Exploring Languages with Interpreters and Functional Programming

H. Conrad Cunningham

27 April 2022

Contents

0 Preface **2**
- 0.1 Dedication . . . 2
  - 0.1.1 July 2018 . . . 2
  - 0.1.2 November 2019 Addendum . . . 2
- 0.2 Course 1 and Course 2 . . . 2
- 0.3 Motivation: "Functional Programming" . . . 3
- 0.4 Motivation: "Exploring Languages with Interpreters" . . . 5
- 0.5 Textbook Prerequisites . . . 5
- 0.6 Author's Perspective . . . 6
- 0.7 The Journey . . . 7
  - 0.7.1 Notes on Functional Programming with Haskell . . . 7
  - 0.7.2 Organization of Programming Languages . . . 9
  - 0.7.3 Exploring Languages with Interpreters and Functional Programming . . . 11

1 Evolution of Programming Languages **14**
- 1.1 Chapter Introduction . . . 14
- 1.2 Evolving Computer Hardware Affects Programming Languages . 14
- 1.3 History of Programming Languages . . . 17
- 1.4 What Next? . . . 22
- 1.5 Exercises . . . 22
- 1.6 Acknowledgements . . . 23
- 1.7 Terms and Concepts . . . 23

2 Programming Paradigms **24**
- 2.1 Chapter Introduction . . . 24
- 2.2 Abstraction . . . 24
  - 2.2.1 What is abstraction? . . . 24
  - 2.2.2 Kinds of abstraction . . . 24
  - 2.2.3 Procedures and functions . . . 25

2.3 What is a Programming Paradigm? 26
2.4 Imperative Paradigm 26
2.4.1 Java 26
2.4.2 Other languages 27
2.5 Declarative Paradigm 27
2.5.1 Functional paradigm 28
2.5.1.1 Haskell 28
2.5.1.2 Other languages 30
2.5.2 Relational (or logic) paradigm 30
2.5.2.1 Prolog 30
2.5.2.2 Other languages 31
2.6 Other Programming Paradigms 31
2.6.1 Procedural paradigm 32
2.6.1.1 Python 32
2.6.1.2 Other languages 33
2.6.2 Modular paradigm 33
2.6.2.1 Python 33
2.6.2.2 Other languages 38
2.6.3 Object-based paradigms 38
2.6.4 Concurrent paradigms 38
2.7 Motivating Functional Programming: John Backus 38
2.7.1 Excerpts from Backus's Turing Award Address [6] 39
2.7.2 Aside on the disorderly world of statements 41
2.7.3 Perspective from four decades later 41
2.8 What Next? 42
2.9 Exercises 42
2.10 Acknowledgements 42
2.11 Terms and Concepts 43

3 Object-Based Paradigms **45**
3.1 Chapter Introduction 45
3.2 Motivation 45
3.3 Object Model 46
3.3.1 Objects 46
3.3.1.1 Essential characteristics 46
3.3.1.2 Important but non-essential characteristics . . . 47
3.3.2 Classes 48
3.3.3 Inheritance 49
3.3.4 Subtype polymorphism 53
3.4 Object-Oriented Paradigm 55
3.4.1 Object-oriented Python example 56
3.4.2 Object-oriented Scala example 58
3.5 Prototype-based Paradigm 58
3.5.1 Prototype concepts 58
3.5.2 Lua as an object-based language 59
3.5.3 Prototype-based Lua example 61

3.5.4 Observations 64
3.6 What Next? 64
3.7 Exercises 65
3.8 Acknowledgements 65
3.9 Terms and Concepts 66

4 First Haskell Programs 68
4.1 Chapter Introduction 68
4.2 Defining Our First Haskell Functions 68
4.2.1 Factorial function specification 68
4.2.2 Factorial function using if-then-else: `fact1` 69
4.2.3 Factorial function using guards: `fact2` 71
4.2.4 Factorial function using pattern matching: `fact3` and `fact4` 71
4.2.5 Factorial function using built-in library function: `fact5` . 72
4.2.6 Testing 73
4.3 Using the Glasgow Haskell Compiler (GHC) 73
4.4 What Next? 75
4.5 Chapter Source Code 75
4.6 Exercises 75
4.7 Acknowledgements 76
4.8 Terms and Concepts 76

5 Types 77
5.1 Chapter Introduction 77
5.2 Type System Concepts 77
5.2.1 Types and subtypes 77
5.2.2 Constants, variables, and expressions 77
5.2.3 Static and dynamic 78
5.2.4 Nominal and structural 78
5.2.5 Polymorphic operations 79
5.2.6 Polymorphic variables 80
5.3 Basic Haskell Types 80
5.3.1 Integers: `Int` and `Integer` 81
5.3.2 Floating point numbers: `Float` and `Double` 82
5.3.3 Booleans: `Bool` 82
5.3.4 Characters: `Char` 83
5.3.5 Functions: `t1 -> t2` 83
5.3.6 Tuples: `(t1,t2,...,tn)` 84
5.3.7 Lists: `[t]` 85
5.3.8 Strings: `String` 85
5.3.9 Advanced Types 85
5.4 What Next? 85
5.5 Exercises 86
5.6 Acknowledgements 88
5.7 Terms and Concepts 89

6 Procedural Abstraction 90
6.1 Chapter Introduction . . . 90
6.2 Procedural Abstraction Review . . . 90
6.3 Top-Down Stepwise Refinement . . . 90
6.3.1 Developing a square root package . . . 90
6.3.2 Making the package a Haskell module . . . 92
6.3.3 Reviewing top-down stepwise refinement . . . 93
6.4 Modular Design and Programming . . . 94
6.4.1 Information-hiding modules and secrets . . . 95
Secret of square root module . . . 95
6.4.2 Contracts: Preconditions and postconditions . . . 95
Contracts of square root module . . . 96
Contracts of `Factorial` module . . . 96
6.4.3 Interfaces for modules . . . 97
Interface of square root module . . . 97
6.4.4 Abstract interfaces for modules . . . 98
Abstract interface of square root module . . . 98
6.4.5 Client-supplier relationship . . . 98
6.4.6 Design criteria for interfaces . . . 99
6.5 What Next? . . . 101
6.6 Chapter Source Code . . . 101
6.7 Exercises . . . 101
6.8 Acknowledgements . . . 101
6.9 Terms and Concepts . . . 102

7 Data Abstraction 103
7.1 Chapter Introduction . . . 103
7.2 Data Abstraction Review . . . 103
7.3 Using Data Abstraction . . . 103
7.3.1 Rational number arithmetic . . . 103
7.3.2 Rational number data representation . . . 105
7.3.3 Rational number modularization . . . 108
7.3.3.1 Module `RationalCore` . . . 108
7.3.3.2 Module `Rational` . . . 109
7.3.3.3 Modularization critique . . . 109
7.3.4 Alternative data representation . . . 109
7.3.5 Haskell information-hiding modules . . . 111
7.3.6 Rational number testing . . . 113
7.4 Module invariants . . . 113
7.4.1 RationalRep modules . . . 114
7.4.1.1 `RationalCore` . . . 115
7.4.1.2 `RationalDeferGCD` . . . 115
7.4.2 Rational modules . . . 116
7.5 What Next? . . . 116
7.6 Chapter Source Code . . . 116
7.7 Exercises . . . 117

7.8 Acknowledgements . . . 119
7.9 Terms and Concepts . . . 120

8 Evaluation Model **121**
8.1 Chapter Introduction . . . 121
8.2 Referential Transparency Revisited . . . 121
8.3 Substitution Model . . . 122
8.4 Time and Space Complexity . . . 126
8.5 Termination . . . 126
8.6 What Next? . . . 127
8.7 Exercises . . . 127
8.8 Acknowledgements . . . 128
8.9 Terms and Concepts . . . 128

9 Recursion Styles and Efficiency **129**
9.1 Chapter Introduction . . . 129
9.2 Linear and Nonlinear Recursion . . . 129
9.2.1 Linear recursion . . . 129
9.2.2 Nonlinear recursion . . . 130
9.3 Backward and Forward Recursion . . . 131
9.3.1 Backward recursion . . . 131
9.3.2 Forward recursion . . . 131
9.3.3 Tail recursion . . . 132
9.4 Logarithmic Recursion . . . 134
9.5 Local Definitions . . . 135
9.6 Using Other Languages . . . 137
9.6.1 Scheme . . . 137
9.6.2 Elixir . . . 138
9.6.3 Scala . . . 140
9.6.4 Lua . . . 141
9.6.5 Elm . . . 142
9.7 What Next? . . . 143
9.8 Chapter Source Code . . . 143
9.9 Exercises . . . 144
9.10 Acknowledgements . . . 146
9.11 Terms and Concepts . . . 146

10 Simple Input and Output (FUTURE) **147**
10.1 Chapter Introduction . . . 147
10.2 What Next? . . . 147
10.3 Exercises . . . 147
10.4 Acknowledgements . . . 147
10.5 Terms and Concepts . . . 147

11 Software Testing Concepts **148**
11.1 Chapter Introduction . . . 148

11.2 Software Requirements Specification 148
11.3 What is Software Testing? 149
11.4 Goals of Testing 149
11.5 Dimensions of Testing 149
11.5.1 Testing levels 150
11.5.2 Testing methods 152
11.5.2.1 Black-box testing 152
11.5.2.2 White-box testing 154
11.5.2.3 Gray-box testing 155
11.5.2.4 Ad hoc testing 155
11.5.3 Testing types 155
11.5.4 Combining levels, methods, and types 156
11.6 Aside: Test-Driven Development 156
11.7 Principles for Test Automation 157
11.8 What Next? 161
11.9 Exercises 161
11.10Acknowledgements 161
11.11Terms and Concepts 161

12 Testing Haskell Programs **162**
12.1 Chapter Introduction 162
12.2 Organizing Tests 162
12.3 Testing Functions 162
12.3.1 Factorial example 162
12.3.2 Arrange 163
12.3.3 Act 164
12.3.4 Assert 164
12.3.5 Aggregating into test script 164
12.4 Testing Modules 166
12.4.1 Rational arithmetic modules example 166
12.4.2 Data representation modules 166
12.4.2.1 Arrange 167
12.4.2.2 Act 169
12.4.2.3 Assert 169
12.4.2.4 Aggregate into test script 170
12.4.2.5 Broken encapsulation 171
12.4.3 Rational arithmetic modules 171
12.4.3.1 Arrange 171
12.4.3.2 Act 172
12.4.3.3 Assert 172
12.4.3.4 Aggregate into test script 172
12.4.4 Reflection on this example 172
12.5 What Next? 173
12.6 Chapter Source Code 173
12.7 Exercises 174
12.8 Acknowledgements 174

12.9 Terms and Concepts 174

13 List Programming 176
13.1 Chapter Introduction 176
13.2 Polymorphic List Data Type 176
13.2.1 List: `[t]` 176
13.2.2 String: `String` 178
13.2.3 Polymorphic lists 180
13.3 Programming with List Patterns 181
13.3.1 Summing a list of integers: `sum'` 181
13.3.2 Multiplying a list of numbers: `product'` 183
13.3.3 Length of a list: `length'` 184
13.3.4 Remove duplicate elements: `remdups` 184
13.3.5 More list patterns 186
13.4 Data Sharing 186
13.4.1 Preconditions for `head` and `tail` 187
13.4.2 Dropping elements from beginning of list 188
13.4.3 Taking elements from the beginning of a list 189
13.5 What Next? 189
13.6 Chapter Source Code 189
13.7 Exercises 189
13.8 Acknowledgements 190
13.9 Terms and Concepts 191

14 Infix Operators and List Examples 192
14.1 Chapter Introduction 192
14.2 Using Infix Operations 192
14.2.1 Appending two lists: `++` 193
14.2.2 Properties of operations 195
14.2.3 Element selection: `!!` 195
14.2.4 Reversing a list: `rev` 196
14.2.5 Tail recursive `reverse` 197
14.3 More Useful List Functions 198
14.3.1 Another list-breaking function: `splitAt` 198
14.3.2 List-combining operations: `zip` and `unzip` 198
14.4 Insertion Sort 199
14.5 What Next? 200
14.6 Chapter Source Code 200
14.7 Exercises 200
14.8 Acknowledgements 208
14.9 Terms and Concepts 208

15 Higher-Order Functions 209
15.1 Chapter Introduction 209
15.2 Generalizing Procedural Abstractions 209
15.3 Defining `map` 210

15.4 Thinking about Data Transformations 211
15.5 Generalizing Function Definitions 212
15.6 Defining `filter` 212
15.7 Defining Fold Right (`foldr`) 214
15.8 Using `foldr` 217
15.9 Defining Fold Left (`foldl`) 218
15.10Using `foldl` 219
15.11Defining `concatMap` (flatmap) 220
15.12What Next? 221
15.13Chapter Source Code 221
15.14Exercises 221
15.15Acknowledgemen4 222
15.16Terms and Concepts 223

16 Haskell Function Concepts 224
16.1 Chapter Introduction 224
16.2 Strictness 224
16.3 Currying and Partial Application 225
16.4 Operator Sections 226
16.5 Combinators 227
16.6 Functional Composition 229
16.7 Function Pipelines 229
16.8 Lambda Expressions 231
16.9 Application Operator `$` 232
16.10Eager Evaluation Using `seq` and `$!` 233
16.11What Next? 234
16.12Chapter Source Code 234
16.13Exercises 235
16.14Acknowledgements 235
16.15Terms and Concepts 236

17 Higher Order Function Examples 237
17.1 Chapter Introduction 237
17.2 List-Breaking Operations 237
17.3 List-Combining operations 238
17.4 Rational Arithmetic Revisited 239
17.5 Mergesort 239
17.6 Divide-and-Conquer Algorithms 241
17.6.1 General strategy 241
17.6.2 As higher-order function 241
17.6.3 Generating Fibonacci sequence 242
17.6.4 Folding a list 243
17.6.5 Finding minimum and maximum of a list 244
17.7 What Next? 245
17.8 Chapter Source Code 245
17.9 Exercises 246

17.10 Wally World Marketplace POP Project 247
17.10.1 Problem description and initial design 247
17.10.2 Prelude functions useful for project 250
17.10.3 POP project exercises 251
17.11 Acknowledgements 253
17.12 Terms and Concepts 254

18 More List Processing 255
18.1 Chapter Introduction 255
18.2 Sequences 255
18.3 List Comprehensions 256
18.3.1 Syntax and semantics 256
18.3.2 Translating list comprehensions 257
18.4 Using List Comprehensions 259
18.4.1 Strings of spaces 259
18.4.2 Prime number test 259
18.4.3 Squares of primes 259
18.4.4 Doubling positive elements 260
18.4.5 Concatenating a list of lists of lists 260
18.4.6 First occurrence in a list 260
18.5 What Next? 261
18.6 Chapter Source Code 261
18.7 Exercises 261
18.8 Acknowledgements 262
18.9 Terms and Concepts 262

19 Systematic Generalization 263
19.1 Chapter Introduction 263
19.2 SCV Analysis 263
19.3 Function Generalization 263
19.4 Developing a Cosequential Processing Family 264
19.4.1 Scope 264
19.4.2 Frozen spots 265
19.4.3 Hot spots 265
19.4.4 Hot spot #1: Variability in total ordering 266
19.4.5 Hot spot #2: Variability in record format 266
19.4.6 Hot spot #3: Independent variability of sequences 267
19.4.7 Hot spot #4: Variability in sequence transformations . . 269
19.4.8 Hot spot #5 :Variability of sequence source/destination . 272
19.4.9 Bag and set operation implementations 272
19.4.10 Sequential file update algorithm 274
19.4.11 Recap 275
19.5 What Next? 275
19.6 Chapter Source Code 276
19.7 Exercises 276
19.8 Acknowledgements 276

19.9 Terms and Concepts . . . 276

20 Problem Solving **277**
20.1 Chapter Introduction . . . 277
20.2 Problem Solving Philosophy . . . 277
20.3 Polya's Insights . . . 277
20.4 Problem-Solving Strategies . . . 278
20.4.1 Solve a more general problem first . . . 278
Examples . . . 279
20.4.2 Solve a simpler problem first . . . 279
Examples . . . 280
20.4.3 Reuse off-the-shelf solutions to standard subproblems . . . 280
Examples . . . 280
20.4.4 Solve a related problem . . . 281
Examples . . . 281
20.4.5 Separate concerns . . . 281
Examples . . . 281
20.4.6 Divide and conquer . . . 282
Examples . . . 282
20.5 What Next? . . . 282
20.6 Chapter Source Code . . . 282
20.7 Exercises . . . 282
20.8 Acknowledgements . . . 283
20.9 Terms and Concepts . . . 283

21 Algebraic Data Types **284**
21.1 Chapter Introduction . . . 284
21.2 Concepts . . . 284
Aside: ADT confusion . . . 284
21.3 Haskell Algebraic Data Types . . . 285
21.3.1 Declaring data types . . . 285
21.3.2 Declaring type `Color` . . . 285
21.3.3 Deriving class instances . . . 286
21.3.4 Exploring more example types . . . 287
21.4 Recursive Data Types . . . 288
21.4.1 Defining a binary tree type . . . 288
21.4.2 Exporting types from modules . . . 289
21.4.3 Defining an alternative binary tree type . . . 290
21.5 Error-handling with `Maybe` and `Either` . . . 291
21.5.1 Handling null references . . . 291
21.5.2 Introducing `Maybe` and `Either` . . . 292
21.5.3 Considering other languages . . . 293
21.6 What Next? . . . 294
21.7 Chapter Source Code . . . 294
21.8 Exercises . . . 294
21.9 Carrie's Candy Bowl Project . . . 298

21.9.1 Problem description and initial design 298
21.9.2 Carrie's Candy Bowl project exercises 299
21.9.3 Candy Bowl alternative exercises 301
21.10 Sandwich DSL Project 301
21.10.1 Project Introduction 301
21.10.2 Developing the Sandwich DSL 301
21.10.3 Sandwich DSL exercise set A 304
21.10.4 Compiling the program for the SueChef controller 305
21.10.5 Sandwich DSL exercise set B 306
21.10.6 Sandwich DSL source code 307
21.11 Exam DSL Project 307
21.11.1 Project introduction 307
21.11.2 Developing the Exam DSL 307
21.11.3 Exam DSL exercise set A 310
21.11.4 Outputting the Exam as HTML 311
21.11.5 Exam DSL project exercise set B 314
21.11.6 Exam DSL source code 314
21.12 Acknowledgements 314
21.13 Terms and Concepts 315

22 Data Abstraction Revisited 317
22.1 Chapter Introduction 317
22.2 Concepts 317
22.3 Example: Doubly Labelled Digraph 317
22.4 Use Case 318
22.5 Defining ADTs 319
22.5.1 Specification 319
22.5.2 Operations 319
22.5.3 Approaches to semantics 320
22.6 Specification of Labelled Digraph ADT 320
22.6.1 Notation 321
22.6.2 Sets 321
22.6.3 Signatures 321
Constructors 322
Mutators 322
Accessors 322
Destructors 322
22.6.4 Semantics 323
22.6.4.1 Interface invariant 323
22.6.4.2 Constructive semantics 323
22.6.5 Haskell module abstract interface 327
22.7 List Implementation 328
22.7.1 Labelled digraph representation 328
22.7.2 Implementation invariant 329
22.7.3 Haskell implementation 329
22.7.4 Improvements to the list implementation 331

22.8 Map Implementation . . . 332
22.8.1 Labelled digraph representation . . . 332
22.8.2 Implementation invariant . . . 333
22.8.3 Haskell module . . . 334
22.8.4 Improvements to the map implementation . . . 335
22.9 What Next? . . . 335
22.10 Chapter Source Code . . . 336
22.11 Exercises . . . 336
22.12 Mealy Machine Simulator Project . . . 337
22.12.1 Project introduction . . . 337
22.12.2 Mealy Machine Simulator project exercises . . . 337
22.13 Acknowledgements . . . 339
22.14 Terms and Concepts . . . 339

23 Overloading and Type Classes **341**
23.1 Chapter Introduction . . . 341
23.2 Polymorphism in Haskell . . . 341
23.3 Why Overloading? . . . 341
23.4 Defining an Equality Class and Its Instances . . . 342
23.5 Type Class Laws . . . 344
23.6 Another Example Class `Visible` . . . 344
23.7 Class Extension (Inheritance) . . . 345
23.8 Multiple Constraints . . . 346
23.9 Built-In Haskell Classes . . . 347
23.10 Comparison to Other Languages . . . 347
23.11 What Next? . . . 348
23.12 Chapter Source Code . . . 348
23.13 Exercises . . . 349
23.14 Acknowledgements . . . 349
23.15 Terms and Concepts . . . 349

24 Future Chapter TBD **351**
24.1 Chapter Introduction . . . 351
24.2 What Next? . . . 351
24.3 Exercises . . . 351
24.4 Acknowledgements . . . 351
24.5 Terms and Concepts . . . 351

25 Proving Haskell Laws **352**
25.1 Chapter Introduction . . . 352
25.2 Referential Transparency Revisited . . . 352
25.3 Stating and Proving Laws . . . 352
25.3.1 Example: `++` associativity and identity element . . . 352
25.3.2 Structural induction proof method . . . 353
25.3.3 Proving associativity of `++` . . . 354
25.3.4 Reviewing proof method . . . 356

25.3.5 Proving identity element for ++ 357
25.4 Example: Relating length and ++ 358
25.5 Example: Relating take and drop 359
25.6 Example: Equivalence of Functions 360
25.7 What Next? 363
25.8 Exercises 363
25.9 Acknowledgements 366
25.10 Terms and Concepts 366

26 Program Synthesis **367**
26.1 Chapter Introduction 367
26.2 Motivation 367
26.3 Fast Fibonacci Function 367
26.4 Sequence of Fibonacci Numbers 369
26.5 Synthesis of drop from take 373
26.6 Tail Recursion Theorem 375
26.7 Finding Better Tail Recursive Algorithms 378
26.8 What Next? 381
26.9 Exercises 381
26.10 Acknowledgements 382
26.11 Terms and Concepts 383

27 Text Processing Example **384**
27.1 Chapter Introduction 384
27.2 Text Processing Example 384
27.2.1 Word processing 389
27.2.2 Paragraph processing 390
27.2.3 Other text processing functions 391
27.3 What Next? 392
27.4 Exercises 392
27.5 Acknowledgements 392
27.6 Terms and Concepts 392

28 Type Inference **393**
28.1 Chapter Introduction 393
28.2 Motivation 393
28.3 Example: Functional Composition 393
28.4 Example: Multiple Use of Polymorphic Function (fst) 394
28.5 Example: Fixpoint (fix) 396
28.6 Example: Incorrect Typing (selfapply) 397
28.7 Other Aspects of Type Inference 398
28.8 What Next? 398
28.9 Exercises 398
28.10 Acknowledgements 398
28.11 Terms and Concepts 399

29 Models of Reduction **400**
29.1 Chapter Introduction 400
29.2 Big-O and Efficiency 400
29.3 Reduction 401
29.3.1 Definition 401
29.3.2 Redexes 401
29.3.3 AOR and NOR 401
29.3.4 Concepts related to AOR and NOR 404
29.3.5 String and graph reduction 406
29.4 Head Normal Form 409
29.5 Pattern Matching 410
29.6 Reduction Order and Space 411
29.7 Choosing a Fold 416
29.8 What Next? 418
29.9 Exercises 418
29.10 Acknowledgements 418
29.11 Terms and Concepts 418

30 Infinite Data Structures **419**
30.1 Chapter Introduction 419
30.2 Infinite Lists 419
30.3 Iterate 420
30.4 Prime Numbers: Sieve of Eratosthenes 421
30.5 Circular Structures 423
30.6 What Next? 424
30.7 Chapter Source Code 424
30.8 Exercises 424
30.9 Acknowledgements 424
30.10 Terms and Concepts 424

31 Future Chapter **425**

32 Future Chapter **425**

33 Future Chapter **425**

34 Future Chapter **425**

35 Future Chapter **425**

36 Future Chapter **425**

37 Future Chapter **425**

38 Future Chapter **425**

39 Future Chapter **425**

40 Language Processing **426**
40.1 Chapter Introduction 426
40.2 Compiler Phases 426
40.3 What Next? 426
40.4 Chapter Source Code 426
40.5 Exercises 426
40.6 Acknowledgements 426
40.7 References 426
40.8 Terms and Concepts 426

41 Calculator: Concrete Syntax **428**
41.1 Chapter Introduction 428
41.2 Concrete Syntax 428
41.3 Grammars 429
41.3.1 Context-free grammars and BNF 429
41.3.2 Derivations 430
41.3.3 Regular grammars 431
41.4 Infix syntax 431
41.5 Prefix syntax 434
41.6 What Next? 436
41.7 Chapter Source Code 436
41.8 Exercises 436
41.9 Acknowledgements 436
41.10 Terms and Concepts 439

42 Calculator: Abstract Syntax and Evaluation **440**
42.1 Chapter Introduction 440
42.2 Abstract Syntax 440
42.2.1 Abstract syntax tree data type 440
42.2.2 Values and variable names 444
42.3 Associative Data Structures 444
42.4 Semantics 445
42.4.1 Environments 445
42.4.2 Values of AST nodes 446
42.4.3 Evaluation function 447
42.5 Simplification 449
42.6 Symbolic Differentiation 449
42.7 What Next? 450
42.8 Chapter Source Code 450
42.9 Exercises 451
42.10 Acknowledgements 453
42.11 Terms and Concepts 454

43 Calculator: Modular Structure **455**
43.1 Chapter Introduction 455
43.2 Module Dependencies 455

43.3 Values Module 455
43.4 Environments Module 456
43.5 Abstract Syntax Module 458
43.6 Evaluator Module 458
43.7 Lexical Analysis Module 459
43.8 Parser Modules 460
43.9 REPL Modules 461
43.10 Code Improvement Modules 462
43.11 What Next? 462
43.12 Chapter Source Code 462
43.13 Exercises 463
43.14 Acknowledgements 463
43.15 Terms and Concepts 463

44 Calculator: Parsing 464
44.1 Chapter Introduction 464
44.2 Parsing 464
44.3 Lexical Analysis 465
44.3.1 Prefix syntax 466
44.3.2 Infix syntax 468
44.4 Recursive Descent Parsing 469
44.4.1 Constructing recursive descent parsers 469
44.4.2 Prefix syntax 472
44.4.2.1 Parse `<expression>` 473
44.4.2.2 Parse `<var>` 474
44.4.2.3 Parse `<val>` 475
44.4.2.4 Parse `<operexpr>` 475
44.4.2.5 Parse `<operandseq>` 476
44.4.2.6 AST construction (`makeExpr`) 477
44.4.3 Infix syntax 478
44.5 Exercises 479
44.6 Chapter Source Code 479
44.7 Acknowledgements 479
44.8 Terms and Concepts 479

45 Parsing Combinators 480
45.1 Chapter Introduction 480
45.2 Developing Parsing Combinators 480
45.2.1 State actions and combinators 480
45.2.2 Completing a combinator library 481
45.2.3 Adding parse tree generations 483
45.3 Standard libraries for parsing 483
45.4 Exercises 483
45.5 Chapter Source Code 483
45.6 Acknowledgements 483
45.7 Terms and Concepts 484

46 Calculator: Compilation **485**
46.1 Chapter Introduction . . . 485
46.2 Stack Virtual Machine . . . 485
46.2.1 Instruction set syntax . . . 485
46.2.2 Instruction set semantics . . . 485
46.2.3 Machine execution . . . 486
46.2.4 Compilation . . . 486
46.3 What Next? . . . 487
46.4 Chapter Source Code . . . 487
46.5 Exercise Set A . . . 487
46.6 Conditional Expressions . . . 488
46.6.1 Extending the Calculator language . . . 488
46.6.2 Extending the stack virtual machine (UNFINISHED) . . . 488
46.6.3 Extending the compiler (UNFINISHED) . . . 489
46.7 Exercise Set B (UNFINISHED) . . . 490
46.8 Acknowledgements . . . 490
46.9 Terms and Concepts . . . 490

47 Imperative Core Language (Future) **491**
47.1 Chapter Introduction . . . 491
47.2 What Next? . . . 491
47.3 Exercises . . . 491
47.4 Acknowledgements . . . 491
47.5 Terms and Concepts . . . 491

48 Appendix I: Review of Relevant Mathematics **492**
48.1 Chapter Introduction . . . 492
48.2 Natural Numbers and Ordering . . . 492
48.3 Functions . . . 493
48.4 Recursive Functions . . . 493
48.5 Mathematical Induction Natural Numbers . . . 494
48.6 Operations . . . 495
48.7 Algebraic Structures . . . 497
48.8 Exercises . . . 498
48.9 Acknowledgements . . . 498
48.10Terms and Concepts . . . 498

References **499**

Copyright (C) 2022, H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

Instability Warning: This version of ELIFP is in work beginning in January 2022. The author may change its structure and content without warning. No changes are planned to the 2018 version upon which the this version is based. The stable version is on the Fall 2018 CSci 450 course website.

Feedback Request: The author plans to publish this textbook eventually. He invites anyone using this book to give him feedback on its current structure and content: to point out typos and other errors or suggest improvements and extensions. He can be contacted at hcc AT cs DOT olemiss DOT edu.

0 Preface

0.1 Dedication

0.1.1 July 2018

I dedicate this textbook to my parents—my mother, Mary Cunningham, and my father, the late Harold "Sonny" Cunningham—and to my wife, Diana Cunningham.

I thank Mother and Dad for their love and encouragement throughout my nearly 64 years on this earth. They taught me the importance of hard work, both physical and mental. They taught me the importance of faith in God and of personal integrity. I hope I have been a good student.

I write this sitting at a desk in my mother's home late one evening in July 2018. I remember a time more than a half century ago when I was struggling with an elementary school writing assignment. Mother wrote an example page that I remember as amazing. I thank her for that encouragement. I still suffer from a deficit of creativity at times, but I was able to write this approximately 400-page textbook.

I look around and see a plaque for an award my father received for serving his church as a Sunday School teacher for 40 years. It reminds me of the many positive contributions he made to his community and his church, many unseen by others. I hope I am also making positive contributions to the various communities, physical and virtual, in which I live and work.

I thank Diana, my wife of 42 years, for her love and friendship—for being my companion on the journey of life. This textbook is an effort that has spanned more than a quarter century. She has lived it nearly as much as I have. Many times she has urged me to stop work and get some sleep, as she did just now.

0.1.2 November 2019 Addendum

My mother passed away in June 2019 at the age of 91 years and 8 months. We miss her dearly! Her family and friends will remember for as long as we live.

We love you, Mom, and look forward to the reunion of our family in heaven.

0.2 Course 1 and Course 2

As the title suggests, I designed this textbook to be used for at least two different kinds of courses:

1. A course on "functional programming" targeted at advanced undergraduate and beginning graduate students who have previously programmed using imperative languages but who have not used functional or relational languages extensively.

This *functional and modular programming* course focuses on parts of Chapter 2 and 80 and Chapters 4-30.

I have been teaching such an elective course at the University of Mississippi since 1991 (CSci 555, Functional Programming). I have been teaching the Haskell programming language since 1993. Some of the content of this textbook evolved from class notes I originally developed for the course in the 1991-6 period.

My approach to the course was initially motivated by the first edition of the classic Bird and Wadler textbook [13,15].

2. A course on programming language organization targeted at a similar audience.

   There are several approaches to teaching the programming languages course. My approach in this textbook focuses on "exploring languages with interpreters". It seeks to guide students to learn how programming languages work by developing interpreters for simple languages.

   This programming language organization course focuses on Chapters 1-3, Chapters 40-49, and parts of Chapters 4-30 as needed.

   Kamin's excellent textbook *Programming Languages: An Interpreter-Based Approach* [108] motivated my approach. But, instead of using Pascal or C to develop the interpreters as Kamin's book did, this textbook primarily uses Haskell. Other influences on my approach are the book by Sestoft [159,160], the online books by Krishnamurthi [114,115], and an early manuscript by Ramsey [148] (which is based on Kamin's book).

   I began experimenting with this approach using the Lua language in my graduate Software Language Engineering (CSci 658) course in Fall 2013. I first taught the interpreter approach (using Lua) in the required undergraduate Organization of Programming Languages (CSci 450) course at the University of Mississippi in 2016. I used Haskell with the interpreter-based approach in 2017 and 2018.

Of course, students must become familiar with basic functional programming and Haskell for Course 2 to be possible.

Most real courses will likely be a mix of the two approaches.

0.3 Motivation: "Functional Programming"

Course type 1 is a course on functional and modular programming.

As a course on *programming*, Course 1 emphasizes the analysis and solution of problems, the development of correct and efficient algorithms and data structures that embody the solutions, and the expression of the algorithms and data structures in a form suitable for processing by a computer. The focus is more on the human thought processes than on the computer execution processes.

As a course on *functional* programming, Course 1 approaches programming as the construction of definitions for (mathematical) functions and (immutable) data structures. Functional programs consist of *expressions* that use these definitions. The execution of a functional program entails the evaluation of the expressions making up the program. Thus this course's focus is on problem solving techniques, algorithms, data structures, and programming notations appropriate for the functional approach.

As a course on *modular* programming, Course 1 approaches the construction of large programs as sets of modules that collaborate to solve a problem. Each module is a coherent collection of function and data type definitions. A module hides its private features, allowing their use only within the module, and exposes its public features, enabling their use by the other modules in the program.

Course 1 is not a course on functional or modular programming *languages*. In particular, it does not undertake an in-depth study of the techniques for implementing such languages on computers. (That is partly covered in Course 2.) The focus is on the concepts for programming, not on the internal details of the technological artifact that executes the programs.

Of course, we want to be able to execute our programs on a computer and, moreover, to execute them efficiently. Thus we must become familiar with some concrete programming language and use an implementation of that language to execute our programs. To be able to analyze program efficiency, we must also become familiar with the basic techniques that are used to evaluate expressions.

The academic community has long been interested in functional programming. In recent years, the practitioner community has also become interested in functional programming techniques and language features. There is growing use of languages that are either primarily functional or have significant functional subsets—such as Haskell, OCaml, Scala, Clojure, F#, Erlang, and Elixir. Most mainstream languages have been extended with new functional programming features and libraries—for example, Java, C#, Python, JavaScript, and Swift. Other interesting research languages such as Elm and Idris are also generating considerable interest.

In this textbook, we use the Haskell 2010 language. Haskell is a "lazy" functional language whose development began in the late 1980's. We also use a set of programming tools based on GHC, the Glasgow Haskell Compiler. GHC is distributed in a "batteries included" bundle called the the Haskell Platform. (That is, it bundles GHC with commonly used libraries and tools.)

Most of the concepts, techniques, and skills learned in this Haskell-based course can be applied in other functional and multiparadigm languages and libraries.

More importantly, any time we learn new approaches to problem solving and programming, we become better programmers in whatever language we are working. A course on functional programming provides a novel, interesting, and, probably at times, frustrating opportunity to learn more about the nature of

the programming task.

Enjoy the "functional programming" aspects of the course and textbook!

0.4 Motivation: "Exploring Languages with Interpreters"

Course type 2 is a course on programming language organization that emphasizes design and implementation of a sequence of interpreters for simple languages.

When we first approach a new programming language, we typically think about the *syntax* of the language—how the external (e.g., textual) representation of the language is structured.

Syntax is important, but the *semantics* is more important. The semantics defines what the language means: how it "behaves" at "runtime".

In Course 2 we primarily focus on the semantics. We express the essential aspects of a expression's structure (i.e., syntax) with an *abstract syntax tree (AST)* and then process the AST to obtain a result. For example, we may have:

- an *interpreter* that takes an AST and *evaluates* it in some environment to obtain its value
- a *transformer* that takes the AST and produces a related but different (e.g., more efficient) program in the same language
- a *compiler* that takes an AST and produces a related program in a different language

By "exploring languages with interpreters", we can better understand the semantics of the programming languages. We can learn to use languages more effectively. We can explore alternative designs and implementations for languages.

This textbook uses functional and modular programming in Haskell—a paradigm and language that most students in Course 2 do not know—to implement the interpreters. Students learn new language concepts by both learning Haskell and by building language processors.

0.5 Textbook Prerequisites

This textbook assumes the reader has basic knowledge and skills in programming, algorithms, and data structures at least at the level of a three-semester introductory computer science sequence. It assumes that the reader has programming experience using a language such as Java, C++, Python, or C#; it does not assume any previous experience in functional programming. (For example, successful completion of at least CSci 211, Computer Science III, at the University of Mississippi should be sufficient.)

This textbook also assumes the reader has basic knowledge and understanding of introductory computer architecture from a programmer's perspective. (For

example, successful completion of at least CSci 223, Computer Organization and Assembly Language, at the University of Mississippi should be sufficient.)

In addition, this course assumes the reader has basic knowledge and skills in mathematics at the level of a college-level course in discrete mathematical structures for computer science students. (For example, concurrent enrollment in Math 301, Discrete Mathematics, at the University of Mississippi should suffice.) The "Review of Relevant Mathematics" chapter (appendix) reviews some of the concepts, terminology, and notation used in this course.

0.6 Author's Perspective

In the 1974 Turing Award Lecture *Computer Programming as an Art*, Donald Knuth said [111]:

> The chief goal of my work as an educator and author is to help people learn to write *beautiful programs*.

In my writing and my teaching, I hope I can emulate Knuth.

I approach writing this textbook, most of my teaching and research for that matter, from the following (opinionated) perspectives:

- The essence of computing science is programming. Programming is fun!

  Of course, by "programming" I do not mean merely "coding"—and definitely not "hacking".

  I view programming as the process: of determining what the problem is, whether a solution is needed, what the desired nature of a solution is, and whether such a solution is feasible, ethical, and socially useful; of devising specific abstractions, algorithms, and information structures that correctly, elegantly, and efficiently solve the problem; of implementing the solution effectively within the concrete resources available and validating that it indeed solves the problem; and of evolving the solution and its implementation to handle changing needs. This, of course, encompasses most of what is traditionally called computer science and software engineering.

- Abstraction is the primary tool we have to deal with the complexity we must face as programmers.

  To become good programmers, we need to learn to develop good abstractions. To learn to develop good abstractions, most of us need to work upward from lots of concrete examples and experiences.

  Perhaps instead of computer science our field should be called abstraction engineering.

- Our programs should be elegant—both conceptually in terms of their design (architecture, algorithms, data structures, use of appropriate abstraction)

and physically in terms of their style (use of language features, layout, use of names, appropriate comments).

- We should rigorously describe what a program must do. For example, we can define a rigorous contract that specifies what the clients of a program must do and what the program must do in response.
- We should construct larger programs as sets of collaborating modules. The modules should be designed and constructed according to the information-hiding and abstract interface principles.
- We should design our programs to be testable and test them thoroughly.
- We should reflect upon what we have done. What about our successes and failures can we observe and exploit in the future? Did our specific problem reveal or reinforce a general principle? What can we do better next time?
- To learn a programming paradigm and language well, we should immerse ourselves in the paradigm and language for a period ot time. We need to learn to think in that paradigm and language even if it is quite different from our previous experiences.
- Although we learn to program in a particular language and paradigm, we should seek to compare how the new concepts, features, and patterns of thought apply to other approaches to programming that we have learned in the past or will in the future.
- Many tasks can be viewed as language design or language processing tasks. Language design and processing are fun!
- As much as feasible, we should make instructional materials accessible (e.g., compatible with screen readers) and available in multiple formats at a low cost.

  Toward that end, I have developed most of my new instructional materials using Pandoc's dialect of Markdown and tools compatible with Pandoc.

0.7 The Journey

Although I only began to write this textbook in Summer 2016, it is a result of a journey I began long ago. Many other writers, colleagues, students, and friends have helped me during this journey.

0.7.1 Notes on Functional Programming with Haskell

I created the course CSci 555, Functional Programming, at the University of Mississippi and first taught it during the Spring 1991 semester.

I adopted the first edition of Bird and Wadler [15] as the initial textbook for the course. I thank Richard Bird and Philip Wadler for writing this excellent

textbook. I thank Jeremy Gibbons for suggesting that book in a response to an inquiry I posted to a Usenet newsgroup in Summer 1990.

I also used Wentworth's RUFL (Rhodes University Functional Language) interpreter and his tutorial [178] in the first two offerings of the course. I thank Peter Wentworth for sending me (unsolicited, in response to my Usenet post) his interpreter and tutorial on a floppy disk through snail mail from the then-sanctioned South Africa.

My approach was also shaped by my research on formal methods and my previous teaching on that topic. I created the course Program Semantics and Derivation (CSci 550) and first taught it in Spring 1990 [40,41]. I followed that with the course Theory of Concurrent Programming (Engr 664), which I first taught in Fall 1990. I thank my dissertation advisor Gruia-Catalin Roman for developing my interests in formal methods, Jan Tijmen Udding for teaching a graduate course on program derivation that piqued my interests, and the other researchers or authors who have influenced my thinking: Edsger Dijkstra, Tony Hoare, David Gries, Mani Chandy, Jayadev Misra, Edward Cohen, and many others.

For the third offering of CSci 555 in Fall 1993, I switched the course to use the Gofer interpreter for the Haskell language. I thank the international committee of researchers, including Simon Peyton Jones, Paul Hudak, Philip Wadler, and others, who have developed and sustained Haskell since the late 1980s. I also thank Mark Jones for developing the lightweight Gofer interpreter and making it and its successor HUGS widely available.

Because of the need for a tutorial like Wentworth's and an unexpected delay in getting copies of the Bird and Wadler textbook [15] from Prentice Hall that semester, I began writing, on an emergency basis, what evolved into my *Notes on Functional Programming with Haskell* [42].

Some parts of the *Notes* were based on my handwritten class notes from the the 1991 and 1992 offerings of the course. Many pages of the *Notes* were written "just-in-time" in late-night sessions before I taught them the next day. I thank Prentice Hall (now Pearson) for its delay in shipping books across the "big pond", my wife Diana Cunningham for tolerating my disruptive schedule, and my Fall 1993 students for not complaining too vigorously about a quite raw set of class notes.

I continued to develop the *Notes* for the Fall 1994 and Fall 1995 offerings of the course. In early 1996, I created a relatively stable version of the *Notes* that I continued to use in subsequent offerings of CSci 555. I thank my students and others who pointed out typos and suggested improvements during the 1993-1996 period.

I thank David Gries for encouraging me to expand these notes into a textbook. I am doing that, albeit over 20 years later than Gries intended.

I formatted the *Notes* using LaTeX augmented by BibTeX for the bibliography and makeIndex for the index. I thank Donald Knuth, Leslie Lamport, and the

many others who have developed and maintained TeX, LaTeX, and the other tools and packages over four decades. They form an excellent system for creating beautiful scientific documents.

I used GNU Emacs for writing and editing the source files for the *Notes*. I thank Richard Stallman and the many others who developed, maintained, and popularized Emacs over more than four decades.

For the Spring 1997 offering of CSci 555, I started using the new HUGS interpreter and the 1st edition of Thompson's textbook [171] (now it its 3rd edition [173]). I thank Simon Thompson for writing his excellent, comprehensive introductory textbook on Haskell programming.

Over the next 17 years, I corrected a few errors but otherwise the *Notes* were stable. However, I did create supplementary notes for CSci 555 and related courses. These drew on the works of Abelson and Sussman [1], Thompson [171–173], Parnas [20,134], and others. I formated these notes with HTML, Microsoft Word and Powerpoint, or plain text.

I decided to use Haskell as one of the languages in the Fall 2014 offering of Organization of Programming Languages (CSci 450). But I needed to change the language usage from the Haskell 98 standard and HUGS to the new Haskell 2010 standard and the Glasgow Haskell Compiler (GHC) and its interactive user interface GHCi. I edited the *Notes* through chapter 10 on Problem Solving to reflect the changes in Haskell 2010.

0.7.2 Organization of Programming Languages

ACM Curriculum '78 [5] influenced how most computer science academic programs were structured when they are established in the 1970s and 1980s. It defined eight core courses, most of which are still prominent in contemporary computer science curricula.

Organization of Programming Languages (CS 8) is one of those core courses. Curriculum '78 describes CS 8 as "an applied course in programming language constructs emphasizing the run-time behavior of programs" and providing "appropriate background for advanced level courses involving formal and theoretical aspects of programming languages and/or the compilation process" [5].

I first taught the required Organization of Programming Languages (CSci 450) course at the University of Mississippi in Fall 1995. I took over that class for another instructor and used the textbook the Department Chair had already selected. The textbook was the 2nd Edition of Sebesta's book [157].

Although the Sebesta book, now in its 11th edition, is probably one of the better and more popular books for CS 8-type courses, I found it difficult for me to use that semester. It and its primary competitors seem to be large, expensive tomes that try to be all things to all instructors and students. I personally find the kind of survey course these books support to be a disjointed hodgepodge. There

is much more material than I can cover well in one semester. I abandoned the Sebesta book mid-way through the semester and have never wanted to use it again.

I had a copy of Kamin's textbook [108] and used two of its interpreters after abandoning Sebesta's book. It seemed to work better than Sebesta. So I ended the semester with a positive view of the Kamin approach.

My only involvement with CSci 450 for the next 18 years was to schedule and staff the course (as Department Chair 2001-15). In 2013, I began planning to teach CSci 450 again.

I decided to try an experiment. I planned to use the Kamin approach for part of the course but to redevelop the interpreters in the Lua language. Lua is a minimalist, dynamically typed language that can support multiple paradigms.

I chose Lua because (a) learning it would be a new experience for almost all of my students, (b) as a small language it should be easy for students to learn, (c) its flexibility would enable me to explore how to extend the language itself to provide new features, (d) its associated LPEG library supported the development of simple parsers, and (e) its use in computer games might make it interesting to students. I thank the Lua authors—Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo—for developing this interesting platform and making it available. I thank Ierusalimschy for developing LPEG and for writing an excellent textbook on Lua programming [105].

I used the Fall 2013 offering of my Software Language Engineering (CSci 658) course to explore Lua programming and interpreters. I thank the students in that class for their feedback and suggestions—Michael Macias, Blake Adams, Cornelius Hughes, Zhendong Zhao, Joey Carlisle, and others.

However, in Summer 2014, I did not believe I was ready to undertake the interpreter-based approach in the large Fall 2014 class. Instead, I planned to try a multiple paradigm survey. I planned to begin with Haskell statically typed functional programming (using my *Notes*), then cover Lua dynamically typed, multiparadigm programming, and then use the logic language Prolog. I had taught Haskell, Lua, and Prolog in elective courses in the past.

I was comfortable with the Haskell part, but I found a required course a more challenging environment in which to teach Haskell than an elective. Covering Haskell took nearly two-thirds of the semester, leaving Lua in one-third, and squeezing out coverage of the logic language and most of the interpreter material.

I was scheduled to teach CSci 450 again in Fall 2016. For this offering, I decided to (a) begin with Lua and then follow with Haskell (the reverse order from 2014) and (b) to use the interpreter approach in the Lua segment. I adopted the 4th edition of Scott's textbook [156] to support the general material (but did not use the book much).

Unfortunately, that offering suffered from immature teaching materials for both

Lua and for the interpreter approach. I was unable to invest sufficient time in Summer 2016 to prepare course materials and revise the interpreters. Also, students, who mostly had experience with Java, had considerable difficulty modifying and debugging the dynamically typed Lua programs with 1000+ lines of code. (For various reasons, I decided to use the new Elm language instead of Haskell in the last three weeks of the semester.)

I thank the students in the Fall 2014 and Fall 2016 CSci 450 classes for giving me valuable feedback on what works and what does not—much more on the latter than the former. I also thank my Teaching Assistant (TA) for CSci 450 in the Fall 20a6 semester, Ajay Sharma, for his assistance. I learned that a large, required course like CSci 450 needs more mature teaching materials and tools than a small, elective course does. It should have been obvious!

0.7.3 Exploring Languages with Interpreters and Functional Programming

In Summer 2016, I participated in the eLearning Training Course (eTC) at the University of Mississippi to become eligible to teach online. As a part of that course, I was expected to prepare a syllabus and at least one module for some class. I chose to focus on CSci 555, Functional Programming.

This stimulated me to begin reorganizing my previous *Notes on Functional Programming with Haskell* to be a textbook for an online course on functional programming. I thank the eTC instructors Patty O'Sullivan and Wan Latartara, for (unintentionally) pushing me to begin developing this textbook.

For the textbook, I expanded the *Notes* by adapting materials I had originally developed for other purposes—such as papers with former graduate students Pallavi (Tadepalli) Darbhamulla, Yi Liu, and Cuihua Zhang—and some notes from my courses on functional programming, multiparadigm programming, software architecture, and software language engineering. I thank Darbhamulla, Liu, and Zhang. I also thank former graduate student James Church (author of a Haskell-based book [32]) for his feedback and encouragement to repackage my class notes as a textbook.

Unfortunately, I devoted too much time to this project in Summer 2016 and not enough to developing Lua-based materials and tools for the Fall 2016 offering of CSci 450, as I discussed above.

The eTC also sensitized me to the need to produce accessible instructional materials (e.g., materials compatible with screen readers for the visually impaired). I decided to expand my use of Pandoc-flavored Markdown and the Pandoc tools for producing materials in a variety of accessible formats (HTML, possibly LaTeX/PDF).

In Summer 2016, I had materials in a variety of formats. The *Notes on Functional Programming with Haskell* used LaTeX, BibTeX, and makeIndex. This is a great format for producing printed scientific documents, but not as good for display on

the Web. Some of my other materials used HTML, which is great for the Web, but not for printed documents. I also had some material in Microsoft Office formats, Pandoc-flavored Markdown, and plain text (e.g., program comments).

Pandoc-flavored Markdown offered a means for achieving both greater flexibility and greater accessibility. Of course, sometimes I have to compromise on the appearance in some formats.

The Pandoc tool uses a language-processing approach, is implemented in Haskell, and supports Lua as its builtin scripting language. So it is a good fit for this textbook project. I thank John MacFarlane and many others who have developed and maintained the excellent Pandoc tools.

In Spring and Summer 2017, I combined the efforts from the previous years and sought to expand the Haskell-based functional programming course materials to include materials for the interpreter-based approach to the programming languages course and new Haskell-related material on type classes.

I redirected the work from developing materials for an online course to developing a textbook for the types of courses I describe in the "Course 1 and Course 2" section above.

In Fall 2017, I taught CSci 450 from the 2017 version of the textbook. Given my more mature materials, it worked better than the Lua-based course the previous year. But that effort identified the need for additional work on the textbook: shorter, more focused chapters, some explicit discussion of software testing, more attention to the language-processing issues, etc.

I thank my Fall 2017 and Fall 2018 TA for CSci 450, Kyle Moore, for his suggestions and corrections in a number of areas. I also thank the Spring 2017 Multiparadigm Programming (CSci 556) and Fall 2017 CSci 450 students for their feedback on the textbook materials.

I thank my long-time colleague, and current Department Chair, Dawn Wilkins, for her general suggestions on the CSci 450 course and the textbook and for the Department's continued support of my textbook writing efforts.

I also thank Armando Suarez, my Spring 2018 TA for Senior Project and student in Software Language Engineering that semester, for his suggestions on my materials and approach to those courses—some of which I have applied to this textbook and its associated courses.

In 2018, I began restructuring the 2017 version of the textbook to better meet the needs of the CSci 450 course. I changed the title to *Exploring Languages with Interpreters and Functional Programming*.

I incorporated additional chapters from the *Notes on Functional Programming with Haskell* and other materials that had not been previously included. I also developed new chapters on software testing, the language processing pipeline, and the Imperative Core language interpreter. I plan to develop additional interpreters, such as one for a Scheme-like language.

Because of this project, I have come to appreciate how much time, effort, and attention to detail must be invested to develop a good programming language organization textbook. I think Samuel Kamin, Robert Sebesta, Michael Scott, Norman Ramsey, Shriram Krishnamurthi, and other authors for their investment in writing and updating their books.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I plan to continue work on this textbook. However, the work went slowly until 2022 because of the COVID-19 pandemic disruptions, my continued work with two PhD students until mid-2021, and various personal factors. In January 2022, I began refining the existing content, integrating separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc features.

As I have noted above, I maintain this preface as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed. I continue to learn how better to apply the Pandoc-related tools to accomplish this.

1 Evolution of Programming Languages

1.1 Chapter Introduction

The goal of this chapter is motivate the study of programming language organization by:

- describing the evolution of computers since the 1940's and its impact upon contemporary programming language design and implementation
- identifying key higher-level programming languages that have emerged since the early 1950's

1.2 Evolving Computer Hardware Affects Programming Languages

To put our study in perspective, let's examine the effect of computing hardware evolution on programming languages by considering a series of questions.

1. *When were the first "modern" computers developed? That is, programmable electronic computers.*

   Although the mathematical roots of computing go back more than a thousand years, it is only with the invention of the programmable electronic digital computer during the World War II era of the 1930s and 1940s that modern computing began to take shape.

   One of the first computers was the ENIAC (Electronic Numerical Integrator and Computer), developed in the mid-1940s at the University of Pennsylvania. When construction was completed in 1946, it cost about $500,000. In today's terms, that is nearly $7,000,000.

   The ENIAC weighed 30 tons, occupied as much space as a small house, and consumed 160 kilowatts of electric power.

   Initially, the ENIAC had no main memory. Instead it had 20 accumulators, each 10 decimal digits wide. Later 100 *words* of core were added.

   Similarly, the ENIAC had no external memory as we know it today. It could read and write stacks of punch cards.

   The ENIAC was not a stored program computer. It was programmed mostly by connecting cables in plugboards. It took several days of careful work to enter one program. The program was only changed every few weeks.

   Aside: Many of the early programmers were women. This is quite a contrast to contemporary programming teams that are mostly male. What happened?

   The ENIAC and most other computers of that era were designed for military purposes, such as calculating firing tables for artillery or breaking

codes. As a result, many observers viewed the market for such devices to be quite small. The observers were wrong!

Electronics technology has improved greatly in 70 years. Today, a computer with the capacity of the ENIAC would be smaller than a coin from our pockets, would consume little power, and cost just a few dollars on the mass market.

2. *How have computer systems and their use evolved over the past 70 years?*

    - Contemporary processors are much smaller and faster. They use much less power, cost much less money (when mass produced), and operate much more reliably.

    - Contemporary "main" memories are much larger in capacity, smaller in physical size, and faster in access speed. They also use much less power, cost much less money, and operate much more reliably.

    - The number of processors per machine has increased from one to many. First, channels and other co-processors were added, then multiple CPUs. Today, computer chips for common desktop and mobile applications have several processors—cores—on each chip, plus specialized processors such as graphics processing units (GPUs) for data manipulation and parallel computation. This trend toward multiprocessors will likely continue given that physics dictates limits on how small and fast we can make computer processors; to continue to increase in power means increasing parallelism.

    - Contemporary external storage devices are much larger in capacity, smaller in size, faster in access time, and cost less.

    - The number of computers available per user has increased from much less than one to many more than one.

    - Early systems were often locked into rooms, with few or no direct connections to the external world and just a few kinds of input/output devices. Contemporary systems may be on the user's desktop or in the user's backpack, be connected to the internet, and have many kinds of input/output devices.

    - The range of applications has increased from a few specialized applications (e.g., code-breaking, artillery firing tables) to almost all human activities.

    - The cost of the human staff to program, operate, and support computer systems has probably increased somewhat (in constant dollars).

3. *How have these changes affected programming practice?*

    - In the early days of computing, computers were very expensive and the cost of the human workers to use them relatively less. Today, the opposite holds. So we need to maximize human productivity.

- In the early days of computing, the slow processor speeds and small memory sizes meant that programmers had to control these precious resources to be able to carry out most routine computations. Although we still need to use efficient algorithms and data structures and use good coding practices, programmers can now bring large amounts of computing capacity to bear on most problems. We can use more computing resources to improve productivity to program development and maintenance. The size of the problems we can solve computationally has increased beyond what would be possible manually.
- In the early days of computing, multiple applications and users usually had to share one computer. Today, we can often apply many processors for each user and application if needed. Increasingly, applications must be able to use multiple processors effectively.
- Security on early systems meant keeping the computers in locked rooms and restricting physical access to those rooms. In contemporary networked systems with diverse applications, security has become a much more difficult issue with many aspects.
- Currently, industry can devote considerable hardware and software resources to the development of production software.

The first higher-level programming languages began to appear in the 1950s. IBM released the first compiler for a programming language in 1957–for the scientific programming language Fortran. Although Fortran has evolved considerably during the past 60 years, it is still in use today.

4. *How have the above changes affected programming language design and implementation over the past 60 years?*

   - Contemporary programming languages often use automatic memory allocation and deallocation (e.g., garbage collection) to manage a program's memory. Although programs in these languages may use more memory and processor cycles than hand-optimized programs, they can increase programmer productivity and the security and reliability of the programs. Think Java, C#, and Python versus C and C++.
   - Contemporary programming languages are often implemented using an interpreter instead of a compiler that translates the program to the processor's machine code–or be implemented using a compiler to a virtual machine instruction set (which is itself interpreted on the host processor). Again they use more processor and memory resources to increase programmer productivity and the security and reliability of the programs. Think Java, C#, and Python versus C and C++.
   - Contemporary programming languages should make the capabilities of contemporary multicore systems conveniently and safely available

to programs and applications. To fail to do so limits the performance and scalability of the application. Think Erlang, Scala, and Clojure versus C, C++, and Java.

- Contemporary programming languages increasingly incorporate declarative features (higher-order functions, recursion, immutable data structures, generators, etc.). These features offer the potential of increasing programming productivity, increasing the security and reliability of programs, and more conveniently and safely providing access to multicore processor capabilities. Think Scala, Clojure, and Java 8 and beyond versus C, C++, and older Java.

As we study programming and programming languages in this and other courses, we need to keep the nature of the contemporary programming scene in mind.

1.3 History of Programming Languages

From the instructor's perspective, key languages and milestones in the history of programming languages include the following.

Note: These descriptions use terminology such as imperative and function that is defined in Chapters 2 and 3 on programming paradigms.

1950's

- Fortran, 1957; imperative; first compiler, math-like language for scientific programming, developed at IBM by John Backus, influenced most subsequent languages, enhanced versions still in use today (first programming language learned by the author in 1974)
- Lisp [121,122,145,182], 1958; mix of imperative and functional features; innovations include being *homoiconic* (i.e., code and data have same format), extensive use of recursion, syntactic macros, automatic storage management, higher-order functions; related to Church's lambda calculus theory, developed at MIT by John McCarthy, influenced most subsequent languages/research, enhanced versions still in use today
- Algol, 1958, 1960; imperative; innovations included nested block structure, lexical scoping, use of BNF to define syntax, call-by-name parameter passing; developed by an international team from Europe and the USA, influenced most subsequent languages
- COBOL, 1959; imperative; focus on business/accounting programming, decimal arithmetic, record data structures, key designer Grace Hopper, still in use today (third language learned by instructor in late 1975)

1960's

- Simula; 1962, 1967; imperative; original purpose for discrete-event simulation, developed in Norway by Ole-Johan Dahl and Kristen Nygaard, Simula 67 is first object-oriented language (in Scandinavian school of

object-oriented languages), Simula 67 influenced subsequent object-oriented languages

- Snobol, 1962; imperative; string processing, patterns as first-class data, backtracking on failure, developed at AT&T Bell Laboratories by David J. Farber, Ralph E. Griswold and Ivan P. Polonsky
- PL/I, 1964; imperative; IBM-designed language to merge scientific (Fortran), business (COBOL), and systems programming (second language learned by the instructor in early 1975)
- BASIC, 1964; imperative; simple language developed for interactive computing in early timesharing and microcomputer environments, developed at Dartmouth College by John G. Kemeny and Thomas E. Kurtz
- Algol 68, 1968; imperative; ambitious and rigorously defined successor to Algol 60; designed by international team, greatly influenced computing science theory and subsequent language designs, but not widely or fully implemented because of its complexity

1970's

- Pascal, 1970; imperative; simplified Algol family language designed by Niklaus Wirth (Switzerland) because of frustration with complexity of Algol 68, structured programming, one-pass compiler, important for teaching in 1980s and 1990s, Pascal-P System virtual machine implemented on many early microcomputers (Pascal used by UM CIS in CS1 and CS2 until 1999)
- Prolog [33,163], 1972; logic (relational); first and most widely used logic programming language, originally developed by a team headed by Alain Colmerauer (France), rooted in first-order logic, most modern Prolog implementations based on the Edinburgh dialect (which ran on the Warren Abstract Machine), used extensively for artificial intelligence research in Europe, influenced subsequent logic languages and also Erlang
- C, 1972; imperative; systems programming language for Unix operating system, widely used today; developed by Dennis Ritchie at AT&T Bell Labs, influenced many subsequent languages (first used by the author in 1977)
- Smalltalk [84], 1972; imperative object-oriented; ground-up object-oriented programming language, message-passing between objects (in American school of object-oriented languages), extensive GUI development environment; developed by Alan Kay and others at Xerox PARC, influenced many subsequent object-oriented languages and user interface approches
- ML, 1973; mostly functional; polymorphic type system on top of Lisp-like language, pioneering statically typed functional programming, algebraic data types, module system; developed by Robin Milner at the University of Edinburgh as the "meta language" for a theorem-proving system, influenced

subsequent functional programming languages, modern dialects include Standard ML (SML), CAML, and OCAML

- Scheme [81,82,183], 1975; mixed functional and imperative; minimalist dialect of Lisp with lexical scoping, tail call optimization, first-class continuations; developed by Guy Steele and Gerald Jay Sussman at MIT, influenced subsequent languages/research
- Icon, 1977; imperative; structured programming successor to Snobol, uses goal-directed execution based on success or failure of expressions; developed by a team led by Ralph Griswold at the University of Arizona

1980's

- C++, 1980; imperative and object-oriented; C with Simula-like classes; developed by Bjarne Stroustrup (Denmark)
- Ada, 1983; imperative and modular; designed by US DoD-funded committee as standard language for military applications, design led by Jean Ichbiah and a team in France, statically typed, block structured, modular, synchronous message passing, object-oriented extensions in 1995 (instructor studied this language while working in the military aerospace industry 1980-83)
- Eiffel, 1985; imperative object-oriented language; designed with strong emphasis on software engineering concepts such as design by contract and command-query separation; developed by Bertrand Meyer (France)
- Objective C, 1986; imperative object-oriented; C with Smalltalk-like messaging; developed by Brad Cox and Tom Love at Stepstone, selected by Steve Jobs' NeXT systems, picked up by Apple when NeXT absorbed, key language for MacOS and iOS
- Erlang, 1986; functional and concurrent; message-passing concurrency on functional programming base (actors), fault-tolerant/real-time systems, dynamic typing, virtual machine, originally used in real-time telephone switches; developed by Joe Armstrong, Robert Virding, and Mike Williams at Ericsson (Sweden)
- Self [158,175], 1986; imperative prototype-based; dialect of Smalltalk, first prototype-based language, used virtual machine with just-in-time compilation (JIT); developed by David Ungar and Randall Smith while at Xerox PARC, Stanford University, and Sun Microsystems, language influenced JavaScript and Lua, JIT influenced Java HotSpot JIT development
- Perl, 1987; imperative; dynamic programming language originally focused on providing powerful text-processing facilities based around regular expressions; developed by Larry Wall

1990's

- Haskell [120,173,179], 1990; purely functional language; non-strict semantics (i.e., lazy evaluation) and strong static typing; developed by an international committee of functional programming researchers, widely used in research community
- Python [[144];] [146]], 1991; imperative, originally object-based; dynamically typed, multiparadigm language; developed by Guido van Rossum (Netherlands)
- Ruby [149,169], 1993; imperative, object-oriented; dynamically typed, supports reflective/metaprogramming and internal domain-specific languages; developed by Yukihiro "Matz" Matsumoto (Japan), popularized by Ruby on Rails web framework, influenced subsequent languages
- Lua [105,116], 1993; imperative; minimalistic language designed for embedding in any environment supporting standard C, dynamic typing, lexical scoping, first-class functions, garbage collection, tail recursion optimization, pervasive table/metatable data structure, facilities for prototype object-oriented programming, coroutines, used as scripting language in games; developed by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes (Brazil)
- R [167,181], 1993; imperative; designed for statistical computing and graphics, open-source implementation of the language S; developed by Ross Ihaka and Robert Gentleman (New Zealand), influenced programming in the data science community
- Java, 1995; imperative object-oriented; statically typed, virtual machine, version 8+ has functional programming features (higher-order functions, streams); developed by Sun Microsystems, now Oracle
- JavaScript, 1995 (standardized as ECMAScript); imperative and prototype-based; designed for embedding in web pages, dynamic typing, first-class functions, prototype-based object-oriented programming, internals influenced by Scheme and Self but using a Java-like syntax; developed by Brendan Eich at Netscape in 12 days to meet a deadline, became popular quickly before language design made clean, evolving slowly because of requirement to maintain backward compatibility
- PHP, 1995; imperative; server-side scripting language fordynamic web applications; originally developed by Rasmus Lerdorf (Canada), evolved organically
- OCaml (originally Objective Caml), 1996; mostly functional with imperative and object-oriented features; a dialect of ML that adds object-oriented constructs, focusing on performance and practical use; developed by a team lead by Xavier Leroy (France)

2000's

- C#, 2001; imperative object-oriented programming; statically typed, language runs on Microsoft's Common Language Infrastructure; developed by Microsoft (in response to Sun's Java)

- F#, 2002; OCaml re-envisioned for Microsoft's Common Language Infrastructure (.Net), replaces OCaml's object and module systems with .Net concepts; developed by a team led by Don Syme at Microsoft Research in the UK

- Scala [132,151], 2003; hybrid functional and object-oriented language; runs on the Java Virtual Machine and interoperates with Java; developed by Martin Odersky's team at EPFL in Switzerland

- Groovy, 2003; imperative object-oriented; dynamically typed "scripting" language, runs on the Java Virtual Machine; originally proposed by James Strachan

- miniKanren [26,27,80], 2005; relational; a family of relational programming languages, developed by Dan Friedman's team at Indiana University, implemented as an extension to other languages (originally Scheme), most popular current usage probably in Clojure

- Clojure [75,94,95], 2007; mixed functional and imperative; Lisp dialect, runs on Java Virtual Machine, Microsoft Common Language Runtime, and JavaScript platform, emphasis on functional programming, concurrency (e.g., software transactional memory), and immutable data structures; developed by Rich Hickey

2010's

- Idris [18,19], 2011 (1.0 release 2017); functional; eagerly evaluated, Haskell-like language with dependent types, incorporating ideas from proof assistants (e.g., Coq), intended for practical programming; developed by Edwin Brady (UK)

- Julia, 2012 (1.0 release 2018); dynamic programming language designed to address high-performance numerical and scientific programming, intended as a modern replacement for MATLAB, Python, and R

- Elixir [68,168], 2012 (1.0 release 2014); functional concurrent programming language; dynamic strong typing, metaprogramming, protocols, Erlang actors, runs on Erlang Virtual Machine, influenced by Erlang, Ruby, and Clojure; developed by a team led by Jose Valim (Brazil)

- Elm [60,70], 2012 (0.19.1 release October 2019); simplified, eagerly evaluated Haskell-like functional programming language that compiles to JavaScript, intended primarily for user-interface programming in a browser, supports reactive-style programming; developed by Evan Czaplicki (original version for his senior thesis at Harvard)

- Rust [110,150], 2012 (1.0 release 2015); imperative; systems programming language that incorporates contemporary language concepts and focuses on safety and performance, meant to replace C and C++; developed originally at Mozilla Research by Graydon Hoare
- PureScript [79,143], 2013 (0.12 release May 2018); mostly functional; an eagerly evaluated language otherwise similar to Haskell, primarily compiles to human-readable JavaScript; originally developed by Phil Freeman
- Swift, 2014; Apple's replacement for Objective C that incorporates contemporary language concepts and focuses on program safety; "Objective C without the C"

The evolution continues!

1.4 What Next?

Computer systems, software development practices, and programming languages have evolved considerably since their beginnings in the 1940s and 1950s. Contemporary languages build on many ideas that first emerged in the early decades of programming languages. But they mix the enduring ideas with a few modern innovations and adapt them for the changing circumstances.

This textbook explores both programming and programming language organization with the following approach:

- emphasize important concepts and techniques that have emerged during the decades since the 1940s
- teach functional and modular programming primarily using the language Haskell, a language that embodies many of the important concepts
- explore the design and implementation of programming languages by building interpreters for simple languages

Chapters 2 and 3 explore the concept of programming paradigms.

1.5 Exercises

1. Choose some programming language not discussed above and investigate the following issues.

   a. When was the language created?
   b. Who created it?
   c. What programming paradigm(s) does it support? (See Chapters 2 and 3 for more information about programming paradigms.)
   d. What are its distinguishing characterists?
   e. What is its primary target domain or group of users?
   f. What are other interesting aspects of the language, its history, use, etc?

2. Repeat the previous exercise for some other language.

1.6 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work in from my previous materials:

- Evolving Computer Hardware Affects Programming Languages from my notes *Effect of Computing Hardware Evolution on Programming Languages*, which were based on a set of unscripted remarks I made in the Fall 2014 offering of CSci 450, Organization of Programming Languages
- History of Programming Languages from my notes *History of Programming Languages*, which were based on a set of unscripted remarks I made in the Fall 2014 offering of CSci 450, Organization of Programming Languages. Those remarks drew on the following:
  - O'Reilly History of Programming Languages poster [125]
  - Wikipedia article on History of Programming Languages [180]

In 2017, I continued to develop this material as a part of Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. These are Chapter 1, Evolution of Programming Languages (this chapter); Chapter 2, Programming Paradigms); chapter 3, Object-Based Paradigms; and Chapter 80 (an appendix), Review of Relevant Mathematics.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

1.7 Terms and Concepts

The evolution of computer hardware since the 1940s; impacts upon programming languages and their subsequent evolution.

2 Programming Paradigms

2.1 Chapter Introduction

The goals of this chapter are to:

- introduce the concepts of procedural and data abstraction
- examine the characteristics and concepts the primary programming paradigms, imperative and declarative (including functional and relational)
- survey other paradigms such as procedural and modular programming

2.2 Abstraction

Programming concerns the construction of appropriate abstractions in a programming language. Before we examine programming paradigms, let's examine the concept of abstraction.

2.2.1 What is abstraction?

As computing scientists and computer programmers, we should remember the maxim:

> *Simplicity is good; complexity is bad.*

The most effective weapon that we have in the fight against complexity is *abstraction*. What is abstraction?

Abstraction is *concentrating on the essentials and ignoring the details.*

Sometimes abstraction is described as *remembering the "what" and ignoring the "how"*.

Large complex problems can only be made understandable by decomposing them into subproblems. Ideally, we should be able to solve each subproblem independently and then compose their solutions into a solution to the larger problem.

In programming, the subproblem solution is often expressed with some kind of abstraction represented in a programming notation. From the outside, each abstraction should be simple and easy for programmers to use correctly. The programmers should only need to know the abstraction's *interface* (i.e., some small number of assumptions necessary to use the abstraction correctly).‘

2.2.2 Kinds of abstraction

Two kinds of abstraction are of interest to computing scientists: *procedural abstraction* and *data abstraction*.

Procedural abstraction: the separation of the logical properties of an *action* from the details of how the action is implemented.

Data abstraction: the separation of the logical properties of *data* from the details of how the data are represented.

In procedural abstraction, programmers focus primarily on the actions to be carried out and secondarily on the data to be processed.

For example, in the top-down design of a sequential algorithm, a programmer first identifies a sequence of actions to solve the problem without being overly concerned about how each action will be carried out.

If an action is simple, the programmer can code it directly using a sequence of programming language statements.

If an action is complex, the programmer can abstract the action into a subprogram (e.g., a procedure or function) in the programming language. The programmer must define the subprogram's name, parameters, return value, effects, and assumptions—that is, define its interface. The programmer subsequently develops the subprogram using the same top-down design approach.

In data abstraction, programmers primarily focus on the problem's data and secondarily on its actions. Programmers first identify the key data representations and develop the programs around those and the operations needed to create and update them.

We address procedural and data abstraction further in Chapters 6 and 7.

2.2.3 Procedures and functions

Generally we make the following distinctions among subprograms:

- A *procedure* is (in its pure form) a subprogram that takes zero or more arguments but does not return a value. It is executed for its effects, such as changing values in a data structure within the program, modifying its reference or value-result arguments, or causing some effect outside the program (e.g., displaying text on the screen or reading from a file).
- A *function* is (in its pure form) a subprogram that takes zero or more arguments and returns a value but that does not have other effects.
- A *method* is a procedure or function often associated with an object or class in an object-oriented program. Some object-oriented languages use the metaphor of message-passing. A method is the feature of an object that receives a message. In an implementation, a method is typically a procedure or function associated with the (receiver) object; the object may be an *implicit parameter* of the method.

Of course, the features of various programming languages and usual practices for their use may not follow the above pure distinctions. For example, a language may not distinguish between procedures and functions. One term or another may be used for all subprograms. Procedures may return values. Functions may

have side effects. Functions may return multiple values. The same subprogram can sometimes be called either as a function or procedure.

Nevertheless, it is good practice to maintain the distinction between functions and procedures for most cases in software design and programming.

2.3 What is a Programming Paradigm?

According to Timothy Budd, a *programming paradigm* is "a way of conceptualizing what it means to perform computation, of structuring and organizing how tasks are to be carried out on a computer" [21:3].

Historically, computer scientists have classified programming *languages* into one of two primary paradigms: *imperative* and *declarative*.

This imperative-declarative taxonomy categorizes programming styles and language features on how they handle state and how they execute programs.

In recent years, many imperative languages have added more declarative features, so the distinction between languages has become blurred. However, the concept of *programming* paradigm is still meaningful.

2.4 Imperative Paradigm

A program in the imperative paradigm has an *implicit state* (i.e., values of variables, program counters, etc.) that is modified (i.e., side-effected or mutated) by *constructs* (i.e., commands) in the source language [101].

As a result, such languages generally have an explicit notion of *sequencing* (of the commands) to permit precise and deterministic control of the state changes.

Imperative programs thus express *how* something is to be computed. They emphasize procedural abstractions.

2.4.1 Java

Consider the following Java program fragment from file `Counting.java`:

```
int count =  0 ;
int maxc  = 10 ;
while (count <= maxc) {
    System.out.println(count) ;
    count = count + 1 ;
}
```

In this fragment, the program's *state* includes at least the values of the variables `count` and `maxc`, the sequence of output lines that have been printed, and an indicator of which statement to execute next (i.e., location or program counter).

The assignment statement changes the value of `count` and the `println` statement adds a new line to the output sequence. These are *side effects* of the execution.

Similarly, Java executes these commands in sequence, causing a change in which statement will be executed next. The purpose of the `while` statement is to cause the statements between the braces to be executed zero or more times. The number of times depends upon the values of `count` and `maxc` and how the values change within the `while` loop.

We call this state *implicit* because the aspects of the state used by a particular statement are not explicitly specified; the state is assumed from the context of the statement. Sometimes a statement can modify aspects of the state that are not evident from examining the code fragment itself.

The Java variable `count` is *mutable* because its value can change. After the declaration, `count` has the value 0. At the end of the first iteration of the `while` loop, it has value 1. After the `while` loop exits, it has a value 10. So a reference to `count` yields different values depending upon the state of the program at that point.

The Java variable `maxc` is also *mutable*, but this code fragment does not change its value. So `maxc` could be replaced by an *immutable* value.

Of course, the Java fragment above must be included within a `main` method to be executed. A `main` method is the entry point of a Java program.

```
public class Counting {
    public static void main(String[] args) {
        /* Java code fragment above */
    }
 }
```

Imperative languages are the "conventional" or "von Neumann languages" discussed by John Backus in his 1977 Turing Award address [6]. (See Section 2.7.) They are suited to traditional computer architectures.

Most of the languages in existence today are primarily imperative in nature. These include Fortran, C, C++, Java, Scala, C#, Python, Lua, and JavaScript.

2.4.2 Other languages

The Scala [132,151] program `CountingImp.scala` is equivalent to the Java program described above. The program `CountingImp2.scala` is also equivalent, except that it makes the `maxc` variable *immutable*. That is, it can be bound to an initial value, but its binding cannot be changed subsequently.

2.5 Declarative Paradigm

A program in the declarative paradigm has *no implicit* state. Any needed state information must be handled explicitly [101].

A program is made up of *expressions* (or terms) that are *evaluated* rather than commands that are executed.

Repetitive execution is accomplished by *recursion* rather than by sequencing.

Declarative programs express *what* is to be computed (rather than how it is to be computed).

The declarative paradigm is often divided into two types: *functional* (or applicative) and *relational* (or logic).

2.5.1 Functional paradigm

In the functional paradigm the underlying model of computation is the mathematical concept of a *function* [101].

In a computation, a function is applied to zero or more arguments to compute a single result; that is, the result is deterministic (or predictable).

2.5.1.1 Haskell

Consider the following Haskell code from file `Counting.hs`:

```
counter :: Int -> Int -> String
counter count maxc
    | count <= maxc = show count ++ "\n"
                      ++ counter (count+1) maxc
    | otherwise     = ""
```

This fragment is similar to the Java fragment above. This Haskell code defines a function `counter` (i.e., a procedural abstraction) that takes two integer arguments, `count` and `maxc`, and returns a string consisting of a sequence of lines with the integers from `count` to `maxc` such that each would be printed on a separate line. (It does not print the string, but it inserts a newline character at the end of each line.)

In the evaluation (i.e., "execution") of a function call, Programming for the {Newton}: Software Development with {NewtonScript}`counter` references the *values* of `count` and `maxc` corresponding to the explicit arguments of the function call. These values are not changed during the evaluation of that function call. However, the values of the arguments can be changed as needed for a subsequent *recursive* call of `counter`.

We call the state of `counter` *explicit* because it is passed in arguments of the function call. These parameters are *immutable* (i.e., their values cannot change) within the body of the function. That is, any reference to `count` or `maxc` within a call gets the same value.

In a pure functional language like Haskell, the names like `count` and `maxc` are said to be *referentially transparent*. In the same context (such as the body of the function), they always have the same value. A name must be defined before it is used, but otherwise the order of evaluation of the expressions within a function body does not matter; they can even be evaluated in parallel.

There are no "loops". The functional paradigm uses recursive calls to carry out a task repeatedly.

As we see in later chapters, referential transparency is probably the most important property of functional programming languages. It underlies Haskell's evaluation model (Chapter 8). It also underlies the ability to state and prove "laws" about Haskell programs (e.g., Chapters 25 and 26). Haskell programmers and Haskell compilers can use the "mathematical" properties of the programs to transform programs that are more efficient.

The above Haskell fragment does not really carry out any actions; it just defines a mapping between the arguments and the return value. We can "execute" the `counter` function above with the arguments 0 and 10 with the following `IO` program.

```
main = do
    putStrLn (counter 0 10)
```

By calling the `main` function from the `ghci` interpreter, we get the same displayed output as the Java program.

Haskell separates pure computation (as illustrated by function `counter`) from computation that has effects on the environment such as input/output (as illustrated by `IO` function `main`).

In most programming languages that support functional programming, functions are treated as *first-class* values. That is, like other data types, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions. (The implementation technique for first-order functions usually involves creation of a *lexical closure* holding the function and its environment.)

In some sense, functional languages such as Haskell merge the concepts of procedural and functional abstraction. Functions are procedural abstractions, but they are also data.

A function that can take functions as arguments or return functions in the result is called a *higher-order function*. A function that does not take or return functions is thus a *first-order function*. Most imperative languages do not fully support higher-order functions.

The higher-order functions in functional programming languages enable regular and powerful abstractions and operations to be constructed. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

Purely functional languages include Haskell, Idris, Miranda, Hope, Elm, and Backus's FP.

Hybrid functional languages with significant functional subsets include Scala, F#, OCaml, SML, Erlang, Elixir, Lisp, Clojure, and Scheme.

Mainstream imperative languages such as Java (beginning with version 8), C#, Python, Ruby, Groovy, Rust, and Swift have recent feature extensions that make them hybrid languages as well.

2.5.1.2 Other languages The Scala [132,151] program `CountingFun.scala` is equivalent to the above Haskell program.

2.5.2 Relational (or logic) paradigm

In the relational (logic) paradigm, the underlying model of computation is the mathematical concept of a *relation* (or a *predicate*) [101].

A computation is the (nondeterministic) association of a group of values—with backtracking to resolve additional values.

2.5.2.1 Prolog Consider the following Prolog [33] code from file `Counting.pl`. In particular, this code runs on the SWI-Prolog interpreter [163].

```
counter(X,Y,S) :- count(X,Y,R), atomics_to_string(R,'\n',S).

count(X,X,[X]).
count(X,Y,[])      :- X  > Y.
count(X,Y,[X|Rs]) :- X < Y, NX is X+1, count(NX,Y,Rs).
```

This fragment is somewhat similar to the Java and Haskell fragments above. It can be used to generate a string with the integers from `X` to `Y` where each integer would be printed on a separate line. (As with the Haskell fragment, it does not print the string.)

This program fragment defines a *database* consisting of four *clauses*.

The clause

```
count(X,X,[X]).
```

defines a *fact*. For any *variable* value `X` and list `[X]` consisting of the single value `X`, `count(X,X,[X])` is asserted to be true.

The other three clauses are *rules*. The left-hand-side of `:-` is true if the right-hand-side is also true. For example,

```
count(X,Y,[]) :- X > Y.
```

asserts that

```
count(X,Y,[])
```

is true when `X > Y`. The empty brackets denote an empty list of values.

As a logic or relational language, we can *query* the database for any missing components. For example,

```
count(1,1,Z).
```

yields the value `Z = [1]`. However,

```
count(X,1,[1]).
```

yields the value `X = 1`. If more than one answer is possible, the program can generate all of them in some nondeterministic order.

So, in some sense, where imperative and functional languages only run a computation in one direction and give a single answer, Prolog can potentially run a computation in multiple directions and give multiple answers.

As with Haskell, the above Prolog fragment does not really carry out any computational actions; it just adds facts to the database and defines general relationships among facts. We can "execute" the query `counter(0,10,S)` above and print the value of `S` using the following rule.

```
main :- counter(0,10,S), write(S).
```

Example relational languages include Prolog, Parlog, and miniKanren.

Most Prolog implementations have imperative features such as the "cut" and the ability to assert and retract clauses.

2.5.2.2 Other languages TODO: Perhaps add a new example using miniKanren [26,27,80] in some reasonable base language–preferably Java, Python, or Scala.

2.6 Other Programming Paradigms

As we noted, the imperative-declarative taxonomy described above divides programming styles and language features on how they handle state and how they are executed.

The computing community often speaks of other paradigms—procedural, modular, object-oriented, concurrent, parallel, language-oriented, scripting, reactive, and so forth. The definitions of these "paradigms" may be quite fuzzy and vary significantly from one writer to another.

Sometimes a term is chosen for "marketing" reasons—to associate a language with some trend even though the language may be quite different from others in that paradigm—or to make a language seem different and new even though it may not be significantly different.

These paradigms tend to divide up programming styles and language features along different dimensions than the primary taxonomy described in Sections 2.4 and 2.5. Often the languages we are speaking of are subsets of the imperative paradigm.

This section briefly discusses some of these paradigms. We discuss the prominent object-based paradigms in the next chapter.

2.6.1 Procedural paradigm

The *procedural paradigm* is a subcategory of the imperative paradigm. It organizes programs primarily using procedural abstractions. A procedural program consists of a sequence of steps that access and modify the program's state.

Some of the steps are abstracted out as subprograms—procedures or functions—that can be reused. In some cases, subprograms may be nested inside other subprograms, thus limiting the part of the program in which the nested subprogram can be called.

The procedural programming approach arose in programming languages such as Fortran, Algol, PL/I, Pascal, and C from the 1950's to the 1970's and beyond. In this chapter, we use the Python programming language to illustrate of its features.

2.6.1.1 Python

Consider the following Python [144] code from file `CountingProc.py`:

```
# File CountingProc.py
def counter(count,maxc):
    def has_more(count,maxc): # new variables
        return count <= maxc
    def adv():
        nonlocal count     # from counter
        count = count + 1
    while has_more(count,maxc):
        print(f'{count}')  # Python 3.6+ string interpolation
        adv()
```

When called as

```
counter(0,10)
```

this imperative Python "procedure" executes similarly to the Java program fragment we examined in Section 2.4.

Python does not distinguish between procedures and functions as we have defined them. It uses the term "function" for both. Both return values and can have side-effects. The value returned may be the special default value `None`.

This Python code uses procedural abstraction more extensively than the earlier Java fragment. The Python procedure encloses the `while` loop in procedure `counter` and abstracts the loop test and incrementing operation into function `has_more` and procedure `adv`, respectively.

Like many procedural languages, Python uses *lexical scope* for variable, procedure, and function names. That is, the *scope* of a name (i.e., range of code in which it can be accessed) begins at the point it is defined and ends at the end of that block of code (e.g., function, class, or module).

Function has_more and procedure adv are encapsulated within counter. They can only be accessed inside the body of counter after their definitions.

Parameters count and maxc of procedure counter can be accessed throughout the body of counter unless hidden by another variable or parameter with the same name. They are hidden within the function has_more, which reuses the names for its parameters, but are accessible within procedure adv.

But to allow assignment to count within the nested procedure adv, the variable must declared as nonlocal in the inner procedure. Otherwise, the assignment would have created a new variable with the name count within the body of procedure adv.

Languages like Python, C, Fortran, Pascal, and Lua are primarily procedural languages, although most have evolved to support other styles.

2.6.1.2 Other languages Scala [132,151] is a hybrid object-functional language that enables function definitions to be nested inside other function definitions. The procedural Scala program CountingProc.scala is equivalent to the Python program above.

2.6.2 Modular paradigm

Modular programming refers more to a design method for programs and program libraries than to languages.

Modular programming means to decompose a program into units of functionality (i.e., modules) that can be developed separately and then recomposed. These *modules* can hide (i.e., encapsulate) key design and implementation details within the modu

The module's *public* features can be accessed through its interface; its *private* features cannot be accessed from outside the module. Thus a module supports the principle of *information hiding.* This method also keeps the interactions among modules at a minimum, maintaining a low degree of coupling.

We discuss modular programming in more depth in Chapters 6 and 7.

A language that provides constructs for defining modules, packages, namespaces, or separate compilation units can assist in writing modular programs.

In this chapter, we examine some aspects of the modular paradigm using the imperative language Python. We examine modular programming in the purely functional programming language Haskell on Chapters 6 and 7 and later chapters.

2.6.2.1 Python

2.6.2.1.1 Using one module First, let's consider the following Python [144] code from file CountingMod.py to illustrate use of modules in Python programs. This module is similar to the procedural program in the previous section.

This modular program, however, has all the functions and procedures at the same level of the Python module (file) instead of most being nested within procedure `counter`. The modular program also uses module-level variables instead of local variables of procedure `counter`.

```
# File CountingMod.py
count =  0
maxc  = 10

def has_more():
    return count <= maxc

def adv():
    global count
    count = count + 1

def counter():
    while has_more():
        print(f'{count}')
        adv()
```

This module creates two module-level *global* variables `count` and `maxc` and defines three module-level Python functions `has_more`, `adv`, and `counter`.

The module assigns initial values to the variables. Their values can be accessed anywhere later in the module unless hidden by parameters or local variables with the same name.

Function `has_more()` tests module-level variables `count` and `maxc` to determine whether there are more items in the sequence.

Procedure `adv()` assigns a new value to the module-level variable `count`. It must declare `count` as `global` so that a new local variable is not created.

Variable `maxc` is also mutable, but this module does not modify its value.

Each module is a separate file that can be imported by other Python code. It introduces a separate name space for variables, functions, and other features.

For example, we can import the module above and execute counter with the following Python code from file `CountingModTest1.py`:

```
from CountingMod import counter
counter()
```

The `from-import` statement imports feature `counter` (a Python function) from the module in file `CountingMod.py`. The imported name `counter` can be used without qualifying it. The other features of `CountingMod` (e.g., `count` and `adv`) cannot be accessed.

As an alternative, we can import the module from file `CountingModTest2.py` as follows:

```
import CountingMod

CountingMod.count = 10
CountingMod.maxc  = 20
CountingMod.counter()
```

This code imports all the features of the module. It requires the variables and functions to be accessed with the name prefix `CountingMod.` (i.e., the module name followed by a period). This approach enables the importing code to modify the values of global variables in the imported module.

In this second example, the importing code can both access and modify the global variables of the imported module.

Python does not enforce the encapsulation of module-level variable or function names. All names are public (i.e., can be imported to other modules). However, programmers can, by convention, designate module-level names as private by beginning the name with a single underscore character `_`. The alternative `import` above will not automatically import such names.

For example, good modular programming practice might suggest that the names `_count`, `_maxc`, `_has_more()`, and `_adv()` be used in the `CountingMod` module above. This naming convention would designate those as private and leave only `counter()` as public.

Most modern languages support "modules" in some way. Other languages (e.g., Standard ML) provide advanced support for modules with the ability to encapsulate features and provide multiple implementations of common interfaces.

2.6.2.1.2 Using multiple modules To see the flexibility of modular programs, let's consider a variant of the above that uses two modules.

The first module—`CountingModA` from file `CountingModA.py`—is shown below.

```
# File CountingModA.py
from Arith import reset, adv, get_count, has_more

def counter():
    while has_more():
        count = get_count()
        print(f'{count}')
        adv()
```

`CountingModA` has similar overall functionality to the `CountingMod` module in the previous example. However, its `counter` procedure uses a `has_more` function, an `adv` procedure, and a new `get_counter` function implemented in a separate

module named Arith. The CountingModA module has no module-level variables and its counter procedure has no local variables.

The second module—Arith from file Arith.py—is shown below.

```
# File Arith.py
_start  =  0
_stop   = 10
_change =  1
_count  = _start

def reset(new_start, new_stop, new_change):
    global _start, _stop, _change, _count
    _start = new_start
    _stop  = new_stop
    _count = _start
    if new_change == 0:
        print('Error: Attempt to reset increment to 0; not reset.')
    else:
        _change = new_change

def adv():
    global _count
    _count = _count + _change

def get_count():
    return _count

def has_more():
    if _change > 0:
        return _count <= _stop
    else:
        return _count >= _stop
```

This module makes the module-level variables private to the module by convention.

By default, module Arith generates the same arithmetic sequence as CountingMod in the previous modular programming example. However, it generalizes CountingMod in the following ways:

- renaming variable count to be _count and variable maxc to be _stop
- replacing the constant 0 in the initialization of variable _count by a new variable _start, which is itself initialized to 0
- replacing the constant 1 in the increment of variable _count by a new variable _change, which is itself initialized to 1

- adding a new function `get_count` that enables a user module (e.g., `CountingModA`) to get the current value of the `_count` variable

  This is called an *accessor* or *getter* function.

- implementing the function `has_more()` and the procedure `adv()` used by module `CountingModA`

  These argumentless public functions operate on `Arith`'s private module-level variables `_start`, `_stop`, `_change`, and `_count`.

- adding a new procedure `reset()` that enables the values of `_start`, `_stop`, `_change`, and `_count` to be reinitialized to new values

Now let's consider an alternative to `Arith`, the second module. Module `Geom` from file `Geom.py` is shown below.

```
# File Geom.py
_start  =   1
_stop   = 100
_change =   2
_count  = _start

def reset(new_start, new_stop, new_change):
    global _start, _stop, _change, _count
    _start = new_start
    _stop  = new_stop
    _count = start
    if abs(new_change) <= 1:
        print('Error: Attempt to set abs(_change) <= 1; not reset.')
    else:
        _change = new_change

def adv():
    global _count
    _count = _count * _change

def get_count():
    return _count

def has_more():
    return _count <= _stop
```

Module `Geom` has essentially the same interface as `Arith`, but it generates a geometric sequence instead of an arithmetic sequence.

To use this module, the only change needed to `CountingModA.py` is to import the module `Geom` instead of `Arith`. This alternative is in module `CountingModG` in file `CountingModG.py`.

This two-level example illustrates the additional flexibility that modular programming can enable.

2.6.2.2 Other languages The modular Scala [132,151] program `CountingMod.scala` is equivalent to the first Python program above. The similar Scala program `CountingMod2.scala` uses a Scala *trait* to define the interface of the module. It is used in manner similar to the second Python program above.

TODO: Probably should show a Java 8+ example for this. Also the Scala might need more update to be similar to new modular Python examples.

2.6.3 Object-based paradigms

The dominant paradigm since the early 1990s has been the *object-oriented paradigm*. Because this paradigm is likely familiar with most readers, we examine it and related object-based paradigms in the next chapter.

2.6.4 Concurrent paradigms

TODO: Perhaps describe a paradigm like actors and give an example in Elixir [68,168].

2.7 Motivating Functional Programming: John Backus

In this book we focus primarily on the functional paradigm—on the programming language Haskell in particular. Although languages that enable or emphasize the functional paradigm have been around since the early days of computing, much of the later interest in functional programming grew from the 1977 Turing Award lecture.

John W. Backus (December 3, 1924 – March 17, 2007) was a pioneer in research and development of programming languages. He was the primary developer of Fortran while a programmer at IBM in the mid-1950s. Fortran is the first widely used high-level language. Backus was also a participant in the international team that designed the influential languages Algol 58 and Algol 60 a few years later. The notation used to describe the Algol 58 language syntax—Backus-Naur Form (BNF)—bears his name. This notation continues to be used to this day.

In 1977, ACM bestowed its Turing Award on Backus in recognition of his career of accomplishments. (This award is sometimes described as the "Nobel Prize for computer science".) The annual recipient of the award gives an address to a major computer science conference. Backus's address was titled "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs".

Although functional languages like Lisp go back to the late 1950's, Backus's

address did much to stimulate research community's interest in functional programming languages and functional programming over the past four decades.

The next subsection gives excerpts from Backus's Turing Award address published as the article "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" [6].

2.7.1 Excerpts from Backus's Turing Award Address [6]

Programming languages appear to be in trouble. Each successive language incorporates, with little cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. ... Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there is a desperate need for a powerful methodology to help us think about programs, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs. ... In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived of it ... [in the 1940's], it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this ... concept.

In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent form the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must either be generated by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must

have been sent, and so on.

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. ...

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our ... old belief that there is only one kind of computer is the basis our our belief that there is only one kind of programming language, the conventional—von Neumann—language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem.

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-at-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment <statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements in the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce

some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on *it* has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. . . .

2.7.2 Aside on the disorderly world of statements

Backus states that "the world of statements is a disorderly one, with few mathematical properties". Even in 1977 this was a bit overstated since work by Hoare on *axiomatic semantics* [96], by Dijkstra on the *weakest precondition (wp) calculus* [63], and by others had already appeared.

However, because of the referential transparency property of purely functional languages, reasoning can often be done in an equational manner within the context of the language itself. We examine this convenient approach later in this book.

In contrast, the *wp*-calculus and other axiomatic semantic approaches must project the problem from the world of programming language statements into the world of predicate calculus, which is much more orderly. We leave this study to courses on program derivation and programming language semantics.

Note: For this author's take on this formal methods topic, see my materials for University of Mississippi course Program Semantics and Derivation (CSci 550) [40,41].

2.7.3 Perspective from four decades later

In his Turing Award Address, Backus went on to describe FP, his proposal for a functional programming language. He argued that languages like FP would allow programmers to break out of the von Neumann bottleneck and find new ways of thinking about programming.

FP itself did not catch on, but the widespread attention given to Backus' address and paper stimulated new interest in functional programming to develop by researchers around the world. Modern languages like Haskell developed partly from the interest generated.

In the 21st Century, the software industry has become more interested in functional programming. Some functional programming features now appear in most mainstream programming languages (e.g., in Java 8+). This interest seems to driven primarily by two concerns:

- managing the complexity of large software systems effectively
- exploiting multicore processors conveniently and safely

The functional programming paradigm is able to address these concerns because of such properties such as referential transparency, immutable data structures, and composability of components. We look at these aspects in later chapters.

2.8 What Next?

This chapter (2) introduced the concepts of abstraction and programming paradigm and surveyed the imperative, declarative, functional, and other paradigms.

Chapter 3 continues the discussion of programming paradigms by examining the object-oriented and related object-based paradigms.

The subsequent chapters use the functional programming language Haskell to illustrate general programming concepts and explore programming language design and implementation using interpreters.

2.9 Exercises

1. This chapter used Haskell (and Scala) to illustrate the functional paradigm. Choose a language such as Java, Python, or C#. Describe how it can be used to write programs in the functional paradigm. Consider how well the language supports tail recursion.

   TODO: Modify question if more examples are given in chapter.

2. This chapter used Python (and Scala) to illustrate the procedural paradigm. Choose a different language such as Java, C, C++, or C#. Describe how it can be used to write programs in the procedural paradigm.

   TODO: Modify question if more examples are given in chapter.

3. This chapter used Python (and Scala) to illustrate the modular paradigm. For the same language chosen for previous exercise, describe how it can be used to write programs in the modular paradigm.

   TODO: Modify question if more examples are given in chapter.

4. Repeat the previous two exercises with a different language.

2.10 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from my previous materials:

- Abstraction (Section 2.2) from the "What is Abstraction?" section of my Data Abstraction notes [46], which I wrote originally for the first C++ (CSci 490) and Java-based (CSci 211) classes at UM in 1996 but expanded

and adapted for other courses in later years. In the mid-to-late 1990s, the Data Abstraction notes drew on my study of a variety of sources (e.g., Bird and Wadler [13], Dale [61], Gries [85]; Horstmann [99,100], Liskov [119], Meyer [128], Mossenbock [129], Parnas [134], and Thomas [170])

- Discussion of the primary programming paradigms (Sections 2.3-2.6) from Chapter 1 of my *Notes on Functional Programming with Haskell* [42], which drew on the taxonomy in Hudak's survey paper [101]. In 2016, I expanded the discussion of the paradigms and included examples. This drew in part from my use and/or teaching of a variety of programming languages since my first programming course in 1974 (e.g., Fortran, Cobol, Pl/I, C, Snobol, Jovial, Ada, Pascal, Haskell, C++, Java, Ruby, Scala, Lua, Elixir, and Python).
- Motivating Functional Programming (Section 2.7) from Chapter 1 of my *Notes on Functional Programming with Haskell* [42]. This includes a long excerpt from the influential Turing Award lecture by John Backus [6].

In 2017, I continued to develop this material as a part of Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* These are Chapter 1, Evolution of Programming Languages; Chapter 2, Programming Paradigms (this chapter); Chapter 3, Object-based Paradigms; and Chapter 80, Review of Relevant Mathematics. I added the examples on procedural and modular programming.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), adding cross-references, and improving the build workflow and use of Pandoc.

In 2022, I aslo revised and expanded the modular programming example

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

2.11 Terms and Concepts

TODO: Update

Abstraction, procedural abstraction, data abstraction, interface, procedures, functions, methods; programming language paradigm, primary paradigms (imperative, declarative, functional, relational or logic language); other paradigms (procedural, modular, object-oriented, concurrent); program state, implicit versus

explicit state, execution of commands versus evaluation of expressions, mutable versus immutable data structures, side effects, sequencing, recursion, referential transparency, first-class values, first-order and higher-order functions, lexical scope, global versus local variables, public versus private features, information hiding, encapsulation, lexical closure; von Neumann computer, von Neumann language, worlds of expressions and statements, axiomatic semantics, weakest precondition calculus.

3 Object-Based Paradigms

3.1 Chapter Introduction

The imperative-declarative taxonomy described in the previous chapter divides programming styles and language features on how they handle state and how they are executed. The previous chapter also mentioned other paradigms such as procedural, modular, object-based, and concurrent.

The dominant paradigm since the early 1990s has been the *object-oriented paradigm*. Because this paradigm is likely familiar with most readers, it is useful to examine it in more detail.

Thus the goals of this chapter are to examine the characteristics of:

- the object-oriented paradigm
- related paradigms such as the object-based, class-based, and prototype-based paradigms

3.2 Motivation

In contemporary practice, most software engineers approach the design of programs from an object-oriented perspective.

The key idea (notion?) in object orientation is the following: The real world can be accurately described as a collection of objects that interact.

This approach is based on the following *assumptions*:

1. Describing large, complex systems as interacting objects make them *easier to understand* than otherwise.
2. The *behaviors* of real world objects tend to be *stable* over time.
3. The different *kinds* of real world objects tend to be *stable*. (That is, new kinds appear slowly; old kinds disappear slowly.)
4. *Changes* tend to be *localized* to a few objects.

Assumption 1 simplifies requirements analysis, software design, and implementation—makes them more reliable.

Assumptions 2 and 3 support reuse of code, prototyping, and incremental development.

Assumption 4 supports design for change.

The object-oriented approach to software development:

- uses the same basic entities (i.e., objects) throughout the software development lifecycle
- identifies the basic objects during analysis

- identifies lower-level objects during design, reusing existing object descriptions where appropriate
- implements the objects as software structures (e.g., Java classes)
- maintains the object behaviors

<a name="ObjectModel>

3.3 Object Model

We discuss object orientation in terms of an object model. Our *object model* includes four basic components:

1. objects (i.e., abstract data structures)
2. classes (i.e., abstract data types)
3. inheritance (hierarchical relationships among abstract data types)
4. subtype polymorphism

Some writers consider *dynamic binding* a basic component of object orientation. Here we consider it an implementation technique for subtype polymorphism.

Now let's consider each of four components of the object model.

3.3.1 Objects

For languages in the object-based paradigms, we require that objects exhibit three essential characterics. Some writers consider one or two other other characteristics as essential. Here we consider these as important but non-essential characteristics of the object model.

3.3.1.1 Essential characteristics An *object* must exhibit three *essential* characteristics:

a. state
b. operations
c. identity

An object is a separately identifiable entity that has a set of operations and a state that records the effects of the operations. An object is typically a *first-class* entity that can be stored in variables and passed to or returned from subprograms.

The *state* is the collection of information held (i.e., stored) by the object.

- It can change over time.
- It can change as the result of an operation performed on the object.

- It cannot change spontaneously.

The various components of the state are sometimes called the *attributes* of the object.

An *operation* is a procedure that takes the state of the object and zero or more arguments and changes the state and/or returns one or more values. Objects permit certain operations and not others.

If an object is *mutable*, then an operation may change the stored state so that a subsequent operation on that object acts upon the modified state; the language is thus imperative.

If an object is *immutable*, then an operation cannot change the stored state; instead the operation returns a new object with the modified state.

Identity means we can distinguish between two distinct objects (even if they have the same state and operations).

As an example, consider an object for a student desk in a simulation of a classroom.

- A student desk is distinct from the other student desks and, hence, has a unique *identity*.
- The relevant *state* might be attributes such as location, orientation, person using, items in the basket, items on top, etc.
- The relevant *operations* might be state-changing operations (called *mutator*, setter, or command operations) such as "move the desk", "seat student", or "remove from basket" or might be state-observing operations (called *accessor*, getter, observer, or query operations) such as "is occupied" or "report items on desktop".

A language is *object-based* if it supports objects as a language feature.

Object-based languages include Ada, Modula, Clu, C++, Java, Scala, C#, Smalltalk, and Python 3.

Pascal (without module extensions), Algol, Fortran, and C are not inherently object-based.

3.3.1.2 Important but non-essential characteristics Some writers require that an object have additional characteristics, but this book considers these as important but *non-essential* characteristics of objects:

d. encapsulation

e. independent lifecycle

The state may be *encapsulated* within the object—that is, not be directly visible or accessible from outside the object.

The object may also have an *independent lifecycle*—that is, the object may exist independently from the program unit that created it. Its lifetime is not determined by the program unit that created it.

We do not include these as essential characteristics because they do not seem required by the object metaphor.

Also, some languages we wish to categorize as object-based do not exhibit one or both of these characteristics. There are languages that use a modularization feature to enforce encapsulation separately from the object (or class) feature. Also, there are languages that may have local "objects" within a function or procedure.

In languages like Python 3, Lua, and Oberon, objects exhibit an independent lifecycle but do not themselves enforce encapsulation. Encapsulation may be supported by the module mechanism (e.g., in Oberon and Lua) or partly by a naming convention (e.g., in Python 3).

In C++, some objects may be local to a function and, hence, be allocated on the runtime stack. These objects are deallocated upon exit from the function. These objects may exhibit encapsulation, but do not exhibit independent lifecycles.

3.3.2 Classes

A *class* is a template or factory for creating objects.

- A class describes a collection of related objects (i.e., *instances* of the class).
- Objects of the same class have common operations and a common set of possible states.
- The concept of class is closely related to the concept of *type*.

A class description includes definitions of:

- operations on objects of the class
- the set of possible states

As an example, again consider a simulation of a classroom. There might be a class `StudentDesk` from which specific instances can be created as needed.

An object-based language is *class-based* if the concept of class occurs as a language feature and every object has a class.

Class-based languages include Clu, C++, Java, Scala, C#, Smalltalk, Ruby, and Ada 95. Ada 83 and Modula are not class-based.

At their core, JavaScript and Lua are object-based but not class-based.

In statically typed, class-based languages such as Java, Scala, C++, and C# classes are treated as types. Instances of the same class have the same (nominal) type.

However, some dynamically typed languages may have a more general concept of type: If two objects have the same set of operations, then they have the same type regardless of how the object was created. Languages such as Smalltalk and Ruby have this characteristic—sometimes informally called *duck typing*. (If it walks like a duck and quacks like a duck, then it is a duck.)

See Chapter 5 for more discussion of types.

3.3.3 Inheritance

A class C *inherits* from class P if C's objects form a subset of P's objects.

- Class C's objects must support all of the class P's operations (but perhaps are carried out in a special way).
- Class C may support additional operations and an extended state (i.e., more information fields).
- Class C is called a *subclass* or a *child* or *derived class*.
- Class P is called a *superclass* or a *parent* or *base* class.
- Class P is sometimes called a *generalization* of class C; class C is a *specialization* of class P.

The importance of inheritance is that it encourages sharing and reuse of both design information and program code. The shared state and operations can be described and implemented in base classes and shared among the subclasses.

As an example, again consider the student desks in a simulation of a classroom. The `StudentDesk` class might be derived (i.e., inherit) from a class `Desk`, which in turn might be derived from a class `Furniture`. In diagrams, there is a convention to draw arrows (e.g., $\longleftarrow$) from the subclass to the superclass.

`Furniture` $\longleftarrow$ `Desk` $\longleftarrow$ `StudentDesk`

The simulation might also include a `ComputerDesk` class that also derives from `Desk`.

`Furniture` $\longleftarrow$ `Desk` $\longleftarrow$ `ComputerDesk`

We can also picture the above relationships among these classes with a class diagram as shown in Figure 3.1.

