Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Intro to semantics

Lecture 1 Tuesday, January 24, 2022

1 Intro to semantics

What is the meaning of a program? When we write a program, we use a sequence of characters to represent the program. But this *syntax* is just how we represent the program: it is not what the program means.

Maybe we could define the meaning of a program to be whatever happens when we execute the program (perhaps using an interpreter, or by compiling it first). But we can have bugs in interpreters and compilers! That is, an interpreter or compiler may not accurately reflect the meaning of a program. So we must look elsewhere for a definition of what a program means.

One place to look for the meaning of a program is in the language specification manual. Such manuals typically give an informal description of the language constructs.

Another option is to give a formal, mathematical definition of the language semantics. A formal mathematical definition can have the following advantages over an informal description.

- **Less ambiguous.** The behavior of the language is clearer, which is useful for anyone who needs to write programs in the language, implement a compiler or interpreter for the language, add a new feature to the language, etc.

- **More concise.** Mathematical concepts and notation can clearly and concise describe a language, and state restrictions on legal programs in the language. For example, the Java Language Specification (2rd edition) devotes a chapter (26 pages) to describing the concept of *definite assignment*, most of which is describing, in English, a dataflow analysis that can be expressed more succinctly using mathematics.

- **Formal arguments.** Most importantly, a formal semantics allows us to state, and prove, program properties that we're interested in. For example: we can state and prove that *all* programs in a language are guaranteed to be free of certain run-time errors, or free of certain security violations; we can state the specification of a program and prove that the program meets the specification (i.e., that the program is guaranteed to produce the correct output for all possible inputs).

However, the drawback of formal semantics is that they can lead to fairly complex mathematical models, especially if one attempts to describe all details in a full-featured modern language. Few real programming languages have a formal semantics, since modeling all the details of a real-world language is hard: real languages are complex, with many features. In order to describe these features, both the mathematics and notation for modeling can get very dense, making the formal semantics difficult to understand. Indeed, sometimes novel mathematics and notation needs to be developed to accurately model language features. So while there can be many benefits to having a formal semantics for a programming language, they do not yet outweigh the costs of developing and using a formal semantics for real programming languages.

There are three main approaches to formally specify the semantics of programming languages:

- operational semantics: describes how a program would execute on an abstract machine;

- denotational semantics: models programs as mathematical functions;

- axiomatic semantics: defines program behavior in terms of the logical formulae that are satisfied before and after a program;

Each of these approaches has different advantages and disadvantages in terms of how mathematically sophisticated they are, how easy it is to use them in proofs, or how easy it is to implement an interpreter or compiler based on them.

2 A simple language of arithmetic expressions

To understand some of the key concepts of semantics, we will start with a very simple language of integer arithmetic expressions, with assignment. A program in this language is an expression; executing a program means evaluating the expression to an integer.

To describe the structure of this language we will use the following domains:

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

Var is the set of program variables (e.g., foo, bar, baz, i, etc.). **Int** is the set of constant integers (e.g., 42, −40, 7). **Exp** is the domain of expressions, which we specify using a BNF (Backus-Naur Form) grammar:

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2$$

Informally, the expression $x := e_1; e_2$ means that x is assigned the value of e_1 before evaluating e_2. The result of the entire expression is that of e_2.

This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 \times 3$. One can build two abstract syntax trees:

```
    +                                      *
   / \                                    / \
  1   *                                  +   3
     / \                                / \
    2   3                              1   2
```

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 expressions:

$$x \mid n \mid (e_1 + e_2) \mid (e_1 \times e_2) \mid x := e_1; e_2$$

However, this also leads to unnecessary clutter and complexity.

Instead, we separate the "concrete syntax" of the language (which specifies how to unambiguously parse a string into program phrases) from the "abstract syntax" of the language (which describes, possibly ambiguously, the structure of program phrases). In this course we will use the abstract syntax and 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 details on parsing, grammars, and ambiguity elimination, see or take the compiler course Computer Science 153.)