMlite
- **Goal:**
  - Familiarize yourself with (a subset of) the LLVM IR
  - Implement a translation down to (inefficient) X86lite

- **Due:** Monday, Feb. 26<sup>th</sup>

*it is nearly too late to*  
***START EARLY!!***

- **MIDTERM EXAM**
  - Thursday, March 1<sup>st</sup> in class
  - Coverage: interpreters / program transformers / x86 / calling conventions / IRs / LLVM / Lexing / Parsing
  - See examples on the web pages

Creating an abstract representation of program syntax.

# PARSING

# Today: Parsing

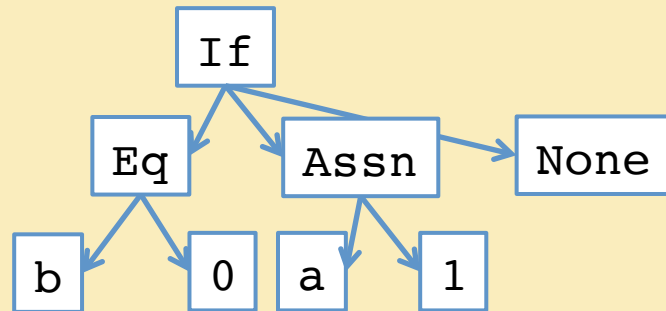
Source Code  
(Character stream)

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

Token stream:

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

Abstract Syntax Tree:



Intermediate code:

```
11:
    %cnd = icmp eq i64 %b, 0
    br i1 %cnd, label %12,
    label %13
12:
    store i64* %a, 1
    br label %13
13:
```

Lexical Analysis

Parsing

Analysis & Transformation

Backend

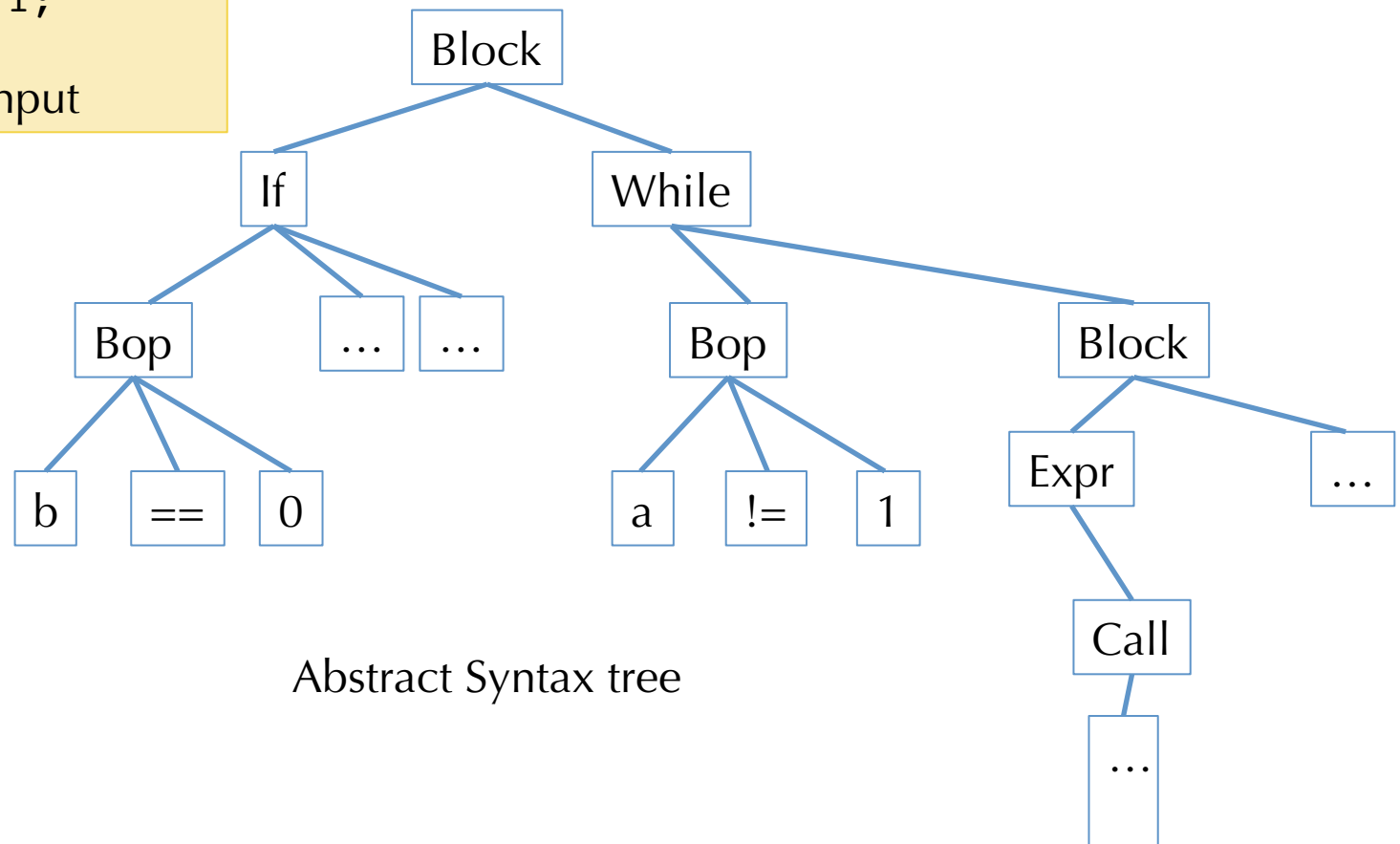
Assembly Code

```
11:
    cmpq %eax, $0
    jeq 12
    jmp 13
12:
    ...
```

