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 further volumes:
http://www.springer.com/series/7592

Peter Sestoft

Programming Language Concepts

Springer

Peter Sestoft
IT University of Copenhagen
Copenhagen, Denmark

ISSN 1863-7310 Undergraduate Topics in Computer Science
ISBN 978-1-4471-4155-6 ISBN 978-1-4471-4156-3 (eBook)
DOI 10.1007/978-1-4471-4156-3
Springer London Heidelberg New York Dordrecht

Library of Congress Control Number: 2012941284

Printed on acid-free paper

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

Preface

This book takes an operational approach to programming language concepts, studying those concepts in interpreters and compilers for some toy languages, and pointing out their relations to real-world programming languages.

What Is Covered Topics covered include abstract and concrete syntax; functional and imperative programming languages; interpretation, type checking, and compilation; peep-hole optimizations; abstract machines, automatic memory management and garbage collection; and the Java Virtual Machine and Microsoft's Common Language Infrastructure (also known as .NET Common Language Runtime).

Some effort is made throughout to put programming language concepts into their historical context, and to show how the concepts surface in languages that the students are assumed to know already; primarily Java or C#.

We do not cover regular expressions and parser construction in much detail. For this purpose, we refer to Torben Mogensen's textbook; see Chap. 3 and its references.

Why Virtual Machines? We do not consider generation of machine code for real microprocessors, nor classical compiler subjects such as register allocation. Instead the emphasis is on virtual stack machines and their intermediate languages, often known as bytecode.

Virtual machines are machine-like enough to make the central purpose and concepts of compilation and code generation clear, yet they are much simpler than present-day microprocessors such as Intel i7 and similar. Full understanding of performance issues in real microprocessors, with deep pipelines, register renaming, out-of-order execution, branch prediction, translation lookaside buffers and so on, requires a very detailed study of their architecture, usually not conveyed by compiler textbooks anyway. Certainly, a mere understanding of the instruction set, such as x86, conveys little information about whether code will be fast or not.

The widely used object-oriented languages Java and C# are rather far removed from the real hardware, and are most conveniently explained in terms of their virtual machines: the Java Virtual Machine and Microsoft's Common Language Infrastructure. Understanding the workings and implementation of these virtual machines sheds light on efficiency issues, design decisions and inherent limitations in Java and C#. To understand memory organization of classic imperative languages, we also study a small subset of C with arrays, pointer arithmetics, and recursive functions.

Why F#? We use the functional language F# as presentation language throughout to illustrate programming language concepts, by implementing interpreters and compilers for toy languages. The idea behind this is two-fold.

First, F# belongs to the ML family of languages and is ideal for implementing interpreters and compilers because it has datatypes and pattern matching and is strongly typed. This leads to a brevity and clarity of examples that cannot be matched by languages without these features.

Secondly, the active use of a functional language is an attempt to add a new dimension to students' world view, to broaden their imagination. The prevalent single-inheritance class-based object-oriented programming languages (namely, Java and C#) are very useful and versatile languages. But they have come to dominate computer science education to a degree where students may become unable to imagine other programming tools, especially such that use a completely different paradigm. Knowledge of a functional language will make the student a better designer and programmer, whether in Java, C# or C, and will prepare him or her to adapt to the programming languages of the future.

For instance, so-called generic types and methods appeared in Java and C# in 2004 but has been part of other languages, most notably ML, since 1978. Similarly, garbage collection has been used in functional languages since Lisp in 1960, but entered mainstream use more than 30 years later, with Java.

Appendix A gives a brief introduction to those parts of F# used in this book. Students who do not know F# should learn those parts during the first third of this course, using the appendix or a textbook such as Hansen and Rischel or a reference such as Syme et al.; see Appendix A and its references.

Supporting Material There are practical exercises at the end of each chapter. Moreover, the book is accompanied by complete implementations in F# of lexer and parser specifications, abstract syntaxes, interpreters, compilers, and run-time systems (abstract machines, in Java and C) for a range of toy languages. This material, and lecture slides in PDF, are available separately from the book's homepage:

http://www.itu.dk/people/sestoft/plc/

Acknowledgements

This book originated as lecture notes for courses held at the IT University of Copenhagen, Denmark. I would like to thank Andrzej Wasowski, Ken Friis Larsen, Hannes Mehnert, David Raymond Christiansen and past and present students, in particular Niels Kokholm and Mikkel Bundgaard, who pointed out mistakes and made suggestions on examples and presentation in earlier drafts. I also owe a big thanks to Neil D. Jones and Mads Tofte who influenced my own view of programming languages and the presentation of programming language concepts.

