

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 <code>fslex</code> and <code>fsyacc</code>	36
3.6.1	Setting up <code>fslex</code> and <code>fsyacc</code> for Command Line Use . .	36
3.6.2	Using <code>fslex</code> and <code>fsyacc</code> in Visual Studio	37
3.6.3	Parser Specification for Expressions	37
3.6.4	Lexer Specification for Expressions	38
3.6.5	The <code>ExprPar.fsyacc.output</code> File Generated by <code>fsyacc</code> . . .	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
 - 11.10 History and Literature 218
 - 11.11 Exercises 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 <code>option</code> 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



<http://www.springer.com/978-1-4471-4155-6>

Programming Language Concepts

Sestoft, P.

2012, XIV, 278 p. 17 illus., Softcover

ISBN: 978-1-4471-4155-6