

## Contents

<b>1</b>	<b>Prologue</b>	<i>page 1</i>
1.1	Why OCaml?	1
1.1.1	A Brief History	2
1.1.2	The Base Standard Library	3
1.1.3	The OCaml Platform	4
1.2	About This Book	4
1.2.1	What to Expect	5
1.2.2	Installation Instructions	5
1.2.3	Code Examples	6
1.3	Contributors	6
	<b>Part I Language Concepts</b>	7
<b>2</b>	<b>A Guided Tour</b>	9
2.1	OCaml as a Calculator	10
2.2	Functions and Type Inference	11
2.2.1	Type Inference	13
2.2.2	Inferring Generic Types	14
2.3	Tuples, Lists, Options, and Pattern Matching	15
2.3.1	Tuples	15
2.3.2	Lists	17
2.3.3	Options	21
2.4	Records and Variants	22
2.5	Imperative Programming	24
2.5.1	Arrays	24
2.5.2	Mutable Record Fields	25
2.5.3	Refs	26
2.5.4	For and While Loops	28
2.6	A Complete Program	29
2.6.1	Compiling and Running	30
2.7	Where to Go from Here	30

viii	<b>Contents</b>	
<b>3</b>	<b>Variables and Functions</b>	31
3.1	Variables	31
3.1.1	Pattern Matching and Let	33
3.2	Functions	34
3.2.1	Anonymous Functions	34
3.2.2	Multiargument Functions	36
3.2.3	Recursive Functions	37
3.2.4	Prefix and Infix Operators	38
3.2.5	Declaring Functions with <code>function</code>	41
3.2.6	Labeled Arguments	42
3.2.7	Optional Arguments	45
<b>4</b>	<b>Lists and Patterns</b>	50
4.1	List Basics	50
4.2	Using Patterns to Extract Data from a List	51
4.3	Limitations (and Blessings) of Pattern Matching	52
4.3.1	Performance	53
4.3.2	Detecting Errors	54
4.4	Using the List Module Effectively	55
4.4.1	More Useful List Functions	58
4.5	Tail Recursion	60
4.6	Terser and Faster Patterns	62
<b>5</b>	<b>Files, Modules, and Programs</b>	66
5.1	Single-File Programs	66
5.2	Multifile Programs and Modules	69
5.3	Signatures and Abstract Types	70
5.4	Concrete Types in Signatures	73
5.5	Nested Modules	74
5.6	Opening Modules	76
5.6.1	Open Modules Rarely	76
5.6.2	Prefer Local Opens	76
5.6.3	Using Module Shortcuts Instead	77
5.7	Including Modules	77
5.8	Common Errors with Modules	79
5.8.1	Type Mismatches	79
5.8.2	Missing Definitions	80
5.8.3	Type Definition Mismatches	80
5.8.4	Cyclic Dependencies	81
5.9	Designing with Modules	82
5.9.1	Expose Concrete Types Rarely	82
5.9.2	Design for the Call Site	82
5.9.3	Create Uniform Interfaces	83

	Contents	ix
	5.9.4 Interfaces Before Implementations	83
<b>6</b>	<b>Records</b>	85
	6.1 Patterns and Exhaustiveness	87
	6.2 Field Punning	89
	6.3 Reusing Field Names	90
	6.4 Functional Updates	93
	6.5 Mutable Fields	94
	6.6 First-Class Fields	95
<b>7</b>	<b>Variants</b>	99
	7.1 Catch-All Cases and Refactoring	102
	7.2 Combining Records and Variants	104
	7.2.1 Embedded Records	107
	7.3 Variants and Recursive Data Structures	108
	7.4 Polymorphic Variants	111
	7.4.1 Example: Terminal Colors Redux	113
	7.4.2 When to Use Polymorphic Variants	117
<b>8</b>	<b>Error Handling</b>	119
	8.1 Error-Aware Return Types	119
	8.1.1 Encoding Errors with Result	120
	8.1.2 Error and Or_error	121
	8.1.3 bind and Other Error Handling Idioms	122
	8.2 Exceptions	124
	8.2.1 Helper Functions for Throwing Exceptions	126
	8.2.2 Exception Handlers	127
	8.2.3 Cleaning Up in the Presence of Exceptions	128
	8.2.4 Catching Specific Exceptions	129
	8.2.5 Backtraces	130
	8.2.6 From Exceptions to Error-Aware Types and Back Again	132
	8.3 Choosing an Error-Handling Strategy	133
<b>9</b>	<b>Imperative Programming</b>	134
	9.1 Example: Imperative Dictionaries	134
	9.2 Primitive Mutable Data	138
	9.2.1 Array-Like Data	138
	9.2.2 Mutable Record and Object Fields and Ref Cells	139
	9.2.3 Foreign Functions	140
	9.3 For and While Loops	140
	9.4 Example: Doubly Linked Lists	141
	9.4.1 Modifying the List	143
	9.4.2 Iteration Functions	144