Contents

1 Introduction . . . 1
- 1.1 Files for This Chapter . . . 1
- 1.2 Meta Language and Object Language . . . 1
- 1.3 A Simple Language of Expressions . . . 2
  - 1.3.1 Expressions Without Variables . . . 2
  - 1.3.2 Expressions with Variables . . . 3
- 1.4 Syntax and Semantics . . . 4
- 1.5 Representing Expressions by Objects . . . 5
- 1.6 The History of Programming Languages . . . 7
- 1.7 Exercises . . . 9
- References . . . 12

2 Interpreters and Compilers . . . 13
- 2.1 Files for This Chapter . . . 13
- 2.2 Interpreters and Compilers . . . 13
- 2.3 Scope and Bound and Free Variables . . . 14
  - 2.3.1 Expressions with Let-Bindings and Static Scope . . . 15
  - 2.3.2 Closed Expressions . . . 16
  - 2.3.3 The Set of Free Variables . . . 16
  - 2.3.4 Substitution: Replacing Variables by Expressions . . . 17
- 2.4 Integer Addresses Instead of Names . . . 19
- 2.5 Stack Machines for Expression Evaluation . . . 21
- 2.6 Postscript, a Stack-Based Language . . . 22
- 2.7 Compiling Expressions to Stack Machine Code . . . 24
- 2.8 Implementing an Abstract Machine in Java . . . 25
- 2.9 History and Literature . . . 26
- 2.10 Exercises . . . 27
- References . . . 29

3 From Concrete Syntax to Abstract Syntax . . . 31
- 3.1 Preparatory Reading . . . 31

3.2 Lexers, Parsers, and Generators . . . 32
3.3 Regular Expressions in Lexer Specifications . . . 33
3.4 Grammars in Parser Specifications . . . 34
3.5 Working with F# Modules . . . 35
3.6 Using `fslex` and `fsyacc` . . . 36
3.6.1 Setting up `fslex` and `fsyacc` for Command Line Use . . . 36
3.6.2 Using `fslex` and `fsyacc` in Visual Studio . . . 37
3.6.3 Parser Specification for Expressions . . . 37
3.6.4 Lexer Specification for Expressions . . . 38
3.6.5 The ExprPar.fsyacc.output File Generated by `fsyacc` . . . 40
3.6.6 Exercising the Parser Automaton . . . 43
3.6.7 Shift/Reduce Conflicts . . . 45
3.7 Lexer and Parser Specification Examples . . . 46
3.7.1 A Small Functional Language . . . 46
3.7.2 Lexer and Parser Specifications for Micro-SQL . . . 47
3.8 A Handwritten Recursive Descent Parser . . . 48
3.9 JavaCC: Lexer-, Parser-, and Tree Generator . . . 50
3.10 History and Literature . . . 52
3.11 Exercises . . . 54
References . . . 56

4 A First-Order Functional Language . . . 57
4.1 Files for This Chapter . . . 57
4.2 Examples and Abstract Syntax . . . 57
4.3 Run-Time Values: Integers and Closures . . . 59
4.4 A Simple Environment Implementation . . . 59
4.5 Evaluating the Functional Language . . . 60
4.6 Evaluation Rules for Micro-ML . . . 62
4.7 Static Scope and Dynamic Scope . . . 64
4.8 Type-Checking an Explicitly Typed Language . . . 65
4.9 Type Rules for Monomorphic Types . . . 68
4.10 Static Typing and Dynamic Typing . . . 70
4.10.1 Dynamic Typing in Java and C# Array Assignment . . . 71
4.10.2 Dynamic Typing in Non-Generic Collection Classes . . . 71
4.11 History and Literature . . . 72
4.12 Exercises . . . 73
References . . . 76

