Preface **XV**

1 Introduction to Some Programming Styles 1

1.1 Functional Programming **1**
1.1.1 Easier Programming . . . 2
1.1.2 Underlying Formalisms . . . 3
1.1.3 Functions as Values . . . 4
1.1.4 Constraints of Functional Programming . . . 5

1.2 Generic Programming **7**
1.2.1 Combining Type Flexibility and Type Precision . . . 8
1.2.2 Combining Components Variability and Link Precision . . . 10

1.3 Object Programming **12**
1.3.1 Effective Modeling . . . 13
1.3.2 Compatible Object Types . . . 13
1.3.3 Objects as Values . . . 14

1.4 Mixing Up Styles Together **14**

2 Quick Introduction to OCaml 15

2.1 The Interactive Environment **15**
2.1.1 Using the Interactive Environment . . . 15
2.1.2 Typing . . . 18
2.1.3 Exploiting the Standard Library . . . 19

2.2 Type Reading **20**
2.2.1 Verifying Existing Definitions . . . 21
2.2.2 Reading Unary Function Types . . . 21
2.2.3 Reading n-ary Function Types . . . 22
2.2.4 Reading Parameterized Types . . . 25

2.3 New Value Definitions **27**

3 Elements of Functional Programming 31

3.1 Basics **31**
3.1.1 Terminology . . . 31
3.1.2 Variables and Bindings . . . 33
3.1.3 Global Binding Definitions . . . 33
3.1.4 Local Binding Definitions . . . 36
3.1.5 Life and Hiding . . . 37
3.1.6 Garbage-Collecting . . . 39
3.1.7 Function Definitions . . . 40
3.1.8 Recursive Function Definitions . . . 41
3.1.9 Local Environments within Functions . . . 42

3.1.10 Functional Values and Closures . . . 45
3.1.11 The `let`'s: a Partitioning System . . . 47

3.2 Evaluation Strategies **48**
3.2.1 Result Independance . . . 48
3.2.2 Value Evaluation . . . 49
3.2.3 Name and Lazy Evaluations . . . 51
3.2.4 Choosing an Evaluation Strategy . . . 54
3.2.5 Exceptions in Evaluation Strategies . . . 55

4 Functions 57

4.1 Forms and Facets of Functions **57**
4.1.1 Functional Parameters and Code Transmission . . . 57
4.1.2 Functions in Data Structures and Code Organization . . . 59
4.1.3 Anonymous Functions . . . 60
4.1.4 Using Anonymous Functions . . . 61
4.1.5 N-ary and Curried Functions . . . 62
4.1.6 Using Curried Functions . . . 63
4.1.7 Tail Recursive Functions . . . 66
4.1.8 Infixation . . . 69

4.2 Functional Genericity **71**
4.2.1 Polymorphisms . . . 71
4.2.2 Generic Functions . . . 72
4.2.3 Generic Functions in Other Languages . . . 75
4.2.4 Generic Predicates . . . 77
4.2.5 The Limits of the Functional Genericity . . . 78

4.3 Function Typing **80**
4.3.1 Explicit Typing . . . 80
4.3.2 Type Abbreviations . . . 81
4.3.3 Using Explicit Typing . . . 82
4.3.4 The Type Inference and its Error Messages . . . 83
4.3.5 The Type Inference and the Search for Errors . . . 85

5 Types 89

5.1 Type Zoology **89**
5.1.1 The Classical Hierarchy . . . 89
5.1.2 The Access Hierarchy . . . 91
5.1.3 Types with Functional or Imperative Nature . . . 93

5.2 Record Types **94**
5.2.1 Defining Record Types . . . 94
5.2.2 Instantiating Record Types . . . 94
5.2.3 Accessing Fields . . . 95
5.2.4 Record Type Possibilities . . . 96

5.2.5 Data Types and Recursion with Record Types . . . 97
5.2.6 The Limits and Advantages of Record Types . . . 99
5.2.7 Type Redefinitions . . . 100

