Concepts, Techniques, and Models of Computer Programming

PETER VAN ROY[1]
Université catholique de Louvain (at Louvain-la-Neuve)
Swedish Institute of Computer Science

SEIF HARIDI[2]
Royal Institute of Technology (KTH)
Swedish Institute of Computer Science

June 5, 2003

---

[1] Email: `pvr@info.ucl.ac.be`, Web: `http://www.info.ucl.ac.be/~pvr`
[2] Email: `seif@it.kth.se`, Web: `http://www.it.kth.se/~seif`

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Contents

List of Figures xvi

List of Tables xxiv

Preface xxvii

Running the example programs xliii

I Introduction 1

1 Introduction to Programming Concepts 3

1.1 A calculator . . . 3
1.2 Variables . . . 4
1.3 Functions . . . 4
1.4 Lists . . . 6
1.5 Functions over lists . . . 9
1.6 Correctness . . . 11
1.7 Complexity . . . 12
1.8 Lazy evaluation . . . 13
1.9 Higher-order programming . . . 15
1.10 Concurrency . . . 16
1.11 Dataflow . . . 17
1.12 State . . . 18
1.13 Objects . . . 19
1.14 Classes . . . 20
1.15 Nondeterminism and time . . . 21
1.16 Atomicity . . . 23
1.17 Where do we go from here . . . 24
1.18 Exercises . . . 24

II General Computation Models 29

2 Declarative Computation Model 31

2.1 Defining practical programming languages 33
2.1.1 Language syntax 33
2.1.2 Language semantics 38
2.2 The single-assignment store 44
2.2.1 Declarative variables 44
2.2.2 Value store 44
2.2.3 Value creation 45
2.2.4 Variable identifiers 46
2.2.5 Value creation with identifiers 47
2.2.6 Partial values 47
2.2.7 Variable-variable binding 48
2.2.8 Dataflow variables 49
2.3 Kernel language 50
2.3.1 Syntax 50
2.3.2 Values and types 51
2.3.3 Basic types 53
2.3.4 Records and procedures 54
2.3.5 Basic operations 56
2.4 Kernel language semantics 57
2.4.1 Basic concepts 57
2.4.2 The abstract machine 61
2.4.3 Non-suspendable statements 64
2.4.4 Suspendable statements 67
2.4.5 Basic concepts revisited 69
2.4.6 Last call optimization 74
2.4.7 Active memory and memory management 75
2.5 From kernel language to practical language 80
2.5.1 Syntactic conveniences 80
2.5.2 Functions (the **fun** statement) 85
2.5.3 Interactive interface (the **declare** statement) 88
2.6 Exceptions 91
2.6.1 Motivation and basic concepts 91
2.6.2 The declarative model with exceptions 93
2.6.3 Full syntax 95
2.6.4 System exceptions 97
2.7 Advanced topics 98
2.7.1 Functional programming languages 98
2.7.2 Unification and entailment 100
2.7.3 Dynamic and static typing 106
2.8 Exercises 108

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

3 Declarative Programming Techniques 113

3.1 What is declarativeness? 117
  3.1.1 A classification of declarative programming 117
  3.1.2 Specification languages 119
  3.1.3 Implementing components in the declarative model 119
3.2 Iterative computation 120
  3.2.1 A general schema 120
  3.2.2 Iteration with numbers 122
  3.2.3 Using local procedures 122
  3.2.4 From general schema to control abstraction 125
3.3 Recursive computation 126
  3.3.1 Growing stack size 127
  3.3.2 Substitution-based abstract machine 128
  3.3.3 Converting a recursive to an iterative computation 129
3.4 Programming with recursion 130
  3.4.1 Type notation 131
  3.4.2 Programming with lists 132
  3.4.3 Accumulators 142
  3.4.4 Difference lists 144
  3.4.5 Queues 149
  3.4.6 Trees 153
  3.4.7 Drawing trees 161
  3.4.8 Parsing 163
3.5 Time and space efficiency 169
  3.5.1 Execution time 169
  3.5.2 Memory usage 175
  3.5.3 Amortized complexity 177
  3.5.4 Reflections on performance 178
3.6 Higher-order programming 180
  3.6.1 Basic operations 180
  3.6.2 Loop abstractions 186
  3.6.3 Linguistic support for loops 190
  3.6.4 Data-driven techniques 193
  3.6.5 Explicit lazy evaluation 196
  3.6.6 Currying 196
3.7 Abstract data types 197
  3.7.1 A declarative stack 198
  3.7.2 A declarative dictionary 199
  3.7.3 A word frequency application 201
  3.7.4 Secure abstract data types 204
  3.7.5 The declarative model with secure types 205
  3.7.6 A secure declarative dictionary 210
  3.7.7 Capabilities and security 210
3.8 Nondeclarative needs 213

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

- 3.8.1 Text input/output with a file 213
- 3.8.2 Text input/output with a graphical user interface 216
- 3.8.3 Stateless data I/O with files 219
- 3.9 Program design in the small 221
  - 3.9.1 Design methodology 221
  - 3.9.2 Example of program design 222
  - 3.9.3 Software components 223
  - 3.9.4 Example of a standalone program 228
- 3.10 Exercises 233

