What is the meaning of a program? When we write a program, we represent it using sequences of characters. But these strings are just *concrete syntax*—they do not tell us what the program actually means. It is tempting to define meaning by executing programs—either using an interpreter or a compiler. But interpreters and compilers often have bugs! We could look in a specification manual. But such manuals typically only offer an informal description of language constructs.

A better way to define meaning is to develop a formal, mathematical definition of the semantics of the language. This approach is unambiguous, concise, and, most importantly, it makes it possible to analyze the possible behaviors of programs, even totally bizarre ones, and develop rigorous proofs about properties of interest.

To get from concrete syntax to a formal mathematical definition, we need to recognize that programs are more than just lists of instructions. They are also mathematical objects. A programming language is a logical formalism, just like first-order logic. Such formalisms typically consist of

- *Syntax*: a strict set of rules telling how to distinguish well-formed expressions from arbitrary sequences of symbols; and
- *Semantics*: a way of interpreting the well-formed expressions. The word "semantics" is a synonym for "meaning" or "interpretation." Although ostensibly plural, it customarily takes a singular verb. Semantics may include a notion of deduction or computation, which determines how the system performs work.

We talked in the last lecture about various different kinds of semantics—static vs dynamic and operational vs denotational vs axiomatic. We will cover each and the trade-offs between them later in the course.

1 Arithmetic Expressions

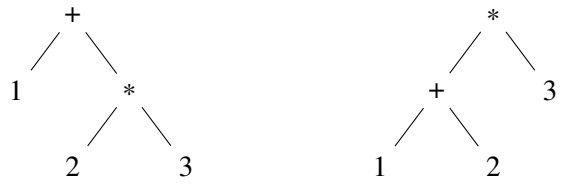
To understand some of the key concepts of semantics, consider a very simple language, ARITH, of integer arithmetic expressions with variable assignment. A program in this language is an expression; executing a program means evaluating the expression to an integer. To describe the syntactic structure of this language we will use variables that range over the following domains:

$$\begin{array}{rcl} \overline{m}, \overline{n} & \in & \textbf{Int} \\ e & \in & \textbf{AExp} \end{array}$$

We put overlines over $\overline{m}$ and $\overline{n}$ to distinguish the syntactic integer symbols in **Int** from the mathematical space of integers $\mathbb{Z}$. **AExp** is the domain of expressions specified using a Backus–Naur Form (BNF) grammar:

$$e \quad ::= \quad n \mid e_1 + e_2 \mid e_1 * e_2$$

This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 * 3$. One can build two abstract syntax trees:



There are several ways to deal with this problem. One is to rewrite the grammar for the same language to make it unambiguous. But that makes the grammar more complex and harder to understand. Another

possibility is to extend the syntax to require parentheses around all addition and multiplication expressions. That also leads to unnecessary clutter and complexity.

Instead we separate the "concrete syntax" of the language (which specifies how to parse a string into program phrases) from the "abstract syntax" (which describes, possibly ambiguously, the structure of program phrases). In this course we will assume that the abstract syntax tree is known. When writing expressions, we will occasionally use parenthesis to indicate the structure of the abstract syntax tree, but the parentheses are not part of the language itself. For instance, we may wish to write "$(1 + 2) * 3$" to indicate the second abstract syntax tree above.

2 Semantics

We will use the basic structures of relations and functions discussed in Lecture 1 to define the semantics of a programming language.

Denotational Semantics. Consider the language of arithmetic expressions in Section 1. We could define a function, lets call it eval, that takes an expression and produces the number it evaluates to. So $\mathsf{eval}(3+(4*2)) = 11$. Here eval is a form of *denotational semantics* for our language. Do to so, we would define the function recursively on **AExp** terms as follows

$$\mathsf{eval}(e) = \begin{cases} n & \text{when } e = \overline{n} \\ \mathsf{eval}(e_1) + \mathsf{eval}(e_2) & = \text{when } e = e_1 + e_2 \\ \mathsf{eval}(e_1) * \mathsf{eval}(e_2) & = \text{when } e = e_1 * e_2. \end{cases}$$

Operational Semantics. Alternatively, we could define a relation that defines how an expression evaluates. This will define an *operational semantics*. For instance, $4 * 2$ could "step to" 8, which we would write as

$$4 * 2 \longrightarrow 8.$$

Here $\longrightarrow \subseteq \mathbf{AExp} \times \mathbf{AExp}$ is a binary relation. If the semantics are entirely deterministic—as they hopefully are here—it may be a (partial) function, but that is not required.