5.3 Inductive Types **101**
5.3.1 Inductive Type Definitions . . . 101
5.3.2 Inductive Type Instanciations . . . 102
5.3.3 Inductive Type Possibilities . . . 103
5.3.4 The Inductivity of Inductive Types . . . 105

5.4 Pattern Matching **106**
5.4.1 Patterns . . . 106
5.4.2 Pattern Matching at Work . . . 106
5.4.3 Filtering . . . 109
5.4.4 The Reasons of Using Filtering . . . 112
5.4.5 The Limits of Pattern Matching and Filtering . . . 114

5.5 Some Applications of Inductive Types **117**
5.5.1 Unions of Distinct Types . . . 117
5.5.2 Clarifying Code . . . 118
5.5.3 Partial Functions . . . 118
5.5.4 Emulated Overloading . . . 119
5.5.5 Defensive Programming: Dense Typing Style . . . 121
5.5.6 Simple Inductive Types and Record Types . . . 123
5.5.7 Structured Expression Typing . . . 124
5.5.8 Grammar Representations . . . 125

5.6 The General Problem of Type Extensibility **129**
5.6.1 Features and Criteria of Type Extensibility . . . 129
5.6.2 Record Type Extensibility . . . 130
5.6.3 Inductive Type Extensibility . . . 131
5.6.4 The Inductive Types and the Explicit Labeled Types . . . 133
5.6.5 Flexible Inductive Types: Polymorphic Variants . . . 135

6 Functional Data Structures 141

6.1 Lists **142**
6.1.1 A Sequential Data Structure Representation . . . 142
6.1.2 Lists : A Predefined Structured Type . . . 144
6.1.3 Some Functions on Lists . . . 145

6.2 Programming Techniques on Functional Structures. I. **148**
6.2.1 Iterators . . . 148
6.2.2 Empty Structures and Partial Functions . . . 149
6.2.3 Associative Tables and Structural Constraints . . . 151

6.3 Trees **152**
6.3.1 Binary Trees . . . 152
6.3.2 Trees : an Example of Composed Data Structure . . . 157

6.3.3 Multiple Compositions . . . 160

6.4 Persistance **160**
6.4.1 A Property of Functional Programming . . . 160
6.4.2 The Advantages of Persistance . . . 163
6.4.3 The Disadvantages of Persistance . . . 165

6.5 Programming Techniques on Functional Structures. II. **166**
6.5.1 Taming the Disadvantages of Persistance . . . 166
6.5.2 Controlling Navigation within Data Structures . . . 170
6.5.3 Tail Recursivity on Lists . . . 173
6.5.4 Tail Recursivity on Trees . . . 174

6.6 Functional Data Structures **176**
6.6.1 The Limits of Inductive Types . . . 176
6.6.2 Ambiguity and Inductive Types . . . 178
6.6.3 Homogeneous and Heterogeneous Data Structures . . . 183
6.6.4 Graphs and Non Functional Data Structures . . . 186

7 Imperative Programming 193

7.1 Exceptions **194**
7.1.1 Partial Functions and Error Handling . . . 194
7.1.2 A Basic Type: the Exceptions . . . 197
7.1.3 Raising Exceptions . . . 197
7.1.4 Catching and Handling Exceptions . . . 199
7.1.5 Exceptions as an Optimisation Tool . . . 200
7.1.6 Exceptions as Values . . . 202

7.2 Procedures and Unit **203**
7.2.1 Procedures Are Not Functions . . . 203
7.2.2 Sequences . . . 205
7.2.3 A Simple `if-then` Construction . . . 206

7.3 Using Procedures **206**
7.3.1 Primitive Debugging Techniques . . . 206
7.3.2 Batch Evaluations . . . 207
7.3.3 Basic Graphics . . . 209
7.3.4 Assertions . . . 210
7.3.5 File Handling or OCaml as a Script Language . . . 211

7.4 Arrays : A Predefined Type **214**
7.4.1 Direct Access and Value Assignment . . . 214
7.4.2 Obtaining New Arrays . . . 215
7.4.3 Internal Iterators on Arrays . . . 216
7.4.4 Loops . . . 216
7.4.5 Strings: An Imperative Type . . . 218