x	<b>Contents</b>	
	9.5 Laziness and Other Benign Effects	145
	9.5.1 Memoization and Dynamic Programming	146
	9.6 Input and Output	152
	9.6.1 Terminal I/O	153
	9.6.2 Formatted Output with printf	154
	9.6.3 File I/O	156
	9.7 Order of Evaluation	158
	9.8 Side Effects and Weak Polymorphism	159
	9.8.1 The Value Restriction	160
	9.8.2 Partial Application and the Value Restriction	161
	9.8.3 Relaxing the Value Restriction	162
	9.9 Summary	164
<b>10</b>	<b>GADTs</b>	166
	10.1 A Little Language	166
	10.1.1 Making the Language Type-Safe	168
	10.1.2 Trying to Do Better with Ordinary Variants	169
	10.1.3 GADTs to the Rescue	170
	10.1.4 GADTs, Locally Abstract Types, and Polymorphic Recursion	172
	10.2 When Are GADTs Useful?	173
	10.2.1 Varying Your Return Type	173
	10.2.2 Capturing the Unknown	176
	10.2.3 Abstracting Computational Machines	177
	10.2.4 Narrowing the Possibilities	180
	10.3 Limitations of GADTs	187
	10.3.1 Or-Patterns	188
	10.3.2 Deriving Serializers	188
<b>11</b>	<b>Functors</b>	191
	11.1 A Trivial Example	191
	11.2 A Bigger Example: Computing with Intervals	193
	11.2.1 Making the Functor Abstract	196
	11.2.2 Sharing Constraints	197
	11.2.3 Destructive Substitution	199
	11.2.4 Using Multiple Interfaces	201
	11.3 Extending Modules	205
<b>12</b>	<b>First-Class Modules</b>	209
	12.1 Working with First-Class Modules	209
	12.1.1 Creating First-Class Modules	209
	12.1.2 Inference and Anonymous Modules	210
	12.1.3 Unpacking First-Class Modules	210
	12.1.4 Functions for Manipulating First-Class Modules	210
	12.1.5 Richer First-Class Modules	211

	Contents	xi
12.1.6	Exposing types	211
12.2	Example: A Query-Handling Framework	213
12.2.1	Implementing a Query Handler	215
12.2.2	Dispatching to Multiple Query Handlers	216
12.2.3	Loading and Unloading Query Handlers	219
12.3	Living Without First-Class Modules	222
<b>13</b>	<b>Objects</b>	<b>223</b>
13.1	OCaml Objects	224
13.2	Object Polymorphism	225
13.3	Immutable Objects	227
13.4	When to Use Objects	228
13.5	Subtyping	228
13.5.1	Width Subtyping	229
13.5.2	Depth Subtyping	229
13.5.3	Variance	230
13.5.4	Narrowing	234
13.5.5	Subtyping Versus Row Polymorphism	235
<b>14</b>	<b>Classes</b>	<b>237</b>
14.1	OCaml Classes	237
14.2	Class Parameters and Polymorphism	238
14.3	Object Types as Interfaces	239
14.3.1	Functional Iterators	242
14.4	Inheritance	243
14.5	Class Types	244
14.6	Open Recursion	245
14.7	Private Methods	246
14.8	Binary Methods	247
14.9	Virtual Classes and Methods	251
14.9.1	Create Some Simple Shapes	251
14.10	Initializers	254
14.11	Multiple Inheritance	254
14.11.1	How Names Are Resolved	254
14.11.2	Mixins	255
14.11.3	Displaying the Animated Shapes	258
<b>Part II</b>	<b>Tools and Techniques</b>	<b>261</b>
<b>15</b>	<b>Maps and Hash Tables</b>	<b>263</b>
15.1	Maps	263
15.1.1	Sets	265
15.1.2	Modules and Comparators	265