In Java and Scala, we can express the above inheritance relationships using the `extends` keyword as follows.

```
class Furniture  // extends cosmic root class for references
{   ...   }      // (java.lang.Object, scala.AnyRef)

class Desk extends Furniture
{   ...   }
```

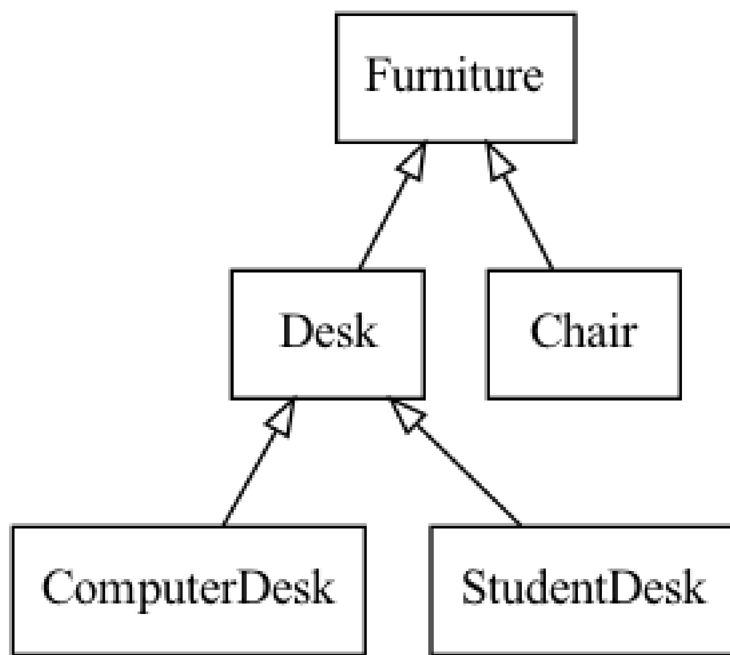


Figure 3.1: Classroom simulation inheritance hierarchy.

```
class StudentDesk extends Desk
{   ...   }

class ComputerDesk extends Desk
{   ...   }
```

Both `StudentDesk` and `ComputerDesk` objects will need operations to simulate a `move` of the entity in physical space. The `move` operation can thus be implemented in the `Desk` class and shared by objects of both classes.

Invocation of operations to `move` either a `StudentDesk` or a `ComputerDesk` will be bound to the general `move` in the `Desk` class.

The `StudentDesk` class might inherit from a `Chair` class as well as the `Desk` class.

`Furniture` ⟵ `Chair` ⟵ `StudentDesk`

Some languages support *multiple inheritance* as shown in Figure 3.2 for `StudentDesk` (e.g., C++, Eiffel, Python 3). Other languages only support a single inheritance hierarchy.

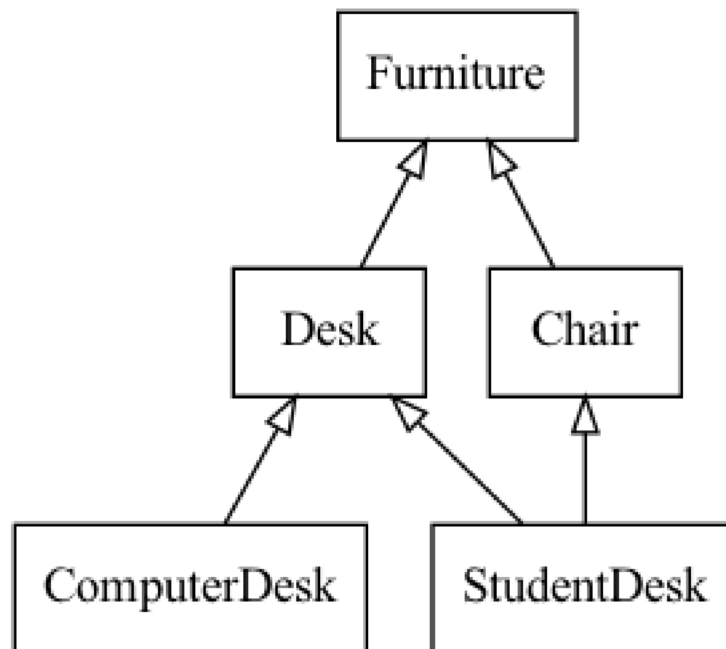


Figure 3.2: Classroom simulation with multiple inheritance.

Because multiple inheritance is both difficult to use correctly and to implement in a compiler, the designers of Java and Scala did not include multiple inheritance

of classes as features. Java has a single inheritance hierarchy with a top-level class named `Object` from which all other classes derive (directly or indirectly). Scala is similar, with the corresponding top-level class named `AnyRef`.

```
class StudentDesk extends Desk, Chair  // NOT VALID in Java
{   ...   }
```

To see some of the problems in implementing multiple inheritance, consider the above example. Class `StudentDesk` inherits from class `Furniture` through two different paths. Do the data fields of the class `Furniture` occur once or twice? What happens if the intermediate classes `Desk` and `Chair` have conflicting definitions for a data field or operation with the same name?

The difficulties with multiple inheritance are greatly decreased if we restrict ourselves to inheritance of class *interfaces* (i.e., the signatures of a set of operations) rather than a supporting the inheritance of the class *implementations* (i.e., the instance data fields and operation implementations). Since interface inheritance can be very useful in design and programming, the Java designers introduced a separate mechanism for that type of inheritance.

The Java **`interface`** construct can be used to define an interface for classes separately from the classes themselves. A Java **`interface`** may inherit from (i.e., `extend`) zero or more other **`interface`** definitions.

```
interface Location3D
{   ...   }

interface HumanHolder
{   ...   }

interface Seat extends Location3D, HumanHolder
{   ...   }
```

A Java **`class`** may inherit from (i.e., `implement`) zero or more interfaces as well as inherit from (i.e., `extend`) exactly one other **`class`**.

```
interface BookHolder
{   ...   }

interface BookBasket extends Location3D, BookHolder
{   ...   }

class StudentDesk extends Desk implements Seat, BookBasket
{   ...   }
```

Figure 3.3 shows this interface-based inheritance hierarchy for the classroom simulation example. The dashed lines represent the **`implements`** relationship.

This definition requires the `StudentDesk` class to provide actual implementations for all the operations from the `Location3D`, `HumanHolder`, `BookHolder`, `Seat`,

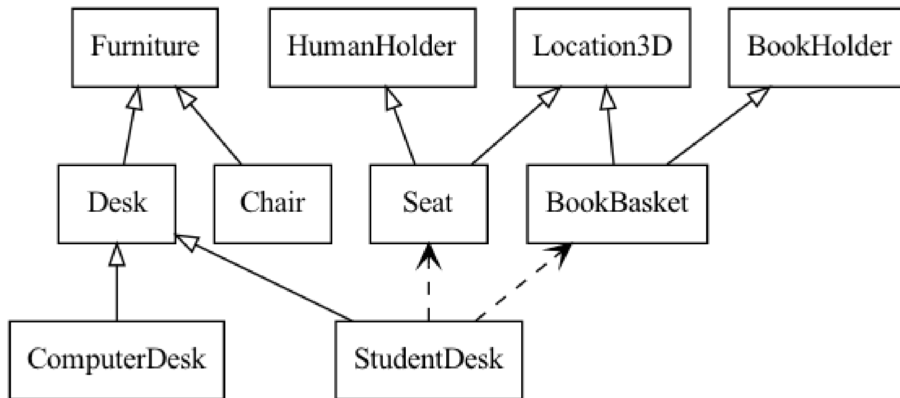


Figure 3.3: Classroom simulation with interfaces.

and `BookBasket` interfaces. The `Location3D` operations will, of course, need to be implemented in such a way that they make sense as part of both the `HumanHolder` and `BookHolder` abstractions.

The Scala `trait` provides a more powerful, and more complex, mechanism than Java's original `interface`. In addition to signatures, a `trait` can define method implementations and data fields. These traits can be added to a class in a controlled, linearized manner to avoid the semantic and implementation problems associated with multiple inheritance of classes. This is called *mixin* inheritance.

Java 8+ generalizes interfaces to allow default implementations of methods.

Most statically typed languages treat subclasses as *subtypes*. That is, if `C` is a subclass of `P`, then the objects of type `C` are also of type `P`. We can *substitute* a `C` object for a `P` object in all cases.

However, the inheritance mechanism in languages in most class-based languages (e.g., Java) does not automatically preserve substitutability. For example, a subclass can change an operation in the subclass to do something totally different from the corresponding operation in the parent class.

3.3.4 Subtype polymorphism

The concept of *polymorphism* (literally "many forms") means the ability to hide different implementations behind a common interface. Polymorphism appears in several forms in programming languages. We will discuss these more later.

Subtype polymorphism (sometimes called *polymorphism by inheritance*, *inclusion polymorphism*, or *subtyping*) means the association of an operation invocation (i.e., procedure or function call) with the appropriate operation implementation in an inheritance (subtype) hierarchy.

This form of polymorphism is usually carried out at run time. That implementation is called *dynamic binding*. Given an object (i.e., class instance) to which an operation is applied, the system will first search for an implementation of the operation associated with the object's class. If no implementation is found in that class, the system will check the superclass, and so forth up the hierarchy until an appropriate implementation is found. Implementations of the operation may appear at several levels of the hierarchy.

The combination of dynamic binding with a well-chosen inheritance hierarchy allows the possibility of an instance of one subclass being substituted for an instance of a different subclass during execution. Of course, this can only be done when none of the extended operations of the subclass are being used.

As an example, again consider the simulation of a classroom. As in our discussion of inheritance, suppose that the `StudentDesk` and `ComputerDesk` classes are derived from the `Desk` class and that a general `move` operation is implemented as a part of the `Desk` class. This could be expressed in Java as follows:

```
class Desk extends Furniture
{   ...
    public void move(...)
    ...
}

class StudentDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}

class ComputerDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}
```

As we noted before, invocation of operations to `move` either a `StudentDesk` or a `ComputerDesk` instance will be bound to the general `move` in the `Desk` class.

Extending the example, suppose that we need a special version of the `move` operation for `ComputerDesk` objects. For instance, we need to make sure that the computer is shut down and the power is disconnected before the entity is moved.

To do this, we can define this special version of the `move` operation and associate it with the `ComputerDesk` class. Now a call to `move`{java} a `ComputerDesk` a object will be bound to the special `move` operation, but a call to `move` a `StudentDesk` object will still be bound to the general `move` operation in the `Desk` class.

The definition of `move` in the `ComputerDesk` class is said to *override* the definition in the `Desk` class.

In Java, this can be expressed as follows:

```
class Desk extends Furniture
{   ...
    public void move(...)
    ...
}

class StudentDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}

class ComputerDesk extends Desk
{   ...
    public void move(...)
    ...
}
```

A class-based language is *object-oriented* if class hierarchies can be incrementally defined by an inheritance mechanism and the language supports polymorphism by inheritance along these class hierarchies.

Object-oriented languages include C++, Java, Scala, C#, Smalltalk, and Ada 95. The language Clu is class-based, but it does not include an inheritance facility.

Other object-oriented languages include Objective C, Object Pascal, Eiffel, and Oberon 2.

3.4 Object-Oriented Paradigm

Now let's consider the object-oriented paradigm more concretely. First, let's review what we mean by an object-oriented language. A language is:

- *object-based* if it supports objects that satisfy the three essential characteristics (state, operations, and identity) as a language feature
- *class-based* if it is object-based, has the concept of class as a language feature, and assigns every object to a class
- *object-oriented* if it is class-based, can define class hierarchies incrementally using an inheritance mechanism, and supports polymorphism by inheritance along these class hierarchies

A class-based language is *object-oriented* if class hierarchies can be incrementally defined by an inheritance mechanism and the language supports polymorphism

by inheritance along these class hierarchies.

3.4.1 Object-oriented Python example

TODO: This example mostly illustates class-based Python. It needs to be extended to show effective use of inheritance and subtyping. Possibly create two differnet subclasses to override the hook methods or leave them abstract and make concrete in subclass–as in Arith and Geom modules for the modular examples.

Python 3 is a dynamically typed language with support for imperative, procedural, modular, object-oriented, and (to a limited extent) functional programming styles [144]. It's object model supports state, operations, identity, and an independent lifecycle. It provides some support for encapsulation. It has classes, single and multiple inheritance, and subtype polymorphism.

Let's again examine the counting problem from Chapter 2 from the standpoint of object-oriented programming in Python 3. The following code defines a class named `CountingOO`. It defines four instance methods and two instance variables.

Note: By *instance variable* and *instance method* we mean variables and instances associated with an object, an instance of a class.

```
class CountingOO:                     # (1)

    def __init__(self,c,m):           # (2,3)
        self.count = c                # (4)
        self.maxc  = m

    def has_more(self,c,m):           # (5)
        return c <= m

    def adv(self):                    # (6)
        self.count = self.count + 1

    def counter(self):                # (7)
        while self.has_more(self.count,self.maxc):
            print(f'{self.count}') # (8)
            self.adv()
```

The following notes explain the numbered items in the above code.

1. By default, a Python 3 class inherits from the cosmic root class `object`. If a class inherits from some other class, then we place the parent class's name in parenthesis after the class name, as with class `Times2` below. (Python 3 supports multiple inheritance, so there can be multiple class names separated by commas.)

2. Python 3 classes do not normally have explicit constructors, but we often

define an initialization method which has the special name `__init__`.

3. Unlike object-oriented languages such as Java, Python 3 requires that the receiver object be passed explicitly as the first parameter of instance methods. By convention, this is a parameter named `self`.

4. An instance of the class `CountingOO` has two instance variables, `count` and `maxc`. Typically, we create these dynamically by explicitly assigning a value to the name. We can access these values in expressions (e.g., `self.count`).

5. Method `has_more()` is a function that takes the receiver object and values for the current count and maximum values and returns `True` if and only there are additional values to generate. (Although an instance method, it does not access the instance's state.)

6. Method `adv()` is a procedure that accesses and modifies the state (i.e., the instance variables), setting `self.count` to a new value closer to the maximum value `self.maxc`.

7. Method `counter()` is a procedure intended as the primary public interface to an instance of the class. It uses function method `has_more()` to determine when to stop the iteration, procedure method `adv()` to advance the variable `count` from one value to the next value, and the `print` function to display the value on the standard output device.

8. Expression `f'{self.count}'` is a Python 3.7 interpolated string.

In terms of the *Template Method* design pattern [83], `counter` is intended as a *template method* that encodes the primary algorithm and is not intended to be overridden. Methods `has_more()` and `adv()` are intended as *hook methods* that are often overriden to give different behaviors to the class.

Consider the following fragment of code.

```
ctr = CountingOO(0,10)
ctr.counter()
```

The first line above creates an instance of the `CountingOO` class, initializes its instance variables `count` and `maxc` to 0 and 10, and stores the referene in variable `ctr`. The call `ctr.counter()` thus prints the values 0 to 10, one per line, as do the programs from Chapter 2.

However, we can create a subclass that overrides the definitions of the hook methods `has_more()` and `adv()` to give quite different behavior without modifying class `CountingOO`.

```
class Times2(CountingOO):    # inherits from CountingOO

    def has_more(self,c,m):  # overrides
        return c != 0 and abs(c) <= abs(m)
```

```
    def adv(self):           # overrides
        self.count = self.count * 2
```

Now consider the following code fragment.

```
ctr2 = Times2(-1,10)
ctr2.counter()
```

This generates the sequence of values -1, -2, -4, and -8, printed one per line.

The call to any method on an instance of class `Times2` is polymorphic. The system dynamically searches up the class hierarchy from `Times2` to find the appropriate function. It finds `has_more()` and `adv()` in `Times2` and `counter()` in parent class `CountingOO`.

The code for this section is in source file `CountingOO.py`.

3.4.2 Object-oriented Scala example

The program `CountingOO.scala` is an object-oriented Scala [131,151] program similar to the Python version given above.

3.5 Prototype-based Paradigm

Classes and inheritance are not the only way to support relationships among objects in object-based languages. Another approach of growing importance is the use of *prototypes*.

3.5.1 Prototype concepts

A *prototype-based* language does not have the concept of class as defined above. It just has objects. Instead of using a class to instantiate a new object, a program copies (or clones) an existing object—the *prototype*—and modifies the copy to have the needed attributes and operations.

Each prototype consists of a collection of *slots*. Each slot is filled with either a data attribute or an operation.

This cloning approach is more flexible than the class-based approach.

In a class-based language, we need to define a new class or subclass to create a variation of an existing type. For example, we may have a `Student` class. If we want to have students who play chess, then we would need to create a new class, say `ChessPlayingStudent`, to add the needed data attributes and operations.

Aside: Should `Student` be the parent `ChessPlayingStudent`? or should `ChessPlayer` be the parent? Or should we have fields of `ChessPlayingStudent` that hold `Student` and `ChessPlayer` objects?

In a class-based language, the boundaries among categories of objects specified by classes should be *crisply defined*. That is, an object is in a particular class or

it is not. Sometimes this crispness may be unnatural.

In a prototype-based language, we simply clone a student object and add new slots for the added data and operations. This new object can be a prototype for further objects.

In a prototype-based language, the boundaries between categories of objects created by cloning may be fuzzy. One category of objects may tend to blend into others. Sometimes this fuzziness may be more natural.

Consider categories of people associated with a university. These categories may include `Faculty`, `Staff`, `Student`, and `Alumnus`. Consider a *student* who gets a BSCS degree, then accepts a *staff* position as a programmer and stays a student by starting an MS program part-time, and then later teaches a course as a graduate student. The same person who started as a student thus evolves into someone who is in several categories later. And he or she may also be a chess player.

Instead of static, class-based inheritance and polymorphism, some languages exhibit prototype-based *delegation*. If the appropriate operation cannot be found on the current object, the operation can be delegated to its prototype, or perhaps to some other related, object. This allows dynamic relationships along several dimensions. It also means that the "copying" or "cloning" may be partly logical rather than physical.

Prototypes and delegation are more basic mechanisms than inheritance and polymorphism. The latter can often be implemented (or perhaps "simulated") using the former.

Self [158,175], NewtonScript [123,130], JavaScript [4,231], Lua [105,116,165], and Io [36,62,164] are prototype-based languages. (Package `prototype.py` can also make Python behave in a prototype-based manner.)

Let's look at Lua as a prototype-based language.

Note: The two most widely used prototype languages are JavaScript and Lua. I choose Lua here because it is simpler and can also execute conveniently from the command line. I have also used Lua extensively in the past and have not yet used JavaScript extensively.

3.5.2 Lua as an object-based language

Lua is a dynamically typed, multiparadigm language [105,116]. The language designers stress the following design principles [117]:

- portability
- embeddability
- efficiency
- simplicity

To realize these principles, the core language implementation:

- can only use standard C and the standard C library
- must be efficient in use of memory and processor time (i.e., keep the interpreter small and fast)
- must support interoperability with C programs in both directions (i.e., can call or be called by C programs)

C is ubiquitous, likely being the first higher-level language implemented for any new machine, whether a small microcontroller or a large multiprocessor. So this implementation approach supports the portability, embeddability, and efficiency design goals.

Because of Lua's strict adherence to the above design principles, it has become a popular language for extending other applications with user-written scripts or templates. For example, it is used for this purpose in some computer games and by Wikipedia. Also, Pandoc, the document conversion tool used in production of this textbook, enables scripts to be written in Lua. (The Pandoc program itself is written in Haskell.)

The desire for a simple but powerful language led the designers to adopt an approach that *separates mechanisms from policy*. As noted on the Lua website [117]:

> A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. For example, although Lua is not a pure object-oriented language, it does provide meta-mechanisms for implementing classes and inheritance. Lua's meta-mechanisms bring an economy of concepts and keep the language small, while allowing the semantics to be extended in unconventional ways.

Lua provides a small set of quite powerful primitives. For example, it includes only one data structure—the *table* (dictionary, map, or object in other languages)—but ensures that it is efficient and flexible for a wide range of uses.

Lua's tables are *objects* as described in Section 3.3. Each object has its own:

- *state* (i.e., values associated with keys)
- *identity* independent of state
- *lifecycle* independent of the code that created it

In addition, a table can have its own *operations* by associating function closures with keys.

Note: By *function closure*, we mean the function's definition plus aspects of its environment necessary (e.g., variables variables outside the function) necessary for the function to be executed.

So a key in the table represents a *slot* in the object. The slot can be occupied by either a data attribute's value or the function closure associated with an operation.

Lua tables do not directly support *encapsulation*, but there are ways to build structures that encapsulate key data or operations.

Lua's *metatable* mechanism, particularly the `__index` *metamethod*, enables an access to an undefined key to be delegated to another table (or to result in a call of a specified function).

Thus tables and metatables enable the prototype-based paradigm as illustrated in the next section.

As in Python 3, Lua requires that the *receiver* object be passed as an argument to object-based function and procedure calls. By convention, it is passed as the first argument, as shown below.

```
obj.method(obj, other_arguments)
```

Lua has a bit of *syntactic sugar*—the `:` operator—to make this more convenient. The following Lua expression is equivalent to the above.

```
obj:method(other_arguments)
```

The Lua interpreter evaluates the expression `obj` to get the receiver object (i.e., table), then retrieves the function closure associated with the key named `method` from the receiver object, then calls the function, passing the receiver object as its first parameter. In the body of the function definition, the receiver object can be referenced by parameter name `self`.

We can use a similar notation to define functions to be methods associated with objects (tables).

3.5.3 Prototype-based Lua example

The Lua code below, from file `CountingPB.lua`, implements a Lua module similar to the Python 3 `CountingOO` class given in Section 3.4.1. It illustrates how to define Lua modules as well as prototypes.

```
-- File CountingPB.lua
local CountingPB = {count = 1, maxc = 0} -- (1)

function CountingPB:new(mixin)           -- (2)
   mixin = mixin or {}                   -- (5)
   local obj = { __index = self }        -- (4)
   for k, v in pairs(mixin) do           -- (5)
      if k ~= "__index" then
         obj[k] = v
      end
   end
   return setmetatable(obj,obj)          -- (6,7)
end
```

```
function CountingPB:has_more(c,m)          -- (2)
   return c <= m
end

function CountingPB:adv()                  -- (2)
   self.count = self.count + 1
end

function CountingPB:counter()              -- (2)
   while self:has_more(self.count,self.maxc) do
      print(self.count)
      self:adv()
    end
end

return CountingPB                          -- (3)
```

The following notes explain the numbered steps in the above code.

1. Create module object `CountingPB` as a Lua table with default values for data attributes `count` and `maxc`. This object is also the top-level prototype object.
2. Define methods (i.e., functions) `new()`, `has_more()`, `adv()`, and `counter()` and add them to the `CountingPB` table. The key is the function's name and the value is the function's closure.

   Method `new()` is the constructor for clones.
3. Return `CountingPB` when the module file `CountingPB.lua` is imported with a `require` call in another Lua module or script file.

Method `new` is what constructs the clones. This method:

4. Creates the clone initially as a table with only the `__index` set to the object that called `new` (i.e., the receiver object `self`).
5. Copies the method `new`'s parameter `mixin`'s table entries into the clone. This enables existing data and method attributes of the receiver object `self` to be redefined and new data and method attributes to be added to the clone.

   If parameter `mixin` is undefined or an empty table, then no changes are made to the clone.
6. Sets the clone's metatable to be the clone's table itself. In step 4, we had set its metamethod `__index` to be the receiver object `self`.
7. Returns the clone object (a table) as is the convention for Lua modules.

If a Lua program accesses an undefined key of a table (or object), then the interpreter checks to see whether the table has a metatable defined.

- If no metatable is set, then the result of the access is a `nil` (meaning undefined).
- If a metatable is set, then the interpreter uses the `__index` metamethod to determine what to do. If `__index` is a table, then the access is delegated to that table. If `__index` is set a function closure, then the interpreter calls that function. If there is no `__index`, then it returns a `nil`.

We can load the `CountingPB.lua` module as follows:

```
local CountingPB = require "CountingPB"
```

Now consider the Lua assignment below:

```
x = CountingPB:new({count = 0, maxc = 10})
```

This creates a clone of object `CountingPB` and stores it in variable `x`. This clone has its own data attributes `count` and `maxc`, but it delegates method calls back to object `CountingPB`.

If we execute the call `x:counter()`, we get the following output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Now consider the Lua assignment:

```
y = x:new({count = 10, maxc = 15})
```

This creates a clone of object in `x` and stores the clone in variable `y`. The `y` object has different values for `count` and `maxc`, but it delegates the method calls to `x`, which, in turn, delegates them on to `CountingPB`.

If we execute the call `y:counter()`, we get the following output:

```
10
11
12
13
14
15
```

Now, consider the following Lua assignment:

```lua
z = y:new( { maxc = 400,
             has_more = function (self,c,m)
                return c ~= 0 and math.abs(c) <= math.abs(m)
             end,
             adv = function(self)
                self.count = self.count * 2
             end,
             bye = function(self) print(self.msg) end
             msg = "Good-Bye!" } )
```

This creates a clone of object `y` that keeps `x`'s current value of `count` (which is 16 after executing `y:counter()`), sets a new value of `maxc`, overrides the definitions of methods `has_more()` and `adv()`, and defines new method `bye()` and new data attribute `msg`.

If we execute the call `z:counter()` followed by `z:bye()`, we get the following output:

```
16
32
64
128
256
Good-Bye!
```

The Lua source code for this example is in file `CountingPB.lua`. The example calls are in file `CountingPB_Test.lua`.

3.5.4 Observations

How does the prototype-based (PB) paradigm compare with the object-oriented (OO) paradigm?

- The OO paradigm as implemented in a language usually enforces a particular discipline or policy and provides syntactic and semantic support for that policy. However, it makes programming outside the policy difficult.
- The PB paradigm is more flexible. It provides lower-level mechanisms and little or no direct support for a particular discipline or policy. It allows programmers to define their own policies, simple or complex policies depending on the needs. These policies can be implemented in libraries and reused. However, PB can result in different programmers or different software development shops using incompatible approaches.

Whatever paradigm we use (OO, PB, procedural, functional, etc.), we should be careful and be consistent in how we design and implement programs.

3.6 What Next?

In Chapters 2 and 3 (this chapter), we explored various programming paradigms.

In Chapter 4, we begin examining Haskell, looking at our first simple programs and how to execute those programs with the interactive interpreter.

In subsequent chapters, we look more closely at the concepts of type introduced in this chapter and abstraction introduced in the previous chapter.

3.7 Exercises

1. This chapter used Python 3 to illustrate the object-oriented paradigm. Choose a language such as Java, C++, or C#. Describe how it can be used to write programs in the object-oriented paradigm. Show the `CountingOO` example in the chosen language.

2. C is a primarily procedural language. Describe how C can be used to implement object-based programs. Show the `CountingOO` example in the chosen language.

3.8 Acknowledgements

Beginning in Summer 2016 and continuing through Fall 2018, I adapted and revised much of this chapter from my previous course materials:

- I adapted the Object-Oriented programming paradigm section from my notes *Introduction to Object Orientation* [52], which I wrote originally for the first C++ (CSci 490) and Java-based (CSci 211) classes at UM in 1996 but expanded and adapted for other courses. These notes—and my approach to object-oriented design and programming in general—were influenced by my study of works by Horstmann [99,100], Budd [22,23], Meyer [128], Thomas [170], Wirfs-Brock [224,225], Beck [11], Bellin [12], the "Gang of Four" [83], Buschmann [25], Fishwick [74], Johnson [106], Schmid [[154]; Schmid1999a], and many other sources (lost in the mists of time).

  Note: In particular, chapters 3-6 of Horstmann's C++ book [99]—on "Implementing Classes", "Interfaces", "Object-Oriented Design", and "Invariants"—considerably influenced my views on object-oriented design and programming as I was learning the approach. Similarly, the first edition of Horstmann and Cornell's *Core Java* [100] influenced my approach to OOP in Java.

- I expanded the Prototype-based programming paradigm section from my 2016 draft notes [43] and Lua example [44] on that topic. These notes and example were influenced by Craig [38], Ierusalimschy [105], Ungar [175], and other sources (e.g., the Lua website [116] and the Wikipedia article "Prototype-based Programming" [211]) and by my experiences developing and my teaching Lua-based courses.

  Note: I chose Lua instead of Javascript because the former is simpler and "cleaner" and could easily execute from the command line. I also

> had considerably more Lua programming experience. I had used it in my Software Language Engineering (CSci 658) course in the Fall 2013 and the Fall 2014 and Fall 2016 offerings of Organization of Programming Languages (CSci 450). (Initially, I planned to use Lua as the programming language for the interpreters in this textbook. However, my experiences with CSci 450 in Fall 2016 suggested that I needed more mature teaching materials for the Fall 2017 class. I decided to instead expand my relatively mature *Notes on Functional Programming with Haskell* [42] and redesign the interpreters to use Haskell.)
>
> TODO: Perhaps add citation for 2013 CSCI 658 course materials? Also identify which Javascript book had influence on my thinking?

In 2017, I incorporated this material into Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In 2018 I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming* (ELIFP). These are Chapter 1, Evolution of Programming Languages; Chapter 2, Programming Paradigms; Chapter 3, Object-Based Paradigms (this chapter); and Chapter 80, Review of Relevant Mathematics.

In 2018, I added the new Python 3 [144] examples in ELIFP Chapters 2 and 3. My approach to Python 3 programming are influenced by Beazley [8], Ramalho [146], and other sources and by my experiences developing and teaching two graduate courses that used Python 3 in 2018.

In 2018, I also revised the discussion of the Lua example and incorporated it into this chapter.

I used Chapter 2 in CSci 450 in Fall 2018 but not this chapter. However, I used both chapters in my Python-based Multiparadigm Programming (CSci 556) course in Fall 2018.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

3.9 Terms and Concepts

Object (state, operations, identity, encapsulation, independent lifecycle, mutable, immutable), object-based language, class, type, class-based language, inheritance, subtype, interface, polymorphism, subtype polymorphism (subtyping,

inclusion polymorphism, polymorphism by inheritance), dynamic binding, object-oriented language, prototype, clone, slot, delegation, prototype-based language, embeddability, Lua tables, metatables, and metamethods, function closure.

4 First Haskell Programs

4.1 Chapter Introduction

The goals of this chapter are to

- introduce the definition of Haskell functions using examples
- illustrate the use of the `ghci` interactive REPL (Read-Evaluate-Print Loop) interpreter

4.2 Defining Our First Haskell Functions

Let's look at our first function definition in the Haskell language, a program to implement the factorial function for natural numbers.

The Haskell source file `Factorial.hs` holds the Haskell function definitions for this chapter. The test script is in source file `TestFactorial.hs`; it is discussed further in Chapter 12 on testing of Haskell programs.

4.2.1 Factorial function specification

We can give two mathematical definitions of factorial, *fact* and *fact'*, that are equivalent for all natural number arguments. We can define *fact* using the product operator as follows:

$$\mathit{fact}(n) = \prod_{i=1}^{i=n} i$$

For example,

$$\mathit{fact}(4) = 1 \times 2 \times 3 \times 4.$$

By definition

$$\mathit{fact}(0) = 1$$

which is the *identity* element of the multiplication operation.

We can also define the factorial function *fact'* with a *recursive* definition (or *recurrence relation*) as follows:

$$\mathit{fact'}(n) = 1,\ \mathit{if}\ n = 0$$
$$\mathit{fact'}(n) = n \times\ \mathit{fact'}(n-1),\ \mathit{if}\ n \geq 1$$

Since the domain of *fact'* is the set of natural numbers, a set over which induction is defined, we can easily see that this recursive definition is well defined.

- For $n = 0$, the base case, the value is simply 1.
- For $n \geq 1$, the value of $\mathit{fact'}(n)$ is recursively defined in terms of $\mathit{fact'}(n-1)$. The argument of the recursive application decreases toward the base case.

In the Review of Relevant Mathematics appendix, we prove that $\mathit{fact}(n) = \mathit{fact'}(n)$ by mathematical induction.

The Haskell functions defined in the following subsections must compute $fact(n)$ when applied to argument value $n \geq 0$.

4.2.2 Factorial function using if-then-else: `fact1`

One way to translate the recursive definition *fact'* into Haskell is the following:

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
          else
              n * fact1 (n-1)
```

- The first line above is the *type signature* for function `fact1`. In general, type signatures have the syntax *object* `::` *type*.

  Haskell type names begin with an uppercase letter.

  The above defines object `fact1` as a function (denoted by the `->` symbol) that takes one argument of type integer (denoted by the first `Int`) and returns a value of type integer (denoted by the last `Int`).

  Haskell does not have a built-in natural number type. Thus we choose type `Int` for the argument and result of `fact1`.

  The `Int` data type is a bounded integer type, usually the integer data type supported directly by the host processor (e.g., 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer (-2^{29} to 2^{29}).

- The declaration for the function `fact1` begins on the second line. Note that it is an equation of the form

  fname parms `=` *body*

  where *fname* is the function's name, *parms* are the function's parameters, and *body* is an expression defining the function's result.

  Function and variable names begin with lowercase letters optionally followed by a sequence of characters each of which is a letter, a digit, an apostrophe (`'`) (sometimes pronounced "prime"), or an underscore (`_`).

  A function may have zero or more parameters. The parameters are listed after the function name without being enclosed in parentheses and without commas separating them.

  The parameter names may appear in the *body* of the function. In the evaluation of a function *application* the actual argument values are substituted for parameters in the *body*.

- Above we define the *body* function `fact1` to be an `if-then-else` *expression*. This kind of expression has the form

`if` *condition* `then` *expression1* `else` *expression2*

where

condition is a Boolean expression, that is, an expression of Haskell type `Bool`, which has either `True` or `False` as its value

expression1 is the expression that is returned when the condition is `True`

expression2 is the expression (with the same type as *expression1*) that is returned when the condition is `False`

Evaluation of the `if-then-else` expression in `fact1` yields the value `1` if argument `n` has the value `0` (i.e., `n == 0`) and yields the value `n * fact1 (n-1)` otherwise.

- The `else` clause includes a recursive application of `fact1`. The whole expression `(n-1)` is the argument for the recursive application, so we enclose it in parenthesis.

  The value of the argument for the recursive application is less than the value of the original argument. For each recursive application of `fact` to a natural number, the argument's value thus moves closer to the termination value `0`.

- Unlike most conventional languages, the *indentation is significant* in Haskell. The indentation indicates the nesting of expressions.

  For example, in `fact1` the `n * fact1 (n-1)` expression is nested inside the `else` clause of the `if-then-else` expression.

- This Haskell function does not match the mathematical definition given above. What is the difference?

  Notice the domains of the functions. The evaluation of `fact1` will go into an "infinite loop" and eventually abort when it is applied to a negative value.

In Haskell there is *only* one way to form more complex expressions from simpler ones: *apply* a function.

Neither parentheses nor special operator symbols are used to denote function application; it is denoted by simply listing the argument expressions following the function name. For example, a function `f` applied to argument expressions `x` and `y` is written in the following *prefix* form:

```
f x y
```

However, the usual prefix form for a function application is not a convenient or natural way to write many common expressions. Haskell provides a helpful bit of syntactic sugar, the *infix* expression. Thus instead of having to write the addition of `x` and `y` as

```
add x y
```

we can write it as

```
x + y
```

as we have since elementary school. Here the symbol + represents the addition function.

Function application (i.e., juxtaposition of function names and argument expressions) has higher precedence than other operators. Thus the expression `f x + y` is the same as `(f x) + y`.

4.2.3 Factorial function using guards: `fact2`

An alternative way to differentiate the two cases in the recursive definition is to use a different equation for each case. If the Boolean *guard* (e.g., `n == 0`) for an equation evaluates to true, then that equation is used in the evaluation of the function. A guard is written following the `|` symbol as follows:

```
fact2 :: Int -> Int
fact2 n
    | n == 0    = 1
    | otherwise = n * fact2 (n-1)
```

Function `fact2` is equivalent to the `fact1`. Haskell evaluates the guards in a top-to-bottom order. The `otherwise` guard always succeeds; thus it's use above is similar to the trailing `else` clause on the `if-then-else` expression used in `fact1`.

4.2.4 Factorial function using pattern matching: `fact3` and `fact4`

Another equivalent way to differentiate the two cases in the recursive definition is to use *pattern matching* as follows:

```
fact3 :: Int -> Int
fact3 0 = 1
fact3 n = n * fact3 (n-1)
```

The parameter pattern `0` in the first *leg* of the definition only matches arguments with value 0. Since Haskell checks patterns and guards in a top-to-bottom order, the `n` pattern matches all nonzero values. Thus `fact1`, `fact2`, and `fact3` are equivalent.

To stop evaluation from going into an "infinite loop" for negative arguments, we can remove the negative integers from the function's domain. One way to do this is by using guards to narrow the domain to the natural numbers as in the definition of `fact4` below:

```
fact4 :: Int -> Int
fact4 n
```

```
    | n == 0 =  1
    | n >= 1 =  n * fact4 (n-1)
```

Function `fact4` is undefined for negative arguments. If `fact4` is applied to a negative argument, the evaluation of the program encounters an error quickly and returns without going into an infinite loop. It prints an error and halts further evaluation.

We can define our own error message for the negative case using an `error` call as in `fact4'` below.

```
fact4' :: Int -> Int
fact4' n
    | n == 0    =  1
    | n >= 1    =  n * fact4' (n-1)
    | otherwise = error "fact4' called with negative argument"
```

In addition to displaying the custom error message, this also displays a stack trace of the active function calls.

4.2.5 Factorial function using built-in library function: `fact5`

The four definitions we have looked at so far use recursive patterns similar to the recurrence relation *fact'*. Another alternative is to use the library function `product` and the list-generating expression `[1..n]` to define a solution that is like the function *fact*:

```
fact5 :: Int -> Int
fact5 n = product [1..n]
```

The list expression `[1..n]` generates a *list* of consecutive integers beginning with `1` and ending with `n`. We study lists beginning with Chapter 13.

The library function `product` computes the product of the elements of a finite list.

If we apply `fact5` to a negative argument, the expression `[1..n]` generates an empty list. Applying `product` to this empty list yields 1, which is the identity element for multiplication. Defining `fact5` to return 1 is consistent with the function *fact* upon which it is based.

Which of the above definitions for the factorial function is better?

Most people in the functional programming community would consider `fact4` (or `fact4'`) and `fact5` as being better than the others. The choice between them depends upon whether we want to trap the application to negative numbers as an error or to return the value 1.

4.2.6 Testing

Chapter 12 discusses testing of the Factorial module designed in this chapter. The test script is `TestFactorial.hs`.

4.3 Using the Glasgow Haskell Compiler (GHC)

See the Glasgow Haskell Compiler Users Guide [92] for information on the Glasgow Haskell Compiler (GHC) and its use.

GHCi is an environment for using GHC interactively. That is, it is a REPL (Read-Evaluate-Print-Loop) command line interface using Haskell. The "Using GHCi" chapter [93] of the GHC User Guide [92] describes its usage.

Below, we show a GHCi session where we load source code file (module) `Factorial.hs` and apply the factorial functions to various inputs. The instructor ran this in a Terminal session on an iMac running macOS 10.13.4 (High Sierra) with `ghc` 8.4.3 installed.

1. Start the REPL.

   ```
   bash-3.2$ ghci
   GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
   ```

2. Load module `Fact` that holds the factorial function definitions. This assumes the `Factorial.hs` file is in the current directory. The `load` command can be abbreviated as just `:l`.

   ```
   Prelude> :load Factorial
   [1 of 1] Compiling Factorial        ( Factorial.hs, interpreted )
   Ok, one module loaded.
   ```

3. Inquire about the type of `fact1`.

   ```
   *Factorial> :type fact1
   fact1 :: Int -> Int
   ```

4. Apply function `fact1` to 7, 0, 20, and 21. Note that the factorial of 21 exceeds the `Int` range.

   ```
   *Factorial> fact1 7
   5040
   *Factorial> fact1 0
   1
   *Factorial> fact1 20
   2432902008176640000
   *Factorial> fact1 21
   -4249290049419214848
   ```

5. Apply functions `fact2`, `fact3`, `fact4`, and `fact5` to 7.

```
*Factorial> fact2 7
5040
*Factorial> fact3 7
5040
*Factorial> fact4 7
5040
*Factorial> fact5 7
5040
```

6. Apply functions `fact1`, `fact2`, and `fact3` to -1. All go into an infinite recursion, eventually terminating with an error when the runtime stack overflows its allocated space.

```
*Factorial> fact1 (-1)
*** Exception: stack overflow
*Factorial> fact2 (-1)
*Factorial> fact3 (-1)
*** Exception: stack overflow
```

7. Apply functions `fact4` and `fact4'` to -1. They quickly return with an error.

```
*Factorial> fact4 (-1)
*** Exception: Factorial.hs:(54,1)-(56,29):
  Non-exhaustive patterns in function fact4
*Factorial> fact4' (-1)
*** Exception: fact4' called with negative argument
CallStack (from HasCallStack):
  error, called at Factorial.hs:64:17 in main:Factorial
```

8. Apply function `fact5` to -1. It returns a 1 because it is defined for negative integers.

```
*Factorial> fact5 (-1)
1
```

9. Set the `+s` option to get information about the time and space required and the `+t` option to get the type of the returned value.

```
*Factorial> :set +s
*Factorial> fact1 20
2432902008176640000
(0.00 secs, 80,712 bytes)
*Factorial> :set +t
*Factorial> fact1 20
2432902008176640000
it :: Int
(0.05 secs, 80,792 bytes)
*Factorial> :unset +s +t
```

```
*Factorial> fact1 20
2432902008176640000
```

10. Exit GHCi.

```
:quit
Leaving GHCi.
```

Suppose we had set the environment variable `EDITOR` to our favorite text editor in the Terminal window. For example, on a MacOS system, your instructor might give the following command in shell (or in a startup script such as `.bash_profile`):

```
export EDITOR=Aquamacs
```

Then the `:edit` command within GHCi allows us to edit the source code. We can give a filename or default to the last file loaded.

```
:edit
```

Or we could also use a `:set` command to set the editor within GHCi.

```
:set editor Aquamacs
...
:edit
```

See the Glasgow Haskell Compiler (GHC) User's Guide [92] for more information about use of GHC and GHCi.

4.4 What Next?

In this chapter (4), we looked at our first Haskell functions and how to execute them using the Haskell interpreter.

In Chapter 5, we continue our exploration of Haskell by examining its built-in types.

4.5 Chapter Source Code

The Haskell source module `Factorial.hs` gives the factorial functions used in this chapter. The test script in source file `TestFactorial.hs` is discussed further in Chapter 12 on testing of Haskell programs.

4.6 Exercises

1. Reimplement functions `fact4` and `fact5` with type `Integer` instead of `Int`. `Integer` is an unbounded precision integer type (discussed in the next chapter). Using `ghci`, execute these functions for values -1, 7, 20, 21, and 50 using `ghci`.

2. Develop both recursive and iterative (looping) versions of a factorial fuunction in an imperative language (e.g., Java, C++, Python 3, etc.)

4.7 Acknowledgements

In Summer 2016, I adapted and revised much of this work in from Chapter 3 of my *Notes on Functional Programming with Haskell* [42] and incorporated it into Chapter 2, Basic Haskell Functional Programming, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs (this chapter); previous Section 2.4 became Section 5.3 in the new Chapter 5, Types; and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction, and Chapter 7, Data Abstraction.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

4.8 Terms and Concepts

TODO: Update

Factorials, function definition and application, recursion, function domains, `error`, `if`, guards, basic types (`Int`, `Integer`, `Bool`, library (Prelude) functions, REPL, `ghci` commands and use.

5 Types

5.1 Chapter Introduction

The goals of this chapter are to:

- examine the general concepts of type systems
- explore Haskell's builtin types

5.2 Type System Concepts

The term *type* tends to be used in many different ways in programming languages. What is a type?

The chapter on object-based paradigms discusses the concept of type in the context of object-oriented languages. This chapter first examines the concept more generally and then examines Haskell's builtin types.

5.2.1 Types and subtypes

Conceptually, a *type* is a set of values (i.e., possible states or objects) and a set of operations defined on the values in that set.

Similarly, a type `S` is (a behavioral) *subtype* of type `T` if the set of values of type `S` is a "subset" of the values in set `T` an set of operations of type `S` is a "superset" of the operations of type `T`. That is, we can safely *substitute* elements of subtype `S` for elements of type `T` because `S`'s operations behave the "same" as `T`'s operations.

This is known as the *Liskov Substitution Principle* [119,205].

Consider a type representing all furniture and a type representing all chairs. In general, we consider the set of chairs to be a subset of the set of furniture. A chair should have all the general characteristics of furniture, but it may have additional characteristics specific to chairs.

If we can perform an operation on furniture in general, we should be able to perform the same operation on a chair under the same circumstances and get the same result. Of course, there may be additional operations we can perform on chairs that are not applicable to furniture in general.

Thus the type of all chairs is a subtype of the type of all furniture according to the Liskov Substitution Principle.

5.2.2 Constants, variables, and expressions

Now consider the types of the basic program elements.

A *constant* has whatever types it is defined to have in the context in which it is used. For example, the constant symbol `1` might represent an integer, a real

number, a complex number, a single bit, etc., depending upon the context.

A *variable* has whatever types its value has in a particular context and at a particular time during execution. The type may be constrained by a declaration of the variable.

An *expression* has whatever types its evaluation yields based on the types of the variables, constants, and operations from which it is constructed.

5.2.3 Static and dynamic

In a *statically typed language*, the types of a variable or expression can be determined from the program source code and checked at "compile time" (i.e., during the syntactic and semantic processing in the front-end of a language processor). Such languages may require at least some of the types of variables or expressions to be *declared* explicitly, while others may be *inferred* implicitly from the context.

Java, Scala, and Haskell are examples of statically typed languages.

In a *dynamically typed language*, the specific types of a variable or expression cannot be determined at "compile time" but can be checked at runtime.

Lisp, Python, JavaScript, and Lua are examples of dynamically typed languages.

Of course, most languages use a mixture of static and dynamic typing. For example, Java objects defined within an inheritance hierarchy must be bound dynamically to the appropriate operations at runtime. Also Java objects declared of type `Object` (the root class of all user-defined classes) often require explicit runtime checks or coercions.

5.2.4 Nominal and structural

In a language with *nominal typing*, the type of value is based on the type *name* assigned when the value is created. Two values have the same type if they have the same type name. A type `S` is a subtype of type `T` only if `S` is explicitly declared to be a subtype of `T`.

For example, Java is primarily a nominally typed language. It assigns types to an object based on the name of the class from which the object is instantiated and the superclasses extended and interfaces implemented by that class.

However, Java does not guarantee that subtypes satisfy the Liskov Substitution Principle. For example, a subclass might not implement an operation in a manner that is compatible with the superclass. (The behavior of subclass objects are this different from the behavior of superclass objects.) Ensuring that Java subclasses preserve the Substitution Principle is considered good programming practice in most circumstances.

In a language with *structural typing*, the type of a value is based on the *structure* of the value. Two values have the same type if they have the "same" structure;

that is, they have the same *public* data attributes and operations and these are themselves of compatible types.

In structurally typed languages, a type `S` is a subtype of type `T` only if `S` has all the public data values and operations of type `T` and the data values and operations are themselves of compatible types. Subtype `S` may have additional data values and operations not in `T`.

Haskell is primarily a structurally typed language.

5.2.5 Polymorphic operations

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (or operator symbols) are associated with implementations of the functions (or operations).

In general, two primary kinds of polymorphism exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at *compile time* based on the declared types of the entities. They exhibit *early binding*.

      Consider the language Java. It overloads a few operator symbols, such as using the + symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

      Haskell's *type class* mechanism, which we examine in a later chapter, implements overloading polymorphism in Haskell. There are similar mechanisms in other languages such as Scala and Rust.

   b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate

implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*. This the polymorphism associated with the class structure in Java.

Haskell does not support subtyping. Its type classes do support *class extension*, which enables one class to inherit the properties of another. However, Haskell's classes are not types.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

   The object-oriented programming (e.g., Java) community often calls this type of polymorphism *generics* or *generic programming*.

   The functional programming (e.g., Haskell) community often calls this simply *polymorphism*.

5.2.6 Polymorphic variables

A *polymorphic variable* is a variable that can "hold" values of different types during program execution.

For example, a variable in a dynamically typed language (e.g., Python) is polymorphic. It can potentially "hold" any value. The variable takes on the type of whatever value it "holds" at a particular point during execution.

Also, a variable in a nominally and statically typed, object-oriented language (e.g., Java) is polymorphic. It can "hold" a value its declared type or of any of the subtypes of that type. The variable is declared with a static type; its value has a dynamic type.

A variable that is a parameter of a (parametrically) polymorphic function is polymorphic. It may be bound to different types on different calls of the function.

5.3 Basic Haskell Types

The type system is an important part of Haskell; the compiler or interpreter uses the type information to detect errors in expressions and function definitions. To each expression Haskell assigns a *type* that describes the kind of value represented by the expression.

Haskell has both built-in types (defined in the language or its standard libraries) and facilities for defining new types. In the following we discuss the primary built-in types. As we have seen, a Haskell type name begins with a capital letter.

In this textbook, we sometimes refer to the types `Int`, `Float`, `Double`, `Bool`, and `Char` as being *primitive* because they likely have direct support in the host processor's hardware.

5.3.1 Integers: `Int` and `Integer`

The `Int` data type is usually an integer data type supported directly by the host processor (e.g., 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer.

The type `Integer` is an unbounded precision integer type. Unlike `Int`, host processors usually do not support this type directly. The Haskell library or runtime system typically supports this type in software.

Haskell integers support the usual literal formats (i.e., constants) and typical operations:

- Infix binary operators such as `+` (addition), `-` (subtraction), `*` (multiplication), and `^` (exponentiation)
- Infix binary comparison operators such as `==` (equality of values), `/=` (inequality of values), `<`, `<=`, `>`, and `>=`
- Unary operator `-` (negation)

For integer division, Haskell provides two-argument functions:

- `div` such that `div m n` returns the integral quotient *truncated toward negative infinity* from dividing `m` by `n`
- `quot` such that `quot m n` returns the integral quotient *truncated toward 0* from dividing `m` bem `n`
- `mod` (i.e., modulo) and `rem` (i.e., remainder) such that

```
(div m n) * n + mod m n == m
(quot m n)* n + rem m n == m
```

To make these definitions more concrete, consider the following examples. Note that the result of `mod` has the same sign as the divisor and `rem` has the same sign as the dividend.

```
div  7 3  == 2
quot 7 3  == 2
mod  7 3  == 1          -- same sign as divisor
rem  7 3  == 1          -- same sign as dividend

div  (-7) (-3) == 2
quot (-7) (-3) == 2
mod  (-7) (-3) == (-1)  -- same sign as divisor
rem  (-7) (-3) == (-1)  -- same sign as dividend
```

```haskell
div  (-7) 3 == (-3)
quot (-7) 3 == (-2)
mod  (-7) 3 == 2        -- same sign as divisor
rem  (-7) 3 == (-1)     -- same sign as dividend

div  7 (-3) == (-3)
quot 7 (-3) == (-2)
mod  7 (-3) == (-2)     -- same sign as divisor
rem  7 (-3) == 1        -- same sign as dividend
```

Haskell also provides the useful two-argument functions `min` and `max`, which return the minimum and maximum of the two arguments, respectively.

Two-arguments functions such as `div`, `rem`, `min`, and `max` can be applied in infix form by including the function name between backticks as shown below:

```haskell
5 `div` 3   -- yields 1
5 `rem` 3   -- yields 2
5 `min` 3   -- yields 3
5 `max` 3   -- yields 5
```

5.3.2 Floating point numbers: `Float` and `Double`

The `Float` and `Double` data types are usually the single and double precision floating point numbers supported directly by the host processor.

Haskell floating point literals must include a decimal point; they may be signed or in scientific notation: `3.14159`, `2.0`, `-2.0`, `1.0e4`, `5.0e-2`, `-5.0e-2`.

Haskell supports the usual operations on floating point numbers. Division is denoted by / as usual.

In addition, Haskell supports the following for converting floating point numbers to and from integers:

- `floor` returns the largest integer less than its floating point argument.
- `ceiling` returns the smallest integer greater than its floating point argument
- `truncate` returns its argumentas an integer truncated toward 0.
- `round` returns it argument as an integer rounded away from 0.
- `fromIntegral` returns its integer argument as a floating point number in a context where `Double` or `Float` is required. It can also return its `Integer` argument as an `Int` or vice versa.

5.3.3 Booleans: `Bool`

The `Bool` (i.e., Boolean) data type is usually supported directly by the host processor as one or more contiguous bits.

The `Bool` literals are `True` and `False`. Note that these begin with capital letters.

Haskell supports Boolean operations such as `&&` (and), `||` (or), and `not` (logical negation).

Functions can match against patterns using the Boolean constants. For example, we could define a function `myAnd` as follows:

```
myAnd :: Bool -> Bool -> Bool
myAnd True  b = b
myAnd False _ = False
```

Above the pattern `_` is a wildcard that matches any value but does not bind a value that can be used on the right-hand-side of the definition.

The expressions in Haskell `if` conditions and guards on function definitions must be `Bool`-valued expressions. They can include calls to functions that return `Bool` values.

5.3.4 Characters: `Char`

The `Char` data type is usually supported directly by the host processor by one or more contiguous bytes.

Haskell uses Unicode for its character data type. Haskell supports character literals enclosed in single quotes—including both the graphic characters (e.g., `'a'`, `'0'`, and `'Z'`) and special codes entered following the escape character backslash \ (e.g., `'\n'` for newline, `'\t'` for horizontal tab, and `'\\'` for backslash itself).

In addition, a backslash character \ followed by a number generates the corresponding Unicode character code. If the first character following the backslash is `o`, then the number is in octal representation; if followed by `x`, then in hexadecimal notation; and otherwise in decimal notation.

For example, the exclamation point character can be represented in any of the following ways: `'!'`, `'\33'`, `'\o41'`, `'\x21'`

5.3.5 Functions: `t1 -> t2`

If `t1` and `t2` are types then `t1 -> t2` is the type of a function that takes an argument of type `t1` and returns a result of type `t2`.

Function and variable names begin with lowercase letters optionally followed by a sequences of characters each of which is a letter, a digit, an apostrophe (`'`) (sometimes pronounced "prime"), or an underscore (`_`).

Haskell functions are *first-class* objects. They can be arguments or results of other functions or be components of data structures. Multi-argument functions are *curried*-–that is, treated as if they take their arguments one at a time.

For example, consider the integer addition operation `(+)`. (Surrounding the binary operator symbol with parentheses refers to the corresponding function.)

In mathematics, we normally consider addition as an operation that takes a *pair* of integers and yields an integer result, which would have the type signature

```
(+) :: (Int,Int) -> Int
```

In Haskell, we give the addition operation the type

```
(+) :: Int -> (Int -> Int)
```

or just

```
(+) :: Int -> Int -> Int
```

since Haskell binds `->` from the right.

Thus `(+)` is a one argument function that takes some `Int` argument and returns a function of type `Int -> Int`. Hence, the expression `((+) 5)` denotes a function that takes one argument and returns that argument plus `5`.

We sometimes speak of this `(+)` operation as being *partially applied* (i.e., to one argument instead of two).

This process of replacing a structured argument by a sequence of simpler ones is called *currying*, named after American logician Haskell B. Curry who first described it.

The Haskell library, called the *standard prelude* (or just Prelude), contains a wide range of predefined functions including the usual arithmetic, relational, and Boolean operations. Some of these operations are predefined as *infix* operations.

5.3.6 Tuples: `(t1,t2,...,tn)`

If `t1`, `t2`, $\cdots$, `tn` are types, where `n` is finite and `n >= 2`, then is a type consisting of *n-tuples* where the various components have the type given for that position.

Each element in a tuple may have different types. The number of elements in a tuple is fixed.

Examples of tuple values with their types include the following:

```
('a',1) :: (Char,Int)
(0.0,0.0,0.0) :: (Double,Double,Double)
(('a',False),(3,4)) :: ((Char, Bool), (Int, Int))
```

We can also define a *type synonym* using the `type` declaration and the use the synonym in further declarations as follows:

```
type Complex = (Float,Float)
makeComplex :: Float -> Float -> Complex
makeComplex r i = (r,i)`
```

A type synonym does not define a new type, but it introduces an alias for an existing type. We can use `Complex` in declarations, but it has the same effect

as using (Float,Float) expect that Complex provides better documentation of the intent.

5.3.7 Lists: [t]

The primary built-in data structure in Haskell is the *list*, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with notation such as [t] to denote a list of zero or more elements of type t.

A list literal is a comma-separated sequence of values enclosed between [and]. For example, [] is an empty list and [1,2,3] is a list of the first three positive integers in increasing order.

We will look at programming with lists in a later chapter.

5.3.8 Strings: String

In Haskell, a *string* is just a list of characters. Thus Haskell defines the data type String as a *type synonym* :

```
type String = [Char]
```

We examine lists and strings in a later chapter, but, because we use strings in a few examples in this subsection, let's consider them briefly.

A String literal is a sequence of zero or more characters enclosed in double quotes, for example, "Haskell programming".

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code \& is used to separate any character sequences that are otherwise ambiguous.

For example, the string literal "Hotty\nToddy!\n" is a string that has two newline characters embedded.

Also the string literal "\12\&3" represents the two-element list ['\12','3'].

The function show returns its argument converted to a String.

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings. We look at manipulating lists and strings in later chapters of this textbook.

5.3.9 Advanced Types

In later chapters, we examine other important Haskell type concepts such as user-defined algebraic data types and type classes.

5.4 What Next?

In this chapter (5), we examined general type systems concepts and explored Haskell's builtin types.

For a similar presentation of the types in the Python 3 language, see reference [45].

In Chapters 6 and 7, we examine methods for developing Haskell programs using abstraction. We explore use of top-down stepwise refinement, modular design, and other methods in the context of Haskell.

5.5 Exercises

For each of the following exercises, develop and test a Haskell function or set of functions.

1. Develop a Haskell function `sumSqBig` that takes three `Double` arguments and yields the sum of the squares of the two larger numbers.

   For example, `(sumSqBig 2.0 1.0 3.0)` yields `13.0`.

2. Develop a Haskell function `prodSqSmall` that takes three `Double` arguments and yields the product of the squares of the two smaller numbers.

   For example, `(prodSqSmall 2.0 4.0 3.0)` yields `36.0`.

3. Develop a Haskell function `xor` that takes two Booleans and yields the "exclusive-or" of the two values. An exclusive-or operation yields `True` when exactly one of its arguments is `True` and yields `False` otherwise.

4. Develop a Haskell Boolean function `implies` that takes two Booleans `p` and `q` and yields the Boolean result `p` ⇒ `q` (i.e., logical implication). That is, if `p` is `True` and `q` is `False`, then the result is `False`; otherwise, the result is `True`.

   Note: This function is sometimes called `nand`.

5. Develop a Haskell Boolean function `div23n5` that takes an `Int` and yields `True` if and only if the integer is divisible by 2 or divisible by 3, but is not divisible by 5.

   For example, `(div23n5 4)`, `(div23n5 6)`, and `(div23n5 9)` all yield `True` and `(div23n5 5)`, `(div23n5 7)`, `(div23n5 10)`, `(div23n5 15)`, `(div23n5 30)` all yield `False`.

6. Develop a Haskell function `notDiv` such that `notDiv n d` yields `True` if and only if integer `n` is not divisible by `d`.

   For example, `(notDiv 10 5)` yields `False` and `(notDiv 11 5)` yields `True`.

7. Develop a Haskell function `ccArea` that takes the *diameters* of two concentric circles (i.e., with the same center point) as `Double` values and yields the area of the space between the circles. That is, compute the area of the larger circle minus the area of the smaller circle. (Hint: Haskell has a builtin constant `pi`.)

For example, `(ccArea 2.0 4.0)` yields approximately `9.42477796`.

8. Develop a Haskell function `mult` that takes two *natural numbers* (i.e., nonnegative integers in `Int`) and yields their product. The function must not use the multiplication (∗) or division (`div`) operators. (Hint: Multiplication can be done by repeated addition.)

9. Develop a Haskell function `addTax` that takes two `Double` values such that `addTax c p` yield `c` with a sales tax of `p` *percent* added.

   For example, `(addTax 2.0 9.0)` yields `2.18`.

   Also develop a function `subTax` that is the inverse of `addTax`. That is, `(subTax (addTax c p) p)` yields `c`.

   For example, `(subTax 2.18 9.0) = 2.0`.

10. The time of day can be represented by a tuple `(hours,minutes,m)` where `hours` and `minutes` are `Int` values with `1 <= hours <= 12` and `0 <= minutes <= 59`, and where `m` is either the string value `"AM"` or `"PM"`.

    Develop a Boolean Haskell function `comesBefore` that takes two time-of-day tuples and determines whether the first is an earlier time than the second.

11. A day on the calendar (usual Gregorian calendar [217] used in the USA) can be represented as a tuple with three `Int` values `(month,day,year)` where the `year` is a positive integer, `1 <= month <= 12`, and `1 <= day <= days_in_month`. Here `days_in_month` is the number of days in the the given `month` (i.e., 28, 29, 30, or 31) for the given `year`.

    Develop a Boolean Haskell function `validDate d` that takes a date tuple `d` and yields `True` if and only if `d` represents a valid date.

    For example, `validDate (8,20,2018)` and `validDate (2,29,2016)` yield `True` and `validDate (2,29,2017)` and `validDate (0,0,0)` yield `False`.

    Note: The *Gregorian calendar* [217] was introduced by Pope Gregory of the Roman Catholic Church in October 1582. It replaced the Julian calendar system, which had been instituted in the Roman Empire by Julius Caesar in 46 BC. The goal of the change was to align the calendar year with the astronomical year.

    Some countries adopted the Gregorian calendar at that time. Other countries adopted it later. Some countries may never have adopted it officially.

    However, the Gregorian calendar system became the common calendar used worldwide for most civil matters. The *proleptic Gregorian calendar* [218] extends the calendar backward in time from 1582. The year 1 BC becomes year 0, 2 BC becomes year -1, etc. The proleptic Gregorian

calendar underlies the ISO 8601 standard used for dates and times in software systems [219].

12. Develop a Haskell function `roman` that takes an `Int`) in the range from 0 to 3999 (inclusive) and yields the corresponding Roman numeral [220] as a string (using capital letters). The function should halt with an appropriate `error` messages if the argument is below or above the range. Roman numerals use the symbols shown in Table 5.1 and are combined by addition or subtraction of symbols.

Table 5.1: Decimal equivalents of Roman numerals.

Roman	=	Decimal
I		1
V		5
X		10
L		50
C		100
D		500
M		1000

For the purposes of this exercise, we represent the Roman numeral for 0 as the empty string. The Roman numerals for integers 1-20 are `I`, `II`, `III`, `IV`, `V`, `VI`, `VII`, `VIII`, `IX`, `X`, `XI`, `XII`, `XIII`, `XIV`, `XV`, `XVI`, `XVII`, `XVII`, `XIX`, and `XX`. Integers 40, 90, 400, and 900 are `XL`, `XC`, `CD`, and `CM`.

13. Develop a Haskell function

```
minf :: (Int -> Int) -> Int
```

that takes a function `g` and yields the smallest integer `m` such that `0 <= m <= 10000000` and `g m == 0`. It should throw an `error` if there is no such integer.

5.6 Acknowledgements

In Summer 2016, I adapted and revised the discussion Surveying the Basic Types from chapter 5 of my *Notes on Functional Programming with Haskell* [42]. In 2017, I incorporated the discussion into Section 2.4 in Chapter 2 Basic Haskell Functional Programming of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming* [54]. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs; previous Section 2.4 became Section 5.3 in the new Chapter 5, Types

(this chapter); and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction, and Chapter 7, Data Abstraction.

In Spring 2018, I wrote the general Type System Concepts section as a part of a chapter that discusses the type system of Python 3 [45] to support my use of Python in graduate CSci 658 (Software Language Engineering) course.

In Summer 2018, I revised the section to become Section 5.2 in Chapter 5 of the Fall 2018 version of ELIFP [54]. I also moved the "Kinds of Polymorphism" discussion from the 2017 List Programming chapter to the new subsection "Polymorphic Operations". This textbook draft supported my Haskell-based offering of the core course CSci 450 (Organization of Programming Languages).

The type concepts discussion draws ideas from various sources:

- my general study of a variety of programming, programming language, and software engineering over three decades [13,20,22,99–101,104,119,128,131,134,135,156,157,171].
- the *Wikipedia* articles on the Liskov Substitution Principle [205], Polymorphism [206], Ad Hoc Polymorphism [208], Parametric Polymorphism [209], Subtyping [210], and Function Overloading [207]

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I also added some discussion on the functions `div`, `quot`, `mod`, `rem`, `fromIntegral`, and `show` because of their usefulness in the exercises in this and later chapters.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

5.7 Terms and Concepts

Type, subtype, Liskov Substitution Principle, types of constants, variables, and expressions, static vs. dynamic types, nominal vs. structural types, polymorphic operations (ad hoc, overloading, subtyping, parametric/generic), early vs. late binding, compile time vs. runtime, polymorphic variables, basic Haskell types (`Int`, `Integer`, `Bool`, `Char`, functions, tuples, lists, `String`), `type` aliases, library (Prelude) functions, proleptic Gregorian calendar system, Roman numerals.

6 Procedural Abstraction

6.1 Chapter Introduction

Chapter 2 introduced the concepts of procedural and data abstraction. This chapter (6) focuses on procedural abstraction. Chapter 7focuses on data abstraction.

The goals of this chapter are to:

- illustrate use of procedural abstraction, in particular of the top-down, stepwise refinement approach to design
- introduce modular programming using Haskell modules

6.2 Procedural Abstraction Review

As defined in Chapter 2, *procedural abstraction* is the separation of the logical properties of an *action* from the details of how the action is implemented.

In general, we abstract an action into a Haskell function that takes zero or more arguments and returns a value but does not have other effects. In later chapters, we discuss how input, output, and other effects are handled in a purely functional manner. (For example, in Chapter 10 we examine simple input and output.)

We also collect one or more functions into a Haskell module with appropriate type definitions, data structures, and local functions. We can explicitly expose some of the features and hide others.

To illustrate the development of a group of related Haskell procedural abstractions in this chapter, we use top-down stepwise refinement.

6.3 Top-Down Stepwise Refinement

A useful and intuitive design process for a small program is to begin with a high-level solution and incrementally fill in the details. We call this process top-down stepwise refinement. Here we introduce it with an example.

6.3.1 Developing a square root package

Consider the problem of computing the nonnegative square root of a nonnegative number x. Mathematically, we want to find the number y such that

$y \geq 0$ and $y^2 = x$.

A common algorithm in mathematics for computing the above y is to use Newton's method of successive approximations, which has the following steps for square root:

1. Guess at the value of y.

2. If the current approximation (guess) is sufficiently close (i.e., good enough), return it and stop; otherwise, continue.

3. Compute an improved guess by averaging the value of the guess y and x/y, then go back to step 2.

To encode this algorithm in Haskell, we work top down to decompose the problem into smaller parts until each part can be solved easily. We begin this *top-down stepwise refinement* by defining a function with the type signature:

```
sqrtIter :: Double -> Double -> Double
```

We choose type `Double` (double precision floating point) to approximate the real numbers. Thus we can encode step 2 of the above algorithm as the following Haskell function definition:

```
sqrtIter guess x                                   -- step 2
    | goodEnough guess x = guess
    | otherwise          = sqrtIter (improve guess x) x
```

We define function `sqrtIter` to take two arguments—the current approximation `guess` and nonnegative number `x` for which we need the square root. We have two cases:

- When the current approximation `guess` is sufficiently close to `x`, we return `guess`.

  We abstract this decision into a separate function `goodEnough` with type signature:

  ```
  goodEnough :: Double -> Double -> Bool
  ```

- When the approximation is not yet close enough, we continue by reducing the problem to the application of `sqrtIter` itself to an improved approximation.

  We abstract the improvement process into a separate function `improve` with type signature:

  ```
  improve :: Double -> Double -> Double
  ```

  To ensure termination of `sqrtIter`, the argument `(improve guess x)` on the recursive call must get closer to termination (i.e., to a value that satisfies its base case).

The function `improve` takes the current `guess` and `x` and carries out step 3 of the algorithm, thus averaging `guess` and `x/guess`, as follows:

```
improve :: Double -> Double -> Double      -- step 3
improve guess x = average guess (x/guess)
```

Function application `improve y x` assumes `x >= 0 && y > 0`. We call this a *precondition* of the `improve y x` function.

Because of the precondition of `improve`, we need to strengthen the precondition of `sqrtIter guess x` to `x >= 0 && guess > 0`.

In `improve`, we abstract `average` into a separate function as follows:

```
average :: Double -> Double -> Double
average x y = (x + y) / 2
```

The new guess is closer to the square root than the previous guess. Thus the algorithm will terminate assuming a good choice for function `goodEnough`, which guards the base case of the `sqrtIter` recursion.

How should we define `goodEnough`? Given that we are working with the limited precision of computer floating point arithmetic, it is not easy to choose an appropriate test for all situations. Here we simplify this and use a tolerance of 0.001.

We thus postulate the following definition for `goodEnough`:

```
goodEnough :: Double -> Double -> Bool
goodEnough guess x = abs (square guess - x) < 0.001
```

In the above, `abs` is the built-in absolute value function defined in the standard Prelude library. We define square as the following simple function (but could replace it by just `guess * guess`):

```
square :: Double -> Double
square x = x * x
```

What is a good initial guess? It is sufficient to just use 1. So we can define an overall square root function `sqrt'` as follows:

```
sqrt' :: Double -> Double
sqrt' x | x >= 0 = sqrtIter 1 x
```

(A square root function `sqrt` is defined in the Prelude library, so a different name is needed to avoid the name clash.)

Function `sqrt' x` has precondition `x >= 0`. This and the choice of `1` for the initial guess ensure that functions `sqrtIter` and `improve` are applied with arguments that satisfy their preconditions.

6.3.2 Making the package a Haskell module

We can make this package into a Haskell module by putting the definitions in a file (e.g., named `Sqrt`) and adding a module header at the beginning as follows:

```
module Sqrt
    ( sqrt' )
where
    -- give the definitions above for functions sqrt',
    --   sqrtIter, improve, average, and goodEnough
```

The header gives the module the name Sqrt and lists the names of the features being *exported* in the parenthesis that follows the name. In this case, only function sqrt' is exported.

Other Haskell modules that *import* the Sqrt module can access the features named in its export list. In the case of Sqrt, the other functions—sqrtIter, goodEnough, and improve)— are local to (i.e., hidden inside) the module.

In this book, we often call the exported features (e.g., functions and types) the module's *public* features and the ones not exported the *private* features.

We can import module Sqrt into a module such as module TestSqrt shown below. By default, the import makes all the definitions exported by Sqrt available within module TestSqrt. The importing module may select the features it wishes to export and may assign local names to the features it does import.

```
module TestSqrt
where

import Sqrt -- file Sqrt.hs, import all public names

main = do
    putStrLn (show (sqrt' 16))
    putStrLn (show (sqrt' 2))
```

In the above Haskell code, the symbol "--" denotes the beginning of an end-of-line comment. All text after that symbol is text ignored by the Haskell compiler.

The Haskell module for the Square root case study is in file Sqrt.hs. Limited, smoke-testing code is in file SqrtTest.hs.

6.3.3 Reviewing top-down stepwise refinement

The program design strategy known as *top-down stepwise refinement* is a relatively intuitive design process that has long been applied in the design of structured programs in imperative procedural languages. It is also useful in the functional setting.

In Haskell, we can apply top-down stepwise refinement as follows.

1. Start with a high-level solution to the problem consisting of one or more functions. For each function, identify its type signature and functional requirements (i.e., its inputs, outputs, and termination condition).

   Some parts of each function may be incomplete—expressed as "pseudocode" expressions or as-yet-undefined functions.

2. Choose one of the incomplete parts. Consider the specified type signature and functional requirements. Refine the incomplete part by breaking it into subparts or, if simple, defining it directly in terms of Haskell

expressions (including calls to the Prelude, other available library functions, or previously defined parts of the algorithm).

When refining an incomplete part, consider the various options according to the relevant design criteria (e.g., time, space, generality, understandability, and elegance).

The refinement of the function may require a refinement of the data being passed.

If it not possible to design an appropriate function or data refinement, back up in the refinement process and readdress previous design decisions.

3. Continue step 2 until all parts are fully defined in terms of Haskell code and data and the resulting set of functions meets all required criteria.

For as long as possible, we should stay with terminology and notation that is close to the problem being solved. We can do this by choosing appropriate function names and signatures and data types. (In other chapters, we examine Haskell's rich set of builtin and user-defined types.)

For stepwise refinement to work well, we must be willing to back up to earlier design decisions when appropriate. We should keep good documentation of the intermediate design steps.

The stepwise refinement method can work well for small programs , but it may not scale well to large, long-lived, general purpose programs. In particular, stepwise refinement may lead to a module structure in which modules are tightly coupled and not robust with respect to changes in requirements.

A combination of techniques may be needed to develop larger software systems. In the next section (6.4), we consider the use of modular design techniques.

6.4 Modular Design and Programming

In the previous section, we developed a Haskell module. In this section, let's consider what a module is more generally.

Software engineering pioneer David Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" [138]. This is a *software engineering* view of a module.

In a programming language like Haskell, a `module` is also a program unit defined with a construct or convention. This is a *programming language* view of a module.

In a programming language, each module may be stored in a separate file in the computer's file system. It may also be the smallest external unit processed by the language's compiler or interpreter.

Ideally, a language's module features should support the software engineering module methods.

6.4.1 Information-hiding modules and secrets

According to Parnas, the goals of *modular design* are to [134]:

1. enable programmers to understand the system by focusing on one module at a time (i.e., *comprehensibility*).
2. shorten development time by minimizing required communication among groups (i.e., *independent development*).
3. make the software system flexible by limiting the number of modules affected by significant changes (i.e., *changeability*).

Parnas advocates the use of a principle he called *information hiding* to guide decomposition of a system into appropriate modules (i.e., work assignments). He points out that the connections among the modules should have as few information requirements as possible [134].

In the Parnas approach, an information-hiding module:

- forms a *cohesive* unit of functionality *separate* from other modules
- *hides* a design decision—its *secret*—from other modules
- *encapsulates* an aspect of system likely to change (its secret)

Aspects likely to change independently of each other should become secrets of separate modules. Aspects unlikely to change can become interactions (connections) among modules.

This approach supports the goal of changeability (goal 2). When care is taken to design the modules as clean abstractions with well-defined and documented interfaces, the approach also supports the goals of independent development (goal 1) and comprehensibility (goal 3).

Information hiding has been absorbed into the dogma of contemporary object-oriented programming. However, information hiding is often oversimplified as merely hiding the data and their representations [177].

The secret of a well-designed module may be much more than that. It may include such knowledge as a specific functional requirement stated in the requirements document, the processing algorithm used, the nature of external devices accessed, or even the presence or absence of other modules or programs in the system [134,136,137]. These are important aspects that may change as the system evolves.

Secret of square root module The secret of the `Sqrt` module in the previous section is the algorithm for computing the square root.

6.4.2 Contracts: Preconditions and postconditions

Now let's consider the semantics (meaning) of functions.

The *precondition* of a function is what the caller (i.e., the client of the function) must ensure holds when calling the function. A precondition may specify the valid combinations of values of the arguments. It may also record any constraints on any "global" state that the function accesses or modifies.

If the precondition holds, the supplier (i.e., developer) of the function must ensure that the function terminates with the *postcondition* satisfied. That is, the function returns the required values and/or alters the "global" state in the required manner.

We sometimes call the set of preconditions and postconditions for a function the *contract* for that function.

Contracts of square root module In the `Sqrt` module defined in the previous section, the exported function `sqrt' x` has the precondition:

```
x >= 0
```

Function `sqrt' x` is undefined for `x < 0`.

The postcondition of the function `sqrt' x` function is that the result returned is the correct mathematical value of the square root within the allowed tolerance. That is, for a tolerance of 0.001:

```
(sqrt x - 0.001)^2 < (sqrt x)^2 < (sqrt x + 0.001)^2
```

We can state preconditions and postconditions for the local functions `sqrtIter`, `improve`, `average`, and `goodEnough` in the `Sqrt` module. These are left as exercises.

The preconditions for functions `average` and `goodEnough` are just the assertion `True` (i.e., always satisfied).

Contracts of `Factorial` module Consider the factorial functions defined in Chapter 4. (These are in the source file `Factorial.hs`.)

What are the preconditions and postconditions?

Functions `fact1`, `fact2`, and `fact3` require that argument `n` be a natural number (i.e., nonnegative integer) value. If they are applied to a negative value for `n`, then the evaluation does not terminate. Operationally, they go into an "infinite loop" and likely will abort when the runtime stack overflows.

If function `fact4` is called with a negative argument, then all guards and pattern matches fail. Thus the function aborts with a standard error message.

Similarly, function `fact4'` terminates with a custom error message for negative arguments.

Thus to ensure normal termination, we impose the precondition

```
n >= 0
```

on all these factorial functions.

The postcondition of all six factorial functions is that the result returned is the correct mathematical value of `n` factorial. For `fact4`, that is:

```
fact4 n = fact'(n)
```

None of the six factorial functions access or modify any global data structures, so we do not include other items in the precondition or postcondition.

Function `fact5` is defined to be 1 for all arguments less than zero. So, if this is the desired result, we can weaken the precondition to allow all integer values, for example,

```
True
```

and strengthen the postcondition to give the results for negative arguments, for example:

```
fact5 n = if n >= 0 then fact'(n) else 1
```

Caveat: In this chapter, we ignore the limitations on the value of the factorial functions' argument `n` imposed by the finite precision of the computer's integer arithmetic. We readdress this issue somewhat in Chapter 12.

6.4.3 Interfaces for modules

It is important for an information-hiding module to have a well-defined and stable interface. What do we mean by interface?

According to Britton et al [20], an *interface* is a "set of assumptions ... each programmer needs to make about the other program ... to demonstrate the correctness of his own program".

A module interface includes the type signatures (i.e., names, arguments, and return values), preconditions, and postconditions of all public operations (e.g., functions).

As we see in Chapter 7, the interface also includes the *invariant* properties of the data values and structures manipulated by the module and the definitions of any new data types exported by the module. An invariant must be part of the precondition of public operations except operations that construct elements of the data type (i.e., constructors). It must also be part of the postcondition of public operations except operations that destroy elements of the data type (i.e., destructors).

As we have seen, in Haskell the `module` not provide direct syntactic or semantic support for preconditions, postconditions, or invariant assertions.

Interface of square root module The interface to the `Sqrt` module in the previous section consists of the function signature:

```haskell
    sqrt' :: Double -> Double
```

where `sqrt' x` has the precondition and postcondition defined above. None of the other functions are accessible outside the module `Sqrt` and, hence, are not part of the interface.

6.4.4 Abstract interfaces for modules

An *abstract interface* is an interface that does not change when one module implementation is substituted for another [20,138]. It concentrates on module's essential aspects and obscures incidental aspects that vary among implementations.

Information-hiding modules and abstract interfaces enable us to design and build software systems with multiple versions. The information-hiding approach seeks to identify aspects of a software design that might change from one version to another and to hide them within independent modules behind well-defined abstract interfaces.

We can reuse the software design across several similar systems. We can reuse an existing module implementation when appropriate. When we need a new implementation, we can create one by following the specification of the module's abstract interface.

Abstract interface of square root module For the `Sqrt` example, if we implemented a different module with the same interface (signatures, preconditions, postconditions, etc.), then we could substitute the new module for `Sqrt` and get the same result.

In this case, the interface is an abstract interface for the set of module implementations.

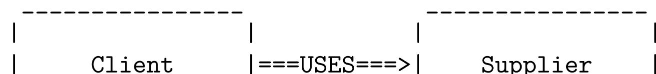
Caveats: Of course, the time and space performance of the alternative modules might differ. Also, because of the nature of floating point arithmetic, it may be difficult to ensure both algorithms have precisely the same termination conditions.

6.4.5 Client-supplier relationship

The design and implementation of information-hiding modules should be approached from two points of view simultaneously:

supplier: the developers of the module—the providers of the services
client: the users of the module—the users of the services (e.g., the designers of other modules)

The *client-supplier relationship* is as represented in the following diagram:



The supplier's concerns include:

- efficient and reliable algorithms and data structures
- convenient implementation
- easy maintenance

The clients' concerns include:

- accomplishing their own tasks
- using the supplier module without effort to understand its internal details
- having a sufficient, but not overwhelming, set of operations.

As we have noted previously, the *interface* of a module is the set of features (i.e., public operations) provided by a supplier to clients.

A precise description of a supplier's interface forms a *contract* between clients and supplier.

The client-supplier contract:

1. gives the responsibilities of the client

   These are the conditions under which the supplier must deliver results—when the *preconditions* of the operations are satisfied (i.e., the operations are called correctly).

2. gives the responsibilities of the supplier

   These are the benefits the supplier must deliver—make the *postconditions* hold at the end of the operation (i.e., the operations deliver the correct results).

The contract

- protects the client by specifying how much must be done by the supplier
- protects the supplier by specifying how little is acceptable to the client

If we are both the clients and suppliers in a design situation, we should consciously attempt to separate the two different areas of concern, switching back and forth between our supplier and client "hats".

6.4.6 Design criteria for interfaces

What else should we consider in designing a good interface for an information-hiding module?

In designing an interface for a module, we should also consider the following criteria. Of course, some of these criteria conflict with one another; a designer must carefully balance the criteria to achieve a good interface design.

Note: These are general principles; they are not limited to Haskell or functional programming. In object-oriented languages, these criteria apply to class interfaces.

- **Cohesion:** All operations must logically fit together to support a single, coherent purpose. The module should describe a single abstraction.
- **Simplicity:** Avoid needless features. The smaller the interface the easier it is to use the module.
- **No redundancy:** Avoid offering the same service in more than one way; eliminate redundant features.
- **Atomicity:** Do not combine several operations if they are needed individually. Keep independent features separate. All operations should be *primitive*, that is, not be decomposable into other operations also in the public interface.
- **Completeness:** All primitive operations that make sense for the abstraction should be supported by the module.
- **Consistency:** Provide a set of operations that are internally consistent in
  - naming convention (e.g.„ in use of prefixes like "set" or "get", in capitalization, in use of verbs/nouns/adjectives),
  - use of arguments and return values (e.g.„ order and type of arguments),
  - behavior (i.e., make operations work similarly).

  Avoid surprises and misunderstandings. Consistent interfaces make it easier to understand the rest of a system if part of it is already known.

  The operations should be consistent with good practices for the specific language being used.
- **Reusability:** Do not customize modules to specific clients, but make them general enough to be reusable in other contexts.
- **Robustness with respect to modifications:** Design the interface of an module so that it remains stable even if the implementation of the module changes. (That is, it should be an abstract interface for an information-hiding module as we discussed above.)
- **Convenience:** Where appropriate, provide additional operations (e.g.„ beyond the complete primitive set) for the convenience of users of the module. Add convenience operations only for frequently used combinations after careful study.

We must trade off conflicts among the criteria. For example, we must balance:

- completeness versus simplicity
- reusability versus simplicity
- convenience versus consistency, simplicity, no redundancy, and atomicity

We must also balance these design criteria against efficiency and functionality.

6.5 What Next?

In this chapter (6), we considered procedural abstraction and modularity in that context.

In Chapter 7, we consider data abstraction and modularity in that context.

6.6 Chapter Source Code

The Haskell source code for this chapter are in files:

- `Sqrt.hs` for the Square Root case study
- `SqrtTest.hs` for (limited) "smoke testing" of the `Sqrt` module
- `Factorial.hs` for the factorial source code from Chapter 4
- `TestFactorial.hs` is an extensive testing module developed in Chapter 12 for the factorial module

6.7 Exercises

1. State preconditions and postconditions for the following internal functions in the `Sqrt` module:

    a. `sqrtIter`
    b. `improve`
    c. `average`
    d. `goodEnough`
    e. `square`

2. Develop recursive and iterative (looping) versions of the square root function from this chapter in one or more primarily imperative languages (e.g., Java, C++, C#, Python 3, or Lua)

6.8 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from my previous materials:

- Using Top-Down Stepwise Refinement (square root module), which is based on Section 1.1.7 of Abelson and Sussman's *Structure and Interpretation of*

Computer Programs [1] and my example implementations of this algorithm in Scala, Elixir, and Lua as well as Haskell.

- Modular Design and Programming from my Data Abstraction [46] and Modular Design [47] notes, which drew ideas over the past 25 years from a variety of sources [20,22,56,58,59,61,99,100,128,129,134,136,137,170,177].

In 2017, I continued to develop this work as sections 2.5-2.7 in Chapter 2, Basic Haskell Functional Programming), of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs; previous Section 2.4 became Section 5.3 in the new Chapter 5, Types; and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction (this chapter), and Chapter 7, Data Abstraction. The discussion of contracts for the factorial functions was moved from the 2017 Evaluation and Efficiency chapter to this chapter.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), adding cross-references, and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

6.9 Terms and Concepts

TODO: Update

Procedural abstraction, top-down stepwise refinement, abstract code, termination condition for recursion, Newton's method, Haskell `module`, module exports and imports, information hiding, module secret, encapsulation, precondition, postcondtion, contract, invariant, interface, abstract interface, design criteria for interfaces, software reuse, use of Haskell modules to implement information-hiding modules, client-supplier contract.

7 Data Abstraction

7.1 Chapter Introduction

Chapter 2 introduced the concepts of procedural and data abstraction. Chapter 6 focuses on procedural abstraction and modular design and programming. This chapter focuses on data abstraction. 4 The goals of this chapter are to:

- illustrate use of data abstraction
- reinforce and extend the concepts of modular design and programming using Haskell modules

The chapter uses the development of a rational arithmetic package to illustrate data abstraction.

7.2 Data Abstraction Review

As defined in Chapter 2, *data abstraction* is the separation of the logical properties of *data* from the details of how the data are represented.

In data abstraction, programmers primarily focus on the problem's data and secondarily on its actions. Programmers first identify the key data entities and develop the programs around those and the operations needed to create and update them.

Data abstraction seeks to make a program robust with respect to change in the data.

7.3 Using Data Abstraction

As in Chapter 6, let's begin the study of this design technique with an example.

7.3.1 Rational number arithmetic

For this example, let's implement a group of Haskell functions to perform rational number arithmetic, assuming that the Haskell library does not contain such a data type. We focus first on the operations we want to perform on the data.

In mathematics we usually write rational numbers in the form $\frac{x}{y}$ where `x` and `y` are integers and $y \neq 0$.

For now, let us assume we have a special type `Rat` to represent rational numbers and a constructor function

```
makeRat :: Int -> Int -> Rat
```

to create a Haskell rational number instance from a numerator `x` and a denominator `y`. That is, `makeRat x y` constructs a Haskell rational number with mathematical value $\frac{x}{y}$, where $y \neq 0$.

Let us also assume we have selector functions `numer` and `denom` with the signatures:

```
numer, denom :: Rat -> Int
```

Functions `numer` and `denom` take a valid Haskell rational number and return its numerator and denominator, respectively.

Requirement: For any `Int` values `x` and `y` where $y \neq 0$, there exists a Haskell rational number `r` such that `makeRat x y == r` and rational number values $\frac{\texttt{numer r}}{\texttt{denom r}} = \frac{\texttt{x}}{\texttt{y}}$.

Note: In this example, we use fraction notation like $\frac{\texttt{x}}{\texttt{y}}$ to denote the mathematical value of the rational number. In constrast, `r` above denotes a Haskell value representing a rational number.

We consider how to implement rational numbers in Haskell later, but for now let's look at rational arithmetic implemented using the constructor and selector functions specified above.

Given our knowledge of rational arithmetic from mathematics, we can define the operations for unary negation, addition, subtraction, multiplication, division, and equality as follows. We assume that the operands `x` and `y` are values created by the constructor `makeRat`.

```
negRat :: Rat -> Rat
negRat x = makeRat (- numer x) (denom x)

addRat, subRat, mulRat, divRat :: Rat -> Rat -> Rat  -- (1)
addRat x y = makeRat (numer x * denom y + numer y * denom x)
                     (denom x * denom y)
subRat x y = makeRat (numer x * denom y - numer y * denom x)
                     (denom x * denom y)
mulRat x y = makeRat (numer x * numer y) (denom x * denom y)
divRat x y  -- (2) (3)
    | eqRat y zeroRat = error "Attempt to divide by 0"
    | otherwise       = makeRat (numer x * denom y)
                                (denom x * numer y)

eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

The above code:

1. combines the type signatures for all four arithmetic operations into a single declaration by listing the names separated by commas
2. introduces the parameterless function `zeroRat` to abstract the constant rational number value 0

Note: We could represent zero as `makeRat 0 1` but choose to introduce a separate abstraction.

3. calls the `error` function for an attempt to divide by zero

These arithmetic functions do not depend upon any specific representation for rational numbers. Instead, they use rational numbers as a *data abstraction* defined by the type `Rat`, constant `zeroRat`, constructor function `makeRat`, and selector functions `numer` and `denom`.

The goal of a data abstraction is to separate the logical properties of *data* from the details of how the data are represented.

7.3.2 Rational number data representation

Now, how can we represent rational numbers?

For this package, we define type synonym `Rat` to denote this type:

```
type Rat = (Int, Int)
```

For example, `(1,7)`, `(-1,-7)`, `(3,21)`, and `(168,1176)` all represent the value $\frac{1}{7}$.

As with any value that can be expressed in many different ways, it is useful to define a single *canonical* (or *normal*) form for representing values in the rational number type `Rat`.

It is convenient for us to choose a Haskell rational number representation `(x,y)` that satisfies all parts of the following **Rational Representation Property**:

- `(x,y)` ∈ `(Int,Int)`
- `y > 0`
- if `x == 0`, then `y == 1`
- `x` and `y` are relatively prime
- rational number value is $\frac{x}{y}$

By *relatively prime*, we mean that the two integers have no common divisors except 1.

This representation keeps the magnitudes of the numerator `x` and denominator `y` small, thus reducing problems with overflow arising during arithmetic operations.

This representation also gives a unique representation for zero. For convenience, we define the name `zeroRat` to represent this constant:

```
zeroRat :: (Int,Int)
zeroRat = (0,1)
```

We can now define constructor function `makeRat x y` that takes two `Int` values (for the numerator and the denominator) and returns the corresponding Haskell rational number in this canonical form.

```
makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ show x ++ "/0" )          -- (1)
makeRat 0 _ = zeroRat
makeRat x y = (x' `div` d, y' `div` d)             -- (2)
    where x' = (signum' y) * x                     -- (3,4)
          y' = abs' y
          d  = gcd' x' y'
```

In the definition of `makeRat`, we use features of Haskell we have not used in the previous examples. the above code:

1. uses the infix `++` (read "append") operator to concatenate two strings

   We discuss `++` in the chapter on infix operations.

2. puts backticks (`) around an alphanumeric function name to use that function as an infix operator

   The function `div` denotes integer division. Above the `div` operator denotes the integer division function used in an infix manner.

3. uses a `where` clause to introduce `x'`, `y'`, and `d` as local definitions within the body of `makeRat`

   These local definition can be accessed from within `makeRat` but not from outside the function. In contrast, `sqrtIter` in the Square Root example is at the same level as `sqrt'`, so it can be called by other functions (in the same Haskell module at least).

   The `where` feature allows us to introduce new definitions in a top-down manner—first using a symbol and then defining it.

4. uses *type inference* for local variables `x'`, `y'`, and `d` instead of giving explicit type definitions

   These parameterless functions could be declared

   ```
   x', y', d :: Int
   ```

   but it was not necessary because Haskell can infer the types from the types involved in their defining expressions.

   Type inference can be used more broadly in Haskell, but explicit type declarations should be used for any function called from outside.

We require that `makeRat x y` satisfy the *precondition*:

```
y /= 0
```

The function generates an explicit error exception if it does not.

As a *postcondition*, we require `makeRat x y` to return a result `(x',y')` such that:

- `(x',y')` satisfies the Rational Representation Property
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

Note: Together the two postcondition requirements imply that $\frac{\texttt{x'}}{\texttt{y'}} = \frac{\texttt{x}}{\texttt{y}}$.

The function `signum'` (similar to the more general function `signum` in the Prelude) takes an integer and returns the integer `-1`, `0`, or `1` when the number is negative, zero, or positive, respectively.

```
signum' :: Int -> Int
signum' n | n == 0     =  0
          | n > 0      =  1
          | otherwise  = -1
```

The function `abs'` (similar to the more general function `abs` in the Prelude) takes an integer and returns its absolute value.

```
abs' :: Int -> Int
abs' n | n >= 0    =  n
       | otherwise = -n
```

The function `gcd'` (similar to the more general function `gcd` in the Prelude) takes two integers and returns their greatest common divisor.

```
gcd' :: Int -> Int -> Int
gcd' x y = gcd'' (abs' x) (abs' y)
    where gcd'' x 0 = x
          gcd'' x y = gcd'' y (x `rem` y)
```

Prelude operation `rem` returns the remainder from dividing its first operand by its second.

Given a tuple `(x,y)` constructed by `makeRat` as defined above, we can define `numer (x,y)` and `denom (x,y)` as follows:

```
numer, denom :: Rat -> Int
numer (x,_) = x
denom (_,y) = y
```

The preconditions of both `numer (x,y)` and `denom (x,y)` are that their arguments `(x,y)` satisfy the Rational Representation Property.

The postcondition of `numer (x,y) = x` is that the rational number values $\frac{\texttt{x}}{\texttt{numer (x,y)}} = \frac{\texttt{x}}{\texttt{y}}$.

Similarly, the postcondition of `denom (x,y) = y` is that the rational number values $\frac{\texttt{denom (x,y)}}{y} = \frac{\texttt{x}}{\texttt{y}}$.

Finally, to allow rational numbers to be displayed in the normal fractional representation, we include function `showRat` in the package. We use function `show`, found in the Prelude, here to convert an integer to the usual string format and use the list operator ++ to concatenate the two strings into one.

```
showRat :: Rat -> String
showRat x = show (numer x) ++ "/" ++ show (denom x)
```

Unlike `Rat`, `zeroRat`, `makeRat`, `numer`, and `denom`, function `showRat` (as implemented) does not use knowledge of the data representation. We could optimize it slightly by allowing it to access the structure of the tuple directly.

7.3.3 Rational number modularization

There are three groups of functions in this package:

1. the six public rational arithmetic functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`
2. the public type `Rat`, constant `zeroRat`, public constructor function `makeRat`, public selector functions `numer` and `denom`, and string conversion function `showRat`
3. the private utility functions called only by the second group, but just reimplementations of Prelude functions anyway

7.3.3.1 Module `RationalCore` As we have seen, data type `Rat`; constant `zeroRat`; functions `makeRat`, `numer`, `denom`, and `showRat`; and the functions' preconditions and postconditions form the *interface* to the *data abstraction*.

The data abstraction hides the information about the representation of the data. We can *encapsulate* this group of functions in a Haskell module as follows. This source code must also be in a file named `RationalCore.hs`.

```
module RationalCore
    (Rat, makeRat, zeroRat, numer, denom, showRat)
where
    -- Rat,makeRat,zeroRat,numer,denom,showRat definitions
```

In terms of the information-hiding approach, the secret of the `RationalCore` module is the rational number data representation used.

We can encapsulate the utility functions in a separate module, which would enable them to be used by several other modules.

However, given that the only use of the utility functions is within the data representation module, we choose not to separate them at this time. We leave them as local functions in the data abstraction module. Of course, we could also eliminate them and use the corresponding Prelude functions directly.

7.3.3.2 Module `Rational`

Similarly, functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat` use the core data abstraction and, in turn, extend the interface to include rational number arithmetic operations.

We can encapsulate these in another Haskell module that imports the module giving the data representation. This module must be in a file named `Rational1.hs`.

```
module Rational1
  ( Rat, zeroRat, makeRat, numer, denom, showRat,
    negRat, addRat, subRat, mulRat, divRat, eqRat )
where
    import RationalCore
    -- negRat,addRat,subRat,mulRat,divRat,eqRat definitions
```

Other modules that use the rational number package can import module `Rational1`.

7.3.3.3 Modularization critique

The modularization described above:

- enables a module to be reused in several different programs
- offers robustness with respect to change

  The data representation and arithmetic algorithms can change independently.
- allows multiple implementations of each module as long as the public (abstract) interface is kept stable
- enables understanding of one module without understanding the internal details of modules it uses
- costs some in terms of extra code and execution efficiency

  But that probably does not matter given the benefits above and the code optimizations carried out by the compiler.

However, the modularization does not hide the representation fully because it uses a concrete data structure—a pair of integers—to represent a rational number. In chapter 21, we see how to use a user-defined data type to hide the representation fully.

7.3.4 Alternative data representation

In the rational number data representation above, constructor `makeRat` creates pairs in which the two integers are relatively prime and the sign is on the numerator. Selector functions `numer` and `denom` just return these stored values.

An alternative representation is to reverse this approach, as shown in the following module (in file `RationalDeferGCD.hs`.)

```haskell
module RationalDeferGCD
    (Rat, zeroRat, makeRat, numer, denom, showRat)
where

type Rat = (Int,Int)

zeroRat :: (Int,Int)
zeroRat = (0,1)

makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ show x ++ "/0" )
makeRat 0 y = zeroRat
makeRat x y = (x,y)

numer :: Rat -> Int
numer (x,y) = x' `div` d
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'

denom :: Rat -> Int
denom (x,y) = y' `div` d
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'

showRat :: Rat -> String
showRat x = show (numer x) ++ "/" ++ show (denom x)
```

This approach defers the calculation of the greatest common divisor until a selector is called.

In this alternative representation, a rational number `(x,y)` must satisfy all parts of the following **Deferred Representation Property**:

- `(x,y)` ∈ `(Int,Int)`
- `y /= 0`
- if `x == 0` , then `y == 1`
- rational number value is $\frac{x}{y}$

Furthermore, we require that `makeRat x y` satisfies the *precondition*:

```haskell
y /= 0
```

The function generates an explicit error condition if it does not.

As a *postcondition*, we require `makeRat x y` to return a result `(x',y')` such that:

- `(x',y')` satisfies the Deferred Representation Property
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

The preconditions of both `numer (x,y)` and `denom (x,y)` are that `(x,y)` satisfies the Deferred Representation Property.

The postcondition of `numer (x,y) = x'` is that the rational number values $\frac{\texttt{x'}}{\texttt{numer (x,y)}} = \frac{\texttt{x}}{\texttt{y}}$.

Similarly, the postcondition of `denom (x,y) = y` is that the rational number values $\frac{\texttt{denom (x,y)}}{y'} = \frac{\texttt{x}}{\texttt{y}}$.

Question:

> What are the advantages and disadvantages of the two data representations?

Like module `RationalCore`, the design secret for this module, `RationalDeferGCD`, is the rational number data representation.

Regardless of which approach is used, the definitions of the arithmetic and comparison functions do not change. Thus the `Rational` module can import data representation module `RationalCore` or `RationalDeferGCD`.

Figure 7.1 shows the dependencies among the modules we have examined in the rational arithmetic example.

We can consider the `RationalCore` and `RationalDeferGCD` modules as two concrete instances (Haskell `module`s) of a more abstract module we call `RationalRep` in the diagram.

The module `Rational` relies on the abstract module `RationalRep` for an implementation of rational numbers. In the Haskell code above, there are really two versions of the Haskell module `Rational` that differ only in whether they import `RationalCore` or `RationalDeferGCD`.

Chapter 21 introduces user-defined (algebraic) data types. Instead of concrete data types (e.g., the `Int` pairs used by the type alias `Rat`), we can totally hide the details of the data representation using modules.

7.3.5 Haskell information-hiding modules

In the Rational Arithmetic example, we defined two information-hiding modules:

1. "RationalRep", whose secret is how to represent the rational number data and whose interface consists of the data type `Rat`, constant `zeroRat`, operations (functions) `makeRat`, `numer`, `denom`, and `showRat`, and the constraints on these types and functions

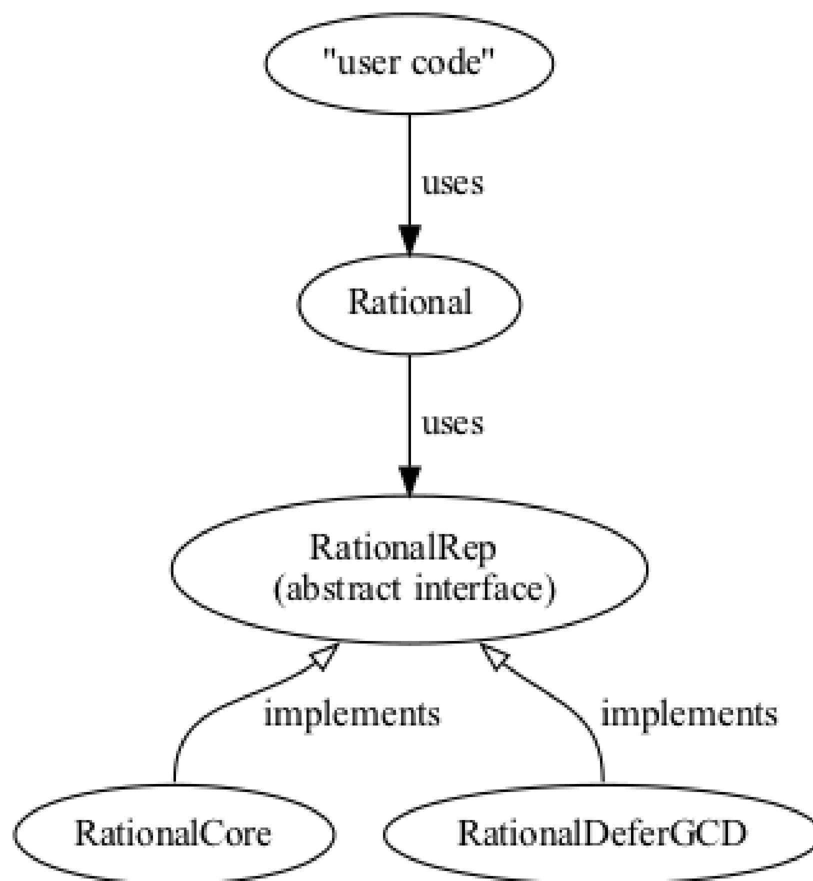


Figure 7.1: Rational package module dependencies.

2. "Rational", whose secret is how to implement the rational number arithmetic and whose interface consists of operations (functions) `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`, the other module's interface, and the constraints on these types and functions

We developed two distinct Haskell modules, `RationalCore` and `RationalDeferGCD`, to implement the "RationalRep" information-hiding module.

We developed one distinct Haskell module, `Rational`, to implement the "Rational" information-hiding module. This module can be paired (i.e., by changing the `import` statement) with either of the other two variants of "RationalRep" module. (Source file `Rational1.hs` imports module `RationalCore`; source file `Rational2.hs` imports module `RationalDeferGCD`.)

Unfortunately, Haskell 2010 has a relatively weak module system that does not support multiple implementations as well as we might like. There is no way to declare that multiple Haskell modules have the same interface other than copying the common code into each module and documenting the interface carefully. We must also have multiple versions of `Rational` that differ only in which other module is imported.

Together the Glasgow Haskell Compiler (GHC) release 8.2 (July 2017) and the Cabal-Install package manager release 2.0 (August 2017) support a new extension, the Backpack mixin package system. This new system remedies the above shortcoming. In this new approach, we would define the abstract module "RationalRep" as a signature file and require that `RationalCore` and `RationalDeferGCD` conform to it.

Further discussion of this new module system is beyond the scope of this chapter.

7.3.6 Rational number testing

Chapter 12 discusses testing of the Rational modules designed in this chapter. The test scripts for the following modules are in the files shown:

- Module `RationalRep`
  - `TestRatRepCore.hs` for module instance `RationalCore`
  - `TestRatRepDefer.hs` for module instance `RationalDeferGCD`
- Module `Rational`
  - `TestRational1.hs` for `Rational` using `RationalCore`.
  - `TestRational2.hs` for `Rational` using `RationalDeferGCD`.

7.4 Module invariants

As we see in the rational arithmetic example, a module that provides a data abstraction must ensure that the objects it creates and manipulates maintain their integrity—always have a valid structure and state.

- The `RationalCore` rational number representation satisfies the Rational Representation Property.
- The `RationalDeferGCD` rational number representation satisfies the Deferred Representation Property.

These properties are *invariants* for those modules. An invariant for the data abstraction can help us design and implement such objects.

Invariant: A logical assertion that must always be true for every "object" created by the public constructors and manipulated only by the public operations of the data abstraction.

Often, we separate an invariant into two parts.

Interface invariant: An invariant stated in terms of the public features and abstract properties of the "object".

Implementation (representation) invariant: A detailed invariant giving the required relationships among the internal features of the implementation of an "object"

An interface invariant is a key aspect of the *abstract interface* of a module. It is useful to the users of the module, as well to the developers.

7.4.1 RationalRep modules

In the Rational Arithmetic example, the *interface invariant* for the "RationalRep" abstract module is the following.

RationalRep Interface Invariant: For any valid Haskell rational number `r`, all the following hold:

- `r` ∈ `Rat`
- `denom r > 0`
- if `numer r == 0`, then `denom r == 1`
- `numer r` and `denom r` are relatively prime
- the (mathematical) rational number value is $\frac{\texttt{numer r}}{\texttt{denom r}}$

We note that the *precondition* for `makeRat x y` is defined above without any dependence upon the concrete representation.

```
y /= 0
```

We can restate the *postcondition* for `makeRat x y = r` generically to require both of the following to hold:

- `r` satisfies the RationaRep Interface Invariant
- rational number `r` 's value is $\frac{\texttt{x}}{\texttt{y}}$

The preconditions of both `numer r` and `denom r` are that their argument `r` satisfies the RationalRep Interface Invariant.

The postcondition of `numer r = x'` is that the rational number value $\frac{\texttt{x'}}{\texttt{denom r}}$ is equal to the rational number value of `r`.

Similarly, the postcondition of `denom r = y'` is that the rational number value $\frac{\texttt{numer r}}{y'}$ is equal to the rational number value of `r`.{.haskell}

An implementation invariant guides the developers in the design and implementation of the internal details of a module. It relates the internal details to the interface invariant.

7.4.1.1 `RationalCore`

We can state an implementation invariant for the `RationalCore` module.

RationalCore Implementation Invariant: For any valid Haskell rational number `r`, all the following hold:

- `r == (x,y)` for some `(x,y)` $\in$ `Rat`
- `y > 0`
- if `x == 0`, then `y == 1`
- `x` and `y` are relatively prime
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

The implementation invariant implies the interface invariant given the definitions of data type `Rat` and selector functions `numer` and `denom`. Constructor function `makeRat` does the work to establish the invariant initially.

7.4.1.2 `RationalDeferGCD`

We can state an implementation invariant for the `RationalDeferGCD` module.

RationalDeferGCD Implementation Invariant: For any valid Haskell rational number `r`, all the following hold:

- `r == (x,y)` for some `(x,y)` $\in$ `Rat`
- `y /= 0`
- if `x == 0`, then `y == 1`
- rational number value is $\frac{\texttt{x}}{\texttt{y}}$

The implementation invariant implies the interface invariant given the definitions of `Rat` and of the selector functions `numer` and `denom`. Constructor function `makeRat` is simple, but the selector functions `numer` and `denom` do quite a bit of work to establish the interface invariant.

7.4.2 Rational modules

The `Rational` abstract module extends the `RationalRep` abstract module with new functionality.