7.5 Records with Mutable Fields **219**

7.6 References **219**
7.6.1 Mutable Elements . . . 219
7.6.2 References: A Basic Type . . . 220
7.6.3 A Typing Problem: The Generic References . . . 222
7.6.4 Imperative Operations and Generic References . . . 223
7.6.5 Using Closures and Generic References . . . 224
7.6.6 Imperative Data Structures and Generic References . . . 224
7.6.7 Non-Generalization and Weak Type Variables . . . 226
7.6.8 Non-Generalization and Function Applications . . . 229
7.6.9 Imperative Data Structures and Weak Type Variables . . . 231

7.7 Choosing to Use the Imperative Style **232**
7.7.1 Several Problems of the Imperative Style . . . 232
7.7.2 The Propagation of the Imperative Style . . . 235
7.7.3 Interfacing OCaml with C . . . 237

8 Functional Programming Techniques **239**

8.1 Incremental Programming **240**
8.1.1 An Example: Analyzing Symbol Sequences . . . 241
8.1.2 A First Prototype . . . 242
8.1.3 More Elaborated Tests . . . 245
8.1.4 A Graphical Representation . . . 248
8.1.5 The Advantages of Incremental Programming . . . 249

8.2 Generalizations **251**
8.2.1 The Idea of Generalizion . . . 251
8.2.2 Transformation Generalizations: Foldings . . . 252
8.2.3 Summing up Generalizations . . . 257
8.2.4 Algorithm Generalizations . . . 258
8.2.5 Type Generalizations . . . 263
8.2.6 Polytypism . . . 265
8.2.7 Generalizations and Reuse . . . 267

8.3 Data-Driven Programming **272**
8.3.1 A Generalized Generalization . . . 272
8.3.2 A Global Programming Technique . . . 275
8.3.3 Imperative Data-Driven Programming . . . 276
8.3.4 Type Extensibility and Data-Driven Programming . . . 278

8.4 Functional Data Representations **279**
8.4.1 Sets as Functions . . . 280
8.4.2 Sub-Spaces as Functions . . . 283
8.4.3 Graphic Objects as Functions . . . 285
8.4.4 Real Numbers as Functions . . . 288

8.5 Evaluation Control **289**
8.5.1 Basics of Evaluation Control . . . 290

8.5.2 Functional Freezing: A New Use of `unit` . . . 290
8.5.3 Freezing and Lazyness . . . 292
8.5.4 The Behavior of Functional Freezing . . . 295
8.5.5 Data Structures with a Lazy Behavior . . . 297
8.5.6 Freezing Evaluation Flow and Tail Recursivity . . . 298

8.6 Lazy Lists or Streams **301**
8.6.1 Freezing a Sequential Data Structure . . . 301
8.6.2 Accessing the Elements of a Stream . . . 303
8.6.3 Variations about Stream Definitions . . . 304
8.6.4 Recursivity and Frozen Infinity . . . 305
8.6.5 Iterators and Stream Compositions . . . 307
8.6.6 Using Frozen Infinity . . . 309
8.6.7 Streams and Lazyness Boundary . . . 311
8.6.8 Generating Values by Streams . . . 313
8.6.9 External Iterators as Streams . . . 314
8.6.10 Infinite Sets as Streams . . . 316
8.6.11 Streams: A Global Programming Technique . . . 318

8.7 Lazy Trees **319**
8.7.1 Freezing a Tree-like Structure . . . 319
8.7.2 Using Lazy Trees . . . 323
8.7.3 Choosing Explicit Lazyness . . . 327

8.8 Continuation Passing Programming **328**
8.8.1 Continuations . . . 328
8.8.2 Continuations Passing and Partial Functions . . . 329
8.8.3 Continuations Passing and Evaluation Control . . . 331
8.8.4 Red-Black Trees . . . 333
8.8.5 2-3 Trees . . . 335

9 Modular Programming 341

9.1 Modules as a General Encapsulating Mean **342**
9.1.1 Module Definitions . . . 342
9.1.2 Accessing the Elements of a Module . . . 343
9.1.3 Data Structures in a Modular Form . . . 344
9.1.4 Remarks about Module Definitions . . . 346
9.1.5 Local Modules . . . 348

