Carroll Morgan

(In-)Formal Methods: the Lost Art... *or*

How to write programs that work

For COMP6721 in 2023T2 November 30, 2022

Preface

What is this text about, and who is it for?

This text is about how to program more effectively, faster, with better results... and how to enjoy doing so.

And it's intended for two kinds of people: those who have some experience of programming, but are not yet experts; and those who *are* experts, but might like to see a possible explanation of *why* they are. Thus its purpose is to introduce (to the first group) and make explicit (to the second group) some coding techniques that can be used to organise, check and maintain large programs, especially those whose correct functioning is crucial (i.e. must have as few bugs as possible).

But –in spite of that emphasis on "large"– the examples used in this text are very small ones; and many will be programs that readers have written already. In fact the more experienced programmers might have written them dozens of times.

Using familiar examples to introduce "(In-)Formal methods" is a deliberate, although possibly unusual choice; and it is quite different from what is usually done.

THE USUAL APPROACH to teaching programming is to introduce a general-purpose, widely used language as a first step, and then to apply it initially to very simple problems. The idea there is to instil familiarity with *some* programming language –it does really not matter so much which one– and then by choosing more and more ambitious examples to move the students to a level where they realise that what they are learning has two main components. One of them is "How to find algorithms."

Algorithms are the steps that a program follows to solve a problem. They are *expressed* in a programming language so that a computer can perform them automatically, fast and reliably. An algorithm is the "essence" of a program.

The other component of this approach to learning to program is of course coding: it's putting an idea, the algorithm (the essence) into words: the program code.

But when programs are large and complex, it is not easy to be sure they will work. First, the algorithm itself might be complex, and not well understood. It could even be wrong. And second, even if the algorithm is not complex (but especially if it is), translating it into code can be tricky too, and sometimes very error prone. It's a bit like "typos" in ordinary writing; but actually it's much worse. A typo (think "misspelling") in a computer program is usually picked up automatically: the rules for programs, their "spelling and grammar", are very exact. Computers detect "syntax errors" in programs very easily.

Copyright © Carroll Morgan 2021

What computers *don't* detect is grammatically correct programs that –actually– don't do what their programmers think they do. Maybe the algorithm is wrong in the first place; maybe it was coded up incorrectly; maybe both. But most people simply accept all that as just a "fact of life" when writing computer programs.

But another approach is often taken by people who *don't* accept "all that... as a fact of life": it's called "Formal Methods".

Formal Methods uses rigorous logic, sometimes in a very sophisticated form, to *prove* mathematically that a program does what it should. Obviously, to prove that, it must be known in the first place what "should" means, what the program is actually supposed to do — and that knowledge must be exact, at least if the proof is to have any rigour at all. So to use Formal Methods properly, the program's *specification* must be written in mathematical language too, not just the proof. And it must be possible to translate the program's code into a mathematical form as well; and then after that more mathematics must be deployed to connect the (mathematised) program to the (mathematical) specification. So it's "mathematics all the way down."

Formal Methods is therefore a very specialised part of Computer Science, and how it works –and especially *why* it works– are fascinating topics in their own right. This "second, alternative" approach to teaching programming, which we are describing in this paragraph, presents Formal Methods first, though perhaps not at its highest intensity, as a sort of "limbering-up exercise" before the students are even allowed to attempt and submit answers to real programming exercises, to "touch the computer at all" so to speak. After that, they are allowed to use the just-learned formal methods on real programs, first small ones and then steadily larger, and they proceed as in the first approach above.

The problem with <u>both</u> of those two approaches...
is that *neither* of them gets us where we want to go. Not the programmers, and not the people who depend on the programs they write. And that's why we don't adopt either of those approaches in this text.

The first approach leads to enormous amounts of time wasted as running programs are tested, debugged, tested again, debugged further... Nights and weekends are lost doing that, and –even then– there are further losses when users suffer because the programs *still* don't work, and their unexpected failures prevent ordinary people from doing their own jobs.

And the second approach simply puts off many people right from the start. You don't *have* to be a mathematician to be a programmer: "everyone knows that" — and actually they are right. Many students therefore reject the Formal-Methods–first approach: they develop a skepticism of the whole idea, and we all end up worse off than we would have been if we just went straight into programming, i.e. without any "method" at all.

What's left is what you will find in this text: neither of those two approaches. Here, we use simple programs, programs that people can write already and, *at the same time* general ideas "lifted" from Formal Methods and made accessible and understandable in an everyday way. This approach really does deserve its title: [1]

(In-)Formal Methods.

[1] "The Lost Art" is explained in the Afterword ⇓.

(In-)Formal Methods shows you how to combine the *ideas* of Formal Methods, a way of thinking about programs, and the *ideas* built into designing algorithms you want to program, into an approach that

- Reduces the number of mistakes you make when coding.
- Helps you to divide a large programming job into smaller pieces, share them with colleagues, and then put the results back together again.
- Leaves behind a description of the essential ideas that were used during the coding (the "documentation"), so that the program can be easily understood and modified by others, later on.
- Helps to understand where an error occurred in the design process, if after all a bug turns up in the running code, so that the programming *process* (e.g. in your organisation, or just your daily life), can be tuned and improved. If a program ends up wrong, you need to fix not only the program *but also the way it was made*. Sometimes that involves "tweaking" the *social* processes involved. (In-)Formal Methods helps you find out which ones need attention.
- Helps to produce modular components that can be reused in many programs, not just the ones they were originally written for.
- Suggests ways in which the program (components) can be tested, both during coding and after it is complete.

The simple programming examples used in this text –the ones you have probably done already in a more traditional way– can be found listed under *Program Examples* in the index ⇓. The programming language used for the examples is Python, but the code can easily be adapted to other, similar languages. (See also App. G.1.)

The text includes more than 200 exercises ⇓, cross-referenced to the topics that gave rise to them, and to each other, and selected answers are given in App. H.

The general algorithmic topics covered include sorting, searching, order of growth (complexity), data structures and encapsulation, loop correctness and concurrency.

But none of those topics is covered exhaustively: it would be more accurate to say that in *this* text they are only "visited". Our purpose here is to introduce and then encourage further interest in more specialised programming topics and techniques –which readers will later discover to have their own authoritative texts– and *at the same time* to give some idea of how to think effectively about those topics and how to use them. We don't tell the whole story here; rather we help people to find and better understand the more comprehensive stories, told by others, that will deepen and complete their knowledge.

Because of the simple examples chosen, the programs we study here are connected to what people already know: they can concentrate on the new *process* we are following, without having to wonder how the example program actually works. Introducing a new topic *and* how to think about it *and* why it works *and* the really difficult problems that it solves, all at the same time... for a general audience, that is often a bad idea...

...because most people learn best in layers, each layer building on the one before, each one benefiting from the proper assimilation of earlier material.

Copyright © Carroll Morgan 2021

> So what I'm aiming for in this text, in this approach, is that when people get a bit further into their programming careers, meet new topics, stretch their capabilities, they will say
>
> *"Oh! Yes — I learned about that in (In-)Formal Methods... and how to think about it."*

And then they will get off to a running start — in the right direction.