- It imports the public interface of the `RationalRep` abstract module and exports those features in its own public interface. Thus it must maintain the interface invariant for the `RationalRep` module it uses.
- It does not add any new data types or constructor (or destructor) functions. So it does not need any new invariant components for new data abstractions.
- It adds one unary and four binary arithmetic functions that take rational numbers and return a rational number. It does so by using the data abstraction provided by the `RationalRep` module. These must preserve the `RationalRep` interface invariant.
- It adds an equality comparison function that takes two rational numbers and returns a `Bool`.

7.5 What Next?

Chapter 6 examined procedural abstraction and stepwise refinement and used the method to develop a square root package.

This chapter (7) examined data abstraction and used the method to develop a rational number arithmetic package. The chapters explored concepts and methods for modular design and programming using Haskell, including preconditions, postconditions, and invariants.

We continue to use these concepts, techniques, and examples in the rest of the book. In particular:

- Chapter 12 examines how to test the modules developed in this chapter.
- Chapter 22 explores the data abstraction concepts and techniques in more depth. In particular, it examines a detailed case study of an abstract data type.

The next chapter, Chapter 8, examines the substitution model for evaluation of Haskell programs and explores efficiency and termination in the context of that model.

7.6 Chapter Source Code

The Haskell source code for this chapter includes the following:

- Two versions of a lower-level "RationalnRep" module that gives implementations of rational number given in the following files.
  - `RationalCore.hs`.

- `RationalDeferGCD.hs`.)

- An upper-level rational arithmetic module given in the following files.

  - `Rational1.hs`, a variant that imports the `RationalCore` module

  - `Rational2.hs`, a variant that imports the `RationalDeferGCD` module

7.7 Exercises

For each of the following exercises, develop and test a Haskell function or set of functions.

1. Develop a Haskell module (or modules) for line segments on the two-dimensional coordinate plane using the *rectangular coordinate* system.

   We can represent a line segment with two points—the starting point and the ending point. Develop the following Haskell functions:

   - constructor `newSeg` that takes two points and returns a new line segment

   - selectors `startPt` and `endPt` that each take a segment and return its starting and ending points, respectively

   We normally represent the plane with a *rectangular coordinate* system. That is, we use two axes—an `x` *axis* and a `y` *axis*—intersecting at a right angle. We call the intersection point the *origin* and label it with 0 on both axes. We normally draw the `x` axis horizontally and label it with increasing numbers to the right and decreasing numbers to the left. We also draw the `y` axis vertically with increasing numbers upward and decreasing numbers downward. Any point in the plane is uniquely identified by its `x`-coordinate and `y`-coordinate.

   Define a data representation for points in the rectangular coordinate system and develop the following Haskell functions:

   - constructor `newPtFromRect` that takes the `x` and `y` coordinates of a point and returns a new point

   - selectors `getx` and `gety` that takes a point and returns the `x` and `y` coordinates, respectively

   - display function `showPt` that takes a point and returns an appropriate `String` representation for the point

   Now, using the various constructors and selectors, also develop the Haskell functions for line segments:

   - `midPt` that takes a line segment and returns the point at the middle of the segment

- display function `showSeg` that takes a line segment and returns an appropriate `String` representation

Note that `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` can be implemented independently from how the points are represented.

2. Develop a Haskell module (or modules) for line segments that represents points using the *polar coordinate* system instead of the rectangular coordinate system used in the previous exercise.

   A polar coordinate system represents a point in the plane by its *radial coordinate* `r` (i.e., the distance from the *pole*) and its *angular coordinate* `t` (i.e., the angle from the *polar axis* in the reference direction). We sometimes call `r` the *magnitude* and `t` the *angle*.

   By convention, we align the rectangular and polar coordinate systems by making the origin the pole, the positive portion of the `x` axis the polar axis, and let the first quadrant (where both `x` and `y` are positive) be the smallest positive angles in the reference direction. That is, with a traditional drawing of the coordinate systems, we measure and the radial coordinate `r` as the distance from the origin measure the angular coordinate `t` counterclockwise from the positive `x` axis.

   Using knowledge of trigonometry, we can convert among rectangular coordinates `(x,y)` and polar coordinates `(r,t)` using the equations:

   ```
   x = r * cos(t)
   y = r * sin(t)
   r = sqrt(x^2 + y^2)
   t = arctan2(y,x)
   ```

   Define a data representation for points in the polar coordinate system and develop the following Haskell functions:

   - constructor `newPtFromPolar` that takes the magnitude `r` and angle `t` as the polar coordinates of a point and returns a new point
   - selectors `getMag` and `getAng` that each take a point and return the magnitude `r` and angle `t` coordinates, respectively
   - selectors `getx` and `gety` that return the `x` and `y` components of the points (represented here in polar coordinates)
   - display functions `showPtAsRect` and `showPtAsPolar` to convert the points to strings using rectangular and polar coordinates, respectively,

   Functions `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` should work as in the previous exercise.

3. Modify the solutions to the previous two line-segment module exercises to enable the line segment functions to be in one module that works properly

if composed with either of the two data representation modules. (The solutions may have already done this.)

4. Modify the solution to the previous line-segment exercise to use the Backpack module system.

5. Modify the modules in the previous exercise to enable the line segment module to work with both data representations in the same program.

6. Modify the solution to the Rational Arithmetic example to use the Backpack module system.

7. State preconditions and postconditions for the functions in abstract module `Rational`.

7.8 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from my previous materials:

- Discussion of the Rational Arithmetic modules mostly from chapter 5 of my *Notes on Functional Programming with Haskell* [42], from my Lua-based implementations, and from section 2.1 of Abelson and Sussman's *Structure and Interpretation of Computer Programs* [1].

- Discussion of modular design and programming issues from my Data Abstraction [46] and Modular Design [47] notes, which drew ideas over the past 25 years from a variety of sources [20,22,56,58,59,61,99,100,128,129,134,136,137,170,177].

In 2017, I continued to develop this work as Sections 2.6-2.7 in Chapter 2, Basic Haskell Functional Programming, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs; previous Section 2.4 became Section 5.3 in the new Chapter 5, Types; and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction, and Chapter 7, Data Abstraction (this chapter).

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

7.9 Terms and Concepts

TODO: Update

Haskell `module`, module exports and imports, module dependencies, rational number arithmetic, data abstraction, properties of data, data representation, precondition, postcondition, invariant, interface invariant, implementation or representation invariant, canonical or normal forms, relatively prime, information hiding, module secret, encapsulation, interface, abstract interface, type inference.

8 Evaluation Model

8.1 Chapter Introduction

This chapter (8) introduces an evaluation model applicable to Haskell programs. As in the previous chapters, this chapter focuses on use of first-order functions and primitive data types.

The goals of this chapter (8) are to:

- describe an evaluation model appropriate for Haskell programs
- enable students to analyze Haskell functions to determine under what conditions they terminate normally and how efficient they are

Building on this model, Chapter 9 informally analyzes simple functions in terms of time and space efficiency and termination. Chapter 29 examines these issues in more depth.

How can we evaluate (i.e., execute) an expression that "calls" a function like the `fact1` function from Chapter 4?

We do this by rewriting expressions using a substitution model, as we see in this chapter. This process depends upon a property of functional languages called referential transparency.

8.2 Referential Transparency Revisited

Referential transparency is probably the most important property of modern functional programming languages.

As defined in Chapter 2, referential transparency means that, within some well-defined context (e.g., a function or module definition), a variable (or other symbol) *always* represents the *same value*.

Because a variable always has the same value, we can replace the variable in an expression by its value or vice versa. Similarly, if two subexpressions have equal values, we can replace one subexpression by the other. That is, "equals can be replaced by equals".

Pure functional programming languages thus use the same concept of a variable that mathematics uses.

However, in most imperative programming languages, a variable represents an address or "container" in which values may be stored. A program may change the value stored in a variable by executing an assignment statement. Thus these mutable variables break the property of referential transparency.

Because of referential transparency, we can construct, reason about, and manipulate functional programs in much the same way we can any other mathematical expressions. Many of the familiar "laws" from high school algebra still hold;

new laws can be defined and proved for less familiar primitives and even user-defined operators. This enables a relatively natural equational style of reasoning using the actual expressions of the language. We explore these ideas further in Chapters 25, 26, and 27.

In contrast, to reason about imperative programs, we usually need to go outside the language itself and use notation that represents the semantics of the language {[41]; [85]].

For our purposes here, referential transparency underlies the substitution model for evaluation of expressions in Haskell programs.

8.3 Substitution Model

The *substitution model* (or *reduction model*) involves rewriting (or reducing) an expression to a "simpler" equivalent form. It involves two kinds of replacements:

- replacing a subexpression that satisfies the left-hand side of an equation by the right-hand side with appropriate substitution of arguments for parameters
- replacing a primitive application (e.g., + or *) by its value

The term *redex* refers to a subexpression that can be reduced.

Redexes can be selected for reduction in several ways. For instance, the redex can be selected based on its position within the expression:

- *leftmost redex first*, where the leftmost reducible subexpression in the expression text is reduced before any other subexpressions are reduced
- *rightmost redex first*, where the rightmost reducible subexpression in the expression text is reduced before any other subexpressions are reduced

The redex can also be selected based on whether or not it is contained within another redex:

- *outermost redex first*, where a reducible subexpression that is not contained within any other reducible subexpression is reduced before one that is contained within another
- *innermost redex first*, where a reducible subexpression that contains no other reducible subexpression is reduced before one that contains others

We will explore these more fully in a Chapter 29. In most circumstances, Haskell uses a *leftmost outermost redex first* approach.

In Chapter 4, we defined factorial function `fact1` as shown below. (The source code is in file `Factorial.hs`){type="text/plain"}.)

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
```

```
        else
            n * fact1 (n-1)
```

Consider the expression from `else` clause in `fact1` with `n` having the value `2`:

```
2 * fact1 (2-1)
```

This has two redexes: subexpressions `2-1` and `fact1 (2-1)`.

The multiplication cannot be reduced because it requires both of its arguments to be evaluated.

A function parameter is said to be *strict* if the value of that argument is always required. Thus, multiplication is strict in both its arguments. If the value of an argument is not always required, then it is *nonstrict*.

The first redex `2-1` is an innermost redex. Since it is the only innermost redex, it is both leftmost and rightmost.

The second redex `fact1 (2-1)` is an outermost redex. Since it is the only outermost redex, it is both leftmost and rightmost.

Now consider the complete evaluation of the expression `fact1 2` using leftmost outermost reduction steps. Below we denote the steps with $\Longrightarrow$ and give the substitution performed between braces.

```
fact1 2
```

$\Longrightarrow$ { replace `fact1 2` using definition }

```
if 2 == 0 then 1 else 2 * fact1 (2-1)
```

$\Longrightarrow$ { evaluate `2 == 0` in condition }

```
if False then 1 else 2 * fact1 (2-1)
```

$\Longrightarrow$ { evaluate `if` }

```
2 * fact1 (2-1)
```

$\Longrightarrow$ { replace `fact1 (2-1)` using definition, add implicit parentheses }

```
2 * (if (2-1) == 0 then 1 else (2-1) * fact1 ((2-1)-1))
```

$\Longrightarrow$ { evaluate `2-1` in condition }

```
2 * (if 1 == 0 then 1 else (2-1) * fact1 ((2-1)-1))
```

$\Longrightarrow$ { evaluate `1 == 0` in condition }

```
2 * (if False then 1 else (2-1) * fact1 ((2-1)-1))
```

$\Longrightarrow$ { evaluate `if` }

```
2 * ((2-1) * fact1 ((2-1)-1))
```

$\Longrightarrow$ { evaluate leftmost `2-1` }

```
     2 * (1 * fact1 ((2-1)-1))
⟹ { replace fact1 ((2-1)-1) using definition, add implicit parentheses }
     2 * (1 * (if ((2-1)-1) == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹ { evaluate 2-1 in condition }
     2 * (1 * (if (1-1) == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹ { evaluate 1-1 in condition }
     2 * (1 * (if 0 == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹ { evaluate 0 == 0 }
     2 * (1 * (if True then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹ { evaluate if }
     2 * (1 * 1)
⟹ { evaluate 1 * 1 }
     2 * 1
⟹ { evaluate 2 * 1 }
     2
```

The rewriting model we have been using so far can be called *string reduction* because our model involves the textual replacement of one string by an equivalent string.

A more efficient alternative is *graph reduction*. In this technique, the expressions are represented as (directed acyclic) expression graphs rather than text strings. The repeated subexpressions of an expression are represented as shared components of the expression graph. Once a shared component has been evaluated once, it need not be evaluated again.

In the example above, subexpression `2-1` is reduced three times. However, all of those subexpressions come from the initial replacement of `fact1 2`. Using graph reduction, only the first of those reductions is necessary.

```
     fact1 2
⟹ { replace fact1 2 using definition }
     if 2 == 0 then 1 else 2 * fact1 (2-1)
⟹ { evaluate 2 == 0 in condition }
     if False then 1 else 2 * fact1 (2-1) }
```

⟹ { evaluate `if` }

```
2 * fact1 (2-1)
```

⟹ { replace `fact1 (2-1)` using definition, add implicit parentheses }

```
2 * (if (2-1) == 0 then 1 else (2-1) * fact1 ((2-1)-1))
```

⟹ { evaluate `2-1` because of condition (3 occurrences in graph) }

```
2 * (if 1 == 0 then 1 else 1 * fact1 (1-1))
```

⟹ { evaluate `1 == 0` }

```
2 * (if False then 1 else 1 * fact1 (1-1))
```

⟹ { evaluate `if` }

```
2 * (1 * fact1 (1-1))
```

⟹ { replace `fact1 ((1-1)` using definition, add implicit parentheses }

```
2 * (1 * (if (1-1) == 0 then 1 else (1-1) * fact1 ((1-1)-1))
```

⟹ { evaluate `1-1` because of condition (3 occurrences in graph) }

```
2 * (1 * (if 0 == 0 then 1 else 0 * fact1 (0-1))
```

⟹ { evaluate `0 == 0` }

```
2 * (1 * (if True then 1 else  0 * fact1 (0-1))
```

⟹ { evaluate `if` }

```
2 * (1 * 1)
```

⟹ { evaluate `1 * 1` }

```
2 * 1
```

⟹ { evaluate `2 * 1` }

```
2
```

In general, the Haskell compiler or interpreter uses a leftmost outermost graph reduction technique. However, if the value of a function's argument is always needed for a computation, then an innermost reduction can be triggered for that argument. Either the programmer can explicitly require this or the compiler can detect the situation and automatically trigger the innermost reduction order.

Haskell exhibits *lazy evaluation*. That is, an expression is not evaluated until its value is needed, if ever. An outermost reduction corresponds to this evaluation strategy.

Other functional languages such as Scala and F# exhibit *eager evaluation*. That is, an expression is evaluated as soon as possible. An innermost reduction corresponds to this evaluation strategy.

8.4 Time and Space Complexity

We state efficiency (i.e., time complexity or space complexity) of programs in terms of the "Big-O" notation and asymptotic analysis.

For example, consider the leftmost outermost graph reduction of function `fact1` above. The number of reduction steps required to evaluate `fact1 n` is `5*n + 3`.

We let the number of steps in a graph reduction be our measure of time. Thus, the *time complexity* of `fact1 n` is O(`n`), which means that the time to evaluate `fact1 n` is bounded above by some (mathematical) function that is proportional to the value of `n`.

Of course, this result is easy to see in this case. The algorithm is dominated by the `n` multiplications it must carry out. Alternatively, we see that evaluation requires on the order of `n` recursive calls.

We let the number of *arguments* in an expression graph be our measure of the *size* of an expression. Then the *space complexity* is the maximum size needed for the evaluation in terms of the input.

This size measure is an indication of the maximum size of the unevaluated expression that is held at a particular point in the evaluation process. This is a bit different from the way we normally think of space complexity in imperative algorithms, that is, the number of "words" required to store the program's data.

However, this is not as strange as it may at first appear. As we in later chapters, the data structures in functional languages like Haskell are themselves *expressions* built by applying constructors to simpler data.

In the case of the graph reduction of `fact1 n`, the size of the largest expression is `2*n + 16`. This is a multiplication for each integer in the range from 1 to `n` plus 16 for the full `if` statement. Thus the space complexity is O(`n`).

The Big-O analysis is an asymptotic analysis. That is, it estimates the order of magnitude of the evaluation time or space as the size of the input approaches infinity (gets large). We often do worst case analyses of time and space. Such analyses are usually easier to do than average-case analyses.

The time complexity of `fact1 n` is similar to that of a loop in an imperative program. However, the space complexity of the imperative loop algorithm is O(1). So `fact1` is not space efficient compared to the imperative loop.

We examine techniques for improving the efficiency of functions below. In Chapter 29, we examine reduction techniques more fully.

8.5 Termination

A recursive function has one or more recursive cases and one or more base (nonrecursive) cases. It may also be undefined for some cases.

To show that evaluation of a recursive function terminates, we must show that each recursive application *always* gets closer to a termination condition represented by a base case.

Again consider `fact1` defined above.

If `fact1` is called with argument `n` greater than 0, the argument of the recursive application in the `else` clause always decreases to `n - 1`. Because the argument always decreases in integer steps, it must eventually reach 0 and, hence, terminate in the first leg of the definition.

If we call `fact1` with argument 0, the function terminates immediately.

What if we call `fact1` with its argument less than 0? We consider this issue below.

8.6 What Next?

This chapter (8) introduced an evaluation model applicable to Haskell programs. It provides a framework for analyzing Haskell functions to determine under what conditions they terminate normally and how efficient they are.

Chapter 9 informally analyzes simple functions in terms of time and space efficiency and termination.

Chapter 29 examines these issues in more depth.

8.7 Exercises

1. Given the following definition of Fibonacci function `fib`, show the reduction of `fib 4`.

    ```
    fib :: Int -> Int
    fib 0          = 0
    fib 1          = 1
    fib n | n >= 2 = fib (n-1) + fib (n-2)
    ```

2. What are the time and space complexities of `fact6` as defined in the previous exercise?

3. Given the following definition of `fact6`, show the reduction of `fact6 2`.

    ```
    fact6 :: Int -> Int
    fact6 n = factIter n 1

    factIter :: Int -> Int -> Int
    factIter 0 r         = r
    factIter n r | n > 0 = factIter (n-1) (n*r)
    ```

4. What are the time and space complexities of `fact6` as defined in the previous exercise?

8.8 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from parts of chapters 1 and 13 of my *Notes on Functional Programming with Haskell* [42]. These chapters had evolved from my study of the work by Bird and Wadler [15], Hudak [101], Wentworth [178], Thompson [171], and others as I was learning functional programming.

In 2017, I continued to develop this work as Chapter 3, Evaluation and Efficiency, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Evaluation and Efficiency chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 3.1-3.2 became the basis for new Chapter 8, Evaluation Model (this chapter), and previous sections 3.3-3.5 became the basis for Chapter 9, Recursion Styles and Efficiency. I also moved the discussion of preconditions and postconditions to the new Chapter 6.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

8.9 Terms and Concepts

Referential transparency, reducible expression (redex), reduction strategies (leftmost vs. rightmost, innermost vs. outermost), string and graph reduction models, time and space complexity, termination preconditions, postconditions, contracts.

9 Recursion Styles and Efficiency

9.1 Chapter Introduction

This chapter () introduces basic recursive programming styles and examines issues of efficiency, termination, and correctness. It builds on the substitution model from Chapter 8, but uses the model informally.

As in the previous chapters, this chapter focuses on use of first-order functions and primitive data types.

The goals of the chapter are to:

- explre several recursive programming styles—linear and nonlinear, backward and forward, tail, and logarithmic—and their implementation using Haskell
- analyze Haskell functions to determine under what conditions they terminate with the correct result and how efficient they are
- explore methods for developing recursive Haskell programs that terminate with the correct result and are efficient in both time and space usage
- compare the basic functional programming syntax of Haskell with that in other languages

9.2 Linear and Nonlinear Recursion

Given the substitution model described in Chapter 8, we can now consider efficiency and termination in the design of recursive Haskell functions.

In this section, we examine the concepts of linear and nonlinear recursion. The following two sections examine other styles.

9.2.1 Linear recursion

A function definition is *linear recursive* if at most one recursive application of the function occurs in any leg of the definition (i.e., along any path from an entry to a return). The various argument patterns and guards and the branches of the conditional expression `if` introduce paths.

The definition of the function `fact4` repeated below is linear recursive because the expression in the second leg of the definition (i.e., `n * fact4 (n-1)`) involves a single recursive application. The other leg is nonrecursive; it is the base case of the recursive definition.

```
fact4 :: Int -> Int
fact4 n
    | n == 0 =  1
    | n >= 1 =  n * fact4 (n-1)
```

What are the precondition and postcondition for `fact4 n`?

As discussed in Chapter 6, we must require a precondition of `n >= 0` to avoid abnormal termination. When the precondition holds, the postcondition is:

> `fact4 n` = *fact'(n)*

What are the time and space complexities of `fact4 n`?

Function `fact4` recurses to a depth of `n`. As we in for `fact1` in Chapter 8, it has *time complexity* O(`n`), if we count either the recursive calls or the multiplication at each level. The *space complexity* is also O(`n`) because a new runtime stack frame is needed for each recursive call.

How do we know that function `fact4 n` terminates?

For a call `fact4 n` with `n > 0`, the argument of the recursive application always decreases to `n - 1`. Because the argument always decreases in integer steps, it must eventually reach 0 and, hence, terminate in the first leg of the definition.

9.2.2 Nonlinear recursion

A *nonlinear recursion* is a recursive function in which the evaluation of some leg requires more than one recursive application. For example, the naive Fibonacci number function `fib` shown below has two recursive applications in its third leg. When we apply this function to a nonnegative integer argument greater than 1, we generate a pattern of recursive applications that has the "shape" of a binary tree. Some call this a *tree recursion*.

```
fib :: Int -> Int
fib 0          = 0
fib 1          = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
```

What are the precondition and postcondition for `fib n`?

For `fib n`, the precondition `n >= 0` to ensure that the function is defined. When called with the precondition satisfied, the postcondition is:

> `fib n` = *Fibonacci*(`n`)

How do we know that `fib n` terminates?

For the recursive case `n >= 2`. the two recursive calls have arguments that are 1 or 2 less than `n`. Thus every call gets closer to one of the two base cases.

What are the time and space complexities of `fib n`?

Function `fib` is combinatorially explosive, having a time complexity O(`fib n`). The space complexity is O(`n`) because a new runtime stack frame is needed for each recursive call and the calls recurse to a depth of `n`.

An advantage of a linear recursion over a nonlinear one is that a linear recursion can be compiled into a *loop* in a straightforward manner. Converting a nonlinear recursion to a loop is, in general, difficult.

9.3 Backward and Forward Recursion

In this section, we examine the concepts of backward and forward recursion.

9.3.1 Backward recursion

A function definition is *backward recursive* if the recursive application is embedded within another expression. During execution, the program must complete the evaluation of the expression after the recursive call returns. Thus, the program must preserve sufficient information from the outer call's environment to complete the evaluation.

The definition for the function `fact4` above is backward recursive because the recursive application `fact4 (n-1)` in the second leg is *embedded within the expression* `n * fact4 (n-1)`. During execution, the multiplication must be done after return. The program must "remember" (at least) the value of parameter `n` for that call.

A compiler can translate a backward linear recursion into a loop, but the translation may require the use of a stack to store the program's *state* (i.e., the values of the variables and execution location) needed to complete the evaluation of the expression.

Often when we design an algorithm, the first functions we come up with are backward recursive. They often correspond directly to a convenient recurrence relation. It is often useful to convert the function into an equivalent one that evaluates more efficiently.

9.3.2 Forward recursion

A function definition is *forward recursive* if the recursive application is *not embedded within another expression*. That is, the *outermost expression is the recursive application* and any other subexpressions appear in the argument lists. During execution, significant work is done as the recursive calls are made (e.g., in the argument list of the recursive call).

The definition for the auxiliary function `factIter` below has two integer arguments. The first argument is the number whose factorial is to be computed. The second argument accumulates the product incrementally as recursive calls are made.

The recursive application `factIter (n-1) (n*r)` in the second leg is on the outside of the expression evaluated for return. The other leg of `factIter` and `fact6` itself are nonrecursive.

```
fact6 :: Int -> Int
fact6 n = factIter n 1

factIter :: Int -> Int -> Int
factIter 0 r         = r
factIter n r | n > 0 = factIter (n-1) (n*r)
```

What are the precondition and postcondition for `factIter n r`?

To avoid termination, `factIter n r` requires `n >= 0`. Its postcondition is that:

`factIter n r = r *` *fact*`(n)`

How do we know that `factIter n r` terminates?

Argument `n` of the recursive leg is at least 1 and decreases by 1 on each recursive call.

What is the time and space complexity of `factIter n r`?

Function `factIter n r` has a time complexity O(`n`). But, if the compiler converts the `factIter` recursion to a loop, the time complexity's constant factor should be smaller than that of `fact4`.

As shown, `factIter n r` has space complexity of O(`n`). But, if the compiler does an innermost reduction on the second argument (because its value will always be needed), then the space complexity of `factIter` becomes O(1).

9.3.3 Tail recursion

A function definition is *tail recursive* if it is *both forward recursive and linear recursive*. In a tail recursion, the last action performed before the return is a recursive call.

The definition of the function `factIter` above is thus tail recursive.

Tail recursive definitions are relatively straightforward to compile into efficient loops. There is no need to save the states of unevaluated expressions for higher level calls; the result of a recursive call can be returned directly as the caller's result. This is sometimes called *tail call optimization* (or "tail call elimination" or "proper tail calls") [202].

In converting the backward recursive function `fact4` to a tail recursive auxiliary function, we added the parameter `r` to `factIter`. This parameter is sometimes called an *accumulating parameter* (or just an *accumulator*).

We typically use an accumulating parameter to "accumulate" the result of the computation incrementally for return when the recursion terminates. In `factIter`, this "state" passed from one "iteration" to the next enables us to convert a backward recursive function to an "equivalent" tail recursive one.

Function `factIter` defines a more general function than `fact4`. It computes a factorial when we initialize the accumulator to 1, but it can compute some multiple of the factorial if we initialize the accumulator to another value. However, the application of `factIter` in `fact6` gives the initial value of 1 needed for factorial.

Consider auxiliary function `fibIter` used by function `fib2` below. This function adds two "accumulating parameters" to the backward nonlinear recursive function `fib` to convert the nonlinear (tree) recursion into a tail recursion. This technique works for Fibonacci numbers, but the same technique will not work in all cases.

```
fib2 :: Int -> Int
fib2 n | n >= 0 = fibIter n 0 1
    where
        fibIter 0 p q         = p
        fibIter m p q | m > 0 = fibIter (m-1) q (p+q)
```

Here we use type inference for `fibIter`. Function `fibIter` could be declared

```
fibIter :: Int -> Int -> Int -> Int
```

but it was not necessary because Haskell can infer the type from the types involved in its defining expressions.

What are the precondition and postcondition for `fibIter n p q`?

To avoid abnormal termination, `fibIter n p q` requires `n >= 0`. When the precondition holds, its postcondition is:

`fibIter n p q =` *Fibonacci*`(n) + (p + q - 1)`

If called with `p` and `q` set to 0 and 1, respectively, then `fibIter` returns:

Fibonacci`(n)`

How do we know that `fibIter n p q` terminates for $n \geq 0$?

The recursive leg of `fibIter n p q` is only evaluated when `n > 0`. On the recursive call, that argument decreases by 1. So eventually the computation reaches the base case.

What are the time and space complexities of `fibIter`?

Function `fibIter` has a time complexity of O(`n`) in contrast to O(`fib n`) for `fib`. This algorithmic speedup results from the replacement of the very expensive operation `fib(n-1) + fib(n-2)` at each level in `fib` by the inexpensive operation `p + q` (i.e., addition of two numbers) in `fibIter`.

Without tail call optimization, `fibIter n p q` has space complexity of O(`n`). However, tail call optimization (including an innermost reduction on the `q` argument) can convert the recursion to a loop, giving O(1) space complexity.

When combined with tail-call optimization and innermost reduction of strict arguments, a tail recursive function may be more efficient than the equivalent

backward recursive function. However, the backward recursive function is often easier to understand and, as we see in Chapter 25, to reason about.

9.4 Logarithmic Recursion

We can define the exponentiation operation `^` in terms of multiplication as follows for integers `b` and `n >= 0`:

`b^n` = $\prod_{i=1}^{i=n} b$

A backward recursive exponentiation function `expt`, shown below in Haskell, raises a number to a nonnegative integer power.

```
expt :: Integer -> Integer -> Integer
expt b 0        = 1
expt b n
    | n > 0     = b * expt b (n-1)  -- backward rec
    | otherwise = error (
                    "expt undefined for negative exponent "
                    ++ show n )
```

Here we use the unbounded integer type `Integer` for the parameters and return value.

Note that the recursive call of `expt` does not change the value of the parameter `b`.

Consider the following questions relative to `expt`.

- What are the precondition and postcondition for `expt b n`?
- How do we know that `expt b n` terminates?
- What are the time and space complexities of `expt b n` (ignoring any additional costs of processing the unbounded integer type)?

We can define a tail recursive auxiliary function `exptIter` by adding a new parameter to accumulate the value of the exponentiation incrementally. We can define `exptIter` within a function `expt2`, taking advantage of the fact that the base `b` does not change. This is shown below.

```
expt2 :: Integer -> Integer -> Integer
expt2 b n | n < 0 = error (
                     "expt2 undefined for negative exponent "
                     ++ show n )
expt2 b n         = exptIter n 1
    where exptIter 0 p = p
          exptIter m p = exptIter (m-1) (b*p) -- tail rec
```

Consider the following questions relative to `expt2`.

- What are the precondition and postcondition for `exptIter n p`?

- How do we know that `exptIter n p` terminates?
- What are the time and space complexities of `exptIter n p`?

The exponentiation function can be made computationally more efficient by squaring the intermediate values instead of iteratively multiplying. We observe that:

```
b^n = b^(n/2)^2   if n is even
b^n = b * b^(n-1) if n is odd
```

Function `expt3` below incorporates this observation into an improved algorithm. Its time complexity is O(`log2 n`) and space complexity is O(`log2 n`). (Here we assume that `log2` computes the logarithm base 2.)

```
expt3 :: Integer -> Integer -> Integer
expt3 _ n | n < 0 = error (
                      "expt3 undefined for negative exponent "
                      ++ show n )
expt3 b n         = exptAux n
    where exptAux 0       = 1
          exptAux n
              | even n    = let exp = exptAux (n `div` 2) in
                                exp * exp      -- backward rec
              | otherwise = b * exptAux (n-1)  -- backward rec
```

Here we are use two features of Haskell we have not used in the previous examples.

- Boolean function `even` returns `True` if and only if its integer argument is an even number. Similarly, `odd` returns `True` when its argument is an odd number.
- The `let` clause introduces `exp` as a local definition within the expression following `in` keyword, that is, within `exp * exp`.

  The `let` feature allows us to introduce new definitions in a bottom-up manner—first defining a symbol and then using it.

Consider the following questions relative to `expt3`.

- What are the precondition and postcondition `expt3 b n`?
- How do we know that `exptAux n` terminates?
- What are the time and space complexities of `exptAux n`?

9.5 Local Definitions

We have used two different language features to add local definitions to Haskell functions: `let` and `where`.

The `let` expression is useful whenever a nested set of definitions is required. It has the following syntax:

`let` *local_definitions* `in` *expression*

A `let` may be used anywhere that an expression my appear in a Haskell program.

For example, consider a function `f` that takes a list of integers and returns a list of their squares incremented by one:

```
f :: [Int] -> [Int]
f [] = []
f xs = let square a = a * a
           one      = 1
           (y:ys)   = xs
       in (square y + one) : f ys
```

- `square` represents a function of one variable.
- `one` represents a constant, that is, a function of zero variables.
- `(y:ys)` represents a pattern match binding against argument `xs` of `f`.
- Reference to `y` or `ys` when argument `xs` of `f` is nil results in an error.
- Local definitions `square`, `one`, `y`, and `ys` all come into scope simultaneously; their scope is the expression following the `in` keyword.
- Local definitions may access identifiers in outer scopes (e.g., `xs` in definition of `(y:ys)`) and have definitions nested within themselves.
- Local definitions may be recursive and call each other.

The `let` clause introduces symbols in a bottom-up manner: it introduces symbols before they are used.

The `where` clause is similar semantically, but it introduces symbols in a top-down manner: the symbols are used and then defined in a `where` that follows.

The `where` clause is more versatile than the `let`. It allows the scope of local definitions to span over several guarded equations while a `let`'s scope is restricted to the right-hand side of one equation.

For example, consider the definition:

```
g :: Int -> Int
g n | check3 == x = x
    | check3 == y = y
    | check3 == z = z * z
                  where check3 = n `mod` 3
                        x      = 0
                        y      = 1
                        z      = 2
```

- The scope of this `where` clause is over *all three guards* and their respective right-hand sides. (Note that the `where` begins in the same column as the `=` rather than to the right as in `rev'`.)

- Note the use of the modulo function `mod` as an infix operator. The backquotes (‘) around a function name denotes the infix use of the function.

In addition to making definitions easier to understand, local definitions can increase execution efficiency in some cases. A local definition may introduce a component into the expression graph that is shared among multiple branches. Haskell uses graph reduction, so any shared component is evaluated once and then replaced by its value for subsequent accesses.

The local variable `check3` introduces a component shared among all three legs. It is evaluated once for each call of `g`.

9.6 Using Other Languages

In this chapter, we have expressed the functions in Haskell, but they are adapted from the classic textbook *Structure and Interpretation of Computer Programs* (SICP) [1], which uses Scheme.

To compare languages, let's examine the `expt3` function in Scheme and other languages.

9.6.1 Scheme

Below is the Scheme language program for exponentiation similar to to `expt3` (called `fast-expt` in SICP [1]). Scheme, a dialect of Lisp, is an impure, eagerly evaluated functional language with dynamic typing.

```
(define (expt3 b n)
    (cond
        ((< n 0) (error `expt3 "Called with negative exponent"))
        (else (expt_aux b n))))

(define (expt_aux b n)
    (cond
    ((= n 0) 1)
    ((even? n) (square (expt3 b (/ n 2))))
    (else (* b (expt3 b (- n 1))))))

(define (square x) (* x x))

(define (even? n) (= (remainder n 2) 0))
```

Scheme (and Lisp) represents both data and programs as s-expressions (nested list structures) enclosed in balanced parentheses; that is, Scheme is *homoiconic*. In the case of executable expressions, the first element of the list may be operator. For example, consider:

```
(define (square x) (* x x))
```

The `define` operator takes two arguments:

- a symbol being defined, in this case a function signature `(square x)` for a function named `square` with one formal parameter named `x`
- an expression defining the value of the symbol, in this case the expression `(* x x)` that multiplies formal parameter `x` by itself and returns the result

The `define` operator has the side effect of adding the definition of the symbol to the environment. That is, `square` is introduced as a one argument function with the value denoted by the expression `(* x x)`.

The conditional expression `cond` gives an if-then-elseif expression that evaluates a sequence of predicates until one evaluates to "true" value and then returns the paired expression. The `else` at the end always evaluates to "true".

The above Scheme code defines the functions `square`, the exponentiation function `expt3`, and the logical predicate `even? {.scheme}`. It uses the primitive Scheme functions `-`, `*`, `/`, `remainder`, and `=` (equality).

We can evaluate the Scheme expression `(expt 2 10)` using a Scheme interpreter (as I did using DrRacket [71,72,140]) and get the value `1024`.

Although Haskell and Scheme are different in many ways—algebraic versus s-expression syntax, static versus dynamic typing, lazy versus eager evaluation (by default), always pure versus sometimes impure functions, etc.—the fundamental techniques we have examined in Haskell still apply to Scheme and other languages. We can use a substitution model, consider preconditions and termination, use tail recursion, and take advantage of first-class and higher-order functions.

Of course, each language offers a unique combination of features that can be exploited in our programs. For example, Scheme programmers can leverage its runtime flexibility and powerful macro system; Haskell programmers can build on its safe type system, algebraic data types, pattern matching, and other features.

The Racket Scheme [140] code for this subsection is in file `expt3.rkt`.

Let's now consider other languages.

9.6.2 Elixir

The language Elixir [68,168] is a relatively new language that executes on the Erlang platform (called the Erlang Virtual Machine or BEAM). Elixir is an eagerly evaluated functional language with strong support for *message-passing concurrent programming*. It is dynamically typed and is mostly pure except for input/output. It has pattern-matching features similar to Haskell.

We can render the `expt3` program into a sequential Elixir program as follows.

```
def expt3(b,n) when is_number(b) and is_integer(n)
                    and n >= 0 do
    exptAux(b,n)
```

```
end

defp exptAux(_,0) do 1 end

defp exptAux(b,n) do
    if rem(n,2) == 0  do    #  i.e. even
        exp = exptAux(b,div(n,2))
        exp * exp           #  backward rec
    else                    #  i.e. odd
        b * exptAux(b,n-1)  #  backward rec
    end
end
```

This code occurs within an Elixir module. The `def` statement defines a function that is exported from the module while `defp` defines a function that is private to the module (i.e., not exported).

A definition allows the addition of guard clauses following `when` (although they cannot include user-defined function calls because of restrictions of the Erlang VM). In function `expt3`, we use guards to do some type checking in this dynamically typed language and to ensure that the exponent is nonnegative.

Private function `exptAux` has two functions bodies. As in Haskell, the body is selected using pattern matching proceeding from top to bottom in the module. The first function body with the header `exptAux(_,0)` matches all cases in which the second argument is `0`. All other situations match the second header `exptAux(b,n)` binding parameters `b` and `n` to the argument values.

The functions `div` and `rem` denote integer division and remainder, respectively.

The Elixir = operator is not an assignment as in imperative languages. It is a pattern-match statement with an effect similar to `let` in the Haskell function.

Above the expression

```
exp = exptAux(b,div(n,2))
```

evaluates the recursive call and then binds the result to new local variable named `exp`. This value is used in the next statement to compute the return value `exp * exp`.

Again, although there are significant differences between Haskell and Elixir, the basic thinking and programming styles learned for Haskell are also useful in Elixir (or Erlang). These styles are also key to use of their concurrent programming features.

The Elixir [68] code for this subsection is in file `expt.ex`.

9.6.3 Scala

The language Scala [132,151] is a hybrid functional/object-oriented language that executes on the Java platform (i.e., on the Java Virtual Machine or JVM). Scala is an eagerly evaluated language. It allows functions to be written in a mostly pure manner, but it allows intermixing of functional, imperative, and object-oriented features. It has a relatively complex static type system similar to Java, but it supports type inference (although weaker than that of Haskell). It interoperates with Java and other languages on the JVM.

We can render the exponentiation function `expt3` into a functional Scala program as shown below. This uses the Java/Scala extended integer type `BigInt` for the base and return values.

```
def expt3(b: BigInt, n: Int): BigInt = {

    def exptAux(n1: Int): BigInt = // b known from outer
        n1 match {
          case 0                 => 1
          case m if (m % 2 == 0) =>  // i.e. even
            val exp = exptAux(m/2)
            exp * exp                // backward rec
          case m                 =>  // i.e. odd
            b * exptAux(m-1)         // backward rec
        }

    if (n >= 0)
        exptAux(n)
    else
        sys.error ("Cannot raise to negative power " + n )
}
```

The body of function `expt3` uses an **`if-else`** expression to ensure that the exponent is non-negative and then calls `exptAux` to do the work.

Function `expt3` encloses auxiliary function `exptAux`. For the latter, the parameters of `expt3` are in scope. For example, `exptAux` uses `b` from `expt3` as a constant.

Scala supports pattern matching using an explicit **`match`** operator in the form:

> *selector* **`match`** { *alternatives* }

It evaluates the *selector* expression and then choses the first *alternative* pattern that matches this value, proceedings top to botton, left to right. We write the alternative as

> **`case`** *pattern* => *expression*

or with a guard as:

```
case pattern if boolean_expression => expression
```

The *expression* may be a sequence of expressions. The value returned is the value of the last expression evaluated.

In this example, the `match` in `exptAux` could easily be replaced by an `if–else if–else` expression because it does not depend upon complex pattern matching.

In Haskell, functions are automatically curried. In Scala, we could alternatively define `expt3` in curried form using two argument lists as follows:

```
def expt3(b: BigInt)(n: Int): BigInt = ...
```

Again, we can use most of the functional programming methods we learn for Haskell in Scala. Scala has a few advantages over Haskell such as the ability to program in a multiparadigm style and interoperate with Java. However, Scala tends to be more complex and verbose than Haskell. Some features such as type inference and tail recursion are limited by Scala's need to operate on the JVM.

The Scala [151] code for this subsection is in file `exptBigInt2.scala`.

9.6.4 Lua

Lua [104,116] is a minimalistic, dynamically typed, imperative language designed to be *embedded as a scripting language* within other programs, such as computer games. It interoperates well with standard C and C++ programs.

We can render the exponentiation function `expt3` into a functional Lua program as shown below.

```
local function expt3(b,n)

    local function expt_aux(n)       -- b known from outer
        if n == 0 then
            return 1
        elseif n % 2 == 0 then       -- i.e. even
            local exp = expt_aux(n/2)
            return exp * exp         -- backward recursion
        else                         -- i.e. odd
            return b * expt_aux(n-1) -- backward recursion
        end
    end

    if type(b) == "number" and type(n) == "number" and n >= 0
                           and n == math.floor(n) then
        return expt_aux(n,1)
    else
        error("Invalid arguments to expt: " ..
              tostring(b) .. "^" .. tostring(n))
```

```
        end
    end
```

Like the Scala version, we define the auxiliary function `expt_aux` inside of function `expt3`, limiting its scope to the outer function.

This function uses with Lua version 5.2. In this and earlier versions, the only numbers are IEEE standard floating point. As in the Elixir version, we make sure the arguments are numbers with the exponent argument being nonnegative. Given that the numbers are floating point, the function also ensures that the exponent is an integer.

Auxiliary function `expt_aux` does the computational work. It differentiates among the three cases using an **`if−elseif−else`** structure. Lua does not have a switch statement or pattern matching capability.

Lua is not normally considered a functional language, but it has a number of features that support functional programming—in particular, first-class and higher order functions and tail call optimization.

In many ways, Lua is semantically similar to Scheme, but instead of having the Lisp-like hierarchical list as its central data structure, Lua provides an efficient, mutable, associative data structure called a table (somewhat like a hash table or map in other languages). Lua does not support Scheme-style macros in the standard language.

Unlike Haskell, Elixir, and Scala, Lua does not have builtin immutable data structures or pattern matching. Lua programs tend to be relatively verbose. So some of the usual programming idioms from functional languages do not fit Lua well.

The Lua [116] code for this subsection is in file `expt.lua`.

9.6.5 Elm

Elm [60,70] is a new functional language intended primarily for *client-side Web programming*. It is currently compiled into JavaScript, so some aspects are limited by the target execution environment. For example, Elm's basic types are those of JavaScript. So integers are actually implemented as floating point numbers.

Elm has a syntax and semantics that is similar to, but simpler than, Haskell. It has a Haskell-like **`let`** construct for local definitions but not a **`where`** construct. It also limits pattern matching to structured types.

Below is an Elm implementation of an exponentiation function similar to the Haskell `expt3` function, except it is limited to the standard integers `Int`. Operator // denotes the integer division operation and % is remainder operator.

```
    expt3 : Int -> Int -> Int
    expt3 b n =
```

```
let
    exptAux m =
        if m == 0 then
            1
        else if m % 2 == 0 then
            let
                exp = exptAux (m // 2)
            in
                exp * exp       -- backward rec
        else
            b  * exptAux (m-1)  -- backward rec
in
    if n < 0 then
        0 -- error?
    else
        exptAux n
```

One semantic difference between Elm and Haskell is that Elm functions must be total—that is, return a result for every possible input. Thus, this simple function extends the definition of `expt3` to return 0 for a negative power. An alternative would be to have `expt3` return a `Maybe Int` type instead of `Int`. We will examine this feature in Haskell later.

The Elm [60] code for this subsection is in file `expt.elm`.

9.7 What Next?

As we have seen in this chapter, we can develop efficient programs using functional programming and the Haskell language. These may require use to think about problems and programming a bit differently than we might in an imperative or object-oriented language. However, the techniques we learn for Haskell are usually applicable whenever we use the functional paradigm in any language. The functional way of thinking can also improve our programming in more traditional imperative and object-oriented languages.

In Chapter 10, we examine simple input/output concepts in Haskell. In Chapters 11 and 12, we examine software testing concepts.

In subsequent chapters, we explore the list data structure and additional programming techniques.

9.8 Chapter Source Code

The Haskell modules for the functions in this chapter are defined in the following source files:

- the factorial functions in `Factorial.hs` (from Chapter 4)

- the other Haskell functions in `RecursionStyles.hs` (with a simple test script in file `TestRecursionStyles.hs`){type="text/plain"}).

9.9 Exercises

1. Show the reduction of the expression `fib 4` substitution model. (This is repeated from the previous chapter.)

2. Show the reduction of the expression `expt 4 3` using the substitution model.

3. Answer the questions (precondition, postcondition, termination, time complexity, space complexity) in the subsection about `expt`.

4. Answer the questions in the subsection about `expt`.

5. Answer the questions in the subsection about `expt2`.

6. Answer the questions in the subsection about `expt3`.

7. Develop a recursive function in Java, C#, Python 3, JavaScript, or C++ that has the same functionality as `expt3`.

8. Develop an iterative, imperative program in Java, C#, Python 3, JavaScript, or C++ that has the same functionality as `expt3`.

For each of the following exercises, develop a Haskell program. For each function, informally argue that it terminates and give Big-O time and space complexities. Also identify any preconditions necessary to guarantee correct operation. Take care that special cases and error conditions are handled in a reasonable way.

7. Develop a backward recursive function `sumTo` such that `sumTo n` computes the sum of the integers from 1 to `n` for `n >= 0`.

8. Develop a tail recursive function `sumTo'` such that `sumTo' n` computes the sum of the integers from 1 to `n` for `n >= 0`.

9. Develop a backward recursive function `sumFromTo` such that `sumFromTo m n` computes the sum of the integers from `m` to `n` for `m <= n`.

10. Develop a tail recursive function `sumFromTo'` such that `sumFromTo' m n` computes the sum of the integers from `m` to `n` for `m <= n`.

11. Suppose we have functions `succ` (successor) and `pred` (predecessor) defined as follows:

    ```
    succ, pred :: Int -> Int
    succ n = n + 1
    pred n = n - 1
    ```

    Develop a function `add` such that `add m n` computes `m + n`. Function `add` cannot use the integer addition or subtraction operations but can use the `succ` ad `pred` functions above.

12. Develop a function `acker` to compute Ackermann's function, which is function A defined in Table 9.1.

Table 9.1: Ackermann's function.

$A(m,n)$	$=$	$n+1,$	if $m=0$
$A(m,n)$	$=$	$A(m-1,1),$	if $m>0$ and $n=0$
$A(m,n)$	$=$	$A(m-1,A(m,m-1)),$	if $m>0$ and $n>0$

13. Develop a function `hailstone` to implement the function shown in Table 9.2.

Table 9.2: Hailstone function.

$hailstone(n)$	$=$	$1,$	if $n=1$
$hailstone(n)$	$=$	$hailstone(n/2),$	if $n>1$, even n
$hailstone(n)$	$=$	$hailstone(3*n+1),$	if $n>1$, odd n

Note that an application of the `hailstone` function to the argument `3` would result in the following "sequence" of "calls" and would ultimately return the result `1`.

```
hailstone 3
  hailstone 10
    hailstone 5
      hailstone 16
        hailstone 8
          hailstone 4
            hailstone 2
              hailstone 1
```

For further thought: What is the domain of the *hailstone* function?

14. Develop the exponentiation function `expt4` that is similar to `expt3` but is tail recursive.

15. Develop the following group of functions.

    - `test` such that `test a b c` is `True` if and only if `a <= b` and no integer is the range from `a` to `b` inclusive is divisible by `c`.
    - `prime` such that `prime n` is `True` if and only if `n` is a prime integer.
    - `nextPrime` such that `nextPrime n` returns the next prime integer greater than `n`

16. Develop function `binom` to compute *binomial coefficients*. That is, `binom n k` returns $\binom{n}{k}$ for integers `n >= 0` and `0 <= k <= n`.

9.10 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from previous work:

- the 2016 Scala version of my notes on *Recursion Syles, Correctness, and Efficiency* [48] (for which previous versions existed in Scala, Elixir, and Lua)
- the Haskell factorial, Fibonacci number, and exponentiation functions from my previous examples in Haskell, Elm, Scala, Elixir, and Lua, which, in turn, were adapted from the Scheme programs in Abelson and Sussman's classic, Scheme-based textbook *SICP* [1]

In 2017, I continued to develop this work as Chapter 3, Evaluation and Efficiency, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Evaluation and Efficiency chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 3.1-3.2 became the basis for new Chapter 8, Evaluation Model, and previous sections 3.3-3.5 became the basis for Chapter 9 (this chapter), Recursion Styles and Efficiency. I also moved some of the discussion of preconditions and postconditions from old chapter 3 to the new chapter 6 and discussion of local definitions from old chapter 4 to new chapter 9 (this chapter).

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

9.11 Terms and Concepts

Recursion styles (linear vs. nonlinear, backward vs. forward, tail, and logarithmic), correctness (precondition, postcondition, and termination), efficiency estimation (time and space complexity), transformations to improve efficiency (auxiliary function, accumulator), homiconic, message-passing concurrent programming, embedded as a scripting language, client-side Web programming.

10 Simple Input and Output (FUTURE)

10.1 Chapter Introduction

This is a stub for a possible future chapter. The Haskell Wikibook [179] Simple input and output page discusses the concepts sufficient for the purposes of this point in the textbook.

10.2 What Next?

TODO

10.3 Exercises

TODO

10.4 Acknowledgements

TODO

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

10.5 Terms and Concepts

TODO

11 Software Testing Concepts

11.1 Chapter Introduction

The goal of this chapter (11) is to survey the important concepts, terminology, and techniques of software testing in general.

Chapter 12 illustrates these techniques by manually constructing test scripts for Haskell functions and modules.

11.2 Software Requirements Specification

The purpose of a software development project is to meet particular needs and expectations of the project's stakeholders.

By *stakeholder*, we mean any person or organization with some interest in the project's outcome. Stakeholders include entities that:

- have a "business" problem needing a solution—the project's sponsors, customers, and users
- care about the broad impacts of the project and its solution—that laws, regulations, standards, best practices, codes of conduct, etc., are followed
- are responsible for the development, deployment, operation, support, and maintenance of the software

A project's stakeholders should create a software requirements specification to state the particular needs and expectations to be addressed.

A *software requirements specification* seeks to comprehensively describe the intended behaviors and environment of the software to be developed. It should address the "what" but not the "how". For example, the software requirements specification should describe the desired mapping from inputs to outputs but not unnecessarily restrict the software architecture, software design, algorithms, data structures, programming languages, and software libraries that can be used to implement the mapping.

Once the requirements are sufficiently understood, the project's developers then design and implement the software system: its software architecture, its subsystems, its modules, and its functions and procedures.

Software testing helps ensure that the software implementation satisfies the design and that the design satisfies the stakeholder's requirements.

Of course, the requirements analysis, design, and implementation may be an incremental. Software testing can also play a role in identifying requirements and defining appropriate designs and implementations.

11.3 What is Software Testing?

According to the Collins English Dictionary [35]:

> A *test* is a deliberate action or experiment to find out how well something works.

The purpose of testing a program is to determine "how well" the program "works"—to what extent the program satisfies its software requirements specification.

Software testing is a "deliberate" process. The tests must be chosen effectively and conducted systematically. In most cases, the *test plan* should be documented carefully so that the tests can be repeated precisely. The results of the tests should be examined rigorously.

In general, the tests should be automated. Testers can use manually written test scripts (as we do in the Chapter 12) or appropriate testing frameworks [126] (e.g., JUnit [166,174] in Java, Pytest [113,133] in Python, and HUnit [90], QuickCheck [89], or Tasty [91] in Haskell).

Testers try to uncover as many defects as possible, but it is impossible to identify and correct all defects by testing. Testing is just one aspect of software quality assurance.

11.4 Goals of Testing

Meszaros [126, Ch. 3] identifies several goals of test automation. These apply more generally to all software testing. Tests should:

- help improve software quality
- help software developers understand the system being tested
- reduce risk
- be easy to develop
- be easy to conduct repeatedly
- be easy to maintain as the system being tested continues to evolve

11.5 Dimensions of Testing

We can organize software testing along three dimensions [161]:

- testing levels
- testing methods
- testing types

We explore these in the following subsections.

11.5.1 Testing levels

Software *testing levels* categorize tests by the applicable stages of software development.

Note: The use of the term "stages" does not mean that this approach is only applicable to the traditional *waterfall* software development process. These stages describe general analysis and design activities that must be carried out however the process is organized and documented.

Ammann and Offutt [2] identify five levels of testing, as shown in Figure 11.1. Each level assumes that the relevant aspects of the level below have been completed successfully.

From the highest to the lowest, the testing levels are as follows.

1. *Acceptance testing* focuses on testing a completed system to determine whether it satisfies the software requirements specification and to assess whether the system is acceptable for delivery.

   The acceptance test team must include individuals who are strongly familiar with the *business requirements* of the stakeholders.

2. *System testing* focuses on testing an integrated system to determine whether it satisfies its overall specification (i.e., the requirements as reflected in the chosen software architecture).

   The system test team is usually separate from the development team.

3. *Integration testing* focuses on testing each subsystem to determine whether its constituent modules communicate as required. For example, do the modules have consistent interfaces (e.g., compatible assumptions and contracts)?

   A subsystem is often constructed by using existing libraries, adapting previously existing modules, and combining these with a few new modules. It is easy to miss subtle incompatibilities among the modules. Integration testing seeks to find any incompatibilities among the various modules.

   Integration testing is often conducted by the development team.

4. *Module testing* focuses on the structure and behavior of each module separately from the other modules with which it communicates.

   A module is usually composed of several *related units* and their associated data types and data structures. Module testing assesses whether the units and other features interact as required and assess whether the module satisfies its specification (e.g., its preconditions, postconditions, and invariants).

   Note: Here we use the term "module" generically. For example, a module in Java might be a `class`, `package`, or `module` (in Java 9) construct. A

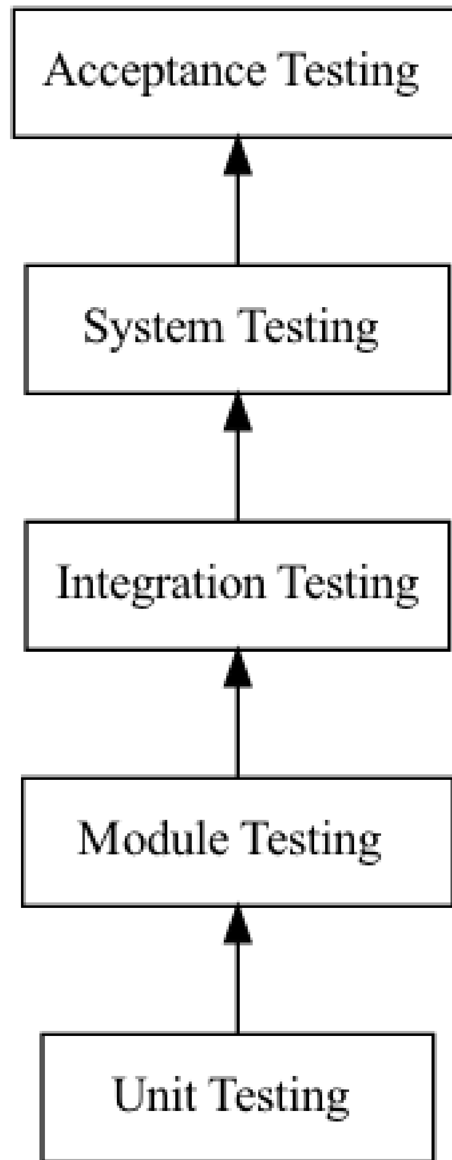


Figure 11.1: Software testing levels.

module in Python 3 might be a code file (i.e., module) or a directory structure of code files (i.e., package). In Haskell, a generic module might be represented as a closely related group of Haskell `module` files.

Module testing is typically done by the developer(s) of a module.

5. *Unit testing* focuses on testing the implementation of each program unit to determine whether it performs according to the unit's specification.

   The term "unit" typically refers to a procedural abstraction such as a function, procedure, subroutine, or method.

   A unit's specification is its "contract", whether represented in terms of preconditions and postconditions or more informally.

   Unit testing is typically the responsibility of the developer(s) of the unit.

In object-based systems, the units (e.g., methods) and the modules (e.g., objects or classes) are often tightly coupled. In this and similar situations, developers often combine unit testing and module testing into one stage called *unit testing* [2,161].

In this book, we are primarily concerned with the levels usually conducted by the developers: unit, module, and integration testing.

11.5.2 Testing methods

Software *testing methods* categorize tests by how they are conducted. The Software Testing Fundamentals website [161] identifies several methods for testing. Here we consider four:

- black-box testing
- white-box testing
- gray-box testing
- ad hoc testing

In this book, we are primarily concerned with black-box and gray-box testing. Our tests are guided by the contracts and other specifications for the unit, module, or subsystem being tested.

11.5.2.1 Black-box testing In *black-box testing*, the tester knows the external requirements specification (e.g., the contract) for the item being tested but does not know the item's internal structure, design, or implementation.

Note: This method is sometimes called closed-box or behavioral testing.

This method approaches the system much as a user does, as a black box whose internal details are hidden from view. Using only the requirements specification and public features of the item, the testers devise tests that check input values to make sure the system yields the expected result for each. They use the item's regular interface to carry out the tests.

Black-box tests are applicable to testing at the higher levels—integration, systems, and acceptance testing—and for use by external test teams.

The method is also useful for unit and module testing, particularly when we wish to test the item against an abstract interface with an explicit specification (e.g., a contract stated as preconditions, postconditions, and invariants).

How do we design black-box tests? Let's consider the possible inputs to the item.

Input domain An item being tested has some number of input parameters—some explicit, others implicit. Each parameter has some domain of possible values.

The *input domain* of the item thus consists of the Cartesian product of the individual domains of its parameters. A *test input* is thus a tuple of values, one possible value for each parameter [2].

For example, consider testing a public instance method in a Java class. The method has zero or more explicit parameters, one implicit parameter (giving the method access to all the associated instance's variables), and perhaps direct access to variables outside its associated instance (static variables, other instances' variables, public variables in other classes, etc.).

In most practical situations, it is impossible to check all possible test inputs. Thus, testers need to choose a relatively small, finite set of input values to test. But how?

Choosing test inputs In choosing test inputs, the testers can fruitfully apply the following techniques [69,126,139].

- Define *equivalence classes* (or partitions) of the possible inputs based on the kinds of behaviors of interest and then choose *representative* members of each class.

  After studying the requirements specification for the item being tested, the tester first groups together inputs that result in the "same" behaviors of interest and then chooses typical representatives of each group for tests (e.g., from the middle of the group).

  The representative values are *normal use* or "happy path" cases that are not usually problematic to implement [2].

  For example, consider the valid integer values for the day of a month (on the Gregorian calendar as used in the USA). It may be useful to consider the months falling into three equivalence classes: 31-day months, 30-day months, and February.

- Choose *boundary values*—values just inside and just outside the edges of an equivalence class (as defined above) or special values that require unusual

handling.

Unlike the "happy path" tests, the boundary values often are values that cause problems [2].

For example, consider the size of a data structure being constructed. The boundary values of interest may be zero, one, minimum allowed, maximum allowed, or just beyond the minimum or maximum.

For a mathematical function, a boundary value may also be at or near a value for which the function is undefined or might result in a nonterminating computation.

- Choose input values that *cover the range of expected results*.

  This technique works from the output back toward the input to help ensure that important paths through the item are handled properly.

  For example, consider transactions on a bank account. The action might be a balance inquiry, which returns information but does not change the balance in the account. The action might be a deposit, which results in a credit to the account. The action might be a withdrawal, which either results in a debit or triggers an insufficient funds action. Tests should cover all four cases.

- Choose input values *based on the model* used to specify the item (e.g., state machine, mathematical properties, invariants) to make sure the item implements the model appropriately.

  For example, a data abstraction should establish and preserve the invariants of the abstraction (as shown in the Rational arithmetic case study in Chapter 7).

Black-box testers often must give attention to tricky practical issues such as appropriate *error handling* and *data-type conversions*.

11.5.2.2 White-box testing In *white-box testing*, the tester knows the internal structure, design, and implementation of the item being tested as well as the external requirements specification.

Note: This method is sometimes called open-box, clear-box transparent-box, glass box, code-based, or structural testing.

This method seeks to test every path through the code to make sure every input yields the expected result. It may use code analysis tools [2] to define the tests or special instrumentation of the item (e.g., a testing interface) to carry out the tests.

White-box testing is most applicable to unit and module testing (e.g., for use by the developers of the unit), but it can also be used for integration and system testing.

11.5.2.3 Gray-box testing

In *gray-box testing*, the tester has *partial* knowledge of the internal structure, design, and requirements of the item being tested as well as the external requirements specification.

Note: "Gray" is the typical American English spelling. International or British English spells the word "grey".

Gray-box testing combines aspects of black-box and white-box testing. As in white-box testing, the tester can use knowlege of the internal details (e.g., algorithms, data structures, or programming language features) of the item being tested to design the test cases. But, as in black-box testing, the tester conducts the tests through the item's regular interface.

This method is primarily used for integration testing, but it can be used for the other levels as well.

11.5.2.4 Ad hoc testing

In *ad hoc testing*, the tester does not plan the details of the tests in advance as is typically done for the other methods. The testing is done informally and randomly, improvised according the creativity and experience of the tester. The tester strives to "break" the system, perhaps in unexpected ways.

This method is primarily used at the acceptance test level. It may be carried out by someone from outside the software development organization on behalf of the client of a software project.

11.5.3 Testing types

Software *testing types* categorize tests by the purposes for which they are conducted. The Software Testing Fundamentals website [161] identifies several types of testing:

- *Smoke testing* seeks to ensure that the primary functions work. It uses of a non-exhaustive set of tests to "smoke out" any major problems.
- *Functional testing* seeks to ensure that the system satisfies all its functional requirements. (That is, does a given input yield the correct result?)
- *Usability testing* seeks to ensure that the system is easily usable from the perspective of an end-user.
- *Security testing* seeks to ensure that the system's data and resources are protected from possible intruders by revealing any vulnerabilities in the system
- *Performance testing* seeks to ensure that the system meets its performance requirements under certain loads.
- *Regression testing* seeks to ensure that software changes (bug fixes or enhancements) do not break other functionality.

- *Compliance testing* seeks to ensure the system complies to required internal or external standards.

In this book, we are primarily interested in functional testing.

11.5.4 Combining levels, methods, and types

A tester can conduct some type of testing during some stage of software development using some method. For example,

- a test team might conduct *functional testing* (a type) at the *system testing* level using the *black-box testing* method to determine whether the system performs correctly
- a programmer might do *smoke testing* (a type) of the code at the *module testing* level using the *white-box testing* method to find and resolve major shortcomings before proceeding with more complete functional testing

As noted above, in this book we are primarily interested in applying *functional testing* (type) techniques at the *unit, module, or integration testing* levels using *black-box or gray-box testing* methods. We are also interested in automating our tests.

11.6 Aside: Test-Driven Development

The traditional software development process follows a *design-code-test* cycle. The developers create a design to satisfy the requirements, then implement the design as code in a programming language, and then test the code.

Test-driven development (TDD) reverses the traditional cycle; it follows a *test-code-design* cycle instead. It uses a test case to drive the writing of code that satisfies the test. The new code drives the restructuring (i.e., refactoring) of the code base to evolve a good design. The goal is for the design and code to grow organically from the tests [10,112].

Beck describes the following "algorithm" for TDD [10].

1. *Add a test* for a small, unmet requirement.

   If there are no unmet requirements, stop. The program is complete.

2. *Run all the tests.*

   If no tests fail, go to step 1.

3. *Write code to make a failing test succeed.*

4. *Run all the tests.*

   If any test fails, go to step 3.

5. *Refactor the code* to create a "clean" design.

6. *Run all the tests.*

   If any test fails, go to step 3.

7. Go to step 1 to start a new cycle.

Refactoring [77] (step 5) is critical for evolving good designs and good code. It involves removing duplication, moving code to provide a more logical structure, splitting apart existing abstractions (e.g., functions, modules, and data types), creating appropriate new procedural and data abstractions, generalizing constants to variables or functions, and other code transformations.

TDD focuses on functional-type unit and module testing using black-box and gray-box methods. The tests are defined and conducted by the developers, so the tests may not cover the full functional requirements needed at the higher levels. The tests often favor "happy path" tests over possible error cases [2].

This book presents programming language concepts using mostly small programs consisting of a few functions and modules. The book does not use TDD techniques directly, but it promotes similar rigor in analyzing requirements. As we have seen in previous chapters, this book focuses on design using contracts (i.e., preconditions, postconditions, and invariants), information-hiding modules, pure functions, and other features we study in later chapters.

As illustrated in Chapter 12, these methods are also compatible with functional-type unit and module testing using black-box and gray-box methods.

11.7 Principles for Test Automation

Based on earlier work on the Test Automation Manifesto [127], Meszaros proposes several principles for test automation [126, Ch. 5]. These focus primarily on unit and module testing. The principles include the following:

1. *Write the tests first.*

   This principle suggests that developers should use Test-Driven Development (TDD) [10] as described in Section 11.6.

2. *Design for testability.*

   Developers should consider how to test an item while the item is being designed and implemented. This is natural when TDD is being used, but, even if TDD is not used, testability should be an important consideration during design and implementation. If code cannot be tested reliably, it is usually bad code.

   The application of this principle requires judicious use of the abstraction techniques, such as those illustrated in Chapters 6 and 7 and in later chapters.

3. *Use the front door first.*

Testing should be done primarily through the standard public interface of the item being tested. A test involves invoking a standard operation and then verifying that the operation has the desired result. (In terms of the testing methods described in Section 11.5.2, this principle implies use of black-box and gray-box methods.)

Special testing interfaces and operations may sometimes be necessary, but they can introduce new problems. They make the item being tested more complex and costly to maintain. They promote unintentional (or perhaps intentional) overspecification of the item. This can limit future changes to the item—or at least it makes future changes more difficult.

Note: *Overspecification* means imposing requirements on the software that are not explicitly needed to meet the users' actual requirements. For example, a particular order may be imposed on a sequence of data or activities when an arbitrary order may be sufficient to meet the actual requirements.

4. *Communicate intent.*

   As with any other program, a test program should be designed, implemented, tested, and documented carefully.

   However, test code is often more difficult to understand than other code because the reader must understand both the test program and the item being tested. The "big picture" meaning is often obscured by the mass of details.

   Testers should ensure they communicate the intent of a set of tests. They should use a naming scheme that reveals the intent and include appropriate comments. They should use standard utility procedures from the testing framework or develop their own utilities to abstract out common activities and data.

5. *Don't modify the system under test.*

   Testers should avoid modifying a "completed" item to enable testing. This can break existing functionality and introduce new flaws. Also, if the tests are not conducted on the item to be deployed, then the results of the tests may be inaccurate or misleading.

   As noted in principles above, it is better to "design for testability" from the beginning so that tests can be conducted through "the front door" if possible.

6. *Keep tests independent.*

   A test should be designed and implemented to be run independently of all other tests of that unit or module. It should be possible to execute a set of tests in any order, perhaps even concurrently, and get the same results for all tests.

Thus each automated test should set up its needed precondition state, run the test, determine whether the test succeeds or fails, and ensure no artifacts created by the test affect any of the other tests.

If one item depends upon the correct operation of a second item, then it may be useful to test the second item fully before testing the first. This dependency should be documented or enforced by the testing program.

7. *Isolate the system under test.*

   Almost all programs depend on some programming language, its standard runtime library, and basic features of the underlying operating system. Most modern software depends on much more: the libraries, frameworks, database systems, hardware devices, and other software that it uses.

   As much as possible, developers and testers should isolate the system being tested from other items not being tested at that time. They should document the versions and configurations of all other items that the system under test depends on. They should ensure the testing environment controls (or at least records) what versions and configurations are used.

   As much as practical, the software developers should encapsulate critical external dependencies within information-hiding components. This approach helps the developers to provide stable behavior over time. If necessary, this also enables the testers to substitute a "test double" for a problematic system.

   A *test double* is a "test-specific equivalent" [126, Ch. 11, Ch. 23] that is substituted for some component upon which the system under test depends. It may replace a component that is necessary but which is not available for safe use in the testing environment. For example, testers might be testing on system that interacts with another that has not yet been developed.

8. *Minimize test overlap.*

   Tests need to cover as much functionality of a system as possible, but it may be counterproductive to test the same functionality more than once. If the code for that functionality is defective, it likely will cause all the overlapping tests to fail. Following up on the duplicate failures takes time and effort that can better be invested in other testing work.

9. *Minimize untestable code.*

   Some components cannot be tested fully using an automated test program. For example, code in graphical user interfaces (GUIs), in multithreaded programs, or in test programs themselves are embedded in contexts that may not support being called by other programs.

   However, developers can design the system so that as much as possible is moved to separate components that can be tested in an automated fashion.

For example, a GUI can perhaps be designed as a "thin" interactive layer that sets up calls to an application programming interface (API) to carry out most of the work. In addition to being easier to test, such an API may enable other kinds of interfaces in addition to the GUI.

10. *Keep test logic out of production code.*

    As suggested above, developers should "design for testability" through "the front door". Code should be tested in a configuration that is as close as possible to the production configuration.

    Developers and testers should avoid inserting special *test hooks* into the code (e.g., `if testing then doSomething`) that will not be active in the production code. In addition to distorting the tests, such hooks can introduce functional or security flaws into the production code and make the program larger, slower, and more difficult to understand.

11. *Verify one condition per test.*

    If one test covers multiple conditions, it may be nontrivial to determine the specific condition causing a failure. This is likely not a problem with a manual test carried out by a human; it may be an efficient use of time to do fewer, broader tests.

    However, tests covering multiple conditions are an unnecessary complication for inexpensive automated tests. Each automated test should be "independent" of others, do its own setup, and focus on a single likely cause of a failure. "

12. *Test concerns separately.*

    The behavior of a large system consists of many different, "small" behaviors. Sometimes a component of the system may implement several of the "small" behaviors. Instead of focusing a test on broad concerns of the entire system, testers should focus a test on a narrow concern. The failure of such a test can help pinpoint where the problem is.

    The key here is to "pull apart" the overall behaviors of the system to identify "small" behaviors that can be tested independently.

13. *Ensure commensurate effort and responsibility.*

    Developing test code that follows all of these principles can exceed the time it took to develop the system under test. Such an imbalance is bad. Testing should take approximately the same time as design and implementation.

    The developers may need to devote more time and effort to "designing for testability" so that testing becomes less burdensome.

    The testers may need to better use existing tools and frameworks to avoid too much special testing code. The testers should consider carefully which

tests can provide useful information and which do not. There is no need for a test if it does not help reduce risk.

11.8 What Next?

This chapter (11) surveyed software testing concepts. Chapter 12 applies them to testing Haskell modules from Chapters 4 and 7.

11.9 Exercises

TODO

11.10 Acknowledgements

I wrote this chapter in Summer 2018 for the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming*.

- The discussion of the dimensions of software testing — levels, methods, and types — draws on the discussion on the Software Testing Fundamentals website [161] and other sources [2,16,69,139].
- The presentation of the goals and principles of test automation draws on the ideas of Meszaros [126,127].
- The description of Test-Driven Development (TDD) "algorithm" is adapted from that of Beck [10] and Koskela [112].

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

11.11 Terms and Concepts

Stakeholder, software requirements specification, test, test plan, testing dimensions (levels, methods, types), testing levels (unit, module, integration, system, and acceptance testing), testing methods (black-box, white-box, gray-box, and ad hoc testing), input domain, test input, input domain equivalence classes, representatives of normal use or "happy path", boundary values, covering the range of expected outputs, testing based on the specification model, error handling, data-type conversions, testing types (smoke testing, functional testing, usability testing, security testing, performance testing, regression testing, compliance testing), test-driven development (TDD), design-code-test vs. test-code-design.

12 Testing Haskell Programs

12.1 Chapter Introduction

The goal of this chapter (12)is to illustrate the testing techniques by manually constructing test scripts for Haskell functions and modules. It builds on the concepts and techniques surveyed in Chapter 11.

We use two testing examples in this chapter:

- the group of factorial functions from Chapters 4 and 9

  The series of tests can be applied any of the functions.

- the rational arithmetic modules from Chapter 7

12.2 Organizing Tests

Testers commonly organize unit tests on a system using the *Arrange-Act-Assert pattern* [10,112].

1. *Arrange:* Select input values from the input domain and construct appropriate "objects" to use in testing the test subject.
2. *Act:* Apply some operation from the test subject to appropriate input "objects".
3. *Assert:* Determine whether or not the result satisfies the specification.

Each test should create the test-specific input "objects" it needs and remove those and any result "objects" that would interfere with other tests.

Note: In this chapter, we use the word "object" in a general sense of any data entity, not in the specific sense defined for object-based programming.

12.3 Testing Functions

In terms of the dimensions of testing described in Chapter 11, this section approaches testing of a group of Haskell functions as follows.

Testing level: unit testing of each Haskell function

Testing method: primarily black-box testing of each Haskell function relative to its specification

Testing type: functional testing of each Haskell function relative to its specification

12.3.1 Factorial example

As an example, consider the set of seven factorial functions developed in Chapters 4 and 9 (in source file `Factorial.hs`). All have the requirement to implement the mathematical function

$fact(n) = \prod_{i=1}^{i=n} i$

for any $n \geq 0$. The specification is ambiguous on what should be the result of calling the function with a negative argument.

12.3.2 Arrange

To carry out black-box testing, we must *arrange* our input values. The factorial function tests do not require any special testing "objects".

We first partition the input domain. We identify two equivalence classes of inputs for the factorial function:

1. the set of nonnegative integers for which the mathematical function is defined and the Haskell function returns that value within the positive `Int` range
2. the set of nonnegative integers for which the mathematical function is defined but the Haskell function returns a value that overflows the `Int` range

The class 2 values result are errors, but integer overflow is typically not detected by the hardware.

We also note that the negative integers are outside the range of the specification.

Next, we select the following values inside the "lower" boundary of class 1 above:

- 0, empty case at the lower boundary
- 1, smallest nonempty case at the lower boundary

Then we choose representative values within class 1:

- 2, one larger than the smallest nonempty case
- 5, arbitrary value representative of values away from the boundary

Note: The choice of two representative values might be considered a violation of the "minimize test overlap" principle from Chapter 11. So it could be acceptable to drop the input of 2. Of course, we could argue that we should check 2 as a possible boundary value.

We also select the value -1, which is just outside the lower boundary implied by the $n \geq 0$ requirement.

All of the factorial functions have the type signature (where `N` is `1`, `2`, `3`, `4`, `4'`, `5`, or `6`):

```
factN :: Int -> Int
```

Thus the `factN` functions also have an "upper" boundary that depends on the maximum value of the `Int` type on a particular machine. The author is testing these functions on a machine with 64-bit, two's complement integers. Thus the largest integer whose factorial is less than 2^{63} is 20.

We thus select input the following input values:

- 20, which is just inside the upper boundary of class 1
- 21, which is just outside class 1 and inside class 2

12.3.3 Act

We can test a factorial function at a chosen input value by simply applying the function to the value such as the following:

```
fact1 0
```

A Haskell function has no side effects, so we just need to examine the integer result returned by the function to determine whether it satisfies the function's specification.

12.3.4 Assert

We can test the result of a function by stating a Boolean expression—an assertion—that the value satisfies some property that we want to check.

In simple cases like the factorial function, we can just compare the actual result for equality with the expected result. If the comparison yields `True`, then the test subject "passes" the test.

```
fact1 0 == 1
```

12.3.5 Aggregating into test script

There are testing frameworks for Haskell (e.g., HUnit [90], QuickCheck [89], or Tasty [91]), but, in this section, we manually develop a simple test script.

We can state a Haskell `IO` program to print the test and whether or not it passes the test. (Simple input and output will eventually be discussed in a Chapter 10. For now, see the Haskell Wikibooks [179] page on "Simple input and output".)

Below is a Haskell `IO` script that tests class 1 boundary values 0 and 1 and "happy path" representative values 2 and 5.

```
pass :: Bool -> String
pass True  = "PASS"
pass False = "FAIL"

main :: IO ()
main = do
    putStrLn  "\nTesting fact1"
    putStrLn ("fact1 0 == 1:       " ++ pass (fact1 0 == 1))
    putStrLn ("fact1 1 == 1:       " ++ pass (fact1 1 == 1))
    putStrLn ("fact1 2 == 2:       " ++ pass (fact1 2 == 2))
    putStrLn ("fact1 5 == 120:     " ++ pass (fact1 5 == 120))
```

The `do` construct begins a sequence of `IO` commands. The `IO` command `putStrLn` outputs a string to the standard output followed by a newline character.

Testing a value below the lower boundary of class 1 is tricky. The specification does not require any particular behavior for -1. As we saw in Chapter 4, some of the function calls result in overflow of the runtime stack, some fail because all of the patterns fail, and some fail with an explicit `error` call. However, all these trigger a Haskell exception.

Our test script can `catch` these exceptions using the following code.

```
putStrLn ("fact1 (-1)  == 1:  "
          ++ pass (fact1 (-1)  == 1))
    `catch` (\(StackOverflow)
                -> putStrLn ("[Stack Overflow] (EXPECTED)"))
    `catch` (\(PatternMatchFail msg)
                -> putStrLn ("[Pattern Match Failure]\n...."
                             ++ msg))
    `catch` (\(ErrorCall msg)
                -> putStrLn ("[Error Call]\n...." ++ msg))
```

To catch the exceptions, the program needs to import the module `Control.Exception` from the Haskell library.

```
import Prelude hiding (catch)
import Control.Exception
```

By catching the exception, the test program prints an appropriate error message and then continues with the next test; otherwise the program would halt when the exception is thrown.

Testing an input value in class 2 (i.e., outside the boundary of class 1) is also tricky.

First, the values we need to test depend on the default integer (`Int`) size on the particular machine.

Second, because the actual value of the factorial is outside the `Int` range, we cannot express the test with Haskell `Int`s. Fortunately, by converting the values to the unbounded `Integer` type, the code can compare the result to the expected value.

The code below tests input values 20 and 21.

```
putStrLn ("fact1 20 == 2432902008176640000:   "
          ++ pass (toInteger (fact1 20) ==
                              2432902008176640000))
putStrLn ("fact1 21 == 51090942171709440000:  "
          ++ pass (toInteger (fact1 21) ==
                              51090942171709440000)
          ++ " (EXPECT FAIL for 64-bit Int)" )
```

The above is a black-box unit test. It is not specific to any one of the seven factorial functions defined in Chapters 4 and 9. (These are defined in the source file `Factorial.hs`.) The series of tests can be applied any of the functions.

The test script for the entire set of functions from Chapters 4 and 9 (and others) are in the source file `TestFactorial.hs`.

12.4 Testing Modules

In terms of the dimensions of testing described in Chapter 11, this section approaches testing of Haskell modules as follows.

Testing level: module-level testing of each Haskell module
Testing method: primarily black-box testing of each Haskell module relative to its specification
Testing type: functional testing of each Haskell module relative to its specification

Normally, module-level testing requires that unit-level testing be done for each function first. In cases where the functions within a module are strongly coupled, unit-level and module-level testing may be combined into one phase.

12.4.1 Rational arithmetic modules example

For this section, we use the rational arithmetic example from Chapter 7.

In the rational arithmetic example, we define two abstract (information-hiding) modules: `RationalRep` and `Rational`.

Given that the `Rational` module depends on the `RationalRep` module, we first consider testing the latter.

12.4.2 Data representation modules

Chapter 7 defines the abstract module `RationalRep` and presents two distinct implementations, `RationalCore` and `RationalDeferGCD`. The two implementations differ in how the rational numbers are represented using data type `Rat`. (See source files `RationalCore.hs` and `RationalDeferGCD.hs`.)

Consider the public function signatures of `RationalRep` (from Chapter 7):

```
makeRat :: Int -> Int -> Rat
numer   :: Rat -> Int
denom   :: Rat -> Int
zeroRat :: Rat
showRat :: Rat -> String
```

Because the results of `makeRat` and `zeroRat` and the inputs to `numer`, `denom`, and `showRat` are abstract, we cannot test them directly as we did the factorial functions Section 12.3. For example, we cannot just call `makeRat` with two

integers and compare the result to some specific concrete value. Similarly, we cannot test `numer` and `denom` directly by providing them some specific input value.

However, we can test both through the abstract interface, taking advantages of the interface invariant.

RationalRep Interface Invariant (from Chapter 7):

: For any valid Haskell rational number `r`, all the following hold:

```
-   `r`{.haskell} $\in$ `Rat`{.haskell}
-   `denom r > 0`{.haskell}
-   if `numer r == 0`{.haskell}, then `denom r == 1`{.haskell}
-   `numer r`{.haskell} and `denom r`{.haskell} are relatively prime
-   the (mathematical) rational number value is
    $\frac{\texttt{numer r}}{\texttt{denom r}}$
```

The invariant allows us to check combinations of the functions to see if they give the expected results. For example, suppose we define `x'` and `y'` as follows:

```haskell
x' = numer (makeRat x y)
y' = denom (makeRat x y)
```

Then the interface invariant and contracts for `makeRat`, `numer`, and `denom` allow us to infer that the (mathematical) rational number values $\frac{\texttt{x'}}{\texttt{y'}}$ and $\frac{\texttt{x}}{\texttt{y}}$ are equal.

This enables us to devise pairs of test assertions such as

```haskell
numer (makeRat 1 2) == 1
denom (makeRat 1 2) == 2
```

and

```haskell
numer (makeRat 4 (-2)) == -2
denom (makeRat 4 (-2)) == 1
```

to indirectly test the functions in terms of their interactions with each other. All the tests above should succeed if the module is designed and implemented according to its specification.

Similarly, we cannot directly test the private functions `signum'`, `abs'`, and `gcd'`. But we try to choose inputs the tests above to cover testing of these functions. (Private functions should be tested as the module is being developed to detect any more problems.)

12.4.2.1 Arrange To conduct black-box testing, we must arrange the input values we wish to test. The module tests do not require any special test objects, but each pair of tests both create a `Rat` object with `makeRat` and select its numerator and denominator with `numer` and `denom`.

However, for convenience, we can define the following shorter names for constants:

```
maxInt  = (maxBound :: Int)
minInt  = (minBound :: Int)
```

TODO: Draw a diagram as discussed

Each pair of tests has two `Int` parameters—the `x` and `y` parameters of `makeRat`. Thus we can visualize the input domain as the integer grid points on an `x-y` coordinate plane using the usual rectangular layout from high school algebra.

We note that any input `x-y` value along the `x`-axis does not correspond to a rational number; the pair of integer values does not satisfy the precondition for `makeRat` and thus result in an `error` exception.

For the purposes of our tests, we divide the rest of the plane into the following additional partitions (equivalence classes):

- the `y`-axis

  Input arguments where `x == 0` may require special processing because of the required unique representation for rational number zero.

- each quadrant of the plane (excluding the axes)

  The `x-y` values in different quadrants may require different processing to handle the `y > 0` and "relatively prime" aspects of the interface invariant.

  Given that the module uses the finite integer type `Int`, we bound the quadrants by the maximum and minimum integer values along each axis.

We identify the following boundary values for special attention in our tests.

- Input pairs along the `x`-axis are outside any of the partitions.

- Input pairs composed of integer values 0, 1, and -1 are on the axes or just inside the "corners" of the quadrants . In addition, these are special values in various mathematical properties.

- Input pairs composed of the maximum `Int` (`maxInt`) and minimum `Int` (`minInt`) values may be near the outer bounds of the partitions.

  Note: If the machine's integer arithmetic uses the two's complement representation, then `minInt` can cause a problem with overflow because its negation is not in `Int`. Because of overflow, `-minInt == minInt`. So we should check both `minInt` and `-maxInt` in most cases.

In addition, we identify representative values for each quadrant. Although we do not partition the quadrants further, in each quadrant we should choose some input values whose (mathematical) rational number values differ and some whose values are the same.

Thus we choose the following `(x,y)` input pairs for testing:

- (0,0), (1,0), and (-1,0) as error inputs along the `x`-axis

- (0,1), (0,-1), (0,9), and (0,-9) as inputs along the `y`-axis
- (1,1), (9,9), and (`maxInt`,`maxInt`) as inputs from the first quadrant and (-1,-1), (-9,-9), and (−`maxInt`,−`maxInt`) as inputs from the third quadrant, all of whom have the same rational number value $\frac{1}{1}$.

  We also test input pairs (`minInt`,`minInt`) and (−`minInt`,−`minInt`), cognizant that the results might depend upon the machine's integer representation.
- (-1,1), (-9,9), and (−`maxInt`,`maxInt`) as inputs from the second quadrant and (1,-1), (9,-9), and (`maxInt`,−`maxInt`) as inputs from the fourth quadrant, all of whom have the same rational number value $-\frac{1}{1}$.

  We also test input pairs (−`minInt`,`minInt`) and (`minInt`,−`minInt`), cognizant that the results might depend upon the machine's integer representation.
- (3,2) and (12,8) as inputs from the first quadrant and (-3,-2) and (-12,-8) as inputs from the third quadrant, all of whom have the same rational number value $\frac{3}{2}$.
- (-3,2) and (-12,8) as inputs from the second quadrant and (3,-2) and (12,-8) as inputs from the fourth quadrant, all of whom have the same rational number value $-\frac{3}{2}$.
- (`maxInt`,1), (`maxInt`,-1), (−`maxInt`,1) and (−`maxInt`,-1) as input values in the "outer corners" of the quadrants.

  We also test input pairs (`minInt`,1) and (`minInt`,-1), cognizant that the results might depend upon the machine's integer representation.

12.4.2.2 Act As we identified in the introduction to this example, we must carry out a pair of actions in our tests. For example,

```
numer (makeRat 12 8)
```

and

```
denom (makeRat 12 8)
```

for the test of the input pair (12,8).

Note: The code above creates each test object (e.g., `makeRat 12 8`) twice. These could be created once and then used twice to make the tests run slightly faster.

12.4.2.3 Assert The results of the test actions must then be examined to determine whether they have the expected values. In the case of the `makeRat-numer-denom` tests, it is sufficient to compare the result for equality with the expected result. The expected result must satisfy the interface invariant.

For the two actions listed above, the comparison are

```
numer (makeRat 12 8) == 3
```

and

```
denom (makeRat 12 8) == 2
```

for the test of the input pair (12,8).

12.4.2.4 Aggregate into test script As with the factorial functions in Section 12.3, we can bring the various test actions together into a Haskell `IO` program. The excerpt below shows some of the tests.

```
pass :: Bool -> String
pass True  = "PASS"
pass False = "FAIL"

main :: IO ()
main =
    do
        -- Test 3/2
        putStrLn ("numer (makeRat 3 2) == 3:              " ++
                  pass (numer (makeRat 3 2) == 3))
        putStrLn ("denom (makeRat 3 2) == 2:              " ++
                  pass (denom (makeRat 3 2) == 2))
        -- Test -3/-2
        putStrLn ("numer (makeRat (-3) (-2)) == 3:        " ++
                  pass (numer (makeRat (-3) (-2)) == 3))
        putStrLn ("denom (makeRat (-3) (-2)) == 2:        " ++
                  pass (denom (makeRat (-3) (-2)) == 2))
        -- Test 12/8
        putStrLn ("numer (makeRat 12 8) == 3:             " ++
                  pass (numer (makeRat 12 8) == 3))
        putStrLn ("denom (makeRat 12 8) == 2:             " ++
                  pass (denom (makeRat 12 8) == 2))
        -- Test -12/-8
        putStrLn ("numer (makeRat (-12) (-8)) == 3:       " ++
                  pass (numer (makeRat (-12) (-8)) == 3))
        putStrLn ("denom (makeRat (-12) (-8)) == 2:       " ++
                  pass (denom (makeRat (-12) (-8)) == 2))
        -- Test 0/0
        putStrLn ("makeRat 0 0 is error:                  "
                  ++ show (makeRat 0 0))
            `catch` (\(ErrorCall msg)
                         -> putStrLn ("[Error Call] (EXPECTED)\n"
                                ++ msg))
```

The first four pairs of tests above check the test inputs (3,2), (-3,-2), (12,8), and (-12,-8). These are four test inputs, drawn from the first and third quadrants, that all have the same rational number value $\frac{3}{2}$.

The last test above checks whether the error pair (0,0) responds with an error exception as expected.

For the full test script (including tests of `showRat`) examine the source file `TestRatRepCore.hs` or `TestRatRepDefer.hs`.

12.4.2.5 Broken encapsulation

So far, the tests have assumed that any rational number object passed as an argument to `numer`, `denom`, and `showRat` is an object returned by `makeRat`.

However, the encapsulation of the data type `Rat` within a `RationalRep` module is just a convention. `Rat` is really an alias for `(Int,Int)`. The alias is exposed when the module is imported.

A user could call a function and directly pass an integer pair. If the integer pair does not satisfy the interface invariant, then the functions might not return a valid result.

For example, if we call `numer` with the invalid rational number value (1,0), what is returned?

Because this value is outside the specification for `RationalRep`, each implementation could behave differently. In fact, `RationalCore` returns the first component of the tuple and `RationalDeferGCD` throws a "divide by zero" exception.

The test scripts include tests of the invalid value (1,0) for each of the functions `numer`, `denom`, and `showRat`.

A good solution to this broken encapsulation problem is (a) to change `Rat` to a user-defined type and (b) only export the type name but not its components. Then the Haskell compiler will enforce the encapsulation we have assumed. We discuss approach in later chapters.

12.4.3 Rational arithmetic modules

TODO: Write section

The interface to the module `Rational` consists of the functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`, the `RationalRep` module's interface. It does not add any new data types, constructors, or destructors.

The `Rational` abstract module's functions preserve the interface invariant for the `RationalRep` abstract module, but it does not add any new components to the invariant.

12.4.3.1 Arrange TODO: Write section

TODO: Draw a diagram to help visualize input domain

12.4.3.2 Act TODO: Write section

12.4.3.3 Assert TODO: Write section

12.4.3.4 Aggregate into test script TODO: Write section

TODO: Discuss `TestRational1.hs` and `TestRational2.hs`

12.4.4 Reflection on this example

TODO: Update after completing chapter

I designed and implemented the `Rational` and `RationalCore` modules using the approach described in the early sections of Chapter 7, doing somewhat ad hoc testing of the modules with the REPL. I later developed the `RationalDeferGCD` module, abstracting from the `RationalCore` module. After that, I wrote Chapter 7 to describe the example and the development process. Even later, I constructed the systematic test scripts and wrote Chapters 11 and 12 (this chapter).

As I am closing out the discussion of this example, I find it useful to reflect upon the process.

- The problem seemed quite simple, but I learned there are several subtle issues in the problem and the modules developed to solve it. As the saying goes, "the devil is in the details".
- In my initial development and testing of these simple modules, I got the "happy paths" right and covered the primary error conditions. Although singer Bobby McFerrin's song "Don't Worry, Be Happy" may give good advice for many life circumstances, it should not be taken too literally for software development and testing.
- In writing both Chapter 7 and this chapter, I realized that my statements of the preconditions, postconditions, and interface invariants of `RationalRep` abstraction needed to be reconsidered and restated more carefully. Specifying a good abstract interface for a family of modules is challenging.
- In developing the systematic test scripts, I encountered other issues I had either not considered sufficiently or overlooked totally:
  - the full implications of using the finite data `Int` data type for the rational arithmetic modules
  - the impact of the underlying integer arithmetic representation (e.g., as two's complement) on the Haskell code

- the effects of calls of functions like `numer`, `denom`, and `showRat` with invalid input data
- a subtle violation of the interface invariant in the `RationalDeferGCD` implementations of `makeRat` and `showRat`
- the value of a systematic input domain partitioning for both developing good tests and understanding the problem

It took me much longer to develop the systematic tests and document them than it did to develop the modules initially. I clearly violated the Meszaros's final principle, "ensure commensurate effort and responsibility" described in the previous chapter (also in Mesazaros [126, Ch. 5]).

For future programming, I learned I need to pay attention to other of Meszaros's principles such as "design for testability", "minimize untestable code", "communicate intent", and perhaps "write tests first" or at least to develop the tests hand-in-hand with the program.

12.5 What Next?

Chapters 11 and 12 examined software testing concepts and applied them to testing Haskell functions and modules from Chapters 4 and 7.

So far we have limited our examples mostly to primitive types. In Chapters 13 and 14, we explore first-order, polymorphic list programming in Haskell.

12.6 Chapter Source Code

The source code for the group of factorial functions from Chapters 4 and 9 is in following
files:

- `Factorial.hs`, the source code for the functions
- `TestFactorial.hs`, the source code for the factorial test script

The source code for the rational arithmetic modules from Chapter 7 is in following files:

- `RationalCore.hs` and `RationalDeferGCD.hs`, the source code for the two implementations of the "RationalRep" abstract module
- `TestRatRepCore.hs` and `TestRatRepDefer.hs`, the test scripts for the two above implementations of the "RationalRep" abstract module
- `Rational1.hs` and `Rational2.hs`, the source code for the `Rational` arithmetic module paired with the two above implementations of the "RationalRep" abstract module
- `TestRational1.hs` and `TestRational2.hs`, the test scripts for the `Rational` module paired with the two "RationalRep" implementations

12.7 Exercises

1. Using the approach of this chapter, develop a black-box unit-testing script for the `fib` and `fib2` Fibonacci functions from Chapter 9. Test the functions with your script.

2. Using the approach of this chapter, develop a black-box unit-testing script for the `expt`, `expt2`, and `expt3` exponentiation functions from Chapter 9. Test the functions with your script.

3. Using the approach of this chapter, develop a black-box unit/module-testing script for the module `Sqrt` from Chapter 6. Test the module with your script.

4. Using the approach of this chapter, develop a black-box unit/module-testing script for the line-segment modules developed in exercises 1-3 of Chapter 7. Test the module with your script.

12.8 Acknowledgements

I wrote this chapter in Summer 2018 for the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming*.

- The presentation builds on the concepts and techniques surveyed in the Chapter 11, which was written at the same time.

- The presentation and use of the Arrange-Act-Assert pattern draws on the discussion in Beck [10] and Koskela [112].

- The testing examples draw on previously existing function and (simple) test script examples and on discussion of the examples in Chapters 4 and 7. However, I did redesign and reimplement the test scripts to be more systematic and to follow the discussion in this new chapter.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

12.9 Terms and Concepts

Test, testing level, testing method, testing type, unit and module testing (levels), black-box and gray-box testing (methods), functional testing (type), arrange-act-assert, input domain, input partitioning, representative values (for equivalence

classes), boundary values, testing based on the specification, Haskell `IO` program, `do`, `putStrLn`, exceptions.

13 List Programming

13.1 Chapter Introduction

This chapter introduces the list data type and develops the fundamental programming concepts and techniques for first-order polymorphic functions to process lists.

The goals of the chapter are to:

- introduce Haskell syntax and semantics for programming constructs related to polymorphic list data structures
- examine correct Haskell functional programs consisting of first-order polymorphic functions that solve problems by processing lists and strings
- explore methods for developing Haskell list-processing programs that terminate and are efficient and elegant
- examine the concepts and use of data sharing in lists

The Haskell module for this chapter is in `ListProg.hs`.

13.2 Polymorphic List Data Type

As we have seen, to do functional programming, we construct programs from collections of pure functions. Given the same arguments, a *pure function* always returns the same result. The function application is thus *referentially transparent*.

Such a pure function does not have *side effects*. It does not modify a variable or a data structure in place. It does not throw an exception or perform input/output. It does nothing that can be seen from outside the function except return its value.

Thus the data structures in purely functional programs must be *immutable*, not subject to change as the program executes.

Functional programming languages often have a number of immutable data structures. However, the most salient one is the list.

We mentioned the Haskell list and string data types in Chapter 5. In this chapter, we look at lists in depth. Strings are just special cases of lists.

13.2.1 List: `[t]`

The primary built-in data structure in Haskell is the list, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with the notation `[t]` to denote a list of zero or more elements of type `t`.

A *list* is is hierarchical data structure. It is either *empty* or it is a pair consisting of a *head element* and a *tail* that is itself a *list* of elements.

The Haskell list is an example of an *algebraic data type*. We discuss that concept in Chapter 21.

A matching pair of empty square brackets (`[]`), pronounced "nil", represents the empty list.

A colon (`:`), pronounced "cons", represents the *list constructor* operation between a *head* element on the left and a *tail* list on the right.

Example lists include:

```
[]
2:[]
3:(2:[])
```

The Haskell language adds a bit of *syntactic sugar* to make expressing lists easier. (By syntactic sugar, we mean notation that simplifies expression of a concept but that adds no new functionality to the language. The new notation can be defined in terms of other notation within the language.)

The cons operations *binds from the right*. Thus

```
5:(3:(2:[]))
```

can be written as:

```
5:3:2:[]
```

We can write this as a comma-separated sequence enclosed in brackets as follows:

```
[5,3,2]
```

Haskell supports two list selector functions, `head` and `tail`, such that

`head (h:t)` $\Longrightarrow$ `h`

where `h` is the head element of list, and

`tail (h:t)` $\Longrightarrow$ `t`

where `t` is the tail list.

Aside: Instead of `head`, Lisp uses `car` and other languages use `hd`, `first`, etc. Instead of `tail`, Lisp uses `cdr` and other languages use `tl`, `rest`, etc.

The Prelude library supports a number of other useful functions on lists. For example, `length` takes a list and returns its length.

Note that lists are *defined inductively*. That is, they are defined in terms of a base element `[]` and the list constructor operation cons (`:`). As you would expect, a form of mathematical induction can be used to prove that list-manipulating functions satisfy various properties. We will discuss in Chapter 25.

13.2.2 String: String

In Haskell, a *string* is treated as a list of characters. Thus the data type String is defined as a *type synonym*:

```
type String = [Char]
```

In addition to the standard list syntax, a String literal can be given by a sequence of characters enclosed in double quotes. For example, "oxford" is shorthand for ['o','x','f','o','r','d']'.

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code \& is used to separate any character sequences that are otherwise ambiguous.

Example: "Hello\nworld!\n" is a string that has two newline characters embedded.

Example: "\12\&3" represents the list ['\12','3'].

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings.

Consider a function to compute the length of a finite string:

```
len :: String -> Int
len s  =  if s == [] then 0 else 1 + len (tail s)
```

Note that the argument string for the recursive application of len is simpler (i.e., shorter) than the original argument. Thus len will eventually be applied to a [] argument and, hence, len's evaluation will terminate.

How efficient is this function (i.e., its time and space complexity)?

Consider the evaluation of the expression len "five" using the evaluation model from Chapter 8.

```
len "five"
```

$\Longrightarrow$

```
if "five" == [] then 0 else 1 + len (tail "five")
```

$\Longrightarrow$

```
if False then 0 else 1 + len (tail "five")
```

$\Longrightarrow$

```
1 + len (tail "five")
```

$\Longrightarrow$

```
1 + len "ive"
```

$\Longrightarrow$

```
    1 + (if "ive" == [] then 0 else 1 + len (tail "ive"))
==>
    1 + (if False then 0 else 1 + len (tail "ive"))
==>
    1 + (1 + len (tail "ive"))
==>
    1 + (1 + len "ve")
==>
    1 + (1 + (if "ve" == [] then 0 else 1 + len (tail "ve")))
==>
    1 + (1 + (if False then 0 else 1 + len (tail "ve")))
==>
    1 + (1 + (1 + len (tail "ve")))
==>
    1 + (1 + (1 + len "e"))
==>
    1 + (1 + (1 + (if "e" == [] then 0 else 1 + len (tail "e"))))
==>
    1 + (1 + (1 + (if False then 0 else 1 + len (tail "e"))))
==>
    1 + (1 + (1 + (1 + len (tail "e"))))
==>
    1 + (1 + (1 + (1 + len "")))
==>
    1 + (1 + (1 + (1 + (if "" == [] then 0 else 1 + len (tail "")))))
==>
    1 + (1 + (1 + (1 + (if True then 0 else 1 + len (tail "")))))
==>
    1 + (1 + (1 + (1 + 0)))
==>
    1 + (1 + (1 + 1))
```

$\Longrightarrow$

```
1 + (1 + 2)
```

$\Longrightarrow$

```
1 + 3
```

$\Longrightarrow$

```
4
```

If `n` is the length of the list `xs`, then `len s` requires `4*n` reduction steps involving the recursive leg (first 16 steps above), 2 steps involving the nonrecursive leg (next 2 steps above), and `n+1` steps involving the additions (last five steps). Thus, the evaluation requires `5*n+3` reduction steps. Hence, the number of reduction steps in proportional to the length of the input list. The time complexity of the function is thus O(`length s`{.haskell]).

The largest expression above is

```
1 + (1 + (1 + (1 + (if "" == [] then 0 else 1 + len (tail "")))))
```

This expression has `n + 2` (6) binary operators, 2 unary operators, and 1 ternary operator. Counting arguments (as discussed in Chapter 8), it has size `2 * (n + 2) + 2 + 3` (or `2*n+9`). Hence, the amount of space required (given lazy evaluation) is also proportional to the length of the input list. The space complexity of the function is thus O(`length s`).

13.2.3 Polymorphic lists

The above definition of `len` only works for strings. How can we make it work for a list of integers or other elements?

For an arbitrary type `a`, we want `len` to take objects of type `[a]` and return an `Int` value. Thus its type signature could be:

```
len :: [a] -> Int
```

If `a` is a variable name (i.e., it begins with a lowercase letter) that does not already have a value, then the type expression `a` used as above is a *type variable*; it can represent an arbitrary type. All occurrences of a type variable appearing in a *type signature* must, of course, represent the same type.

An object whose type includes one or more type variables can be thought of having many different types and is thus described as having a *polymorphic type*. (The next subsection gives more detail on polymorphism in general.)

Polymorphism and first-class functions are powerful abstraction mechanisms: they allow irrelevant detail to be hidden.

Other examples of polymorphic list functions from the Prelude library include:

```haskell
head :: [a] -> a
tail :: [a] -> [a]
(:)  :: a -> [a] -> [a]
```

13.3 Programming with List Patterns

In the factorial examples in Chapter 4, we used integer patterns and guards to break out various cases of a function definition into separate equations. Lists and other data types may also be used in patterns.

Pattern matching helps enable the *form of the algorithm* to match the *form of the data structure*. Or, as others may say, it helps in *following types to implementations*.

This is considered elegant. It is also pragmatic. The structure of the data often suggests the algorithm that is needed for a task.

In general, lists have two cases that need to be handled: the empty list and the nonempty list. Breaking a definition for a list-processing function into these two cases is usually a good place to begin.

13.3.1 Summing a list of integers: `sum'`

Consider a function `sum'` to sum all the elements in a finite list of integers. That is, if the list is

$$v_1, v_2, v_3, \cdots, v_n,$$

then the sum of the list is the value resulting from inserting the addition operator between consecutive elements of the list:

$$v_1 + v_2 + v_3 + \cdots + v_n.$$

Because addition is an *associative* operation (that is, $(x + y) + z = x + (y + z)$ for any integers x, y, and z), the above additions can be computed in any order.

What is the sum of an empty list?

Because there are no numbers to add, then, intuitively, zero seems to be the proper value for the sum.

In general, if some binary operation is inserted between the elements of a list, then the result for an empty list is the *identity* element for the operation. Since $0 + x = x = x + 0$ for all integers x, zero is the identity element for addition.

Now, how can we compute the sum of a nonempty list?

Because a nonempty list has at least one element, we can remove one element and add it to the sum of the rest of the list. Note that the "rest of the list" is a simpler (i.e., shorter) list than the original list. This suggests a recursive definition.

The fact that Haskell defines lists recursively as a cons of a head element with a tail list suggests that we structure the algorithm around the structure of the *beginning* of the list.

Bringing together the two cases above, we can define the function `sum'` in Haskell as follows. This is similar to the Prelude function `sum`.

```
{- Function sum' sums a list of integers. It is similar to
   function sum in the Prelude.
-}
sum' :: [Int] -> Int
sum' []     = 0            -- nil list
sum' (x:xs) = x + sum' xs -- non-nil list
```

- As noted previously, all of the text between the symbol "`--`" and the end of the line represents a *comment*; it is ignored by the Haskell interpreter.

  The text enclosed by the `{-` and `-}` is a block comment, that can extend over multiple lines.

- This definition uses two *legs*. The equation in the first leg is used for nil list arguments, the second for non-nil arguments.

- Note the `(x:xs)` pattern in the second leg. The "`:`" denotes the list constructor operation *cons*.

  If this pattern succeeds, then the head element of the list argument is bound to the variable `x` and the tail of the list argument is bound to the variable `xs`. These bindings hold for the right-hand side of the equation.

- The use of the cons in the pattern simplifies the expression of the case. Otherwise the second leg would have to be stated using the `head` and `tail` selectors as follows:

  ```
  sum' xs = head xs + sum' (tail xs)
  ```

- We use the simple name `x` to represent items of some type and the name `xs`, the same name with an `s` (for sequence) appended, to represent a list of that same type. This is a useful convention (adopted from the classic Bird and Wadler textbook [15]) that helps make a definition easier to understand.

- Remember that patterns (and guards) are tested in the order of occurrence (i.e., left to right, top to bottom). Thus, in most situations, the cases should be listed from the most specific (e.g., nil) to the most general (e.g., non-nil).

- The length of a non-nil argument decreases by one for each successive recursive application. Thus (assuming the list is finite) `sum'` will eventually be applied to a `[]` argument and terminate.

For a list consisting of elements 2, 4, 6, and 8, that is, `2:4:6:8:[]`, function `sum'` computes

```
2 + (4 + (6 + (8 + 0)))
```

giving the integer result 20.

Function `sum'` is backward linear recursive; its time and space complexity are both O(`n`), where `n` is the length of the input list.

We could, of course, redefine this to use a tail-recursive auxiliary function. With *tail call optimization*, the recursion could be converted into a loop. It would still be O(`n`) in time complexity (but with a smaller constant factor) but O(1) in space.

13.3.2 Multiplying a list of numbers: `product'`

Now consider a function `product'` to multiply together a finite list of integers.

The product of an empty list is 1 (which is the identity element for multiplication).

The product of a nonempty list is the head of the list multiplied by the product of the tail of the list, except that, if a 0 occurs anywhere in the list, the product of the list is 0.

We can thus define `product'` with two base cases and one recursive case, as follows. This is similar to the Prelude function `product`.

