

Contents

Preface	xi
I Everyday programs	1
1 Programs that work	3
1.1 Introduction	3
1.2 Jack be nimble, Jack be quick...	4
1.3 When does a program “work”?	5
1.4 Requirements and specifications: pre- and postconditions	6
1.5 Satisfying a specification: assertions	8
1.6 Turn the handle, and watch the program flow	10
1.7 Assertions for loops: invariants	11
1.8 Avoiding infinite loops: variants	14
1.9 Summary	16
1.10 Exercises	18
2 Using invariants to design loops	25
2.1 Introduction	25
2.2 The longest <i>Good</i> subsegment problem	25
2.3 First attempt: operational intuition and diagrams	26
2.4 Second attempt: operational intuition and a caterpillar	27
2.5 Third attempt: use an invariant to design the program	28
2.6 Exercises	31
3 Finding invariants	33
3.1 Introduction	33
3.2 Split a conjunct	33
3.3 Introduce a new variable	35
3.4 “Tail” invariants	40
3.5 Cascading invariants: one leads to another	45
3.6 Invariants should be “inductive”	48
3.7 Exercises	49
4 Finding variants	55
4.1 Introduction: simple variants	55
4.2 Lexicographic variants: not so simple	57
4.3 Structural variants	58

4.4	Well-founded variants, and even “real valued”	59
4.5	Exercises	60
5	Checking assignments and conditionals	65
5.1	Introduction	65
5.2	Checking assignments	65
5.3	The “do nothing” statement skip	67
5.4	Checking conditionals	68
5.5	Exercises	70
6	Summary of Part I	73
6.1	What’s not obvious	73
6.2	What <i>is</i> obvious	75
II	Data structures and their encapsulation	77
7	Introduction to Part II	79
7.1	Data vs. data <i>structures</i>	79
7.2	Data encapsulation	80
8	Coupling invariants	81
8.1	Introduction	81
8.2	Implementing sets as sequences	81
8.3	Sentinels as “high-level” data	83
8.4	Writing abstract data-types	86
8.5	Coding concrete data-types	87
8.6	Exercises	91
9	Case study in coupling invariants: Fibonacci numbers	93
9.1	Introduction: definition vs. implementation	93
9.2	Linear-time Fibonacci	94
9.3	Introducing matrices, but still linear time	94
9.4	Achieving logarithmic-time with matrices	95
9.5	Removing auxiliary variables	96
9.6	Summary	98
9.7	Exercises	98
10	Encapsulated data-types: how exactly is it done?	101
10.1	Introduction	101
10.2	Data-type invariants	101
10.3	Rules for encapsulation: the basics summarised	107
10.4	Encapsulation rules justified	110
10.5	More advanced techniques	112
10.6	Exercises	113
11	Case study: the Mean Calculator	117
11.1	Requirements and specification of the calculator	117
11.2	Implementation of the calculator	119
11.3	Exercises	120
12	Summary of Part II	123
12.1	What is obvious	123

12.2	Specifications vs. implementations: how are they connected?	124
12.3	What is <i>not</i> obvious	126
12.4	Exercises	128
III	Concurrency — and how to check it	131
13	What is “concurrency”?	133
13.1	Introduction to concurrent programs	133
13.2	A small example: automatic teller machines	134
13.3	Atomicity of expressions and assignments	136
13.4	What simple locks guarantee	137
13.5	What simple locks actually do	138
13.6	Where simple locks came from	138
13.7	Critical sections and other techniques	139
13.8	Concurrency and interference	143
13.9	Exercises	144
14	The Owicki–Gries method	147
14.1	Introduction	147
14.2	Controlled interleaving	147
14.3	Checking <i>local</i> correctness: what’s <i>old</i>	148
14.4	Checking global invariants: what’s obvious	149
14.5	Checking <i>global</i> correctness: what’s <i>new</i>	149
14.6	Exercises	150
15	Critical sections with Owicki–Gries	153
15.1	Introduction	153
15.2	Using <code>await</code> statements with a single Boolean	153
15.3	Deadlock, livelock and starvation	156
15.4	Exercises	158
16	Peterson’s algorithm for mutual exclusion	161
16.1	Introduction	161
16.2	Implementation based on a queue	162
16.3	Eliminating multiple assignments	165
16.4	No deadlock, no starvation in Peterson’s algorithm	167
16.5	Peterson’s algorithm without locks	167
16.6	Exercises	168
17	Garbage collection on the fly	171
17.1	Introduction	171
17.2	What “garbage” is — in a computer	172
17.3	Why garbage needs to be collected	173
17.4	How garbage is collected	174
17.5	Origins of this presentation	175
17.6	Using a Boolean <i>mark-bit</i> to control scanning	178
17.7	Ensuring local correctness of the Scanner	179
17.8	Ensuring global correctness of the Scanner	180
17.9	Using a <i>count</i> to control scanning	182
17.10	Counting blacks non-atomically	185
17.11	Atomicity of the Mutator, in two steps	187