xii	<b>Contents</b>	
	15.1.3	Why Do We Need Comparator Witnesses? 267
	15.1.4	The Polymorphic Comparator 269
	15.1.5	Satisfying <code>Comparator.S</code> with <code>[@@deriving]</code> 270
	15.1.6	Applying <code>[@@deriving]</code> to Maps and Sets 272
	15.1.7	Trees 273
	15.2	Hash Tables 274
	15.2.1	Time Complexity of Hash Tables 274
	15.2.2	Collisions with the Polymorphic Hash Function 275
	15.3	Choosing Between Maps and Hash Tables 276
<b>16</b>	<b>Command-Line Parsing</b>	280
	16.1	Basic Command-Line Parsing 280
	16.1.1	Defining an Anonymous Argument 281
	16.1.2	Defining Basic Commands 281
	16.1.3	Running Commands 282
	16.1.4	Multi-Argument Commands 284
	16.2	Argument Types 285
	16.2.1	Defining Custom Argument Types 286
	16.2.2	Optional and Default Arguments 286
	16.2.3	Sequences of Arguments 288
	16.3	Adding Labeled Flags 289
	16.4	Grouping Subcommands Together 291
	16.5	Prompting for Interactive Input 293
	16.6	Command-Line Autocompletion with <code>bash</code> 294
	16.6.1	Generating Completion Fragments from Command 295
	16.6.2	Installing the Completion Fragment 295
	16.7	Alternative Command-Line Parsers 296
<b>17</b>	<b>Concurrent Programming with <code>Async</code></b>	297
	17.1	<code>Async</code> Basics 297
	17.1.1	Using <code>Let</code> Syntax 300
	17.1.2	<code>Ivars</code> and <code>Upon</code> 301
	17.2	Example: An Echo Server 303
	17.2.1	Improving the Echo Server 306
	17.3	Example: Searching Definitions with <code>DuckDuckGo</code> 309
	17.3.1	URI Handling 309
	17.3.2	Parsing JSON Strings 310
	17.3.3	Executing an HTTP Client Query 310
	17.4	Exception Handling 312
	17.4.1	Monitors 314
	17.4.2	Example: Handling Exceptions with <code>DuckDuckGo</code> 316
	17.5	Timeouts, Cancellation, and Choices 318
	17.6	Working with System Threads 320

	Contents	xiii
17.6.1	Thread-Safety and Locking	323
<b>18</b>	<b>Testing</b>	<b>325</b>
18.1	Inline Tests	326
18.1.1	More Readable Errors with <code>test_eq</code>	327
18.1.2	Where Should Tests Go?	328
18.2	Expect Tests	329
18.2.1	Basic Mechanics	329
18.2.2	What Are Expect Tests Good For?	330
18.2.3	Exploratory Programming	331
18.2.4	Visualizing Complex Behavior	333
18.2.5	End-to-End Tests	336
18.2.6	How to Make a Good Expect Test	339
18.3	Property Testing with Quickcheck	339
18.3.1	Handling Complex Types	341
18.3.2	More Control with Let-Syntax	342
18.4	Other Testing Tools	343
18.4.1	Other Tools to Do (Mostly) the Same Things	343
18.4.2	Fuzzing	344
<b>19</b>	<b>Handling JSON Data</b>	<b>345</b>
19.1	JSON Basics	345
19.2	Parsing JSON with Yojson	346
19.3	Selecting Values from JSON Structures	348
19.4	Constructing JSON Values	351
19.5	Using Nonstandard JSON Extensions	353
19.6	Automatically Mapping JSON to OCaml Types	354
19.6.1	ATD Basics	354
19.6.2	ATD Annotations	355
19.6.3	Compiling ATD Specifications to OCaml	355
19.6.4	Example: Querying GitHub Organization Information	357
<b>20</b>	<b>Parsing with OCamllex and Menhir</b>	<b>361</b>
20.1	Lexing and Parsing	362
20.2	Defining a Parser	363
20.2.1	Describing the Grammar	364
20.2.2	Parsing Sequences	365
20.3	Defining a Lexer	367
20.3.1	OCaml Prelude	367
20.3.2	Regular Expressions	368
20.3.3	Lexing Rules	368
20.3.4	Recursive Rules	370
20.4	Bringing It All Together	371

