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.

Also in this series

Iain D. Craig
Object-Oriented Programming Languages: Interpretation
978-1-84628-773-2

Max Bramer
Principles of Data Mining
978-1-84628-765-7

Hanne Riis Nielson and Flemming Nielson
Semantics with Applications: An Appetizer
978-1-84628-691-9

Michael Kifer and Scott A. Smolka
Introduction to Operating System Design and Implementation: The OSP 2 Approcah
978-1-84628-842-5

Phil Brooke and Richard Paige
Practical Distributed Processing
978-1-84628-840-1

Frank Klawonn
Computer Graphics with Java
978-1-84628-847-0

David Salomon
A Concise Introduction to Data Compression
978-1-84800-071-1

David Makinson
Sets, Logic and Maths for Computing
978-1-84628-844-9

Orit Hazzan
Agile Software Engineering
978-1-84800-198-5

Pankaj Jalote
A Concise Introduction to Software Engineering
978-1-84800-301-9

Alan P. Parkes
A Concise Introduction to Languages and Machines
978-1-84800-120-6

Gilles Dowek

Principles of Programming Languages

Springer

Gilles Dowek
École Polytechnique
France

Undergraduate Topics in Computer Science ISSN 1863-7310
ISBN: 978-1-84882-031-9 e-ISBN: 978-1-84882-032-6
DOI: 10.1007/978-1-84882-032-6

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

Library of Congress Control Number: 2008943965

Based on course notes by Gilles Dowek published in 2006 by L'Ecole Polytechnique with the following title: "Les principes des langages de programmation."

Printed on acid-free paper

Springer Science+Business Media
springer.com

The author wants to thank François Pottier, Philippe Baptiste, Julien Cervelle, Albert Cohen, Olivier Delande, Olivier Hermant, Ian Mackie, François Morain, Jean-Marc Steyaert and Paul Zimmermann for their remarks on a first version of this book.

Preface

We've known about algorithms for millennia, but we've only been writing computer programs for a few decades. A big difference between the Euclidean or Eratosthenes age and ours is that since the middle of the twentieth century, we express the algorithms we conceive using formal languages: programming languages.

Computer scientists are not the only ones who use formal languages. Optometrists, for example, prescribe eyeglasses using very technical expressions, such as "OD: -1.25 (-0.50) 180° OS: -1.00 (-0.25) 180°", in which the parentheses are essential. Many such formal languages have been created throughout history: musical notation, algebraic notation, etc. In particular, such languages have long been used to control machines, such as looms and cathedral chimes.

However, until the appearance of programming languages, those languages were only of limited importance: they were restricted to specialised fields with only a few specialists and written texts of those languages remained relatively scarce. This situation has changed with the appearance of programming languages, which have a wider range of applications than the prescription of eyeglasses or the control of a loom, are used by large communities, and have allowed the creation of programs of many hundreds of thousands of lines.

The appearance of programming languages has allowed the creation of artificial objects, programs, of a complexity incomparable to anything that has come before, such as steam engines or radios. These programs have, in return, allowed the creation of other complex objects, such as integrated circuits made of millions of transistors, or mathematical proofs that are hundreds of thousands of pages long. It is very surprising that we have succeeded in writing such complex programs in languages comprising such a small number of constructs — assignment, loops, etc. — that is to say in languages barely more sophisticated than the language of prescription eyeglasses.

Programs written in these programming languages have the novelty of not only being understandable by humans, which brings them closer to the scores used by organists, but also readable by machines, which brings them closer to the punch cards used in Barbarie organs.

The appearance of programming languages has therefore profoundly impacted our relationship with language, complexity, and machines.

This book is an introduction to the principles of programming languages. It uses the Java language for support. It is intended for students who already have some experience with computer programming. It is assumed that they have learned some programming empirically, in a single programming language, other than Java.

The first objective of this book will then be to learn the fundamentals of the Java programming language. However, knowing a single programming language is not sufficient to be a good programmer. For this, you must not only know several languages, but be able to easily learn new ones. This requires that you understand universal concepts like functions or cells, which exist in one form or another in all programming languages. This can only be done by comparing two or more languages. In this book, two comparison languages have been chosen: Caml and C. Therefore, the goal is not for the students to learn three programming languages simultaneously, but that with the comparison with Caml and C, they can learn the principles around which programming languages are created. This understanding will allow them to develop, if they wish, a real competence in Caml or in C, or in any other programming language.

Another objective of this book is for the students to begin acquiring the tools which permit them to precisely define the meaning of the program. This precision is, indeed, the only means to clearly understand what happens when a program is executed, and to reason in situations where complexity defies intuition. The idea is to describe the meaning of a statement by a function operating on a set of states. However, our expectations of this objective remain modest: students wishing to pursue this goal will have to do so elsewhere.

The final objective of this course is to learn basic algorithms for lists and trees. Here too, our expectations remain modest: students wishing to pursue this will also have to look elsewhere.

