LECTURE 4

Syntax

SPECIFYING SYNTAX

Programming languages must be very well defined – there's no room for ambiguity.

Language designers must use formal syntactic and semantic notation to specify the rules of a language.

In this lecture, we will focus on how syntax is specified.

SPECIFYING SYNTAX

We know from the previous lecture that the front-end of the compiler has three main phases:

- Scanning
- Parsing
- Semantic Analysis

Syntax Verification

SPECIFYING SYNTAX

- Scanning
  - Identifies the valid *tokens*, the basic building blocks, within a program.
- Parsing
  - Identifies the valid patterns of tokens, or *constructs*.

So how do we specify what a valid token is? Or what constitutes a valid construct?

REGULAR EXPRESSIONS

Tokens can be constructed from regular characters using just three rules:

1. Concatenation.
2. Alternation (choice among a finite set of alternatives).
3. Kleene Closure (arbitrary repetition).

Any set of strings that can be defined by these three rules is a *regular set*.
Regular sets are generated by *regular expressions*.

REGULAR EXPRESSIONS

Formally, all of the following are valid regular sets (let R and S be regular sets and let Σ be a finite set of symbols):

- The empty set.
- The set containing the empty string ϵ.
- The set containing a single literal character α from the alphabet Σ.
- Concatenation: RS is the set of strings obtained by concatenation of one string from R with a string from S.
- Alternation: R|S describes the union of R and S.
- Kleene Closure: R* is the set of strings that can be obtained by concatenating any number of strings from R.

REGULAR EXPRESSIONS

You can either use parentheses to avoid ambiguity or assume Kleene star has the highest priority, followed by concatenation then alternation.

Examples:

- $a^* = \{\epsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$
- $a \mid b^* = \{\epsilon, a, b, bb, bbb, bbbb, \ldots\}$
- $(ab)^* = \{\epsilon, ab, abab, ababab, abababab, \ldots\}$
- $(a|b)^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

REGULAR EXPRESSIONS

Write a regular expression for each of the following:

- Zero or more c’s followed by a single a or a single b.
- Binary strings starting and ending with 1.
- Binary strings containing at least 3 1’s.

REGULAR EXPRESSIONS

Write a regular expression for each of the following:

- Zero or more c's followed by a single a or a single b.
  - c*(a|b)
- Binary strings starting and ending with 1.
  - 1|1(0|1)*1
- Binary strings containing at least 3 1's.
  - 0*10*10*1(0|1)*

REGULAR EXPRESSIONS

Let’s look at a more practical example. Say we want to write a regular expression to identify valid numbers.

Some things to consider:

- Numbers can be any number of digits long, but must not start with 0.
- Numbers can only be positive.
- Numbers can be integers or real.
- Numbers can be represented by scientific notation (i.e. 2.9e8).

REGULAR EXPRESSIONS

number → *integer* | *real*

integer → *non_zero_digit digit**

real → *integer exponent* | *decimal* (*exponent* | ϵ)

decimal → integer (. *digit*) *digit**

exponent → (e | E) (+ | - | ϵ) *integer*

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

REGULAR EXPRESSIONS

So our number tokens are well-defined by the *number* symbol, which makes use of the other symbols to build larger expressions.
Any valid pattern generated by expanding out the *number* symbol is a valid number.
Note: while our rules build upon one another, no symbol is defined in terms of itself, even indirectly.

number → *integer* | *real*

integer → *non_zero_digit digit**

real → *integer exponent* | *decimal* (*exponent* | ϵ)

decimal → integer (. *digit*) *digit**

exponent → (e | E) (+ | - | ϵ) *integer*

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

CONTEXT-FREE GRAMMARS

We can completely define our tokens in terms of regular expressions, but more complicated constructs necessitate the ability to self-reference.

This self-referencing ability takes the form of *recursion*.

The set of strings that can be defined by adding recursion to regular expressions is known as a *Context-Free Language*.

Context-Free Languages are generated by *Context-Free Grammars*.

CONTEXT-FREE GRAMMARS

We've seen a little bit of context-free grammars, but let's flesh out the details.

Context-free grammars are composed of rules known as *productions*.

Each production has left-hand side symbols known as *non-terminals,* or *variables*.

On the right-hand side, a production may contain *terminals* (tokens) or other non-terminals.

One of the non-terminals is named the *start symbol*.

expr → `id` | `number` | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

This notation is known as Backus-Naur Form.

