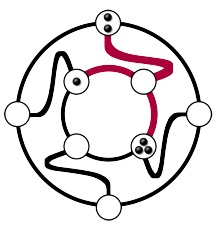
Carroll Morgan

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

How to write programs that work

For COMP6721 in 2022T2 July 25, 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.

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 both 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 a selection of their answers is given in App. H. (A complete set of answers is available, but many of them are reserved as a teaching resource.)

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.

> 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, and by Annabelle McIver.

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.

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

4 Finding variants **49**
4.1 Introduction . . . 49
4.2 Simple variants . . . 49
4.3 Lexicographic variants . . . 51
4.4 Structural variants . . . 52
4.5 General variants . . . 52
4.6 ***Exercises*** . . . 53

5 Assignments and conditionals **57**
5.1 Introduction . . . 57
5.2 Assignments . . . 57
5.3 Do nothing . . . 59
5.4 Conditionals . . . 59
5.5 ***Exercises*** . . . 61

6 Summary of Part I **65**
6.1 What's not obvious . . . 65
6.2 What *is* obvious . . . 67

II Data structures **69**

7 Introduction **71**
7.1 Data vs. data *structures* . . . 71
7.2 Data encapsulation . . . 72

8 Coupling invariants **73**
8.1 Introduction . . . 73
8.2 Implementing sets as sequences . . . 73
8.3 Sentinels . . . 75
8.4 Writing abstract data-types . . . 77
8.5 Coding concrete data-types . . . 79
8.6 ***Exercises*** . . . 81

9 Fibonacci numbers **83**
9.1 Definition vs. implementation . . . 83
9.2 Linear-time Fibonacci . . . 84
9.3 Introducing matrices . . . 84
9.4 Logarithmic-time Fibonacci . . . 85
9.5 Remove auxiliary variables . . . 86
9.6 Summary . . . 88
9.7 ***Exercises*** . . . 88

10 Encapsulated data types **91**
10.1 Introduction . . . 91
10.2 Data-type invariants . . . 91
10.3 Rules for encapsulation . . . 95
10.4 Rules justified . . . 98
10.5 More advanced techniques . . . 100
10.6 ***Exercises*** . . . 101

11 The Mean Calculator **105**
11.1 Specification . . . 105
11.2 Implementation . . . 107
11.3 ***Exercises*** . . . 108

12 Summary of Part II **109**
12.1 What is obvious . . . 109
12.2 Specifications and implementations . . . 110
12.3 What is *not* obvious . . . 112
12.4 ***Exercises*** . . . 114

III Concurrency **117**

13 What is "concurrency"? **119**
13.1 Examples of concurrent programs . . . 119
13.2 A minimal concurrent example . . . 120
13.3 What simple locks guarantee . . . 122
13.4 What simple locks actually do . . . 123
13.5 Critical sections . . . 123
13.6 Concurrency and interference . . . 124
13.7 ***Exercises*** . . . 124

14 The Owicki-Gries method **127**
14.1 Introduction . . . 127
14.2 Controlled interleaving . . . 127
14.3 Checking *local* correctness . . . 128
14.4 Checking global *invariants* . . . 128
14.5 Checking *global* correctness . . . 129
14.6 ***Exercises*** . . . 130

15 Critical sections with Owicki-Gries **131**
15.1 Introduction . . . 131
15.2 Single Boolean . . . 131
15.3 Deadlock... . . . 133
15.4 ***Exercises*** . . . 135

16 Peterson's algorithm **137**
16.1 Introduction . . . 137
16.2 Implementation based on a queue . . . 139
16.3 Eliminating multiple assignments . . . 141
16.4 Liveness of Peterson's algorithm . . . 142
16.5 Peterson's algorithm without locks . . . 143
16.6 ***Exercises*** . . . 143

17 Garbage collection on the fly **145**
17.1 Introduction . . . 145
17.2 What "garbage" is . . . 146
17.3 Why garbage needs to be collected . . . 147
17.4 How garbage is collected . . . 147
17.5 Origins of this presentation . . . 148
17.6 Using a Boolean flag . . . 151

17.7 Local correctness . . . 152
17.8 Global correctness . . . 153
17.9 Using a count rather than a flag . . . 155
17.10 Evaluating `|blacks|` non-atomically . . . 159
17.11 Atomicity of the Mutator, in two steps . . . 161
17.12 The completed program . . . 165
17.13 ***Exercises*** . . . 166

IV Checking programs automatically **171**

18 Machine-assisted program checking **173**
18.1 Introduction and rationale . . . 173
18.2 Automated checking of assertions . . . 174
18.3 Interactive program checking . . . 178
18.4 Exercises . . . 178

Afterword **179**

Notes for teachers **180**

Appendices **181**

A Drill exercises **183**
A.1 Sequences and operators on them . . . 183
A.2 Substitution . . . 184
A.3 Invariants . . . 185
A.4 Variants . . . 185
A.5 Propositions, sets and predicates . . . 186
A.6 Thinking outside the box . . . 187
A.7 Hoare triples "in the small" . . . 187
A.8 Hoare triples in the large(r) . . . 188
A.9 Whole-program drills . . . 189

B Rules for checking programs **191**
B.1 The basic programs . . . 191
B.2 Assertions . . . 192
B.3 Rules for checking larger program fragments . . . 193
B.4 Rules for checking loops . . . 194
B.5 Concurrency . . . 197
B.6 Exotica . . . 197
B.7 ***Exercises*** . . . 199

C Data refinement **201**
C.1 Adding auxiliary variables . . . 201
C.2 Imposing a (data type) invariant . . . 201
C.3 Removing auxiliary variables . . . 202
C.4 Other manipulations . . . 202
C.5 An example . . . 202
C.6 Getting your improved datatype to compile . . . 204

C.7 ***Exercises*** 205

D The arithmetic of conditions **207**
D.1 Introduction and rationale 207
D.2 Why is my program correct? 207
D.3 How do I write my program in the first place? 209
D.4 Calculating with conditions 210
D.5 Simple calculations in logic 212
D.6 Terms 213
D.7 Simple formulae 214
D.8 Propositions, and propositional formulae 215
D.9 Operator precedence 215
D.10 Calculation with logical formulae 217
D.11 ***Exercises*** on propositions 218
D.12 Quantifiers 221
D.13 ***Exercises*** on quantifiers 222
D.14 (General) formulae 223

E Some helpful logical identities **225**
E.1 Some basic propositional rules 225
E.2 Some basic predicate rules 228
E.3 ***Exercises*** on rules for logic 232
E.4 Epilogue on notation and terminology 232

F Illustration of garbage collection **235**
F.1 Mark-and-sweep 235
F.2 Woodger's scenario 235

G Python-specific issues **241**
G.1 Block structure (and assertions) 241
G.2 `for`-loops vs. `while`-loops 241
G.3 Object orientation in Python 243
G.4 Exercises 243

H Exercise answers suppressed in this copy **245**

I Drill answers suppressed in this copy **247**

Index **249**

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 85

10.1 An array-based implementation of `Set` 94
10.2 Abstract definition of a more robust class `Set` 97
10.3 Concrete version of `Set`, without `assert` 98
10.4 Two versions of class `Set` 99

15.1 Mutual exclusion with `await` and single Boolean 131
15.2 Mutual exclusion with `await` and auxiliary variable `t` 132

16.1 Framework for critical-section program 138
16.2 Peterson's algorithm implemented with a queue 139
16.3 Encoding queue `qs` as three Booleans `t1`,`t2` and `t`. 140
16.4 Peterson's algorithm. 142

17.1 Reference-counting misses some garbage 149
17.2 Why *garbage collection on the fly* needs mark-bits 150
17.3 Interference between Mutator and Scanner 154
17.4 Interference between Mutator and Scanner 156
17.5 Scanner Version 2 157
17.6 "Approximate" black-counting specification 159
17.7 Scanner Version 3 160
17.8 Scanner Version 4: final version 165
17.9 Scanner propagation alone, without assertions 166

18.1 Binary-search program in Dafny 175
18.2 Dafny verifies binary search 175

18.3 Dafny rejects incorrect binary search 176
18.4 Dafny rejects incorrect binary search 176
18.5 Dafny rejects incorrect binary search 176

D.1 Intersection of sets: a Venn diagram 211
D.2 Intersection of *disjoint* sets, i.e. with no elements in common 211
D.3 Some terms 213
D.4 Some simple formulae 214
D.5 Truth tables for propositional connectives 216
D.6 Some propositional formulae 216
D.7 Some general formulae 224

F.1 Illustration of mark-sweep 236
F.2 Woodger's scenario 239

Part I

Everyday programs

Chapter 1

Programs that work

1.1 Introduction

This first part of the text deals with "ordinary" programs comprising assignment statements, conditionals and loops, and simple data structures (mainly). The language for the program examples is Python, but the techniques apply to other "conventional" languages that also have those features. Program text will be written in `this font`.[1]

Part of the novelty (perhaps) of what we are doing will be that we use "conditions", which can be thought of as Boolean-valued expressions, to write down what we expect the programs to do. What we call "checking" a program is simply making sure that the program satisfies those conditions. The conditions' syntax will be in Python as well, i.e. just as you would find a condition in an `if`-statement: thus we'll write `x==2` instead of $x{=}2$, and `x!=2` instead of $x{\neq}2$.

Although our aim is to write programs that work, we will in fact begin our study of "everyday programs" with a famous example of one that *didn't* work. It's not hard of course to find programs that don't work — we all write them every day, and the whole point of this text is to help us to do that less often.

More unusual, though, is to find a "doesn't work" program that was in the pockets of millions of people. So that will be our first example.

1.2 Jack be nimble, Jack be quick...

The following program was part of the operating system for a portable music player, of which millions were sold around the world. The variable `d` (for "day") was kept up to date by the player's hardware, giving at all times the number of days elapsed since "epoch" 1 January 1980. When the player was switched on, on of its first tasks was to convert that day-number `d` to a date in calendar style: "the $\texttt{D}^{th}$ day of month `M` in year `YYYY`".

Here is the code for calculating the `YYYY`:[2]

See Ex. 1.16.

[1] And comments about Python specifically will be mainly in footnotes, from here on.
[2] The original program was written in C; aside from that, this version in Python is the exact code except for the added comments.

```
# -- Calculate the year number y for today,
# -- where d is the number of days since 1/1/1980.
y= 1980
while d>365:
  if leapYear(y):
    if d>366:
      d= d-366
      y= y+1
  else:
    d= d-365
    y= y+1
# The current year is y.
```
(1.1)

It was done so that the calendar date could be displayed on the player's screen. (Subsequent code dealt with month and day-in-month.)

Yet on 31 December in the first leap year after the music player was released, every single one of these players, in the whole world, froze when it was turned on. Can you see why?

There are many posts about this error on the internet, and many blog comments attached to them where the program is "fixed" by one reader only to have the next reader find a new mistake in the supposed repair. But the "why" of this error is not the actual mistake in the program: the real cause of it is that programmers are often not given the chance to learn how to avoid simple mistakes like this in the first place. It's the process that *led* to this program that needs to be debugged.

This program should never have been installed. And even if the programmer was new, and inexperienced (but we do not know), more experienced colleagues should have been able to ask simple questions –as part of the review process– that would have revealed the problem.

> The problem is the culture that allowed this program to be written, installed and deployed.
>
> *Jack should have been encouraged to be less nimble, been shown how to be more careful.*

See Ex. 1.16.
See Ex. 3.1.

Can we do better than Jack? Yes. This book shows how.

1.3 When does a program "work"?

Suppose we have a sequence `A[0:N]` of numbers, indexed from `0` inclusive to `N` exclusive, and we want to calculate their sum. Does this program work?

```
s= 0 # Start from zero.
for n in range(N):
  s= s+A[n] # Add the current element.
```
(1.2)

To answer that question, we check various things.

Is `s` correctly initialised? *Yes.* Are all the sequence elements considered, once exactly? And does the indexing remain within the bounds of the sequence? (We remind ourselves that the elements of `A[0:N]` are indexed from `0` up to `N-1`, and that `n in range(N)` iterates over `0,1,...,N-1`.) So, *Yes.* Is the correct operator `+` being used in the loop body? *Yes.*

So –overall– *Yes*. It does work. For a program of this (tiny) size, we can be sure we've checked everything.

Or does it work, really... *What does that mean?* It does run without crashing. But you can't say that something works (or does not) *unless* you know what it's supposed to do. And there's nothing "officially" associated with Prog. 1.2 that says what it's actually *for*. For that, all we have is the text "...we have a sequence `A[0:N]` of numbers, and we want to calculate their sum." written in a paragraph nearby (here, just above it). When asking whether a program works, we have to point not only to the program, but also a description of what it's supposed to do. (See Sec. 18.2.3.)

So it's probably a good idea to make a (rough) description of the program's purpose into *part of the program itself*, say as a comment at its beginning. That would give us

```
# -- Sum the elements of sequence A[0:N].
s= 0                                                    (1.3)
for n in range(N): s= s+A[n]    .
```

(We have removed the two comments already there, which were not really doing anything useful; and so the `for`-loop can now be all on one line.)

Now we *can* now ask simply "Does Prog. 1.3 work?" and it *does* make sense, provided we know we are following the convention that the comment at the front (for now, its *only* comment) is supposed to give the program's purpose. That is, the first line is "officially" where the purpose is to be stated, the *requirements*; and we have here the convention that the " `# --` " identifies it as such. (Any other similar convention would do.)

More than that, though, we'll also write a comment at the *end* of the program that states, perhaps in more precise and technical terms, what the final state of the program is supposed to be: that gives us the program

```
# -- Sum the elements of sequence A[0:N].
s= 0
for n in range(N): s= s+A[n]                            (1.4)
# s == ∑A[0:N]    ,
```

where $\sum$ means "the sum of". The more technical comment is at the end because it is telling us what we expect to be true at the end of the program's execution, and it can be thought of as a more precise, more detailed version of the `# --` comment at the beginning. See Ex. 1.15.

In later sections we will see many more reasons that putting a "postcondition" there, at the end, is a good idea.

1.4 Requirements and specifications, pre- and postconditions

In careful programming, the comment

```
# -- Sum the elements of sequence A[0:N].              (1.5)
```

might be called the *requirements*, and the comment

```
# s == ∑A[0:N]                                          (1.6)
```

would be called a *postcondition*. The requirements say what the program is supposed to do, and the postcondition states something that is supposed to be true (a *condition*) at its end (*post*). That is, the requirements (1.5) express what our customer (or boss) wants the program to do: and "gathering" requirements is an important topic on its own. The postcondition (1.6) states what the program will have achieved when it has ended.

Obviously, requirements and specifications are related; but they do not have exactly the same purpose. We will explain at the end of this section what the difference between those two things is.

For now, however, we look at a third feature — *preconditions*. They go at the front of the program, and interact with postconditions in the following way. Suppose we have the slightly different program shown here, for finding the largest element in a sequence:

```
# -- Find the largest element of sequence A[0:N].
m= A[0]
for n in range(1,N): m= m max A[n]
# m == MAX A[0:N]    ,
```
(1.7)

where in general a `max` b is the maximum of a and b and `MAX` A is the overall maximum of a sequence A of values. [3]

If we check this program, as we checked Prog. 1.2 at the beginning of Sec. 1.3, we would check similar things; but here we have deliberately introduced a slight complication. The requirements comment `# --` doesn't make sense if the sequence is empty, because there can't be a *largest* element if there is no element at all. So –if we are to meet the requirements– we must somehow make sure that this program is not run when `A` is empty, that is when `N` is zero. That's what the precondition is for.

Just as a postcondition states what must be true at a program's end, a *precondition* states what must be true at its beginning. To make it clear which is which, we will (temporarily) write PRE and POST within them:

```
# PRE 0<N
m= A[0]
for n in range(1,N): m= m max A[n]
# POST m == MAX A[0:N]    .
```
(1.8)

You could read the whole thing as

> "If `0<N` is true at the beginning of program (PRE)
>
> ```
> m= A[0]
> for n in range(1,N): m= m max A[n]    ,
> ```
>
> then `m ==` the largest element in `A[0:N]` will be true at its end (POST)."

Reading it that way, i.e. saying "if" this "then" that, makes it clear who has responsibility for making those conditions true: the person who is *using* the program must ensure that the precondition is met *if* the program is to operate correctly — that is, after all, exactly what "if" means. And the person who wrote the program must make sure that (if its precondition holds) *then* its postcondition will hold once the program ends. In the case of Prog. 1.8, that means "If `0<N` then the program will set `m` to the largest element in `A[0:N]`."

[3] In Python "max" is written as a function, thus here `max(m,A[n])`.

And if Prog. 1.8 is run when its precondition does not hold, what then? In that case, the program (and its programmer) is absolved from any responsibility for what happens. That takes us back to requirements: does the program meet them? They were there in Prog. 1.7, that is

```
# -- Find the largest element of sequence A[0:N].
```
,

and it's now clear that they are too strong: the requirements should read instead

```
# -- Find the largest element of non-empty sequence A[0:N].
```
(1.9)

and our overall program becomes

See Ex. 1.14.

```
# -- Find the largest element of non-empty sequence A[0:N].
# PRE 0<N
m= A[0]
for n in range(1,N): m= m max A[n]
# POST m == MAX A[0:N]
```
(1.10)
.

That leaves "specifications" as the last keyword in the title of this section. We'll call the pre- and postcondition pair together the *specification*: you could think of it as

$$\textit{precondition} \implies (\text{program}) \implies \textit{postcondition} \quad ,$$

where if a program is sitting between the two conditions then it should "satisfy" the specification that the pre- and postcondition together express. The difference, between the requirements Prog. 1.9 and the specification $\textit{precondition} \Rightarrow \cdots \Rightarrow \textit{postcondition}$, is then that the requirements can be thought of as the purpose of the program, and the specification can be thought of as a contract — a contract that the programmer has checked the program is guaranteed to meet.

It's crucial that the specification imply that the requirements are met. And of course it's just as crucial that the program be guaranteed to satisfy its specification. Those are however two separate things. In the next section, we will look at the second one: checking the program meets, "satisfies" its specification. That is what the programmer does.

1.5 Satisfying a specification: assertions

We now return to the original Prog. 1.2, but suppose additionally that `N` is 3 (an arbitrary choice) so that we can "unroll" the program. Because `A[0:N]` is now `A[0:3]`, that is `[A[0],A[1],A[2]]`, we get

```
# PRE True
s= 0
s= s+A[0]
s= s+A[1]
s= s+A[2]
# POST s == ∑A[0:3]
```
(1.11)
,

where we have put a precondition `True` to indicate unambiguously that the precondition hasn't simply been left off by accident. Does this program (the middle four lines) satisfy its specification (the first and last lines)?

It's intuitively clear that it does (once we remember –again– that the last element of `A[0:3]` is `A[2]`). But there is a now sense in which we can "prove" that, even

though for such a small program it seems hardly worth it. But we do it anyway, just to show how it is done. We begin by splitting it up into smaller pieces.

The program (1.11) is actually four smaller programs one after the other; and we can give each one a specification of its own. Instead of writing them as conventional comments, here we write them between brackets $\{\cdots\}$ to help separate them textually from the programs in between. Their meaning is the same:

```
                {PRE True}     s= 0     {POST s == ∑A[0:0]}
{PRE s == ∑A[0:0]}  s= s+A[0]  {POST s == ∑A[0:1]}
{PRE s == ∑A[0:1]}  s= s+A[1]  {POST s == ∑A[0:2]}
{PRE s == ∑A[0:2]}  s= s+A[2]  {POST s == ∑A[0:3]}
```

The precondition of the first program is `True`, meaning that we are assuming nothing (because `True` is true no matter what the variables' values are). After that, though, the *post*-condition of each smaller program is the *pre*-condition of the next one, until –at the end– the final postcondition, i.e. of the final statement, is the postcondition of the whole original program.

Let's look at the third small program `s= s+A[1]` now, the one in the middle, all on its own. If we write it vertically, we can include our conditions as comments again:

```
# PRE s == ∑A[0:1]
s= s+A[1]                                                    (1.12)
# POST s == ∑A[0:2]     .
```

Its precondition `s == ∑A[0:1]` will be true just before it is executed, because it is exactly the postcondition of the previous program `s= s+A[0]` . And if this fragment is itself correct, it will establish *its* postcondition `s == ∑A[0:2]`, which is then ready for the `s= s+A[2]` that follows. So each fragment "does its bit", and the next fragment assumes that has been done.

But *does* the small program Prog. 1.12 work? (Yes.) How do we check that?

To find out whether Prog. 1.12 works, we take the postcondition `s == ∑A[0:2]` and simply make the textual substitution given by the assignment statement, i.e. replacing `s` by `s+A[1]`. It's as simple as that, just doing it with a text editor: *find* `s`, then *replace* `s` *with* `s+A[1]`. Literally: just find, then replace. The new text is `s+A[1] == ∑A[0:2]` . Once that's done, we check that the new, edited text is implied by (⇛) the precondition; that is we check that

See Drill A.3.

```
                    implies
—antecedent—                    —consequent—                 (1.13)
s == ∑A[0:1]        ⇛       s+A[1] == ∑A[0:2]      ,
```

which is to say that "Whenever (left of ⇛) the *antecedent* is true, then so is (right of ⇛) the *consequent*." After some simplification, that boils down to just

```
(∑A[0:1]) + A[1]      ==      ∑A[0:2]      ,
```

and of course the checks for the other three small programs (i.e. including the initialisation `s= 0`) are just as simple, as you can see in App. B.1.1.

But now here is the magic: although we checked those program fragments separately,[4] we can be sure they will work when put all together — without checking anything further.

[4] They could even have been checked by different people, at different times and in different places.

All of that is an example of decomposing one *fairly* simple problem into four *extremely* simple ones. Later however we will use the same technique to decompose not-so-simple problems into a number of simple-but-not-trivial ones; and later still, we will decompose complex problems into a number of less complicated ones, and those into less complicated ones still... until eventually we reach (at last) the extremely trivial again.

Overall, we decompose large single complicated problems into a (large) number of very simple ones: it's like a tree with a large trunk (a complicated problem) whose branches split, and split again — getting smaller and smaller until we reach the leaves (many simple problems).

Because we have got used to preconditions and postconditions, and that we can write them either as comments `# ···` or between braces `{ ··· }` as we like, and that we can suppress the PRE's and POST's — we'll now write just

```
{ True }
s= 0
{ s == ∑A[0:0] }
s= s+A[0]

{ s == ∑A[0:1] }
s= s+A[1]
{ s == ∑A[0:2] }

s= s+A[2]
{ s == ∑A[0:3] }
```

→ Recall (1.12) above. (1.14)

for the example above. The `{···}` are called *assertions*, and each one is the postcondition `{`POST `···}` of the program statement before it and the precondition `{`PRE `···}` of the one after: an example is Lines 5–7 of the above, which together give our earlier (1.12) once we re-insert the PRE's and the POST's and use the vertical format. If we wrote the whole thing out in that style, it would become

```
# PRE True
s= 0
# POST s == ∑A[0:0]

# PRE s == ∑A[0:0]
s= s+A[0]
# POST s == ∑A[0:1]

# PRE s == ∑A[0:1]
s= s+A[1]
# POST s == ∑A[0:2]

# PRE s == ∑A[0:2]
s= s+A[2]
# POST s == ∑A[0:3]
```

→ *These are the same.* (1.15)

,

and that is why the whole, single program is correct provided each of its four smaller components is correct.

1.6 Making it happen: just "turn the handle"

An easy way to visualise (1.14) is to imagine the program's execution flowing from one statement to the next: we just follow the program's progress, the program counter's position, as it moves from the program's beginning to its end.

When the program counter "flows through a *statement*," the statement is executed; and when it flows through an *assertion*, that assertion must be true. The second of those –that the assertion must be true– is the key to rigorous reasoning about programs like this. [5]

If each statement does its job properly, then the truth of its precondition guarantees the truth of its postcondition, which becomes in turn the precondition of the next statement... and so on. Taken all together, the truth of the very first precondition, i.e. of the first statement in the program and thus the precondition of the *whole* program, leads inevitably via this "baton passing" of must-be-true assertions to the truth of the postcondition of the very last statement — which is the postcondition of the whole program.

It's like lining up dominoes, checking the adjacent pairs separately to be sure that each one's falling is certain to strike the next. If *all* adjacent pairs have been checked, even if by different people at different times, and no matter how many there are, then pushing the very first "precondition domino" will –eventually– cause the postcondition domino to fall.

1.7 Assertions for loops

Our original Prog. 1.2 from Sec. 1.3 was a loop, and to discuss its correctness we added pre- and postconditions (1.15) and "unrolled" the loop. We now repeat that, but this time we include the variable `n` that is initialised and then incremented "behind the scenes" by the `for n in range(N)` . The initial (multiple) assignment `s,n= 0,0` sets both `s` and `n` to 0 and establishes the postcondition (of the initialisation), i.e. that `s == ∑A[0:n]` which, because `n==0` at that point, is the same as our earlier `s == ∑A[0:0]`. And all the subsequent assertions `s == ∑A[0:1]` and `s == ∑A[0:2]` and `s == ∑A[0:3]` are also replaced by the same `s == ∑A[0:n]`, and for the same reason:

↓ *Now these are all the same.*

```
{PRE True}
s,n= 0,0
{POST n==0 and s == ∑A[0:n]} 6

{PRE s == ∑A[0:n]}
s,n= s+A[0],n+1
{POST n==1 and s == ∑A[0:n]}

{PRE s == ∑A[0:n]}
s,n= s+A[1],n+1
{POST n==2 and s == ∑A[0:n]}

{PRE s == ∑A[0:n]}
s,n= s+A[2],n+1
{POST n==3 and s == ∑A[0:n]}
```
(1.16)

,

[5] That second, complementary point of view was proposed by RW Floyd — and it changed *everything*.

But now we can go even further: we can replace the `A[0]` and `A[1]` etc. in the assignments by `A[n]` in all cases, and we can remove the `n==0` and `n==1` etc. from the assertions. That gives

```
{PRE True}
s,n= 0,0
{POST s == ∑A[0:n]}
```
→ *This is the initialisation, but...*

```
{PRE s == ∑A[0:n]}
s,n= s+A[n],n+1
{POST s == ∑A[0:n]}
```
→ *...this one, and*

```
{PRE s == ∑A[0:n]}
s,n= s+A[n],n+1
{POST s == ∑A[0:n]}
```
→ *...this one and*

```
{PRE s == ∑A[0:n]}
s,n= s+A[n],n+1
{POST s == ∑A[0:n]}
```
→ *...this one are all the same.*

```
# But how do we know that n==3 here?
```

And so we find that all the component programs except the initialisation –and their assertions– are the same. If we could somehow add the postcondition `{ n==3 }` at the very end, we would be able to assert `{ s == ∑A[0:3] }` there as well.

And so, as a final step, to recover our final `# n==3`, we'll "roll (1.16) up" again, but use a `while`-loop rather than a `for`-loop: the advantage of the `while`-loop is that it makes the incrementing and testing of the loop index `n` explicit. With coloured program statements, to make them stand out from the assertions, that gives us

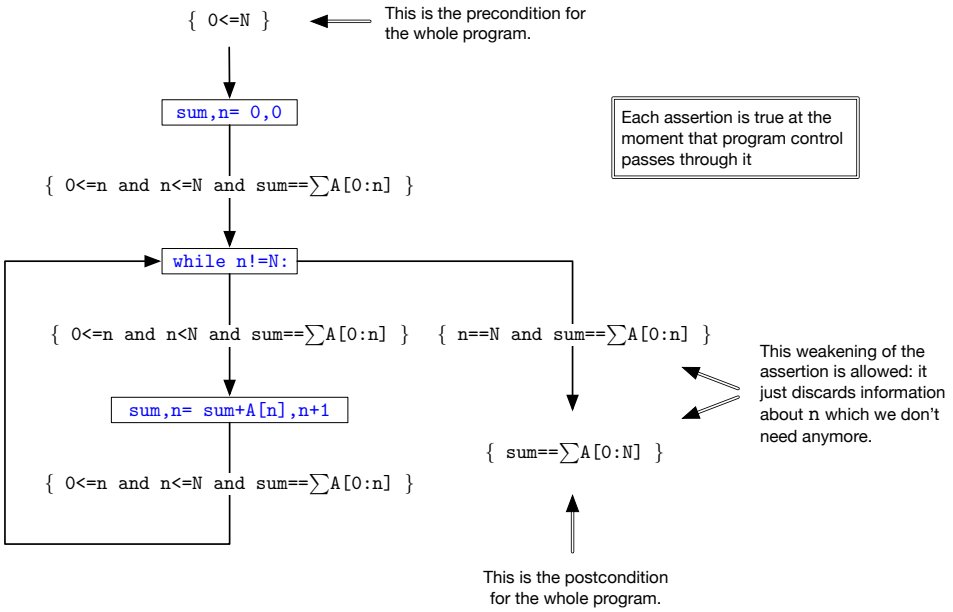
See Ex. 1.2. See Ex. 1.8. See Ex. 1.9.

```
{PRE 0<=N}
s,n= 0,0
{POST 0<=n<=N and s==∑A[0:n]}

{PRE 0<=n<=N and s==∑A[0:n]}
while n!=N:
  {POST n!=N and 0<=n<=N and s==∑A[0:n]}

  {PRE 0<=n<N and s==∑A[0:n]}
  s,n= s+A[n],n+1
  {POST 0<=n<=N and s==∑A[0:n]}

{POST n==N and 0<=n<=N and s==∑A[0:n]}

{POST s==∑A[0:N]}    .
```
(1.17)

Notice that the `while` statement itself has *two* postconditions, because it can send the program execution in either of two directions: one of them enters the loop, where the condition `n!=N` is added; and the other exits the loop, where the opposite condition `n==N` is added. They are underlined for emphasis. (The underlined `n==N` at the end is supplied by the fact that the loop condition must be `False` at that point.)

See Ex. 1.5.

[6] In Python "`and`" is used rather than for example "`&&`" for *conjunction*, to make a single condition that holds just when both of its components (conjuncts) do.



The overall precondition of the program is `0<=N`, at the top; and its overall postcondition is `s==∑A[0:N]`, at the bottom.

If you imagine starting the program in a state where its precondition `0<=N` is true, and then travelling through the program as it executes, you'll notice two things:

(a) Every time you reach an assertion, the program variables make it evaluate to `True` (as a Boolean).

(b) Every time you travel through a statement, it will make its postcondition true (the assertion after it) provided its precondition was true (the assertion before it).

As a result, when the program is finished and you have therefore reached the *overall* postcondition, i.e. of the whole program, it will be true too.

Figure 1.1 Flowchart for Prog. 1.17

Again, the program has been split up into simpler pieces; but this time the number of pieces does not depend on `N`, as it did before (where for example there were four pieces when `N` was 3). Instead of having to write out an instance of the loop body for every time it's executed, we can write it just once — because the pre- and postconditions are the same every time:

– Initialisation

$$\{\text{PRE } 0{\le}N\}\ s,n{=}\ 0,0\ \{\text{POST } 0{\le}n{\le}N \text{ and } s{=}{=}\textstyle\sum A[0{:}n]\}$$

If we treat the assignment statement just as we did at (1.13), and make the substitution, we see that we just have to check that

$$0{\le}N \quad \Longrightarrow \quad 0{\le}0 \text{ and } 0{\le}N \text{ and } 0{=}{=}\textstyle\sum A[0{:}0] \quad ,$$

which is indeed easy.

– Test for loop entry

$$\{\text{PRE } \cdots\}\ \text{while } \underline{n{!}{=}N}\ \{\ \underset{\downarrow}{\underline{n{!}{=}N}} \text{ and } \cdots\ \underset{\downarrow}{n{\le}N}\ \cdots\ \}\ \text{(loop body)}$$

The effect of entering the loop is to add the loop-entry condition `n!=N` to the loop's precondition (at ↓).

– Loop body

$$\{\text{PRE } \cdots\ n{<}N\ \cdots\}\ s,n{=}\ s{+}A[n],n{+}1\ \{\text{POST } \cdots\}$$

Here the precondition is not *textually* the same as the postcondition before, but it is equivalent: if `n<=N` and `n!=N` then `n<N` (and vice versa).

We must check that

$$\begin{array}{ll} & 0{\le}n{<}N \text{ and } s{=}{=}\textstyle\sum A[0{:}n] \\ \Rrightarrow & 0{\le}(n{+}1) \text{ and } (n{+}1){\le}N \text{ and } s{+}A[n]{=}{=}\textstyle\sum A[0{:}n{+}1] \quad . \end{array}$$

– Loop exit

$$\{\text{PRE } \cdots\}\ \text{while } \underline{n{!}{=}N}\ \text{(exit loop)}\ \{\text{POST } \underline{n{=}{=}N} \text{ and } \cdots\}$$

Here the effect of *not* entering the loop is to add the negation of the loop-entry condition to the loop's postcondition, that is `n==N` to get the postcondition for the whole loop.

– End of program

$$\{\text{POST } s{=}{=}\textstyle\sum A[0{:}N]\}$$

This *second* postcondition is the postcondition for the whole program, and it is strictly weaker than the postcondition passed to it from the `while`-loop: we are not interested in `n` anymore.

It is always allowed to have several assertions in a row, as long as each one is implied by (⇛) the one before.

Each of those five parts corresponds to taking one step in the execution of the program, as shown in the flowchart of Fig. 1.1.

Even the "weakening of the final postcondition" (the final item above) corresponds to a step in the flowchart, though it is not a step in the program (because there is no program statement there). But it makes sense, because if the first assertion implies the second, and the first is true, then you will find that the second is true as well.

1.8 Avoiding infinite loops

Our original summing program (1.3) cannot loop forever, because it's a "`for`-loop", that is a loop where the number of iterations, and the values of the index variable, are fixed in advance. (That is an advantage of `for`-loops.) As long as the code in the loop body does not assign to the index (and it should not), the loop will eventually end. Here it is again, with assertions added:

```
# -- Sum the elements of sequence A[0:N].
s= 0
for n in range(N): # s==∑A[0:n]          ← the invariant          (1.18)
  s= s+A[n]
# s==∑A[0:N]    .
```

We have split the `for`-loop into two lines so that there is a convenient place to write the invariant, the assertion `s==∑A[0:n]` . It's called the *invariant* of the loop, because (as we saw above, with the `while`-loop), it's true every time the `for` (or `while`) is reached — and that is precisely what allowed us to "roll up" the separate program fragments (for `n` being 0,1,2 etc.) into a single piece of code.

For "`while`-loops" on the other hand it is possible that the loop will never end: and usually, at least in simple programs (like Prog. 1.1), that's a mistake. But luckily –again, for simple programs– there is an easy way to check that an infinite loop cannot happen. We look for an integer expression that can never become negative, and we show that each execution of the loop body must execute assignment statements that strictly decrease that expression. In the `while`-loop version of the summing program, which was

```
# -- Sum the elements of sequence A[0:N].
# 0<=N
s,n= 0,0
while n!=N: # 0<=n<=N and s==∑A[0:n]     ← the invariant          (1.19)
  s,n= s+A[n],n+1
# s==∑A[0:N]    ,
```

the *variant* is the expression is `N-n` . From the invariant's `n<=N` we see that `N-n` can never be negative; and from the loop body's `n= n+1` we see that it must decrease every time the loop body is executed (since `N` does not change). With some practice, one simply says that `n` on its own is an "increasing" variant (which therefore must be bounded *above*).

And now if we go all the way back to (1.1) from Sec. 1, which was

```
# -- Calculate the year number for today,
# -- where y is the number of days since 1/1/1980.
y= 1980
while d>365:
  if leapYear(y):
    if d>366:
      d= d-366
      y= y+1
  else:
    d= d-365
    y= y+1
# The current year is y.
```

[7]

we can see that it *has no variant*: it cannot, because when `y` is a leap year and `d==366`, the first `if`-test is `True` and the second one is `False`; and in that case the loop body makes no assignments at all — it can't decrease anything, variant or otherwise.

That concludes our introduction to checking programs, making sure that they work: the *ideas* are extremely simple, though often overlooked (and –surprisingly– not taught). They become complicated only when they are applied to complicated programs. But –as we will see– keeping invariants and variants in mind when writing complicated programs can actually help to *find* the program, and indeed often can result in programs that are simpler than they might have been otherwise.

1.9 Summary

Here is a summary of what we have covered so far — and you can do a lot with just this alone.

(a) **What are requirements?**
Requirements state informally, "in English" what the program is for, what it is supposed to do. They should not be too technical.

(b) **What are comments good for?**
Comments help people understand the program: they can state what the program is supposed to do; they can explain why certain code is the way it is; and they can give conditions (assertions, pre- and postconditions, invariants) that help to check that the program works.

(c) **What are assertions? When are they supposed to be true?**
Assertions are Boolean expressions that should be true whenever the program is executing at that point. Preconditions and postconditions are special cases of assertions.

(d) **What are preconditions?**
Preconditions are assertions written before a program fragment. They state with reasonable rigour the conditions that the fragment relies on in its initial state.

(e) **What are postconditions?** Postconditions are assertions written after a program fragment. They state with similar rigour the conditions that a the fragment must establish when it reaches its final state, provided the precondition(s) were met in its initial state.

(f) **What are specifications?**
Specifications are precondition/postcondition pairs.

(g) **What does it mean for a program to satisfy its specification?**
A program satisfies a specification if it treats the pre- and postcondition as described at (d) and (e) just above, and checking the resulting program succeeds. The rules for checking programs are summarised in App. B.

(h) **When can you say a program "works" ?**
A program works just when it satisfies its specification.

(i) **How do you show that an assignment statement satisfies a a specification?**
Treat the assignment statement as a textual substitution, replacing in the specification's postcondition all occurrences of the variables on its left-hand side by the corresponding expressions on the right-hand side. Then check that the specification's precondition implies the result of that substitution.

(j) **How do we decompose big complicated programs into smaller, simple ones?** We split the program between statements, inserting an assertion in between: it becomes the postcondition of the first statement and the precondition of the second. Each precondition/statement/postcondition is treated as a small program on its own, and checked with respect to its own precondition and postcondition. As with the "tree" analogy mentioned earlier, those smaller programs can themselves be split again.

(k) **What's the connection between specifications and requirements?**
Requirements are the interface between the programmer and the client, and can be informal — but they should be clear. Specifications state in more rigorous terms what a program should do, and they should imply the requirements; but that can only be checked informally.

The rigour of specification is what makes them useful for checking reliably that the program works. Check carefully that the program satisfies its specification, by splitting the program up into smaller pieces connected by assertions; but check also that the specification meets the requirements. Here is a final example:

requirements — Find the (integer) quotient `//` and remainder `%` of non-negative `num` and positive `den`.

specification — `{PRE den>0} ··· {POST q==num//den and r==num%den}`

See Ex. 1.20.

program (that works)

```
# -- Find the quotient and remainder of num and den.
# PRE num>=0 and den>0
q,r= 0,num
while r>=den: # INV q*den+r==num and 0<=r
  q,r= q+1,r-den
{ q*den+r==num and 0<=r<den }
# POST q==num//den and r==num%den
```
(1.20)

We wrote "INV" to remind us that the assertion there is the loop invariant.

1.10 *Exercises*

Exercise 1.1 This program sets `m` to the maximum value among `a`,`b` and `c`:

```
if a>=b:
  if a>=c: m= a
  else:    m= c
else:
  if b<=c: m= c
  else:    m= b    ,
```
(1.21)

and a conventional approach to checking it would be to examine all four paths it might follow when run. Write out the four paths, and the checks you would have to do.

If you extended this program in the same style, to include a fourth variable `d`, how many paths would you have to check then? See Ex. 5.4.

Exercise 1.2 (p. 11) In Prog. 1.17 there was the program fragment See Ex. 1.3. See Ex. 1.4. See Ex. 1.6.

```
{PRE 0<=n<N and s==∑A[0:n]}
s,n= s+A[n],n+1
{POST 0<=n<=N and s==∑A[0:n]}    .
```

If you perform the substitution of `s+A[n]` for `s` and `n+1` for `n`, as the assignment statement suggests, what exact condition do you get? Just do the substitution: do not simplify it (yet).

Exercise 1.3 (p. 17) Use arithmetic and facts about sequences (and $\sum$) to simplify your answer to Ex. 1.2 as much as you can. See Ex. 1.2.

Exercise 1.4 (p. 17) Split the program of Ex. 1.2 into two assignments, one after the other: See Ex. 1.2.

```
{PRE 0<=n<N and s==∑A[0:n]}
s= s+A[n]
{ What assertion goes here? }
n= n+1
{POST 0<=n<=N and s==∑A[0:n]}    .
```

Use the substitution implied by the second assignment `n= n+1` to *tell you* what the assertion that should go between the statements could be: that way, you do not have to guess. Then simplify it. And *only then* perform the substitution, on that new assertion, that is implied by the first statement `s= s+A[n]` ; and, finally, check that it's implied by the precondition. See Ex. 1.6.

Exercise 1.5 (p. 11) In Prog. 1.17, where have we checked that sequence `A` is always being accessed within bounds?

Exercise 1.6 (p. 17) Split the program of Ex. 1.2 into two assignments, one after the other as in Ex. 1.4, but this time the other way around:

```
{PRE 0<=n<N and s==∑A[0:n]}
n= n+1
{ What assertion goes here? }
s= s+A[n]
{POST 0<=n<=N and s==∑A[0:n]}    .
```

Use substitution to tell you the assertion that should go between the statements. Then perform the substitution implied by the first statement. Is it implied by the precondition? Does the program check? See Ex. 1.4. See Ex. 1.7.

See Ex. 1.6.

Exercise 1.7 (p. 17) If the program in Ex. 1.6 does not check, i.e. the intermediate assertion you found is *not* implied by the precondition, can you by some other means find an intermediate assertion that does show that the program work? If not, what does that tell you about the assertion that substitution found for you?

Exercise 1.8 (p. 11) The following program is like Prog. 1.17 except that it goes in the opposite direction, from high `n` to low. The presentation is simplified by removing some of the assertions, and most of the PRE's and POST's; and some question marks "`?`" have been introduced for you to fill in:

```
{PRE 0<=N}

s,n= 0,N
{ 0<=n<=N and s==∑A[?:?] }                  # Start here.
while n!=0:
  { 0<n<=N and s==∑A[?:?] }
  n= n-1                                                        (1.22)
  { ??? }       # And calculate this from the one below.
  s= s+A[n]
  { 0<=n<=N and s==∑A[?:?] }                 # Then here.
{ n==0 and s==∑A[?:?] }

{POST s==∑A[0:N]}    .
```

Fill in the missing pieces indicated by `?`'s.

Remember that it is easier to calculate assertions *backwards*, working from post-conditions to preconditions. For example, the assertion marked `Start here.` is the loop "invariant", and you should figure it out first, and check that it is established by the initial assignment (given the program's overall precondition `0<=N`). Then the assertion marked `Then here.` , at the end, should be exactly the same. Only after all that, use substitution to calculate the intermediate assertion in the middle.

Exercise 1.9 (p. 11) The comments in this version of Prog. 1.17 should be enough for checking that it works:

```
# -- Calculate the sum of sequence A[0:N].
# 0<=N
s,n= 0,0
while n!=N: # s==∑A[0:n]                                        (1.23)
  s,n= s+A[n],n+1
# s==∑A[0:N]    .
```

The reason is that the requirements comment can be checked against the other two "assertion" comments, to see that the program is doing what the customer wants. Then the assertion comments can be turned into actual assertions, with the obvious "housekeeping" inequalities on `n` added. Then the intermediate assertions can be generated by substitution, and the implications checked by arithmetic and sequence-based reasoning. The PRE and POST labels are not necessary — we use them just to explain how the reasoning works. Any assertion is simply considered to be a *pre*condition of the statement after it, and a *post*condition of the one before it.

In the end, the only significant fact turns out to be that

```
0<=n<N    ⇒    ∑A[0:(n+1)] == ∑A[0:n] + A[n]    ,
```

which is actually a law of sequences: that for any two sequences `A` and `B` we have See Ex. 1.11.

$$\sum\texttt{(A+B)} \quad == \quad \sum\texttt{A} + \sum\texttt{B} \quad , \tag{1.24}$$

where (+) sticks two sequences together, "concatenates" them.

Starting from Prog. 1.23, go through the process of generating all the assertions needed to check that it works.

You should do an exercise like this at least once in your life. Later, you will develop your own style: you will learn which assertions you should include and which ones you can safely leave out, which implications you should check carefully and which ones you can just "eyeball".

And indeed there are are program-checking tools that can generate many assertions for you automatically, at least the simple ones. But *before* you use those tools, you must know how to do it yourself! Otherwise you will never understand what is actually going on.

Exercise 1.10 The two programs

```
# PRE N>=0
s,n= 0,0
while n<=N: # INV s==∑_{i=0}^{n} i²
  s,n= s+n*n,n+1
# POST s==∑_{i=0}^{N} i²
```
(1.25)

and

```
# PRE N>=0
s= 0
for n in range(N+1): # INV s==∑_{i=0}^{n} i²
  s= s+n*n
# POST s==∑_{i=0}^{N} i²
```
(1.26)

have the same pre- and postconditions, and the same invariant. The loop body is executed the same number of times (`N+1`) and for the same values of `n` (from `0` to `N` inclusive).

And they assign the same final value to `s`.

But what is the final value of `n` in Prog. 1.25? In Prog. 1.26? *Hint:* See App. G.2.

Exercise 1.11 (p. 19) In Ex. 1.9 we mentioned a "law" (1.24) that related addition and sequence concatenation. How would you use that law to argue that the sum of the empty sequence must be zero, i.e. that $\sum$`[] == 0` ?

What is the product $\prod$`[]` of the empty sequence? See Ex. 1.12. See Ex. 1.13.

Exercise 1.12 (p. 19) Any square matrix (of numbers) has a *determinant*. For the matrices

$$|a| \quad \text{and} \quad \begin{vmatrix} a & b \\ c & d \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

the determinants are a and $ad-bc$ and

$$a\begin{vmatrix} e & f \\ h & i \end{vmatrix} + b\begin{vmatrix} f & d \\ i & g \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

respectively.

What is the determinant of the empty matrix?

Exercise 1.13 (p. 19) By analogy with `+` and $\sum$, and similarly `*` and $\prod$, define `max` to be the maximum of two numbers, and `MAX` to be the maximum of a sequence. Thus `1 max 2 == 2` and `MAX[1,3,2] == 3`.

What is the maximum `MAX[]` of the empty sequence?

Exercise 1.14 (p. 7) Program (1.10) finds the largest element of a sequence, and it was introduced as an example to motivate *pre*conditions, because it worked only on non-empty sequences: there cannot be a largest element of a sequence if the sequence has no elements at all. If we re-phrase its requirements slightly, however, we can make the program more general — provided we assume for now that the programming language includes the value that was the answer to Ex. 1.13. The program becomes

```
# -- Find the maximum of all elements in A[0:N].
# PRE 0<=N
m= ?
for n in range(N): # m == MAX A[0:n]
  { What goes here? }
  m= m max A[n]
  { What goes here? }
# m == MAX A[0:N]
```

Note that we are alternating freely between writing assertions as `{ ··· }` between brackets and assertions as comments `# ···` . They are the same thing: but we are following a convention that the `{ ··· }` for is used for checking that a program works, and the `# ···` form is used for conditions that we would like to remain in the program afterwards, as comments, for documentation. Thus, in the above, the comment that is the loop invariant, `# m == MAX A[0:n]` , should be part of the program's documentation — but also an assertion of course that we actually use when reasoning about whether the program works. On the other had, the `{ What goes here? }` assertions we can probably remove once we are happy they have done their job.