9.2 Signatures: Types for Modules **349**
9.2.1 Inferred Signatures . . . 349
9.2.2 Signatures Definitions . . . 350
9.2.3 Abstract Data Types . . . 351
9.2.4 Implemented Elements in Signatures . . . 352
9.2.5 Signatures to Type Modules . . . 353
9.2.6 Modules are Instances of Signatures . . . 356

9.3 Signatures Writing Techniques **358**
9.3.1 Signatures as Specification and Interface Means . . . 358
9.3.2 The Limits of Signature Specifications . . . 358
9.3.3 Overcoming the Limits of Signature Specifications . . . 359
9.3.4 Signature Generalizations . . . 360
9.3.5 Signatures for Functional and Imperative Styles . . . 361

9.4 Public and Private Elements of Modules **362**
9.4.1 Public Elements . . . 362
9.4.2 Private Elements . . . 363
9.4.3 Signature Compatibility . . . 364
9.4.4 Public Exceptions . . . 365
9.4.5 Public Types . . . 366
9.4.6 Private Types and Abstract Data Types . . . 367

9.5 Abstract Data Types **369**
9.5.1 The Reasons of Information Hiding and Substitutivity . . . 369
9.5.2 To Adapt Data Types . . . 370
9.5.3 To Optimize Data Types by Memoization . . . 372
9.5.4 Information Hiding and Generic Predefined Predicates . . . 375
9.5.5 Information Hiding and Optimization by Imperative Features . . . 376
9.5.6 "Public" Abstract Types . . . 379
9.5.7 "Public" Abstract Types and Type Constraints . . . 382
9.5.8 Conceptual Signature Generalizations . . . 384
9.5.9 Naming Tactics for Abstract Data Types . . . 386
9.5.10 Universal Type Parameters and Abstract Types . . . 388

9.6 Module Linking by Inclusions **392**
9.6.1 Module and Signature Inclusions . . . 392
9.6.2 Inclusions and Inheritance . . . 393
9.6.3 Inclusions and Adaptations . . . 395

9.7 Module Linking by Sub-Components **396**
9.7.1 Sub-Modules . . . 396
9.7.2 Module Openings . . . 397
9.7.3 Abstract Sub-Modules . . . 399
9.7.4 Signature Linking by Abstract Sub-Modules . . . 399
9.7.5 Abstract Sub-Modules and Constraints Inclusions . . . 401
9.7.6 Hiding Types of Sub-Modules . . . 402
9.7.7 Flattening Modular Levels . . . 404
9.7.8 Modular Aggregates of Data Types . . . 404
9.7.9 Modular Associations of Data Types . . . 406
9.7.10 A More Complete Example: The "Apple Men" . . . 409

9.8 Information Hiding in Question **411**
9.8.1 Choosing to Hide . . . 411
9.8.2 The "Record-Modules" . . . 413
9.8.3 The *a posteriori* Information Hiding . . . 415
9.8.4 Information Hiding and Module Associations . . . 416

9.8.5 The Limits of Type Constraints . . . 417
9.8.6 The Extensibility of Value Representations . . . 418

9.9 Modules as External Files and Batch Compilation 420
9.9.1 Loading Modules in the Interaction Loop . . . 420
9.9.2 Module Batch Compilation . . . 422
9.9.3 Compiled Modules and the Interaction Loop . . . 424
9.9.4 Compiled Modules and Side-Effects . . . 425
9.9.5 Super-Modules as Packages . . . 426

10 Generic Modular Programming: Functors 431

10.1 Functors 432
10.1.1 The Idea of Module Functions . . . 432
10.1.2 Functor Definitions . . . 433
10.1.3 Using Functor Parameters . . . 434
10.1.4 Applying Functors . . . 435
10.1.5 Explicit Typing and Functors . . . 436
10.1.6 Sharing Types and Type Transmission . . . 437
10.1.7 Non-Modular Parameters . . . 438
10.1.8 Functors : a Typed Generic Modular Programming . . . 439