DERIVATIONS

So, how do we use the context-free grammar to generate syntactically valid strings of terminals (or tokens)?

1. Begin with the start symbol.
2. Choose a production with the start symbol on the left side.
3. Replace the start symbol with the right side of the chosen production.
4. Choose a non-terminal A in the resulting string.
5. Replace A with the right side of a production whose left side is A.
6. Repeat 4 and 5 until no non-terminals remain.

DERIVATIONS

Let's do a practice *derivation* with our grammar.
We'll derive the string "(base1 + base2) * height/2".
The start symbol is *expr*.

expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

Each string of symbols in the steps of the derivation is called a *sentential form*.
The final sentential form is known as the *yield*.

DERIVATIONS

To save a little bit of room, we can write:

$expr \Rightarrow^{*}$ (id + id) * id / number

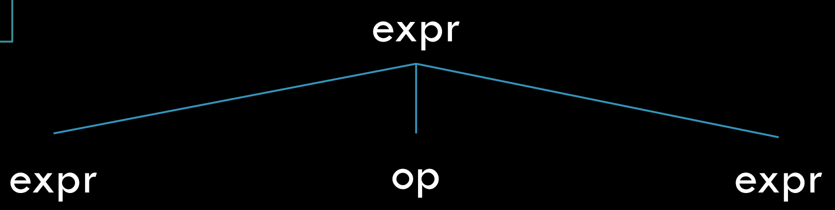
"derives after zero or more replacements"

Note that in this derivation, we replaced the right-hand side consistently, leading to a *right-most derivation*. There are alternative derivation methods.

PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

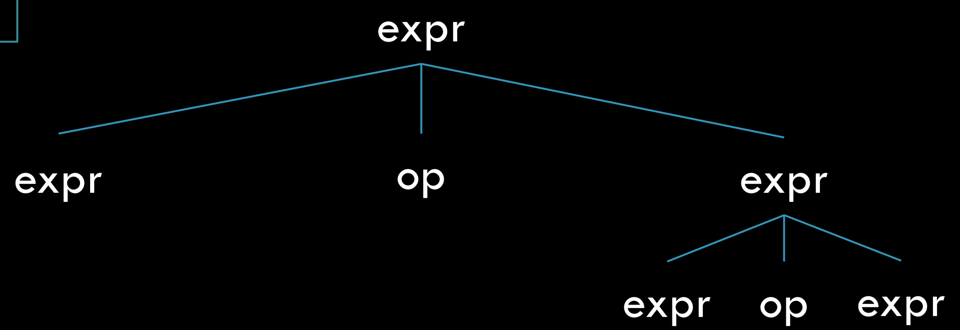
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

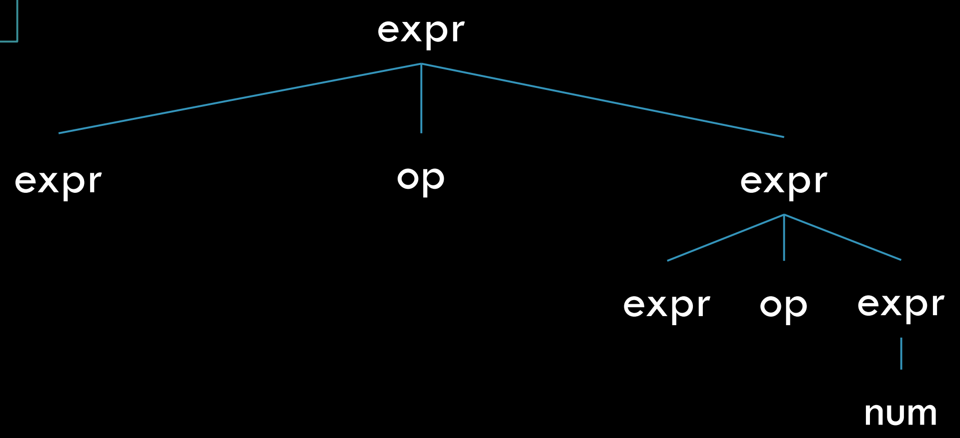
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