Organisation and use

The text is organised into four major parts, and the first is the most important: *Everyday Programs*. It could be used on its own for an introductory course.

Exercises are given throughout, and those with answers included in the text are marked ⇓. The remainder can be used in class, for students to solve.

The material is substantially based on the "(In-)Formal Methods" course that is given at The University of New South Wales, in Sydney, and elsewhere; and there are mini-project–sized assignments associated with the course (but not given here) which can take up to a week to complete each. It's not hard to fit 3 of them into say a 9-week term. They include using the "IFM techiques" explained here to develop, program and test...

- An "optimal paragraph formatter", such as is used by LaTeX.
- A Unix "diff" protype.
- A circular-buffer–based copy utility, developed using IFM but "tested" by an automatic program-checker .
- More...

Acknowledgments

I'm grateful for the support of the School of Computer Science and Engineering, which allowed me the flexibility to try out this approach on undergraduate students at the University of New South Wales (seven times now...) and similarly the Trustworthy Systems Group, at the time part of CSIRO's Data61 but now at UNSW, who allowed me time to teach the course, to write this text and which provided the right intellectual environment for an experiment of this kind and –indeed– provided many of the students who over several years have attended and helped to improve the course.

Improvements and corrections to the text have been suggested by the students of COMP6721 in 2021 and 2022, by Annabelle McIver and by Thomas Kunc.

Tom in particular was essential in holding the delivery and presentation of the course together in 2022 as we made the transition from a small class size to a much larger one, suggesting the right approach to the current student cohort and how we should structure our automated assessment of their work.

Dedication

This introduction to *reasoning* about programs –rather than just writing them– is dedicated to the men and women whose insights into how to employ both simplicity *and* power, at the most fundamental level, will eventually be recognised by *everyone* as the true determinant of how computers should be designed, programmed and deployed. Only a small number of them are cited here directly: they are indicated by bold-face entries in the index.

 Copyright © Carroll Morgan 2021

Contents

Preface iii
What is this text about, and who is it for? . . . iii
Organisation and use . . . vii

List of Figures xvi

I Everyday programs 1

1 Programs that work 3
1.1 Introduction . . . 3
1.2 Jack be nimble, Jack be quick... . . . 3
1.3 When does a program "work"? . . . 4
1.4 Requirements and specifications . . . 5
1.5 Assertions . . . 7
1.6 Making it happen . . . 10
1.7 Assertions for loops . . . 10
1.8 Avoiding infinite loops . . . 14
1.9 Summary . . . 15
1.10 ***Exercises*** . . . 16

2 Using invariants to design loops 23
2.1 Introduction . . . 23
2.2 The longest *Good* subsegment . . . 23
2.3 First program . . . 24
2.4 Second program . . . 26
2.5 Third program . . . 27
2.6 ***Exercises*** . . . 29

3 Finding invariants 31
3.1 Introduction . . . 31
3.2 Split a conjunct . . . 31
3.3 Introduce a new variable . . . 33
3.4 Simplify the postcondition . . . 38
3.5 Cascading invariants . . . 42
3.6 ***Exercises*** . . . 45

Copyright © Carroll Morgan 2021

4 Finding variants **51**
4.1 Introduction . . . 51
4.2 Simple variants . . . 51
4.3 Lexicographic variants . . . 53
4.4 Structural variants . . . 54
4.5 General variants . . . 54
4.6 ***Exercises*** . . . 55

5 Assignments and conditionals **59**
5.1 Introduction . . . 59
5.2 Assignments . . . 59
5.3 Do nothing . . . 61
5.4 Conditionals . . . 61
5.5 ***Exercises*** . . . 63

6 Summary of Part I **67**
6.1 What's not obvious . . . 67
6.2 What *is* obvious . . . 69

II Data structures **71**

7 Introduction **73**
7.1 Data vs. data *structures* . . . 73
7.2 Data encapsulation . . . 74

8 Coupling invariants **75**
8.1 Introduction . . . 75
8.2 Implementing sets as sequences . . . 75
8.3 Sentinels . . . 77
8.4 Writing abstract data-types . . . 79
8.5 Coding concrete data-types . . . 81
8.6 ***Exercises*** . . . 83

9 Fibonacci numbers **85**
9.1 Definition vs. implementation . . . 85
9.2 Linear-time Fibonacci . . . 86
9.3 Introducing matrices . . . 86
9.4 Logarithmic-time Fibonacci . . . 87
9.5 Remove auxiliary variables . . . 88
9.6 Summary . . . 90
9.7 ***Exercises*** . . . 90

10 Encapsulated data types **93**
10.1 Introduction . . . 93
10.2 Data-type invariants . . . 93
10.3 Rules for encapsulation . . . 97
10.4 Rules justified . . . 100
10.5 More advanced techniques . . . 102
10.6 ***Exercises*** . . . 103

11 The Mean Calculator **107**
11.1 Specification . . . 107
11.2 Implementation . . . 109
11.3 ***Exercises*** . . . 110

12 Summary of Part II **111**
12.1 What is obvious . . . 111
12.2 Specifications and implementations . . . 112
12.3 What is *not* obvious . . . 114
12.4 ***Exercises*** . . . 116

III Concurrency **119**

13 What is "concurrency"? **121**
13.1 Examples of concurrent programs . . . 121
13.2 A minimal concurrent example . . . 122
13.3 What simple locks guarantee . . . 124
13.4 What simple locks actually do . . . 125
13.5 Critical sections . . . 125
13.6 Concurrency and interference . . . 126
13.7 ***Exercises*** . . . 126

14 The Owicki-Gries method **129**
14.1 Introduction . . . 129
14.2 Controlled interleaving . . . 129
14.3 Checking *local* correctness . . . 130
14.4 Checking global *invariants* . . . 130
14.5 Checking *global* correctness . . . 131
14.6 ***Exercises*** . . . 132

15 Critical sections with Owicki-Gries **133**
15.1 Introduction . . . 133
15.2 Single Boolean . . . 133
15.3 Deadlock... . . . 135
15.4 ***Exercises*** . . . 137

16 Peterson's algorithm **139**
16.1 Introduction . . . 139
16.2 Implementation based on a queue . . . 141
16.3 Eliminating multiple assignments . . . 143
16.4 Liveness of Peterson's algorithm . . . 144
16.5 Peterson's algorithm without locks . . . 145
16.6 ***Exercises*** . . . 145

17 Garbage collection on the fly **147**
17.1 Introduction . . . 147
17.2 What "garbage" is . . . 148
17.3 Why garbage needs to be collected . . . 149
17.4 How garbage is collected . . . 149
17.5 Origins of this presentation . . . 150
17.6 Using a Boolean flag . . . 153

 Copyright © Carroll Morgan 2021

17.7 Local correctness . . . 154
17.8 Global correctness . . . 155
17.9 Using a count rather than a flag . . . 157
17.10 Evaluating `|blacks|` non-atomically . . . 161
17.11 Atomicity of the Mutator, in two steps . . . 163
17.12 The completed program . . . 167
17.13 ***Exercises*** . . . 168