What are those two `{ What goes here? }` assertions? Use them to show that the program works.

Exercise 1.15 (p. 5) A simple example of a *continued fraction* is $1 + \frac{1}{2}$, which has the value $3/2$. If we go two more steps we have

$$1 + \frac{1}{2 + \frac{1}{3}} \qquad \text{and} \qquad 1 + \frac{1}{2 + \frac{1}{3 + \frac{1}{4}}} \quad ,$$

which have values $10/7$ and $43/30$.

Suppose we represent a continued fraction as a (finite) non-empty sequence of integers `A[0:N]` where `A[0]` is the leading term, and elements `A[1:N]` give the subsequent denominators. Thus `[a]` would be a and `[a,b]` would be $a + \frac{1}{b}$ and `[a,b,c]` would be $a + \frac{1}{b + \frac{1}{c}}$ and so on.

Write a program in the style of this chapter that, given a sequence of positive integers, calculates the value of the continued fraction it represents.

"In the style of this chapter" means

(a) Include an initial *requirements* comment.

(b) Write the *precondition* of the whole program as a comment at its beginning.

(c) Write the *postcondition* of the whole program as a comment at its end.

(d) Give an *invariant* for your loop; write it as a `# INV` -comment immediately after the colon on the `for`-loop or `while`-loop header.

Note that the *way* you write these assertion/comments does not have to be super-rigorous: it's the ideas they express that are important.

After all, the summation $\sum$ was introduced into as assertion at (1.4) in the running example for this chapter, wholly without rigour — we never defined it, except informally in words. But we did use a property of it, which turned to be all we needed: it was the "law" (1.24) (stated without proof) in Ex. 1.9.

The whole idea of checking programs this way is to reduce the job systematically to a number of much smaller and, crucially, *separate and independent* checks for which "eyeballing" suffices for each one on its own. A good example of that technique was given at (1.17).

Exercise 1.16 (p. 3) Use the invariant "Today is `d` days from 1/1/`y` ." to write a version of Prog. 1.1 that actually works; assume that the function `DiY(y)` gives the number of days in Year `y`. Be sure to state somewhere whether 1/1/80 is Day 0 or Day 1. (Should that be in the requirements, or in the precondition, or in the postondition?) See Ex. 1.19.

Do not use `break` to escape the loop; instead call `DiY()` twice if necessary. See Ex. 1.17.

What is the variant, and how do you know it decreases? Note that (integer) variants do not have to be non-negative: it's enough to show that they are bounded below. "Non-negative" is just the special case where the below-bound is zero. See Ex. 3.1.

Exercise 1.17 (p. 21) In your answer to Ex. 1.16 you might have called function `DiY(y)` in two places. Figure out how to use assertions to check loops that use `break`, and then write a version of Prog. 1.1 that uses `break`, and so needs only one call of `DiY()`. Introduce an extra variable `n` if you need it. See Ex. 1.18.

What about `continue`? See Ex. G.2.

Exercise 1.18 (p. 21) Make a flowchart for your answer to Ex. 1.17 in the style of Fig. 1.1. Check that all assertions are true as program control passes through them.

See Ex. 1.16.

Exercise 1.19 Suppose that the date today is 1 January 2021, so that "1 day ago" would be 31 December 2020, and "366 days ago" would be 1 January 2020 (because 2020 was a leap year). Assume (as above) that you have a function `DiY` such that that `DiY(y)` is the number of days in Year `y`.

Use the invariant

`G+DiY(y)>g` `and` "Day `1` in Year `y` is `g` days ago."

to write a program that uses the natural number `G` (for "`aGo`") to set variables `d` (for "day") and `y` (for "year") so that "`G` days ago" (from 1 January 2021) is the `d`-th day of Year `y`, where 1 January in any year is Day 1 (in that year). Thus for example with input `G==365` initially, the program should establish `y==2020` and `d==2` finally. The following "skeleton" gives a hint for the program structure:

```
# PRE G>=0 and "Today is 1 January 2021."
{ FFF and "Today is 1 January 2021." }
g= EEE
{ AAA and "Day 1 in Year 2021 is g days ago." }
y= CCC

{ AAA and BBB }
while kkk:
  { AAA and BBB and KKK }
  { KKK and "Day 1 in Year y-1 is g+DiY(y-1) days ago." }
  y= HHH
  { KKK and "Day 1 in Year y is g+DiY(y) days ago." }
  g= GGG
  { AAA and BBB }
{ LLL and AAA and BBB }

{ 1<=g+1-G<=DiY(y) and "Day g+1-G of Year y is G days ago." }
d= JJJ
# POST III and "Day d of Year y is G days ago."
```
(1.27)

Write the program by using the invariant, and various other assertion-manipulation techniques, to supply the missing pieces *XXX*.

See Ex. 1.21.

What is the variant?

Exercise 1.20 (p. 16) What is the variant that makes sure that the loop in Prog. 1.20 of Sec. 1.9(k) does not iterate forever?

Exercise 1.21 Look carefully at the precondition of Prog. 1.27 in Ex. 1.19, and use the checking rules to see whether perhaps it could be weaker. In particular, are there situations in which `G<0` but Prog. 1.27 is still guaranteed to establish its postcondition?

See Ex. 1.19.
See Ex. 1.16.

Now, more generally than in Ex. 1.19, suppose that today is Day `D` in Year `Y` (so that in Ex. 1.19 we would have `D==1 and Y==2021`). Otherwise, the purpose of the program is the same, to find `d` and `y` so that "`G` days ago" (from today) is the `d`-th day of Year `y`. How would you approach writing that program? And what is its precondition?

Chapter 2

Using invariants to design loops

2.1 Introduction

In this chapter, we will go through the program-design process in three *different* ways, but for the *same* problem. The first two will use a conventional approach (although we do mention invariants, after the fact, to help check them); but the third uses an invariant *before* the fact, to help design the program in the first place. [1]

2.2 The longest *Good* subsegment

Assume we have a Boolean function *Bad*() whose argument is a sequence of integers of length exactly 3, for some (not yet made precise) property of "badness". Here are some examples of what *Bad*($[a, b, c]$) might be:

- All equal: $a = b = c$.
- In a run, up or down: either $a+1 = b$ and $b+1 = c$, or $a-1 = b$ and $b-1 = c$.
- Able to make a triangle: thus $a+b+c \geq 2(a \max b \max c)$.
- The negation of any of the above.

Suppose further we are working (as usual) with a sequence `A[0:N]` and that we have a Boolean function `Bad(n)` that calculates *Bad*(`A[n]`, `A[n+1]`, `A[n+2]`). (Note that *Bad* itself has just the one argument, but it examines `A` in three places.) It is a programming error –subscript out of range– if `n<0` or `n+3>N`.

Now a subsegment of `A[0:N]` is any consecutive run `A[i:j]` of elements with `0<=i<=j<=N` . We say that such a subsegment of `A` is *Good* just when it has no bad subsegments inside it: a *Good* subsegment of `A` can potentially be any length up to the length `N` of `A` itself. And any subsegment of length 2 or less is *Good*, since it is too short to have a *Bad* subsegment inside it. (Remember that *Bad* subsegments have length exactly 3 and, since they can overlap, it's clear that `A` can contain anywhere from 0 up to `N-2` of them.)

[1] The problem itself is from a first-year university course "Introduction to Programming in *C*".

The requirements for this program might be written

```
# -- Determine the length of the longest Good subsegment
# -- of sequence A[0:N], where a Good subsegment has no
# -- subsegment of three (consecutive) elements inside it
# -- that are Bad.
# -- Assume that Bad(n) determines whether A[n:n+3] is Bad,
# -- for 0<=n<=N-3.
```

Its specification might be

```
# PRE Bad(n) is well defined for 0<=n and n+3<=N.
...
# POST good == largest value such that
# .... there is a low with 0<=low<=N and
# .... there's no n with low<=n and n+3<=low+good
# .... and Bad(n).
```

Notice that the requirements state that `Bad` must actually calculate *Bad*, but the specification does not: it makes no demands on the function `Bad` except that it be well defined for `0<=n and n+3<=N`. This is deliberate: given the many possible definitions of *Bad*, it might be a distraction if the programming task were to depend on what `Bad()` actually calculates.

We now design the first of our three programs for this specification.

2.3 First program: locate *Bad*'s, then measure *Good*'s

A straightforward approach to solving this problem might be along these lines.

First go through all of `A` and store its "bad positions" `b` in an extra array `B`. Then find the largest difference between any two adjacent elements of `B`. Then –after some careful thought– set `good` based on that largest difference.

Notice the operational phrasing "go through..." and the slightly fuzzy "based on" (One more than that? One less? Exactly that?) A typical approach might be to code up the program at this point, using the guess above, and see whether the answer looked right. It's only an assignment statement, after all, and if it contains a one-off error — well, it can be "tweaked", based on testing runs, without affecting the structure of the rest of the program. (Really?)

With the above intuitions, the program is easy to code: it's shown in Fig. 2.1. There are in that program however a number of careful calculations necessary to figure out the length of *Good* subsegments as they are discovered, plus the special cases of "no `Bad` subsegments" and "exactly one `Bad` subsegment" (when the first and the last `Bad`'s are the same), and finally the *Good* subsegments that run from the start of `A`, or right up to the end of `A`.

It's "easy to code", this program; but perhaps it's not so easy to check.

```
# Sequence B must have length at least N-2.

bads= 0 # Number of Bad subsegments found.
for b in range(N-2): # Find starting points of Bad subsegments.
  if Bad(b): B[bads],bads= b,bads+1
  # Now know where the Bad's are.

if bads==0: good= N # Special case.
else:
  good= B[0]+2 # First Bad subsegment gives first Good subsegment.
  for n in range(1,bads): # Subsequent Bad subsegments...
    good= good max (B[n]-B[n-1]+1) # ...give subsequent Good's.
  good= good max (N-B[bads-1]-1) # Very last Bad subsegment.
```

(2.1)

The first part of the program records the starting indices of all `Bad` triples in `A`. Its invariant is that `B[:bads]` records all such indices found so far, i.e. in `A[:n]`.

For the second part of the program there are three cases. If there are no `Bad`'s at all, then the whole of `A` is *Good*, and has length `N`.

If not, then the first *Good* subsegment begins at `A[0]` and ends at `A[B[0]+1]` (inclusive), and so has length `B[0]+2`. And the last *Good* subsegment (which –remember– might be the same as the first one!) begins at `A[B[bads-1]+1]` and ends at `A[N-1]` (again inclusive), and so has length `N-B[bads-1]+1)`.

In between, the invariant is that `good` is correct for all `Bad`'s recorded in `B[:b]` — that is, again "so far".

Figure 2.1 First program — 9 lines of code, and extra sequence `B` (Sec. 2.3)

```
a0,a1,good= -1,-1,0 # Pretend there is a Bad subsegment at -1.
while True:
  a1= a1+1 # Look for the next Bad subsegment.
  if a1+3>N: # There isn't one.
    good= good max (N-a0-1) # Take last a0 into account.
    break
  if Bad(a1): # Found one: update good from a0,a1 and carry on.
    good,a0= good max (a1-a0+1), a1
```
(2.2)

This second program improves the first in two major ways: it doesn't need an extra sequence `B`, and there are fewer special cases. A minor improvement is that it has only two `max`'es instead of three.

Think of `a0` and `a1` as being the tail and head resp. of a caterpillar. The tail always rests on a *Bad* subsegment; and on each iteration the head advances to the next *Bad* subsegment further along. Then `good` is updated by `max`'ing it with the current length of the caterpillar; and finally the caterpillar brings its tail `a0` up to its head `a1`. Then the process repeats.

At the beginning, the caterpillar is bunched up at -1, in effect pretending there's a *Bad* subsegment there. At the end, the caterpillar pretends there's a *Bad* subsegment at `N`.

Figure 2.2 Second program — 8 lines of code, no extra sequence (Sec. 2.4)

2.4 Second program: using a caterpillar

The second program can be derived from the first. We realise that having the extra array `B` is useful for separating the problem into sub-problems, but logically speaking it is a waste of space: a more space-efficient program uses a "conceptual" `B` whose currently interesting element is maintained as you go along. After all, that's the only part of `B` you need.

It's a nice idea, and it improves Prog. 2.1 so that it becomes Prog. 2.2 in Fig. 2.1 — and so it's probably worth the effort. And the result, given in Fig. 2.2, might even be considered a clearer coding than the first program: the caterpillar analogy really helps to see what's going on. It's slightly marred, though, by the mysterious "`-1`" initialisations.

Its invariant is that as `a0` and `a1` move through the indices of `A`, the subsegment `A[a0:a1]` is a non-extendable *Good* subsegment, and the value of `good` is always up-to-date with respect to `A[:a1]`.

2.5 Third program: let an invariant *design* the loop for you

We saw in Chp. 1 that assertions –a special kind of comment– can help to check that a program works. (That's why we gave invariants to help checking the two earlier version of this program.) Special cases were *preconditions*, which state what a program(mer) can assume about the initial state, and *postconditions*, state what the final state must satisfy and *invariants*, which state what should be true every time a loop condition is evaluated (to decide whether to (re-)enter the loop or not).

The first two are external, because they are directly related to the requirements: from the programmer's point of view they are "given". The invariants, on the other hand (one per loop) help to see *why* the loop is doing the right thing.

But invariants can do more — they can also be used to *design* the loop; and the loop works because the invariant was used to *make* it, not (just) to check it. We are about to see how the example of this chapter shows this nicely.

Let's write for brevity `lgs(n)` for the length of the largest *Good* subsegment in the prefix `A[:n]` of `A`, so that the program's specification becomes simply

```
# PRE Bad(n) is well defined for 0<=n and n+3<=N.
...
# POST good == lgs(N)    .
```

For many simple programs (like this one) we can find a candidate invariant immediately by splitting the postcondition: it becomes `n==N and good == lgs(n)`, referring now to a new variable `n` that will be the index for the loop we are about to create. (It is the technique "iterate up" from Sec. 3.3.1 to come.) The `n==N` is the negation of the loop condition (if we use a `while`-loop) or the actual termination condition (if we use a `for`-loop). Just that small step allows us to begin writing the program — but we write it *from the outside in* rather than from front to rear, or in one part and then another. Our first step is

```
# PRE Bad(n) is well defined for 0<=n and n+3<=N.
n,good= 0,0
{ good==lgs(n) }
while n!=N: # INV good==lgs(n) and 0<=n<=N ←invariant
  {PRE good==lgs(n) and n<N}
  ...                                          } ← This specifies the loop body.
  {POST good==lgs(n+1) and n<N}
  n= n+1            ↙       ↙
  {POST good==lgs(n) and n<=N}
{ n==N and good == lgs(n) }
# POST good == lgs(N)    .
```

In some places we leave out parts of `0<=n<=N` in the invariant to reduce clutter; but we put them in where they're important.

The programming is now reduced to finding the loop body, the "inside": the rest of the program, since the "outside", is already there. And the specification of the loop body (at "}←" above) tells us exactly what it must do. If `good==lgs(n)` at this point, and we are considering `A[n]`, which we know is well defined because of `n<N`, how do we "update" `good` so that it now takes `A[n]` into account as well?

Updating `good` means increasing it if a longer *Good* subsegment is found than the ones already considered. All the subsegments of `A[:n]` were considered (says the

invariant), and so if `good` must change it must be because of a *Good* subsegment that ends at the newly considered element `A[n]`. So we add a new variable `low` to keep track of that, and an second invariant

`Inv2: A[low:n] is the longest` *Good* `suffix of A[:n]` .

(See Sec. 3.5 to come for a more general discussion of this.) We'll call it `Inv2(n)`, and our earlier invariant we'll now rename to `Inv1(n)`. Both of them apply. Our program becomes

```
# PRE Bad(n) is well defined for 0<=n and n+3<=N.
good,low= 0,0
{ Inv1(0) and Inv2(0) }
n= 0
while n!=N: # Inv1(n) and Inv2(n)          ← invariants
  {PRE Inv1(n) and Inv2(n)}
  ...                                      } ← This specifies how low must be updated.
  {POST Inv1(n) and Inv2(n+1)}
                ↕
  {PRE  Inv1(n) and Inv2(n+1)}
  good= good max (n+1-low)                 } ← This is correct by inspection.
  {POST Inv1(n+1) and Inv2(n+1)}

  {PRE  Inv1(n+1) and Inv2(n+1)}
  n= n+1
  {POST Inv1(n) and Inv2(n)}               ← invariants re-established
{ n==N and good == lgs(n) }                ← loop exit: not (n!=N) and Inv1(n)
# POST good == lgs(N)    .
```

These are the same:

Notice ↕ how the postcondition of the `low`-update becomes the precondition of the `good`-update, and tells us what that `good`-update must be *even before we have figured out how to update* `low` *itself*...

...which is what we now do: we update `low` as its specification directs. If the last three elements of `A[:n+1]`, that is `A[n-2]`, `A[n-1]` and `A[n]` are `Bad`, then `low` must become `n-1`. (Remember that `A[n-1:n+1]`, of length only 2, cannot be `Bad`.) On the other hand, if they are not `Bad` then `low` need not be changed. Bearing in mind that the subsegment must be of length 3 –and that there must be room for it– the final jigsaw piece is

`if 2<=n and Bad(n-2): low= n-1` .

If we insert that in the above, and then remove all the "scaffolding" –that is, the assertions we used, along the way, to help us *design* the program– the result is Fig. 2.3.

See Ex. 2.2.
See Ex. 2.1.

```
# PRE Bad(n) is well defined for 0<=n and n+3<=N.

# Inv1(n):  good is the length of a longest Good subsegment of A[:n].
# Inv2(n):  A[low:n] is the longest Good suffix of A[:n].

good,low,n= 0,0,0
while n!=N:                        # Inv1(n) and Inv2(n) and 0<=n<=N
  if n>=2 and Bad(n-2): low= n-1
  good,n= good max (n+1-low), n+1

# POST Inv1(N).
```

(2.3)

Figure 2.3 Third program — only 4 lines, no array, one max. (Sec. 2.5)

2.6 *Exercises*

Exercise 2.1 (p. 28) Explain why Prog. 2.3 of Fig. 2.3 cannot index `A` out-of-bounds with its call of `Bad`. Then write the program as a `for`-loop.

Since a subsegment must have length at least 3 to be `Bad`, is it true that the program always sets `good` to at least 2?

Exercise 2.2 (p. 28) Modify the postcondition for Prog. 2.3 from Sec. 2.5 so that the subsegment `A[start:end]`, that is `A[start]`, `A[start+1]`,...,`A[end-1]`, is an actual *Good* subsequence of maximum length. Modify the invariant to take account of that. And write the program that results.

(There should not be too much to change.)

Chapter 3

Finding invariants

3.1 Introduction

Invariants were introduced in Sec. 1.7 as assertions used specifically for designing and checking loops; and indeed we devoted all of the previous chapter (Chp. 2) to a single extended example of how that is done. Using "conventional" design techniques gave programs that were twice as long as the one we found, finally, by choosing an invariant *first*, before beginning to code.

And so invariants turn out to be one of the major skills that lead to simple, efficient and evidently correct programs, easy to check — even when the problems those programs appear at first to be difficult, or at least "fiddly" and "tricky" (as in the last chapter).

This chapter is more general, giving you some of the techniques used to find invariants *before* you write the program, then using them to help design the program so that it is "automatically" easy to check — precisely because it was designed with checking in mind.

We begin with the most common invariant-finding technique: splitting a conjunct.

3.2 Split a conjunct

This first invariant-finding technique is the simplest, and indeeed the most common. We introduce it by returning to our very first topic — date finding (Sec. 1.2). But this time we will concentrate on months rather than years.

Suppose we have a function `DiM(m)` that gives for any month `m` from 1 to 12 inclusive the number of days in that month: thus `DiM(1)==31` and `DiM(6)==30` etc. We ignore leap years for now.

Here is a program that converts a day-in-year `d`, again starting at 1, to a month-

number m and a day-in-month d:

```
# -- Given the day-number in a year,
# -- find the month-number
# -- and the day-number in that month.
# -- Ignore leap years.

# PRE 1<=d<=365
# PRE DiM() contains correct values.
...
# POST This month is m and d is the day number in it.
```

We begin by rewriting the postcondition as

```
Today is d days from the 1st of m, and 1<=d and d<=DiM(m).
```

and we see that the postcondition is in three pieces, joined by and's — they're called "conjuncts". We will take the first two of them for the invariant, and establish the third by making its opposite (i.e. its negation not) the condition of the loop: that is, we "split" the three conjuncts between the second and the third. That gives

```
# PRE 1<=d and Today is d days from the 1st of January.
m= 1
while d>DiM(m): # INV 1<=d and Today is d days from the 1st of m.
...
# d<=DiM(m) and 1<=d and Today is d days from the 1st of m.
# POST Today is day d in month m this year.
```

Notice that when m==1, as it is initially, the sum DiM(1)+···+DiM(m-1) is zero, because it is the sum of no terms. It is strictly less than d, because we have 1<=d from the precondition.

See Ex. 1.11.

Now that we've found an invariant, our job is to code a loop body that maintains it. That loop body is d,m= d-DiM(m),m+1, giving the program

```
m= 1
while d>DiM(m): # Inv: 1<=d and Today is d days from the 1st of m.
  d,m= d-DiM(m),m+1            # Maintain the invariant.

# d>DiM(m) must be False if the loop has finished.  And so...
# d<=DiM(m) and (still) 1<=d and Today is d days from the 1st of m.
# POST Today is month m and day d.
```

(3.1)

See Ex. 3.1.

And that's our program done.

Recalling Jack's difficulties in Sec. 1, however, we will check carefully that this program does not loop forever. The variant for the loop (Sec. 1.8) is d itself, which the invariant bounds below by 1. And the subtraction d= d-DiM(m) strictly decreases d, because from the overall precondition (that DiM() contains correct values), we know that DiM(m)>=1; and so that is sufficient for a strict decrease. So this loop cannot go on forever.

We must check also, of course, that the invariant really is respected — that is, the loop must give the right answer when it *does* terminate. We will do that carefully using the rule for checking assignments: so we are therefore checking the program fragment

```
# PRE d>DiY(m) and 1<=d and Today is d days from the 1st of m.
d,m= d-DiM(m),m+1
# POST 1<=d and Today is d days from the 1st of m.
```

where we note the extra conjunct `d>DiY(m)` in the precondition: it is the `while`-condition. If it were not true, we could not *be* in the loop.

As at (1.13) earlier, to check an assignment we make the substitution and then check an implication. In this case the implication is

```
    d>DiY(m) and 1<=d and Today is d days from the 1st of m.
⇛   1<=d-DiM(m) and Today is d-DiM(m) days from the 1st of m+1    .
```

Because `d>DiY(m)` ⇛ `1<=d-DiM(m)`, we can simplify that implication to

```
    Today is d days from the 1st of m.
⇛   Today is d-DiM(m) days from the 1st of m+1    ,
```

which (again) follows from the precondition that `DiM()` contains correct values.

3.3 Introduce a new variable

The second common invariant-finding technique is often used together with the first, and the result is a program with a loop that goes up or down in regular steps.

3.3.1 Iterating up, or down

Most of the examples we have done so far use an invariant that could be found by introducing a new variable. The sum'ing and max'ing programs in Chp. 1 were all of the form

```
# PRE A is a sequence of length N.
...
# POST r == Fun(A)    ,
```

in other words "Assign to result-variable `r` the value of some function *Fun* applied to the whole sequence `A`." Function *Fun* might be "the sum of" ($\sum$) or "the product of" ($\prod$) or "the maximum of" (MAX); and in each case the invariant-finding strategy is

(a) Replace (just) `A` in the postcondition by the more explicit `A[0:N]`.

(b) Introduce a new variable `n` and rewrite the postcondition as

```
r == Fun(A[:n]) and n==N    ,
```

which makes `not n==N`, that is `n!=N` become the loop condition.

(c) Split the conjunct (Sec. 3.2 just above), and use the invariant

```
0<=n<=N and r == Fun(A[:n])    .
```

The program becomes

```
# PRE A is a sequence of length N.
n,r= 0, Fun([])
while n!=N: # Inv: 0<=n<=N and r == Fun(A[:n])
  {PRE 0<=n<N and r == Fun(A[:n])}
  ··· # Examine A[n]; update r.
  {POST r == Fun(A[:n+1])}
  n= n+1 # Invariant restored.
  { r == Fun(A[:n]) }
{ n==N and r == Fun(A[:n]) }
{ r == Fun(A[0:N]) }
# POST r == Fun(A)
```
(3.2)

which "iterates up" from 0 towards N.

See Ex. 3.2.

For the analogous "iterating down" technique, see Ex. 3.2.

For a not-completely-trivial example of iterating up, let *Fun* be "the largest absolute difference between any two adjacent elements" — call it lad. From the above, once we have followed the routine steps just above, we are left with the loop body

```
{PRE 0<=n<N and r == lad(A[:n])}
···
{POST r == lad(A[:n+1])}
n= n+1
```

in which the assignment ··· is clearly `r= r max abs(A[n]-A[n-1])` . Or is it?

The value max'ed with r has to be –writing very carefully– the absolute difference between two adjacent elements where the second one is A[n]. But unless 0<=n-1, there isn't such a pair. So we must write `if n>=1: r= r max abs(A[n]-A[n-1])` .

And we mustn't forget the initialisation, which should include `r= lad([])` . Since our program produces only non-negative results, we can take zero –the least non-negative number– for lad([]), and the program becomes

```
# PRE 0<=N
n,r= 0,0
while n!=N: # Inv: 0<=n<=N and r == lad(A[:n])
  if n>=1: r= r max abs(A[n]-A[n-1])
  n= n+1
# POST r == lad(A)
```
(3.3)

See Ex. 3.2.

3.3.2 A note on efficiency

It's true that Prog. 3.3 contains the irritating feature that it tests `n>=1` again and again, where we can easily see that only one test is required. That can easily be avoided — just rewrite it as See Ex. 3.3.

```
# PRE 0<=N
r= 0
if 0!=N:
  n= 1                  ↓ Note!
  while n!=N: # Inv: 1<=n<=N and r == lad(A[:n])
    r= r max abs(A[n]-A[n-1])
    n= n+1
# POST r == lad(A)     ,
```
(3.4)

but –Note!– this should not be just a "quick fix". There's no point in carefully checking a program and then risking its correctness –all that work– with a casual, off-the-cuff adjustment: so we will do it in stages, and alter the invariant to take account of what we do. Quick fixes should be avoided... and we'll use this program as an example of how to proceed carefully.

The first stage is to "unroll" the loop in Prog. 3.3 just once, so that the program becomes

```
n,r= 0,0
if n!=N:                                        # n==0 here.
  if n>=1: r= r max abs(A[n]-A[n-1])             # and here.
  n= n+1
  while n!=N:
    if n>=1: r= r max abs(A[n]-A[n-1])
    n= n+1
```

and –crucially– it does not have to be checked again: if the loop was correct before unrolling, it's still correct afterwards.

The next step is to replace `n` with `0` in the places where the unrolling has made `n==0` true. (Just replace it: don't simplify yet. Small, careful steps are the key...) That gives

```
n,r= 0,0
if 0!=N:                                        # n==0 here.
  if 0>=1: r= r max abs(A[0]-A[0-1])             # and here.
  n= 0+1                                        # and here.
  while n!=N:
    if n>=1: r= r max abs(A[n]-A[n-1])
    n= n+1    ,
```

where the illegal sequence-indexing `A[0-1]` is not a problem: behind an `if 0>=1:` that is an `if False`, it is dead code. See Ex. 3.3.

The third step is to simplify what we now have, trivially *and therefore reliably*, to Prog. 3.4 above. We removed the redundant initialisation `n= 0` and the "dead" `if 0>=1:` and simplified the `0+1` to just `1`. A final step might be to turn the `while`-loop into into a `for`-loop. See Ex. 3.4.

Just as in loops "with or without early exits" (mentioned elsewhere), it is a a good strategy to write programs first in a way that makes them easy to check. Only after

that do you make small changes, if necessary, for efficiency. There's a good chance that the extra checking for those changes will be quite small, and the earlier checks "come along for the ride".

3.3.3 Iterating towards the middle: Binary Search

In this example we look at iterations that go both up *and* down: it is based on Ex. 3.5.

See Ex. 3.5.

Our program is given a sequence `A[0:N]` that is sorted in ascending order, and a value `x`, and it must find where `x` occurs in the sequence:

```
# -- Find the location of a given value in a sorted sequence.
# PRE Sequence A[0:N] is sorted into ascending order.
# POST A[:n]<x and x<=A[n:]     .
```

In general `A[l:h]<x` means that *all* elements in that subsegment of `A` are strictly less than x, and similarly for other comparisons: for example `A[l:h]==x` means that all values in `A[l:h]` are equal to `x` (and hence also to each other).

See Ex. 3.5.

In Ex. 3.5 you can work out the conclusions that can be drawn from the result, i.e. from the postcondition of the above program. Here we will concentrate on how to discover a good invariant for it. We introduce a new variable `m`, and replace one of the `n`'s by that — and then add a conjunct `m==n` so that the new postcondition implies the one we really want. The postcondition is now

```
A[:m]<x and x<=A[n:]  and m==n    .
```

Then using "split a conjunct" from Sec. 3.2 above, we can add some more code to our growing program. As usual, we try to work "from the outside in", and we get

```
# PRE Sequence A[:N] is sorted into ascending order.
m,n= 0,N
while m!=n: # Inv: 0<=m<=n<=N and A[:m]<x and x<=A[n:]
  ··· # Need to look at some A[p] in A[m:n].
# POST A[:n]<x and x<=A[n:]     .
```

The "Need to look at..." comment is a temporary note to ourselves, a hint to our own intuition: since the invariant says for example that `x` is *not* in `A[:m]` –it is strictly greater than all of them– we must have "looked at those elements already". Similar reasoning suggests that we have looked at `A[n:]` already, and therefore that the only only place left to look is `A[m:n]`, the "middle" segment

Being precise at this stage is quite helpful, as we are about to see. It is indeed true that the elements we need still to look at are "the ones in the middle" — but since we *can* say exactly, precisely, which ones those are, we should do it. The earlier that the accessible information is brought to light, the easier our later steps will be.

And indeed our first payoff is to know how to choose `p`. If it's to index into `A[m:n]`, it must satisfy `m<=p<n`: that's direct and obvious, without any "Is it $<$ or $\leq$?" -guesswork involved. All that is taken care of for us. (Remember that subsegment indexing is inclusive/exclusive.) Our precise identification of the subsequence of possible "`p` targets" is what told us that the inequality must be `<=···<`, that is less-than-or-equal-to on the left, and strict less-than on the right. No head-scratching is required.

Now the one thing that everybody knows about binary search is that somewhere in the middle there is a statement something like `p= (m+n)//2`, where `//` is integer

division (rounding down). And then `A[p]` is compared with `x`. But that "something like"... Should we maybe round up? Does it matter? The remainder of our program is now

See Ex. 3.8.

```
# PRE 0<=m<n<=N and A[:m]<x and x<=A[n:]
p= (m+n)//2
if   A[p]<x:  { A[:m]<x and A[p]<x  and x<=A[n:] } ···
else:         { A[:m]<x and x<=A[p] and x<=A[n:] } ···
# n-m has decreased
# POST 0<=m<n<=N and A[:m]<x and x<=A[n:]      ,
```

where with `# n-m` *has decreased* we are stating that the variant is `n-m` and we are using it to check that the loop does not continue forever. Notice that the precondition of this remaining portion has `m<n`, which comes from the invariant's `m<=n` and the loop condition `m!=n`. We can't check `{ n-m` *has decreased* `}` yet, because we have not see what the `if`-branches do; but by writing it now, we make sure we won't miss it.

As usual, we try to reason in small pieces: here, that means "one `if`-branch at a time." The first branch (with inessential conjuncts removed) is now

```
# PRE A[:m]<x and m<=p and A[p]<x
m= p+1                                                    (3.5)
# POST A[:m]<x and n-m has decreased      ,
```

where the `p+1` comes from our careful observation that we must set `m` so that `p` is the last index included in `[:m]`, and that last index is `m-1`. That is, to make `m-1==p` we must set `p= m+1`. That being so, we see that `n-m` *has decreased* is satisfied because `m` has increased and `n` has not changed.

When we move to the second `if`-branch, i.e. the `else` part, we can forget about the `then` part: it's already done, and we already know it's right. So we just look at

```
# PRE x<=A[n:]  and p<n and x<=A[p]
n= p                                                      (3.6)
# POST x<=A[n:]  and n-m has decreased      ,
```

all on its own, where `n= p` is justified because `A[n]` is the first element of `A[n:]` (not `A[n+1]` or `A[n-1]`), and because `p<n` ensures that the assignment decreases `n-m`.

Once we put the whole thing together, we have

See Ex. 8.3.

```
# -- binary search:
# -- Find the location of a given value in a sorted sequence.
# PRE Sequence A[0:N] is sorted into ascending order.

m,n= 0,N
while m!=n: # INV 0<=m<=n<=N and A[:m]<x and x<=A[n:]
  p= (m+n)//2                                             (3.7)
  if A[p]<x: m= p+1
  else: n= p
  # VAR n-m has decreased.

# POST A[:n]<x  and  x<=A[n:]      ,
```

where as usual most of the assertions we used along the way have been removed — because we can regenerate them, if we need to, from the ones left behind. The VAR identifies the variant, which we leave as part of the program's documentation because of its importance for this program and the fact that it's not *completely* obvious.

See Ex. 3.9.

3.4 Simplify the postcondition

In this section we will look at two examples. Neither of them is obvious if coded without choosing an invariant first, i.e. to begin designing our program *before* coding begins. Our first example is finding an exponential in logarithmic time.

3.4.1 The logarithmic-time exponential

A base `B` is to be raised to the exponent `E`: the starting point is

```
# -- Raise a given base to a non-negative integer power.
# PRE E:int and 0<=E
...
# POST p==B**E    ,
```

where `**` means "to the power of". By introducing new variable `e`, an invariant `p==B**e` and the loop postcondition `e==E` we then have

```
# PRE 0<=E
p,e= 1,0
while e!=E: { Inv: p==B**e }
  p,e= p*B,e+1
{ p==B**e and e==E }
# POST p==B**E    ,
```
(3.8)

which (in spite of this section's title) takes time *linear* (not logarithmic) in `E`. We have used the "iterate up" (Sec. 3.3.1) strategy to design this simple program. Even in this simple case, however, having an invariant helps: we are more likely to get the initialisation (`p,e= 1,0`) and the loop guard (`e!=E`) right first time... if we have an invariant to refer to.

Indeed, the invariant-finding techniques so far have all been of the form "modify the postcondition" in this or that way (as we did just above). But now we will do that more generally: we "simplify" the postcondition; but how we do that depends of course on what it is in the first place.

And now we design a faster program, taking time only $O(\log \texttt{E})$ — for that we use instead the invariant `B**E == p*b**e` in which `b,e= B,E` have captured the original values, and the goal `B**E` is simplified to finding `b**e` for smaller and smaller `e`. The key insight is that when `e` is even, the calculation can be sped up enormously by using `b,e= b*b,e//2` instead of the `p,e= p*B,e+1` we used in Prog. 3.8 above — the remaining exponent is halved, rather than simply being decreased by 1. When `e` is odd, however, we proceed (more or less) as above (which was the point of doing that one first). That gives the alternative (and *much* faster) program

```
# PRE 0<=E
p,b,e= 1,B,E
while e!=0: { Inv: B**e == p*b**e }
  if e%2==0: b,e= b*b,e//2
  else: p,e= p*b,e-1
{ B**E == p*b**e and e==0 }
# POST p==B**E    .
```
(3.9)

To see just how much faster it is, we note that Prog. 3.8 takes 1,000 iterations for 2^{1000}, but the more efficient Prog. 3.9 needs only 10. If `print(e)` is inserted at

the beginning of the loop body for Prog. 3.9, and `print(p)` at the end of the whole program, we see

```
1000
500
250
125
124
62
31
30
15
14
7
6
3
2
1
1071508607186267320948425049060001810561404811705533607443750388 37 ···
0351051124936122493198378815695858127594672917553146825187145285 69 ···
2314043598457757469857480393456777482423098542107460506237114187 79 ···
5418215304647498358194126739876755916554394607706291457119647768 65 ···
42167660429831652624386837205668069376 ,
```

where the last five lines are taken together, one large 306-digit integer. See Ex. 3.10.

3.4.2 The strict majority

Our second example in this section again illustrates simplifying the postcondition; and it's even more surprising. The obvious algorithm (we will see) is $O(N \log N)$; but with the invariant we will use to design a "non-obvious" version, that can be reduced to linear, that is $O(N)$.

A value `x` is a *strict majority* in sequence `A[0:N]` just when it occurs strictly more than half the time. Thus a is a strict majority in `[a,b,a]`, because $2 > 3/2$ — but it is not a strict majority in `[a,b,a,b]` (because $2 \not> 4/2$). As special cases, the empty list `[]` has *no* strict majority, because every element occurs 0 times, which is not strictly more than half its length (also 0); and in the singleton list `[a]` the element a is a strict majority, because it occurs strictly more than $1/2$ times.

The obvious approach, mentioned above, is to sort the list –which takes time $O(N \log N)$— and then look for the longest run of equal elements. The point of sorting is not in fact to put the elements in order; it is just to make sure that equal elements come together in subsegments, and so can easily be counted. See Ex. 3.20.

But –in spite of all that– we will use quite a different method. We start by writing `csm(x,S)`, for "conditional strict majority" to mean the value `x` is a strict majority in the sequence `S` *if there is one*. (If there is no strict majority in `S`, then `csm(x,S)` allows `x` to be anything at all.)

Our first step in designing the program is then

```
# -- If there is a value occurring strictly more than half the time
# -- in A[0:N] then set x to that value.
# PRE  0<=N
...
# POST csm(x,A[0:N])    ,
```
(3.10)

and the straightforward realisation mentioned above could be made by taking a sorting program "off the shelf", and then using the invariant-finding techniques "replace a constant by a variable" and "split a conjunct" (Secs. 3.2 and 3.3) to code

```
# PRE 0<=N
# Put a "sort sequence A" program fragment here. [1]
# POST A is sorted.

# PRE A is sorted.
c,e= 0,0
# Inv1(0): x occurs c times, and is
#          a most frequent occurrence in A[0:0];
# Inv2(0): and e indexes the start of the longest
#          all-equal suffix of A[0:0].
for n in range(N): # Inv1(n) and Inv2(n)
  if n>=1 and A[n]!=A[n-1]: e= n
  if n+1-e>c: x,c= A[n],n+1-e
# POST x is a most frequently occurring value in A.

# POST csm(x,A[0:N])    .
```
(3.11)

See Ex. 12.4.

If `c>N/2` then `x` is a strict majority; otherwise, there is no strict majority.

Program 3.11 takes time $O(\mathtt{N}\log\mathtt{N})$ however, because its first part –the sorting– takes that long to sort `A`. (The second part of the program is only $O(\mathtt{N})$.)

But our second realisation of Prog. 3.10 will take only linear time. The crucial insight that leads to it is this:

> If some sequence has a *strict* majority, and you remove two unequal elements from it, then what's left has that same strict majority.

A counting argument suffices for that: since the removed elements are unequal, at most one of them can be a strict majority — and so its count decreases by at most 1. But the length of the overall sequence decreases by 2 — and so the same strict majority will remain in what's left, because `c>N/2` $\Rightarrow$ `(c-1)>(N-1)/2`. Our sequence `S` here will be `A[0:N]` initially, and then will become smaller and smaller as we simulate removing elements from it.

It is in that sense that we are simplifying the problem — we make the part of `A` we are considering shorter and shorter, until it becomes so short that `csm` is obvious. To organise that into a neat program, we take as invariant that for any value `x` we have

```
Inv3(n):  If A has a strict majority, then
          [x]*r+A[n:]  has the same strict majority    ,
```
(3.12)

[1] Look ahead to Sec. 12.2 and Ex. 12.4 to see how to put a specification here.

```
# -- If there is a value occurring strictly more
# -- than half the time in A[0:N]
# -- then set x to that value.
# PRE 0<=N

r= 0
for n in range(N): # Inv3(n)                                        (3.13)
  if   r==0:     x,r= A[n],1                  # case r==0
  elif x!=A[n]:  r= r-1            # case r>0 and x!=A[n]
  else:          r= r+1            # case r>0 and x==A[n]
{ Inv3(N) }

# POST csm(x,A)
```

Note that `x` is not explicitly initialised!

Figure 3.1 The conditional strict majority found in linear time

where `[x]*r` is the sequence in which `x` is repeated exactly `r` times. That whole expression is our "simulation" of the part of `A` we are still considering: the entire suffix `A[n:]` together with `r` copies of some `x` that occurred in `A[:n]`. The rest of `A[:n]` has been removed.

The invariant is established by `r,n= 0,0` — and when finally we have `n==N`, the invariant tells us that if `A` has a strict majority, then `[x]*r+A[N:N]`, that is `[x]*r` has that same strict majority, which has to be `x`.

The body of the loop (omitting the increment of `n`) will be

```
# PRE Inv3(n)
if    r==0:    x,r= A[n],1                  # case r==0
elif: x!=A[n]: r= r-1            # case r>0 and x!=A[n]
else:          r= r+1            # case r>0 and x==A[n]
# POST Inv3(n+1)    ,
```

and we can see that it maintains the invariant by examining the cases below:

— **case** `r==0` Here we have `[x]*r+A[n:]  == [A[n]]*1+A[n+1:]`, because both are equal to `A[n:]` itself.

— **case** `r>0 and x!=A[n]` Here we remove the two unequal elements `x` and `A[n]`.

— **case** `r>0 and x==A[n]` Here we record one more occurrence of `x`.

Notice how much simpler it is to examine the cases each one on its own, rather than trying to solve the whole construction at once. The overall program is given in Fig. 3.1. A slightly shorter version is given in Ex. 3.13.

See Ex. 3.11.
See Ex. 3.12.
See Ex. 3.13.

3.5 Cascading invariants: one leads to another

A "cascade" of invariants occurs when, having found one invariant, we discover that in order to maintain it we need a second one. (In fact we saw that in the example of Sec. 2.5, where we first suggested `Inv` but found that we needed another invariant to maintain it: then `Inv` became `Inv1` and `Inv2` was introduced to help maintain `Inv1`.)

The general situation is that when we have found one invariant, say `Inv1(n)`, we might find that we need another invariant `Inv2(n)` to take the step from `Inv1(n)` to `Inv1(n+1)`. The typical pattern is in the loop body is

```
{ Inv1(n) and Inv2(n) }
...                          # Update Inv2 to hold for n+1.
{ Inv1(n) and Inv2(n+1) }
...                          # Update Inv1 to hold for n+1,
                             # given that Inv2 already does.
{ Inv1(n+1) and Inv2(n+1) }    ,
```

and indeed we saw this already in Programs 2.3 and 3.11. Invariant `Inv1` was found using one of the techniques above, and then `Inv2` was derived from that. Here is another example: the maximum segment-sum. It is to...

```
# -- Find the largest sum of any contiguous subsegment of A[0:N].
# PRE 0<=N
...
# POST m= mss A    ,
```

where mss –the maximum segment sum– of a sequence `A` is the largest value of $\sum$`A[l:h]` for any `0<=l<=h<=len(A)`.

See Ex. 3.14.

Of course there is a straightforward program for this, that is

```
m= 0                                          # m for "max"
for l in range(N+1):                           # l for "low"
  for h in range(l,N+1):                      # h for "high"
    s= 0                                      # s for "sum"
    for n in range(l,h): s= s+A[n]
    m= m max s    ,
```
(3.14)

which simply sums *all* subsegments –including the empty ones– and records the largest sum found. Unfortunately however it takes time $O(\mathtt{N}^3)$ to do so. But we will have a closer look at Prog. 3.14 anyway.

Although Prog. 3.14 looks simple enough not to need invariants, it's still good practice (in both senses) to put them in: if the program really is simple, its invariants will be simple too; and even simple invariants help to prevent one-off indexing errors and similar.

For this program, it's clear that it checks the subsegment sums in the order `A[0:0]`, `A[0:1]`, ... `A[0:N]`, then `A[1:1]`, ... and finally `A[N-1:N]`, `A[N:N]`, and so for brevity we'll write `muh(l,h)` for

```
the Maximum segment sum in the above order
Up to just before Here    ,
```
(3.15)

See Ex. 4.6.

where "here" is `(l,h)` – and so, because of "just before", the sum $\sum$`A[l:h]` is itself

```
# PRE 0<=N
n,s= 0,0
while n!=N: # Inv1(n): 0<=n<=N and s==mss(A[:n])
  { 0<=n<N and m==mss(A[:n]) }
  ...                                    } ← Code needed here.
  { 0<=n<N and m==mss(A[:n+1]) }
  n= n+1
# POST s= mss A    ,
```

Figure 3.2 Outer structure of maximum segment sum algorithm

not yet included. With its invariants, and one helpful extra assertion, Prog. 3.14 becomes

```
m= 0
for l in range(N+1):                  # Inv1: m==muh(l,0)
  for h in range(l,N+1):              # Inv2: m==muh(l,h)
    s= 0
    for n in range(l,h):              # Inv3: s==∑A[l:n]
      s= s+A[n]
    { s==∑A[l:h] }    ← Put this postcondition in Inv2 ?
    m= m max s    .
```

Now `Inv3` suggests that the program's efficiency can be improved to $O(N^2)$ by moving its associated assertion `s==∑A[l:h]` into `Inv2`, and that gives

```
m= 0
for l in range(N+1): # Inv1: m==muh(l,N)
  s= 0
  for h in range(l,N+1): # Inv2: m==muh(l,h) and s==∑A[l:h]
    m= m max s
    s= s+A[h] ← Oops!      ,
```
(3.16)

which is shorter, and faster — except for the fact that the `s= s+A[h]` would index `A[N]` –out of bounds– on the very last iteration of the inner loop. That did not occur in Prog. 3.14; why has it occurred here? See Ex. 3.14. See Ex. 3.15.

Instead of pursuing that issue now we'll use a different invariant altogether, based on *iterating up* (Sec. 3.3.1). We aim for a linear, that is $O(N)$ program, and begin with the program of Fig. 3.2, whose the loop body taken alone is this, where for brevity we now write just `mss(n)` etc., that is leaving out the `A[0:]`:

```
{PRE 0<=n<N and m==mss(n)}
...                                  } ← This specifies the missing loop body.
{POST 0<=n<N and m==mss(n+1)}
```

In `mss(n)`, all the subsegments `A[l:h]` with `0<=l<=h<=n` have been considered; and for `mss(n+1)` we must therefore take into account all subsegments `A[l:n+1]` with `0<=l<=n+1`, because they are exactly the subsegments of `A[:n+1]` that are not subsegments of `A[:n]`. And that gives us our second invariant.