5 Higher-Order Functions . . . 77
5.1 Files for This Chapter . . . 77
5.2 Higher-Order Functions in F# . . . 77
5.3 Higher-Order Functions in the Mainstream . . . 78
5.3.1 Higher-Order Functions in Java . . . 78
5.3.2 Higher-Order Functions in C# . . . 80
5.3.3 Google MapReduce . . . 81
5.4 A Higher-Order Functional Language . . . 81

5.5 Eager and Lazy Evaluation 82
5.6 The Lambda Calculus 83
5.7 History and Literature 86
5.8 Exercises 86
References 91

6 Polymorphic Types 93
6.1 Files for This Chapter 93
6.2 ML-Style Polymorphic Types 93
6.2.1 Informal Explanation of ML Type Inference 94
6.2.2 Which Type Parameters May Be Generalized 95
6.3 Type Rules for Polymorphic Types 97
6.4 Implementing ML Type Inference 99
6.4.1 Type Equation Solving by Unification 102
6.4.2 The Union-Find Algorithm 102
6.4.3 The Complexity of ML-Style Type Inference 103
6.5 Generic Types in Java and C# 104
6.6 Co-Variance and Contra-Variance 105
6.6.1 Java Wildcards 107
6.6.2 C# Variance Declarations 108
6.6.3 The Variance Mechanisms of Java and C# 109
6.7 History and Literature 109
6.8 Exercises 109
References 113

7 Imperative Languages 115
7.1 Files for This Chapter 115
7.2 A Naive Imperative Language 115
7.3 Environment and Store 117
7.4 Parameter Passing Mechanisms 118
7.5 The C Programming Language 120
7.5.1 Integers, Pointers and Arrays in C 120
7.5.2 Type Declarations in C 122
7.6 The Micro-C Language 123
7.6.1 Interpreting Micro-C 125
7.6.2 Example Programs in Micro-C 125
7.6.3 Lexer Specification for Micro-C 125
7.6.4 Parser Specification for Micro-C 128
7.7 Notes on Strachey's *Fundamental Concepts* 130
7.8 History and Literature 133
7.9 Exercises 133
References 136

8 Compiling Micro-C 137
8.1 Files for This Chapter 137
8.2 An Abstract Stack Machine 138
8.2.1 The State of the Abstract Machine 138

8.2.2 The Abstract Machine Instruction Set 139
8.2.3 The Symbolic Machine Code 140
8.2.4 The Abstract Machine Implemented in Java 141
8.2.5 The Abstract Machine Implemented in C 142
8.3 The Structure of the Stack at Run-Time 143
8.4 Compiling Micro-C to Abstract Machine Code 144
8.5 Compilation Schemes for Micro-C 144
8.6 Compilation of Statements 146
8.7 Compilation of Expressions 149
8.8 Compilation of Access Expressions 149
8.9 Compilation to Real Machine Code 150
8.10 History and Literature 150
8.11 Exercises 151
References 153

9 Real-World Abstract Machines 155
9.1 Files for This Chapter 155
9.2 An Overview of Abstract Machines 155
9.3 The Java Virtual Machine (JVM) 157
9.3.1 The JVM Run-Time State 157
9.3.2 The JVM Bytecode 159
9.3.3 The Contents of JVM Class Files 159
9.3.4 Bytecode Verification 162
9.4 The Common Language Infrastructure (CLI) 163
9.5 Generic Types in CLI and JVM 165
9.5.1 A Generic Class in Bytecode 165
9.5.2 Consequences for Java 166
9.6 Decompilers for Java and C# 168
9.7 Just-in-Time Compilation 169
9.8 History and Literature 171
9.9 Exercises 171
References 173

10 Garbage Collection 175
10.1 Files for This Chapter 175
10.2 Predictable Lifetime and Stack Allocation 175
10.3 Unpredictable Lifetime and Heap Allocation 176
10.4 Allocation in a Heap 177
10.5 Garbage Collection Techniques 178
10.5.1 The Heap and the Freelist 178
10.5.2 Garbage Collection by Reference Counting 179
10.5.3 Mark-Sweep Collection 180
10.5.4 Two-Space Stop-and-Copy Collection 181
10.5.5 Generational Garbage Collection 182
10.5.6 Conservative Garbage Collection 183
10.5.7 Garbage Collectors Used in Existing Systems 184

