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 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:

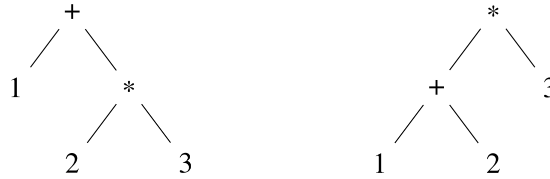
$$\begin{aligned} x, y, z &\in \mathbf{Var} \\ m, n &\in \mathbf{Int} \\ e &\in \mathbf{Exp} \end{aligned}$$

Var is a set of program variables (e.g., *foo* or *i*), **Int** is the set of integers, and **Exp** is the domain of expressions specified using a Backus–Naur Form (BNF) grammar:

$$\begin{aligned} e \quad ::= \quad & x \\ \mid \quad & n \\ \mid \quad & e_1 + e_2 \\ \mid \quad & e_1 * e_2 \\ \mid \quad & x := e_1 \ ; e_2 \end{aligned}$$

Informally, $x := e_1 \ ; e_2$ means that the program should evaluate e_1, assign the value to x, then evaluate e_2 and return the result.

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 Binary Relations and Functions

To define languages precisely, we will make heavy use of some foundational mathematical objects: binary relations and functions.

2.1 Binary Relations

A binary relation R is some way of connecting elements of two (not-necessarily-distinct) domains A and B. The Cartesian product (or cross product) $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. A binary relation on $A \times B$ is simply a subset $R \subseteq A \times B$. We sometimes say "a is related to b" when $(a, b) \in R$, sometimes denoted $a\ R\ b$.

For a few examples, the smallest binary relation is the empty relation $\varnothing$ consisting of no pairs, and the largest binary relation on $A \times B$ is $A \times B$ itself. The *identity relation* on A is $\{(a, a) \mid a \in A\} \subseteq A \times A$, which relates every element a to itself, and nothing else.

And important operation on binary relations is *relational composition*

$$R \ ; S = \{(a, c) \mid \exists b.\ (a, b) \in R \text{ and } (b, c) \in S\}$$

2.2 Functions

A (*total*) *function* (or *map*) is a binary relation $f \subseteq A \times B$ in which every element of A is associated with exactly one element of B. If f is such a function, we write

$$f : A \to B.$$

In other words, $f : A \to B$ is a binary relation $f \subseteq A \times B$ such that, for every element $a \in A$, there is exactly one pair $(a, b) \in f$ with first component a. Note that they may be any number of pairs (including zero) with any given b.

The set A is called the *domain* of f, while B is the *codomain* (or *range*). The *image* of f is the set of elements in B that come from at least one element of A under f.

$$\begin{aligned} f(A) &\triangleq \{b \in B \mid f(a) = b \text{ for some } a \in A\} \\ &= \{f(a) \mid a \in A\} \end{aligned}$$

The notation $f(A)$ is standard, albeit somewhat of an abuse.

Function composition is the operator that means we should apply one function and then aother. If $f : A \to B$ and $g : B \to C$ are functions, then $g \circ f : A \to C$ is the function

$$(g \circ f)(x) \triangleq g(f(x)).$$

Viewing functions as a special case of binary relations, functional composition is the same as relational composition, but the order is reversed in the notation: $g \circ f = f \,;\, g$.

A *partial function* $f : A \rightharpoonup B$ (note the shape of the arrow) is a function $f : A' \to B$ defined on some subset $A' \subseteq A$. The notation $\mathrm{dom}(f)$ refers to A', the domain of f. If $f : A \to B$ is total, then $\mathrm{dom}(f) = A$.

A function $f : A \to B$ is said to be *injective* (or *one-to-one*) if $a \neq b$ implies $f(a) \neq f(b)$. That is, for any $b \in B$, there is *at most* one $a \in A$ that f maps b. The function is *surjective* (or *onto*) if for every $b \in B$, there is some $a \in A$ such that $f(a) = b$—that is, $f(A) = B$. That is, for any $b \in B$, there is *at least* one $a \in A$ that f maps b. A function that is both injective and surjective—there is exactly one $a \in A$ that maps to each $b \in B$—is said to be *bijective*.

Representations of Functions. Mathematically, a function is equal to its *extension*, which is the set of all its (input, output) pairs. One way to describe a function is to describe its extension directly, usually by specifying some mathematical relationship between the inputs and outputs. This is called an *extensional* representation. Another way is to give an *intensional*[1] representation, which is essentially a program or evaluation procedure to compute the output corresponding to a given input. The main differences are

- there can be more than one intensional representation of the same function, but there is only one extension;
- intensional representations typically give a method for computing the output from a given input, whereas extensional representations need not concern themselves with computation (and seldom do).

A central issue in semantics—and a good part of this course—is concerned with how to go from an intensional representation to a corresponding extensional representation.

3 Semantics

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

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 $\mathsf{eval}(3 + (4 * 2)) = 11$ and $\mathsf{eval}(i := 6 + 1 \,;\, 2 * 3 * i) = 42$. Here eval is a form of *denotational semantics* for our language. But this still leaves some questions unanswered. For instance, what is $\mathsf{eval}(x + 2)$? Does it matter what came before it? How about what comes after it?

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

$$4 * 2 \longrightarrow 8.$$

[1]Note the spelling; *intensional* and *intentional* are not the same!

Here $\longrightarrow \subseteq \textbf{Exp} \times \textbf{Exp}$ 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. Variables are also a concern. We could easily say

$$(i = 6 + 1 \; ; \; 2 * 3 * i) \longrightarrow (i = 7 \; ; \; 2 * 3 * i),$$

but then what does this step to? There are a few options, including substituting a value in for i in the rest of the expression and changing $\longrightarrow$ to relate not only expressions, but pairs of expressions and stores that track the values of variables.

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.