Define `mes(n)` –the maximum *end* sum– to be the maximum sum of all suffixes of `A[:n]` and define `Inv2(n)` to be `e==mes(n)`. Our loop body becomes (with some

```
# PRE 0<=N
n,m,e= 0,0,0
while n!=N: # INV 0<=n<=N and m==mss(n) and e==mes(n)
  e= e+A[n] max 0 # Establish e==mes(n+1) given e==mes(n).
  m= m max e # Establish m==mes(n+1) given m==mes(n) and e==mes(n+1).
  n= n+1
# POST m= mss A
```

With the above as a guide, we can write the program as below, a `for`-loop, slightly more concisely. However it is usually easier to *check* a program in its original `while` form.

```
# PRE 0<=N
m,e= 0,0
for n in range(N): # INV 0<=n<=N and m==mss(n) and e==mes(n)
  e= e+A[n] max 0
  m= m max e
# POST m= mss A
```

This version of maximum segment sum runs in linear time, and has no problems with indexing errors.

Figure 3.3 The maximum segment sum

assertions suppressed)

```
{ e==mes(n) }
...
{ m==mss(n) and e==mes(n+1) }
m= m max e
{ m==mss(n+1) }    ,
```

and to "fill in the dots" we realise that there are two possible values for `mes(n+1)` — either it is `e+A[n]`, since `e` is the earlier maximum, or it is 0 in the case that `e+A[n]` is negative. The latter case corresponds to taking the empty suffix `A[n+1:n+1]`.

For example, when `A[:n+1]` is `[3,-4,2,1,5]`, thus with `A[n]==A[4]==5`, the `mes(n+1)` will become `(2+1)+5`, that is `8` because `2+1` was the `mes` beforehand, i.e. in `A[:n]`. But if `A[:n+1]` was instead `[3,-4,2,1,-5]`, then we would take `0` for `mes(n+1)` — since adding the final `-5` to the greatest suffix sum (`2+1`) of `A[0:4]` that we had before would still be less than we can achieve by just summing the empty subsegment.

Thus the missing piece of the loop body is `e= e+A[n] max 0`. Putting it all together gives the program in Fig. 3.3.

See Ex. 3.16.
See Ex. 3.18.
See Ex. 3.19.

3.6 *Exercises*

Exercise 3.1 (p. 32) Combine Prog. 3.1 with your answer to Ex. 1.16 to write *and check* this program

```
# -- Given that today is d days from 1 January 1980,
# -- determine today's date in the form d/m/y.
# -- Assume correctly operating functions DiY(y) and DiM(m).
# PRE  ???
# POST ???
```

Fill in the precondition and postcondition yourself, making sure they agree with the requirements. Take leap years into account, supplying your own functions `DiY(y)` and `DiM(m,y)`.

Use `break` if you wish; but make sure it does not interfere with your program-checking. One approach to that is to code first *without* `break`'s, because the checking is easier with "pure" `while`-loops, and it is more reliable. Once that is done, modify your program to use `break`'s, if it helps to avoid repeated tests or function calls; but carry over your assertions as much as possible.

The `DiMY` function could easily be implemented based a fixed sequence and using `m-1` as index: `[31,28,31,30,31,30,31,31,30,31,30,31]` .

Exercise 3.2 (p. 34) Replay the invariant-finding strategy of Sec. 3.3.1 for a program that "iterates down".

Exercise 3.3 (p. 35) Rewrite the "tweaked" code of Prog. 3.4 as a `for`-loop.

Exercise 3.4 (p. 35) Would there be anything gained by unrolling Prog. 3.4 a *second* time?

Exercise 3.5 (p. 36) For `0<=n<=N` let

```
# PRE 0<=N and A[0:N] is sorted in ascending order.
...
# POST 0<=n<=N and A[:n]<x and x<=A[n:]
```

be the specification of a program that finds value `x` within `A[0:N]` if it is there. Read `A[:n]<x` as "Every element of `A[:n]` is less than `x`." etc.

(a) Why is `A[0:0]<x` always true, no matter what the value `x` is?

(b) If `x` is strictly less than every element of `A`, what will the final value of `n` be?

(c) If `x` is strictly greater than all of `A`, what will the final value of `n` be?

(d) Describe informally what the final value of `n` will be if neither of the above holds, but still `x` does not occur anywhere in `A`.

(e) Describe informally what the final value of `n` will be if `x` occurs in `A` exactly once.

(f) Describe informally what the final value of `n` will be if `x` occurs in `A` more than once.

(g) Write an `if`-statement that branches *after* the above program depending on whether `x` is in `A` or not: that is, complete

```
# PRE 0<=n<=N and A[:n]<x and x<=A[n:]
if ???: # x is in A.
else:   # x is not in A.
```

Exercise 3.6 Suppose `A[0:N]` is a non-empty sequence of strings, i.e. with `0<N`, and that we want to find the longest common prefix of all of them: the longest string `lcp` that is at the beginning of every `A[n]`. For example, if `A` were

```
[ "The cat sat on the mat."
, "The cat said nothing."
, "The cat stole my homework."
]
```
(3.17)

then `lcp` should be set to `"The cat s"`. We can (clearly) concentrate just on the length `p` say of `lcp`, since once that's known the actual prefix itself is known too.

An invariant that might apply is that "All `A`'s have a common prefix of length `p`." whence initialising `p` to 0 is obvious, and the loop should increment `p` until it reaches the first value where the prefixes no longer agree. Checking that agreement requires a second (inner) loop whose invariant would be something like "The $\texttt{p}^{th}$ character of each string in `A[:n]` all agree.", and the guard would be that `n!=N`.

See whether you can write a program based on those two invariants. (Hint: since the condition for the outer loop might not be a simple test, it could be that a `while True:` ··· `break` structure would help.)

Exercise 3.7 As an alternative to the approach in Ex. 3.6, a startling simplification is possible if you do a bit of thinking about longest common prefixes *beforehand*, i.e. before you even go near invariants, yet alone a program.

See Ex. 3.6.

The "longest common prefix of ···" can be built from a single binary operator that gives the longest common prefix of just two strings, because the latter is *associative*: the longest common prefix of `K+1` strings is the longest common prefix of... the longest common prefix of the first `K` strings... and then one more: the $\texttt{K+1}^{st}$ string on its own. In the example above (3.6), the longest common prefix of the first two is `"The cat sa"`, but the longest common prefix of all three is then the longest common prefix of `"The cat sa"` and `"The cat stole my homework."`.

Use that insight to formulate a programming solution with the loops in the opposite order, i.e. with outer invariant "The length of the longest common prefix of all the strings `A[:n]` is `p`." , and see what results. (You still still need two loops: but they could well be simpler than what you got with the approach of Ex. 3.6.)

Exercise 3.8 (p. 37) As we remarked in Sec. 3.3, the integer division `p= (m+n)//2` rounds down, so that for example `3//2 == 1` — it is the *floor* of the actual quotient `1.5`. How would you write an expression that rounds up instead? (Can you do it without using the "ceiling" function?)

Would our binary search have worked if we had rounded up instead of down? If so, say why; if not, explain how it might fail and –in particular– state where our process of program-checking would have discovered the error before it was too late.

Exercise 3.9 (p. 37) Where in our development of binary search did we use that `A` is in ascending order?

Exercise 3.10 (p. 39) The program

```
B,E= 2,10
for e in range(E): { Inv: p==B**e }
  p= p*B
{ p==B**e and e==E }      ← incorrect!
print(p,e)
```

prints `1024 9` — which is the correct answer for 2^{10} but gives a misleading impression. And its assertion is incorrect. Why? *Hint*: See App. G.2.

Exercise 3.11 (p. 41) In Prog. 3.13 the result variable `x` is not assigned-to at all in the case where `N==0`. How can the program be considered to be "working" in that case?

Exercise 3.12 (p. 41) In Prog. 3.13 it is possible that the final value of `r` is 0, even when `N` is not zero: in that case, the program has established only that `csm(x,[x]*0+[])`, that is `csm(x,[x]*0+[])`, which is trivially `True` for all possible values of `x`.

That is, in this case the program guarantees nothing at all about `x` finally, even though it has been busily assigning to `x` throughout the run.

How can the program be said to be "working" in that case?

Exercise 3.13 (p. 41) Explain why Prog. 3.13 is equivalent to this slightly smaller program — only 4 lines:

```
# -- Find a conditional strict majority in A[0:N]: see Sec. 3.4.2.
r= 0
for n in range(N): # Inv3(n): see Prog. 3.12.
  if r==0 or x==A[n]: x,r= A[n],r+1
  else: r= r-1
```

With `A,N= [0,1,1,0],4 ··· print(x)` placed around this program, it prints `1`. With `A,N= [0,1,0],3 ··· print(x)` it prints `0`. And

`A= [0,1,0,1,1,0,1,0,1,1,1,0,1,0,1,1,0,0,0,0,1]` prints `1`, whereas
`A= [0,1,0,1,1,0,1,0,1,1,1,0,1,0,0,1,0,0,0,0,1]` prints `0`.

Exercise 3.14 (p. 42) The difficulties with Program 3.16 in Sec. 3.5 arise partly from its examining *every* subsegment, including all the empty ones, that is `A[n:n]` for every `n` from 0 up to `N` inclusive. If we skipped those (which sum to zero anyway, all of them), we could use `range(l,N)` for the inner loop and avoid the index-out-of-bounds. But skipping the empty segments is perhaps a bad idea. Why?

Instead, rewrite the inner `for`-loop as a `while True: ··· break ···` loop, and so avoid the subscript error while continuing to visit every segment, including all the empty ones. Include invariants and helpful assertions.

Exercise 3.15 (p. 43) In Prog. 3.16 a bug was introduced *("Oops!")* by what seemed to be a perfectly reasonable manipulation of the program just before. What is the root cause of that problem?

See Ex. 3.17.

Exercise 3.16 (p. 44) Use the strategy of Sec. 3.5 to write programs that consider sums of *non-empty* segments only: both the $O(\mathtt{N}^2)$ and $O(\mathtt{N})$ versions. Use the value $-\infty$ if necessary; if you do use it, say why.

Exercise 3.17 (p. 48) If you used $-\infty$'s in your answer to Ex. 3.16, write new versions without them in which you assume that `A` itself is non-empty, i.e. that `0<N`.

Do it by modifying your answers to Ex. 3.16 in just one place, and explain why you do not have to check the program again.

What's the new invariant for `m`? It can't be just `m == mss(n)` anymore, because `mss(0)!=A[0]` and so it would not be `True` initially.

Exercise 3.18 (p. 44) Use the strategy of Sec. 3.5 to write an $O(\mathtt{N})$ program for the maximum segment *product* in `A[0:N]`: both the $O(\mathtt{N}^2)$ and $O(\mathtt{N})$ versions. Include empty segments.

Hint: You will have to "cascade" twice.

Suggestion: Try figuring out what the invariants will be before writing *any* code.

See Ex. 3.20.

Exercise 3.19 (p. 44) Write `lr(n)` for the length of the longest run (subsegment of equal values) in sequence `A[0:N]`. Make and check a program to find the length of the longest run in all of `A`, that is to establish the postcondition `{POST l==lr(N)}`.

Hint: First use Sec. 3.3.1 to suggest an invariant `Inv1`; then use Sec. 3.5 to suggest a second invariant `Inv2`. *Only then* write the program code: it should practically check itself.

Exercise 3.20 (p. 48)

Write `sr(n)` for the length of the *shortest* run of equal values in the prefix `A[:n]` of `A` that cannot be extended on either side. Write and check a program to find the length of the shortest such run in all of `A`, that is to establish the postcondition `{POST l==sr(N)}`.

Hint: First use Sec. 3.3.1 to suggest an invariant `Inv1`; but then use choose the same second invariant `Inv2` that you used in Ex. 3.19. Then write the program code.

Use ∞ in your code (since you're `min`'ing) if it's convenient: don't try to program your way around it. .Test your program however by defining `INF` to be something suitable. What?

Chapter 4

Finding variants

4.1 Introduction

We have by now seen that there are two aspects to checking programs: one is to make sure they have the right answer when they terminate, for which invariants and assertions generally are the main tools; and the other is to make sure they actually *do* terminate. For the latter, we use a different tool: "variants".

Proving termination is *usually* quite easy — which is precisely why it is sometimes overlooked. (Remember Jack in Sec. 1.2.) Easy or hard, the accepted technique for checking program termination is the *variant*, in its simplest form an integer expression that every loop iteration must decrement by at least one, but which can never become negative.

We will as usual begin with the simple cases.

4.2 Simple variants

"Simple" variants are ones like those we have seen already in our earlier examples, integer expressions that strictly decrease but cannot become negative: they are the "other half" of loop checking. That is, the first half (using an invariant) is being able to check that a loop produces the right final state — if it produces a final state at all. That is called *partial correctness*. But how might it not produce a final state? When it iterates forever.

A variant is how we check that it does terminate — which is of course *termination*. And if a loop checks for partial correctness *and* it checks for termination, then it checks for what is called *total correctness*.

Unlike an **in**variant (which is a condition that must be `True` at the beginning of each loop iteration), a variant is an integer expression (not a Boolean) that every iteration of the loop is guaranteed to decrease strictly but at the same time is also guaranteed never to make negative. With those two guarantees, the loop cannot iterate forever.

A typical example of a variant is is found in Prog. 1.22 from Ex. 1.8, partly repro-

duced here:

```
# PRE 0<=N
s,n= 0,N
{ 0<=n<=N }                          # Start the loop at n==N.
while n!=0:  # VAR n
  { 0<n==V }
  n= n-1
  s= s+A[n]
  { 0<=n<V }
# n==0                               Finish the loop when n==0.
# POST s==∑A
```
(4.1)

This program obviously terminates: variable `n` starts at `N` and goes down in steps of 1 until it reaches 0. And in fact the expression `n` all by itself is the variant in this case: it is an integer; it is decreased strictly on each iteration; and it cannot go below 0. The `V` is an "auxiliary variable" which we can think of as capturing the value of the variant `n` at the beginning of each iteration, allowing us to check at the end of the loop body that it has decreased but not below 0, that is `0<=n<V`. The postcondition in this case depends on (the value of) `V` in the precondition).

Another example is in Prog. 1.19, where however the iteration went up instead of down, from `n==0` up to `n==N`; in that case the "actual" variant was `N-n`. In general, a variant can be any integer expression *expr* that either always increases strictly or always decreases strictly (but not a mixture) and which is bounded by a constant in the direction that it is moving: if it's decreasing and bounded below by `L`, the "actual" variant is *expr*`-L`. If it's increasing and bounded above by `H`, then the actual variant is `H-`*expr*.

Even simple variants might not be simple to find, however. For example, in Euclid's algorithm for calculating the greatest common divisor, the gcd of two positive integers `A` and `B`, the loop body decreases either `a` or `b` (but never both). Thus neither `a` nor `b` can be used as the variant on its own, since on any given iteration it might be the other one that decreases. The algorithm is

```
                                     # Euclid's algorithm
# PRE 0<A and 0<B
a,b= A,B
while a!=b:  { Invariant: 0<a and 0<b and gcd(a,b)==gcd(A,B) }
  if a<b: b= b-a
  else: { b<a } a= a-b
# POST a == b == gcd(A,B)
```
(4.2)

See Ex. 4.1.

The important invariant of this loop for partial correctness is `gcd(A,B)==gcd(a,b)`, maintained because

```
gcd(A,B) == gcd(a-b,b) == gcd(a,b-a)     ,
```

and the loop checks as a whole because `gcd(A,B) == gcd(a,b) == a == b` when the loop terminates with the loop condition negated — that is, when `not (a!=b)`.

But here we will concentrate on the variant. It is not either `a` or `b` on its own, because –as we observed above– sometimes one is decreased and sometimes the other. The variant is in fact the sum `a+b`, which is decreased whichever branch of the `if` is taken. It is a *strict* decrease (i.e. of at least one) because we know from the invariant that both `a` and `b` are strictly positive. And we know also from the invariant of the loop that the variant can never become negative: in fact it is always at least 2.

See Ex. 4.1.
See Ex. 4.2.
See Ex. 4.4.

4.3 Lexicographic variants: not so simple

Another kind of variant is called "lexicographic", because it resembles the order of words in a dictionary. Here is an example, where we use the old-style British "imperial" currency: twelve pence make a shilling (12d = 1s) and twenty shillings make a pound (20s = £1). This program models an *ATM* that allows withdrawals until the account is empty, and then terminates:

```
# Allow withdrawals from a bank account until it is empty.
# PRE L>=0 and S>=0 and D>=0

while L>0 or S>0 or D>0:
  read(l,s,d) # Requested withdrawal.
  # Assume input validation ensures 0<=l and 0<=s<20 and 0<=d<12,
  # and 0<l or 0<s or 0<d.
  L,S,D= L-l,S-s,D-d                          # See Ex. 4.7.
  S= S + D//12                  # // is integer division.
  D= D%12                               # % is remainder.
  L= L + S//20
  S= S%20
  if L>=0 write("Please take your cash.")
  else:
    write("Insufficient funds.")
    break # Possible fraud: ATM closed.
```

(4.3)

See Ex. 4.7.

The program is (deliberately) a bit cryptic, because such complications do occur — and we want to be able to check such programs regardless. Here are are concentrating only on whether the bank account will eventually be empty (and not so much on whether the debits are being done correctly — but see Prog. 4.5 in Ex. 4.5). See Ex. 4.5.

What is the variant for termination? It cannot be any of `L` or `S` or `D` on its own, because for each variable separately there are situations in which it does not decrease: withdrawing 1d from £1/0s/0d leaves 19s/11p, where the number of shillings and the number of pence have both increased. The same example shows that it cannot be `L+S+D` either, because it increases the sum from 1+0+0 = 1 to 19+11 = 30.

The variant (of course!) is the number of pence in the account altogether, that is `(20*L+S)*12+D` or, written out as `240*L+12*S+D`. And with some calculations that can be shown to decrease strictly with each withdrawal.

But that is a lot of annoying arithmetic — and an easier approach is to use a *lexicographic* variant, in which the three variables are ordered by importance –first `L` then `S` then `D` in this case– and less important variables may increase as long as there is a more important one that decreased at the same time: in the example above both `S` and `D` increased, but that decreased the lexicographic variant `(L,S,D)` because the more important (in both cases) variable `L` decreased from 1 to 0. See Ex. 4.6.

In more detail: if `l>=L` then `break` terminates the loop immediately; otherwise if `l==0`, so that `L` does not decrease, then `S` or `D` decreases; and if `S` does not, then `D` must. And that is sufficient, without any arithmetic at all. See Ex. 4.8.

Some lexicographic variants are so obvious that we don't notice they are there: for example decreasing an integer considered as a string of three digits `H`,`T`,`U` for "hundreds, tens, units" uses as variant the actual integer the digits together represent. We don't notice it because, in this case, the multipliers are all the same (10) instead of the unusual 20,12 from the pounds/shillings/pence example. See Ex. 4.10.
See Ex. 4.9.

And as for "dictionary order": as you move from "banana" to "apple" in the dictionary, the second letter has "increased" from "a" to "p"; but the (more important) first letter decreased from "b" to "a". Here all of the multipliers are all 27 (if you count blanks). One word is "less than" another if the first letter where they differ is going in the right direction, i.e. towards "a".

See Ex. 4.11.
See Ex. 4.9.

4.4 Structural variants

In programming languages where structured datatypes can be declared directly, something like

```
DATA tree: Leaf int | Node tree tree
```
, [1] (4.4)

a variant function can be defined on the new datatype directly (`tree` in this case): any value in the type is strictly greater than any of its components. For example `Node (Leaf 9) (Node (Leaf 17) (Leaf 23))` is strictly greater than both `Leaf 9` and `Node (Leaf 17) (Leaf 23)`, since they are its two direct components; and the second of those is in turn greater than both `Leaf 17` and `Leaf 23`. The values `Leaf 9` and `Leaf 27` and `Leaf 23` are not greater than anything, and thus play the role of zero in the simple variant. They are however not minimum elements: rather they are (all three) minimal elements: the difference is that a minimum element is less than *all others*, whereas a minimal element has nothing less than *it*. For example, in a family tree the "age in seconds" order has a minimum element, the youngest person (barring conincidences); but the "is a descendant of" order usually has only minimal elements, those with no children.

See Ex. 4.14.

In languages without declarations like (4.4) directly, but where types like that can still be represented by other means (for example sequences represented as linked lists with pointers, or classes and their instances), structural variants can still be used *provided* you have as an invariant that the representation is correct.

As for lexicographic variants, it is often possible to reduce a structural variant to a simple one: in (4.4) for example the simple variant would be the number of `Node`'s in the tree (or equivalently the number of `Leaf`'s, with minimum 1). But using the structural variant directly avoids having to go that extra conceptual step.

See Ex. 4.16.
See Ex. 4.15.

4.5 General variants

In general a variant's value can come from any "well founded" set — all the techniques above are special cases of that.

A *well founded set* is a partially ordered set without infinitely descending chains; and a (non-strict) *partial order* is like a *total* (i.e. ordinary) order except that there can be pairs of elements that are neither greater than nor less than each other. A good example of a partial order, perhaps the simplest one, is sets with the subset order, where it's easy to find two sets that are not equal but neither contains the other (like $\{0,1\}$ and $\{1,2\}$). An *infinitely descending chain* is a sequence of elements, each one strictly less than the one before, that goes on forever: an example of an infinitely descending chain on the total order of integers is $0 > -1 > -2 > \cdots$.

Both partial- and total orders are reflexive, anti-symmetric and transitive: *reflexive* means that any element is less than or equal to itself; *anti-symmetric* means that if

[1] This kind of definition does not seem to be possible in Python, unless you use classes at the same time.

two elements are less than or equal to each other, then they must in fact be equal; and *transitive* means that if a is greater than or equal to b, and b to c, then a is greater than or equal to c.

See Ex. 4.17.
See Ex. 4.18.
See Ex. 4.12.
See Ex. 4.13.
See Ex. 4.19.

4.6 *Exercises*

Exercise 4.1 (p. 50) On the `else` branch of the conditional in Prog. 4.2 there is the assertion `{ b<a }`, a reminder of the fact that the `0<a` part of the invariant will be preserved by the subtraction `a= a-b`.

Why is it `{ b<a }` rather than the exact negation `{ b<=a }` of the `if` condition? And why is that important?

Exercise 4.2 (p. 50) You are given a (rectangular) matrix of numbers, positive negative or zero; and you can choose any row or any column and negate all the numbers in it. Show that by doing that repeatedly, with rows and/or columns of your choice, you can eventually bring the matrix into a state where no row or column has a strictly negative sum.

The difficulty of course is that negating a row (or column) might disturb the numbers in the other columns (or rows) that it crosses, possibly changing their sum from positive to negative (just as fixing one face in a Rubik's Cube can disturb other faces).

This activity can be expressed as a program

```
while "Some row or column has a negative sum.":
  "Choose a row or a column."  # ← What's your strategy here?
  "Negate every number in it."
# POST "No row or column has a negative sum."
```

Can you find a strategy that works, and a variant that checks it?

Exercise 4.3 For safety during a pandemic, queueing people are required to stand on painted circles along a line: each circle is 1.5m from the next. But sometimes several people stand together on the same circle, and have to be asked to move apart: one of them must move one circle forward, and the other one circle backward.

A single police officer does this repeatedly: if two or more people are on the same circle, one is asked to move one circle forward, and the other one circle backward. [2] That might of course create a new multiple occupancy in either direction, and even so the original circle might still be multiply occupied: for example `[0,1,2+2,1,1]` might become `[0,1+1,2,1+1,1]`, in which case there are now three multiple occupancies `1+1`, `2` and `1+1` instead of the original single one.

Show that if there are only finitely many people (and even if there are infinitely many circles), and the police officer deals with just one pair of multiple-occupying people at a time *in any order*, eventually the line will have at most one person per circle — each one of them safely distant from all others.

[2] Strictly speaking, moving just one of the people would be enough to separate her from the others remaining on that circle; but then termination might not be assured. Why not?

Exercise 4.4 (p. 50) There is a train yard with strings of coupled carriages in it: let that be represented by a sequence `cs` of positive integers with each integer being the length of a string of coupled carriages. Thus there are initially `len(cs)` strings of carriages in the yard, and $\sum$`cs` carriages in total. Your aim is to empty the yard.

But only single carriages can be removed from the train yard, one at a time; and the only other move allowed is to separate some string of carriages into two (smaller) strings.

This activity can be expressed as a program

```
while cs!=[]:
  "Choose any single string c of carriages in cs."
  if c==1: "Remove that c from cs (i.e. from the yard)."
  else: # It's not a single carriage.
    "Choose any positive c1,c2 so that c==c1+c2."
    "Replace c in cs with [c1,c2]."  # Break c into two pieces.
# POST The train yard is empty: cs==[] .
```

Show that now matter how the two "any" choices are made above, the above program is guaranteed to terminate. (Note: This is not "Find a strategy to empty the yard." Rather it to show that *any* strategy will empty the yard.)

Exercise 4.5 (p. 51) Integer quotient (`//`) and remainder (`%`) are related by this equality: when `d!=0` we have

```
(q//d)*d + q%d == q    .
```
(4.5)

Use that to check this program:

```
{PRE ss == 20*L + S}
L= L-l + (S-s)//20
S = (S-s)%20
{POST ss == 20*(L+l) + (S+s)}    .
```

What's the significance of the program?

Hint: Use substitution (Sec. B.1.1) to check the assignments, and work from post-condition to precondition. Don't forget (4.5).

Exercise 4.6 (p. 51) In Sec. 3.5 (at 3.15) there was an "up to" (just before here) -order introduced, used to express invariants for a program

```
...
for l in range(N+1):
  for h in range(l,N+1):
    # Process A[l,h].
...
```

that examined every subsegment `A[0:0],A[0:1],...` of a given sequence `A`. Can you relate that to the idea of a lexicographic order, such as might be used for a variant?

Exercise 4.7 (p. 51) What happens in Prog. 4.3 if `D= D-d` makes `D` negative?

Exercise 4.8 (p. 51) Give some other examples of lexicographic variants in everyday life, and how you could reduce them to a simple variant.

Exercise 4.9 (p. 52) Explain the fact that "dictionary order" does not work as a lexicographic variant if the words can be arbitrarily long.

If however there is a maximum length (however large), give a simple argument that dictionary order does work as a (lexicographic) variant. See Ex. 4.10.

Exercise 4.10 (p. 51) We remarked in Sec. 4.3 earlier that three-digit non-negative integers `HTU` (hundreds, tens, units) give a lexicographic order with each `H` being "worth" 10 `T`'s and each `T` being worth 10 `U`'s. Yet in Ex. 4.9 it's stated that lexicographic order does not work for variants if the "words" can be arbitrarily long.

But non-negative integer variants *do* work for checking termination, no matter how many digits they have. What's going on?

Exercise 4.11 (p. 52) Suppose in the example of "banana" and "apple" the lexicographic variant was over six-letter words, including trailing blanks but no other punctuation. Thus "apple" would be "apple␣" — and '␣' would be letter 0, then 'a' would be 1 and finally 'z' would be 26. If you converted "banana" and "apple␣" to simple non-negative integer variants, what would they become? See Ex. 4.12.

Does it decrease strictly from "banana" to "apple␣"?

Exercise 4.12 (p. 53) Suppose variables `a:A` and `b:B` come from types `A`,`B` respectively, and that both of those types have infinitely many values (yet each one separately is still well founded). Thus the "reduce to an integer variant by multiplying" strategy won't work. Is the lexicographic variant "`a` then `b`" permissible even so? See Ex. 4.11. See Ex. 4.13.

Exercise 4.13 (p. 53) Does this program terminate? How (if so) can you reduce it to a simple variant in the style of Ex. 4.8? See Ex. 4.8.

```
# PRE All variables read are non-negative integers.
read(a,b,c)            # a,b,c can be arbitrarily large.
while not(a==b==c==0):
  if c>0: c= c-1
  else:
    read(c)              # c can be arbitrarily large.
    if b>0: b= b-1
    else:
      read(b)            # b can be arbitrarily large.
      { a>0 } a= a-1
# POST a==b==c==0
```

Why does `a>0` equal `True` where indicated?

Exercise 4.14 (p. 52) Give an some examples of genealogical family trees which have minimum (rather than just minimal) elements in the "is a descendant" order.

Exercise 4.15 (p. 52) A (finite, non-circular) list of characters could be defined as the structured type

```
DATA list: Empty | Cons char list
```
,

and for example `Cons 'a' (Cons 'b' Empty)` would be greater than both `Empty` and `Cons 'b' Empty`.

How would you reduce this structural variant to a simple one? Is this equivalently a lexicographic variant?

Exercise 4.16 (p. 52) If you reduce a structural variant to an integer variant (as suggested in Ex. 4.15 or, earlier, by counting the number of `Leaf`'s in a tree), it's possible that several of the original data-type values could reduce to the same integer. Does that matter?

Exercise 4.17 (p. 53) Why is any *finite* partially ordered set well founded?

Exercise 4.18 (p. 53) Is the set of all sets of integers well founded under subset? What about the set of all *finite* sets of integers?

Hint: Is $\{0, 1, 2, \ldots\}$ a strict superset of $\{1, 2, 3, \ldots\}$?

Exercise 4.19 (p. 53) The inhabitants of Flatland are polygons of at least three sides: and the more sides they have, the more their status. The least respected are therefore the equilateral triangles.

When a polygon dies, as they all eventually do, with its last gasp it "spawns" baby polygons, arbitrarily many but all of strictly lesser status than it had: a dying hexagon might spawn a million pentagons, a hundred squares and a billion triangles. But lowly triangles do not spawn: since there are no polygons of degree 2,1 or 0, when a triangle dies it leaves no legacy at all.

Suppose the polygon species began with a single *ur*-polygon, a circle with infinitely many sides (thus a polygon only on a technicality — but there was no one there to complain). When it died, it left behind an enormous (but finite) number of real polygons of arbitrarily large (but finite) status.

In spite of all the polygons that the *ur*-polygon created, will Flatland's polygon-species nevertheless eventually die out altogether?

Chapter 5

Checking assignments and conditionals

5.1 Introduction

So far, our checking of assignments and conditionals has been partly (but not wholly) intuitive, because we have been concentrating on the bigger issue: how to design and check loops. Here however we will look at these basic building blocks more closely.

5.2 Checking assignments

At (1.13) in Sec. 1.5 we checked an assignment statement `s= s+A[1]` (Prog. 1.12), repeated here:

```
# PRE s == ∑A[0:1]
s= s+A[1]                                                    (5.1)
# POST s == ∑A[0:2]      .
```

By substituting `s+A[1]` for `s` in the postcondition `s == ∑A[0:2]`, just as a text editor would, we got `(s+A[1]) == ∑A[0:2]`, and we had to show that the resulting assertion was implied by the precondition. Again we use that reasoning here: because `(∑A[0:1])+A[1] == ∑A[0:2]` we have the implication

$$\texttt{s == }\textstyle\sum\texttt{A[0:1]} \quad \Rrightarrow \quad \texttt{s+A[1] == }\textstyle\sum\texttt{A[0:2]} \quad , \tag{5.2}$$

and because of that (and we need no other reason), we know that Prog. 5.1 works. Notice the *two* steps we have done here, however: the first one was substitution, where we did not need to know about sequences at all; and the second one was about sequences and addition, and we did not need to know about substitution at all.

Here are some further examples:

(a) `x= 1 {POST x==1}` works because `True` $\Rrightarrow$ `1==1`. (Remember that the default precondition is `True`.) Here the `1==1` comes from substitution, but its truth comes from arithmetic. And because `1==1` is unconditionally `True`, this program would work no matter what its precondition was.

(b) `{PRE x==1} x= x+1 {POST x==2}` works because `x==1` $\Rrightarrow$ `(x+1)==2`. Here the precondition has to be what it is: the default `True` would not work.

(c) {PRE x==X} x= x-1 {POST x<X} works because x==X ⇛ (x-1)<X. The X in the precondition is also a variable, but one that is not in the program body x= x+1: that's how we write postconditions that depend on the precondition.

(d) {PRE 0<n==V} n= n-1 {POST 0<=n<V} works because 0<n==V ⇛ 0<=(n-1)<V, and is the typical check you might do for the variant in a loop that is iterating down. We're using V for the initial Value of n.

(e) {PRE V==n<N} n= n+1 {POST V<n<=N} works because V==n<N ⇛ V<(n+1)<=N, and is the typical check you might do for a variant in a loop that is iterating up.

See Ex. 5.1.

(f) {PRE x==X and y==Y} x,y= y,x {POST x==Y and y==X} works because of the implication (actually equivalence — but implication is enough)

x==X and y==Y ⇛ y==Y and x==X .

(g) {PRE x==X and y==Y} x= y; y= x {POST x==Y and y==X} does *not* work, however. With two statements one after the other, we pass the assertions along as usual: the program is actually

```
{ x==X and y==Y }
x= y
{ ??? }
y= x
{POST x==Y and y==X}    ,
```

and there is no "intermediate" assertion ??? which, placed in the middle, would make both

```
      {PRE x==X and y==Y} x= y {POST ???}
and             {PRE ???} y= x {POST x==Y and y==X}
```

work. One of them must fail.

A good (perhaps desperate?) guess for ??? comes from the substitution we'd carry out for the second assignment: that would give us { x==Y and x==X }, which does look unlikely: we are not allowed to assume that X and Y are equal. But if we apply the first assignment's substitution to that anyway, we get { y==Y and y==X }, which is not implied by x==X and y==Y unless indeed x==y.

So not only do we see that x= y; y= x does not swap x and y in general, we discover the precise conditions when actually it would: when x and y are equal already. [1]

(h) As our final example, we show that this well-known "swap x and y" program works:

{PRE x==X and y==Y} t= x; x= y; y= t {POST x==Y and y==X}

[1] In Python, the swap can be done with x,y= y,x, a multiple assignment. But we are pretending that multiple assignment is not available here, because not all languages have it. For example *C* does not.

As usual for chains of assignments, we figure out the assertions from the end towards the beginning, just one substitution after another: we get

Reason from back to front.

```
    {PRE x==X and y==Y} ⇛
↑   { y==Y and x==X }                  # and finally this ...
    t= x
↑   { y==Y and t==X }                    # and then this,
    x= y
↑   { x==Y and t==X }                     # then do this,
    y= t
↑   {POST x==Y and y==X}                    # Start here,
```

... after which we show the implication `x==X and y==Y` ⇛ `y==Y and x==X`, which is what in general allows putting one assertion after another: the first must imply (⇛) the second. In this case however, they are actually equivalent (≡). Alternatively, we could imagine that there is a `skip` at the beginning, giving

```
    {PRE x==X and y==Y} ⇛
    skip
⇛   { y==Y and x==X }    ,            # and finally this ...
    ⋮
```

and then the implication comes from the explanation of how to check `skip`, given in Sec. 5.3 just below.

See Ex. 5.2.

5.3 The "do nothing" statement `skip`

The program {PRE *pre*} `skip` {POST *post*} works just when *pre* ⇛ *post*, because `skip` does nothing at all. [2]

Like zero (as a number), it's not terribly useful on its own. (Who wants to have zero oranges?) But it is useful in checking programs, just as zero is useful in checking arithmetic. One use for it is explaining why, when two assertions are placed directly after another as we saw just above, the first must imply the second.

5.4 Checking conditionals

We have already used conditionals extensively, that is `if` *condition*: while writing and checking programs (Secs. 2.2, 3.3.3, 3.4.1, 3.4.2). But occasionally it's an `if` itself that needs checking. We'll start with an example. This program sorts `a`,`b`,`c` into ascending order:

```
if a>b: a,b= b,a
{ a<=b }
if b>c: b,c= c,b                  } → Concentrate on this step.
{ a<=c and b<=c }
if a>b: a,b= b,a
# POST a<=b<=c    .
```

Let's check the second statement more closely: pulled out of context, and presented for independent checking, it becomes See Ex. 5.6.

```
# PRE a<=b
if b>c: b,c= c,b
# POST a<=c and b<=c     .
```

We see that if the `if` condition `b>c` is `False`, then `b<=c` holds, and so it can be in the postcondition without action on our part. (You can see it there as the second conjunct.) But where does the first conjunct `a<=c` come from? And why don't we have the precondition's `a<=b` any more? These questions are answered by looking more closely at how `if`'s are checked in detail (when we have to).

The first conjunct comes from taking the precondition `a<=b` and the (failed) `if`-condition `b<=c` together: they give `a<=b<=c`, and the `a<=c` in the postcondition is a consequence of that.

The reason we must discard the `a<=b` however –it is *not* in the postcondition– is that it does not (necessarily) hold if the `if` condition is `True`, because `b,c` are then swapped, and `b`'s value can decrease, potentially invalidating `a<=b`.

Let's leave that for a moment, though, while we look at the other branch of the `if`. The overall precondition, and the truth of the condition `b>c`, together give the first extra assertion (1), and applying the assignment `b,c= c,b` as a substitution to the postcondition `a<=c and b<=c` gives the second extra assertion (2):

```
# PRE a<=b
if b>c: { a<=b and b>c }                    # (1)
  { a<=b and c<=b }                         # (2)
  b,c= c,b
# POST a<=c and b<=c     ,
```

Since the assertions are next to each other, we must check as usual that the first implies the second, that is that `a<=b and b>c` $\Rightarrow$ `a<=b and c<=b`; and it does.

And now we can see, by analogy, that what we should be checking in the case that the `if` condition is `False` is the implication `a<=b and not b>c` $\Rightarrow$ `a<=b and b<=c`; and that holds too.

The overall pattern of checking an `if` is easier to see if we include the `else` part of the conditional explicitly: to check See Ex. 5.3.

```
# PRE pre
if cond: thenBranch
else:    elseBranch
# POST post
```

we check each branch separately: we check both

```
# PRE pre and cond
thenBranch
# POST post
```

and

```
# PRE pre and not cond
elseBranch
# POST post     ,
```

[2] In Python `skip` is called `pass`.

which is precisely what we did in the example above — but because in that case the second branch was actually `skip`, we could check it with an implication, because checking `skip` is always done with an implication, as we saw in Sec. 5.3.

Most of the time, conditionals don't need such careful attention; but sometimes they do. (The small program above is one of them; and what if we had a fourth variable `d` as well?) An example is the Binary-Search program from Sec. 3.3.3, in particular checking that it does not go into an infinite loop. The variant for that program is `n-m`, and we need to check that it decreases strictly in each iteration. Here is the part of the Binary-Search program where that happens, with assertions concentrating on the variant:

```
# PRE 0<=m<n<=N and V==n-m
p= (m+n)//2
{ m<=p<n }
if A[p]<x:  m= p+1
else:       n= p
# POST 0<=n-m<V
```

Once we apply the above, and leave out the conjuncts we don't need, the two checks for the branches are

```
# PRE 0<V==n-m and m<=p<n
m= p+1
# POST 0<=n-m<V
```

and

```
# PRE 0<V==n-m and m<=p<n
n= p
# POST 0<=n-m<V     ,
```

and the implications that result from them (after carrying out the substitutions) are separately

$$\texttt{0<V==n-m and m<=p<n} \quad \Rrightarrow \quad \texttt{0<=n-(p+1)<V}$$
$$\text{and} \quad \texttt{0<V==n-m and m<=p<n} \quad \Rrightarrow \quad \texttt{0<=p-m<V} \quad .$$

Both of those hold.

Notice how we introduced the auxiliary variable `V` so that we could check that it was decreased by the loop body.

5.5 *Exercises*

Exercise 5.1 (p. 58) Example (e) in Sec. 5.2 can be (re-)written as a decreasing variant, bounded below by 0: it becomes `{PRE V==(N-n)>0} n= n+1 {POST V>(N-n)>=0}` — that is, the (decreasing) variant is `N-n`, not just `n` on its own. What is the implication that you have to check for that?

Exercise 5.2 (p. 59) Show that the program

```
{PRE x==X and y==Y}  x= y-x; y= y-x; x= x+y  {POST x==Y and y==X}
```

works. *Hint*: Remember to figure out the assertions from the end of the program towards the beginning. Simplify the conditions, if you can, as you go along.

See Ex. 5.5.

Exercise 5.3 (p. 60) Here is how assertions are placed inline when checking conditional statements:

```
# PRE pre
if cond:
  {PRE cond and pre}
  ...
  {POST post}
else:
  {PRE not cond and pre}
  ...
  {POST post}
# POST post     .
```

Give a similar procedure for `if` on its own (no `else`). *Hint*: Use the above, with `skip` for the else part. (See also App. B.3.3.)

Then show how to check `if` and `elif` and `else`.

See Ex. 1.1.

Exercise 5.4 This program sets `m` to the largest value among `a`,`b`,`c` and `d`:

```
m= a
if b>m: m= b
if c>m: m= c
if d>m: m= d
```

Add assertions between the statements that will allow the program to be easily checked. *Hint*: Write `[a,b,c]<=d` for `a<=d and b<=d and c<=d` etc.

How many paths need to be checked in this program? If you added an extra variable `e`, how much longer would the program have to be?

Exercise 5.5 Use your techniques from Ex. 5.3 to check the program

```
{PRE True} if a>b: a,b= b,a {POST a<=b}     .
```

Then check the "sort three variables" program

```
{PRE True}                    ← True is the default precondition.
if a>b: a,b= b,a
if b>c: b,c= c,b
if a>b: a,b= b,a
{POST a<=b<=c}
```

Hint: Use Sec. 5.4 and the substitution technique for assignments, working from the overall postcondition backwards towards the `True` precondition (which can be omitted — it's the default). You will find ⇛-constraints that suggest what the intermediate assertions should be.

Exercise 5.6 (p. 60) Add assertions to check the "sort *four* variables" program

```
{PRE True}
if a>b: a,b= b,a
if b>c: b,c= c,b
if c>d: c,d= d,c
if a>b: a,b= b,a
if b>c: b,c= c,b
if a>b: a,b= b,a
{POST a<=b<=c<=d}    .
```

Hint: See the hint for Ex. 5.4, which suggests e.g. that the `a<=c and b<=c` from there could be written more concisely as `[a,b]<=c`. See Ex. 5.7.

Exercise 5.7 (p. 63) The following (incomplete) Bubble Sort program uses the ideas of Ex. 5.6 to sort a whole sequence `a[0:N]` of `N` values instead of just four variables `a,b,c,d`. Its invariants are suggested by the assertions you would have used in your answer to Ex. 5.6. See Ex. 5.6.
See Ex. 5.8.

```
{PRE ···}
for i in range(?1?): # INV Inv1
  for j in range (?2?): # INV Inv2
    if ?3?: ?4?
  { ?5? }
# A[0:N] are in their final sorted positions.
# POST The whole of A is sorted.     ,
```

where `Inv1` is "`A[i:] are in their final sorted positions.`" and `Inv2` is `A[:j]<=A[j]` and

(a) `range(?1?)` should establish `Inv1` initially, and imply that `A[0:N]` are in their final sorted positions when the outer `for`-loop has terminated. What should `?1?` be?

Hint: Use `range(high,low,-1)` "iterate down" in the `for`-loop.

(b) `range(?2?)` should establish `Inv2` initially, and imply `?5?` when the inner `for`-loop has terminated. What are `?2?` and `?5?` and why?

(c) The code of the inner-loop body will be

```
{PRE A[:j]<=A[j]}
if ?3?: ?4?
{POST A[:j+1]<=A[j+1]}
```
(5.3)

What do `?3?` and `?4?` have to be so that the code checks?

(d) Explain in words why your completed Prog. 5.3 checks. Note however that Bubble Sort is quite inefficient.

Hint: Do the `else` part first.

See Ex. 5.7.

Exercise 5.8 Another elementary (and similarly inefficient) sorting program is Insertion Sort:

```
{PRE ···}
for i in range(?1?): # INV A[i:]  are in order.
  for j in range(?2): # INV Inv2 ?6?
    if ?3?: ?4?
  { ?5? }
{ A[0:N] are in order. }
{POST ···}    .
```

Complete the missing assertions and code fragments. (Note the new one, `?6?`.) Like Bubble Sort, Insertion Sort is quite inefficient.

Exercise 5.9 This simple program sorts a sequence `A[0:N]` by continuing to swap until no more swaps can be done. What's its invariant `?I?`? What's its variant `?V?`? What should the assertion `???` be?

Remember that an invariant is a Boolean condition, whereas a variant is a non-negative integer.

```
# -- Sort sequence A[0:N] into ascending order.
# PRE 0<=N
while True: # Var: ?V?
  noSwaps= True
  for n in range(N-1): # Inv(n): ?I?
    if A[n]>A[n+1]:
      A[n],A[n+1]= A[n+1],A[n]
      noSwaps= False
  { Inv(N-1) }
  if noSwaps: { ??? } break
# POST A[0:N] is sorted
```

Like Bubble Sort and Insertion sort, Exchange Sort is quite inefficient.

Chapter 6

Summary of Part I

6.1 What's not obvious

Programming is both a vocational skill [1] and an engineering discipline. As a vocation, people learn to "code" (so-called to make it sound less forbidding?) at school, and possibly from their parents, siblings and friends too. Coding is not very hard — and so by the time they start university, most whose interests lie there are already quite good at it; and a substantial subset of those have discovered that –more than that– they have a natural aptitude for programming, and even computer science in general.

There is a difference though between a vocation and a discipline: it's the difference from being able to build a garden shed yourself (with help perhaps from your family) and *organising* the construction of an office block or a bridge or an aeroplane (with help from perhaps hundreds of people you've never met before). In the first case, you have an instinctive understanding of what to do, based on (years of) experience, and you have a built-in rapport with the people who are helping you: you probably eat breakfast with them every day. In the second case, instinct is only a start, and you might have never met the team who are working for you. It's the difference between being able to tune your motorcycle yourself, and being able to design one from scratch, and then organise its being manufactured and sold.

That difference, the small vs. the large, the instinct vs. the science, is why there are 4-year university courses in Engineering, indeed in various different *kinds* of engineering, and professional bodies that accreditate engineers so that we can trust what they build, and what they organise others to build. What is the analogue for programming?

Unlike engineering in general, the *technical* gap between vocation and discipline in programming is actually

astonishingly small,

although the organisational and social gap remains as large as it is for any other branch of engineering. And the technical gap is very easily bridged — if only you are given the chance. We need accreditation there too.

The techniques explained in our Part I show that –for example– instead of figuring

[1] "Vocational" skills pertain to some occupation, trade or business or profession. [Macquarie Dictionary]

out how to write a loop by following the conventional thought processes

- Imagining what one of its iterations must do ("It must add another one of the array elements to the current running total.")
- Working backwards from the initialisation to determine what the first step must be ("After the first addition, the running total must be the first element alone: so we start at zero.")
- Figuring out what the last iteration must do in order to make the loop complete ("We mustn't forget to add the very last element, so the index on the last iteration must be `N-1`... Or is it `N`?") (6.1)
- Checking the edge cases ("Does it still work if the sequence is empty?")