```
product' :: [Integer] -> Integer
product' []     = 1
product' (0:_)  = 0
product' (x:xs) = x * product' xs
```

Note the use of the wildcard pattern underscore "`_`" in the second leg above. This represents a "don't care" value. In this pattern it matches the tail, but no value is bound; the right-hand side of the equation does not need the actual value.

0 is the *zero element* for the multiplication operation on integers. That is, for all integers x:

$$0 * x = x * 0 = 0$$

For a list consisting of elements 2, 4, 6, and 8, that is, `2:4:6:8:[]`, function `product'` computes:

```
2 * (4 * (6 * (8 * 1)))
```

which yields the integer result 384.

For a list consisting of elements 2, 0, 6, and 8, function `product'` "short circuits" the computation as:

```
2 * 0
```

Like `sum'`, function `product'` is backward linear recursive; it has a worst-case time complexity of O(`n`), where n is the length of the input list. It terminates because the argument of each successive recursive call is one element shorter than the previous call, approaching the first base case.

As with `sum'`, we could redefine this to use a tail-recursive auxiliary function, which could evaluate in O(`n`) space with tail call optimization.

Note that `sum'` and `product'` have similar computational patterns. In Chapter 15, we look at how to capture the commonality in a single higher-order function.

13.3.3 Length of a list: `length'`

As another example, consider the function for the length of a finite list that we discussed earlier (as `len`). Using list patterns we can define `length'` as follows:

```
length' :: [a] -> Int
length' []     = 0                -- nil list
length' (_:xs) = 1 + length' xs -- non-nil list
```

Note the use of the wildcard pattern underscore "`_`". In this pattern it matches the head, but no value is bound; the right-hand side of the equation does not need the actual value.

Given a finite list for its argument, does this function terminate? What are its time and space complexities?

This definition is similar to the definition for `length` in the Prelude.

13.3.4 Remove duplicate elements: `remdups`

Consider the problem of removing adjacent duplicate elements from a list. That is, we want to replace a group of adjacent elements having the same value by a single occurrence of that value.

As with the above functions, we let the form of the data guide the form of the algorithm, following the type to the implementation.

The notion of adjacency is only meaningful when there are two or more of something. Thus, in approaching this problem, there seem to be three cases to consider:

- The argument is a list whose first two elements are duplicates; in which case one of them should be removed from the result.
- The argument is a list whose first two elements are not duplicates; in which case both elements are needed in the result.
- The argument is a list with fewer than two elements; in which case the remaining element, if any, is needed in the result.

Of course, we must be careful that sequences of more than two duplicates are handled properly.

Our algorithm thus can examine the first two elements of the list. If they are equal, then the first is discarded and the process is repeated recursively on the list remaining. If they are not equal, then the first element is retained in the result and the process is repeated on the list remaining. In either case the remaining list is one element shorter than the original list. When the list has fewer than two elements, it is simply returned as the result.

If we restrict the function to lists of integers, we can define Haskell function `remdups` as follows:

```
remdups :: [Int] -> [Int]
remdups (x:y:xs)
    | x == y = remdups (y:xs)
    | x /= y = x : remdups (y:xs)
remdups xs = xs
```

- Note the use of the pattern `(x:y:xs)`. This pattern match succeeds if the argument list has at least two elements: the head element is bound to `x`, the second element to `y`, and the tail list to `xs`.
- Note the use of guards to distinguish between the cases where the two elements are equal (==) and where they are not equal (/=).
- In this definition the case patterns overlap, that is, a list with at least two elements satisfies both patterns. But since the cases are evaluated top to bottom, the list only matches the first pattern. Thus the second pattern just matches lists with fewer than two elements.

What if we wanted to make the list type polymorphic instead of just integers?

At first glance, it would seem to be sufficient to give `remdups` the polymorphic type `[a] -> [a]`. But the guards complicate the situation a bit.

Evaluation of the guards requires that Haskell be able to compare elements of the polymorphic type `a` for equality (==) and inequality (/=). For some types these comparisons may not be supported. (For example, suppose the elements are functions.) Thus we need to restrict the polymorphism to types in which the comparisons are supported.

We can restrict the range of types by using a *context* predicate. The following type signature restricts the polymorphism of type variable `a` to the built-in *type class* `Eq`, the group of types for which both equality (==) and inequality (/=) comparisons have been defined:

```
remdups :: Eq a => [a] -> [a]
```

Another useful context is the class `Ord`, which is an *extension* of class `Eq`. `Ord` denotes the class of objects for which the relational operators <, <=, >, and >= have been defined in addition to == and /=.

Note: Chapter 22 explores the concepts of type class, instances, and overloading in more depth.

In most situations the type signature can be left off the declaration of a function. Haskell then attempts to infer an appropriate type. For `remdups`, the type inference mechanism would assign the type `Eq [a] => [a] -> [a]` . However, in general, it is good practice to give explicit type signatures.

Like the previous functions, `remdups` is backward linear recursive; it takes a number of steps that is proportional to the length of the list. This function has a recursive call on both the duplicate and non-duplicate legs. Each of these recursive calls uses a list that is shorter than the previous call, thus moving closer to the base case.

13.3.5 More list patterns

Table 13.1 shows Haskell parameter patterns, corresponding arguments, and the results of the attempted match.

Table 13.1: Example Haskell parameter patterns and match results.

Pattern	**Argument**	**Succeeds?**	**Bindings**
`1`	`1`	yes	none
`x`	`1`	yes	`x` ← `1`
`(x:y)`	`[1,2]`	yes	`x` ← `1`, `y` ← `[2]`
`(x:y)`	`[[1,2]]`	yes	`x` ← `[1,2]`, `y` ← `[]`
`(x:y)`	`["olemiss"]`	yes	`x` ← `"olemiss"`, `y` ← `[]`
`(x:y)`	`"olemiss"`	yes	`x` ← `'o'`, `y` ← `"lemiss"`
`(1:x)`	`[1,2]`	yes	`x` ← `[2]`
`(1:x)`	`[2,2]`	no	none
`(x:_:_:y)`	`[1,2,3,4,5,6]`	yes	`x` ← `1`, `y` ← `[4,5,6]`
`[]`	`[]`	yes	none
`[x]`	`["Cy"]`	yes	`x` ← `"Cy"`
`[1,x]`	`[1,2]`	yes	`x` ← `2`
`[x,y]`	`[1]`	no	none
`(x,y)`	`(1,2)`	yes	`x` ← `1`, `y` ← `2`

13.4 Data Sharing

Suppose we have the declaration:

```
xs = [1,2,3]
```

As we learned in the data structures course, we can implement this list as a singly linked list `xs` with three cells with the values `1`, `2`, and `3`, as shown in Figure 13.1.

Consider the following declarations (which are illustrated in Figure 13.1):

```
ys = 0:xs
zs = tail xs
```

where

- `0:xs` returns a list that has a new cell containing `0` in front of the previous list
- `tail xs` returns the list consisting of the last two elements of `xs`

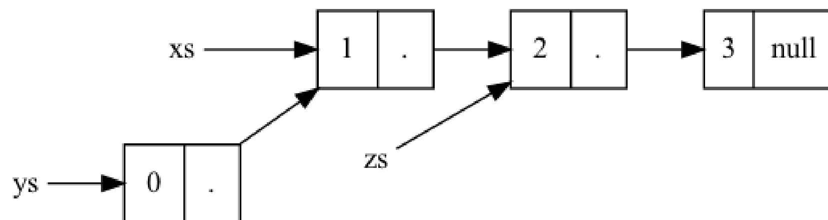


Figure 13.1: Data sharing in lists.

If the linked list `xs` is immutable (i.e., the values and pointers in the three cells cannot be changed), then neither of these operations requires any copying.

- The first just constructs a new cell containing `0`, links it to the first cell in list `xs`, and initializes `ys` with a reference to the new cell.
- The second just returns a reference to the second cell in list `xs` and initializes `zs` with this reference.
- The original list `xs` is still available, unaltered.

This is called *data sharing*. It enables the programming language to implement immutable data structures efficiently, without copying in many key cases.

Also, such functional data structures are *persistent* because existing references are never changed by operations on the data structure.

Consider evaluation of the expression `head xs`. It must create a copy of the head element (in this case `1`). The result does not share data with the input list.

Similarly, the list returned by function `remdups` (defined above) does not share data with its input list.

13.4.1 Preconditions for `head` and `tail`

What should `tail` return if the list is nil?

One choice is to return a nil list `[]`. However, it seems illogical for an empty list to have a tail.

Consider a typical usage of the `tail` function. It is normally an error for a program to attempt to get the tail of an empty list. Moreover, a program can efficiently check whether a list is empty or not. So, in this case, it is better to consider `tail` a partial function.

Thus, Haskell defines both `tail` and `head` to have the precondition that their parameters are non-nil lists. If we call either with a nil list, then it will terminate execution with a standard error message.

13.4.2 Dropping elements from beginning of list

We can generalize `tail` to a function `drop'` that removes the first `n` elements of a list as follows, (This function is called `drop` in the Prelude.)

```
drop' :: Int -> [a] -> [a]
drop' n xs | n <= 0 = xs
drop' _ []          = []
drop' n (_:xs)      = drop' (n-1) xs
```

Consider the example:

`drop 2 "oxford"` $\Longrightarrow \cdots$ `"ford"`

This function takes a different approach to the "empty list" issue than `tail` does. Although it is illogical to take the `tail` of an empty list, dropping the first element from an empty list seems subtly different. Given that we often use `drop'` in cases where the length of the input list is unknown, dropping the first element of an empty list does not necessarily indicate a program error.

Suppose instead that `drop'` would trigger an error when called with an empty list. To avoid this situation, the program might need to determine the length of the list argument. This is inefficient, usually requiring a traversal of the entire list to count the elements. Thus the choice for `drop'` to return a nil is also pragmatic.

The `drop'` function is tail recursive. The result list shares space with the input list.

The `drop'` function terminates when either the list argument is empty or the integer argument is 0 or negative. The function eventually terminates because each recursive call both shortens the list and decrements the integer.

What is the time complexity of `drop'`?

There are two base cases.

- For the first leg, the function must terminate in O(`max 1 n`) steps.

- For the second leg, the function must terminate within O(`length xs`) steps. So the function must terminate within O(`min (max 1 n) (length xs)`) steps.

What is the space complexity of `drop'`?

This tail recursive function evaluates in constant space when optimized.

13.4.3 Taking elements from the beginning of a list

Similarly, we can generalize `head'` to a function `take` that takes a number `n` and a list and returns the first `n` elements of the list.

```
take' :: Int -> [a] -> [a]
take' n _ | n <= 0 = []
take' _ []         = []
take' n (x:xs)     = x : take' (n-1) xs
```

Consider the following questions for this function?

- What is returned when the list argument is nil?
- Does evaluation of this function terminate?
- Does the result share data with the input?
- Is the function tail recursive?
- What are its time and space complexities?

Consider the example:

`take 2 "oxford"` $\Longrightarrow \cdots$ `"ox"`

13.5 What Next?

This chapter (13) examined programming with the list data type using first-order polymorphic functions. Chapter 14 continues the discussion of list programming, introducing infix operations and more examples.

13.6 Chapter Source Code

The Haskell module for this chapter is in `ListProg.hs`.

13.7 Exercises

1. Answer the following questions for the `take'` function defined in this chapter:

    - What is returned when the list argument is nil?
    - Does evaluation of the function terminate?
    - Does the result share data with the input?

- Is the function tail recursive?
- What are its time and space complexities?

2. Write a Haskell function `maxlist` to compute the maximum value in a nonempty list of integers. Generalize the function by making it polymorphic, accepting a value from any ordered type.

3. Write a Haskell function `adjpairs` that takes a list and returns the list of all pairs of adjacent elements. For example, `adjpairs [2,1,11,4]` returns `[(2,1), (1,11), (11,4)]`.

4. Write a Haskell function `mean` that takes a list of integers and returns the mean (i.e., average) value for the list.

5. Write the following Haskell functions using tail recursion:

   a. `sum''` with same functionality as `sum'`

   b. `product''` with the same functionality as `product'`

13.8 Acknowledgements

In Summer 2016, I adapted and revised much of this work from previous work:

- Chapter 5 of my *Notes on Functional Programming with Haskell* [42], which is influenced by Bird [13–15] and Wentworth [178]
- My notes on *Functional Data Structures (Scala)* [50], which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [29] and its associated materials [30,31]

In 2017, I continued to develop this work as Chapter 4, List Programming, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous List Programming chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 4.1-4.4 became the basis for new Chapter 13 (this chapter), List Programming, and previous sections 4.5-4.8 became the basis for Chapter 14, Infix Operators and List Programming Examples. I moved the discussion of "kinds of polymorphism" to new Chapter 5 and "local definitions" to new Chapter 9.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

13.9 Terms and Concepts

Type class (`Eq`, `Ord`, context predicate), lists (polymorphic, immutable, persistent, data sharing, empty/nil, nonempty), string, list and string operations (cons, head, tail, pattern matching, wildcard pattern, length), inductive definitions, operator binding, syntactic sugar, type synonym, type variable, type signature, follow the types to implementations, let the form of the data guide the form of the algorithm, associativity, identity element, zero element, termination, time and space complexity, adjacency,

14 Infix Operators and List Examples

14.1 Chapter Introduction

This chapter introduces Haskell infix operations and continues to develop techniques for first-order polymorphic functions to process lists.

The goals of the chapter are to:

- introduce Haskell syntax and semantics for infix operations
- examine correct Haskell functional programs consisting of first-order polymorphic functions that solve problems by processing lists and strings
- explore methods for developing Haskell list-processing programs that terminate and are efficient and elegant.

The Haskell module for this chapter is in `ListProgExamples.hs`.

14.2 Using Infix Operations

In Haskell, a *binary operation* is a function of type `t1 -> t2 -> t3` for some types `t1`, `t2`, and `t3`.

We usually prefer to use *infix* syntax rather than prefix syntax to express the application of a binary operation. Infix operators usually make expressions easier to read; they also make statement of mathematical properties more convenient.

Often we use several infix operators in an expression. To ensure that the expression is not ambiguous (i.e., the operations are done in the desired order), we must either use parentheses to give the order explicitly (e.g., `((y * (z+2)) + x)`) or use syntactic conventions to give the order implicitly.

Typically the application order for adjacent operators of different kinds is determined by the relative *precedence* of the operators. For example, the multiplication (`*`) operation has a higher precedence (i.e., binding power) than addition (`+`), so, in the absence of parentheses, a multiplication will be done before an adjacent addition. That is, `x + y * z` is taken as equivalent to `(x + (y * z))`.

In addition, the application order for adjacent operators of the same binding power is determined by a *binding* (or *association*) order. For example, the addition (`+`) and subtraction `-` operations have the same precedence. By convention, they bind more strongly to the *left* in arithmetic expressions. That is, `x + y - z` is taken as equivalent `((x + y) - z)`.

By convention, operators such as exponentiation (denoted by `^`) and cons bind more strongly to the *right*. Some other operations (e.g., division and the relational comparison operators) have no default binding order—they are said to have *free* binding.

Accordingly, Haskell provides the statements `infix`, `infixl`, and `infixr` for declaring a symbol to be an infix operator with free, left, and right binding,

respectively. The first argument of these statements give the precedence level as an integer in the range 0 to 9, with 9 being the strongest binding. Normal function application has a precedence of 10.

The operator precedence table for a few of the common operations from the Prelude is shown below. We introduce the ++ operator in the next subsection.

```
infixr 8 ^                          -- exponentiation
infixl 7 *                          -- multiplication
infix  7 /                          -- division
infixl 6 +, -                       -- addition, subtraction
infixr 5 :                          -- cons
infix  4 ==, /=, <, <=, >=, >       -- relational comparisons
infixr 3 &&                         -- Boolean AND
infixr 2 ||                         -- Boolean OR
```

14.2.1 Appending two lists: ++

Suppose we want a function that takes two lists and returns their concatenation, that is, *appends* the second list after the first. This function is a binary operation on lists much like + is a binary operation on integers.

Further suppose we want to introduce the infix operator symbol ++ for the append function. Since we want to evaluate lists lazily from their heads, we choose right binding for both cons and ++. Since append is, in a sense, an extension of cons (:), we give them the same precedence:

```
infixr 5 ++
```

Consider the definition of the append function. We must define the ++ operation in terms of application of already defined list operations and recursive applications of itself. The only applicable simpler operation is cons.

As with previous functions, we follow the type to the implementation—let the form of the data guide the form of the algorithm.

The cons operator takes an element as its left operand and a list as its right operand and returns a new list with the left operand as the head and the right operand as the tail.

Similarly, ++ must take a list as its left operand and a list as its right operand and return a new list with the left operand as the initial segment and the right operand as the final segment.

Given the definition of cons, it seems reasonable that an algorithm for ++ must consider the structure of its left operand. Thus we consider the cases for nil and non-nil left operands.

- If the left operand is nil, then the function can just return the right operand.

- If the left operand is a cons (that is, non-nil), then the result consists of the left operand's head followed by the append of the left operand's tail to the right operand.

In following the type to the implementation, we use the form of the left operand in a pattern match. We define ++ as follows:

```
infixr 5 ++

(++) :: [a] -> [a] -> [a]
[] ++ xs     = xs              -- nil left operand
(x:xs) ++ ys = x:(xs ++ ys)    -- non-nil left operand
```

Above we use infix patterns on the left-hand sides of the defining equations.

For the recursive application of ++, the length of the left operand decreases by one. Hence the left operand of a ++ application eventually becomes nil, allowing the evaluation to terminate.

Consider the evaluation of the expression `[1,2,3] ++ [3,2,1]`.

```
[1,2,3] ++ [3,2,1]
```

$\Longrightarrow$

```
1:([2,3] ++ [3,2,1])
```

$\Longrightarrow$

```
1:(2:([3] ++ [3,2,1]))
```

$\Longrightarrow$

```
1:(2:(3:([] ++ [3,2,1])))
```

$\Longrightarrow$

```
1:(2:(3:[3,2,1]))
```

$=$

```
[1,2,3,3,2,1]
```

The number of steps needed to evaluate `xs ++ ys` is proportional to the length of `xs`, the left operand. That is, the time complexity is O(`n`), where `n` is the length `xs`.

Moreover, `xs ++ ys` only needs to copy the list `xs`. The list `ys` is shared between the second operand and the result. If we did a similar function to append two (mutable) arrays, we would need to copy both input arrays to create the output array. Thus, in this case, a linked list is more efficient than arrays!

Consider the following questions:

- What is the precondition of `xs ++ ys`?

- Is ++ tail recursive?
- What is the space complexity of ++?

14.2.2 Properties of operations

The append operation has a number of useful algebraic properties, for example, associativity and an identity element.

Associativity of ++: For any finite lists `xs`, `ys`, and `zs`, `xs ++ (ys ++ zs) == (xs ++ ys) ++ zs`.

Identity for ++: For any finite list `xs`, `[] ++ xs = xs = xs ++ []`.

We will prove these and other properties in Chapter 25.

Mathematically, the list data type and the binary operation ++ form a kind of abstract algebra called a *monoid*. Function ++ is *closed* (i.e., it takes two lists and gives a list back), is associative, and has an identity element.

Similarly, we can state properties of combinations of functions. We can prove these using techniques we study in Chapter 25. For example, consider the functions defined above in this chapter.

- For all finite lists `xs`, we have the following distribution properties:

```
sum' (xs ++ ys)     = sum' xs + sum' ys
product' (xs ++ ys) = product' xs * product' ys
length' (xs ++ ys)  = length' xs + length' ys
```

- For all natural numbers `n` and finite lists `xs`,

```
take n xs ++ drop n xs = xs
```

14.2.3 Element selection: `!!`

As another example of an infix operation, consider the list selection operator `!!`. The expression `xs!!n` selects element `n` of list `xs` where the head is in position 0. It is defined in the Prelude similar to the way `!!` is defined below:

```
infixl 9 !!

(!!) :: [a] -> Int -> a
xs     !! n | n < 0 = error "!! negative index"
[]     !! _         = error "!! index too large"
(x:_)  !! 0         = x
(_:xs) !! n         = xs !! (n-1)
```

Consider the following questions concerning the element selection operator:

- What is the precondition for element selection?
- Does evaluation terminate?
- Is the operator tail recursive?
- Does the result share any data with the input list?

- What are its time and space complexities?

14.2.4 Reversing a list: `rev`

Consider the problem of reversing the order of the elements in a list.

Again we can use the structure of the data to guide the algorithm development. If the argument is nil, then the function returns nil. If the argument is non-nil, then the function can append the head element at the back of the reversed tail.

```
rev :: [a] -> [a]
rev []     = []              -- nil argument
rev (x:xs) = rev xs ++ [x]  -- non-nil argument
```

Given that evaluation of ++ terminates, we note that evaluation of `rev` also terminates because all recursive applications decrease the length of the argument by one.

How efficient is this function?

Consider the evaluation of the expression `rev "bat"`.

```
rev "bat"
```

$\Longrightarrow$

```
(rev "at") ++ "b"
```

$\Longrightarrow$

```
((rev "t") ++ "a") ++ "b"{.haskell}
```

$\Longrightarrow$

```
(((rev "") ++ "t") ++ "a") ++ "b"
```

$\Longrightarrow$

```
(("" ++ "t") ++ "a") ++ "b"
```

$\Longrightarrow$

```
("t" ++ "a") ++ "b"
```

$\Longrightarrow$

```
('t':("" ++ "a")) ++ "b"
```

$\Longrightarrow$

```
"ta" ++ "b"
```

$\Longrightarrow$

```
't':("a" ++ "b")
```

$\Longrightarrow$

```
't':('a':("" ++ "b"))
```

⟹

```
't':('a':"b")
```

=

```
"tab"
```

The evaluation of `rev` takes O(`n^2`) steps, where `n` is the length of the argument. There are O(`n`) applications of `rev`; for each application of `rev` there are O(`n`) applications of `++`.

The initial list and its reverse do not share data.

Function `rev` has a number of useful properties, for example the following.

Distribution: For any finite lists `xs` and `ys`, `rev (xs ++ ys) = rev ys ++ rev xs`.

Inverse: For any finite list `xs`, `rev (rev xs) = xs`.

Also, for any finite lists `xs` and `ys` and natural numbers `n`, we can state properties such as:

```
rev (xs ++ ys) = rev ys ++ rev xs
take n (rev xs)  = rev (drop (length xs - n) xs)
```

14.2.5 Tail recursive reverse

Most of the list function definitions examined so far are *backward recursive*. That is, for each case the recursive applications are embedded within another expression. Operationally, significant work is done after the recursive call returns.

Now let's look at the problem of reversing a list again to see whether we can devise a more efficient *tail recursive* solution.

As we have seen, the common technique for converting a backward linear recursive definition like `rev` into a tail recursive definition is to use an *accumulating parameter* to build up the desired result incrementally. A possible definition follows:

```
rev' [] ys     = ys
rev' (x:xs) ys = rev' xs (x:ys)
```

In this definition parameter `ys` is the accumulating parameter. The head of the first argument becomes the new head of the accumulating parameter for the tail recursive call. The tail of the first argument becomes the new first argument for the tail recursive call.

We know that `rev’` terminates because, for each recursive application, the length of the first argument decreases toward the base case of `[]`.

We note that `rev xs` is equivalent to `rev’ xs []`. We can prove this using the techniques in Chapter 25.

To define a single-argument replacement for `rev`, we can embed the definition of `rev’` as an *auxiliary* function within the definition of a new function `reverse’`. (This is similar to function `reverse` in the Prelude.)

```
reverse' :: [a] -> [a]
reverse' xs = rev xs []
              where rev []     ys = ys
                    rev (x:xs) ys = rev xs (x:ys)
```

The `where` clause introduces the local definition `rev’` that is only known within the right-hand side of the defining equation for the function `reverse’`.

What is the time complexity of this function?

The evaluation of `reverse’` takes O(`n`) steps, where `n` is the length of the argument. There is one application of `rev’` for each element; `rev’` requires a single cons operation in the accumulating parameter.

Where did the increase in efficiency come from?

Each application of `rev` applies ++, a linear time (i.e., O(`n`) function. In `rev’`, we replaced the applications of ++ by applications of cons, a constant time (i.e., O(1)) function.

In addition, a compiler or interpreter that does tail call optimization can translate this tail recursive call into a loop on the host machine.

14.3 More Useful List Functions

14.3.1 Another list-breaking function: `splitAt`

Above we defined list-breaking functions `take'` and `drop'`. It is sometimes useful to have a single function that breaks a list into two parts.

The function `splitAt` (shown below as `splitAt'`) takes an integer `n` and a list and returns a pair whose first component is the first `n` elements of the list and second component is the list remaining after the first `n` elements are removed.

```
splitAt' :: Int -> [a] -> ([a],[a])
splitAt' n xs =  (take' n xs, drop' n xs)
```

Can we write an alternative definition that makes only one pass over argument `xs`? (That is, it does not call `take'` and `drop'`.)

14.3.2 List-combining operations: `zip` and `unzip`

Another useful function in the Prelude is `zip` (shown below as `zip’`) which takes two lists and returns a list of pairs of the corresponding elements. That is, the

function fastens the lists together like a zipper. It's definition is similar to `zip'` given below:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys -- zip.1
zip' _      _      = []                 -- zip.1
```

Function `zip` applies a *tuple-forming* operation to the corresponding elements of two lists. It stops the recursion when either list argument becomes nil. Putting the recursive case first enabled the two bases cases to be combined into one leg.

Example: `zip [1,2,3] "oxford"` $\Longrightarrow \cdots$ `[(1,'o'),(2,'x'),(3,'f')]`

Similarly, function `unzip` in the Prelude takes a list of pairs and returns a pair of lists. It's definition is similar to `unzip'` below.

```
unzip' :: [(a,b)] -> ([a],[b])
unzip' []         = ([],[])
unzip' ((x,y):ps) = (x:xs, y:ys)
                    where (xs,ys) = unzip' ps
```

The Prelude includes versions of `zip` and `unzip` that handle the tuple-formation for triples. Librart `Data.List` includes functions for up to seven input lists: `zip4` $\cdots$ `zip7` and `unzip4` $\cdots$ `unzip7`.

14.4 Insertion Sort

Consider a function to sort the elements of a list into ascending order.

A list is *ascending* if every element is <= all of its successors in the list. Successor means an element that occurs later in the list (i.e., away from the head). A list is *increasing* if every element is < its successors. Similarly, a list is *descending* or *decreasing* if every element is >= or >, respectively, its successors.

A simple algorithm to do this is *insertion sort*. To sort a non-empty list with head `x` and tail `xs`, sort the tail `xs` and then insert the element `x` at the right position in the result. To sort an empty list, just return it.

If we restrict the function to integer lists, we get the following Haskell functions:

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)

insert :: Int -> [Int] -> [Int]
insert x []     = [x]
insert x xs@(y:ys)
    | x <= y    = (x:xs)
    | otherwise = y : (insert x ys)
```

Insertion sort has a (worst and average case) time complexity of O(`n^2`) where `n` is the length of the input list. (Function `isort` requires *n* consecutive recursive calls; each call uses function `insert` which itself requires on the order of `n` recursive calls.)

Now suppose we want to generalize the sorting function and make it polymorphic. We cannot just add a type parameter `a` and substitute it for `Int` everywhere. Not all Haskell types can be compared on a *total ordering* (<, <=, >, and >= as well).

We need to constrain the polymorphism to types in class `Ord`, as follows:

```
isort' :: Ord a => [a] -> [a]
isort' []     = []
isort' (x:xs) = insert' x (isort' xs)

insert' :: Ord a => a -> [a] -> [a]
insert' x []    = [x]
insert' x xs@(y:ys)
    | x <= y    = (x:xs)
    | otherwise = y : (insert' x ys)
```

We could define `insert'` inside `isort'` and avoid the separate type parameterization. But `insert` is separately useful, so it is reasonable to leave it external.

Consider the following questions:

- How do we know `insert'` terminates?
- What are the time and space complexities of `insert'`?
- How do we know `isort'` terminates?
- What are the time and space complexities of `isort'`?

14.5 What Next?

Chapters 13 and 14 explored use of first-order polymorphic functions to process lists in Haskell.

Chapters 15, 16, and 17 examine higher-order function concepts in Haskell.

14.6 Chapter Source Code

The Haskell module for this chapter is in `ListProgExamples.hs`.

14.7 Exercises

1. Answer the following questions for the `++` operation defined in this chapter:

    - Is ++ tail recursive?

- What is the space complexity?

1. Answer the following questions concerning the element selection operator defined in this chapter.

    - What is the precondition for element selection?
    - Does evaluation terminate?
    - Is the operator tail recursive?
    - Does the result share any data with the input list?
    - What are its time and space complexities?

2. Write a version of function `splitAt'` that makes only one pass over the input list (that is, does not call `take'` and `drop'`).

3. Answer the following questions for the `isort'` and `insert'` functions.

    - How do we know `insert'` terminates?
    - What are the time and space complexities of `insert'`?
    - How do we know `isort'` terminates?
    - What are the time and space complexities of `isort'`?

4. Hailstone functions.

    a. (This part is repeated from Chapter 9.) Develop a function `hailstone` to implement the function shown in Table 14.1.

    Table 14.1: Hailstone Function.

$hailstone(n)$	$=$	$1,$	if $n = 1$
$hailstone(n)$	$=$	$hailstone(n/2),$	if $n > 1$, even n
$hailstone(n)$	$=$	$hailstone(3 * n + 1),$	if $n > 1$, odd n

    Note that an application of the `hailstone` function to the argument `3` would result in the following "sequence" of "calls" and would ultimately return the result `1`.

    ```
    hailstone 3
        hailstone 10
            hailstone 5
                hailstone 16
                    hailstone 8
                        hailstone 4
                            hailstone 2
                                hailstone 1
    ```

    For further thought: What is the domain of the *hailstone* function?

    b. Write a Haskell function that computes the results of the `hailstone` function for each element of a list of positive integers. The value

returned by the `hailstone` function for each element of the list should be displayed.

c. Modify the `hailstone` function to return the function's "path."

   That is, each application of this path function should return a list of integers instead of a single integer. The list returned should consist of the arguments of the successive calls to the `hailstone` function necessary to compute the result. For example, the `hailstone 3` example above should return `[3,10,5,16,8,4,2,1]`.

5. Number base conversion.

   a. Write a Haskell function `natToBin` that takes a natural number and returns its binary representation as a list of `0`'s and `1`'s with the most significant digit at the head. For example, `natToBin 23` returns `[1,0,1,1,1]`. (Note: Prelude function `rem` returns the remainder from dividing its first argument by its second. Enclosing the function name in backquotes as in `` `rem` `` allows a two-argument function to be applied in an infix form.)

   b. Generalize `natToBin` to function `natToBase` that takes a base `b`

      (`b \geq 2`) and a natural number and returns the base `b` representation of the natural number as a list of integer digits with the most significant digit at the head. For example, `natToBase 5 42` returns `[1,3,2]`.

   c. Write Haskell function `baseToNat`, the inverse of the `natToBase` function. For any base `b` (`b \geq 2`) and natural number `n`:

```
baseToNat b (natToBase b n) = n
```

6. Write a Haskell function `merge` that takes two increasing lists of integers and merges them into a single increasing list (without any duplicate values). A list is *increasing* if every element is less than (<) its successors. Successor means an element that occurs later in the list, i.e., away from the head. Generalize the function by making it polymorphic.

7. Design a module of set operations. Choose a Haskell representation for sets. Implement functions to make sets from lists and vice versa, to insert and delete elements from sets, to do set union, intersection, and difference, to test for equality and subset relationships, to determine cardinality, and so forth.

8. Bag module.

   Mathematically, a *bag* (or *multiset*) is a function from some arbitrary set of elements (the domain) to the set of nonnegative integers (the range). We interpret the nonnegative integer as the number of occurrences of the element in the bag. Zero means the element does not occur.

From another perspective, a bag is an unordered collection of elements. Each element may occur one or more times in the bag. (It is like a set except values can occur multiple times.)

For example, `{| "time', "time", "and", "time", "again" |}` is a bag containing 5 strings. There are 3 occurrences of string `"time"` and 1 occurrence each of strings `"and"` and `"again"`.

`{| 11, 2, 3, 7, 5 |}` is a bag of prime numbers. It is also a set because each element occurs exactly once.

We can represent a bag in many ways in Haskell. Using lists, we could represent a bag with a simple (unordered) list of elements, an ordered list of elements, an unordered or an ordered list of tuples which pair an element with the (nonzero) number of times it occurs, etc. A bag could also be represented with other data structures such as a `Map` from library `Data.Map`.

Choose some representation for polymorphic bags. You may assume that the elements in the domain are totally ordered (i.e., are from a type that is an instance of class `Ord`), but otherwise the elements can be of any type.

For example, if you use a list representation, you might define the type synonym:

```haskell
type Bag a = [a]
```

Develop a data abstraction (information-hiding) module that encapsulates the representation of the data structure used to store the elements inside the module.

The module should include the following public functions. This interface should be the same even if you change the representation of the data internally.

a. `newBag` returns a new bag with no elements (i.e., empty).

b. `listToBag` takes a list of elements and returns a bag containing exactly those elements. The number of occurrences of an element in the list and in the resulting bag is the same.

c. `bagToList` takes a bag and returns a list containing exactly the elements occurring in the bag. The number of occurrences of an element in the bag and in the resulting list is the same.

   Note: It is not required that:

   ```haskell
   bagToList (listToBag xs) == xs
   ```

   But it is required that both sides have the same numbers of the same elements.

d. `isEmpty` takes a bag and returns `True` if the bag has no elements and returns `False` otherwise.

e. `isElem` takes an element and a bag and returns `True` if the element occurs in the bag and returns `False` otherwise.

f. `size` takes a bag and returns its cardinality (i.e., the total number of occurrences of all elements).

g. `occursBag` takes an element and a bag and returns the number of occurrences of the element in the bag.

h. `insertElem` takes an element and a bag and returns the bag with the element inserted. Bag insertion either adds a single occurrence of a new element to the bag or increases the number of occurrences of an existing element by one.

i. `deleteElem` takes an element and a bag and returns the bag with the element deleted. Bag deletion removes a single occurrence of an element from the bag, decreases the number of occurrences of an existing element by one, or does not change the bag if the element does not occur.

j. `eqBag` takes two bags and returns `True` if the two bags are equal (i.e., the same elements and same number of occurrences of each) and returns `False` otherwise.

   Note: If `bagToList xs == bagToList ys`, then `eqBag xs ys`. However, if `eqBag xs ys`, it is not required that `bagToList xs == bagToList ys`.

k. `unionBag` takes two bags and returns their bag union. The union of bags X and Y contains all elements that occur in either X or Y; the number of occurrences of an element in the union is the number in X or in Y, whichever is greater.

l. `intersectBag` takes two bags and returns their bag intersection. The intersection of bags X and Y contains all elements that occur in both X and Y; the number of occurrences of an element in the intersection is the number in X or in Y, whichever is lesser.

m. `sumBag` takes two bags and returns their bag sum. The sum of bags X and Y contains all elements that occur in X or Y; the number of occurrences of an element is the sum of the number of occurrences in X and Y.

n. `diffBag` takes two bags and returns the bag difference, first argument minus the second. The difference of bags X and Y contains all elements of X that occur in Y fewer times; the number of occurrences of an element in the difference is the number of occurrences in X minus the number in Y.

o. `subBag` takes two bags and returns `True` if the first is a subbag of the second and `False` otherwise. X is a subbag of Y if every element of X occurs in Y at least as many times as it does in X.

p. `bagToSet` takes a bag and returns a list containing exactly the *set* of elements contained in the bag. Each element occurring one or more times in the bag will occur exactly once in the list returned.

9. Develop a bag module as described in the previous exercise, but use a different internal representation than you used in the previous exercise. The new module should have the same public interface as the previous module.

10. Unbounded precision arithmetic module for natural numbers (i.e., nonnegative integers). Do not use the builtin `Integer` type.

    a. Define a type synonym `BigNat` to represent these unbounded precision natural numbers as lists of `Int`. Let each element of the list denote a decimal digit of the "big natural" number represented, with the *least* significant digit at the head of the list and the remaining digits given in order of *increasing* significance. For example, the integer value 22345678901 is represented as `[1,0,9,8,7,6,5,4,3,2,2]`.

       Use the following "canonical" representation:

       the value `0` is represented by the list `[0]` and positive numbers by a list without "leading" `0` digits (i.e., `126` is `[6,2,1]` not `[6,2,1,0,0]`). You may use the nil list `[]` to denote an error value.

       Define a Haskell module with basic arithmetic operations, including the following functions. Make sure that `BigNat` values returned by these functions are in canonical form.

       - `intToBig` takes a nonnegative `Int` and returns the `BigNat` with the same value.
       - `strToBig` takes a `String` containing the value of the number in the "usual" format (i.e., decimal digits, left to right in order of *decreasing* significance with zero or more leading spaces, but with no spaces or punctuation embedded within the number) and returns the `BigNat` with the same value.
       - `bigToStr` takes a `BigNat` and returns a `String` containing the value of the number in the "usual" format (i.e., left to right in order of *decreasing* significance with no spaces or punctuation).
       - `bigComp` takes two `BigNat`s and returns the `Int` value `-1` if the value of the first is less than the value of the second, the value `0` if they are equal, and the value `1` if the first is greater than the second.
       - `bigAdd` takes two `BigNat` s and returns their sum as a `BigNat`.

- `bigSub` takes two `BigNat` s and returns their difference as a `BigNat`, first argument minus the second.
- `bigMult` takes two `BigNat`s and returns their product as a `BigNat`.

b. Use the package to generate a table of factorials for the naturals 0 through 25. Print the values from the table in two *right-justified* columns, with the number on the left and its factorial on the right. (Allow about 30 columns for `25!`.)

c. Use the package to generate a table of Fibonacci numbers for the naturals 0 through 50.

d. Generalize the package to handle signed integers. Add the following new function:

- `bigNeg` returns the negation of its `BigNat` argument.

e. Add the following functions to the package:

- `bigDiv` takes two `BigNat`s and returns, as a `BigNat`, the quotient from dividing the first argument by the second.
- `bigRem` takes two `BigNat`s and returns, as a `BigNat`, the remainder from dividing the first argument by the second.

11. Define the following set of text-justification functions. You may want to use Prelude functions like `take`, `drop`, and `length`.

- `spaces' n` returns a string of length `n` containing only space characters (i.e., the character `' '`).
- `left' n xs` returns a string of length `n` in which the string `xs` begins at the head (i.e., left end).

  Examples: `left' 3 "ab"` yields `"ab "`; `left' 3 "abcd"` yields `"abc"`.
- `right' n xs` returns a string of length `n` in which the string `xs` ends at the tail (i.e., right end).

  Examples: `right' 3 bc` yields `bc`; `right' 3 abcd` yields `bcd`.
- `center' n xs` returns a string of length `n` in which the string `xs` is approximately centered.

  Example: `center' 4 "bc"` yields `" bc "`.

12. Consider simple mathematical expressions consisting of integer constants, variable names, parentheses, and the binary operators `+`, `-`, `*`, and `/`. For the purposes of this exercise, an *expression* is a string that satisfies the following (extended) BNF grammar and lexical conventions:

- The characters in an input string are examined left to right to form "lexical tokens". The tokens of the expression "language" consist of *addOp*s, *mulOp*s,*identifier*s, *number*s, and left and right parentheses.
- An expression may contain space characters at any position except within a lexical token.
- An *addOp* token is either a "+" or a "-"; a *mulOp* token is either a "*" or a "/".
- An *identifier* `q` is a string of one or more contiguous characters such that the leftmost character is a letter and the remaining characters are either letters, digits, or underscore characters.

  Examples: "`Hi1`", "`lo23_1`", "`this_is_2_long`"
- A *number* is a string of one or more contiguous characters such that all (including the leftmost) are digits.

  Examples: "`1`", "`23456711`"
- All *identifier* and *number* tokens extend as far to the right as possible. For example, consider the string

  "`A123 12B3+2 )`". (Note the space and right parenthesis characters). This string consists of the six tokens "`A123`", "`12`", "`B3`", "`+`", "`2`", and "`)`".

Define a Haskell function `valid` that takes a `String` and returns `True` if the string is an *expression* as described above and returns `False` otherwise.

Hints:

- If you need to return more than one value from a function, you can do so by returning a tuple of those values. This tuple can be decomposed by Prelude functions such as `fst` and `snd`.
- Use of the `where` or `let` features can simplify many functions. You may find Prelude functions such as `span`, `isSpace`, `isDigit`, `isAlpha`, and `isAlphanum` useful.
- You may want to consider organizing your program as a simple *recursive descent* recognizer for the expression language.

13. Extend the mathematical expression recognizer of the previous exercise to *evaluate* integer expressions with the given syntax. The four binary operations have their usual meanings.

    Define a function `eval e st` that evaluates expression `e` using symbol table `st`. If the expression `e` is syntactically valid, `eval` returns a pair `(True,val)` where `val` is the value of `e`. If `e` is not valid, `eval` returns `(False,0)`.

The symbol table consists of a list of pairs, in which the first component of a pair is the variable name (a string) and the second is the variable's value (an integer).

Example: `eval "(2+x) * y" [("y",3),("a",10),("x",8)]` yields `(True,30)`.

14.8 Acknowledgements

In Summer 2016, I adapted and revised much of this work from previous work:

- Chapter 5 of my *Notes on Functional Programming with Haskell* [42] which is influenced by Bird [13–15] and Wentworth [178]
- My notes on *Functional Data Structures (Scala)* [50], which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [29] and its associated materials [30,31]

In 2017, I continued to develop this work as Chapter 4, List Programming, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous List Programming chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 4.1-4.4 became the basis for new Chapter 13, List Programming, and previous sections 4.5-4.8 became the basis for Chapter 14, Infix Operators and List Programming Examples (this chapter).

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

14.9 Terms and Concepts

Binary operation, infix operation, properties of operators (associative, identity, zero, inverse, distribution), precedence (left, right, free binding).

15 Higher-Order Functions

15.1 Chapter Introduction

The previous chapters discussed first-order programming in Haskell. This chapter "kicks it up a notch" (to quote chef Emeril Lagasse) by adding powerful new abstraction facilities.

The goals of this chapter (15) are to:

- introduce first-class and higher-order functions
- construct a library of useful higher-order functions to process lists

This chapter continues the emphasis on Haskell programs that are correct, terminating, efficient, and elegant.

The chapter approaches the development of higher-order functions by generalizing a set of first-order functions having similar patterns of computation.

The Haskell module for this chapter is in file `HigherOrderFunctions.hs`.

15.2 Generalizing Procedural Abstractions

A function in a programming language is a *procedural abstraction*. It separates the logical properties of a computation from the details of how the computation is implemented. It abstracts a pattern of behavior and encapsulates it within a program unit.

Suppose we wish to perform the *same* computation on a set of *similar* data structures. As we have seen, we can encapsulate the computation in a function having the data structure as an argument. For example, the function `length'` computes the number of elements in a list of any type.4 Suppose instead we wish to perform a *similar* (but not identical) computation on a set of *similar* data structures. For example, we want to compute the sum or the product of a list of numbers. In this case, we may can pass the operation itself into the function.

This kind of function is called a *higher-order function*. A higher-order function is a function that takes functions as arguments or returns functions in a result. Most traditional imperative languages do not fully support higher-order functions.

In most functional programming languages, functions are treated as *first class* values. That is, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions. Historically, imperative languages have not treated functions as first-class values. (Recently, many imperative languages, such as Java 8, have added support for functions as first-class values.)

The higher-order functions in Haskell and other functional programming languages enable us to construct regular and powerful abstractions and operations.

By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

This can increase programmer productivity and program reliability because such programs are shorter, easier to understand, and constructed from well-tested components.

Higher-order functions can also increase the *modularity* of programs by enabling simple program fragments to be "glued together" readily into more complex programs.

In this chapter, we examine several common patterns and build a library of useful higher-order functions.

15.3 Defining `map`

Consider the following two functions, noting their type signatures and patterns of recursion.

The first, `squareAll`, takes a list of integers and returns the corresponding list of squares of the integers.

```
squareAll :: [Int] -> [Int]  squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = (x * x) : squareAll xs
```

The second, `lengthAll,`q takes a list of lists and returns the corresponding list of the lengths of the element lists; it uses the Prelude function `length`.

```
lengthAll :: [[a]] -> [Int]
lengthAll []       = []
lengthAll (xs:xss) = (length xs) : lengthAll xss
```

Although these functions take different kinds of data (a list of integers versus a list of polymorphically typed lists) and apply different operations (squaring versus list length), they exhibit the same pattern of computation. That is, both take a list of some type and apply a given function to each element to generate a resulting list of the same length as the original.

The combination of polymorphic typing and higher-order functions allow us to abstract this pattern of computation into a standard function.

We can abstract the pattern of computation common to `squareAll` and `lengthAll` as the (broadly useful) function `map`, which we define as follows. (In this chapter, we often add a suffix to the base function names to avoid conflicts with the similarly named functions in the Prelude. Here we use `map'` instead of `map`.)

```
map' :: (a -> b) -> [a] -> [b]  -- map in Prelude
map' f []     = []
```

```haskell
map' f (x:xs) = f x : map' f xs
```

Function `map` *generalizes* `squareAll`, `lengthAll`, and similar functions by adding a higher-order parameter for the operation applied and making the input and the output lists polymorphic. Specifically, he function takes a function `f` of type `a -> b` and a list of type `[a]`, applies function `f` to each element of the list, and produces a list of type `[b]`.

Thus we can *specialize* `map` to give new definitions of `squareAll` and `lengthAll` as follows:

```haskell
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x

lengthAll2 :: [[a]] -> [Int]
lengthAll2 xss = map' length xss
```

Consider the following questions.

- Under what circumstances does `map' f xs` terminate? Do we have to assume anything about `f`? about `xs`?
- What is the time complexity of `map f xs`?
- What is the time complexity of `squareAll2 xs`? Of `lengthAll2 xs`?

15.4 Thinking about Data Transformations

Above we define `map` as a recursive function that transforms the elements of a list one by one. However, it is often more useful to think of `map` in one of two ways:

1. as a powerful list operator that transforms every element of the list. We can combine `map` with other powerful operators to quickly construct powerful list processing programs.

   We can consider `map` as operating on every element of the list "simultaneously". In fact, an implementation could use separate processors to transform each element: this is essentially the `map` operation in Google's `mapReduce` distributed "big data" processing framework.

   Referential transparency and immutable data structures make parallelism easier in Haskell than in most imperative languages.

2. as a operator node in a dataflow network. A stream of data objects flows into the `map` node. The `map` node transforms each object by applying the argument function. Then the data object flows out to the next node of the network.

   The lazy evaluation of the Haskell functions enables such an implementation.

Although in the early parts of these notes we give attention to the details of recursion, learning how to *think like a functional programmer* requires us to think about large-scale transformations of collections of data.

15.5 Generalizing Function Definitions

Whenever we recognize a computational pattern in a set of related functions, we can *generalize the function* definition as follows:

1. Do a *scope-commonality-variability (SCV) analysis* on the set of related functions [37].

   That is, identify what is to be included and what not (i.e., the *scope*), the parts of functions that are the same (i.e., the *commonalities* or *frozen spots*), and the parts that differ (the *variabilities* or *hot spots*).

2. Leave the commonalities in the generalized function's body.

3. Move the variabilities into the generalized function's header—its type signature and parameter list.

   - If the part moved to the generalized function's parameter list is an expression, then make that part a function with a parameter for each local variable accessed.

   - If a data type potentially differs from a specific type used in the set of related functions, then add a type parameter to the generalized function.

   - If the same data value or type appears in multiple roles, then consider adding distinct type or value parameters for each role.

4. Consider other approaches if the generalized function's type signature and parameter list become too complex.

   For example, we can introduce new data or procedural abstractions for parts of the generalized function. These may be in the same module of the generalized function or in an appropriately defined separate module.

15.6 Defining `filter`

Consider the following two functions.

The first, `getEven`, takes a list of integers and returns the list of those integers that are even (i.e., are multiples of 2). The function preserves the relative order of the elements in the list.

```
getEven :: [Int] -> [Int]
getEven []       = []
getEven (x:xs)
```

```
    | even x      = x : getEven xs
    | otherwise = getEven xs
```

The second, doublePos, takes a list of integers and returns the list of doubles of the positive integers from the input list; it preserves the relative order of the elements.

```
doublePos :: [Int] -> [Int]
doublePos []          = []
doublePos (x:xs)
    | 0 < x       = (2 * x) : doublePos xs
    | otherwise = doublePos xs
```

Function even is from the Prelude; it returns True if its argument is evenly divisible by 2 and returns False otherwise.

What do these two functions have in common? What differs?

- Both take a list of integers and return a (possibly shorter) list of integers.

  However, the fact they use integers is not important; the key fact is that they take and return lists of the same element type.

- Both return an empty list when its input list is empty.

- In both, the relative orders of elements in the output list is the same as in the input list.

- Both select some elements to copy to the output and others not to copy.

  Function getEven selects elements that are even numbers and function doublePos selects elements that are positive numbers.

- Function doublePos doubles the value copied and getEven leaves the value unchanged.

Using the generalization method outlined above, we abstract the pattern of computation common to getEven and doublePos as the (broadly useful) function filter found in the Prelude. (We call the function filter' below to avoid a name conflict.)

```
filter' :: (a -> Bool) -> [a] -> [a]   -- filter in Prelude
filter' _ []     = []
filter' p (x:xs)
    | p x        = x : xs'
    | otherwise = xs'
                  where xs' = filter' p xs
```

Function filter takes a predicate p of type a -> Bool and a list of type [a] and returns a list containing those elements that satisfy p, in the same order as the input list. Note that the keyword where begins in the same column as the = in the defining equations; thus the scope of the definition of xs' extends over *both* legs of the definition.

Function `filter` does not incorporate the doubling operation from `doublePos`. We could have included it as another higher-order parameter, but we leave it out to keep the generalized function simple. We can use the already defined `map` function to achieve this separately.

Therefore, we can specialize `filter` to give new definitions of `getEven` and `doublePos` as follows:

```haskell
getEven2 :: [Int] -> [Int]
getEven2 xs = filter' even xs

doublePos2 :: [Int] -> [Int]
doublePos2 xs = map' dbl (filter' pos xs)
                where dbl x  = 2 * x
                      pos x = (0 < x)
```

Note that function `doublePos2` exhibits both the `filter` and the `map` patterns of computation.

The standard higher-order functions `map` and `filter` allow us to restate the three-leg definitions of `getEven` and `doublePos` in just one leg each, except that `doublePos` requires two lines of local definitions. In Chapter 16, we see how to eliminate these simple local definitions as well.

- Under what circumstances does `filter' p xs` terminate? Do we have to assume anything about `p`? about `xs`?
- What is the time complexity of `filter' p xs`? space complexity?
- What is the time complexity of `getEven2 xs`? space complexity?
- What is the time complexity of `doublePos2 xs`? space complexity?

15.7 Defining Fold Right (`foldr`)

Consider the `sum` and `product {.haskell}` functions we defined in Chapter 4, ignoring the short-cut handling of the zero element in `product`.

```haskell
sum' :: [Int] -> Int              -- sum in Prelude
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer -- product in Prelude
product' []     = 1
product' (x:xs) = x * product' xs
```

Both `sum'` and `product'` apply arithmetic operations to integers. What about other operations with similar pattern of computation?

Also consider a function `concat` that concatenates a list of lists of some type into a list of that type with the order of the input lists and their elements preserved.

```haskell
concat' :: [[a]] -> [a]   -- concat in Prelude
concat' []       =  []
concat' (xs:xss) =  xs ++ concat' xss
```

For example,

```haskell
sum' [1,2,3]           = (1 + (2 + (3 + 0)))
product' [1,2,3]       = (1 * (2 * (3 * 1)))
concat' ["1","2","3"] = ("1" ++ ("2" ++ ("3" ++ "")))
```

What do `sum'`, `product'`, and `concat'` have in common? What differs?

All exhibit the same pattern of computation.

- All take a list.

  But the element type differs. Function `sum'` takes a list of `Int` values, `product'` takes a list of `Integer` values, and `concat'` takes a polymorphic list.

- All insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value.

  But the binary operation differs. Function `sum'` applies integer addition, `product'` applies integer multiplication, and `concat'` applies ++.

- All group the operations from the right to the left.

- Each function returns some value for an empty list. The function extends nonempty input lists to implicitly include this value as the "rightmost" value of the input list.

  But the actual value differs.

  Function `sum'` returns integer 0, the (right) identity element for addition.

  Function `product'` returns 1, the (right) identity element for multiplication.

  Function `concat'` returns `[]`, the (right) identity element for ++.

  In general, this value could be something other than the identity element.

- All return a value of the same element type as the input list.

  But the input type differs, as we noted above.

This group of functions inserts operations of type `a -> a -> a` between elements a list of type `[a]`.

But these are special cases of more general operations of type `a -> b -> b`. In this case, the value returned must be of type `b` in the case of both empty and nonempty lists.

We can abstract the pattern of computation common to `sum'`, `product'`, and `concat'` as the function `foldr` (pronounced "fold right") found in the Prelude. (Here we use `foldrX`{.haskell} to avoid the name conflict.)

```
foldrX :: (a -> b -> b) -> b -> [a] -> b  -- foldr in Prelude
foldrX f z []     = z
foldrX f z (x:xs) = f x (foldrX f z xs)
```

Function `foldr`:

- uses two type parameters `a` and `b`—one for the type of elements in the list and one for the type of the result
- passes in the general binary operation `f` (with type `a -> b -> b`) that combines (i.e., folds) the list elements
- passes in the "seed" element `z` (of type `b`) to be returned for empty lists

The `foldr` function "folds" the list elements (of type `a`) into a value (of type `b`) by "inserting" operation `f` between the elements, with value `z` "appended" as the rightmost element.

Often the seed value `z` is the right identity element for the operation, but `foldr` may be useful in some circumstances where it is not (or perhaps even if there is no right identity).

For example, `foldr f z [1,2,3]` expands to `f 1 (f 2 (f 3 z))`, or, using an infix style:

```
1 `f` (2 `f` (3 `f` z))
```

Function `foldr` does not depend upon `f` being associative or having either a right or left identity.

Function `foldr` is backward recursive. If the function application is fully evaluated, it needs a new stack frame for each element of the input list. If its list argument is long or the folding function itself is expensive, then the function can terminate with a *stack overflow* error.

In Haskell, `foldr` is called a *fold* operation. Other languages sometimes call this a *reduce* or *insert* operation.

We can specialize `foldr` to restate the definitions for `sum'`, `product'`, and `concat'`.

```
sum2 :: [Int] -> Int              -- sum
sum2 xs = foldrX (+) 0 xs

product2 :: [Int] -> Int          -- product
product2 xs = foldrX (*) 1 xs

concat2:: [[a]] -> [a]            -- concat
concat2 xss = foldrX (++) [] xss
```

As further examples, consider the folding of the Boolean operators `&&` ("and") and `||` ("or") over lists of Boolean values as Prelude functions `and` and `or` (shown as `and'` and `or'` below to avoid name conflicts):

```haskell
and', or' :: [Bool] -> Bool  -- and, or in Prelude
and' xs = foldrX (&&) True xs
or'  xs = foldrX (||) False xs
```

Although their definitions look different, `and'` and `or'` are actually identical to functions `and` and `or` in the Prelude.

Consider the following questions.

- Under what circumstances does `foldrX f z xs` terminate? Do we have to assume anything about `f`? about `xs`?
- What is the time complexity of `product2`? of `concat2`?

15.8 Using `foldr`

The fold functions are very powerful. By choosing an appropriate folding function argument, many different list functions can be implemented in terms of `foldr`.

For example, we can implement `map` using `foldr` as follows:

```haskell
map2 :: (a -> b) -> [a] -> [b] -- map
map2 f xs = foldr mf [] xs
    where mf y ys = (f y) : ys
```

The folding function `mf y ys = (f y):ys` applies the mapping function `f` to the next element of the list (moving right to left) and attaches the result on the front of the processed tail. This is a case where the folding function `mf` does not have a right identity, but where `foldr` is quite useful.

We can also implement `filter` in terms of `foldr` as follows:

```haskell
filter2 :: (a -> Bool) -> [a] -> [a]  -- filter
filter2 p xs = foldr ff [] xs
    where ff y ys = if p y then (y:ys) else ys
```

The folding function `ff y ys = if p x then (y:ys) else ys` applies the filter predicate `p` to the next element of the list (moving right to left). If the predicate evaluates to `True`, the folding function attaches that element on the front of the processed tail; otherwise, it omits the element from the result.

We can also use `foldr` to compute the length of a polymorphic list.

```haskell
length2 :: [a] -> Int  -- length
length2 xs  = foldr len  0  xs
    where len _ acc = acc + 1
```

This uses the `z` parameter of `foldr` to initialize the count to 0. Higher-order argument `f` of `foldr` is a function that takes an element of the list as its left argument and the previous accumulator as its right argument and returns the accumulator incremented by 1. In this application, `z` is not the identity element for `f` but is a convenient beginning value for the counter.

We can construct an "append" function that uses `foldr` as follows:

```
append2 :: [a] -> [a] -> [a]   -- ++
append2 xs ys = foldr (:) ys xs
```

Here the the list that `foldr` operates on the first argument of the append. The `z` parameter is the entire second argument and the folding function is just `(:)`. So the effect is to replace the `[]` at the end of the first list by the entire second list.

Function `foldr` 1s a backward recursive function that processes the elements of a list one by one. However, as we have seen, it is often more useful to think of `foldr` as a powerful list operator that reduces the element of the list into a single value. We can combine `foldr` with other operators to conveniently construct list processing programs.

15.9 Defining Fold Left (`foldl`)

We designed function `foldr` as a backward linear recursive function with the signature:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

As noted:

```
foldr f z [1,2,3] ==  f 1 (f 2 (f 3 z))
                  ==  1 `f` (2 `f` (3 `f` z))
```

Consider a function `foldl` (pronounced "fold left") such that:

```
foldl f z [1,2,3]  ==  f (f (f z 1) 2) 3
                   ==  ((z `f` 1) `f` 2) `f` 3`
```

This function folds from the left. It offers us the opportunity to use parameter `z` as an accumulating parameter in a tail recursive implementation. This is shown below as `foldlX`, which is similar to `foldl` in the Prelude.

```
foldlX :: (a -> b -> a) -> a -> [b] -> a  -- foldl in Prelude
foldlX f z []     = z
foldlX f z (x:xs) = foldlX f (f z x) xs
```

] Note how the second leg of `foldlX` implements the left binding of the operation. In the recursive call of `foldlX` the "seed value" argument is used as an accumulating parameter.

Also note how the types of `foldr` and `foldl` differ.

Often the beginning value of `z` is the left identity of the operation `f`, but `foldl` (like `foldr`) can be a quite useful function in circumstances when it is not (or when `f` has no left identity).

15.10 Using foldl

If $\oplus$ is an associative binary operation of type `t -> t -> t` with identity element `z` (i.e., $\oplus$ and `t` form the algebraic structure know as a monoid), then, for any `xs`,

```
foldr ( ⊕) z xs = foldl ( ⊕) z xs
```

The classic Bird and Wadler textbook [15] calls this property the *first duality theorem*.

Because +, *, and ++ are all associative operations with identity elements, `sum`, `product`, and `concat` can all be implemented with either `foldr` or `foldl`.

Which is better?

Depending upon the nature of the operation, an implementation using `foldr` may be more efficient than `foldl` or vice versa.

We defer a more complete discussion of the efficiency until we study evaluation strategies further in Chapter 29.

As a rule of thumb, however, if the operation $\oplus$ is *nonstrict* in either argument, then it is usually better to use `foldr`. That form takes better advantage of lazy evaluation.

If the operation $\oplus$ is *strict* in both arguments, then it is often better (i.e., more efficient) to use the optimized version of `foldl` called `foldl'` from the standard Haskell module `Data.List`.

The append operation ++ is nonstrict in its second argument, so it is better to use `foldr` to implement `concat`.

Addition and multiplication are strict in both arguments, so we can implement `sum` and `product` functions efficiently with `foldl'`, as follows:

```
import Data.List   -- to make foldl' available
sum3, product3 :: Num a =>  [a] -> a -- sum, product
sum3 xs     = foldl' (+) 0 xs
product3 xs = foldl' (*) 1 xs
```

Note that we generalize these functions to operate on polymorphic lists with a base type in class `Num`. Class `Num` includes all numeric types.

Function `length3` uses `foldl`. It is like `length2` except that the arguments of function `len` are reversed.

```
length3 :: [a] -> Int   -- length
length3 xs   = foldl len 0  xs
    where len acc _ = acc + 1
```

However, it is usually better to use the `foldr` version `length2` because the folding function `len` is nonstrict in the argument corresponding to the list.

We can also implement list reversal using `foldl` as follows:

```
reverse2 :: [a] -> [a]  -- reverse
reverse2 xs = foldl rev [] xs
    where rev acc x = (x:acc)
```

This gives a solution similar to the tail recursive `reverse` function from Chapter 14. The `z` parameter of function `foldl` is initially an empty list; the folding function parameter `f` of `foldl` uses `(:)` to "attach" each element of the list as the new head of the accumulator, incrementally building the list in reverse order.

Although cons is nonstrict in its right operand, `reverse2` builds up that argument from `[]`, so `reverse2` cannot take advantage of lazy evaluation by using `foldr` instead.

To avoid a stack overflow situation with `foldr`, we can first apply `reverse` to the list argument and then apply `foldl` as follows:

```
foldr2 :: (a -> b -> b) -> b -> [a] -> b  -- foldr
foldr2 f z xs = foldl flipf z (reverse xs)
    where flipf y x = f x y
```

The combining function in the call to `foldl` is the same as the one passed to `foldr` except that its arguments are reversed.

15.11 Defining `concatMap` (flatmap)

The higher-order function `map` applies its function argument `f` to every element of a list and returns the list of results. If the function argument `f` returns a list, then the result is a list of lists. Often we wish to flatten this into a single list, that is, apply a function like `concat` defined in Section 15.7.

This computation is sufficiently common that we give it the name `concatMap`. We can define it in terms of `map` and `concat` as

```
concatMap' :: (a -> [b]) -> [a] -> [b]
concatMap' f xs = concat (map f xs)
```

or by combining `map` and `concat` into one `foldr` as:

```
concatMap2 :: (a -> [b]) -> [a] -> [b]
concatMap2 f xs = foldr fmf [] xs
    where fmf x ys = f x ++ ys
```

Above, the function argument to `foldr` applies the `concatMap` function argument `f` to each element of the list argument and then appends the resulting list in front of the result from processing the elements to the right.

We can also define `filter` in terms of `concatMap` as follows:

```
filter3 :: (a -> Bool) -> [a] -> [a]
filter3 p xs = concatMap' fmf xs
```

```haskell
        where fmf x = if p x then [x] else []
```

The function argument to `concatMap` generates a one-element list if the filter predicate `p` is true and an empty list if it is false.

Some other languages (e.g., Scala) call the `concatMap` function by the name `flatmap`.

15.12 What Next?

This chapter introduced the concepts of first-class and higher-order functions and generalized common computational patterns to construct a library of useful higher-order functions to process lists.

Chapter 16 continues to examine those concepts and their implications for Haskell programming.

15.13 Chapter Source Code

The Haskell module for this chapter is in file `HigherOrderFunctions.hs`.

15.14 Exercises

1. Suppose you need a Haskell function `times` that takes a list of integers (type `Integer`) and returns the product of the elements (e.g., `times [2,3,4]` returns `24`). Define the following Haskell functions.

    a. Function `times1` that uses the Prelude function `foldr` (or `foldr'` from this chapter).

    b. Function `times2` that uses backward recursion to compute the product. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `product`.)

    c. Function `times3` that uses forward recursion to compute the product. (Hint: use a tail-recursive auxiliary function with an accumulating parameter.)

    d. Function `times4` that uses function `foldl'` from the Haskell library `Data.List`.

2. For each of the following specifications, define a Haskell function that has the given arguments and result. Use the higher order library functions (from this chapter) such as `map`, `filter`, `foldr`, and `foldl` as appropriate.

    a. Function `numof` takes a value and a list and returns the number of occurrences of the value in the list.

    b. Function `ellen` takes a list of character strings and returns a list of the lengths of the corresponding strings.

c. Function `ssp` takes a list of integers and returns the sum of the squares of the positive elements of the list.

3. Suppose you need a Haskell function `sumSqNeg` that takes a list of integers (type `Integer`) and returns the sum of the squares of the negative values in the list.

   Define the following Haskell functions. Use the higher order library functions (from this chapter) such as `map`, `filter`, `foldr`, and `foldl` as appropriate.

   a. Function `sumSqNeg1` that is backward recursive. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `sum`.)

   b. Function `sumSqNeg2` that is tail recursive. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `sum`.)

   c. Function `sumSqNeg3` that uses standard prelude functions such as `map`, `filter`, `foldr`, and `foldl`.

   d. Function `sumSqNeg4` that uses list comprehensions (Chapter 18).

4. Define a Haskell function

```haskell
scalarprod :: [Int] -> [Int] -> Int
```

   to compute the scalar product of two lists of integers (e.g., representing vectors).

   The *scalar product* is the sum of the products of the elements in corresponding positions in the lists. That is, the scalar product of two lists `xs` and `ys`, of length `n`, is:

$$\sum_{i=0}^{i=n} xs_i * ys_i$$

   For example, `scalarprod [1,2,3] [3,3,3]` yields `18`.

5. Define a Haskell function `map2` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

15.15 Acknowledgemen4

In Summer 2016, I adapted and revised much of this work from the following sources:

- Chapter 6 of my *Notes on Functional Programming with Haskell* [42] which is influenced by Bird [13–15] and Wentworth [178]

- My notes on *Functional Data Structures (Scala)* [50], which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [29] and its associated materials [30,31]

In 2017, I continued to develop this work as Chapter 5, Higher-Order Functions, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Higher-Order Functions chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 5.1-5.2 became the basis for new Chapter 15 (this chapter), Higher-Order Functions, section 5.3 became the basis for new Chapter 16, Haskell Function Concepts, and previous sections 5.4-5.6 became the basis for new Chapter 17, Higher-Order Function Examples.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

15.16 Terms and Concepts

Procedural abstraction, functions (first-class, higher-order), modularity, interface, function generalization and specialization, scope-commonality-variability (SCV) analysis, hot and frozen spots, data transformations, think like a functional programmer, common functional programming patterns (map, filter, fold, concatMap), duality theorem, strict and nonstrict functions.

16 Haskell Function Concepts

16.1 Chapter Introduction

Chapter 15 introduced the concepts of first-class and higher-order functions and generalized common computational patterns to construct a library of useful higher-order functions to process lists.

This chapter continues to examine those concepts and their implications for Haskell programming. It explores strictness, currying, partial application, combinators, operator sections, functional composition, inline function definitions, evaluation strategies, and related methods.

The Haskell module for Chapter 16 is in file `FunctionConcepts.hs`.

16.2 Strictness

In the discussion of the fold functions, Chapter 15 introduced the concept of strictness. In this section, we explore that in more depth.

Some expressions cannot be reduced to a simple value, for example, `div 1 0`. The attempted evaluation of such expressions either return an error immediately or cause the interpreter to go into an "infinite loop".

In our discussions of functions, it is often convenient to assign the symbol $\bot$ (pronounced "bottom") as the value of expressions like `div 1 0`. We use $\bot$ is a polymorphic symbol—as a value of every type.

The symbol $\bot$ is not in the Haskell syntax and the interpreter cannot actually generate the value $\bot$. It is merely a name for the value of an expression in situations where the expression cannot really be evaluated. It's use is somewhat analogous to use of symbols such as ∞ in mathematics.

Although we cannot actually produce the value $\bot$, we can, conceptually at least, apply any function to $\bot$.

If `f` $\bot = \bot$, then we say that the function is *strict*; otherwise, it is *nonstrict* (sometimes called *lenient*).

That is, a strict argument of a function must be evaluated before the final result can be computed. A nonstrict argument of a function may not need to be evaluated to compute the final result.

Assume that lazy evaluation is being used and consider the function `two` that takes an argument of any type and returns the integer value two.

```
two :: a -> Int
two x = 2
```

The function `two` is nonstrict. The argument expression is not evaluated to compute the final result. Hence, `two` $\bot = 2$.

Consider the following examples.

- The arithmetic operations (e.g., +) are strict in both arguments.
- Function `rev` (discussed in Chapter 14) is strict in its one argument.
- Operation ++ is strict in its first argument, but nonstrict in its second argument.
- Boolean functions && and || from the Prelude are also strict in their first arguments and nonstrict in their second arguments.

```
(&&), (||) :: Bool -> Bool -> Bool
False && x = False  -- second argument not evaluated
True  && x = x

False || x = x
True  || x = True   -- second argument not evaluated
```

16.3 Currying and Partial Application

Consider the following two functions:

```
add :: (Int,Int) -> Int
add (x,y) = x + y

add' :: Int -> (Int -> Int)
add' x y  = x + y
```

These functions are closely related, but they are not identical.

For all integers `x` and `y`, `add (x,y) == add' x y`. But functions `add` and `add'` have different types.

Function `add` takes a 2-tuple `(Int,Int)` and returns an `Int`. Function `add'` takes an `Int` and returns a function of type `Int -> Int`.

What is the result of the application `add 3`? An error.

What is the result of the application `add' 3`? The result is a function that "adds 3 to its argument".

What is the result of the application `(add' 3) 4`? The result is the integer value `7`.

By convention, function application (denoted by the juxtaposition of a function and its argument) binds to the left. That is, `add' x y = ((add' x) y)`.

Hence, the higher-order functions in Haskell allow us to replace any function that takes a tuple argument by an equivalent function that takes a sequence of simple arguments corresponding to the components of the tuple. This process is called *currying*. It is named after American logician Haskell B. Curry, who first exploited the technique.

Function `add'` above is similar to the function `(+)` from the Prelude (i.e., the addition operator).

We sometimes speak of the function `(+)` as being *partially applied* in the expression `((+) 3)`. In this expression, the first argument of the function is "frozen in" and the resulting function can be passed as an argument, returned as a result, or applied to another argument.

Partially applied functions are very useful in conjunction with other higher-order functions.

For example, consider the partial applications of the relational comparison operator `(<)` and multiplication operator `(*)` in the function `doublePos3`. This function, which is equivalent to the function `doublePos` discussed in Chapter 15, doubles the positive integers in a list:

```
doublePos3 :: [Int] -> [Int]
doublePos3 xs = map ((*) 2) (filter ((<) 0) xs)
```

Related to the concept of currying is the *property of extensionality*. Two functions `f` and `g` are extensionally equal if `f x == g x` for all `x`.

Thus instead of writing the definition of `g` as

```
f, g :: a -> a
f x = some_expression

g x = f x
```

we can write the definition of `g` as simply:

```
g = f
```

16.4 Operator Sections

Expressions such as `((*) 2)` and `((<) 0)`, used in the definition of `doublePos3` in Section 16.3, can be a bit confusing because we normally use these operators in infix form. (In particular, it is difficult to remember that `((<) 0)` returns `True` for positive integers.)

Also, it would be helpful to be able to use the division operator to express a function that halves (i.e., divides by two) its operand. The function `((/) 2)` does not do it; it divides 2 by its operand.

We can use the function `flip` from the Prelude to state the halving operation. Function `flip` takes a function and two additional arguments and applies the argument function to the two arguments with their order reversed.

```
flip' :: (a -> b -> c) -> b -> a -> c  -- flip in Prelude
flip' f x y = f y x
```

Thus we can express the halving operator with the expression `(flip (/) 2)`.

Because expressions such as `((<) 0)` and `(flip (/) 2)` are quite common in programs, Haskell provides a special, more compact and less confusing, syntax.

For some infix operator ⊕ and arbitrary expression `e`, expressions of the form `(e ⊕)` and `( ⊕e)` represent `(( ⊕) e)` and `(flip ( ⊕) e)`, respectively. Expressions of this form are called *operator sections*.

Examples of operator sections include:

> `(1+)` is the successor function, which returns the value of its argument plus 1.
>
> `(0<)` is a test for a positive integer.
>
> `(/2)` is the halving function.
>
> `(1.0/)` is the reciprocal function.
>
> `(:[])` is the function that returns the singleton list containing the argument.

Suppose we want to sum the cubes of list of integers. We can express the function in the following way:

```
sumCubes :: [Int] -> Int
sumCubes xs = sum (map (^3) xs)
```

Above `^` is the exponentiation operator and `sum` is the list summation function defined in the Prelude as:

```
sum = foldl' (+) 0  -- sum
```

16.5 Combinators

The function `flip` in Section 16.4 is an example of a useful type of function called a combinator.

A *combinator* is a function without any free variables. That is, on right side of a defining equation there are no variables or operator symbols that are not bound on the left side of the equation.

For historical reasons, `flip` is sometimes called the `C` combinator.

There are several other useful combinators in the Prelude.

The combinator `const` (shown below as `const’`) is the constant function constructor; it is a two-argument function that returns its first argument. For historical reasons, this combinator is sometimes called the `K` combinator.

```
const' :: a -> b -> a  -- const in Prelude
const' k x = k
```

Example: `(const 1)` takes any argument and returns the value 1.

Question: What does `sum (map (const 1) xs)` do?

Function `id` (shown below as `id’`) is the identity function; it is a one-argument function that returns its argument unmodified. For historical reasons, this function is sometimes called the `I` combinator.

```
id' :: a -> a  -- id in Prelude
id' x = x
```

Combinators `fst` and `snd` (shown below as `fst’` and `snd’`) extract the first and second components, respectively, of 2-tuples.

```
fst' :: (a,b) -> a  -- fst in Prelude
fst' (x,_) = x

snd' :: (a,b) -> b  -- snd in Prelude
snd' (_,y) = y
```

Similarly, `fst3`, `snd3`, and `thd3` extract the first, second, and third components, respectively, of 3-tuples.

TODO: Correct above statement. No longer seems correct. `Data.Tuple.Select` `sel1`, `sel2`, `sel2`, etc. Investigate and rewrite.

An interesting example that uses a combinator is the function `reverse` as defined in the Prelude (shown below as `reverse’`):

```
reverse' :: [a] -> [a]          -- reverse in Prelude
reverse' = foldlX (flip' (:)) []
```

Function `flip (:)` takes a list on the left and an element on the right. As this operation is folded through the list from the left it attaches each element as the new head of the list.

We can also define combinators that convert an uncurried function into a curried function and vice versa. The functions `curry'` and `uncurry'` defined below are similar to the Prelude functions.

```
curry' :: ((a, b) -> c) -> a -> b -> c     --Prelude curry
curry' f x y =  f (x, y)

uncurry' :: (a -> b -> c) -> ((a, b) -> c) --Prelude uncurry
uncurry' f p =  f (fst p) (snd p)
```

Two other useful combinators are `fork` and `cross` [Bird 2015]. Combinator `fork` applies each component of a pair of functions to a value to create a pair of results. Combinator `cross` applies each component of a pair of functions to the corresponding components of a pair of values to create a pair of results. We can define these as follows:

```
fork :: (a -> b, a -> c) -> a -> (b,c)
fork (f,g) x = (f x, g x)
```

```haskell
cross :: (a -> b, c -> d) -> (a,c) -> (b,d)
cross (f,g) (x,y) = (f x, g y)
```

16.6 Functional Composition

The functional composition operator allows several "smaller" functions to be combined to form "larger" functions. In Haskell, this combinator is denoted by the period (`.`) symbol and is defined in the Prelude as follows:

```haskell
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Composition's default binding is from the right and its precedence is higher than all the operators we have discussed so far except function application itself.

Functional composition is an associative binary operation with the identity function `id` as its identity element:

```haskell
 f . (g . h) = (f . g) . h
 id . f   = f . id
```

16.7 Function Pipelines

As an example, consider the function `count` that takes two arguments, an integer `n` and a list of lists, and returns the number of the lists from the second argument that are of length `n`. Note that all functions composed below are single-argument functions: `length`, `(filter (== n))`, `(map length)`.

```haskell
count :: Int -> [[a]] -> Int
count n  -- unprimed versions from Prelude
    | n >= 0    = length . filter (== n) . map length
    | otherwise = const 0   -- discard 2nd arg, return 0
```

We can think of the point-free expression `length . filter (== n) . map length` as defining a *function pipeline* through which data flows from right to left.

TODO: Draw a diagram showing the data flow network (right to left?)

1. The pipeline takes a polymorphic list of lists as input.
2. The `map length` component of the pipeline replaces each inner list by its length.
3. The `filter (== n)` component takes the list created by the previous step and removes all elements not equal to `n`.
4. The `length` component takes the list created by the previous step and determines how many elements are remaining.
5. The pipeline outputs the value computed by the previous component. The number of lists within the input list of lists that are of length `n`.

Thus composition is a powerful form of "glue" that can be used to "stick" simpler functions together to build more powerful functions. The simpler functions in this case include partial applications of higher order functions from the library we have developed.

As we see above in the definition of `count`, partial applications (e.g., `filter (== n)`), operator sections (e.g., `(== n)`), and combinators (e.g., `const`) are useful as *plumbing* the function pipeline.

Remember the function `doublePos` that we discussed in earlier sections.

```
doublePos3 xs = map ((*) 2) (filter ((<) 0) xs)
```

Using composition, partial application, and operator sections we can restate its definition in point-free style as follows:

```
doublePos4 :: [Int] -> [Int]
doublePos4 = map (2*) . filter (0<)
```

Consider a function `last` to return the last element in a non-nil list and a function `init` to return the initial segment of a non-nil list (i.e., everything except the last element). These could quickly and concisely be written as follows:

```
last' = head . reverse            -- last in Prelude
init' = reverse . tail . reverse -- init in Prelude
```

However, since these definitions are not very efficient, the Prelude implements functions `last` and `init` in a more direct and efficient way similar to the following:

```
last2 :: [a] -> a    -- last in Prelude
last2 [x]    = x
last2 (_:xs) = last2 xs

init2 :: [a] -> [a]  -- init in Prelude
init2 [x]    = []
init2 (x:xs) = x : init2 xs
```

The definitions for Prelude functions `any` and `all` are similar to the definitions show below; they take a predicate and a list and apply the predicate to each element of the list, returning `True` when any and all, respectively, of the individual tests evaluate to `True`.

```
any', all' :: (a -> Bool) -> [a] -> Bool
any' p = or' . map' p   -- any in Prelude
all' p = and' . map' p  -- all in Prelude
```

The functions `elem` and `notElem` test for an object being an element of a list and not an element, respectively. They are defined in the Prelude similarly to the following:

```haskell
elem', notElem' :: Eq a => a -> [a] -> Bool
elem'    = any . (==)  -- elem in Prelude
notElem' = all . (/=)  -- notElem in Prelude
```

These are a bit more difficult to understand since `any`, `all`, `==`, and /= are two-argument functions. Note that expression `elem x xs` would be evaluated as follows:

```
     elem’ x xs
⟹ { expand elem’ }
     (any’ . (==)) x xs
⟹ { expand composition }
     any’ ((==) x) xs
```

The composition operator binds the first argument with `(==)` to construct the first argument to `any’`. The second argument of `any’` is the second argument of `elem’`.

16.8 Lambda Expressions

Remember the function `squareAll2` we examined in Chapter 15:

```haskell
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x
```

We introduced the local function definition `sq` to denote the function to be passed to `map`. It seems to be a waste of effort to introduce a new symbol for a simple function that is only used in one place in an expression. Would it not be better, somehow, to just give the defining expression itself in the argument position?

Haskell provides a mechanism to do just that, an anonymous function definition. For historical reasons, these nameless functions are called *lambda expressions*. They begin with a backslash `\{.haskell}` and have the syntax:

`\` *atomicPatterns* `->` *expression*

For example, the squaring function (`sq`) could be replaced by a lambda expression as `(\x -> x * x)`. The pattern `x` represents the single argument for this anonymous function and the expression `x * x` is its result.

Thus we can rewrite `squareAll2` in point-free style using a lambda expression as follows:

```haskell
squareAll3 :: [Int] -> [Int]
squareAll3 = map' (\x -> x * x)
```

A lambda expression to average two numbers can be written `(\x y -> (x+y)/2)`.

An interesting example that uses a lambda expression is the function `length` as defined in the Prelude—similar to `length4` below. (Note that this uses the optimized function `foldl'` from the standard Haskell `Data.List` module.)

```
length4 :: [a] -> Int    -- length in Prelude
length4  = foldl' (\n _ -> n+1) 0
```

The anonymous function `(\n _ -> n+1)` takes an integer "counter" and a polymorphic value and returns the "counter" incremented by one. As this function is folded through the list from the left, this function counts each element of the second argument.

16.9 Application Operator $

In Haskell, function application associates to the left and has higher binding power than any infix operator. For example, for some function two-argument function `f` and values `w`, `x`, `y`, and `z`

```
w + f x y * z
```

is the same as

```
w + (((f x) y) * z)
```

given the relative binding powers of function application and the numeric operators.

However, sometimes we want to be able to use function application where it associates to the right and binds less tightly than any other operator. Haskell defines the `$` operator to enable this style, as follows:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Thus, for single argument functions `f`, `g`, and `h`,

```
f $ g $ h $ z + 7
```

is the same as

```
(f (g (h (z+7))))
```

and as:

```
(f . g . h) (z+7)
```

Similarly, for two-argument functions `f'`, `g'`, and `h'`,

```
f' w $ g' x $ h' y $ z + 7
```

is the same as

```
((f' w) ((g' x) ((h' y) (z+7))))
```

and as:

```
(f' w . g' x . h' y) (z+7)
```

For example, this operator allows us to write

```
foldr (+) 0 $ map (2*) $ filter odd $ enumFromTo 1 20
```

where Prelude function `enumFromTo m n` generates the sequence of integers from `m` to `n`, inclusive.

16.10 Eager Evaluation Using `seq` and `$!`

Haskell is a lazily evaluated language. That is, if an argument is nonstrict it may never be evaluated.

Sometimes, using the technique called *strictness analysis*, the Haskell compiler can detect that an argument's value will always be needed. The compiler can then safely force eager evaluation as an optimization without changing the meaning of the program.

In particular, by selecting the `-O` option to the Glasgow Haskell Compiler (GHC), we can enable GHC's code optimization processing. GHC will generally create smaller, faster object code at the expense of increased compilation time by taking advantage of strictness analysis and other optimizations.

However, sometimes we may want to force eager evaluation explicitly without invoking a full optimization on all the code (e.g., to make a particular function's evaluation more space efficient). Haskell provides the primitive function `seq` that enables this. That is,

```
seq :: a -> b -> b
x `seq` y = y
```

where it just returns the second argument except that, as a side effect, `x` is evaluated before `y` is returned. (Technically, `x` is evaluated to what is called *head normal form*. It is evaluated until the outer layer of structure such as `h:t` is revealed, but `h` and `t` themselves are not fully evaluated. We study evaluation strategies further in Chapter 29.

Function `foldl`, the "optimized" version of `foldl` can be defined using `seq` as follows

```
foldlP :: (a -> b -> a) -> a -> [b] -> a  -- Data.List.foldl'
foldlP f z []     = z
foldlP f z (x:xs) = y `seq` foldl' f y xs
                    where y = f z x
```

That is, this evaluates the `z` argument of the tail recursive application eagerly.

Using `seq`, Haskell also defines `$!`, a strict version of the `$` operator, as follows:

```haskell
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

The effect of `f $! x` is the same as `f $ x` except that `$!` eagerly evaluates the argument `x` before applying function `f` to it.

We can rewrite `foldl'` using `$!` as follows:

```haskell
foldlQ :: (a -> b -> a) -> a -> [b] -> a  -- Data.List.foldl'
foldlQ f z []     = z
foldlQ f z (x:xs) = (foldlQ f $! f z x) xs
```

We can write a tail recursive function to sum the elements of the list as follows:

```haskell
sum4 :: [Integer] -> Integer  -- sum in Prelude
sum4 xs = sumIter xs 0
    where sumIter []     acc = acc
          sumIter (x:xs) acc = sumIter xs (acc+x)
```

We can then redefine `sum4` to force eager evaluation of the accumulating parameter of `sumIter` as follows:

```haskell
sum5 :: [Integer] -> Integer -- sum in Prelude
sum5 xs = sumIter xs 0
    where sumIter []     acc = acc
          sumIter (x:xs) acc = sumIter xs $! acc + x
```

However, we need to be careful in applying `seq` and `$!`. They change the semantics of the lazily evaluated language in the case where the argument is nonstrict. They may force a program to terminate abnormally and/or cause it to take unnecessary evaluation steps.

16.11 What Next?

Chapter 15 introduced the concepts of first-class and higher-order functions and generalized common computational patterns to construct a library of useful higher-order functions to process lists.

This chapter (16}) continued to examine those concepts and their implications for Haskell programming by exploring concepts and features such as strictness, currying, partial application, combinators, operator sections, functional composition, inline function definitions, and evaluation strategies.

Chapter 17 looks at additional examples that use these higher-order programming concepts.

16.12 Chapter Source Code

The Haskell module for Chapter 16 is in file `FunctionConcepts.hs`.

16.13 Exercises

1. Define a Haskell function

   ```
   total :: (Integer -> Integer) -> Integer -> Integer
   ```

   so that `total f n` gives `f 0 + f 1 + f 2 + ... + f n`. How could you define it using `removeFirst`?

2. Define a Haskell function `map2` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

3. Define a Haskell function `fmap` that takes a value and a list of functions and returns the list of results from applying each function to the argument value. (For example, `fmap 3 [((*) 2), ((+) 2)]` yields `[6,5]`.)

4. Define a Haskell function `composeList` that takes a list of functions and composes them into a single function. (Be sure to give the type signature.)

16.14 Acknowledgements

In Summer 2016, I adapted and revised much of this work from the following sources:

- Chapter 6 of my *Notes on Functional Programming with Haskell* [42] which is influenced by Bird [13–15] and Wentworth [178]
- My notes on *Functional Data Structures (Scala)* [50], which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [29] and its associated materials [30,31]

In Summer 2016, I also added the following,a drawing on ideas from [14, Ch. 6, 7] and [173, Ch. 11]:

- expanded discussion of combinators and functional composition
- new discussion of the `seq`, `$`, and `$!` operators

In 2017, I continued to develop this work as Chapter 5, Higher-Order Functions, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Higher-Order Functions chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 5.1-5.2 became the basis for new Chapter 15, Higher-Order Functions, section 5.3 became the basis for new Chapter 16, Haskell Function Concepts (this chapter), and previous sections 5.4-5.6 became the basis for new Chapter 17, Higher-Order Function Examples.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials,

reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

16.15 Terms and Concepts

Strict and nonstrict functions, bottom, strictness analysis, currying, partial application, operator sections, combinators, functional composition, property of extensionality, pointful and point-free styles, plumbing, function pipeline, lambda expression, application operator `$`, eager evaluation operators `seq` and `$!`, head-normal form.

17 Higher Order Function Examples

17.1 Chapter Introduction

Chapters 15 and 16 introduced the concepts of first-class and higher-order functions and their implications for Haskell programming.

The goals of this chapter (17) are to:

- continue to explore first-class and higher-order functions by examining additional library functions and examples
- examine how to express general problem-solving strategies as higher-order functions, in particular the divide-and-conquer strategy

17.2 List-Breaking Operations

In Chapter 13, we looked at the list-breaking functions `take` and `drop`. The Prelude also includes several higher-order list-breaking functions that take two arguments, a predicate that determines where the list is to be broken and the list to be broken.

Here we look at Prelude functions `takeWhile` and `dropWhile`. As you would expect, function `takeWhile` "takes" elements from the beginning of the list "while" the elements satisfy the predicate and `dropWhile` "drops" elements from the beginning of the list "while" the elements satisfy the predicate. The Prelude definitions are similar to the following:

```
takeWhile':: (a -> Bool) -> [a] -> [a] -- takeWhile in Prelude
takeWhile'  p []   = []
takeWhile'  p (x:xs)
    | p x        = x : takeWhile' p xs
    | otherwise  = []

dropWhile' :: (a -> Bool) -> [a] -> [a] -- dropWhile in Prelude
dropWhile'  p []   = []
dropWhile'  p xs@(x:xs')
    | p x        = dropWhile' p xs'
    | otherwise  = xs
```

Note the use of the pattern `xs@(x:xs')` in `dropWhile'`. This pattern matches a non-nil list with `x` and `xs'` binding to the head and tail, respectively, as usual. Variable `xs` binds to the entire list.

As an example, suppose we want to remove the leading blanks from a string. We can do that with the expression:

```
dropWhile ((==) ' ')
```

As with `take` and `drop`, the above functions can also be related by a "law". For all finite lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs = xs
```

Prelude function `span` combines the functionality of `takeWhile` and `dropWhile` into one function. It takes a predicate `p` and a list `xs` and returns a tuple where the first element is the longest prefix (possibly empty) of `xs` that satisfies `p` and the second element is the remainder of the list.

```
span' :: (a -> Bool) -> [a] -> ([a],[a]) -- span in Prelude
span' _ xs@[]       =  (xs, xs)
span' p xs@(x:xs')
    | p x           =  let (ys,zs) = span' p xs' in (x:ys,zs)
    | otherwise     =  ([],xs)
```

Thus the following "law" holds for all finite lists `xs` and predicates `p` on same type:

```
span p xs == (takeWhile p xs, dropWhile p xs)
```

The Prelude also includes the function `break`, defined as follows:

```
break' :: (a -> Bool) -> [a] -> ([a],[a]) -- break in Prelude
break' p =  span (not . p)
```

17.3 List-Combining operations

In Chapter 14, we also looked at the function `zip`, which takes two lists and returns a list of pairs of the corresponding elements. Function `zip` applies an operation, in this case *tuple-construction*, to the corresponding elements of two lists.

We can generalize this pattern of computation with the function `zipWith` in which the operation is an argument to the function.

```
zipWith' :: (a->b->c) -> [a]->[b]->[c] -- zipWith in Prelude
zipWith' z  (x:xs)  (y:ys) = z x y : zipWith' z xs ys
zipWith' _  _       _      = []
```

Using a lambda expression to state the tuple-forming operation, the Prelude defines `zip` in terms of `zipWith`:

```
zip'' :: [a] -> [b] -> [(a,b)]  -- zip
zip'' = zipWith' (\x y -> (x,y))
```

Or it can be written more simply as:

```
zip''' :: [a] -> [b] -> [(a,b)]  -- zip
zip''' = zipWith' (,)
```

The `zipWith` function also enables us to define operations such as the scalar product of two vectors in a concise way.

```
sp :: Num a => [a] -> [a] -> a
sp xs ys = sum' (zipWith' (*) xs ys)
```

The Prelude includes `zipWith3` for triples. Library `Data.List` has versions of `zipWith` that take up to seven input lists: `zipWith3` ⋯ `zipWith7`.

17.4 Rational Arithmetic Revisited

Remember the rational number arithmetic package developed in Chapter 7. In that package's `Rational` module, we defined a function `eqRat` to compare two rational numbers for equality using the appropriate set of integer comparisons.

```
eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

We could have implemented the other comparison operations similarly.

Because the comparison operations are similar, they are good candidates for us to use a higher-order function. We can abstract out the common pattern of comparisons into a function that takes the corresponding integer comparison as an argument.

To compare two rational numbers, we can express their values in terms of a common denominator (e.g., `denom x * denom y`) and then compare the numerators using the integer comparisons. We can thus abstract the comparison into a higher-order function `compareRat` that takes an appropriate integer relational operator and the two rational numbers.

```
compareRat :: (Int -> Int -> Bool) -> Rat -> Rat -> Bool
compareRat r x y = r (numer x * denom y) (denom x * numer y)
```

Then we can define the rational number comparisons in terms of `compareRat`. (Note that we redefine function `eqRat` from the package Chapter 7.)

```
eqRat,neqRat,ltRat,leqRat,gtRat,geqRat :: Rat -> Rat -> Bool
eqRat   = compareRat (==)
neqRat  = compareRat (/=)
ltRat   = compareRat (<)
leqRat  = compareRat (<=)
gtRat   = compareRat (>)
geqRat  = compareRat (>=)
```

The Haskell module for the revised rational arithmetic module is in `RationalHO.hs`. The module `TestRationalHO.hs` is an extended version of the standard test script from Chapter 12 that tests the standard features of the rational arithmetic module plus `eqRat`, `neqRat`, and `ltRat`. (It does not currently test `leqRat`, `gtRat`, or `geqRat`.)

17.5 Mergesort

We defined the insertion sort in Chapter 14. It has an average-case time complexity of O(`n^2`) where `n` is the length of the input list.

We now consider a more efficient function to sort the elements of a list into ascending order: *mergesort*. Mergesort works as follows:

- If the list has fewer than two elements, then it is already sorted.
- If the list has two or more elements, then we split it into two sublists, each with about half the elements, and sort each recursively.
- We merge the two ascending sublists into an ascending list.

We define function `msort` to be a polymorphic, higher-order function that has two parameters. The first (`less`) is the comparison operator and the second (`xs`) is the list to be sorted. Function `less` must be defined for every element that appears in the list to be sorted.

```
msort :: Ord a => (a -> a -> Bool) -> [a] -> [a]
msort _    []         = []
msort _    [x]        = [x]
msort less xs = merge less (msort less ls) (msort less rs)
    where n       = (length xs) `div` 2
          (ls,rs) = splitAt n xs
          merge _ [] ys     = ys
          merge _ xs []     = xs
          merge less ls@(x:xs) rs@(y:ys)
              | less x y  = x : (merge less xs rs)
              | otherwise = y : (merge less ls ys)
```

By nesting the definition of `merge`, we enabled it to directly access the the parameters of `msort`. In particular, we did not need to pass the comparison function to `merge`.

Assuming that `less` evaluates in constant time, the time complexity of `msort` is O(`n * log2 n`), where `n` is the length of the input list and `log2` is a function that computes the logarithm with base 2.

- Each call level requires splitting of the list in half and merging of the two sorted lists. This takes time proportional to the length of the list argument.
- Each call of `msort` for lists longer than one results in two recursive calls of `msort`.
- But each successive call of `msort` halves the number of elements in its input, so there are O(`log2 n`) recursive calls.

So the total cost is O(`n * log2 n`). The cost is independent of distribution of elements in the original list.

We can apply `msort` as follows:

```
msort (<) [5, 7, 1, 3]
```

Function `msort` is defined in curried form with the comparison function first. This enables us to conveniently specialize `msort` with a specific comparison function. For example,

```
descendSort :: Ord a => [a] -> [a]
descendSort = msort (\ x y -> x > y)     -- or (>)
```

17.6 Divide-and-Conquer Algorithms

The mergesort (`msort`) function in Section 17.5 uses the divide-and-conquer strategy to solve the sorting problem. In this section, we exmine that strategy in more detail.

17.6.1 General strategy

For some problem `P`, the general strategy for *divide-and-conquer algorithms* has the following steps:

1. *Decompose* the problem `P` into subproblems, each like `P` but with a smaller input argument.
2. *Solve* each subproblem, either directly or by recursively applying the strategy.
3. *Assemble* the solution to `P` by combining the solutions to its subproblems.

The advantages of divide-and-conquer algorithms are that they:

- can lead to efficient solutions.
- allow use of a "horizontal" parallelism. Similar problems can be solved simultaneously.

We examined the meregesort algorithm in Section 17.5. Other well-known divide-and-conquer algorithms include quicksort, binary search, and multiplication [15:6.4]. In these algorithms, the divide-and-conquer strategy leads to more efficient algorithms.

For example, consider searching for a value in a list. A simple *sequential search* has a time complexity of O(`n`), where n denotes the length of the list. Application of the divide-and-conquer strategy leads to *binary search*, a more efficient O(`log2 n`) algorithm.

17.6.2 As higher-order function

As a general pattern of computation, the divide and conquer strategy can be expressed as the following higher-order function:

```
divideAndConquer :: (a -> Bool)          -- trivial
                      -> (a -> b)        -- simplySolve
                      -> (a -> [a])      -- decompose
```

```
                         -> (a -> [b] -> b) -- combineSolutions
                         -> a               -- problem
                         -> b

divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
    = solve problem
      where solve p
              | trivial p = simplySolve p
              | otherwise = combineSolutions p
                               (map solve (decompose p))
```

If the problem is trivially simple (i.e., `trivial p` holds), then it can be solved directly using `simplySolve`.

If the problem is not trivially simple, then it is decomposed using the `decompose` function. Each subproblem is then solved separately using `map solve`. The function `combineSolutions` then assembles the subproblem solutions into a solution for the overall problem.

Sometimes `combineSolutions` may require the original problem description to put the solutions back together properly. Hence, the parameter `p` in the function definition.

Note that the solution of each subproblem is completely independent from the solution of all the others.

If all the subproblem solutions are needed by `combineSolutions`, then the language implementation could potentially solve the subproblems simultaneously. The implementation could take advantage of the availability of multiple processors and actually evaluate the expressions in parallel. This is "horizontal" parallelism as described above.

If `combineSolutions` does not require all the subproblem solutions, then the subproblems cannot be safely solved in parallel. If they were, the result of `combineSolutions` might be *nondeterministic*, that is, the result could be dependent upon the relative order in which the subproblem results are completed.

Now let's use the function `divideAndConquer` to define a few functions.

17.6.3 Generating Fibonacci sequence

First, let's define a Fibonacci function. Consider the following definition (adapted from Kelly [109:77–78]). This function is inefficient, so it is given here primarily to illustrate the technique.

```
fib :: Int -> Int
fib n = divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
        where trivial 0                = True
```

```
trivial 1                   = True
trivial (m+2)               = False
simplySolve 0               = 0
simplySolve 1               = 1
decompose m                 = [m-1,m-2]
combineSolutions _ [x,y] = x + y
```

17.6.4 Folding a list

Next, let's consider a folding function (similar to `foldr` and `foldl`) that uses the function `divideAndConquer`. Consider the following definition (also adapted from Kelly [109:79–80]).

```
fold :: (a -> a -> a) -> a -> [a] -> a
fold op i =
    divideAndConquer trivial simplySolve decompose
                     combineSolutions
    where trivial xs              = length xs <= 1
          simplySolve []          = i
          simplySolve [x]         = x
          decompose xs            = [take m xs, drop m xs]
                                    where m = length xs / 2
          combineSolutions _ [x,y] = op x y
```

This function divides the input list into two almost equal parts, folds each part separately, and then applies the operation to the two partial results to get the combined result.

The `fold` function depends upon the operation `op` being *associative*. That is, the result must not be affected by the order in which the operation is applied to adjacent elements of the input list.

In `foldr` and `foldl`, the operations are not required to be associative. Thus the result might depend upon the right-to-left operation order in `foldr` or left-to-right order in `foldl`.

Function `fold` is thus a bit less general. However, since the operation is associative and `combineSolutions` is strict in all elements of its second argument, the operations on pairs of elements from the list can be safely done in parallel,

Another divide-and-conquer definition of a folding function is the function `fold'` shown below. It is an optimized version of `fold` above.

```
fold' :: (a -> a -> a) -> a -> [a] -> a
fold' op i xs = foldt (length xs) xs
                where foldt _ []  =   i
                      foldt _ [x] =   x
                      foldt n ys  = op (foldt m (take m ys))
                                       (foldt m' (drop m ys))
```

```
                                       where  m  = n / 2
                                              m' = n - m
```

17.6.5 Finding minimum and maximum of a list

Now, consider the problem of finding both the minimum and the maximum values in a nonempty list and returning them as a pair.

First let's look at a definition that uses the left-folding operator.

```
sMinMax :: Ord a => [a] -> (a,a)
sMinMax (x:xs) = foldl' newmm (x,x) xs
                 where newmm (y,z) u = (min y u, max z u)
```

Let's assume that the comparisons of the elements are expensive and base our time measure on the number of comparisons. Let n denote the length of the list argument and `time` be a time function

A singleton list requires no comparisons. Each additional element adds two comparisons (one `min` and one `max`).

```
time n | n == 1 = 0
       | n >= 2 = time (n-1) + 2
```

Thus `time n == 2 * n - 2`.

Now let's look at a divide-and-conquer solution.

```
minMax :: Ord a => [a] -> (a,a)
minMax [x]   = (x,x)
minMax [x,y] = if x < y then (x,y) else (y,x)
minMax xs    = (min a c, max b d)
               where m     = length xs / 2
                     (a,b) = minMax (take m xs)
                     (c,d) = minMax (drop m xs)
```

Again let's count the number of comparisons for a list of length `n`.

```
time n | n == 1 = 0
       | n == 2 = 1
       | n > 2  = time (floor (n/2)) + time (ceiling (n/2)) + 2
```

For convenience suppose `n = 2^k` for some `k >= 1`.

```
time n = 2 * time (n/2) + 2
       = 2 * (2 * time (n/4) + 2) + 2
       = 4 * time (n/4) + 4 + 2
       = ...
       = 2^(k-1) * time 2 + sum [ 2^i | i <- [1..(k-1)] ]
       = 2^(k-1) + 2 * sum [ 2^i | i <- [1..(k-1)] ]
                 -     sum [ 2^i | i <- [1..(k-1)] ]
       = 2^(k-1) + 2^k - 2
```

```
= 3 * 2^(k-1) - 2
= 3 * (n/2) - 2
```

Thus the divide and conquer version takes 25 percent fewer comparisons than the left-folding version.

So, if element comparisons are the expensive in relation to to the `take`, `drop`, and `length` list operations, then the divide-and-conquer version is better. However, if that is not the case, then the left-folding version is probably better.

Of course, we can also express `minMax` in terms of the function `divideAndConquer`.

```
minMax' :: Ord a => [a] -> (a,a)
minMax' = divideAndConquer trivial simplySolve decompose
                           combineSolutions
          where n                 = length xs
                m                 = n/2
                trivial xs        = n <= 2
                simplySolve [x]   = (x,x)
                simplySolve [x,y] =
                    if x < y then (x,y) else (y,x)
                decompose xs =
                    [take m xs, drop m xs]
                combineSolutions _ [(a,b),(c,d)] =
                    (min a c, max b d)
```

17.7 What Next?

Chapters 15, 16, and 17 (this chapter) examined higher-order list programming concepts and features.

Chapter 18 examines list comprehensions, an alternative syntax for higher-order list processing that is likely comfortable for programmers coming from an imperative programming background.

17.8 Chapter Source Code

The Haskell module for list-breaking, list-combining, and mergesort functions is in file `HigherOrderExamples.hs`.

The Haskell module for the revised rational arithmetic module is in `RationalHO.hs`. The module `TestRationalHO.hs` is an extended version of the standard test script from Chapter 12.

TODO: Reconstruct source code for divide-and-conquer functions and place links here and in text above. May also want to break out mergesort into a separate module.

17.9 Exercises

1. Define a Haskell function

```
removeFirst :: (a -> Bool) -> [a] -> [a]
```

   so that `removeFirst p xs` removes the first element of `xs` that has the property `p`.

2. Define a Haskell function

```
removeLast :: (a -> Bool) -> [a] -> [a]
```

   so that `removeLast p xs` removes the last element of `xs` that has the property `p`.

   How could you define it using `removeFirst`?

3. A list `s` is a *prefix* of a list `t` if there is some list `u` (perhaps nil) such that `s ++ u == t`. For example, the prefixes of string `"abc"` are `""`, `"a"`, `"ab"`, and `"abc"`.

   A list `s` is a *suffix* of a list `t` if there is some list `u` (perhaps nil) such that `u ++ s == t`. For example, the suffixes of `"abc"` are `"abc"`, `"bc"`, `"c"`, and `""`.

   A list `s` is a *segment* of a list `t` if there are some (perhaps nil) lists `u` and `v` such that `u ++ s ++ v = t`. For example, the segments of string `"abc"` consist of the prefixes and the suffixes plus `"b"`.

   Define the following Haskell functions. You may use functions appearing early in the list to implement later ones.

   a. Define a function `prefix` such that `prefix xs ys` returns `True` if `xs` is a prefix of `ys` and returns `False` otherwise.

   b. Define a function `suffixes` such that `suffixes xs` returns the list of all suffixes of list `xs`. (Hint: Generate them in the order given in the example of `"abc"` above.)

   c. Define a function `indexes` such that `indexes xs ys` returns a list of all the positions at which list `xs` appears in list `ys`. Consider the first character of `ys` as being at position 0. For example, `indexes "ab" "abaabbab"` returns `[1,4,7]`. (Hint: Remember functions like `map`, `filter`, `zip`, and the functions you just defined.)

   d. Define a function `sublist` such that `sublist xs ys` returns `True` if list `xs` appears as a segment of list `ys` and returns `False` otherwise.

4. Assume that the following Haskell type synonyms have been defined:

```
type Word = String -- word, characters left-to-right
type Line = [Word] -- line, words left-to-right
```

```haskell
type Page = [Line] -- page, lines top-to-bottom
type Doc  = [Page] -- document, pages front-to-back
```

Further assume that values of type `Word` do not contain any space characters. Implement the following Haskell text-handling functions:

a. `npages` that takes a `Doc` and returns the number of `Page`s in the document.

b. `nlines` that takes a `Doc` and returns the number of `Line`s in the document.

c. `nwords` that takes a `Doc` and returns the number of `Word`s in the document.

d. `nchars` that takes a `Doc` and returns the number of `Char`s in the document (not including spaces of course).

e. `deblank` that takes a `Doc` and returns the `Doc` with all blank lines removed. A blank line is a line that contains no words.

f. `linetext` that takes a `Line` and returns the line as a `String` with the words appended together in left-to-right order separated by space characters and with a newline character `'\n'` appended to the right end of the line. (For example, `linetext ["Robert", "Khayat"]` yields `"Robert Khayat\n"`.)

g. `pagetext` that takes a `Page` and returns the page as a `String`—applies `linetext` to its component lines and appends the result in a top-to-bottom order.

h. `doctext` that takes a `Doc` and returns the document as a `String`—applies `pagetext` to its component lines and appends the result in a top-to-bottom order.

i. `wordeq` that takes a two `Doc`s and returns `True` if the two documents are *word equivalent* and `False` otherwise. Two documents are word equivalent if they contain exactly the same words in exactly the same order regardless of page and line structure. For example, `[[["Robert"],["Khayat"]]]` is word equivalent to `[[["Robert","Khayat"]]]`.

17.10 Wally World Marketplace POP Project

17.10.1 Problem description and initial design

Wally World Marketplace (WWM) is a "big box" store selling groceries, dry goods, hardware, electronics, etc. In this project, we develop part of a point-of-purchase (POP) system for WWM.

The barcode scanner at a WWM POP—i.e., checkout counter—generates a list of barcodes for the items in a customer's shopping cart. For example, a cart

with nine items might result in the list:

```
[ 1848, 1620, 1492, 1620, 1773, 2525, 9595, 1945, 1066 ]
```

Note that there are two instances of the item with barcode `1620`.

The primary goal of this project is to develop a Haskell module `WWMPOP` (in file `WWMPOP.hs`) that takes a list of barcodes corresponding to the items in a shopping cart and generates the corresponding printable receipt. The module consists of several functions that work together. We build these incrementally in a somewhat bottom-up manner.

Let's consider how to model the various kinds of "objects" in our application. The basic objects include:

- barcodes for products, which we represent as integers
- prices of products, which we represent as integers denoting cents
- names of products, which we represent as strings

We introduce the following Haskell type aliases for these basic objects above:

```
type BarCode = Int
type Price   = Int
type Name    = String
```

We associate barcodes with the product names and prices using a "database" represented as a list of tuples. We represent this price list database using the following type alias:

```
type PriceList = [(BarCode,Name,Price)]
```

An example price list database is:

```
database :: PriceList
database = [ (1848, "Vanilla yogurt cups (4)",    188),
             (1620, "Ground turkey (1 lb)",       316),
             (1492, "Corn flakes cereal",         299),
             (1773, "Black tea bags (100)",       307),
             (2525, "Athletic socks (6)",         825),
             (9595, "Claw hammer",                788),
             (1945, "32-in TV",                 13949),
             (1066, "Zero sugar cola (12)",       334),
             (2018, "Haskell programming book",  4495)
           ]
```

To generate a receipt, we need to take a list of barcodes from a shopping cart and generate a list of prices associated with the items in the cart. From this list, we can generate the receipt.

We introduce the type aliases:

```haskell
type CartItems  = [BarCode]
type CartPrices = [(Name,Price)]
```

We thus identify the need for a Haskell function

```haskell
priceCart :: PriceList -> CartItems -> CartPrices
```

that takes a database of product prices (i.e., a price list) and a list of barcodes of the items in a shopping cart and generates the list of item prices.

Of course, we must determine the relevant sales taxes due on the items and determine the total amount owed. We introduce the following type alias for the bill:

```haskell
type Bill = (CartPrices, Price, Price, Price)
```

The three `Price` items above are for Subtotal, Tax, and Total amounts associated with the purchase (printed on the bottom of the receipt).

We thus identify the need for a Haskell function

```haskell
makeBill :: CartPrices -> Bill
```

that takes the list of item prices and constructs a `Bill` tuple. In carrying out this calculation, the function uses the following n constant:

```haskell
taxRate :: Double
taxRate = 0.07
```

Given a bill, we must be able to convert it to a printable receipt. Thus we introduce the Haskell function

```haskell
formatBill ::  Bill -> String
```

that takes a bill tuple and generates the receipt. It uses the following named constant for the width of the line:

```haskell
lineWidth :: Int
lineWidth = 34
```

Given the above functions, we can put the above functionality together with the Haskell function:

```haskell
makeReceipt :: PriceList -> CartItems -> String
```

that does the end-to-end conversion of a list of barcodes to a printed receipt given an applicable price database, tax rate, and line width.

Given the example shopping cart items and price list database, we get the following receipt when printed.