IV Checking programs automatically 173

18 Machine-assisted program checking 175
18.1 Introduction and rationale . . . 175
18.2 Automated checking of assertions . . . 176
18.3 Interactive program checking . . . 180
18.4 Exercises . . . 180

Afterword 181

Notes for teachers 182

Appendices 183

A Drill exercises 185
A.1 Sequences and operators on them . . . 185
A.2 Substitution . . . 186
A.3 Invariants . . . 187
A.4 Variants . . . 187
A.5 Propositions, sets and predicates . . . 188
A.6 Thinking outside the box . . . 189
A.7 Hoare triples "in the small" . . . 189
A.8 Hoare triples in the large(r) . . . 190
A.9 Whole-program drills . . . 191

B Rules for checking programs 193
B.1 The basic programs . . . 193
B.2 Assertions . . . 194
B.3 Rules for checking larger program fragments . . . 195
B.4 Rules for checking loops . . . 196
B.5 Concurrency . . . 199
B.6 Exotica . . . 199
B.7 ***Exercises*** . . . 201

C Data refinement 203
C.1 Adding auxiliary variables . . . 203
C.2 Imposing a (data type) invariant . . . 203
C.3 Removing auxiliary variables . . . 204
C.4 Other manipulations . . . 204
C.5 An example . . . 204
C.6 Getting your improved datatype to compile . . . 206

C.7 ***Exercises*** . . . 207

D The arithmetic of conditions **209**
D.1 Introduction and rationale . . . 209
D.2 Why is my program correct? . . . 209
D.3 How do I write my program in the first place? . . . 211
D.4 Calculating with conditions . . . 212
D.5 Simple calculations in logic . . . 214
D.6 Terms . . . 215
D.7 Simple formulae . . . 216
D.8 Propositions, and propositional formulae . . . 217
D.9 Operator precedence . . . 217
D.10 Calculation with logical formulae . . . 219
D.11 ***Exercises*** on propositions . . . 220
D.12 Quantifiers . . . 223
D.13 ***Exercises*** on quantifiers . . . 224
D.14 (General) formulae . . . 225

E Some helpful logical identities **227**
E.1 Some basic propositional rules . . . 227
E.2 Some basic predicate rules . . . 230
E.3 ***Exercises*** on rules for logic . . . 234
E.4 Epilogue on notation and terminology . . . 234

F Illustration of garbage collection **237**
F.1 Mark-and-sweep . . . 237
F.2 Woodger's scenario . . . 237

G Python-specific issues **243**
G.1 Block structure (and assertions) . . . 243
G.2 `for`-loops vs. `while`-loops . . . 243
G.3 Object orientation in Python . . . 245
G.4 Exercises . . . 245

H Exercise answers suppressed in this copy **247**

I Drill answers suppressed in this copy **249**

Index **251**

List of Figures

1.1 Flowchart for summing a sequence 12

2.1 Finding longest *Good* subsegment: first attempt 25
2.2 Finding longest *Good* subsegment: second attempt 26
2.3 Finding longest *Good* subsegment: recommended method 29

3.1 The conditional strict majority found in linear time 41
3.2 Outer structure of maximum segment sum algorithm 43
3.3 The maximum segment sum 44

9.1 Fibonacci as matrix exponential 87

10.1 An array-based implementation of `Set` 96
10.2 Abstract definition of a more robust class `Set` 99
10.3 Concrete version of `Set`, without `assert` 100
10.4 Two versions of class `Set` 101

15.1 Mutual exclusion with `await` and single Boolean 133
15.2 Mutual exclusion with `await` and auxiliary variable `t` 134

16.1 Framework for critical-section program 140
16.2 Peterson's algorithm implemented with a queue 141
16.3 Encoding queue `qs` as three Booleans `t1`,`t2` and `t`. 142
16.4 Peterson's algorithm. 144

17.1 Reference-counting misses some garbage 151
17.2 Why *garbage collection on the fly* needs mark-bits 152
17.3 Interference between Mutator and Scanner 156
17.4 Interference between Mutator and Scanner 158
17.5 Scanner Version 2 159
17.6 "Approximate" black-counting specification 161
17.7 Scanner Version 3 162
17.8 Scanner Version 4: final version 167
17.9 Scanner propagation alone, without assertions 168

18.1 Binary-search program in Dafny 177
18.2 Dafny verifies binary search 177

Copyright © Carroll Morgan 2021

18.3 Dafny rejects incorrect binary search 178
18.4 Dafny rejects incorrect binary search 178
18.5 Dafny rejects incorrect binary search 178

D.1 Intersection of sets: a Venn diagram 213
D.2 Intersection of *disjoint* sets, i.e. with no elements in common 213
D.3 Some terms 215
D.4 Some simple formulae 216
D.5 Truth tables for propositional connectives 218
D.6 Some propositional formulae 218
D.7 Some general formulae 226

F.1 Illustration of mark-sweep 238
F.2 Woodger's scenario 241

Index

`[low:high]`
index range in sequence is inclusive/exclusive, **4**
$\sum$ means "the sum of", **5**
`# -` gives the program's requirements, **5**
$\Rightarrow$ is "implies"
not usually having an equivalent symbol for Booleans in a program, 219
$\Rightarrow$ is "implies", **8**
$\Rrightarrow$ is "implies" as well (for now), **8**
but is actually entailment, 219
`//` is (integer) quotient, **16**
`%` is remainder, **16**
`+` is sequence concatenation, **75**
`+` is sequence concatenation, **19**
`**` means "to the power of", **38**
`[x]*r` makes a sequence of `r` `x`'s, **41**, 96∘
$\equiv$ is "iff" (for now), **61**
but is actually equivalence,
and does not operate between Booleans, **219**
$\in$ membership
of bag, **108**
of set, **75**
`-` difference
of two bags, **108**
of two sets, **76**
$\cup$ union
of two sets, **75**
`{}` is the empty set, **75**
`+` sums two bags, **108**
$\perp$ is "bottom"
meaning "no thread", **124**, 137
`x:[`*pre*`,`*post*`]`, specification, **113**
`$b` as "lock variable" for `b`, **124**
$\wedge$ is conjunction, corresponding to `and` between Booleans in a program, 214, **227**
$\vee$ is disjunction, corresponding to `or` between Booleans in a program, **217**, 220, 227
$\Leftrightarrow$ is iff-and-only-if, corresponding to `==` in a program, **217**
$\neg$ is negation, corresponding to `not` applied to a Boolean in a program, **217**
$\forall$ for all, *see* quantification
$\exists$ there exists, *see* quantification
$\rightarrow$ is implication too (but here we write it $\Rightarrow$), **235**
$\vDash$ is semantic entailment (but here we write it $\Rrightarrow$), **235**
$\vdash$ is logical entailment (but here we write it $\Rrightarrow$ as well), **235**

