UNIVERSITY OF LIVERPOOL

Introduction to Semantics

This module is concerned with the *meaning* (semantics) of programs. Obviously, anyone who can write a program in some programming language has at least an *informal* understanding of what that program means. There are, however, some problems raised by such informal semantics:

- How can we be sure that what the programmer thinks the program does is what it actually does?
- How can we prove (or give a rigorous argument that) the program does what it's meant to do?

The second point is the problem of **program verification**, and is important for safety-critical software (e.g., air-traffic control software, missile-guidance systems, system-control software in nuclear power plants, etc.). To give an idea of the importance of program verification in such cases, in the 1990's the US Department of Defense introduced a policy that all software developed for the Department should be formally verified against its specifications.

Formal semantics provides a way of reasoning rigorously about programs: one way of describing the meaning of a program is as a mathematical object; its mathematical properties can then be used to show that it meets its specification, or that it is equivalent to another program (such equivalences are useful in program transformation, where a programmer begins with a program that is obviously correct but perhaps very inefficient, and transforms this program into an equivalent, more efficient program). Alternatively, a more direct route to proving the correctness of a program is to describe its semantics in terms of the *logical* properties it satisfies, and from those properties infer that it meets its specification. In this module, we'll look at both mathematical and logical approaches to defining the semantics of programs.

There are three main kinds of semantics:

Operational semantics
: describes the meaning of a program in terms of how it is executed, for example by specifying the individual steps in its computation, or by describing an abstract machine that evaluates the program.

Denotational semantics
: describes the meaning of a program as a mathematical object, typically as a function that takes a state of a computer (its initial state before the program is run) and returns an updated state (the state of the computer that results from executing the program.)

Axiomatic semantics
: describes the logical properties of programs; typically, an axiomatic semantics says that if some property holds before a program is run, then some other property will hold after the program has run. For example, the semantics of a program to compute the minimum of two numbers x and y might state that if x is less than y then (on completion of the program) the

value stored in a variable min is *x*.

In this module, we will look at both denotational and axiomatic semantics.

More generally, we can speak of the semantics not just of programs, but of a programming language. By this we mean that each syntactic construct in the language can be given a formal semantics (i.e., there would be a formal semantics for assignments, conditionals, loops, etc.). The semantics of a program written in the language is then derived from the semantics of its component parts (i.e., its assignments, loops, and so on). When the semantics of all programs are described in terms of the semantics of their components, we call this **compositional semantics**. The issue of program verification is one of correctness; when we have a semantics for a programming language, one form of the correctness issue is whether a language is correctly implemented (i.e., can we verify the correctness of the compiler or interpreter that implements the language on a particular platform). A satisfactory answer can really only be given when the language has been given a formal semantics.

A related issue is that of programming language design, and it is here that formal semantics can play an important role. A formal semantics for a language can help separate the various elements that play a role in the language design. A key example is given by the programming language Algol, whose design was recorded in the Algol Report, a document that was groundbreaking in that it described both the syntax and the semantics of the language, and also clearly separated the syntax of the language from its semantics. One of the great contributions of the Algol Report was the introduction of Backus-Naur Form (named after John Backus and Peter Naur, two of the authors of the Algol Report) to give a formal description of the syntax of the language.

We will use Backus-Naur Form (BNF) to describe the syntax of a simple programming language whose semantics we consider in later lectures. Before doing so, we list the following examples of the uses of formal semantics:

- Some programming languages (such as Ada) support **parallel assignment**, where two or more variables can be assigned to simultaneously. For example, in Ada,

  ```
  x,y := 0,1
  ```

  has the effect of assigning 0 to x and 1 to y. Moreover, these assignments are done all at once: 0 is assigned to x at the same time as 1 is assigned to y. For example,

  ```
  x,y := y,x
  ```

  has the effect of swapping the values of x and y, which is clearly different from

  ```
  x := y ;
  y := x
  ```

  which simply sets both x and y to the initial value of y. It is fairly clear what parallel assignment does, and even from this brief description, you probably have a good, though informal, understanding of the concept. Informal understanding is typically based upon examples, and the examples given tend to be straightforward. It is often possible, however, to think up examples that test this informal understanding to the limit; for example, what are we to make of

  ```
  x,x := 0,1
  ```

Clearly, something is wrong here, and the program cannot set x to both 0 and 1. Maybe the program should assign either 0 or 1 to x, although there's no good reason to prefer one possibility or the other. Probably the best solution is to say that this is syntactically incorrect, and have the compiler flag this as a syntax error, but in the absence of formal syntactic and semantic descriptions of the language, this choice is rather arbitrary.