```
         Wally World Marketplace

Vanilla yogurt cups (4).......1.88
Ground turkey (1 lb)..........3.16
```

```
Toasted oat cereal...........2.99
Ground turkey (1 lb).........3.16
Black tea bags (100).........3.07
Athletic socks (6)...........8.25
Claw hammer..................7.88
32-in. television..........139.49
Zero sugar cola (12).........3.34

Subtotal...................176.26
Tax.........................12.34
Total......................188.60
```

The above Haskell definitions are collected into the source file `WWMPOP_skeleton.hs`.

The exercises in Section 17.10.3 guide you to develop the above functions incrementally.

17.10.2 Prelude functions useful for project

In the exercises in Section 17.10.3, you may want to consider using some of the following:

- numeric functions from the Prelude library such as such as:
  - `div`, integer division truncated toward negative infinity, and `quot`, integer division truncated toward 0
  - `rem` and `mod` satisfy the following for `y /= 0`

    ```
    (x `quot` y)*y + (x `rem` y) == x
    (x `div`  y)*y + (x `mod` y) == x
    ```

  - `floor`, `ceiling`, `truncate`, and `round` that convert real numbers to integers; `truncate` truncates toward 0 and `round` rounds away from 0
  - `fromIntegral` converts integers to `Double` (and from `Integer` to `Int`)
  - `show` converts numbers to strings
- first-order list functions (Chapters 13 and 14) from the Prelude–such as `head`, `tail`, `++`, -`take`, `drop`, `length`, -`sum`, and `product`
- Prelude function `replicate :: Int -> a -> [a]` such that `replicate n e` returns a list of `n` copies of `e`
- higher-order list functions (Chapters 15, 16, and 17) from the Prelude such as `map`, `filter`, `foldr`, `foldl`, and `concatMap`
- list comprehensions (Chapter 18)—not necessary for solution but may be convenient

17.10.3 POP project exercises

Note: Most of the exercises in this project can be programmed without direct recursions. Consider the Prelude functions listed in the previous subsection.

Also remember that the character code `'\n'` is the newline character; it denotes the end of a line in Haskell strings.

This project defines several type aliases and the constants `lineWidth` and `taxRate` that should be defined and used in the exercises. You should start with the template source file `WWMPOP_skeleton.hs` to develop your own `WWMPOP.hs` solution.

1. Develop the Haskell function

   ```
   formatDollars :: Price -> String
   ```

   that takes a `Price` in cents and formats a string in dollars and cents. For example, `formatDollars 1307` returns the string `13.07`. (Note the `0` in `07`.)

2. Using `formatDollars` above, develop the Haskell function

   ```
   formatLine :: (Name, Price) -> String
   ```

   that takes an item and formats a line of the receipt for that item. For example,

   ```
   formatLine ("Claw hammer",788)
   ```

   yields the string:

   ```
   "Claw hammer...................7.88\n"
   ```

   This string has length `lineWidth` not including the newline character. The space between the item's name and cost is filled using `'.'` characters.

3. Using the `formatLine` function above, develop the Haskell function

   ```
   formatLines :: CartPrices -> String
   ```

   that takes a list of priced items and formats a string with a line for each item. (In general, the resulting string will consist of several lines, each ending with a newline.)

4. Develop the Haskell function

   ```
   calcSubtotal :: CartPrices -> Price
   ```

   that takes a list of priced items and calculates the sum of the prices (i.e., the subtotal).

5. Develop the Haskell function

   ```
   formatAmt :: String -> Price -> String
   ```

that takes a label string and a price amount and generates a line of the receipt for that label

For example,

```
formatAmt "Total" 18860
```

generates the string:‘

```
"Total......................188.60\n"`.
```

6. Develop the Haskell function

```
formatBill :: Bill -> String
```

that takes a `Bill` tuple and generates a receipt string.

7. Develop the Haskell function

```
look :: PriceList -> BarCode -> (Name,Price)
```

that takes a price list database and a barcode for an item and looks up the name and price of the item.

If the `BarCode` argument does not occur in the `PriceList`, then `look` should return the tuple `("None",0)`.

8. Now develop the Haskell function

```
priceCart :: PriceList -> CartItems -> CartPrices
```

defined above.

9. Now develop the Haskell function

```
makeBill :: CartPrices -> Bill
```

defined above. It takes a list of priced items and generates a bill tuple. It uses the `taxRate` constant.

10. Now develop the Haskell function

```
makeReceipt :: PriceList -> CartItems -> String
```

defined above. This function defines the end-to-end processing that takes the list of items from the shopping cart and generates the receipt.

11. Develop Haskell functions

```
addPL    :: PriceList -> BarCode -> (Name,Price)
                -> PriceList
removePL :: PriceList -> BarCode -> PriceList
```

Function `removePL` takes an “old” price list and a barcode to remove and returns a “new” price list with any occurrences of that barcode removed.

Function `addPL` takes an "old" price list, a barcode, and a name/price pair to add and returns a price list with the item added. (If the the barcode is already in the list, the old entry should be removed.)

17.11 Acknowledgements

In Summer 2016, I adapted and revised the following to form a chapter on Higher-Order Functions:

- Chapter 6 of my *Notes on Functional Programming with Haskell* [42], which is influenced by Bird [13–15] and Wentworth [178]
- My notes on *Functional Data Structures (Scala)* [50], which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [29] and its associated materials [30,31]

In 2017, I continued to develop this work as Chapter 5, Higher-Order Functions, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Higher-Order Functions chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming* (ELIFP), Previous sections 5.1-5.2 became the basis for new Chapter 15, Higher-Order Functions, section 5.3 became the basis for new Chapter 16, Haskell Function Concepts, and previous sections 5.4-5.6 became the basis for new Chapter 17 (this chapter), Higher-Order Function Examples.

In Fall 2018, I developed the Wally World Marketplace POP project. It was motivated by a similar project in Thompson's textbook [173] that I had used in my courses. I designed the project and its exercises to allow for the possibility of automatic grading.

In Summer 2018, I also adapted and revised Chapter 14 of my *Notes on Functional Programming with Haskell* [42] to form Chapter 29 (Divide and Conquer Algorithms) of ELIFP. These previous notes drew on the presentations in the 1st edition of the Bird and Wadler textbook [15], Kelly's dissertation [109], and other functional programming sources.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I also merged the previous ELIFP Chapter 29 (Divide and Conquer Algorithms) and the Wally World Marketplace project into an expanded Chapter 17 (this chapter) of the revised ELIFP.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document

to HTML, PDF, and other forms as needed.

17.12 Terms and Concepts

List-breaking (splitting) operators, list-combining operators, rational arithmetic, merge sort, divide and conquer, horizontal parallelism, divide and conquer as higher-order function, sequential search binary search, simply solve, decompose, combine solutions, Fibonacci sequence, nondeterministic, associative.

18 More List Processing

18.1 Chapter Introduction

Previous chapters examined first-order and higher-order list programming. In particular, Chapter 15 explored the standard higher order functions such as `map`, `filter`, and `concatMap` and Chapter 16 explored function concepts such as function composition.

This chapter examines list comprehensions. This feature does not add new power to the language; the computations can be expressed with combinations of features from the previous chapters. But list comprehensions are often easier to write and to understand than equivalent compositions of `map`, `filter`, `concatMap`, etc.

The source file for the code in this chapter is in file `MoreLists.hs`.

18.2 Sequences

Haskell provides a compact notation for expressing arithmetic sequences.

An arithmetic sequence (or progression) is a sequence of elements from an enumerated type (i.e., a member of class `Enum`) such that consecutive elements have a fixed difference. `Int`, `Integer`, `Float`, `Double`, and `Char` are all predefined members of this class.

- `[m..n]` produces the list of elements from `m` up to `n` in steps of one if `m <= n`. It produces the nil list otherwise.

  Examples:

  - `[1..5]` $\Longrightarrow$ `[1,2,3,4,5]`
  - `[5..1]` $\Longrightarrow$ `[]`

  This feature is implemented with Prelude function `enumFromTo` applied as `enumFromTo m n`.

- `[m,m'..n]` produces the list of elements from `m` in steps of `m'-m`. If `m' > m` then the list is increasing up to `n`. If `m' < m`, then it is decreasing.

  Examples:

  - `[1,3..9]` $\Longrightarrow$ `[1,3,5,7,9]`
  - `[9,8..5]` $\Longrightarrow$ `[9,8,7,6,5]`
  - `[9,8..11]` $\Longrightarrow$ `[]`

  This feature is implemented with Prelude function `enumFromThenTo` applied as `enumFromThenTo m' m n`.

- `[m..]` and `[m,m'..]` produce potentially infinite lists beginning with `m` and having steps 1 and `m'-m` respectively.

  These features are implemented with Prelude functions `enumFrom` applied as `enumFrom m` and `enumFromThen` applied as `enumFromThen m m'`.

Of course, we can provide our own functions for sequences. Consider the following function to generate a geometric sequence.

A geometric sequence (or progression) is a sequence of elements from an ordered, numeric type (i.e., a member of both classes `Ord` and `Num`) such that consecutive elements have a fixed ratio.

```
geometric :: (Ord a, Num a) => a -> a -> a -> [a]
geometric r m n | m > n     = []
                | otherwise = m : geometric r (m*r) n
```

Example: `geometric 2 1 10` ⟹ `[1,2,4,8]`

18.3 List Comprehensions

18.3.1 Syntax and semantics

The *list comprehension* is another powerful and compact notation for describing lists. A list comprehension has the form

`[` *expression* `|` *qualifiers* `]`

where *expression* is any Haskell expression.

The *expression* and the *qualifiers* in a comprehension may contain variables that are local to the comprehension. The values of these variables are bound by the *qualifiers*.

For each group of values bound by the qualifiers, the comprehension generates an element of the list whose value is the *expression* with the values substituted for the local variables.

There are three kinds of *qualifiers* that can be used in Haskell: generators, filters, and local definitions.

1. A *generator* is a qualifier of the form

   pat `<-` *exp*

   where *exp* is a list-valued expression. The generator extracts each element of *exp* that matches the pattern *pat* in the order that the elements appear in the list; elements that do not match the pattern are skipped.

   Example:

   - `[ n*n | n <- [1..5]]` ⟹ `[1,4,9,16,25]`

2. A *filter* is a Boolean-valued expression used as a *qualifier* in a list comprehension. These expressions work like the `filter` function; only values that make the expression `True` are used to form elements of the list comprehension.

   Example:

- `[ n*n | even n ]` ⟹ `(if even n then [n*n] else [])`

  Above variable `n` is global to this expression, not local to the comprehension.

3. A *local definition* is a qualifier of the form

   let *pat* = *expr*

   introduces a local definition into the list comprehension.

   Example:

   - `[ n*n | let n = 2 ]` ⟹ `[4]`

The real power of list comprehensions come from using several qualifiers separated by commas on the right side of the vertical bar `|`.

- Generators appearing later in the list of qualifiers vary more quickly than those that appear earlier. Speaking operationally, the generation "loop" for the later generator is nested within the "loop" for the earlier.

  Example:

  – `[ (m,n) | m<-[1..3], n<-[4,5] ]` ⟹
    `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

- Qualifiers appearing later in the list of qualifiers may use values generated by qualifiers appearing earlier, but not vice versa.

  Examples:

  – `[ n*n | n<-[1..10], even n ]` ⟹ `[4,16,36,64,100]`

  – `[ (m,n) | m<-[1..3], n<-[1..m] ]` ⟹
    `[ (1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]`

- The generated values may or may not be used in the *expression*.

  Examples:

  – `[ 27 | n<-[1..3]]` ⟹ `[27,27,27]`

  – `[ x | x<-[1..3], y<-[1..2]]` ⟹ `[1,1,2,2,3,3]`

18.3.2 Translating list comprehensions

List comprehensions are syntactic sugar. We can translate them into core Haskell features by applying the following identities.

1. For any expression `e`,

   ```
   [ e | True ]
   ```

   is equivalent to:

   ```
   [ e ]
   ```

2. For any expression `e` and qualifier `q`,

   ```
   [ e | q ]
   ```

   is equivalent to:

   ```
   [ e | q, True ]
   ```

3. For any expression `e`, boolean `b`, and sequence of qualifiers `Q`,

   ```
   [ e | b, Q ]
   ```

   is equivalent to:

   ```
   if b then [ e | Q ] else []
   ```

4. For any expression `e`, pattern `p`, list-valued expression `l`, sequence of qualifiers `Q`, and fresh variable `ok`,

   ```
   [ e | p <- l, Q ]
   ```

   is equivalent to:

   ```
   let ok p = [ e | Q ] -- p is a pattern
       ok _ = []
   in concatMap ok l
   ```

5. For any expression `e`, declaration list `D`, and sequence of qualifiers `Q`,

   ```
   [ e | let D, Q ]
   ```

   is equivalent to:

   ```
   let D in [ e | Q ]
   ```

Function `concatMap` and boolean value `True` are as defined in the Prelude.

As we saw in a previous chapter, `concatMap` applies a list-returning function to each element of an input list and then concatenates the resulting list of lists into a single list. Both `map` and `filter` can be defined in terms of `concatMap`.

Consider the list comprehension:

```
[ n*n | n<-[1..10], even n ]
```

a. Apply identity 4:

   ```
   let ok n = [ n*n | even n ]
       ok _ = []
   in concatMap ok [1..10]
   ```

b. Apply identity 2:

   ```
   let ok n = [ n*n | even n, True ]
       ok _ = []
   in concatMap ok [1..10]
   ```

c. Apply identity 3:

```
let ok n = if (even n) then [ n*n | True ]
    ok _ = []
in concatMap ok [1..10]
```

d. Apply identity 1:

```
let ok n = if (even n) then [ n*n ]
    ok _ = []
in concatMap ok [1..10]
```

18.4 Using List Comprehensions

This section gives several examples where list comprehensions can be used to solve problems and express the solutions conveniently.

18.4.1 Strings of spaces

Consider a function `spaces` that takes a number and generates a string with that many spaces.

```
spaces :: Int -> String
spaces n = [ ' ' | i<-[1..n]]
```

Note that when `n < 1` the result is the empty string.

18.4.2 Prime number test

Consider a Boolean function `isPrime` that takes a nonzero natural number and determines whether the number is *prime*. (Remember that a prime number is a natural number whose only natural number factors are 1 and itself.)

```
isPrime :: Int -> Bool
isPrime n | n > 1 = (factors n == [])
    where factors m = [ x | x<-[2..(m-1)], m `mod` x == 0 ]
isPrime   _       =  False
```

18.4.3 Squares of primes

Consider a function `sqPrimes` that takes two natural numbers and returns the list of squares of the prime numbers in the inclusive range from the first up to the second.

```
sqPrimes :: Int -> Int -> [Int]
sqPrimes m n = [ x*x | x<-[m..n], isPrime x ]
```

Alternatively, this function could be defined using `map` and `filter` as follows:

```
sqPrimes' :: Int -> Int -> [Int]
sqPrimes' m n = map (\x -> x*x) (filter isPrime [m..n])
```

18.4.4 Doubling positive elements

We can use a list comprehension to define (our, by now, old and dear friend) the function `doublePos`, which doubles the positive integers in a list.

```
doublePos5 :: [Int] -> [Int]
doublePos5 xs = [ 2*x | x<-xs, 0 < x ]
```

18.4.5 Concatenating a list of lists of lists

Consider a program `superConcat` that takes a list of lists of lists and concatenates the elements into a single list.

```
superConcat :: [[[a]]] -> [a]
superConcat xsss = [ x | xss<-xsss, xs<-xss, x<-xs ]
```

Alternatively, this function could be defined using Prelude functions `concat` and `map` and functional composition as follows:

```
superConcat' :: [[[a]]] -> [a]
superConcat' = concat . map concat
```

18.4.6 First occurrence in a list

Consider a function `position` that takes a list and a value of the same type. If the value occurs in the list, `position` returns the position of the value's first occurrence; if the value does not occur in the list, `position` returns 0.

Strategy: *Solve a more general problem first, then use it to get the specific solution desired.*

In this problem, we generalize the problem to finding *all* occurrences of a value in a list. This more general problem is actually easier to solve.

```
positions :: Eq a => [a] -> a -> [Int]
positions xs x = [ i | (i,y)<-zip [1..length xs] xs, x == y]
```

Function `zip` is useful in pairing an element of the list with its position within the list. The subsequent filter removes those pairs not involving the value `x`. The "zipper" functions can be very useful within list comprehensions.

Now that we have the positions of all the occurrences, we can use `head` to get the first occurrence. Of course, we need to be careful that we return 0 when there are no occurrences of `x` in `xs`.

```
position :: Eq a => [a] -> a -> Int
position xs x = head ( positions xs x ++ [0] )
```

Because of lazy evaluation, this implementation of `position` is not as inefficient as it first appears. The function `positions` will, in actuality, only generate the head element of its output list.

Also because of lazy evaluation, the upper bound `length xs` can be left off the generator in `positions`. In fact, the function is more efficient to do so.

18.5 What Next?

This chapter (18) examined list comprehensions. Although they do not add new power to the language, programs involving comprehensions are often easier to write and to understand than equivalent compositions of other functions.

Chapters 19 and 20 discuss problem solving techniques. Chapter 19 discusses systematic generalization of functions. Chapter 20 surveys various problem-solving techniques uses in this textbook and other sources.

18.6 Chapter Source Code

The source file for the code in this chapter is in file `MoreLists.hs`.

18.7 Exercises

1. Show the list (or string) yielded by each of the following Haskell list expressions. Display it using fully specified list bracket notation, e.g., expression `[1..5]` yields `[1,2,3,4,5]`.

   a. `[7..11]`

   b. `[11..7]`

   c. `[3,6..12]`

   d. `[12,9..2]`

   e. `[ n*n | n <- [1..10], even n ]`

   f. `[ 7 | n <- [1..4] ]`

   g. `[ x | (x:xs) <- [Did, you, study?] ]`

   h. `[ (x,y) | x <- [1..3], y <- [4,7] ]`

   i. `[ (m,n) | m <- [1..3], n <- [1..m] ]`

   j. `take 3 [ [1..n] | n <- [1..] ]`

2. Translate the following expressions into expressions that use list comprehensions. For example, `map (*2) xs` could be translated to `[ x*2 | x <- xs ]`.

   a. `map (\x -> 2*x-1) xs`

   b. `filter p xs`

   c. `map (^2) (filter even [1..5])`

   d. `foldr (++) [] xss`

e. `map snd (filter (p . fst) (zip xs [1..]))`

18.8 Acknowledgements

In 2016 and 2017, I adapted and revised my previous notes to form Chapter 7, More List Processing and Problem Solving, in the 2017 version of this textbook. In particular, I drew the information on More List Processing from:

- chapter 7 of my *Notes on Functional Programming with Haskell* [42]

In Summer 2018, I divided the 2017 More List Processing and Problem Solving chapter back into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 7.2-7.3 (essentially chapter 7 of [42]) became the basis for new Chapter 18, More List Processing (this chapter), and the Problem Solving discussion became the basis for new Chapter 20, Problem Solving.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

18.9 Terms and Concepts

Sequence (arithmetic, geometric), list comprehension (generator, filter, local definition, multiple generators and filters), syntactic sugar, translating list comprehensions to function calls, prime numbers, solve a harder problem first.

<a name-“Ch19”>

19 Systematic Generalization

19.1 Chapter Introduction

This chapter is incomplete!

TODO: Write missing sections

- Cleanup function generalization description — make it fit better with the discussion in chapter 15 and elsewhere
- Add eager evaluated version of `merge4b`, perhaps rename `coseq`
- Complete sequential file update example
- Update code and include link `Conseq.hs`

19.2 SCV Analysis

In Chapter 15, we examined *families* of related functions to define generic, higher-order functions to capture the computational pattern for each family. In this chapter, we approach *function generalization* more systematically.

The systematic function generalization approach begins with a prototype member of the potential family [55,57]. As in a Schmid's similar method for building object-oriented software frameworks [153]; [154,155], we apply *Scope-Commonality-Variability (SCV) analysis* [37] to the potential family represented by this prototype and produce four outputs.

1. *scope*: the boundaries of the family. That is, we identify what we should address and what we can ignore, what is in the family and what is not.

2. *terminology*: the definitions of the specialized terms, or concepts, relevant to the family.

3. *commonalities*: the aspects of the family that do not change from one member to another. We seek to reuses these among family members. We sometimes call these the *frozen spots*.

4. *variabilities*: the aspects of the family that may vary from one member to another. We sometimes call these the *hot spots*.

In SCV analysis, we must seek to identify all the implicit assumptions about elements in the family. These implicit assumptions need to be made explicit in family's design and implementation.

19.3 Function Generalization

Once we have the above, we incrementally transform the prototype function for each of the hot spots [154].

A generalizing transformation may replace specific values or data types at a hot spot by parameters. We may make a type more abstract, perhaps making it polymorphic. Or we may break a type into several types if it plays potentially different roles.

Similarly, a generalizing transformation may replace fixed, specialized operations at a hot spot by abstract operations. We may make an abstract operation a higher-order parameter of the generalized function.

TODO: Better tie this chapter's technique to tht given back in Section 15.5 on generalizing function definitions.

19.4 Developing a Cosequential Processing Family

19.4.1 Scope

Cosequential processing concerns the coordinated processing of two ordered sequences to produce some result, often a third ordered sequence [64,67,76,87]. Key requirements include:

- Both input sequences must be ordered according to the same total ordering.
- The processing should be incremental, where only a few elements of each sequence (perhaps just one) are examined at a time.

This important family includes the ascending merge needed in merge sort, set and bag operations, and sequential file update applications.

Consider a function `merge0` that takes two ascending sequences of integers and merges them together to form a third ascending sequence.

```
merge0 :: [Int] -> [Int] -> [Int]  -- xs, ys
merge0 [] ys = ys
merge0 xs [] = xs
merge0 xs@(x:xs') ys@(y:ys')
    | x < y    = x : merge0 xs' ys
    | x == y   = x : merge0 xs' ys'
    | x > y    = y : merge0 xs  ys'
```

The `merge0` function must satisfy a number of properties:

- *Precondition*: The two input lists must be in ascending order.
- *Postcondition*: The output list must also be in ascending order. The number of times an element appears in the output list is the maximum number of times it appears within one of the two input lists.
- *Termination*: The sum of the lengths of the two input sequences must decrease by at least one for each call of the recursive function.

For the cosequential processing family, let take function `merge0` as the prototype member.

Aside: The `merge0` function differs from the merge function we used in the merge sort funtion in Chapter 17. For merge sort, the `x == y` leg would need to remove the head of only one of the input lists, i.e., be defined as either `x : merge0 xs' ys` or `x : merge0 xs ys'`.

19.4.2 Frozen spots

Considering the scope and examining the prototype function `merge0`, we identify the following frozen spots for the family of functions:

1. The input consists of two sequences ordered by the same total ordering.
2. The output consists of a sequence ordered by the same total ordering as the input sequences.
3. The processing is incremental. Each step examines the current element from each input sequence and advances at least one of the input sequences for subsequent steps.
4. Each step compares the current elements from the input sequences to determine what action to take at that step.

The merge function represents the frozen spots of the family. It gives the common behavior of family members and the relationships among the various elements of the hot spot subsystems.

A *hot spot subsystem* consists of a set of Haskell functions, types, and class definitions that add the desired variability into the merge function.

19.4.3 Hot spots

Again considering the scope and examining the prototype function `merge0`, we can identify the following hot spots:

1. Variability in the total ordering used for the input and output sequences, i.e., of the comparison operators and input sequence type.
2. The ability to have more complex data entities in the input and output sequences, i.e., variability in "record" format.
3. The ability to vary the input and output sequence structures independently of each other.
4. Variability in the transformations applied to the data as it passes into the output.
5. Variability in the sources of the input sequences and destination of the output sequence.

We need to be careful to avoid enumerating hot spots that are unlikely to be needed in an application.

Now let's analyze each hot spot, design a hot spot subsystem, and carry out the appropriate transformations to generalize the Haskell program.

19.4.4 Hot spot #1: Variability in total ordering

In the function `merge0`, the input and output sequences are restricted to elements of type `Int` and the comparison operations, hence, to the integer comparisons.

TODO: May need to explain these generalizations according to the rules in Section 15.5.

The responsibility associated with hot spot #1 is to enable the base type of the sequences to be any type upon which an appropriate ordering is defined. In this transformation, we still consider all three sequences as containing simple values of the same type.

We can generalize the function to take and return sequences of any ordered type by making the type of the list polymorphic. Using a type variable `a`, we can redefine the type signature to be `[a]`.

However, we need to constrain type `a` to be a type for which an appropriate total ordering is defined. We do this by requiring that the type be restricted to those in the predefined Haskell type class `Ord`. This class consists of the group of types for which all six relational operators are defined.

The function resulting from generalization step is `merge1`.

```
merge1 :: Ord a => [a] -> [a] -> [a]  -- xs, ys
merge1 []  ys = ys
merge1 xs  [] = xs
merge1 xs@(x:xs') ys@(y:ys')
    | x <  y    = x : merge1 xs' ys
    | x == y    = x : merge1 xs' ys'
    | x >  y    = y : merge1 xs  ys'
```

This function represents the frozen spots of the cosequential processing framework. The implementation of class `Ord` used in a program is hot spot #1. To satisfy the requirement represented by frozen spot #1, we require that the two lists `xs` and `ys` be in ascending order.

Note that, if we restrict `merge1` polymorphic type `a` to `Int`, then:

```
merge1 xs ys == merge0 xs ys
```

That is, the generic function `merge1` can be *specialized* to be equivalent to `merge0`.

19.4.5 Hot spot #2: Variability in record format

The `merge1` function works with sequences of any type that have appropriate comparison operators defined. This allows the elements to be of some built-in

type such as `Int` or `String` or some user-defined type that has been declared as an instance of the `Ord` class. Thus each individual data item is of a single type.

In general, however, applications in this family will need to work with data elements that have more complex structures. We refer to these more complex structures as *records* in the general sense, not just the Haskell data structure by that name.

The responsibility associated with hot spot #2 is to enable the elements of the sequences to be values with more complex structures, i.e., records. Each record is composed of one or more fields of which some subset defines the key. The value of the key provides the information for ordering the records within that sequence.

In this transformation, we still consider all three sequences as containing simple values of the same type. We abstract the `key` as a function on the record type that returns a value of some `Ord` type to enable the needed comparisons. We transform the `merge1` function by adding `key` as a higher-order parameter.

The function resulting from this generalization is `merge2`.

```
merge2 :: Ord b => (a -> b) -> [a] -> [a] -> [a]  -- key, xs, ys
merge2 key []  ys  = ys
merge2 key xs  []  = xs
merge2 key xs@(x:xs') ys@(y:ys')
    | key x <  key y = x : merge2 key xs' ys
    | key x == key y = x : merge2 key xs' ys'
    | key x >  key y = y : merge2 key xs  ys'
```

The higher-order parameter `key` represents hot spot #2 in the generalized function design.

Hot spot #1 is the implementation of Haskell class `Ord` for values of type `b`.

To satisfy the requirement represented by frozen spot #1, the sequence of keys corresponding to each input sequence, i.e., `map key xs` and `map key ys`, must be in ascending order.

Also note that

```
merge2 id xs ys == merge1 xs ys
```

where `id` is the identity function. Thus `merge1` is a specialization of `merge2`.

19.4.6 Hot spot #3: Independent variability of sequences

In `merge2`, the records are of the same type in all three sequences. The key extraction function is also the same for all sequences.

Some cosequential processing applications, however, require that the record structure vary among the sequences. For example, the sequential file update application usually involves a master file and a transaction file as the inputs and

a new master file as the output. The master records and transaction records usually carry different information.

The responsibility associated with hot spot #3 is to enable the three sequences to be varied independently. That is, the records in one sequence may differ in structure from the records in the others.

This requires separate key extraction functions for the two input sequences. These must, however, still return key values from the same total ordering. Because the data types for the two input sequences may differ and both may differ from the output data type, we must introduce record transformation functions that convert the input data types to the output types.

The function resulting from the transformation is `merge3`.

```
merge3 :: Ord d => (a -> d) -> (b -> d)     -- kx, ky
                -> (a -> c) -> (b -> c) -- tx, ty
                -> [a] -> [b] -> [c]    -- xs, ys
merge3 kx ky tx ty xs ys   = mg xs ys
    where
        mg [] ys         = map ty ys
        mg xs []         = map tx xs
        mg xs@(x:xs') ys@(y:ys')
            | kx x <  ky y = tx x : mg xs' ys
            | kx x == ky y = tx x : mg xs' ys'
            | kx x >  ky y = ty y : mg xs  ys'
```

Higher-order parameters `kx` and `ky` are the *key* extraction functions for the first and second inputs, respectively. Similarly, `tx` and `ty` are the corresponding functions to *transform* those inputs to the output.

Hot spot #3 consists of these four functions. In some sense, this transformation subsumes hot spot #2.

To avoid repetition of the many unchanging arguments in the recursive calls, the definition of `merge3` uses an auxiliary function definition `mg`.

The nonrecursive legs use the higher-order library function `map`. To satisfy the requirement represented by frozen spot #1, the sequence of keys corresponding to each input sequence, i.e., `map kx xs` and `map ky ys`, must be in ascending order.

If `xs` and `ys` are of the same type, then it is true that:

```
merge3 key key id id xs ys ==  merge2 key xs ys
```

Thus `merge2` is a specialization of `merge3`.

19.4.7 Hot spot #4: Variability in sequence transformations

Function `merge3` enabled simple one-to-one, record-by-record transformations of the input sequences to create the output sequence. Such simple transformations are not sufficient for practical situations.

For example, in the sequential file update application, each key may be associated with no more than one record in the master file. However, there may be any number of update transactions that must be performed against a master record before the new master record can be output. Thus, there needs to be some local state maintained throughout the processing of all the transaction records associated with one master record.

Before we address the issue of this variation directly, let us generalize the merge function to make the state that currently exists (i.e., the evolving output list) explicit in the parameter list.

To do this, we replace the backward linear recursive `merge3` function by its *tail recursive* generalization. That is, we add an *accumulating parameter* `ss` that is used to collects the output during the recursive calls and then to generate the final output when the end of an input sequence is reached.

The initial value of this argument is normally a nil list, but it does enable some other initial value to be prepended to the output list. This transformation is shown as function `merge4a` below.

```
merge4a :: Ord d => (a -> d) -> (b -> d)         -- kx, ky
                 -> (a -> c) -> (b -> c)         -- tx, ty
                 -> [c] -> [a] -> [b] -> [c] -- ss, xs, ys
merge4a kx ky tx ty ss xs ys = mg ss xs ys
    where
        mg ss [] ys       = ss ++ map ty ys
        mg ss xs []       = ss ++ map tx xs
        mg ss xs@(x:xs') ys@(y:ys')
            | kx x <  ky y = mg (ss ++ [tx x]) xs' ys
            | kx x == ky y = mg (ss ++ [tx x]) xs' ys'
            | kx x >  ky y = mg (ss ++ [ty y]) xs  ys'
```

Note that the following holds:

```
merge4a kx ky tx ty ss xs ys == ss ++ merge3 kx ky tx ty xs ys
```

Thus function `merge3` is a specialization of `merge4a`.

Unfortunately, building up the state `ss` requires a relatively expensive appending to the end of a list (e.g., `ss ++ [tx x]` in the third leg).

Now consider hot spot #4 more explicitly. The responsibility associated with the hot spot is to enable the use of more general transformations on the input sequences to produce the output sequence.

To accomplish this, we introduce an explicit *state* to record the relevant aspects of the computation to some position in the two input sequences. Each call of the merge function can examine the current values from the input sequences and update the value of the state appropriately for the next call.

In some sense, the merge function "folds" together the values from the two input sequences to compute the state. At the end of both input sequences, the merge function then transforms the state into the output sequence.

To accomplish this, we can generalize `merge4a`. We generalize the accumulating parameter `ss` in `merge4a` to be a parameter `s` that represents the state. We also replace the two simple record-to-record transformation functions `tx` and `ty` by more flexible transformation functions `tl`, `te`, and `tg`, that update the state in the three guards of the recursive leg and functions `tty` and `ttx` that update the state when the first and second input sequences, respectively, become empty.

For the "equals" guard, the amount that the input sequences are advanced also becomes dependent upon the state of the computation. This is abstracted as functions `nex` on the first input sequence and `ney` on the second. To satisfy the requirement represented by frozen spot #3, the pair of functions `nex` and `ney` must make the following progress requirement true for each call of `mg`:

```
if (kx x == ky y) then
    (length (nex s xs) < length xs) ||
    (length (ney s ys) < length ys)
else True
```

That is, the client of the framework must ensure that at least one of the input sequences will be advanced by at least one element. We also introduce the new function `res` to take the final state of the computation and return the output sequence.

The above transformation results in function `merge4b`.

```
merge4b :: Ord d => (a -> d) -> (b -> d) -> -- kx, ky
                    (e -> a -> b -> e) ->   -- tl
                    (e -> a -> b -> e) ->   -- te
                    (e -> a -> b -> e) ->   -- tg
                    (e -> [a] -> [a])  ->   -- nex
                    (e -> [b] -> [b]) ->    -- ney
                    (e -> a -> e) ->        -- ttx
                    (e -> b -> e) ->        -- tty
                    (e -> [c]) -> e ->      -- res, s
                    [a] -> [b] -> [c]       -- xs, ys
merge4b kx ky tl te tg nex ney ttx tty res s xs ys
    =  mg s xs ys
    where
        mg s [] ys       = res (foldl tty s ys)
        mg s xs []       = res (foldl ttx s xs)
```

```
mg s xs@(x:xs') ys@(y:ys')
    | kx x <  ky y = mg (tl s x y) xs' ys
    | kx x == ky y = mg (te s x y) (nex s xs) (ney s ys)
    | kx x >  ky y = mg (tg s x y) xs ys
```

The function uses the Prelude function `foldl` in the first two legs. This function continues the computation beginning with the state computed by the recursive leg and processes the remainder of the nonempty input sequence by "folding" the remaining elements as defined in the functions `ttx` and `tty`.

As was the case for `merge3`, frozen spot #1 requires that `map kx xs` and `map ky ys` be in ascending order for calls to `merge4b`.

Hot spot #4 consists of the eight functions `tl`, `te`, `tg`, `ttx`, `tty`, `nex`, `ney`, and `res`. The following property also holds:

```
merge4b kx ky
    (\ss x y -> ss ++ [tx x])  -- tl
    (\ss x y -> ss ++ [tx x])  -- te
    (\ss x y -> ss ++ [ty y])  -- tg
    (\ss xs -> tail xs)        -- nex
    (\ss ys -> tail ys)        -- ney
    (\ss x -> ss ++ [x])       -- ttx
    (\ss y -> ss ++ [y])       -- tty
    id ss xs ys                -- res, ss, xs, ys
== merge4a kx ky tx ty ss xs ys
== ss ++ merge3 kx ky tx ty xs ys
```

That is, we can define the general transformation functions so that they have the same effect as the record-to-record transformations of `merge4a`. The statement of this property uses the anonymous functions (lambda expression) feature of Haskell.

Thus function `merge3` is a specialization of `merge4a`, which in turn is a specialization of function `merge4b`.

A problem with the above "implementation" of `merge3` is that the `merge4b` parameters `tl`, `te`, `tg`, `ttx`, and `tty` all involve an expensive operation to append to the end of the list `ss`.

An alternative would be to build the state sequence in reverse order and then reverse the result as shown below.

```
merge4b kx ky
    (\ss x y -> reverse (tx x) ++ ss)  -- tl
    (\ss x y -> reverse (tx x) ++ ss)  -- tl
    (\ss x y -> reverse (ty y) ++ ss)  -- tg
    (\ss xs -> tail xs)                -- nex
    (\ss ys -> tail ys)                -- ney
    (\ss x -> x : ss)                  -- ttx
```

```
        (\ss y -> y : ss)                  -- tty
        reverse ss xs ys                   -- res, ss, xs, ys
    == merge4a kx ky tx ty ss xs ys
    == ss ++ merge3 kx ky tx ty xs ys
```

TODO: Possibly include a version with selective eager evaluation (similar to below) and rename coseq.

```
merge4c :: Ord d => (a -> d) -> (b -> d) -> -- kx, ky
                    (e -> a -> b -> e) ->   -- tl
                    (e -> a -> b -> e) ->   -- te
                    (e -> a -> b -> e) ->   -- tg
                    (e -> [a] -> [a])  ->   -- nex
                    (e -> [b] -> [b]) ->    -- ney
                    (e -> a -> e) ->        -- ttx
                    (e -> b -> e) ->        -- tty
                    (e -> [c]) -> e ->      -- res, s
                    [a] -> [b] -> [c]       -- xs, ys
merge4b kx ky tl te tg nex ney ttx tty res s xs ys
        =  mg s xs ys
    where
        mg s [] ys       = res (foldl tty s ys)
        mg s xs []       = res (foldl ttx s xs)
        mg s xs@(x:xs') ys@(y:ys')
            | kx x <  ky y = (mg $! (tl s x y)) xs' ys
            | kx x == ky y = (mg $! (te s x y)) ((nex $! s) xs)
                                                ((ney $! s) ys)
            | kx x >  ky y = (mg $! (tg s x y)) xs ys'
```

19.4.8 Hot spot #5 :Variability of sequence source/destination

Hot spot #5 concerns the ability to take the input sequences from many possible sources and to direct the output to many possible destinations.

In the Haskell merge functions, these sequences are represented as the pervasive polymorphic list data type. The redirection is simply a matter of writing appropriate functions to produce the input lists and to consume its output list. No changes are needed to the `merge4b` function itself.

Of course, for any expressions (e.g., function calls) `ex` and `ey` that generate the input sequence arguments `xs` and `ys` of `merge4b`, it must be the case that sequences `map kx ex` and `map ky ey` are ascending.

19.4.9 Bag and set operation implementations

TODO: Possibly reexpress some of the lambdas above with standard combinators.

Mathematically, a *bag* (also called a *multiset*) is an unordered collection of elements in which each element may occur one or more times. We can model

a bag as a total function (called the *multiplicity function*) over the domain of elements to the natural numbers where the numbers 0 and above denote the number of occurrences of the element in the bag.

A *set* is thus a bag for which there is no more than one occurence of any element.

If we restrict the elements to a Haskell data type that is an instance of class `Ord`, we can represent a bag by an ascending list of values and a set by an increasing list of values. With this representation, we can implement the bag an set operations as special cases of cosequential processing.

The *intersection* of two bags consists of only the elements that occur in both bags such that the number of occurrences is the minimum number for the two input bags. We can express the bag intersection of two ascending lists in terms of `merge4b` as follows.

```
bagIntersect xs ys =
    merge4b id id
        (\s x y -> s)
        (\s x y -> x:s)
        (\s x y -> s)
        (\s xs -> tail xs)
        (\s ys -> tail ys)
        (\s x -> s)
        (\s y -> s)
        reverse [] xs ys
```

This function only adds an element to the output when it occurs in both input lists.

If we require the two input lists to be increasing, the above also implements set intersection.

The *sum* of two bags consists of the elements that occur in either bag such that the number of occurrences is the total number for both bags. We can express the bag sum of two ascending lists in terms of `merge4b` as follows.

```
bagSum xs ys =
    merge4b id id
        (\s x y -> x:s)
        (\s x y -> x:y:s)
        (\s x y -> y:s)
        (\s xs -> tail xs)
        (\s ys -> tail ys)
        (\s x -> x:s)
        (\s y -> y:s)
        reverse [] xs ys
```

The *union* of two bags consists of the elements that occur in either bag such that the number of occurrences is the minimum number in the two input lists.

The prototype function `merge0` implements this operation on ascending lists.

The *subtraction* of bag `B` from bag `A`, denoted `A - B`, consists of only the elements that occur in both bags such that the number of occurrences is the number of occurrences in `A` minus those in `B`.

Questions:

- How can we represent set union in terms of `merge4b`?
- How can we represent a merge function that can be used in the merge sort of two lists (whose elements are from an instance of class `Ord`)?
- How can we implement a bag union function `bagUnion` in terms of `merge4b`?
- How can we implement a bag subtraction function `bagSub xs ys` in terms of `merge4b`?
- If the elements of the input lists are not instances of class `Ord`, how can we implement bag union? bag intersection?

19.4.10 Sequential file update algorithm

TODO: Describe this!?

```
-- Simple Master-Transaction Update
-- Master increasing list [(Account,Amount)]
--   with Account < maxAccount
-- Transaction ascending list [(Account,Amount)]
--   with Account < maxAccount
-- Result is new Master increasing list [(Account,Amount)]
--  with Account < maxAccount

type Account = Int
type Amount = Integer

seqUpdate :: [(Account,Amount)] -> [(Account,Amount)]
             -> [(Account,Amount)]
seqUpdate = merge4c fst        fst
                    masterlt   mastereq  mastergt
                    masternext transnext
                    notrans    nomaster
                    getResult  initState

initState = ([],[])

maxAccount = maxBound :: Account

masterlt (out,[])    m t = (m:out,[])
    -- no transactions for this master
```

```haskell
masterlt (out,[cur]) m t = (cur:out,[])
    -- processed all transactions for this master

mastereq (out,[])        (ma,mb) (_,tc)
    = (out, [(ma,mb+tc)])   -- first transaction
mastereq (out,[(sa,sb)]) (_,_)   (_,tc)
    = (out, [(sa,sb+tc)])   -- subsequent transaction

mastergt (_,_) m t
    = error ("Transactions not ascending at " ++ show t)

masternext (_,_) ms = ms -- do not advance master on eq

transnext  (_,_) ts = let ys = tail ts
                      in if null ys then [(maxAccount,0)] else ys
                      -- advance transaction on eq
                      -- force master with final (maxAccount,0)

notrans (out,[])    m = (m:out,[])
notrans (out,[cur]) m = (m:cur:out,[])

nomaster ([],[])  t  -- only for empty master list
    = error ("Unmatched transaction " ++ show t)
nomaster (out,[]) (maxInt,_) -- transaction list ended
    = (out,[])
nomaster _        t
    = error ("Unmatched transaction " ++ show t)

getResult (nms,[]) = reverse nms
```

19.4.11 Recap

This case study illustrates the function generalization method. It begins with a simple Haskell program to merge two ascending lists of integers into third ascending list of integers. This program is generalized in a step by step fashion to produce a new program that is capable of carrying out any operation from the family of cosequential processing programs.

Although some members of the cosequential processing family can be rather complicated, the family has the characteristic that the primary driver for the algorithm can be concisely stated as a simple loop (i.e., recursive function).

19.5 What Next?

TODO

19.6 Chapter Source Code

TODO

19.7 Exercises

TODO

19.8 Acknowledgements

In Summer 2016, I adapted and revised the discussion of function generalization from my and Pallavi Tadepalli's paper "Using Function Generalization to Design a Cosequential Processing Framework" [57] to form partial Chapter 6, Developing Functional Programs, in the 2017 version of the ELIFP textbook [54]. (Cunningham, Liu, and Tadepalli [55] presents related work.) Schmid's work [153–155] on the generalization of object-oriented programs to design software frameworks motivated us to apply similar concepts to functional programs.

The cosequential processing problem was motivated by my use of Folk's presentation of that algorithm [76] in my File Structures (CSci 211) course at the University of Mississippi in the early 1990s. The papers by Dijkstra [64] and Dwyer [67] further informed that work.

In Summer 2018, I repurposed the partial Developing Functional Programs chapter and made it Chapter 19, Systematic Generalization, in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g.„ using CSS), constructing a bibliography (e.g.„ using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

19.9 Terms and Concepts

TODO

20 Problem Solving

20.1 Chapter Introduction

This Chapter is incomplete.

TODO: - Add Chapter Introduction - Give additional and improved examples. - Add What Next?, Chapter Source Code, and Exercises sections

20.2 Problem Solving Philosophy

I approach computing science with the following philosophy:

- Programming is the essence of computing science.
- Problem solving is the essence of programming.

Here I consider programming as the process of analyzing a problem and formulating a solution suitable for execution on a computer. The solution should be correct, elegant, efficient, and robust. It should be expressed in a manner that is understandable, maintainable, and reusable. The solution should balance generality and specificity, abstraction and concreteness.

In my view, programming is far more than just coding. It subsumes the concerns of algorithms, data structures, and software engineering. It uses programming languages and software development tools. It uses the intellectual tools of mathematics, logic, linguistics, and computing science theory. Etc.

20.3 Polya's Insights

The mathematician George Polya (1887–1985), a Professor of Mathematics at Stanford University, said the following in the preface to his book *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving* [142].

> Solving a problem means finding a way out of a difficulty, a way around an obstacle, attaining an aim which was not immediately attainable. Solving problems is the specific achievement of intelligence, and intelligence is the specific gift of mankind: solving problems can be regarded as the most characteristically human activity. ...
>
> Solving problems is a practical art, like swimming, or skiing, or playing the piano: you learn it only by imitation and practice. ... if you wish to learn swimming you have to go into the water, and if you wish to become a problem solver you have to solve problems.
>
> If you wish to derive the most profit from your effort, look out for such features of a problem at hand as may be useful in handling the problems to come. A solution that you have obtained by your own effort or one that you have read or heard, but have followed with

> real interest and insight, may become a *pattern* for you, a model that you can imitate with advantage in solving similar problems. . . .
>
> Our knowledge about any subject consists of *information* and *know-how*. If you have genuine *bonafide* experience of mathematical work on any level, elementary or advanced, there will be no doubt in your mind that, in mathematics, know-how is much more important than mere possession of information. . . .
>
> What is know-how in mathematics? The ability to solve problems—not merely routine problems but problems requiring some degree of independence, judgment, originality, creativity. Therefore, the first and foremost duty . . . in teaching mathematics is to emphasize *methodical work in problem solving.*

What Polya says for mathematics holds just as much for computing science.

In the book *How to Solve It* [141], Polya states four phases of problem solving. These steps are important for programming as well.

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan, checking each step.
4. Reexamine and reconsider the solution. (And, of course, reexamine the understanding of the problem, the plan, and the way the plan was carried out.)

20.4 Problem-Solving Strategies

There are many problem-solving strategies applicable to programming in general and functional programming in particular. We have seen some of these in the earlier chapters and will see others in later chapters. In this section, we highlight some of the general techniques.

20.4.1 Solve a more general problem first

The first strategy is to *solve a more general problem first.* That is, we solve a "harder" problem than the specific problem at hand, then use the solution of the "harder" problem to get the specific solution desired.

Sometimes a solution of the more general problem is actually easier to find because the problem is simpler to state, more symmetrical, or less obscured by special conditions. The general solution can often be used to solve other related problems.

Often the solution of the more general problem can actually lead to a more efficient solution of the specific problem.

Examples We have already seen one example of this technique: finding the first occurrence of an item in a list in Chapter 18.

First, we devised a program to find all occurrences in a list. Then we selected the first occurrence from the set of all occurrences. (Lazy evaluation of Haskell programs means that this use of a more general solution differs very little in efficiency from a specialized version.)

We have also seen several cases where we have generalized problems by adding one or more *accumulating parameters*. These "harder" problems can lead to more efficient tail recursive solutions.

For example, consider the tail recursive Fibonacci program we developed in Chapter 9. We added two extra arguments to the function.

```
fib2 :: Int -> Int
fib2 n | n >= 0 = fibIter n 0 1
    where
        fibIter 0 p q         = p
        fibIter m p q | m > 0 = fibIter (m-1) q (p+q)
```

Another approach is to use the *tupling* technique. Instead of adding extra arguments, we add extra results.

For example, in the Fibonacci program `fastfib` below, we compute `(fib n, fib (n+1))` instead of just `fib n`. This is a harder problem, but it actually gives us more information to work with and, hence, provides more opportunity for optimization. (We formally derive this solution in Chapter 26.)

```
fastfib :: Int -> Int
fastfib n | n >= 0 = fst (twofib n)

twofib :: Int -> (Int,Int)
twofib 0 = (0,1)
twofib n = (b,a+b)
    where (a,b) = twofib (n-1)
```

20.4.2 Solve a simpler problem first

The second strategy is to *solve a simpler problem first*. After solving the simpler problem, we then adapt or extend the solution to solve the original problem.

Often the mass of details in a problem description makes seeing a solution difficult. In the previous technique we made the problem easier by finding a more general problem to solve. In this technique, we move in the other direction: we find a more specific problem that is similar and solve it.

At worst, by solving the simpler problem we should get a better understanding of the problem we really want to solve. The more familiar we are with a problem,

the more information we have about it, and, hence, the more likely we will be able to solve it.

At best, by solving the simpler problem we will find a solution that can be easily extended to build a solution to the original problem.

Examples Consider a program to convert a positive integer of up to six digits to a string consisting of the English words for that number. For example, `369027` yields the string:

```
three hundred and sixty-nine thousand and twenty-seven
```

To deal with the complexity of this problem, we can work as follows:

a. Solve the problem of converting a two-digit number to words. (The single digit numbers and numbers in teens are special cases.)
b. Then extend the two-digit solution to three digits. (This can basically use the solution to part "a" twice.)
c. Then extend three-digit solution to to six digits. (This can basically use the solution to part "b" twice.)

See Section 4.1 of the classic Bird and Wadler textbook [15] for the details of this problem and a solution.

TODO: May want to create some code for this problem rather than just refer to an old textbook.

The process of generalizing first-order functions into higher-order functions is another example of this "solve a simpler problem first" strategy. Recall how we motivated the development of the higher-order library functions such as `map`, `filter`, and `foldr` in Chapter 15. Also consider the function generalization approach used in the cosequential processing case study in Chapter 19.

20.4.3 Reuse off-the-shelf solutions to standard subproblems

The third strategy is to *reuse an off-the-shelf solutions to a standard subproblem.*

We have been doing this all during this semester, especially since we began began studying polymorphism and higher-order functions.

The basic idea is to identify standard patterns of computation (e.g., standard Prelude functions such as `length`, `take`{.haskell, `zip`{.haskell, `map`{.haskell, `filter`{.haskell, `foldr`{.haskell) that will solve some aspects of the problem and then combine (e.g., using function composition) these standard patterns with your own specialized functions to construct a solution to the problem.

Examples We have seen several examples of this technique in this textbook and its exercises.

See section 4.2 of the classic Bird and Wadler textbook [15] for a case study that develops a package of functions to do arithmetic on variable length integers. The functions take advantage of several of the standard Prelude functions.

20.4.4 Solve a related problem

The fourth strategy is to *solve a related problem*. After solving the related problem, we then transform the solution of the related problem to get a solution to the original problem.

Perhaps we can find an entirely different problem formulation (i.e., stated in different terms) for which we can readily find a solution. Then that solution can be converted into a solution to the problem at hand.

Examples For example, we can recast a problem in terms of mathematical or logical frameworks (e.g., sets, relations, graphs, finite state machines, grammars, algebraic structures, differential equations, etc.), solve the corresponding problem in those terms, and then interpret the result for the original problem. The simplification provided by the frameworks may reveal solutions that are obscured in the details of the problem. We can also take advantage of the theory and techniques that have been found previously for the mathematical frameworks.

Consider the problem of breaking a string of text into the list of its component lines.

This is not trivial. However, the "inverse" problem is trivial. All that is needed to convert a list of lines to a string of text is to insert linefeed characters between the lines.

We can first solve the inverse problem (line-folding) and then use it to calculate what the line-breaking program must be. (See Section 4.3 of the Bird and Wadler textbook [15] and a Chapter 27 in this textbook.)

20.4.5 Separate concerns

The fifth strategy is to *separate concerns*. That is, we partition the problem into logically separate problems, solve each problem separately, then combine the solutions to the subproblems to construct a solution to the problem at hand.

As we have seen in the above strategies, when a problem is complex and difficult to attack directly, we search for simpler, but related, problems to solve, then build a solution to the complex problem from the simpler problems.

Examples We have seen examples of this approach in earlier chapters and homework assignments. We separated concerns when we used stepwise refinement to develop a square root function, data abstraction in the rational number case study, and function pipelines.

Consider the development of a program to print a calendar for any year in various formats. We can approach this problem by first separating it into two independent subproblems:

a. building a calendar
b. formatting the output

After solving each of these simpler problems, the more complex problem can be solved easily by combining the two solutions. (See Section 4.5 of the classic Bird and Wadler textbook [15] for the details of this problem and a solution.)

20.4.6 Divide and conquer

The sixth strategy is *divide and conquer*. This is a special case of the "solve a simpler problem first" strategy. In this technique, we must divide the problem into subproblems that are the same as the original problem except that the size of the input is smaller.

This process of division continues recursively until we get a problem that can be solved trivially, then we combined we reverse the process by combining the solutions to subproblems to form solutions to larger problems.

Examples Examples of divide and conquer from earlier chapters include the logarithmic exponentiation function `expt3` in Chapter 9 and the merge sort function `msort` in Chapter 17.

Another common example of the divide and conquer approach is binary search. (See Section 6.4.3 of the classic Bird and Wadler textbook [15].)

Chapter 17 examines divide and conquer algorithms in terms of a higher order function that captures the pattern.

There are, of course, other strategies that can be used to approach problem solving.

20.5 What Next?

TODO

20.6 Chapter Source Code

TODO

20.7 Exercises

TODO

20.8 Acknowledgements

In 2016 and 2017, I adapted and revised my previous notes to form Chapter 7, More List Processing and Problem Solving, in the 2017 version of this textbook. In particular, I drew the information on Problem Solving from:

- chapter 10 of my *Notes on Functional Programming with Haskell* for discussion of problem-solving techniques in section 7.4

  Chapter 10 drew on chapters 4 and 6 of Bird [15], chapter 4 of [173], and two of Polya's books [141,142].

- part of chapter 12 of *Notes on Functional Programming with Haskell* for discussion of the tupling technique incorporated into subsection 7.4.2.1

  The tupling discussion originally drew from Bird [15] and Hoogerwoord [98].

In Summer 2018, I divided the previous More List Processing and Problem Solving chapter back into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 7.2-7.3 became the basis for new Chapter 18, More List Processing, and previous section 7.4 (essentially the two items above) became the basis for new Chapter 20 (this chapter), Problem Solving.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

20.9 Terms and Concepts

Problem solving, Polya, information, know-how, bonafide experience, problem-solving strategies, solve a more general (harder) problem first, accumulating parameters, tupling, solve a simpler problem first, reuse an off-the-shelf solution, higher-order functions, stepwise refinement, data abstraction, solve a related problem, separate concerns, divide and conquer.

<

21 Algebraic Data Types

21.1 Chapter Introduction

The previous chapters have primarily used Haskell's primitive types along with tuples, lists, and functions.

The goals of this chapter (21) are to:

- describe how to define and use of Haskell's (user-defined) algebraic data types
- introduce improvements in error-handling using `Maybe` and `Either` types
- present a few larger programming projects

Algebraic data types enable us to conveniently leverage the power of the type system to write safe programs. We extensively these in the remainder of this textbook.

The Haskell source module for this chapter is in file `AlgDataTypes.hs`.

TODO: It might be better to factor the source code into multiple files.

21.2 Concepts

An *algebraic data type* [24,173,204] is a type formed by combining other types, that is, it is a *composite* data type. The data type is created by an algebra of operations of two primary kinds:

- a *sum* operation that constructs values to have one variant among several possible variants. These sum types are also called *tagged*, *disjoint union*, or *variant* types.

  The combining operation is the alternation operator, which denotes the choice of one but not both between two alternatives.

- a *product* operation that combines several values (i.e., *fields*) together to construct a single value. These are *tuple* and *record* types.

  The combining operation is the Cartesian product from set theory.

We can combine sums and products recursively into arbitrarily large structures.

An *enumerated type* is a sum type in which the constructors take no arguments. Each constructor corresponds to a single value.

Aside: ADT confusion

Although sometimes the acronym ADT is used for both, an *algebraic data type* is a different concept from an *abstract data type* [61,203].

- We specify an algebraic data type with its *syntax* (i.e., structure)—with rules on how to compose and decompose them.

- We specify an abstract data type with its *semantics* (i.e., meaning)—with rules about how the operations behave in relation to one another.

  The modules we build with abstract interfaces, contracts, and data abstraction, such as the Rational Arithmetic modules from Chapter 7, are abstract data types.

Perhaps to add to the confusion, in functional programming we sometimes use an algebraic data type to help define an abstract data type. We do this in the Carrie's Candy Bowl project at the end of this chapter. We consider these techniques more fully in Chapter 22.

21.3 Haskell Algebraic Data Types

21.3.1 Declaring data types

In addition to the built-in data types we have discussed, Haskell also allows the definition of new data types using declarations of the form:

`data` **Datatype a*1* a*2*** $\cdots$ **a*n*** `=` **Cnstr*1*** `|` **Cnstr*2*** `|` $\cdots$ `|` **Cnstr*m***

where:

- **Datatype** is the name of a new type constructor of *arity **n** (n $\geq$ 0).* As with the built-in types, the name of the data type must begin with an uppercase letter.
- **a*1***, **a*2***, $\cdots$ **a*n*** are distinct type variables representing the ***n*** parameters of the data type. These begin with lowercase letters (by convention at the beginning of the alphabet).
- **Cnstr*1***, **Cnstr*2***, $\cdots$, **Cnstr*m*** are the ***m*** ($m \geq$ 1$) data constructors that describe the ways in which the elements of the new data type are constructed. These begin with uppercase letters.

The `data` definition can also end with an optional `deriving` that we discuss below.

21.3.2 Declaring type `Color`

For example, consider a new data type `Color` whose possible values are the colors on the flag of the USA. The names of the data constructors (the color constants in this case) must also begin with capital letters.

```
data Color = Red | White | Blue
             deriving (Show, Eq)
```

`Color` is an example of an *enumerated type*, a *sum* type that consists of a finite sequence of *nullary* (i.e., the arity—number of parameters—is zero) data constructors.

We can use the type and data constructor names defined with `data` in declarations, patterns, and expressions in the same way that the built-in types can be used.

```
isRed :: Color -> Bool
isRed Red = True
isRed _   = False
```

Data constructors can also have associated values. For example, the constructor `Grayscale` below takes an integer value.

```
data Color' = Red' | Blue' | Grayscale Int
              deriving (Show, Eq)
```

Constructor `Grayscale` implicitly defines a constructor function with the type.

21.3.3 Deriving class instances

The optional `deriving` clauses in the above definitions of `Color` and `Color'` are very useful. They declare that these new types are automatically added as instances of the type classes listed.

Note: Chapter 23 explores the concepts of type class, instance, and overloading in more depth.

In the above cases, `Show` and `Eq` enable objects of type `Color` to be converted to a `String` and compared for equality, respectively.

The Haskell compiler derives the body of an instance syntactically from the data type declaration. It can derive instances for classes `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, and `Show`.

The derived instances of type class `Eq` include the `(==)` and `(/=)` methods.

Type class `Ord` extends `Eq`. In addition to `(==)` and `(/=)`, a derived instance of `Ord` also includes the `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min` methods. The ordered comparison operators use the order of the constructors given in the `data` statement, from smallest to largest, left to right. These comparison operators are strict in both arguments.

Similarly, a derived `Enum` instance assigns successive integers to the constructors increasing from 0 at the left. In addition to this, a derived instance of `Bounded` assigns `minBound` to the leftmost and `maxBound` to the rightmost.

The derived `Show` instance enables the function `show` to convert the data type to a syntactically correct Haskell expression consisting of only the constructor names, parentheses, and spaces. Similarly, `Read` enables the function `read` to parse such a string into a value of the data type.

For example, the data type `Bool` might be defined as:

```
data Bool = False | True
            deriving (Ord, Show)
```

Thus `False < True` evaluates to `True` and `False > True` evaluates to `False`. If `x == False`, then `show x` yields the string `False`.

21.3.4 Exploring more example types

Consider a data type `Point` that has a type parameter. The following defines a polymorphic type; both of the values associated with the constructor `Pt` must be of type `a`. Constructor `Pt` implicitly defines a constructor function of type `a -> a -> Point a`.

```
data Point a = Pt a a
               deriving (Show, Eq)
```

As another example, consider a polymorphic set data type that represents a set as a list of values as follows. Note that the name `Set` is used both as the type constructor and a data constructor. In general, we should not use a symbol in multiple ways. It is acceptable to double use only when the type has only one constructor.

```
data Set a = Set [a]
             deriving (Show, Eq)
```

Now we can write a function `makeSet` to transform a list into a `Set`. This function uses the function `nub` from the `Data.List` module to remove duplicates from a list.

```
makeSet :: Eq a => [a] -> Set a
makeSet xs = Set (nub xs)
```

As we have seen previously, programmers can also define type synonyms. As in user-defined types, synonyms may have parameters. For example, the following might define a matrix of some polymorphic type as a list of lists of that type.

```
type Matrix a = [[a]]
```

We can also use special types to encode error conditions. For example, suppose we want an integer division operation that returns an error message if there is an attempt to divide by 0 and returns the quotient otherwise. We can define and use a union type `Result` as follows:

```
data Result a = Ok a | Err String
                deriving (Show, Eq)

divide :: Int -> Int -> Result Int
divide _  0 = Err "Divide by zero"
divide x  y = Ok (x `div` y)
```

Then we can use this operation in the definition of another function `f` that returns the maximum `Int` value `maxBound` when a division by 0 occurs.

```
f :: Int -> Int -> Int
f x y  = return (divide x y)
         where return (Ok z)  = z
               return (Err s) = maxBound
```

The auxiliary function `return` can be avoided by using the Haskell `case` expression as follows:

```
f' x y =
    case divide x y of
        Ok z  -> z
        Err s -> maxBound
```

This case expression evaluates the expression `divide x y`, matches its result against the patterns of the alternatives, and returns the right-hand-side of the first matching patter.

Later in this chapter we discuss the `Maybe` and `Either` types, two polymorphic types for handling errors defined in the Prelude.

21.4 Recursive Data Types

Types can also be recursive.

21.4.1 Defining a binary tree type

For example, consider the user-defined type `BinTree`, which defines a binary tree with values of a polymorphic type.

```
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
                 deriving (Show, Eq)
```

This data type represents a binary tree with a value in each node. The tree is either "empty" (denoted by `Empty`) or it is a "node" (denoted by `Node`) that consists of a value of type `a` and "left" and "right" subtrees. Each of the subtrees must themselves be objects of type `BinTree`.

Thus a binary tree is represented as a three-part "record" as shown in on the left side of Figure 21.1. The left and right subtrees are represented as nested binary trees. There are no explicit "pointers".

Consider a function `flatten` to return the list of the values in binary tree in the order corresponding to a left-to-right in-order traversal. Thus expression

```
flatten (Node (Node Empty 3 Empty) 5
              (Node (Node Empty 7 Empty) 1 Empty))
```

yields `[3,5,7,1]`.

```
flatten :: BinTree a -> [a]
flatten Empty        = []
flatten (Node l v r) = flatten l ++ [v] ++ flatten r
```

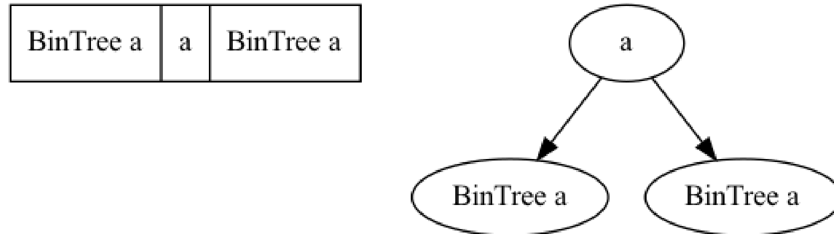


Figure 21.1: Binary tree `BinTree`.

The second leg of `flatten` requires two recursive calls. However, as long as the input tree is finite, each recursive call receives a tree that is simpler (3.g., shorter) than the input. Thus all recursions eventually terminate when `flatten` is called with an `Empty` tree.

Function `flatten` can be rendered more efficiently using an accumulating parameter and cons as in the following:

```
flatten' :: BinTree a -> [a]
flatten' t = inorder t []
             where inorder Empty xs        = xs
                   inorder (Node l v r) xs =
                       inorder l (v : inorder r xs)
```

Auxiliary function `inorder` builds up the list of values from the right using cons.

To extend the example further, consider a function `treeFold` that folds an associative operation `op` with identity element `i` through a left-to-right in-order traversal of the tree.

```
treeFold :: (a -> a -> a) -> a -> BinTree a -> a
treeFold op i Empty        = i
treeFold op i (Node l v r) = op (op (treeFold op i l) v)
                                (treeFold op i r)
```

21.4.2 Exporting types from modules

If an algebraic data type is defined in a module, we can export the type and make it available to users of the module. Suppose the `BinTree` type and the functions above are defined in a Haskell module named `BinaryTrees`. Then the following module header would export the type `BinTree`, the three explicitly defined functions, and the functions generated for the `Eq` and `Show` classes.

```
module BinaryTrees
    ( BinTree, flatten, flatten', treeFold )
```

```
where  -- implementation details of type and functions
```

This module definition makes the type `BinTree` and its two constructors `Node` and `Empty` available for use in a module that imports `BinaryTrees`.

If we want to make the type `BinTree` available but not its constructors, we can use the following module header:

```
module BinaryTrees
    ( BinTree(..), flatten, flatten', treeFold )
where  -- implementation details of type and functions
```

With `BinTree(..)` in the export list, `BinTree` values can only be constructed and examined by functions defined in the module (including the automatically generated functions). Outside the module, the `BinTree` values are "black boxes" that can be passed around or stored.

If the `BinaryTrees` module is designed and implemented as an information-hiding module as described in Chapter 7, then we also call this an *abstract data type*. We discuss these data abstractions in more detail in Chapter 22.

21.4.3 Defining an alternative binary tree type

Now let's consider a slightly different formulation of a binary tree: a tree in which values are only stored at the leaves.

```
data Tree a = Leaf a | Tree a :^: Tree a
              deriving (Show, Eq)
```

This definition introduces the constructor function name `Leaf` as the constructor for leaves and the infix construction operator "`:^:`" as the constructor for internal nodes of the tree. (A constructor operator symbol must begin with a colon.)

These constructors allow such trees to be defined conveniently. For example, the tree

```
((Leaf 1 :^: Leaf 2) :^: (Leaf 3 :^: Leaf 4))
```

generates a complete binary tree with height 3 and the integers 1, 2, 3, and 4 at the leaves.

Suppose we want a function `fringe`, similar to function `flatten` above, that displays the leaves in a left-to-right order. We can write this as:

```
fringe :: Tree a -> [a]
fringe (Leaf v)  = [v]
fringe (l :^: r) = fringe l ++ fringe r
```

As with `flatten` and `flatten'` above, function `fringe` can also be rendered more efficiently using an accumulating parameter as in the following:

```
fringe' :: Tree a -> [a]
fringe' t = leaves t []
```

```haskell
            where leaves (Leaf v)  = ((:) v)
                  leaves (l :^: r) = leaves l . leaves r
```

Auxiliary function `leaves` builds up the list of leaves from the right using cons.

21.5 Error-handling with `Maybe` and `Either`

Before we examine `Maybe` and `Either`, let's consider a use case.

21.5.1 Handling null references

An *association list* is a list of pairs in which the first component is some *key* (e.g., a string) and the second component is the *value* associated with that key. It is a simple form of a *map* or *dictionary* data structure.

Suppose we have an association list that maps the name of a student (a key) to the name of the student's academic advisor (a value). The following function `lookup'` carries out the search recursively.

```haskell
lookup' :: String -> [(String,String)] -> String
lookup' key ((x,y):xys)
    | key == x  =  y
    | otherwise =  lookup' key xys
```

But what do we do when the key is not in the list (e.g., the list is empty)? How do we define a leg for `lookup' key []` ?

1. Leave the function undefined for that pattern?

   In this case, evaluation will halt with a "non-exhaustive pattern" error message.

2. Put in an explicit `error` call with a custom error message?

3. Return some default value of the advisor such as `"NONE"`?

4. Return a *null reference*?

The first two approaches either halt the entire program or require use of the exception-handling mechanism. However, in any language, both abnormal termination and exceptions should be avoided except in cases in which the program is unable to continue. The lack of an assignment of a student to an advisor is likely not such an extraordinary situation.

Exceptions break referential transparency and, hence, negate many of the advantages of purely functional languages such as Haskell. In addition, Haskell programs can only catch exceptions in `IO` programs (i.e., the outer layers that handle input/output).

The third approach only works when there is some value that is not valid. This is not a very general approach.

The fourth approach, which is not available in Haskell, can be an especially unsafe programming practice. British computing scientist Tony Hoare, who introduced the null reference into the Algol type system in the mid-1960s, calls that his “billion dollar mistake” [97] because it “has led to innumerable errors, vulnerabilities, and system crashes”.

What is a safer, more general approach than these?

21.5.2 Introducing `Maybe` and `Either`

Haskell includes the union type `Maybe` (from the Prelude and `Data.Maybe`) which can be used to handle such cases.

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

The `Maybe` algebraic data type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented by `Just a`) or it is empty (represented by `Nothing`).

The `Maybe` type is a good way to handle errors or exceptional cases without resorting to an `error` call.

Now we can define a general version of `lookup'` using a `Maybe` return type. (This is essentially function `lookup` from the Prelude.)

```
lookup'' :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup'' key [] =  Nothing
lookup'' key ((x,y):xys)
   | key == x    =  Just y
   | otherwise   =  lookup'' key xys
```

Suppose `advisorList` is an association list pairing students with their advisors and `defaultAdvisor` is the advisor the student should consult if no advisor is officially assigned. We can look up the advisor with a call to `lookup` and then pattern match on the `Maybe` value returned. (Here we use a `case` expression.)

```
whoIsAdvisor :: String -> String
whoIsAdvisor std =
    case lookup std advisorList of
        Nothing   -> defaultAdvisor
        Just prof -> prof
```

The `whoIsAdvisor` function just returns a default value in place of `Nothing`. The function

```
fromMaybe :: a -> Maybe a -> a
```

supported by the `Data.Maybe` library has the same effect. Thus we can rewrite `whoIsAdvisor` as follows:

```haskell
whoIsAdvisor' std =
    fromMaybe defaultAdvisor $ lookup std advisorList
```

Alternatively, we could use `Data.Maybe` functions such as:

```haskell
isJust    :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
fromJust  :: Maybe a -> a   -- error if Nothing
```

This allows us to rewrite `whoIsAdvisor` as follows:

```haskell
whoIsAdvisor'' std =
    let ad = lookup std advisorList
    in if isJust ad then fromJust ad else defaultAdvisor
```

If we need more fine-grained error messages, then we can use the union type `Either` defined as follows:

```haskell
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

The `Either a b` type represents values with two possibilities: a `Left a` or `Right b`. By convention, a `Left` constructor usually contains an error message and a `Right` constructor a *correct* value.

As with `fromMaybe`, we can use similar `fromRight` and `fromLeft` functions from the `Data.Either` library to extract the `Right` or `Left` values or to return a default value when the value is represented by the other constructor.

```haskell
fromLeft  :: a -> Either a b -> a
fromRight :: b -> Either a b -> b
```

Library module `Data.Either` also includes functions to query for the presence of the two constructors.

```haskell
isLeft    :: Either a b -> Bool
isRight   :: Either a b -> Bool
```

21.5.3 Considering other languages

Most recently designed languages include a *maybe* or *option* type [214]. Scala [131,151] has an `Option` case class [29:4,51], Rust [110,150] has an `Option` enum, and Swift has an `Optional` class, all of which are similar to Haskell's `Maybe`. The functional languages Idris [18,19], Elm [60,70], and PureScript [79,143] also have Haskell-like `Maybe` algebraic data types.

The concept of *nullable type* [212] is closely related to the option type. Several older languages support this concept (e.g., `Optional` in Java 8, `None` in Python [144,146], and `?` type annotations in C#).

When programming in an object-oriented language that does not provide an option/maybe type, a programmer can often use the *Null Object design pattern*

[162,213,229] to achieve a similar result. This well-known pattern seeks to "encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior" [162]. That is, the object must be of the correct type. It must be possible to apply all operations on that type to the object, but the operations should have neutral behaviors, with no side effects. The null object should actively do nothing!

21.6 What Next?

This chapter (21) added Haskell's algebraic data types to our programming toolbox. Chapter 22 sharpens the data abstraction tools introduced in Chapter 7 by using algebraic data types from this chapter. Chapter 23 adds type classes and overloading to the toolbox.

The remainder of this chapter includes a number of larger exercises and projects.

21.7 Chapter Source Code

The Haskell source module for this chapter is in file `AlgDataTypes.hs`.

21.8 Exercises

1. For trees of type `Tree`, implement a tree-folding function similar to `treeFold`.

2. For trees of type `BinTree`, implement a version of `treeFold` that uses an accumulating parameter. (Hint: `foldl`.)

3. In a binary search tree all values in the left subtree of a node are less than the value at the node and all values in the right subtree are greater than the value at the node.

   Given binary search trees of type `BinTree`, implement the following Haskell functions:

   a. `makeTree` that takes a list and returns a perfectly balanced (i.e., minimal height) `BinTree` such that `flatten (makeTree xs) = sort xs`. Prelude function `sort` returns its argument rearranged into ascending order.

   b. `insertTree` that takes an element and a `BinTree` and returns the `BinTree` with the element inserted at an appropriate position.

   c. `elemTree` that takes an element and a `BinTree` and returns `True` if the element is in the tree and `False` otherwise.

   d. `heightTree` that takes a `BinTree` and returns its height. Assume that height means the number of levels in the tree. (A tree consisting of exactly one node has a height of `1`.)

e. `mirrorTree` that takes a `BinTree` and returns its mirror image. That is, it takes a tree and returns the tree with the left and right subtrees of every node swapped.

f. `mapTree` that takes a function and a `BinTree` and returns the `BinTree` of the same shape except each node's value is computed by applying the function to the corresponding value in the input tree.

g. `showTree` that takes a `BinTree` and displays the tree in a parenthesized, left-to-right, in-order traversal form. (That is, the traversal of a tree is enclosed in a pair of parentheses, with the traversal of the left subtree followed by the traversal of the right subtree.)

Extend the package to support both insertion and deletion of elements. Keep the tree balanced using a technique such the AVL balancing algorithm.

4. Implement the package of functions described in the previous exercise for the data type `Tree`.

5. Each node of a general (i.e., multiway) tree consists of a label and a list of (zero or more) subtrees (each a general tree). We can define a general tree data type in Haskell as follows:

```
data Gtree a = Node a [Gtree a]
```

For example, tree `(Node 0 [ ])` consists of a single node with label `0`; a more complex tree `Node 0 [Node 1 [ ], Node 2 [ ], Node 3 []]` consists of root node with three single-node subtrees.

Implement a "map" function for general trees, i.e., write Haskell function

```
mapGtree :: (a -> b) -> Gtree a -> Gtree b
```

that takes a function and a `Gtree` and returns the `Gtree` of the same shape such that each label is generated by applying the function to the corresponding label in the input tree.

6. We can introduce a new Haskell type for the natural numbers (i.e., nonnegative integers) with the statement

```
data Nat = Zero | Succ Nat
```

where the constructor `Zero` represents the value 0 and constructor `Succ` represents the "successor function" from mathematics. Thus `(Succ Zero)` denotes 1, `(Succ (Succ Zero))` denotes 2, and so forth. Implement the following Haskell functions.

a. `intToNat` that takes a nonnegative `Int` and returns the equivalent `Nat`, for example, `intToNat 2` returns `Succ (Succ Zero)`.

b. `natToInt` that takes a `Nat` and returns the equivalent value of type `Int`, for example, `natToInt Succ (Succ Zero)` returns `2`.

c. `addNat` that takes two `Nat` values and returns their sum as a `Nat`. This function cannot use integer addition.

d. `mulNat` that takes two `Nat` values and returns their product as a `Nat`. This function cannot use integer multiplication or addition.

e. `compNat` that takes two `Nat` values and returns the value -1 if the first is less than the second, 0 if they are equal, and 1 if the first is greater than the second. This function cannot use the integer comparison operators.

7. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Att (Seq a) a
```

`Nil` represents the empty sequence. `Att xz y` represents the sequence in which *last element* `y` is "attached" at the right end of the *initial sequence* `xz`.

Note that `Att` is similar to the ordinary "cons" (`:`) for Haskell lists except that elements are attached at the opposite end of the sequences. `(Att (Att (Att Nil 1) 2) 3)` represents the same sequence as the ordinary list `(1:(2:(3:[])))`.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists.

a. `lastSeq` takes a nonempty `Seq` and returns its last (i.e., rightmost) element.

b. `initialSeq` takes a nonempty `Seq` and returns its initial sequence (i.e., sequence remaining after the last element removed).

c. `lenSeq` takes a `Seq` and returns the number of elements that it contains.

d. `headSeq` takes a nonempty `Seq` and returns its head (i.e., leftmost) element.

e. `tailSeq` takes a nonempty `Seq` and returns the `Seq` remaining after the head element is removed.

f. `conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

g. `appSeq` takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

h. `revSeq` takes a `Seq` and returns the `Seq` with the same elements in reverse order.

i. `mapSeq` takes a function and a `Seq` and returns the `Seq` resulting from applying the function to each element of the sequence in turn.

j. `filterSeq` that takes a predicate and a `Seq` and returns the `Seq` containing only those elements that satisfy the predicate.

k. `listToSeq` takes an ordinary Haskell list and returns the `Seq` with the same values in the same order (e.g., `headSeq (listToSeq xs) = head xs` for nonempty `xs`.)

l. `seqToList` takes a `Seq` and returns the ordinary Haskell list with the same values in the same order (e.g., `head (seqToList xz) = headSeq xz` for nonempty `xz`.)

8. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Unit a | Cat (Seq a) (Seq a)
```

The constructor `Nil` represents the empty sequence; `Unit` represents a single-element sequence; and `Cat` represents the "concatenation" (i.e., append) of its two arguments, the second argument appended after the first.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists. (Do not convert back and forth to lists.)

a. `toSeq` that takes a list and returns a corresponding `Seq` that is balanced.

b. `fromSeq` that takes a `Seq` and returns the corresponding list.

c. `appSeq` that takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

d. `conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

e. `lenSeq` that takes a `Seq` and returns the number of elements that it contains.

f. `revSeq` that takes a `Seq` and returns a `Seq` with the same elements in reverse order.

g. `headSeq` that takes a nonempty `Seq` and returns its head (i.e., leftmost or front) element. (Be careful!)

h. `tailSeq` that takes a nonempty `Seq` and returns the `Seq` remaining after the head is removed.

i. `normSeq` that takes a `Seq` and returns a `Seq` with unnecessary embedded `Nil` values removed. (For example, `normSeq (Cat (Cat Nil (Unit 1)) Nil)` returns `(Unit 1)`.)

j. `eqSeq` that takes two `Seq` "trees" and returns `True` if the sequences of values are equal and returns `False` otherwise. Note that two `Seq` "trees" may be structurally different yet represent the same sequence of values.

   For example, `(Cat Nil (Unit 1))` and `(Cat (Unit 1) Nil)` have the same sequence of values (i.e., `[1]`). But `(Cat (Unit 1) (Unit 2))` and `(Cat (Unit 2) (Unit 1))` do not represent the same sequence of values (i.e., `[1,2]` and `[2,1]`, respectively).

   Also `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` has the same sequence of values as `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` (i.e., `[1,2,3]`).

In general what are the advantages and disadvantages of representing lists this way?

21.9 Carrie's Candy Bowl Project

21.9.1 Problem description and initial design

Carrie, the Department's Administrative Assistant, has a candy bowl on her desk. Often she fills this bowl with candy, but the contents are quickly consumed by students, professors, and staff members. At a particular point in time, the candy bowl might contain several different kinds of candy with one or more pieces of each kind or it might be empty. Over time, the kinds of candy in the bowl varies.

In this project, we model the candy, the candy bowl, and the "operations" that can be performed on the bowl and develop it as a Haskell module.

What about the candy?

In general, we want to be able to identify how many pieces of candy we have of a particular kind (e.g., we may have two Snickers bars and fourteen Hershey's Kisses) but do not need to distinguish otherwise between the pieces. So distinct identifiers for the different kinds of candy should be sufficient.

We can represent the different kinds of candy in several different ways. We could use strings, integer codes, the different values of an enumerated type, etc. In different circumstances, we might want to use different representations.

Thus we model the kinds of candy to be a polymorphic parameter of the candy bowl. However, we can contrain the polymorphism on the kinds of candy to be a Haskell type that can be compared for equality (i.e., in class `Eq`) and converted to a string so that it can be displayed (i.e., in class `Show`).

What about the candy bowl itself?

A candy bowl is some type of collection of pieces of candy with several possible representations. We could use a list (either unordered) of the pieces of candy,

an association list (unordered or ordered) pairing the kinds of candy with the numbers of pieces of each, a Data.Map structure (from the Haskell library), or some other data structure.

Thus we want to allow the developers of the candy bowl to freely choose whatever representation they wish or perhaps to provide several different implementations with the same interface. We will leave this hidden inside the Haskell module that implements an abstract data type.

Thus, a Haskell module that implements the candy bowl can define a polymophic algebraic data type CandyBowl a and export its name, but not export the implementation details (i.e., the constructors) of the type. For example, a represenation built around a list of kinds of candy could be defined as:

```haskell
data CandyBowl a = Bowl [a]
```

Or a representation using an association list can be defined as:

```haskell
data CandyBowl a = Bowl [(a,Int)]
```

Thus, to export the CandyBowl but hide the details of the representation, the module would have a header such as:

```haskell
module CarrieCandyBowl
    ( CandyBowl(..), -- function names exported
    )
where  -- implementation details of type and functions
```

Some of the possible representations require the ability to order the types of candy in some way. Thus, we further constrain the polymorphic type parameter to class Ord instead of simply Eq. (Above, we also constrained it to class Show.)

21.9.2 Carrie's Candy Bowl project exercises

Your task for this project to develop a Haskell module CarrieCandyBowl (in a file CarrieCandyBowl.hs), as described above. You must choose an appropriate internal representation for the data type CandyBowl and implement the public operations (functions) defined below. In addition to exporting the public functions and data type name, the module may contain whatever other internal data and function definitions needed for theimplemenation.

An initial Haskell source code for this project is in file CarrieCandyBowl_skeleton.hs.

You may use a function you have completed to implement other functions in the list (as long as you do not introduce circular definitions).

1. `newBowl :: (Ord a,Show a) => CandyBowl a`
   creates a new empty candy bowl.
2. `isEmpty :: (Ord a,Show a) => CandyBowl a -> Bool`
   returns True if and only if the bowl is empty.

3. `putIn :: (Ord a,Show a) => CandyBowl a -> a -> CandyBowl a`
   adds one piece of candy of the given kind to the bowl.

   For example, if we use strings to represent the kinds, then

   ```
   putIn bowl "Kiss"
   ```

   adds one piece of candy of kind `"Kiss"` to the `bowl`.

4. `has :: (Ord a,Show a) => CandyBowl a -> a -> Bool`
   returns `True` if and only if one or more pieces of the given kind of candy is in the bowl.

5. `size :: (Ord a,Show a) => CandyBowl a -> Int`
   returns the total number of pieces of candy in the bowl (regardless of kind).

6. `howMany :: (Ord a,Show a) => CandyBowl a -> a -> Int`
   returns the count of the given kind of candy in the bowl.

7. `takeOut :: (Ord a,Show a) => CandyBowl a -> a -> Maybe (CandyBowl a)`
   attempts to remove one piece of candy of the given kind from the bowl (so it can be eaten). If the bowl contains a piece of the given kind, the function returns the value `Just bowl`, where `bowl` is the bowl with the piece removed. If the bowl does not contain such a piece, it returns the value `Nothing`.

8. `eqBowl :: (Ord a,Show a) => CandyBowl a -> CandyBowl a -> Bool`
   returns `True` if and only if the two bowls have the same contents (i.e., the same kinds of candy and the same number of pieces of each kind).

9. `inventory :: (Ord a,Show a) => CandyBowl a -> [(a,Int)]`
   returns a Haskell list of pairs `(k,n)`, where each kind `k` of candy in the bowl occurs once in the list with `n > 0`. The list should be arranged in *ascending order* by kind.

   For example, if there are two `"Snickers"` and one `"Kiss"` in the bowl, the list returned would be `[("Kiss",1),("Snickers",2)]`.

10. `restock :: (Ord a,Show a) => [(a,Int)] -> CandyBowl a`
    creates a new bowl such that for any `bowl`:

    ```
    eqBowl (restock (inventory bowl)) bowl  ==  True
    ```

11. `combine :: (Ord a,Show a) => CandyBowl a -> CandyBowl a -> CandyBowl a`
    pours the two bowls together to form a new “larger” bowl.

12. `difference :: (Ord a,Show a) => CandyBowl a -> CandyBowl a -> CandyBowl a`
    returns a bowl containing the pieces of candy in the first bowl that are not in the second bowl.

    For example, if the first bowl has four `"Snickers"` and the second has one `"Snickers"`, then the result will have three `"Snickers"`.

13. `rename :: (Ord a,Show a) => CandyBowl a -> (a -> b) -> CandyBowl b` takes a bowl and a renaming function, applies the renaming function to all the kind values in the bowl, and returns the modified bowl.

    For example, for some mysterious reason, we might want to reverse the strings for the kind names: `f xs = reverse xs`. Thus `"Kiss"` would become `"ssiK"`. Then `rename f bowl` would do the reversing of all the names.

21.9.3 Candy Bowl alternative exercises

TODO: Maybe specify reimplentations with a different data rep, perhaps requiring a map.

21.10 Sandwich DSL Project

21.10.1 Project Introduction

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work.

These special-purpose languages are often called *domain-specific languages* (or DSLs) [53]. (For more discussion of the DSL concepts, terminology, and techniques, see the introductory chapter of the Notes on Domain-Specific Languages [53].)

In this project, we design and implement a simple *internal DSL* [53]. This DSL describes simple "programs" using a set of Haskell algebraic data types. We express a program as an *abstract syntax tree* (AST) [53] using the DSLs data types.

In this project, we first build a package of functions for creating and manipulating the abstract syntax trees. We then extend the package to translate the abstract syntax trees to a sequence of instructions for a simple "machine".

21.10.2 Developing the Sandwich DSL

Suppose Emerald de Gassy, the owner of the Oxford-based catering business Deli-Gate, hires us to design a domain-specific language (DSL) for describing sandwich platters. The DSL scripts will direct Deli-Gate's robotic kitchen appliance SueChef (Sandwich and Utility Electronic Chef) to assemble platters of sandwiches.

In discussing the problem with Emerald and the Deli-Gate staff, we discover the following:

- A sandwich platter consists of zero or more sandwiches. (Zero? Why not! Although a platter with no sandwiches may not be a useful, or profitable,

case, there does not seem to be any harm in allowing this degenerate case. It may simplify some of the coding and representation.)

- Each sandwich consists of layers of ingredients.
- The categories of ingredients are breads, meats, cheeses, vegetables, and condiments.
- Available breads are white, wheat, and rye.
- Available meats are turkey, chicken, ham, roast beef, and tofu. (Okay, tofu is not a meat, but it is a good protein source for those who do not wish to eat meat. This is a college town after all.)
- Available cheeses are American, Swiss, jack, and cheddar.
- Available vegetables are tomato, lettuce, onion, and bell pepper.
- Available condiments are mayo, mustard, relish, and Tabasco. (Of course, this being the South, the mayo is Blue Plate Mayonnaise and the mustard is a Creole mustard.)

Let's define this as an internal DSL—in particular, by using a relatively *deep embedding* [53].

What is a sandwich? ... Basically, it is a stack of ingredients.

Should we require the sandwich to have a bread on the bottom? ... Probably. ... On the top? Maybe not, to allow "open-faced" sandwiches. ... What can the SueChef build? ... We don't know at this point, but let's assume it can stack up any ingredients without restriction.

For simplicity and flexibility, let's define a Haskell data type `Sandwich` to model sandwiches. It wraps a possibly empty list of ingredient layers. *We assume the head of the list to be the layer at the top of the sandwich.* We derive `Show` so we can display sandwiches.

```
data Sandwich = Sandwich [Layer]
                deriving Show
```

Note: In this project, we use the same name for an algebraic data type and its only constructor. Above the `Sandwich` after `data` defines a type and the one after the "`=`" defines the single constructor for that type.

Data type `Sandwich` gives the specification for a sandwich. When "executed" by the SueChef, it results in the assembly of a sandwich that satisfies the specification.

As defined, the `Sandwich` data type does not require there to be a bread in the stack of ingredients. However, we add function `newSandwich` that starts a sandwich with a bread at the bottom and a function `addLayer` that adds a new ingredient to the top of the sandwich. We leave the implementation of these functions as exercises.

```haskell
newSandwich :: Bread -> Sandwich
addLayer    :: Sandwich -> Layer -> Sandwich
```

Ingredients are in one of five categories: breads, meats, cheeses, vegetables, and condiments.

Because both the categories and the specific type of ingredient are important, we choose to represent both in the type structures and define the following types. A value of type `Layer` represents a single ingredient. Note that we use names such as `Bread` both as a constructor of the `Layer` type and the type of the ingredients within that category.

```haskell
data Layer     = Bread Bread           | Meat Meat
               | Cheese Cheese         | Vegetable Vegetable
               | Condiment Condiment
                 deriving (Eq,Show)

data Bread     = White | Wheat | Rye
                 deriving (Eq, Show)

data Meat      = Turkey | Chicken | Ham | RoastBeef | Tofu
                 deriving (Eq, Show)

data Cheese    = American | Swiss | Jack | Cheddar
                 deriving (Eq, Show)

data Vegetable = Tomato | Onion | Lettuce | BellPepper
                 deriving (Eq, Show)

data Condiment = Mayo | Mustard | Ketchup | Relish | Tabasco
                 deriving (Eq, Show)
```

We need to be able to compare ingredients for equality and convert them to strings. Because the automatically generated default definitions are appropriate, we derive both classes `Show` and `Eq` for these ingredient types.

We do not derive `Eq` for `Sandwich` because the default element-by-element equality of lists does not seem to be the appropriate equality comparison for sandwiches.

To complete the model, we define type `Platter` to wrap a list of sandwiches.

```haskell
data  Platter = Platter [Sandwich]
                deriving Show
```

We also define functions `newPlatter` to create a new `Platter` and `addSandwich` to add a sandwich to the `Platter`. We leave the implementation of these functions as exercises.

```haskell
newPlatter  :: Platter
addSandwich :: Platter -> Sandwich -> Platter
```

21.10.3 Sandwich DSL exercise set A

Please put these functions in a Haskell module `SandwichDSL` (in a file named `SandwichDSL`.) You may use functions defined earlier in the exercises to implement those later in the exercises.

1. Define and implement the Haskell functions `newSandwich`, `addLayer`, `newPlatter`, and `addSandwich` described above.

2. Define and implement the Haskell query functions below that take an ingredient (i.e., `Layer`) and return `True` if and only if the ingredient is in the specified category.

   ```haskell
   isBread     :: Layer -> Bool
   isMeat      :: Layer -> Bool
   isCheese    :: Layer -> Bool
   isVegetable :: Layer -> Bool
   isCondiment :: Layer -> Bool
   ```

3. Define and implement a Haskell function `noMeat` that takes a sandwich and returns `True` if and only if the sandwich contains no meats.

   ```haskell
   noMeat :: Sandwich -> Bool
   ```

4. According to a proposed City of Oxford ordinance, in the future it may be necessary to assemble all sandwiches in *Oxford Standard Order (OSO)*: a slice of bread on the bottom, then zero or more meats layered above that, then zero or more cheeses, then zero or more vegetables, then zero or more condiments, and then a slice of bread on top. The top and bottom slices of bread must be of the same type.

   Define and implement a Haskell function `inOSO` that takes a sandwich and determines whether it is in OSO and another function `intoOSO` that takes a sandwich and a default bread and returns the sandwich with the same ingredients ordered in OSO.

   ```haskell
   inOSO   :: Sandwich -> Bool
   intoOSO :: Sandwich -> Bread -> Sandwich
   ```

   Hint: Remember Prelude functions like `dropWhile`.

   Note: It is impossible to rearrange the layers into OSO if the sandwich does not include exactly two breads of the same type. If the sandwich does not include any breads, then the default bread type (second argument) should be specified for both. If there is at least one bread, then the bread type nearest the *bottom* can be chosen for both top and bottom.

5. Suppose we store the current prices of the sandwich ingredients in an association list with the following type synonym:

   ```
   type PriceList = [(Layer,Int)]
   ```

   Assuming that the price for a sandwich is base price plus the sum of the prices of the individual ingredients, define and implement a Haskell function `priceSandwich` that takes a price list, a base price, and a sandwich and returns the price of the sandwich.

   ```
   priceSandwich :: PriceList -> Int -> Sandwich -> Int
   ```

   Hint: Consider using the `lookup` function from the Prelude. The library `Data.Maybe` may also include helpful functions.

   Use the following price list as a part of your testing:

   ```
   prices = [ (Bread White, 20), (Bread Wheat, 30),
              (Bread Rye, 30),
              (Meat Turkey, 100), (Meat Chicken, 80),
              (Meat Ham, 120), (Meat RoastBeef, 140),
              (Meat Tofu, 50),
              (Cheese American, 50), (Cheese Swiss, 60),
              (Cheese Jack, 60), (Cheese Cheddar, 60),
              (Vegetable Tomato, 25), (Vegetable Onion, 20),
              (Vegetable Lettuce, 20), (Vegetable BellPepper,25),
              (Condiment Mayo, 5), (Condiment Mustard, 4),
              (Condiment Ketchup, 4), (Condiment Relish, 10),
              (Condiment Tabasco, 5)
            ]
   ```

6. Define and implement a Haskell function `eqSandwich` that compares two sandwiches for equality.

   What does equality mean for sandwiches? Although the definition of equality could differ, you can use "bag equality". That is, two sandwiches are equal if they have the same number of layers (zero or more) of each ingredient, regardless of the order of the layers.

   ```
   eqSandwich :: Sandwich -> Sandwich -> Bool
   ```

   Hint: The "sets" operations in library `Data.List` might be helpful

7. Give the Haskell declaration needed to make `Sandwich` an instance of class `Eq`. You may use `eqSandwich` if applicable.

21.10.4 Compiling the program for the SueChef controller

In this section, we look at compiling the `Platter` and `Sandwich` descriptions to issue a sequence of commands for the SueChef's controller.

The SueChef supports the special instructions that can be issued in sequence to its controller. The data type `SandwichOp` below represents the instructions.

```
data SandwichOp = StartSandwich    | FinishSandwich
                | AddBread Bread   | AddMeat Meat
                | AddCheese Cheese | AddVegetable Vegetable
                | AddCondiment Condiment
                | StartPlatter | MoveToPlatter | FinishPlatter
                  deriving (Eq, Show)
```

We also define the type `Program` to represent the sequence of commands resulting from compilation of a `Sandwich` or `Platter` specification.

```
data Program = Program [SandwichOp]
               deriving Show
```

The flow of a program is given by the following pseudocode:

```
StartPlatter
for each sandwich needed
    StartSandwich
    for each ingredient needed
        Add ingredient on top
    FinishSandwich
    MoveToPlatter
FinishPlatter
```

Consider a sandwich defined as follows:

```
Sandwich [ Bread Rye, Condiment Mayo, Cheese Swiss,
           Meat Ham, Bread Rye ]
```

The corresponding sequence of SueChef commands would be the following:

```
[ StartSandwich, AddBread Rye, AddMeat Ham, AddCheese Swiss,
  AddCondiment Mayo, AddBread Rye, FinishSandwich, MoveToPlatter ]
```

21.10.5 Sandwich DSL exercise set B

Add the following functions to the module `SandwichDSL` developed in the Sandwich DSL Project exercise set A.

1. Define and implement a Haskell function `compileSandwich` to convert a sandwich specification into the sequence of SueChef commands to assemble the sandwich.

   ```
   compileSandwich :: Sandwich -> [SandwichOp]
   ```

2. Define and implement a Haskell function `compile` to convert a platter specification into the sequence of SueChef commands to assemble the sandwiches on the platter.

```
compile :: Platter -> Program
```

21.10.6 Sandwich DSL source code

The Haskell source code for this project is in file:

- `SandwichDSL_base.hs`

21.11 Exam DSL Project

21.11.1 Project introduction

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work.

These special-purpose languages are often called *domain-specific languages* (or DSLs) [53]. (For more discussion of the DSL concepts, terminology, and techniques, see the introductory chapter of the Notes on Domain-Specific Languages [53].)

In this project, we design and implement a simple *internal DSL* [53]. This DSL describes simple "programs" using a set of Haskell algebraic data types. We express a program as an *abstract syntax tree* (AST) [53] using the DSL's data types.

The package first builds a set of functions for creating and manipulating the abstract syntax trees for the exams. It then extends the package to translate the abstract syntax trees to HTML.

21.11.2 Developing the Exam DSL

Suppose Professor Harold Pedantic decides to create a DSL to encode his (allegedly vicious) multiple choice examinations. Since his course uses Haskell to teach programming language organization, he wishes to implement the language processor in Haskell. Professor Pedantic is too busy to do the task himself. He is also cheap, so he assigns us, the students in his class, the task of developing a prototype.

In the initial prototype, we do not concern ourselves with the concrete syntax of the Exam DSl. We focus on design of the AST as a Haskell algebraic data type. We seek to design a few useful functions to manipulate the AST and output an exam as HTML.

First, let's focus on multiple-choice questions. For this prototype, we can assume a question has the following components:

- the text of the question

- a group of several choices for the answer to the question, exactly one of which should be be a correct answer to the question
- a group of tags identifying topics covered by the question

Let's define this as an internal DSL—in particular, by using a relatively *deep embedding* [53].

We can state a question using the Haskell data type `Question`, which has a single constructor `Ask`. It has three components—a list of applicable topic tags, the text of the question, and a list of possible answers to the question.

```
type QText    = String
type Tag      = String
data Question = Ask [Tag] QText [Choice] deriving Show
```

We use the type `QText` to describe the text of a question. We also use the type `Tag` to describe the topic tags we can associate with a question.

We can then state a possible answer to the question using the data type `Choice`, which has a single constructor `Answer`. It has two components—the text of the answer and a Boolean value that indicates whether this is a correct answer to the question (i.e., `True`) or not.

```
type AText    = String
data Choice   = Answer AText Bool deriving (Eq, Show)
```

As above, we use the type `AText` to describe the text of an answer.

For example, we could encode the question "Which of the following is a required course?" as follows.

```
Ask ["curriculum"]
    "Which of the following is a required course?"
    [ Answer "CSci 323" False,
      Answer "CSci 450" True,
      Answer "CSci 525" False ]
```

The example has a single topic tag `"curriculum"` and three possible answers, the second of which is correct.

We can develop various useful functions on these data types. Most of these are left as exercises.

For example, we can define a function `correctChoice` that takes a `Choice` and determines whether it is marked as a correct answer or not.

```
correctChoice :: Choice -> Bool
```

We can also define function `lenQuestion` that takes a question and returns the number of possible answers are given. This function has the following signature.

```
lenQuestion :: Question -> Int
```

We can then define a function to check whether a question is valid. That is, the question must have:

- a non-nil text
- at least 2 and no more than 10 possible answers
- exactly one correct answer

It has the type signature.

```
validQuestion :: Question -> Bool
```

We can also define a function to determine whether or not a question has a particular topic tag.

```
hasTag :: Question -> Tag -> Bool
```

To work with our lists of answers (and other lists in our program), let's define function `eqBag` with the following signature.

```
eqBag :: Eq a => [a] -> [a] -> Bool
```

This is a "bag equality" function for two polymorphic lists. That is, the lists are collections of elements that can be compared for equality and inequality, but not necessarily using ordered comparisons. There may be elements repeated in the list.

Now, what does it mean for two questions to be equal?

For our prototype, we require that the two questions have the same question text, the same collection of tags, and the same collection of possible answers with the same answer marked correct. However, we do not require that the tags or possible answers appear in the same order.

We note that type `Choice` has a derived instance of class `Eq`. Thus we can give an `instance` definition to make `Question` an instance of class `Eq`.

```
instance Eq Question where
-- fill in the details
```

Now, let's consider the examination as a whole. It consists of a title and a list of questions. We thus define the data type `Exam` as follows.

```
type Title = String
data Exam = Quiz Title [Question] deriving Show
```

We can encode an exam with two questions as follows.

```
Quiz "Curriculum Test" [
     Ask ["curriculum"]
         "Which one of the following is a required course?"
         [ Answer "CSci 323" False,
           Answer "CSci 450" True,
           Answer "CSci 525" False ],
```

```
Ask ["language","course"]
    "What one of the following languages is used in CSci 450?"
    [ Answer "Lua" False,
      Answer "Elm" False,
      Answer "Haskell" True ]
]
```

We can define function `selectByTags` selects questions from an exam based on the occurrence of the specified topic tags.

```
selectByTags :: [Tag] -> Exam -> Exam
```

The function application `selectByTags tags exam` takes a list of zero or more `tags` and an `exam` and returns an exam with only those questions in which at least one of the given `tags` occur in a `Question`'s tag list.

We can define function `validExam` that takes an exam and determines whether or not it is valid. It is valid if and only if all questions are valid. The function has the following signature.

```
validExam :: Exam -> Bool
```

To assist in grading an exam, we can also define a function `makeKey` that takes an exam and creates a list of `(number,letter)` pairs for all its questions. In a pair, `number` is the problem number, a value that starts with `1` and increases for each problem in order. Similarly, `letter` is the answer identifier, an uppercase alphabetic character that starts with `A` and increases for each choice in order. The function returns the tuples arranged by increasing problem number.

The function has the following signature.

```
makeKey :: Exam -> [(Int,Char)]
```

For the example exam above, `makeKey` should return `[(1,'B'),(2,'C')]`.

21.11.3 Exam DSL exercise set A

Define the following functions in a module named `ExamDSL` (in a file named `ExamDSL.hs`).

1. Develop function `correctChoice :: Choice -> Bool` as defined above.
2. Develop function `lenQuestion :: Question -> Int` as defined above.
3. Develop function `validQuestion :: Question -> Bool` as defined above.
4. Develop function `hasTag :: Question -> Tag -> Bool` as defined above.
5. Develop function `eqBag :: Eq a => [a] -> [a] -> Bool` as defined above.

6. Give an `instance` declaration to make data type `Question` an instance of class `Eq`.
7. Develop function `selectByTags :: [Tag] -> Exam -> Exam` as defined above.
8. Develop function `validExam :: Exam -> Bool` as defined above.
9. Develop function `makeKey :: Exam -> [(Int,Char)]` as defined above.

21.11.4 Outputting the Exam as HTML

Professor Pedantic wants to take an examination expressed with the Exam DSL, as described above, and output it as HTML.

Again, consider the following `Exam` value.

```
Quiz "Curriculum Test" [
     Ask ["curriculum"]
         "Which one of the following courses is required?"
         [ Answer "CSci 323" False,
           Answer "CSci 450" True,
           Answer "CSci 525" False ],
     Ask ["language","course"]
         "What one of the following is used in CSci 450?"
         [ Answer "Lua" False,
           Answer "Elm" False,
           Answer "Haskell" True ]
     ]
```

We want to convert the above to the following HTML.

```
<html lang="en">
<body>
<h1>Curriculum Test</h1>
<ol type="1">
<li>Which one of the following courses is required?
<ol type="A">
<li>CSci 323</li>
<li>CSci 450</li>
<li>CSci 525</li>
</ol>
</li>
<li>What one of the following is used in CSci 450?
<ol type="A">
<li>Lua</li>
<li>Elm</li>
<li>Haskell</li>
</ol>
```

```
</li>
</ol>
</body>
</html>
```

This would render in a browser something like the following.

Curriculum Test

1. Which one of the following courses is required?
   A. CSci 323
   B. CSci 450
   C. CSci 525
2. What one of the following is used in CSci 450?
   A. Lua
   B. Elm
   C. Haskell

Professor Pedantic developed a module of HTML template functions named SimpleHTML to assist us in this process. (See file SimpleHTML.hs.)