`A[low:high]`
(sub)segment, 29
"`...<x`" "means the whole segment is `<x`", **36**
index range is inclusive/exclusive, 29, 36, 45
`A[0]` is the head of `A`, 141
`A[1:]` is the tail of `A`, 141
`A[n:n]` is the empty sequence, 45, 48
J-R **Abrial**, 181
absorption, 227
abstract data-type, 79–80
abstract/concrete hybrid, *qv*
access modifier, *see* object orientation
accreditation

Copyright © Carroll Morgan 2021

of Engineers, 67
of programmers, 67
`acquire()` in Python, 123↓
aeroplane, *see* building
algorithm
can be incorrectly coded, iv
can be wrong in principle, iv
is the "essence" of a program, iii
(for) all ∀, *see* quantification
"all the way down", *see* mathematics
"`and`" is conjunction, *qv*
antecedent
is `False` means the implication is `True`, 135
left-hand side of implication, **8**, 217
anti-symmetric, **54**
arithmetic, of Booleans, *qv*
arity of operator, function or predicate, **215**
"arrays" in Python, 96
`assert` statement, 97, **114**
at checking time, 95
at run time, 95
how to check, **200**
if it fails, allows unpredictable behaviour, 103
in Python, 95↓
assertion `{···}`, **9**, 7–9, 15, **108**
brackets vs. comment, 20, 117
checked from end towards beginning, 61, 63, 179
compile-time vs. run-time, 176
"housekeeping", 18
how to check, **200**
intermediate, 18, 60, 179
pre- and postconditions are special cases of, 15
several in a row, 13, 61, 62
used as "scaffolding", 28
vertical vs. horizontal format, 9
assignment statement
checked by textual substitution, 8, 13, 16, 17, *see* exercises, 32, 33, 59–61, 64, **193**, 231
examples of, 59
multiple, 10, 86
not available in *C*, 60↓
associativity
of ∧, 227
of ∨, 219, 227
of longest common prefix, 46
of multiplication, 87
`assume` statement
how to check, **200**
with frame, **114**
without frame, **114**
ATM, 129
automated teller machine, 122
in Imperial currency, 53
invariant for, 122
many active concurrently, 122
atomic action, **123**, 122–123, 125, 129, 143
written on a single line, 132, 143, 155
atomicity
implicit if two or more statements are on the same line, 199
removing it, 163–165
auxiliary variable, 81–84, 88–89, 134∘, 134, 197
adding, 81
does not appear in running code, 135
label-like, 143, 144
removing, 82, 88
use in postcondition, 60
for variants, 52, 60, 63
`await` statement, 133–135
how to check, **134**, **199**
implemented with lock, 136

`Bad()`, *Bad*() subsegment, **23**, 23–24
bag, *see* multi-set
basic data types, *qv*
baton passing, *see* program checking
of `True` assertions, 10
JL **Bentley**, 177
binary search, 36–37, 45–47, 63, 74, 76, 83, 93, 102, 104, 112, 177–179
interpreting postcondition, 45
overflow bug, 177↓, 180
trees, 94
black nodes, *see* garbage collection
blame
finding who is responsible for an error, 97
body
of a quantification, **223**
Boolean
arithmetic, 84
expression, 3

honorary, 142, 154↓
Boolean arithmetic, 84
bound variable, 223, **224**, 231
bounds checking, *see* "everyday" errors
`break` to exit loop, 21, 66, 197, 245, 246
bubble sort, 65
is inefficient, 65
bugs
found using `assert` statements, 200
in specification, 109
introduced by carelessness, 163
never-ending `while`-loop, 14
their main source, 179
building
a garden shed, an office block, an aeroplane, 67
vocational skill, 67↓

C programming language, 3↓, 23↓, 148, 150, 195↓
has no built-in sets, 75↓
but Python does, 75↓
has no multiple assignments, 60↓, 90↓, 90, 91
but Python does, 86
pointers, 148
CAS, *see* compare and swap
cascading invariants, 42–44, 86
case analysis
in `if` statements
reason about each `if`-case separately, 37, 41
caterpillar analogy, for *Good* subsegments, 26
`ceil`-ing function, 46
checking
all possible surrounding programs, 114
automatically with a verifier, 164
conditionals, 64
effort can double with each new variable added, 68
in isolation, 62
loop bodies, 13
circular reasoning, 132
not allowed, 131
only apparent, 131, 132
circular structure
made from pointers, 150, 151∘, 168
class definition
data-type invariants, 95
global constants, 103
global variables (references to), 103
local
variables, 79
overall precondition, 95
parameter, 95
procedure precondition, 95
coding
reducing mistakes, v
what coding really is, iii
coffee cups, disappearing, 147, 149
Collector, 149, **150**
comments, v
good comments, 15
some comments are not very good, 5
"What does this bit of code do?" vs. "What's true here?", 69
common sense is built on sound engineering principles, 111
commutative, 227
commutativity
of ∨, 219
of quantifiers, 232
compare and swap, abbreviated *CAS*, **125**, 133, 135, 139
complement, 228
completeness, **236**
concatenation, *see* sequence
concrete data-type, 81–83
concurrency
atomic action
for compare and swap, 125
atomicity, 123
critical section, **125**, 125, 133–137, **139**
implemented with lock, 125
implemented with queue, 141
thread cannot remain indefinitely within it, 141, 144
deadlock, 135–137, 141, 144
fair scheduling, 137
global variables, 139
"in the large" vs. "in the small", 147
in the world generally, 122
interleaving, 121
livelock, 135–137
liveness, 135–137, 141
locks, 123–125

Copyright © Carroll Morgan 2021

mutual exclusion, **133**, 133–135, **139**
implemented with queue, 141
of *ATM*'s, 122
purpose and examples, 121–123
safety, 135–137, 141, 144, 145
sharing variables, 121
simplest example, 121
starvation, 135–137, 141, 144, 146
thread, **121**, 123, 133
within an operating system, 147
condition
used for checking programs, 3
written using Python syntax, 3
conditional (`if`)
examples of checking, 61–63
how to check
with `elif`, 64
with `else`, 64, **196**
without `else`, **195**
conditional strict majority `csm`, 39, 47
conjunct, 11↓, **32**
splitting two, 31
splitting more than two, 32
expression next to "`and`" or ∧, **217**
conjunction
written "`and`" or ∧, 11↓, 32, 217, 227
conjunctive normal form, **229**
consequent
right-hand side of implication, **8**, 217
constant symbols in logic
are as usual, 215
written in sans-serif, 215
`continue` to restart loop, 21, 245
continued fraction, *see* program exercises
contrapositive, 229
counting black nodes, *see* garbage collection
coupling invariants, **76**, 75–84, 86, 88, 91, 93–95, 101, 103, 105, 109, 110, 114–116, 137, 245
explain how data-type operations cooperate, 76
for implementing mutual exclusion, 142
global variables (references to), 103
keeping only the last six digits, 91
linking matrices to scalars, 90
course assignments
a sample of, vii
critical section, *see* concurrency
`csm`, *see* conditional strict majority