4 Declarative Concurrency **237**
- 4.1 The data-driven concurrent model 239
  - 4.1.1 Basic concepts 241
  - 4.1.2 Semantics of threads 243
  - 4.1.3 Example execution 246
  - 4.1.4 What is declarative concurrency? 247
- 4.2 Basic thread programming techniques 251
  - 4.2.1 Creating threads 251
  - 4.2.2 Threads and the browser 251
  - 4.2.3 Dataflow computation with threads 252
  - 4.2.4 Thread scheduling 256
  - 4.2.5 Cooperative and competitive concurrency 259
  - 4.2.6 Thread operations 260
- 4.3 Streams 261
  - 4.3.1 Basic producer/consumer 261
  - 4.3.2 Transducers and pipelines 263
  - 4.3.3 Managing resources and improving throughput 265
  - 4.3.4 Stream objects 270
  - 4.3.5 Digital logic simulation 271
- 4.4 Using the declarative concurrent model directly 277
  - 4.4.1 Order-determining concurrency 277
  - 4.4.2 Coroutines 279
  - 4.4.3 Concurrent composition 281
- 4.5 Lazy execution 283
  - 4.5.1 The demand-driven concurrent model 286
  - 4.5.2 Declarative computation models 290
  - 4.5.3 Lazy streams 293
  - 4.5.4 Bounded buffer 295
  - 4.5.5 Reading a file lazily 297
  - 4.5.6 The Hamming problem 298
  - 4.5.7 Lazy list operations 299
  - 4.5.8 Persistent queues and algorithm design 303
  - 4.5.9 List comprehensions 307
- 4.6 Soft real-time programming 309

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

4.6.1 Basic operations 309
4.6.2 Ticking 311
4.7 Limitations and extensions of declarative programming 314
4.7.1 Efficiency 314
4.7.2 Modularity 315
4.7.3 Nondeterminism 319
4.7.4 The real world 322
4.7.5 Picking the right model 323
4.7.6 Extended models 323
4.7.7 Using different models together 325
4.8 The Haskell language 327
4.8.1 Computation model 328
4.8.2 Lazy evaluation 328
4.8.3 Currying 329
4.8.4 Polymorphic types 330
4.8.5 Type classes 331
4.9 Advanced topics 332
4.9.1 The declarative concurrent model with exceptions 332
4.9.2 More on lazy execution 334
4.9.3 Dataflow variables as communication channels 337
4.9.4 More on synchronization 339
4.9.5 Usefulness of dataflow variables 340
4.10 Historical notes 343
4.11 Exercises 344

5 Message-Passing Concurrency 353
5.1 The message-passing concurrent model 354
5.1.1 Ports 354
5.1.2 Semantics of ports 355
5.2 Port objects 357
5.2.1 The `NewPortObject` abstraction 358
5.2.2 An example 359
5.2.3 Reasoning with port objects 360
5.3 Simple message protocols 361
5.3.1 RMI (Remote Method Invocation) 361
5.3.2 Asynchronous RMI 364
5.3.3 RMI with callback (using thread) 364
5.3.4 RMI with callback (using record continuation) 366
5.3.5 RMI with callback (using procedure continuation) 367
5.3.6 Error reporting 367
5.3.7 Asynchronous RMI with callback 368
5.3.8 Double callbacks 369
5.4 Program design for concurrency 370
5.4.1 Programming with concurrent components 370

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

5.4.2 Design methodology 372
5.4.3 List operations as concurrency patterns 373
5.4.4 Lift control system 374
5.4.5 Improvements to the lift control system 383
5.5 Using the message-passing concurrent model directly 385
5.5.1 Port objects that share one thread 385
5.5.2 A concurrent queue with ports 387
5.5.3 A thread abstraction with termination detection 390
5.5.4 Eliminating sequential dependencies 393
5.6 The Erlang language 394
5.6.1 Computation model 394
5.6.2 Introduction to Erlang programming 395
5.6.3 The `receive` operation 398
5.7 Advanced topics 402
5.7.1 The nondeterministic concurrent model 402
5.8 Exercises 407

6 Explicit State 413
6.1 What is state? 416
6.1.1 Implicit (declarative) state 416
6.1.2 Explicit state 417
6.2 State and system building 418
6.2.1 System properties 419
6.2.2 Component-based programming 420
6.2.3 Object-oriented programming 421
6.3 The declarative model with explicit state 421
6.3.1 Cells 422
6.3.2 Semantics of cells 424
6.3.3 Relation to declarative programming 425
6.3.4 Sharing and equality 426
6.4 Abstract data types 427
6.4.1 Eight ways to organize ADTs 427
6.4.2 Variations on a stack 429
6.4.3 Revocable capabilities 433
6.4.4 Parameter passing 434
6.5 Stateful collections 438
6.5.1 Indexed collections 439
6.5.2 Choosing an indexed collection 441
6.5.3 Other collections 442
6.6 Reasoning with state 444
6.6.1 Invariant assertions 444
6.6.2 An example 445
6.6.3 Assertions 448
6.6.4 Proof rules 449

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

6.6.5 Normal termination . . . 452
6.7 Program design in the large . . . 453
6.7.1 Design methodology . . . 454
6.7.2 Hierarchical system structure . . . 456
6.7.3 Maintainability . . . 461
6.7.4 Future developments . . . 464
6.7.5 Further reading . . . 466
6.8 Case studies . . . 467
6.8.1 Transitive closure . . . 467
6.8.2 Word frequencies (with stateful dictionary) . . . 475
6.8.3 Generating random numbers . . . 476
6.8.4 "Word of Mouth" simulation . . . 481
6.9 Advanced topics . . . 484
6.9.1 Limitations of stateful programming . . . 484
6.9.2 Memory management and external references . . . 485
6.10 Exercises . . . 487