# Parsing: Finding Syntactic Structure

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

Source input

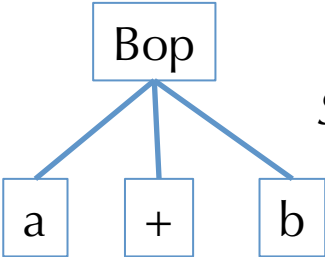


Abstract Syntax tree

# Syntactic Analysis (Parsing): Overview

- Input: stream of tokens (generated by lexer)
- Output: abstract syntax tree
- Strategy:
  - Parse the token stream to traverse the “concrete” syntax
  - During traversal, build a tree representing the “abstract” syntax
- Why abstract? Consider these three *different* concrete inputs:

$a + b$   
 $(a + ((b)))$   
 $((a) + (b))$



```
graph TD; Bop[Bop] --> a[a]; Bop --> plus[+]; Bop --> b[b];
```

Same abstract syntax tree
- Note: parsing doesn't check many things:
  - Variable scoping, type agreement, initialization, ...

# Specifying Language Syntax

- First question: how to describe language syntax precisely and conveniently?
- Last time: we described tokens using regular expressions
  - Easy to implement, efficient DFA representation
  - Why not use regular expressions on tokens to specify programming language syntax?
- Limits of regular expressions:
  - DFA's have only finite # of states
  - So... DFA's can't "count"
  - For example, consider the language of all strings that contain balanced parentheses – easier than most programming languages, but not regular.
- So: we need more expressive power than DFA's

# CONTEXT FREE GRAMMARS



# Context-free Grammars

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

$$S \mapsto (S)S$$

$$S \mapsto \varepsilon$$

Note: Once again we have to take care to distinguish meta-language elements (e.g. “ $S$ ” and “ $\mapsto$ ”) 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:  $S \mapsto (S)S \mapsto ((S)S)S \mapsto ((\varepsilon)S)S \mapsto ((\varepsilon)S)\varepsilon \mapsto ((\varepsilon)\varepsilon)\varepsilon = (())$
- You can replace the “*nonterminal*”  $S$  by its definition anywhere
- A context-free grammar accepts a string iff there is a derivation from the start symbol

# CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical token or  $\epsilon$ )
  - A set of *nonterminals* (e.g.,  $S$  and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of productions:  $\text{LHS} \mapsto \text{RHS}$ 
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals

- Example: The balanced parentheses language:

$$S \mapsto (S)S$$

$$S \mapsto \epsilon$$

- How many terminals? How many nonterminals? Productions?

## Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

$$\begin{array}{l} S \mapsto E + S \quad | \quad E \\ E \mapsto \text{number} \quad | \quad ( S ) \end{array}$$

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

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

$S \mapsto E + S$

$S \mapsto E$

$E \mapsto \text{number}$

$E \mapsto (S)$

4 productions

2 nonterminals: S, E

4 terminals: (, ), +, number

Start symbol: S

# Derivations in CFGs

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

- $\underline{S} \mapsto \underline{E} + S$

$$\mapsto (\underline{S}) + S$$

$$\mapsto (\underline{E} + S) + S$$

$$\mapsto (1 + \underline{S}) + S$$

$$\mapsto (1 + \underline{E} + S) + S$$

$$\mapsto (1 + 2 + \underline{S}) + S$$

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

$$\mapsto (1 + 2 + (\underline{S})) + S$$

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

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

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

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

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

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

$$S \mapsto E + S \mid E$$

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

For arbitrary strings  $\alpha, \beta, \gamma$  and production rule  $A \mapsto \beta$   
a single step of the derivation is:


$$\alpha A \gamma \mapsto \alpha \beta \gamma$$

( *substitute*  $\beta$  for an occurrence of  $A$  )

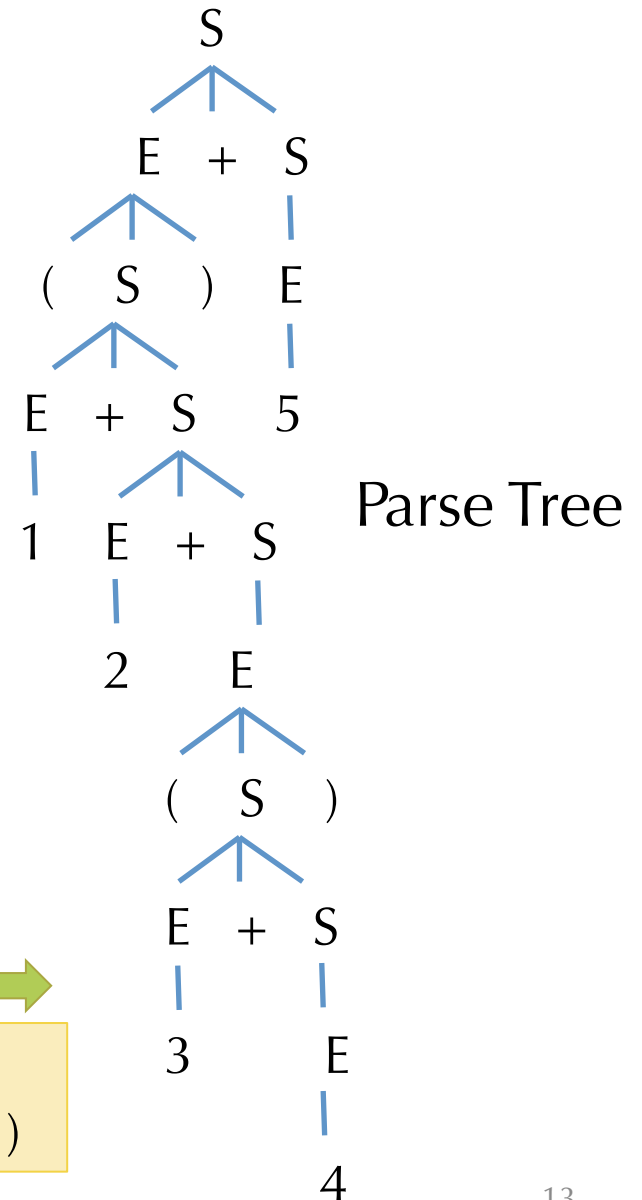
In general, there are many possible derivations for a given string

Note: Underline indicates symbol being expanded.