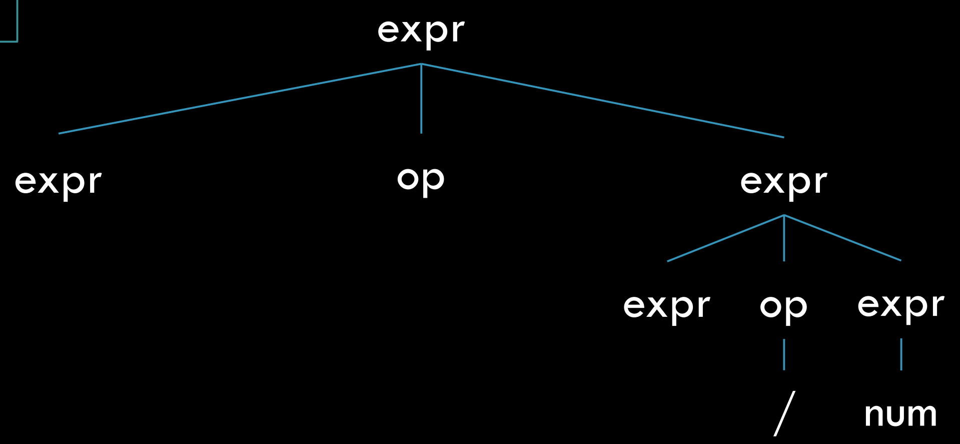
expr → *expr op expr*
→ *expr op expr op expr*
→ *expr op expr op* number
→ *expr op expr* / number
→ *expr op* id / number
→ *expr* * id / number
→ (*expr*) * id / number
→ (*expr op expr*) * id / number
→ (*expr op* id) * id / number
→ (*expr* + id) * id / number
→ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

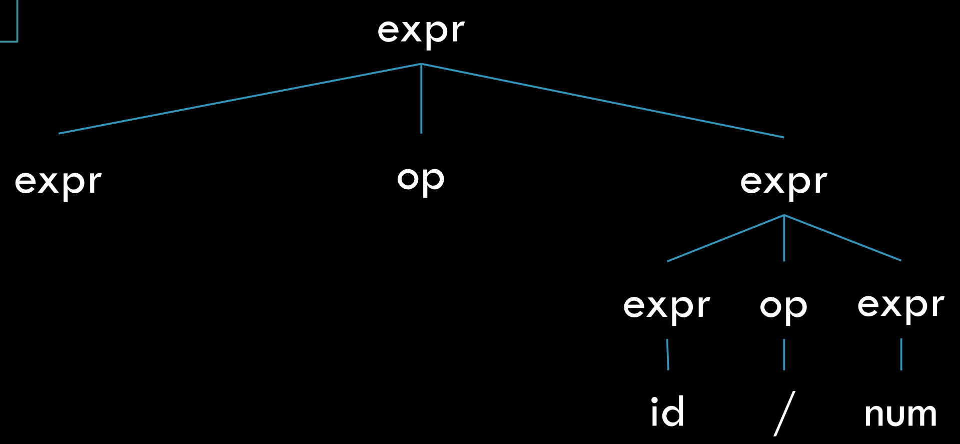
expr → *expr op expr*
→ *expr op expr op expr*
→ *expr op expr op* number
→ *expr op expr* / number
→ *expr op* id / number
→ *expr* * id / number
→ (*expr*) * id / number
→ (*expr op expr*) * id / number
→ (*expr op* id) * id / number
→ (*expr* + id) * id / number
→ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

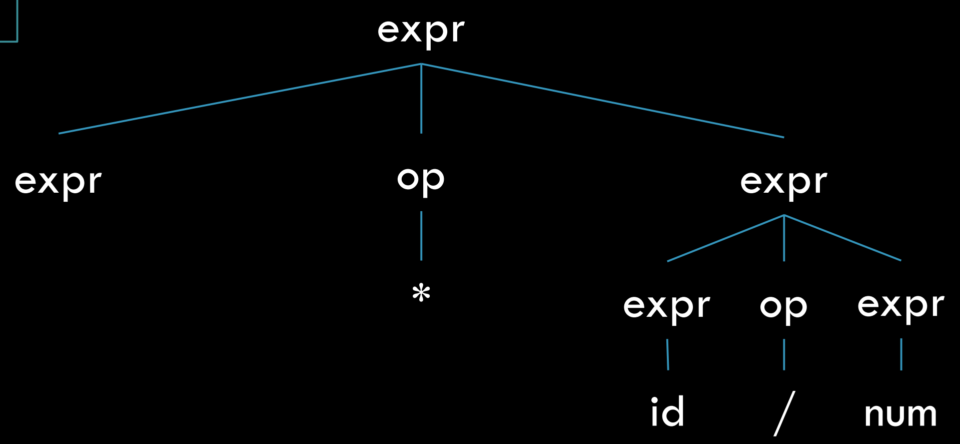
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