10.6 Programming with a Garbage Collector . . . 185
10.6.1 Memory Leaks . . . 185
10.6.2 Finalizers . . . 186
10.6.3 Calling the Garbage Collector . . . 186
10.7 Implementing a Garbage Collector in C . . . 186
10.7.1 The List-C Language . . . 186
10.7.2 The List-C Machine . . . 189
10.7.3 Distinguishing References from Integers . . . 190
10.7.4 Memory Structures in the Garbage Collector . . . 191
10.7.5 Actions of the Garbage Collector . . . 192
10.8 History and Literature . . . 193
10.9 Exercises . . . 194
References . . . 198

11 Continuations . . . 201
11.1 Files for This Chapter . . . 201
11.2 Tail-Calls and Tail-Recursive Functions . . . 201
11.2.1 A Recursive But Not Tail-Recursive Function . . . 201
11.2.2 A Tail-Recursive Function . . . 202
11.2.3 Which Calls Are Tail Calls? . . . 203
11.3 Continuations and Continuation-Passing Style . . . 204
11.3.1 Writing a Function in Continuation-Passing Style . . . 204
11.3.2 Continuations and Accumulating Parameters . . . 205
11.3.3 The CPS Transformation . . . 206
11.4 Interpreters in Continuation-Passing Style . . . 206
11.4.1 A Continuation-Based Functional Interpreter . . . 207
11.4.2 Tail Position and Continuation-Based Interpreters . . . 209
11.4.3 A Continuation-Based Imperative Interpreter . . . 209
11.5 The Frame Stack and Continuations . . . 211
11.6 Exception Handling in a Stack Machine . . . 211
11.7 Continuations and Tail Calls . . . 213
11.8 Callcc: Call with Current Continuation . . . 214
11.9 Continuations and Backtracking . . . 215
11.9.1 Expressions in Icon . . . 215
11.9.2 Using Continuations to Implement Backtracking . . . 216
11.10History and Literature . . . 218
11.11Exercises . . . 219
References . . . 223

12 A Locally Optimizing Compiler . . . 225
12.1 Files for This Chapter . . . 225
12.2 Generating Optimized Code Backwards . . . 225
12.3 Backwards Compilation Functions . . . 226
12.3.1 Optimizing Expression Code While Generating It . . . 227
12.3.2 The Old Compilation of Jumps . . . 229
12.3.3 Optimizing a Jump While Generating It . . . 230

12.3.4 Optimizing Logical Expression Code 232
12.3.5 Eliminating Dead Code 233
12.3.6 Optimizing Tail Calls 234
12.3.7 Remaining Deficiencies of the Generated Code 237
12.4 Other Optimizations 237
12.5 A Command Line Compiler for Micro-C 238
12.6 History and Literature 239
12.7 Exercises 240
References 243

Appendix A Crash Course in F# 245
A.1 Files for This Chapter 245
A.2 Getting Started 245
A.3 Expressions, Declarations and Types 246
A.3.1 Arithmetic and Logical Expressions 246
A.3.2 String Values and Operators 248
A.3.3 Types and Type Errors 248
A.3.4 Function Declarations 250
A.3.5 Recursive Function Declarations 251
A.3.6 Type Constraints 251
A.3.7 The Scope of a Binding 251
A.4 Pattern Matching 252
A.5 Pairs and Tuples 253
A.6 Lists 254
A.7 Records and Labels 255
A.8 Raising and Catching Exceptions 256
A.9 Datatypes 257
A.9.1 The `option` Datatype 258
A.9.2 Binary Trees Represented by Recursive Datatypes 259
A.9.3 Curried Functions 260
A.10 Type Variables and Polymorphic Functions 260
A.10.1 Polymorphic Datatypes 261
A.10.2 Type Abbreviations 262
A.11 Higher-Order Functions 263
A.11.1 Anonymous Functions 263
A.11.2 Higher-Order Functions on Lists 264
A.12 F# Mutable References 265
A.13 F# Arrays 266
A.14 Other F# Features 267
A.15 Exercises 267
References 269

Index 271