Day 1.

1. Defining a Language

What is a programming language?

- well-defined
- representation of (originally: abstraction for)
- computation (originally: instructions to a thing that computes)

So, let’s build one!

Two aspects of language definition:

- **Syntax**
  - from the Greek συνταξίς (“suntaxis”) coordination
  - most technical questions about syntax—parsing and printing—well-studied
  - see compilers for mechanisms, theory of computation for theoretical aspects
  - not particularly the focus of this course: parsers and printers will generally be provided

- **Semantics**
  - from the Greek σημαντικός (“sēmantikos”) significant
  - many open questions—much of PL theory revolves around questions of defining and approximating program semantics
  - variety of techniques—from the very mathematical (interpreting programs as mathematical functions) to the very empirical (programs mean what the compiler/hardware do)
  - this class—theory of language semantics; compilers—practice of language semantics
  - can we ever really get away from translation?

- Most semantic concerns independent of syntactic concerns in programming languages

2. Arithmetic Expressions (Part 1)

*Model of computation:* grade school arithmetic.

Have to define syntax, even if it’s not the point of the course. Levels of syntax:

- input stream/characters \(( \underline{18 + 5} ) \times 2\)
- lexemes/words \(( \underline{18 + 5} ) \times 2\)
- terms/sentences \(( \underline{18 + 5} ) \times 2\)

Underlining convention: language being defined is underlined, meta-notation written normally. (Broken regularly from now on.)

Our approach: define the terms of a language; leave remaining syntactic concerns implicit.

Terms, intuitively: sums, products, constants. How to make formal?
Mathematical description: Let the set \( \mathcal{E} \) be the smallest set such that

1. For all integers \( z \in \mathbb{Z} \), \( z \in \mathcal{E} \);
2. If \( e_1, e_2 \in \mathcal{E} \), then \( e_1 + e_2 \in \mathcal{E} \); and,
3. If \( e_1, e_2 \in \mathcal{E} \), then \( e_1 \times e_2 \in \mathcal{E} \)

System of inference rules:

\[
\begin{align*}
\frac{z \in \mathbb{Z}}{z \in \mathcal{E}} & \quad \frac{e_1 \in \mathcal{E}}{e_1 + e_2 \in \mathcal{E}} & \frac{e_2 \in \mathcal{E}}{e_1 \times e_2 \in \mathcal{E}}
\end{align*}
\]

BNF (Backus-Naur form) rules:

\[\mathcal{E} \ni e ::= z \mid e_1 + e_2 \mid e_1 \times e_2\]

Key ideas:

- Each defines the same notion
- Each is \textit{compositional}: bigger terms are built out of smaller terms
  - Operations on terms will be defined the same way: recursive functions are the natural consequence of compositional definition
- Still have to disambiguate our \textit{representation} of terms, but parentheses \&c. are in our \textit{meta-}notation, not in terms themselves

Happy surprise: (almost) direct correspondence between mathematical formalism and executable Haskell

```haskell
data Expr = Const Int | Plus Expr Expr | Times Expr Expr
```

Some functions:

```haskell
eval :: Expr -> Int
eval (Const z) = z
eval (Plus e1 e2) = eval e1 + eval e2
eval (Times e1 e2) = eval e1 * eval e2
```

```haskell
pp :: Expr -> String
pp (Const z) = show z
pp (Plus e1 e2) = "(" ++ pp e1 ++ " + " ++ pp e2 ++ ")"
pp (Times e1 e2) = "(" ++ pp e1 ++ " \times " ++ pp e2 ++ ")"
```

Key ideas:

- Pattern matching: always your friend
- Recursion: always your other friend
- Summary: structure of computation parallels structure of data