1 Introduction

What is a program? Is it just something that tells the computer what to do? Yes, but there is much more to it than that. The basic expressions in a program must be interpreted somehow, and a program's behavior depends on how they are interpreted. We must have a good understanding of this interpretation, otherwise it would be impossible to write programs that do what is intended.

It may seem like a straightforward task to specify what a program is supposed to do when it executes. After all, basic instructions are pretty simple. But in fact this task is often quite subtle and difficult. Programming language features often interact in ways that are unexpected or hard to predict. Ideally it would seem desirable to be able to determine the meaning of a program completely by the program text, but that is not always true, as you well know if you have ever tried to port a C program from one platform to another. Even for languages that are nominally platform-independent, meaning is not necessarily determined by program text. For example, consider the following Java fragment.

```
class A { static int a = B.b + 1; }
class B { static int b = A.a + 1; }
```

First of all, is this even legal Java? Yes, although no sane programmer would ever write it. So what happens when the classes are initialized? A reasonable educated guess might be that the program goes into an infinite loop trying to initialize `A.a` and `B.b` from each other. But no, the initialization terminates with initial values for `A.a` and `B.b`. So what are the initial values? Try it and find out, you may be surprised. Can you explain what is going on? This simple bit of pathology illustrates the difficulties that can arise in describing the meaning of programs. Luckily, these are the exception, not the rule.

Programs describe computation, but they are more than just lists of instructions. They are mathematical objects as well. 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.

In this course we will see some of the formal tools that have been developed for describing these notions precisely. The course consists of three major components:

- *Dynamic semantics*—methods for describing and reasoning about what happens when a program runs.
- *Static semantics*—methods for reasoning about programs *before* they run. Such methods include type checking, type inference, and static analysis. We would like to find errors in programs as early as possible. By doing so, we can often detect errors that would otherwise show up only at runtime, perhaps after significant damage has already been done.
- *Language features*—applying the tools of dynamic and static semantics to study actual language features of interest, including some that you may not have encountered previously.

We want to be as precise as possible about these notions. Ideally, the more precisely we can describe the semantics of a programming language, the better equipped we will be to understand its power and limitations and to predict its behavior. Such understanding is essential not only for writing correct programs, but also for building tools like compilers, optimizers, and interpreters. Understanding the meaning of programs allows us to ascertain whether these tools are implemented correctly. But the very notion of *correctness* is subject to semantic interpretation.

It should be clear that the task before us is inherently mathematical. Initially, we will characterize the semantics of a program as a function that produces an output value based on some input value. Thus, we will start by presenting some mathematical tools for constructing and reasoning about functions.

1.1 Binary Relations and Functions

Denote by $A \times B$ the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$. A *binary relation* on $A \times B$ is just a subset $R \subseteq A \times B$. The sets A and B can be the same, but they do not have to be. The set A is called the *domain* and B the *codomain* (or *range*) of R. The smallest binary relation on $A \times B$ is the null 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 $\{(a, a) \mid a \in A\} \subseteq A \times A$.

An important operation on binary relations is *relational composition*

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

where the codomain of R is the same as the domain of S.

A (*total*) *function* (or *map*) is a binary relation $f \subseteq A \times B$ in which each 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, a function $f : A \to B$ is a binary relation $f \subseteq A \times B$ such that for each element $a \in A$, there is exactly one pair $(a, b) \in f$ with first component a. There can be more than one element of A associated with the same element of B, and not all elements of B need be associated with an element of A.

The set A is the *domain* and B is the *codomain* or *range* of f. The *image* of f is the set of elements in B that come from at least one element in A under f:

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

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

The operation of *functional composition* is: if $f : A \to B$ and $g : B \to C$, then $g \circ f : A \to C$ is the function

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

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 $\mathsf{dom}\, f$ refers to A', the domain of f. If $f : A \to B$ is total, then $\mathsf{dom}\, f = A$.

A function $f : A \to B$ is said to be *one-to-one* (or *injective*) if $a \neq b$ implies $f(a) \neq f(b)$ and *onto* (or *surjective*) if every $b \in B$ is $f(a)$ for some $a \in A$.

1.2 Representation 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.

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