If we return to the original goal of formal semantics, we can see that neither operational semantics nor axiomatic semantics answer the question “what is the meaning of this piece of code?” head on. Operational semantics answers it in this way:

Semantics can be expressed as a set of logical inference rules that explain what happens when a particular code segment is executed. Expression evaluation is a relation between the syntactic form, a store (modeled as a function) and the set of values that result. The semantics of a command are expressed as evaluations of the immediate components of the command and their combined affect on the store. Essentially a program means a logical derivation that unfolds the execution step by step.

On the other hand, axiomatic semantics answers the question thus:

Semantics can be expressed by relating properties of the store before an after the code segment is executed. Axioms express these relationships by choosing the weakest pre-condition on the store (and the strongest post-condition). Essentially the meaning of a program is its specification (constraints on initial and final stores), broken down into intermediate constraints on stores after each command.

However, the usual understanding of meaning is that an object can be understood in terms of another object that can stand in for the original object, and yet is somehow simpler, or more transparent. A typical way to explain this is the meaning triangle, due to Ogden and Richards in a book called “The Meaning of Meaning”, written in 1923. In this triangle, a human reasoner is the mediator between a thing and the symbol or symbols we use to represent and talk about that thing. A picture can help:

In the world of language, we might use the word “dog” to symbolize a dog, but what is in our head is neither the actual dog, nor merely the word; it is the concept of a dog, i.e. what it means to be a dog (four legs, fur, wags tail etc.) In the world of programming languages, we can draw the picture like this:

The model of computation is the way we understand a program, but that is different from either the source code or the program itself. The source code and model of computation remain fixed in our three methods; it is just the realization of the program that differs. But what is a program? It certainly is not like a dog, which is a physical object,
although thinking of a program as a collection of silicon chips and electron flows is theoretically possibly. Actually that is pretty much how a computer engineer views a program. So if it is not to be thought of as a physical object, we are left with abstract objects – hence formal semantics. For operational semantics, the abstract object is a derivation involving inference rules and changes to a store function. For axiomatic semantics it is a proof using axioms and sometimes difficult gluing inferences (the weakening and strengthening rules).

Neither of these is really an object in the normal sense, i.e. a static entity which sits there waiting to be observed. Another way to think of this, is to think of the meaning of a dog in abstract terms. We might say that a dog is the set of chemical and physical transformations of a fertilized egg; this is somewhat like the operational approach. Or we might say that a dog is just the sum of its behavioral properties, and these properties can be broken down into properties of its parts. So a dog is, in highly simplified form, a walking, barking, eating, sleeping object. This is rather like the axiomatic approach; it is descriptive of a dog’s behavior and this description can be broken down until there are parts which cannot be described, merely accepted for what they are (like axioms). So barking is the forced movement of air past vocal chords, and so on.

Can we not simply place the source code of a program into correspondence with an abstract object, and let the abstract object stand in for the source code? This is the essence of the denotational approach. We will choose abstract, mathematical objects from these correspondences. The abstraction will be sets and functions, and we will let the source code denote these abstractions, and they will be static objects that we can explore and analyze as such.

THE BIG PICTURE

The denotational approach is actually in four parts. These are:

1. Abstract syntax. As usual syntax comes first.
2. Semantic domains. Define the set or sets that will be used to represent the values that the language will handle.
3. Semantic algebras. Add useful operations to the domains so that the sets can be handled in sensible ways. Adding operations to a domain is very like creating an abstract data type.
4. Create valuation functions. This is the heart of the method. Each valuation function will map a piece of syntax (the right-hand alternatives from the abstract syntax definition) to a semantic domain. Most often these will be functions, which are, in any case, just sets of pairs.

We will be looking at a variety of languages, including the two basic ones we looked at in operational and axiomatic semantics, but heading towards languages with types, sub-programs and different systems of scope.