A function application to_html lang content wraps the content (HTML in a string) inside a pair of HTML tags <html> and </html> with lang attribute set to langtype, defaulting to English (i.e., "en"). This function and the data types are defined in the following.

```
type HTML     = String
data LangType = English | Spanish | Portuguese | French
                deriving (Eq, Show)
langmap = [ (English,"en"), (Spanish,"es"), (Portuguese,"pt"),
            (French,"fr") ]

to_html :: LangType -> HTML -> HTML
to_html langtype content =
    "<html lang=\"" ++ lang ++ "\">" ++ content ++ "</html>"
        where lang = case lookup langtype langmap of
                         Just l  -> l
                         Nothing -> "en"
```

For the above example, the to_html function generates the the outer layer:

```
<html lang="en"> ... </html>
```

Function application to_body content wraps the content inside a pair of HTML tags <body> and </body>.

```
to_body :: HTML -> HTML
to_body content = "<body>" ++ content ++ "</body>"
```

Function application to_heading level title wraps string title inside a pair of HTML tags <hN> and </hN> where N is in the range 1 to 6. If level is outside

this range, it defaults to the nearest valid value.

```
to_heading:: Int -> String -> HTML
to_heading level title = open ++ title ++ close
    where lev   = show (min (max level 1) 6)
          open  = "<h"  ++ lev ++ ">"
          close = "</h" ++ lev ++ ">"
```

Function application `to_list listtype content` wraps the `content` inside a pair of HTML tags `<ul>` and `</ul>` or `<ol>` and `</ol>`. For `<ol>` tags, it sets the `type` attribute based on the value of the `listtype` argument.

```
data ListType = Decimal | UpRoman | LowRoman
              | UpLettered | LowLettered | Bulleted
                deriving (Eq, Show)

to_list :: ListType -> HTML -> HTML
to_list listtype content = open ++ content ++ close
    where
        (open,close) =
            case listtype of
                Decimal     -> ("<ol type=\"1\">", "</ol>")
                UpRoman     -> ("<ol type=\"I\">", "</ol>")
                LowRoman    -> ("<ol type=\"i\">", "</ol>")
                UpLettered  -> ("<ol type=\"A\">", "</ol>")
                LowLettered -> ("<ol type=\"a\">", "</ol>")
                Bulleted    -> ("<ul>",            "</ul>")
```

Finally, function application `to_li content` wraps the `content` inside a pair of HTML tags `<li>` and `</li>`.

```
to_li :: HTML -> HTML
to_li content = "<li>" ++ content ++ "</li>"
```

By importing the `SimpleHTML` module, we can now develop functions to output an `Exam` as HTML.

If we start at the leaves of the `Exam` AST (i.e., from the `Choice` data type), we can define a function `choice2html` function as follows in terms of `to_li`.

```
choice2html :: Choice -> HTML
choice2html (Answer text _) = to_li text
```

Using `choice2html` and the `SimpleHTML` module, we can define `question2html` with the following signature.

```
question2html :: Question -> HTML
```

Then using `question2html` and the `SimpleHTML` module, we can define `question2html` with the following signature.

```
exam2html :: Exam -> HTML
```

Note: These two functions should add newline characters to the HTML output so that they look like the examples at the beginning of the "Outputting the Exam" section. Similarly, it should not output extra spaces. This both makes the string output more readable and makes it possible to grade the assignment using automated testing.

For example, the output of `question2html` for the first `Question` in the example above should appear as the following when printed with the `putStr` input-output command.

```
<li>Which one of the following courses is required?
<ol type="A">
<li>CSci 323</li>
<li>CSci 450</li>
<li>CSci 525</li>
</ol>
```

In addition, you may want to output the result of `exam2html` to a file to see how it displays in a browser a particular `exam`.

```
writeFile "output.html" $ exam2html exam
```

21.11.5 Exam DSL project exercise set B

Add the following functions to the module `ExamDSL` developed in the Exam DSL Project exercise set A.

1. Develop function `question2html :: Question -> HTML` as defined above.

2. Develop function `exam2html :: Exam -> HTML` as defined above.

21.11.6 Exam DSL source code

The Haskell source code for this project is in files:

- `ExamDSL_base.hs`, which is the skeleton to flesh out for a solution to this project

- `SimpleHTML.hs`, which is the module of HTML string templates

21.12 Acknowledgements

In Summer 2016, I adapted and revised much of this work from the following sources:

- Chapter 8 of my *Notes on Functional Programming with Haskell* [42], which is influenced by Bird and Wadler [15], Hudak [102], Wentworth [178], and likely other sources.

- My *Functional Data Structures (Scala)* [50], which is based, in part, on Chapter 3 of the book *Functional Programming in Scala* [29] and associated resources [30,31]

In 2017, I continued to develop this work as Chapter 8, Algebraic Data Types, of my 2017 Haskell-based programming languages textbook. I added discussion of the `Maybe` and `Either` types. For this work, I studied the Haskell `Data.Maybe` and `Data.Either` documentation, Chapter 4 on Error Handling in Chiusano [29], and articles on the Option Type [214] and the Null Object Pattern [162,213,229].

In Summer 2018, I revised this as Chapter 21, Algebraic Data Types, in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming* [54].

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I added the descriptions of three projects: Carrie's Candy Bowl, Sandwich DSL, and Exam DSL. These are adapted from homework assignments I have given in the past.

I devised the first version of Carrie's Candy Bowl project for an exam question in the Lua-based, Fall 2013 CSci 658 (Software Language Engineering) class. It is algebraic data type reformulation of the Bag project I had assigned several times in CSci 555 (Functional Programming) since the mid-1990s. I revised the problem for use in the Scala-based CSci 555 class in 2016 and for later use in the Haskell-bsed CSci 450 (Organization of Programming Languages) in Fall 2018.

I devised the first version of the Sandwich DSL problem for an exam question in the Lua-based, Fall 2013 CSci 658 (Software Language Engineering) class. I subsequently developed a full Haskell-based project for the Fall 2014 CSci 450 (Organization of Programming Languages) class. I then converted the case study to use Scala for the Scala-based, Spring 2016 CSci 555 (Functional Programming). I revised Haskell-based version for the Fall 2017 CSci 450 class.

I developed the Exam DSL project description in Fall 2018 motivated by the Sandwich DSL project and a set of questions I gave on an exam.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

21.13 Terms and Concepts

Types, algebraic data types (composite), sum (tagged, disjoint union, variant, enumerated), product (tuple, record), arity, nullary, recursive types, algebraic data types versus abstract data types, syntax, semantics, pattern matching, null

reference, safe error handling, `Maybe` and `Either` “option” types, Null Object design pattern, association list (map, dictionary), key, value.

22 Data Abstraction Revisited

22.1 Chapter Introduction

This chapter (22) revisits the specification, design, and implementation of data abstraction modules in Haskell. It follows the general approach introduced in Chapter 7 but uses algebraic data types introduced in Chapter 21 to represent the data. An algebraic data enables the Haskell module implementing the abstraction to encapsulate the details of the data structure.

The goals of this chapter are to:

- reinforce the methods for specification and design of data abstractions
- illustrate how to use Haskell modules and algebraic data types to enforce the encapsulation of a module's implementation secrets
- introduce additional concepts and terminology for data abstractions

The concepts and terminology in this chapter are mostly general. They are applicable to most any language. Here we look specifically at Haskell and focus on the details of one application. (I have implemented basically the same data abstraction module in Scala and Elixir.)

22.2 Concepts

Chapter 7 used the term *data abstraction*.

This chapter uses the related term *abstract data type* [61] to refer to a data abstraction encapsulated in an information-hiding module. The data abstraction module defines and exports a user-defined type (i.e., an algebraic data type) and a set of operations (i.e., functions) on that type. The type is abstract in the sense that its concrete representation is hidden; only the module's operations may manipulate the representation directly.

For convenience, this chapter sometimes uses acronym *ADT* to refer to an abstract data type.

In Chapters 6 and 7, we explored the concepts of contracts, which include preconditions and postconditions for the functions in the module and interface and implementation) invariants for the data created and manipulated by the module. For convenience, this chapter refers to these as the *abstract model* for the ADT.

22.3 Example: Doubly Labelled Digraph

In this chapter, we develop a family of *doubly labelled digraph* data structures.

As a *graph*, the data structure consists of a finite set of *vertices (nodes)* and a set of *edges*. Each edge connects two vertices. (Some writers require that the set of vertices be nonempty, but here we prefer to allow an empty graph to have

no vertices. But the question remains whether such a graph with no vertices is pointless concept.)

As a *directed graph* (or *digraph*), each pair of vertices has at most one edge connecting them; the edge has a direction from one of the edges to the other.

As a *doubly labelled* graph, each vertex and each edge has some user-defined data (i.e., labels) attached.

This chapter draws on the discussion of digraphs and their specification in Chapters 1 and 10 of the Dale and Walker book *Abstract Data Types* [61].

22.4 Use Case

For what purpose can we use a doubly labelled digraph data structure?

One concrete use case is to represent the game world in an implementation of an adventure game.

For example, in the Wizard's Adventure Game from Chapter 5 of *Land of Lisp: Learn to Program in Lisp, One Game at a Time* [7], the game's rooms become vertices, passages between rooms become edges, and descriptions associated with rooms or passages become labels on the associated vertex or edge (as shown in Figure 22.1).

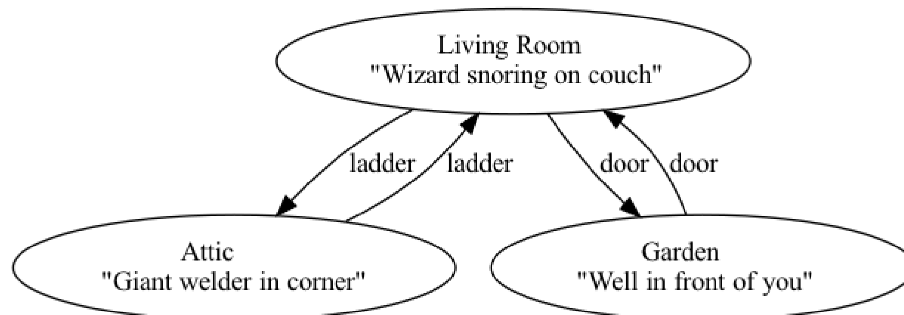


Figure 22.1: Labelled digraph for Wizard's Adventure game.

Aside: By using a digraph to model the game world, we disallow multiple passages directly from one room to another. By changing the graph to a *multigraph*, we can allow multiple directed edges from one vertex to another.

The Adventure game must create and populate the game world initially, but it does not typically modify the game world during play. It maintains the game state (e.g., player location) separately from the game world. A player moves from room to room during play; the labelled digraph provides the static structure and descriptions of the game world.

22.5 Defining ADTs

How can we define an abstract data type?

The behavior of an ADT is defined by a set of operations that can be applied to an *instance* of the ADT.

Each operation of an ADT can have inputs (i.e., parameters) and outputs (i.e., results). The collection of information about the names of the operations and their inputs and outputs is the *interface* of the ADT.

22.5.1 Specification

To specify an ADT, we need to give:

1. the *name* of the ADT
2. the *sets* (or domains) upon which the ADT is built

   These include the type being defined and the auxiliary types (e.g., primitive data types and other ADTs) used as parameters or return values of the operations.
3. the *signatures* (syntax or structure) of the operations
   - name
   - input sets (i.e., the types, number, and order of the parameters)
   - output set (i.e., the type of the return value)
4. the *semantics* (or meaning) of the operations

Note: In this chapter, we more state the specification of the data abstraction more systematically than in Chapter 7. But we are doing essentially the same things we did for the Rational Arithmetic modules in Chapter 7.

22.5.2 Operations

We categorize an ADT's operations into four groups depending upon their functionality:

- A *constructor* (sometimes called a creator, factory, or producer function) constructs and initializes an instance of the ADT.
- A *mutator* (sometimes called a modifier, command, or setter function) returns the instance with its state changed.
- An *accessor* (sometimes called an observer, query, or getter function) returns information from the state of an instance without changing the state.
- A *destructor* destroys an instance of the ADT.

We normally list the operations in that order.

For a language with immutable data structures like Haskell, a mutator returns a distinct new instance of the ADT with a state that is a modified version of the original instance's state. That is, we are taking an applicative (or functional or referentially transparent) approach to ADT specifications.

Note: Of course, in an imperative language, a mutator can change the state of an instance in place. That may be more efficient, but it tends to be less safe. It also tends to make concurrent use of an abstract data type more problematic.

Technically speaking, a destructor is not an operation of the ADT. We can represent the other types of operations as functions on the sets in the specification. However, we cannot define a destructor in that way. But destructors are of pragmatic importance in the implementation of ADTs, particularly in languages that do not have automatic storage reclamation (i.e., garbage collection).

22.5.3 Approaches to semantics

There are two primary approaches for specifying the semantics of the operations:

- The *axiomatic* (or *algebraic*) approach gives a set of logical rules (properties or axioms) that relate the operations to one another. The meanings of the operations are defined implicitly in terms of each other.

- The *constructive* (or *abstract model*) approach describes the meaning of the operations explicitly in terms of operations on other abstract data types. The underlying *model* may be any well-defined mathematical model or a previously defined ADT.

In some ways, the axiomatic approach is the more elegant of the two approaches. It is based in the well-established mathematical fields of abstract algebra and category theory. Furthermore, it defines the new ADT independently of other ADTs. To understand the definition of the new ADT it is only necessary to understand its axioms, not the semantics of a model.

However, in practice, the axiomatic approach to specification becomes very difficult to apply in complex situations. The constructive approach, which builds a new ADT from existing ADTs, is the more useful methodology for most practical software development situations.

In this chapter, we use the constructive approach.

22.6 Specification of Labelled Digraph ADT

Now let's look at a constructive specification of the doubly labelled digraph.

First, we specify the ADT as an implementation-independent abstraction. The *secret* of the ADT module is the data structure used internally to implement the doubly labelled digraph.

Then, we examine two implementations of the abstraction:

- using Haskell lists to represent the vertex and edge sets
- using a Haskell `Map` to map a vertex to the set of outgoing edges from that vertex

Before we specify the ADT, let's define the mathematical notation we use. We choose notation that can readily be used in comments in program.

22.6.1 Notation

We use the the plain-text specification notation to describe the abstract data type's model and its semantics. The following document summarizes this notation:

- **Plain Text Specification Notation (PTSN)** HTML
  PDF

TODO:

- Make sure the specification in this chapter and source code use the PTSN.
- For the full ELIFP book, it might be better to make the PTSN document an appendix chapter (81?).
- Perhaps use more traditional logic and math notation [40] in the ELIFP book and PTSN in the code.

22.6.2 Sets

We *name* the abstract data type being defined to be `Digraph`.

We specify that this abstract data type be represented by a Haskell algebraic data type `Digraph a b c`, which has three *type* parameters (i.e., sets):

1. `VertexType`, the set of possible vertices (i.e., vertex identifiers) in the `Digraph`
2. `VertexLabelType`, the set of possible labels on vertices in the `Digraph`
3. `EdgeLabelType`, the set of possible labels on edges in the `Digraph`

Given this ADT defines a digraph, edges can be identified by ordered pairs (tuples) of vertices. Values from the above types, in particular the labels, may have several components.

22.6.3 Signatures

We define the following operations on the Labelled Digraph ADT (shown below as Haskell function signatures).

Given the primary use case described above, we specify a constructor to create an empty graph (`new_graph`), a mutator to add a new vertex (`add_vertex`), and mutator to add a new edge between existing vertices (`add_edge`).

We also specify mutators to remove vertices (`remove_vertex`) and edges (`remove_edge`) and to update the labels on vertices (`update_vertex`) and edges (`update_edge`). (Note: In the identified use case, these are likely used less often than the mutators that add new vertices and edges.)

Constructors We specify a single constructor with the following signature:

```
new_graph :: Digraph a b c
```

Mutators We specify six mutators with the following signatures:

```
add_vertex    :: Digraph a b c -> a -> b -> Digraph a b c
remove_vertex :: Digraph a b c -> a -> Digraph a b c
update_vertex :: Digraph a b c -> a -> b -> Digraph a b c
add_edge      :: Digraph a b c -> a -> a -> c -> Digraph a b c
remove_edge   :: Digraph a b c -> a -> a -> Digraph a b c
update_edge   :: Digraph a b c -> a -> a -> c -> Digraph a b c
```

Accessors We specify query functions to check whether the labelled digraph is empty (`is_empty`), has a given vertex (`has_vertex`), and has an edge between two vertices (`has_edge`).

We also specify accessors to retrieve the label associated with a given vertex (`get_vertex`) and edge (`get_edge`).

Given the identified use case, we also specify accessors to return lists of all vertices in the graph (`all_vertices`) and of just their labels (`all_vertices_labels`) and to return lists of all outgoing edges from a vertex (`from_edges`) and of just their labels (`from_edges_labels`).

We thus specify nine accessors with the following signatures:

```
is_empty      :: Digraph a b c -> Bool
get_vertex    :: Digraph a b c -> a -> b
has_vertex    :: Digraph a b c -> a -> Bool
get_edge      :: Digraph a b c -> a -> a -> c
has_edge      :: Digraph a b c -> a -> a -> Bool
all_vertices  :: Digraph a b c -> [a]
from_edges    :: Digraph a b c -> a -> [a]
all_vertices_labels :: Digraph a b c -> [(a,b)]
from_edges_labels   :: Digraph a b c -> a -> [(a,c)]
```

TODO: Consider changing `get_vertex` and `get_edge` to return `Maybe b` and `Maybe c`, respectively.

Destructors Given the identified use case and that Haskell uses garbage collection, no destructor seems to be needed in most cases.

22.6.4 Semantics

We *model* the state of the instance of the Labelled Digraph ADT with an abstract value `G` such that `G = (V,E,VL,EL)` with `G`'s components satisfying the following **Labelled Digraph Properties**.

- `V` is a finite subset of values from the set `VertexType`. `V` denotes the vertices (or nodes) of the digraph.
- Any two elements of `V` can be compared for *equality*.
- `E` is a binary relation on the set `V`. A pair `(v1,v2) IN E` denotes that there is a directed edge from `v1` to `v2` in the digraph.

  Note that this model allows at most one (directed) edge from a vertex `v1` to vertex `v2`. It allows a directed edge from a vertex to itself.

  Also, because vertices can be compared for equality, any two edges can also be compared for equality.
- `VL` is a total function from set `V` to the set `VertexLabelType`.
- `EL` is a total function from set `E` to the set `EdgeLabelType`.

22.6.4.1 Interface invariant We define the following interface invariant for the Labelled Digraph ADT:

> *Any valid labelled digraph instance `G`, appearing in either the arguments or return value of a public ADT operation, must satisfy the Labelled Digraph Properties.*

22.6.4.2 Constructive semantics We specify the various ADT operations below using their type signatures, preconditions, and postconditions. Along with the interface invariant, these comprise the (implementation-independent) specification of the ADT (i.e., its abstract interface).

In these assertions, for a digraph `g` that satisfies the invariants, `G(g)` denotes its abstract model`(V,E,VL,EL)` as described above. The value `Result` denotes the return value of function.

TODO: Consider in what order these should appear.

- Constructor `new_graph` creates and returns a new empty instance of the graph ADT.

  Precondition:

  `True`

  Postcondition:

  `G(Result) == ({},{},{},{})`

- Accessor `is_empty g` returns `True` if and only if graph `g` is empty.

  Precondition:

  ```
  G(g) = (V,E,VL,EL)
  ```

  Postcondition:

  ```
  Result == (V == {} && E == {})
  ```

- Mutator `add_vertex g nv nl` inserts vertex `nv` with label `nl` into graph `g` and returns the resulting graph.

  Precondition:

  ```
  G(g) = (V,E,VL,EL) && nv NOT_IN V
  ```

  Postcondition:

  ```
  G(Result) == (V UNION {nv}, E, VL UNION {(nv,nl)}, EL)
  ```

- Mutator `remove_vertex g ov` deletes vertex `ov` from graph `g` and returns the resulting graph.

  Precondition:

  ```
  G(g) =  (V,E,VL,EL) && ov IN V
  ```

  Postcondition:

  ```
  G(Result) == (V', E', VL', EL')
          where V'  = V  - {ov}
                E'  = E  - {(ov,*),(*,ov)}
                VL' = VL - {(ov,*)}
                EL' = EL - {((ov,*),*),((*,ov),*)}
  ```

- Mutator `update_vertex g ov nl` changes the label on vertex `ov` in graph `g` to be `nl` and returns the resulting graph.

  Precondition:

  ```
  G(g) =  (V,E,VL,EL) && ov IN V
  ```

  Postcondition:

  ```
  G(Result) == (V - {ov}, E, VL', EL)
      where VL' = (VL - {(ov,VL(ov))}) UNION {(ov,nl)}
  ```

- Accessor `get_vertex g ov` returns the label from vertex `ov` in graph `g`

  TODO: If signature changed to return `Maybe`, change precondition and postcondition appropriately.

  Precondition:

  ```
  G(g) = (V,E,VL,EL) && ov IN V
  ```

  Postcondition:

```
Result == VL(ov)
```

- Accessor `has_vertex g ov` returns `True` if and only if `ov` is a vertex of graph `g`.

  Precondition:

  ```
  G(g) = (V,E,VL,EL)
  ```

  Postcondition:

  ```
  G(Result) == ov IN V
  ```

- Mutator `add_edge g v1 v2 nl` inserts an edge from vertex `v1` to vertex `v2` in graph `g` and returns the resulting graph.

  Precondition:

  ```
  G(g) = (V,E,VL,EL) && v1 IN V && v2 IN V &&
  (v1,v2) NOT_IN E
  ```

  Postcondition:

  ```
  G(Result) == (V, E', VL, EL')
      where E'  = E  UNION {(v1,v2)}
            EL' = EL UNION {((v1,v2),nl)}
  ```

- Mutator `remove_edge g v1 v2` deletes the edge from vertex `v1` to vertex `v2` from graph `g` and returns the resulting graph.

  Precondition:

  ```
  G(g) =  (V,E,VL,EL) V - {ov} && (v1,v2) IN E
  ```

  Postcondition:

  ```
  G(Result) == (V, E - {(v1,v2)}, VL, EL - { ((v1,v2),*) }
  ```

- Mutator `update_edge g v1 v2 nl` changes the label on the edge from vertex `v1` to vertex `v2` in graph `g` to have label `nl` and returns the resulting graph.

  Precondition:

  ```
  G(g) = (V,E,VL,EL) && (v1,v2) IN E
  ```

  Postcondition:

  ```
  G(Result) == (V, E, VL, EL')
      where EL' == (EL - {((v1,v2),*)}) UNION {((v2,v2),nl)
  ```

- Accessor `get_edge g v1 v2` returns the label on the edge from vertex `v1` to vertex `v2` in graph `g`.

  TODO: If signature changed to return `Maybe`, change precondition and postcondition appropriately.

Precondition:

```
G(g) = (V,E,VL,EL) && (v1,v2) IN E
```

Postcondition:

```
Result == EL((v1,v2))
```

- Accessor `has_edge g v1 v2` returns `True` if and only if there is an edge from a vertex `v1` to a vertex `v2` in graph `g`.

  Precondition:

  ```
  G(g) = (V,E,VL,EL)
  ```

  Postcondition:

  ```
  Result == (v1,v2) IN E
  ```

- Accessor `all_vertices g` returns a sequence of all the vertices in graph `g`. The returned sequence is represented by a builtin Haskell list.

  Precondition:

  ```
  G(g) = (V,E,VL,EL)
  ```

  Postcondition:

  ```
  (ForAll ov: ov IN Result <=> ov IN V) &&
  length(Result) == size(V)
  ```

- Accessor `from_edges g v1` returns a sequence of all vertices `v2` such that there is an edge from vertex `v1` to vertex `v2` in graph `g`. The returned sequence is represented by a builtin Haskell list.

  Precondition:

  ```
  G(g) = (V,E,VL,EL) && v1 IN V
  ```

  Postcondition:

  ```
  (ForAll v2: v2 IN Result <=> (v1,v2) IN E) &&
  length(Result) == (# v2 :: (v1,v2) IN E)
  ```

  TODO: Function `from_edges g v1` should return `[]` when `v1` does not appear in `g`, so that it can work well with the Wizard's Adventure game. We should redefine the precondition and postcondition to specify this behavior.

- Accessor `all_vertices_labels g` returns a sequence of all pairs `(v,l)` such that `v` is a vertex and `l` is it's label in graph `g`. The returned sequence is represented by a builtin Haskell list.

  Precondition:

  ```
  G(g) = (V,E,VL,EL)
  ```

Postcondition:

```
(ForAll v, l: (v,l) IN Result <=> (v,l) IN VL) &&
length(Result) == size(VL)
```

- Accessor `from_edges_labels g v1` returns a sequence of all pairs `(v2,l)` such that there is an edge `(v1,v2)` labelled with `l` in graph `g`.

  Precondition:

  ```
  G(g) = (V,E,VL,EL) && v1 IN V
  ```

  Postcondition:

  ```
  (ForAll v2, l :: (v2,l) IN Result <=> ((v1,v2),l) IN EL)
  && length(Result) == (# v2 :: (v1,v2 ) IN E)
  ```

  TODO: Function `from_edges_labels g v1` should return `[]` when `v1` does not appear in `g`, so that it can work well with the Wizard's Adventure game. We should redefine the precondition and postcondition to specify this behavior.

22.6.5 Haskell module abstract interface

Below we state the header for a Haskell module `Digraph_XXX` that implements the Labelled Digraph ADT. The module name suffix `XXX` denotes the particular implementation for a data representation, but the signatures and semantics of the operations are the same regardless of representation.

The module *exports* data type `Digraph`, but its constructors are not exported. This allows modules that import `Digraph_XXX` to use the data type without knowing how the data type is implemented.

If we had `Digraph(..)` in the export list, then the data type and all its constructors would be exported.

The intention of this interface is to constrain the type parameters of `Digraph a b c` so that:

- Type `a` (i.e., type `VertexType`) must be in Haskell class `Eq`. This is essentially required by the interface invariant (i.e., the Labelled Digraph Properties).
- Types `a`, `b`, and `c` (i.e., types `VertexType`, `VertexLabelType`, and `EdgeLabelType`) must be in Haskell class `Show`. This contraint enables the vertices and labels to be displayed as text.

It may be desirable (or necessary) for an implementation to further constrain the type parameters. For example, some implementations may need to constrain `VertexType` to be from class `Ord` (i.e., totally ordered). It does not seem to restrict the generality of the ADT significantly to require that vertices be drawn from a totally ordered set such as integers, strings, or enumerated types.

```haskell
module DigraphADT_XXX
  ( Digraph      --constraints (Eq a, Show a, Show b, Show c)
  , new_graph    --Digraph a b c
  , is_empty     --Digraph a b c -> Bool
  , add_vertex   --Digraph a b c -> a -> b -> Digraph a b c
  , remove_vertex--Digraph a b c -> a -> Digraph a b c
  , update_vertex--Digraph a b c -> a -> b -> Digraph a b c
  , get_vertex   --Digraph a b c -> a -> b
  , has_vertex   --Digraph a b c -> a -> Bool
  , add_edge     --Digraph a b c -> a -> a -> c -> Digraph a b c
  , remove_edge  --Digraph a b c -> a -> a -> Digraph a b c
  , update_edge  --Digraph a b c -> a -> a -> c -> Digraph a b c
  , get_edge     --Digraph a b c -> a -> a -> c
  , has_edge     --Digraph a b c -> a -> a -> Bool
  , all_vertices --Digraph a b c -> [a]
  , from_edges   --Digraph a b c -> a -> [a]
  , all_vertices_labels--Digraph a b c -> [(a,b)]
  , from_edges_labels  --Digraph a b c -> a -> [(a,c)]
  )
where  -- definitions for the types and functions
```

Note: The Glasgow Haskell Compiler (GHC) release 8.2 (July 2017) and the Cabal-Install package manager release 2.0 (August 2017) support a new mixin package system called Backpack. This extension would enable us to define an abstract module "DigraphADT" as a signature file with the above interface. Other modules can then implement this abstract interface thus giving a more explicit and flexible definition of this abstract data type.

22.7 List Implementation

This section gives an implementation of the ADT that uses Haskell lists to represent the vertex and edge sets.

22.7.1 Labelled digraph representation

We represent the List implementation of the Labelled Digraph ADT as an instance of the Haskell algebraic data type `Digraph` as shown below. (Remember that type variable `a` is `VertexType`, `b` is `VertexLabelType`, and `c` is `EdgeLabelType`.)

```haskell
data Digraph a b c = Graph [(a,b)] [(a,a,c)]
```

In an instance (`Graph vs es`):

- `vs` is a list of tuples `(v,vl)` where
  - `v` has `VertexType` and represents a vertex of the digraph
  - `vl` has `VertexLabelType` and is the unique label associated with vertex `v`

- a vertex `v` occurs at most once in `vs` (i.e., `vs` encodes a function from vertices to vertex labels)

- `es` is a list of tuples `((v1,v2),el)` where
  - `v1` and `v2` are vertices occurring in `vs`, representing a directed edge from `v1` to `v2`
  - `el` has `EdgeLabelType` and is the unique label associated with edge `(v1,v2)`
  - an edge `(v1,v2)` occurs at most once in `es` (i.e., `es` encodes a function from edges to edge labels)

In terms of the abstract model, `vs` encodes `VL` directly and, because `VL` is a total function on `V`, it encodes `V` indirectly. Similarly, `es` encodes `EL` directly and `E` indirectly.

Of course, there are many other ways to represent the graph as lists. This representation is biased for a context where, once built, the labelled digraph is relatively static and the most frequent operations are the retrieval of labels attached to vertices or edges. That is, it is biased toward the Adventure game use case.

Given that all the type parameters must be of class `Show`, we also define `Digraph` to also be of class `Show` as defined below.

```
instance (Show a, Show b, Show c) =>
                    Show (Digraph a b c) where
    show (Graph vs es) =
        "(Digraph " ++ show vs ++ ", " ++ show es ++ ")"
```

22.7.2 Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labelled Digraph ADT:

> *Any Haskell `Digraph` value `(Graph vs es)` with abstract model `G = (V,E,VL,EL)`, appearing in either the arguments or return value of an operation, must also satisfy the following:*

```
(ForAll v, l :: (v,l) IN vs  <=> (v,l) IN VL ) &&
(ForAll v1, v2, m :: (v1,v2,m) IN es  <=> ((v1,v2),m) IN EL )
```

22.7.3 Haskell implementation

The code in this section shows a list-based implementation for several of the operations related to vertices.

The Haskell module for the list representation of the Labelled Digraph ADT is in source file `DigraphADT_List.hs`. A simple smoke test driver module is in source file `DigraphADT_TestList.hs`.

The implementations of constructor new_graph and accessor is_empty are straightforward.

```
new_graph :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c
new_graph = Graph [] []

is_empty :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> Bool
is_empty (Graph [] _ ) = True
is_empty _             = False
```

Function has_vertex just needs to search through the list of vertices to determine whether or not the vertex occurs. It relies upon VertexType being in class Eq.

```
has_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> Bool
has_vertex (Graph vs _) ov =
    not (null [ n | (n,_) <- vs, n == ov])
```

Because of lazy evaluation, the list comprehension only needs to evaluate far enough to find the occurrence of the vertex in the list.

To add a new vertex and its label to the graph, add_vertex must return a new graph with the new vertex-label pair added to the head of the vertex list. To meet the specification, it must not allow a vertex to be added if the vertex already occurs in the list.

```
add_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b -> Digraph a b c
add_vertex g@(Graph vs es) nv nl
    | not (has_vertex g nv) = Graph ((nv,nl):vs) es
    | otherwise             = error has_nv
    where has_nv =
        "Vertex " ++ show nv ++ " already in digraph"
```

Function remove_vertex is a bit trickier with this representation. To remove an existing vertex and its label from the graph, remove_vertex must return a new graph with that vertex's tuple removed from the list of vertices and with any outgoing edges also removed from the list of edges.

```
remove_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> Digraph a b c
remove_vertex g@(Graph vs es) ov
    | has_vertex g ov = Graph ws fs
    | otherwise       = error no_ov
    where ws    = [ (w,m)     | (w,m) <- vs, w /= ov ]
          fs    = [ (v1,v2,m) |
                    (v1,v2,m) <- es, v1 /= ov, v2 /= ov ]
          no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

The implementation of `remove_vertex` filters all occurrences of the vertex from the list of vertices. Given the implementation invariant, this is not necessary. However, this potentially adds some safety to the implementation at the possible expense of execution time.

For an existing vertex in the list of vertices, function `update_vertex` replaces the old label with the new label. Like `remove_vertex`, it potentially processes the entire list of vertices and makes the change to all occurrences, when the implementation invariant would allow it to stop on the first (and only) occurrence.

```
update_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b -> Digraph a b c
update_vertex g@(Graph vs es) ov nl
    | has_vertex g ov = Graph (map chg vs) es
    | otherwise       = error no_ov
    where chg (w,m) = (if w == ov then (ov,nl) else (w,m))
          no_ov     = "Vertex " ++ show ov ++ " not in digraph"
```

For an existing vertex, function `get_vertex` retrieves the label. Because of lazy evaluation, the search of the list of vertices stops with the first occurrence.

TODO: Modify appropriately if changed to `Maybe` return.

```
get_vertex :: (Eq a, Show a, Show b, Show c) =>
                    Digraph a b c -> a -> b
get_vertex (Graph vs _)  ov
    | not (null ls) = head ls
    | otherwise     = error no_ov
    where ls    = [ l | (w,l) <- vs, w == ov]
          no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

TODO: Modify source file appropriately if changed to `Maybe` return.

The remainder of the functions are defined in file `DigraphADT_List.hs`..

We can create an empty labelled digraph `g0` having `Int` identifiers for vertices, `Int` labels for vertices, and `Int` labels for edges as follows:

```
g0  = (new_graph :: Digraph Int Int Int)
```

Then we can add a new vertex with identifier `1` and vertex label `101` as follows:

```
g1  = add_vertex g0 1 101
```

22.7.4 Improvements to the list implementation

TODO: Consider whether to make any of the following changes to the specification and implementation above.

Based on the list-based design and implementation above, what improvements should we consider? Here are some possibilities.

1. As described above, the current list implementations of functions such as `remove_vertex` and `update_vertex` do some unnecessary work with respect to the implementation invariant. This could be eliminated.

2. The data representation (i.e., implementation invariant) could be changed to allow, for example, multiple occurrences of vertices in the vertex list. This would avoid the checks of `has_vertex` in `add_vertex` and `update_vertex`. Then, as it does above, `remove_vertex` needs to remove all occurrences of the vertex.

   Other functions would need to be modified accordingly so that they only access the first occurrence of a vertex (especially the `all_vertices` and `all_vertices_labels` functions).

   A similar change could be made to the list of edges.

   Note: The Labelled Diagraph ADT specification does not specify what the behavior should be when the referenced vertex or edge is not defined. The change suggested in this item gives non-error behavior to those situations. Perhaps a better alternative would be to change the general ADT specification to require specific behaviors in those cases.

3. Most of the functions throw an `error` exception when the vertex they reference does not exist. A better Haskell design would redefine these functions to return a `Maybe` or `Either` value. This would eliminate most of the `has_vertex` checks and make the functions defined on all possible inputs.

   This would require changes to the overall Labelled Digraph ADT specification and its abstract interface.

4. New functions could be added to the Labelled Digraph ADT—such as an equality check on graphs, a constructor that creates a copy of an existing graph, or functions to apply various graph algorithms.

5. Existing functions could be eliminated. For example, if the graph is only constructed and used for retrieval, then the remove and update functions could be eliminated.

22.8 Map Implementation

This section gives an implementation of the ADT that uses a Haskell `Map` to map a vertex to the set of outgoing edges from that vertex

22.8.1 Labelled digraph representation

We represent the Map implementation of the Labelled Digraph ADT as an instance of the Haskell algebraic data type `Digraph` as shown below. (Remember that type variable `a` is `VertexType`, `b` is `VertexLabelType`, and `c` is `EdgeLabelType`.)

```
import qualified Data.Map.Strict as M

data Digraph a b c = Graph (M.Map a (b,[(a,c)]))
```

In the data constructor `(Graph m)`, `m` is an instance of `Data.Map.Strict`. This collection is set of key-value pairs implemented as a balanced tree, giving logarithmic access time.

An instance of `(Graph m)` corresponds to the abstract model as follows:

- The keys for the `Map m` collection are of `VertexLabelType`.

  The interface invariant requires that `VertexType` be in class `Eq`. The implementation based on `Data.Map.Strict` further constrains vertices to be in subclass `Ord` because the vertices are the keys of the `Map`.

  TODO: Consider restricting the Digraph spec to require `Ord`.

- `Map m` is defined for all keys `v1` in vertex set `V` and undefined for all other keys.

- For some vertex `v1`, the value of `m` at key `v1` is a pair `(l,es)` where

  - `l` is an element of `VertexLabelType` and is the unique label associated with `v1`, that is, `l = VL(v1)`.

  - `es` is the list of all tuples `(v2,el)` such that `(v1,v2) IN E`, `el IN EdgeLabelType`, and `el = EL((v1,v2))`. That is, `(v1,v2)` is an edge and `el` is its unique label.

Given that all the type parameters must be of class `Show`, we also define `Digraph` to also be of class `Show` as defined below.

```
instance (Show a, Show b, Show c) => Show (Digraph a b c) where
    show (Graph m) = "(Digraph " ++ show (M.toAscList m) ++ ")"
```

22.8.2 Implementation invariant

Given the above description, we then define the following implementation (representation) invariant for the list-based version of the Labelled Digraph ADT:

> *Any Haskell `Digraph` value `(Graph m)` with abstract model `G = (V,E,VL,EL)`, appearing in either the arguments or return value of an operation, must also satisfy the following:*

```
(ForAll v1, l, es ::
    ( m(v1) defined && m(v1) == (l,es) ) <=>
    ( VL(v1) == l &&
        (ForAll v2, el :: (v2,el) IN es <=>
                          EL((v1,v2)) == el) ) )
```

22.8.3 Haskell module

The code in this section shows a map-based implementation for the same operations we examined for the list-based implementation.

The Haskell module for the map representation of the Labelled Digraph ADT is in source file `DigraphADT_Map.hs`.. A simple smoke test driver module is in source file `DigraphADT_TestMap.hs`..

Constructor `new_graph` and accessors `is_empty` and `has_vertex` are just wrappers for functions from `Data.Map.Strict`.

```
new_graph :: (Ord a, Show a, Show b, Show c) =>
               Digraph a b c
new_graph = Graph M.empty

is_empty :: (Ord a, Show a, Show b, Show c) =>
                Digraph a b c -> Bool
is_empty (Graph m) = M.null m

has_vertex :: (Ord a, Show a, Show b, Show c) =>
                Digraph a b c -> a -> Bool
has_vertex (Graph m) ov = M.member ov m
```

To add a new vertex and label to the graph, `add_vertex` must return a graph with the new key-value pair inserted into the existing graph's `Map`. The value consists of the label paired with a nil list of adjacent edges. To meet the specification, it must not allow a vertex to be added if the vertex already occurs in the list.

```
add_vertex :: (Ord a, Show a, Show b, Show c) =>
                Digraph a b c -> a -> b -> Digraph a b c
add_vertex g@(Graph m) nv nl
    | not (has_vertex g nv) = Graph (M.insert nv (nl,[]) m)
    | otherwise             = error has_nv
    where has_nv =
        "Vertex " ++ show nv ++ " already in digraph"
```

Except for making sure the vertex to be deleted is the graph, function `remove_vertex` is just a wrapper for the `Data.Map.Strict.delete` function.

```
remove_vertex :: (Ord a, Show a, Show b, Show c) =>
                   Digraph a b c -> a -> Digraph a b c
remove_vertex g@(Graph m) ov
    | has_vertex g ov = Graph (M.delete ov m)
    | otherwise       = error no_ov
    where no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

If the argument vertex is in the graph, then function `update_vertex` retrieves its old label and edge list and then reinserts the new label paired with the same

edge list.

```
update_vertex :: (Ord a, Show a, Show b, Show c) =>
                     Digraph a b c -> a -> b -> Digraph a b c
update_vertex g@(Graph m) ov nl
    | has_vertex g ov =
        Graph (M.insert ov (upd (M.lookup ov m)) m)
    | otherwise       = error no_ov
    where upd (Just (ol,edges)) = (nl,edges)
          upd _                 = error no_entry
          no_ov    = "Vertex " ++ show ov ++ " not in digraph"
          no_entry =
              "Missing/malformed value for vertex " ++ show ov
```

For an existing vertex, function `get_vertex` retrieves the associated value and extracts the label.

```
get_vertex :: (Ord a, Show a, Show b, Show c) =>
                  Digraph a b c -> a -> b
get_vertex g@(Graph m)  ov
    | has_vertex g ov = getlabel (M.lookup ov m)
    | otherwise       = error no_ov
    where getlabel (Just (ol,_)) = ol
          no_ov = "Vertex " ++ show ov ++ " not in digraph"
```

The remainder of the functions are defined in file `DigraphADT_Map.hs`.

The Map-based functions can be called in the same manner as the List-based function, except that the vertices must be in class `Ord`.

22.8.4 Improvements to the map implementation

All the improvements suggested for the list-based implementation apply to the map-based implementation except for the first.

For large graphs, the map-based implementation should perform better than the list-based implementation.

For large graphs with many outgoing edges on each vertex, it might be useful to implement the edge-list itself with a `Map`.

22.9 What Next?

This chapter (22) revisited the issues of specification, design, and implementation of data abstractions as modules in Haskell. It used a labelled digraph data structure as the example.

Although we may not specify all subsequent Haskell modules as systematically as we did in this chapter, we do use the modular style of programming in the various interpreters developed in Chapter 41 and following.

In the future, we plan to implement a Adventure game on top of the ADT implemented in this chapter.

22.10 Chapter Source Code

TODO

22.11 Exercises

TODO: If the `Maybe` improvement is not done above for errors, put that here as an exercise.

1. Restate the preconditions and postconditions for functions `from_edges` and `from_edges` so that they must return empty lists when the argument vertex `v1` is not in the vertex set. (See the notes on these operations in the semantic specification above.)

2. Develop a comprehensive test script for the Labelled Digraph ADT implementations using blackbox, module-level, functional testing as described in Chapters 11 and 12.

3. Adapt the Haskell Labelled Digraph ADT interface and it two implementations to use GHC's Backpack module system.

4. Specify a similar Labelled Digraph ADT as a Java interface.

5. Give two different implementations of the Labelled Digraph ADT in Java using the specification from the previous exercise.

6. Specify a similar Labelled Digraph ADT as a Python 3 module.

7. Give two different implementations of the Labelled Digraph ADT in Python using the specification from the previous exercise.

8. Choose one of the improvements described in the "Improvements in the list implementation" subsection and change the specification and list implementation as needed for the improvement.

9. Choose one of the improvements and change the specification and map implementation as needed for the improvement.

10. Give a full specification (similar to the one for the Labeled Digraph ADT) in this chapter) for the Carrie's Candy Bowl project in Chapter 21<!-22—>. That is, give the name, set, signatures, and constructive semantics. If helpful, you may use the mathematical concept of bag.

11. Specify a doubly labelled directed multigraph data structure to replace the doubled labelled digraph. (That is, allow multiple directed edges from one vertex to another.)

12. Give an implementation of the doubly labelled directed multigraph specified in the previous exercise.

22.12 Mealy Machine Simulator Project

22.12.1 Project introduction

In this project, you are asked to design and implement Haskell modules to represent Mealy Machines and to simulate their execution.

This kind of machine is a useful abstraction for simple controllers that listen for input events and respond by generating output events. For example in an automobile application, the input might be an event such as "fuel level low" and the output might be command to "display low-fuel warning message".

In the theory of computation, a *Mealy Machine* is a *finite-state automaton* whose output values are determined both by its current state and the current input. It is a *deterministic finite state transducer* such that, for each state and input, at most one transition is possible.

Appendix A of the Linz textbook [118] defines a Mealy Machine mathematically by a tuple

$$M = (Q, \Sigma, \Gamma, \delta, \theta, q_0)$$

where

Q is a finite set of internal states
Σ is the input alphabet (a finite set of values)
Γ is the output alphabet (a finite set of values)
$\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
$\theta : Q \times \Sigma \longrightarrow \Gamma$ is the output function
q_0 is the initial state of M (an element of Q)

In an alternative formulation, the transition and output functions can be combined into a single function:

$$\delta : Q \times \Sigma \longrightarrow Q \times \Gamma$$

We often find it useful to picture a finite state machine as a *transition graph* where the states are mapped to vertices and the transition function represented by directed edges between vertices labelled with the input and output symbols.

22.12.2 Mealy Machine Simulator project exercises

1. Specify, design, and implement a general representation for a Mealy Machine as a Haskell module implementing an abstract data type. It should hide the representation of the machine and should have, at least, the following public operations.

    - `newMachine s` creates a new machine with initial (and current) state `s` and no transitions.

      Note: This assumes that the state, input, and output sets are exactly those added with the mutator operations below. An alternative would

be to change this function to take the allowed state, input, and output sets.

- `addState m s` adds a new state `s` to machine `m` and returns an `Either` wrapping the modified machine or an error message.
- `addTransition m s1 in out s2` adds a new transition to machine `m` and returns an `Either` wrapping the modified machine or an error message. From state `s1` with input `in` the modified machine outputs `out` and transitions to state `s2`.
- `addResets m` adds all reset transitions to machine `m` and returns the modified machine. From state `s1` on input `in` the modified machine outputs `out` and transitions to state `s2`. This operation makes the transition function a total function by adding any missing transitions from a state back to the initial state.
- `setCurrent m s` sets the current state of machine `m` to `s` and returns an `Either` wrapping the modified machine or an error message.
- `getCurrent m` returns the current state of machine `m`.
- `getStates m` returns a list of the elements of the state set of machine `m`.
- `getInputs m` returns a list of the input set of machine `m`.
- `getOutputs m` returns a list of the output set of machine `m`.
- `getTransitions m` returns a list of the transition set of machine `m`. Tuple `(s1,in,out,s2)` occurs in the returned list if and only if, from state `s1` with input `in`, the machine outputs `out` and moves to state `s2`.
- `getTransitionsFrom m s` returns an `Either` wrapping a list of the set of transitions enabled from state `s` of machine `m` or an error message.

2. Given the above implementation for a Mealy Machine, design and implement a separate Haskell module that simulates the execution of a Mealy Machine. It should have, at least, the following new public operations.

   - `move m in` moves machine `m` from the current state given input `in` and returns an `Either` wrapping a tuple `(m',out)` or an error message. The tuple gives the modified machine `m'` and the output `out`.
   - `simulate m ins` simulates execution of machine `m` from its current state through a sequence of moves for the inputs in list `ins` and returns an `Either` wrapping a tuple `(m',outs)` or an error message. The tuple gives the modified machine `m'` after the sequence of moves and the output list `outs`.

Note: It is possible to use a Labelled Digraph ADT module in the implementation of the Mealy Machine.

3. Implement a Haskell module that uses a different representation for the Mealy Machine. Make sure the simulator module still works correctly.

22.13 Acknowledgements

In Spring 2017, I created a Labelled Digraph ADT document by adapting and revising comments from the Haskell implementations of the Labelled Digraph abstract data type. I had specified the ADT and developed the implementations as my solution for Assignment #1 in CSci 556 (Multiparadigm Programming) in Spring 2015. I also included some content from my notes on Data Abstraction [46].

(In addition to the list- and map-based Haskell implementations of the Labelled Digraph ADT, I developed a list-based implementation in Elixir in Spring 2015 and two Scala-based implementations in Spring 2016.)

In Spring 2017, I also created a Mealy Machine Simulator Exercise document by adapting and revising a project I had assigned in the Scala-based offering of CSci 555 (Functional Programming) in Spring 2016.

In 2018, I merged and revised these documents to become new Chapter 22, Data Abstraction Revisited, in the textbook *Exploring Languages with Interpreters and Functional Programming*.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

22.14 Terms and Concepts

Data abstraction; abstract data type (ADT), instance; specification of ADTs using name, sets, signatures, and semantics; constructor, accessor, mutator, and destructor operations; axiomatic and constructive semantics; abstract model (contract, precondition, postcondition, interface and implementation invariant, abstract interface); use of Haskell module hiding features to implement the abstract data type's interface; using mathematical concepts to model the data abstraction (graph, digraph, labelled graph, multigraph, set, sequence, bag, total and partial functions, relation); graph data structure; adventure game.

Mealy Machine, simulator, finite-state automaton (machine), deterministic finite state transducer, state, transition, transition graph.

23 Overloading and Type Classes

23.1 Chapter Introduction

Chapter 5introduced the concept of overloading. Chapters 13 and 21 introduced the related concepts of type classes and instances.

The goals of this chapter (23) and a planned future chapter are to explore these concepts in more detail.

The concept of type class was introduced into Haskell to handle the problem of comparisons, but it has had a broader and more profound impact upon the development of the language than its original purpose. This Haskell feature has also had a significant impact upon the design of subsequent languages (e.g., Scala [132,151] and Rust [110,124,150]) and libraries.

TODO: This chapter, including the Introduction, should be revised after deciding how to handle issues such as functors, monads, etc.

23.2 Polymorphism in Haskell

Chapter 5 surveyed the different kinds of polymorphism. Haskell implements two of these kinds:

1. *Parametric polymorphism* (usually just called "polymorphism" in functional languages), in which a single function definition is used for all types of arguments and results.

   For example, consider the function `length :: [a] -> Int` , which returns the length of any finite list.

2. *Overloading*, in which the same name refers to different functions depending upon the type.

   For example, consider the `(+)` function, which can add any supported number.

Chapter 13 examined parametric polymorphism. Chapter 21 introduced type classes briefly in the context of algebraic data types. This chapter better motives type classes and explores them more generally.

23.3 Why Overloading?

Consider testing for membership in a Boolean list, where `eqBool` is an equality-testing function for Boolean values.

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x []     = False
elemBool x (y:ys) = eqBool x y || elemBool x ys
```

We can define `eqBool` using pattern matching as follows:

```haskell
eqBool :: Bool -> Bool -> Bool
eqBool True  False = False
eqBool False True  = False
eqBool _     _     = True
```

The above is not very general. It works for booleans, but what if we want to handle lists of integers? or of characters? or lists of lists of tuples?

The aspects of `elemBool` we need to generalize are the type of the input list and the function that does the comparison for equality.

Thus let's consider testing for membership of a general list, with the equality function as a parameter.

```haskell
elemGen :: (a -> a -> Bool) -> a -> [a] -> Bool
elemGen eqFun x []     = False
elemGen eqFun x (y:ys) = eqFun x y || elemGen eqFun x ys
```

This allows us to define `elemBool` in terms of `elemGen` as follows:

```haskell
elemBool :: Bool -> [Bool] -> Bool
elemBool = elemGen eqBool
```

But really the function `elemGen` is *too general* for the intended function. Parameter `eqFun` could be any

```haskell
a -> a -> Bool
```

function, not just an equality comparison.

Another problem is that equality is a meaningless idea for some data types. For example, comparing functions for equality is a computationally intractable problem.

The alternative to the above to make `(==)` (i.e., equality) an overloaded function. We can then restrict the polymorphism in `elem`'s type signature to those types for which `(==)` is defined.

We introduce the concept of *type classes* to to be able to define the group of types for which an overloaded operator can apply.

We can then restrict the polymorphism of a type signature to a class by using a *context* constraint as `Eq a =>` is used below:

```haskell
elem :: Eq a => a -> [a] -> Bool
```

We used context constraints in previous chapters. Here we examine how to define the type classes and associate data types with those classes.

23.4 Defining an Equality Class and Its Instances

We can define class `Eq` to be the set of types for which we define the `(==)` (i.e., equality) operation.

For example, we might define the `class` as follows, giving the type *signature(s)* of the associated function(s) (also called the operations or *methods* of the class).

```
class Eq a where
    (==) :: a -> a -> Bool
```

A type is made a member or *instance* of a class by defining the signature function(s) for the type. For example, we might define `Bool` as an `instance` of `Eq` as follows:

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

Other types, such as the primitive types `Int` and `Char` , can also be defined as instances of the class. Comparison of primitive data types will often be implemented as primitive operations built into the computer hardware.

An instance declaration can also be declared with a context constraint, such as in the equality of lists below. We define equality of a list type in terms of equality of the element type.

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x == y && xs == ys
     _     == _      =  False
```

Above, the == on the left sides of the equations is the operation being defined for lists. The `x == y` comparison on the right side is the previously defined operation on elements of the lists. The `xs == ys` on the right side is a recursive call of the equality operation for lists.

Within the class `Eq` , the `(==)` function is overloaded. The definition of `(==)` given for the types of its actual operands is used in evaluation.

In the Haskell standard prelude, the class definition for `Eq` includes both the equality and inequality functions. They may also have *default* definitions as follows:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    -- Minimal complete definition: (==) or (/=)
    x /= y  = not (x == y)
    x == y  = not (x /= y)
```

In the case of class `Eq` , inequality is defined as the negation of equality and vice versa.

An instance declaration must *override* (i.e., redefine) at least one of these functions (in order to break the circular definition), but the other function may either be left with its default definition or overridden.

23.5 Type Class Laws

Of course, our expectation is that any operation `(==)` defined for an instance of `Eq` should implement an "equality" comparison. What does that mean?

In mathematics, we expect equality to be an *equivalence relation*. That is, equality comparisons should have the following properties *for all* values `x`, `y`, and `z` in the type's set.

- **Reflexivity**: `x == x` is `True`.
- **Symmetry**: `x == y` if and only if `y == x`.
- **Transitivity**: if `x == y` and `y == z`, then `x == z`.

In addition, `x /= y` is expected to be equivalent to `not (x == y)` as defined in the default method definition.

Thus class `Eq` has these *type class laws* that every instance of the class should satisfy. The developer of the instance should ensure that the laws hold.

As in many circumstances, the reality of computing may differ a bit from the mathematical ideal. Consider Reflexivity. If `x` is infinite, then it may be impossible to implement `x == x`. Also, this property might not hold for floating point number representations.

23.6 Another Example Class `Visible`

TODO: Perbhaps replace this example (which follows Thompson, ed. 2) with a better one.

We can define another example class `Visible`, which might denote types whose values can be displayed as strings. Method `toString` represents an element of the type as a `String`. Method `size` yields the size of the argument as an `Int`.

```
class Visible a where
    toString :: a -> String
    size     :: a -> Int
```

We can make various data types instances of this class:

```
instance Visible Char where
    toString ch  = [ch]
    size _       = 1

instance Visible Bool where
    toString True  = "True"
    toString False = "False"
    size _         = 1

instance Visible a => Visible [a] where
    toString = concat . map toString
    size     = foldr (+) 1 . map size
```

What type class laws should hold for `Visible`?

There are no constraints on the conversion to strings. However, `size` must return an `Int`, so the "size" of the input argument must be finite and bounded by the largest value in type `Int`.

23.7 Class Extension (Inheritance)

Haskell supports the concept of *class extension*. That is, a new class can be defined that *inherits* all the operations of another class and adds additional operations.

For example, we can derive an ordering class `Ord` from the class `Eq`, perhaps as follows. (The definition in the Prelude may differ from the following.)

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a
    -- Minimal complete definition: (<) or (>)
    x <= y                 = x < y || x == y
    x <  y                 = y > x
    x >= y                 = x > y || x == y
    x >  y                 = y < x
    max x y   | x >= y     = x
              | otherwise  = y
    min x y   | x <= y     = x
              | otherwise  = y
```

With the above, we define `Ord` as a *subclass* of `Eq`; `Eq` is a *superclass* of `Ord`.

The above default method definitions are circular: < is defined in terms of > and vice versa. So a complete definition of `Ord` requires that at least one of these be given an appropriate definition for the type. Method == must, of course, also be defined appropriately for superclass `Eq`.

What type class laws should apply to instances of `Ord`?

Mathematically, we expect an instance of class `Ord` to implement a *total order* on its type set. That is, given the comparison operator (i.e., binary relation) `<=`, then the following properties hold *for all* values `x`, `y`, and `z` in the type's set.

- **Reflexivity**: `x <= x` is `True`.
- **Antisymmetry**: `x <= y` and `y <= x`, then `x == y`.
- **Transitivity**: if `x <= y` and `y <= z`, then `x <= z`.
- **Trichotomy** (comparability, totality): `x <= y` or `y <= x`.

A relation that satisfied the first three properties above is a *partial order*. The fourth property requires that all values in the type's set can be compared by `<=`.

In addition to the above laws, we expect == (and /=) to satisfy the `Eq` type class laws and <, >, >=, `max`, and `min` to satisfy the properties (i.e., default method

definitions) given in the class `Ord` declaration.

As an example, consider the function `isort'` (insertion sort), defined in a previous chapter. It uses class `Ord` to constrain the list argument to ordered data items.

```
isort' :: Ord a => [a] -> [a]
isort' []     = []
isort' (x:xs) = insert' x (isort' xs)

insert' :: Ord a => a -> [a] -> [a]
insert' x []    = [x]
insert' x (y:ys)
    | x <= y    = x:y:ys
    | otherwise = y : insert' x ys
```

23.8 Multiple Constraints

Haskell also permits classes to be constrained by two or more other classes.

Consider the problem of sorting a list and then displaying the results as a string:

```
vSort :: (Ord a,Visible a) => [a] -> String
vSort = toString . isort'
```

To sort the elements, they need to be from an ordered type. To convert the results to a string, we need them to be from a `Visible` type.

The multiple contraints can be over two different parts of the signature of a function. Consider a program that displays the second components of tuples if the first component is equal to a given value:

```
vLookupFirst :: (Eq a,Visible b) => [(a,b)] -> a -> String
vLookupFirst xs x = toString (lookupFirst xs x)

lookupFirst :: Eq a => [ (a,b) ] -> a -> [b]
lookupFirst ws x  = [ z | (y,z) <- ws, y == x ]
```

Multiple constraints can occur in an instance declaration, such as might be used in extending equality to cover pairs:

```
instance (Eq a,Eq b) => Eq (a,b) where
    (x,y) == (z,w) =  x == z && y == w
```

Multiple constraints can also occur in the definition of a class, as might be the case in definition of an ordered visible class.

```
class (Ord a,Visible a) => OrdVis a

vSort :: OrdVis a => [a] -> String
```

The case where a class extends two or more classes, as above for `OrdVis` is called *multiple inheritance*.

Instances of class `OrdVis` must satisfy the type class laws for classes `Ord` and `Visible`.

23.9 Built-In Haskell Classes

See Section 6.3 of the Haskell 2010 Language Report [120:6.3] for discussion of the various classes in the Haskell Prelude library.

23.10 Comparison to Other Languages

Let's compare Haskell concept of type class with the class concept in familiar object-oriented languages such as Java and C++.

- In Haskell, a class is a collection of types. In Java and C++, class and type are similar concepts.

  For example, Java's static type system treats the collection of objects defined with a **`class`** construct as a (nominal) type. A **`class`** can be used to implement a type. However, it is possible to implement classes whose instances can behave in ways outside the discipline of the type (i.e., not satisfy the Liskov Substitution Principle [119,205]).

- Haskell classes are similar in concept to Java and C++ abstract classes except that Haskell classes have no data fields. (There is no multiple inheritance from classes in Java, of course.)

- Haskell classes are similar in concept to Java interfaces. Haskell classes can give default method definitions, a feature that was only added in Java 8 and beyond.

- Instances of Haskell classes are types, not objects. They are somewhat like concrete Java or C++ classes that extend abstract classes or concrete Java classes that implement Java interfaces.

- Haskell separates the definition of a type from the definition of the methods associated with that type. A class in Java or C++ usually defines both a data structure (the member variables) and the functions associated with the structure (the methods). In Haskell, these definitions are separated.

- The methods defined by a Haskell class correspond to the instance methods in Java or virtual functions in a C++ class. Each instance of a class provides its own definition for each method; class defaults correspond to default definitions for a virtual function in the base class. Of course, Haskell class instances do not have implicit receiver object or mutable data fields.

- Methods of Haskell classes are bound statically at compile time, not dynamically bound at runtime as in Java.
- C++ and Java attach identifying information to the runtime representation of an object. In Haskell, such information is attached logically instead of physically to values through the type system.
- Haskell does not support the C++ overloading style in which functions with different types share a common name.
- The type of a Haskell object cannot be implicitly coerced; there is no universal base class such as Java's `Object` which values can be projected into or out of.
- There is no access control (such as public or private class constituents) built into the Haskell class system. Instead, the module system must be used to hide or reveal components of a class. In that sense, it is similar to the object-oriented language Component Pascal [17,176] (which is a variant of Oberon-2 [129]) and to the imperative systems programming language Rust [[110]; McNamara2021; [150]].

Type classes first appeared in Haskell, but similar concepts have been implemented in more recently designed languages.

- The imperative systems programming language Rust [[110]; McNamara2021; [150] supports traits, a limited form of type classes.
- The object-functional hybrid language Scala[132,151] has implicit classes and parameters, which enable a type enrichment programming idiom similar to type classes.
- The functional language PureScript [79,143] supports Haskell-like type classes.
- The dependently typed functional language Idris [18,19] supports interfaces, which are, in some ways, a generalization of Haskell's ty.pe classes.
- Functional JavaScript libraries such as Ramda [147] have type class-like features.

23.11 What Next?

This chapter (23) motivated and explored the concepts of overloading, type classes, and instances in Haskell and compared them to features in other languages.

Chapter 24 further explores the profound impact of type classes on Haskell.

23.12 Chapter Source Code

The source code for this chapter is in file `TypeClassMod.hs`.

23.13 Exercises

TODO

23.14 Acknowledgements

In Spring 2017, I adapted and revised this chapter from my previous notes on this topic [42]. I based the previous notes, in part, on the presentations in:

- Chapter 12 of the Second edition of Simon Thompson's textbook *Haskell: The Craft of Functional Programming* [172]
- Section 5 of *A Gentle Introduction to Haskell Version 98* [103]

For new content on Haskell typeclass laws, I read the discussions of typeclass laws on:

- Typeclassopedia [230]
- StackOverflow
- Reddit

I also reviewed the mathematical definitions of equality, equivalence relations, and total orders on sites as Wolfram MathWorld [226,227,and 228] and Wikipedia [221–223].

In Summer and Fall 2017, I continued to develop this work as Chapter 9, Overloading and Type Classes, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the Overloading and Type Classes chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Most of the existing content became Chapter 23, Overloading and Type Classes. I moved the planned content on advanced type class topics (functors, monads) to a planned future chaper.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

23.15 Terms and Concepts

Polymorphism in Haskell (parametric polymorphism, overloading); Haskell type system concepts (type classes, overloading, instances, signatures, methods, default definitions, context constraints, class extension, inheritance, subclass,

superclass, overriding, multiple inheritance, class laws) versus related Java/C++ type system concepts (abstract and concrete classes, objects, inheritance, interfaces); mathematical concepts (equivalence relation, reflexivity, symmetry, antisymmetry, transitivity, trichotomy, total and partial orders).

24 Future Chapter TBD

24.1 Chapter Introduction

TODO

24.2 What Next?

TODO

24.3 Exercises

TODO

24.4 Acknowledgements

TODO

24.5 Terms and Concepts

TBD

25 Proving Haskell Laws

25.1 Chapter Introduction

The goal of this chapter is to show how to state and prove Haskell "laws".

This chapter depends upon the reader understanding Haskell's polymorphic, higher-order list programming concepts (e.g., from Chapters 4-5, 8-9, and 13-17), but it is otherwise independent of other preceding chapters.

The chapter provides useful tools that can be used in stating and formally proving function and module contracts (Chapters 6, 7, and 22) and type class laws (Chapter 23). It supports reasoning about program generalization (Chapter 19) and type inference (Chapter 24).

The following two chapters on program synthesis (Chapters 26 and 27) build on the concepts and techniques introduced by this chapter.

25.2 Referential Transparency Revisited

Referential transparency is probably the most important property of purely functional programming languages like Haskell.

Chapter 2 defines referential transparency to mean that, within some well-defined context, a variable (or other symbol) always represents the same value. This allows one expression to be replaced by an equivalent expression or, more informally, "equals to be replaced by equals".

Chapter 8 shows how referential transparency underpins the evaluation (i.e., substitution or reduction) model for Haskell and similar functional languages.

In this chapter, we see that referential transparency allows us to state and prove various "laws" or identities that hold for functions and to use these "laws" to transform programs into equivalent ones. Referential transparency underlies how we reason about Haskell programs.

25.3 Stating and Proving Laws

As a purely functional programming language, Haskell supports mathematical reasoning mostly within the programming language itself. We can state properties of functions and prove them using a primarily equational, or calculational, style of proof. The proof style is similar to that of high school trigonometric identities.

25.3.1 Example: ++ associativity and identity element

We have already seen a number of these laws. Again consider the append operator (++) for *finite lists* from Chapter 14.

```
infixr 5 ++
```

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ xs     = xs               -- append.1
(x:xs) ++ ys = x:(xs ++ ys)     -- append.2
```

The append operator ++: has two useful properties that we have already seen.

Associativity: For any finite lists `xs`, `ys`, and `zs`,
    `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.
Identity: For any finite list `xs`,
    `[] ++ xs = xs = xs ++ []`.

Note: The above means that the append operator ++ and the set of finite lists form the algebraic structure called a *monoid*.

How do we prove these properties?

25.3.2 Structural induction proof method

The answer is, of course, *induction*. But we need a type of induction that allows us to prove theorems over the set of all finite lists. In fact, we have already been using this form of induction in the informal arguments that the list-processing functions terminate.

Induction over the natural numbers is a special case of a more general form of induction called *structural induction*. This type of induction is over the syntactic structure of recursively (inductively) defined objects. Such objects can be partially ordered by a complexity ordering from the most simple (minimal) to the more complex.

If we think about the usual axiomization of the natural numbers (i.e., Peano's postulates), then we see that `0` is the only simple (minimal) object and that the successor function `((+) 1)` is the only constructor.

In the case of finite lists, the only simple object is the nil list `[]` and the only constructor is the cons operator `(:)`.

To prove a proposition `P(x)` holds for any finite object `x`, one must prove the following cases.

Base cases: That `P(e)` holds for each simple (minimal) object `e`.

Inductive cases: That, for all object constructors `C`, if `P(x)` holds for some arbitrary object(s) `x`, then `P(C(x))` also holds.

That is, we can *assume* `P(x)` holds, then *prove* that `P(C(x))` holds. This shows that the constructors preserve proposition 'P.

To prove a proposition `P(xs)` holds for any finite list `xs`, the above reduces to the following cases.

Base case `xs = []`: That `P([])` holds.

Inductive case `xs = (a:as)`. That, if `P(as)` holds, then `P(a:as)` also holds.

One, often useful, strategy for discovering proofs of laws is the following:

- Determine whether induction is needed to prove the law. Some laws can be proved directly from the definitions and other previously proved laws.
- Carefully choose the induction variable (or variables).
- Identify the base and inductive cases.
- For each case, use *simplification* independently on each side of the equation. Often, it is best to start with the side that is the most complex.

  Simplification means to substitute the right-hand side of a *definition* or the induction hypothesis for some expression matching the left-hand side.
- Continue simplifying each expression as long as possible.

  Often we can show that the two sides of an equation are the same or that simple manipulations (perhaps using previously proved laws) will show that they are the same.
- If necessary, identify subcases and prove each subcase independently.

A formal proof of a case should, in general, be shown as a calculation that transforms one side of the equation into the other by substitution of equals for equals.

This formal proof can be constructed from the calculation suggested in the above

25.3.3 Proving associativity of ++

Now that we have the mathematical machinery we need, let's prove that ++ is associative for all finite lists. The following proofs assume that all arguments of the functions are defined.

Prove: For any finite lists `xs`, `ys`, and `zs`,
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.

Proof:

There does not seem to be a non-inductive proof, thus we proceed by structural induction over the finite lists. But on which variable(s)?

By examining the definition of ++, we see that it has two legs differentiated by the value of the left operand. The right operand is not decomposed. To use this definition in the proof, we need to consider the left operands of the ++ in the associative law.

Thus we choose to do the induction on `xs`, the leftmost operand, and consider two cases—a base case and an inductive case.

Base case `xs = []`:

First, we simplify the left-hand side.

```
    [] ++ (ys ++ zs)
= { append.1 (left to right), omit outer parentheses }
    ys ++ zs
```

We do not know anything about `ys` and `zs`, so we cannot simplify further.

Next, we simplify the right-hand side.

```
    ([] ++ ys) ++ zs
= { append.1 (left to right), omit parentheses around ys }
    ys ++ zs
```

Thus we have simplified the two sides to the same expression.

Of course, a formal proof can be written more elegantly as:

```
    [] ++ (ys ++ zs)
= { append.1 (left to right) }
    ys ++ zs
= { append.1 (right to left, applied to left operand) }
    ([] ++ ys) ++ zs
```

Thus the base case is established.

Note the equational style of reasoning. We proved that one expression was equal to another by beginning with one of the expressions and repeatedly substituting "equals for equals" until we got the other expression.

Each transformational step was justified by a definition, a known property, or (as we see later) the induction hypothesis. We normally do not state justifications like "omit parentheses" or "insert parentheses". We show these justifications for these steps in braces in the equational arguments. This style follows the common practice in the program derivaton community [40,40,85].

In the inductive case, we find it helpful to state both the inductive assumption and the proof goal explicitly, as we do below.

Inductive case `xs = (a:as)`:

Assume `as ++ (ys ++ zs) = (as ++ ys) ++ zs`;
prove `(a:as) ++ (ys ++ zs) = ((a:as) ++ ys) ++ zs`.

First, we simplify the left-hand side.

```
    (a:as) ++ (ys ++ zs)
= { append.2 (left to right) }
```

```
    a:(as ++ (ys ++ zs))
= { induction hypothesis }
    a:((as ++ ys) ++ zs)
```

We do not know anything further about `as`, `ys`, and `zs`, so we cannot simplify further.

Next, we simplify the right-hand side.

```
    ((a:as) ++ ys) ++ zs
= { append.2 (left to right, on inner ++) }
    (a:(as ++ ys)) ++ zs
= { append.2 (left to right, on outer ++) }
    a:((as ++ ys) ++ zs)
```

Thus we have simplified the two sides to the same expression.

Again, a formal proof can be written more elegantly as follows.

```
    (a:as) ++ (ys ++ zs)
= { append.2 (left to right) }
    a:(as ++ (ys ++ zs))
= { induction hypothesis }
    a:((as ++ ys) ++ zs)
= { append.2 (right to left, on outer ++) }
    (a:(as ++ ys)) ++ zs
= { append.2 (right to left, on inner ++) }
    ((a:as) ++ ys) ++ zs
```

Thus the inductive case is established.

Therefore, we have proven the ++ associativity property. **Q.E.D.**

The above proof and the ones that follow assume that the arguments of the functions are all defined (i.e., not equal to $\perp$).

25.3.4 Reviewing proof method

You should practice writing proofs in the “more elegant” form given above. This end-to-end calculational style is more useful for synthesis of programs.

Reviewing what we have done, we can identify the following guidelines:

- Determine whether induction is really needed.

- Choose the induction variable carefully.
- Be careful with parentheses.

  Substitutions, comparisons, and pattern matches must done with the fully parenthesized forms of definitions, laws, and expressions in mind, that is, with parentheses around all binary operations, simple objects, and the entire expression. We often omit "unneeded" parentheses to make the expression more readable.

- Start with the more complex side of the equation.

  That gives us more information with which to work.

25.3.5 Proving identity element for ++

Now let's prove the identity property.

Prove: For any finite list `xs`,
`[] ++ xs = xs = xs ++ []`.

Proof:

The equation `[] ++ xs = xs` follows directly from `append.1`. Thus we consider the equation `xs ++ [] = xs`, which we prove by structural induction on `xs`.

Base case `xs = []`:

```
[] ++ []
```

= { `append.1` (left to right) }

```
[]
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `as ++ [] = as`; *prove* `(a:as) ++ [] = (a:as)`.

```
(a:as) ++ []
```

= { `append.2` (left to right) }

```
a:(as ++ [])
```

= { induction hypothesis }

```
a:as
```

This establishes the inductive case.

Therefore, we have proved that `[]` is the *identity element* for ++. **Q.E.D.**

25.4 Example: Relating `length` and `++`

Suppose that the list `length` function is defined as follows (from Chapter 13}).

```
length :: [a] -> Int
length []     =  0                -- length.1
length (_:xs) =  1 + length xs    -- length.2
```

Prove: For all finite lists `xs` and `ys`:
`length (xs++ys) = length xs + length ys`.

Proof:

Because of the way ++ is defined, we choose `xs` as the induction variable.

Base case `xs = []`:

```
length [] + length ys
```

= { `length.1` (left to right) }

```
0 + length ys
```

= { 0 is identity for addition }

```
length ys
```

= { `append.1` (right to left) }

```
length ([] ++ ys)
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `length (as ++ ys) = length as + length ys`;
prove `length ((a:as) ++ ys) = length (a:as) + length ys`.

```
length ((a:as) ++ ys)
```

= { `append.2` (left to right) }

```
length (a:(as ++ ys))
```

= { `length.2` (left to right) }

```
1 + length (as ++ ys)
```

= { induction hypothesis }

```
1 + (length as + length ys)
```

= { associativity of addition }

```
(1 + length as) + length ys
```

= { `length.2` (right to left, value of `a` arbitrary) }

```
length (a:as) + length ys
```

This establishes the inductive case.

Therefore, `length (xs ++ ys) = length xs + length ys`. **Q.E.D.**

Note: The proof above uses the associativity and identity properties of integer addition.

25.5 Example: Relating `take` and `drop`

Remember the definitions for the list functions `take` and `drop` from Chapter 13}.

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []                  -- take.1
take _ []         = []                  -- take.2
take n (x:xs)     = x : take (n-1) xs -- take.3

drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs                 -- drop.1
drop _ []          = []                 -- drop.2
drop n (_:xs)      = drop (n-1) xs     -- drop.3
```

Prove: For any natural numbers `n` and finite lists `xs`,
`take n xs ++ drop n xs = xs`.

Proof:

Note that both `take` and `drop` use both arguments to distinguish the cases. Thus we must do an induction over all natural numbers `n` and all finite lists `xs`.

We would expect four cases to consider, the combinations from `n` being zero and nonzero and `xs` being nil and non-nil. But an examination of the definitions for the functions reveal that the cases for `n = 0` collapse into a single case.

Base case `n = 0`:

```
take 0 xs ++ drop 0 xs
```

= { `take.1`, `drop.1` (both left to right) }

```
[] ++ xs
```

= { ++ identity `xs` }

```
xs
```

This establishes the case.

Base case `n = m+1`, `xs = []`:

```
take (m+1) [] ++ drop (m+1) []
```

= { `take.2`, `drop.2` (both left to right) }

```
[] ++ []
```

= { ++ identity }

```
[]
```

This establishes the case.

Inductive case `n = m+1`, `xs = (a:as)`:

Assume `take m as ++ drop m as = as`;
prove `take (m+1) (a:as) ++ drop (m+1) (a:as) = (a:as)`.

```
take (m+1) (a:as) ++ drop (m+1) (a:as)
```

= { `take.3`, `drop.3` (both left to right) }

```
(a:(take m as)) ++ drop m as
```

= { `append.2` (left to right) }

```
a:(take m as ++ drop m as)
```

= { induction hypothesis }

```
(a:as)
```

This establishes the case.

Therefore, the property is proved. **Q.E.D.**

25.6 Example: Equivalence of Functions

What do we mean when we say two functions are equivalent?

Usually, we mean that the "same inputs" yield the "same outputs". For example, single argument functions `f` and `g` are equivalent if `f x` = `g x` for all `x`.

In Chapter 14. we defined two versions of a function to reverse the elements of a list. Function `rev` uses backward recursion and function `reverse` (called `reverse'` in Chapter 14) uses a forward recursive auxiliary function `rev'`.

```
rev :: [a] -> [a]
rev []      =  []                          -- rev.1
rev (x:xs)  =  rev xs ++ [x]               -- rev.2

reverse :: [a] -> [a]
reverse xs  =  rev' xs []                  -- reverse.1
    where rev' [] ys     = ys              -- reverse.2
          rev' (x:xs) ys = rev' xs (x:ys)  -- reverse.3
```

To show `rev` and `reverse` are equivalent, we must prove that, for all finite lists `xs`:

```
rev xs = reverse xs
```

If we unfold (i.e., simplify) `reverse` one step, we see that we need to prove:

```
rev xs = rev' xs []
```

Thus let's try to prove this by structural induction on `xs`.

Base case `xs = []`:

```
    rev []
= { rev.1 (left to right) }
    []
= { reverse.2 (right to left) }
    rev' [] []
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `rev as = rev' as []`; *prove* `rev (a:as) = rev' (a:as) []`.

First, we simplify the left side.

```
    rev (a:as)
= { rev.2 (left to right) }
    rev as ++ [a]
```

Then, we simplify the right side.

```
    rev' (a:as) []
= { reverse.3 (left to right) }
    rev' as [a]
```

Thus we need to show that `rev as ++ [a] = rev' as [a]`. But we do not know how to proceed from this point.

Maybe another induction. But that would probably just bring us back to a point like this again. We are stuck!

Let's look back at `rev xs = rev' xs []`. This is difficult to prove directly. Note the asymmetry, one argument for `rev` versus two for `rev'`.

Thus let's look for a new, more symmetrical, problem that might be easier to solve. Often it is easier to find a solution to a problem that is symmetrical than one which is not.

Note the place we got stuck above (proving `rev as ++ [a] = rev' as [a]`) and also note the equation `reverse.3`. Taking advantage of the identity element for `++`, we can restate our property in a more symmetrical way as follows:

```
rev xs ++ [] = rev' xs []
```

Note that the constant `[]` appears on both sides of the above equation. We can now apply the following generalization heuristic [41,85]. (That is, we try to solve a "harder" problem.)

Heuristic: *Replace constant by variable*

> That is, generalize by replacing a constant (or any subexpression) by a new variable.

Thus we try to prove the more general proposition:

```
rev xs ++ ys = rev' xs ys
```

The case `ys = []` gives us what we really want to hold. Intuitively, this new proposition seems to hold. Now let's prove it formally. Again we try structural induction on `xs`.

Base case `xs = []`:

```
    rev [] ++ ys
```

= { `rev.1` (left to right) }

```
    [] ++ ys
```

= { `append.1` (left to right) }

```
    ys
```

= { `reverse.2` (right to left) }

```
    rev' [] ys
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `rev as ++ ys = rev' as ys` for any finite list `ys`; *prove* `rev (a:as) ++ ys = rev' (a:as) ys`.

```
    rev (a:as) ++ ys
```

= { `rev.2` (left to right) }

```
    (rev as ++ [a]) ++ ys
```

= { ++ associativity, Note 1 }

```
    rev as ++ ([a] ++ ys)
```

= { singleton law, Note 2 }

```
    rev as ++ (a:ys)
```

= { induction hypothesis }

```
    rev' as (a:ys)
```

= { `reverse.3` (right to left) }

```
rev' (a:as) ys
```

This establishes the inductive case.

Notes:

1. We could apply the induction hypothesis here, but it does not seem profitable. Keeping the expressions in terms of `rev` and `++` as long as possible seems better; we know more about those expressions.

2. The *singleton law* is `[x] ++ xs = x:xs` for any element `x` and finite list `xs` of the same type. Proof of this is left as an exercise for the reader.

Therefore, we have proved `rev xs ++ ys = rev' xs ys` and, hence:

```
rev xs = reverse xs
```

The key to the performance improvement here is the solution of a "harder" problem: function `rev'` does both the reversing and appending of a list while `rev` separates the two actions.

25.7 What Next?

This chapter illustrated how to state and prove Haskell "laws" about already defined functions.

Chapters 26} and 27} on *program synthesis* illustrate how to use similar reasoning methods to synthesize (i.e., derive or calculate) function definitions from their specifications.

25.8 Exercises

This set of exercises uses functions defined in this and previous chapters including the following:

- Functions `map`, `filter`, `foldr`, `foldl`, and `concatMap` are defined in Chapter 15.

- Functional composition, identity combinator `id`, and function `all` are defined in Chapter 16}.

- Functions `takeWhile` and `dropWhile` are defined in Chapter 17.

Prove the following properties using the proof methods illustrated in this chapter.

1. Prove for all `x` of some type and finite lists `xs` of the same type (i.e., the *singleton law*):

   ```
   [x] ++ xs  =  (x:xs)
   ```

2. Consider the definition for `length` given in the text of this chapter and the following definition for `len`:

```
len :: Int -> [a] -> Int
len n [ ]     = n                -- len.1
len n (_:xs) = len (n+1) xs   -- len.2
```

Prove for any finite list `xs`: `len 0 xs = length xs`.

3. Prove for all finite lists `xs` and `ys` of the same type:

   ```
   reverse (xs ++ ys)  =  reverse ys ++ reverse xs
   ```

   Hint: The function `reverse` (called`reverse'` in Chapter 14.) uses forward recursion. Backward recursive definitions are generally easier to use in inductive proofs. In Chapter 14., we also defined a backward recursive function `rev` and proved that `rev xs = reverse xs` for all finite lists `xs`. Thus, you may find it easier to substitute `rev` for `reverse` and instead prove:

   ```
   rev (xs ++ ys)  =  rev ys ++ rev xs
   ```

4. Prove for all finite lists `xs` of some type:

   ```
   reverse (reverse xs)  =  xs
   ```

5. Prove for all natural numbers `m` and `n` and all finite lists `xs`:

   ```
   drop n (drop m xs)  =  drop (m+n) xs
   ```

6. Consider the rational number package from Chapter 7.. Prove for any `Rat` value `r` that satisfied the interface invariant for the abstract module `RationalRep`:

   ```
   addRat r zeroRat  =  r  =  addRat zeroRat r
   ```

7. Consider the two definitions for the Fibonacci function in Chapter 9. Prove for any natural number `n`:

   ```
   fib n  =  fib' n
   ```

   Hint: First prove, for `n` ≥ 2:

   ```
   fib'' n p q = fib'' (n-2) p q + fib'' (n-1) p q
   ```

8. Prove that the `id` function is the identity element for functional composition. That is, for any function `f :: a ->     b`, prove:

   ```
   f . id  =  f  =  id . f
   ```

9. Prove that functional composition is associative. That is, for any function `f :: a -> a`, `g :: a -> a`, and `h :: a -> a`, prove:

   ```
   (f . g) . h  =  f . (g . h)
   ```

10. Prove for all finite lists `xs` and `ys` of the same type and function `f` on that type:

    ```
    map f (xs ++ ys)  =  map f xs ++ map f ys
    ```

11. Prove for all finite lists `xs` and `ys` of the same type and predicate `p` on that type:

```
filter p (xs ++ ys)  =  filter p xs ++ filter p ys
```

12. Prove for all finite lists `xs` and `ys` of the same type and all predicates `p` on that type:

```
all p (xs ++ ys)  =  (all p xs) && (all p ys)
```

The definition for `&&` is as follows:

```
(&&) :: Bool -> Bool -> Bool
False && x = False   -- second argument not evaluated
True  && x = x       -- second argument returned
```

13. Prove for all finite lists `xs` of some type and predicates `p` and `q` on that type:

```
filter p (filter q xs)  =  filter q (filter p xs)
```

14. Prove for all finite lists `xs` and `ys` of the same type and for all functions `f` and values `a` of compatible types:

```
foldr f a (xs ++ ys)  =  foldr f (foldr f a ys) xs
```

15. Prove for all finite lists `xs` of some type and all functions `f` and `g` of conforming types:

```
map (f . g) xs  =  (map f . map g) xs
```

16. Prove for all finite lists of finite lists `xss` of some base type and function `f` on that type:

```
map f (concat xss)  =  concat (map (map f) xss)
```

17. Prove for all finite lists `xs` of some type and functions `f` on that type:

```
map f xs  =  foldr ((:) .f) [] xs
```

18. Prove for all lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs  =  xs
```

19. Prove that, if `***` is an associative binary operation of type `t -> t` with identity element `z` (i.e., a monoid), then:

```
foldr (***) z xs  =  foldl (***) z xs
```

20. Consider the Haskell type for the natural numbers given in an exercise in Chapter 21.

```
data Nat = Zero | Succ Nat
```

For the functions defined in that exercise, prove the following:

   a. Prove that `intToNat` and `natToInt` are inverses of each other.

b. Prove that `Zero` is the (right and left) identity element for `addNat`.

c. Prove for any `Nat`s `x` and `y`:

```
addNat (Succ x) y  =  addNat x (Succ y)
```

d. Prove associativity of addition on `Nat`'s. That is, for any `Nat`s `x`, `y`, and `z`:

```
addNat x (addNat y z)  =  addNat (addNat x y) z
```

e. Prove commutativity of addition on `Nat`'s. That is, for any `Nat`s `x` and `y`:

```
addNat x y  =  addNat y x
```

25.9 Acknowledgements

In Summer 2018, I adapted and revised this chapter from Chapter 11 of my *Notes on Functional Programming with Haskell* [42].

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [15] and other functional programming sources [13,14,98,171,178]. They were also influenced by my research, study, and teaching related to program specification, verification, derivation, and semantics [[28]; [34]; [39]; [40]; [41]; [64]; [65]; [66]; [85]; [86]; [107]; vanGesteren1990].

I incorporated this work as new Chapter 25, Proving Haskell Laws, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my

post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

25.10 Terms and Concepts

Referential transparency, equational reasoning, laws, definition, simplification, calculation, associativity, identity, monoid, singleton law, equivalence of functions.

26 Program Synthesis

26.1 Chapter Introduction

Chapter 25 illustrated how to state and prove Haskell "laws" about already defined functions.

This chapter (26) illustrates how to use similar reasoning methods to synthesize (i.e., derive or calculate) function definitions from their specifications.

Chapter 27 applies these program synthesis techniques to a larger set of examples on text processing.

26.2 Motivation

This chapter deals with program synthesis.

In the *proof* of a property, we take an existing program and then demonstrate that it satisfies some property.

In the *synthesis* of a program, we take a property called a *specification* and then synthesize a program that satisfies it [15]. (Program synthesis is called program *derivation* in other contexts, such as in the Gries textbook [85] and my *Notes on Program Semantics and Derivation* [41].)

Both proof and synthesis require essentially the same reasoning. Often a proof can be turned into a synthesis by simply reversing a few of the steps, and vice versa.

26.3 Fast Fibonacci Function

This section is based on Bird and Wadler [15:5.4.5,15:5.5] and Hoogerwoord [98:4.5].

A (second-order) Fibonacci sequence is the sequence in which the first two elements are 0 and 1 and each successive element is the sum of the two immediately preceding elements:

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

As we have seen in Chapter 9, we can take the above informal description and define a function to compute the `n`th element of the Fibonacci sequence. The definition is straightforward. Unfortunately, this algorithm is quite inefficient, O(`fib n`).

```
fib :: Int -> Int
fib 0          = 0
fib 1          = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
```

In Chapter 9, we also developed a more efficient, but less straightforward, version by using two accumulating parameters. This definition seemed to be "pulled out of thin air". Can we synthesize a definition that uses the more efficient algorithm from the simpler definition above?

Yes, but we use a slightly different approach than we did before. We can improve the performance of the Fibonacci computation by using a technique called *tupling* [98] as we saw in Chapter 20.

The tupling technique can be applied to a set of functions with the same domain and the same recursive pattern. First, we define a new function whose value is a tuple, the components of which are the values of the original functions. Then, we proceed to calculate a recursive definition for the new function.

This technique is similar to the technique of adding accumulating parameters to define a new function.

Given the definition of `fib` above, we begin with the specification [15]:

```
twofib n = (fib n, fib (n+1))
```

and synthesize a recursive definition by using induction on the natural number `n`.

Base case `n = 0`:

```
twofib 0
```

= { specification }

```
(fib 0, fib (0+1))
```

= { arithmetic, `fib.1`, `fib.2` }

```
(0,1)
```

This gives us a definition for the base case.

Inductive case `n = m+1`:

Given that there is a definition for `twofib m` that satisfies the specification

```
twofib m = (fib m, fib (m+1))
```

calculate a definition for `twofib (m+1)` that satisfies the specification.

```
twofib (m+1)
```

= { specification }

```
(fib (m+1), fib ((m+1)+1))
```

= { arithmetic, `fib.3` }

```
(fib (m+1), fib m + fib (m+1))
```

= { abstraction }

```
(b,a+b)
where (a,b) = (fib m, fib (m+1))
```

= { induction hypothesis }

```
(b,a+b)
where (a,b) = twofib m
```

This gives us a definition for the inductive case.

Bringing the cases together and rewriting `twofib (m+1)` to get a valid pattern, we synthesize the following definition:

```
twofib :: Int -> (Int,Int)
twofib 0         = (0,1)
twofib n | n > 0 = (b,a+b)
                   where (a,b) = twofib (n-1)

fastfib :: Int -> Int
fastfib n = fst (twofib n)
```

Above `fst` is the standard prelude function to extract the first component of a pair (i.e., a 2-tuple).

The key to the performance improvement is solving a "harder" problem: computing `fib n` and `fib (n+1)` at the same time. This allows the values needed to be "passed forward" to the "next iteration".

In general, we can approach the synthesis of a function using the following method.

- Devise a specification for the function in terms of defined functions, data, etc.
- Assume the specification holds.
- Using proof techniques (as if proving the specification), calculate an appropriate definition for the function.
- As needed, break the synthesis calculation into cases motivated by the induction "proof" over an appropriate (well-founded) set (e.g., over natural numbers or finite lists). The inductive cases usually correspond to recursive legs of the definition.

26.4 Sequence of Fibonacci Numbers

Now let's consider a function to generate a list of the elements `fib 0` through `fib n` for some natural number `n`. A simple backward recursive definition follows:

```
allfibs :: Int -> [Int]
allfibs 0         = [0]                          -- allfibs.1
allfibs n | n > 0 = allfibs (n-1) ++ [fib n]     -- allfibs.2
```

Using `fastfib`, each `fib n` calculation is O(`n`). Each ++ call is also O(`n`). The `fib` and the ++ are "in sequence", so each call of `allfibs` is just O(`n`). However, there are O(`n`) recursive calls of `allfibs`, so the overall complexity is O(`n^2`).

We again attempt to improve the efficiency by tupling We begin with the following specification for `fibs`:

```
fibs n = (fib n, fib (n+1), allfibs n)
```

We already have definitions for the functions on the right-hand side, `fib` and `allfibs`. Our task now is to synthesize a definition for the left-hand side, `fibs`.

We proceed by induction on the natural number `n` and consider two cases.

Base case `n = 0`:

```
fibs 0
```

= { `fibs` specification }

```
(fib 0, fib (0+1), allfibs 0)
```

= { `fib.1`, `fib.2`, `allfibs.1` }

```
(0,1,[0])
```

This gives us a definition for the base case.

Inductive case `n = m+1`

Given that there is a definition for `fibs m` that satisfies the specification

```
fibs m = (fib m, fib (m+1), allfibs m)
```

calculate a definition for `fibs (m+1)` that satisfies the specification.

```
fibs (m+1)
```

= { `fibs` specification }

```
(fib (m+1), fib (m+2), allfibs (m+1))
```

= { `fib.3`, `allfibs.2` }

```
(fib (m+1), fib m + fib (m+1), allfibs m ++ [fib (m+1)])
```

= { abstraction }

```
(b,a+b,c++[b])
where (a,b,c) = (fib m, fib (m+1), allfibs m)
```

= { induction hypothesis }

```
(b,a+b,c++[b])
where (a,b,c) = fibs m
```

This gives us a definition for the inductive case.

Bringing the cases together, we get the following definitions:

```
fibs :: Int -> (Int,Int,[Int])
fibs 0         = (0,1,[0])
fibs n | n > 0 = (b,a+b,c++[b])
                 where (a,b,c) = fibs (n-1)

allfibs1 :: Int -> [Int]
allfibs1 n = thd3 (fibs n)
```

Above `thd3` is the standard prelude function to extract the third component of a 3-tuple.

We have eliminated the O(`n`) `fib` calculations, but still have an O(`n`) append (++) within each of the O(`n`) recursive calls of `fibs`. This program is better, but is still O(`n^2`).

Note that in the `c ++ [b]` expression there is a single element on the right. Perhaps we could build this term backwards using cons, an O(1) operation, and then reverse the final result.

We again attempt to improve the efficiency by tupling. We begin with the following specification for `fibs`:

```
fibs' n = (fib n, fib (n+1), reverse (allfibs n))
```

For convenience in calculation, we replace `reverse` by its backward recursive equivalent `rev`.

```
rev :: [a] -> [a]
rev []     = []              -- rev.1
rev (x:xs) = rev xs ++ [x]   -- rev.2
```

We again proceed by induction on `n` and consider two cases.

Base case `n = 0`:

```
fibs' 0
```

= { `fibs'` specification }

```
(fib 0, fib (0+1), rev (allfibs 0))
```

= { `fib.1`, `fib.2`, `allfibs.1` }

```
(0,1, rev [0])
```

= { `rev.2` }

```
(0,1, rev [] ++ [0])
```

= { `rev.1`, `append.1` }

```
(0,1,[0])
```

This gives us a definition for the base case.

Inductive case `n = m+1`:

Given that there is a definition for `fibs' m` that satisfies the specification

```
fibs' m = (fib m, fib (m+1), allfibs m)
```

calculate a definition for `fibs' (m+1)` that satisfies the specification.

```
    fibs' (m+1)
= { fibs' specification }
    (fib (m+1), fib (m+2), rev (allfibs (m+1)))
= { fib.3, allfibs.2 }
    (fib (m+1), fib m + fib (m+1), rev (allfibs m ++ [fib (m+1)]))
= { abstraction }
    (b, a+b, rev (allfibs m ++ [b]))
    where (a,b,c) = (fib m, fib (m+1), rev (allfibs m))
= { induction hypothesis }
    (b, a+b, rev (allfibs m ++ [b]))
    where (a,b,c) = fibs' m
= { rev (xs ++ [x]) = x : rev xs, Note 1 }
    (b, a+b, b : rev (allfibs m))
    where (a,b,c) = fibs' m
= { substitution }
    (b, a+b, b:c)
    where (a,b,c) = fibs' m
```

This gives us a definition for the inductive case.

Note 1: The proof of `rev (xs ++ [x]) = x : rev xs` is left as an exercise.

Bringing the cases together, we get the following definition:

```
fibs' :: Int -> (Int,Int,[Int])
fibs' 0         = (0,1,[0])
fibs' n | n > 0 = (b,a+b,b:c)
                   where (a,b,c) = fibs' n

allfibs2 :: Int -> [Int]
allfibs2 n = reverse (thd3 (fibs' n))
```

Function `fibs'` is O(`n`). Hence, `allfibs2'` is O(`n`).

Are further improvements possible?

Clearly, function `fibs'` must generate an element of the sequence for each integer in the range `[0..n]`. Thus no complexity order improvement is possible.

However, from our previous experience, we know that it should be possible to avoid doing a reverse by using a tail recursive auxiliary function to compute the Fibonacci sequence. The investigation of this possible improvement is left to the reader.

For an O(`log2 n`) algorithm to compute `fib n`, see Kaldewaij's textbook on program derivation [107:5.2]. (As in Chapter 9, we assume `log2` is a function that computes the logarithm with base 2.)

26.5 Synthesis of `drop` from `take`

Suppose that we have the following definition for the list function `take`, but no definition for `drop`.

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take' (n-1) xs
```

Further suppose that we wish to synthesize a definition for `drop` that satisfies the following specification for any natural number `n` and finite list `xs`.

```
take n xs ++ drop n xs = xs
```

We proved this as a property earlier, given definitions for both `take` and `drop`. The synthesis uses induction on both `n` and `xs` and the same cases we used in the proof.

Base case `n = 0`:

```
xs
```

= { specification, substitution for this case }

```
take 0 xs ++ drop 0 xs
```

= { `take.1` }

```
[] ++ drop 0 xs
```

= { `++` identity }

```
drop 0 xs
```

This gives the equation `drop 0 xs = xs`.

Base case `n = m+1`:

```
[]
```

= { specification, substitution for this case }

```
take (m+1) [] ++ drop (m+1) []
```

= { `take.2` }

```
    [] ++ drop (m+1) []
= { ++ identity }
    drop (m+1) []
```

This gives the defining equation `drop (m+1) [] = []`. Since the value of the argument `(m+1)` is not used in the above calculation, we can generalize the definition to `drop _ [] = []`.

Inductive case `n = m+1, xs = (a:as)`:

Given that there is a definition for `drop m as` that satisfies the specification:

```
    take m as ++ drop m as = as
```

calculate an appropriate definition for `drop (m+1) (a:as)` that satisfies the specification.

```
    (a:as)
= { specification, substitution for this case }
    take (m+1) (a:as) ++ drop (m+1) (a:as)
= { take.3 }
    (a:(take m as)) ++ drop (m+1) (a:as)
= { append.2 }
    a:(take m as ++ drop (m+1) (a:as))
```

Hence, `a:(take m as ++ drop (m+1) (a:as)) = (a:as)`.

```
    a:(take m as ++ drop (m+1) (a:as)) = (a:as)
≡ { axiom of equality of lists (Note 1) }
    take m as ++ drop (m+1) (a:as) = as
≡ { m ≥ 0, specification }
    take m as ++ drop (m+1) (a:as) = take m as ++ drop m as
≡ { equality of lists (Note 2) }
    drop (m+1) (a:as) = drop m as
```

Because of the induction hypothesis, we know that `drop m as` is defined. This gives a definition for this case.

Notes:

0. The symbol ≡ denotes logical equivalence (i.e., if and only if) and is pronounced "equivales".
1. `(x:xs) = (y:ys) ≡ x = y && xs = ys`. In this case `x` and `y` both equal `a`.

2. `xs ++ ys = xs ++ zs` ≡ `ys = zs` can be proved by induction on `xs` using the Note 1 property.

Bringing the cases together, we get the definition that we saw earlier.

```
drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs              -- drop.1
drop _ []          = []              -- drop.2
drop n (_:xs)      = drop (n-1) xs   -- drop.3
```

26.6 Tail Recursion Theorem

In Chapter 14, we looked at two different definitions of a function to reverse the elements of a list. Function `rev` uses a straightforward backward linear recursive technique and `reverse` uses a tail recursive auxiliary function. We proved that these definitions are equivalent.

```
rev :: [a] -> [a]
rev []       =   []              -- rev.1
rev (x:xs)   =   rev xs ++ [x]   -- rev.2

reverse :: [a] -> [a]
reverse xs    =   rev' xs []                        -- reverse.1
    where rev' [] ys      = ys                      -- reverse.2
          rev' (x:xs) ys   = rev' xs (x:ys)         -- reverse.3
```

Function `rev'` is a *generalization* of `rev`. Is there a way to calculate `rev'` from `rev`?

Yes, by using the Tail Recursion Theorem for lists. We develop this theorem in a more general setting than `rev`.

The following is based on Hoogerwoord [98:4.7].

For some types `X` and `Y`, let function `fun` be defined as follows:

```
fun :: X -> Y
fun x | not (b x) = f x                  -- fun.1
      | b x       = h x *** fun (g x)    -- fun.2
```

- Functions `b`, `f`, `g`, `h`, and `***` are *not* defined in terms of `fun`.
- `b :: X -> Bool` such that, for any `x`, `b x` is defined whenever `fun x` is defined.
- `g :: X -> X` such that, for any `x`, `g x` is defined whenever `fun x` is defined and `b x` holds.
- `h :: X -> Y` such that, for any `x`, `h x` is defined whenever `fun x` is defined and `b x` holds.

- `(***) :: Y -> Y -> Y` such that operation `***` is defined for all elements of `Y` and is an associative operation with left identity `e`.
- `f :: X -> Y` such that, for any `x`, `f x` is defined whenever `fun x` is defined and `not (b x)` holds.
- `X` with relation $\prec$ admits induction (i.e., $\langle$`X`$, \prec\rangle$ is a *well-founded ordering*).
- For any `x`, if `fun x` is defined and `b x` holds, then `g x` $\prec$ `x`.

Note that both `fun x` and the recursive leg `h x *** fun (g x)` have the general structure `y *** fun z` for some expressions `y` and `z` (i.e., `fun x = e *** fun x`). Thus we specify a more general function `fun'` such that

```
fun' :: Y -> X -> Y
fun' y x = y *** fun x
```

and such that `fun'` is defined for any `x` $\in$ `X` for which `fun x` is defined.

Given the above specification, we note that:

```
    fun' e x
= { fun' specification }
    e *** fun x
= { e is the left identity for *** }
    fun x
```

We proceed by induction on the type `X` with $\prec$. (We are using *well-founded induction*, a more general form of induction than we have used before.

We have two cases. The base case is when `not (b x)` holds for argument `x` of `fun'`. The inductive case is when `b x` (i.e, `g x` $\prec$ `x`).

Base case `not (b x)`: (That is, `x` is a minimal element of `X` under $\prec$.)

```
    fun' y x
= { fun' specification }
    y *** fun x
= { fun.1 }
    y *** f x
```

Inductive case `b x`: (That is, `g x` $\prec$ `x`.)

Given that there is a definition for `fun' y (g x)` that satisfies the specification for any `y`

```
    fun' y (g x) = y *** fun (g x)
```

calculate a definition for `fun' y x` that satisfies the specification.

```
    fun' y x
= { fun' specification }
    y *** fun x
= { fun.2 }
    y *** (h x *** fun (g x))
= { *** associativity }
    (y *** h x) *** fun (g x)
= { g x ≺ x, induction hypothesis }
    fun' (y *** h x) (g x)
```

Thus we have synthesized the following tail recursive definition for function `fun'` and essentially proved the Tail Recursion Theorem shown below.

```
fun' :: Y -> X -> Y
fun' y x | not (b x) = y *** f x              -- fun'.1
         | b x       = fun' (y *** h x) (g x) -- fun'.2
```

Note that the first parameter of `fun'` is an *accumulating parameter*.

Tail Recursion Theorem: If `fun`, `fun'`, and `e` are defined as given above, then `fun x = fun' e x`.

Now let's consider the `rev` and `rev'` functions again. First, let's rewrite the definitions of `rev` in a form similar to the definition of `fun`.

```
rev :: [a] -> [a]
rev xs | xs == [] = []                          -- rev.1
       | xs /= [] = rev (tail xs) ++ [head xs]  -- rev.2
```

For `rev` we substitute the following for the components of the `fun` definition:

- `fun x` ← `rev xs`
- `b x` ← `xs /= []`
- `g x` ← `tail xs`
- `h x` ← `[head xs]`
- `l *** r` ← `r ++ l` (Note the flipped operands,)
- `f x` ← `[]`
- `l` ≺ `r` ← `(length l) < (length r)`
- `e` ← `[]`
- `fun' y x` ← `rev' xs ys` (Note the flipped arguments.)

Thus, by applying the tail recursion theorem, `fun'` becomes the following:

```
rev' :: [a] -> [a] -> [a]
rev' xs ys
    | xs == [] = ys                             -- rev'.1
    | xs /= [] = rev' (tail xs) ([head xs]++ys)  -- rev'.2
```

From the Tail Recursion Theorem, we conclude that `rev xs = rev' xs []`.

Why would we want to convert a backward linear recursive function to a tail recursive form?

- A tail recursive definition is sometimes more space efficient (as we saw in Chapter 9).

  This is especially the case if the strictness of an accumulating parameter can be exploited (as we saw in Chapters 9 and 15).

- A tail recursive definition sometimes allows the replacement of an "expensive" operation (requiring many steps) by a less "expensive" one. (For example, ++ is replaced by cons in the transformation from `rev` to `rev'`.)

- A tail recursive definition can be transformed (either by hand or by a compiler) into an efficient loop.

- A tail recursive definition is usually more general than its backward linear recursive counterpart. Sometimes we can exploit this generality to synthesize a more efficient definition. (We see an example of this in the next subsection.)

26.7 Finding Better Tail Recursive Algorithms

This section is adapted from Cohen [34:11.3].

Although the Tail Recursion Theorem is important, the technique we used to develop it is perhaps even more important. We can sometimes use the technique to transform one tail recursive definition into another that is more efficient [98].

Consider exponentiation by a natural number power. The operation ** can be defined recursively in terms of multiplication as follows:

```
infixr 8 **
(**) :: Int -> Int -> Int
m ** 0         = 1              -- **.1
m ** n | n > 0 = m * (m ** n)   -- **.2
```

For `(**)` we substitute the following for the components of the `fun` definition of the previous subsection:

- `fun x` ← `m ** n`
- `b x` ← `n > 0` (Applied only to natural numbers.)
- `g x` ← `n - 1`

- `h x` ← `m`
- `l *** r` ← `l * r`
- `f x` ← `1`
- `l` ≺ `r` ← `l < r`
- `e` ← `1`
- `fun' y x` ← `exp a m n`

Thus, by applying the Tail Recursion Theorem, we define the function `exp` such that

```
exp a m n = a * (m ** n)
```

and, in particular:

```
exp 1 m n = m ** n
```

The resulting function `exp` is defined as follows (for `n >= 0`):

```
exp :: Int -> Int -> Int -> Int
exp a m 0 = a                -- exp.1
exp a m n = exp (a*m) m n    -- exp.2
```

In terms of time, this function is no more efficient than the original version; both require O(`n`) multiplies. (However, by exploiting the strictness of the first parameter, `exp` can be made more space efficient than `**`.)

Note that `exp` algorithm converges upon the final result in steps of one. Can we take advantage of the generality of `exp` and the arithmetic properties of exponentiation to find an algorithm that converges in larger steps?

Yes, we can by using the technique that we used to develop the Tail Recursion Theorem. In particular, let's try to synthesize an algorithm that converges logarithmically (in steps of half the distance) instead of linearly.

Speaking operationally, we are looking for a "short cut" to the result. To find this short cut, we use the "maps" that we have of the "terrain". That is, we take advantage of the properties we know about the exponentiation operator.

We thus attempt to find expressions `x` and `y` such that

```
exp x y (n/2) = exp a m n
```

where "/" represents division on integers.

For the base case where `n = 0`, this is trivial. We proceed with a calculation to discover values for `x` and `y` that make

```
exp x y (n/2) = exp a m n
```

when `n > 0` (i.e., in the inductive case). In doing this we can use the specification for `exp` (i.e.,`exp a m n = a * (m ** n)`).

```
    exp x y (n/2)
= { exp specification }
    x * (y ** (n/2))
= { Choose y = m ** 2 (Note 1) }
    x * ((m ** 2) ** (n/2))
```

Note 1: The strategy is to make choices for `x` and `y` that make

```
    x * (y ** (n/2))
```

and

```
    a * (m ** n)
```

equal. This choice for `y` is toward getting the `m ** n` term.

Because we are dealing with integer division, we need to consider two cases because of truncation.

Subcase `even n` (for `n > 0`):

```
    x * ((m ** 2) ** (n/2))
= { arithmetic properties of exponentiation, n even }
    x * (m ** n)
= { Choose x = a, toward getting a * (m ** n) }
    a * (m ** n)
= { exp specification }
    exp a m n
```

Thus, for even `n`, we derive:

```
    exp a m n = exp a (m*m) (n/2)
```

We optimize and replace `m ** 2` by `m * m`.

Subcase `odd n` (for `n > 0`): That is, `n/2 = (n-1)/2`.

