23 Cambridge Computer Science Texts

# A practical introduction to denotational semantics

Lloyd Allison
*Department of Computer Science, University of Western Australia*

CAMBRIDGE
UNIVERSITY PRESS

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211 USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia



First published 1986
Reprinted 1995

*British Library cataloguing in publication data*

Allison, Lloyd
A practical introduction to denotational semantics. – (Cambridge computer science text)

1. Programming languages (Electronic computers) – Syntax 2. Programming languages (Electronic computers) – Semantics
I. Title
005.13 QA76.7

*Library of Congress cataloguing in publication data*

Allison, Lloyd.
A practical introduction to denotational semantics.
(Cambridge computer science texts; 23)
Bibliography
Includes indexes.
1. Programming languages (Electronic computers) – Semantics. I. Title. II. Series.
QA76.7.A45 1987 005.13 86-12961

ISBN 0 521 30689 2 hard covers
ISBN 0 521 31423 2 paperback

Transferred to digital printing 2003

MP

# Contents

# 1
# Introduction

Denotational semantics is a formal method for defining the semantics of programming languages. It is of interest to the language designer, compiler writer and programmer. These individuals have different criteria for judging such a method – it should be concise, unambiguous, open to mathematical analysis, mechanically checkable, executable and readable depending on your point of view. Denotational semantics cannot be all things to all people but it is one attempt to satisfy these various aims. It is a formal method because it is based on well-understood mathematical foundations and uses a rigorously defined notation or meta-language.

The complete definition of a programming language is divided into *syntax*, *semantics* and sometimes also *pragmatics*. Syntax defines the structure of legal sentences in the language. Semantics gives the meaning of these sentences. Pragmatics covers the use of an implementation of a language and will not be mentioned further.

In the case of syntax, context-free grammars expressed in Backus–Naur form (*BNF*) or in syntax diagrams have been of great benefit to computer scientists since Backus and Naur [44] formally specified the syntax of Algol-60. Now all programming languages have their syntax given in this way. The result has been 'cleaner' syntax, improved parsing methods, parser-generators and better language manuals. As yet no semantic formalism has achieved such popularity and the semantics of a new language is almost invariably given in natural language.

The typical problem facing a programmer is to write a program which will transform data satisfying some properties or assertions 'P' into results satisfying 'Q'.

$$\{P\}\ \text{program}\ \{Q\}$$

The language of the assertions is predicate logic. This formulation treats a program as a predicate transformer.

Concentrating on the predicates in the transformation leads to the *axiomatic* style of semantics. This was suggested by Floyd [17] and

formalized and developed by Hoare [24], Dijkstra [13] and many others. The method is readable and very useful to programmers and designers of algorithms. It is intimately connected with the discipline of structured programming. It has, not insurmountable, difficulties in defining some features of programming languages, notably **goto**s and side-effects in expressions. There is heated debate, not to be taken up here, as to whether this is a drawback of the method or an indication that these features are hard to use and dangerous. Note that interest in predicate logic has created the programming language Prolog [9] (Ch 9).

Concentrating on the program as a function mapping inputs satisfying P into results satisfying Q leads to *operational* and denotational semantics. Operational semantics imagines the program running on an abstract machine. The machine may be quite unlike any real computer, either low-level, simple and easy to analyse, or high-level with an easy translation from the programming language. The machine and the translation must be specified. Such a definition is most useful to a compiler writer if the abstract machine is close to the real hardware. To be useful mathematically it may require quite different properties.

Denotational semantics recognizes the subtle distinction between a function as a probably infinite set of ordered pairs $\{\langle \text{input}_i, \text{output}_i \rangle\}$ and an algorithm as a finite description of the function. A program is the algorithm written in some particular programming language. A program stands for, or denotes, a function. A denotational semantics of a programming language gives the mapping from programs in the language to the functions *denoted*.

***Example***

$\text{factorial} = \{\langle 0,1\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,6\rangle, \langle 4,24\rangle, \ldots\}$

$\text{fact(n)} \quad = \text{if n} = 0 \text{ then } 1 \text{ else n} \times \text{fact(n} - 1)$

A good semantics should confirm that program fact denotes the factorial function.

Denotational semantics is written in *λ-notation* which is the λ-calculus of Church [7] with data-types. It has a well-developed mathematical theory and the foundations have been thoroughly investigated. The method is concise and powerful enough to describe the features of current programming languages. Mosses [40] gives a definition of Algol-60 and Tennent [63] gives one for Pascal, for example. Such definitions are only readable with practice, the notation being equivalent to a powerful but

terse programming language. It is more suitable for the language designer and implementor than the programmer.

McCarthy [34] based the programming language Lisp on the $\lambda$-calculus, and other languages, particularly in the Algol family, show a similar influence. Lisp was perhaps the first programming language designed on mathematical semantic principles. Much of the motivation for the wider application of denotational semantics to all programming languages came from Strachey [59, 60]. Scott [54, 55, 56] solved many of the mathematical problems raised concerning the existence and consistency of objects defined in a semantics. One benefit of denotational semantics is that it can be 'mechanized'. Mosses' SIS [41] is an interpreter for denotational semantics that enables a definition to be run – used to execute programs in the defined language. Paulson's compiler-compiler [49] translates a definition into a compiler. Certain functional programming languages such as ML [20] are very close to the typed $\lambda$-notation and can be used to write, and run, denotational definitions.