xiv	<b>Contents</b>	
<b>21</b>	<b>Data Serialization with S-Expressions</b>	374
21.1	Basic Usage	374
21.1.1	S-Expression Converters for New Types	376
21.2	The Sexp Format	378
21.3	Preserving Invariants	379
21.4	Getting Good Error Messages	382
21.5	Sexp-Conversion Directives	383
21.5.1	@sexp.opaque	383
21.5.2	@sexp.list	384
21.5.3	@sexp.option	385
21.5.4	Specifying Defaults	385
<b>22</b>	<b>The OCaml Platform</b>	388
22.1	A Hello World OCaml Project	389
22.1.1	Setting Up an Opam Local Switch	389
22.1.2	Choosing an OCaml Compiler Version	390
22.1.3	Structure of an OCaml Project	391
22.1.4	Defining Module Names	391
22.1.5	Defining Libraries as Collections of Modules	392
22.1.6	Writing Test Cases for a Library	392
22.1.7	Building an Executable Program	393
22.2	Setting Up an Integrated Development Environment	394
22.2.1	Using Visual Studio Code	394
22.2.2	Browsing Interface Documentation	395
22.2.3	Autoformatting Your Source Code	396
22.3	Publishing Your Code Online	396
22.3.1	Defining Opam Packages	396
22.3.2	Generating Project Metadata from Dune	397
22.3.3	Setting up Continuous Integration	398
22.3.4	Other Conventions	399
22.3.5	Releasing Your Code into the Opam Repository	400
22.4	Learning More from Real Projects	401
<b>Part III</b>	<b>The Compiler and Runtime System</b>	403
<b>23</b>	<b>Foreign Function Interface</b>	405
23.1	Example: A Terminal Interface	406
23.2	Basic Scalar C Types	410
23.3	Pointers and Arrays	411
23.3.1	Allocating Typed Memory for Pointers	412
23.3.2	Using Views to Map Complex Values	413
23.4	Structs and Unions	414
23.4.1	Defining a Structure	414



	Contents	xv
	23.4.2 Adding Fields to Structures	415
	23.4.3 Incomplete Structure Definitions	415
	23.4.4 Defining Arrays	418
23.5	Passing Functions to C	419
	23.5.1 Example: A Command-Line Quicksort	420
23.6	Learning More About C Bindings	422
	23.6.1 Struct Memory Layout	423
<b>24</b>	<b>Memory Representation of Values</b>	<b>424</b>
	24.1 OCaml Blocks and Values	425
	24.1.1 Distinguishing Integers and Pointers at Runtime	425
	24.2 Blocks and Values	427
	24.2.1 Integers, Characters, and Other Basic Types	428
	24.3 Tuples, Records, and Arrays	428
	24.3.1 Floating-Point Numbers and Arrays	429
	24.4 Variants and Lists	430
	24.5 Polymorphic Variants	431
	24.6 String Values	432
	24.7 Custom Heap Blocks	432
	24.7.1 Managing External Memory with Bigarray	433
<b>25</b>	<b>Understanding the Garbage Collector</b>	<b>435</b>
	25.1 Mark and Sweep Garbage Collection	435
	25.2 Generational Garbage Collection	436
	25.3 The Fast Minor Heap	436
	25.3.1 Allocating on the Minor Heap	437
	25.4 The Long-Lived Major Heap	438
	25.4.1 Allocating on the Major Heap	439
	25.4.2 Memory Allocation Strategies	439
	25.4.3 Marking and Scanning the Heap	441
	25.4.4 Heap Compaction	442
	25.4.5 Intergenerational Pointers	443
	25.5 Attaching Finalizer Functions to Values	445
<b>26</b>	<b>The Compiler Frontend: Parsing and Type Checking</b>	<b>447</b>
	26.1 An Overview of the Toolchain	447
	26.1.1 Obtaining the Compiler Source Code	448
	26.2 Parsing Source Code	449
	26.2.1 Syntax Errors	449
	26.2.2 Generating Documentation from Interfaces	450
	26.3 Preprocessing with ppx	451
	26.3.1 Extension Attributes	452
	26.3.2 Commonly Used Extension Attributes	453
	26.3.3 Extension Nodes	454

---

26.4	Static Type Checking	454
26.4.1	Displaying Inferred Types from the Compiler	455
26.4.2	Type Inference	456
26.4.3	Modules and Separate Compilation	460
26.4.4	Wrapping Libraries with Module Aliases	462
26.4.5	Shorter Module Paths in Type Errors	464
26.5	The Typed Syntax Tree	465
26.5.1	Examining the Typed Syntax Tree Directly	465
<b>27</b>	<b>The Compiler Backend: Bytecode and Native code</b>	<b>468</b>
27.1	The Untyped Lambda Form	468
27.1.1	Pattern Matching Optimization	468
27.1.2	Benchmarking Pattern Matching	470
27.2	Generating Portable Bytecode	472
27.2.1	Compiling and Linking Bytecode	474
27.2.2	Executing Bytecode	474
27.2.3	Embedding OCaml Bytecode in C	475
27.3	Compiling Fast Native Code	476
27.3.1	Inspecting Assembly Output	477
27.3.2	Debugging Native Code Binaries	480
27.3.3	Profiling Native Code	483
27.3.4	Embedding Native Code in C	485
27.4	Summarizing the File Extensions	486
	<i>Index</i>	487