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, for the most part, 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, with properties and behavior that we can attempt to described formally. For most mathematical structures we encounter, familiar mathematical tools like sets and sequences are adequate to describe the properties of the structure. In this course we will see some of the formal tools that have been developed for describing precisely what programs are and what they do.

This course is mostly about the *semantics* of programs and programming languages. "Semantics" is a synonym for "meaning" or "interpretation". We want to be very precise about this notion, because it is necessary for understanding the behavior of programs. It 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.

There are three major components to this course.

- Dynamic semantics. We will study methods for describing and reasoning about what happens when a program runs.
- Static semantics. We will also study 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. We will apply methods for dynamic and static semantics to study actual language features of interest, including some interesting features that many students may have not seen before.

At the start, we will mostly be characterizing the semantics of a program as a function that produces an output value(s) based on some input value(s). More generally, real programs are reactive and interact with their inputs arriving from the environment. Describing reactive programs is more challenging, although we can view the reactive behavior of the program again as a function of the inputs it receives from the environment as it runs. Thus, to describe program semantics, we will build up 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 $\emptyset$ consisting of no pairs, and the largest binary relation on $A \times B$ is $A \times B$ itself. The *identity relation* on A is $ID = \{(a, a) \mid a \in A\} \subseteq A \times A$.

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. There can be more than one element of A associated with the same 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.

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:

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

This is also sometimes denoted $f(A)$, although this is an abuse of notation.

A *partial function* $f : A \rightharpoonup B$ 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.

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

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

This course is quite a bit about how to get from an intensional representation to a corresponding extensional representation.

2 The Lambda Calculus

The lambda calculus (or λ-calculus[2], λ=Greek "L") was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s to describe functions in an unambiguous and compact manner. The lambda calculus provides *intensional* representations of functions.

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

[2] Why λ? Church wanted to separate the bound variables from the unbound (free) variables and he used a caret on top of the bound variables. $f(x) = x + yx^2$ was represented as $\hat{x}.x + yx^2$. Apparently, the printers could not handle the caret and it moved to the front and became a λ; the expression became $\lambda x.\, x + yx^2$

Real programming languages such as Lisp, Scheme, Haskell and ML are based on the lambda calculus, although there are differences as well. Lisp was the first of these.

It is common to use lambda notation in conjunction with other operators and values in some domain, such as $\lambda x.\, x + 2$, but the *pure* λ-calculus has only λ-terms and only the operators of functional abstraction and functional application, nothing else. In the pure λ-calculus, λ-terms act as functions that take other λ-terms as input and produce λ-terms as output. Nevertheless, it is possible to code common data structures such as booleans, integers, lists, and trees as λ-terms. The λ-calculus is computationally powerful enough to represent and compute any computable function. It is thus equivalent to Turing machines in computational power.

2.1 Syntax

The following is the syntax of the pure λ-calculus. A λ-term is defined inductively as follows. There is a countable set of *variables* **Var**.

1. Any variable $x \in \mathbf{Var}$ is a λ-term.
2. If e is a λ-term, then so is $\lambda x.\, e$ (functional abstraction).
3. If e_1 and e_2 are λ-terms, then so is $e_1 \cdot e_2$ (functional application).

We often write $e_1\, e_2$ or $e_1(e_2)$ for $e_1 \cdot e_2$. Intuitively, this term represents the result of applying of e_1 as a function to e_2 as its input. The term $\lambda x.\, e$ represents a function with input parameter x and body e.

We can abbreviate the above definition in BNF (Backus–Naur form) as

$$e \;::=\; x \;\mid\; e_1\; e_2 \;\mid\; \lambda x.\, e.$$

In mathematics it is common to define a function f by writing down what its value on a typical input would be. For example, we might define the squaring function on integers by writing $f(x) = x^2$. This is the same as writing $f = \lambda x.\, x^2$. In mathematics, one often writes $x \mapsto x^2$ to denote the same function *anonymously*, that is, without giving it a separate name; this is the same as writing $\lambda x.\, x^2$.

Here are some examples of λ-terms. The identity function is $ID = \lambda x.\, x$. A function that ignores its argument and return the identity function is $\lambda x.\, \lambda a.\, a$. This is the same as $\lambda x.\, ID$.

Parentheses are used to show explicitly how to parse expressions, but we also assign a precedence to the operators in order to save parentheses. The application operator $\cdot$ binds tighter than λ-abstraction. Another way to view this is that the body of a λ-abstraction extends as far to the right as it can. For example, $\lambda x.\, x\ \lambda y.\, y$ should be parsed as $\lambda x.\, (x\ \lambda y.\, y)$, not $(\lambda x.\, x)\ (\lambda y.\, y)$. If you want the latter, you have to use explicit parentheses. Application is *left-associative*, which means that $e_1\, e_2\, e_3$ should be parsed as $(e_1\, e_2)\, e_3$.

It's never a bad idea to include parentheses if you aren't sure.

2.2 Scope, Bound and Free Occurrences, Closed Terms

The *scope* of the abstraction operator λx shown in the term $\lambda x.\, e$ is the body e. An occurrence of a variable y in a term is said to be *bound* in that term if it occurs in the scope of an abstraction operator λy; otherwise, it is *free*. A bound occurrence of y is *bound to* the abstraction operator λy with the smallest scope in which it occurs. Note that a variable can have bound and free occurrences in the same term, and can have bound occurrences that are bound to different abstraction operators.

A term is *closed* if all variables are bound.

In the term

$$\lambda x.\, (x\ (\lambda y.\, y\ a)\ x)\ (\lambda x.\, x\ y),$$

all three occurrences of x are bound. The first two are bound to the first λx, and the last is bound to the second λx. The first occurrence of y is bound, the a is free, and the last y is free, since it is not in the scope of any λy.

2.3 Higher-Order Functions

In lambda calculus, we can define functions that can take functions as arguments and return functions as results. Thus functions are *first-class values*. For example, the term $\lambda f.\, f\ 5$ represents a function that takes another function f as an argument and applies it to 5. The term $\lambda v.\, \lambda f.\, f\ v$ represents a function that takes an argument v and returns a function that calls its argument on v. A function that takes a pair of functions and returns their composition is represented by $\lambda f.\, \lambda g.\, \lambda x.\, g(fx)$.

In fact, every λ-term represents a function, since any λ-term can appear on the left-hand side of an application operator.

2.4 Multi-Argument Functions and Currying

We would like to allow multiple arguments to a function, as for example in $(\lambda x, y.\, x + y)(5, 2)$. However we can consider this an abbreviation for $(\lambda x.\, \lambda y.\, x + y)\ 5\ 2$. That is, instead of the function taking two arguments and adding them, the function takes only the first argument and returns a function that takes the second argument and then adds the two arguments. In general, $\lambda x_1 \ldots x_n.\, e$ is considered an abbreviation for $\lambda x_1.\, \lambda x_2.\, \lambda x_3.\, \ldots \lambda x_n.\, e$. Thus we consider the multi-argument version of the λ-calculus as just syntactic sugar. The "desugaring" transformation

$$\begin{array}{rcl} \lambda x_1 \ldots x_n.\, e & \Rightarrow & \lambda x_1.\, \lambda x_2.\, \lambda x_n.\ e \\ e_0\ (e_1, \ldots, e_n) & \Rightarrow & e_0\ e_1\ e_2\ \cdots\ e_n \end{array}$$

for this particular form of sugar is called *currying* (after Haskell B. Curry).

3 Preview

Next time we will discuss capture-avoiding (safe) substitution the computational rules of the λ-calculus, namely α-, β-, and η-reduction. This is the *calculus* part of the λ-calculus.