7 Object-Oriented Programming 493
7.1 Motivations . . . 495
7.1.1 Inheritance . . . 495
7.1.2 Encapsulated state and inheritance . . . 497
7.1.3 Objects and classes . . . 497
7.2 Classes as complete ADTs . . . 498
7.2.1 An example . . . 499
7.2.2 Semantics of the example . . . 500
7.2.3 Defining classes . . . 501
7.2.4 Initializing attributes . . . 503
7.2.5 First-class messages . . . 504
7.2.6 First-class attributes . . . 507
7.2.7 Programming techniques . . . 507
7.3 Classes as incremental ADTs . . . 507
7.3.1 Inheritance . . . 508
7.3.2 Static and dynamic binding . . . 511
7.3.3 Controlling encapsulation . . . 512
7.3.4 Forwarding and delegation . . . 517
7.3.5 Reflection . . . 522
7.4 Programming with inheritance . . . 524
7.4.1 The correct use of inheritance . . . 524
7.4.2 Constructing a hierarchy by following the type . . . 528
7.4.3 Generic classes . . . 531
7.4.4 Multiple inheritance . . . 533
7.4.5 Rules of thumb for multiple inheritance . . . 539
7.4.6 The purpose of class diagrams . . . 539
7.4.7 Design patterns . . . 540

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

7.5 Relation to other computation models 543
  7.5.1 Object-based and component-based programming 543
  7.5.2 Higher-order programming 544
  7.5.3 Functional decomposition versus type decomposition . . . 547
  7.5.4 Should everything be an object? 548
7.6 Implementing the object system 552
  7.6.1 Abstraction diagram 552
  7.6.2 Implementing classes 554
  7.6.3 Implementing objects 555
  7.6.4 Implementing inheritance 556
7.7 The Java language (sequential part) 556
  7.7.1 Computation model 557
  7.7.2 Introduction to Java programming 558
7.8 Active objects 563
  7.8.1 An example 564
  7.8.2 The `NewActive` abstraction 564
  7.8.3 The Flavius Josephus problem 565
  7.8.4 Other active object abstractions 568
  7.8.5 Event manager with active objects 569
7.9 Exercises 574

8 Shared-State Concurrency 577
8.1 The shared-state concurrent model 581
8.2 Programming with concurrency 581
  8.2.1 Overview of the different approaches 581
  8.2.2 Using the shared-state model directly 585
  8.2.3 Programming with atomic actions 588
  8.2.4 Further reading 589
8.3 Locks 590
  8.3.1 Building stateful concurrent ADTs 592
  8.3.2 Tuple spaces (“Linda”) 594
  8.3.3 Implementing locks 599
8.4 Monitors 600
  8.4.1 Bounded buffer 602
  8.4.2 Programming with monitors 605
  8.4.3 Implementing monitors 605
  8.4.4 Another semantics for monitors 607
8.5 Transactions 608
  8.5.1 Concurrency control 610
  8.5.2 A simple transaction manager 613
  8.5.3 Transactions on cells 616
  8.5.4 Implementing transactions on cells 619
  8.5.5 More on transactions 623
8.6 The Java language (concurrent part) 625

8.6.1 Locks . . . 626
8.6.2 Monitors . . . 626
8.7 Exercises . . . 626

9 Relational Programming 633
9.1 The relational computation model . . . 635
9.1.1 The **`choice`** and **`fail`** statements . . . 635
9.1.2 Search tree . . . 636
9.1.3 Encapsulated search . . . 637
9.1.4 The `Solve` function . . . 638
9.2 Further examples . . . 639
9.2.1 Numeric examples . . . 639
9.2.2 Puzzles and the n-queens problem . . . 641
9.3 Relation to logic programming . . . 644
9.3.1 Logic and logic programming . . . 644
9.3.2 Operational and logical semantics . . . 647
9.3.3 Nondeterministic logic programming . . . 650
9.3.4 Relation to pure Prolog . . . 652
9.3.5 Logic programming in other models . . . 653
9.4 Natural language parsing . . . 654
9.4.1 A simple grammar . . . 655
9.4.2 Parsing with the grammar . . . 656
9.4.3 Generating a parse tree . . . 656
9.4.4 Generating quantifiers . . . 657
9.4.5 Running the parser . . . 660
9.4.6 Running the parser “backwards” . . . 660
9.4.7 Unification grammars . . . 661
9.5 A grammar interpreter . . . 662
9.5.1 A simple grammar . . . 663
9.5.2 Encoding the grammar . . . 663
9.5.3 Running the grammar interpreter . . . 664
9.5.4 Implementing the grammar interpreter . . . 665
9.6 Databases . . . 667
9.6.1 Defining a relation . . . 668
9.6.2 Calculating with relations . . . 669
9.6.3 Implementing relations . . . 671
9.7 The Prolog language . . . 673
9.7.1 Computation model . . . 674
9.7.2 Introduction to Prolog programming . . . 676
9.7.3 Translating Prolog into a relational program . . . 681
9.8 Exercises . . . 684

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

III Specialized Computation Models 687

10 Graphical User Interface Programming 689
- 10.1 Basic concepts 691
- 10.2 Using the declarative/procedural approach 692
  - 10.2.1 Basic user interface elements 693
  - 10.2.2 Building the graphical user interface 694
  - 10.2.3 Declarative geometry 696
  - 10.2.4 Declarative resize behavior 697
  - 10.2.5 Dynamic behavior of widgets 698
- 10.3 Case studies 699
  - 10.3.1 A simple progress monitor 699
  - 10.3.2 A simple calendar widget 700
  - 10.3.3 Automatic generation of a user interface 703
  - 10.3.4 A context-sensitive clock 707
- 10.4 Implementing the GUI tool 712
- 10.5 Exercises 712