Dafny, 176–179
data refinement, *qv*
data structures, 73–74
triples, 73
data types
basic, **73**
primitive, **73**
structured, **73**
their encapsulation, 74, 76, 78
data-type encapsulation, 93–105
as a "locked box", 90, 115
case studies
pocket calculator, 107–110
set (data type), 93–100
global variables (references to), 103
hides internal state, 90
local
procedures, 96∘, 96, 98
variables, 79, 98
more advanced techniques, 102–103
rules for
justification of, 100–102
summary of, 97–100
data-type invariant, **95**, 93–97
always results in a refinement, 102, 105
ensures consistency, 93
established by class initialisation, 95
implicit pre- and postcondition of every class procedure (method), 95
maintained by every class procedure (method), 95
might not be maintained, 245
sometimes allowed to be broken, 104
data-type representation
vs. encapsulation, 94
date
calculating "today" from day number, 3, 4, 21, 22, 32, 45
calculation has no variant?, 15, 21
leap-year bug, 3, 14
De Morgan's laws, **228**
deadlock, *see* concurrency

debugging the programming *process*, v, 4
defensive programming, 97, 99, 200
determinant of empty matrix, *see* exercises
dictionary order, 54
EW **Dijkstra**, 151↓, 163↓, 181, 214↓
  Golden Rule, 230↓, 230
`DiM()`, days in month, 31
disasters and their causes, 108
disjunct
  expression next to "`or`" or ∨, **217**
disjunction
  written "`or`" or ∨, 217, 220, 227
disjunctive normal form, **229**
distribution, 228
distributive rule
  from arithmetic, 223
division, *see* quotient and remainder
dominoes, falling, *see* program checking
dynamic memory allocation, *see* `malloc()`, `free()`

∑ (Greek capital "sigma") means "the sum of", **5**
efficiency and neatness, *see* time complexity
  after correctness is easier, 35–36
  "tweaking" for, v, 24, 35, 43, 84, 89
empty matrix, *see* exercises, determinant of
empty sequence or subsegment
  largest element of, 6, 7
  maximum of, 20
  product of, 19
  sum of, 19, 44, 48
encapsulated data type, *see* data-type encapsulation
entailment (⇛), **8**, **219**
  different from implication ⇒, 217↓, 219
  distribution of, 220
equivalence (≡)
  chain of, 219
  does not operate between Booleans, 219
  is not a propositional connective, 219, 230↓
Euclid's algorithm gcd, 52
"everyday" errors, 17
  index out of bounds, 4, 17, 23, 43, 48, 145
    is sometimes acceptable, 97
  uninitialised variable, 47
exchange sort, 66
  is inefficient, 66
executing code "in your head", 80
exercises (assorted)
  `assert` statement, 103
  assertion `{···}`
    brackets vs. comment, 117
  assignment-statement checking, 17
  binary search (degenerate), 83
  breaking data-type invariants, 104
  delete `del()` from the Mean Calculator, 110
  determinant of empty matrix, 19
  facts about sequences and ∑, 17
  linear search, 83, 84
  loop invariants, 83
  minimum vs. minimal, 58
  postcondition, 83
  precondition, 83
  `remove()` element from set, 103
  requirements, 83
  set membership, 83
  specification used as program text, 116
  summing a sequence, 19
    if empty, 19
  `undo()` procedure for Mean Calculator, 110
  variants, 83
existential quantification, *qv*
(there) exists ∃, *see* quantification
expressions
  how to check for undefined, **195**
"eyeballing" an implication, 19, 21

`F`, free chain start, *see* garbage collection
false
  unit of ∨, 228
  zero of ∧, 228
false describes no states, **216**
"false", *see* "true"
Fibonacci numbers, **85**, 85–91
  in logarithmic time, 88–89
finite sets of integers, 58
`floor` function, 46

Copyright © Carroll Morgan 2021

flowchart, 10, 13
checking *between* the boxes vs. checking *inside* the boxes, 69
very helpful intuitively, 69
flowchart, making one, 21
RW **Floyd**, **10**, 181
for all ∀, *see* quantification
`for`-loop
assertions for, 47
cannot execute forever, 14
how to check, **198**
vs. `while`-loop, 47, 103, 198, 243
check the `while`, then convert to `for`, 44
`for`-loop vs. `while`-loop, 19
Formal Methods
a fascinating topic in its own right, iv
informally, v
"The Lost Art", *qv*
what Formal Methods is, iv
formula, in logic
atomic, **216**
general, **225**
propositional, 217
frame of a specification, 113
`free()`, `malloc()`, 148 ff
free chain `F`, *see* garbage collection
free variable, **224**, 231
"fresh" means "not already used for something else", **125**
frozen music player, 4
function symbols
are as usual, 215
function symbols in logic
written in sans-serif, 215
function, higher-order, **217**

garbage
caused by pointers, 148
is in the heap, 148
sometimes called "space leak", 149
what garbage is, 148–149
why garbage needs collection, 149
garbage collection, 147–171
Collector, **149**, 149
complete vs. safe, 168
`F`, free chain start, **153**
free chain, 149
starts at `F`, **153**
global correctness, 166
label-like auxiliary variable, 164
mark-sweep, 149
Mutator, **150**
might be more than one, 170
Mutator-Collector interference is not possible, 150
Mutator-Scanner interference, 150, 152∘, 153, 155, 156∘, 156, 158∘, 161
`N`, null pointer target, **153**
nodes, **151**
black, **153**
counting black nodes, 161–163
white, **153**
null pointer target `N`, **153**
"on the fly", **150**
reachable node, 149, **151**
reference counting, 150
root node, **151**
safe vs. complete, 168
Scanner, **149**, 153–155, 157–160
global correctness, 155–157, 159–160, 162, 165, 169
invariants for, 154, 155, 157, 163, 165, 166, 168
local correctness, 154–155, 157–159, 161–162, 164–166, 169
variants for, 154, 157, 168
Version 1 (no), 154, 157
Version 2 (ok, but), 159, 160
Version 3 (ok, but), 161, 162
Version 4 (yes), 168
Version 4 (yes, finally), 167
Scanner-Collector interference, 169
swinging a pointer, **151**
garden shed, building, *qv*
GC, *see* global correcteness
`gcd`, greatest common divisor, **52**
global
postcondition, *qv*
precondition, *qv*
global correctness, 126, **131**, 131, 169, 199
abbreviated *GC*, 131
global invariance, 130–131
global invariants, **130**, 142, 144
compare globally correct, stable, 146
global variables, 103, 139
globally correct
compare stable, global invariants,