This suggestion raises a couple of questions. First, we need to handle sub-expressions stepping. We would probably want to allow $3 + (4 * 2) \longrightarrow 3 + 8$, for example. Second, what about the term $(1 + 2) * (3 + 4)$. Which sub-expression should we step first? While with integer arithmetic it won't matter, in some languages it does, so we need to define precisely the order of evaluation for sub-expressions.

We define the small step operational semantics compactly using a set *inference rules*. The following rules define left-to-right evaluation.

$$\frac{n_1 + n_2 = m}{\overline{n_1} + \overline{n_2} \longrightarrow \overline{m}} \qquad \frac{n_1 * n_2 = m}{\overline{n_1} * \overline{n_2} \longrightarrow \overline{m}}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \qquad \frac{e_1 \longrightarrow e_1'}{e_1 * e_2 \longrightarrow e_1' * e_2} \qquad \frac{e \longrightarrow e'}{\overline{n} + e \longrightarrow \overline{n} + e'} \qquad \frac{e \longrightarrow e'}{\overline{n} * e \longrightarrow \overline{n} * e'}$$

The meaning of an inference rule is that if the facts above the line holds, then the fact below the line holds. The fact above the line are called *premises*; the fact below the line is called the *conclusion*. The rules without premises are *axioms*; and the rules with premises are *inductive* rules.

Axiomatic Semantics. We may also want to specify certain properties about a program. For instance, we may want to prove that $e_1 + e_2$ is even whenever both e_1 and e_2 are even (or when both are odd), and that $e_1 * e_2$ is even whenever either e_1 or e_2 is even. We could do that by stating properties about the even-ness of

programs and how they combine. This idea leads us to *axiomatic semantics* and will define the program as a relation between things that are true before and after executing it.

We will see how to do all of the above later in the semester. But first, we need to understand the basic math of how we can formally build up and manipulate those definitions. We will primarily use *induction*, which we will talk about in the next couple of lectures.

3 Inductive Definitions

The definitions of both **AExp** and $\longrightarrow$ above appear to reference themselves. Is that allowed? The short answer is "yes," but we need to unpack exactly what this inductive definition means.

An *inductively-defined* set A is one that is described using a finite collection of axioms and inductive (inference) rules. Axioms of the form

$$\frac{}{a \in A}$$

indicate that a is in the set A. Inductive rules

$$\frac{a_1 \in A \qquad \cdots \qquad a_n \in A}{a \in A}$$

indicate that if $a_1, \ldots, a_n$ are *all* elements of A, then a is also an element of A.

Rather than providing an immediate definition, these inference rules lay out a set of requirements that A must satisfy. Loosely speaking, we can prove that some value a is in the set A if we can do so using any *finite* number of applications of these rules. The finite requirement means we must "bottom out" at an axiom (rather than an inductive rule) eventually. Then A is the set of elements that we can prove are in A in this manner.

Here are two examples of inductive sets.

Example 1. The BNF grammar for **AExp** above is essentially shorthand for the following inference rules.

$$\frac{}{n \in \textbf{AExp}} \qquad \frac{e_1 \in \textbf{AExp} \qquad e_2 \in \textbf{AExp}}{e_1 + e_2 \in \textbf{AExp}} \qquad \frac{e_1 \in \textbf{AExp} \qquad e_2 \in \textbf{AExp}}{e_1 * e_2 \in \textbf{AExp}}$$

Example 2. The natural numbers $\mathbb{N}$ can be defined as an inductive set from the rules

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{succ(n) \in \mathbb{N}}$$

4 Formalizing Inductive Sets

The description above provides a guide for how inductive sets work, but how can we formalize the definition of **AExp** (or any other inductive set) from that intuition? To do so, we use the inference rules to build up the inductive set one step at a time.