11 Distributed Programming 713
- 11.1 Taxonomy of distributed systems 716
- 11.2 The distribution model 718
- 11.3 Distribution of declarative data 720
  - 11.3.1 Open distribution and global naming 720
  - 11.3.2 Sharing declarative data 722
  - 11.3.3 Ticket distribution 723
  - 11.3.4 Stream communication 725
- 11.4 Distribution of state 726
  - 11.4.1 Simple state sharing 726
  - 11.4.2 Distributed lexical scoping 728
- 11.5 Network awareness 729
- 11.6 Common distributed programming patterns 730
  - 11.6.1 Stationary and mobile objects 730
  - 11.6.2 Asynchronous objects and dataflow 732
  - 11.6.3 Servers 734
  - 11.6.4 Closed distribution 737
- 11.7 Distribution protocols 738
  - 11.7.1 Language entities 738
  - 11.7.2 Mobile state protocol 740
  - 11.7.3 Distributed binding protocol 742
  - 11.7.4 Memory management 743
- 11.8 Partial failure 744
  - 11.8.1 Fault model 745
  - 11.8.2 Simple cases of failure handling 747
  - 11.8.3 A resilient server 748

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

11.8.4 Active fault tolerance 749
11.9 Security 749
11.10 Building applications 751
11.10.1 Centralized first, distributed later 751
11.10.2 Handling partial failure 751
11.10.3 Distributed components 752
11.11 Exercises 752

12 Constraint Programming 755
12.1 Propagate and search 756
12.1.1 Basic ideas 756
12.1.2 Calculating with partial information 757
12.1.3 An example 758
12.1.4 Executing the example 760
12.1.5 Summary 761
12.2 Programming techniques 761
12.2.1 A cryptarithmetic problem 761
12.2.2 Palindrome products revisited 763
12.3 The constraint-based computation model 764
12.3.1 Basic constraints and propagators 766
12.4 Computation spaces 766
12.4.1 Programming search with computation spaces 767
12.4.2 Definition 767
12.5 Implementing the relational computation model 777
12.5.1 The **`choice`** statement 778
12.5.2 Implementing the `Solve` function 778
12.6 Exercises 778

IV Semantics 781

13 Language Semantics 783
13.1 The shared-state concurrent model 784
13.1.1 The store 785
13.1.2 The single-assignment (constraint) store 785
13.1.3 Abstract syntax 786
13.1.4 Structural rules 787
13.1.5 Sequential and concurrent execution 789
13.1.6 Comparison with the abstract machine semantics 789
13.1.7 Variable introduction 790
13.1.8 Imposing equality (tell) 791
13.1.9 Conditional statements (ask) 793
13.1.10 Names 795
13.1.11 Procedural abstraction 795

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

13.1.12 Explicit state 797
13.1.13 By-need triggers 798
13.1.14 Read-only variables 800
13.1.15 Exception handling 801
13.1.16 Failed values 804
13.1.17 Variable substitution 805
13.2 Declarative concurrency 806
13.3 Eight computation models 808
13.4 Semantics of common abstractions 809
13.5 Historical notes 810
13.6 Exercises 811

V Appendices 815

A Mozart System Development Environment 817
A.1 Interactive interface 817
A.1.1 Interface commands 817
A.1.2 Using functors interactively 818
A.2 Batch interface 819

B Basic Data Types 821
B.1 Numbers (integers, floats, and characters) 821
B.1.1 Operations on numbers 823
B.1.2 Operations on characters 824
B.2 Literals (atoms and names) 825
B.2.1 Operations on atoms 826
B.3 Records and tuples 826
B.3.1 Tuples 827
B.3.2 Operations on records 828
B.3.3 Operations on tuples 829
B.4 Chunks (limited records) 829
B.5 Lists 830
B.5.1 Operations on lists 831
B.6 Strings 832
B.7 Virtual strings 833

C Language Syntax 835
C.1 Interactive statements 836
C.2 Statements and expressions 836
C.3 Nonterminals for statements and expressions 838
C.4 Operators 838
C.4.1 Ternary operator 841
C.5 Keywords 841
C.6 Lexical syntax 843

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

C.6.1 Tokens . . . 843
C.6.2 Blank space and comments . . . 843

D General Computation Model 845
D.1 Creative extension principle . . . 846
D.2 Kernel language . . . 847
D.3 Concepts . . . 848
D.3.1 Declarative models . . . 848
D.3.2 Security . . . 849
D.3.3 Exceptions . . . 849
D.3.4 Explicit state . . . 850
D.4 Different forms of state . . . 850
D.5 Other concepts . . . 851
D.5.1 What's next? . . . 851
D.5.2 Domain-specific concepts . . . 851
D.6 Layered language design . . . 852

Bibliography 853

Index 869

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

List of Figures

1.1 Taking apart the list `[5 6 7 8]` 7
1.2 Calculating the fifth row of Pascal's triangle 8
1.3 A simple example of dataflow execution 17
1.4 All possible executions of the first nondeterministic example . . . 21
1.5 One possible execution of the second nondeterministic example . . 23