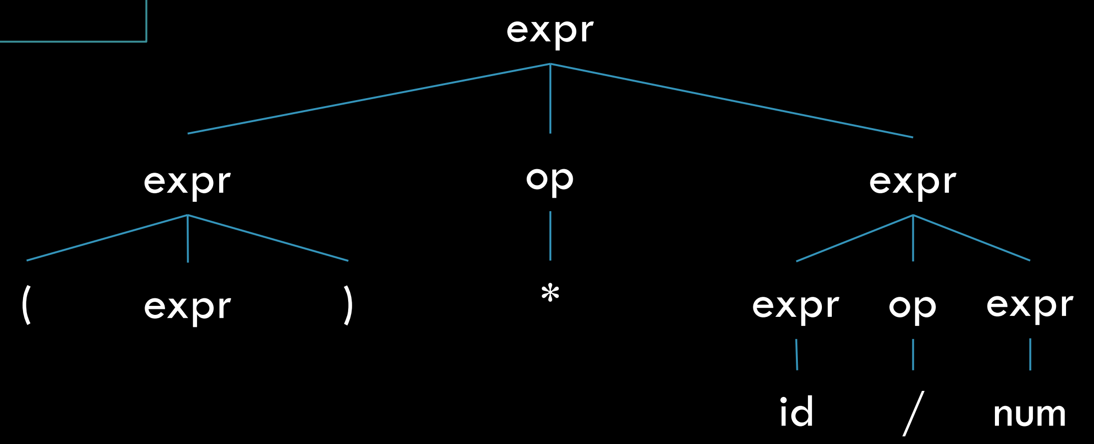
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

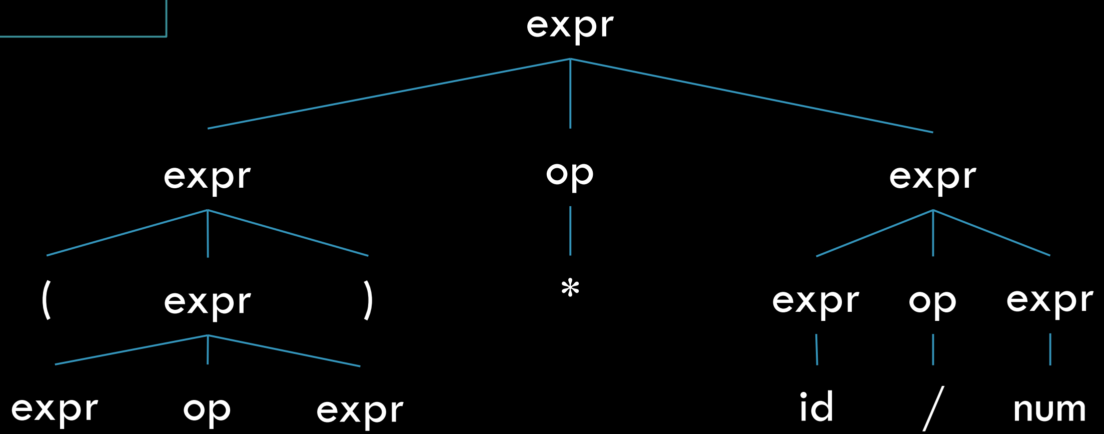
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

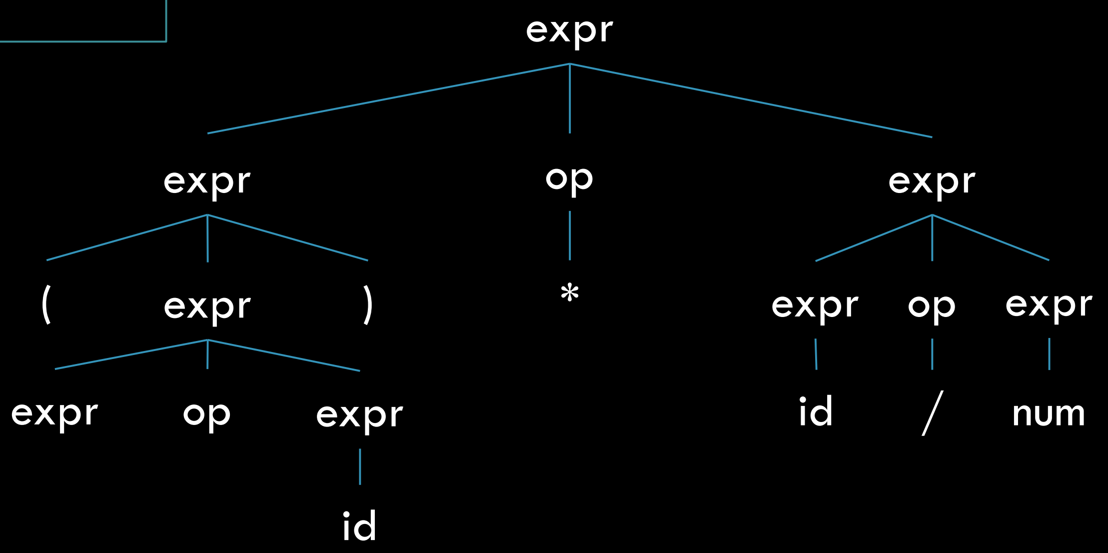
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