- In the late 1970s, Joseph Goguen and Rod Burstall developed the algebraic specification language Clear. This was a language intended to allow one to describe formally the desired properties of a program or software system (its specification). Clear had an elegant, clean semantics, and in fact the language was designed with its semantics in mind (rather than having the semantics developed after the language was designed). Clear had many features supporting hierarchical composition of specifications (i.e., constructs that allowed specifications to be built from other component specifications). Many of these features were used later by Bernd Krieg-Brueckner in the design of the programming language Ada's "generic packages" (and probably these ideas resurfaced in the 1990's in the development of the C Standard Template Libraries).

- The programming language Java was designed to be a platform-independent language suitable for writing programs that could be downloaded across the Internet and run in a browser (as Applets). Of course, no language can be truly platform-independent, as programs have to run on some physical machine, but Java has a high degree of platform independence that is largely achieved through the use of teh Java Virtual Machine. Java source code is compiled, but the target language of the compilation process is not the machine code of some particular chip-set, but the machine code (so-called "byte-code") of an abstract machine (a stack automaton). This process of compiling to an abstract machine effectively provides Java with an operational semantics. Moreover, this semantics is useful in proving properties of the language such as the enforcement of Java's security model.

Syntax and Backus-Naur Form

See this tutorial on bnf by Lars Marius Garshol.

Syntax and Semantics of Binary Numbers

As you know, numbers can be represented as binary numbers (in "base 2") using strings of 0s and 1s. For an example that develops the syntax and semantics of binary numbers, see the hand-out (Chapter 1 of Tennent's book).

The Syntax of a Simple Imperative Language

We end by specifying the syntax of a simple imperative language that we will use in later lectures to illustrate axiomatic and denotational semantics.

There are five syntactic classes:

- <**ZZ**> - numbers in standard decimal notation, for example 4, 58, etc. (see the first exercise below).
- <**Var**> - variable (or identifier) names; see the second exercise below).
- <**Exp**> - arithmetic expressions, built up from numbers and variables using operations for addition, subtraction and multiplication.
- <**Tst**> - Boolean expressions; these include equality tests on two arithmetic expressions and testing whether the value of one arithmetic expression is less than or equal to another. There are also the logical "and", "or" and "not" operations.
- <**Pgm**> - programs ("commands"), including assignment, sequential composition, conditionals and while-loops.

These syntactic classes are defined by the following BNF rules:

<**Exp**> ::= <**ZZ**> | <**Var**> | <**Exp**> + <**Exp**>
| <**Exp**> - <**Exp**> | <**Exp**> * <**Exp**>

<**Tst**> ::= true | false
| <**Exp**> = <**Exp**> | <**Exp**> <= <**Exp**>
| not <**Tst**>
| <**Tst**> and <**Tst**> | <**Tst**> or <**Tst**>

<**Pgm**> ::= skip | <**Var**> := <**Exp**> | <**Pgm**> ; <**Pgm**>
| if <**Tst**> then <**Pgm**> else <**Pgm**> fi
| while <**Tst**> do <**Pgm**> od

The program skip is a program that does nothing: it simply terminates immediately. It is useful, for example, in defining an "if-then" construct using the "if-then-else-fi" conditional:

if *B* then *C* = if *B* then *C* else skip fi

Exercises

1. Give a BNF definition of decimal numbers (i.e., numbers in "base 10").

2. An **alphanumeric character** is either a digit (0,1,...,9) or a letter (a,b,...,z). In many programming languages, variable names can be any string of alphanumeric characters that doesn't begin with a digit. Give a BNF description of a syntactic class <**Var**> of such variable names.

3. Give a BNF definition of Java method signatures; i.e., the method declarations that can be given in an interface, for example:

```
public void myMethod(int i);

String concatenate(String s1, String s2);
```

   You may assume that all parameters either have type int or String, and that return-types are either int, String or void (why would it be difficult to allow any type or class name to be used?).

4. In the language defined by the BNF clauses above, write a program that sets a variable max to the larger of the values stored in variables x and y (i.e., computes the maximum of x and y).

5. In the same language, write a program that sets a variable sum to the sum of numbers $0+1+2+...+n$, where n is the number stored in the variable n.

Review Questions

1. What is the difference between syntax and semantics? (It's hard to be precise about this, but it's worthwhile thinking about it, as it's fundamental to this course.)

2. What, briefly, are the differences between operational, axiomatic, and denotational semantics?

3. What are some of the uses of formal semantics?

Grant Malcolm

Last modified: Fri May 11 14:58:48 BST 2007