2.1 From characters to statements 33
2.2 The context-free approach to language syntax 35
2.3 Ambiguity in a context-free grammar 36
2.4 The kernel language approach to semantics 39
2.5 Translation approaches to language semantics 42
2.6 A single-assignment store with three unbound variables 44
2.7 Two of the variables are bound to values 44
2.8 A value store: all variables are bound to values 45
2.9 A variable identifier referring to an unbound variable 46
2.10 A variable identifier referring to a bound variable 46
2.11 A variable identifier referring to a value 47
2.12 A partial value 47
2.13 A partial value with no unbound variables, i.e., a complete value . 48
2.14 Two variables bound together 48
2.15 The store after binding one of the variables 49
2.16 The type hierarchy of the declarative model 53
2.17 The declarative computation model 62
2.18 Lifecycle of a memory block 76
2.19 Declaring global variables 88
2.20 The Browser 90
2.21 Exception handling 92
2.22 Unification of cyclic structures 102

3.1 A declarative operation inside a general computation 114
3.2 Structure of the chapter 115
3.3 A classification of declarative programming 116
3.4 Finding roots using Newton's method (first version) 121
3.5 Finding roots using Newton's method (second version) 123

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

3.6 Finding roots using Newton's method (third version) 124
3.7 Finding roots using Newton's method (fourth version) 124
3.8 Finding roots using Newton's method (fifth version) 125
3.9 Sorting with mergesort 140
3.10 Control flow with threaded state 141
3.11 Deleting node `Y` when one subtree is a leaf (easy case) 156
3.12 Deleting node `Y` when neither subtree is a leaf (hard case) 157
3.13 Breadth-first traversal 159
3.14 Breadth-first traversal with accumulator 160
3.15 Depth-first traversal with explicit stack 160
3.16 The tree drawing constraints 162
3.17 An example tree 162
3.18 Tree drawing algorithm 164
3.19 The example tree displayed with the tree drawing algorithm . . . 165
3.20 Delayed execution of a procedure value 181
3.21 Defining an integer loop 186
3.22 Defining a list loop 186
3.23 Simple loops over integers and lists 187
3.24 Defining accumulator loops 188
3.25 Accumulator loops over integers and lists 189
3.26 Folding a list 190
3.27 Declarative dictionary (with linear list) 199
3.28 Declarative dictionary (with ordered binary tree) 201
3.29 Word frequencies (with declarative dictionary) 202
3.30 Internal structure of binary tree dictionary in `WordFreq` (in part) 203
3.31 Doing `S1={Pop S X}` with a secure stack 208
3.32 A simple graphical I/O interface for text 217
3.33 Screen shot of the word frequency application 228
3.34 Standalone dictionary library (file `Dict.oz`) 229
3.35 Standalone word frequency application (file `WordApp.oz`) 230
3.36 Component dependencies for the word frequency application . . . 231

4.1 The declarative concurrent model 240
4.2 Causal orders of sequential and concurrent executions 242
4.3 Relationship between causal order and interleaving executions . . 242
4.4 Execution of the **`thread`** statement 245
4.5 Thread creations for the call `{Fib 6}` 254
4.6 The Oz Panel showing thread creation in `{Fib 26 X}` 255
4.7 Dataflow and rubber bands 256
4.8 Cooperative and competitive concurrency 259
4.9 Operations on threads 260
4.10 Producer-consumer stream communication 261
4.11 Filtering a stream 264
4.12 A prime-number sieve with streams 264

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

4.13 Pipeline of filters generated by `{Sieve Xs 316}` 266
4.14 Bounded buffer 267
4.15 Bounded buffer (data-driven concurrent version) 267
4.16 Digital logic gates 272
4.17 A full adder 273
4.18 A latch 275
4.19 A linguistic abstraction for logic gates 276
4.20 Tree drawing algorithm with order-determining concurrency . . . 278
4.21 Procedures, coroutines, and threads 280
4.22 Implementing coroutines using the `Thread` module 281
4.23 Concurrent composition 282
4.24 The by-need protocol 287
4.25 Stages in a variable's lifetime 289
4.26 Practical declarative computation models 291
4.27 Bounded buffer (naive lazy version) 296
4.28 Bounded buffer (correct lazy version) 296
4.29 Lazy solution to the Hamming problem 298
4.30 A simple 'Ping Pong' program 310
4.31 A standalone 'Ping Pong' program 311
4.32 A standalone 'Ping Pong' program that exits cleanly 312
4.33 Changes needed for instrumenting procedure P1 317
4.34 How can two clients send to the same server? They cannot! 319
4.35 Impedance matching: example of a serializer 326

5.1 The message-passing concurrent model 356
5.2 Three port objects playing ball 359
5.3 Message diagrams of simple protocols 362
5.4 Schematic overview of a building with lifts 374
5.5 Component diagram of the lift control system 375
5.6 Notation for state diagrams 375
5.7 State diagram of a lift controller 377
5.8 Implementation of the timer and controller components 378
5.9 State diagram of a floor 379
5.10 Implementation of the floor component 380
5.11 State diagram of a lift 381
5.12 Implementation of the lift component 382
5.13 Hierarchical component diagram of the lift control system 383
5.14 Defining port objects that share one thread 386
5.15 Screenshot of the 'Ping-Pong' program 386
5.16 The 'Ping-Pong' program: using port objects that share one thread 387
5.17 Queue (naive version with ports) 388
5.18 Queue (correct version with ports) 389
5.19 A thread abstraction with termination detection 391
5.20 A concurrent filter without sequential dependencies 392