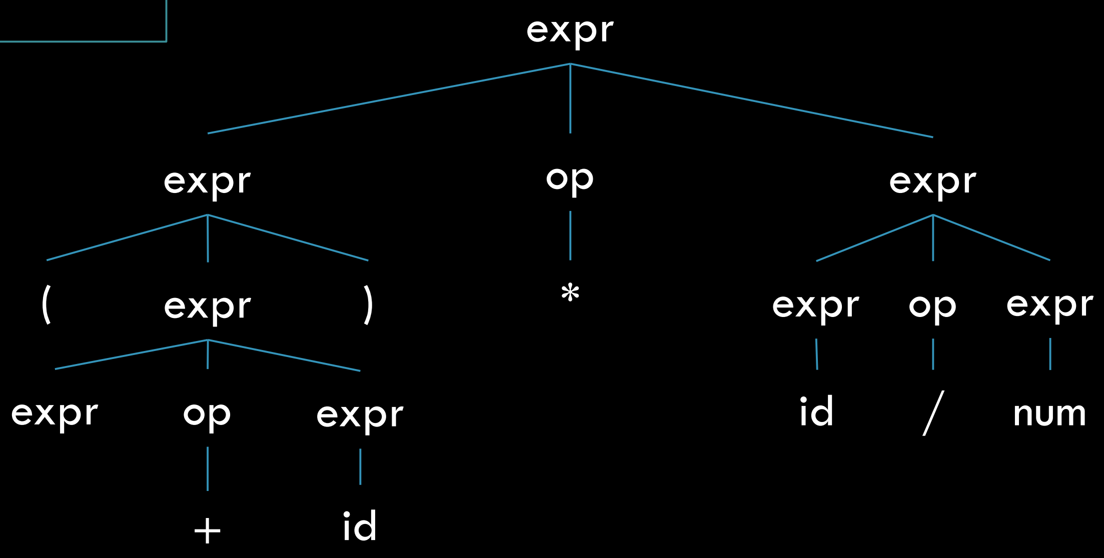
expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

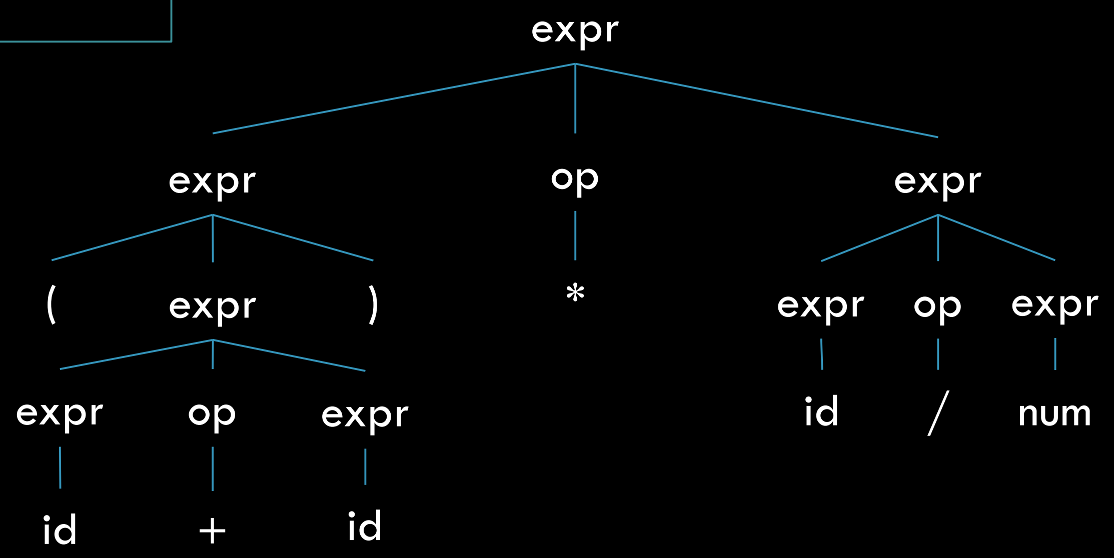
expr → *expr op expr*
→ *expr op expr op expr*
→ *expr op expr op* number
→ *expr op expr* / number
→ *expr op* id / number
→ *expr* * id / number
→ (*expr*) * id / number
→ (*expr op expr*) * id / number
→ (*expr op* id) * id / number
→ (*expr* + id) * id / number
→ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

