Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

For other volumes:
http://www.springer.com/series/7592

Gilles Dowek • Jean-Jacques Lévy

Introduction to the Theory of Programming Languages

Springer

Gilles Dowek
Labo. d'Informatique
École polytechnique
route de Saclay
91128 Palaiseau
France
gilles.dowek@polytechnique.edu

Jean-Jacques Lévy
Centre de Recherche Commun
INRIA-Microsoft Research
Parc Orsay Université
28 rue Jean Rostand
91893 Orsay Cedex
France
jean-jacques.levy@inria.fr

The work was first published in 2006 by Les editions de l'École polytechnique with the following title: 'Introduction à la théorie des langages de programmation'. The translator of the work is Maribel Fernandez.

ISSN 1863-7310
ISBN 978-0-85729-075-5 e-ISBN 978-0-85729-076-2
DOI 10.1007/978-0-85729-076-2
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

What Is the Theory of Programming Languages?

The ultimate, definitive programming language has not been created yet, far from it. Almost every day a new language is created, and new functionalities are added to existing languages. Improvements in programming languages contribute to making programs more reliable, shorten the development time, and make programs easier to maintain. Improvements are also needed to satisfy new requirements, such as the development of parallel, distributed or mobile programs.

The first thing that we need to describe, when defining a programming language, is its *syntax*. Should we write `x := 1` or `x = 1`? Should we put brackets after an `if` or not? More generally, what are the strings of symbols that can be used as a program? There is a useful tool for this: the notion of a *formal grammar*. Using a grammar, we can describe the syntax of the language in a precise way, and this makes it possible to build programs to check the syntactical correctness of programs.

But it is not sufficient to know what a syntactically correct program is in order to know what is going to happen when we run the program. When defining a programming language, it is also necessary to describe its *semantics*, that is, the expected behaviour of the program when it is executed. Two languages may have the same syntax but different semantics.

The following is an example of what is meant (informally) by semantics. Function evaluation is often explained as follows. "*The result* `V` *of the evaluation of an expression of the form* `f` e_1 `...` e_n, *where the symbol* `f` *is a function defined by the expression* `f` x_1 `...` x_n `=` `e'`, *is obtained in the following way. First, the arguments* e_1`,` `...,` e_n *are evaluated, returning values* W_1`,` `...,` W_n. *Then, these values are associated to the variables* x_1`,` `...,` x_n, *and finally the expression* `e'` *is evaluated. The value* `V` *is the result of this evaluation*."

This explanation of the semantics of the language, expressed in a natural language (English), allows us to understand what happens when a program is executed, but is it precise? Consider, for example, the program

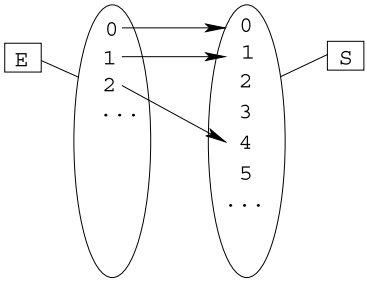
```
f x y = x
g z = (n = n + z; n)
n = 0; print(f (g 2) (g 7))
```

Depending on the way we interpret the explanation given above, we can deduce that the program will result in the value `2` or in the value `9`. This is because the natural language explanation does not indicate whether we have to evaluate `g 2` before or after `g 7`, and the order in which we evaluate these expressions is important in this case. Instead, the explanation should have said: "the arguments $e_1, \ldots, e_n$ are evaluated *starting from* e_1" or else "*starting from* e_n".

If two different programmers read an ambiguous explanation, they might understand different things. Even worse, the designers of the compilers for the language might choose different conventions. Then the same program will give different results depending on the compiler used.

It is well known that natural languages are too imprecise to express the syntax of a programming language, a formal language should be used instead. Similarly, natural languages are too imprecise to express the semantics of a programming language, and we need to use a formal language for this.

What is the semantics of a program? Let us take for instance a program `p` that requests an integer, computes its square, and displays the result of this operation. To describe the behaviour of this program, we need to describe a relation `R` between the input value and the associated output.



The semantics of this program is, thus, a relation `R` between elements of the set `E` of input values and elements of the set `S` of output values, that is, a subset of $E \times S$.

The semantics of a program is then a binary relation. The semantics of a programming language is, in turn, a ternary relation: "the program `p` with input value `e` returns the output value `s`". We denote this relation by $p,\ e \hookrightarrow s$. The program `p` and the input `e` are available before the execution of the program starts. Often, these two elements are paired in a *term* `p e`, and the semantics of the language assigns a value to this term. The semantics of the language is then a binary relation $t \hookrightarrow s$.

To express the semantics of a programming language we need a language that can express relations.

When the semantics of a program is a functional relation, that is, for each input value there is at most one output value, we say that the program is *deterministic*. Video games are examples of non-deterministic programs, since some randomness is necessary to make the game enjoyable. A language is deterministic if all the programs that can be written in the language are deterministic, or equivalently, if the semantics is a functional relation. In this case, it is possible to define its semantics using a language to define functions instead of a language to define relations.

Acknowledgements

The authors would like to thank Gérard Assayag, Antonio Bucciarelli, Roberto Di Cosmo, Xavier Leroy, Dave MacQueen, Luc Maranget, Michel Mauny, François Pottier, Didier Rémy, Alan Schmitt, Élodie-Jane Sims and Véronique Viguié Donzeau-Gouge.

Contents