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

5.21 Translation of `receive` without time out 400
5.22 Translation of `receive` with time out 401
5.23 Translation of `receive` with zero time out 402
5.24 Connecting two clients using a stream merger 404
5.25 Symmetric nondeterministic choice (using exceptions) 407
5.26 Asymmetric nondeterministic choice (using `IsDet`) 407

6.1 The declarative model with explicit state 422
6.2 Five ways to package a stack 429
6.3 Four versions of a secure stack 430
6.4 Different varieties of indexed collections 439
6.5 Extensible array (stateful implementation) 443
6.6 A system structured as a hierarchical graph 456
6.7 System structure – static and dynamic 458
6.8 A directed graph and its transitive closure 466
6.9 One step in the transitive closure algorithm 467
6.10 Transitive closure (first declarative version) 469
6.11 Transitive closure (stateful version) 471
6.12 Transitive closure (second declarative version) 472
6.13 Transitive closure (concurrent/parallel version) 474
6.14 Word frequencies (with stateful dictionary) 476

7.1 An example class `Counter` (with **`class`** syntax) 498
7.2 Defining the `Counter` class (without syntactic support) 499
7.3 Creating a `Counter` object 500
7.4 Illegal and legal class hierarchies 508
7.5 A class declaration is an executable statement 509
7.6 An example class `Account` 510
7.7 The meaning of "private" 513
7.8 Different ways to extend functionality 517
7.9 Implementing delegation 519
7.10 An example of delegation 521
7.11 A simple hierarchy with three classes 525
7.12 Constructing a hierarchy by following the type 527
7.13 Lists in object-oriented style 528
7.14 A generic sorting class (with inheritance) 529
7.15 Making it concrete (with inheritance) 530
7.16 A class hierarchy for genericity 530
7.17 A generic sorting class (with higher-order programming) 531
7.18 Making it concrete (with higher-order programming) 532
7.19 Class diagram of the graphics package 534
7.20 Drawing in the graphics package 536
7.21 Class diagram with an association 537
7.22 The Composite pattern 541

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

7.23 Functional decomposition versus type decomposition 548
7.24 Abstractions in object-oriented programming 553
7.25 An example class `Counter` (again) 554
7.26 An example of class construction 555
7.27 An example of object construction 556
7.28 Implementing inheritance 557
7.29 Parameter passing in Java 562
7.30 Two active objects playing ball (definition) 563
7.31 Two active objects playing ball (illustration) 564
7.32 The Flavius Josephus problem 565
7.33 The Flavius Josephus problem (active object version) 566
7.34 The Flavius Josephus problem (data-driven concurrent version) . 568
7.35 Event manager with active objects 570
7.36 Adding functionality with inheritance 571
7.37 Batching a list of messages and procedures 572

8.1 The shared-state concurrent model 580
8.2 Different approaches to concurrent programming 582
8.3 Concurrent stack 586
8.4 The hierarchy of atomic actions 588
8.5 Differences between atomic actions 589
8.6 Queue (declarative version) 591
8.7 Queue (sequential stateful version) 592
8.8 Queue (concurrent stateful version with lock) 593
8.9 Queue (concurrent object-oriented version with lock) 594
8.10 Queue (concurrent stateful version with exchange) 595
8.11 Queue (concurrent version with tuple space) 596
8.12 Tuple space (object-oriented version) 597
8.13 Lock (non-reentrant version without exception handling) 598
8.14 Lock (non-reentrant version with exception handling) 598
8.15 Lock (reentrant version with exception handling) 599
8.16 Bounded buffer (monitor version) 604
8.17 Queue (extended concurrent stateful version) 606
8.18 Lock (reentrant get-release version) 607
8.19 Monitor implementation 608
8.20 State diagram of one incarnation of a transaction 615
8.21 Architecture of the transaction system 619
8.22 Implementation of the transaction system (part 1) 621
8.23 Implementation of the transaction system (part 2) 622
8.24 Priority queue 624
8.25 Bounded buffer (Java version) 627

9.1 Search tree for the clothing design example 637
9.2 Two digit counting with depth-first search 640

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

9.3 The n-queens problem (when $n = 4$) 642
9.4 Solving the n-queens problem with relational programming 643
9.5 Natural language parsing (simple nonterminals) 658
9.6 Natural language parsing (compound nonterminals) 659
9.7 Encoding of a grammar 664
9.8 Implementing the grammar interpreter 666
9.9 A simple graph 669
9.10 Paths in a graph 671
9.11 Implementing relations (with first-argument indexing) 672

10.1 Building the graphical user interface 693
10.2 Simple text entry window 694
10.3 Function for doing text entry 695
10.4 Windows generated with the `lr` and `td` widgets 695
10.5 Window generated with `newline` and `continue` codes 696
10.6 Declarative resize behavior 697
10.7 Window generated with the `glue` parameter 698
10.8 A simple progress monitor 700
10.9 A simple calendar widget 701
10.10Automatic generation of a user interface 703
10.11From the original data to the user interface 704
10.12Defining the read-only presentation 705
10.13Defining the editable presentation 705
10.14Three views of FlexClock, a context-sensitive clock 707
10.15Architecture of the context-sensitive clock 707
10.16View definitions for the context-sensitive clock 710
10.17The best view for any size clock window 711

11.1 A simple taxonomy of distributed systems 717
11.2 The distributed computation model 718
11.3 Process-oriented view of the distribution model 720
11.4 Distributed locking 727
11.5 The advantages of asynchronous objects with dataflow 733
11.6 Graph notation for a distributed cell 741
11.7 Moving the state pointer 741
11.8 Graph notation for a distributed dataflow variable 742
11.9 Binding a distributed dataflow variable 742
11.10A resilient server 748

