PART I

BACKGROUND

1

A question of semantics

The goal of this chapter is to give the reader a glimpse of the applications and problem areas that have motivated and to this day continue to inspire research in the important area of computer science known as programming language semantics.

1.1 Semantics is the study of meaning

Programming language semantics is the study of *mathematical models of and methods for describing and reasoning about the behaviour of programs.*

The word *semantics* has Greek roots[1] and was first used in linguistics. Here, one distinguishes among *syntax*, the study of the structure of languages, *semantics*, the study of meaning, and *pragmatics*, the study of the use of language.

In computer science we make a similar distinction between syntax and semantics. The languages that we are interested in are *programming languages* in a very general sense. The 'meaning' of a program is its behaviour, and for this reason programming language semantics is the part of programming language theory devoted to the study of *program behaviour*.

Programming language semantics is concerned only with purely internal aspects of program behaviour, namely what happens within a running program. Program semantics does not claim to be able to address other aspects of program behaviour – e.g. whether or not a program is user-friendly or useful.

In this book, when we speak of semantics, we think of *formal semantics*,

[1] The Greek word (transliterated) is *semantikós*, meaning 'significant'. The English word 'semantics' is a singular form, as are 'physics', 'mathematics' and other words that have similar Greek roots.

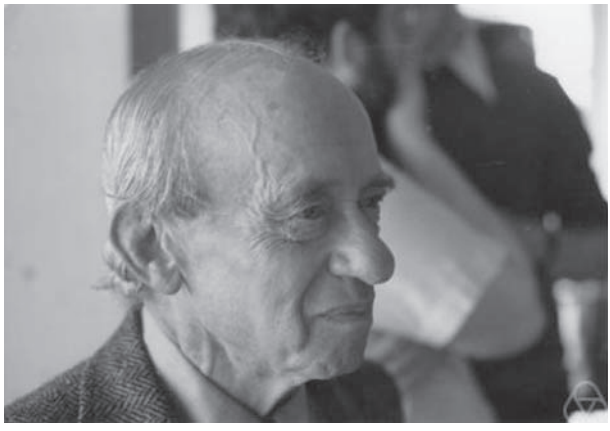


Figure 1.1 Alfred Tarski

understood as an approach to semantics that relies on precise mathematical definitions.

Formal semantics arose in the early twentieth century in the context of mathematical logic. An early goal of mathematical logic was to provide a precise mathematical description of the language of mathematics, including the notion of truth. An important contributor in this area was the logician Alfred Tarski (Figure 1.1) (Tarski, 1935).

Many of the first insights and a lot of the fundamental terminology used in programming language semantics can be traced back to the work of Tarski. For instance, the important notion of *compositionality* – that the meaning of a composite language term should be defined using the meanings of its immediate constituents – is due to him. So is the insight that we need to use another language, a *metalanguage*, to define the semantics of our target language.

1.2 Examples from the history of programming languages

The area of programming language semantics came into existence in the late 1960s. It was born of the many problems that programming language designers and implementors encountered when trying to describe various constructs in both new and existing programming language.

The general conclusion that emerged was that an informal semantics, however precise it may seem, is not sufficient when it comes to defining the behaviour of programs.

1.2.1 ALGOL 60

The programming language ALGOL 60 was first documented in a paper from 1960 (Backus and Naur, 1960), now often referred to simply as 'the ALGOL 60 report'. ALGOL 60 was in many ways a landmark in the evolution of programming languages.

Firstly, the language was the result of very careful work by a committee of prominent researchers, including John Backus, who was the creator of FORTRAN, John McCarthy, the creator of Lisp, and Peter Naur, who became the first Danish professor of computer science. Later in their careers, Backus, McCarthy and Naur all received the ACM Turing Award for their work on programming languages.

Secondly, ALGOL 60 inspired a great many subsequent languages, among them Pascal and Modula.

Thirdly, ALGOL 60 was the first programming language whose syntax was defined formally. The notation used was a variant of context-free grammars, later known as Backus–Naur Normal Form (BNF).

However, as far as the semantics of ALGOL 60 is concerned, Backus and his colleages had to rely on very detailed descriptions in English, since there were as yet no general mathematical theories of program behaviour. It turned out to be the case that even a group of outstanding researchers (who for the most part were mathematicians) could not avoid being imprecise, when they did not have access to a formalized mathematical theory of program behaviour. In 1963 the ALGOL 60 committee therefore released a revised version of the ALGOL 60 report (Backus and Naur, 1963) in which they tried to resolve the ambiguities and correct the mistakes that had been found since the publication of the original ALGOL 60 report.

However, this was by no means the end of the story. In 1967, Donald E. Knuth published a paper (Knuth, 1967) in which he pointed out a number of problems that still existed in ALGOL 60.