... instead you might do better by remembering Program 1.18 (for example), and send those thought processes in a different, and startlingly more efficient direction once you get the hang of it:

- The *invariant* is that the running total is the sum of the sequence prefix so far. And to preserve that invariant you simply add the current element to the sum, and lengthen the prefix by 1 position.
- At the beginning, "the prefix so far" is empty, so the running total is 0. (6.2)
- At the end, "the prefix so far" must be the whole sequence.
- The edge cases are handled automatically.

You might not have heard of a loop invariant before: but you have now. Are invariants "unconventional"? Perhaps. But most of the time they make your programs easier to write, easier to check and sometimes even more efficient. And soon they will be the conventional process for you.

But the difference between (6.1) and (6.2) above is not necessarily "an amount of work". Indeed, for small programs the amount of work is about the same. For larger programs, however, the second one (6.2) is actually *less* work in the end, when you factor in the debugging and other program maintenance that might have to be done. Instead of "an amount of work", the difference is a *point of view* — plus the astounding advantage that the (6.2)-approach brings, these three extra benefits:

1. Approach (6.2) scales up.

   Checking a program in terms of "What steps does it carry out in this situation? And this situation? And this situation that my customer mentioned only yesterday?", as in (6.1), becomes harder at a much faster rate than simply the size of the problem. Remember Exercise 1.1, that showed how in that small program, every new variable *doubled* the amount of checking needed — if done in the conventional style.

2. Approach (6.2) makes it easier for team members to collaborate.

   The idea of documenting "What's *true* at this point in the program?", as in (6.2), instead of "What does this bit of code do?", as in (6.1), is the glue that

binds together large teams of programmers, and that allows large programs to be divided into pieces that can be handled separately.

3. For very important programming projects, where failure is extremely expensive, even catastrophic, the approach of (6.2) can be to some extent be automated. (See Part IV.) For (6.1), that is much more difficult.

Because the difference in effort required, –i.e. to move from thinking in the (6.1)-style to thinking in the (6.2)-style– is *so small* but makes *so much difference*, we have started in Part I with programs that you already know and understand — and that means you can concentrate on how you are thinking about them, rather than having to worry at the same time about what those programs are actually doing.

Later, however, we will see the payoff that "What's true here?" thinking brings. As just one example, the program in Chp. 16 is just four lines line (plus another four that are a symmetric copy of the first four): but it took 15 years to discover that program, and the person who did is still celebrated today. [2] There are more than 50 program-execution paths to think about were we to check those four lines of code conventionally.

See Ex. 16.9.

What's *not* obvious about these insights into thinking about your programs in a different way? The only non-obvious thing (before you read this) was that it's possible to do it.

You were not told; and nobody showed you how.

6.2 What *is* obvious

Once you know about this style of thinking about programming, and have had some practice, it is indeed pretty obvious in most cases how to do it. Figure 1.1 showed a flowchart for summing a sequence (Prog. 1.18 again). And it's very clear what's going on: the operations of the program occur within the boxes –rectangles for assignments, diamonds for tests– and the lines (with arrows where necessary) show how the program moves from one action to the next.

Inside each box is written what it does; and you can simulate the program's actions, execute it "in your head" just by following the lines around.

But there is more to these flowcharts: on the lines *between* the boxes is written "what is true when the program is travelling along this line". The principal reason it is so much easier to use those for checking –to use what's *between* the boxes rather than what's *in* them– is that if you use the boxes alone, then to check whether a particular box is doing the right thing, you must look at the box before, to see what *it* did; and for that one, you must look at the one before it, and so on.

On the other hand, if you use what's written on the lines between the boxes, you need only check whether the box successfully takes you from what was written on its incoming line to what is written on its outgoing line. If that checks, then this box is working, and you do not need to check *other* boxes to see whether *this* box works. And even more significantly,

> If you check *all* the boxes in that style ("Does it take you from its incoming line to its outgoing line?") even if you check them all *separately*, even at different times, even with different people... and they all work, then the whole program works too.

[2] Forty years later, at time of writing.

That is, those other boxes can be checked by other people, or yourself on other days; and what's written on the lines, our pre- and postconditions, are the "glue" that puts all the checks together, what reminds you tomorrow what you have to check then (even if by then you have by then forgotten the details of what you checked today). Or it tells your colleagues what they have to check, separately, in order for the whole thing to work properly when it's assembled.

The more you practise checking programs by using the techniques we've introduced in this Part I, the more you will appreciate it, in both senses of that word: you will be glad you were told about it, and you will understand how it works.

Part II

Data structures and their encapsulation

Chapter 7

Introduction

7.1 Data vs. data *structures*

The data values of a program are stored in its variables: in the simplest cases they are things like numbers (integers, floating point), characters, Booleans: those are called "basic", or "primitive" data *types*. Using those basic types, "building" on them, gives us things like *sequences* of integers, characters, Booleans, strings, and *sets* of such things, and *tuples* of them and so on. There you get more than just the basic: you get data *structures*.

As an example, let's take people's names and ages, both primitive types, [1] and make a data structure from them: we can build a directory using a sequence variable `dir` as a sequence of triples. It might be initialised with the assignment

```
dir= [("Alex", "Smith", 35), ("John", "Jones", 60)]    ,
```

so that it stores that information for just two people: Alex Smith is aged 35; and John Jones is aged 60. The data structure here, the type of `dir`, is a sequence of triples, with each tuple based on two basic types: two strings and an integer.

The family name of the first person in `dir` is `dir[0][1]`, where `dir[0]` is the first person (because we index lists from 0), and `···[1]` is the second component of that triple — `"Smith"`.

You can do a lot with that structure. But if your directory `dir` grows to contain thousands of entries, or you want to store middle names as well, then two issues arise:

1. Having a simple sequence of entries might make it quite inefficient to search the whole directory: using linear search might take too long.

2. Expanding the directory to include middle names (with `""` for "none") would give things like

   ```
   dir=  [("Alex", "", "Smith", 35),
          ("John", "Paul", "Jones", 60)]    ,
   ```

   and the `dir[0][1]`, to access the family name "Smith" of Person 0, would have to be changed to `dir[0][2]` –everywhere, throughout your whole program– because all the family names are now in the third position.

[1] Strictly speaking, the names' type is not basic, because it is a sequence of characters. We'll ignore that however: where you start is to some extent flexible. Isn't a character in turn a sequence of bits?

The solution to both problems is to use procedures and functions to access and update the data structure; and then when you want to have a faster lookup you might alter the `search` function to use binary search instead of linear search (Sec. 3.3.3). But you would at the same time have to alter the `add` procedure to put new entries in the right place, so that the sequence of triples is always sorted (since binary search doesn't work, otherwise). All the various procedures would have to be modified in a consistent way.

The second problem (a simpler one) is also solved by using procedures and functions: in that case you would probably have to make only one change: the "`getFamilyName`" function would have its "`···[1]`" altered to a "`···[2]`".

7.2 Data encapsulation

Data encapsulation is the practice of gathering all the code for the declaration, initialisation, access and updating of a data type, all of it, together in one place for each conceptual data structure. So for example the declaration and initialisation of `dir` just above, and its related functions, would all be grouped together. Data related to another concept (say a directed graph) would also be grouped together, but in its own, separate encapsulation.

The advantage of doing all that, and the topic of this Part II, is how that encapsulation is organised and how –in particular– changes like the two mentioned above (many entries, middle name), and others, can then be carried out in a way that does not undo the checking that might *already* have been done in the surrounding program that used the data structure in its original form. And it is as well likely to take less time to carry out than it would have without the encapsulation.

Very simple examples will be used to illustrate the process, pretending we have discovered the idea of encapsulation, and its advantages, for the first time. See Sec. G.3 however for a brief discussion of how encapsulation has evolved into Object-Oriented programming, and the extra concerns that apply.

As in Part I, the key tool is assertions and invariants.

Chapter 8

Coupling invariants

8.1 Introduction

Invariants –we recall– are assertions that are intended to describe things that are "always true", like the condition that "always holds" every time your program begins a particular loop's body. Those are *loop* invariants, of course.

But invariants are used in many places, and for many purposes: here we look at "coupling" invariants that state the relationship that "always holds" between two different representations of the same data.

We'll use "sets" as a running example of coupling invariants.

8.2 Implementing sets as sequences

Our first illustration of coupling invariants concerns writing code at a "low level" that expresses an idea at "higher level". The low level is sequences; the higher level is sets.

Suppose we can write programs that use finite sets directly, with statements like `ss= {}` that assigns the empty set to set `ss`,[1] or like `ss= ss∪{s}` that adds element `s` to set `ss`, or like the conditional `s∈ss` that determines whether `s` is in `ss`.

This is the "higher level".

The reality at the "lower level" however might be that actually you *can't* use sets directly in your code, because your programming language doesn't support them:[2] but you still have to write that program. In that case, you might decide to represent the set `ss` as a sequence `qs`, and write your code directly at that lower level, translating as you go: you'd write

- `qs= []` to initialise the "set" to the empty set, and
- `qs= qs+[s]` to add an element `s` to the set, using sequence concatenation `+`, and

[1] In Python you must write `set()` for the empty set: the two-bracket notation is used for empty "dictionary". We will use `{}` here, however.

[2] Python *does* support sets; but here we are pretending it does not: and the language you are *actually* using might not. Sets are not build in to *C*, for example. The techniques we'll use here apply to more or less any language.

- the small program [3]

```
b= False
for q in qs:
  if q==s: b= True; break
```
(8.1)

  for setting Boolean `b` depending on whether `s` is in `qs`: instead of a single statement, we must use a small linear-search loop to search the "set" for the element. In this case the low level is the code you are writing, and the high level is in your head.

See Ex. 8.1.

So now imagine that you wanted to add a new operation, to take an element *out* of the set, i.e. to remove it: with sets directly, i.e. "in your head", you are thinking `ss= ss-{s}`. But that's not what you write: you are implementing the set in code directly, as you go, as a sequence `qs`. And so you must now decide whether to take "just one" `s` out of `qs`, or "all of them". How do you decide?

In the implementation above, you would have to take them all out, and to learn that you have to do that, you would have to examine the code of `add` (which might be pages away) to see whether it is possible for elements to be repeated in `qs`. And indeed we would find that adding `s` "a second time" will include it twice.

And –another variation– what if for efficiency of searching you decided to keep the sequence `qs` in ascending order? In that case, you could use binary search for membership checking, but a loop would be necessary for insertion (and deletion).

This example simply reminds us that if we are using a collection of procedures and low-level data to implement a higher-level concept, the collection must be consistent: its members must cooperate with each other. Here we will show that to do that it's a good idea to group them all together, and to introduce a "coupling invariant" to state clearly and explicitly what "cooperating" with each other actually means.

In fact most programs (and programmers) carry out "representation" exercises like the above in almost every program. Often it is trivial, and easy to keep track of. But a clear advantage of putting all the information in one place, and having a coupling invariant, is that it makes the consistency checking so much easier. If say there were 5 procedures associated with the concrete representation of the data, then checking each against all the others would amount to 10 checks. With a coupling invariant, it would be only 5 checks; if there were 10 procedures, it would be 45 checks vs. 10. The more procedures there are, the bigger the difference becomes.

As remarked in the Introduction to this chapter (Sec. 8.1), we have seen assertions already as "What's true at this point in the program." comments; and we have seen a more specialised use of them as "What's true every time this loop begins?" What we will now see is how assertions can describe the connection between the variables we would *like* to have –like sets– and the variables we actually *do* have, in this case sequences. Those assertions are called *coupling invariants*, and the coupling invariant for our example above is

> The elements in the sequence `qs`, ignoring repeats, are the members of the set `ss` that we are actually thinking of as we write the program. (8.2)

It's a condition on variables' values, just like the others we have written, in this case containing both the variables `ss` and `qs` — that is, both the high level and the low

[3] Python implements `in` for sequences directly; but many languages do not. We are pretending here that Python does not.

level, the "abstract" and the "concrete". It tells us for example that `add(s)` can append `s` to `qs` even if `s` is already there; and it tells us that `remove` must check the whole sequence in order to find every occurrence.

8.3 Sentinels as "high level" data

Another use of coupling invariants is to implement "sentinels".

A *sentinel* is an extra, "special" value added to data in order to simplify program code. For example, we might want to find program code "`...`" such that this program would check:

```
# -- Find the position of some value in a given sequence, if it's there.
# PRE 0<=N
...
# POST A[n]==x if x∈A else n==N
```
(8.3)

Notice that the requirements are (deliberately) a bit loose, but do state the general purpose of the program. The postcondition however is more precise: it says which sequence (`A`) is to be searched, which variable (`n`) is to receive the result, and how "`x` is not there" is to be indicated.

We'll do it with linear search.

Linear search is not a hard program. But it would be slightly easier to code if we knew that `x` was in `A` somewhere, because the search for `x` would then be guaranteed to succeed — we would not have to worry about "falling off the end". See Ex. 3.5. See Ex. 8.2. See Ex. 8.3.

To use a sentinel for that, we imagine extending `A` with an extra element `x` at position `N`, "beyond the end" of `A` — that is, we would use a sequence `B[0:N+1]` that was equal to `A+[x]`. We'd therefore have `B[n]==A[n]` when `0<=n<N` but `B[N]==x` in particular. Searching for the `x` that *might* be in `A` is then the same as searching for the `x` that *must* be in `B`... somewhere. If `x` is not in `A` at all, then it won't be in `B[0:N]` either, in which case our program will set `n` to `N` — that's exactly what the sentinel `B[N]==x` is for.

Here is the program that uses our "imaginary", high-level `B` with its sentinel:

```
# -- Find x in B[0:N+1].
# PRE 0<=N and x∈B              ← Notice the extra precondition.
n= 0
while x!=B[n]:
  n= n+1
# POST 0<=n<=N and x∉B[:n] and B[n]==x
```
(8.4)

The extra "`x∉B[:n]`" in the postcondition is important: without it we could simply write the code `n= N`! And as usual the postcondition is just the invariant `and` the negated loop condition. See Ex. 8.4.

What we will now look at is how to translate that Prog. 8.4 to a program about our "actual", low-level sequence `A` instead of the imaginary `B`. And that is done with a coupling invariant.

As we have said, a coupling invariant –like other invariants– states a condition that the program maintains: it links two or more variables together. In Sec. 8.2 the variables were `ss` and `qs`; here they are `B` and `A`.

In the earlier example of Sec. 8.2, we wanted to think in terms of sets, but had to program in terms of sequences. The coupling invariant expressed that the "set we

are thinking of" is the set "that the sequence represents". In addition, the coupling invariant might express extra constraints like "the set has no more than `N` elements", or "the sequence is sorted" or "the sequence does not contain repeated elements". We don't do that now, but we will return to it in Sec. 10.2.

In this example we do something much simpler: we use a coupling invariant to link our original `A` to our fictitious `B`. Indeed we mentioned it above already: it's just

```
B == A+[x]    .
```
(8.5)

Since neither `A` nor `B` is being updated by our linear-search program, we do not have to figure out corresponding updates to keep the coupling invariant true. We do however need it to convert references to `B` into corresponding references to `A`. There are three places that `B` is referred to in Prog. 8.4:

- in the precondition, with `x∈B`;
- in the loop condition, with `x!=B[n]`; and
- in the postcondition, with `x∉B[:n] and B[n]==x` .

One of the principal advantages of invariants generally (i.e. loop invariants as well) is that they allow us to deal with things one at a time, knowing that if each thing separately respects the invariant then all of them together will respect it as well. In the case of a loop invariant, that translates into checking that the invariant is established by the loop initialisation (one thing), then checking that the loop body maintains it (another thing), and then being able to use the fact that the invariant holds when the loop terminates — in effect the "all of them together" payoff. (In Sec. 8.2 it was the point about having to do only 5 checks instead of 10.)

The same thing will happen with this small program:

- In the precondition `x∈B` becomes `x∈(A+[x])`, which is trivially `True`. So it can be deleted altogether from the precondition.
- The loop condition `x!=B[n]` becomes `x!=(A+[x])[n]`, which we deal with in two cases: either `n==N` or `n!=N`. In the `n==N` case it's trivially `False`; but in the other case it is just `x!=A[n]`. Thus the loop condition translates to

  ```
        n==N and False
  or    n!=N and x!=A[n]    ,
  ```

  and that simplifies to just `n!=N and x!=A[n]` .

  See Ex. 8.5.

- In the postcondition, the `x∉B[:n] and B[n]==x` becomes

  ```
  x∉(A+[x])[:n] and (A+[x])[n]==x    ,
  ```

  and again we can separate it into the two cases. This time we get

  ```
        n==N and x∉A and True
  or    n!=N and x∉A[:n] and A[n]==x    .
  ```

  If we discard the "`and True`", and weaken the postcondition by discarding the `x∉A[:n]` because the requirements didn't ask for the *first* occurrence of `x`, then we get

  ```
  # POST A[n]==x if n!=N else x∉A    .
  ```

When we make those substitutions, replacing references to `B` by corresponding references to `A` as figured out just above, we get the linear search program

```
# -- Find x in A[0:N].
# PRE 0<=N
n= 0                                                    (8.6)
while n!=N and A[n]!=x: n= n+1
# POST A[n]==x if n!=N else x∉A    .
```

Curiously however, the postcondition of this program is not the same as the postcondition of Prog. 8.3 above.

See Ex. 8.6.
See Ex. 8.7.

We are not of course saying that Prog. 8.6 is complicated, or that it could not have been written in this form directly. What we *are* saying is that here we have used the (coupling) invariant to *construct* that program; and if Prog. 8.4 (the "`B` version") is correct then Prog. 8.6 must be as well — there's no need here to check it again.

8.4 Writing abstract data-types

We now put the ideas of the two previous sections together: having simultaneously an "abstract" and a "concrete" view of our data (sets vs. sequences in Sec. 8.2); and using a coupling invariant to transform one program into another (usually abstract, using sequence `B` with its imaginary extra element, transformed into the concrete use of sequence `A` in Sec. 8.3).

We begin by returning to our set example (Sec. 8.2) and collecting the set-valued variable `ss`, and the functions that use it, all together in one place: [4]

```
class Set: # Pretend that sets cannot be implemented directly:
  local ss= {}                          # Start empty, yet
  def makeEmpty:
    ss= {}                            # we have no "{}"...
  def add(s):                                               (8.7)
    ss= ss∪{s}                              # ...nor "∪",
  def isIn(s):
    return s∈ss                                 # nor "∈".
```

This code does not itself make a set: it is more like declaring a `Set` type. To make a set instead, we write things like `ss1= Set()` to make a set-valued variable, in this case called `ss1`. With `ss2= Set()` we could make a second one.

See Ex. 10.2.

Given that (we are pretending) there are no "real" sets in our programming language, the code above is fantasy: it won't compile, and we can't run it.

So it's tempting to call Prog. 8.7 above "pseudocode" — but it is much more than that. Here's why. Suppose we have a program that uses our (more than) "pseudo-

[4] We're using something like the Python syntax for declaring a "class", and will explain it as we go. Remember though that we are pretending that the statements `ss= ss∪{s}` and similar are *not* Python (even though actually they are), because sets might not be in whatever language we are actually using. `local` is not a keyword of Python, where global/local is determined implicitly. We use it here to make reasoning easier.

code" like this:

```
{PRE True}
ss1= Set()
ss1.add(2*2)                                                    (8.8)
has4 = ss1.isIn(4)
{POST has4}    .
```

It initialises set `ss1` to empty, adds the value `2*2` to the set, and then sets Boolean `has4` to whether the element `4` is in the set. Can we check this program, even though we can't compile it, can't run it?

Yes. We can check this program with the methods we have already, by inserting assertions in between the statements. We "inline" the procedure calls, that is we replace the call to the procedure with the code that's *in* the procedure (with the appropriate substitutions done) — and then we check *that*. As usual, we start from the bottom (the postcondition) and work towards the top (the precondition), getting

```
{PRE True}                          # And see that indeed it checks. ←
{ 4∈{4} }                                        # Simplify.  ↑
{ 4∈{}∪{4} }                  # Inline local, substitute.  ↑
ss1= Set()
{ 4∈ss1.ss∪{4} }                                 # Arithmetic.  ↑
{ 4∈ss1.ss∪{2*2} }      # Inline add(2*2), substitute.  ↑      (8.9)
ss1.add(2*2)
{ 4∈ss1.ss }                                  # Inline isIn(4).  ↑
{ ss1.isIn(4) }                                  # Substitute.  ↑
has4 = ss1.isIn(4)              # Work towards the front.  ↑
{POST has4}    ,                   # Start checking here.  ↑
```

and we see that indeed it checks. But in doing that check, we have done something that ordinary pseudocode cannot do, and that is why this is more than pseudocode. *You cannot reason rigorously about pseudocode.* The best you can do is "execute it in your head."

> We have reasoned about code (the set operations) that is not only not really there, it can *never* be there in the language we imagine we are using. (8.10)
>
> But –in spite of that– we can check our programs anyway.

This is what the abstract/concrete approach allows us to do. If we check the abstract, and construct the concrete using the methods of this Part, we do not need to check the concrete: the checking is done during the construction, a beneficial side-effect of the methods we are using.

The reason we call Prog. 8.7 an "abstract" data type is that it *is* a data type, and we can reason about it, but it *abstracts* from the details of the particular programming language we might be using.

8.5 Coding concrete data-types

At (8.10) we saw that we were able to check a real program Prog. 8.8 without actually having real, runnable code for the data it was using. We did it at Prog. 8.9 by using the abstract version of the data type *as if* it had been real code. But in the end do we need real code, in a real programming language that we can compile and run on a real computer, if ever we are to use our program. And so we now use the ideas of Sec. 8.3 do do the complementary thing: we reason about the abstract data-type *itself* without having to worry about the programs (however many and varied they might be) that might be using it.

That's worth emphasising: on the one hand, we can check programs that use the data *without* having the actual code that manipulates the data — we did that in Sec. 8.4 above. But we are now about to see that we can implement the actual code that manipulates the data *without* having to see the programs that use the data. That is, we won't have to look at "use cases" at all, things like (8.9), while we are figuring out how to implement the encapsulation in Prog. 8.7.

It's coupling invariants that make it possible to separate the two activities: writing the code without having an implementation of the data type; and implementing the data type without knowing anything about the code that's using it. Here the coupling invariant was given in Sec. 8.2 at (8.2), and we will use it just as we did in Sec. 8.3 for sentinels, where `B` was abstract and `A` was concrete.

Translating an abstract data-type to a concrete data-type is done in three steps, and we will use both Sec. 8.2 and Sec. 8.3 as examples.

8.5.1 Add concrete (auxiliary) variables

We'll start with the `set` data-type, and convert it gradually into one that uses sequences instead. Our first step is to add the concrete, sequence-valued variables –in this case just `qs`, a single sequence– to the abstract data-type, and add statements that maintain the coupling invariant; that is, we will temporarily have *both* `ss` and `qs`; but to begin with `qs` will be only an auxiliary variable, one that has no effect on the execution of the procedures (methods) of the data type. We have (temporarily) a kind of abstract/concrete hybrid. This first step gives

```
class Set:
  local ss= {} # Abstract.      } → Together.
  local qs= [] # Concrete       }

  # Coupling invariant is established.

  def makeEmpty:
    # PRE If coupling invariant holds here,
    ss,qs= {},[]
    # POST then it holds here too.                              (8.11)

  def add(s):
    # PRE If coupling invariant holds here,
    ss,qs= ss∪{s},qs+[s]
    # POST then it holds here too.

  def isIn(s):
    # PRE Coupling invariant holds here, before.
    return s∈ss # And ensures the correct value is returned.
```

See Ex. 8.8.

In the code above (8.11), the coupling invariant must hold after any initialisation (`local`), before any read-only function (`isIn`), and before and after an updating procedure (`makeEmpty`,`add`).

That was the first step.

8.5.2 Use the coupling invariant

The second step in the translation procedure is to use the coupling invariant to swap the roles of the abstract- and concrete variables. In Prog. 8.11 just above the abstract variable `ss` is "real" and the concrete `qs` is auxiliary. We want to swap that around, to make `qs` become real and `ss` become auxiliary; and so we must remove `ss` from the `return` of `isIn` while preserving the meaning of what is returned. For that we introduce the loop from Prog. 8.1. Provided it terminates, it has no effect on `IsIn` at all:

```
class Set:
  ⋮
  def isIn(s):
    # PRE Coupling invariant holds here, before.
    b= False
    for q in qs:
      if q==s: b= True; break
    { b == (s∈qs) }     ⟵ Established by the loop.
    # Coupling invariant holds here, which gives us
    { b == (s∈ss) }     ⟸ this also.
    return s∈ss
    # We can now replace the  return s∈ss  by  return b .
```
(8.12)

The loop establishes the postcondition $\longleftarrow$ and the coupling invariant tells us that $\Longleftarrow$ holds too. And then the $\Longleftarrow$ allows us to replace the `return s∈ss` by `return b`, which eliminates the reference to `ss`, making it auxiliary.

See Ex. 8.9.

8.5.3 Remove abstract (auxiliary) variables

After our changes in Sec. 8.5.2 just above, we now have that `ss` is auxiliary and `qs` is "real", exactly the opposite of before. And because it's `ss` that is now auxiliary, we can remove it altogether; that gives

```
class Set:  # This is the actual implementation.
  local qs= []

  def makeEmpty: qs= []
  def add(s):    qs= qs+[s]

  def isIn(s):
    b= False
    for q in qs:
      if q==s: b= True; break
    return b    .
```
(8.13)

The question is now "How do we re-check Prog. 8.8 –that *uses* the abstract `Set`– with this new (concrete) version (8.13) that actually *implements* `Set`?"

The (surprising) answer is that *we don't have to (re-)check at all!* We checked it already, at Prog. 8.9 in its abstract form. That is, we checked our program using the abstract data-type, but when we run it we supply the concrete data-type — *and we can be sure that the check is still valid...*

> ...provided the abstract- and concrete data-types together respect the coupling invariant,
>
> *and you follow the rules illustrated here, and summarised in Chapter 10 below.*

The reason for splitting this process into two pieces –write the surrounding program in terms of the abstract data-type; implement the abstract data-type using concrete types– is that the surrounding program is more easily checked with the abstract data-type, because we are reasoning at a higher level, as in Prog. 8.9. But for the actual code, we need the concrete data-type, because the abstract code might not be supported by our programming language. See Ex. 8.10.

8.6 *Exercises*

Exercise 8.1 (p. 74) Introduce requirements, pre- and postconditions for Prog. 8.1, and then add a loop invariant so you can check it.

Exercise 8.2 (p. 75) Find the code for Prog. 8.3 directly (i.e. without using a sentinel), using the invariant

```
0<=n<=N and x∉A[:n]
```

.

Exercise 8.3 (p. 37) In Prog. 3.7, binary search, we used the assignment statement `p= (m+n)//2` to establish the assertion `m<=p<n` given that `m<n` held: that is See Ex. 8.2.

```
# PRE m<n
p= (m+n)//2
# POST m<=p<n
```

checked. But doesn't the simpler

```
# PRE m<n
p= m
# POST m<=p<n
```

also check? If so, then the resulting program as a whole also checks, i.e. it searches for `x` in `A`, just as binary search did. But what kind of search is it *really*?

Exercise 8.4 (p. 75) What is the invariant for Prog. 8.4? What is the variant? Check your program carefully with the invariant and variant you have chosen.

Hint: Be especially careful with the upper bound for `n` in the variant.

Exercise 8.5 (p. 76) Explain why

```
n==N and False    or    n!=N and x!=A[n]
```

simplifies to just `n!=N and x!=A[n]` .

Exercise 8.6 (p. 77) The postcondition of Program 8.6 is not the same as the postcondition of Prog. 8.3, which was our original goal. Indeed both of them are missing `0<=n<=N`, which really should be there. But they are *still* not the same.

Give an example that shows they are not, explain "methodologically" how it happened and what the consequences might be.

Exercise 8.7 (p. 77) Give the most concise formulation you can of the condition

```
x∉A if n==N else x==A[n] and x∉A[:n] .
```

Exercise 8.8 (p. 79) What does it mean to be an *auxiliary* variable? Why is `qs` an auxiliary variable in Prog. 8.11?

Exercise 8.9 (p. 80) To check Prog. 8.12, in particular the assertion `{ b == (s∈ss) }`, we need an invariant for the linear-searching `for`-loop in `isIn`. What is it?

What checking rule do we use (from App. B.2.2) to move from (⟵) to (⟸) ?

And why does it check?

Exercise 8.10 (p. 81) Often the the concrete data-type can be simplified, a sort of "tweaking" process, once the abstract variables are gone — and that is the case with `isIn` above at (8.13) above. What is the simplified code for `inIn`? Why is it all right to change it now?

Chapter 9

Case study in coupling invariants: Fibonacci numbers

9.1 Definition vs. implementation

The Fibonacci numbers are well known: they start with 1,1, and then carry on with 2,3,5,8,13... where each new one is the sum of the two just before. Their usual definition is that

$$\texttt{fib}\,(n{+}2) \;=\; \texttt{fib}\,(n{+}1) + \texttt{fib}\,(n)\,, \quad \text{with } \texttt{fib}\,(1) = \texttt{fib}\,(0) = 1 \quad , \tag{9.1}$$

and that leads directly to an easy (recursive) program to calculate them:

```
# -- Calculate the n-th Fibonacci number
# PRE n>=0
def fib(n):
  if n==0: return 1
  elsif n==1: return 1
  else: return fib(n-1)+fib(n-2)
# POST  fib(n) returns fib(n)    .
```
(9.2)

The above function is "recursive" because it can call itself, but it is still meaningful: the reason it is not a circular definition is that it does not *always* call itself. Eventually, it stops doing that, and starts to construct the answer: the check that it *does* stop, eventually, is done with a variant; and the check that it returns the right answer, when it stops, is done with an invariant. And it's an easy check.

See Ex. 9.1.

What makes Prog. 9.2 impractical though, as an implementation of (9.1), is that for `n` of any reasonable size it's simply too inefficient: to calculate $\texttt{fib}\,(0)$ and $\texttt{fib}\,(1)$ takes one call of `fib` (each), but to calculate $\texttt{fib}\,(2)$ takes three; and $\texttt{fib}\,(3)$ takes five. The number of calls is exponential in `n`.

See Ex. 9.2.

The more efficient program that is typically used to calculate Fibonacci numbers is instead the one given in the next section, and we will explain it in detail, even though it's well known, because it illustrates two of our invariant-finding techniques. And it takes time *linear* in n.

The aim of this chapter however (from Sec. 9.4 on) is to show how, using a coupling invariant, we can calculate `fib`(n) in *logarithmic* time. The final program is given at (9.10).

9.2 Linear-time Fibonacci

The usual program for calculating Fibonacci numbers is this one:

```
# -- Calculate the N-th Fibonacci number.
{PRE N>=0}
f,g= 1,0
for n in range(N): { Inv: f==fib(n) and g==fib(n-1) }
  f,g= f+g,f
{POST f==fib(N)}
```
(9.3)

where we note that in the multiple assignment `f,g= f+g,f` the *initial* values of `f` and `g` are used on the right hand side.

We have introduced the convenient fiction that `fib(-1)==0`. Checking –indeed designing– this program is an example of *iterate up* and *cascading invariants* from Secs. 3.5 and 3.3.1 — the first step, iterate up, is to propose the invariant `f==fib(n)`. But then to code the loop body, which must calculate the next value `fib(n+1)` of `f` from its current value `fib(n)`, you discover that you need `fib(n-1)` as well, and that is the second step: the "cascaded invariant" is `g==fib(n-1)`.

See Ex. 9.3.
See Ex. 9.4.

9.3 Introducing matrices, but still linear time

If we look at the loop body of Prog. 9.3 just above, that is `f,g= f+g,f`, we see that it is equivalently the matrix multiplication

$$\begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix} \quad ;$$

and so we can re-write Prog. 9.3 as shown in Fig. 9.1. It has become a simple exponential-calculating program (and does not need *cascading invariants* any more, because the `n` and `n-1` case are combined together in the column vector for `f` and `g`). Still Prog. 9.4, like Prog. 9.3, takes linear time.

But it *is* an exponential-calculating program — and our earlier Program 3.9 calculated `B**E` for scalars in the much more efficient *logarithmic* time, using a halving-and-squaring strategy. (It was the straightforward, but slower algorithm Prog. 3.8 that took time linear in `E`.) With the cleverer invariant `B**E==p*b**e` (though it was not an obvious one) the faster, logarithmic-time program was easy to write and check.

Using a coupling invariant, we can therefore transform the program Prog. 9.4 of Fig. 9.1 into a much faster program that calculates the $\texttt{N}^{th}$ Fibonacci number in logarithmic time. It uses matrices rather than integers, but is otherwise structurally the same as Prog. 3.9.

That is, the coupling invariant will allow us transform the logarithmic-time exponentiation program on numbers in to one that works the same way on matrices — and then we will then have a logarithmic-time Fibonacci calculator.

```
{PRE N>=0}
( f )=  ( 1 )
( g )   ( 0 )
for n in range(N): # INV Inv                                    (9.4)
    ( f )  =  ( 1  1 ) ( f )
    ( g )     ( 1  0 ) ( g )
{POST f==fib(N)}    ,
```

where Inv is

$$\begin{pmatrix} \texttt{f} \\ \texttt{g} \end{pmatrix} \;\texttt{==}\; \begin{pmatrix} \texttt{1} & \texttt{1} \\ \texttt{1} & \texttt{0} \end{pmatrix}^{\texttt{n}} \begin{pmatrix} \texttt{1} \\ \texttt{0} \end{pmatrix} \quad . \tag{9.5}$$

This linear-time Fibonacci program, using matrices, is a simple example of *iterate up* from Sec. 3.5.

Figure 9.1 Fibonacci as matrix exponential

9.4 Logarithmic-time Fibonacci

Below is Prog. 3.9, the logarithmic-time exponential calculator from Sec. 3.4.1 for simple numbers, that is scalars, repeated here for convenience. We have changed its `E` to an `N`:

```
# PRE 0<=N
p,b,n= 1,B,N
while n!=0: { Inv: B**N == p*b**n }
  if n%2==0: b,n= b*b,n//2                                      (9.6)
  else: p,n= p*b,n-1
{ B**N == p*b**n and n==0 }
# POST p==B**N    .
```

The checking we did for this program, in Sec. 3.4.1 earlier, relied mainly on associativity of multiplication for ordinary numbers, i.e. that $a(bc) = (ab)c$. In particular, checking the assignments in the loop body (with substitution, as usual), that they maintained the loop invariant, required things like

```
        p*b**n == p*(b*b)**(n//2)          when e is even
  and   p*b**n == p*b*b**(n-1)               in any case.
```

Since matrix multiplication is associative, just as it is for scalars, we should be able to use the same checking strategy. (Matrix multiplication is not commutative, however.)

We will replace the "abstract" matrix-typed variables `b` and `p` by "concrete" integer variables, using one concrete variable for each position in a matrix. The concrete variables are `b00`, `b01`, `b10`, `b11` and `p00`, `p01`, `p10` and `p11`, and the *coupling invariant*

will be

$$\texttt{b} == \begin{pmatrix} \texttt{b00} & \texttt{b01} \\ \texttt{b10} & \texttt{b11} \end{pmatrix}$$

$$\text{and} \quad \texttt{p} == \begin{pmatrix} \texttt{p00} & \texttt{p01} \\ \texttt{p10} & \texttt{p11} \end{pmatrix} \quad ,$$

so that the ordinary multiplications in Prog. 9.6 will be replaced by matrix multiplications. The result is this program:

```
# Variable N is given.
b00,b01,b10,b11= 1,1,1,0              # from (9.5) above
p00,p01,p10,p11= 1,0,0,1    # the matrix equivalent of 1
n= N

while n!=0:
  if n%2==0:
    b00,b01,b10,b11= b00*b00+b01*b10, \
                     b00*b01+b01*b11, \
                     b10*b00+b11*b10, \
                     b10*b01+b11*b11
    n= n//2
  else:
    p00,p01,p10,p11= p00*b00+p01*b10, \
                     p00*b01+p01*b11, \
                     p10*b00+p11*b10, \
                     p10*b01+p11*b11
    n= n-1
f= p00
```

(9.7)

where the final assignment `f= p00` implements the multiplication of the final matrix by the initial (column) vector `(1 0)`.

And indeed this program Prog. 9.7 calculates `fib(N)` in *logarithmic* time. But it can be simplified...

9.5 Remove auxiliary variables

In the final assignment to `f` in Prog. 9.7 just above, only one of the four `p`'s appears on the right-hand side. In fact, in the whole program the variables `p10` and `p11` appear only in assignments to themselves, which means they are (jointly) auxiliary. But `p01` is used in an assignment to `p00` — so it is not auxiliary, and must remain. If we remove the auxiliaries `p10` and `p11`, however, we get the new, smaller (but equivalent) program

```
b00,b01,b10,b11= 1,1,1,0              # from (9.5) above
p00,p01= 1,0
n= N

while n!=0:
  if n%2==0:
    b00,b01,b10,b11= b00*b00+b01*b10, \
                     b00*b01+b01*b11, \
                     b10*b00+b11*b10, \
                     b10*b01+b11*b11
    n= n//2
  else:
    p00,p01= p00*b00+p01*b10, p00*b01+p01*b11
    n= n-1
f= p00    .
```
(9.8)

But there is more. In fact two further variables can be removed, once we notice that the program satisfies the loop invariant

```
b00 == b10+b11    and    b01==b10    .
```
(9.9)

See Ex. 9.5.

So far, that invariant has not been used — we didn't even know it was there. But we will use it now, because it means we can remove `b00` and `b01`, replacing them with the right-hand sides just above. Doing this is another example of "tweaking" — we have already checked the program, but we are now making it slightly neater (and more efficient).

That gives the program

```
b10,b11,p00,p01,n= 1,0,1,0,N
while n!=0:
  if n%2==0:
    b10,b11= b10*(b10+2*b11), b10*b10+b11*b11
    n= n//2
  else:
    p00,p01= p00*b00+p01*b10, p00*b01+p01*b11
    n= n-1
f= p00    ,
```

which (still) calculates `fib(N)` in time logarithmic in `N`. We can finish off by renaming the variables to reduce the clutter, a further simplification: say `b10` to just `b`, and `b11` to `c`, and similarly for the `p`'s. Then we get the following compact (and intriguing) program, one that is *already checked*: it is

```
# -- Calculate fib(N) in logarithmic time
{PRE N>=0}
b,c,f,g,n= 1,0,1,0,N
while n!=0:                                                    (9.10)
  if n%2==0: b,c,n= b*(b+2*c),   b*b+c*c, n//2
  else:      f,g,n= f*b+f*c+b*g, f*b+g*c, n-1
{POST f==fib(N)}    ,
```

where the renaming of `p00` to `f` allowed us to delete the final assignment `f= p00`.[1]

[1] If you transliterate this to *C*, you will have to convert the multiple assignments into single assign-

9.6 Summary

What we saw in this chapter was the transformation of a whole "program" (actually function) from abstract- to concrete form. The abstract form was matrices, and the concrete form was ordinary numbers, i.e. scalars. (An added point of interest was that checking the abstract program, the matrices, had already been done when *it* was constructed in its scalar form, i.e. ordinary exponentiation of scalars. Only because of that check –its explicit reliance on associativity of multiplication, for example– could we just "carry it over" to matrices, without having to start again from scratch. It's the assertions and invariants that made that possible.)

Similarly, in Chp. 8.3 we transformed a whole (sequential search) program from abstract form, using `B`, to its concrete form using `A`. In both cases, transformation of the program/procedure/function was transforming "everything" — except that its inputs and outputs, or initial- and final state, were not transformed. Only the "internal workings" were affected, and they were not visible from the "outside", which is one reason that such transformations work.

In Chp. 8.2 however, the various procedures of the `Set` class *do* input and output their internal state, in the sense that each one passes its final state onto the next, where it is initial state; and it's essential for their correct working that those internals are not exposed to the outside world, that is to the surrounding program. That's achieved by the "encapsulation" that we will now discuss in more detail in the next chapter. The internal state that is *shared* by all those functions and procedures (e.g. `clear`, `add` and `isIn` for the `Set`'s) is *hidden*, and can be accessed only by the procedures and functions supplied. It's as if the state is put in a box, on exit from one of the procedures, and the box is locked: only entering another procedure from the same encapsulation can unlock it. *Users* of the encapsulation, the surrounding program, cannot get into the box.

9.7 *Exercises*

Exercise 9.1 (p. 83) How would you adapt our invariant/variant techniques so that you could check recursive procedures and functions?

Exercise 9.2 (p. 83) How many calls of `fib` in Prog. 9.2 does it take to calculate `fib(10)`?

Give a simple argument to show that `fib(n)` itself is exponential in `n`, and then express the number of calls needed to calculate `fib(n)` in terms of `fib` itself... and conclude that the number of calls needed is also exponential.

Hint: "Exponential" is this context means "is at least c^n for *some* $c > 1$." You don't have to give the actual c.

See Ex. 9.4.

Exercise 9.3 (p. 84) Fill in the detailed assertions for checking Prog. 9.3, but do it without using multiple assignment `f,g=···` (for example because you are programming in *C*, which does not have them). Instead introduce a "temporary" variable `t` to carry out the update as (three) separate assignments in the loop body.

ments successively.

Exercise 9.4 (p. 84) Repeat Ex. 9.3 but this time do not introduce an extra variable `t`. Instead, do something "tricky" with `f,g` to get the effect of the multiple assignment but using only (two) single assignments. (This is how you would have to do it in *C*.) Fill in the detailed assertions that you need. See Ex. 9.3.

Hint: This is an excellent small problem for practising how to check assignments. Do it! You might also need App. B.2.2.

Exercise 9.5 (p. 87) Show that (9.9) is an invariant of the loop in Prog. 9.7.

Hint: Do it using substitutions for the assignments, as explained in App. B.1.1. Although it's tempting to write down equations based on what you think the loop body is doing, it's a bad idea: it is then very easy to become confused between which variables are "before" and which are "after".

Much better is to use the substitution approach, which avoids all that, and gives the same (elementary) algebra anyway. Actually, it organises your work so that in the end you are doing the *right* elementary algebra.

Exercise 9.6 The variables of Prog. 9.10 are `b,c,p,q` and of course `n` and `N`. Use a coupling invariant

```
b' == b%1000000 and ··· and q' == q%1000000
```

to transform Prog. 9.10 into one that that calculates only the last 6 digits of `fib`(N).

Since keeping only the last 6 digits prevents the numbers from growing arbitrarily large, you can run this program for huge values of N.

What are the last 6 digits of `fib`(1000000)?

What are the last 6 digits of `fib`(1000000000)?

Chapter 10

Encapsulated data types: how exactly is it done?

10.1 Introduction

In earlier chapters, we introduced "coupling invariants" as a way of tying the abstract- and concrete versions of a data type together. Like loop invariants (we saw), the key idea is something that is "always true". Here we will look a bit behind the scenes, and see that coupling invariants are a special case of something more generally useful, the "data type" invariant. They describe something about a data-type representation that is... always true.

10.2 Data-type invariants [1]

Before summarising all the material above, we introduce "data-type invariants", which ensure that a data type is used consistently in a way that preserves an important property of it. It's the "consistency" that matters.

If you decide to represent a set as a sorted sequence, because you want membership tests to be efficient (e.g. using binary search) then you must make sure that when you add an element to the set, it is put in its proper place in the (sorted) sequence that represents it. How does that "need" (for being sorted) in `find` reach an entirely different function `add` and become a requirement (to maintain "is sorted") there?

In fact, using a sorted-sequence representation for sets could easily be a *second* stage of abstract–data-type translation: the first, a "quick and dirty" implementation of `ss` in terms of a sequence `qs` with possibly repeated elements, is only a prototype; then the second, implemented when time allows (or the need arises), is a more sophisticated sequence version with a *sorted* sequence `sqs`, with all the procedures altered systematically to be consistent with that. As we will see, the second translation does not have to refer to the original `ss` version: its work was done already when we introduced `qs`. The second translation works directly from `qs`, imposing the *additional* constraint that the sequence be kept in order.

All of that is, admittedly, trivial: but writing code that works is as much an exercise in care, and good organisation, as it is in brilliant insight. As we've already remarked, it's for that reason that when a data-type representation contributes in many places to a program, it's helpful to gather its use all in one place textually, that is within

a class: then it becomes an encapsulation. The advantage of doing that is is simple: it makes it easy to see whether the data type is being used consistently, because all references to it are in one place textually. You do not have to search through the source file to find them. And if the compiler enforces the encapsulation properly, you *know* that the data has been gathered in that one place — because you would get a compile-time error otherwise.

Not so trivial however, but still very important, is that when the data is gathered, i.e. encapsulated this way, it is possible to check that an abstract data-type has been correctly implemented by a concrete version of it (as sets were by sequences, above) again *without* searching through the rest of the program: it's all done in one place, inside the encapsulation. And if, later, you want to change the representation (abstractly still sets, but perhaps now implemented concretely as binary-search trees), that too is done in one place — the same place, inside the encapsulation.

With all that in mind, we now look at data-type invariants specifically, continuing with the `Set` example.

See Ex. 8.10.

In Program 8.13 we gave an implementation for `Set`: it had been specified, earlier, in Prog. 8.7, where we showed (Prog. 8.9) how in that earlier form it could have been used to help write a program that *used* a set, for its own purposes. (It was the discussion about "more than pseudocode".) The implementation we provided, in terms of sequences, was guaranteed not to invalidate any of the checks that might have been performed on the program using the abstract version. "For its own purposes" means that we don't know how the set was used, and –because we followed the abstract-to-concrete transformation that a coupling invariant enables– we don't care. That was the theme of Sec. 8.5.

But we might want to go even further: the implementation we have chosen uses sequences, which are possibly expensive to implement (because they are copied around): for efficiency, we might use arrays instead. Arrays are like sequences, but are allocated just once and do not move around afterwards; but their disadvantage is that they are not easily extended. Can we implement our `Set` datatype in Prog. 8.7 with a fixed-size array, instead of with sequences as we did in Program 8.13? How do we find out?

What we find out is that no, actually, we cannot implement Prog. 8.7 that way. Not as it currently stands. Whatever the size of the array we used, say some `N` no matter how big, if some surrounding program calls procedure `add(s)` for `N+1` times in a row with a different, i.e. a new `s` each time, the `N`-length array inside the encapsulation would have to store `N+1` values. And that is impossible.