```
    x * ((m ** 2) ** ((n-1)/2))
= { arithmetic properties of exponentiation }
    x * (m ** (n-1))
= { Choose x = a * m, toward getting a * (m ** n) }
    (a * m) * (m ** (n-1))
= { arithmetic properties of exponentiation }
    a * (m ** n)
```

= { exp specification }

```
exp a m n
```

Thus, for odd n, we derive:

```
exp a m n = exp (a*m) (m*m) (n/2)
```

To differentiate the logarithmic definition for exponentiation from the linear one, we rename the former to `exp'`. We have thus defined `exp'` as follows (for `n >= 0`):

```
exp' :: Int -> Int -> Int -> Int
exp' a m 0              = a                 -- exp'.1
exp' a m n
    | even n = exp' a     (m*m) (n/2)       -- exp'.2
    | odd n  = exp' (a*m) (m*m) ((n-1)/2)   -- exp'.3
```

Above we showed that `exp a m n = exp' a m n`. However, execution of `exp'` converges faster upon the result: O(`log2 n`) steps rather than O(n)‘.

Note: Multiplication and division of integers by natural number powers of 2, particularly 2^1, can be implemented on must current computers by arithmetic left and right shifts, respectively, which are faster than general multiplication and division.

26.8 What Next?

Chapter 27 applies the program synthesis techniques developed in this chapter to a larger set of examples on text processing.

No subsequent chapter depends explicitly upon the program synthesis content from these chapters. However, if practiced regularly, the techniques explored in this chapter can enhance a programmer's ability to solve problems and construct correct functional programming solutions.

26.9 Exercises

1. The following function computes the integer base 2 logarithm of a positive integer:

   ```
   lg :: Int -> Int
   lg x | x == 1 = 0
        | x > 1  = 1 + lg (x/2)
   ```

   Using the tail recursion theorem, write a definition for `lg` that is tail recursive.

2. Synthesize the recursive definition for `++` from the following specification:

   ```
   xs ++ ys  =  foldr (:) ys xs
   ```

3. Using tupling and function `fact5` from Chapter 4, synthesize an efficient function `allfacts` to generate a list of factorials for natural numbers `0` through parameter `n`, inclusive.

4. Consider the following recursive definition for natural number multiplication:

   ```
   mul :: Int -> Int -> Int
   mul m 0     = 0
   mul m (n+1) = m + mul m n
   ```

   This is an O(`n`) algorithm for computing `m * n`. Synthesize an alternative operation that is O(`log2 n`). Doubling (i.e., `n*2`) and halving (i.e., `n/2` with truncation) operations may be used but not multiplication (`*`) in general.

5. Derive a "more general" version of the Tail Recursion Theorem for functions of the shape

   ```
   func ::  X -> Y
   func x | not (b x) = f x                -          -- func.1
          | b x       = h x *** func (g x) +++ d.x -- func.2
   ```

   where functions `b`, `f`, `g`, and `h` are constrained as in the definition of `fun` in the Tail Recursion Theorem. Be sure to identify the appropriate constraints on `d`, `***`, and `+++` including the necessary properties of `***` and `+++`.

26.10 Acknowledgements

In Summer 2018, I adapted and revised this chapter and the next from Chapter 12 of my *Notes on Functional Programming with Haskell* [42].

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [15] and other functional programming sources [13,14,98,171,178]. They were also influenced by my research, study, and teaching related to program specification, verification, derivation, and semantics [[28]; [34]; [39]; [40]; [41]; [64]; [65]; [66]; [85]; [86]; [107]; vanGesteren1990].

I incorporated this work as new Chapter 26, Program Synthesis (this chapter), and new Chapter 27, Text Processing, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document

to HTML, PDF, and other forms as needed.

26.11 Terms and Concepts

TODO

27 Text Processing Example

27.1 Chapter Introduction

Chapter 26 illustrates how to synthesize function definitions from their specifications.

This chapter (27) applies these program synthesis techniques to a larger set of examples on text processing.

27.2 Text Processing Example

In this section we develop a text processing package similar to the one in Section 4.3 of the Bird and Wadler textbook [15]. The text processing package in the Haskell standard Prelude is slightly different in its treatment of newline characters.

A textual document can be viewed in many different ways. At the lowest level, we can view it as just a character string and define a type synonym as follows:

```haskell
type Text = String
```

However, for other purposes, we may want to consider the document as having more structure (i.e., view it as a sequence of words, lines, paragraphs, pages, etc). We sometimes want to convert the text from one view to another.

Consider the problem of converting a `Text` document to the corresponding sequence of lines. Suppose that in the `Text`{.haskell document, the newline characters `'\n'`{.haskell serve as *separators* of lines, not themselves part of the lines. Because each line is a sequence of characters, we define a type synonym `Line` as follows:

```haskell
type Line = String
```

We want a function `lines'` that will take a `Text` document and return the corresponding sequence of lines in the document. The function has the type signature:

```haskell
lines' :: Text -> [Line]
```

For example, the Haskell expression

```haskell
lines' "This has\nthree\nlines"
```

yields:

```haskell
["This has", "three ", "lines"]
```

Writing function `lines'` is not trivial. However, its inverse `unlines'` is quite easy. Function `unlines'` takes a list of `Line`s, inserts a newline character between each pair of adjacent lines, and returns the `Text` document resulting from the concatenation.

```
unlines' :: [Line] -> Text
```

Let's see if we can develop `lines'` from `unlines'`.

The basic computational pattern for function `unlines'` is a folding operation. Because we are dealing with the construction of a list and the list constructors are nonstrict in their right arguments, a `foldr` operation seems more appropriate than a `foldl` operation.

To use `foldr`, we need a binary operation that will append two lines with a newline character inserted between them. The following, a bit more general, operation `insert'` will do that for us. The first argument is the element that is to be inserted between the two list arguments.

```
insert' :: a -> [a] -> [a] -> [a]
insert' a xs ys = xs ++ [a] ++ ys -- insert.1
```

Informally, it is easy to see that `(insert' a)` is an associative operation but that it has no right (or left) identity element.

Given that `(insert' a)` has no identity element, there is no obvious "seed" value to use with `fold`. Thus we will need to find a different way to express `unlines'`.

If we restrict the domain of `unlines'` to non-nil lists of lines, then we can use `foldr1`, a right-folding operation defined over non-empty lists (in the Prelude). This function does not require an identity element for the operation. Function `foldr1` can be defined as follows:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Note: There is a similar function (in the Prelude), `foldl1` that takes a non-nil list and does a left-folding operation.

Thus we can now define `unlines'` as follows:

```
unlines' :: [Line] -> Text
unlines' xss = foldr1 (insert' '\n') xss
```

Given the definition of `unlines'`, we can now specify what we want `lines'` to do. It must satisfy the following specification for any *non-nil* `xss` of type `[Line]`:

```
lines' (unlines' xss) = xss
```

That is, `lines'` is the inverse of `unlines'` for all non-nil arguments.

The first step in the synthesis of `lines'` is to guess at a possible structure for the `lines'` function definition. Then we will attempt to calculate the unknown pieces of the definition.

Because `unlines'` uses a right-folding operation, it is reasonable to guess that its inverse will also use a right-folding operation. Thus we speculate that `lines'` can be defined as follows, given an appropriately defined operation `op` and "seed value" `a`.

```
lines' :: Text -> [Line]
lines' = foldr op a
```

Because of the definition of `foldr` and type signature of `lines'`, function `op` must have the type signature

```
op :: Char -> [Line] -> [Line]
```

and `a` must be the right identity of `op` and hence have type `[Line]`.

The task now is to find appropriate definitions for `op` and `a`.

From what we know about `unlines'`, `foldr1`, `lines'`, and `foldr`, we see that the following identities hold. (These can be proved, but we do not do so here.)

```
unlines' [xs]         = xs                  -- unlines.1
unlines' ([xs]++xss) =
    insert' '\n' xs (unlines' xss)          -- unlines.2

lines' []             = a                   -- lines.1
lines' ([x]++xs)      = op x (lines' xs)    -- lines.2
```

Note the names we give each of the above identities (e.g., `unlines.1`). We use these equations to justify our steps in the calculations below.

Next, let us calculate the unknown identity element `a`. The strategy is to transform `a` by use of the definition and derived properties for `unlines'` and the specification and derived properties for `lines'` until we arrive at a constant.

```
    a
= { lines.1 (right to left) }
    lines' []
= { unlines'.1 (right to left) with xs = [] }
    lines' (unlines' [[]])
= { specification of lines' (left to right) }
    [[]]
```

Therefore we define `a` to be `[[]]`. Note that because of `lines.1`, we have also defined `lines'` in the case where its argument is `[]`.

Now we proceed to calculate a definition for `op`. Remember that we assume `xss /= []`.

As above, the strategy is to use what we know about unlines' and what we have assumed about lines' to calculate appropriate definitions for the unknown parts of the definition of lines'. We first expand our expression to bring in unlines'.

```
    op x xss
= { specification for lines' (right to left) }
    op x (lines' (unlines' xss))
= { lines.2 (right to left) }
    lines' ([x] ++ unlines' xss)
```

Because there seems no other way to proceed with our calculation, we distinguish between cases for the variable x. In particular, we consider the case where x is the line separator and the case where it is not, i.e., x == '\n' and x /= '\n'.

Case x == '\n':

Our strategy is to absorb the '\n' into the unlines', then apply the specification of lines'.

```
    lines' ("\n" ++ unlines' xss)
= { [] is the identity for ++ }
    lines' ([]  ++ "\n" ++ unlines' xss)
= { insert.1 (right to left) with a == '\n' }
    lines' (insert' '\n' [] (unlines' xss))
= { unlines.2 (right to left) }
    lines' (unlines' ([[]] ++ xss))
= { specification of lines' (left to right) }
    [[]] ++ xss
```

Thus op '\n' xss = [[]] ++ xss.

Case x /= '\n':

Our strategy is to absorb the [x] into the unlines', then apply the specification of lines.

```
    lines' ([x] ++ unlines' xss)
= { Assumption xss /= [], let xss = [ys] ++ yss }
    lines' ([x] ++ unlines' ([ys] ++ yss))
= { unlines.2 (left to right) with a = '\n' }
    lines' ([x] ++ insert' '\n' ys (unlines' yss))
```

= { insert.1 (left to right) }

```
lines' ([x] ++ (ys ++ "\n" ++ unlines' yss))
```

= { ++ associativity }

```
lines' (([x] ++ ys) ++ "\n" ++ unlines' yss)
```

= { insert.1 (right to left) }

```
lines' (insert' '\n' ([x]++ys) (unlines' yss))
```

= { unlines.2 (right to left) }

```
lines' (unlines' ([[x]++ys] ++ yss))
```

= { specification of lines' (left to right) }

```
[[x]++ys] ++ yss
```

Thus, for `x /= '\n'` and `xss /= []`:

```
op x xss = [[x] ++ head xss] ++ (tail xss)
```

To generalize `op` like we did `insert'` and give it a more appropriate name, we define `op` to be `breakOn '\n'` as follows:

```
breakOn :: Eq a => a -> a -> [[a]] -> [[a]]
breakOn a x []              = error "breakOn applied to nil"
breakOn a x xss | a == x    = [[]] ++ xss
                | otherwise = [[x] ++ ys] ++ yss
                              where (ys:yss) = xss
```

Thus, we get the following definition for `lines'`:

```
lines' :: Text -> [Line]
lines' xs = foldr (breakOn '\n') [[]] xs
```

Let's review what we have done in this example. We have synthesized `lines'` from its specification and the definition for `unlines'`, its inverse. Starting from a precise, but non-executable specification, and using only equational reasoning, we have derived an executable definition of the required function.

The technique used is a familiar one in many areas of mathematics:

1. We guessed at a form for the solution.
2. We then calculated the unknowns.

Note: The definition of `lines` and `unlines` in the standard Prelude treat newlines as line *terminators* instead of line separators. Their definitions follow.

```
lines :: String -> [String]
lines "" =  []
lines s  = l : (if null s' then [] else lines (tail s'))
           where (l, s') = break ('\n'==) s
```

```
unlines :: [String] -> String
unlines  = concat . map (\l -> l ++ "\n")
```

27.2.1 Word processing

Let's continue the text processing example from the previous subsection a bit further. We want to synthesize a function to break a text into a sequence of words.

For the purposes here, we define a *word* as any nonempty sequence of characters not containing a space or newline character. That is, a group of one or more spaces and newlines separate words. We introduce a type synonym for words.

```
type Word = String
```

We want a function `words'` that breaks a line up into a sequence of words. Function `words'` thus has the following type signature:

```
words' :: Line -> [Word]
```

For example, expression

```
words' "Hi there"
```

yields:

```
["Hi", "there"]
```

As in the synthesis of `lines'`, we proceed by defining the "inverse" function first, then we calculate the definition for `words'`.

All `unwords'` needs to do is to insert a space character between adjacent elements of the sequence of words and return the concatenated result. Following the development in the previous subsection, we can thus define `unwords'` as follows.

```
unwords' :: [Word] -> Line
unwords' xs = foldr1 (insert'  ' ') xs
```

Using calculations similar to those for `lines'`, we derive the inverse of `unwords'` to be the following function:

```
foldr (breakOn' ') [[]]
```

However, this identifies zero-length words where there are adjacent spaces. We need to filter those out.

```
words' :: Line -> [Word]
words' = filter (/= []) . foldr (breakOn' ') [[]]
```

Note that

```
words' (unwords' xss)  =  xss
```

for all `xss` of type `[Word]`, but that

```haskell
unwords' (words' xs)  =   xs
```

for some `xs` of type `Line`. The latter is undefined when `words' {.haskell} xs` returns `[]`. Where it is defined, adjacent spaces in `xs` are replaced by a single space in `unwords' (words' xs)`.

Note: The functions `words` and `unwords` in the standard Prelude differ in that `unwords [] = []`, which is more complete.

27.2.2 Paragraph processing

Let's continue the text processing example one step further and synthesize a function to break a sequence of lines into paragraphs.

For the purposes here, we define a *paragraph* as any nonempty sequence of nonempty lines. That is, a group of one or more empty lines separate paragraphs. As above, we introduce an appropriate type synonym:

```haskell
type Para = [Line]
```

We want a function `paras'` that breaks a sequence of lines into a sequence of paragraphs:

```haskell
paras' :: [Line] -> [Para]
```

For example, expression

```haskell
paras' ["Line 1.1","Line 1.2","","Line 2.1"]
```

yields:

```haskell
[["Line 1.1","Line 1.2"],["Line 2.1"]]
```

As in the synthesis of `lines'` and `words'`, we can start with the inverse and calculate the definition of `paras'`. The inverse function `unparas'` takes a sequence of paragraphs and returns the corresponding sequence of lines with an empty line inserted between adjacent paragraphs.

```haskell
unparas' :: [Para] -> [Line]
unparas' = foldr1 (insert' [])
```

Using calculations similar to those for `lines'` and `words'`, we can derive the following definitions:

```haskell
paras' :: [Line] -> [Para]
paras' = filter (/= []) . foldr (breakOn []) [[]]
```

The `filter (/= [])` operation removes all "empty paragraphs" corresponding to two or more adjacent empty lines.

Note: There are no equivalents of `paras'` and '`unparas'` in the standard prelude. As with `unwords`, `unparas'` should be redefined so that `unparas' [] = []`, which is more complete.

27.2.3 Other text processing functions

Using the six functions in our text processing package, we can build other useful functions.

1. **Count the lines in a text.**

```
countLines :: Text -> Int
countLines = length . lines'
```

2. **Count the words in a text.**

```
countWords :: Text -> Int
countWords = length . concat . (map words') . lines'
```

   An alternative using a list comprehension is:

```
countWords xs =
    length [ w | l <- lines' xs, w <- words' l]
```

3. **Count the paragraphs in a text.**

```
countParas :: Text -> Int
countParas = length . paras' . lines'
```

4. **Normalize text by removing redundant empty lines and spaces.**

   The following functions take advantage of the fact that `paras'` and `words'` discard empty paragraphs and words, respectively.

```
normalize :: Text -> Text
normalize = unparse . parse

parse :: Text -> [[[Word]]]
parse = (map (map words')) . paras' . lines'

unparse :: [[[Word]]] -> Text
unparse = unlines' . unparas' . map (map unwords')
```

   We can also state `parse` and `unparse` in terms of list comprehensions.

```
parse xs =
    [ [words' l | l <- p] | p <- paras' (lines' xs) ]

unparse xssss =
    unlines' (unparas' [ [unwords' l | l<-p] | p<-xssss])
```

Section 4.3.5 of the Bird and Wadler textbook [15] goes on to build functions to fill and left-justify lines of text.

27.3 What Next?

Chapter 26 illustrates how to synthesize (i.e., derive or calculate) function definitions from their specifications. This chapter (27) applies these program synthesis techniques to larger set of examples on text processing.

No subsequent chapter depends explicitly upon the program synthesis content from these chapters. However, if practiced regularly, the techniques explored in this chapter can enhance a programmer's ability to solve problems and construct correct functional programming solutions.

27.4 Exercises

TODO

27.5 Acknowledgements

In Summer 2018, I adapted and revised this chapter and the next from Chapter 12 of my *Notes on Functional Programming with Haskell* [42].

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [15] and other functional programming sources [13,14,98,171,178]. They were also influenced by my research, study, and teaching related to program specification, verification, derivation, and semantics [[28]; [34]; [39]; [40]; [41]; [64]; [65]; [66]; [85]; [86]; [107]; vanGesteren1990].

I incorporated this work as new Chapter 26, Program Synthesis, and new Chapter 27, Text Processing (this chapter), in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

27.6 Terms and Concepts

Program synthesis, synthesizing a function from its inverse, text processing, line, word, paragraph, terminator, separator.

28 Type Inference

28.1 Chapter Introduction

The goal of this chapter (28) is to show how type inference works. It presents the topic using an equational reasoning technique.

This chapter depends upon the reader understanding Haskell polymorphic, higher-order function concepts (e.g., from studying Chapters 13-17), but it is otherwise independent of other chapters. No subsequent chapter depends explicitly upon this content.

28.2 Motivation

How can we deduce the type of a Haskell expression?

To get the general idea, let's look at a few examples.

Note: The discussion here is correct for monomorphic functions, but it is a bit simplistic for polymorphic functions. However, it should be of assistance in understanding how types are assigned to Haskell expressions.

28.3 Example: Functional Composition

Expressed in prefix form, functional composition can be defined with the equation:

```
(.) f g x = f (g x)
```

We begin the process of type inference by assigning types to the parameter names and to the function's defining expression (i.e., its result). We introduce new type names `t1`, `t2`, `t3` and `t4` for the components of `(.)` as follows:

```
          f :: t1    -- parameter 1 of (.)
          g :: t2    -- parameter 2 of (.)
          x :: t3    -- parameter 3 of (.)
    f (g x) :: t4    -- defining expression for (.)
```

The type of `(.)` is therefore given by:

```
(.) :: t1 -> t2 -> t3 -> t4
```

We are not finished because there are certain relationships among the new types that must be taken into account. To see what these relationships are, we use the following inference rules.

- **Application rule:** If `f x :: t`, then we can deduce `x :: t'` and `f :: t' -> t` for some new type `t'`.
- **Equality rule:** If both `x :: t` and `x :: t'` for some variable `x` , then we can deduce `t = t'`.

- **Function rule:** If `(t -> u) = (t' -> u')`, then we can deduce `t = t'` and `u = u'`.

Using the *application rule* on `{.haskell} f (g x) :: t4`, we introduce a new type `t5` such that:

```
g x :: t5
  f :: t5 -> t4
```

Using the *application rule* for `g x :: t5`, we introduce another new type `t6` such that:

```
x :: t6
g :: t6 -> t5
```

Using the *equality rule* on the two types deduced for each of `f`, `g`, and `x`, respectively, we get the following identities:

```
t1 = (t5 -> t4)     -- f
t2 = (t6 -> t5)     -- g
t3 = t6             -- x
```

For function `(.)`, we thus deduce the type signature:

```
(.) :: (t5 -> t4) -> (t6 -> t5) -> t6 -> t4
```

If we replace the type names by Haskell generic type variables that follow the usual naming convention, we get:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

28.4 Example: Multiple Use of Polymorphic Function (`fst`)

Now let's consider the function definition:

```
f x y = fst x + fst y
```

Note that the names `(+)` and `fst` occur on the right side of the definition, but do not occur on the left.

From the Haskell Prelude, we can see that:

```
(+) :: Num a => a -> a -> a
fst :: (a, b) -> a
```

The `Num a` context contrains the polymorphism on type variable `a`.

We must be careful. The two occurrences of the polymorphic function `fst` in the definition for `f` need not bind the type variables `a` and `b` to the same concrete types. For example, consider the expression:

```
fst (2, True) + fst (1, "hello")
```

This expression is well-typed despite the fact that the first occurrence of `fst` has the type

```
Num a => (a,Bool) -> a
```

and the second occurrence has type

```
Num a => (a, [Char]) -> a
```

Furthermore, the two occurrences of the type variable `a` are not, in general, required to bind to the same type. (However, as we will see, they do in this expression because of the addition operation.)

To handle the situation with the multiple applications of `fst`, we use the following rule.

- **Polymorphic use rule:** If a polymorphic function is applied multiple times in an expression, then the type of each occurrence is determined independently, with each assigned new type variables.

Following the *polymorphic use* rule, we rewrite the definition of `f` in the form

```
f x y = fst1 x + fst2 y
```

and assume two different instantiations of the generic type of `fst`:

```
fst1 :: (u1, u2) -> u1
fst2 :: (v1, v2) -> v1
```

After making the above transformation, we proceed by assigning types to the parameters and definition of `f`, introducing three new types:

```
              x :: t1    -- parameter 1 of f
              y :: t2    -- parameter 2 of f
fst1 x + fst2 y :: t3    -- defining expression for f
```

Thus we have the following type for `f`:

```
f :: t1 -> t2 -> t3
```

Now we can rewrite the defining expression for `f` fully in prefix form to get:

```
(+) (fst1 x) (fst2 y)
```

Then, using the *application rule* on the above expression, we deduce:

```
    (fst2 y) :: t4
(+) (fst1 x) :: t4 -> t3
```

Using the *application rule* on `(fst2 y) :: t4`, we get:

```
   y :: t5
fst2 :: t5 -> t4
```

Similarly, using the *application rule* on `(+) (fst1 x) :: t4 -> t3`, we get:

```
(fst1 x) :: t6
     (+) :: t6 -> t4 -> t3
```

Going further and applying the *application rule* to `(fst1 x) :: t6`, we deduce:

```
   x :: t7
fst1 :: t7 -> t6
```

Now we have introduced types for all the symbols appearing in the definition of function `f`. We begin simplification by using the *equality rule* for `x`, `y`, `fst1`, `fst2`, and `(+)`, respectively. We thus deduce the type equations:

```
                    t1 = t7                     -- x
                    t2 = t5                     -- y
     ((u1, u2) -> u1)  =  (t7 -> t6)           -- fst1
     ((v1, v2) -> v1)  =  (t5 -> t4)           -- fst2
(Num a => a -> a -> a) = (t6 -> t4 -> t3)      -- (+)
```

Now, using the function rule on the last three equations above, we derive:

```
t7 = (u1, u2)
t6 = u1

t5 = (v1, v2)
t4 = v1

t3 = t4 = t6 = v1 = u1 = (Num a => a)
```

We had assigned type `f :: t1 -> t2 -> t3` originally. Substituting from the above, we deduce the following type:

```
f :: Num a => (a, u2) -> (a, v2) -> a
```

Finally, we can replace the type names `u2` and `v2` by Haskell generic type variables that follow the usual naming convention. We get the following inferred type for function `f`:

```
f :: Num a => (a, b) -> (a, c) -> a
```

28.5 Example: Fixpoint (`fix`)

For this example, consider the definition:

```
fix f = f (fix f)
```

To deduce a type for `fix`, we proceed as before and introduce types for the parameters and defining expression of `f`:

```
          f :: t1     -- parameter of fix
f (fix f) :: t2     -- defining expression for fix
```

Thus, `fix` has the type:

```
fix :: t1 -> t2
```

Using the *application rule* on the expression `f (fix f)`, we obtain:

```
(fix f) :: t3
      f :: t3 -> t2
```

Then using the *application rule* on the expression `fix f`, we get:

```
    f :: t4
  fix :: t4 -> t3
```

Using the *equality rule* on `f` and `fix`, we deduce:

```
t1 = t4 = (t3 -> t2)        -- f
(t1 -> t2) = (t4 -> t3)     -- fix
```

Then, using the *function rule* on the second equation, we obtain the identities:

```
t1 = t4
t2 = t3
```

Since `fix :: t1 -> t3`, we derive the type:

```
fix :: (t3 -> t3) -> t3
```

If we replace `t3` by a Haskell generic type variable that follows the usual naming convention, we get the following inferred type for `fix`:

```
fix :: (a -> a) -> a
```

28.6 Example: Incorrect Typing (`selfapply`)

Finally, let us consider an example in which the typing is wrong. Let us define `selfapply` as follows:

```
selfapply f = f f
```

Proceeding as in the previous examples, we introduce new types for the parameters and defining expression of `f`:

```
  f :: t1    -- parameter of selfapply
f f :: t2    -- defining expression for selfapply
```

Thus we have the type:

```
selfapply :: t1 -> t2
```

Using the *application rule* on `f f`, we get:

```
f :: t3
f :: t3 -> t2
```

But the *equality rule* for `f` tells us that:

```
t1 = t3 = (t3 -> t2)
```

or just

```
t1 = (t1 -> t2)
```

However, the equation `t1 = (t1 -> t2)` does not possess a solution for `t1` and the definition of `selfapply` is thus rejected by the type checker.

28.7 Other Aspects of Type Inference

Haskell function definitions must also conform to the following rules.

- **Guard rule:** Each guard must be an expression of type `Bool`.
- **Tuple rule:** The type of a tuple of elements is the tuple of their respective types.

28.8 What Next?

This chapter is largely independent of other chapters. No subsequent chapter depends explicitly upon this content.

28.9 Exercises

TODO

28.10 Acknowledgements

In Spring 2017, I adapted and revised this chapter from my previous HTML notes on this topic. (These were supplementary notes for a course based on [42].) I based the previous notes on the presentations in:

- Section 2.8 of the book *Introduction to Functional Programming* (First Edition) by Richard Bird and Philip Wadler [15]
- Chapter 9 of the book *Haskell: The Craft of Functional Programming* (First Edition) by Simon Thompson [171]

I thank MS student Hongmei Gao for helping me prepare the first version of the previous notes in Spring 2000.

In Summer 2018, I incorporated this work as new Chapter 24, Type Inference, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I reordered the Chapters, making this Chapter 28 (instead of Chapter 24).

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

28.11 Terms and Concepts

Type inference, function, polymorphism, type variable, function composition, fixpoint, application rule, equality rule, function rule, polymorphic use rule, guard rule, tuple rule.

29 Models of Reduction

29.1 Chapter Introduction

TODO:

- Complete introduction and other missing pieces.

- Redraw LaTeX figures so they appear in formats other than LaTeX/PDF.
- Remove or explain any unnecessary redundancies between this chapter and chapters 8, 9, etc.
- Consider whether to replace the use of Haskell as pseudo-math notation.
- Check section breaks and titles.

29.2 Big-O and Efficiency

We state efficiency (i.e., time complexity or space complexity) of programs in terms of the "Big-O" notation and asymptotic analysis.

For example, consider the list-reversing functions `rev` and `reverse` that we have looked at several times. We stated that the number of steps required to evaluate `rev xs` is, in the worst case, "on the order of" `n^2` where `n` denotes the length of list `xs`. We let the number of steps be our measure of time and write

> T(`rev xs`) = O(`n^2`)

to mean that the time to evaluate `rev xs` is bounded by some (mathematical) function that is proportional to the square of the length of list `xs`.

Similarly, we write

> T(`reverse xs`) = O(`n`)

to mean that the time (i.e., number of steps) to evaluate `reverse xs` is bounded by some function that is proportional to the length of `xs`.

Note: These expressions are not really equalities. We write the more precise expression

> T(`reverse xs`)

on the left-hand side and the less precise expression O(`n`) on the right-hand side.

For short lists, the performance of `rev` and `reverse` are similar. But as the lists get long, `rev` requires considerably more steps than `reverse`.

The Big-O analysis is an asymptotic analysis. That is, it estimates the order of magnitude of the evaluation time as the size of the input approaches infinity (i.e., gets large). We often do worst case analyses of time. Such analyses are usually easier to do than average-case analyses.

29.3 Reduction

29.3.1 Definition

The terms *reduction*, *simplification*, and *evaluation* all denote the same process: rewriting an expression in a "simpler" equivalent form. That is, they involve two kinds of replacements:

- the replacement of a subterm that satisfies the left-hand side of an equation by the right-hand side with appropriate substitution of arguments for parameters. (This is sometimes called β-reduction.)
- the replacement of a primitive application (e.g., `+` or `*`) by its value. (This is sometimes called δ-reduction.)

29.3.2 Redexes

The term *redex* refers to a subterm of an expression that can be reduced.

An expression is said to be in *normal form* if it cannot be further reduced.

Some expressions cannot be reduced to a value. For example, `1/0` cannot be reduced; an error message is usually generated if there is an attempt to evaluate (i.e., reduce) such an expression.

For convenience, we sometimes assign the value $\perp$ (pronounced "bottom") to such error cases to denote that their values are undefined. Remember that this value cannot be manipulated within a computer.

Redexes can be selected for reduction in several ways. For instance, the redex can be selected based on its position within the expression:

- **leftmost redex first**—where the leftmost reducible subterm in the expression text is reduced before any other subterms are reduced
- **rightmost redex first**—where the rightmost reducible subterm in the expression text is reduced before any other subterms are reduced

The redex can also be selected based on whether or not it is contained within another redex:

- **outermost redex first**—where a reducible subterm that is not contained within any other reducible subterm is reduced before one that is contained within another
- **innermost redex first**—where a reducible subterm that contains no other reducible subterm is reduced before one that contains others

29.3.3 AOR and NOR

The two most often used reduction orders are:

- **applicative order reduction (AOR)**—where the leftmost innermost redex is reduced first
- **normal order reduction (NOR)**—where the leftmost outermost redex is reduced first.

To see the difference between AOR and NOR consider the following functions:

```
fst :: (a,b) -> a
fst (x,y) = x

sqr :: Int -> Int
sqr x = x * x
```

Now consider the following reductions.

First, reduce the expression with **AOR**:

```
     fst (sqr 4, sqr 2)
⟹ { sqr }
     fst (4*4, sqr 2)
⟹ { * }
     fst (16, sqr 2)
⟹ { sqr }
     fst (16, 2*2)
⟹ { * }
     fst (16, 4)
⟹ { fst }
     16
```

Thus AOR requires 5 reductions.

Second, reduce the expression with **NOR**:

```
     fst (sqr 4, sqr 2)
⟹ { fst }
     sqr 4
⟹ { sqr }
     4*4
⟹ { * }
     16
```

Thus NOR requires 3 reductions.

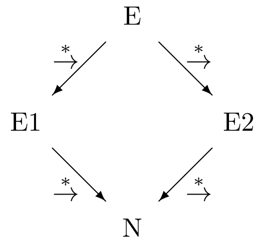
In this example NOR requires fewer steps because it avoids reducing the unneeded second component of the tuple.

The number of reductions is different, but the result is the same for both reduction sequences.

In fact, this is always the case. If any reduction terminates (and not all do), then the resulting value will always be the same.

(Consequence of) Church-Rosser Theorem: If an expression can be reduced in two different ways to two normal forms, then these normal forms are the same (except that variables may need to be renamed).

The *diamond property* for the reduction relation $\rightarrow$ states that if an expression `E` can be reduced to two expressions `E1` and `E2`, then there is an expression `N` which can be reached (by repeatedly applying $\rightarrow$) from both `E1` and `E2`. We use the symbol $\stackrel{*}{\rightarrow}$ to represent the *reflexive transitive closure* of $\rightarrow$. (`E` $\stackrel{*}{\rightarrow}$ `E1` means that `E` can be reduced to `E1` by some finite, possibly zero, number of reductions.)



Some reduction orders may fail to terminate on some expressions. Consider the following functions:

```
answer :: Int -> Int
answer n = fst (n+n, loop n)

loop :: Int -> [a\]
loop n = loop (n+1)
```

First, reduce the expression with **AOR**:

```
 answer 1
```

$\Longrightarrow$ { answer }

```
 fst (1+1,loop 1)
```

$\Longrightarrow$ { + }

```
 fst (2,loop 1)
```

$\Longrightarrow$ { `loop` }

```
fst (2,loop (1+1))
```

$\Longrightarrow$ { + }

```
fst (2,loop 2)
```

$\Longrightarrow$ { `loop` }

```
fst (2,loop (2+1))
```

$\Longrightarrow$ { + }

```
fst (2,loop 3)
```

$\Longrightarrow$ $\cdots$ Does not terminate normally

Second, reduce the expression with **NOR**:

```
answer 1
```

$\Longrightarrow$ { `answer` }

```
fst (1+1,loop 1)
```

$\Longrightarrow$ { `fst` }

```
1+1
```

$\Longrightarrow$ { + }

```
2
```

Thus NOR requires 3 reductions.

If an expression `E` has a normal form, then a normal order reduction of `E` (i.e., leftmost outermost) is guaranteed to reach the normal form (except that variables may need to be renamed).

29.3.4 Concepts related to AOR and NOR

There are several concepts in functional programming languages related to AOR:

- ***Applicative order reduction (AOR)*** Reduce leftmost innermost redex first.
- ***Eager evaluation*** Evaluate any expression that can be evaluated regardless of whether the result is ever needed. (For example, arguments of a function are evaluated before the function is called.)
- ***Strict semantics*** A function is only defined if all of its arguments are defined. For example, multiplication is only defined if both of its operands are defined, `5 * \bot = \bot`.

- ***Call-by-value parameter passing*** Evaluate the argument expression and bind its value to the function's parameter.

Similarly, there are several concepts in functional programming languages related to NOR:

- ***Normal order reduction (NOR)*** Reduce leftmost outermost redex first.
- ***Lazy evaluation*** Do not evaluate an expression unless its result is needed.
- ***Nonstrict (lenient) semantics*** A function may have a value even if some of its arguments are undefined. (For example, tuple construction is not strict in either parameter. That is, `(⊥,x)` $\neq \bot$ and `(x,⊥)` $\neq \bot$.)
- ***Call-by-name parameter passing*** Pass the unevaluated argument expression to the function; evaluate it upon each reference.

Note that in the absence of side-effects (e.g., when we have referential transparency, call-by-name gives the same result as call-by-value.

In general, call-by-name parameter passing is inefficient. However, a referentially transparent language can replace call-by-name parameter passing with the equivalent, but more efficient, *call-by-need* method.

In the call-by-need method, the unevaluated argument expression is passed to the function as in call-by-name. The first reference to the corresponding parameter causes the expression to be evaluated; subsequent references just use the value computed by the first reference. Thus the expression is only evaluated when needed and then only once.

Consider the `sqr` program again.

```
sqr x = x \* x
```

First, reduce the expression with **AOR**:

```
 sqr (4+2)
⟹ { + }
 sqr 6
⟹ { sqr }
 6 * 6
⟹ { * }

 36
```

Thus AOR requires 3 reductions.

Second, reduce the expression with **NOR**:

```
     sqr (4+2)
⟹ { sqr }
     (4+2) * (4+2)
⟹ { + }
     6 * (4+2)
⟹ { + }
     6 * 6
⟹ { * }
36
```

Thus NOR requires 4 reductions.

Here NOR is less efficient than AOR. What is the problem?

The argument `(4+2)` is reduced twice because the parameter appeared twice on the right-hand side of the definition.

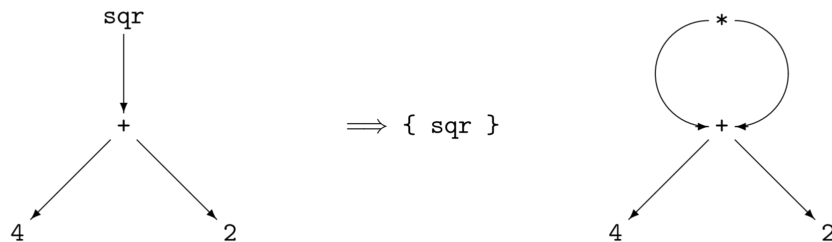
29.3.5 String and graph reduction

The rewriting strategy we have been using so far can be called *string reduction* because our model involves the textual replacement of one string by an equivalent string.

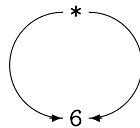
A more efficient alternative is *graph reduction.* In this technique, the expressions are represented as (directed acyclic) expression graphs rather than text strings. The repeated subterms of an expression are represented as shared components of the expression graph. Once a shared component has been evaluated, it need not be evaluated again. Thus leftmost outermost (i.e., normal order) graph reduction is a technique for implementing call-by-need parameter passing.

The Haskell interpreter uses a graph reduction technique.

Consider the leftmost outermost graph reduction of the expression `sqr (4+2)`.



$\Longrightarrow$ { + }



36

$\Longrightarrow$ { * }

Note: In a graph reduction model, normal order reduction never performs more reduction steps than applicative order reduction. It may perform fewer. And, like all outermost reduction techniques, it is guaranteed to terminate if any reduction sequence terminates.

As we see above, parameters that repeatedly occur on the right-hand side introduce shared components into the expression graph. A programmer can also introduce shared components into a function's expression graph by using **`where`** or **`let`** to define new symbols for subexpressions that occur multiple times in the defining expression. This potentially increases the efficiency of the program .

Consider a program to find the solutions of the following equation:

$$a * x^2 + b * x + c = 0$$

Using the quadratic formula the two solutions are:

$$\frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

Expressing this formula as a Haskell program to return the two solutions as a pair, we get:

```
roots :: Float -> Float -> Float -> (Float,Float)
roots a b c = ( (-b-d)/e, (-b+d)/e )
    where d = sqrt (sqr b - 4 * a * c)
          e = 2 * a
```

Note the explicit definition of local symbols for the subexpressions that occur multiple times.

Function `sqr` is as defined previously and `sqrt` is a primitive function defined in the standard prelude.

In one step, the expression `roots 1 5 3` reduces to the expression graph shown on the following page. For clarity, we use the following in the graph:

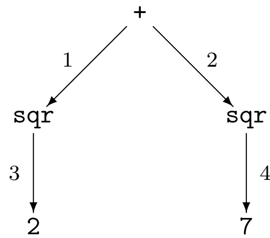
- `tuple-2` denotes the pair forming operator `( , )`.
- `div` denotes division (on `Float`).
- `sub` denotes subtraction.
- `neg` denotes unary negation.

The application `roots 1 5 3` reduces to the following expression graph:

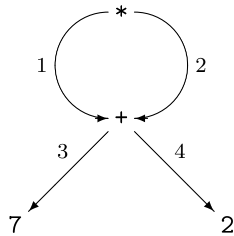
(Drawing Not Currently Available)

We use the total number of *arguments* as the measure of the *size* of a term or graph.

Example: `sqr 2 + sqr 7` has size 4.



Example: `x * x where x = 7 + 2` has size 4.



Note: This size measure is an indication of the size of the unevaluated expression that is held at a particular point in the evaluation process. This is a bit different

from the way we normally think of space complexity in an imperative algorithms class, that is, the number of "words" required to store the program's data.

However, this is not as strange as it may first appear. Remember that data structures such as lists and tuples are themselves *expressions* built by applying constructors to simpler data.

29.4 Head Normal Form

Sometimes we need to reduce a term but not all the way to normal form.

Consider the expression `head (map sqr [1..7])` and a normal order reduction.

```
    head (map sqr [1..7])
⟹ { [1..7] }
    head (map sqr (1:[2..7]))
⟹ { map.2 }
    head (sqr 1 : map sqr [2..7])
⟹ { head }
    sqr 1
⟹ { sqr }
    1 * 1
⟹ { * }
    1
```

Note that the expression `map sqr [1..7]` was reduced but not all the way to normal form. However, any term that is reduced must be reduced to *head normal form*.

A term is in *head normal form* if:

- it is not a redex
- it cannot become a redex by reducing any of its subterms

If a term is in normal form, then it is in head normal form, but not vice versa.

Any term of form `(e1:e2)` is in head normal form, because regardless of how far `e1` and `e2` are reduced, no reduction rule applies to `(e1:e2)`. The cons operator is the primitive list constructor; it is not defined in terms of anything else.

However, a term of form `(e1:e2)` is only in normal form if both `e1` and `e2` are in their normal forms.

Similarly, any term of the form `(e1,e2)` is in head normal form. The tuple constructor is a primitive operation; it is not defined in terms of anything else.

However, a term of the form `(e1,e2)` is in normal form only if both `e1` and `e2` are.

Whether a term needs to be reduced further than head normal form depends upon the context.

Example: In the reduction of the expression `head (map sqr [1..7])`, the term `map sqr [1..7]` only needed to be reduced to head normal form, that is, to the expression `sqr 1 : map sqr [2..7]`.

However, `appendChan stdout (show (map sqr [1..7])) exit done` would cause reduction of `map sqr [1..7]` to normal form.

29.5 Pattern Matching

For reduction using equations that involve pattern matching, the leftmost outermost (i.e., normal order) reduction strategy is not, by itself, sufficient to guarantee that a terminating reduction sequence will be found if one exists.

Consider function `zip'`.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' (a:as) (b:bs) = (a,b) : zip' as bs
zip'  _      _     = []
```

Now consider a leftmost outermost (i.e., normal order) reduction of the expression `zip' (map sqr []) (loop 0)`, where `sqr` and `loop` are as defined previously.

```
     zip' (map sqr []) (loop 0)
⟹ { map.1, to determine if first arg matches (a:as) }
     zip' [] (loop 0)
⟹ { zip'.2 }
     []
```

Alternatively, consider a rightmost outermost reduction of the same expression.

```
     zip' (map sqr []) (loop 0)
⟹ { loop, to determine if second arg matches (b:bs) }
     zip' (map sqr []) (loop (0+1))
⟹ { + }
     zip' (map sqr []) (loop 1)
⟹ { loop }
     zip' (map sqr []) (loop (1+1))
⟹ { + }
```

```
zip’ (map sqr []) (loop 2)
```

$\Longrightarrow \cdots$ Does not terminate normally

Pattern matching should not cause an argument to be reduced unless absolutely necessary; otherwise nontermination could result.

Pattern-matching reduction rule: Match the patterns left to right. Reduce a subterm only if required by the pattern.

In `zip’ (map sqr []) (loop 0)` the first argument must be reduced to head normal form to determine whether it matches `(a:as)` for the first leg of the definition. It is not necessary to reduce the second argument unless the first argument match is successful.

Note that the second leg of the definition, which uses two anonymous variables for the patterns, does not require any further reduction to occur in order to match the patterns.

The expressions

```
zip' (map sqr [1,2,3]) (map sqr [1,2,3])
```

and

```
zip' (map sqr [1,2,\]) []
```

both require their second arguments to be reduced to head normal form in order to determine whether the arguments match `(b:bs)`.

Note that the first does match and, hence, enables the first leg of the definition to be used in the reduction. The second expression does not match and, hence, disables the first leg from being used. Since the second leg involves anonymous patterns, it can be used in this case.

- Normal order graph reduction `e_{0}` $\Longrightarrow$ `e_{1}` $\Longrightarrow$ `e_{2}` $\Longrightarrow \cdots \Longrightarrow$ `e_{n}`
- Time = number of reduction steps (n)
- Space = size of the largest expression graph `e_{i}`

Most lazy functional language implementations more-or-less correspond to graph reduction.

29.6 Reduction Order and Space

It is always the case that the number of steps in an *outermost graph reduction* $\leq$ the number of steps in an *innermost reduction* of the same expression.

However, sometimes a combination of innermost and outermost reductions can save on space and, hence, on implementation overhead.

Consider the following definition of the factorial function. (This was called `fact3` in Chapter 4.)

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Now consider a normal order reduction of the expression `fact 3`.

```
    fact 3
==> { fact.2 }
    3 * fact (3-1)
==> { -, to determine pattern match }
    3 * fact 2
==> { fact.2 }
    3 * (2 * fact (2-1))
==> { -, to determine pattern match }
    3 * (2 * fact 1)
==> { fact.2 }
    3 * (2 * (1 * fact (1-1))) MAX SPACE!
==> { -, to determine pattern match }
    3 * (2 * (1 * fact 0))
==> { fact.1 }
    3 * (2 * (1 * 1))
==> { * }
    3 * (2 * 1)
==> { * }
    3 * 2
==> { * }
    6
```

We define the following measures of the

- **Time:** Count reduction steps. 10 for this example.

  In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3n+1` reductions. O(`n`).

- **Space:** Count arguments in longest expression. 4 binary operations, 1 unary operation, hence size is 9 for this example.

  In general, 1 multiplication for each `n > 01` plus 1 subtraction and one application of `fact`. Thus `2n + 3` arguments. O(`n`).

Note that function `fact` is strict in its argument. That is, evaluation of `fact` *always* requires the evaluation of its argument.

Since the value of the argument expression `n-1` in the recursive call is eventually needed (by the pattern match), there is no reason to delay evaluation of the expression. That is, the expression could be evaluated eagerly instead of lazily. Thus any work to save this expression for future evaluation would be avoided.

Delaying the computation of an expression incurs overhead in the implementation. The delayed expression and its calling environment (i.e., the values of variables) must be packaged so that evaluation can resume correctly when needed. This packaging—called a *closure*, *suspension*, or *recipe*—requires both space and time to be set up.

Furthermore, delayed expressions can aggravate the problem of *space leaks*.

The implementation of a lazy functional programming language typically allocates space for data dynamically from a memory *heap*. When the heap is exhausted, the implementation searches through its structures to recover space that is no longer in use. This process is usually called *garbage collection*.

However, sometimes it is very difficult for a garbage collector to determine whether or not a particular data structure is still needed. The garbage collector thus retains some unneeded data. These are called space leaks.

Aside: Picture bits of memory oozing out of the program, lost to the program forever. Most of these bits collect in the bit bucket under the computer and are automatically recycled when the interpreter restarts. However, in the past a few of these bits leaked out into the air, gradually polluting the atmosphere of functional programming research centers. Although it has not be scientifically verified, anecdotal evidence suggests that the bits leaked from functional programs, when exposed to open minds, metamorphose into a powerful intellectual stimulant. Many imperative programmers have observed that programmers who spend a few weeks in the vicinity of functional programs seem to develop a permanent distaste for imperative programs and a strange enhancement of their mental capacities.

Aside continued: As environmental awareness has grown in the functional programming community, the implementors of functional languages have begun to develop new leak-avoiding designs for the language processors and garbage collectors. Now the amount of space leakage has been reduced considerably. Although it is still a problem. Of course, in the meantime a large community of programmers have become addicted to the intellectual stimulation of functional programming. The number of addicts in the USA is small, but growing. FP

traffickers have found a number of ways to smuggle their illicit materials into the country. Some are brought in via the Internet from clandestine archives in Europe; a number of professors and students are believed to be cultivating a domestic supply. Some are smuggled from Europe inside strange red-and-white covered books (but that source is somewhat lacking in the continuity of supply). Some are believed hidden in Haskell holes; others in a young nerd named Haskell's pocket protector. (Haskell is Miranda's younger brother; she was the first one who had any comprehension about FP.)

Aside ends: Mercifully.

Now let's look at a tail recursive definition of factorial.

```
fact' :: Int -> Int -> Int
fact' f 0 = f
fact' f n = fact' (f*n) (n-1)
```

Because of the Tail Recursion Theorem, we know that `fact' 1 n = fact n` for any natural `n`.

Now consider a normal order reduction of the expression `fact' 1 3`.

```
    fact' 1 3
==> { fact'.2 }
    fact' (1 * 3) (3 - 1)
==> { -, to determine pattern match }
    fact' (1 * 3) 2
==> { fact'.2 }
    fact' ((1 * 3) * 2) (2 - 1)
==> { -, to determine pattern match }
    fact' ((1 * 3) * 2) 1
==> { fact'.2 }
    fact' (((1 * 3) * 2) * 1) (1 - 1) MAX SPACE!
==> { -, to determine pattern match }
    fact' (((1 * 3) * 2) * 1) 0
==> { fact'.1 }
    ((1 * 3) * 2) * 1
==> { * }
    (3 * 2) * 1
==> { * }
```

```
    6 * 1
⟹ { 6 }
    6
```

- **Time:** Count reduction steps. 10 for this example, same as for `fact`.

  In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3*n+1` reductions. O(`n`).

- **Space:** Count arguments in longest expression. 4 binary operations, 1 two-argument function, hence size is 10 for this example.

  In general, 1 multiplication for each `n > 0` plus 1 subtraction and one application of `fact'`. Thus `2*n+4` arguments. O(`n`).

Note that function `fact’` is strict in both arguments. The second argument of `fact’` is evaluated immediately because of the pattern matching. The first argument’s value is eventually needed, but its evaluation is deferred until after the `fact’` recursion has reached its base case.

Perhaps we can improve the space efficiency by forcing the evaluation of the first argument immediately as well. In particular, we try a combination of outermost and innermost reduction.

```
    fact’ 1 3
⟹ { fact’.2 }
    fact’ (1 * 3) (3 - 1)
⟹ { *, innermost }
    fact’ 3 (3 - 1)
⟹ { -, to determine pattern match }
    fact’ 3 2
⟹ { fact’.2 }
    fact’ (3 * 2) (2 - 1)
⟹ { *, innermost }
    fact’ 6 (2 - 1)
⟹ { -, to determine pattern match }
    fact’ 6 1
⟹ { fact’.2 }
    fact’ (6 * 1) (1 - 1)
⟹ { *, innermost }
    fact’ 6 (1 - 1)
```

⟹ { -, to determine pattern match }

```
fact' 6 0
```

⟹ { fact'.1 }

```
6
```

- **Time:** Count reduction steps. 10 for this example. Same as for previous two reduction sequences.

  In general, 3 for each `n > 0`, 1 for `n = 0`. Thus `3*n+1` reductions. O(`n`).

- **Space:** Count arguments in longest expression.

  For any `n > 0`, the longest expression consists of one multiplication, one subtraction, and one call of `fact'`. Thus the size is constantly 6. O(1).

How to decrease space usage and implementation overhead.

1. The compiler could do *strictness analysis* and automatically force eager evaluation of arguments that are always required.

   This is done by many compilers. It is sometimes a complicated procedure.

2. The language could be extended with a feature that allows the programmer to express strictness explicitly.

In Haskell, reduction order can be controlled by use of the special function `strict`.

A term of the form `strict f e` is reduced by first reducing expression `e` to head normal form, and then applying function `f` to the result. The term `e` can be reduced by normal order reduction, unless, of course, it contains another call of `strict`.

The following definition of `fact'` gives the mixed reduction order given in the previous example. That is, it evaluates the first argument eagerly to save space.

```
fact' :: Int -> Int -> Int
fact' f 0 = f
fact' f n = (strict fact' (f*n)) (n-1)
```

29.7 Choosing a Fold

Remember that earlier we defined two folding operations. Function `foldr` is a backward linear recursive function that folds an operation through a list from the tail (i.e., right) toward the head. Function `foldl` is a tail recursive function that folds an operation through a list from the head (i.e., left) toward the tail.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

The first duality theorem (as given in the Bird and Wadler textbook [15]) states the circumstances in which one can replace `foldr` by `foldl` and vice versa.

If $\oplus$ is a associative binary operation of type `t -> t` with identity element `z`, then:

First duality theorem: If $\oplus$ is a associative binary operation of type `t -> t` with identity element `z`, then:

```
foldr (⊕) z xs = foldl (⊕) z xs
```

Thus, often we can use either `foldr` or `foldl` to solve a problem. Which is better?

We discussed this problem before, but now we have the background to understand it a bit better.

Clearly, eager evaluation of the second argument of `foldl`, which is used as an accumulating parameter, can increase the space efficiency of the folding operation. This optimized operation is called `foldl’` in the standard prelude.

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z []     = z
foldl' f z (x:xs) = strict (foldl' f) (f z x) xs
```

Suppose that `op` is *strict in both arguments* and can be computed in O(1) time and O(1) space. (For example, + and $*$ have these characteristics.) If `n = length xs`, then both `foldr op i xs` and `foldl op i xs` can be computed in O(`n`) time and O(`n`) space.

However, `foldl’ op i xs`) requires O(`n`) time and O(1) space. The reasoning for this is similar to that given for `fact’`.

Thus, in general, `foldl’` is the better choice for this case.

Alternatively, suppose that `op` is *nonstrict in either argument*. Then `foldr` is usually more efficient than `foldl`.

As an example, consider operation `||` (i.e., logical-or). The `||` operator is strict in the first argument, but not in the second. That is, `True || x = True` without having to evaluate `x`.

Let `xs = [x_1, x_2, x_3, ... x_n]` such that ($\exists i : 1 \leq i \leq n ::$ `x_i == True`) $\wedge$ ($\forall j : 1 \leq < i ::$ `x_i == False`))

Suppose `x_i` is the minimum i satisfying the above existential.

```
foldr (||) False xs
```

$\Longrightarrow$ { many steps }

```
x_1 || (x_2 || ( ... || (x_i || ( ... || (x_n || False) ... )
```

Because of the nonstrict definition of `||`, the above can stop after the `x_i` term is processed. None of the list to the right of `x_i` needs to be evaluated.

However, a version which uses `foldl` must process the entire list.

```
foldl (||) False xs
```

$\Longrightarrow$ { many steps }

```
( ... ( False || x_i) || x_2) || ... ) || x_i) || ... ) || x_n
```

In this example, `foldr` is clearly more efficient than `foldl`.

29.8 What Next?

TODO

29.9 Exercises

TODO

29.10 Acknowledgements

TODO History of chapter in FP class.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I adapted and revised this chapter from Chapter 13 of my *Notes on Functional Programming with Haskell* [42]. I had included some some of this discussion in Chapter 8 in 2016 and later.

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [15:6.1–6.3], [73:6], and other functional programming sources.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

29.11 Terms and Concepts

TODO

30 Infinite Data Structures

30.1 Chapter Introduction

One particular benefit of *lazy evaluation* is that functions in Haskell can manipulate "infinite" data structures. Of course, a program cannot actually generate or store all of an infinite object, but lazy evaluation will allow the object to be built piece-by-piece as needed and the storage occupied by no-longer-needed pieces to be reclaimed.

This chapter explores Haskell programming techniques for infinite data structures such as lists.

TODO: Write Introduction, including goals of chapter.

TODO: - Complete chapter. Improve the writing. - Update and expand discussion of infinite computations. - Recreate the missing Haskell source code files for this chapter. Ensure it works for Haskell 2010.

30.2 Infinite Lists

Reference: This section is based, in part, on discussions in the classic Bird and Wadler textbook [15:7.1] and Wentworth's tutorial [178].

In Chapter 18 , we looked at generators for infinite arithmetic sequences such as `[1..]` and `[1,3..]`. These infinite lists are encoded in the functions that generate the sequences. The sequences are only evaluated as far as needed.

For example, `take 5 [1..]` yields:

```
[1,2,3,4,5]
```

Haskell also allows infinite lists of infinite lists to be expressed as shown in the following example which generates a table of the multiples of the positive integers.

```
multiples :: [[Int]]
multiples = [ [ m*n | m<-[1..]] | n <- [1..] ]
```

Thus `multiples` represents an infinite list, as shown below (not valid Haskell code):

```
[ [1, 2, 3, 4, 5, ... ],
  [2, 4, 6, 8,10, ... ],
  [3, 6, 9,12,14, ... ],
  [4, 8,12,16,20, ... ],
  ...
]
```

However, if we evaluate the expression

```
take 4 (multiples !! 3)
```

we get the terminating result:

```
[4,8,12,16]
```

Note: Remember that the operator `xs !! n` returns element `n` of the list `xs` (where the head is element `0`).

Haskell's infinite lists are not the same as *infinite sets* or *infinite sequences* in mathematics. *Infinite lists* in Haskell correspond to *infinite computations* whereas infinite sets in mathematics are simply definitions.

In mathematics, set $\{x^2 \mid x \in \{1,2,3\} \wedge x^2 < 10\} = \{1,4,9\}$.

However, in Haskell, the expression

```
show [ x * x | x <- [1..], x * x < 10 ]
```

yields:

```
[1,4,9
```

This is a computation that never returns a result. Often, we assign this computation the value `1:4:9:`⊥ (where ⊥, pronounced "bottom" represents an undefined expression).

But the expression

```
takeWhile (<10) [ x * x | x <- [1..] ]
```

yields:

```
[1,4,9]
```

30.3 Iterate

Reference: This section is based in part on a discussion in the classic Bird and Wadler textbook [15:7.2].

In mathematics, the notation f^n denotes the function f composed with itself n times. Thus, $f^0 = id$, $f^1 = f$, $f^2 = f.f$, $f^3 = f.f.f$, $\cdots$.

A useful function is the function `iterate` such that (not valid Haskell code):

```
iterate f x = [x, f x, f^2 x, f^3 x, ... x ]
```

The Haskell standard Prelude defines `iterate` recursively as follows:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

For example, suppose we need the set of all powers of the integers.

We can define a function `powertables` would expand as follows (not valid Haskell code):

```
[ [1, 2, 4,  8, ...
  [1, 3, 9, 27, ...
  [1, 4,16, 64, ...
  [1, 5,25,125, ...
  ...
]
```

Using `iterate` we can define `powertables` compactly as follows:

```
powertables :: [[Int]]
powertables = [ iterate (*n) 1 | n <- [2..]]
```

As another example, suppose we want a function to extract the decimal digits of a positive integer. We can define `digits` as follows:

```
digits :: Int -> [Int]
digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (/10)
```

Let's consider how `digits 178` evaluates (not actual reduction steps).

```
digits 178
```

$\Longrightarrow$

```
reverse . map (mod10) . takeWhile (/= 0) [178,17,1,0,0,
...]
```

$\Longrightarrow$

```
reverse . map (mod10) [178,17,1]
```

$\Longrightarrow$

```
reverse [8,7,1]
```

$\Longrightarrow$

```
[1,7,8]
```

30.4 Prime Numbers: Sieve of Eratosthenes

Reference: This is based in part on discussions in the classic Bird and Wadler textbook [15:7.3] and Wentworth's tutorial [178, Ch. 9].

The Greek mathematician Eratosthenes described essentially the following procedure for generating the list of all *prime numbers*. This algorithm is called the *Sieve of Eratosthenes*.

1. Generate the list 2, 3, 4, $\cdots$
2. Mark the first element p as prime.
3. Delete all multiples of p from the list.
4. Return to step 2.

Not only is the 2-3-4 loop infinite, but so are steps 1 and 3 themselves.

There is a straightforward translation of this algorithm to Haskell.

```
primes :: [Int]
primes = map head (iterate sieve [2..])

sieve (p:xs) = [x | x <- xs, x `mod` p /= 0 ]
```

Note: This uses an intermediate infinite list of infinite lists; even though it is evaluated lazily, it is still inefficient.

We can use function `primes` in various ways, e.g., to find the first 1000 primes or to find all the primes that are less than 10,000.

```
take 1000 primes
takeWhile (<10000) primes
```

Calculations such as these are not trivial if the computation is attempted using arrays in an "eager" language like Pascal—in particular it is difficult to know beforehand how large an array to declare for the lists.

However, by *separating the concerns*, that is, by keeping the computation of the primes separate from the application of the boundary conditions, the program becomes quite modular. The same basic computation can support different boundary conditions in different contexts.

Now let's transform the `primes` and `sieve` definitions to eliminate the infinite list of infinite lists. First, let's separate the generation of the infinite list of positive integers from the application of `sieve`.

```
primes        = rsieve [2..]

rsieve (p:ps) = map head (iterate sieve (p:ps))
```

Next, let's try to transform `rsieve` into a more efficient definition.

```
    rsieve (p:ps)
= { rsieve }
    map head (iterate sieve (p:ps))
= { iterate }
    map head ((p:ps) : (iterate sieve (sieve (p:ps)) ))
= { map.2, head }
    p : map head (iterate sieve (sieve (p:ps)) )
= { sieve }
    p : map head (iterate sieve [x | x <- ps, x `mod` p /= 0 ])
= { rsieve }
```

```
p : rsieve [x | x <- ps, x `mod` p /= 0 ]
```

This calculation gives us the new definition:

```
rsieve (p:ps) = p : rsieve [x | x <- ps, x `mod` p /= 0 ]
```

This new definition is, of course, equivalent to the original one, but it is slightly more efficient in that it does not use an infinite list of infinite lists.

30.5 Circular Structures

Reference: This section is based, in part, on discussions in classic Bird and Wadler textbook [15:7.6] and of Wentworth's tutorial [178, Ch. 9].

Suppose a program produces a data structure (e.g., a list) as its output. And further suppose the program feeds that output structure back into the input so that later elements in the structure depend on earlier elements. These might be called *circular*, *cyclic*, or *self-referential* structures.

Consider a list consisting of the integer one repeated infinitely:

```
ones = 1:ones
```

As an expression graph, `ones` consists of a cons operator with two children, the integer 1 on the left and a recursive reference to `ones` (i.e., a self loop) on the right. Thus the infinite list `ones` is represented in a finite amount of space.

Function `numsFrom` below is a perhaps more useful function. It generates a list of successive integers beginning with `n`:

```
numsFrom :: Int -> [Int]
numsFrom n = n : numsFrom (n+1)
```

Using `numsFrom` we can construct an infinite list of the natural number multiples of an integer `m`:

```
multiples :: Int -> [Int]
multiples m  = map ((*) m) (numsFrom 0)
```

Of course, we cannot actually process all the members of one of these infinite lists. If we want a terminating program, we can only process some finite initial segment of the list. For example, we might want all of the multiples of 3 that are at most 2000:

```
takeWhile ((>=) 2000) (multiples 3)
```

We can also define a program to generate a list of the Fibonacci numbers in a circular fashion similar to `ones`:

```
fibs :: [Int]
fibs = 0 : 1 : (zipWith (+) fibs (tail fibs))
```

Proofs involving infinite lists are beyond the current scope of this textbook. See the Bird and Wadler textbook for more information [15].

TODO: Finish Chapter

30.6 What Next?

TODO

30.7 Chapter Source Code

TODO

30.8 Exercises

TODO

30.9 Acknowledgements

In Summer 2018, I adapted and revised this chapter from chapter 15 of my *Notes on Functional Programming with Haskell* [42].

These previous notes drew on the presentations in the 1st edition of the Bird and Wadler textbook [15], Wentworth's tutorial [178], and other functional programming sources.

I incorporated this work as new Chapter 30, Infinite Data Structures, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

30.10 Terms and Concepts

Infinite data structures, lazy evaluation, infinite sets, infinite sequences, infinite lists, infinite computations, bottom $\perp$, `iterate`, prime numbers, Sieve of Eratosthenes, separation of concerns, circular/cyclic/self-referential structures.

31 Future Chapter

32 Future Chapter

33 Future Chapter

34 Future Chapter

35 Future Chapter

36 Future Chapter

37 Future Chapter

38 Future Chapter

39 Future Chapter

40 Language Processing

40.1 Chapter Introduction

This is a stub for a future chapter. Only a figure exists so far.

40.2 Compiler Phases

See Figure 40.2

40.3 What Next?

TODO

40.4 Chapter Source Code

TODO if applicable

40.5 Exercises

ODO

40.6 Acknowledgements

TODO

40.7 References

TODO

40.8 Terms and Concepts

TODO

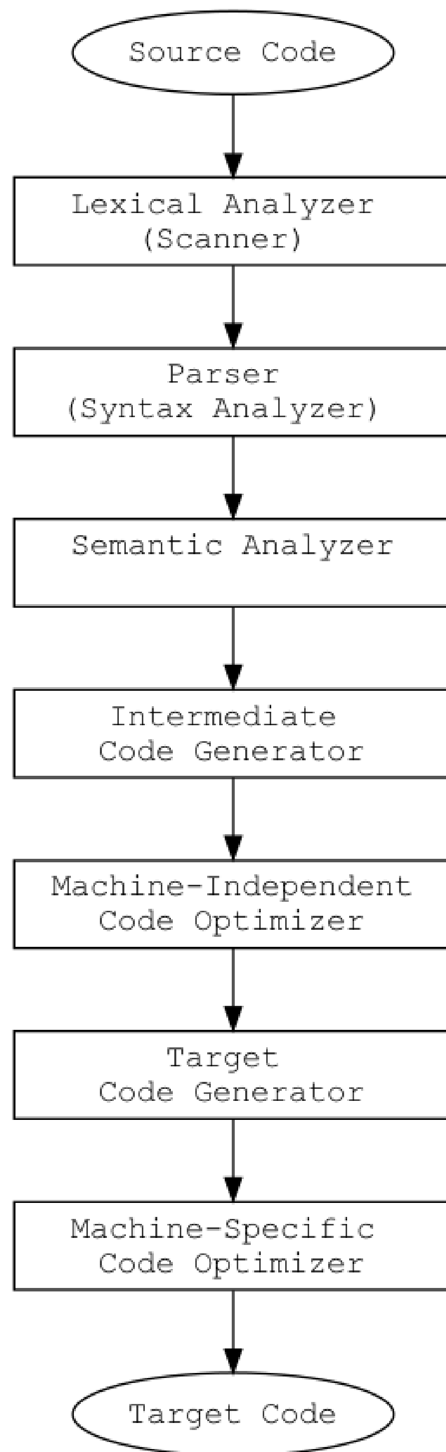


Figure 40.1: Phases of compilation.

41 Calculator: Concrete Syntax

41.1 Chapter Introduction

Chapter 40 surveyed the overall language processing pipeline.

Beginning with this chapter, we explore language concepts processing techniques in the context of a simple case study. The case study uses a language of simple arithmetic expressions, a language we call the *ELI (Exploring Languages with Interpreters) Calculator language*.

- Chapter 41 introduces the formal concepts related to concrete syntax. It gives two different concrete syntaxes for the ELI Calculator language.
- Chapter 42 introduces the concepts of abstract syntax and language semantics. It represents both concrete syntaxes of the ELI Calculator language with the same abstract syntax encoded as a Haskell algebraic data type. It defines the semantics of the language using a Haskell function that evaluates (i.e., interprets) the abstract syntax expressions.
- Chapter 43 surveys the modular design and implementation of the ELI Calculator language application.
- Chapter 44 considers lexical analysis and parsing of the concrete syntaxes to generate the corresponding abstract syntax trees
- Chapter 45 explores the construction of a set of parsing combinators.
- Chapter 46 looks at a simple Stack Virtual Machine with an instruction set represented as another algebraic data type and how to translate (i.e., compile), how to execute the machine, and how to translate the abstract syntax trees to sequences of instructions.

We will extend the language with other features in later chapters.

TODO: Give chapter's goals explicitly.

The goals of this chapter are to:

- TODO

41.2 Concrete Syntax

The ELI Calculator language can be represented as human-readable text strings in forms similar to traditional mathematical and programming notations. The structure of these textual expressions is called the *concrete syntax* [193] of the expressions.

In this case study, we examine two possible concrete syntaxes: a familiar infix syntax and a (probably less familiar) parenthesized prefix syntax.

But, first, let's consider how we can describe the syntax of a language.

41.3 Grammars

We usually describe the syntax of a language using a *formal grammar* [118,184].

Formally, a formal grammar consists of a tuple (V, T, S, P), where:

- V is a finite set of *variable* (or *nonterminal*) symbols
- T is a finite set of *terminal* symbols (called the *alphabet*)
- $S \in V$ is the *start* (or *goal*) symbol
- P is a finite set of *production* rules
- V and T are disjoint

Production rules describe how the grammar transforms one sequence of symbols to another. The rules have the general form

$$x \rightarrow y$$

where x and y are sequences of symbols from $V \cup T$ such that x has length of at least one symbol.

A *sentence* in a language consists of any finite sequence of symbols that can be generated from the start symbol of a grammar by a finite sequence of productions from the grammar.

We call a sequence of productions that generates a sentence a *derivation* for that sentence.

Any intermediate sequence of symbols in a derivation is called a *sentential form*.

The *language* generated by the grammar is the set of all sentences that can be generated by the grammar.

41.3.1 Context-free grammars and BNF

To express the syntax of programming languages, we normally restrict ourselves to the family of *context-free grammars* (and its subfamilies) [118,184,185] context free. In a context-free grammar (CFG), the production rules have the form

$$A \rightarrow y$$

where $A \in V$ and y is a sequence of zero or more symbols from $V \cup T$. This means that an occurence of nonterminal A can be replaced by the sequence x.

We often express a grammar using a metalanguage such as the *Backus-Naur Form (BNF)* or *extended Backus-Naur Form (BNF)* [78,186,187].

For example, consider the following BNF description of a grammar for the unsigned binary integers:

```
<binary> ::= <digit>
<binary> ::= <digit> <binary>
<digit>  ::= '0'
<digit>  ::= '1'
```

The nonterminals are the symbols shown in angle brackets: `<binary>` and `<digit>`.

The terminals are the symbols shown in single quotes: `'0'` and `'1'`.

The production rules are shown with a nonterminal on the left side of the metasymbol `::=` and its replacement sequence of nonterminal and terminal symbols on the right side.

Unless otherwise noted, the start symbol is the nonterminal on the left side of the first production rule.

For multiple rules with the same left side, we can use the `|` metasymbol to write the alternative right sides concisely. The four rules above can be written as follows:

```
<binary> ::= <digit> | <digit> <binary>
<digit>  ::= '0' | '1'
```

We can also use the extended BNF metasymbols:

- `{` and `}` to denote that the symbols between the braces are repeated zero or more times
- `[` and `]` to denote that the symbols between the brackets are optional (i.e., occur at most once)

41.3.2 Derivations

Consider a derivation of the sentence `101` using the grammar for unsigned binary numbers above.

- Start symbol — `<binary>`
- Apply rule 2 — `<digit> <binary>`
- Apply rule 2 — `<digit> <digit> <binary>`
- Apply rule 3 — `<digit> 0 <binary>`
- Apply rule 4 — `1 0 <binary>`
- Apply rule 1 — `1 0 <digit>`
- Apply rule 4 — `1 0 1`

This is not the only possible derivation for `101`. Let's consider a second derivation of `101`.

- Start symbol — `<binary>`
- Apply rule 2 — `<digit> <binary>`
- Apply rule 4 — `1 <binary>`
- Apply rule 2 — `1 <digit> <binary>`
- Apply rule 3 — `1 0 <binary>`
- Apply rule 1 — `1 0 <digit>`
- Apply rule 4 — `1 0 1`

The second derivation applies the same rules the same number of times, but it applies them in a different order. This case is called the *leftmost derivation* because it always replaces the leftmost nonterminal in the sentential form.

Both of the above derivations can be represented by the *derivation tree* (or *parse tree*) [118,193] shown in Figure 41.1. (The numbers below the nodes show the rules applied.)

41.3.3 Regular grammars

The grammar above for binary numbers is a special case of a context-free grammar called a *right-linear grammar* [118,188]. In a right-linear grammar, *all* productions are of the forms

$A \rightarrow xB$
$A \rightarrow x$

where A and B are nonterminals and x is a sequence of zero or more terminals. Similarly, a *left-linear grammar* [118,188] must have *all* productions of the form:

$A \rightarrow Bx$
$A \rightarrow x$

A grammar that is either right-linear or left-linear is called a *regular grammar* [118,189].

(Note that *all* productions in a grammar must satisfy either the right- or left-linear definitions. They cannot be mixed.)

We can recognize sentences in a regular grammar with a simple "machine" (program)—a *deterministic finite automaton (DFA)* [118,191].

In general, we must use a more complex "machine"—a *pushdown automaton (PDA)*[118,192]—to recognize a context-free grammar.

We leave a more detailed study of regular and context-free grammars to courses on formal languages, automata, or compiler construction.

Now let's consider the concrete syntaxes for the ELI Calculator language—first infix, then prefix.

41.4 Infix syntax

An *infix syntax* for expressions is a syntax in which most binary operators appear between their operands as we tend to write them in mathematics and in programming languages such as Java and Haskell. For example, the following are intended to be valid infix expressions:

```
3
-3
x
```

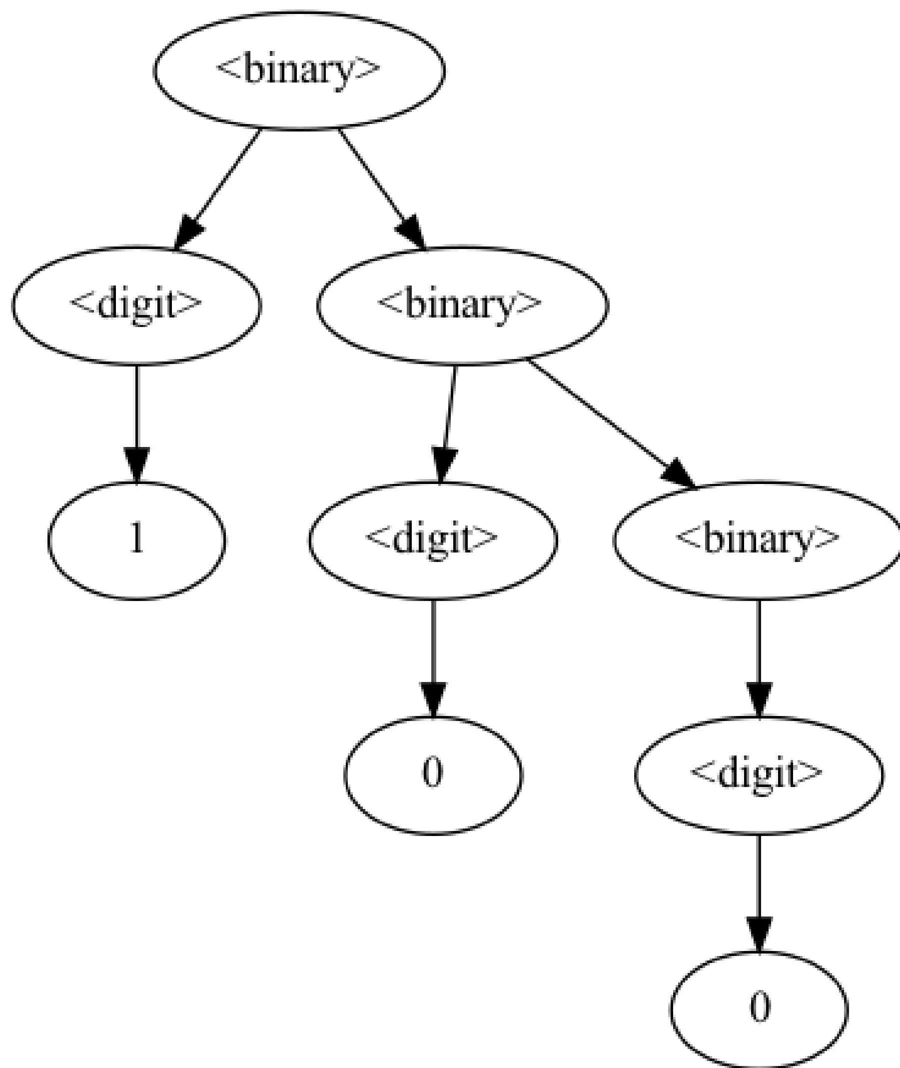


Figure 41.1: Derivation (parse) tree for binary number 101.

```
1+1
x + 3
(x + y) * (2 + z)
```

For example, we can present the concrete syntax of our core Calculator language with the grammar below. Here we just consider expressions made up of decimal integer constants; variable names; binary operators for addition, subtraction, multiplication, and division; and parentheses to delimit nested expressions.

We express the upper levels of the infix expression's syntax with the following context-free grammar where <expression> is the start symbol.

```
<expression> ::= <term> { <addop> <term> }
<term>       ::= <factor> { <mulop> <factor> }
<factor>     ::= <var> | <val>
               | '(' <expression> ')'
<val>        ::= [ '-' ] <unsigned>
<var>        ::= <id>
<addop>      ::=  '+'  |  '-'
<mulop>      ::=  '*'  |  '/'
```

Normally we want operators such as multiplication and division to bind more tightly than addition and subtraction. That is, we want expression `x + y * z` to have the same meaning as `x + (y * z)`. To accomplish this in the context-free grammar, we position <addop> in a higher-level grammar rule than <mulop>.

We can express the lower (lexical) level of the expression's grammar with the following production rules:

```
<id>         ::=  <firstid>  |  <firstid> <idseq>
<idseq>      ::=  <restid>   |  <restid> <idseq>
<firstid>    ::=  <alpha>    | '_'
<restid>     ::=  <alpha>    | '_'  | <digit>
<unsigned>   ::=  <digit>    |  <digit> <unsigned>
<digit>      ::=  any numeric character
<alpha>      ::=  any alphabetic character
```

The variables <digit> and <alpha> are essentially terminals. Thus the above is a regular grammar. (We can also add the rules for recognition of <addop> and <mulop> and rules for recognition of the terminals `(`, `)`, and `-` to the regular grammar.)

We assume that identifiers and constants extend as far to the "right" as possible. That is, an <id> begins with an alphabetic or underscore character and extends until it is terminated by some character other than an alphabetic, numeric, or underscore character (e.g., by whitespace or special character). Similarly for <unsigned>.

Otherwise, the language grammar ignores whitespace characters (e.g., blanks, tabs, and newlines). The language also supports end of line comments, any

characters on a line following a `--` (double dash).

We can use a *parsing* program (i.e., a *parser*) to determine whether a concrete expression (e.g., `1 + 1`) satisfies the grammar and to build a corresponding *parse tree* [118,194].

Aside: In a previous section, we use the term derivation tree to refer to a tree that we construct from the root toward the leaves by applying production rules from the grammar. We usually call the same tree a parse tree if we construct it from the leaves (a sentence) toward the root.

Figure 41.2 shows the parse tree for infix expression `1 + 1`. It has `<expression>` at its root. The children of a node in the parse tree depend upon the grammar rule application needed to generate the concrete expression. Thus the root `<expression>` has either one child—a `<term>` subtree—or three children—a `<term>` subtree, an `<addop>` subtree, and an `<expression>` subtree.

If the parsing program returns a boolean result instead of building a parse tree, we sometimes call it a *recognizer* program.

41.5 Prefix syntax

An alternative is to use a *parenthesized prefix syntax* for the expressions. This is a syntax in which expressions involving operators are of the form

```
( op operands )
```

where `op` denotes some "operator" and `operands` denotes a sequence of zero or more expressions that are the arguments of the given operator. This is a syntax similar to the language Lisp.

In this syntax, the examples from the section on the infix syntax can be expressed something like:

```
3
3
x
(+ 1 1)
(+ x 3)
(* (+ x y) (+ 2 z))
```

We express the upper levels of a prefix expression's syntax with the following context-free grammar, where `<expression>` is the start symbol.

```
<expression> ::=  <var> | <val> | <operexpr>
<var>        ::=  <id>
<val>        ::=  [ "-" ] <unsigned>
<operexpr>   ::=  '(' <operator> <operandseq> ')'
<operandseq> ::=  { <expression> }
<operator>   ::=  '+'  |  '*'  |  '-'  |  '/' | ...
```

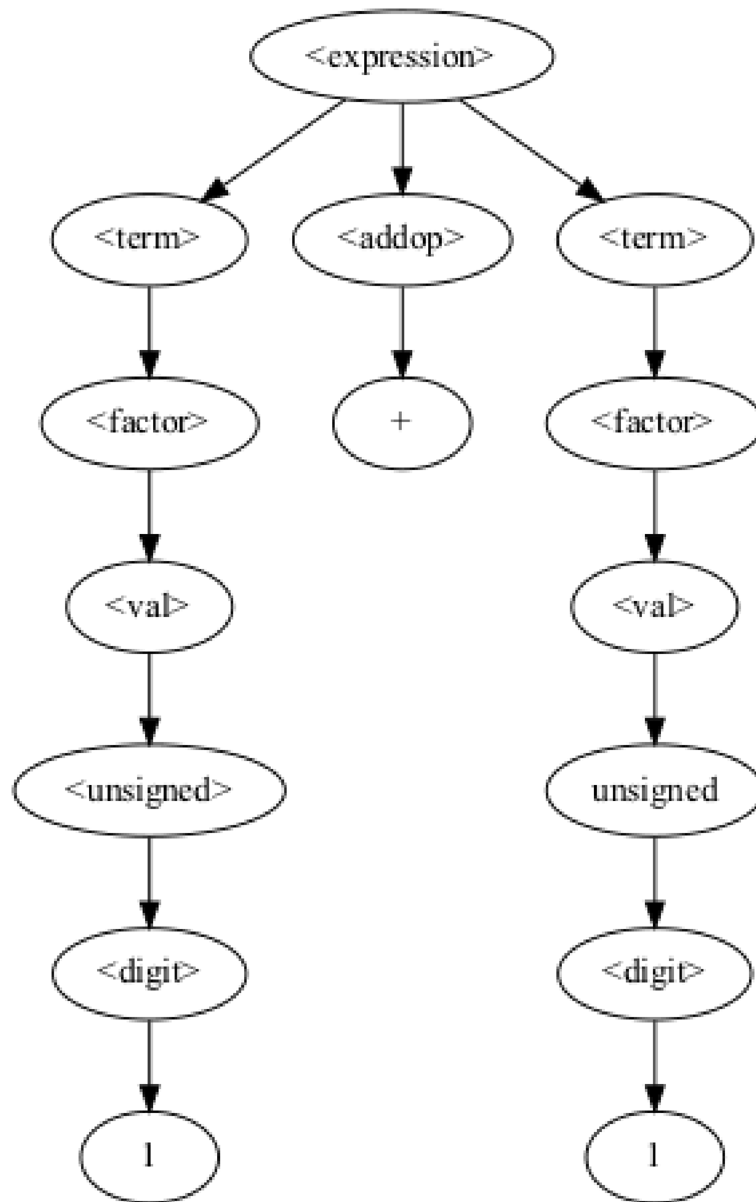


Figure 41.2: Parse tree for infix 1 + 1.

We can express the lower (lexical) level of the expression's grammar with basically the same regular grammar as with the infix syntax. (We can also add the rule for recognition of `<operator>` and for recognition of the terminals `(`, `)`, and `-` to the regular grammar

The parse tree for prefix expression `(+ 1 1)` is shown in Figure 41.3,

Because the prefix syntax expresses all operations in a fully parenthesized form, there is no need to consider the binding powers of operators. This makes parsing easier.

The prefix also makes extending the language to other operators—and keywords—much easier. Thus we will primarily use the prefix syntax in this and other cases studies.

We return to the problem of parsing expressions in a later chapter.

41.6 What Next?

This chapter (41) introduced the formal concepts related to a language's concrete syntax. It also introduced the *ELI (Exploring Languages with Interpreters) Calculator language*, which is the simple language we use in the following five chapters.

Chapter 42 examines the concepts of abstract syntax and evaluation, using the ELI Calculator language as an example.

41.7 Chapter Source Code

TODO if needed

41.8 Exercises

TODO

41.9 Acknowledgements

Chapters 41-46 of this book explore the ELI Calculator language and general concepts and techniques for language processing. I initially developed the ELI Calculator language (then called the Expression Language) case study for the Haskell-based offering of CSci 556, Multiparadigm Programming, in Spring 2017. I continued this work during Summer and Fall 2017 for the Fall 2017 offering of CSci 450, Organization of Programming Languages. I based the ELI Calculator language case study on ideas drawn, in part, from the following:

- the 2016 version of my Scala-based Expression Tree Calculator case study from my *Notes on Scala for Java Programmers* [49] (which was itself adapted from the the tutorial [152])

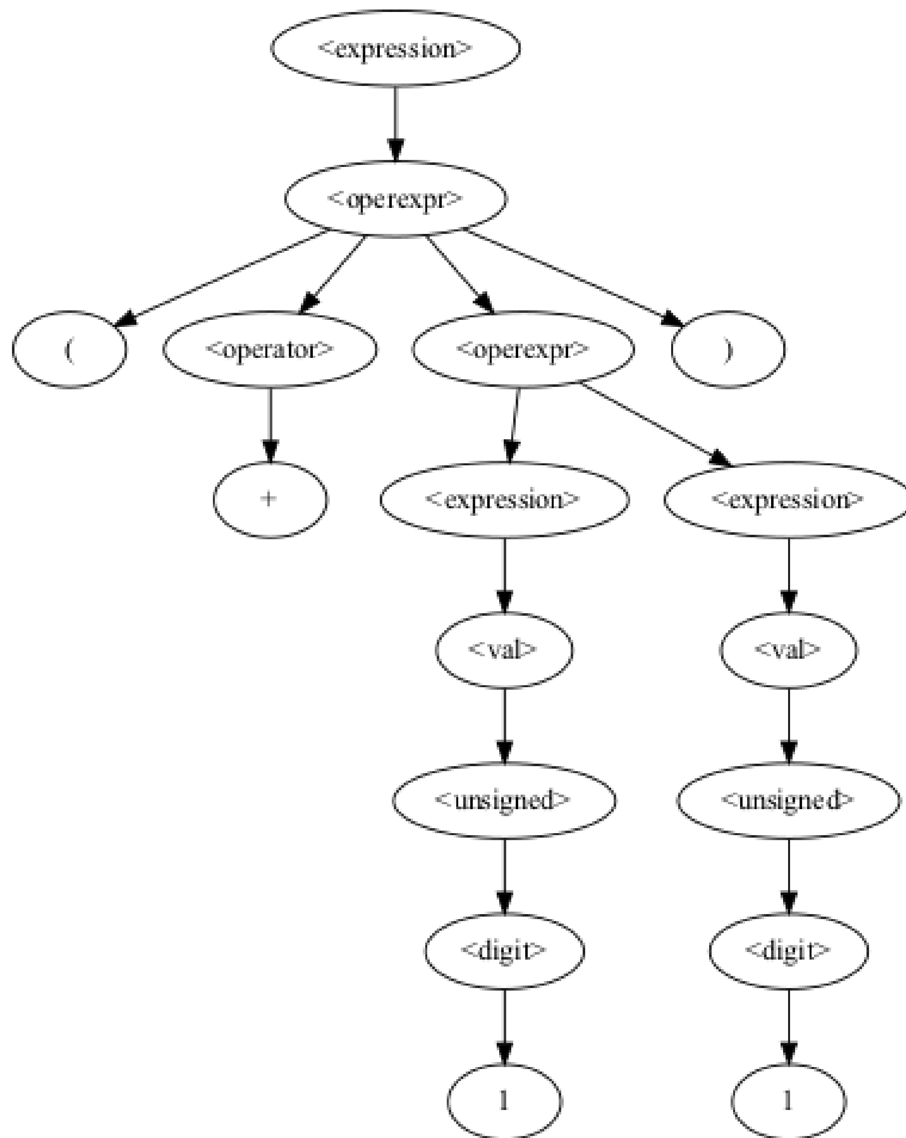


Figure 41.3: Parse tree for prefix (+ 1 1).

- the Lua-based Expression Language 1 and Imperative Core interpreters I developed for the Fall 2016 CSci 450 course
- chapters 1, 2, and 4 of Samuel Kamin's textbook *Programming Languages: An Interpreter-Based Approach* [108] and my work to implement three (Core, Lisp, and Scheme) of Kamin's interpeters in Lua in 2013
- sections 8.3 and 9.6 of the classic Richard Bird and Philip Wadler's textbook *Introduction to Functional Programming* [15]
- sections 14.2, 16.1, 17.5, and 18.3 of Simon Thompson's textbook [173]
- chapters 1-4 and 8 of Peter Sestoff's textbook *Programming Language Concepts* [159]
- chapters 21 (Recursive Descent Parser) and 22 (Parser Combinator) of Martin Fowler and Parsons's book *Domain-Specific Languages* [78].
- section 3.2 (Predictive Parsing) of Andrew W. Appel's textbook *Modern Compiler Implementation in ML* [3].
- chapters 6 (Purely Functional State) and 9 (Parser Combinators) from Paul Chiusano and Runar Bjarnason's *Functional Programming in Scala* [29].
- sections 1.2, 3.3, and 5.1 of Peter Linz's textbook *Formal Languages and Automata* [118]
- the Wikipedia articles on Formal Grammar [184], Linear Grammar {[188]}, Regular Grammar [189], Context-Free Grammar [185], Backus-Naur Form [186], Extended Backus-Naur Form [187], Parsing [194], Parse Tree [193], Recursive Descent Parser [196], LL Parser [195], Lexical Analysis [199], Finite-state Machine [190], Deterministic Finite Automaton [191], Push-down Automaton [192], Abstract Syntax [197], Abstract Syntax Tree [198], Stack Machine [200], Reverse Polish Notation [201], Association List [215], and Associative Array [216].

For the 2017 textbook, I organized this work into three chapters:

10. Expression Language Syntax and Semantics
11. Expression Language Parsing
12. Expression Language Compilation (a partial chapter)

In Summer 2018, I divided the previous Expression Language Syntax and Semantics chapter into three new chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*.

- Previous section 10.2 became new Chapter 41, Calculator Concrete Syntax.
- Previous sections 10.3-5 and 10.7-8 became new Chapter 42, Calculator Abstract Syntax and Evaluation.

- Previous sections 10.6 and 10.9 became new Chapter 43, Calculator Modular Structure, and were expanded.

In Fall 2018, I divided the 2017 Expression Language Parsing chapter into two new chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.*

- Previous sections 11.1-11.4 became new Chapter 44, Calculator Parsing.
- Previous sections 11.6-11.7 became new Chapter 45, Parsing Combinators.
- Previous section 11.5 was merged into new Chapter 43, Calculator Modular Structure.

In Fall 2018, I also renumbered previous chapter 12 to become new Chapter 46.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on the ELIFP textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

41.10 Terms and Concepts

Syntax, concrete syntax, formal grammar (variable and terminal symbols, alphabet, start or goal symbol), production rule, sentence, sentential form, language, context-free grammar, Backus-Naur Form (BNF), derivation, leftmost derivation, derivation tree, right-lean and right-linear grammar, regular grammar, deterministic finite automaton (DFA), pushdown automaton (PDA), infix and prefix syntaxes, lexical level, parsing, parser, parse tree, infix and prefix syntax.

42 Calculator: Abstract Syntax and Evaluation

42.1 Chapter Introduction

Chapter 41 introduced formal concepts related to concrete syntax and gave two different concrete syntaxes for the ELI Calculator language.

This chapter (42) introduces the concepts related to abstract syntax and language semantics. It encodes the essential structure of any ELI Calculator expression as a Haskell algebraic data type and defines the semantics operationally using a Haskell evaluation function. The abstract syntax also enables the expression to be transformed in various ways, such as converting it to a simpler expression while maintaining an equivalent value.

TODO: Rethink statement of goals below.

The goals of this chapter are to:

- explore the concepts of abstract syntax, abstract syntax trees, and expression evaluation
- define the semantics of the ELI Calculator language by designing its abstract syntax and an evaluation function
- examine techniques for abstract syntax tree simplification and manipulation

42.2 Abstract Syntax

The *abstract syntax* of an expression seeks to represent only the essential aspects of the expression's structure, ignoring nonessential, representation-dependent details of the concrete syntax [159,197].

For example, parentheses represent structural details in the concrete syntaxes given in Chapter 41. This structural information can be represented directly in the abstract syntax; there is no need for parentheses to appear in the abstract syntax.

We can represent arithmetic expressions conveniently using a tree data structure, where the nodes represent operations (e.g., addition) and leaves represent values (e.g., constants or variables). This representation is called a *abstract syntax tree* (AST) for the expression [159,198].

42.2.1 Abstract syntax tree data type

In Haskell, we can represent an abstract syntax trees using algebraic data types. Such types often enable us to express programs concisely by using pattern matching.

For the ELI Calculator language, we define the `Expr` algebraic data type—in the Abstract Syntax module (`AbSynCalc`)—to describe the abstract syntax tree.

```haskell
import Values ( ValType, Name )

data Expr = Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Var Name
          | Val ValType
            -- deriving Show?

instance Show Expr where
    show (Val v)   = show v
    show (Var n)   = n
    show (Add l r) = showParExpr "+" [l,r]
    show (Sub l r) = showParExpr "-" [l,r]
    show (Mul l r) = showParExpr "+" [l,r]
    show (Div l r) = showParExpr "/" [l,r]

showParExpr :: String -> [Expr] -> String
showParExpr op es =
    "(" ++ op ++ " " ++ showExprList es ++ ")"

showExprList :: [Expr] -> String
showExprList es = Data.List.intercalate " " (map show es)
```

Above in type `Expr`, the constructors `Add`, `Sub`, `Mul`, and `Div` represent the addition, subtraction, multiplication, and division, respectively, of the two operand subexpressions, `Var` represents a variable with a name, and `Val` represents a constant value.

Note that this abstract syntax is similar to the (Lisp-like) parenthesized prefix syntax described in Chapter 41.

We make type `Expr` an instance of class `Show`. We do not derive or define an instance of the `Eq` class because direct structural equality of trees may not be how we want to define equality comparisons.

We can thus express the example expressions from the Concrete Syntax chapter as follows:

```haskell
Val 3                   -- 3
Val (-3)                -- -3
Var "x"                 -- x
Add (Val 1) (Val 1)     -- 1+1
Add (Var "x") (Val 3)   -- x + 3
                        -- (x + y) * (2 - z)
Mul (Add (Var "x") (Var "y")) (Sub (Val 2) (Var "z"))
```

Figures 42.1 and 42.2 show abstract syntax trees for two example expressions

above.

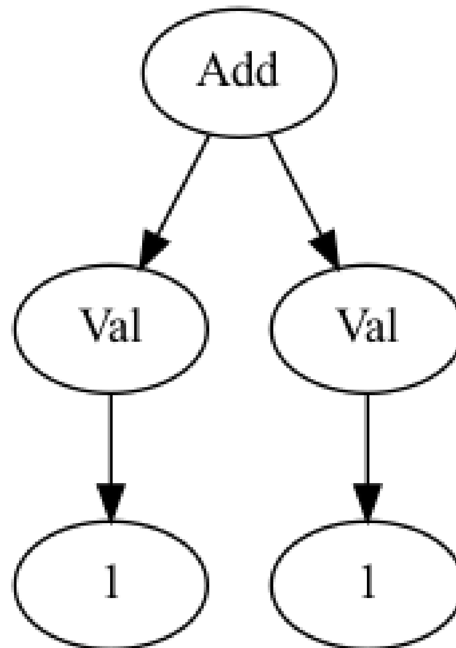


Figure 42.1: Abstract syntax tree for `1 + 1` and `(+ 1 1)`**

In Chapter 44 on parsing, we develop parsers for both the prefix and infix syntaxes. Both parsers construct abstract syntax trees using the algebraic data type `Expr`.

42.2.2 Values and variable names

The ELI Calculator language restricts values to `ValType`. The `Values` module indirectly defines this type synonym to be `Int`.

The abstract syntax allows a name to be represented by any string (i.e., type alias `Name`, which is defined to be `String` in the `Values` module). We likely want to restrict names to follow the usual "identifier" syntax. The parser for the concrete syntax should enforce this restriction. Or we could define Haskell functions to parse and construct identifiers, such as the functions below.

```
import Data.Char ( isAlpha, isAlphaNum )

getId :: String -> (Name,String)
getId []          = ([],[])
getId xs@(x:_)
    | isFirstId x = span isRestId xs
```

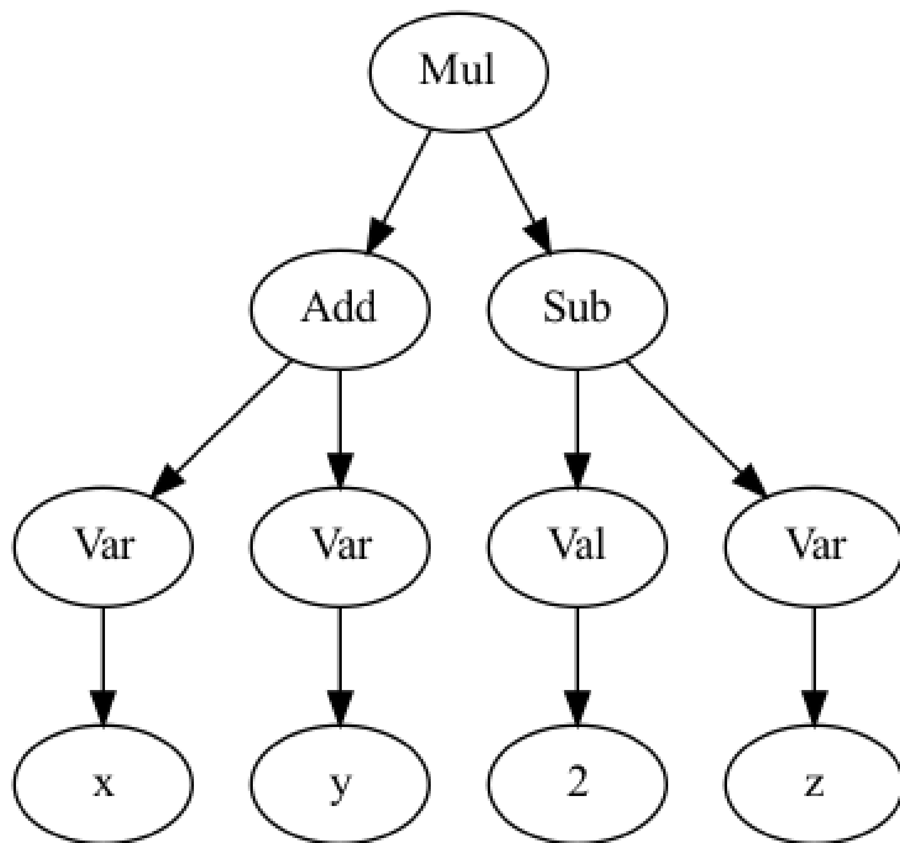


Figure 42.2: Abstract syntax tree for `(x + y) * (2 - z)` and `(* (+ x y) (- 2 z))`

```haskell
        | otherwise   = ([],xs)
    where
        isFirstId c = isAlpha c    || c == '_'
        isRestId  c = isAlphaNum c || c == '_'

identifier :: String -> Maybe Name
identifier xs =
    case getId xs of
        (xs@(_:_),[]) -> Just xs
        otherwise     -> Nothing
```

The getId function takes a string and parses an identifier at the beginning of the string. A valid identifier must begin with an alphabetic or underscore character and continue with zero or more alphabetic, numeric, or underscore characters.

The getId function uses the higher order function span to collect the characters that form the identifier. This function takes a predicate and returns a pair, of which the first component is the prefix string satisfying the predicate and the second is the remaining string.

In Chapter 44, we examine how to parse an expression's concrete syntax to build an abstract syntax tree.

42.3 Associative Data Structures

In language processing, we often need to associate some key (e.g., a variable name) with its value. There are several names for this type of data structure—*associative array* [215], *dictionary*, *map*, *symbol table*, etc.

As we saw in Chapter 21, an *association list* is a simple list-based implementation of this concept [215]. It is a list of pairs in which the first component is the *key* (e.g., a string) and the second component is the *value* associated with the key.

The Prelude function lookup, shown below (and in Chapter 21), searches an association list for a key and returns a Maybe value. If it finds the key, it wraps the associated value in a Just; if it does not find the key, it returns a Nothing.

```haskell
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup _   []    =  Nothing
lookup key ((x,y):xys)
    | key == x   =  Just y
    | otherwise  =  lookup key xys
```

For better performance with larger dictionaries, we can replace an association list by a more efficient data structure such as a Data.Map.Map. This structure implements the dictionary structure as a size-balanced tree. It provides a lookup function with essentially the same interface.

Of course, imperative languages might use a mutable *hash table* to implement a dictionary.

42.4 Semantics

Consider the evaluation of the ELI Calculator language abstract syntax trees as defined above.

42.4.1 Environments

To evaluate an expression, we must determine the current value of each variable occurring in the expression. That is, we must evaluate the expression in some *environment* that associates the variable names with their values.

For example, consider the expression `x + 3`. It might be evaluated in an environment that associates the value `5` with the variable `x`, written `{ x -> 5 }`. The evaluation of this expression yields the value `8`.

The environment `{ x -> 5 }` can be expressed in a number of ways in Haskell. Here we choose to represent it as a simple association list as follows:

```
[("x",5)]
```

This list associates a variable name in the first component with its integer value in the second component.

Looking up a key in an association list is an O(`n`) operation where `n` denotes the number of key-value pairs.

As noted above, a good alternative to the association list is a `Map` from the `Data.Map` library. It implements the dictionary as an immutable, size-balanced tree, thus its `lookup` function is an O(`log2 n`) operation.

In the ELI Calculator language implementation, we encapsulate the representation of the environment in the `Environments` module. This module exports the following type synonym and functions:

```
type AnEnv a =[(Name,a)]

newEnv     :: AnEnv a
toList     :: AnEnv a -> [(Name,a)]
getBinding :: Name -> AnEnv a -> Maybe a
hasBinding :: Name -> AnEnv a -> Bool
newBinding :: Name -> a -> AnEnv a -> AnEnv a
setBinding :: Name -> a -> AnEnv a -> AnEnv a
bindList   :: [(Name,a)] -> AnEnv a -> AnEnv a
```

For the purposes of our evaluation program, we can then define a specific environment with the type synonym `Env` in the Evaluator (`EvalCalc`) module as follows:

```
import Values       ( ValType, Name, defaultVal )
import AbSynExpr    ( Expr(..) )
import Environments ( AnEnv, Name, newEnv, toList, getBinding,
```

```
                      hasBinding, newBinding, setBinding,
                      bindList )

type Env = AnEnv ValType
```

42.4.2 Values of AST nodes

We express the *semantics* (i.e., meaning) of the various ELI Calculator language expressions (i.e., nodes of the AST) as follows.

- `c` evaluates to the constant (`NumType`) value `c`.
- `Var n` evaluates to the value of variable `n` in the environment, generating an error if the variable is not defined.
- `Add l r` evaluates to the sum of the values of the expression trees `l` and `r`.
- `Sub l r` evaluates to the difference between the values of the expression trees `l` and `r`.
- `Mul l r` evaluates to the product of the values of the expression trees `l` and `r`.
- `Div l r` evaluates to the quotient of the values of the expression trees `l` and `r`. Division by zero is not defined.

Operations `Add`, `Sub`, `Mul`, and `Div` are *strict*. They are undefined if any of their subexpressions are undefined.

42.4.3 Evaluation function

We can thus define a Haskell *evaluation function* (i.e., *interpreter*) for the ELI Calculator language as follows.

This function in the Evaluator module (`EvalCalc`) does a *post-order traversal* of the abstract syntax tree, first computing the values of the child subexpressions and then computing the value of of a node. The value is returned wrapped in an `Either`, where the `Left` constructor represents an error message and the `Right` constructor a good value.

```
import Values       ( ValType, Name, defaultVal )
import AbSynExpr    ( Expr(..) )
import Environments ( AnEnv, Name, newEnv, toList, getBinding,
                      hasBinding, newBinding, setBinding,
                      bindList )
type EvalErr = String
type Env     = AnEnv ValType

eval :: Expr -> Env -> Either EvalErr ValType
eval (Val v) _   = Right v
eval (Var n) env =
```

```haskell
        case getBinding n env of
            Nothing -> Left ("Undefined variable " ++ n)
            Just i  -> Right i
    eval (Add l r) env =
        case (eval l env, eval r env) of
            (Right lv, Right rv) -> Right (lv + rv)
            (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
            (x@(Left le),  _   ) -> x
            (_,      y@(Left le)) -> y
    eval (Sub l r) env =
        case (eval l env, eval r env) of
            (Right lv, Right rv) -> Right (lv - rv)
            (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
            (x@(Left le),  _   ) -> x
            (_,      y@(Left le)) -> y
    eval (Mul l r) env =
        case (eval l env, eval r env) of
            (Right lv, Right rv) -> Right (lv * rv)
            (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
            (x@(Left le),  _   ) -> x
            (_,      y@(Left le)) -> y
    eval (Div l r) env =
        case (eval l env, eval r env) of
            (Right _,  Right 0 ) -> Left "Division by 0"
            (Right lv, Right rv) -> Right (lv `div` rv)
            (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
            (x@(Left le),  _   ) -> x
            (_,      y@(Left le)) -> y
```

Consider an example with a simple `main` function below (that could be added to the `EvalExpr` module) that evaluates the example expressions from a previous section. (See the extended Evaluator module (`EvalCalcExt`).)

```haskell
    main =
      do
        let env = [("x",5), ("y",7),("z",1)]
        let exp1 = Val 3           -- 3
        let exp2 = Var "x"         -- x
        let exp3 = Add (Val 1) (Val 2)     -- 1+2
        let exp4 = Add (Var "x") (Val 3)   -- x + 3
        let exp5 = Mul (Add (Var "x") (Var "y"))
                       (Add (Val 2) (Var "z")) -- (x + y) * (2 + z)
        putStrLn ("Expression: " ++ show exp1)
        putStrLn ("Evaluation with x=5, y=7, z=1:  "
                  ++ show (eval exp1 env))
        putStrLn ("Expression: " ++ show exp2)
        putStrLn ("Evaluation with x=5, y=7, z=1:  "
```

```
              ++ show (eval exp2 env))
    putStrLn ("Expression: " ++ show exp3)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
              ++ show (eval exp3 env))
    putStrLn ("Expression: " ++ show exp4)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
              ++ show (eval exp4 env))
    putStrLn ("Expression: " ++ show exp5)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
              ++ show (eval exp5 env))
```

When `main` is called, it first computes he values of the various expressions in the environment `{ x -> 5, y -> 7 }` and then prints their results.

```
Expression: 3
Evaluation with x=5, y=7, z=1:  Right 3
Expression: x
Evaluation with x=5, y=7, z=1:  Right 5
Expression: (+ 1 2)
Evaluation with x=5, y=7, z=1:  Right 3
Expression: (+ x 3)
Evaluation with x=5, y=7, z=1:  Right 8
Expression: (* (+ x y) (+ 2 z))
Evaluation with x=5, y=7, z=1:  Right 36
```

42.5 Simplification

TODO: Should the discussion of Simplification and Differentiation be in the main line of the chapter or separated into a project (or projects) with exercises? Simplfication is related to the global

An expression may be more complex than necessary. We can *simplify* it, perhaps with the intention of *optimizing* its evaluation.

An operation whose operands are constants can be simplified by replacing it by the appropriate constant. For example, `Add (Val 3) (Val 4)` is the same semantically as `Val 7`.

Similarly, we can take advantages of an operation's identity element and other mathematical properties to simplify expressions. For example, `Add (Val 0) (Var "x")` is the same as `Var "x"`.

We can thus define a skeletal function `simplify` as follows. As with `eval`, the `simplify` function traverses the abstract syntax tree using a post-order traversal.