12.1 Constraint definition of *Send-More-Money* puzzle 762
12.2 Constraint-based computation model 765
12.3 Depth-first single solution search 768
12.4 Visibility of variables and bindings in nested spaces 770
12.5 Communication between a space and its distribution strategy . . . 775
12.6 Lazy all-solution search engine `Solve` 779

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

13.1 The kernel language with shared-state concurrency 787
B.1 Graph representation of the infinite list `C1=a|b|C1` 832
C.1 The ternary operator "`. :=`" 840

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

List of Tables

2.1 The declarative kernel language 50
2.2 Value expressions in the declarative kernel language 51
2.3 Examples of basic operations 56
2.4 Expressions for calculating with numbers 82
2.5 The **`if`** statement 83
2.6 The **`case`** statement 83
2.7 Function syntax 85
2.8 Interactive statement syntax 88
2.9 The declarative kernel language with exceptions 94
2.10 Exception syntax 95
2.11 Equality (unification) and equality test (entailment check) 100

3.1 The descriptive declarative kernel language 117
3.2 The parser's input language (which is a token sequence) 166
3.3 The parser's output language (which is a tree) 167
3.4 Execution times of kernel instructions 170
3.5 Memory consumption of kernel instructions 176
3.6 The declarative kernel language with secure types 206
3.7 Functor syntax 224

4.1 The data-driven concurrent kernel language 240
4.2 The demand-driven concurrent kernel language 285
4.3 The declarative concurrent kernel language with exceptions 332
4.4 Dataflow variable as communication channel 337
4.5 Classifying synchronization 340

5.1 The kernel language with message-passing concurrency 355
5.2 The nondeterministic concurrent kernel language 403

6.1 The kernel language with explicit state 423
6.2 Cell operations 423

7.1 Class syntax 501

8.1 The kernel language with shared-state concurrency 580

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

9.1 The relational kernel language 635
9.2 Translating a relational program to logic 649
9.3 The extended relational kernel language 673

11.1 Distributed algorithms 740

12.1 Primitive operations for computation spaces 768

13.1 Eight computation models 809

B.1 Character lexical syntax 822
B.2 Some number operations 823
B.3 Some character operations 824
B.4 Literal syntax *(in part)* 825
B.5 Atom lexical syntax 825
B.6 Some atom operations 826
B.7 Record and tuple syntax (*in part*) 826
B.8 Some record operations 828
B.9 Some tuple operations 829
B.10 List syntax (*in part*) 829
B.11 Some list operations 831
B.12 String lexical syntax 832
B.13 Some virtual string operations 833

C.1 Interactive statements 836
C.2 Statements and expressions 836
C.3 Nestable constructs (no declarations) 837
C.4 Nestable declarations 837
C.5 Terms and patterns 838
C.6 Other nonterminals needed for statements and expressions 839
C.7 Operators with their precedence and associativity 840
C.8 Keywords 841
C.9 Lexical syntax of variables, atoms, strings, and characters 842
C.10 Nonterminals needed for lexical syntax 842
C.11 Lexical syntax of integers and floating point numbers 842

D.1 The general kernel language 847

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

Preface

Six blind sages were shown an elephant and met to discuss their experience. "It's wonderful," said the first, "an elephant is like a rope: slender and flexible." "No, no, not at all," said the second, "an elephant is like a tree: sturdily planted on the ground." "Marvelous," said the third, "an elephant is like a wall." "Incredible," said the fourth, "an elephant is a tube filled with water." "What a strange piecemeal beast this is," said the fifth. "Strange indeed," said the sixth, "but there must be some underlying harmony. Let us investigate the matter further."
– Freely adapted from a traditional Indian fable.

"A programming language is like a natural, human language in that it favors certain metaphors, images, and ways of thinking."
– Mindstorms: Children, Computers, and Powerful Ideas [141], *Seymour Papert* (1980)

One approach to study computer programming is to study programming languages. But there are a tremendously large number of languages, so large that it is impractical to study them all. How can we tackle this immensity? We could pick a small number of languages that are representative of different programming paradigms. But this gives little insight into programming as a unified discipline. This book uses another approach.

We focus on programming *concepts* and the *techniques* to use them, not on programming languages. The concepts are organized in terms of computation models. A computation model is a formal system that defines how computations are done. There are many ways to define computation models. Since this book is intended to be practical, it is important that the computation model should be directly useful to the programmer. We will therefore define it in terms of concepts that are important to programmers: data types, operations, and a programming language. The term computation model makes precise the imprecise notion of "programming paradigm". The rest of the book talks about computation models and not programming paradigms. Sometimes we will use the phrase programming model. This refers to what the programmer needs: the programming techniques and design principles made possible by the computation model.

Each computation model has its own set of techniques for programming and

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

reasoning about programs. The number of different computation models that are known to be useful is much smaller than the number of programming languages. This book covers many well-known models as well as some less-known models. The main criterium for presenting a model is whether it is useful in practice.

Each computation model is based on a simple core language called its *kernel language*. The kernel languages are introduced in a progressive way, by adding concepts one by one. This lets us show the deep relationships between the different models. Often, just adding one new concept makes a world of difference in programming. For example, adding destructive assignment (explicit state) to functional programming allows us to do object-oriented programming.