# From Derivations to Parse Trees

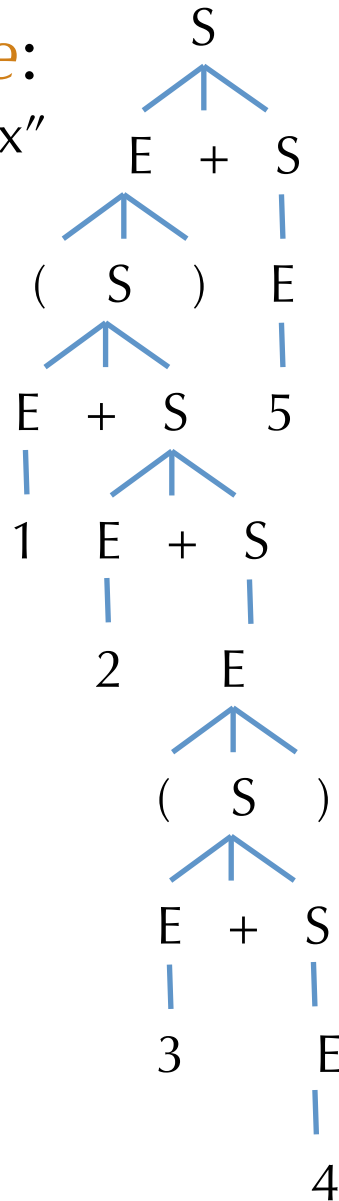
- Tree representation of the derivation
- Leaves of the tree are terminals
  - In-order traversal yields the input sequence of tokens
- Internal nodes: nonterminals
- No information about the *order* of the derivation steps
- $(1 + 2 + (3 + 4)) + 5$  

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

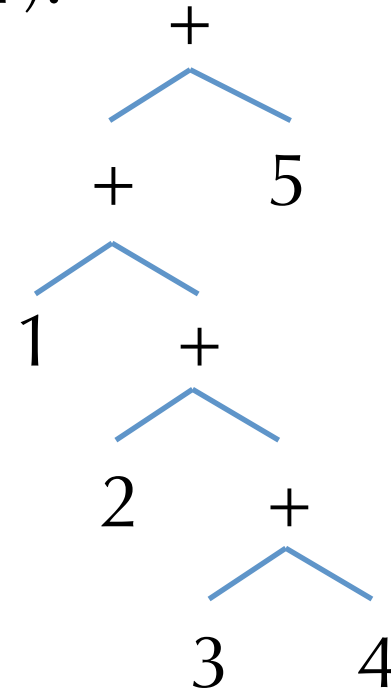


# From Parse Trees to Abstract Syntax

- *Parse tree*:  
“concrete syntax”



- *Abstract syntax tree*  
(AST):



- Hides, or *abstracts*,  
unnneeded 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:

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

- Rightmost derivation:

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

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



# Loops and Termination

- Some care is needed when defining CFGs
- Consider:

$$\begin{array}{l} S \mapsto E \\ E \mapsto S \end{array}$$

- 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:  $S \mapsto ( S )$

- 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 “chain” 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:  $1 + 2 + 3$

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

Leftmost derivation: Rightmost derivation:

$\underline{S} \mapsto \underline{E} + S$

$\mapsto 1 + \underline{S}$

$\mapsto 1 + \underline{E} + S$

$\mapsto 1 + 2 + \underline{S}$

$\mapsto 1 + 2 + \underline{E}$

$\mapsto 1 + 2 + 3$

$\underline{S} \mapsto E + \underline{S}$

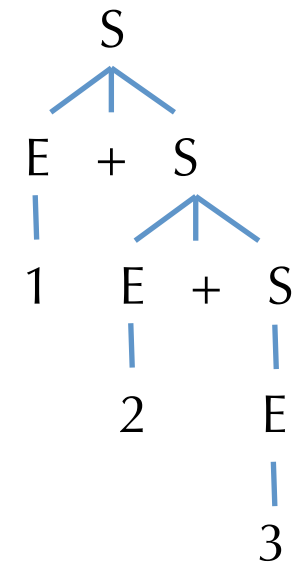
$\mapsto E + E + \underline{S}$

$\mapsto E + E + \underline{E}$

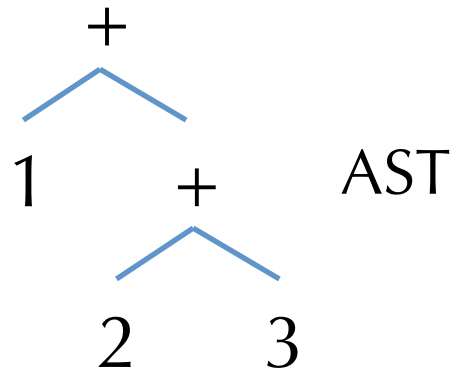
$\mapsto E + \underline{E} + 3$

$\mapsto \underline{E} + 2 + 3$

$\mapsto 1 + 2 + 3$



Parse Tree



AST

# Associativity

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

$$\begin{array}{l} S \mapsto E + S \mid E \\ E \mapsto \text{number} \mid ( S ) \end{array}$$

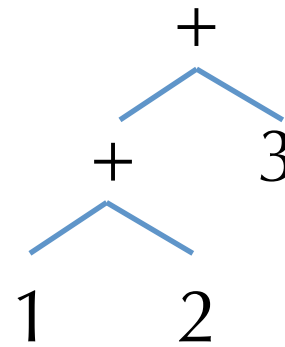
- How would you make '+' left associative?
- What are the trees for " $1 + 2 + 3$ "?

# Ambiguity

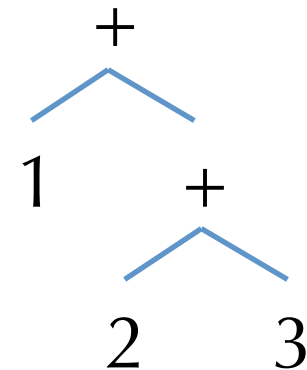
- Consider this grammar:

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

- Claim: it accepts the *same* set of strings as the previous one.
- What's the difference?
- Consider these *two* leftmost derivations:
  - $\underline{S} \mapsto \underline{S} + S \mapsto 1 + \underline{S} \mapsto 1 + \underline{S} + S \mapsto 1 + 2 + \underline{S} \mapsto 1 + 2 + 3$
  - $\underline{S} \mapsto \underline{S} + S \mapsto \underline{S} + S + S \mapsto 1 + \underline{S} + S \mapsto 1 + 2 + \underline{S} \mapsto 1 + 2 + 3$
- One derivation gives left associativity, the other gives right associativity to '+'
  - Which is which?



AST 1



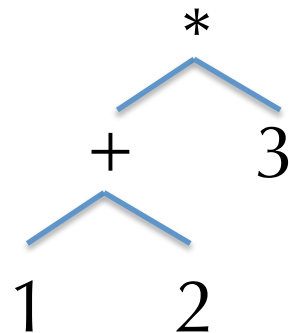
AST 2

# Why do we care about ambiguity?

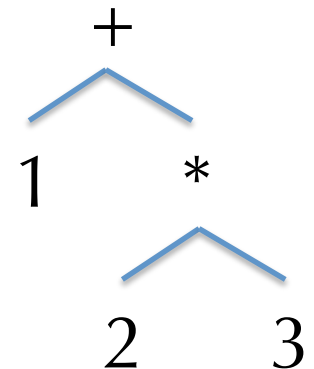
- The '+' operation is associative, so it doesn't matter which tree we pick. Mathematically,  $x + (y + z) = (x + y) + z$ 
  - But, some operations aren't associative. Examples?
  - Some operations are only left (or right) associative. Examples?
- Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*
- Consider:

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

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



vs.



# 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:

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

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

$$\begin{array}{lll} S_0 \mapsto & S_0 + S_1 & \mid S_1 \\ S_1 \mapsto & S_2 * S_1 & \mid S_2 \\ S_2 \mapsto & \text{number} & \mid ( S_0 ) \end{array}$$

# 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