Thus we will have to change the abstract `Set` specification slightly, and we introduce some new features as we do it. Here is the new version of the abstract data-type, the specification, written out all together in one place:

```
class Set(N):
  # INV |ss|<=N                    # Data-type invariant.
  local { N>=0 } ss= {}          # Data-type precondition.
  def makeEmpty:
    ss= {}
  def add(s):
    assert |ss|!=N # This must hold when add() is called.
    ss= ss∪{s}
  def isIn(s):
    return s∈ss
```
(10.1)

There are four new features. The FIRST is a parameter `N` for the whole `Set` class — it is the size of the set the class must be able to represent. The SECOND is a precondition `N>=0` for the whole class: it refers to the parameter `N`. Obviously it is pointless asking the set to be able to store at least a negative number of elements.

The THIRD new feature is the assertion `|ss|!=N` placed at the beginning of `add(s)`, which states that unless that condition holds whenever `add(s)` is called, the result of the call could be arbitrary. That is, if the calling procedure does not respect the assertion, then the `Set` data type, in particular its `add(s)` procedure, is allowed to fail.

See Ex. 10.5.
See Ex. 10.6.
See Ex. 10.7.

Such `assert` statements are common in programming languages, [2] and principally intended as an aid to debugging at runtime: when executed at runtime, they check the condition given and, if that condition evaluates to `False`, the program is halted at that point — usually with a helpful error message that the programmer attached to the `assert` statement (and, at the very least, an indication of which `assert` it was, e.g. a line number and a trace-back). They are especially useful for catching bugs that are caused by coding errors in one place, but would not actually crash the program until it had reached some other place, possibly far away.

But `assert` statements can also be used at *checking* time, i.e. before the program is ever run: when you *check* an assert statement, you are making sure that its condition can never *be* `False`. But you do that while writing the program, not (as above) while running it. And that is how we use it here. (We discuss assertions further in Chp. 12, and give the rule for checking them in App. B.6.1.)

By placing `assert |ss|!=N` at the beginning of `add(s)` we are forcing anyone who is checking a program that *uses* this class to make sure that `add(s)` is never called when `|ss|==N`, because otherwise *that* program would fail. And it's this new, slightly weaker specification of `Set()` that *can* after all be implemented with arrays, provided that the check for duplicates is done: if the encapsulation stores each element in the array only once, then the `Set`-*user's* checks will make sure that `add` is never called in a situation where it would cause the array to overfill. That is, the assertion is imposing an obligation on the `Set`-user, not on the `Set`-implementer.

The FOURTH new feature is a *data-type invariant*: it states a property that

(a) Must be established by the initialisation. (It is, in (10.1), given the precondition for the whole class.)

(b) Must be `True` at the end of any procedure. (It is, in (10.1), provided it. . .)

(c) (. . .) Can be assumed to hold at the beginning of any procedure. (It is, provided that only procedures in the class can assign to its variables. Whatever procedure of the class that was called immediately before the current one must by (b) have left the coupling invariant `True`.)

Here the data-type invariant is `|ss|<=N`.

The data-type invariant, if there is one, can be considered a checking obligation on the whole class: it is an implicit pre- and in the postcondition of every procedure — with the exception that it is not a precondition of the initialisation. The initialisation must *establish* the data-type invariant using only the precondition for the initialisation itself. And all three procedures `makeEmpty()`, `add(s)` and `isIn(s)` must be checked to see that they *maintain* the data-type invariant:

- The check for the initialisation is `{`PRE `N>=0} ss= {} {`POST `|ss|<=N}`. Its precondition is the precondition for the whole class.

[2] Python has them.

```
class Set(N):
  # INV Array "ar" contains no repeated values.
  # INV 0<=n<=N
  local { N>=0 } ar= [0]*N   # "ar" has constant size N;
  local n=  0               # but only ar[:n] is in use.
  local def find(s):                    # local procedure
    for i in range(n):
      if ar[i]==s: return i
    return n

  def makeEmpty: n= 0

  def add(s):
    i= find(s)
    if i==n: ar[n],n= s,n+1
  def isIn(s):
    return find(s)!=n
```

(10.2)

Figure 10.1 An array-based implementation of `Set`

- The check for `makeEmpty` is {PRE `|ss|<=N`} `ss= {}` {POST `|ss|<=N`}. Its precondition is the data-type invariant, ensuring that `N>=0`; that ensures the truth of the postcondition, since `|{}|==0`.
- The check for `isIn` is {PRE `|ss|<=N`} `return s∈ss` {POST `|ss|<=N`}. Here the check is trivial, because the procedure does not change `ss`.
- The check for `add` is

  {PRE `|ss|<=N and |ss|!=N`}
  `ss= ss∪{s}`
  {POST `|ss|<=N`} ,

  where the `|ss|!=N` has been included in the precondition because it is an obligation on the caller who is –by the very presence of that extra conjunct– warned that only `N` distinct elements may be added to this set.[3] The precondition can be simplified to {PRE `|ss|<N`} but it is written out to show where its two components come from: the first is the data-type invariant, and the second is the initial assertion in `add` itself.

See Ex. 10.7.

See Ex. 10.1.
See Ex. 10.2.

In Fig. 10.1 is a possible concrete implementation of the abstract `Set` above, now using arrays and taking advantage of the `assert` we have just been discussing:[4]

Most of that implementation is straightforward — except perhaps for the behaviour of `add()` when "too many" elements are added. What does `add` actually do in that case? And where has the `assert` gone?

(The "local" procedure `find` cannot be accessed from outside the encapsulation, and has no data-type-invariant restrictions.)

[3] Actually, what it literally says is that no element may be passed to `add` when the size of the set is already `N`. As we will see later, that condition can be slightly weakened.
[4] We continue to write in the Python style; but we will use the sequence `ar` only in "array like" ways.

If `ar` is full (that is, if `n==N`), and a new element is added, one that isn't the set `ss` already (the abstract view), then the array update `ar[n]= x` will actually be `ar[N]= x`, which is a run-time error (the concrete view): *index out-of-bounds*.

But that *is acceptable* in this situation (though inconvenient) because the assertion `|ss|!=N` was in the *abstract* specification of `add(s)`, at its beginning: it states that if "too many" elements are added, then the user cannot place any reliance on the outcome. And it's the abstract encapsulation that the user is looking at. If that user has checked the program, the assertion will make sure that the "too many elements" error can never be committed by the user's program.

In practice, however, an instance of "defensive programming", we would still probably rewrite procedure `add()` as

```
⋮
def add(s):
  i= find(s)
  if i==n<N: ar[n],n= s,n+1
  else: assert False, "Too many elements added."
⋮  ,
```
(10.3)

where the `assert condition, message` statement, executed at run-time, evaluates the condition and, if it is not `True`, halts the program at that point and prints the message.[5] Here, the `assert` is being used in both ways: at checking time, it should ensure that the user writes code that calls `add(s)` properly. But if the user does not check that code, and `add(s)` is –after all– called improperly, at runtime, then the `assert` will catch the error at that point, and prevent it from propagating further. (And it will place the blame for the error where it belongs: with the caller.)

Better late...

See Ex. 10.3.
See Ex. 10.4.

10.3 Rules for encapsulation: the basics summarised

Now that we have dealt with data-type invariants, we can bring the earlier material of this Part II together: the principal idea of encapsulation is that data –at least, non-trivial data– should be grouped textually together with the code that accesses and changes it. Aside from the fairly obvious general advantage of having things all in one place, there is the technical advantage that the representation of the data, or the trade-offs between speed of access and complexity of coding, can be done without affecting other parts of the program. It could be something as simple as deciding not to keep duplicates in a sequence that's representing a set (mentioned in Sec. 8.2, this would speed up deletions but slow down insertions), or something as complex as storing the set as a balanced binary tree (which would slow down insertions and deletions, but speed up searches).

As long as the rules below are followed, changes like that can be made behind the scenes, even when the software is already delivered and running in the field. But what are the rules, and what precisely do they guarantee?

The basic rules are given here; more advanced possibilities are discussed in Sec. 10.5.

Rules for encapsulated data types:

[5] This is Python's version of `assert`.

(a) The data should be gathered and declared in one place, and enclosed by an explicit indication of where the data-type definition begins and ends. [6]

This is the "encapsulation". The compiler should be able to check (with scope rules etc.) that this is done.

(b) Within the encapsulation there are a number of declarations:

(i) Declaration of all the "local" variables used to represent the datatype. These should not be accessible directly from outside the encapsulation, i.e. not directly from the surrounding program. (A program that tries to do that should generate a compile-time error.) The only way local variables can be read or written should via procedures and functions declared *within* the encapsulation. Sometimes these variables are called "attributes". We are calling them the *local* variables.

(ii) Declaration of procedures and functions that the surrounding program can use to access the local variables. The simplest form of these are sometimes called "get- and set methods"; but in general they may access/update the local variables in any way. Sometimes these procedures/functions are called "methods".

(iii) Declaration of *local* procedures and functions that can be used by the procedures and functions of (ii), but may not themselves be called directly from the surrounding program. (Again, a program that does that should generate a compile-time error.) An example was `find` in Fig. 10.1.

(iv) Declaration of formal parameters (if desired) of the encapsulation as a whole, that will be determined by the surrounding program when the data type is initialised.

(c) The encapsulation may have a data-type invariant (which we have labelled with INV). If several are given, they all apply. Each (non-`local`) procedure or function may assume the invariant holds when it is called, and must re-establish it (if necessary) before its return to the surrounding program; but it does not have to maintain it "in between" call and return (though it often will maintain it) — *except* that it must not call one of its own non-local procedures (either directly or indirectly) if the invariant is not true at that point of call. (Why not? See App. G.3.)

See Ex. G.1.

The data-type invariant usually refers only to the local variables of the encapsulation and possibly the formal parameters of the whole encapsulation. (See Sec. 10.5.)

(d) The procedures and functions may have assertions (written either `assert` − or `{` − `}`), and it is that it is the *caller's* obligation to satisfy them. (An example is the assertion `|ss|!=N` in the `add(s)` in Prog. 10.1 for the datatype `Set`.) They can be written anywhere within the procedure or function, but are usually put at its beginning.

To repeat: it is the *user* of the encapsulation that must make sure those assertions –the extra ones, not part of the data-type invariant– don't fail.

[6] We have been using Python's `class` declaration for that.

```
class Set(N):
  # INV |ss|<=N
  local { N>=0 } ss= {}
  def makeEmpty():
    ss= {}
  def add(s):
    assert s∉ss ⇒ |ss|!=N
    ss= ss∪{s}
  def isIn(s):
    return s∈ss
```
(10.4)

Figure 10.2 Abstract definition of a more robust class `Set`

(e) The local variables may have an initialisation, and an assertion that refers to the formal parameters of the class. Again, it is the class-user's responsibility to ensure that the assertion is `True` when the class is initialised. Given the assertion, the initialisation must establish the data-type invariant, if there is one.

Our `Set` data-type gives us two examples of encapsulation as described above. One we have called "abstract"; the other is "concrete". Figure 10.2 gives the abstract version (a slight variation on Prog. 10.1).

In Fig. 10.2 the local variable is (simply) the set `ss`, and it is within the explicit indication of the data-type scope: beginning immediately after the `class Set:` declaration, and ending when the outer indentation-level is resumed. The surrounding program cannot refer to `ss` directly. That is (a).

Its declaration is given immediately after the `local`. That is (i). Often[7] the type of the local variable would be given here.

The procedures and functions that access the local variable `ss` are `makeEmpty`, `add` and `isIn`. That is (ii). There are no local procedures or functions (iii).

The formal parameter of the size-limited `Set` class is `N`, the maximum size of the set. That is (iv).

The data-type invariant of this class is `|ss|<=N`, that the size of the set `ss` may not exceed `N`. That is (c). All the externally accessible functions `makeEmpty`, `add` and `isIn` may assume it holds when they are called.

Procedure `add(s)` has an assertion `s∉ss ⇒ |ss|!=N` which it is the obligation of the surrounding program to make `True` when it calls `add(s)`. (It's slightly weaker than the assertion in Prog. 10.1.) If it is not `True`, the surrounding program will not check. If for some reason the surrounding program was not checked, and `add(s)` was called anyway, then it might crash — and it is allowed to do so. (But recall defensive programming as in Prog. 10.3.) That is (d).

[7] In Python, this typing is sometimes implicit.

```
class Set(N):
  # INV Array ar[:n] contains no repeated values.
  # INV 0<=n<=N
  local { N>=0 } ar= [0]*N
  local n= 0
  local def find(s):                    # local procedure
    for i in range(n):
      if ar[i]==s: return i                                    (10.5)
    return n
  def makeEmpty: n= 0
  def add(s):  No explicit assumption.
    i= find(s)
    if i==n: ar[n],n= s,n+1
  def isIn(s):
    return find(s)!=n
```

Figure 10.3 Concrete version of `Set`, without `assert`

The initialisation `ss= {}` has an assertion `N>=0`, which it is the surrounding program's responsibility to make `True` when the class is initialised. That is (e).

In Fig. 10.3 –a second example– is the concrete version of the `Set` class, the array implementation Prog. 10.2 that we have seen before:

The concrete version in Fig. 10.3 has two encapsulation features that the abstract version did not need. It has a local procedure `find` , that is not accessible from the surrounding program; and the procedure `add(s)` has only an "implicit" assumption, which (we will see) turns out to be the same as the explicit assumption in the abstract version. In concrete terms, though, if it were made explicit it would be `n==N` $\Rightarrow$ `s∈ar`, because on the concrete level `|ss|` is `n` and `s∈ss` is `s∈ar[:n]`.

See Ex. 10.10.
See Ex. 10.11.
See Ex. 10.12.

10.4 Rules for encapsulation: their justification

In Sec. 10.3, the rules for encapsulation were simply stated: to motivate them, we now look at Fig. 10.4. If a program –*any* program– is successfully checked using the left-hand class, then replacing the left-hand class by the right-hand class must not invalidate that check. And it's precisely the rules for encapsulation that make that possible: by following them, we make sure that no user of the right-hand class can ever be sure that "There's not a real set `ss` in there."

And the process can be repeated: we could at a later stage replace the `Set` class a second time, putting in a bit more programming effort to keep the array in ascending order, in order to speed up membership tests. And *still* no surrounding program would be affected. [8]

See Ex. 10.13.

[8] We are concentrating only on the values returned by the procedures, not how fast they run or how much space they take — and that is called "functional correctness". The other issues are important too, but are not covered by our encapsulation rules here.

```
class Set(N):

  # INV |ss|<=N

  local { N>=0 } ss= {}

  def makeEmpty():  ss= {}

  def add(s):
    assert |ss|!=N
    ss= ss∪{s}

  def isIn(s):
    return s∈ss
```

```
class Set(N):
  # INV Array "ar[:n]" has no repeated values.
  # INV 0<=n<=N

  local { N>=0 } ar= [0]*N
  local n= 0

  def makeEmpty: n= 0

  def add(s):
    i= find(s)
    if i==n: ar[n],n= s,n+1

  def isIn(s):
    return find(s)!=n

  local def find(s):
    for i in range(n):
      if ar[i]==s: return i
    return n
```

Replacing the left-hand implementation of a fixed-size set by the right-hand one cannot invalidate the checking of any surrounding program.

(We are using the original, stronger `assert` on the left.)

Figure 10.4 Two versions of class `Set`

Indeed, this second step is the motivating example we used in Sec. 8.2 and returned to in Sec. 8.4. But we can go further: to replace sequence `qs` by a *sorted* sequence `sqs` with no repeating variables, we would use the coupling invariant

```
Sequences qs and sqs contain the same elements
(ignoring repeats in qs), but sqs is sorted into
ascending order and does not contain repeats.
```

The resulting new concrete encapsulation `Set` of sets would be

```
class Set:
  # INV Sequence sqs is sorted, without repeats.
  local sqs= []                    # Internal variable.
  local def binarySearch(x):
    # PRE sqs is sorted (from data-type invariant)
    ...                ← Fill this in later (Chp. 12.2).
    # POST sqs[:n]<x<=sqs[n:] and 0<=n<=len(sqs)
    return n
  def makeEmpty: sqs= []
  def add(s):
    n= binarySearch(s)
    if n<len(sqs) and s==sqs[n]: return
    else:  sqs= sqs[:n]+[s]+sqs[n:]
  def isIn(s): return binarySearch(s)!=len(sqs)    ,
```
(10.6)

but functionally speaking it would be indistinguishable from the original `qs` or `ss` version.

See Ex. 10.8.
See Ex. 10.9.
See Ex. 10.14.

The relation between the left- and the right-hand programs in Fig. 10.4 is called "refinement". Saying that once piece of code, whether standing alone or an encapsulated data-type, is a *refinement* of another, means that any check that succeeds on the first (less refined) one will succeed on the second (more refined) one too. If the two pieces of code have exactly the same effect, then that is equality, a trivial kind of refinement.[9] An example is that the sorted-sequence `sqs` class is equal to the original (unordered) sequence `sq` class which, in turn, is equal to the original set `ss` version of the `Set` class: the equality is in the sense that no surrounding program can tell the difference.

Adding a data-type invariant to a data type *always* results in a refinement of it. (Remember that being equal is a special case of refinement.)

But the array-based implementation `ar` of the class is *not equal to* the fixed-size set `ss` one in Fig. 10.4 — it is actually *better*, because the `ss` version can fail if `add` is called when `|ss|==N` but the `ar` version can fail only if `n==N` *and* `s` is not in `ar` already. A refinement that's not equality is called a *proper* refinement.

We can now give a more precise justification for the rules of data-type encapsulation, in terms of refinement: it is that

> If you follow the rules in Sec. 8.5 for replacing an abstract data-type definition by a concrete one, and the encapsulations themselves follow the rules given in Sec. 10.3, then the concrete version will be a refinement of the abstract version.
>
> The translation of an abstract- to a concrete data-type is called *data* refinement.

It is also true that imposing an extra data-type invariant, even if you don't change the local variables, produces a refinement of the original data type. That is effectively what we did between the versions of `qs` and `sqs` — the extra data-type invariant was that the sequence was sorted, and that was the only change. (Indeed, as a final step we could rename `sqs` back to `qs`.)

See Ex. 10.14.
See Ex. 10.16.

10.5 Rules for encapsulation: more advanced techniques

Some of the "basic" rules given in Sec. 10.3 can be relaxed, for more advanced situations. In this section we explain some of those situations.

> More advanced rules for encapsulations and their data-refinements
>
> (a) The data type may declare variables that are directly accessible from the surrounding program, as long as they cannot be *assigned to* by the surrounding program, nor their declarations changed during a data-refinement.
>
> (b) The surrounding program may refer to the formal parameters of the encapsulation; but it may not change them. (For example, the surrounding program may read the set-size `N` in Prog. 10.2; but it may not assign to it.)

[9] In the same way, equality (=) is a trivial form of (≤).

(c) The coupling invariant may refer to global variables, i.e. those in the surrounding program, provided that the surrounding program does not assign to those globals. (The best way to ensure this is to use a programming language that allows declarations of constants, say labelled `const`, so that an assignment to a `const` is a syntax error.)

The same applies to the special case of imposing a further data-type invariant, without changing the variables. It too may refer to global constants. See Ex. 10.15.

(d) The precondition(s) for the encapsulation (assertions before the initialising statements) can refer to variables outside the encapsulation, with no restrictions. They do not have to be constants.

(e) The new version may have more procedures/functions than the old; but they must use the same coupling invariant.

See App. C for a concise summary of all the above rules, in their most basic form.

10.6 *Exercises*

Exercise 10.1 (p. 94) In Prog. 10.2 the last statement of `find(s)` is `return n`. Would it not be clearer to have `return i` instead, to match the `return` in the loop body?

Hint: See (G.1) and Sec. B.4.5.

Exercise 10.2 (p. 94) Add a `remove(s)` procedure to the abstract `Set` type of Prog. 8.7. Then add a corresponding `remove(s)` procedure to the concrete `Set` type of Prog. 10.2. Make sure the data-type invariant is respected.

What should the specification of `remove(s)` do if `s` is not in `ss`? See Ex. 10.14.

Exercise 10.3 (p. 95) What does the concrete code (10.3) do when adding `s` to set `ss` in the case that `s`$\in$`ss` already?

Exercise 10.4 (p. 95) In Prog. 10.3, it would probably be clearer to put the `assert` just before the `if`, rather than in its `else` branch. What would that look like?

Why however is the "behaviour is unpredictable" still important?

Exercise 10.5 (p. 93) The assertion `|ss|!=N` at the beginning of the abstract `add(s)` is stronger than it needs to be. The reason it's there at all is that `add(s)` is obliged to preserve the data-type invariant `|ss|<=N`, and it does preserve it. But let's look at it more closely.

The place that `|ss|<=N` must hold is actually at the *end* of `add(s)` — so let's put the assertion there instead. That gives

```
def add(s):
  ss= ss∪{s}
  assert |ss|<=N
```
(10.7)

Then we use the program-equality

`x=` *expr*`; assert` *cond* = `assert` *cond'*`; x=` *expr*

where *cond'* is *cond* with `x` replaced by *expr*. It's an example of "program algebra" — an equality is asserted between programs rather than, as in normal algebra, between numbers.

Use the program equality above to rewrite Prog. 10.7 in the form

```
def add(s):
  assert ???
  ss= ss∪{s}    .
```
(10.8)

Do it by first using the program algebra to find out what `???` is, and only then simplifying it. What do you get? Does it make sense?

Exercise 10.6 (p. 93) Use the data-type invariant and your answer to Ex. 10.5 to explain why the assertion at the beginning of `add(s)` can be just `s∉ss ⇒ |ss|!=N` .

See Ex. 10.5.

Exercise 10.7 (p. 93) Is `add(s)` allowed to break the data-type invariant if it is called when `n==N`, that is when the `assert n!=N` at its beginning is not `True`?

Discuss both the abstract and concrete cases.

Hint: Think carefully about what "allowed to" means.

Exercise 10.8 (p. 100) Exercise 10.2 added an extra procedure `remove(s)` to the `set` data type. Is it all right in general to add new procedures? Is it all right to remove them?

See Ex. 10.9.

Exercise 10.9 (p. 102) Suppose the hidden procedure `binarySearch` in Prog. 10.6 were changed to visible. Does that break the rules?

No, it does not break the rules, because that procedure was not there before and so could just be considered to be a new procedure. Or...

Yes, it does break the rules, because it reveals the inner structure of the representation: searching for some `s` could reveal its position in `sqs`, which a surrounding program could then depend on. In fact you could search for all the possible `s`'s, one by one, and so deduce the exact value of `sqs`.

Which is correct? Either? Both?

Exercise 10.10 (p. 98) In the discussion of the concrete version (10.5) of the `Set` datatype, the explicit version of the assertion of `add(s)` was said to be `n==N ⇒ s∈ar`. Why is it that?

And why, in the very next sentence, is `s∈ar[:n]` written, instead of just `s∈ar`?

See Ex. 10.10.

Exercise 10.11 (p. 98) In spite of the fact that the concrete version (10.5) of the `Set` datatype does not need an explicit assert, it still might be a good idea to write the assertion `assert n==N ⇒ s∈ar` there explicitly .

See Ex. 10.12.

Why?

See Ex. 10.10.

Exercise 10.12 (p. 102) In spite of Ex. 10.11, it might be that there's no point in adding the assertion explicitly, as suggested, because no one would ever see it. Is that true?

See Ex. 10.11.

Exercise 10.13 (p. 98) Take any data type, and impose on it an additional data-type invariant INV `False`.

If the rules of this section have been followed by the data type, the new version is at least as good as the old: no surrounding program would be adversely affected. Yet the *implementor* of that new version now finds an implicit `assert False` at the beginning of every procedure, and so can place any code at all inside the procedure: infinite loops, array-accesses out of bounds, divisions by zero... or simply `skip`!

What has gone wrong with our method?

Exercise 10.14 (p. 100) Start with the array-based implementation of `Set`, as in Fig. 10.4, and refine it so that it keeps the array `ar` in ascending order.

Add a procedure `remove(s)` that removes `s` from the set, if it is there, and does nothing otherwise. See Ex. 10.2.

Exercise 10.15 (p. 101) What is the difference between a variable in the encapsulation that is accessible (by cannot be altered) from outside, as in (a), and a variable outside the encapsulation, as in (c), that can be referred to by the coupling invariant but cannot be altered by anything.

Exercise 10.16 (p. 100) The left- and right-hand sides of Fig. 10.4 look wholly unrelated: the left-hand one is in terms of so-called "sets", which we are assuming our programming language doesn't even implement; the right-hand one uses an "array", and doesn't have sets –whether implementable or not– anywhere. How can we say that they might be "equal"?

Chapter 11

Case study: the Mean Calculator

11.1 Specification of the calculator

This case-study in data-type encapsulation models a pocket calculator with a "find the average" function.

> First you `clear` it, then you `enter` a collection of numbers, and then you press "`mean`" — the calculation will display the mean (average) of the numbers you entered.
> After that, you can go further –enter more numbers– or clear and start again. You can also remove a number if you decide, along the way, that it should not have been included. (11.1)

The paragraph above is presented as a unit of its own, set off from the surrounding text, because it serves as the "user manual" for this calculator, and is written in English so that it can be understood by anyone who is likely to want to calculate averages. It's the *requirements*, in our earlier sense.

The program text below is similar, but is written in a Python-like programming language. We say "Python like" because it uses the "multi-set", or "bag" data type, which we assume our whatever[1] -like programming language does not have or, at least cannot compile directly to running code. (Multi-sets will be explained below.) But –again– it is not "pseudo-code"; again, it is more than that.

This code agrees with the requirements above –as it is supposed to– but it says quite a bit more, and it is more precise too. Think of requirements as being used to decide whether to buy a thing, and the more detailed specification as being a description of

[1] We could just as well give a C-like or a Java-like specification, and the following comments would also apply.

how it is to be used once you have bought it.

```
class MeanCalculator:                          # Using bags.
  local nb

  def clear: nb= ⟅⟆                             # Empty bag.
  def enter(n): nb= nb+⟅n⟆                  # Add single element.
  def del(n): nb= nb-⟅n⟆                 # Remove single element.

  def mean():                                  # Calculate mean.
    assert nb!=⟅⟆
    return ∑nb / |nb|
```
(11.2)

If our programming language for specifying has a precise meaning, as this one does, it can –and should– be used for writing specifications in the style above. Because it is simple (no complicated control-structures used) and has powerful and intuitive data types (like multi-sets), it can be clear and precise at the same time. As such, it serves a dual rule: it serves the customer, or user, who might have detailed questions about "What happens in this-or-that unusual situation?" And it serves the programmer, who ultimately must create code that behaves as the specification says it should.

For example, answering questions like "What happens when I press `mean` but haven't entered any numbers?" should not be part of the requirements (11.1). It's too detailed, a distraction, and is dealing with something that might never happen and which –if it does– will probably not cause a disaster. A similar question is "What happens if I delete a number that I did not enter?" The same remarks apply as just above.

See Ex. 11.4.

Both those questions are answered however by the program text at (11.2). In the first case, there is `assert nb!=⟅⟆` , a statement which means (as we have seen) "If the condition `nb!=⟅⟆` is not `True`, then the calculator (i.e. program) can behave arbitrarily after this point."

To answer the other question, we use the properties of multi-sets. A *multi-set*, or *bag* is like a set except that it can have a given element more than once. (Alternatively, it is like a sequece where you have forgotten the order.) Specific multi-sets, like `⟅n⟆` above, are written with "bag brackets" ⟅−⟆ rather than set brackets {−}, and for example ⟅1, 2, 3, 2⟆ has one 1, two 2's and one 3 and overall size 4, whereas the set {1, 2, 3, 2} has one each of 1,2 and 3 and has size 3. For bag `nb` and potential element `n`, we write `n∈nb` for the number of times `n` occurs in `nb`; if `n` is not in `nb` at all, then `n∈nb` is 0. The sum of two bags takes the sum of the memberships, so that `n∈(nb1+nb2)` is `(n∈nb1)+(n∈nb2)`, and the difference is similar except that membership cannot become negative:

```
      n∈(nb1-nb2)
==    if (n∈nb1)>=(n∈nb2) then (n∈nb1)-(n∈nb2) else 0    .
```

A notational convenience is to be able to write `n∈nb` where a Boolean is expected, in which case we understand it as shorthand for `(n∈nb)!=0`; and if we write `n∉nb` for a bag `nb` we always mean `(n∈nb)==0`.

With bags now understood, we can see how Prog. 11.2 answers the question "What happens if I delete a number `n` that I did not enter?" In that case, the bag `nb` is unchanged, because `n∈nb` cannot become negative.

11.2 Implementation of the calculator

It is not hard to implement the `MeanCalculator` using sequences (if the programming language provides them). But actual pocket calculators probably do not do that. Instead they would keep only the sum of the numbers entered, giving the following as a possible concrete implementation

```
class MeanCalculator:            # Using a running total.
  local s,c

  def clear: s,c= 0,0
  def enter(n): s,c= s+n,c+1
  def del(n): s,c= s-n,c-1            # Hmm... See below.

  def mean():
    assert c!=0
    return s/c    ,
```
(11.3)

which has been designed using the coupling invariant `s==∑nb and c==|nb|` . But as part of that design process, we would have found ourselves checking the fragment See Ex. 11.1.

```
{PRE s==∑nb and c==|nb|}
nb,s,c= nb-⟅n⟆,s-n,c-1
{POST s==∑nb and c==|nb|}    ,
```
(11.4)

which expresses that the coupling invariant is maintained; and it would *not* have checked. It checks only when `n∈nb`, because only then does the number of `n`'s in the abstract `nb` actually decrease; but the concrete `c` is decreased unconditionally. A gross failure of the check occurs when some `n` is removed from the empty bag. In the specification, the bag remains empty; but in our "implementation" the count `c` becomes −1. A natural question –faced with the check failure– is to try to fix the implementation, or the coupling invariant or both, so that the check succeeds. But in fact that's not possible: if the implementation is recording only the sum and count of the entered numbers, there is no way it can remove a specific number. Instead, we have discovered a bug in the *specification*, and that is what we must fix. See Ex. 11.4. See Ex. 12.2. See Ex. 12.3.

One way to repair the specification would be to remove the `del` button on the calculator altogether. Another would be to replace it with an `undo` button that would remove the most recently entered number (but could be used only once). A third way (which we will now do) is to alter the specification so that it requires the deleted number to be one that was entered: we'd add that as an assertion to the abstract version

```
def del(n):
  assert n∈nb
  nb= nb-⟅n⟆    ,
```
(11.5)

and when the coupling invariant and concrete variables are added we'd then have

```
def del(n):
  assert s==∑nb and c==|nb|
  assert n∈nb
  nb= nb-⟅n⟆
  s,c= s-n,c-1
  assume s==∑nb and c==|nb|    .
```
(11.6)

Removing the abstract variables would give us simply

See Ex. 11.3.
See Ex. 11.4.

```
def del(n):
  s,c= s-n,c-1
```
(11.7)

for the concrete implementation — which, remarkably, is just what we had before. But it checks now, because what it checks *against* has been changed.

See Ex. 11.2.

11.3 *Exercises*

Exercise 11.1 (p. 107) Is the `assert c!=0` necessary in `mean()` in Prog. 11.3?

Exercise 11.2 (p. 108) In the code of Prog. 11.7 it's clear that calling `del` "too often" could make the variable `c` negative. Yet `c` is supposed to be the number of elements in the bag, whose mean will eventually be asked for. How can a bag have a negative number of elements? What's going on? Is the code of Prog. 11.7 correct?

Exercise 11.3 (p. 108) Use the rules in App. C to remove the `assert`'s and the `assume` from Prog. 11.6.

Exercise 11.4 (p. 108) In the discussion of the failed check at (11.4), it was suggested that the `del` operation be replaced by an `undo`. Do that, and give both the bag- and the sum version of the `MeanCalculator` class.

Then explain what happens (is "allowed" to happen) if `undo` is (i) pressed twice in a row, or is (ii) pressed when the bag is empty.

Hint: You might have to add extra variable(s).

Then give the coupling invariant that you've used to connect your specification and implementation.

Hint: It might have to refer to the extra variable(s) you might have added.

Chapter 12

Summary of Part II

12.1 What is obvious

This Part II has been about organising the data in your programs. (Part I was about "organising" the programs themselves.) The main point is that it is good style to group conceptually related variables, and their procedures and functions, together in one place: it's as simple as that, and it is the part of the "why" of Object-Oriented Programming. (See Sec. G.3.) Further features of object-oriented programming are "inheritance" (where more specific types can be derived from others) and "persistence" (where internal data of a procedure can be retained from one invocation to the next). [1]

For example, when the programmer is thinking about a "set" of values that the algorithm relies on conceptually, the program code that implements that set within an array can then be considered easily all at once, without having to search through a file or leaf through a printout.

It would be silly for example in Prog. 10.5 to put the declaration of our set-implementing array `ar[0:N]` in one place, and yet to put the `n`, that indicates how many variables are currently in it, in some other place, and then maybe its initialisation `ar= [0]*N` somewhere else again. It's common sense *not* to do that.

Much of "common sense" however turns out to have sound engineering principles behind it. In this case, the grouping together of the related data and procedures enables systematic and *practical* techniques that simplify the understanding, checking and maintenance of large programs, by separating their components into portions that can be dealt with one at a time. In particular, one of the main themes of this part is the way in which the implementation of a sophisticated data type can be altered "behind the scenes" by taking advantage of an absolute guarantee that it won't invalidate an already-checked program.

We will return to that issue of "already checked" in Sec. 12.3 below. In between, however, we examine some interesting possibilities of abstract vs. concrete and specification vs. implementation that have suggested by the way we have been organising our data types.

[1] We don't those aspects of object orientation in this text.

12.2 Specifications and implementations: how are they connected?

See Ex. 10.14.

In Fig. 10.4 we showed two version of the `Set` data-type: the left-hand one was in terms of size-limited sets directly, using a set-valued variable `ss`; and the right-hand one used *two* variables: an array `ar` and a "How full is it?" variable `n`.

The left-hand "abstract" version in Fig. 10.4 made it clear what could be done with the set, and it could also answer more obscure issues like "What happens if you try to remove an element that is not there?" (once we add a `remove()` procedure as well, as in Ex. 10.2).

See Ex. 10.2.

The larger issue here is, however, that the abstract version allows a programmer to check a surrounding program, using that abstract version, *even though* the set-valued variable `ss` might not be supported directly in the programming language being used. That was the point of our example Prog. 8.9, and it is again an example of "more than (just) pseudocode". It's an astonishing advantage of abstraction, and –at least at first– it might not be obvious that it's even possible.

The right-hand "concrete" version in Fig. 10.4 shows how to write the `Set` data type in the programming language we actually have. That is, the left-hand side "specifies" and the right-hand side "implements". That they correspond is precisely what the Rules for Encapsulation in Sec. 10.3 achieve: the rules were *designed* to make that possible.

Inside the right-hand version, however, we find another opportunity for specification vs. implementation. The `find(s)` function is local, not accessible from outside the encapsulation, and we used it in two places to see whether element `s` was already in the set. (Remember, the set is `ss` on the left and `ar[:n]` on the right.) While coding that implementation, it was a bit of a nuisance to have to "take time out" to write the body of `find(s)`, the linear search with its `for`-loop, where really all we wanted to do was to say what `find(s)` was for, what it was supposed to do. We had to get even *more* concrete there, temporarily, deciding whether to search the array from its beginning (`range(n)`) or its end (`range(n-1,-1,-1)`) because sought-for elements are perhaps more likely to have been added recently, or even to implement binary search if we had decided to keep the array `ar` in ascending order.

Having to dive briefly into those details is distracting and, because it is energy-sapping, it is also error prone. Better would be to be able to say what `find(s)` must do without (at this stage) having to say how it does it. Chapter 10 has a nice example of that:[2] in Program 10.6, rather than write out the code of `binarySearch`, we wrote just "`...`" but at the same time gave a pre- and a postcondition that would be checked later, when the `...`'s were actually filled in. (They could even be supplied by someone else, say a more junior programmer: we would use the PRE and the POST to check the code of our class; the junior programmer would use the very same PRE and the POST to check the code replacing the `...`.)

[2] There is also a nice example in Sec. 17.10 of Part III to come, where a "count the black-coloured nodes" specification is introduced, which we don't bother to implement until we have checked that counting those black nodes actually does the job that the surrounding program requires it to do.

In Fig. 10.4,using the same technique would mean that we would write

```
local def find(s):
# PRE 0<=n<=N
...
# POST  if s∈ar[:n]                                    (12.1)
#         then 0<=i<n and ar[i]==s
#         else i==n
return i    ,
```

and we wouldn't write the actual `for`-loop until later. The postcondition is all we need to know in order to use function `find` correctly within `add` and `isIn`.

We will now –finally– introduce an abbreviation that makes all that easier. We have been writing

```
# PRE pre
prog                                                   (12.2)
# POST post
```

for some specific precondition *pre*, postcondition *post* and program *prog* and asserting "This program works." if whenever *prog* is started in a state satisfying *pre*, then it will finish in a state satisfying *post*. And we have been saying that "after the fact", when we have our program *prog* already and we want to check it.

But now we are seeing that it would also be useful to be able to write "We must find a program *prog* so that (12.2) works." in a similar way, when we know what *pre* and *post* are but (of course) not *prog*: for that, we have been writing

```
# PRE pre
...                                                    (12.3)
# POST post
```

with the dots indicating where our code must be put in order to check it.

The "streamlining" is to introduce an abbreviation for the whole of Prog. 12.3. By `x:[`*pre*`,`*post*`]` we mean "some code that, when replacing `...` just above, will make Prog. 12.3 work while changing only variable(s) `x` in the process." Thus `x` can be a list if we want: it is called the *frame*, and the whole thing is called a *specification* — it has a frame, a precondition and a postcondition. Although we don't know (and at this stage, don't care) what that code actually is, because we know what "Prog. 12.3 works" means, we *do* know that the code –whatever it turns out to be– will have these properties:

(a) If its initial state does not satisfy *pre*, then it can do anything at all. Given that the specification says clearly what is expected of the initial state, that seems fair.

(b) If the initial state *does* satisfy *pre*, then the code must make sure that its final state satisfies *post* while changing only variable(s) `x`. The specification says that clearly too.

As an extra convenience, however, we introduced

(c) The code can change only variables in the list `x` — and that allows us to introduce some useful further abbreviations:

(d) The specification `[`*pre*`,True]` (with no `x:`) that insists on the initial state's satisfying *pre*, but changes no variables at all, must do nothing if indeed *pre* holds; but if *pre* does not hold, it can crash. We write that `assert` *pre* . See Ex. 12.1.

(e) The specification `x:[`*post*`]`, where the precondition is missing, takes a default precondition of `True`. And furthermore...

(f) That specification `x:[`*post*`]`, allowing *any* initial state but establishing *post* while changing only `x`, we can write `assume x:`*post* .

If it is allowed to change *no* variables at all then we leave the `x:` off, writing just `assume` *post* .

Once we put (c)–(f) all together, we have a nice vocabulary for specifying what programs must do, and writing their specifications at the moment we realise we need them. We do that *before* we go on to find the code. And so our motivating example (12.1) can be replaced by the simple

```
local def find(s):
  i:[0<=n<=N,
     (0<=i<N and ar[i]==s) if s∈ar[:n] else i==n]
  return i
```

See Ex. 12.2.
See Ex. 12.4.
See Ex. 12.6.

It could also be written

```
local def find(s):
  assert 0<=n<=N
  assume i:(0<=i<N and ar[i]==s if s∈ar[:n] else i==n)
  return i
```

See Ex. 12.5.

Now that we have a concise way of writing specifications, we can even give rules for using them directly in checking programs (App. B.6.3).

12.3 What is *not* obvious

It might not be obvious why the rules of Sec. 10.3 work, i.e. *why* they guarantee that for all possible surrounding programs a concrete implementation data-type will be as good as the abstract data-type that it came from. How could we ever check *all* possible surrounding programs?

The coupling invariant is the key, and it works in a similar way to a loop invariant which, after all, guarantees that once the loop has finished the invariant will still be `True` — no matter how many iterations have occurred (even none).

For the loop invariant, you reason "If there are no iterations, the invariant is `True` after the loop because it was `True` before." And "If there is just one iteration, then it's `True` after the loop because we have checked that a single iteration preserves the invariant."

And if there are two iterations, then it's still `True` because that's just one more iteration than one iteration, and we already know (just above) that the invariant is still `True` after just one iteration. So it's still `True` now, as well." And from three on, it's the same argument again, each time referring to the one before. [3]

[3] This is a proof by induction, written out in words.

For the abstract- and concrete data-types, the reasoning is similar. You imagine *both* of them executing in parallel, as in Sec. 8.5.1 where we (temporarily) had both the abstract and the concrete variables together, and then you see that no matter how many times the operations of the data type are called, and with no matter what parameters, the abstract- and the concrete version will give the same answer (if they are functions) and will result in final states whose abstract- and concrete parts correspond according to the coupling invariant. In more detail:

- At the beginning, the abstract initialisation and the concrete initialisation are both executed (in parallel). If the abstract precondition is `True`, then the concrete precondition must be `True` too. And the rules say that executing the two of them together must establish the coupling invariant. So the coupling invariant is `True` after initialisation, provided the datatype's abstract precondition was `True`.

- When for the *very first* time an operation on the data type is called, the data-type invariant is `True` (from just above). And so (1) the two operations, the abstract and the concrete, return the same answer (if they are functions), and (2) the coupling invariant is re-established afterwards.

- When for the *second* time an operation on the data type is called, the data-type invariant is still `True` from the first time, because the scope rules do not allow the surrounding program to invalidate it between the two calls. [4] And so again the two operations, the abstract and the concrete, return the same answer (if functions) , and the coupling invariant is re-established afterwards, ready for the *third* operation. [5]

- And so it continues, in a way similar to the description we gave for the loop invariant.

Some of the rules for encapsulation are necessary to make the argument above work. The remainder –mainly to do with certain variables not being assigned to– are what make sure that the surrounding program cannot falsify the coupling invariant. (Recall our earlier comment in Sec. 9.6 about encapsulation being like "putting the internal state in a locked box".) If it could, we would not be able to string the constructs together as we did, with each operation being able to assume the coupling invariant at its start, because it was (re-)established by the operation before, at its end. Specifically

- We don't allow the surrounding program to assign to the encapsulation's local variables, because we have no way of checking that those assignments will preserve the coupling invariant. (Remember — we don't even know what the surrounding program is.)

- And we don't allow the coupling invariant to refer to non-constant variables in the surrounding program because, again, that program might change them, and so falsify the coupling invariant.

[4] That is, the surrounding program cannot change the encapsulated variariables because it cannot even refer to them.
[5] Again... This is a proof by induction, written out in words.

12.4 *Exercises*

Exercise 12.1 (p. 112) Can the specification `[`*pre*`,True]` change the program variables if *pre* is `False`, i.e. if the program is allowed to crash? There is no `x:` there...

Exercise 12.2 (p. 112) The code below is from `add(s)` in the right-hand encapsulation in Fig. 10.4, but replacing the code `i= find(s)` with its actual body, assuming that body has been written as a specification, as in

```
# Code of add(s), with find written as a specification.
# PRE (s∉ar[:n] ⇒ n!=N) and inv
i:[0<=n<=N,                                  ← specification of find
   if s∈ar[:n] then (0<=i<n and ar[i]==s) else i==n]
if i==n: ar[n],n= s,n+1                      ← implementation of add(s)
# POST s∈ar[:n] and inv    ,
```
(12.4)

where *inv* is the data-type invariant

```
# INV Array ar[:n] contains no repeated values.
# INV 0<=n<=N
```

of the right-hand encapsulation in Fig. 10.4 that we are starting from.

We have surrounded it with PRE and POST, so that we can check that `add(s)` really does add `s` to the set. Using only the specification of `find` (i.e. as given above), check that Prog. 12.4 works.

See Ex. 12.3.

Remember though that this is only an *exercise*, because `add` in its concrete version does not *have* to be checked: we know already that it works — because it has been constructed, using a coupling invariant, from the abstract version while following the rules of Sec. 12.3.

Exercise 12.3 (p. 114) In Ex. 12.2, the PRE for our proposed check contains the condition `s∉ar[:n] ⇒ n!=N`. But that happens to be precisely the condition required for termination of the concrete code for `add(s)` — if it does not hold, then `add(s)` will attempt to access `ar[N]`, which gives a subscript-out-of-range error.

See Ex. 12.2.

Doesn't it seem like cheating, therefore, to include that condition in the PRE? It's like assuming the very thing you need for the check to go through, which surely defeats the whole purpose of the check.

Where does the condition really come from, and why isn't it cheating to include it?

Exercise 12.4 (p. 112) Go all the way back to Prog. 3.11 (length of the longest run) and insert a specification where the comment says "Put a 'sort sequence `A`' program fragment here." How precise do we need to be?

Exercise 12.5 (p. 112) Explain informally why `x:[`*pre*`,`*post*`]` is the same as

```
assert pre
assume x:post    .
```

Exercise 12.6 (p. 112) What is the difference between writing `{` *cond* `}` and writing `assert` *cond*?

Part III

Concurrency — and how to check it

Chapter 13

What is "concurrency"?

13.1 Examples of concurrent programs

Concurrent programs (or systems) are *groups* of conventional programs ("conventional" as in "like those we have considered up to now") that execute *all at the same time* and *interact with each other as they run.* (We have so far been considering each program executing alone.) The kinds of interaction we will now see are where two or more programs share variables between them, and can read and update those variables independently of each other.

Here is perhaps the simplest possible example:

```
# Initially:
  x==0
# One (conventional) program:      call it Thread 1:
  x= x+1
# Another (conventional) program:  call it Thread 2:
  x= x*2
```

← *concurrent system*

Programs that run concurrently, also described as "running in parallel", are sometimes called "threads", which we abbreviate *Th*1 and *Th*2 for the two above. A typical way of indicating that two programs together are running in parallel is to write

$$Th1 \;||\; Th2 \quad , \tag{13.1}$$

with "||" suggesting "in parallel"; and the effect is to allow each of the two programs to execute as it usually would... except that at any moment the variables of one program might be altered "behind its back" by the other program.

In the case above, we can ask "What will the final value of `x` be if the concurrent system in Prog. 13.1 is run from an initial state `x==0`?" The answer depends on which of the two threads goes first: if *Th*1 (Thread 1) goes first, then `x` will be `(0+1)*2 == 2` afterwards; but if *Th*2 goes first, then `x` will be `(0*2)+1 == 1`.

This –obviously– can be quite tricky to deal with, especially if the programs are complicated even on their own. The "interleaving" style of computation allows the threads' individual statements to execute in any order that respects their individual coding, and there can be many, very many possibilities for that interleaving. For example if each of our threads *Th1*, *Th2* had *two* statements instead of just one, say `A` then `B` in the first and `C` then `D` in the second, then the possible execution

interleavings would be `ABCD`, `ACBD`, `ACDB`, `CABD`, `CADB`, `CDAB`, that is six possibilities in all. (If they were not run in parallel, there would be only two possibilities: first one then the other, or first the other then the one, that is `ABCD` or `CDAB`. But when there are six possibilities, checking the program would –in principle– require checking all six of those interleavings.)

See Ex. 16.9.