17.12 The completed program	191
17.13 Exercises	191
IV Machine-assisted program checking, and testing	195
18 Machine-assisted program checking	197
18.1 Why by-hand checking is only a beginning	197
18.2 Automated checking of assertions in Dafny	198
18.3 Exercises	203
19 Program testing	205
19.1 Why is testing <i>always</i> necessary?	205
19.2 From expectations to actual behaviour, in five steps	205
19.3 How assertions and testing can work together	210
19.4 A cautionary tale	211
19.5 Exercises	213
Afterword	215
The “Lost Art” — <i>What</i> was lost, and why?	215
Background, evolution, experience and prospects	216
Support and suggestions for teachers	220
Sources, literature, future directions	222
Appendices	227
A Drill exercises	229
A.1 What are “drills”? And what are they for?	229
A.2 Substitution	230
A.3 Sequences and operators on them	231
A.4 Invariants	232
A.5 Variants	233
A.6 Propositions, sets and predicates	234
A.7 Thinking outside the box	235
A.8 Hoare triples “in the small”	235
A.9 Hoare triples in the large(r)	236
A.10 Whole-program drills	238
B Summary of rules for checking programs	241
B.1 The basic programs	241
B.2 Assertions: “What’s true here?”	242
B.3 Rules for checking larger program fragments	244
B.4 Rules for checking loops: while and for	245
B.5 Concurrency: await , and global correctness	248
B.6 Exotica: assert and assume statements, and specifications	249
B.7 Exercises	251
C Data refinement: the real story	253
C.1 Introduction	253
C.2 Step-by-step from abstract to concrete	253
C.3 Other manipulations of assertions, assumptions and specifications	256

C.4	An example: <code>add(s)</code> to a size-limited <code>Set</code>	257
C.5	Postscript on justification of <i>dti</i> 's and their rules	260
C.6	Exercises	261
D	The “arithmetic” of conditions	263
D.1	Introduction and rationale	263
D.2	Why is my program correct?	264
D.3	How do I write my program in the first place?	265
D.4	<i>Calculating</i> with conditions: we start with sets	267
D.5	Simple calculations in logic	269
D.6	Terms in logic are like expressions in programming	270
D.7	Simple formulae are like (in)equalities in programming	271
D.8	Propositions, and propositional formulae	272
D.9	Operator precedence	274
D.10	Calculation with logical formulae	274
D.11	Quantifiers	276
D.12	(General) formulae	278
D.13	Exercises on propositions	278
D.14	Exercises on quantifiers	280
E	Some helpful logical identities	281
E.1	Introduction	281
E.2	Some basic propositional rules	281
E.3	Some basic quantifier rules	285
E.4	Epilogue on logical notation and terminology	288
E.5	Exercises	290
F	Illustration of heap behaviour during garbage collection	293
F.1	Mark-and-sweep garbage collection	293
F.2	Woodger's scenario	293
G	Python-specific issues	299
G.1	Multiple assignments	299
G.2	Block structure (and assertions)	299
G.3	<code>for</code> -loops vs. <code>while</code> -loops in Python	300
G.4	Object orientation in Python, and in general	301
G.5	Exercises	301
H	Answers to selected drills	303
I	Answers to selected exercises	309
	Bibliography	335
	Index	339