When stepping from one model to the next, how do we decide on what concepts to add? We will touch on this question many times in the book. The main criterium is the *creative extension principle*. Roughly, a new concept is added when programs become complicated for technical reasons unrelated to the problem being solved. Adding a concept to the kernel language can keep programs simple, if the concept is chosen carefully. This is explained further in Appendix D. This principle underlies the progression of kernel languages presented in the book.

A nice property of the kernel language approach is that it lets us use different models together in the same program. This is usually called *multiparadigm programming*. It is quite natural, since it means simply to use the right concepts for the problem, independent of what computation model they originate from. Multiparadigm programming is an old idea. For example, the designers of Lisp and Scheme have long advocated a similar view. However, this book applies it in a much broader and deeper way than was previously done.

From the vantage point of computation models, the book also sheds new light on important problems in informatics. We present three such areas, namely graphical user interface design, robust distributed programming, and constraint programming. We show how the judicious combined use of several computation models can help solve some of the problems of these areas.

Languages mentioned

We mention many programming languages in the book and relate them to particular computation models. For example, Java and Smalltalk are based on an object-oriented model. Haskell and Standard ML are based on a functional model. Prolog and Mercury are based on a logic model. Not all interesting languages can be so classified. We mention some other languages for their own merits. For example, Lisp and Scheme pioneered many of the concepts presented here. Erlang is functional, inherently concurrent, and supports fault tolerant distributed programming.

We single out four languages as representatives of important computation models: Erlang, Haskell, Java, and Prolog. We identify the computation model of each language in terms of the book's uniform framework. For more information about them we refer readers to other books. Because of space limitations, we are

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

not able to mention all interesting languages. Omission of a language does not imply any kind of value judgement.

Goals of the book

Teaching programming

The main goal of the book is to teach programming as a unified discipline with a scientific foundation that is useful to the practicing programmer. Let us look closer at what this means.

What is programming?

We define *programming*, as a general human activity, to mean the act of extending or changing a system's functionality. Programming is a widespread activity that is done both by nonspecialists (e.g., consumers who change the settings of their alarm clock or cellular phone) and specialists (computer programmers, the audience of this book).

This book focuses on the construction of software systems. In that setting, programming is the step between the system's specification and a running program that implements it. The step consists in designing the program's architecture and abstractions and coding them into a programming language. This is a broad view, perhaps broader than the usual connotation attached to the word programming. It covers both programming "in the small" and "in the large". It covers both (language-independent) architectural issues and (language-dependent) coding issues. It is based more on concepts and their use rather than on any one programming language. We find that this general view is natural for teaching programming. It allows to look at many issues in a way unbiased by limitations of any particular language or design methodology. When used in a specific situation, the general view is adapted to the tools used, taking account their abilities and limitations.

Both science and technology

Programming as defined above has two essential parts: a technology and its scientific foundation. The technology consists of tools, practical techniques, and standards, allowing us to *do* programming. The science consists of a broad and deep theory with predictive power, allowing us to *understand* programming. Ideally, the science should explain the technology in a way that is as direct and useful as possible.

If either part is left out, we are no longer doing programming. Without the technology, we are doing pure mathematics. Without the science, we are doing a craft, i.e., we lack deep understanding. Teaching programming correctly therefore means teaching both the technology (current tools) and the science (fundamental

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.

concepts). Knowing the tools prepares the student for the present. Knowing the concepts prepares the student for future developments.

More than a craft

Despite many efforts to introduce a scientific foundation, programming is almost always taught as a craft. It is usually taught in the context of one (or a few) programming languages (e.g., Java, complemented with Haskell, Scheme, or Prolog). The historical accidents of the particular languages chosen are interwoven together so closely with the fundamental concepts that the two cannot be separated. There is a confusion between tools and concepts. What's more, different schools of thought have developed, based on different ways of viewing programming, called "paradigms": object-oriented, logic, functional, etc. Each school of thought has its own science. The unity of programming as a single discipline has been lost.

Teaching programming in this fashion is like having separate schools of bridge building: one school teaches how to build wooden bridges and another school teaches how to build iron bridges. Graduates of either school would implicitly consider the restriction to wood or iron as fundamental and would not think of using wood and iron together.

The result is that programs suffer from poor design. We give an example based on Java, but the problem exists in all existing languages to some degree. Concurrency in Java is complex to use and expensive in computational resources. Because of these difficulties, Java-taught programmers conclude that concurrency is a fundamentally complex and expensive concept. Program specifications are designed around the difficulties, often in a contorted way. But these difficulties are not fundamental at all. There are forms of concurrency that are quite useful and yet as easy to program with as sequential programs (for example, stream programming as exemplified by Unix pipes). Furthermore, it is possible to implement threads, the basic unit of concurrency, almost as cheaply as procedure calls. If the programmer were taught about concurrency in the correct way, then he or she would be able to specify for and program in systems without concurrency restrictions (including improved versions of Java).

The kernel language approach

Practical programming languages scale up to programs of millions of lines of code. They provide a rich set of abstractions and syntax. How can we separate the languages' fundamental concepts, which underlie their success, from their historical accidents? The kernel language approach shows one way. In this approach, a practical language is translated into a *kernel language* that consists of a small number of *programmer-significant* elements. The rich set of abstractions and syntax is encoded into the small kernel language. This gives both programmer and student a clear insight into what the language does. The kernel language has a simple formal semantics that allows reasoning about program correctness and

Copyright © 2001-3 by P. Van Roy and S. Haridi. All rights reserved.