But before going on to show that checking concurrent programs is not (quite) as bad as the above makes it sound, we should at least make it clear why we need concurrent programs at all. In the next section Sec. 13.2 we give a simple reason for that: it is that most activities (in the world) go on concurrently, and although many do not interact with each other very much, in some cases there is intense interaction, as Sec. 13.2 will illustrate: thousands of bank *ATM*'s throughout a whole country are active at the same time, and they interact with each other through the bank accounts that they access. If they were kept separate, it would be as if every bank transaction had to be done at head office, where there was exactly one bank teller and a (very long) single queue.

At the other extreme, in Chp. 17 we give an example of concurrent activities within a single computer, deep in the operating system. Here the activities are not physically separated from each other but, for efficiency, they must both be active most of the time: it's not practical (as we explain there) to make one execute separately from the other.

So we must live with concurrent programs, and therefore we must check them. This part of our text shows how that is done, mainly in Chp. 14 and Chp. 15.

13.2 A minimal concurrent example: automatic teller machines

Suppose integer `b` is your bank balance, and there are two *ATM*'s (automated teller machines) from which you can make withdrawals. We can imagine that their code could be something like this, where we're using `b` to represent your bank balance, stored centrally, and `c1` and `c2` are variables local to each *ATM*:

```
# ATM1                              # ATM2
if c1<=b: # Check balance.          if c2<=b # Check balance.
  b= b-c1 # Cash withdrawn.           b= b-c2 # Cash withdrawn.     .
```
(13.2)

Each *ATM* checks that you have sufficient funds and, if you do, delivers your cash into `c1`,`c2` respectively. If you use one *ATM* and then the other (and you have at least `c1+c2` in your account), which you could call *serial* use, then afterwards you will have `b-c1-c2` in your account, and `c1+c2` in your pocket. If initially you have nothing in your pocket and `B` in your bank account, that is `c1==c2==0 and b=B`, then `B==b+c1+c2` is an invariant of this system, you and the bank considered together, and it is maintained no matter which *ATM* you visit first. A second invariant is `b>=0` that you cannot withdraw more than you have in your account.

See Ex. 13.1.

Suppose now that you and a friend visit the two *ATM*'s at the same time, which we could call *concurrent* use, and that each *ATM* communicates with the bank twice: first it checks your balance, and then it notifies the bank that it dispensed cash to you. Let's say that your account has $100, i.e. initially `B==b==100`, and both you and your friend ask to withdraw the whole $100 from your account — that is `c1==c2==100`. Each *ATM* asks the bank whether you have $100 in your account, and each receives the answer "Yes." Those are the first two actions.

The third and fourth actions are that both you and your friend withdraw the $100, and your balance is finally *minus* $100 — so that you are overdrawn. Although the first invariant `B==b+c1+c2` is maintained, the second invariant `b>=0` has been broken.

The simple example, and others like it, above motivates the first three of the several interesting issues that concurrency introduces: they are "atomicity", "locks" and (as we saw earlier) "threads".

We begin with *atomicity*. What went wrong in concurrent scenario above was that the statement-by-statement execution order of the activities of the two *ATM*'s was interleaved in an unexpected way:

```
if c1<=b:                          # ATM1 checks c1<=b.
                  if c2<=b:        # ATM2 checks c2<=b.
b= b-c1                        # Cash withdrawn at ATM1.
                  b= b-c2      # Cash withdrawn at ATM2.
```
(13.3)

If the visits had been serial, that is all of one before any of the other, then the order would instead have been say

```
if c1<=b:                          # ATM1 checks c1<=b
b= b-c1                        # Cash withdrawn at ATM1.
                  if c2<=b:        # ATM2 checks c2<=b
                  skip     # Cash not withdrawn at ATM2.
```

where we remember that `if`'s whose conditions are `False` can be thought of as executing `else: skip` . Each of those smaller pieces is called *atomic* because it is executed by the bank in one piece: it cannot be further subdivided.

Fixing the problem illustrated by Prog. 13.3 above brings us to the second issue introduced by the *ATM* example: the use of *locks*. Obviously what real banks must do, instead of Prog. 13.2, is this: [1]

```
# ATM1                             # ATM2
lock b # Lock the account.         lock b # Lock the account.
if c1<=b:                          if c2<=b
  b= b-c1                            b= b-c2
unlock b # Unlock the account.     unlock b # Unlock the account.
```
(13.4)

The effect is to convert the whole of the program text shown for *ATM1* into a single atomic action (and the same for *ATM2*) — thus *ATM2* cannot "interrupt *ATM1* in the middle" (or vice versa).

The third issue is *threads*. The two programs *ATM1* and *ATM2* above are called threads and, in general, a thread gives a guaranteed execution order for its *own* constituent atomic actions only. Thus the code of Thread *ATM1* guarantees that `if c1<=b:` will occur before `b= b-c1` (or the `else`); but it has nothing to say about `if c2<=b:` and `b= b-c2`. That's the concern of Thread *ATM2*.

[1] In Python there are function calls `acquire()` and `release()` corresponding to the (fictitious) commands `lock` and `unlock` that we're using here for these examples. There many different schemes for concurrency and locks: we are concentrating just on the essentials.

13.3 What simple locks guarantee

The `lock b` / `unlock b` combination introduced in Prog. 13.4 just above is guaranteed to execute so that there cannot be two `lock`'s in a row, i.e. without an `unlock` in between. Usually that means that the `lock`'s and `unlock`'s occur in alternation, i.e. `lock`, `unlock`, `lock`, `unlock`... although that can be disturbed by two `unlock`'s in a row. (If that happens, though, it could mean there is a programming error somewhere.)

The simple scheme above separates the behaviour of the locks themselves from the goal that using the locks is supposed to achieve: that's a conceptual advantage; but it means that the programmer does have to put the locks in the right place. We now look at how a programmer can check that, with simple locks like these.

With each variable that can be locked, say `b` (which is here an integer, but it doesn't matter what it is), we associate a special extra lock variable `$b` whose possible values are thread names (or abbreviations for them) plus a special value ⊥ that means "no thread". Command `lock b` executes `$b= T` where `T` is the name of the thread executing it, but only if `$b==`⊥ beforehand. (What `lock b` does in the other case, i.e. when `$b!=`⊥, we will see in a moment.) Command `unlock b` executes `$b=` ⊥ unconditionally.

With that, we can add assertions to Prog. 13.4 as follows, using abbreviations `1`,`2` for thread names *ATM1* and *ATM2* that are assigned to `$b`. That gives

```
# ATM1                                  # ATM2
lock b # Lock account b.                lock b # Lock account b.
{ $b==1 }                               { $b==2 }
if c1<=b:                               if c2<=b
  { $b==1 and c1<=b }                     { $b==2 and c2<=b }
  b= b-c1                                 b= b-c2
  { $b==1 and b>=0 }                      { $b==2 and b>=0 }
unlock b # Unlock account b.            unlock b # Unlock account b.
{ $b==⊥ }                               { $b==⊥ }
```
(13.5)

and we can now see that the invariant `b>=0` is maintained, because

{PRE `$b==1 and c1<=b`} `b= b-c1` {POST `$b==1 and b>=0`}

works (and similarly for *ATM2*) — except that we must explain why *ATM2* cannot falsify the precondition `{ $b==1 and c1<=b }` just above. Here are the reasons; we deal with `$b==1` and `c1<=b` separately.

See Ex. 15.1.
See Ex. 15.2.

For *ATM2* to falsify `$b==1` in *ATM1* it must execute an assignment to `$b`, that is either `$b= 2` from `lock b` in *ATM2*, or `$b=` ⊥ from `unlock b`. But `lock` can execute only when `$b==`⊥ (and, again, here `$b` is not ⊥ — instead `$b==1`); and `unlock` in *ATM2* has precondition `$b==2`, so it cannot execute either. So *ATM2* cannot falsify `$b==1` in *ATM1*.

For *ATM2* to falsify `c1<=b` in *ATM1*, it must execute an assignment to `b` or to `c1`, and only `b= b-c2` assigns to `b` in *ATM2* (and there is no assignment in *ATM2* to `c1`). Like `unlock` in *ATM2*, it has precondition `$b==2`, so it cannot execute. So *ATM2* cannot falsify `c1<=b` in *ATM1*.

That reasoning checks that the problems in Prog. 13.3 cannot occur anymore, if we use locks in the right way. But what do they actually do?

13.4 What simple locks actually do

Actual implementations of locking depend ultimately on a hardware instruction that reads *and* writes as one atomic action: a typical example is "compare and swap", abbreviated *CAS*, which assigns a new value to a variable only if its current value equals one that is given; if the current value does not equal the one given, no assignment occurs. It must read the current value and then, if it's equal to the given value, write the new value *without* being interrupted in between: it reads, compares, then (possibly) writes — all executed as a single atomic action on the hardware level.

Thus we could implement `lock b` in some thread `T` by using a tight loop

```
while $b!=T:
  "Compare $b with ⊥ and                ← CAS, atomic in hardware.
   assign T to $b if they are equal."
```
(13.6)

This is sometimes called a "spin lock" implementation, because the thread "spins" around the loop until it acquires the lock. The key observation is that even if some other thread `U` is trying to lock `b` at the same time, only one of `T`,`U` can succeed: if the code above sets `$b` to `T`, then `$b` cannot be changed to `U` by a *CAS* in Thread `U` because its comparison of `$b` with ⊥ in `U` will fail, and no assignment of `U` to `$b` will be done at that point.

See Ex. 13.2.
See Ex. 13.3.
See Ex. 13.4.

13.5 Critical sections and other techniques

Locks are typically used to provide "critical sections" within threads: a *critical section* is usually a small code fragment appearing within each of several threads that must never be simultaneously active with any similar fragment in one of the other threads. Critical sections are precisely what the code fragments `if c1<=b: b= b-c1` and `if c2<=b: b= b-c2` are in threads *ATM1* and *ATM2*. A group of critical sections is easily implemented with locks by associating a lock `l` (for some fresh `l`) with each group, and enclosing each individual critical section with `lock l ··· unlock l` .

Introducing a critical section group is similar to making the enclosed fragment atomic, but it is not quite the same: critical-section fragments from *different* groups (i.e. different "`l`" names) can be active simultaneously, and this is important for efficiency. If for example we simply made `if c1<=b: b= b-c1` atomic, then every *ATM* for the bank would have to take turns, even if accessing different accounts: only one withdrawal could be taking place at a time among *all* the bank's *ATM*'s. To avoid that, a separate lock is associated with each bank account.

Critical sections are therefore an example of a typical approach to concurrency problems, and a the above is a "pattern" of solving them with locks. Another example of a pattern is "synchronised methods" where data that requires serial access (one thread after another, not interleaved) is collected within a class, and access to it us via methods that act as critical sections for that a particular instance (object) of the class.

13.6 Concurrency and interference

The example above, with its final annotations at Prog. 13.5, shows how seriously we must take into account the extra work that can be involved in checking concurrent programs, even quite small ones. As well as the usual checks we have already learned in Part I, we now have an extra set of checks to do: that atomic program fragments in one thread do not "interfere" with assertions in another thread. That is precisely what we did at the end of Sec. 13.3. In more detail:

(a) To show that invariant `b>=0` was maintained by the assignment `b= b-c1` in *ATM1*, we needed to refer to the assignment's precondition `c1<=b`. This will be called *local* checking (or checking for *local* correctness), and is what we did in Part I.

(b) Even though that precondition was established (as a postcondition) by the conditional `if c1<=b` just before, it might be invalidated by execution of `b= b-c2` in *ATM2*. That would be an example of interference, i.e. Thread *ATM2* interfering with (an assertion in) *ATM1*. Thus, more precisely,

> *interference* occurs between threads when a *statement* in one thread invalidates (changes from `True` to `False`) an *assertion* in another thread.

Note in particular that interference is not merely one thread's changing a variable that another thread accesses: indeed that is normal, for without that the threads could not cooperate at all. To interfere, you must falsify an assertion.

Making sure that interference does not happen is *global* checking (or checking for *global* correctness), and is the main new feature of this section.

(c) To show that the interference cannot happen, we relied on the fact that all the potentially interfering statements in *ATM2* could execute only when `$b==2`, and that as well as `c1<=b` we had `$b==1` (i.e. that `$b` was *not* 2) at the point we were checking for interference.

In the next chapter we draw all that together, discussing how our ordinary checks and our checks for non-interference fit together. The method is called "Owicki-Gries", named after its inventors.

See Ex. 13.5.

13.7 *Exercises*

Exercise 13.1 (p. 120) Alter the code of (13.2) so that it models the *total* cash you have, no matter how many times you visit the *ATM*'s. What is the new invariant?

Exercise 13.2 (p. 123) The spin lock in Prog. 13.6 is inefficient because in a multi-programming environment (i.e. a single core), where a single *CPU* is shared between threads, the time the lock spends "spinning" is time that could have been used by other threads for something more important. Even in a multi-core environment, the spinning is still using (extra) electricity and possible contributing to bus contention. What is usually done to prevent that?

Exercise 13.3 (p. 123) We have learned that infinite loops are to be avoided, and that variants are used to show that a loop terminates. Yet the loop in Prog. 13.6 appears not to terminate, and has no evident variant. Does it terminate? Does it have a variant?

Exercise 13.4 (p. 123) Check that

```
{PRE $b==1 and c1<=b}  b= b-c1  {POST $b==1 and b>=0}
```

works.

Exercise 13.5 (p. 124) In Prog. 13.5 the extra variable `$b` had three possible values: $\bot$, 1 and 2. But the only tests of that variable in the code were whether it was $\bot$ or not: there was no test "Is it 1?" or "Is it 2?" So we could merge those two values into a single one, having now $\bot$ for "unlocked" and $\top$ for "'locked". The program would still work.

What would be a *disadvantage* of merging 1 and 2 into a single value $\top$? (See also Sec. 15.2 to come.)

Chapter 14

The Owicki–Gries method

14.1 Introduction

In Sec. 13.1 we saw just how difficult it is to check concurrent programs that are sharing variables: the reason was the very large number of "interleavings" in which their statements could execute. And because you cannot predict which interleaving will happen –it could be different on each run– the techniques of Part I would have to be used to check every one of them.

See Ex. 16.9.

Luckily, the techniques of Part I *can* be carried over into the concurrent setting, once we concentrate not on "what happens" when the program executes, but rather on *what is true* as it executes. That is the difference we have been emphasising so much, between thinking "What does this statement do?" and "What does this statement make true?", between "What did the previous iteration of this loop do?" and "What does *every* iteration of this loop make true?"

The key is to examine how the activities of other thread affect the *checking* of this thread, rather than the execution of it. In this chapter we will see how that is done. The main innovation is explained beginning in Sec. 14.5.

14.2 Controlled interleaving

Assume we have a collection of threads *Th1*, *Th2*... *ThN*, like the *ATM*'s we looked at above (but which had only the two threads *ATM1* and *ATM2*). All those programs will run concurrently, which means that their atomic program fragments will be interleaved arbitrarily *except that* within each thread individually, the fragments execute in the order determined by the code of the thread itself.

Further, each of the threads has an assertion

(a) At every point within the thread where program control can pass from one atomic fragment to the next, [1]

(b) At the thread's very beginning and

(c) At the thread's very end;

and before the whole collection of threads is a single global precondition; and after the whole collection of threads is a single global postcondition.

[1] For us, that is usually *between* lines of code.

As we saw in Part I, each assertion mentioned in (a) is a postcondition for the fragment immediately before, and a precondition for the one immediately after.

And finally, there is a single (global) invariant that applies to the whole collection of threads together. There's an example of all this in Fig. 15.1 of Sec. 15.2 ahead. (Have a look.)

How exactly do we check a system like this? We will discuss that in stages, in the next three sections 14.3–14.5.

14.3 Checking *local correctness*: what's old

For "local correctness" we check each thread separately, entirely on its own — and, while this is being done, we have only "local" concerns. It is exactly what did in Part I for sequential programs (i.e. not concurrent), and that is because an ordinary (sequential) program is a special case of a concurrent system: it is just that it has only one thread.

Here are the three steps:

(a) We check every program fragment

$$\{\text{PRE } pre\} \; prog \; \{\text{POST } post\} \quad ,$$

where we recall that the PRE and the POST are there just to remind us of the role that those assertions play. In the actual thread code we would possibly find simply { *pre* } *prog* { *post* } . This is called "local correctness", abbreviated *LC*, and it is done just as we did in Part I.

Assertions *within* a line are checked for local correctness too.

In particular, we do not need to consider (yet) the "other" threads that might interfere.

(b) We also must show that the global precondition (of the whole system) implies the very first (local) precondition of each thread.

(c) Finally, we must show that the conjunction of all the local postconditions, i.e. all of them taken together, implies the global postcondition.

If we wish, we can assume as many (or as few) of the global invariant(s) while doing those checks: i.e. for (a) we could check

$$\{\text{PRE } pre \text{ and } gInv1 \text{ and } gInv2 \cdots\} \; prog \; \{\text{POST } post\}$$

which, because it has a stronger precondition, might be slightly easier to do.

14.4 Checking global *invariants*

To check that the global invariants really *are* invariant, however, we make sure that each one is established initially, and that it is preserved by every atomic fragment in the whole system, no matter which thread that fragment is in. That is, for this part we ignore the thread structure altogether and simply consider the statements separately.

In more detail: we check that the global precondition implies each global invariant

$$\begin{array}{lll} gPre & \Rrightarrow & gInv1 \\ gPre & \Rrightarrow & gInv2 \\ & \vdots & , \end{array}$$

and we can even use one invariant to help with another, as in

$$\begin{array}{lll} gPre & \Rrightarrow & gInv1 \\ gPre \texttt{ and } gInv1 & \Rrightarrow & gInv2 \end{array} \tag{14.1}$$

provided we don't do so circularly. For example

$$\begin{array}{lll} gPre \texttt{ and } gInv1 & \Rrightarrow & gInv2 \\ gPre \texttt{ and } gInv2 & \Rrightarrow & gInv1 \end{array} \tag{14.2}$$

is obviously not allowed.

Then for every atomic fragment *prog* in the system, whichever thread it might be in, we check

$$\texttt{\{PRE } pre \texttt{ and } gInv\texttt{\}} \; prog \; \texttt{\{POST } gInv\texttt{\}} \quad .$$

Just as in (14.1), if there are several global invariants then we can use some of them to help check others — and here we can do that even if it looks circular. For example

$$\begin{array}{ll} & \texttt{\{PRE } pre \texttt{ and } gInv1\texttt{\}} \; prog \; \texttt{\{POST } gInv2\texttt{\}} \\ \text{and} & \texttt{\{PRE } pre \texttt{ and } gInv2\texttt{\}} \; prog \; \texttt{\{POST } gInv1\texttt{\}} \end{array} \tag{14.3}$$

is allowed.

See Ex. 14.2.
See Ex. 14.1.

14.5 Checking *global correctness*: what's new

The idea of "global correctness" is the principal innovation of the Owicki-Gries method; it is that –as mentioned in Sec. 14.1– we look at how the *checking* of a thread is interfered with, or not, by the activities of other threads.

Global correctness is where it is checked that the activity of those "other threads" does not interfere "too much" in the local reasoning we have already checked (i.e. in Secs. 14.3 and 14.4). For every assertion `{` *assn* `}` (pre- or postcondition), wherever it might be (but not the global precondition, nor the global postcondition nor the global invariants), global correctness checks

$$\texttt{\{PRE } assn \texttt{ and } pre\texttt{\}} \; prog \; \texttt{\{POST } assn\texttt{\}} \quad , \tag{14.4}$$

where `{PRE` *pre*`}` *prog* `{POST ···}` is any program fragment in any other thread. (We do not care about *prog*'s postcondition here.) Obviously we need only consider *prog*'s that assign to variables that actually in the *assn*. Also, we can appeal to global invariant(s) by checking the easier

$$\texttt{\{PRE } assn \texttt{ and } pre \texttt{ and } gInv\texttt{\}} \; prog \; \texttt{\{POST } assn\texttt{\}} \quad ,$$

This is what is called *global correctness*, abbreviated *GC*.

And that –amazingly– is all that Owicki-Gries entails. In the next chapter, we will see examples of how it all works.

See Ex. 14.3.
See Ex. 14.4.
See Ex. 14.5.

14.6 *Exercises*

Exercise 14.1 (p. 129) The situation in (14.3) certainly *seems* circular: it looks like we are using *gInv1* to prove *gInv2* and similarly *gInv2* to prove *gInv1* — as we disallowed in (14.3). But here it *is* allowed. Why?

See Ex. 14.5.

Exercise 14.2 (p. 129) We will follow the convention that statements written all on one line are atomic, i.e. that *stmt1*`;` *stmt2* cannot be "interrupted at the semicolon". Suppose however that there is an assertion there, for example

stmt1`; {` *assn* `}` *stmt2* .

Does *assn* have to be checked for local correctness, i.e. that it is established by *stmt1*? Does it have to be checked for global correctness, i.e. that it cannot be falsified by a statement in another thread? Do global invariants have to be checked against *stmt1* on its own (and similarly *stmt2* on its own)?

Exercise 14.3 (p. 129) Why do we *not* have to check (14.4) when *assn* is the global pre- or postcondition?

Exercise 14.4 (p. 129) Why do we not have to check (14.4) when *assn* is a global invariant?

Exercise 14.5 (p. 129) In Sec. 14.5 we allow the precondition *pre* from another thread to be used when we are checking an assertion *assn* in this thread, which is of course the precondition of the fragment following it. But we will also check that that following fragment in this thread, whose precondition is *assn*, does not interfere with the assertion *assn* from the other thread.

See Ex. 14.1.

Why is that not circular reasoning?

Chapter 15

Critical sections with Owicki-Gries

15.1 Introduction

We discussed critical sections in Sec. 13.5, and introduced some ad-hoc notation (using variables `$b`) for them. In this chapter, we show how the Owicki-Gries method makes it possible to check that locks are doing what they are supposed to do.

15.2 Using a single Boolean

Consider two threads, each in a never-ending loop, as in Fig. 15.1. *"Other business"* is arbitrary code; *"critical section"* must eventually terminate; and neither of those accesses the variable `c` (nor any other variable that we might later introduce to control access to the critical section). *Mutual exclusion* is what we say we have achieved if the critical sections can never be active simultaneously.

The system in Fig. 15.1 uses an abstract version of the "compare and swap" that we discussed above in Sec. 13.4. Here we have written

```
await not c: c= True
```
(15.1)

to mean "Wait here until `c` is `False`, and then non-interruptably set `c` to `True`." The variable `c` means intuitively to us that "Some thread is in its critical section." and so the `await` statement prevents a thread from entering the critical section when another thread is already there, just as *CAS* did in Secs. 13.4 and 13.5. But the code in Fig. 15.1 does not (yet) contain enough information for us to show that it achieves mutual exclusion.

```
# Th1
"other business 1"
await not c: c= True
  "Critical section 1"
c= False
```

```
# Th2
"other business 2"
await not c: c= True
  "Critical section 2"
c= False
```

Figure 15.1 Mutual exclusion with `await` and single Boolean

```
                                        c= False      ← overall precondition

# Th1 repeats this forever.             # Th2 repeats this forever.

"other business 1"                      "other business 2"
await not c: c,t= True,1                await not c: c,t= True,2
{ c and t==1 }                          { c and t==2 }
  "Critical section 1"                    "Critical section 2"
{ t==1 }                                { t==2 }
c= False                                c= False
```

Because we cannot have `t==1` and `t==2` true at the same time, mutual exclusion is achieved — provided we successfully check local- and global correctness of the assertions.

The variable `t` is auxiliary, and therefore does not appear in the actual running program.

Figure 15.2 Mutual exclusion with `await` and auxiliary variable `t`

The general `await` causes execution to pause (so that some other thread is scheduled instead), unless (or until) the condition holds. The condition is evaluated atomically and –if it is true– execution proceeds to the `await` body without any potential interruption from other threads and, furthermore, the entire body of the `await` is executed without interruption as well. On the other hand — if execution *was* paused, then when the thread is rescheduled the condition is evaluated again (and so on...)

By analogy with conditional statements (as in Sec. 5.4 but without `else`), to check the `await`

{PRE *pre*} `await` *cond*: *awaitBody* {POST *post*}

we check only

{PRE *pre* `and` *cond*} *awaitBody* {POST *post*} , (15.2)

where –crucially– there is no `else` branch, and no implication to check. That's the difference between `if` and `await`: if the `if`-condition is `False`, then `else`-branch (default `skip`) is taken; but if the `await`-condition is `False`, the thread simply waits until it becomes `True`, which (of course) must be done by some *other* thread. An `await` never "takes `else`" — because it doesn't have one.

See Ex. 15.1.
See Ex. 15.2.

In Fig. 15.2 we've extended Fig. 15.1 by adding an auxiliary variable `t` (for "thread") and, with that, we can use Owicki-Gries to show that mutual exclusion is achieved. The reason `t` is auxiliary here is that it is never read and never written at runtime: it participates only in the assertions, i.e. while we are checking. But that is enough to check mutual exclusion at runtime.

Here's how it works. For local correctness we use the "how to check `await`" at (15.2) just above: in *Th1* to check

{PRE `True`} `await not c: c,t= True,1` {POST `c and t= 1`}

we check only

{PRE `True and not c`} `c,t= True,1` {POST `c and t==1`} ,

which is obvious. (In fact it doesn't need the precondition.) And checking *Th2* is the same. (What's the precondition for, then? See below.)

For global correctness we check two things: the first is the surprising

```
{ c and t=1 }  await not c: c,t= True,2  { c and t=1 }
```
(15.3)

where the statement `t= 2` occurs in the `await` body, but the postcondition contains `t==1`. From (15.2) however, checking requires

```
{ c and t=1 and not c }  c,t= True,2  { c and t=1 }    .
```

And that check *succeeds* because the precondition `c and t=1 and not c` is identically `False`, meaning that the `await` cannot execute yet: it must... wait. (And there is where we used the `await`'s precondition: recall above.)

The second is

```
{ c and t=1 and t=2 }  c= False  { c and t=1 }    ,
```

which again has precondition `False`, and so again checks successfully. The above Prog. 15.3 probably seems quite mysterious: it seems to be saying that we must show that `··· t= 2 ···` establishes `t==1`. But actually it does *not* say that. It says

> If `c and t==1` holds
> and `await not c: c,t= True,2` executes
> then `c and t==1` will still be `True`.
(15.4)

The key to understanding that is to realise that when `c and t==1` holds, the `await not c: c,t= True,2` *cannot* execute. And so (15.4) is vacuously true: its antecedent (the "If... executes" part) is `False`.

And now we know we have mutual exclusion because if both *Th1* and *Th2* were in their critical sections at the same time, the assertions show us that we would have `t==1 and t==2` , which cannot happen.

There is a clear conceptual connection between this auxiliary `t` and the lock values 1,2 assigned to the variable `$b` used in Prog. 13.5. But the difference is that those values are used in the actual running code of Prog. 13.5, whereas neither `t` nor its possible values will appear in the actual running code of Fig. 15.2. (See Ex. 13.5, which suggests that 1,2 might be replaced by a single value $\top$ in Prog. 13.5.)

See Ex. 13.5.
See Ex. 15.3.

15.3 Deadlock and starvation

It seems obvious that establishing critical sections must rely ultimately on hardware locks, at least at some level: as we saw in Sec. 13.4, typically that is a instruction that both reads and writes and is guaranteed to be atomic. We mentioned compare and swap as one example, where the "critical section" is between the comparison with the existing value and its (possible) replacement by a new one.

Similarly an `await` *cond*: *awaitBody* can be implemented by (spin) locks in the style of Prog. 13.6 by

```
while True:
  lock c
  "Set c to cond."      # This might be several statements.
  if c: break
  unlock c
awaitBody
unlock
```
(15.5)

where locking `c` must guarantee that the evaluation of *cond* is atomic, no matter how complicated it might be, and that the execution of *awaitBody* begins while *cond* is true and is carried out atomically.

But it is actually not true that critical sections must rely ultimately on hardware locks, at least not in the obvious way. The following code implements a critical section without using locks at all. The `await`'s have no bodies, and their conditions are single Boolean variables — there's no need to lock anything: [1]

```
# Th1 repeats this forever.          # Th2 repeats this forever.

"other business 1"                   "other business 2"
c1= True                             c2= True
await not c2:                        await not c1:
{ c1 and not c2 }                    { c2 and not c1 }
  "Critical section 1"                 "Critical section 2"
{ c1 and not c2 }                    { c2 and not c1 }
c1= False                            c2= False
```
(15.6)

Indeed `await not c2` is easily implemented by `while c2: skip`.

There is nevertheless a problem with Prog. 15.6 — if both threads set their Booleans to `True`, and then both `await`, then they will wait forever. This is called *deadlock* — the system doesn't crash; it just... stops. If however the `await` is implemented as suggested just above, with a spin lock, then instead the whole system might loop forever, checking Booleans that will never change: that is called *livelock*. We must figure out how to check that neither deadlock nor livelock occurs.

That the critical sections are indeed mutually exclusive is called *safety*, meaning that the dangerous situation, in which both threads are in their critical section at the same time, cannot occur. But the system (15.6) as a whole does not satisfy *liveness* — that eventually something useful will be done.

The possibility of deadlock is enough on its own to make Prog. 15.6 unsatisfactory; and so perhaps we do need hardware locks at some level. And yet the simple implementation of critical sections in Fig. 15.1, where implicitly hardware locks are used to implement `await not c: c= True`, has its own problems, though they are not so obvious.

[1] Actually, it is not *quite* true that no locking is ultimately required: at a very low level, it is. Here it would be at the hardware level where a single memory location (e.g. containing `c1` is not read-from and written-to at the same time. The bakery algorithm, however...

Suppose for example that the two threads were interleaved in this pattern:

```
      c==False                                  # initially
Th1                                 Th2

OB1                                 OB2         # other businesses
await not c: c= True                                # successful
                                    await not c     # unsuccessful
  CS1                                         # Critical section 1
c= False                                                              (15.7)
OB1
await not c: c= True                                # successful
                                    await not c     # unsuccessful
  CS1                                         # Critical section 1
c= False
```

⋮

This system is safe (as we already know), but it is not live — it is possible for one of the threads to "starve" the other: it is possible for *Th1* to execute *CS1* repeatedly, but *Th2* never gets to execute *CS2* even though it "wants" to (is at its `await`). This is essentially a problem of "fairness", where the interleaving is unfair in the sense that *Th2* is allowed to execute its `await` only when the `await` condition –at that particular moment– is `False`. It's unfair in the same sense as it would be unfair to make an offer repeatedly to someone, but only when you know they cannot accept it at that moment.

See Ex. 16.2.

In the next chapter we will see how to avoid all those problems — achieving all three properties of safety, absence of deadlock and absence of starvation, at least for two threads.

15.4 *Exercises*

Exercise 15.1 (p. 132) If the programs of Fig. 15.2 and (13.5) were executed in lock step, there would be a correspondence between the values of `c`,`t` in the first and `$b` in the second. What is the strongest condition (over all three variables) that describes that correspondence?

Exercise 15.2 (p. 132) Explain why simply *knowing* that the use of a lock in Prog. 13.5 has established critical sections in the two threads (in the sense of Fig. 15.2) allows you to conclude that the invariant `b>=0` is maintained.

Exercise 15.3 (p. 133) Why don't we need "bottom" $\perp$ as a possible value for `t` in the Owick-Gries version of a critical sections?

Exercise 15.4 Show that Prog. 15.6 implements critical sections.

Chapter 16

Peterson's algorithm for mutual exclusion

16.1 Introduction

In Sec. 13.5 we began the discussion of critical sections and mutual exclusion: a *critical section* (recall) is a portion of code that must be allowed to execute *without* being interrupted by other threads' executing code of the same kind. That is, all those code fragments in the whole system must be *mutually excluded* from each other, so that no two can be active at the same time. As we saw in the rest of Chp. 13, that is such an important programming tool that there are special hardware instructions (like *CAS*) for doing it, or by controlling interrupts (i.e. turning them on and off).

It might be surprising therefore that it is possible to program mutual exclusion *without* special instructions, or manipulation of interrupts. Although it isn't much done in practice, the "puzzle" of figuring out *how* to do it is a very good showcase for many of the techniques we have spent the earlier part of this book learning and practising: assertions, invariants, data abstraction, careful checking. As a special case, we will set it as an exercise right here. Do try it for yourself before looking at the (partial) answer!

Exercise 16.1 Using only single-assignment statements to simple variables (i.e. no multiple assignments, no expressions referring to more than one global variable, no sequences or other non-primitive data types) and body-free `await`-statements (i.e. `await` *cond* `do skip`), program a two-thread mutual-exclusion protocol that is safe (the two threads cannot be in the critical section at the same time), cannot deadlock (the two threads cannot both be "stuck", trying to enter the critical section), that does not stop one thread from entering the critical section simply because the other thread is not participating in the protocol at all (i.e. is busy elsewhere), and does not allow one thread to overtake the other indefinitely (i.e. no "queue jumping").

The structure should be as in the code skeleton in Fig. 16.1, and when complete should not be more than say 8 statements in each thread. And the solution should be symmetric, i.e. the code for *Th2* should be the same as for *Th1* except for swapping 1's and 2's in the obvious way.

```
Some initialisation # outside of both threads

# Th1 repeats this forever.

  "Other business 1"
single-assignment statements # ...
# ...  to shared (or not) variables
# ...  to indicate that Th1 wants to enter the CS

await cond1 # with no body
  "Critical section 1"
single-assignment statements # to leave the CS


# Th2 repeats this forever.

  "Other business 2"
single-assignment statements # prepare to enter

await cond2 # enter
  "Critical section 2"
single-assignment statements # leave
```

Figure 16.1 Framework for critical-section program

```
qs==[] # queue initially empty

# Th1 repeats this forever.

"other business 1"
qs= qs+[1]            #  Th1 joins qs at rear.
await qs[0]==1:    # qs[0] is the head of qs.
  "Critical section 1"
qs= qs[1:]         # qs[1:] is the tail of qs.

# Th2 repeats this forever.

"other business 2"
qs= qs+[2]            #  Th2 joins qs at rear,
await qs[0]==2:       #  waits until at front,
  "Critical section 2"
qs= qs[1:]                    # and leaves qs.
```
(16.1)

Figure 16.2 Peterson's algorithm implemented with a queue

16.2 Implementation based on a queue

Peterson's algorithm achieves mutual exclusion between two threads (safety), while avoiding the deadlock and starvation problems mentioned in Sec. 15.3 of the last chapter (i.e. achieving not only safety, but also liveness). That is (we will see), it is a complete answer to Ex. 16.1.

The key idea is to think abstractly about what you really want, and then to build that with the tools you have: so we begin by imagining that the two threads cooperate by using a two-place queue to decide which of them enters the critical section, and when. [1] The scheme is shown in Fig. 16.2, where it is implemented by a two-place queue. See Ex. 16.3.

Informal checking for SAFETY relies on `qs[0]` always being the name of the current thread in the critical section: i.e. that fact is invariant. Since `qs[0]` cannot have more than one value, there cannot be more than one thread there. See Ex. 16.4.

ABSENCE OF DEADLOCK relies on the fact that if both threads are at their `await`'s, then the queue is full and they cannot both be *not* at the head.

And ABSENCE OF STARVATION relies on the fact that once (say) *Th1* has joined `qs` it must eventually reach the front, provided *Th2* does not stay in its critical section forever (always assumed — otherwise there is no hope of achieving liveness at all). See Ex. 16.4.

The reason however that we don't immediately take Prog. 16.1 as "the solution" however is precisely because it *is* abstract: the manipulations of `qs` itself need protection from interference. To join the queue for example, a thread has to write its identity into the queue *and* increase an index, and it must not be interrupted by another thread while it is doing that. So the scheme Prog. 16.1 "as is" would just push the critical-section issue to a lower level.

[1] This approach is due to Jay Misra. But it is not yet a complete answer to Ex. 16.1, because it uses a sequence variable which *itself* requires mutual exclusion to be manipulated safely. In Ex. 16.1, only simple variables were allowed.

```
                    not t1 and not t2          # initialisation

# Th1                                          # Th2

OB1                                            OB2
t1,t= True,2        # atomic                   t2,t= True,1        # atomic
await not t2 or t==1                           await not t1 or t==2
CS1                                            CS2
t1= False                                      t2= False
```

The two multiple assignments, and the "`or`" in the `await` conditions, look like they might themselves need critical sections. But we will discover that they do not.

Figure 16.3 Encoding queue `qs` as three Booleans `t1`,`t2` and `t`.

To get from abstract to concrete, we use the data-abstraction techniques of Part II to implement the sequence `qs` in terms of more elementary variables: we will introduce a coupling invariant that links the "abstract" `qs` to a "concrete" representation as three simple Booleans `t1`, `t2` and `t`. Our first step for that is to realise that `qs` can have only 5 values: either it's empty, or it has just 1 or 2 in it, or it has both 1 and 2 but in either order. (This is a global invariant, and has to be checked.) Our coupling invariant is then that `t1` means that 1 is in `qs` (somewhere); and similarly `t2` means that 2 is in `qs`. And `t` (an "honorary" Boolean, so-called since it has only two values, which we could call `True` and `False`) applies only when *both* 1 and 2 are in `qs` — it indicates which of the two is in `qs[0]`, i.e. is at the head of the queue. (If `qs` is not full, then `t` has no significance.) With that coupling invariant we convert Prog. 16.1 into the program shown in Fig. 16.3, using the translations that the coupling invariant determines, following the techniques of Part II for the `qs`-manipulations and -tests to express them in terms of the three Booleans:

(a) Initially the queue is empty: so we have `t1==t2==False`.

(b) `qs= qs+[1]`, i.e. *Th1* joins the queue, becomes `t1,t= True,2`.

(c) `qs= qs+[2]` becomes `t2,t= True,1`.

(d) `await qs[0]==1`, i.e. *Th1* waits until it is at the head of the queue, becomes `await not t2 or t==1`.

(e) `await qs[0]==2` becomes `await not t1 or t==2`.

(f) `qs= qs[1:]`, i.e. leave the queue (either thread), becomes `t1= False` in *Th1* and `t2= False` in *Th2*.

Using those translations gives us Fig. 16.3, where there is no longer a queue variable `qs` — but still we have not yet eliminated the possible need for "lower level" critical sections: there are still multiple assignments, and there are still compound `await` statements that must be split into smaller pieces. That is the topic of Secs. 16.3 and 16.5 to come.

See Ex. 16.6.

16.3 Eliminating the multiple assignments

The code in Fig. 16.3 has two multiple assignments: they are `t1,t= True,2` and `t2,t= True,1`. Each is written on a single line, which is how we indicate that it's atomic; and because of that (1) we could check the local correctness with a single multiple substitution, and (2) we would not have to check for interference *between* the two assignments — they have no "between".

But to implement that "at run time" where we have only single-assignment atomicity, we would have to put the two statements in a critical section of their own: in the first case for example, we would have to write

```
lock
  t1= True
  t= 2
unlock
```
→ Indentation documents the critical section that the `lock`/`unlock` establishes. (16.2)

where the two statements in the critical section –between the `lock` and the `unlock`– can be in either order precisely because the locking is there. If we remove the lock, however, which is what we want to do, then the order of the two statements becomes very important — because interference might happen between them. It is even possible that neither order works (but luckily we will discover that one of them does).

See Ex. 16.5.

Thus when we split those assignments in Fig. 16.3, we have only two possibilities:

See Ex. 16.7.

```
t= 2          t= 1
t1= True      t2= True
```
(16.3)

or

```
t1= True      t2= True
t= 2          t= 1
```
.

The first of those (16.3) however is no safe: mutual exclusion is no longer guaranteed.

See Ex. 16.2.

Informal reasoning to show that the second alternative *is* safe would be that (on the left) Thread *Th1* is "about to" execute `t= 2` and that, if it is, variable `t1` cannot be set to `False` before it does so — because setting `t1` to `False` can be done only by *Th1* itself, and *Th1* is "here" (between the assignments) and not "there" (at the end of *CS1*). And the problem with the first alternative is now clear: it is that `t` *can* be set to 1, falsifying `t==2`, while *Th1* is between the assignments, because the `t= 1` is done by *Th2*.

But "informal" is only helpful, not conclusive. To check carefully, we add a label-like auxiliary variable. For two program fragments as below, the `L:` –designed to look like a label– is shorthand for introducing an auxiliary Boolean variable `L` that is true in between the two statements, and nowhere else. That is, the left-hand side is an abbreviation for the right:

```
    stmt1                stmt1; L= True  # atomic
L:  stmt2                stmt2; L= False # atomic
```

With that we can be more precise with "about to" and introduce two auxiliaries `L1`,`L2` in that style, giving Fig. 16.4 based on Fig. 16.3. In effect the `L` means that the thread is "poised" to execute *stmt2*, but has not yet done so.

See Ex. 16.8.

Although the assertions in Fig. 16.4 are long (three conjuncts, one of them a triple disjunct), they are easy to motivate: on the left, the `not L1` is there because control

```
                    not t1 and not t2          # initialisation

# Th1                                          # Th2

OB1                                            OB2
t1= True                                       t2= True
L1: t= 2                                       L2: t= 1
await not t2 or t==1                           await not t1 or t==2
{ not L1 and t1 and                            { not L2 and t2 and
  (not t2 or L2 or t==1) }                       (not t1 or L1 or t==2) }
CS1                                            CS2
t1= False                                      t2= False
```

The atomic double assignments from Fig. 16.3 have been split into single assignments.

The two assertions before *CS1* and *CS2* are inconsistent, and so establish mutual exclusion between the critical sections.

Figure 16.4 Peterson's algorithm.

is not at that "label" (and it is globally correct, since only *Th1* can affect `L1`); the `t1` is from the assignment `t1= True` (and is globally correct, because *Th2* does not assign to `t1`), and the `not t2 or L2 or t==1` is *implied by* the `await` condition `not t2 or t==1` — but it has been weakened by including the extra `(or L2)` precisely in order to make it globally correct:

- if `not t2` is falsified by *Th2*'s assignment `t2= True`, the disjunct as a whole is still true, because `L2` is now true;
- and if `L2` is falsified by *Th2*'s moving beyond the label `L2:` , then the disjunct as a whole is still true because `t==1` becomes true;
- and `t==1` cannot be falsified by *Th2*, since *Th2*'s only assignment to `t` is `t= 1`.

See Ex. 16.9.
See Ex. 16.8.

Figure 16.4 is Peterson's algorithm; and we have now established that it is safe.

16.4 Liveness of Peterson's algorithm

It is easy to check that Peterson's algorithm cannot deadlock: if both `await` conditions are `False`, we have `t!=1 and t!=2` which is impossible, because 1 and 2 are the only values assigned to `t`. (More precisely, we have that `t==1 or t==2` is a global invariant.)

See Ex. 16.10.

For absence of starvation, we must show that `not t2 or t==1` must eventually become –and remain– `True` if *Th2* continues to execute (and vice versa): for that we must assume that *Th2* cannot execute indefinitely in its critical section.

Once *Th2* leaves its critical section, it must eventually reach *OB2*; and while it remains there `not t2` is true, so that *Th1* can enter the critical section. But if *Th2* leaves *OB2*, then eventually `t==1` must become true and remain true — and again *Th1* can enter the critical section.

Thus *Th1* cannot be starved by *Th2*.

16.5 Peterson's algorithm without locks

In Fig. 16.3 it was remarked that there were two places where Peterson's algorithm might need locks for its implementation: one was the atomic multiple assignments, dealt with in Sec. 16.3; and the other was the multiple tests in the two `await`'s, which we look at now.

For the "multiple tests" we recall from Prog. 15.5 in Sec. 15.3 that an `await` can be implemented with a lock, no matter how complicated the condition, because the lock can be used to make evaluation of the condition atomic. In Prog. 15.6 however it was shown that if the condition is based on a single variable, a lock is not necessary.

The full truth lies between those two extremes: an `await` can be implemented without a lock provided its condition refers only to a *single* variable from another thread: that variable is read once, and then the calculation of the condition is based on the value read. But if two (or more) variables must be read from other threads, there is a risk that the thread performing the two reads could be interrupted between one read and the other.

Now Peterson's *Th1*-`await` does read two variables, both `t2` and `t`, and both can be written by *Th2*. Yet it can be implemented without a lock because the condition `t==1` is *stable* in *Th1*, meaning that *Th2* cannot falsify it. A lock-free implementation of `await not t2 or t==1` in *Th1* is therefore possible: it is

```
while True:
  if not t2: break                    (16.4)
  if t==1:   break    ,
```

where we have written the two tests sequentially to emphasise that they are evaluated non-atomically (i.e. instead of writing `if not t2 or t==1:`).

See Ex. 16.14.
See Ex. 16.11.
See Ex. 16.12.
See Ex. 16.13.
See Ex. 16.15.

16.6 *Exercises*

Exercise 16.2 (p. 135) Set out a schedule in the style of Prog. 15.7 that shows the first alternative at (16.3) above for splitting `t1,t= True,2` is not safe.

Exercise 16.3 (p. 139) Add assertions to Prog. 16.1 in Fig. 16.2 that are sufficiently detailed to check its safety carefully. Remember that manipulations of `qs` must have preconditions that ensure absence of "everyday" errors: if the queue is empty, for example, you can't check its first element, nor can you take its tail.

Exercise 16.4 (p. 139) Explain in detail why deadlock cannot occur in the Peterson's-algorithm system of Fig. 16.2, i.e. that `qs[0]!=1` and `qs[0]!=2` can never both be `False`. How do we know that `qs[0]` is defined?

See Ex. 16.3.

Exercise 16.5 (p. 141) Using Prog. 16.2 as inspiration (or otherwise), suggest a nice syntax for critical sections. Propose a checking rule (in the style of App. B) that would be used for it.

Exercise 16.6 (p. 140) In (b) and (c), where one thread joins the queue `qs` at its end, the variable `t` is set to the identity of the *other* thread — even though that thread might not be in the queue at all.

Why?

Exercise 16.7 (p. 141) Are there really only two possibilities for splitting the double assignment in Peterson's algorithm? Can you think of a third? Does it work?

Exercise 16.8 (p. 141) Check that the two assertions mentioned in Fig. 16.4 are indeed inconsistent, so establishing that *Th1* and *Th2* cannot simultaneously be in their critical sections.

Hint: See App. D.

Exercise 16.9 (p. 142) Consider just one "run" of each of *Th1* and *Th2* in Fig. 16.4: ignore the *OB* and *CS*, so that each thread is just four lines long. If you checked mutual exclusion for every single possible execution path possibly arising from running the two threads in parallel, how many paths would you have to check? Since each path is 4 lines long, how many pre-post checks (i.e. Hoare triples) would that checking require altogether?

Hint: It is more than 500.

Exercise 16.10 (p. 142) Check that the *negations* of the two `await` consitions mentioned in Fig. 16.4 are indeed inconsistent, so establishing that *Th1* and *Th2* cannot deadlock.

Hint: Remember to consider global invariants; and see App. D.

Exercise 16.11 (p. 143) What is the difference between a condition's being *stable*, being *globally correct* and being *globally invariant*?