146
KF **Gödel**, 179↓
Golden Rule, *see* EW Dijkstra
Good subsegment, 29
greatest common divisor `gcd`, 52
variant, 52
D **Gries**, 151↓, 181

handle, turning it, 10
head-scratching
"Use the assertions, Luke.", 36, 155
heap, *see* garbage (collection)
hint in proof, 219
CAR **Hoare**, 146, 181, 189–191
"housekeeping" assertions, *qv*
how to check programs, summary and definitions, **193**
hybrid of abstract- and concrete, 81

idempotent, 219, 227
`if ··· else: skip`, 63
`if`-branches
reason about each `if`-case separately, 37, 41
IFM, acronym for "(In-)Formal Methods", vii
"Imperial" currency: pounds, shillings and pence, 53
implication
does not usually have an equivalent symbol for Booleans in a program, 217
five different meanings of *??*, 236
written (⇒), **8**
different from entailment ⇛, 219
implicit increment, *see* `for`-loop
(In-)Formal Methods, iii
some of its benefits, 68
what it is, v
indexing is from zero, 4
"indexitis", 192↓
infinite loop, *see* loop variants
infinitely descending chain, **54**
infinity ∞
getting rid of, 48
used as value in program, 48, 49
inlining, *see* procedure calls
insertion sort, 66
is inefficient, 66
interference, **126**, 126, 131, 141, 143, 199
between Mutator and Scanner, *see* garbage collection
interleaving, 123, 129–130, 137
between lines of code, 129↓
intermediate assertion, 18
internal state
hidden by encapsulation, 90
internet, program "fixes", 4
interruption
without, 134
interrupts, 126
controlling them for mutual exclusion, 139
introduce a new variable, 33–37, 40
introductory course, vii
invalid, *see* valid
invariants (and finding them), 21, 31–49
associated with locks, 124
cascading, 42–44, 86
choose them *before* coding, 31
coupling, *qv*
data-type, *qv*
even simple ones are helpful, 38, 42
for recursion, 85, 90
global, *qv*
helps to design loops, 27
introduce new variable, 33–37
must usually be supplied by the programmer, 179
simplify postcondition, 38–41
split conjunct, 31–33
tail invariants, 47
involution, **228**
isolation, checking components in, *qv*
iterating
down, 45, 60, 63
towards the middle, 36–37
up, 33, 43, 60, 86, 154

Jack
"be nimble", 3
should be more careful, 4, 32, 51, 157
Java programming language, 245

DE **Knuth**, 177
König's Lemma, *see* termination

label-like auxiliary variable, *qv*
`lad`, *see* largest absolute difference

Copyright © Carroll Morgan 2021

L **Lamport**, 151↓
largest absolute difference lad, 34
largest element in a sequence, 6, 7
  in a non-empty sequence, 7
LC, *see* local correcteness
leap-year bug, *see* date, 3, 14
length of longest run lr, 40, 48, 116
length of shortest run sr, 49
lexicographic variant, **53**, *see also* termination, 57
  its underlying order, 56
  sometimes can be reduced to simple, 57
  sometimes can't be reduced to simple, 57
lgs(n), longest *Good* subsegment, **27**
linear search, 74, 76, 77, 79, 83, 84, 112
  sometimes too inefficient, 73
linear-time exponential, 38
list of variables, *qv*
livelock, *see* concurrency
liveness, *see* concurrency
local
  correctness, 126, 169
    abbreviated *LC*, 130
  procedures in encapsulation, 96∘, 96, 98
  variables in encapsulation, 79, 98, 108
local is not a declaration in Python, 79↓
local def, 96∘
  find(s), 112
lock variable
  variable $b associated with lock b, 133
lock-free implementation, 146
"locked box", *see* data-type encapsulation
locks
  adding them, 122–123
  implemented with *CAS*, 125
  lock statement, 123
  lock variable
    variable $b associated with lock b, 124
  unlock statement, 123
  what locks guarantee, 124
log-time exponential, 38
logarithmic, *see* time complexity
longest run, *see* program examples
longest *Good* subsegment lgs(n), 23–28
loop
  checking for entry, 13
  checking initialisation, 13
  checking the body (invariant maintained, variant decreased), 13
  conventional thought processes for its design, 68
  early exit with break, 35
  exit, 13
  (avoiding) infinite, 22, *see* loop variants
  invariants, 10–13, 23–49, 93
  unconventional thought processes for its design, 68
  unrolling
    preserves correctness, 35
  variants, 14–15, 51, 55, 60, 63, 66, 197
    for spinlock, 127
"The Lost Art", iv↓, 175, 181
lr, longest run, **48**

majority voting csm, 47–48
malloc(), free(), 148 ff, 153
AJ **Martin**, 151↓
mathematics all the way down, iv, 174–181
matrix multiplication
  used to calculate Fibonacci numbers, 86
max takes the maximum of two elements, 6
MAX takes the maximum of many elements, 6
maximum
  of empty sequence, 20
  of three values, 16
  of four values, 64
  of whole sequence, 16, 20
maximum end sum mes, 43
maximum segment product xsp, 48
maximum segment sum mss, 42–44
  in $O(N^3)$, 42
  in $O(N^2)$, 43
  in $O(N)$, 44, 48
  specification, 42
maximum up to here muh, **42**
The Mean Calculator, 107–110
mes, *see* maximum end sum

method
get- and set- , 98
of a class, 81
minimal has no others less than it, 54, 58
minimum is less than all others, 54, 58
J **Misra**
and Peterson's algorithm, 141
modularity and re-use, v
Modus Ponens, **235**
is a rule of inference, 236
motorcycle
tuning your own, 67
`mss`, maximum segment sum, **42**
`muh`, *see* maximum up to here
multi-set (also bag), 107, **108**
multiple assertions, how to check
on one line, **194**
on successive lines, **194**
multiple assignment, *qv*
music player, frozen, 4
Mutator, **150**
mutual exclusion, 133, 134∘

`N`, null-pointer target, *see* garbage collection
negation, written "`not`" or ¬, 217
nodes, *see* garbage collection
non-empty-segment sum, 48
null pointer `N`, *see* garbage collection

OB abbreviates "other business", 133
object orientation, 245
What's it for?, 111
. . . is not discussed generally in this text, 111↓
access modifier, 245
package, 245
private, 245
protected, 245
public, 245
class, 245
attribute, 245
instance, 245
method, 245
static, 245
in `Python`, 245
one public procedure calling another, 245
office block, *see* building
one-point laws, **232**
other business, abbreviated *OB*, 133
outside-in design, *see* program
overflow bug, *see* binary search
S **Owicki**, 151↓, 181
Owicki-Gries Method, 126, 129–137

partial correctness, **51**
partial order, **54**
`pass`, *see* `skip`
pence, *see* Imperial
GL **Peterson**, 139–146
his mutual-exclusion algorithm, 141–145
"poised" between atomic statements, 143
cannot deadlock, 144
complete code for, 144∘
is safe, 144
label-like auxiliary variable, 143
prevents starvation, 144
pointers, *see C* programming language
null, swinging, *see* garbage collection
"poised" between atomic statements, *see* Peterson's algorithm
polygons puzzle, 58
POST indicates postcondition, 6
postcondition, *see also* program, 6, 15, 21
assertion labelled POST, 6
at the end of the program, 5
global, 129, 130
interpreting, 45
that depends on the precondition, 52, 60
use of auxiliary variable, 52, 60
`#` POST ···, 7
pounds, shillings and pence, *see* Imperial, program examples, *see* program examples
to the power of `**`, **38**
PRE indicates precondition, 6
precedence
of operators, **217**
precondition, *see also* program, 6, 15, 21
assertion labelled PRE, 6
default is `True`, 7, 8, 59, 64
global, 129, 130
`#` PRE ···, 7
predicate
in logic, **216**