It seems that an axiomatic definition and a denotational definition make good partners; the former for the programmer, the latter for the language designer. The theme of Donahue's book [14] is that such definitions can be shown to be consistent with each other and he does this for a large subset of Pascal.

Note that denotational semantics is part of a wide movement including model theory in logic, philosophy and linguistics. For example, Montague semantics [15] attempts to give a denotational style of semantics for a subset of English. Logicians are concerned with the objects that names, variables and predicates stand for and with what it means for a statement to be true.

This introduction has omitted much, in particular extensions of grammars to include semantics. *Attribute grammars* associate attribute evaluation functions with grammar rules. These can be used operationally to evaluate to code, or denotationally to evaluate to functions. *Two-level grammars* [65] are also powerful enough to specify the input–output behaviour of programs [8].

In the remainder of the book, after some motivating examples, there is an introduction to the $\lambda$-notation, data-types and the mathematical foundations of denotational semantics. This is followed by applications to the definition of programming-language features. The notation of even denotational semantics is somewhat arbitrary (!) and throughout it is shown how definitions can be programmed in conventional languages and executed.

## 1.1 An example

To give the flavour of denotational semantics, the classic example of decimal numerals follows. Decimal numerals form a language, **Num**, over the *alphabet* $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. It can be defined by the *grammar*

$$v ::= v\delta \mid \delta$$
$$\delta ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

The symbol '::=' can be read as 'is' or 'can be replaced by'. The '|' can be read as 'or'. A digit $\delta$ is a 0 or a 1 or a 2 and so on. A numeral $v$ is a numeral followed by a digit or it is a single digit. The Greek letters $\delta$ and $v$ are syntactic variables over parts of the language **Num**.

The decimal numerals are usually taken to stand for, or taken to *denote*, integers which are abstract objects. This conventional interpretation can be made formal by giving a valuation function **V**:

$$\mathbf{V}: \mathbf{Num} \to \mathbf{Int}$$
$$\mathbf{V}[\![v\delta]\!] = 10 \times \mathbf{V}[\![v]\!] + \mathbf{V}[\![\delta]\!]$$
$$\mathbf{V}[\![0]\!] = 0 \quad \mathbf{V}[\![1]\!] = 1$$
$$\mathbf{V}[\![2]\!] = 2 \quad \mathbf{V}[\![3]\!] = 3$$
$$\mathbf{V}[\![4]\!] = 4 \quad \mathbf{V}[\![5]\!] = 5$$
$$\mathbf{V}[\![6]\!] = 6 \quad \mathbf{V}[\![7]\!] = 7$$
$$\mathbf{V}[\![8]\!] = 8 \quad \mathbf{V}[\![9]\!] = 9$$

**V** is a function from the sentences in the language **Num** to the integers **Int**. **V** is defined on a case-by-case analysis of the alternatives in the grammar for **Num**. Elements of the language are enclosed in the special brackets $[\![$ and $]\!]$ to distinguish them from the meta-language outside. Inside the brackets are strings. The integers outside the brackets are in *italics*. 7 is a character which denotes the integer 7. It is impossible to write anything down without using names and so we are forced to adopt some such convention.

The value of a particular numeral can now be calculated:

$$\begin{aligned}\mathbf{V}[\![123]\!] &= 10 \times \mathbf{V}[\![12]\!] + 3 \\ &= 10 \times (10 \times \mathbf{V}[\![1]\!] + 2) + 3 \\ &= 123\end{aligned}$$

This may prompt the reaction 'so what, isn't that obvious?' The reply is that we should be pleased that the formal definition agrees with intuition in simple cases. This is a feature of good theories. The formalism is needed when intuition is not strong enough. The reader who does not realize that the statement 7 = '7' is not only not true, but is an error in Pascal, may have missed the point. Note that **V** has captured the essence of positional notation, and that $\mathbf{V}[\![123]\!] = \mathbf{V}[\![0123]\!]$ and so on.

The definition of **V** can be used to design a piece of code that exists in most compilers:

```
if ch in ['0'..'9'] then
begin n := ord(ch) - ord('0');
      ch := nextch {return next char and advance input↑};
      while ch in ['0'..'9'] do
      begin n := n*10 + ord(ch) - ord('0');
            ch := nextch
      end
end
```

Numerals that occur in a program must have their values calculated by the compiler.

In what follows the languages examined will be more interesting and the things they are mapped onto will be more complex. The general scheme, however, is always to map a language onto a collection of abstract objects.

## 1.2 Exercises

1. If the first case in the definition of **V** is changed to

   $\mathbf{V}[\![v\delta]\!] = -10 \times \mathbf{V}[\![v]\!] + \mathbf{V}[\![\delta]\!]$

   what are the new characteristics of **V**? What advantage does this new interpretation of **Num** have?

2. Give a grammar and a semantic valuation function for roman numerals made up of I (one), V (five) and X (ten) only. It will be simpler to have a grammar which allows unusual but comprehensible numerals such as

   IIII five
   IIIIV zero
   VV ten

   although

   IVX

   should not be allowed as it is ambiguous – four or six.

   To include the full range of roman numerals L (fifty), C (hundred), D (five hundred), M (thousand), follows a similar pattern and just makes the solution larger.