Exercise 16.12 (p. 143) Why would

```
while True:
  if cond1: break
  if cond2: break
```
(16.5)

not be a correct implementation of `await` *cond1* `or` *cond2* if neither *cond1* nor *cond2* were stable?

See Ex. 16.14.
See Ex. 16.15.

Exercise 16.13 (p. 143) Why couldn't we simply write `while t2 and t!=1: skip` for Prog. 16.4?

Exercise 16.14 (p. 143) Why do we know that Program 16.4 is a correct implementation of `await not t2 or t==1`?

See Ex. 16.15.
See Ex. 16.12.

Exercise 16.15 (p. 143) Give a lock-free implementation of `await` *cond1* `and` *cond2* where *cond1* is stable and each of the two conditions refers to at most one shared variable (but not necessarily the same one). Be sure to take starvation into account.

See Ex. 16.14.
See Ex. 16.12.

Chapter 17

Garbage collection on the fly

17.1 Introduction

In Sec. 13.1 we mentioned that concurrency occurs both "in the large", e.g. between thousands of *ATM*s accessing the same bank account from widely separated locations, but also "in the small", deep within the operating system of a single isolated computer. The latter is the subject of this chapter: garbage collection "on the fly".

We explain below what "garbage" is (in a computer), and what "on the fly" means — but in broad terms the problem here is that for efficient execution of some computer programs it's convenient not to force them to "clean up after themselves" as they go. It takes too much time, and it might not be necessary anyway. But sometimes it *is* necessary, especially if the program runs for a long time. In that case, the straightforward solution is to pause the program temporarily, and clean up everything at that point. Then the program is allowed to resume.

An alternative solution, but much more complicated, is to have an extra "clean up" program running separately in parallel, all the time; and then the main program does not have to be paused. But if those two things are active at the same time, then you are dealing with concurrency; and you must make sure that the extra program does not interfere with the main program. Think of leaving empty coffee cups around the house all day, because you (the main program) are too busy to take them to the kitchen yourself; and at the end of the day someone (perhaps you?) collects them all at once. That's simple and efficient — unless at some point during the day you go to make your next coffee and find that there are no clean cups at all.

The more complicated "on the fly" [1] alternative is to have a separate "parallel" person whose sole job during the day is to search continually for used empty cups, take them to the kitchen, clean them and return them to the shelves. Interference –which we must prevent– is in this case the situation where you turn away from your nearly-but-not-quite empty cup, just for a moment — and when you turn back to take that very last sip... the cup has disappeared.

This chapter is about how to implement the on-the-fly, collect-in-parallel algorithm for computers in a way that allows it to be checked rigorously that only garbage is collected, and never items that are still in use.

[1] "On the fly" is perhaps by analogy with in-flight refueling, i.e. without having to land and wait.

17.2 What "garbage" is

Garbage is a feature of programs that use "pointer variables" either explicitly, or implicitly behind the scenes. Pointer variables refer to other variables indirectly, i.e. to their location and not directly to their value. The simplest example of a pointer (here in *C* rather than Python) is the variable `p` that refers indirectly to `n` below:

```
int n= 0;           // Declare integer n; initialise it to 0.
int *p;                     // Declare pointer-to-integers p.
p= &n;                     // Assign "the address of n" to p.
*p= *p+1;          // Increment the integer that p points to.
// Because p was pointing to n, it is n that was incremented.
{POST n==1}
```

Pointers are however not very often used as above, to access variables (like `n`) that were available directly by name already (i.e. are "in the stack"). We could have achieved the same effect much more simply with `n= n+1`. Instead pointers are combined with "dynamic memory allocation" where, rather than declare an integer at compile time (say via `int n`, so that it can be referred to directly), instead space is allocated for the integer at runtime: the space comes from an area called "the heap", which is reserved precisely to allow such run-time allocations. The integer has no name, but does have an address; and then the above program becomes

```
int *p;                     // Declare pointer-to-integers p.
// Allocate space from the "heap" for an integer;
// and make p point to it.
p= malloc(sizeof(int));
*p= 1;              // Set the integer that p points to to 1.
```
(17.1)

Typically, the heap (and pointers) are used where the amount of memory a program will use is not known until it is run (because for example it might depend on how much data the program is given). And that is where the "garbage" problem comes from: it occurs when a program allocates space (using `malloc()`) but then loses it by "losing" the value of variables (like `p`) that point to it. If for example we used `p` to allocate a *second* integer in Prog. 17.1 above, getting something like

```
int *p;                     // Declare pointer-to-integers p.
p= malloc(sizeof(int));
*p= 1;              // Set the integer that p points to to 1.
p= malloc(sizeof(int));             // Garbage created here.
*p= 2;      // Set this second integer that p points to to 2.    ,
```

then the first integer is still `1` (somewhere on the heap) but the program cannot find it: its address *was* in `p`, but is no longer. The "`1`" has become garbage, an integer-sized space the program has taken from the heap but can no longer use.

The garbage problem above is avoided by returning the no-longer-needed space to

the heap, just before the second assignment to p. The result would be

```
int *p;
p= malloc(sizeof(int));
*p= 1;
free(p);          // Give the integer-space back to the heap.
p= malloc(sizeof(int));                  // No garbage now.
*p= 2;    .
```
(17.2)

This is precisely the "clean up your coffee cup as you go" scenario from above.

17.3 Why garbage needs to be collected

If all programs were written as carefully as Prog. 17.2 was, garbage would not need to be collected: in effect, `free()` is "putting the garbage in the bin" instead of just dropping it onto the floor. But knowing when something is garbage, and when it is not, can be tricky. For example, although

```
p= malloc(sizeof(int)); *p= 1; free(p)
```

is fine (i.e. the garbage is properly "tidied up", not just thrown away), the related

```
p= malloc(sizeof(int)); *p= 1; q= p; free(p)
```

is *not* fine: when p is freed here, it is not garbage — the address formerly in p is still held by q, and the integer returned to the heap can be accessed via *q even though in the meantime it might have been allocated (`malloc()`'d) to something else.

Because of that (and other complications), the `free()`'ing of no-longer-needed space is often not done when it should be... the lesser of two evils; and so the garbage builds up, invisible and unreachable but still taking up space. This is known as a "space leak", and eventually the program might run out of memory — even though the memory it is actually *using* might be quite small.

17.4 How garbage is collected

There are two main ways garbage is collected.

The first way, called *mark and sweep*, is to pause the execution of the program, and then let a "Scanner" go through the heap, marking all reachable nodes: if a marked node points to an unmarked node, then the latter becomes marked also; and then the check is repeated, propagating the markings. Once there is a check in which no marks are propagated, a Collector then sweeps through the heap, moving all unmarked (hence unreachable) nodes to a "free chain", which is where `malloc()` gets them from when the program needs them (again). That process is illustrated in App. F.1.

In more detail: the program is paused, and all nodes are set to unmarked, thus considered as potential garbage. Then the "root nodes" that are explicit and named program variables (like p in Prog. 17.1 above) are marked as "reachable"; and that process is repeated, marking more and more nodes reachable from other reachable nodes, until no new nodes are marked. Then everything (still) unmarked is considered garbage, and the Collector sweeps through the whole heap, moving those unmarked nodes to the free chain. Once that is done, the program is resumed. Since the in-use

markings were propagated from the program variables through the heap until they could not go any further, all that was left, unreachable, was garbage.

The main disadvantage of the mark-sweep method, as described above, is the pausing of the program: anyone (or -thing) *using* the program –a self-driving car for example– is likely to be at risk while it is paused for garbage collection.

The second main way of collecting garbage is *reference counting*: a newly allocated portion is given a reference count of 1; every time its address is copied into another pointer the reference count is increased by 1; every time a pointer is reassigned to point elsewhere, the count of what it *used* to point to is decreased by 1, and the count of where it *now* points increased by 1; and when any count reaches 0, the portion is `free()`'d automatically. [2]

The main disadvantage of reference counting is that circular structures, i.e. those that point to themselves although nothing else does, will not be collected. Here is an example (still in *C*), illustrated in Fig. 17.1:

```
typedef struct node struct node* ptr; Node;
Node *p,*q;
p= malloc(sizeof(Node));       // *p's reference count is 1.
q= malloc(sizeof(Node));       // *q's reference count is 1.
p->ptr= q;                  // *q's reference count is now 2.
q->ptr= p;                  // *p's reference count is now 2.
p= NULL;                 // *p's reference count is back to 1.
q= NULL;                 // *q's reference count is back to 1.
// Both *p and *q are garbage, but their reference counts are not 0.
```
(17.3)

See Ex. 17.1.

Garbage collection "on the fly", this chapter's topic, is a variation on mark-sweep method — but (astonishingly) it *can be* carried out without pausing the program. That is, the marking and propagating –the scanning– occurs *concurrently* with the continuing activity of the program: the program might slow down slightly, but it does not at any time stop altogether. The challenge of on-the-fly collection however is to make sure that the "Mutator" (the program, which changes the pointers) and the Scanner (which marks non-garbage) do not interfere with each other. An example of such interference is shown in Fig. 17.2. There is no danger of interference between the Mutator and Collector, however, since the portions freed by the Collector cannot (by definition) be reached by the Mutator.

See Ex. 17.2.
See Ex. 17.22.

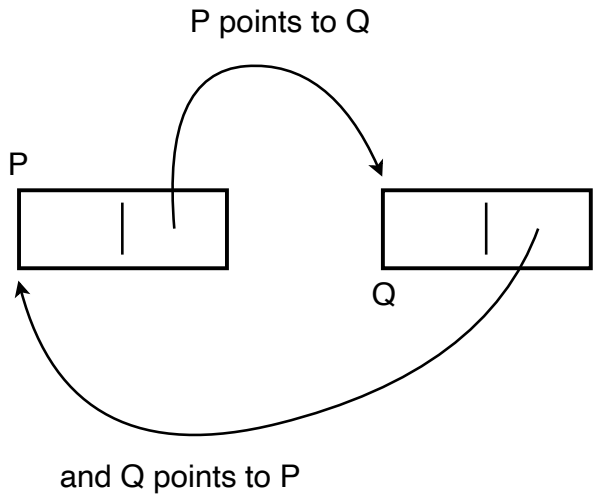
17.5 Origins of this presentation

Pointer-generated garbage is not itself the problem: in Sec. 17.4 we saw some solutions to collecting it.

The issue here is *concurrency*. What we present in this chapter concerns finding a concurrent program that solves an easy to understand but still very difficult problem, and to check that the program works. Ideally, the "finding" and the "checking" go together, one supporting the other:

> " One of the problems that have to be dealt with is organizing the cooperation of ... concurrent processes so as to keep exclusion and synchronization constraints extremely weak, in spite of very frequent manipulations (by all processes involved) of a large shared data space. The problem of garbage collection was selected as one of the most challenging problems

[2] Python uses reference counting.



Both nodes `P`, `Q` above are "pointed to", and so their reference counts will be (at least) 1 in each case. But if they are *exactly* 1, i.e. not pointed-to from anywhere else, then they are both garbage.

Yet looking for reference-count 0's won't find them.

Figure 17.1 Circular structure: garbage, but reference counts not zero

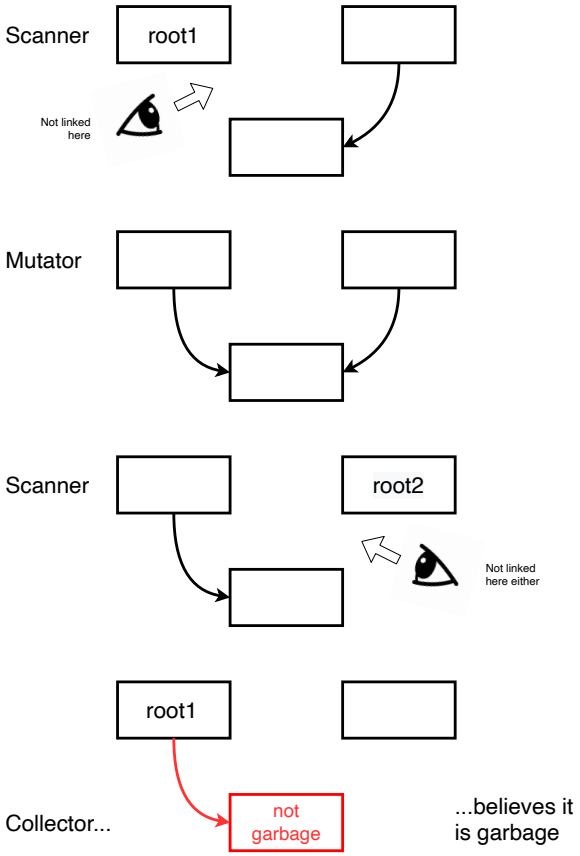
in this respect (and hopefully a very instructive one). Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap possible. [3] "

That is, although concurrent garbage collection "on the fly" is a real, practical problem, it was chosen by the authors of the above mainly as an exercise in writing *any* (complicated) program and being sure that it worked. (That is of course the point of this book!) And our first step –following those authors– is to be very clear about the context, in this case the behaviour of pointers and what programs do with them. So let us begin with that.

A *node* is a structure containing pointers (and possibly other data, that we won't worry about); if a node is declared directly in a program, it is a *root* node, because the program does not need to follow pointers to find it. But nodes can allocated at runtime as well, and "pointed to". A node is said to be *reachable* just when either it is a root, or it can be found by following a chain of pointers from a root. Here's what programs can do with pointers:

(a) A reachable pointer `p` (see below) can be "swung", that is changed from pointing to "this" node to pointing instead to "that" reachable node. (Note the "reachable" — a pointer cannot be swung unless the target of the swing –the node it is *about* to point to– is reachable already: see Sec. 17.8 for why that's important.)

[3] This quote comes from the original article by Dijkstra, Lamport, Martin, Scholten and Steffens, later modified by van der Schnepscheut. Our presentation here comes from the latter. None of those authors used Owicki-Gries reasoning, though, because it had only just been invented and was not yet widely known. Later, however, Gries himself did: but still our presentation is based mainly on van de Schnepscheut's.



The Scanner looks on the left (top sketch), and the bottom node seems to be garbage: but in fact it is reachable from the right-hand root. When the Scanner looks on the right (two steps later, third sketch), however, the bottom node still seems to be garbage: but now it is reachable from the left-hand root, because the Mutator swung the pointer between the Scanner's two visits (second sketch), "behind the Scanner's back" so to speak.

Then –last sketch, at bottom– the Collector, believing the unfortunate node to be garbage because it found no link on either side, collects it — in error. It is not garbage.

Figure 17.2 Interference between Mutator and Scanner

(b) A reachable pointer can be changed from pointing to "this" node to pointing instead to a new node (`malloc()`). (Remember that `malloc()` is the *C*-name for the procedure by which new nodes are allocated at runtime.)

(c) A reachable null pointer can be changed to point to a reachable node.

(d) A reachable null pointer can be changed to point to a new node.

(e) A reachable pointer can be set to null.

Cases (a), (b) and (e) can create garbage; Case (c) and (d) "recycle" it.

Following the original authors, we now simplify the problem by reducing those five cases above to just one. Instead of having a "null" pointer, we have a special root node called `N` that points to itself: all other previously-called-null pointers point to `N` instead. And the "free" nodes are themselves in a chain starting from a special root node `F`, whose last element points to `N`. With those modifications, the only operation on pointers we must consider is (a). That's how the original five cases are reduced to just one.

The other original Cases (b)–(e) of course sometimes become *several* instances of the single remaining Case (a): to implement (b) for example we would replace `p` with the pointer within `F` that points to the head node `h` say of the free chain (provided it is not `N`, in which case we are out of memory); then `F`'s pointer would be updated to point to the tail `t` say (found within node `h`) of the free chain; then the pointers within `h` would be both set to point to `N`; and finally `p` would be set to point to `h`. See Ex. 17.3.

17.6 Use a Boolean "mark-bit" to control scanning

The point of the example in Fig. 17.2 was to show that the Scanner will need to mark the nodes somehow, because otherwise it could not possibly notice the interference of the Mutator — that is, we cannot escape marking altogether. But we will need only 1 extra bit per node. And so we assume every node is marked with a colour, either white W or black B and, during scanning, white will mean "not yet reached" and black will mean "reachable from a root".

The *scanning* phase goes through all nodes in some sequence (directly in the heap, not following pointers). Call the nodes pointed to by a given node its "successors". We begin with a "whitening" phase

(i) Go through the heap, one node at a time (in any order — just make sure that all of them are examined). Set the colour of every node to white W.

And then we have a "scanning" phase:

(ii) Set the colour of every root node to black B. That includes `N` and `F`, but also any node that is declared explicitly as a variable in the Mutator program (like `p` and `q` in Prog. 17.3).

(iii) Propagate the B colouring by scanning through all the nodes again (and again), and for every B-marked node encountered, set its successors –the nodes it points to– to B as well. (Note — after that, however, do not follow the pointer to the successors; instead keep scanning nodes in the original sequence.) When all nodes have been scanned on this pass...

(iv) Go back and do it again if necessary: repeat the previous step until an entire scan is completed in which no W nodes were changed to B.

For "until no W nodes were changed to B", we will use a Boolean flag `c` to record whether any changes were made.

Once the whole scanning process is complete, there should be no B-marked nodes that point to W nodes anymore, anywhere; and, since all roots are B, we know that all W nodes are not reachable, and the scanning can stop. The *Collector* phase then begins, and goes through all the nodes again in some sequence, transferring all W nodes to the free chain that begins at root `F`. (And the process begins again, with whitening at (i) above).

But –remember– while all this scanning and collecting is going on, the *Mutator* –our application program that is possibly *creating* the garbage– is "out there" potentially messing things up by executing pointer-swing operations of type (a) from above. It is (potentially) *interfering* with the scanning/collecting activity: in Owicki-Gries terms, actually, it is potentially interfering with the *assertions* that we will be using to check the garbage-collection thread.

The postcondition that we need for scanning, for collection to be safe, is "White nodes are garbage." But "garbage" means "not reachable from a root", and all the roots are black. So we can take as our postcondition "No black node points to a white one." The variant for the inner loop (iii) will be "number of nodes left to scan"; the variant for the outer loop we will see below.

See Ex. 17.17.

See Ex. 17.5.

17.7 Checking local correctness

We now look at the scanning phase in more detail. For *local correctness*, the Scanner code uses a "Boolean" `c` to note whether any B↦W's were removed.[4] For now, we imagine that the Scanner is executing on its own, in particular without the Mutator running at the same time. It is[5]

See Ex. 17.4.

```
# Scanner (Version 1): Find and remove all B↦W's.
c= W
repeat
  c= B
  for n in nodes: # In some order: doesn't matter what.
    if n.colour==B:
      if n.left.colour ==W: n.left.colour, c= B,W
      if n.right.colour==W: n.right.colour,c= B,W
until c==B
```

(17.4)

The inner loop's variant is "the number of nodes not yet examined in `nodes`". The variant for the outer loop is "the number of W's *including* `c`".

See Ex. 17.6.

Now we consider the two invariants. Since for the outer loop we want postcondition "No black node in `nodes` points to a white node." we simply take as our invariant

If `c==B` then no black node in `nodes` points to a white node. (17.5)

Finding an invariant for the inner loop is in the style of "iterating up" (Sec. 3.3.1) where the iteration is from "no nodes scanned" to "all nodes scanned". Writing B↦W for "black points to white", it is

If there was a node in the nodes scanned so far with B↦W, then `c==W`. (17.6)

[4] We call it "Boolean" because it is two-valued, and because of what it means. For convenience in checking, however, we make it a colour, either B or W. It's an "honorary" Boolean.

[5] Python does not have a `repeat` loops; but they are easily implemented with `while True:` and `break`.

On termination of the `for`-loop, "nodes scanned so far" will be "all nodes", and so the postcondition of the inner loop is simply

> If there is any node at all with B↦W, then `c==`W.

(That is a nice example of the "split a conjunct" style of invariant from Sec. 3.2.) Then the `until` condition of the outer loop `c==`B gives the overall postcondition

> There is no node with B↦W.

Finally, we return to the inner invariant (17.6) — and here is perhaps the only bit of checking that is not routine. The assignment `n.left.colour=` B can *introduce* a B↦W into the "nodes scanned so far" because `n.left` itself (rather than `n`) might already have been scanned. If *it* was white and had a white successor, then nothing would have been changed: but now we have made it black, and so have introduced a B↦W into the already scanned portion of `nodes`. That is why we must set `c` to W.

The above effect is supported by our intuition that the only reason we must make multiple scans is that some nodes might "point backwards" into the already scanned portion of `nodes`; and so we need another scan to go back and repair those, i.e. to change those B↦W's to B↦B's; but those repairs might introduce B↦W's even further back, and so on. That's why the Scanner must keep scanning until `c` remains B.

We have now checked local correctness of the Scanner.

17.8 Checking global correctness

Although we now see (from above) that the Scanner checks-out in isolation, we must check global correctness too: remember that in reality the Mutator is active at the same time. We must now imagine that pointer swings in the Mutator are occurring as the Scanner (Prog. 17.4) executes. Although we wrote the Scanner's inner-loop conditionals on single lines (for neatness), we will allow interference between the `if` condition and its then-branch. Multiple assignments however, also on one line, will for now however remain atomic. In Fig. 17.3 we illustrate what might go wrong if the Scanner and the Mutator are executing in parallel, i.e. at the same time: it's a counter-example to our efforts so far.

See Ex. 17.7.
See Ex. 17.8.

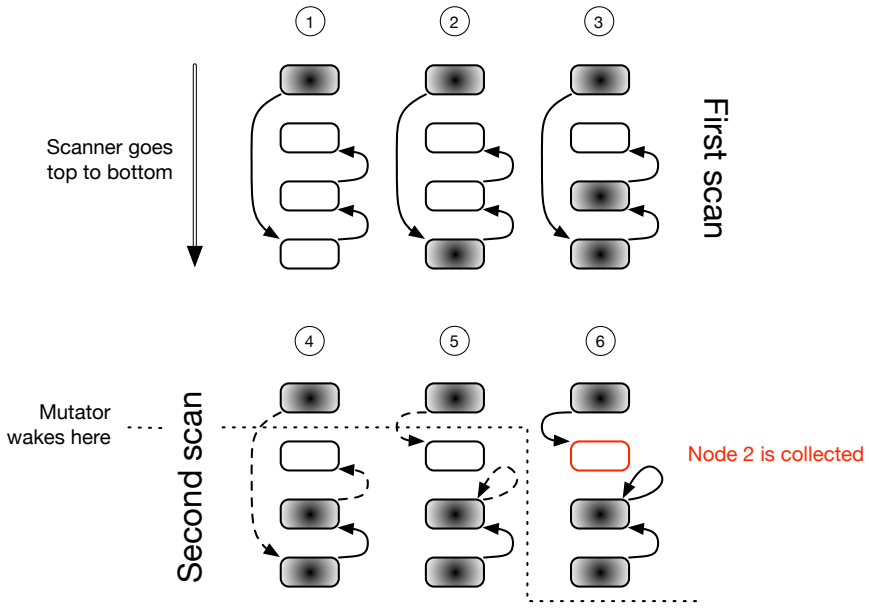
The counter-example (its existence) means that our program Prog. 17.4 is wrong — but how do we find the error? Just "head-scratching" is not enough, not effective: instead, we look at each of the assertions we used for local correctness and ask "Is this the one that is not globally correct?"

It's the inner-loop invariant (17.6) that is not globally correct: the Mutator can swing a pointer in an already-scanned node and make it point to an arbitrary white node somewhere else. (In retrospect, that does seem obvious.)

And what about the outer-loop invariant (17.5)? In fact it *is* globally correct: the Mutator cannot make its antecedent `c==`B true, because the Mutator cannot assign to `c` at all. Can it make its consequent false: that "no black node in `nodes` points to a white node"? It cannot, at least not when `c==`B — for if `c==`B then *all* white nodes are unreachable, and so the Mutator cannot swing any pointer to one of them. (Remember (a) in Sec. 17.5.)

There are two possible ways we might fix the interference with the inner invariant. One is that the Mutator is allowed access to `c` and executes `c=` W every time it swings a pointer. But this is terribly inefficient: the Scanner would have to re-scan if even just one pointer had been swung by the Mutator; and in fact the Scanner's

See Ex. 17.21.



Here is a counter-example for Prog. 17.4. We concentrate on just one successor for each node, and the Scanner visits the nodes from top to bottom.

At (1) we see the four nodes: the root, Node 1 at the top, points to Node 4; and each of the others points to the one before, except for Node 2 which points to `N`, i.e. is null. (We omit the null pointers from the picture.)

In the first inner loop of the Scanner, Node 4 is blackened immediately, giving (2); and then at the very end Node 3 is blackened too (3), because Node 4 was black.

Since some nodes were blackened, a second scan is begun. But just after its first step (where the Scanner sees that Node 4 is already black), the Mutator interferes by swinging Node 1's pointer from Node 4 to Node 2, and Node 3's pointer from Node 2 to itself, giving (5).

The Scanner then resumes and completes its scan, reaching (6) — and terminating, because it did not blacken anything. Node 2, though it is reachable from Node 1, is now up for collection because it is still white.

Figure 17.3 Interference between Mutator and Scanner

variant might not decrease, which would mean that the scan could go on forever. (Remember Jack, from Sec. 1.2.)

The other option is that when the Mutator swings a pointer it *atomically* colours the new target black. In Fig. 17.3 that would mean that the Mutator's action in Stage (4) would make Node 2 black in Stage 5, preventing it from being collected in error. But –for atomicity– every pointer-swing-and-blacken would then have to be put inside a critical section, a lock-then-unlock which, however small in size, is likely to be too inefficient in general because there are potentially too many executions of them.

But –surprisingly– even *with* atomicity here the "solution" so far is not correct: see Fig. 17.4. The problem is that the *target*-blackening done by the Mutator might itself introduce a B↦W into "nodes scanned so far", breaking the invariant. (Again that seems obvious once it's pointed out: but –ask yourself– what made sure that it *was* pointed out? It was "checking the invariant". That's why we do it.)

Because of those failures, we will have to do something more sophisticated than the straightforward "Use a 'Has anything changed?' flag".

17.9 Using a count to control scanning

"More sophisticated" is encouraging: it is much better than "start again from scratch".

Although the "changed-flag `c`" approach of the previous sections did not work, we have still made progress in our understanding of the problem and its possible solution. First, we learned from it that the Mutator must communicate somehow back to the Scanner that a node has been blackened: that suggests our use of B-counting, introduced below. Second –as we will see- the invariants and variants we used in Secs. 17.7 and 17.8 will suggest the invariants and variants we will use for the counting version: we will not have to discover them from scratch.

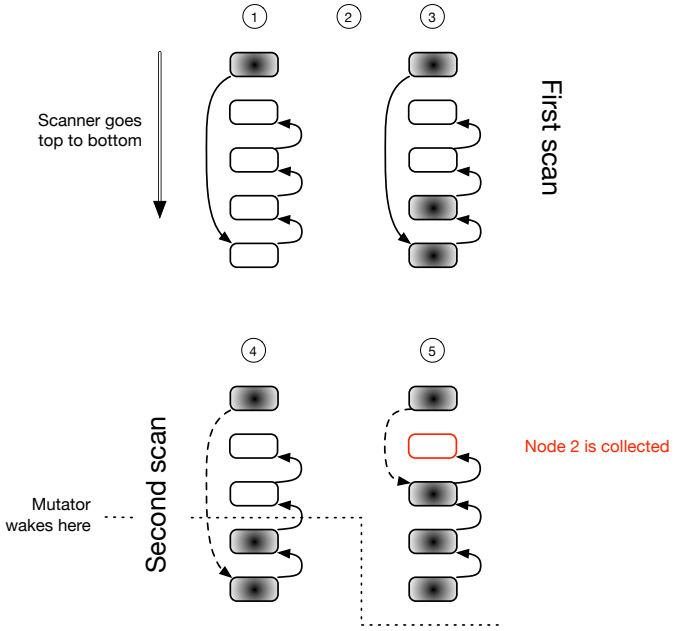
Thus we rewrite the Scanner so that it controls termination not with a "Boolean" but rather based on whether the number of black nodes in `nodes` has increased: we use two variables `nB`,`oB` for "new blacks" and "old blacks" respectively, and assume –temporarily– that the expression `|blacks|` gives atomically the exact number of black nodes in all of `nodes`. [6]

As mentioned above, the intuitive reason for this approach is that part of our earlier problem (with Scanner Version 1) was that the Mutator could not access the Scanner's "Boolean" variable `c` in order to indicate that by blackening a node it might have introduced a B↦W into the already scanned portion of `nodes`. Our counting approach here gets around that because a newly blackened node, whether caused by the Scanner *or by the Mutator*, will be detected automatically in the Scanner simply by an increase of `|blacks|`. In this way the Mutator is communicating with the Scanner implicitly.

So we write just `|blacks|` for "the number of black nodes in `nodes`", and –as stated above– we assume for the moment it is evaluated atomically. (Relaxing that assumption will be dealt with further below: for now, we just accept it.) We also continue to assume that the Mutator swings-and-blackens atomically. (That too will be relaxed later.) Our new Scanner (Version 2) is in Fig. 17.5.

The assertions we introduce are the invariants `Inv1` and `Inv2` for the inner loop, and `Inv3` for the outer loop. (As before, for `repeat`-loops we do not require the invariant to be true on the very first iteration.)

[6] The idea of counting rather than "flagging" comes from Ben-Ari, and was also used by van der Snepscheut.



Here is a counter-example for Prog. 17.4 even when the Mutator's swing-then-blacken is made atomic.

The Mutator wakes up midway through the second scan, and swings Node 1's pointer to Node 3, blackening Node 3 (atomically) at the same time.

When the Scanner resumes it finds Node 4 is black, but the node it points to, Node 3, *is already black* — so the Scanner simply carries on, doing nothing here. And when it completes its second scan, no node has been blackened (by it), so the overall scan terminates. Node 2, not garbage, will then be collected.

The problem, now revealed, is that the Scanner does not "notice" node-blackening done by the Mutator, and so might terminate early. That's the motivation for the approach taken in Sec. 17.9.

Figure 17.4 Interference between Mutator and Scanner

```
          # Scanner (Version 2): Find and remove all B↦W's.
Atomic →  nB= |blacks|
          repeat
            oB= nB
            for n in nodes: # In some order: doesn't matter what.
              # Inv1: If |blacks|==oB then no B↦W's in already-scanned.
              # Inv2: |blacks|>=oB.
              if n.colour==B:
                n.left.colour=  B
                n.right.colour= B
Atomic →    nB= |blacks|
            # Inv3: if nB==oB then no B↦W's anywhere.
            # Inv4: |blacks|>=nB.
          until nB==oB
          # POST No B↦W's anywhere.
```
(17.7)

Note that the two inner `if`'s of Prog. 17.4 have been removed in Prog. 17.7: with a blacks-counting scheme they are not necessary.

Figure 17.5 Scanner Version 2

Here is our reasoning for local correctness of those assertions:

(a) `Inv1` is true initially because nothing is (yet) scanned: already-scanned is empty.

(b) `Inv2` is true initially because of the initial assignment `oB= nB= |blacks|`, for the first iteration of the outer loop, and because of `Inv4` for subsequent iterations.

(c) `Inv1` is maintained because, from `Inv2`, the antecedent `|blacks|==oB` can never change from `False` to `True`; and if the consequent is made false by an assignment `n.--.colour= B`, then `|blacks|` increases and the antecedent becomes false as well.

(d) `Inv3` is true because of `Inv1` and the `for`-loop termination condition that "already scanned" is now "anywhere", and the assignment `nB= |blacks|` just before.

(e) `Inv4` is true because of the assignment `nB= |blacks|` just before.

(f) The overall postcondition is true because of `Inv3` and the `until`-condition.

Now we must do global correctness: we must consider the effect of the (still including some atomicity) Mutator, which can introduce a B↦W anywhere by swinging a pointer to a new target that it changes (at the same time) from W to B. (The possibly introduced B↦W would be from the newly-B target to some other W node.) If it does that, however, it must increase `|blacks|` at the same time (atomically, for now). It can never decrease `|blacks|`.

Here is our reasoning for global correctness:

(a) Inv1 is globally correct for the same reason given at (c) above. There the "interference" was from within the body of the `for`-loop in the Scanner itself; here it is from the Mutator.

(b) Inv2 and Inv4 are globally correct because the Mutator can only blacken, never whiten nodes.

(c) Inv3 and POST are globally correct because "No B↦W's anywhere." cannot be falsified by the Mutator: it blackens a node (possibly introducing a new B↦W from it to somewhere else, as explained just above) only if the node is currently white but is pointed-to by some other black node... in which case there was a B↦W already.

But –actually– when checking global correctness, strictly speaking we have to check the intermediate assertions as well — even the "invisible" ones that we tend to skip over while doing local correctness, because they were so obviously right. In this case, those assertions are

```
if n.colour==B: { n.colour==B }
  n.left.colour=  B
  n.right.colour= B
else: { n.colour==W } skip    ,
```
(17.8)

where we have even made explicit the hidden `else`-part of the one-branch `if`-statement. And in Prog. 17.8 just above, in fact the assertion `n.colour==W` is *not* globally correct, because the Mutator can change a node's colour from W to B.

If it does, though, then `|blacks|` will increase; and so the solution to this global-correctness problem is to weaken the `else`-assertion to

```
|blacks|==oB ⇒ n.colour==W    ,
```

which is (still) locally correct, because it is weaker than before (but, more explicitly, because the code of Prog. 17.8 makes its consequent `True`). And it is (now) globally correct because if the Mutator falsifies the consequent –changes `n.colour` from W to B– then the antecedent `|blacks|==oB` is `False`.

But there is another "invisible" assertion we must check, shown here in Prog. 17.9

```
if n.colour==B:
  n.left.colour=  B { n is not B↦W }
  n.right.colour= B { n is not B↦W }    ,
```
(17.9)

because this node `n` is going to change from unscanned to scanned (at the end of tis iteration of the inner loop), and it (still) must not be B↦W when that happens. It might in principle become B↦W if the Mutator swings pointer `n.--` after this code but just before the end of the inner loop; but the target of the swing must have colour B once the swing has been carried out, because the target is blackened atomically (for the moment) as the swing occurs.

Thus Scanner Version 2 (with its atomic `|blacks|`) and the Mutator (still with its atomic swing-and-blacken) is both locally and globally correct. It works! And yet...

We still have two things to do to reach a practical and efficient solution: we must count the blacks non-atomically, and we must allow the Mutator to swing then blacken non-atomically.

```
   Auxiliary → cB= |blacks| # The number of blacks when we start.
Specification → nB:[ PRE cB<=|blacks|, POST cB<=nB<=|blacks|]
```
(17.10)

The point of having a specification here is that we can use it to check carefully that the surrounding program will work, given this approximate-count program fragment, without (yet) actually having to code it up. If we find out that if fact the approximate count is not sufficient to check the surrounding program, then there is no point in implementing it — and we have saved ourselves some work.

Figure 17.6 "Approximate" black-counting specification

17.10 Evaluating |blacks| non-atomically

In general there is no way to "count the blacks atomically" without actually imposing a critical section around the counting loop, which (as we've already mentioned) is exactly what we're trying to avoid: that would hardly be "on the fly". But –we notice– the Mutator's interference with `|blacks|` is limited: it can not decrease it. And that turns out to be enough to solve the problem.

We introduce an *auxilary* variable `cB` (for "current blacks") to capture the actual value of `|blacks|`, and we then count the actual blacks in a non-atomic loop, say into a variable `nB` (for "new blacks"). The result will be that after the count-loop is complete we'll have only `nB>=cB`, more than were there when the count started, because new blacks might have been created "in front of our count" as we were counting. But also we'll have only only `nB<=|blacks|`, because new blacks might have been created "behind our count", i.e. that we did not notice.

The reason that will still be good enough is that we can use it –before we start the scan– to capture in say `oB` ("old blacks") a black-count that is possibly too low; then after the scan we capture in say `nB` ("new blacks") a block-count that is possibly too high. If "maybe too low" $\leq$ "just right" $\leq$ "maybe too high" *and* "maybe too low" and "maybe too high" are equal, then all three must be equal. That is, if it turns out after all that `oB==nB`, we know that they both equal `|blacks|` at that moment, and that the number of blacks did not increase.

Rather than write that block-counting code now, however, we will *specify* it and then use (only) its properties (i.e. its specification) to make a Scanner Version 3 and check that. Only if Version 3 works, we will bother to *implement* our specification of block counting. The "approximate counting" program fragment is shown in Fig. 17.6, where the specification stands for some code, which we will write later, that would check if placed between that pre- and postcondition pair.

Using Prog. 17.10, we write Scanner Version 3 as in Fig. 17.7. Remember that the rationale is that if approximate black-count before (which was possibly too low) is equal to the approximate black-count after (possibly too high), then in fact the black-count cannot have changed.

Many of our checks for the earlier Version 2 still go through. But we will check them all again anyway:

Local correctness:

(a) Unchanged — `Inv1` is true initially because nothing is (yet) scanned.

(b) `Inv2` is true initially because `nB<=|blacks|` is established before the outer loop and there is an assignment `ob= nB`. We continue to rely on `Inv4` and that assignment for subsequent iterations.

```
                # Scanner (Version 3): Find and remove all B↦W's.
                nB= 0
                repeat
                  oB= nB
                  for n in nodes:
                    # Inv1: If |blacks|==oB then no B↦W's in already-scanned.
                    # Inv2: |blacks|>=oB.
                    if n.colour==B:
                      n.left.colour=  B
                      n.right.colour= B                                    (17.11)
Auxiliary →     cB= |blacks|
      New →     # oB<=cB and (if cB==oB then no B↦W's anywhere) # LC and GC.
                nB:[ PRE cB<=|blacks|, POST cB<=nB<=|blacks| ]
                # Inv3: if nB==oB then no B↦W's anywhere.
                # Inv4: |blacks|>=nB.
                until nB==oB
                # POST No B↦W's anywhere.
```

Figure 17.7 Scanner Version 3

(c) Unchanged — `Inv1` is maintained because, from `Inv2`, the antecedent `|blacks|==oB` can never change from `False` to `True`; and if the consequent is made false by an assignment `n.--.colour= B`, then `|blacks|` increases and the antecedent becomes false as well.

(new) The new assertion's `oB<=cB` comes from `Inv2` and the assignment `cB= |blacks|` just before, and the

```
If cB==oB then no B↦W's anywhere.
```

is true because of `Inv1` and the `for`-loop termination condition that "already scanned" is now "anywhere", and the assignment `cB= |blacks|` just before. (Similar reasoning was used for `Inv3` at (d) in Version 2.)

(d) `Inv3` now depends on `oB<=cB` from the first half of the new assertion, and `cB<=nB` from the postcondition of the specification, which together give `nB==oB => cB==oB`. Then the second half of the new assertion can be used.

(e) `Inv4` is true because of the specification's postcondtion.

(f) Unchanged — The overall postcondition is true because of `Inv3` and the `until`-condition.

Global correctness:

The global-correctness arguments are unchanged.

So that establishes that Scanner Version 3 works; and all that's left is to implement the specification for approximate counting of blacks, since we have now established that approximate counting is good enough for checking the Scanner. It is the straight-

See Ex. 17.9.
See Ex. 17.10.

forward

```
# PRE cB<=|blacks|          ← precondition of specification above
nB= 0
for n in Nodes:
  if n.colour==B: nB= nB+1                                        (17.12)
# POST cB<=nB<=|blacks|  ← postcondition of specification above
```

All that's left now is to deal with the (impractical) atomicity of the Mutator.

17.11 Atomicity of the Mutator, in two steps

17.11.1 Step 1: the main part of the scanning loop

The very last issue we must deal with is that currently our Mutator swings one of the pointers in some node `s` ("source") to a new node `t` ("target") and blackens the target *at the same time*. That is, the Mutator's sole (and atomic) command is "swing and blacken", as follows:

```
# s,t are both reachable nodes.
s.left,t.colour= t,B                    # ...or s.right    ,      (17.13)
```

where the two assignments are done together. The issue is thus that, in order to avoid having to put those two assignments in a critical section, we have to break them into two *separate* assignments: either

```
t.colour= B                 or          s.left= t
s.left= t                               t.colour= B    .          (17.14)
```

The left alternative is "blacken then swing", and the right is "swing then blacken". (That is "and" has become "then".) At first sight, since the motivation for doing them both at once, i.e. atomically (as at 17.13), was to preserve the Scanner's inner-loop invariant

```
INV1: If |blacks|==oB then no B↦W's in scanned.    ,              (17.15)
```

it seems safer to blacken first, so that if indeed `t.colour` is changed from W to B then the antecedent `|blacks|==oB` of INV1 just above is falsified *before* the subsequent swing falsifies the consequent.

Yet that turns out to be a mistake, as the original designers of this algorithm discovered the hard way and described as follows:

> " To keep invariant that there are no B↦W's in the scanned nodes, during the marking cycle, we at first made our Mutator atomic.
>
> Encouraged by this success, we then tried to keep the invariant with a *non*-atomic Mutator, that is one in which the Mutator's action was split into two separate atomic operations: one for redirecting the pointer and another for blackening the new target. To be sure of maintaining that invariant, the Mutator had to blacken the target first, and only then swing the pointer.
>
> *This non-atomic "finer grained" solution –although presented in a way sufficiently convincing to fool ourselves– contained a bug...* [7] "

See Ex. 17.11.

[7] This quote is freely adapted from the article by Dijkstra et al. (with italics added). The bug was discovered by Stenning and Woodger; an analogue for our current solution is shown in App. F.2.

The bug is essentially that if the target is blackened *before* the pointer is swung *and then* an entire whiten-scan-collect process is carried out... when control is returned finally to the Mutator and it actually swings the pointer –the second of the two assignments it must carry out– its target will have been changed to W by the initial "Whiten everything." phase of the new scan (instead of remaining black as it was in the previous scan). And so the Mutator might have created a B↦W in `scanned` after all. App. F.2 sets out the sequence in detail.

See Ex. 17.12.

The only only alternative remaining is the right-hand, "swing then blacken". Before we reason about it, we make a further restriction, and then an observation:

See Ex. 17.21.

(a) RESTRICTION: We assume there is only *one* Mutator. (But there are extensions of this approach that work for multiple Mutators.) The practical implication of that is that we can now assume that the Mutator's interference always alternates `swing`, `blacken`, `swing`, `blacken`,... In particular, there are never two `swing`'s in a row.

(b) OBSERVATION: When the (single) Mutator carries out a `swing`, the (new) target must be reachable. (Remember "reachable" in (a) of Sec. 17.5, and the pre-condition of Prog. 17.13.) That means that if the target `t` is W (which is the "problem case"), there must be a B↦W step somewhere on the path to `t` from a root node. [8]

But if the swing occurs during the inner Scanner loop, where `Inv1` holds, that B↦W cannot be already scanned (when `|blacks|==oB`): thus it must be unscanned, and so the Scanner's completing its inner loop will blacken it eventually, even if the Mutator doesn't — forcing a re-scan because `|blacks|` will have increased. This observation is what "saves the day".

We won't be completely rigorous from this point on, because the detail seems to be quite intricate and so would require very careful checking indeed (such as an automated program verifier might help with). [9] However...

See Ex. 17.13.

For (a) we will introduce a label-like Boolean auxiliary `P` (for "poised") that usually `False`, but becomes briefly `True` just (and only) when the Mutator has swung but not yet marked. That is, the Mutator (17.14, right-hand alternative) becomes

```
# Atomic multiple assignments, with auxiliary label-Boolean P.

{ not P  and s,t both reachable }
s.left= t      # Swing.

{ P }
P: t.colour= B # Blacken.
{ not P }    .
```
(17.16)

Our new version of the original INV1 (17.15), taking advantage of `P` and the OBSERVATION above, is now this `Inv1A`:

```
# This Inv1A replaces Inv1 from (17.15).
|blacks|==oB ⇒
      no B↦W in scanned
  or  P and t.colour==W and B↦W in unscanned    ,
```
(17.17)

[8] It's reachability of `t` that implies the existence of a path from a root node to it. It doesn't matter which root node it is: what matters is that the root node is B yet `t` is W.

[9] See van de Snepscheut's treatment (on which this presentation is based) for complete rigour.

and we first look at its local correctness in the Scanner.

The first thing we must check is that `Inv1A` holds at the beginning of the inner scanning loop: since `scanned` is empty there, `no` B↦W `in scanned` holds trivially.

The second thing we check is that it suffices for `no` B↦W `anywhere` when the inner loop ends with `|blacks|==oB`. And at that point it's `unscanned` that is empty, leaving `|blacks|==oB` ⇒ `no` B↦W `in scanned` — but now `scanned` is `anywhere`. It's not *quite* trivial in this case, but still convincing.

What's left for *local* correctness is to check that `Inv1A` is invariant, i.e. is *maintained* by the inner loop. But the only update the inner loop can make is to change some node's colour from W to B, which falsifies `Inv1A`'s antecedent. So the invariance is pretty clear too — *except* for the hidden transfer at the very end of the loop of the current node `n` from `unscanned` to `scanned`. We will come back to that tricky transfer in Sec. 17.11.2 below: we leave it for now [10] and turn to *global* correctness of `Inv1A`: we must check *both* statements in the Mutator, and because of the removed atomicity they must be checked *separately*. For the `swing` we have as precondition `not P` and `t is reachable` , and from `Inv1A` itself (because of `not P`) we have `|blacks|==oB` ⇒ `no` B↦W `in scanned`. Afterwards, however, we will have `P`, and so we must show that `swing` under these circumstances establishes as well

```
|blacks|==oB ⇒
      no B↦W in scanned
  or  t.colour==W  and  B↦W in unscanned    .
```

Now if `t.colour==`B, i.e. the swing is to B, then `no` B↦W `in scanned` is preserved; but if the swing is to W then, because of `swing`'s precondition that `t` is reachable, there must be a B↦W somewhere on the path from a root to `t` and, from our precondition for `swing` in this context, that B↦W must be in `unscanned`. (That was not trivial.)

For the other step in the Mutator we again assume `Inv1A`, but now also `blacken`'s precondition `P` [11] — and then `t.colour` is set to B and `P` to `False`. In the case that `t.colour==`B already, the blackening has no effect, preserving `Inv1A` trivially; if however `t.colour==`W , then blackening `t` will falsify `Inv1A`'s antecedent `|blacks|==oB`, again preserving it trivially.

17.11.2 Step 2: moving from one node to the next in the scanning loop

This last step (deferred from † in Sec. 17.11.1 above) turns out to be the trickiest one: when the current node `n` has been dealt with, i.e. at the end of the `for`-loop body, it is removed from `unscanned` and added to `scanned`. How do we know that local correctness of `Inv1A` is preserved by *that*?

There's a obvious risk, because although the Scanner sets both `n.left.colour` and `n.colour.right` to B, by the time the end of the `for`-loop is reached a Mutator swing could have set (say) `n.left.colour==`W, i.e. *after* the Scanner's assignment `n.left.colour=` B. And although we know from global correctness –in spite of the Mutator's interference– that `Inv1A` still holds before the transfer, the transfer itself could invalidate `no` B↦W `in scanned` (because `n`, set to B↦B and about to be added to `scanned`, has been changed to B↦W by a Mutator swing); or the

[10] By "hidden" is meant that it happens, but (in Python) there is no source text like "`choose the next n for the loop`" that can be annotated with assertsions.