Copyright © Carroll Morgan 2021

vs. predicate symbol, 216
predicate symbols
written in sans-serif, 216
primitive data types, *qv*
procedure calls
inlining, 80
program
checking
dividing a big programming problem into a tree of smaller ones, v, 8, 16, 17, 69
. . . then putting the pieces back together, 8
like baton passing, 10
like falling dominoes, 10
counter, 10
flowing, *see* flowchart, 10
its purpose stated in requirements, 5
outside-in design, 27, 36
postcondition, 5–7
precondition, 5–7
default is `True`, 8
requirements, 5–7
specification, 7
"What is a program for?", 5
"When does a program work?", 4, 15
program algebra, 48, 104
program components
`for`-loop vs. `while`-loop, 11, 243
assignment statement `x=` *exp*, 59–61
`break`, 21, 26∘, 45, 48, 53, 66, 76, 82, 136, 145, 154↓, 245
its interaction with invariants, 21, 245
conditional (`if`), 61–63
`continue`, 21, 245
its interaction with invariants, 245
`for`-loop, 11
`skip` (do nothing), 61
`while`-loop, 11
program examples
ATM
in Imperial currency, 53
many active concurrently, 122
binary search, 36–37, 47, 63, 74, 76, 93, 102, 104, 177
bubble sort, 65
circular structure
made from pointers, 168
critical section, 133, 137
date
calculating "today" from day number, 3, 21, 22, 32
calculation has no variant?, 15
Euclid's algorithm `gcd`, 52
exchange sort, 66
Fibonacci numbers, 85, 86
in log-time, 87–89
recursive version is inefficient, 85
`for`-loop vs. `while`-loop, 19, 244
garbage collection, 147–171
Good subsegment, 23
caterpillar analogy, 26
greatest common divisor `gcd`, 52
insertion sort, 66
largest absolute difference `lad`, 34
largest element in a sequence, 6, 7
in a non-empty sequence, 7
leap-year bug, 3, 14
length of longest run `lr`, 40
length of shortest run `sr`, 49
linear search, 74, 76, 77, 79
linear-time exponential, 38
log-time exponential, 38, 86, 87
log-time Fibonacci, 85–91
longest *Good* subsegment `lgs(n)`, 23–28
loop
checking body (invariant maintained, variant decreased), 13
checking for entry, 13
checking initialisation, 13
majority voting `csm`, 39–41, 47–48
maximum
of three values, 16
of four values, 64
of whole sequence, 20
maximum segment sum `mss`
in $O(N^3)$, 42
in $O(N^2)$, 43
in $O(N)$, 44, 48
mutual exclusion, 133, 134∘
Peterson's mutual-exclusion algorithm, *see* concurrency, *qv*
quotient and remainder (finding), 16, 56
rectangular-matrix puzzle, 55
search

binary, *qv*
linear, *qv*
set (concrete)
represented by fixed-size array, 96∘
sorting
three variables, 61, 64
four variables, 65
summing a sequence, 4, 5, 11, 14, 18, 52
swap two variables, 63
train-carriages puzzle, 56
unrolling a loop, 7, 9, 10
vertical format for assertions, 9
`while`-loop vs. `for`-loop, 19
program exercises
binary search, 45–83
Boolean arithmetic, 84
bubble sort, 65
checking conditionals, 64
circular reasoning, 132
continued fraction, 20
coupling invariants, 91, 110, 116, 137
critical section, 127, 137
date
calculating "today" from day number, 22, 45
calculation has no variant?, 21
empty sequence or subsegment
sum of, 48
"everyday" errors, 17
exchange sort, 66
flowchart, making one, 21
`for`-loop, assertions for, 47
global correctness, 169
index out of bounds, 17
infinity ∞ used as value in program, 48
insertion sort, 66
intermediate assertion, 18
iterating
down, 45
iterating down, 18
length of longest run `lr`, 48
local
correctness, 169
lock-free implementation, 146
maximum
of empty sequence, 20
of three values, 16
of four values, 64
of whole sequence, 16, 20
maximum segment product `xsp`, 48
non-empty-segment sum, 48
program algebra, 48, 104
quadratic complexity, 48
(all roads lead to) Rome, 47
rounding up or -down, 46
safety, 145
sequential composition of programs
how to check, 17, 63
shortest run `sr`, 49
sorting, 45, 65
three variables, 64
four variables, 65
a whole sequence, 65, 66
spin lock, 126
stable condition, 146
summing a sequence, 18
swap two variables, 63
tail invariant, *see* invariants
termination, 58
dictionary order, 57
lexicographic variant, 57
lexicographic variant sometimes can be reduced to simple, 57
multi-set ordering, **58**
polygons puzzle, 58
quotient and remainder (finding), 56
social-distance puzzle, 55
train-carriages puzzle, 56
well founded sets, 57, 58
train-carriages puzzle, *see also* termination
uninitialised variable, 47
variants (and finding them), 168
decreasing, 63
for infinite loop, 127
increasing, 22
program review, 4
proof by induction, 114↓
proof, hint in, *qv*
propositional connective, 217
prototype
"quick and dirty" refined later to a more sophisticated implementation, 93
pseudocode

Copyright © Carroll Morgan 2021

does not allow rigorously reasoning, 80
more than, 79–80, 94, 107, 112
Python, 3
caveats — take care, 243
`font`, **3**
mentioned mainly in footnotes, 3↓
used in this text, v, 243
uses reference counting for garbage collection, 150↓

quadratic complexity, 48
quantification
body of, **223**
existential (∃), **224**
nested, **225**
typed, **224**
universal (∀), **223**
untyped, **224**
"quick and dirty", *see* prototype
quick fixes
avoid them, 35
quotient and remainder (finding), 56
integer quotient `//`, *qv*
remainder `%`, *qv*