expr ⟶ *expr op expr*
⟶ *expr op expr op expr*
⟶ *expr op expr op* number
⟶ *expr op expr* / number
⟶ *expr op* id / number
⟶ *expr* * id / number
⟶ (*expr*) * id / number
⟶ (*expr op expr*) * id / number
⟶ (*expr op* id) * id / number
⟶ (*expr* + id) * id / number
⟶ (id + id) * id / number



PARSE TREES FROM DERIVATIONS

Consider the following: “length * width * height”

From our grammar, we can generate two equally acceptable parse trees.

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

AMBIGUOUS DERIVATIONS

Consider the following: “length * width * height”

From our grammar, we can generate two equally acceptable parse trees.

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

Grammars that allow more than one parse tree for the same string are said to be *ambiguous*.
Parsers must, in practice, generate special rules for disambiguation.

AMBIGUOUS DERIVATIONS

Context-free grammars can be structured such that derivations are more efficient for the compiler.

Take the example of arithmetic expressions. In most languages, multiplication and division take precedence over addition and subtraction. Also, associativity tells us that operators group left to right.

We could allow ambiguous derivations and let the compiler sort out the precedence later or we could just build it into the structure of the parse tree.

AMBIGUOUS DERIVATIONS

Previously, we had:

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

Building in associativity and operator precedence:

expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → id | number | - *factor* | (*expr*)
add_op → + | -
mult_op → * | /

AMBIGUOUS DERIVATIONS

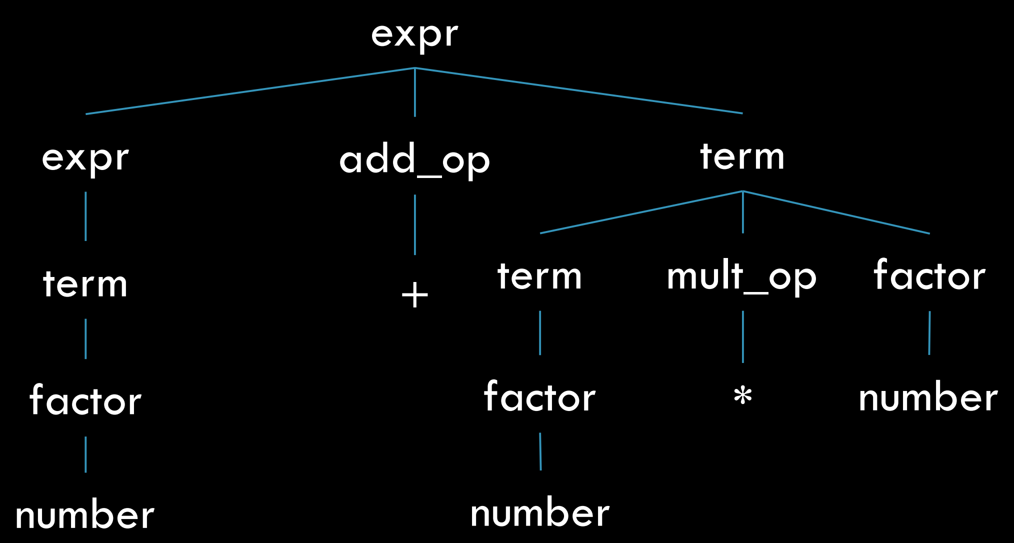
Previously, we had:

expr → id | number | - *expr* | (*expr*) | *expr op expr*
op → + | - | * | /

Building in associativity and operator precedence:

expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → id | number | - *factor* | (*expr*)
add_op → + | -
mult_op → * | /

Example: 3 + 4 * 5



NEXT LECTURE

Scanning
Finite Automata: NFAs and DFAs
Implementing a Scanner