4.1 The Rule Operator

The approach we will take is to build a *rule operator* R on subsets of S, the space of possible elements of A. We what to define R such that $R(B)$ is the set members of A that can be inferred directly from the members of B using only a single rule. That is,

$$R(B) \triangleq \left\{ x \in S \;\middle|\; x_1, \ldots, x_n \in B \text{ and } \frac{x_1 \quad \cdots \quad x_n}{x} \text{ is a rule instance} \right\}$$

With this definition, $R(\varnothing)$ is the set of members that can be inferred from nothing. $R(R(\varnothing))$ is the set of members that can be inferred from $R(\varnothing)$. Note that $R(\varnothing) \subseteq R(R(\varnothing))$ because they are inferred from $\varnothing$, which is a subset of $R(\varnothing)$. Indeed, for any n, we can show that $R^n(\varnothing) \subseteq R^{n+1}(\varnothing)$.

More generally, we can show that R is *monotone*, meaning that

$$B \subseteq C \implies R(B) \subseteq R(C).$$

This is because, if $x \in R(B)$, that means there is some rule instance $\frac{x_1 \quad \cdots \quad x_n}{x}$ where $\{x_i\}_{i=1}^n \subseteq B$. Since $B \subseteq C$, the premises are also in C, so by the definition of R, $x \in R(C)$.

But when is $A \subseteq S$ the precise set defined by a collection of inductive rules? At the very least, we would like A to satisfy the following two properties.

1. *A is R-consistent:* $A \subseteq R(A)$. That is, every element of A should be there as a result of applying a rule to prove its inclusion.
2. *A is R-closed:* $R(A) \subseteq A$. That is, everything the rules say should be in A should actually be there.

These two properties together mean that $A = R(A)$. That is, A must be a *fixed point* of the function R.

There are two natural questions to ask: Does R have a fixed point? Is that fixed point unique, and if not, which should we use?

4.2 Least Fixed Points

In fact, every monotone set operator R always has at least one fixed point—a property shown by the Knaster–Tarski theorem, which we will not cover. There may, however, be many fixed points of the same monotone function R. Among its fixed points, there must be a unique *least fixed point* that is minimal with respect to set inclusion $\subseteq$. That is, the least fixed point is one that is a subset of every other fixed point. We take A to be this least fixed point.

There are two different ways to construct the least fixed point, "from below" and "from above":

$$A_* \triangleq \bigcup_{n=0}^{\infty} R^n(\varnothing)$$

$$A^* \triangleq \bigcap \{B \subseteq S \mid R(B) \subseteq B\}$$

The set A_* is the union of all sets of the form $R^n(\varnothing)$. That is, it is the set of elements one can prove is in A using any finite number of applications of any inductive rule instances.

The set A^* is the intersection of all all R-closed subsets of S. It consists of all elements of S that are in every R-closed set.

It turns out that these two sets are equal, so we define $A \triangleq A_* = A^*$. The proof, which is a special case of the Knaster–Tarski theorem, is relatively straightforward.

Theorem 1. *Using the definitions above, $A_* = A^*$.*

Proof. First we show that $A^* \subseteq A_*$. For this, it is sufficient to show that A_* is R-closed. Unfolding the definition, we get

$$R(A_*) = R\left(\bigcup_{n=0}^{\infty} R^n(\varnothing)\right) = \bigcup_{n=1}^{\infty} R^n(\varnothing).$$

Since $R^0(\varnothing) = \varnothing$, we get $A_* = \varnothing \cup R(A) = R(A)$.

Now we show that $A_* \subseteq A^*$. To prove this, we show that if B is R-closed, then $R^n(\varnothing) \subseteq B$ for all n. We prove this by induction on n using the monotonicity of R.

For $n = 0$, $R^0(\varnothing) = \varnothing \subseteq B$. If $R^n(\varnothing) \subseteq B$, then

$$\begin{aligned} R^{n+1}(\varnothing) &= R(R^n(\varnothing)) \\ &\subseteq R(B) && \text{(by monotonicity of } R\text{)} \\ &\subseteq B && \text{(by } R\text{-closure of } B\text{).} \end{aligned}$$

Since $R^n(\varnothing) \subseteq B$ for all n,

$$A_* = \bigcup_{n=0}^{\infty} R^n(\varnothing) \subseteq B.$$

Since this holds for all R-closed sets B, we have that $A_* \subseteq A^*$. □