One such problem had to do with global variables in procedures. Figure 1.2 illustrates the nature of the problem. The procedure `awkward` is a procedure returning an integer value.

```
integer procedure awkward
begin comment x is a global variable
 x := x+1
 awkward := 3
end  awkward
```

Figure 1.2 An ALGOL 60 procedure. What is its intended behaviour?

The procedure `awkward` manipulates the value of a global variable and therefore has a side effect. However, the ALGOL 60 report does not explain whether or not side effects are allowed in procedures. Because of this, there is also no explanation of how arithmetic expressions should be evaluated if they contain side effects.

Let us consider a global variable `x` whose value is 5 and assume that we now want to find the value of the expression `x+awkward`. Should we evaluate `x` before or after we evaluate `awkward`? If we evaluate `x` first, the value of the expression will be 8; should we evaluate `x` after having called `awkward`, we get the value 9!

One consequence of Knuth's paper was that the ALGOL 60 committee went back to the drawing board to remove the ambiguities. The main reason why it took so long to discover these problems was that the language designers had no precise, mathematical criterion for checking whether or not all aspects of the language had been defined.

1.2.2 Pascal

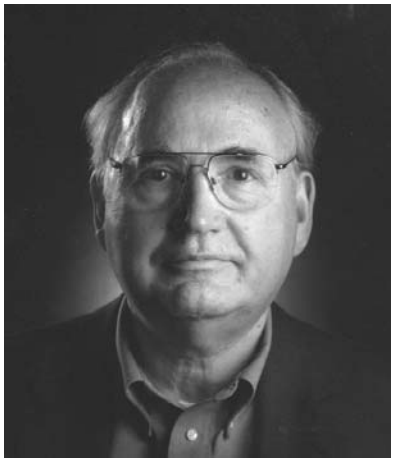
Pascal, a descendant of the Algol family, was created by Niklaus Wirth and first documented in a book written with Kathleen Jensen (Jensen and Wirth, 1975). Ever since then, Pascal has been a common introductory language in computer science degree programmes around the world.

Even though great care was taken in the exposition of the language features, Pascal also suffers from the problems associated with an informal semantics. In particular, there are problems with explaining scoping rules – in fact, the scoping rules are barely explained in the book. There is mention of global variables; however, nowhere in the text is it explained what a global variable is, let alone what its scope should be. Nor are there any rules that specify that a variable must be declared before it is used!

All existing implementations of Pascal assume this (except for pointer variables), but the declaration-before-use convention is not part of the original definition of the language.

1.3 Different approaches to program semantics

The development of a mathematical theory of program semantics has been motivated by examples such as the ones given above. There are several ways of providing such a mathematical theory, and they turn out to be related.



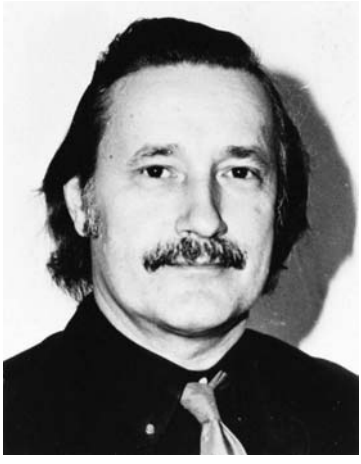


Figure 1.3 Dana Scott (left) and Christopher Strachey (right)

Denotational semantics was the first mathematical account of program behaviour; it arose in the late 1960s (Strachey, 1966, 1967; Scott and Strachey, 1971) and was pioneered by Dana Scott and Christopher Strachey (Figure 1.3), who at the time were both working at Oxford University.

In denotational semantics, the behaviour of a program is described by defining a function that assigns meaning to every construct in the language. The meaning of a language construct is called its *denotation*. Typically, for an imperative program, the denotation will be a *state transformation*, which is a function that describes how the final values of the variables in a program are found from their initial values.

Structural operational semantics – the main topic of this book – came into existence around 1980 and is due to Gordon Plotkin (Figure 1.4), who gave the first account of his ideas in a set of lecture notes written during his sabbatical at Århus University in 1980 (Plotkin, 1981). An important early contribution is that of Robin Milner (Figure 1.5), who used Plotkin's approach to give a labelled semantics to the process calculus CCS (Calculus of Communication Systems) (Milner, 1980). Plotkin (2004) gives a detailed account of the origins and early history of the area.

In structural operational semantics one specifies the behaviour of a program by defining a transition system whose transition relation describes the evaluation steps of a program. One of the underlying motivations for this approach was that it is possible to give a simple account of concurrent programs; previous attempts to give a semantic description of even simple

Figure 1.4 Gordon Plotkin

parallel programming languages had used denotational semantics and had turned out to be quite complicated.