`range(low,high)` is inclusive/exclusive, 4
`range(1,N)`, 6
default `low` is zero, 4
reachable node, *see* garbage collection
rectangular-matrix puzzle, 55
recursion, 85
refinement
data-, **102**
equality is a special case, 102
guaranteed by adding data-type invariants, 102
of code, **102**
proper, **102**
reflexive, **54**
`release()` in Python, 123↓
`remove()` element from set, 103
`repeat`-loop, 154↓
invariant need not be true on initialisation, 154, 157
requirements, 5, 7, **15**, 21
examples, 24
indicated by "`# -`", 5
should not be too detailed or technical, 15, 107, 108
vs. specification, 16
whether to buy a thing, vs. how to use it, 108
rolling-up a loop, 11
(all roads lead to) Rome, *see* program exercises
root node, **149**
rounding up or -down, 46
Rubik's Cube, 55
rule of inference, *see* Modus Ponens

safety, *see* concurrency
satisfaction modulo theories (*SMT*), 179↓
Scanner, **149**
CS **Scholten**, 151↓
search
binary (logarithmic), *qv*
linear, *see* program examples
(sub-)segment, *see* sequence
sentinel, **77**, 77–79
sequence concatenation `+`, **19**, **75**, 76
sequence or subsegment
`A[0:N]`, 4
`A[low:high]` is inclusive/exclusive, **4**, 29, 36, 45
empty, *qv*
is `A[n:n]`, 45, 48
"falling off the end", 77
sequential composition of programs
how to check, 17, 63, **195**
in Python, in *C*, 195↓
sequential programs (not concurrent), 130
serial access (to a class), 125
serial use (in a concurrent system), 122, 123
set (concrete)
represented by fixed-size array, 96∘
set data-type
abstract, 79
compared with concrete, 99
as primitive
difference `-`, **76**
membership ∈, **76**
union ∪, **75**
concrete
represented by fixed-size array, 94, 96∘
represented by sequence, 75–77, 81, 93
empty `{}`
written `set()` in Python, 75

shillings, *see* Imperial
shortest run sr, **49**, *see also* program exercises, 246
$\sum$ means "the sum of", **5**
simple formula, *see* formula, atomic
simplify postcondition, 38–41
`skip`
  called `pass` in Python, 62↓, 194↓
  checking by implication, 61, 63, **194**
  default `else` of conditional, 63, 123
  does nothing, 61
SMT, *see* satisfaction modulo theories
JLA **van de Schnepscheut**, 151↓
social-distance puzzle, *see also* termination, 55
sorting, 45, 65
  three variables, 61, 64
  four variables, 65
  a whole sequence, 65, 66
  bubble sort, 65
  exchange sort, 66
  insertion sort, 66
  inefficient, *see* bubble sort, exchange sort, insertion sort
soundness, **235**
space leak, *see* garbage
specification, *see also* program, 5–9, 15, 40↓, 161
  as contract, 7
  examples, 24
  how to check, **200**
  implementing, 169
  satisfying, 15
  used as program text, 112–114, 116, 161∘, 161, 169
    vs. `assert` then `assume`, 116
  vs. requirements, 16
  written `x:[`*pre*`,`*post*`]`, **113**
  written in mathematical language, iv
spin lock, 126, 135, 136
split a conjunct, 31–33, 40, 155
spot (·) in quantification, **223**
sr, shortest run, **49**
stable condition, 145, 146
  compare globally correct, global invariants, 146
starvation, *see* concurrency
state space of program
  maps variables to values, 215
static class, *see* object orientation
EFM **Steffens**, 151↓
N **Stenning**, 163↓
structural variant
  sometimes can be reduced to simple, *see also* termination, 58
structured datatypes, 54
subscript out of range, *see* "everyday" errors
substitution
  explains quantifiers, 231
  for checking assignment statements, *qv*
  into formula (notation for), 231
  into quantified formulae, 232
  kept separate from reasoning about assertions themselves, 59
  variable capture, 232
sum of no terms is zero, 32
summing a sequence, 4, 5, 11, 14, 18
  loop variant for, 52
swap two variables, 60, 63
swinging a pointer, *see* garbage collection
synchronised methods, 125
syntax errors, iii

tail invariant, *see* invariants
term, in logic, **215**
termination, 27, 51–55, 58, 116, 155, 157, 159, 162, 197, 198
  dictionary order, 57
  lexicographic variant, 57
    sometimes can be reduced to simple, 57
  multi-set (also bag) ordering, **58**
  polygons puzzle, 58
  quotient and remainder (finding), 56
  rectangular-matrix puzzle, 55
  social-distance puzzle, 55
  structural variant
    sometimes can be reduced to simple, 58
  train-carriages puzzle, 56
  well founded sets, 57, 58
testing is necessary, v, 175
there exists ∃, *see* quantification
thread, **121**, 122–123
  guarantees execution order of its components, 123, 129

Copyright © Carroll Morgan 2021

time complexity (efficiency)
logarithmic $O(\log N)$, 38, 39, 86
linear $O(N)$, 39, 40, 43, 44, 48, 85
$O(N \log N)$, 40
quadratic $O(N^2)$, 43, 48
cubic $O(N^3)$, 42
exponential $O(c^N)$, 85, **90**
today, *see* date
total correctness, **51**
total order, **54**
train-carriages puzzle, 56
transitive, **54**, 219
dividing a big programming problem into a *tree* of smaller ones, 9, 16
True
written True as value of logical formula in a state, **216**
"true"
used as English word, 210
written `true` as a logical constant symbol, **216**
written `True` as Boolean in Python, **209**, 216
written True as value of logical formula in a state, **216**, 222
`true` describes all states, **216**
unit of $\wedge$, 228
zero of $\vee$, 228
`True` as default precondition, *qv*
Trustworthy Systems Group, vii
truth table, 217
"tweaking", *see* efficiency and neatness
for correctness (really?), 24

`undo()` procedure for Mean Calculator, 110
uninitialised variable, 47
unit of an operator, 228
universal quantification ($\forall$), *qv*
universally valid, 230
The University of New South Wales, vii
unlock
twice in a row, 124
unrolling a loop, 7, 9, 10, *qv*
ur-polygon, *qv*

"valid" means correct, in logic, 214↓
variable
bound, *qv*, **224**
free, *qv*
in logic written x, 215
in programs written `x`, 215
list of, in quantification, 223
variable capture, *see* substitution
variants (and finding them), **14**, 51–55, 168
bounded above, 22
bounded below, 32
can never be negative, 14
decreasing, 63
dictionary order, 53, 57
for infinite loop, 127
for recursion, 85, 90
increasing, 14, 22, 52, 83
vs. decreasing, 52
lexicographic, **53**, 53–54
multi-set ordering, 58
must strictly decrease, 14, 32, 52
simple, 51–56
structural, 54
use auxiliary variable with, to check decrease, 60, 63, 66
well founded, 54
when there can't be one, 15
verifier, *see* checking programs automatically
vertical format for assertions, 9
vocational skill, 67↓

well formed formula (*wff*), **235**
well founded sets, **54**, *see also* termination, 57, 58
wff, well formed formula, **235**
What does this do? comment, *qv*
"What happens?" vs. "What is true?", 129
What's true here? comment, *qv*
`while`-loop
vs. `for`-loop, 243
how to check partial correctness, **196**
how to check termination, **197**
how to check total correctness, **198**
with `break` statements, **197**
`while`-loop vs. `for`-loop, 19
white nodes, *see* garbage collection
M **Woodger**, 163↓
"working" program, 15

`x:[`*pre*`,`*post*`]`, specification, **113**
used as program text, 113

`yield()`, 126

zero of an operator, *see* unit, 228
Zune, *see* program examples (leap-year bug)

 Copyright © Carroll Morgan 2021