Day 2

1. Defining a Language

We have: a programming language is a

- 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 "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 "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 $\underline{(}\ \underline{1}\ \underline{8}\ \underline{+}\ \underline{5}\ \underline{)}\ \underline{\times}\ \underline{2}$
- lexemes/words $\underline{(}\ \underline{18}\ \underline{+}\ \underline{5}\ \underline{)}\ \underline{\times}\ \underline{2}$
- terms/sentences $(\ \underline{18}\ \underline{+}\ \underline{5}\ )\ \underline{\times}\ \underline{2}$

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 $\mathcal{T}$ be the *smallest* set such that
  1. For all integers $z \in \mathbb{Z}$, $z \in \mathcal{T}$;
  2. If $t_1, t_2 \in \mathcal{T}$, then $t_1 \underline{+} t_2 \in \mathcal{T}$; and,
  3. If $t_1, t_2 \in \mathcal{T}$, then $t_1 \underline{\times} t_2 \in \mathcal{T}$
- System of *inference rules*:

$$\frac{}{z \in \mathcal{T}}\,(z \in \mathbb{Z}) \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 \underline{+} t_2 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 \underline{\times} t_2 \in \mathcal{T}}$$

- *BNF* (Backus-Naur form) rules:

$$\mathcal{T} \ni t ::= z \mid t_1 \underline{+} t_2 \mid t_1 \underline{\times} t_2$$

Key ideas:

- Each defines the same notion
- Each is *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 *representation* of terms, but parentheses &c. are in our *meta*-notation, not in terms themselves

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

```
data Term = Const Int | Plus Term Term | Times Term Term
```

Some functions:

```
eval :: Term → Int
eval (Const z)     = z
eval (Plus t1 t2)  = eval t1 + eval t2
eval (Times t1 t2) = eval t1 * eval t2

pp :: Term → String
pp (Const z)     = show z
pp (Plus t1 t2)  = "(" ++ pp t1 ++ ") + (" ++ pp t2 ++ ")"
pp (Times t1 t2) = "(" ++ pp t1 ++ ") * (" ++ pp t2 ++ ")"
```

Key ideas:

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