```
simplify :: Expr -> Expr
simplify (Add l r) =
    case (simplify l, simplify r) of
        (Val 0, rr)    -> rr
```

```
            (ll, Val 0)    -> ll
            (Val x, Val y) -> Val (x+y)
            (ll, rr)       -> Add ll rr
simplify (Mul l r) =
    case (simplify l, simplify r) of
            (Val 0, rr)    -> Val 0
            (ll, Val 0)    -> Val 0
            (Val 1, rr)    -> rr
            (ll, Val 1)    -> ll
            (Val x, Val y) -> Val (x*y)
            (ll, rr)       -> Mul ll rr
simplify t@(Var _) = t
simplify t@(Val _) = t
```

In an exercise, you are asked to complete the development of this function.

See the incomplete Process AST module (`ProcessAST`) for the sample code in this section and the next one.

42.6 Symbolic Differentiation

Suppose that we redefine the `Expr` type to support double precision floating point (i.e., `Double`) values.

Then let's consider *symbolic differentiation* of the arithmetic expressions. Thinking back to our study of differential calculus, we identify the following rules for differentiation:

- The derivative of a sum is the sum of the derivatives.
- The derivative of a product of two operands is the sum of the product of (a) the first operand and the derivative of the second and (b) the second operand and the derivative of the first.
- The derivative of some variable `v` is 1 if differentiation is relative to `v` and is 0 otherwise.
- The derivative of a constant is 0.

We can directly translate these rules into a skeletal Haskell function that uses the above data types, as follows:

```
deriv :: Expr -> Name -> Expr
deriv (Add l r) v = Add (deriv l v) (deriv r v)
deriv (Mul l r) v = Add (Mul l (deriv r v)) (Mul r (deriv l v))
deriv (Var n)   v
      | v == n    = Val 1
deriv _         _ = Val 0
```

See the incomplete Process AST module (`ProcessAST`) for the sample code in this section.

42.7 What Next?

Chapter 41 presented concrete syntax concepts, illustrating them with two different concrete syntaxes for the ELI Calculator language.

This chapter (42) presented abstract syntax trees as structures for representing the essential features of the syntax in a form that can be evaluated directly. The same abstract syntax can encode either of the two concrete syntaxes for the ELI Calculator language.

Chapter 44 introduces lexical analysis and parsing as techniques for processing concrete syntax expressions to generate the equivalent abstract syntax trees.

Before we look at parsing, let's examine the overall modular structure of the ELI Calculator language interpreter in Chapter 43.

42.8 Chapter Source Code

This chapter involves several of the ELI Calculator language modules:

- Abstract Syntax module (`AbSynCalc`)—to describe the abstract syntax tree.
- `Values` module
- `Environments` module.
- Evaluator (`EvalCalc`) module
- extended Evaluator module (`EvalCalcExt`).)

It also has the incomplete Process AST module (`ProcessAST`) related to the simplification and differentiation discussion and exercises.

42.9 Exercises

1. Extend the abstract syntax tree data type `Expr`, which is defined in the Abstract Syntax module (`AbSynCalc`), to add new operations `Neg` (negation), `Min` (minimum), `Max` (maximum), and `Exp` (exponentiation).

   ```
   data Expr = ...
             | Neg Expr
             | Min Expr Expr
             | Max Expr Expr
             | Exp Expr Expr
             ...
               deriving Show
   ```

   Then extend the `eval` function, which is defined in the Evaluator module (`EvalCalc`), to add these new operations with the following informal semantics:

- `Neg e` negates the value of expression `e`. For example, `Neg (Val 1)` yields `(Val (-1))`.
- `Min l r` yields the smaller value of expression `l` and expression `r`.
- `Max l r` yields the larger value of expression `l` and `r`.
- `Exp l r` raises the value of expression `l` to a power that is the value of expression `r`. It is undefined for a negative exponent value `r`.

These operations are all strict; they only have values if all their subexpressions also have values.

2. Extend the `simplify` function to support operations `Sub` and `Div` and the new operations given in the previous exercise.

   This function should simplify the abstract syntax tree by evaluating subexpressions involving only constants (not evaluating variables) and handling special values like identity and zero elements.

3. Extend the `simplify` function from the previous exercise in other ways. For example, take advantage of mathematical properties such as *associativity* (`(x + y) + z = x + (y + z)`), *commutativity* (`x + 1 = 1 + x`), and *idempotence* (`x min x = x`).

4. Extend the abstract syntax tree data type `Expr` to include the binary operators `Eq` (equality) and `Lt` (less-than comparison), logical unary operator `Not`, and the ternary conditional expression `If` (if-then-else).

```
data Expr = ...
          | Eq  Expr Expr
          | Lt  Expr Expr
          | Not Expr
          | If  Expr Expr Expr
          ...
            deriving Show
```

   Then extend the `eval` function to implement these new operations.

   This extended language does not have Boolean values. We represent "false" by integer 0 and "true" by a nonzero integer, canonically by 1.

   We can express the informal semantics of the new ELI Calculator language expressions as follows:

   - `Eq l r` yields the value 1 if expressions `l` and `r` have the same value; it yields the value 0 if `l` and `r` have different values.
   - `Lt l r` yields the value 1 if the value of expression `l` is smaller than the value of expression `r`; it yields the value 0 if `l` is greater than or equal to `r`.

- `Not i` yields 1 if the value of expression `i` is 0; it yields the value 0 if `i` is nonzero.
- `If c l r` first evaluates expression `c`. If `c` has a nonzero value, the `If` yields the value of expression `l`. If `c` has value 0, the `If` yields the value of expression `r`.

Operations `Eq`, `Lt`, and `Not` are strict for all subexpressions; that is, they are undefined if any subexpression is undefined.

Operation `If` is strict in its first subexpression `c`.

Note: The constants `falseVal` and `trueVal` and the functions `boolToVal` and `valToBool` in the `Values` module may be helpful. (The intention of the `Values` module is to keep the representation of the values hidden from the rest of the interpreter. In particular, these constants and functions these are to help encapsulate the representation of booleans as the underlying values.)

5. Extend the abstract syntax tree data type `Expr` from the previous exercise (which defines operator `If`) to include a `Switch` expression.

```
data Expr = ...
          | Switch Expr Expr [Expr]
          ...
            deriving Show
```

   Then extend the `eval` function to implement this new operation.

   We can express the informal semantics of this new ELI Calculator language expression as follows:

   - `Switch n def exs` first evaluates expression `n`. If the value of `n` is greater than or equal to 0 and less than `length exs`, then the `Switch` yields the value of the `n`th expression in list `exs` (where the first element is at index 0). Otherwise, the `Switch` yields the value of the default expression `def`.

6. Develop an object-oriented program (e.g., in Java) to carry out the same functionality as the `Expr` data type and `eval` function described in this chapter. That is, define a class hierarchy that corresponds to the `Expr` data type and use the message-passing style to implement the needed classes and instances.

7. Extend the object-oriented program from the previous exercise to the `Neg`, `Min`, `Max`, and `Exp` as described in an earlier exercise.

8. Extend the object-oriented program from the previous exercise to implement the `Eq`, `Lt`, `Not`, and `If` as described in another earlier exercise.

9. Extend the object-oriented program above to implement simplification.

10. For this exercise, redefine the `Expr` data type above to hold `Double` constants instead of `Int`. In addition to `Add`, `Mul`, `Sub`, `Div`, `Neg`, `Min`, `Max`, and `Exp`, extend the data type and `eval` function to include the trigonometric operators `Sin` and `Cos` for sine and cosine.

11. Using the extended `Double` version of `Expr` from the previous exercise, extend function `deriv` to support all the operators in the data type.

42.10 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

42.11 Terms and Concepts

Abstract syntax, abstract syntax tree (AST), associative data structure, environment, value, semantics, evaluation function, interpreter, simplification, optimization, symbolic differentiation, associativity, commutativity (symmetry), idempotence.

43 Calculator: Modular Structure

43.1 Chapter Introduction

TODO: Write missing pieces and flesh out other sections

43.2 Module Dependencies

An ELI Calculator interpreter consists of seven modules. The dependencies among modules as shown in Figure 43.1. (The module at the tail of an arrow depends on the module at the head.)

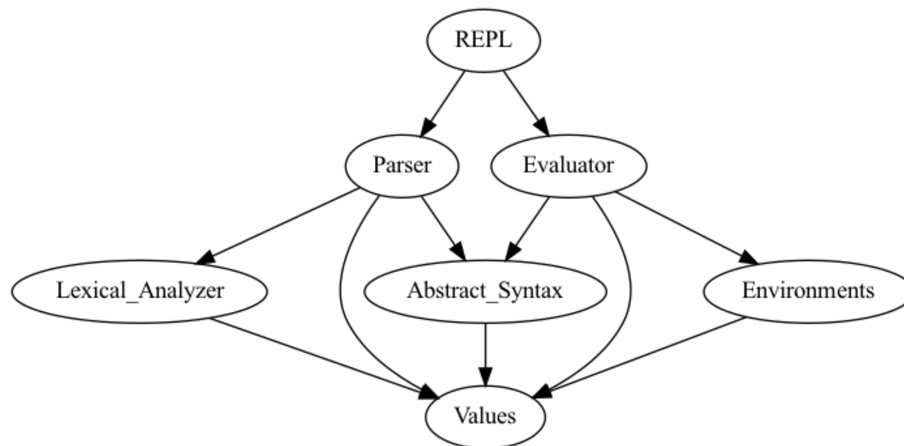


Figure 43.1: ELI Calculator language module dependencies.

We examine each module in the following sections.

TODO: Some of these are concrete modules intended for direct use by all implementations. Some are concrete modules intended for use by just ELI Calculator. Some are "abstract modules" intended to define an interface for implementation by each language as needed. Some may, in some sense, define a module role (e.g., same secret) that must be satisfied for all languages, but which may have a different abstract interface. Etc. This probably should be clarified for each module after study and thought.

43.3 Values Module

The *Values* module `Values` was introduced in Chapter 42. It encapsulates the definitions and functions that know the specific representation of an ELI language's data. Other modules for that language should use its public features to enable the representation to be changed easily.

The *secret* of the information-hiding module Values is the specific representation for the values supported by the language.

This module currently supports both the ELI Calculator language and the ELI Imperative Core language we examine in later chapters. For both languages, the only type of values supported are integers. Booleans are encoded as integers.

The Values module's *abstract interface* includes the following public features:

- Type `ValType` is the type of the values in the ELI language.
- Constant `defaultVal` is the default value for ELI language variables when no value is specified.

  Note: A *constant* is an argumentless function in Haskell.
- Constants `falseVal` and `trueVal` are the ELI language's canonical representations for false and true as `ValType` values, respectively.
- Function `boolToVal` converts Haskell `Bool` values `False` and `True` to `falseVal` and `trueVal`, respectively.
- Function `valToBool v` converts ELI language value `v` to Haskell `False` and `True` appropriately.

  `falseVal` is mapped to Haskell `False`. Any other value is mapped to Haskell `True`; we call these *truthy* values.

If a language supports types other than integers, then that language will need a variant of the Values module that redefines `ValType` accordingly and perhaps defines additional public functions. However, the redefined module should seek to preserve the secret and other features of the abstract interface.

The interface also includes the following, which are intended for the exclusive use of the lexical analysis module to support finite range integers (e.g., a string representation of an integer that is beyond the range of `Int`).

- Type `NumType` is the actual type used to represent integers.
- Function `toNumType` takes a string of digits `numstr` and returns an `Either String NumType` where `Left` wraps an error message and `Right` wraps `numstr` interpreted as a `NumType` value.

TODO: Review how integer constant overflow is handled and seek to encapsulate the representation better. Also might comment that the knowledge of the value representation is probably shared between the Values and Lexical Analysis modules.

The Values module does not depend upon any other modules. All other current modules depend upon it directly except the user-interface module REPL.

43.4 Environments Module

An *environment* is a mapping between a name and its value.

The *Environments* module `Environments` was introduced in Chapter 42. It encapsulates the definitions and functions that know the specific representation of an environment for an ELI language. Other modules should use its public features to enable the representation to be changed easily.

The *secret* of the information-hiding module Environments is the specific representation for the environments used by the language's interpreter. This module currently supports both the ELI Calculator and the ELI Imperative Core languages (defined in future chapters). Given that the "value" is a polymorphic parameter, it should work for most languages unless the nature of names changes significantly.

- The ELI Calculator language creates a single global environment consisting of a set of `(Name,ValType)` pairs that map variables to their values.
- The ELI Imperative Core language (which also supports function definitions and function calls) creartes three different environments, all of which are implemented with the Environments module:
  - a global variable environment consisting of a set of `(Name,ValType)` pairs (as above)
  - a global function definition environment consisting of a set of '`Name`-function definition pairs
  - a local parameter environment like the global variable environment except holding the values of the parameters for a function call

The Environments module's *abstract interface* includes the following public features.

- Type `AnEnv a` is the type of an environment whose values have polymorphic parameter type `a`.
- Type `Name` is imported from the `Values` module and reexported.
- Constructor function `newEnv` returns a new empty environment.
- Mutator function `newBinding` adds a new name-value binding to an environment.
- Mutator function `setBinding` changes the value of an existing name in an environment.
- Mutator function `bindList` takes a list of name-value pairs and adds a new binding for each to an environment.
- Accessor function `toList` returns an association list equivalent to the environment.
- Accessor function `getBinding` returns the value associated with a given name.

- Query function `hasBinding` returns `True` if and only if the given name is bound in the environment.

The Environments module depends upon the Values module and the Evaluator module depends upon it.

43.5 Abstract Syntax Module

The *Abstract Synax* module `AbSynCalc` module was introduced in Chapter 42. It centralizes the abstract syntax definition for the ELI Calculator language so it can be imported where needed.

The abstract syntax consists of algebraic data type definitions. The semantics of the abstract syntax tree is known by modules that must create (e.g., parser) and use (e.g., evaluator) the abstract syntax trees.

TODO: Review how the AST semantics is handled to see if it can be better encapsulated. But remember that too much abstraction may make the pedagogical goals more difficult to achieve (e.g., exercises to add new elements to the abstract syntax and semantics).

The ELI Calculator Language's Abstract Syntax module defines and exports the algebraic data type `Expr` and implements it as an instance of class `Show`. Values of type `Expr` are the abstract syntax trees for the ELI Calculator language.

The module also exports types `ValType` and `Name` that it imports from the Values module.

The equivalent modules for other languages must define the abstract syntax for that language using appropriate algebraic data types that are instances of `Show`. They should, however, use

The Abstract Syntax module depends upon the Values module and the Evaluator and Parser modules depend upon it.

43.6 Evaluator Module

The *Evaluator* module `EvalCalc` was introduced in Chapter 42. It encapsulates the definition of the evaluation function (i.e., the semantics) of the ELI Calculator language.

TODO: Consider how to handle the extensions to the Evaluator module in Chapter 42 for simplification and differentiation (i.e., `ProcessAST` module).

The *secret* of the `EvalCalc` is the implementation of the semantics of the language, including the specifics of the environment. Currently, some aspects of the language semantics are not completely encapsulated within the Evaluator module; they are shared with the Parser module (which creates the abstract syntax trees initially).

TODO: Explore whether the semantics can be better encapsulated and continue to meet the pedagogical goals of the interpreter.

The Evaluator module's *abstract interface* includes the following public features.

TODO: Perhaps simply call this an "interface" because it is not likely used by more than one concrete implementation.

- Evaluation function `eval` takes an ELI Calculator abstract syntax tree (i.e., an `Expr`) and returns its value in the environment.
- Type `Env` defines the environment (i.e., mapping of variable names to their values) for the ELI Calculator language.
- Constant `lastVal` is the variable name whose value in the environment is the result of the most recent expression evaluation.
- Constructor function `newEnviron` creates a new environment that is empty except that variable `lastVal` is set to `Values.defaultVal`.
- Query function `hasNameBinding` returns `True` if and only if the given name is defined in the environment.
- Mutator function `newNameBinding` that creates a new variable in the environment and gives it a value.
- Mutator function `setNameBinding` that sets an existing variable in the environment to a new value.
- Accessor function `getNameBinding` retrieves the value of a variable from the environment.
- Accessor function `showEnviron` displays all the variables and their values in the environment.
- Type `EvalErr` represents error messages arising from evaluation.
- Types `ValType` and `Name` are imported from the Values module and reexported.
- Type `Expr` is imported from the Abstract Syntax module and reexported.

TODO: Comment on how the above secret should be preserved and might need to be modified for other ELI languages.

The Evaluator module depends directly upon the Abstract Syntax, Environments, and Values modules. The language's user-interface module REPL depends upon it. However, as noted above, the Evaluator and Parser modules currently share some aspects of the language semantics.

43.7 Lexical Analysis Module

The *Lexical Analyzer* module `LexCalc` is introduced in Chapter 44. It is common to both the prefix and infix parsers for the ELI Calculator language.

The *secret* of this module is the lexical structure of the concrete language syntax.

The Lexical Analyzer module's *abstract interface* consists of the following public features.

- Algebraic data type `Token` describes the smallest units of the syntax processed by the parser, such as identifiers, operator symbols, parentheses, etc.
- Function `showTokens` is a convenience function that shows a list of tokens as a string.
- Function `lexx` takes a string and returns the corresponding list of lexical tokens, but it does not distinguish among identifiers, keywords, and operators.
- Function `lexer` takes a string and returns the corresponding list of lexical tokens, distinguishing among identifiers, keywords, and operators.
- Type `NumType` is imported from the Values module and reexported; it is the actual type used to represent integers.
- Type `Name`{.haskell is from the Values module and reexported; it is the type that represents "names" such as identifiers and operator symbols.

TODO: Consider whether the above should just be an interface rather than an abstract interface. Also how should the secret and interface be preserved and modified for other languages. Also consider what I should say below about the special dependence upon the Values module and any sharing of information about values.

The Lexical Analyzer module depends upon the Values module and the Parser module depends upon it.

43.8 Parser Modules

Chapter 44 introduces two alternative implementations of the *Parser* abstract module for the ELI Calculator language. These implementations correspond to the two different concrete syntaxes given in Chapter 41. Both use the same Lexical Analyzer.

- Module `ParsePrefixCalc` parses an ELI Calculator language *prefix* expression and generates the equivalent abstract syntax tree.
- Module `ParseInfixCalc` parses an ELI Calculator language *infix* expression and generates the equivalent abstract syntax tree,

The *secret* of the abstract parser module is how the input syntax is recognized and translated to the abstract syntax.

The Parser abstract module's *abstract interface* consists of the following public features.

- Function `parse` takes an input string, parses it according to the corresponding ELI Calculator language concrete syntax and returns an `Either` item wrapping the `Expr` abstract syntax tree (`Right`) or an error message (`Left`).
- Function `parseExpression` takes a `Token` list, parses an `Expr` from the beginning of the list, and returns a pair consisting of
  - an `Either` wrapping the `Expr` abstract syntax tree found (`Right` or an error message (`Right`
  - the `Token` list remaining after the `Expr`.
- Type `ParErr` is the type of the error messages.
- Function `trimComment` trims an end-of-line comment from a line of text.
- Function `getName` takes a string and returns a `Just` wrapping a `Name` if it is a valid identifier or a `Nothing` if any non-identifier characters occur.
- Function `getValue` extracts an identifier from the beginning of a string and returns the identifier and the remaining string.
- Types `ValType` and `Name` are imported from the Values module and reexported.
- Type `Expr` is imported from the Abstract Syntax module and reexported.

TODO: Comment on how the above secret should be preserved and might need to be modified for other ELI languages.

The Parser module depends directly upon the Lexical Analyzer, Abstract Syntax, and Values modules. The language's user-interface module REPL depends upon it. However, as noted above, the Evaluator and Parser modules currently share some aspects of the language semantics.

43.9 REPL Modules

A REPL (Read-Evaluate-Print Loop) is a command line user interface with the following cycle of steps:

1. *Read* an input from the command line.

   If the input is an exit command, exitloop ; else continue.

2. *Evaluate* the expression after parsing.
3. *Print* the resulting value.
4. *Loop* back to step 1.

The *secret* of the REPL modules is how the user interacts with the interpreter.

The ELI Calculator language interpreter provides two REPL modules:

- `PrefixCalcREPL` that uses the Calculator language's prefix syntax
- `InfixCalcREPL` that uses the Calculator languages's infix syntax

In addition to accepting ELI Calculator expressions, they accept the REPL commands `:set`, `:display`, and `:quit`.

TODO: What about `:use`? Do I need to elaborate on the commands further? Probably.

TODO: The REPL functions need to be refactored. Also the issue of the `:use` command versus a `use` expression in the language needs to be reconsidered.

The REPL module depends directly upon the Parser and Evaluator modules. No other modules depend upon it.

43.10 Code Improvement Modules

TODO: Consider how this should be presented in both Chapter 42 and 43.

In addition, the partially implemented *Process AST* module includes the skeleton `simplify` and `deriv` functions discussed in Chapter 42.

This module is "wrapper" for the `EvalCalc` module currently.

43.11 What Next?

TODO

43.12 Chapter Source Code

The ELI Calculator language interpreter includes the following source code modules:

- *Values* module `Values`
- *Environments* module `Environments`
- *Abstract Synax* module `AbSynCalc`
- *Evaluator* module `EvalCalc`
- *Lexical Analyzer* module `LexCalc`
- *Parser* modules
  - Prefix parser `ParsePrefixCalc`
  - Infix parser `ParseInfixCalc`
- *REPL* modules
  - Prefix REPL `PrefixCalcREPL`
  - Infix REPL `InfixCalcREPL`

- Skeleton simplify and derivative module `ProcessAST`

43.13 Exercises

TODO

43.14 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

43.15 Terms and Concepts

TODO

44 Calculator: Parsing

44.1 Chapter Introduction

The *ELI Calculator language* case study examines how we can represent and process simple arithmetic expressions using Haskell.

In Chapter 41, we described two different concrete syntaxes for expressions written as text. In Chapter 42, we defined the abstract syntax represented as an algebraic data type and the language semantics with an evaluation function.

Chapter 40 introduced the general concepts of lexical analysis and parsing. In this chapter, we design and implement a hand-coded lexical analyzer and two hand-coded recursive descent parsers for the two concrete syntaxes given in Chapter 41. The parsers also construct the corresponding abstract syntax trees.

44.2 Parsing

TODO: Add citations for some of the key terms?

A programming language processor uses a parser to determine whether a program satisfies the grammar for the language's concrete syntax. The parser typically constructs some kind of internal representation of the program to enable further processing.

A common approach to parsing is to divide it into at least two phases:

- A lexical analyzer converts the sequence of characters into a sequence of low-level syntactic units called *tokens*. The grammar describing the tokens is usually a *regular grammar*, which can be processed efficiently using a *finite state machine*.
- A parser converts the sequence of tokens into an initial *semantic model* (e.g., into an *abstract syntax tree* supported by a *symbol table*). The grammar describing the language's full syntax is typically a *context-free gramma*r, which requires more complex mechanisms to process.,

If the language has aspects that cannot be described with a context-free grammar, then additional phases may be needed to handle issues such as checking types of variables and expressions and ensuring that variables are declared before they are used.

Of course, regular grammars are context-free grammars, so a separate lexical analyzer is not required. But use of a separate lexical analyzer often leads to a simpler parser and better performance.

However, some approaches to parsing, such as the use of parser combinators, can conveniently handle lexical issues as a part of the parser.

In this chapter, we use the two-stage approach to parsing of the ELI Calculator language. We define a lexical analyzer and parsers constructed using a technique

called recursive descent parsing. The parsers construct abstract syntax trees using the algebraic data type defined in Chapter 42 (i.e., in the Abstract Syntax module).

Chapter 45 generalizes the recursive descent parsers to a set of parsing combinatiors.

44.3 Lexical Analysis

TODO: Lexical analyzer code, etc., probably needs to be updated to reflect handling of finite integer errors.

In computing science, *lexical analysis* [199] is typically the process of reading a sequence of characters from a language text and assembling the characters into a sequence of *lexemes*, the smallest meaningful syntactic units. In a natural language like English, the lexemes are typically the words of the language.

The output of lexical analysis is a sequence of *lexical tokens* (usually just called *tokens*). A token associates a syntactic category with a lexeme. In a natural language, the syntactic category may be the word's part of speech (noun, verb, etc.).

We call the program that carries out the lexical analysis a *lexical analyzer*, *lexer*, *tokenizer*, or *scanner*. (However, the latter term actually refers to one phase of the overall process.)

In a programming language, the syntactic categories of tokens consist of entities such as identifiers, integer literals, and operators.

The "whitespace" characters such as blanks, tabs, and newlines are usually not tokens themselves. Instead, they are delimiters which define the boundaries of the other lexemes. However, in some programming languages, the end of a line or the indentation at the beginning of a line have implicit structural meaning in the language.

Consider the ELI Calculator language infix syntax. The character sequence

```
30   +   ( x1 * 2)
```

includes seven tokens:

- integer literal `30`
- addition operator `+`
- left parenthesis symbol `(`
- identifier `x1`
- multiplication operator `*`
- integer literal `2`
- right parenthesis symbol `)`

Tokenization has two stages—a scanner and an evaluator.

A *scanner* processes the character sequence and breaks it into lexeme strings. It usually recognizes a language corresponding to a *regular grammar*, one of the simplest classes of grammars, and is, hence, based on a *finite state machine* [190]. However, in some cases, a scanner may require more complex grammars and processors.

A token *evaluator* determines the syntactic category of the lexeme string and tags the token with this syntactic information.

Sometimes a lexical analyzer program combines the two stages into the same algorithm.

44.3.1 Prefix syntax

Now let's consider a lexical analyzer for the prefix syntax for the ELI Calculator language.

File `LexCalc.hs` gives an example Haskell module that implements a lexical analyzer for this concrete syntax.

The ELI Calculator language's prefix syntax includes the following syntactic categories: identifiers, keywords, integer literals, operators, left parenthesis, and right parenthesis.

The *left* and *right parenthesis* characters are the only lexemes in those two syntactic categories, respectively.

An *identifier* is the name for variable or other entity. We define an identifier to begin with an alphabetic or underscore character and include all contiguous alphabetic, numeric, or underscore characters that follow. It is delimited by a whitespace or another character not allowed in an identifier.

As a sequence of characters, a *keyword* is just an identifier in this language, so the scanner does not distinguish between two categories. The lexical analyzer subsequently separates out keywords by checking each identifier against the list of keywords.

An *integer literal* begins with a numeric character and includes all contiguous numeric characters that follow. It is delimited by a whitespace or nonnumeric character.

We plan to extend this language with additional operators. To enable flexible use of the scanner, we design it to collect all contiguous characters from a list of supported operator characters. Of course, we exclude alphabetic, numeric, underscore, parentheses, and similar characters from the list for the prefix ELI Calculator language.

The lexer subsequently compares each scanned operator against a list of valid operators to remove invalid operators.

The language uses keywords in similar ways to operators, so the lexer also subsequently tags keywords as operators. The current lexical analyzer does not use the TokKey token category.

The LexCalc module defines a Token algebraic data type, defined below, to represent the lexical tokens. The constructors identify the various syntactic categories.

```
import Values ( NumType, Name, toNumType )
-- e.g., NumType = Int , Name = String

data Token = TokLeft          -- left parenthesis
           | TokRight         -- right parenthesis
           | TokNum NumType   -- unsigned integer literal
           | TokId Name       -- names of variables, etc.
           | TokOp Name       -- names of primitive functions
           | TokKey Name      -- keywords (no use currently)
           | TokOther String  -- other characters
             deriving (Show, Eq)
```

The function lexx, shown below, incorporates the scanner and most of the lexeme evaluator functionality. It takes a string and returns a list of tokens.

```
import Data.Char ( isSpace, isDigit, isAlpha, isAlphaNum )

lexx :: String -> [Token]
lexx []  = []
lexx xs@(x:xs')
    | isSpace x   = lexx xs'
    | x == ';'    = lexx (dropWhile (/='\n') xs')
    | x == '('    = TokLeft  : lexx xs'
    | x == ')'    = TokRight : lexx xs'
    | isDigit x   = let (num,rest) = span isDigit xs
                    in (TokNum (convertNumType num)) : lexx rest
    | isFirstId x = let (id,rest) = span isRestId xs
                    in  (TokId id) : lexx rest
    | isOpChar x  = let (op,rest) = span isOpChar xs
                    in (TokOp op) : lexx rest
    | otherwise   = (TokOther [x]) : lexx xs'
    where
        isFirstId c = isAlpha c    || c == '_'
        isRestId  c = isAlphaNum c || c == '_'
        isOpChar  c = elem c opchars

opchars = "+-*/~<=>!&|@#$%^?:"  -- not " ' ` ( ) [ ] { } , . ;
```

Function lexx pattern matches on the first character of the string and then collects any additional characters of the token using the higher order function

`Data.Char.span`. Function `span` breaks the string into two part—the prefix consisting of all contiguous characters that satisfy its predicate and the suffix beginning with the first character that does not.

Boolean function `isOpChar` returns `True` for characters potentially allowed in operator symbols. These are defined in the string `opchars`{.haskell{, which makes this aspect of the scanner relatively easy to modify.

Function `lexer`, shown below, calls `lexx` and then carries out the following transformations on the list of tokens:

- `TokId` tokens for keywords are transformed into the corresponding `TokOp` tokens (as defined in association list `keywords {.haskell}`)
- `TokOp` tokens for valid operators (as defined in association list `opmap`) are transformed if needed and invalid operators are transformed into `TokOther` tokens

The lexer does not generate error messages. Instead it tags characters that do not fit in any lexeme as a `TokOther` token. The parser can use these as needed (e.g., to generate error messages).

```
lexer :: String -> [Token]
lexer xs = markSpecials (lexx xs)

markSpecials :: [Token] -> [Token]
markSpecials ts = map xformTok ts

xformTok :: Token -> Token
xformTok t@(TokId id)
    | elem id keywords   = TokOp id
    | otherwise          = t
xformTok t@(TokOp op)
    | elem op primitives = t
    | otherwise          = TokOther op
xformTok t               = t

keywords   = []                -- none defined currently
primitives = ["+","-","*","/"]
```

In the above code, the function `xformTok` transforms any identifier that is a defined keyword into an operator token, leaves other identifiers and defined primitive operators alone, and marks everything else with the token type `TokOther`.

44.3.2 Infix syntax

The lexer for the prefix syntax given in the previous subsection can also be used for the simple infix syntax. However, future extensions of the language may require differences in the lexers.

44.4 Recursive Descent Parsing

A *recursive descent parser* is an approach to parsing languages that has relatively simple grammars [78,196].

It is a *top-down parser*, a type of parser that begins with start symbol of the grammar and seeks to determine the parse tree by working down the levels of the parse tree toward the program (i.e., sentence).

By contrast, a *bottom-up* parser first recognizes the low-level syntactic units of the grammar and builds the parse tree upward from these leaves toward the root (i.e., start symbol). Bottom-up parsers support a wider range of grammars and tend to be more efficient for production compilers. However, their development tends to be less intuitive and more complex. We leave discussion of these parsers to courses on compiler construction.

A recursive descent parser consists of a set of mutually recursive functions. It typically includes one hand-coded function for each nonterminal of the grammar and one clause for each production for that nonterminal.

The recursive descent approach works well when the grammar can be transformed into an LL(k) (especially LL(1)) grammar [195]. Discussion of these techniques are left to courses on compiler construction.

For an LL(1) grammar, we can write recursive descent parsers that can avoid backtracking to an earlier point in the parse to start down another path.

For example, consider a simple grammar with with rules:

```
S ::= A | B
A ::= C D
B ::= { E }  -- zero or more occurrence of E
C ::= [ F ]  -- zero or one occurrence of F
D ::= '1' | '@' S
E ::= '3'
F ::= '2'
```

Consider the nonterminal `S`, which has alternatives `A` and `B`.

- Alternative `A` can begin with terminal symbols `1`, `2`, or `@`.
- Alternaive `B` can begin with terminal symbol `3` or be empty.

These sets of first symbols are disjoint, so the parser can distinguish among the alternatives based on the first terminal symbol. (Hence, the grammar is backtrack-free.)

44.4.1 Constructing recursive descent parsers

A simple recognizer for the grammar above could include functions similar to those shown below. We consider the five different situations for nonterminals `S`, `A`, `B`, `C`, and `E`.

In the Haskell code, a parsing function takes a `String` with the text of the expression to be processed and returns a tuple `(Bool,String)` where the first component indicates whether or not the parser succeeded (i.e., the output of the parse) and the second component gives the new state of the input.

If the first component is `True`, then the second component holds the input remaining after the parse. If the first component is `False`, then the second component is the remaining part of the input to be processed after the parser failed.

Of course, instead of strings, the parser could work on lists of tokens or other symbols.

1. Alternatives: `S ::= A | B`

   ```
   parseS :: String -> (Bool,String)  -- A | B
   parseS xs =
       case parseA xs of              -- try A
           (True,  ys) -> (True, ys)  -- A succeeds
           (False, _ ) ->
               case parseB xs of      -- else try B
                   (True, ys) -> (True,  ys) -- B succeeds
                   (False, _) -> (False, xs) -- both A & B fail
   ```

   Function `parseS` succeeds whenever any alternative succeeds. Otherwise, it continues to check subsequent alternatives. It fails if the final alternative fails.

   If there are more than two alternatives, we can nest each additional alternative more deeply within the conditional structure. (That is, we replace the `parseB` failure case value with a `case` expression for the third option. Etc.)

2. Sequencing: `A ::= C D`

   ```
   parseA :: String -> (Bool,String)  -- C D
   parseA xs =
       case parseC xs of              -- try C
           (True,  ys) ->
               case parseD ys of      -- then try D
                   (True, zs) -> (True,  zs) -- C D succeeds
                   (False, _) -> (False, xs) -- D fails
           (False, _ ) -> (False,xs)  -- C fails
   ```

   Function `parseA` fails whenever any component fails. Otherwise, it continues to check subsequent components. It succeeds when the final component succeeds.

   If there are more than two components in sequence, we nest each additional component more deeply within the conditional structure. (That is, we replace `parseD xs` with `case parseD xs of ....`)

3. Repetition zero or more times: `B ::= { E }`

   ```
   parseB :: String -> (Bool,String)   -- { E }
   parseB xs =
       case parseE xs of               -- try E
           (True,  ys) -> parseB ys    -- one E, try again
           (False, _)  -> (True,xs)    -- stop, succeeds
   ```

   Function `parseB` always succeeds if `parseE` terminates. However, it may succeed for zero occurrences of `E` or for some positive number of occurrences.

4. Optional elements: `C ::= [ F ]`

   ```
   parseC :: String -> (Bool,String)   -- [ F ]
   parseC xs =
       case parseF xs of               -- try F
           (True,  ys) -> (True,ys)
           (False, _ ) -> (True,xs)
   ```

   Function `parseC` always succeeds if `parseF` terminates. However, it may succeed for at most one occurrence of `F`.

5. Base cases to parse low-level syntactic elements: `E ::= '3'`

   ```
   parseE :: String -> (Bool,String)
   parseE (x:xs') = (x == '3', xs')
   parseE xs      = (False,    xs )
   ```

On success in any of these cases, the new input state is the string remaining after the successful alternative.

On failure, the input state should be left unchanged by any of the functions.

To use the above templates, it may sometimes be necessary to refactor the rules that involve more than one of the above cases. For example, consider the rule

```
D ::= '1' | '@' S
```

which consists of two alternatives, the second of which is itself a sequence. To see how to apply the templates straightforwardly, we can refactor `D` to be the two rules:

```
D  ::= '1' | DS
DS ::= '@' S
```

In addition to the above parsers for the various rules, we might have a function `parse` that calls the top-level parser (`parseS`) and ensures that all the input is parsed.

```
parse :: String -> Bool
parse xs =
    case parseS xs of
```

```
      (True,  []) -> True
      (_,      _ ) -> False
```

See file ParserS03.hs for experimental Haskell code for this example recursive descent parser.

To have a useful parser, the above prototype functions likely need to be modified to build the intermediate representation and to return appropriate error messages for unsuccessful parses.

The above prototype functions use Haskell, but a similar technique can be used with any language that supports recursive function calls.

44.4.2 Prefix syntax

This subsection describes an example recursive descent parser for the ELI Calculator language's prefix syntax. The complete code for the ParsePrefixCalc module is given in the file ParsePrefixCalc.hs.

As given in Chapter 41, the prefix parser embodies the the following grammar:

```
<expression> ::=  <var> | <val> | <operexpr>
<var>        ::=  <id>
<val>        ::=  [ '-' ] <unsigned>
<operexpr>   ::=  '(' <operator> <operandseq> ')'
<operandseq> ::=  { <expression> }
<operator>   ::=  '+'  |  '*'  |  '-'  |  '/' | ...
```

The ParserPrefixCalc module imports and uses the LexCalc module for lexical analysis. In particular, it uses the algebraic data type Token, types NumType and Name, and function lexer.

```
import Values ( NumType, Name, toNumType )

data Token = TokLeft            -- left parenthesis
           | TokRight           -- right parenthesis
           | TokNum NumType     -- unsigned integer literal
           | TokId Name         -- names of variables, etc.
           | TokOp Name         -- names of primitive functions
           | TokKey Name        -- keywords
           | TokOther String    -- other characters
             deriving (Show, Eq)

lexer :: String -> [Token]
```

For the prefix grammar above, the nonterminals <id> and <unsigned> and the terminals are parsed into their corresponding tokens by the lexical analyzer.

TODO: Update this code and reference. The incomplete module TestPrefix06 (in file TestPrefix06.hs{type="text/plain") provides some testing of the prefix

parser.

The output of the parser is an abstract syntax tree constructed with the algebraic data type `Expr` defined in the previous chapter. This is in the Abstract Syntax module.

```haskell
import Values ( ValType, Name )

data Expr = Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Var Name
          | Val ValType
```

44.4.2.1 Parse `<expression>` Now let's build a recursive descent parser using the method described in the previous subsection. We begin with the start symbol `<expression>`.

The parsing function `parseExpression`, shown below, implements the following BNF rule:

```
<expression> ::= <var> | <val> | <operexpr>
```

It uses the recursive descent template #1 with three alternatives.

```haskell
type ParErr = String

parseExpression :: [Token] -> (Either ParErr Expr, [Token])
parseExpression xs =
    case parseVar xs of
        r@(Right _, _) -> r  -- <var>
        _ ->
          case parseVal xs of
              r@(Right _, _) -> r  -- <val>
              _ ->
                  case parseOperExpr xs of
                      r@(Right _, _) -> r  -- <operexpr>
                      (Left m, ts)  -> (missingExpr m ts, ts)

missingExpr m ts  =
    Left ("Missing expression at  " ++ (showTokens (pref ts))
           ++ "..\n..Nested error { " ++ m ++ " }")
```

Function `parseExpression` takes a `Token` list and attempts to parse an `<expression>`. If the parse succeeds, the function returns a pair consisting of the `Right` value of an `Either` wrapping the corresponding `Expr` abstract syntax tree and the list of input `Token`s remaining after the `Expr`. If the parse

fails, then the function returns an error in a Left value for theEither and the unchanged list of input Tokens.

We define an auxiliary function missingExpr to generate an appropriate error message.

The function parse, shown below, is the primary entry point for the ParsePrefixCalc module. It first calls the lexical analysis function lexer (from the module LexCalc) on the input list of characters and then calls the parsing function parseExpression with the corresponding list of tokens.

If a parsing error occurs or if there are leftover tokens, then the function returns an appropriate error message.

```
parse :: String -> Either ParErr Expr
parse xs =
    case lexer xs of
        [] -> incompleteExpr xs
        ts ->
            case parseExpression ts of
               (ex@(Right _), []) -> ex
               (ex@(Left  _), _ ) -> ex
               (ex, ss)           -> extraAtEnd ex ss

incompleteExpr xs =
    Left ("Incomplete expression: " ++ xs)

extraAtEnd ex xs =
   Left ("Nonspace token(s) \"" ++ (showTokens xs) ++
         "\" at end of the expression \"" ++ (show ex) ++ "\"")
```

44.4.2.2 Parse <var> Function parseVar implements the BNF rule:

```
<var> ::= <id>
```

Variable <id> denotes an identifier token recognized by the lexer. So we implement function parseVar as a base case of the recursive descent parser (i.e., template #5).

```
parseVar :: [Token] -> (Either ParErr Expr, [Token])
parseVar ((TokId id):ts) = (Right (Var id),ts)
parseVar ts              = (missingVar ts, ts)

missingVar ts =
    Left ("Missing variable at " ++ (showTokens (pref ts)))
```

Function parseVar has the same type signature as parseExpression. It attempts to match an identifier token at the front of the token sequence. If it finds an identifier, it transforms the token to a Var expression and returns it with the

remaining token list. Otherwise, it returns an error message and the unchanged token list.

44.4.2.3 Parse `<val>`

Function `parseVal` implements the BNF rule:

```
<val> ::= [ '-' ] <unsigned>
```

To implement this rule, we can refactor it into two rules that correspond to the recursive descent template functions:

```
<val>      ::= <optminus> <unsigned>
<optminus> ::= [ '-' ]
```

Then `<val>` can be implemented using the sequencing (#2) prototype, `<optminus>` using the optional element (#4) prototype, and `<unsigned>` and `-` using base case (#5) prototypes.

However, `<unsigned>` denotes a numeric token and `-` denotes a single operator token. Thus we can easily implement `parseVal` as a base case of the recursive descent parser.

```
parseVal :: [Token] -> (Either ParErr Expr, [Token])
parseVal ((TokNum i):ts)             = (Right (Val i), ts)
parseVal ((TokOp "-"):(TokNum i):ts) = (Right (Val (-i)), ts)
parseVal ts                          = (missingVal ts, ts)

missingVal ts =
    Left ("Missing value at " ++ (showTokens (pref ts)))
```

Function `parseVal` has the same type signature as `parseExpression`. It attempts to match a numeric token, which is optionally preceded by a negative sign, at the front of the token sequence. If it finds this, it transforms the tokens to a `Val` expression and returns the expression and the remaining token list. Otherwise, it returns an error message and the unchanged token list.

44.4.2.4 Parse `<operexpr>`

Function `parseOperExpr` implements following BNF rule:

```
<operexpr>   ::=  "(" <operator> <operandseq> ")"
```

It uses a modified version of recursive descent template #2 for sequences of terms.

```
parseOperExpr :: [Token] -> (Either ErrMsg Expr, [Token])
parseOperExpr xs@(TokLeft:(TokOp op):ys) =  -- ( <operator>
    case parseOperandSeq ys of              -- <operandseq>
        (args, zs) ->
            case zs of                      -- )
              (TokRight:zs') -> (makeExpr op args, zs')
              zs'            -> (missingRParen zs, xs)
```

```haskell
-- ill-formed <operexpr>s
parseOperExpr (TokLeft:ts)       = (missingOp ts, ts)
parseOperExpr (TokRight:ts)      = (invalidOpExpr ")", ts)
parseOperExpr ((TokOther s):ts) = (invalidOpExpr s, ts)
parseOperExpr ((TokOp op):ts)   = (invalidOpExpr op, ts)
parseOperExpr ((TokId s):ts)    = (invalidOpExpr s, ts)
parseOperExpr ((TokNum i):ts)   = (invalidOpExpr (show i), ts)
parseOperExpr []                 = (incompleteExpr, [])

missingRParen ts =
    Left ("Missing `)` at " ++ (show (take 3 ts)))
missingOp ts     =
    Left ("Missing operator at " ++ (show (take 3 ts)))
invalidOpExpr s  =
    Left ("Invalid operation expression beginning with " ++ s)
incompleteExpr   = Left "Incomplete expression"
```

Function `parseOperExpr` has the same type signature as `parseExpression`. It directly matches against the first two tokens to see whether they are a left parenthesis and an operator, respectively, rather than calling separate functions to parse each. If successful, it then parses zero or more operands and examines the last token to see whether it is a right parenthesis.

If the operator expression is ill-formed, the function returns an appropriate error message.

The function `parseOperExpr` delegates the construction of the corresponding `Expr` (i.e., abstract syntax tree) to function `makeExpr`, which we discuss later in the subsection.

The values yielded by the components of `<operexpr>` must be handled differently than the previous components of expressions we have examined. They are not themselves `Expr` values.

- `(` and `)` denote the structure of the expression but do not have any output.
- `<operator>` does not itself yield a complete `Expr`. It must be combined with some number of operands to yield an expression. The number varies depending upon the particular operator. We pass a string to `makeExpr` to denote the operator.
- `<operandseq>` yields a possibly empty list of `Expr` values. We pass an `Expr` list to `makeExpr` to denote the operands. <

44.4.2.5 Parse `<operandseq>`

Function `parseOperandSeq` implements the BNF rule:

```
<operandseq> ::=  { <expression> }
```

It uses the recursive descent template #3 for repeated symbols.

```haskell
parseOperandSeq :: [Token] -> ([Expr],[Token])
parseOperandSeq xs =
    case parseExpression xs of
        (Left  _,       _ ) -> ([],xs)
        (Right ex, ys) ->
            let (exs,zs) = parseOperandSeq ys
            in  (ex:exs,zs)
```

The function `parseOperandSeq` takes a token list and collects a list of 0 or more operand `Expr`s. An empty list means that no operands were found.

44.4.2.6 AST construction (makeExpr) Operators in the current abstract syntax take a fixed number of operands. `Add` and `Mul` each take two operands, but a negation operator would take one operand and a conditional "if" operation would take three.

However, the current concrete prefix syntax does not distinguish among the different operators and the number of operands they require. It allows any operator in an `<operexpr>` to have any finite number of operands.

We could, of course, define a grammar that distinguishes among the operators, but we choose to keep the grammar flexible, thus enabling easy extension. We handle the operator-operand matching in the `makeExpr` function using data structures to define the mapping.

Thus, function `makeExpr` takes the operator string and a list of operand `Expr`s and constructs an appropriate `Expr`. It uses function `arity` to determine the number of operands required for the operator and then calls the appropriate `opConsN` function to construct the `Expr`.

```haskell
makeExpr :: String -> [Expr] -> Either ErrMsg Expr
makeExpr op exs =
    case arity op of
        0 -> opCons0 op exs  -- not implemented
        1 -> opCons1 op exs
        2 -> opCons2 op exs
        3 -> opCons3 op exs
        4 -> opCons4 op exs  -- not implemented
        5 -> opCons5 op exs  -- not implemented
        _ -> opConsX op exs  -- not implemented
```

Function `arity` takes an operator symbol and returns the number of operands that operator requires. It uses the `arityMap` association list to map the operator symbols to the number of arguments expected.

```haskell
import Data.Maybe

arityMap = [ ("+",2), ("-",2), ("*",2), ("/",2) ]
           -- add (operator,arity) pairs as needed
```

```
arity :: String -> Int
arity op = fromMaybe (-1) (lookup op arityMap)
```

Function `opCons2` takes a binary operator string and an operand list with two elements and returns the corresponding `Expr` structure wrapped in a `Right`. An error is denoted by passing back an error message wrapped in a `Left`.

```
assocOpCons2 =
    [ ("+",Add), ("-",Sub), ("*",Mul), ("/",Div) ]
     -- add new pairs as needed

opCons2 :: String -> [Expr] -> Either ParErr Expr
opCons2 op exs =
    case length exs of
        2 -> case lookup op assocOpCons2 of
                 Just c  -> Right (c (exs!!0) (exs!!1))
                 Nothing -> invalidOp op
        n -> arityErr op n

invalidOp op =
    Left ("Invalid operator '" ++ op ++ "'")
arityErr op n =
    Left ("Operator '" ++ op ++ "' incorrectly called with "
          ++ (show n) ++ " operand(s)")
```

Currently, the only supported operators are the binary operators `+`, `-`, `*`, and `/`. These map to the binary `Expr` constructors `Add`, `Sub`,`Mul`, and `Div`. (These are two-argument functions.)

If we extend the supported operators, then we must extend the definitions of `arityMap` and `assocOpCons2` and add new definitions for `opConsN` and `assocOpConsN` for other arities `N`. (We may also need to modify the `LexCalc` module and the definition of `Expr`.)

For now, we respond to unknown operators using function `opConsX` and return an appropriate error message. (In the future, this function may be redefined to support operators with variable numbers of operands.)

```
opConsX :: String -> [Expr] -> Either ErrMsg Expr
opConsX op exs = unsupportedOp op

unsupportedOp op = Left ("Unsupported operator '" ++ op ++ "'")
```

44.4.3 Infix syntax

TODO: Update the parser to reflect the grammar change and recursive descent explanation.

TODO: Describe the recursive descent infix parser in module ParseInfixCalc.hs.

An incomplete module that does some testing is TestInfix03.hs.

44.5 Exercises

TODO

44.6 Chapter Source Code

TODO

44.7 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

44.8 Terms and Concepts

TODO

45 Parsing Combinators

45.1 Chapter Introduction

TODO

45.2 Developing Parsing Combinators

In Chapter 44, we examined a set of prototype parsing functions and then used them as patterns for hand-coding of recursive descent parsing functions. We can benefit by generalizing these functions and collecting them into a library.

45.2.1 State actions and combinators

Consider `parseS`, one of the prototype parsing functions from a previous section. It parses the grammar rule `S ::= A | B`, which has two alternatives.

```
parseS :: String -> (Bool,String)
parseS xs =
    case parseA xs of                -- try A
        (True,  ys) -> (True, ys)   -- A succeeds
        (False, _ ) ->
            case parseB xs of       -- else try B
                (True, ys) -> (True,  ys) -- B succeeds
                (False, _) -> (False, xs) -- both A,B fail
```

Note that `parseS` and the other prototype parsing functions have the type:

```
String -> (Bool,String)
```

The occurrence of type `String` in the argument of the function represents the *state* of the input before evaluation of the function; the second occurrence of `String` represents the state after evaluation. The type `Bool` represents the *result* of the evaluation.

In an imperative program, the state is often left implicit and only the result type is returned. However, in a purely functional program, we must also make both the state change explicit.

Functions that have a type similar to `parseS` are called *state actions* or *state transitions*. We can generalize this parsing state transition as a function type:

```
type Parser a b = a -> (b,a)
```

In the case of `parseS`, we specialize this to:

```
Parser String Bool
```

In the case of richer parsing case studies for the prefix and infix parsers, we specialize this type as:

```
Parser [Token] (Either ErrMsg Expr)
```

Given the `Parser` type, we can define a set of *combinators* that allow us to combine simpler parsers to construct more complex parsers. These combinators can pass along the state implicitly, avoiding some tedious and repetitive work.

We can define a combinator `parseAlt` that generalizes the `parseS` prototype function above. It implements a recognizer, so we fix type `b` to `Bool`, but leave type argument `a` general.

```
parseAlt :: Parser a Bool -> Parser a Bool -> Parser a Bool
parseAlt p1 p2 =
    \xs ->
        case p1 xs of
            (True,  ys) -> (True, ys)
            (False, _ ) ->
                case p2 xs of
                    (True,  ys) -> (True,  ys)
                    (False, _ ) -> (False, xs)
```

Note the use of the anonymous function in the body. Function `parseAlt` takes two `Parser` values and then returns a `Parser` value. The `Parser` function returned binds in the two component function values. When this function is applied to the parser input (which is the argument of the anonymous function), it applies the two component parsers as needed.

We can easily redefine `parseS` in terms of the `parseAlt` combinator and simpler parsers `parseA` and `parseB`.

```
parseS = parseAlt parseA parseB
```

Given parsing input `inp`, we can invoke the parser with the expression:

```
parseS inp
```

Note that this formulation enables us to handle the passing of state among the component parsers implicitly, much as we can in an imperative computation. But it still preserves the nature of purely functional computation.

45.2.2 Completing a combinator library

Now consider the `parseA` prototype, which implements a two-component sequencing rule `A ::= C D`.

```
parseA xs =
    case parseC xs of    -- try C
        (True,  ys) ->   -- then try D
            case parseD ys of
                (True, zs) -> (True,  zs)   -- C D succeeds
                (False, _) -> (False, xs)   -- both C, D fail
        (False, _ ) -> (False,xs)   -- C fails
```

As with `parseS`, we can generalize `parseA` as a combinator `parseSeq`.

```
parseSeq :: Parser a Bool -> Parser a Bool -> Parser a Bool
parseSeq p1 p2 =
    \xs ->
        case p1 xs of
            (True,  ys) ->
                case p2 ys of
                    t@(True, zs) -> t
                    (False,  _ ) -> (False, xs)
            (False, _ ) -> (False,  xs)
```

Thus we can redefine `parseA` in terms of the `parseSeq` combinator and simpler parsers `parseC` and `parseD`.

```
parseA = parseSeq parseC parseD
```

Similarly, we consider the `parseB` prototype, which implements a repetition rule `B ::= { E }`.

```
parseB xs =
    case parseE xs of              -- try E
        (True,  ys) -> parseB ys   -- try again
        (False, ys) -> (True,xs)   -- stop
```

As above, we generalize this as combinator `parseStar`.

```
parseStar :: Parser a Bool -> Parser a Bool
parseStar p1 =
    \xs ->
        case p1 xs of
            (True,  ys) -> parseStar p1 ys
            (False, _ ) -> (True, xs)
```

We can redefine `parseB` in terms of combinator `parseStar` and simpler parser `parseE`.

```
parseB = parseStar parseB
```

Finally, consider parsing prototype `parseC`, which implements an optional rule `C ::= [ F ]`.

```
parseC xs =
   case parseF xs of              -- try F
        (True,  ys) -> (True,ys)
        (False, _ ) -> (True,xs)
```

We generalize this pattern as `parseOpt`, as follows.

```
parseOpt :: Parser a Bool -> Parser a Bool
parseOpt p1 =
    \xs ->
        case p1 xs of
```

```
                (True,  ys) -> (True, ys)
                (False, _ ) -> (True, xs)
```

We can thus redefine `parseC` in terms of simpler parser `parseF` and combinator `parseOpt`.

```
    parseC = parseOpt parseF
```

In this simple example grammar, function `parseD` is a simple instance of a sequence and `parseE` and `parseF` are simple parsers for symbols. These can be directly implemented as basic parsers, as before. However, the technique work if these are more complex parsers built up from combinators.

For convenience and completeness, we include extended alternative and sequencing combinators and parsers that always fail or always succeed.

```
    parseAltList :: [Parser a Bool] -> Parser a Bool
    parseSeqList :: [Parser a Bool] -> Parser a Bool
    parseFail, parseSucceed :: Parser a Bool
```

The combinators in this library are in the Haskell module ParserComb.hs. A module that does some testing is TestParserComb.hs.

TODO: Update and document the Parser Combinator library code.

45.2.3 Adding parse tree generations

TODO: Expand this library to allow returns of "parse trees" and error messages.

45.3 Standard libraries for parsing

TODO

There are a number of relatively standard parsing combinator libraries—e.g., the library Parsec. Readers who wish to develop other parsers may want to study that library.

45.4 Exercises

TODO

45.5 Chapter Source Code

TODO

45.6 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I developed the parsing combinators in this chapter primarily using the approach of Fowler and Parsons [78], with some influence by Chiusano and Bjarnason [29]. I generalized the concrete parsing functions from Chapter 44 to construct the combinators.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

45.7 Terms and Concepts

TODO

46 Calculator: Compilation

46.1 Chapter Introduction

This is a partially developed chapter.

TODO: - Add goals to intro. - Complete and revise the conditional expression sections as needed (e.g., the compilation subsection does not discuss the handling of labels/addresses sufficiently) - Consider adding separate compilation units and linking of units together

46.2 Stack Virtual Machine

Consider a stack virtual machine [[200]} as a means for executing the ELI Calculator language expressions. The operation of this machine is similar to the operation of a calculator that uses Reverse Polish Notation [201] (or postfix notation) such as the calculators from Hewlett-Packard.

46.2.1 Instruction set syntax

Consider a stack-based virtual machine with a symbolic instruction set defined by the following abstract syntax:

```
data SInstr = SVal Int
            | SVar String
            | SPop
            | SSwap
            | SDup
            | SAdd
            | SMul
              deriving (Show, Eq)
```

46.2.2 Instruction set semantics

Suppose the state of the virtual machine consists an *evaluation stack* of values and a program counter indicating the next instruction to be executed. Further suppose the above instructions have the following semantics. The machine executes much like a calculator that uses "reverse Polish notation".

- `SVal i` pushes value `i` onto the top of the evaluation stack.
- `SVar v` pushes the value of "variable" `v` from the current environment onto the top of the evaluation stack. (Here we are simulating a memory with the environment.)
- `SPop` removes the top element from the stack. (That is, if the stack from the top is `10:xs`, then the resulting stack is `xs`.)
- `SSwap` exchanges the top two elements on the stack. (That is, if the stack from the top is `10:20:xs`, then the resulting stack is `20:10:xs`.)

- SDup pushes another copy of the top element onto the stack. (That is, if the stack from the top is 10:xs, then the resulting stack is 10:10:xs.)
- SAdd pops the top two elements from the stack, adds the second to the first, and pushes the result back on top of the stack. (That is, if the stack from the top is 10:20:xs then the resulting stack is 30:xs.)
- SMul pops the top two elements from the stack, multiplies the second times the first, and pushes the result back on top of the stack. (That is, if the stack from the top is 10:20:xs then the resulting stack is 200:xs.)

We extend this instruction set later to provide other operations.

46.2.3 Machine execution

We can define a simple skeletal execution mechanism for the Stack Virtual Machine as follows. Function execSInstr takes the state, environment, and instruction and returns the modified state and environment. (This version does not modify the environment, but a version in the future may do so.)

```
data SState = SState [Int] Int
              deriving (Show, Eq)

execSInstr :: SState -> Env -> SInstr -> (SState, Env)
execSInstr (SState es pc) env (SVal i) =
    (SState (i:es) (pc+1), env)
execSInstr (SState es pc) env (SVar v) =
    case lookup v env of
        Just i  -> (SState (i:es) (pc+1), env)
        Nothing -> error ("Variable " ++ show v ++ " undefined")
execSInstr (SState es pc) env SPop  =
    (SState es pc, env) -- REPLACE
execSInstr (SState es pc) env SSwap =
    (SState es pc, env) -- REPLACE
execSInstr (SState es pc) env SDup  =
    (SState es pc, env) -- REPLACE
execSInstr (SState es pc) env SAdd  =
    case es of
        (r:l:xs) -> (SState ((l+r):xs) (pc+1), env)
        _        -> error ("Cannot Add. Stack too short: " ++ show es)
execSInstr (SState es pc) env SMul = (SState es pc, env) -- REPLACE
```

46.2.4 Compilation

We can translate the ELI Calculator language to the instruction set as follows. We call this process *code generation* and call the whole process of converting from source code to the instruction set *compilation*.

We consider compilation of the Calculator langauge to the stack virtual machine in Exercise Set A.

TODO: Does reference [88] fit here?

46.3 What Next?

TODO

46.4 Chapter Source Code

The source code module for this section is in file SInstr-2.hs.

46.5 Exercise Set A

In this exercise set, we consider the Stack Virtual Machine and translation of the ELI Calculator language's abstract syntax trees to equivalent sequences of instructions.

1. Complete the development of the function `execSInstr`, adding the code for the `SPop`, `SSwap`, `SDup`, and `SMul` instructions.

2. Extend the Stack Virtual Machine instruction set (i.e., `SInstr`) to support the extensions to the `Expr` data type defined in Exercise Set A (i.e., `Sub`, `Div`, `Neg`, `Min`, and `Max`). The operators take top value as their *right* operands and the value under that as the *left* operand.

3. Develop a Haskell function

   ```
   execSeq :: SState -> Env -> [SInstr] -> (SState, Env)
   ```

   that executes a sequence of Stack Virtual Machine instructions given the initial state and environment. (Although the machine in this case study so far does not modify the environment, allow for the future possibility of modification. A later exerces may extend the ELI Calculator language to add assignment statements, imperative loops, and variable and function declarations.)

   Also develop a function `exec` that executes a sequence of instructions from an initially empty stack with the given environment and returns the result on top of the stack after execution. (You may use `execSeq`.)

   ```
   exec :: Env -> [SInstr] -> Int
   ```

4. Develop a Haskell function

   ```
   compile :: Expr -> [SInstr]
   ```

   that translates the extended expression tree from Exercise Set A to a sequence of Stack Virtual Machine instructions as extended in this exercise set.

5. Develop a Haskell function `compGo` that takes an expression tree, simplifies, compiles, and executes it using the given environment. You may use the functions `exec` and `compile` from the previous exercises.

```
compGo :: Env -> Expr -> Int
```

46.6 Conditional Expressions

Let's examine how to extend the ELI Calculator language to include comparisons and conditional expressions.

46.6.1 Extending the Calculator language

TODO: This was introduced as a operator in a previous chapter.

Suppose that we redefine `Expr` to include binary operators `Eq` (equality) and `Lt` (less-than comparison), logical unary operator `Not`, and the ternary conditional expression `If` (if-then-else).

```
data Expr = ...
          | Eq  Expr Expr
          | Lt  Expr Expr
          | Not Expr
          | If  Expr Expr Expr
          ...
            deriving Show
```

This extended language does not have Boolean values. We represent "false" by integer 0 and "true" by a nonzero integer, primarily by 1.

We express the semantics of the various ELI Calculator language expressions as follows:

- `Eq l r` evaluates to the value 1 if `l` and `r` have the same value and to 0 otherwise.
- `Lt l r` evaluates to the value 1 if the value of `l` is smaller then the value of `r` and to 0 otherwise.
- `Not i` evaluates to 1 if `i` is zero and evaluates to 0 if `i` is nonzero.
- `If c l r` first evaluates `c`; if `c` is nonzero, the `if` evaluates to the value of `l`; otherwise the `if` evaluates to the value of `r`.

46.6.2 Extending the stack virtual machine (UNFINISHED)

TODO: This discussion in the remainder of the Conditional Expression section is not complete! In particular, the discussion of labels/addresses must be clarified and expanded—probably changed.

Suppose we redefine SInstr, the Stack Virtual Machine to include the new instructions:

```
data SInstr = ...
            | SEq
            | SLt
            | SLnot
            | SLabel String
            | SGo    String
            | SIfZ   String
            | SIfNZ  String
             deriving (Show, Eq)
```

These Stack Virtual Machine instructions execute as follows:

- SEq pops the top two values from the stack; if the values are equal, it pushes a 1 onto the stack; otherwise, it pushes a 0. (For example, if the stack from the top is 3:4:xs, the resulting stack is 0:xs.)
- SLt pops the top two values from the stack; if the second value is smaller than the top value, it pushes a 1 onto the stack; otherwise, it pushes a 0. (For example, if the stack from the top is 3:4:xs, the resulting stack is 0:xs.)
- SLnot pops the top value from the stack; if the top is 0, it pushes 1 back onto the stack; if it is nonzero, it pushed 0 back onto the stack. (For example, if the stack from the top is 0:xs, the resulting stack is 1:xs. If the stack is 7:xs, then the result is 0:xs.)
- SLabel n does not change the stack. It is a pseudo-instruction to enable a jump to this point in the program using label n.
- SGo n makes the next instruction to be executed the one labelled n; it does not change the stack.
- SIfZ n pops the value from the top of the stack; if this value is zero, then the next instruction executed will be the one labelled n; otherwise the next instruction is the one following the SIfZ instruction.
- SIfNZ n pops the value from the top of the stack; if this value is nonzero, then it makes the next instruction executed the one labelled n; otherwise the next instruction is the one following the SIfNZ instruction.

46.6.3 Extending the compiler (UNFINISHED)

TODO

We can translate the expression

```
If (Eq (Var "x") (Val 1)) (Val 10) (Val 20)
```

to a sequence of Stack Virtual Machine instructions such as:

```
[ SVar "x", SVal 1, SEq, SIfZ "else", SVal 10, SGo "end",
  SLabel "else', SVal 20, SLabel "end" ]
```

Of course, each `If` needs a unique set of labels.

46.7 Exercise Set B (UNFINISHED)

TODO

1. Extend the `eval` function to support the `Eq`, `Lt`, `Not`, and `If` operators.
2. Extend the `simplify` function to support the `Eq`, `Lt`, `Not`, and `If` operators.
3. Extend the data type `Expr` and the `eval` function to support the other comparison operators `Ne` (not equal), `Le` (less or equal), `Gt` (greater than), and `Ge` (greater or equal) and the logical operators `And` and `Or`.
4. Extend the `simplify` function to support the comparison operators `Ne`, `Le`, `Gt`, and `Ge` and the logical operators `And` and `Or` added in the previous exercise.
5. (UNFINISHED) Extend the `execSInstr`, `execSeq`, and `exec` functions from Exercise Set C to include the new Stack Virtual Machine instructions.
6. (UNFINISHED) Extend the `compile` and `compileGo` functions from Exercise Set C to include support for `Eq`, `Lt`, and `Not`.
7. (UNFINISHED) Extend the `compile` and `compileGo` functions from the previous exercise to include expressions `Ne`, `Le`, `Gt`, `Ge`, `And`, `Or`, and `If`. Each of these may need to be translated to a sequence of Stack Virtual Machine instructions.

46.8 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

46.9 Terms and Concepts

TODO

47 Imperative Core Language (Future)

This will likely be more than one chapter.

47.1 Chapter Introduction

TODO

47.2 What Next?

TODO

47.3 Exercises

TODO

47.4 Acknowledgements

TODO

47.5 Terms and Concepts

TBD

48 Appendix I: Review of Relevant Mathematics

48.1 Chapter Introduction

Students studying from this textbook should already have sufficient familiarity with the relevant mathematical concepts from the usual prerequisite courses. However, they may need to relate the mathematics with the programming constructs in functional programming.

The goal of this chapter is to review the mathematical concepts of functions and a few other mathematical concepts used in these notes. The concept of function in functional programming corresponds closely to the mathematical concept of function.

TODO: Add discussion of logic needed for specification and statement of laws?

TODO: Add appropriate citations.

48.2 Natural Numbers and Ordering

Several of the examples in these notes use natural numbers.

For this study, we consider the set of *natural numbers* N to consist of 0 and the positive integers.

Inductively, $n \in N$ if and only if one of the following holds

- $n = 0$
- There exists $m \in N$ such that $m = S(n)$

where S is the *successor* function, which returns the next element.

Furthermore,

- No element is the successor of more one other natural number.
- 0 is not the successor of any natural number. That is, it is the least (base) element.

The natural numbers thus form a *totally ordered* set in conjunction with the binary relation $\leq$ (less or equal). That is, the relation $\leq$ satisfies the following properties on set N:

- $n \leq n$ for all $n \in N$ (*reflexivity*)
- $m \leq n$ and $n \leq m$ implies $m = n$ (*antisymmetry*)
- $m \leq n$ and $n \leq p$ implies $m \leq p$ (*transitivity*)
- Either $m \leq n$ or $n \leq m$ for all $m, n \in N$ (*trichotomy*)

It is also a *partial ordering* because it satisfies the first three properties above.

For all $m, n \in N$, we can define the other ordering relations in terms of $=$, $\neq$, and $\leq$ as follows:

- $m < n$ (less) to mean $m \leq n$ and $m \neq n$. We say that m is smaller (or simpler) than n.
- $m > n$ (greater) to mean $n \leq m$ and $n \neq m$. We say that m is larger (or more complex) than n.
- $m \geq n$ (greater or equal) to mean the same as $n \leq m$

48.3 Functions

As we have studied in mathematics courses, a *function* is a mapping from a set A into a set B such that each element of A is mapped into a unique element of B.

- The set A (on which f is defined) is called the *domain* of f.
- The set of all elements of B into which f maps elements of A is called the *range* (or *codomain*) of f, and is denoted by $f(A)$.

If f is a function from A into B, then we write:

$$f : A \to B$$

We also write the equation

$$f(a) = b$$

to mean that the *value* (or *result*) from *applying* function f to an element $a \in A$ is an element $b \in B$.

If a function

$$f : A \to B$$

and $A \subseteq A'$, then we say that f is a *partial function* from A' to B and a *total function* from A to B. That is, there are some elements of A' on which f may be undefined.

48.4 Recursive Functions

Informally, a *recursive function* is a function defined using recursion.

In computing science, *recursion* is a method in which an "object" is defined in terms of smaller (or simpler) "objects" of the same type. A recursion is usually defined in terms of a recurrence relation.

A *recurrence relation* defines an "object" x_n as some combination of zero or more other "objects" x_i for $i < n$. Here $i < n$ means that i is smaller (or simpler) than n. If there is no smaller object, then n is a base object.

For example, consider a recursive function to compute the sum s of the first n natural numbers.

We can define a recurrence relation for s with the following equations:

$s(n) = 0,\ if\ n = 0$
$s(n) = n + s(n-1),\ if\ n \geq 1$

For example, consider $s(3)$,

$s(3) = 3+s(2) = 3+(2+s(1)) = 3+(2+(1+s(0))) = 3+(2+(1+0)) = 6$

48.5 Mathematical Induction Natural Numbers

We can give two mathematical definitions of factorial, *fact* and *fact'*, that are equivalent for all natural number arguments.

We can define *fact* using the product operator as follows:

$fact(n) = \prod_{i=1}^{i=n} i$

We can also define the factorial function *fact'* with a *recursive* definition (or *recurrence relation*) as follows:

$fact'(n) = 1,\ if\ n = 0$
$fact'(n) = n\times\ fact'(n-1),\ if\ n \geq 1$

It is, of course, easy to see that the recurrence relation definition is equivalent to the previous definition. But how can we prove it?

To prove that the above definitions of the factorial function are equivalent, we can use *mathematical induction* over the natural numbers.

Mathematical induction: To prove a logical proposition $P(n)$ holds for any natural number n, we must show two things:

- For the *base case* $n = 0$, show that $P(0)$ holds.
- For the *inductive case* $n = m + 1$, show that, if $P(m)$ holds for some natural number m, then $P(m + 1)$ also holds.

The $P(m)$ assumption is called the *induction hypothesis*.

Now let's prove that the two definitions *fact* and *fact'* are equivalent.

Prove For all natural numbers n, $fact(n) = fact'(n)$.

Base case $n = 0$**.**

$fact(0)$

$=$ { definition of *fact* (left to right) }

$(\Pi i : 1 \leq i \leq 0 : i)$

$=$ { empty range for Π, 1 is the identity element of $\times$ }

1

$=$ { definition of *fact'* (first leg, right to left) }

fact'(0)

Inductive case $n = m + 1$**.**
Given induction hypothesis *fact*(m) = *fact'*(m), prove *fact*$(m+1)$ =*fact'*$(m+1)$.

fact'$(m+1)$

= { definition of *fact* (left to right) }

$(\Pi i : 1 \leq i \leq m+1 : i)$

= { $m+1 > 0$, so $m+1$ term exists, split it out }

$(m+1) \times (\Pi i : 1 \leq i \leq m : i)$

= { definition of *fact* (right to left) }

$(m+1)\times$ *fact*(m)

= { induction hypothesis }

$(m+1)\times$ *fact'*(m)

= { $m+1 > 0$, definition of *fact'* (second leg, right to left) }

fact'$(m+1)$

Therefore, we have proved *fact*(n) = *fact'*(n) for all natural numbers n. QED

In the inductive step above, we explicitly state the induction hypothesis and assertion we wish to prove in terms of a different variable name (m instead of n) than the original statement. This helps to avoid the confusion in use of the induction hypothesis that sometimes arises.

We use an equational style of reasoning. To prove that an equation holds, we begin with one side and prove that it is equal to the other side. We repeatedly "substitute equals for equal" until we get the other expression.

Each transformational step is justified by a definition, a known property of arithmetic, or the induction hypothesis.

The structure of this inductive argument closely matches the structure of the recursive definition of *fact'*.

What does this have to do with functional programming? Many of the functions we will define in these notes have a recursive structure similar to *fact'*. The proofs and program derivations that we do will resemble the inductive argument above.

Recursion, induction, and iteration are all manifestations of the same phenomenon.

48.6 Operations

A function

$\oplus : (A \times A) \rightarrow A$

is called a *binary operation on A*. We usually write binary operations in *infix* form:

$a \oplus a$'

We often call a two-argument function of the form

$\oplus : (A \times B) \rightarrow C$

a binary operation as well. We can write this two argument function in the equivalent *curried* form:

$\oplus : A \rightarrow (B \rightarrow C)$

The curried form shows a multiple-paramenter function in a form where the function takes the arguments one at a time, returning the resulting function with one fewer arguments.

Let $\oplus$ be a binary operation on some set A and x, y, and z be elements of A. We can define the following kinds of properties.

- Operation $\oplus$ is *closed* on A if and only if $x \oplus y \in A$ for any $x, y \in A$. That is, the operation is a total function on its domain.
- Operation $\oplus$ is *associative* if and only if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for $x, y, z \in A$.
- Operation $\oplus$ is *commutative* (also called *symmetric*) if and only if $x \oplus y = y \oplus x$ for $x, y \in A$.
- An element e of set A is
  - a *left identity* of $\oplus$ if and only if $e \oplus x = x$ for any $x \in A$
  - a *right identity* of $\oplus$ if and only if $x \oplus e = x$ for any $x \in A$
  - an *identity* of $\oplus$ if and only if it is both a left and a right identity.

  An identity of an operation is called a *unit* of the operation.
- An element z of set A is
  - a *left zero* of $\oplus$ if and only if $z \oplus x = z$ for any $x \in A$
  - a *right zero* of $\oplus$ if and only if $x \oplus z = z$ for any $x \in A$
  - a *zero* of $\oplus$ if and only if it is both a right and a left zero
- If e is the identity of $\oplus$ and $x \oplus y = e$ for some x and y, then
  - x is a *left inverse* of y
  - y is a *right inverse* of x.

  Elements x and y are inverses of each other if $x \oplus y = e = y \oplus x$.
- An element x of set A is *idempotent* if $x \oplus x = x$.

If all elements of A are idempotent with respect to $\oplus$, then $\oplus$ is called *idempotent*.

For example, the addition operation + on natural numbers is closed, associative, and commutative and has the identity element 0. It has neither a left or right zero element and the only element with a left or right inverse is 0. If we consider the set of all integers, then all elements also have inverses.

Also, the multiplication operation * on natural numbers (or on all integers) is closed, associative, and commutative and has identity element 1 and zero element 0. Only value 1 has a left or right inverse.

However, the subtraction operation on natural numbers is not closed, associative, or commutative and has neither a left nor right zero. The value 0 is subtraction's right identity, but subtraction has no left identity. Each element is its own right and left inverse. If we consider all integers, then the operation is also closed.

Also, the "logical and" and "logical or" operations are idempotent with respect to the set of Booleans.

48.7 Algebraic Structures

An *algebraic structure* consists of a set of values, a set of one or more operations on those values, and properties (or "laws") of the operation on the set. We can characterize algebraic structures by the operations and their properties on the set of values.

If we focus on a binary operation $\oplus$ on a set A, then we can define various algebraic structures based on their properties.

- If $\oplus$ is closed on A, then then $\oplus$ and A form a *magma*.
- A magma in which $\oplus$ is an *associative* operation forms a *semigroup*.
- A semigroup in which $\oplus$ has an *identity* element forms a *monoid*.
- A monoid in which every element of A has an inverse forms a *group*.
- A monoid in which $\oplus$ is *commutative* forms a *commutative monoid* (or *Abelian monoid*).
- A group in which $\oplus$ is *commutative* forms an *Abelian group*.

For example, addition on natural numbers forms a commutative monoid and on integers forms an Abelian group.

Note: Above we describe a few common *group-like* algebraic structures, that is, algebras with one operation and one set. If we consider two operations on one set (e.g. $\oplus$ on $\otimes$), then we have various *ring-like* algebraic structures. By adding other operations, we have various other kinds of algebraic structures. If we consider more than one set, then we moved from a *single-sorted* (or *first-order*) algebra to a *many-sorted* algebra.

48.8 Exercises

TODO: Add

48.9 Acknowledgements

I adapted and revised much of this work in Summer and Fall 2016 from Chapter 2 of my *Notes on Functional Programming with Haskell* [42].

In Summer and Fall 2017, I continued to develop this material as a part of Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* These are Chapter 1, Evolution of Programming Languages; Chapter 2, Programming Paradigms; Chapter 3, Object-based Paradigms; and Chapter 80, Review of Relevant Mathematics (this background chapter).

In Spring 2019, I expanded the discussion of algebraic structures a bit.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

48.10 Terms and Concepts

TODO: Add

References

[1] Harold Abelson and Gerald Jockay Sussman. 1996. *Structure and interpretation of computer programs (SICP)* (Second ed.). MIT Press, Cambridge, Massachusetts, USA. Retrieved from https://mitpress.mit.edu/sicp/

[2] Paul Ammann and Jeff Offutt. 2017. *Introduction to software testing*. Cambridge University Pres, Cambridge, UK.

[3] Andrew W. Appel. 1998. *Modern compiler implementation in ML*. Cambridge University Press, Cambridge, UK.

[4] Luis Atencio. 2021. *The joy of JavaScript*. Manning, Shelter Island, New York, USA.

[5] Richard H. Austing, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes. 1979. Curriculum '78: Recommendations for the undergraduate program in computer science: A report of the ACM curriculum committee on computer science. *Communications of the ACM* 22, 3 (1979).

[6] John Backus. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs (1977 Turing Award address). *Communications of the ACM* 21, 8 (1978), 613–641.

[7] Conrad Barski. 2011. *Land of Lisp: Learn to program in Lisp, one game at a time*. No Starch Press, San Francisco, California, USA.

[8] David Beazley. 2013. Python 3 metaprogramming (tutorial). Retrieved from http://www.dabeaz.com/py3meta/

[9] David Beazley and Brian K. Jones. 2013. *Python cookbook* (Third ed.). O'Reilly Media, Sebastopol, California, USA.

[10] Kent Beck. 2003. *Test-driven development: By example*. Addison-Wesley, Boston Massachusetts, USA.

[11] Kent Beck and Ward Cunningham. 1989. A laboratory for teaching object-oriented thinking. *ACM SIGPLAN Notices* 24, 10 (1989), 1–6.

[12] David Bellin and Susan Suchman Simone. 1997. *The CRC Card book*. Addison-Wesley, Boston, Massachusetts, USA.

[13] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[14] Richard Bird. 2015. *Thinking functionally with Haskell* (First ed.). Cambridge University Press, Cambridge, UK.

[15] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[16] Rex Black. 2007. *Pragmatic software testing*. Wiley, Indianapolis, Indiana, USA.

[17] BlackBox Framework Center. 2022. What is BlackBox/Component Pascal? Retrieved from https://blackboxframework.org/

[18] Edwin Brady. 2017. *Type-driven development with Idris.* Manning, Shelter Island, New York, USA.

[19] Edwin Brady. 2022. Idris: A language for type-driven development. Retrieved from https://www.idris-lang.org

[20] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[21] Timothy Budd. 1995. *Multiparadigm programming in Leda.* Addison-Wesley, Boston, Massachusetts, USA.

[22] Timothy Budd. 2000. *Understanding object-oriented programming with Java* (Updated ed.). Addison-Wesley, Boston, Massachusetts, USA.

[23] Timothy Budd. 2002. *An introduction to object oriented programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA. Retrieved from https://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/toc.pdf

[24] Rod M. Burstall, David B. MacQueen, and Donald T Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on Lisp and functional programming*, 136–143.

[25] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-oriented software architecture: A system of patterns.* Wiley, Hoboken, New Jersey, USA.

[26] William E. Byrd. 2009. Relational programming in miniKanren: Techniques, applications, and implementations. PhD thesis. Indiana University, Bloomington, Indiana, USA.

[27] William E. Byrd and Contributers. 2022. miniKanren,org. Retrieved from http://minikanren.org/

[28] K. Mani Chandy and Jayadev Misra. 1988. *Parallel program design: A foundation.* Addison-Wesley, Boston, Massachusetts, USA.

[29] Paul Chiusano and Runar Bjarnason. 2015. *Functional programming in Scala* (First ed.). Manning, Shelter Island, New York, USA.

[30] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala exercises, hints, and answers. Retrieved from https://github.com/fpinscala/fpinscala

[31] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala community guide and chapter notes. Retrieved from https://github.com/fpinscala/fpinscala/wiki

[32] James Church. 2015. *Learning Haskell data analysis.* Packt Publishing Ltd, Birmingham, UK.

[33] William F. Clocksin and Christopher S. Mellish. 2012. *Programming in Prolog: Using the ISO standard* (Fifth ed.). Springer, Berlin, Germany.

[34] Edward Cohen. 1990. *Programming in the 1990's: An introduction to the calculation of programs.* Springer, New York, New York, USA.

[35] Collins Learning. 2022. Collins english dictionary. Retrieved from https://www.collinsdictionary.com/us/dictionary/english/

[36] Steve Dekorte amd Contributers. 2022. Io: A programming language. Retrieved from https://iolanguage.org/

[37] James Coplien, Daniel Hoffman, and David Weiss. 1998. Commonality and variability in software engineering. *IEEE Software* 15, 6 (1998), 37–45.

[38] Iain D. Craig. 2007. *Object-oriented programming languages*. Springer, London, UK.

[39] H. Conrad Cunningham. 1989. The shared dataspace approach to concurrent computation: The Swarm programming model, notation, and logic. PhD thesis. Washington University, Department of Computer Science, St. Louis, Missouri, USA.

[40] H. Conrad Cunningham. 2006. *A programmer's introduction to predicate logic*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/PredicateLogicNotes/Programmers_Introduction_to_Predicate-Logic.pdf

[41] H. Conrad Cunningham. 2006. *Notes on program semantics and derivation*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/reports/umcis-1994-02.pdf

[42] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/Notes_FP_Haskell/Notes_on_Functional_Programming_with_Haskell.pdf

[43] H. Conrad Cunningham. 2017. CSci 450 fundamental concepts: Prototype-based programming paradigm. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/2016fall/notes/Fundamentals/01Fundamental450.html#prototype-based

[44] H. Conrad Cunningham. 2017. CSci 450 fundamental concepts: Prototype-based programming example. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/2016fall/notes/450lectureNotes.html#prototype

[45] H. Conrad Cunningham. 2018. Python 3 reflexive metaprogramming. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci658/notes/PythonMetaprogramming/Py3RefMeta.html

[46] H. Conrad Cunningham. 2019. *Notes on data abstraction.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DataAbstraction/DataAbstraction.html

[47] H. Conrad Cunningham. 2019. *Notes on modular design.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ModularDesign/ModularDesign.html

[48] H. Conrad Cunningham. 2019. *Recursion concepts and terminology: Scala version.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/RecursionStyles/Scala/RecursionStylesScala.html

[49] H. Conrad Cunningham. 2019. *Notes on Scala for Java programmers.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScaldFP/ScalaForJava/ScalaForJava.html

[50] H. Conrad Cunningham. 2019. *Functional data structures (Scala).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS03/FunctionalDS.html

[51] H. Conrad Cunningham. 2019. *Handling errors without exceptions (Scala).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS04/ErrorHandling.html

[52] H. Conrad Cunningham. 2019. *Object-oriented software development.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/OOSoftDev/OOSoftDev.html

[53] H. Conrad Cunningham. 2022. *Notes on domain-specific languages.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DSLs/NotesDSLs.html

[54] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[55] H. Conrad Cunningham, Yi Liu, and Pavalli Tadepalli. 2006. Framework design using function generalization: A binary tree traversal case study. In *Proceedings of the ACM SouthEast conference*, Melbourne, Florida, USA, 312–318.

[56] H. Conrad Cunningham, Yi Liu, and Jingyi Wang. 2010. Designing a flexible framework for a table abstraction. In *Data engineering: Mining, information, and intelligence*, Yupo Chan, John Talburt and Terry M. Talley (eds.). Springer, New York, New York, USA, 279–314.

[57] H. Conrad Cunningham and Pallavi Tadepalli. 2006. Using function generalization to design a cosequential processing framework. In *Proceedings of the 39th Hawaii international conference on system sciences (HICSS'06)*, IEEE, Kauai, Hawaii, USA.

[58] H. Conrad Cunningham and Jingyi Wang. 2001. Building a layered framework for the table abstraction. In *Proceedings of the ACM symposium on applied computing*, Las Vegas, Nevada, USA.

[59] H. Conrad Cunningham, Cuihua Zhang, and Yi Liu. 2004. Keeping secrets within a family: Rediscovering Parnas. In *Proceedings of the international conference on software engineering research and practice (SERP)*, CSREA Press, Las Vegas, Nevada, USA, 712–718.

[60] Evan Czaplicki. 2022. Elm: A delightful language for reliable web applications. Retrieved from https://elm-lang.org

[61] Nell Dale and Henry M. Walker. 1996. *Abstract data types: Specifications, implementations, and applications.* D. C. Heath, Lexington, Massachusetts, USA.

[62] Steve Dekorte. 2005. Io: A small programming language. In *Companion to the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA)*, San Diego, California, USA, 166–167.

[63] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18, 8 (1975), 453–457.

[64] Edsger W. Dijkstra. 1976. Updating a sequential file. In *A discipline of programming.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 117--122.

[65] Edsger W. Dijkstra and Wim H. J. Feijen. 1988. *A method of programming.* Addison-Wesley, Boston Massachusetts, USA.

[66] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate calculus and program semantics.* Springer, New York, New York, USA.

[67] Barry Dwyer. 1981. One more time—how to update a master file. *Communications of the ACM* 24, 1 (1981), 3–8.

[68] Elixir Team. 2022. Elixir. Retrieved from https://elixir-lang.org

[69] Gerard D. Everett and Raymond McLeod Jr. 2007. *Software testing: Testing across the entire software development life cycle.* Wiley, Hoboken, New Jersey, USA.

[70] Richard Feldman. 2020. *Elm in action.* Manning, Shelter Island, New York, USA.

[71] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Krishnamurthi Shriram. 2014. *How to design programs* (Second ed.). MIT Press, Cambridge, Massachusetts, USA. Retrieved from https://htdp.org/

[72] Matthias Felleisen, David Van Horn, and Conrad Barski. 2013. *Realm of Racket: Learn to program, one game at a time!* No Starch Press, San Francisco, California, USA.

[73] Anthony J. Field and Peter G. Harrison. 1988. *Functional programming*. Addison-Wesley, Boston, Massachusetts, USA.

[74] Paul A. Fishwick. 1995. *Simulation model design and execution: Building digital worlds*. Addison-Wesley, Boston, Massachusetts, USA.

[75] Michael Fogus and Chris Houser. 2011. *The joy of Clojure*. Manning, Shelter Island, New York, USA.

[76] Michael J. Folk, Bill Zoellick, and Greg Riccardi. 1998. *File structures: An ObjectOriented approach with C++* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA.

[77] Martin Fowler. 1999. *Refactoring: Improving the design of existing code* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[78] Martin Fowler and Rebecca Parsons. 2010. *Domain specific languages*. Addison-Wesley, Boston, Massachusetts, USA.

[79] Phil Freeman. 2017. *Purescript by example: Functional programming for the web*. Leanpub, Victoria, British Columbia, Canada. Retrieved from https://book.purescript.org/

[80] Daniel P. Friedman, William E. Byrd, Oleg Kiselolyov, and Jason Hemenn. 2018. *The reasoned schemer* (Second ed.). MIT Press, Cambridge, Massachusetts, USA.

[81] Daniel P. Friedman and Matthias Felleisen. 1995. *The little schemer* (Fourth ed.). MIT Press, Cambridge, Massachusetts, USA.

[82] Daniel P. Friedman and Matthias Felleisen. 1995. *The seasoned schemer* (Second ed.). MIT Press, Cambridge, Massachusetts, USA.

[83] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Boston, Massachusetts, USA.

[84] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The language and its implementation*. Addison-Wesley, Boston, Massachusetts, USA.

[85] David Gries. 1981. *Science of programming*. Springer, New York, New York, USA.

[86] David Gries and Fred B. Schneider. 1993. *A logical approach to discrete math*. Springer, New York, New York, USA.

[87] H. Conrad Cunningham Yi Liu and Cuihua Zhang. 2006. Using classic problems to teach Java framework design. *Science of Computer Programming* 59, 1-2 (2006), 147–169.

[88] C. L. Hamblin. 1962. Translation to and from Polish Notation. *The Computer Journal* 5, 3 (1962), 210–213. Retrieved from https://doi.org/10.1093/comjnl/5.3.210

[89] Haskell Organization. 2020. QuickCheck: Automatic checking of Haskell programs. Retrieved from https://hackage.haskell.org/package/QuickCheck

[90] Haskell Organization. 2021. HUnit: A unit testing framework for Haskell. Retrieved from https://hackage.haskell.org/package/HUnit

[91] Haskell Organization. 2021. Tasty: Modern and extensible testing framework. Retrieved from https://hackage.haskell.org/package/tasty

[92] Haskell Organization. 2022. Glasgow Haskell Compiler (GHC) user's guide. Retrieved from https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/

[93] Haskell Organization. 2022. Using GHCi: Chapter 3, GHC user's guide. Retrieved from https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html

[94] Rich Hickey. 2020. A history of Clojure. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 71 (2020), 1–46.

[95] Rich Hickey. 2022. The Clojure programming language. Retrieved from https://clojure.org/

[96] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12, 10 (1969), 576–580.

[97] Tony Hoare. 2009. Null references: The billion dollar mistake (presentation). Retrieved from https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[98] Robert R. Hoogerwoord. 1989. The design of functional programs: A calculational approach. PhD thesis. Eindhoven Technical University, Eindhoven, The Netherlands.

[99] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++*. Wiley, Indianapolis, Indiana, USA.

[100] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[101] Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21, 3 (1989), 359–411.

[102] Paul Hudak and Joseph H. Fasel. 1992. A gentle introduction to Haskell. *ACM SIGPLAN Notices* 27, 5 (May 1992), 1–52.

[103] Paul Hudak, John Peterson, Joseph H. Fasel, and Reuben Thomas. 2000. A gentle introduction to Haskell 98. Retrieved from https://www.haskell.org/tutorial/

[104] Roberto Ierusalimschy. 2013. *Programming in Lua* (Third ed.). Lua.org, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil.

[105] Roberto Ierusalimschy. 2016. *Programming in Lua* (Fourth ed.). Lua.org, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil.

[106] Ralph Johnson and Brian Foote. 1988. Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2 (1988), 22–35. Retrieved from http://www.laputan.org/drc/drc.html

[107] Anne Kaldewaij. 1990. *Programming: The derivation of algorithms.* Prentice Hall, New York, New York, USA.

[108] Samuel N. Kamin. 1990. *Programming languages: An interpreter-based approach.* Addison-Wesley, Boston, Massachusetts, USA.

[109] Paul H. J. Kelly. 1989. *Functional programming for loosely-coupled multiprocessors.* MIT Press, Cambridge, Massachusetts, USA.

[110] Steve Klabnik, Carol Nichols, and Conributers. 2019. *The Rust programming language* (Rust 2018th ed.). No Starch Press, San Francisco, California, USA. Retrieved from https://doc.rust-lang.org/book/

[111] Donald E. Knuth. 1974. Computer programming as an art. *Communications of the ACM* 17, 12 (1974), 667–673.

[112] Lasse Koskela. 2013. *Effective unit testing.* Manning, Shelter Island, New York, USA.

[113] Holger Krekel and Pytest-dev Team. 2022. pytest: Helps you write better programs. Retrieved from https://docs.pytest.org/

[114] Shriram Krishnamurthi. 2007. *Programming languages: Application and interpretation* (First ed.). Brown University, Department of Computer Science, Providence, Rhode Island, USA. Retrieved from http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/

[115] Shriram Krishnamurthi, Benjamin S. Lerner, and Joe Gibbs Politz. 2020. *Programming and programming languages* (Unmaintained final draft ed.). Brown University, Department of Computer Science, Providence, Rhode Island, USA. Retrieved from http://cs.brown.edu/courses/cs173/2012/book/

[116] LabLua, PUC-Rio. 2022. Lua: The programming language. Retrieved from https://www.lua.org/

[117] LabLua PUC-Rio. 2022. About Lua. Retrieved from https://www.lua.org/about.html

[118] Peter Linz. 2017. *Formal languages and automata* (Sixth ed.). Jones & Bartlett, Burlington, Massachusetts, USA.

[119] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[120] Simon Marlow (Ed.). 2010. Haskell 2010 language report. Retrieved from https://www.haskell.org/definition/haskell2010.pdf

[121] John McCarthy. 1978. History of LISP. *ACM SIGPLAN Notices* 8 (1978), 217–223. Retrieved from https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/PAPERS/mccarthy.pdf

[122] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. 1962. *LISP 1.5 programmer's manual*. MIT Press, Cambridge, Massachusetts, USA. Retrieved from https://apps.dtic.mil/sti/pdfs/AD0406138.pdf

[123] Julie McKeehan and Neil Rhodes. 1994. *Programming for the Newton: Software development with NewtonScript*. Morgan Kaufmann, Waltham, Massachusetts, USA.

[124] Tim McNamara. 2021. *Rust in action: Systems programming concepts and techniques*. Manning, Shelter Island, New York, USA.

[125] O'Reilly Media. 2004. History of programming languages poster. Retrieved from https://www.cs.toronto.edu/~gpenn/csc324/PLhistory.pdf

[126] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Addison-Wesley, Boston, Massachusetts, USA.

[127] Gerard Meszaros, Shaun M. Smith, and Jennitta Andrea. 2003. The test automation manifest. In *Proceedings of the conference on extreme programming and agile methods*, Springer, New Orleans, Louisiana, USA, 73–81. Retrieved from https://pdfs.semanticscholar.org/b42f/337557b91e5a4daa571fe19a8e937d9ac03d.pdf

[128] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[129] Hanspeter Mossenbock. 1995. *Object-oriented programming in Oberon-2*. Springer, Berlin, Germany.

[130] NewtonScript.org. 2022. NewtonScript. Retrieved from http://newtonscript.org/

[131] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (First ed.). Artima, Inc., Walnut Creek, California, USA.

[132] Martin Odersky, Lex Spoon, and Bill Venners. 2021. *Programming in Scala* (Fifth ed.). Artima, Inc., Walnut Creek, California, USA.

[133] Brian Okken. 2017. *Python testing with pytest: Simple, rapid, effective, and scalable* (First ed.). Pragmatic Bookshelf, Raleigh, North Carolina, USA.

[134] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[135] David L. Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976), 1–9.

[136] David L. Parnas. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* SE-5, 1 (1979), 128–138.

[137] David L. Parnas. 1979. The modular structure of complex systems. *IEEE Transactions on Software Engineering* SE-11, 13 (1979), 128–138.

[138] David L. Parnas. 2001. Some software engineering principles. In *Software fundamentals: Collected papers by David L. Parnas*, Daneil M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.

[139] William E. Perry. 2006. *Effective methods for software testing.* Wiley, Indianapolis, Indiana, USA.

[140] PLT Inc. 2022. Racket. Retrieved from https://www.racket-lang.org

[141] George Polya. 1957. *How to solve it: A new aspect of mathematical method* (Second ed.). Princeton Unversity Press.

[142] George Polya. 1981. *Mathematical discovery: On understanding, learning, and teaching problem solving* (Combined ed.). Wiley, Hoboken, New Jersey, USA.

[143] purescript.org. 2022. PureScript: A strongly-typed functional programming language that compiles to javascript. Retrieved from https://www.purescript.org/

[144] Python Software Foundation. 2022. Python. Retrieved from https://www.python.org/

[145] Christian Queinnec. 2003. *Lisp in small pieces.* Cambridge University Press, Cambridge, UK.

[146] Luciano Ramalho. 2013. *Fluent Python: Clear, concise, and effective programming.* O'Reilly Media, Sebastopol, California, USA.

[147] ramdajs.com. 2022. Ramda: A practical functional library for JavaScript programmers. Retrieved from https://ramdajs.com/

[148] Normam Ramsey and Samuel L. Kamin. 2013. *Programming languages: Build, prove, and compare* (Draft ed.). Cambridge University Press, Cambridge, UK.

[149] Ruby Community. 2022. Ruby: A programmer's best friend. Retrieved from https://www.ruby-lang.org

[150] Rust Team. 2022. Rust: A language empowering everyone to build reliable and efficient software. Retrieved from https://www.rust-lang.org/

[151] Scala Language Organization. 2022. The Scala programming language. Retrieved from https://www.scala-lang.org/

[152] Michel Schinz and Phillipp Haller. 2016. A Scala tutorial for Java. Retrieved from https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

[153] Hans Albrecht Schmid. 1996. Creating applications from components: A manufacturing framework design. *IEEE Software* 13, 6 (1996), 67–75.

[154] Hans Albrecht Schmid. 1997. Systematic framework design by generalization. *Communications of the ACM* 40, 10 (1997), 48–51.

[155] Hans Albrecht Schmid. 1999. Framework design by systematic generalization. In *Building application frameworks: Object-oriented foundations of framework design*, Mohamed E. Fayad, Douglas C. Schmidt and Ralph E. Johnson (eds.). Wiley, Hoboken, New Jersey, USA, 353--378.

[156] Michael L. Scott. 2015. *Programming language pragmatics* (Third ed.). Morgan Kaufmann, Waltham, Massachusetts, USA.

[157] Robert W. Sebesta. 1993. *Concepts of programming languages* (Second ed.). Benjamin/Cummings, Boston, Massachusetts, USA.

[158] SelfLanguage.org. 2022. Self: Fun through simplicity. Retrieved from https://selflanguage.org/

[159] Peter Sestoft. 2012. *Programming language concepts* (First ed.). Springer, London, UK.

[160] Peter Sestoft. 2017. *Programming language concepts* (Second ed.). Springer, London, UK.

[161] Software Testing Fundamentals (SFT). 2021. Software testing guide and tutorial. Retrieved from https://softwaretestingfundamentals.com/

[162] Source Making. 2022. Null object design pattern. Retrieved from https://sourcemaking.com/design_patterns/null_object

[163] SWI-Prolog Organization. 2022. SWI-Prolog: Robust, mature, free Prolog for the real world. Retrieved from https://www.swi-prolog.org/

[164] Bruce Tate. 2010. *Seven languages in seven weeks: A pragmatic guide to learning programming languages*. Pragmatic Bookshelf, Raleigh, North Carolina, USA.

[165] Bruce Tate, Ian Dees, Frederic Daoud, and Jack Moffitt. 2014. *Seven more languages in seven weeks: Languages that are shaping the future*. Pragmatic Bookshelf, Raleigh, North Carolina, USA.

[166] The JUnit Team. 2022. JUnit 5: The fifth major version of the programmer-friendly testing framework for Java and the JVM. Retrieved from https://junit.org/junit5/

[167] The R Foundation. 2022. R: The R project for statistical programming. Retrieved from https://www.r-project.org/

[168] Dave Thomas. 2018. *Programming Elixir >= 1.6: Functional |> concurrent |> pragmatic |> fun*. Pragmatic Bookshelf, Raleigh, North Carolina, USA.

[169] David Thomas, Chad Fowler, and Andrew Hunt. 2004. *Programming Ruby* (Second ed.). Pragmatic Bookshelf, Raleigh, North Carolina, USA.

[170] Pete Thomas and Ray Weedom. 1995. *Object-oriented programming in Eiffel*. Addison-Wesley, Boston, Massachusetts, USA.

[171] Simon Thompson. 1996. *Haskell: The craft of programming* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[172] Simon Thompson. 1999. *Haskell: The craft of programming* (Second ed.). Addison-Wesley, Boston, Massachusetts, USA.

[173] Simon Thompson. 2011. *Haskell: The craft of programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA.

[174] Catalin Tudose. 2020. *JUnit in action* (Third ed.). Manning, Shelter Island, New York, USA.

[175] David Ungar and Randall B. Smith. 1987. Self: The power of simplicity. In *Proceedings of the ACM conference on object-oriented programming systems, languages and applications (OOPSLA'87)*, Orlando, Florida, USA, 227–242.

[176] Stanley J. Warford. 2014. *Computing fundamentals: The theory and practice of software design with BlackBox Component Builder*. Creative Commons, Attribution-ShareAlike (CC BY-SA). Retrieved from https://cslab.pepperdine.edu/warford/ComputingFundamentals/

[177] David M. Weiss. 2001. Introduction: On the criteria to be used in decomposing systems into modules. In *Software fundamentals: Collected papers by David L. Parnas*, Daniel M. Hoffman and David M. Weiss (eds.). Addison-Wesley, Boston, Massachusetts, USA.

[178] E. Peter Wentworth. 1990. *Introduction to functional programming using RUFL*. Rhodes University, Department of Computer Science, Grahamstown, South Africa.

[179] Wikibooks: Open Books for the World. 2019. Haskell. Retrieved from https://en.wikibooks.org/wiki/Haskell

[180] Wikpedia: The Free Encyclopedia. 2022. History of programming languages. Retrieved from https://en.wikipedia.org/wiki/History_of_programming_languages

[181] Wikpedia: The Free Encyclopedia. 2022. R (programming language). Retrieved from https://en.wikipedia.org/wiki/R_(programming_language)

[182] Wikpedia: The Free Encyclopedia. 2022. Lisp (programming language). Retrieved from https://en.wikipedia.org/wiki/Lisp_(programming_language)

[183] Wikpedia: The Free Encyclopedia. 2022. Scheme (programming language). Retrieved from https://en.wikipedia.org/wiki/Scheme_(programming_language)

[184] Wikpedia: The Free Encyclopedia. 2022. Formal grammar. Retrieved from https://en.wikipedia.org/wiki/Formal_grammar

[185] Wikpedia: The Free Encyclopedia. 2022. Context-free grammar. Retrieved from https://en.wikipedia.org/wiki/Context-free_grammar

[186] Wikpedia: The Free Encyclopedia. 2022. Backus-naur form. Retrieved from https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[187] Wikpedia: The Free Encyclopedia. 2022. Extended backus-naur form. Retrieved from https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

[188] Wikpedia: The Free Encyclopedia. 2022. Linear grammar. Retrieved from https://en.wikipedia.org/wiki/Linear_grammar

[189] Wikpedia: The Free Encyclopedia. 2022. RegularGrammar. Retrieved from https://en.wikipedia.org/wiki/Regular_grammar

[190] Wikpedia: The Free Encyclopedia. 2022. Finite-state machine. Retrieved from https://en.wikipedia.org/wiki/Finite-state_machine

[191] Wikpedia: The Free Encyclopedia. 2022. Deterministic finite automaton. Retrieved from https://en.wikipedia.org/wiki/Deterministic_finite_automaton

[192] Wikpedia: The Free Encyclopedia. 2022. Pushdown automaton. Retrieved from https://en.wikipedia.org/wiki/Pushdown_automaton

[193] Wikpedia: The Free Encyclopedia. 2022. Parse tree. Retrieved from https://en.wikipedia.org/wiki/Parse_tree

[194] Wikpedia: The Free Encyclopedia. 2022. Parsing. Retrieved from https://en.wikipedia.org/wiki/Parsing

[195] Wikpedia: The Free Encyclopedia. 2022. LL parser. Retrieved from https://en.wikipedia.org/wiki/LL_parser

[196] Wikpedia: The Free Encyclopedia. 2022. Recursive descent parser. Retrieved from https://en.wikipedia.org/wiki/Recursive_descent_parser

[197] Wikpedia: The Free Encyclopedia. 2022. Abstract syntax. Retrieved from https://en.wikipedia.org/wiki/Abstract_syntax

[198] Wikpedia: The Free Encyclopedia. 2022. Abstract syntax tree. Retrieved from https://en.wikipedia.org/wiki/Abstract_syntax_tree

[199] Wikpedia: The Free Encyclopedia. 2022. Lexical analysis. Retrieved from https://en.wikipedia.org/wiki/Lexical_analysis

[200] Wikpedia: The Free Encyclopedia. 2022. Stack machine. Retrieved from https://en.wikipedia.org/wiki/Stack_machine

[201] Wikpedia: The Free Encyclopedia. 2022. Reverse polish notation. Retrieved from https://en.wikipedia.org/wiki/Reverse_Polish_notation

[202] Wikpedia: The Free Encyclopedia. 2022. Tail call. Retrieved from https://en.wikipedia.org/wiki/Tail_call

[203] Wikpedia: The Free Encyclopedia. 2022. Abstract data type. Retrieved from https://en.wikipedia.org/wiki/Abstract_data_type

[204] Wikpedia: The Free Encyclopedia. 2022. Algebraic data type. Retrieved from https://en.wikipedia.org/wiki/Algebraic_data_type

[205] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_prin ciple

[206] Wikpedia: The Free Encyclopedia. 2022. Polymorphism (computer science). Retrieved from https://en.wikipedia.org/wiki/Polymorphism _(computer_science)

[207] Wikpedia: The Free Encyclopedia. 2022. Function overloading. Retrieved from https://en.wikipedia.org/wiki/Function_overloading

[208] Wikpedia: The Free Encyclopedia. 2022. Ad hoc polymorphism. Retrieved from https://en.wikipedia.org/wiki/Ad_hoc_polymorphism

[209] Wikpedia: The Free Encyclopedia. 2022. Parametric polymophism. Retrieved from https://en.wikipedia.org/wiki/Parametric_polymorphism

[210] Wikpedia: The Free Encyclopedia. 2022. Subtyping. Retrieved from https://en.wikipedia.org/wiki/Subtyping

[211] Wikpedia: The Free Encyclopedia. 2022. Prototype-based programming. Retrieved from https://en.wikipedia.org/wiki/Prototype-based_progra mming

[212] Wikpedia: The Free Encyclopedia. 2022. Nullable type. Retrieved from https://en.wikipedia.org/wiki/Nullable_type

[213] Wikpedia: The Free Encyclopedia. 2022. Null object pattern. Retrieved from https://en.wikipedia.org/wiki/Null_object_pattern

[214] Wikpedia: The Free Encyclopedia. 2022. Option type. Retrieved from https://en.wikipedia.org/wiki/Option_type

[215] Wikpedia: The Free Encyclopedia. 2022. Association list. Retrieved from https://en.wikipedia.org/wiki/Association_list

[216] Wikpedia: The Free Encyclopedia. 2022. Associative array. Retrieved from https://en.wikipedia.org/wiki/Associative_array

[217] Wikpedia: The Free Encyclopedia. 2022. Gregorian calendar. Retrieved from https://en.wikipedia.org/wiki/Gregorian_calendar

[218] Wikpedia: The Free Encyclopedia. 2022. Proleptic Gregorian calendar. Retrieved from https://en.wikipedia.org/wiki/Proleptic_Gregorian_cale ndar

[219] Wikpedia: The Free Encyclopedia. 2022. ISO 8601. Retrieved from https://en.wikipedia.org/wiki/ISO_8601

[220] Wikpedia: The Free Encyclopedia. 2022. Roman numerals. Retrieved from https://en.wikipedia.org/wiki/Roman_numerals

[221] Wikpedia: The Free Encyclopedia. 2022. Equality (mathematics). Retrieved from https://en.wikipedia.org/wiki/Equality_(mathematics)

[222] Wikpedia: The Free Encyclopedia. 2022. Total order. Retrieved from https://en.wikipedia.org/wiki/Total_order

[223] Wikpedia: The Free Encyclopedia. 2022. Equivalence relation. Retrieved from https://en.wikipedia.org/wiki/Equivalence_relation

[224] Rebecca Wirfs-Brock and Alan McKean. 2003. *Object design: Roles, responsibilities, and collaborations*. Addison-Wesley, Boston, Massachusetts, USA.

[225] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. 1990. *Designing object-oriented software*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[226] Wolfram Research, Inc. 2022. Equal. Retrieved from https://mathworld.wolfram.com/Equal.html

[227] Wolfram Research, Inc. 2022. Equivalence relation. Retrieved from https://mathworld.wolfram.com/EquivalenceRelation.html

[228] Wolfram Research, Inc. 2022. Totally ordered set. Retrieved from https://mathworld.wolfram.com/TotallyOrderedSet.html

[229] Bobby Woolf. 1997. Null object. In *Pattern languages of program design 3*, Robert Martin, Dirk Riehle and Frank Buschmann (eds.). Addison-Wesley, Boston, Massachusetts, USA, 5–18.

[230] Brent Yorgey. 2009. The typeclassopedia. *The Monad.Reader* 12, 13 (2009), 17–68. Retrieved from https://wiki.haskell.org/Typeclassopedia

[231] Nicholas C. Zakas. 2014. *The principles of object-oriented JavaScript*. No Starch Press, San Francisco, California, USA.