10.2 Functor Programming Techniques and Basic Design Patterns 440
10.2.1 Generic Modular Inheritance . . . 440
10.2.2 Generic Modular Adaptations . . . 442
10.2.3 Generic Modular Decorations . . . 443
10.2.4 Generic Modular Composites . . . 445
10.2.5 Generic Data Types . . . 447

10.3 Functors and the Genericity Power 448
10.3.1 Signatures and Universal Type Parameters . . . 448
10.3.2 Signatures and Abstract Types . . . 449
10.3.3 Going from one Genericity to Another? . . . 451

10.4 Functors and Batch Compilation 453

10.5 n-ary Functors 456
10.5.1 n-ary Functors Definitions . . . 456
10.5.2 Applying n-ary Functors . . . 457
10.5.3 A More Complete Example: Generic Graphs . . . 461
10.5.4 Type Constraints over Functors Parameters . . . 463
10.5.5 Sharing Types between Functors Parameters . . . 467
10.5.6 A Problem: Sharing Types vs. Information Hiding . . . 469
10.5.7 The "Data Type Nursery" Technique . . . 470

10.6 Functor Signatures 472
10.6.1 Types for Functors . . . 472
10.6.2 Specification and Functor Signatures . . . 475

10.7 Functors of Functors: Higher-Order Functors **476**
10.7.1 A First Example of Higher-Order Functors . . . 476
10.7.2 Type Constraints over Functor Parameters . . . 477
10.7.3 Tuning Genericity by Higher-Order Functors . . . 477
10.7.4 Remarks about Functor Generalizations . . . 480
10.7.5 Modules and Functors Are Almost First Class Citizen . . . 481

10.8 Choosing Generic Modular Programming **482**
10.8.1 Functors: Another Functional Programming . . . 483
10.8.2 Functors: A Tool for Software Engineering . . . 483
10.8.3 Generic Modular Programming in Question . . . 487

10.9 A Generic Architecture Representation **489**
10.9.1 Software Development and Generic Programming . . . 489
10.9.2 Specification Graphs and Architecture Automata . . . 491
10.9.3 Representing Signature Compatibilities . . . 494

10.10 Examples of Architecture Automata **497**
10.10.1 Generic Priority Queues . . . 497
10.10.2 Signature Hierarchies Management . . . 500
10.10.3 Generic Production Lines . . . 501
10.10.4 Remarks about Architecture Automata . . . 505

11 Object-Oriented Programming 507

11.1 Why Do One Need a New Kind of Values? **507**
11.1.1 Objects as Records . . . 508
11.1.2 Objects as Modules . . . 509
11.1.3 Objects as Functions . . . 511

11.2 Objects and Classes **513**
11.2.1 Objects Definitions . . . 513
11.2.2 Accessing Object Elements . . . 515
11.2.3 The Object Self . . . 516
11.2.4 Object Types . . . 516
11.2.5 Object Type Compatibility . . . 518
11.2.6 Open Types . . . 520
11.2.7 Classes as Object Constructors . . . 521
11.2.8 Open Types and Constraint Genericity . . . 525

11.3 Some Class Mechanisms **525**
11.3.1 Inheritance . . . 525
11.3.2 Generic Classes . . . 526
11.3.3 Constraint Generic Classes . . . 529
11.3.4 Redefinitions and Dynamic Links . . . 529
11.3.5 Class Types . . . 531
11.3.6 Abstract Methods . . . 533

11.4 Some Object Oriented Techniques **534**

11.4.1 Object and Functional Programming 534
11.4.2 Using Object Type Compatibility 535
11.4.3 Typing n-ary Methods 540
11.4.4 Abstract Data Types as Classes 543
11.4.5 Object and Modular Programming 545
11.4.6 Object and Generic Modular Programming 546

11.5 Choosing Object Programming **550**
11.5.1 The Main Advantages of Object Oriented Programming 550
11.5.2 Object Oriented Programming and Recursive Components 550
11.5.3 The Extensibility and Object Oriented Programming 552
11.5.4 The Main Drawbacks of Object Oriented Programming 553