Contents

1. Imperative Core 1
1.1 Five Constructs 1
1.1.1 Assignment 1
1.1.2 Variable Declaration 3
1.1.3 Sequence 5
1.1.4 Test 6
1.1.5 Loop 6
1.2 Input and Output 7
1.2.1 Input 7
1.2.2 Output 7
1.3 The Semantics of the Imperative Core 8
1.3.1 The Concept of a State 8
1.3.2 Decomposition of the State 9
1.3.3 A Visual Representation of a State 10
1.3.4 The Value of Expressions 11
1.3.5 Execution of Statements 13

2. Functions 19
2.1 The Concept of Functions 19
2.1.1 Avoiding Repetition 19
2.1.2 Arguments 21
2.1.3 Return Values 22
2.1.4 The `return` Construct 23
2.1.5 Functions and Procedures 24
2.1.6 Global Variables 25
2.1.7 The Main Program 25

2.1.8 Global Variables Hidden by Local Variables 27
2.1.9 Overloading 28
2.2 The Semantics of Functions 29
2.2.1 The Value of Expressions 30
2.2.2 Execution of Statements 31
2.2.3 Order of Evaluation 34
2.2.4 Caml 34
2.2.5 C 36
2.3 Expressions as Statements 37
2.4 Passing Arguments by Value and Reference 37
2.4.1 Pascal 39
2.4.2 Caml 40
2.4.3 C 41
2.4.4 Java 45

3. Recursion 47
3.1 Calling a Function from Inside the Body of that Function 47
3.2 Recursive Definitions 48
3.2.1 Recursive Definitions and Circular Definitions 48
3.2.2 Recursive Definitions and Definitions by Induction 49
3.2.3 Recursive Definitions and Infinite Programs 49
3.2.4 Recursive Definitions and Fixed Point Equations 51
3.3 Caml 53
3.4 C 54
3.5 Programming Without Assignment 55

4. Records 59
4.1 Tuples with Named Fields 59
4.1.1 The Definition of a Record Type 60
4.1.2 Allocation of a Record 60
4.1.3 Accessing Fields 62
4.1.4 Assignment of Fields 62
4.1.5 Constructors 64
4.1.6 The Semantics of Records 65
4.2 Sharing 66
4.2.1 Sharing 66
4.2.2 Equality 68
4.2.3 Wrapper Types 68
4.3 Caml 73
4.3.1 Definition of a Record Type 73
4.3.2 Creating a Record 73
4.3.3 Accessing Fields 74

4.3.4 Assigning to Fields . . . 74
4.4 C . . . 76
4.4.1 Definition of a Record Type . . . 76
4.4.2 Creating a Record . . . 76
4.4.3 Accessing Fields . . . 77
4.4.4 Assigning to Fields . . . 77
4.5 Arrays . . . 79
4.5.1 Array Types . . . 79
4.5.2 Allocation of an Array . . . 80
4.5.3 Accessing and Assigning to Fields . . . 80
4.5.4 Arrays of Arrays . . . 82
4.5.5 Arrays in Caml . . . 83
4.5.6 Arrays in C . . . 84

5. Dynamic Data Types . . . 85
5.1 Recursive Records . . . 85
5.1.1 Lists . . . 85
5.1.2 The `null` Value . . . 86
5.1.3 An Example . . . 86
5.1.4 Recursive Definitions and Fixed Point Equations . . . 88
5.1.5 Infinite Values . . . 89
5.2 Disjunctive Types . . . 90
5.3 Dynamic Data Types and Computability . . . 92
5.4 Caml . . . 92
5.5 C . . . 94
5.6 Garbage Collection . . . 96
5.6.1 Inaccessible Cells . . . 96
5.6.2 Programming without Garbage Collection . . . 98
5.6.3 Global Methods of Memory Management . . . 100
5.6.4 Garbage Collection and Functions . . . 102

6. Programming with Lists . . . 103
6.1 Finite Sets and Functions of a Finite Domain . . . 103
6.1.1 Membership . . . 103
6.1.2 Association Lists . . . 104
6.2 Concatenation: Modify or Copy . . . 105
6.2.1 Modify . . . 105
6.2.2 Copy . . . 109
6.2.3 Using Recursion . . . 111
6.2.4 Chemical Reactions and Mathematical Functions . . . 111
6.3 List Inversion: an Extra Argument . . . 112
6.4 Lists and Arrays . . . 114

6.5 Stacks and Queues . . . 114
6.5.1 Stacks . . . 115
6.5.2 Queues . . . 118
6.5.3 Priority Queues . . . 119

7. Exceptions . . . 121
7.1 Exceptional Circumstances . . . 121
7.2 Exceptions . . . 122
7.3 Catching Exceptions . . . 122
7.4 The Propagation of Exceptions . . . 123
7.5 Error Messages . . . 124
7.6 The Semantics of Exceptions . . . 124
7.7 Caml . . . 125

8. Objects . . . 127
8.1 Classes . . . 127
8.1.1 Functions as Part of a Type . . . 127
8.1.2 The Semantics of Classes . . . 129
8.2 Dynamic Methods . . . 129
8.3 Methods and Functional Fields . . . 132
8.4 Static Fields . . . 132
8.5 Static Classes . . . 133
8.6 Inheritance . . . 134
8.7 Caml . . . 137

9. Programming with Trees . . . 139
9.1 Trees . . . 139
9.2 Traversing a Tree . . . 142
9.2.1 Depth First Traversal . . . 143
9.2.2 Breadth First Traversal . . . 145
9.3 Search Trees . . . 146
9.3.1 Membership . . . 146
9.3.2 Balanced Trees . . . 149
9.3.3 Dictionaries . . . 151
9.4 Priority Queues . . . 152
9.4.1 Partially Ordered Trees . . . 152
9.4.2 Partially Ordered Balanced Trees . . . 153

Index . . . 157