A central insight of this approach, and one to which we shall return repeatedly throughout this book, is that one can describe the evaluation steps of a syntactic entity (such as a program) in a structural fashion, that is, by means of an inductive definition based on the abstract syntax.

Axiomatic semantics is due to Tony Hoare (Hoare, 1969; Apt, 1981) (Figure 1.6) and, like denotational semantics, it is a product of the late 1960s. Here one describes a language construct by means of mathematical logic. More precisely, one defines a set of rules that describe the assertions that must hold before and after the language construct has been executed.

Algebraic semantics is related to denotational semantics and describes the behaviour of a program using universal algebra (Guessarian, 1981; Goguen and Malcolm, 1996). The members of the research collective behind the OBJ specification language, with Joseph Goguen (Figure 1.7) as a prominent contributor, have been important figures in the development of this approach.

These four approaches to programming language semantics are not rivals. Rather, they complement each other. Some approaches are more suitable than others in certain situations. For instance, it is much easier to describe

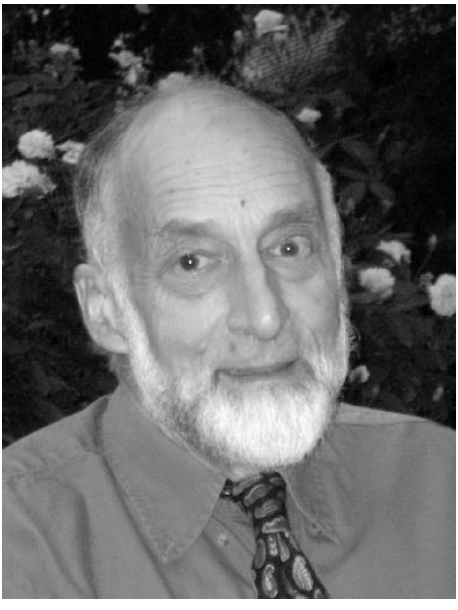


Figure 1.5 Robin Milner

Figure 1.6 Tony Hoare

parallel and nondeterministic program behaviour using structural operational semantics than by means of denotational semantics.

There are many precise mathematical results relating the four approaches. In this book we give an example of such a result in Chapter 15, where we

Figure 1.7 Joseph Goguen

show that the structural operational semantics and the denotational semantics of the **Bims** language are equivalent in a very precise sense.

1.4 Applications of program semantics

The area of program semantics has turned out to be extremely useful in situations where it is important to give a precise description of the behaviour of a program. Here are some prominent examples.

1.4.1 Standards for implementation

The formal semantics of a programming language is not meant as an alternative to the informal descriptions of programming constructs found in introductory programming textbooks. A formal semantics serves a very different purpose, namely to act as a yardstick that any implementation must conform to.

The examples mentioned in Section 1.2 all helped make computer scientists aware of the fact that *only a precise semantic definition can provide an exhaustive and implementation-independent account of all aspects of a programming language*. Such an account is particularly necessary if one is a 'superuser' of the language whose task is to implement an interpreter or a compiler or, in general, to create a language-dependent programming

environment. A prominent example of a formal semantic definition is the operational semantics of Standard ML (Milner *et al.*, 1997), due to Robin Milner, Robert Harper, David MacQueen and Mads Tofte. Later, a lot of effort went into providing a suitable formal semantics of Java and C#; there are now denotational as well as operational semantics of Java (Alves-Foss, 1999) and C# (Börger *et al.*, 2005).

1.4.2 Generating interpreters and compilers

A precise definition of the semantics of a programming language will specify how a program in the language is to be executed. As a consequence, it is a natural step to construct a compiler/interpreter generator which, when given a definition of the semantics of some language L, will generate a compiler (or interpreter) for L.

Such compiler/interpreter generators have existed for many years. The first such systems were based on denotational semantics (Mosses, 1976; Paulson, 1982); later systems have also used variants of structural operational semantics (Pettersson, 1999; Diehl, 2000; Chalub and Braga, 2007). In general, the idea is not to replace standard compiler implementations as such but to provide a tool for the language developer.

In Appendix B of this book we give a number of small examples that describe how one can create an interpreter directly from a structural operational semantics.

1.4.3 Verification and debugging – lessons learned

Many software systems today are safety-critical in the sense that an execution error may have very unpleasant and wide-ranging consequences. One would of course like to be able to predict such events to prevent them from ever occurring.

In the natural sciences the use of mathematical models allows scientists and engineers to predict many events with great precision. Engineers use the mathematically based theories from physics to design bridges in such a way that these do not collapse and meteorologists use mathematical models of the atmosphere to make weather forecasts.

Similarly, we would like to use mathematically based theories to reason in a precise manner about the behaviour of programs. Programming language semantics makes this possible.

The following examples demonstrate what can happen if safety-critical software is not subjected to analyses of this kind.