[11] Recall our comment above about checking the two statements in the Mutator separately: we are not *assuming* that this `blacken` occurs immediately after a `swing`: in principle, there could be two `blacken`'s in a row. To avoid having to consider that here, in practice, is the main reason for introducing `P`.

transfer could invalidate B↦W `in unscanned` (because `n`, about to be removed from `unscanned`, is the very B↦W that `Inv1A` needs to be there).

More precisely, the hidden "transfer `n` from `unscanned` to `scanned`" does not maintain `Inv1A` on its own (a failure of local correctness). We need to add some assertions to the scanner's inner-loop body, in effect to "save" `Inv1A` from falsification caused by the transfer.

See Ex. 17.14.

To do that, we formulate and establish that a *stronger* condition than just the invariant `Inv1A` itself holds just before the transfer: we add assertions `{ n.-.colour==B }` to the loop-body to capture the effect of the node-blackening it performs: referring to Prog. 17.11, we see that it becomes (in part)

```
{ Inv1A }
if n.colour==B:
  n.left.colour= B
  { Inv1A } { n.left.colour==B }
  n.right.colour= B
  { Inv1A } { n.right.colour==B }    .
```
(17.18)

But the problem now is (of course) that those new assertions, while locally correct, are not globally correct: they can be falsified by the Mutator. And so we must weaken them, to make them globally correct, while nevertheless keeping them strong enough to establish the condition that Ex. 17.14 established we need just before the transfer.

See Ex. 17.14.

A nice technique of finding *how* to weaken assertions so that they become globally correct is to experiment with how they are weakened by the interference they suffer (in this case, from the Mutator). In Ex. 17.15 we show how to do that rigorously. For now, however, we will justify the transfer of `n` quite informally, so that we can concentrate on the principal ideas that help *systematically* (Ex. 17.15) to find the assertions that a truly careful check requires.

See Ex. 17.15.
See Ex. 17.16.

We begin our informal discussion by assuming that program control is "positioned" just before the implicit transfer of `n` from `unscanned` to `scanned`, and we reason as follows:

(a) If both `n.left.colour` and `n.right.colour` are B, then the invariant is clearly unaffected: a B↦W is neither added to `scanned` nor removed from `unscanned` by the transfer of `n`: it is B↦B, not B↦W.

(b) If however (say) `n.left.colour` is W, then it must have been swung there by the mutator *since* it was set to B by a scanner swing. When that happened, however, it must have been B↦B and so could not *itself* have been the B↦W asserted to be in `unscanned` — that "justifying" B↦W must be some other.

(c) And that B↦W must be still there (in `unscanned`) because, if it was changed to B↦B by a colour-assignment, then `|blacks|` would have increased; but if it was changed by a second swing (the only other possibility), the Mutator must have carried out a mark in between and so increased `|blacks|`.

(d) Because the justifying B↦W is still there in `unscanned`, at the end of the loop, and is not `n` itself, it is therefore safe to transfer `n` from `unscanned` to `scanned`.

See Ex. 17.18.
See Ex. 17.19.

```
# -- Scanner (Version 4): Set all reachable nodes to B

# Initialisation.
for n in nodes: n.colour= W # Whiten everything.
for n in roots: n.colour= B # Blacken roots.

# Propagation.
nB= 0
repeat
  oB= nB
  for n in nodes:
    # Inv1 and Inv2
    if n.colour==B:
      n.left.colour=  B
      n.right.colour= B
  nB= 0                                  } → count blacks
  for n in Nodes:                        }
    if n.colour==black: nB= nB+1         }
  # Inv3 and Inv4
until nB==oB
# POST No B↦W's anywhere.

# Collection
for all n in nodes:
  if n.colour= W:  # Add n to free list.
    n.left= F
    F= n

# Inv1A: See (17.17).
# Inv2:  |blacks|>=oB
# Inv3:  if nB==oB then no B↦W's anywhere
# Inv4:  |blacks|>=nB
```

(17.19)

The specification of the count (from Prog. 17.11) has been replaced by its implementation (17.12); and the auxiliary `cB` has been removed, because it is no longer needed.

Figure 17.8 Scanner Version 4: final version

17.12 The completed program

The final version of the Garbage Collector is given in Fig. 17.8; in Fig. 17.9 the propagation part of the Scanner is given alone, without assertions, to emphasise how much careful reasoning is required even for short programs — when they are concurrent.

See Ex. 17.17.
See Ex. 17.22.

```
# Propagation alone.
nB= 0
repeat
  oB= nB
  for n in nodes: # Propagate B.
    if n.colour==B:
      n.left.colour=  B
      n.right.colour= B
  # Propagation complete: did anything change?
  nB= 0 # Recalculate nB.
  for n in Nodes:
    if n.colour==black: nB= nB+1
until nB==oB # ...until nothing changed.
```

(17.20)

Figure 17.9 Scanner propagation alone, without assertions

17.13 *Exercises*

Exercise 17.1 (p. 148) Give an example of the circular-structure problem of Prog. 17.3 in Python.

Exercise 17.2 (p. 148) A garbage-collection method is *safe* if it never collects nodes that are still in use, i.e. are not actually garbage. Is mark-sweep safe?

Is reference-counting safe? If you believe reference counting is safe, state an invariant that establishes its safety beyond any doubt.

See Ex. 17.17.

Similarly, a garbage-collection method is *complete* if it collects all garbage: none left behind. Is reference-counting complete?

Is mark-sweep complete when the Mutator is paused? Is it complete "on the fly"? *Hint*: See Fig. 17.4.

Exercise 17.3 (p. 151) Why is the problem not made *more* complex by having to carry out what was originally one instance of operation (b) by four instances of other operations?

Exercise 17.4 (p. 152) The `c= W` just before the `repeat` in Prog. 17.4 is clearly not necessary, because `c= B` is executed immediately after. So we can (and will) remove it. But why is it useful for now?

Exercise 17.5 (p. 152) Is "All white nodes are garbage." the same as "All garbage is white." or not? If not, does it matter?

Exercise 17.6 (p. 152) Check carefully that the variant is strictly decreased in the outer loop of Prog. 17.4.

Exercise 17.7 (p. 153) In Prog. 17.4 we did *not* assume that the step from evaluating the `if`-condition to the `if`-body was atomic (athough we did write it all on the one line). The assertion `{ n.left.colour==W }` after the `if` (and similarly for `right`) must therefore be checked for correctness. It is trivially locally correct; but it is obviously not globally correct, since the mutator might execute `n.left.colour= B` if `n.left.colour` were the target of its pointer swing. That being so, *we cannot leave that assertion in our program.* But what can replace it?

What assertion should be there? Would `{ True }` do? After all, can't matter that if `n.left.colour` is set to `B` when it's `B` already, surely. But if `{ True }` is good enough, why do we need the `if n.left.colour==W:` at all? See Ex. 17.8.

Exercise 17.8 (p. 153) Suppose we removed *both* `if`'s from the inner-loop code, so that the inner loop simply blackens everything. Wouldn't that be simpler? See Ex. 17.7.

In fact it's *too* simple: then the inner loop would run at most twice, after which the outer loop would terminate with all nodes black (and, trivially, all white nodes would be garbage).

What essential property would that trivial loop *not* have, and how do we prove that the actual loop has it? See Ex. 17.17.

Exercise 17.9 (p. 160) The specification

```
nB:[ PRE cB<=|blacks|, POST cB<=nB<=|blacks| ]
```

is easily implemented by the simple assignment `nB= cB`. And we have already checked that Scanner Version 3 works if that specification is met. Yet it clearly *cannot* work in that case, because it negligently declines to count anything at all. Or does it work after all...

Can you clear that up?

Exercise 17.10 (p. 160) Check Prog. 17.12 for local correctness, and for global correctness with respect to the Mutator.

Exercise 17.11 (p. 161) The Owicki-Gries method requires each thread's assertions to be checked for global correctness, that is to be checked that they cannot be falsified by assignments in other threads. Most of the concurrency-related reasoning in this chapter –at least the subtler parts– has been doing just that, checking that the Mutator does not interfere (destructively) with the assertions in the Scanner/Collector.

But don't we have to check also that the Scanner/Collector does not interfere with the Mutator? Did we do that?

Exercise 17.12 (p. 162) Re-do the explanation of Woodger's Scenario in App. F.2, but with the Mutator (now) correctly swinging before blackening. Is the bug avoided?

Exercise 17.13 (p. 162) Give rigorous reasoning for Sec. 17.11. See Ex. 17.15.

Exercise 17.14 (p. 164) What we need is that `Inv1A` holds at the actual end of the `for`-loop, not only just before the transfer of `n`. Formulate a multiple-assignment statement to account for the (implied) transfer, and then use the Assignment Rule (Sec. B.1.1) to figure out what must hold at the *apparent* end of the `for`-loop, i.e. immediately before the hidden assignment you have just brought out into the open.

That will be needed for the actual rigorous check asked for in Ex. 17.16. See Ex. 17.16.

Exercise 17.15 (p. 164) To weaken the assertion `{ n.left.colour==B }` so that it becomes globally correct, we imagine the effect that the Mutator might have on it, and *disjoin* (i.e. `or`) that to the original assertion because –after all– perhaps the Mutator did not interfere, allowing the original assertion to persist. Write a single assertion `?1?` so that both

$$\begin{array}{rl} & \texttt{\{PRE n.left.colour==B\} swing \{POST ?1?\}} \\ \textit{and} & \texttt{\{PRE n.left.colour==B\} blacken \{POST ?1?\}} \end{array} \tag{17.21}$$

hold. Make it as strong as you can, without being too complicated. And then check to see whether

$$\texttt{n.left.colour or ?1?} \tag{17.22}$$

is globally correct. If it is, you are done; if not, apply the same procedure to (17.22) as you did to (17.21), this time to discover a `?2?`, and check whether

$$\texttt{n.left.colour or ?1? or ?2?} \tag{17.23}$$

is globally correct.

Hint: Just doing it once might indeed not be enough. But twice, that is (17.18), should achieve (local and) global correctness. Note also, during the second check, that if `P` holds then a `swing` cannot occur: that simplifies things a lot. (Recall (14.4) from the explanation of global correctness in Sec. 14.5, and look carefully at the assertions in the Mutator at (17.16).)

See Ex. 17.14.

Exercise 17.16 (p. 164) Show that the inner-loop assertion you discovered in Ex. 17.15 entails your answer to Ex. 17.14.

Exercise 17.17 (p. 165) We have shown that the on-the-fly garbage collector here is safe, since all white nodes are garbage. But is it complete: is all garbage white?

If not, figure out whether it is possible for a piece of garbage to remain forever uncollected. If not, for how long can a piece of garbage escape collection?

See Ex. 17.19.

Exercise 17.18 (p. 164) Can you find a "hole" in the informal argument given in Sec. 17.11.2?

Exercise 17.19 (p. 168) Can you fix the "hole" you discovered Ex. 17.18 with a more careful, but still informal argument?

See Ex. 17.18.

Exercise 17.20 In Sec. 17.11.2 there were four "Why?" questions. Can you answer them here? Remember that there is only one Mutator.

See Ex. 17.21.

Exercise 17.21 (p. 168) In fact the counter-example in Fig. 17.3 does not apply if there is only one mutator: it uses two pointer-swings in succession and, between the two, the Mutator must have blackened its target and the Scanner will "notice" it, forcing a re-scan. The counter-example in Fig. 17.4 however still applies.

Give the simplest counter-example you can for our program 17.20's being *incorrect* if there are *two* mutators. *Hint:* You don't need many nodes for this.

Exercise 17.22 (p. 165) The Collector runs after the Scanner, and moves all W-nodes to the free list, which is a singly linked list starting with root node `F` (and using only one of its pointers). It might be something like this:

```
for n in nodes:
  if n.colour==W:
    n.left= F
    n.colour= B
    F.left= n
```
(17.24)

But the Mutator is still executing while this collecting going on: might it interfere with the code above?

Further, the Mutator might be acquiring a new node from the free list at the very same time as the Collector is adding a node to the free list: the Mutator's code (corresponding to "`malloc()`" in *C*) would be something like

```
newNode= F
F= F.left    .
```
(17.25)

Discuss (informally, perhaps) how these potential problems might be avoided.

Part IV

Checking programs automatically

Chapter 18

Machine-assisted program checking

18.1 Introduction and rationale

Up to this point, we have concentrated exclusively on by-hand checking of fairly small programs, though in some cases they are still quite complex — however small they might be. We have done that because, generally speaking, most programmers have not been taught the basic insights needed to see how correctness-checking can be done cheaply and effectively, nor have they been given the opportunity to see the improvements it brings not only in the quality of their programs, but in the satisfaction they experience from really, seriously understanding the good job they have done.

And being unaware of those techniques and insights is not the programmers' fault. It is the fault of their teachers: remember "The Lost Art".

One of the reasons that rigorous program checking was lost, became rarely taught, is that it was originally promoted in an all-or-nothing way. And the cost of rigour, the "all" option, in many cases was simply too high, too hard, too complicated; and so we were left with the "nothing". The *art* therefore is in knowing how to use a light touch, to operate *between* those two extremes, to concentrate just on the assertions you really need: write them informally at least; or even just be aware that they are there. That is what makes the difference, and that is in a nutshell what this text is intended to demonstrate: how astonishingly effective that can be.

Even so, it's easy to make mistakes in the *checking* process itself, even for small programs. (And that's one of the reasons that program-testing remains so important, whether you checked your program or not.) Who has not occasionally made mistakes in adding up a shopping bill by hand?

With larger, or more complex programs, indeed the risk of "checking-time" errors becomes too large to tolerate for programs whose correct functioning is vital, whose failure could lead to death or disaster. In those cases, we can use special-purpose programs to "check our checks", and in this Part we look at three styles of those checking-checkers. An analogy worth remembering is that even when we have learned the theory of differential equations, applying it to electrical circuits, or statics and dynamics applied to buildings and and bridges, we still use pocket calculators for doing the arithmetic.

We'll look at two styles of computer-assisted program-checking tools, picking for each a typical example of its type. The tools chosen here are not necessarily the best, nor even the most common or famous. The idea is simply to give you an idea of what's possible. Both are however are in serious use (at time of writing), when the expense is warranted.

18.2 Automated checking of assertions

18.2.1 Compile-time vs. run-time

The first tool is closest in style to what we have done in this text (but until now by hand). *Dafny* is a programming language, similar in style to Python or *C*, but with the important difference that the assertion statements in Dafny programs are checked *at compile time*. We discuss below how that is done — but we stress here that a Dafny program with an assertion that could fail at runtime cannot in fact be run at all: the compiler will refuse to produce code for it. Thus for example the Python program

```
x= 0
print("Hello!!!")
assert x==1
```

would *not* fail at runtime with an assertion error if it were processed in the Dafny style. Instead Dafny would produce a compile-time error, looking roughly like this: [1] Notice that the print statement is *not* executed, because the program is never run:

```
File assertionError.dfy
Line 3 character 1: assertion violation.
Dafny program verifier finished with 0 verified, 1 error.
```

On the other hand, if the program were processed entirely by Python, instead we would get something like this:

```
Hello!!!
Traceback (most recent call last):
  File "assertionError.py", line 3, in <module>
    assert x==1
AssertionError    ,
```

where we can see that the program *is* run, the `"Hello!!!"` *is* printed, and only then does `x==1` evaluate to `False` and give a run-time assertion error.

That's too late, if you're on your way to Mars.

[1] This output is edited from what Dafny actually produces, in order to make the point with as little syntactic distraction as possible. Below we will see actual examples.

```
                method BinarySearch(A: seq<int>, a: int) returns (n: nat)

precondition →      requires forall i,j:: 0<=i<j<|A| ==> A[i]<=A[j];
                    ensures n<=|A|;
postconditions →    ensures forall i: 0<=i<n   ==> A[i]<a;
                    ensures forall i: n<=i<|A| ==> a<=A[i];
                {   var low,high:= 0,|A|;
                    while (low!=high)
                        invariant 0<=low<=high<=|A|;                    ← "housekeeping"
loop invariants →       invariant forall i:: 0<=i<low    ==> A[i]<a;    ← A[:low]<a
                        invariant forall i:: high<=i<|A| ==> a<=A[i];   ← a<=A[high:]
loop variant →          decreases high-low;
                    {   var mid:= (low+high)/2;
loop body →             if (A[mid]<a) { low:= mid+1; }
                        else          { high:= mid;  }
                    }
                    n:= low;
                }
```

Figure 18.1 Dafny program

```
Running dafny BinarySearch.dfy
Dafny 3.2.0.30713
Dafny program verifier finished with 2 verified, 0 errors
```

Figure 18.2 Dafny checking of Fig. 18.1

18.2.2 Binary search in Dafny

The program text in Fig. 18.1 is "real" Dafny, our binary-search program from Sec. 3.3.3 but re-written using Dafny syntax. It's a "method" (rather than `def`), whose precondition (labelled `requires`) is that the sequence is in order, and whose postcondition (`ensures`) is what we discussed in Ex. 3.5. The loop `invariant` (three conjuncts) is given just after the `while` keyword, and the loop variant (`decreases`) is just after that.

See Ex. 3.5.

Binary search is small program, but is not trivial. Yet Dafny checks it *automatically*, with no further help from the programmer, and says that it verifies (Fig. 18.2). [2]

In Fig. 18.3 is Dafny's output if we introduce a deliberate mistake, replacing `A[mid]<a` by `A[mid]<=a`. For Fig. 18.4 we instead replace `low:= mid+1` by `low:= mid`;

[2] In *Programming Pearls* (1986, p36), Jon Bentley writes that professional programmers who were given a couple of hours to code binary search, in a language of their choice, at the end –in almost all cases– reported that they had found correct code for the task. But once they tested their programs (for about half-an-hour each), apparently some 90% found bugs.

Bentley goes on to say that Knuth in his *Sorting and Searching* (Sec. 6.2.1) writes that the first binary-search program was published in 1946, but the first published *bug-free* version did not appear until 1962.

(And that's leaving aside the famous overflow bug that was discovered later — see Ex. 18.1.)

```
Running dafny BinarySearch1.dfy
BinarySearch1.dfy(30,12):
  Error: This loop invariant might not be maintained by the loop.
Related message: loop invariant violation.
Dafny program verifier finished with 1 verified, 1 error.
```

Figure 18.3 Dafny checking of Fig. 18.1 with an incorrect test

```
Running dafny BinarySearch2.dfy
BinarySearch2.dfy(28,1):
  Error: decreases expression might not decrease.
Dafny program verifier finished with 1 verified, 1 error.
```

Figure 18.4 Dafny checking of Fig. 18.1 with an incorrect assignment

```
Running dafny BinarySearch3.dfy
BinarySearch3.dfy(30,12):
  Error:  This loop invariant might not be maintained by the loop.
BinarySearch.dfy(31,12):
  Error:  This loop invariant might not be maintained by the loop.
Dafny program verifier finished with 1 verified, 2 errors
```

Figure 18.5 Dafny checking of Fig. 18.1 with a missing precondition

and in Fig. 18.5 we leave off the requirement (i.e. precondition) that the sequence be sorted.

18.2.3 Limitations

To use Dafny effectively, one obviously must supply pre- and postconditions for the code that is to be checked — because, otherwise, how do we know what it's supposed to do? (Recall Sec. 1.3.) **One limitation**, therefore, is that the "requirements" cannot be written (only) informally, not at least if they are to be checked: Dafny does not understand English. The pre- and postconditions must therefore be written in *Dafny's* language of assertions, which is more or less standard predicate calculus (with some concessions to the use of a reasonable character set, for example `forall` rather than "∀"). This issue however is common to all computerised program-checking tools.

A more interesting question however is "How many of the *intermediate* assertions must the programmer supply?" The answer is "It depends." For anything non-trivial, the programmer is usually responsible for finding at least the loop *invariants* and adding them to the program text. In many cases, however, Dafny can find the *variant* itself. And assertions that can be hand-generated by working backwards through assignment statements and/or conditionals — they too can often be found by Dafny itself, so that they need not be written down: it's just substitution, and even a text editor can do that.

However the assertions are found –automatically or programmer-introduced– there does remain the problem of showing that one assertion implies another. In other words, Dafny transforms *your* problem of program correctness into *its* problem of proving theorems in logic, using essentially the rules of App. B for that transformation. [3]

And so a second, and major, limitation is here: Dafny cannot prove all truths even (for example) about simple arithmetic, no matter how much time you give it. Remarkably, it has been shown that *no* computer program (as we currently understand them) can be sure of finding all such proofs, i.e. neither Dafny nor anything else. [4] In spite of that, the scope of things that *can* be proved automatically is constantly being extended, and each advance there leads to "spin off" advances in the program-checking tools that use them.

18.2.4 The payoff

In spite of the limitations above, the overwhelming payoff of a "machine-checked check" is that you can trust it, provided you are clear about the assumptions you are making: that the requirements (pre- and post) are correctly formulated, that the hardware you are running your program on functions correctly, that the compiler you use to translate your program to machine code is itself error-free... And that might seem to be quite a few provisos.

But the main source of error in computer applications remains the bugs introduced by the programmer during coding, or inherent in (what turns out to be) an incorrect algorithm.

Those last potential errors are greatly reduced by (In-)Formal Methods, even more so if backed up by computer-assisted proof-checking. It really is worth it.

[3] The technique it uses for that is "satisfaction modulo theories" (SMT), which is an advanced topic worth reading about if you are interested in "under the hood" material.
[4] This is Gödel's (first) incompleteness theorem (1931).

18.3 Interactive program checking

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

18.4 Exercises

Exercise 18.1 The binary-search code in Fig. 18.1 contains `mid:= (low+high)/2`, the infamous binary-search "overflow bug", where even if `low` and `high` were within range of the computer's representation of integers (say $2^{31}-1$), the intermediate expression `low+high` might overflow.

Why did Dafny not flag that as an error?

Afterword

Most of the program-checking techniques explained here [5] were invented in the late 1960's and the 1970's by J-R Abrial, EW Dijkstra, RW Floyd, D Gries, CAR Hoare and S Owicki and –for a while– it looked like they would change radically the way in which programming was taught.

But they did not — and now, in general, these techniques are not taught at all.

Hence "The Lost Art".

Why are they not taught? What happened is that the sophistication required to apply Formal Methods "properly", even to small programs, was beyond the reach of most people who were still perfectly capable of becoming reasonably good programmers anyway. And that's what we've got: those are our IT Professionals today.

And there is a smaller group of people, those fascinated by the whole idea of Formal Methods and inclined to study it for its own sake; they became highly sought-after specialists and are nowadays the experts who develop, check and implement programs and systems whose failure would be too expensive to tolerate. They work for Apple, Microsoft, Google, Amazon, NASA, Trustworthy Systems... and although we might never see those people, we do depend on them. They not only check their programs, but they use *other* programs to "check their checks", to verify that they haven't made mathematical mistakes (Part IV). And some of them *write* those programs that check the checks. (And then those programs, that check the programs that check the checks, must themselves be checked: it really is "mathematics all the way down".)

This text however, the one you are reading, is for the rest of us. It "rescues" the methods lost in the '70's, lifts the bar for what "reasonably good" means, and shows today's programmers that they can use the experts' methods in an *informal* way, and that the result is not to make their programming harder, but actually *easier*. And more effective.

[5] ...and there are many techniques, more advanced, that we do not mention here.

Notes for teachers

Sketchy — just bullet points for now.

- Examples first — and only *then* the rules that make them work. The examples will build the intuition for "what's going on", and that intuition will help the students understand the reason the rules make sense.

  The rules for conditionals, i.e. for `if`-statements, are particularly sensitive to being introduced at the right moment: more sensitive than loops, in my opinion. The *rules* for conditionals are sometimes more complicated than the increased confidence they actually achieve. Until then, let the students think of them as atomic — and entice the clever ones to wonder "If there's a rule for assignment, and for loops, why isn't there one for conditionals?" Make them *want* it before you give it to them.

- Present conventional programs as well as their celebrity-cousins,[6] and use invariants for *both*. That diminishes the "novelty" of assertion-based reasoning and so reinforces the idea that it can be used very widely, even for everyday programs.

- Give examples of actual runs. Python is widespread, and easy to use: tempt the students to try out their own assertion-based ideas by actually programming them and running them. Do not however get into input and output (in Python): it's a distraction. Keen students will do that for themselves anyway — and one must absolutely avoid any suggestion that we are talking about "Programming in Python".

- Try not to talk about "verifying" and "correctness". Try not to use those words at all. Students already know that they must check their programs: it's just that they are often doing it in an inefficient way. Pretend you are teaching them a better way to do what they already do, rather than advising them to discard something they have learned and replace it with what you claim is better.

  Early-stage programmers don't do verifying/proving at all, at least not by that name. Many of them will be unfamiliar with (and wary of) proofs of *anything*, of *any* kind. "Proving a program" doesn't make sense, not at all, from that perspective.

  Instead, they *check* their programs: stick to that nomenclature. It's what they do already, although they haven't learned yet how to do it in any other way than "executing the program in their heads", tearing their hair out because they can't find the bug and then, finally, choosing good test cases.

  Those are good beginnings. This material is to take them further.

[6] The "celebrity cousins" are the programs that you've never heard of, very tricky to program even though usually quite short, that are used by FM proselytisers to dazzle their audience. It's a mistake: it gives the impression that FM is just for fancy stuff.

Appendices

Appendix A

Drill exercises

A.1 Sequences and operators on them

A *sequence* `A` has some length `N>=0` and (in Python) is indexed from `A[0]` inclusive to `A[N]` exclusive: thus the elements are `A[0]`, `A[1]`,... `A[N-1]`. Usually all the elements have the same type. (That's not always true; but we will assume it here.)

Python's subsequence notation `A[l:h]` means (again in the in/ex-clusive style) `A[l]`, `A[l+1]`,... `A[h-1]`, and there are various conventions for "default" range values: a missing `l` is 0; and a missing `h` is `N`. Just `A` on its own is the same as `A[0:N]`, and `[]` is the empty sequence. All but the first element (the "tail") is `A[1:]`.

Python allows also *negative* sequence indices, which "count backwards from the end" so that for example `A[-1]` is `A`'s last element and `A[:-1]` is all but the last element. [1]

Now speaking mathematically for a moment: for associative binary operators on the sequence's type, like say $(+)$ for numbers, there are corresponding general operators that act on a whole (sub-sequence) at once, like $(\sum)$. The drill exercises below are about that. Some of the mathematical operators are implemented directly in Python (like $(+)$ that becomes `+` of course, and $(\sum)$ becomes `sum()`.. And some are not: but that doesn't stop us from using them in assertions — as long as we know what they mean.

Assume in the exercises that all indices have "sensible" values: thus in 1. for example we assume `0<=l<=h<=N`.

Drill A.1 Suppose that `A` has length `N`.

1. How long is `A[l:h]`?
2. Simplify $\sum$`A[l:m]` + $\sum$`A[m:h]`.
3. Simplify $\sum$`A[l:m]` + `A[m]`.
4. How many elements does `A[n:n]` have?
5. What is $\sum$`A[n:n]`?
6. Why is it sensible to write it just `[]`, without mentioning `A`?

[1] Note however that "out of order" indices in Python give the empty sequence, not an error, as do subsequence selectors that are out of bounds. Since the conventions in other languages might differ, it's best not to rely on those behaviours.

7. If for all `0<=n<N` we have `A[n]==n`, what is $\sum$`A`?
8. How many elements does `A[:]` have?
9. Write $\prod$ for the product of all elements in a sequence. With the same values for `A` as in 7., what is $\prod$`A`, where by `A` alone we mean "the whole thing `A[0:N]`"? What is $\prod$`A[1:]`?
10. Are `[]` and `A[0:0]` and `A[1:1]` and ... `A[N:N]` all equal? What is $\prod$`A[n:n]`?

Drill A.2 Continue to assume we are working over sequences of numbers, and that mathematically we have both positive and negative infinity, i.e. both ∞ and $-\infty$. Write MAX and MIN for the whole-sequence generalisations of the binary operators max and min.

1. Simplify `MAX A[l:m]` max `MAX A[m:h]`.
2. Simplify `MAX A[l:l]` max `MAX A[l:h]`.
3. Given your two answers just above, what must `MAX A[l:l]` be? And therefore `MAX []`?
4. Repeat 1.–3. for MIN.

A.2 Substitution

Substitution is the key operation for checking whether assignments work. But in these drills we practise substitutions on their own. (See Sec. E.2.1.) Each of the following exercises should be done in two stages: first, perform the substitution *exactly as it is*, putting additionally parentheses $(\cdots)$ around the new text. Second, use "ordinary" reasoning to simplify the result. Here are two examples, written in the ordinary mathematical style:

1. Substitute 1 for x in $x+1$.[2] *Answer:* First $(1)+1$ and then simplify to 2.
2. Substitute $y+1$ for x in $2x$. *Answer:* First $2(y+1)$ and then (if you want to) simplify to $2y+2$.

Drill A.3 (p. 8) Do these substitutions, using Python syntax, and simplify them if you can. When simplifying, assume `x` is an integer.

1. Substitute `x+1` for `x` in `x==0`.
2. Substitute `x//2` for `x` in `x==2`.
3. Substitute `2` for `x` in `x==2`.
4. Substitute `2` for `x` in `0==x`.
5. Substitute `x-y-1` for `y` in `x-y>=3`.
6. Substitute `x*x-2*x+1` for `y` in `y==0`.

[2] You could also say "Replace x by 1 in $x+1$.

7. Substitute `x*x-3*x+2` for `y` in `y==0`.
8. Substitute `t` for `y` in `x==Y and y==X`.
9. Substitute `y` for `x` in your answer to 8.
10. Substitute `x` for `t` in your answer to 9.

A.3 Invariants

Drill A.4 Suppose that `A[0:N]` is an array of integers.

1. What would be a good invariant for a loop that summed `A` from low-to-high index? (Make up your own variable names, but keep them simple.)
2. What would be a good invariant for loop that summed `A` from high-to-low index?
3. What would be a good invariant for loop that found the largest element in `A`, assuming `A` is not empty?
4. What would be a good invariant for loop that found the smallest element in `A`, assuming `A` is not empty?
5. What postcondition should be the last statement in your your program to 1. ,2. ?
6. What precondition should be the first statement in your your program to 3. ,4. ?
7. What would be a good invariant for a loop that set Boolean `b` to whether integer `a` occurred in `A`?
8. What would be a good invariant for a loop that set integer `n` to the index of the first occurrence of `a` in `A`, if there is one? (And what would therefore be a good value for `n` when there is no `a` in `A`?)
9. What would be a good invariant for a loop that found the first element of `A` that was greater than the *average* of all elements before it? Would that program need a precondition?

A.4 Variants

We will re-use the questions from Sec. A.3, concentrating this time on the variant.

Drill A.5 Suppose again that `A[0:N]` is an array of integers. Remember that variants (for now) must be integer-valued and either strictly decreasing or strictly increasing (but not a mixture) on each loop iteration — and they must be bounded below or above in the direction they are moving.

1. What would be a good variant for the loop in Question 1. of Drill A.4 that summed `A` from low-to-high index?
2. What would be a good variant for the loop in Question 2. of Drill A.4 that summed `A` from high-to-low index?

3. What would be a good variant for loop in Question 3. of Drill A.4 that found the largest element in `A`, assuming `A` is not empty? Do you need the precondition for your choice?

4. What would be a good variant for loop in Question 4. of Drill A.4 that found the smallest element in `A`, assuming `A` is not empty?

5. What would be a good variant for a loop in Question 7. of Drill A.4 that set Boolean `b` to whether integer `a` occurred in `A`?

6. What would be a good variant for a loop in Question 8. of Drill A.4 that set integer `n` to the index of the first occurrence of `a` in `A`, if there is one?

7. (Harder) What would be a good variant for a loop that set integer `n` to the index of the first occurrence of `a` in `A` when it is known that there is one? The program's precondition would be `{`PRE "Element `a` occurs in `A`."`}` and the loop entry would probably be `while A[n]!=n: ···`.

   You must be sure to establish that the variant is bounded (probably above).

A.5 Propositions, sets and predicates

In these drills we will use mainly Python syntax for logic (as elsewhere in the main text) except for "implies" ⇒.

Drill A.6 State for each of the following implications whether it is `True` or `False`:

1. `True` ⇒ `True`
2. `True` ⇒ `False`
3. `False` ⇒ `True`
4. `False` ⇒ `False`

Drill A.7 It's a simple rule of elementary set theory that for two sets A, B their intersection $A \cap B$ is a subset of each of the two sets A, B separately: that is, we have both $A \cap B \subseteq A$ and $A \cap B \subseteq B$.

But is that still true when A and B have no elements in common: that is when their intersection is empty (i.e. they are *disjoint*)? If yes, say why; if no, give a counter-example.

Drill A.8 Consider the statement (in English) "If A is true then also B is true." where A, B stand for any claims at all. In logic (e.g. in an assertion) we would write it $A \Rightarrow B$. Which of the following are equivalent to it?

1. $B \Rightarrow A$
2. `not` A `or` B
3. `not` $B \Rightarrow$ `not` A
4. A `or` B
5. `not` $A \Rightarrow$ `not` B
6. A `and` B

Drill A.9 For each of the following statements (in English), give the correct logical equivalent. (You might need "`not`" in some cases.)

1. if `A` then `B`.
2. `A` if `B`.
3. `A` unless `B`.
4. `A` if and only if `B`.
5. `A` just when `B`.
6. `A` only if `B`.
7. `A` iff `B`.

Drill A.10 State for each of the following implications whether it is `True` for all values of integers `x,y`:

1. `x<=y and y<=x` ⇒ `x==y`
2. `x<y and y<x` ⇒ `x==y`
3. `x<y and y<x` ⇒ `x!=y`
4. `x<y or y<x` ⇒ `x!=y`

Note that the first three are `and` but the last is `or`.

Drill A.11

Consider the statement "If you can't see my mirrors, then I can't see you.". Which of the following are equivalent statements?

1. "If I can't see you, then you can't see my mirrors."
2. "You can see my mirrors or I can't see you."
3. "If I can see you, then you can see my mirrors."
4. "You can't see my mirrors, or I can't see you."
5. "If you can see my mirrors, then I can see you."
6. "You can't see my mirrors, and I can't see you."

A.6 Thinking outside the box

Drill A.12 When people were figuring out how to reason with negative numbers, a question arose about multiplying two negative numbers. If $-1 \times 2 = -2$, i.e. multiplication by -1 takes $+2$ down to -2, shouldn't multiplication of an (already) negative number by -1 go "even more negative"?

Give a *rigorous* argument that the product of two negative numbers should be positive. You can use any general algebraic facts about *non-negative* numbers in order to do so.

A.7 Hoare triples "in the small"

Drill A.13 Which of the following Hoare triples are correct? For those that aren't, explain why not.

1. `{ x==0 } x= x+1 { x==1 }`
2. `{ x==1 } x= x+1 { x==0 }`
3. `{ True } x= x+1 { x==1 }`
4. `{ True } x= x+1 { x==0 }`
5. `{ False } x= x+1 { x==1 }`
6. `{ False } x= x+1 { x==0 }`

Drill A.14 Fill-in the missing assertions. Simplify them arithmetically if you can; but do the substitution first.

1. `{ ??? } x= x+1 { x==1 }`
2. `{ ??? } x= x//2 { x==4 }`
3. `{ ??? } x= x*x-2*x+1 { y==0 }`
4.
```
{ ??? }
y= x*x-3*x+1
{ y==0 }
```
5. `{ ??? } x= x*x+1 { y==0 }`

6.

```
{ ??? }
x= x+y
{ ??? }
y= x-y
{ x==Y and y==X }
```

7.

```
{ ??? }
x= x+y
{ ??? }
y= x-y
{ ??? }
x= x-y
{ x==Y and y==X }
```

8.

```
{ ??? }
s= x
{ ??? }
s= s+y
{ ??? }
s= s+z
{ s== x+y+z }
```

9.

```
{ ??? }
p= 1
{ ??? }
p= p*x
{ ??? }
p= p*y
{ ??? }
p= p*z
{ p== x*y*z }
```

A.8 Hoare triples in the large(r)

Drill A.15

Here is an initialised loop, with some assertions:

```
{ pre }
init
{ Inv }
while test:
  { test and Inv }
  body
  { Inv }
{ post }
```
(A.1)

Assume the assertions are correctly placed, which means that if the Prog. A.1 starts in a state where *pre* is `True`, then every time the program's flow of control reaches an annotation, that annotation will be `True`. What are the implications and Hoare triples that therefore must be correct?

Drill A.16 Here's the same loop as in Drill A.6, but with different assertions: [3]

```
{ A }
init
{ B }
while test:
  { C }
  body
  { D }
{ E }
```
(A.2)

[3] That is, whatever *init*, *test* and *body* might be, they are the same in both cases: the programs are the same; the assertions are different.

Which of the following Hoare triples and implications must be correct in order for Prog. A.2 above to have been correctly annotated?

1. { A } *init* { B }
2. B `and` *test* $\Rightarrow$ C
3. B `and not` *test* $\Rightarrow$ E
4. { C } *body* { D }
5. D `and` *test* $\Rightarrow$ C
6. D `and not` *test* $\Rightarrow$ E

Drill A.17 Give invariants that could be used to annotate the following loops correctly for (partial) correctness. You may write the invariants in English if you wish, as long as they are precise and clear. You do not have to write out the loop again: just give the invariant.

Give also for each program a variant that could be used to establish its successful termination which, together with its partial correctness, establishes total correctness.

In each case `A` is a sequence of length `N`.

1.

```
n= 0
while n!=N and A[n]!=x:
  n= n+1
{ "If x is in A at all, then its first occurrence is at A[n]." }
```

2.

```
n,c= N,0
while n!=0:
  n= n-1
  if A[n]==x: c= c+1
{ "c is the number of occurrences of x in A." }
```

3.

```
n,l,e= 0,0,0
while n!=N:
  if A[n]!=x: e= 0
  else: e= e+1
  n,l= n+1,l max e
{ "l is the length of a longest run of consecutive x's in A." }
```
(A.3)

A.9 Whole-program drills

Drill A.18 Modify your invariant from Drill A.6(3.) to give an invariant for a program that establishes "`l` is the length of a longest run of almost-consecutive `x`'s in `A`." where a run has "almost-consecutive `x`'s" just when it never contains two or more consecutive non-`x`'s. For example, these are almost-consecutive `x` runs:

`[]`, `x`, `xx`, `y`, `xy`, `yx`, `xyx`, `yxx`, `xxy`, `yxyxyxyxyxy` ...

These are examples of runs that are *not* almost-consecutive `x`:

`yy`, `xyy`, `yyx`, `xxxxxxxxxxxxxxyyxxxxxxxxxxxxxx` ...

Drill A.19 Using your answer to Drill A.18, change *only* the if-statement in Prog. A.3 to make a program that establishes the postcondition of Drill A.18 above.

Drill A.20 Suppose `B` gummy bears (*gummibären*) are to be handed out to `C` children. If `C` divides `B` exactly, then each child should receive `B/C` (whole!) bears. But if the division is not exact, then some children will receive more bears than others; yet no two children should have bear-numbers more than one apart.

Let `bL` (for "low") and `bH` be $\lfloor$`B/C`$\rfloor$ and $\lceil$`B/C`$\rceil$ respectively, where $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are the floor- and ceiling functions from arithmetic. If every child receives only `bL` bears, then clearly `B` bears *will* be enough overall — but some bears might be left over. On the other hand, if every child receives `pH` bears, then `B` bears might *not* be enough. So the solution is to give each child c some (whole) number b_c of bears with `bL <=` b_c `<= bH`. But how do we decide when to "go low" (with `bL`) and when to "go high" (with `bH`)? [4]

Below is a program that does that.

```
b,c= B,C # Number of bears; number of children.
while c!=0:
  "Set r to some integer such that  floor(b/c)<=r<=ceil(b/c) ."
  "Hand out r bears to the next child."  # (A.6) below.
  b,c= b-r,c-1
# Each child gets between bL and bH bears; no bears left over.
```
(A.4)

We want to show that Prog. A.4 checks, that indeed

$$\text{all the bears are handed out; and} \tag{A.5}$$

$$\text{each child receives between } \texttt{bL} \text{ and } \texttt{bH} \text{ bears.} \tag{A.6}$$

(Remember that `bL` and `bH` are fixed, and are no more than 1 apart.)

Hint: Consider an invariant resembling [5]

```
      bL<=⌊b/c⌋ and ⌈b/c⌉<=bH
and   B == b + "the number of bears handed out already"
```
(A.7)

[4] Do you suffer from "indexitis"? Did you feel the urge to write "child c_i" and "`bL <=` b_{c_i} `<= bH`" here?
[5] You might have to alter it slightly...

Appendix B

Summary of rules for checking programs

In this section we gather all the checking rules for programs in one place, for easy reference.

Remember that one strategy for checking an already written program is to apply these checking rules to the smallest pieces of your program (the assignment statements, principally), and then use the combining rules –the rules for sequential composition, conditionals and loops– to work your way outwards.

On the other hand, if you are using these rules to help design programs, then the order is reversed: you figure out what the pre- and postconditions (and invariants) might be that would make your overall program work, and then you use these rules, working inwards, to fill in the smaller pieces.

If some of the checks are intricate, then it might help to refer to the rules in App. D for simplifying conditions.

B.1 The basic programs

B.1.1 Assignment statements

Assignment statements are checked by first doing a substitution (as if with a text editor), and then checking an implication: the assignment

$$\{\text{PRE } pre\}\ \texttt{x=}\ expr\ \{\text{POST } post\}$$

checks just if $pre \Rightarrow post'$, where the $post'$ is the original $post$ but with all occurrences of `x` replaced by $expr$. For multiple assignments, the replacements are done simultaneously.

Program 1.12 in Sec. 1.5 gives an example.

B.1.2 Do nothing — `skip`

The `skip` program literally does nothing, and is not often used in actual code. But –as for zero in artihmetic– it is helpful to understand how it is checked, and `skip` helps to understand other checking rules later on (for example the `else`-less conditional App. B.3.2, and multiple assertions on separate lines App. B.2.2). The rule is that

{PRE *pre*} `skip` {POST *post*}

checks successfully if *pre* $\Rrightarrow$ *post* .[1]

See p. 59 and p. 61 for examples.

B.2 Assertions

Sometimes its useful to write assertions without program fragments in between. Here is how they are checked.

B.2.1 Multiple assertions on one line

Two (or more) assertions `{` *assn1* `}` `{` *assn2* `}` on the same line are treated as single assertion `{` *assn1* `and` *assn2* `}` .

If the assertion is a postcondition, then both *assn1* and *assn2* must check from the statement before; If it's a precondition, then either or both *assn1* and *assn2* may be used to check the statement after.

B.2.2 Multiple assertions on successive lines

If assertions occur one after the other on separate lines, the it must be checked that each one implies the one that follows it. The very first one should be checked from the statement before; and the last one may be used to check the statement after. Thus

```
{ assn1 }
{ assn2 }      ← Imagine a skip here.
```

checks if `{` *assn1* $\Rrightarrow$ *assn2* `}`. It is exactly as if there were a `skip` in between, as in

```
{PRE assn1}
skip
{POST assn2}     ,
```

and you applied the rule from App. B.1.2 above.

A common pattern for multiple assertions mixes both arrangements: it is

```
{ inv } { not cond }
{ post }     ,
```

which is often found at the end of a loop. The required check, as explained above, is

inv `and not` *cond* $\Rrightarrow$ *post* .

[1] In Python `skip` is called `pass`.

B.2.3 Assertions for expressions: are they well defined?

When an expression (say in an assignment statement, or in a condition) could possibly cause an error, it must be checked that the precondition of the statement implies the expression to be defined. Thus for example use of division `n/d` requires that `d` is not zero, that is that `d!=0` occurs in (or is implied by) the precondition.

Another example is the use of `A[n]` indexing into to sequence `A[0:N]`, which requires `0<=n<N` in the precondition.

See Ex. 16.3 on p. 143 for examples.

B.3 Rules for checking larger program fragments

B.3.1 Sequential composition

Sequential composition of two programs *prog1* and *prog2* is the program that first executes *prog1* and then executes *prog2*.[2] That means that the postcondition of the first one must imply (or be equal to) the precondition of the second: thus the composition

{PRE *pre*}
prog1
prog2
{POST *post*}

checks just if there is an assertion *assn* "in between" so that

{PRE *pre*} *prog1* {POST *assn*}
and {PRE *assn*} *prog2* {POST *post*}

both check on their own. When used in annotating a program with assertions, instead of writing `{` *assn* `}` twice, as above, it can be written just once, between the two program fragments, as here:

{PRE *pre*}
prog1
`{` *assn* `}`
prog2
{POST *post*} .

See p. 8 for examples.

B.3.2 Conditional without `else`

The simple "no-`else`" conditional requires a separate check of its body, but with the precondition now including the `if`'s condition (which makes the check easier). It must also be checked that the precondition with the *negated* condition implies the overall postcondition directly: so

{PRE *pre*} `if` *cond*`:` *prog* {POST *post*}

checks just if both

{PRE *pre* `and` *cond*} *prog* {POST *post*}

[2] In Python, sequential composition is either newline or semicolon, In *C* it is just writing the two programs next to each other, because semicolons in *C* are mandatory.

checks, *and* the implication

pre `and not` *cond* $\Rrightarrow$ *post*

holds.

See p. 59 for examples, and Ex. 5.3.

B.3.3 Conditional with `else`

See Ex. 5.3.

The conditional without `else` (just above) is a special case of the conditional *with* `else`, but where the `else`-branch is actually `skip`. (This is an example of where `skip` is useful understanding other checking rules.) If the `else`-branch is *not* `skip` (or even if it is `skip`), then this rule is used:

{PRE *pre*}
`if` *cond*`:` *prog1*
`else:` *prog2*
{POST *post*}

checks if both

{PRE *pre* `and` *cond*} *prog1* {POST *post*}
and {PRE *pre* `and not` *cond*} *prog2* {POST *post*}

See Ex. 5.3.

both check.

See p. 60 for examples.

B.4 Rules for checking loops

In general, loops have two things to check: their partial correctness, that they establish the postcondition *if* they terminate; and that they actually do terminate. Partial correctness and termination are together called total correctness.

B.4.1 Partial correctness for `while`-loops

To check

{PRE *pre*}
`while` *cond*`:`
  body
{POST *post*} ,

an invariant must be found, a condition that is true before and after each iteration of the body. This was explained in Chp. 2, and finding those invariants was the subject of Chp. 3.

Thus to check the loop above we must find an condition *inv* and check it in these

three ways:

$$pre \Rrightarrow inv \qquad \texttt{\# The invariant is true initially.}$$

$$\left.\begin{array}{ll} & \{\text{PRE } cond \texttt{ and } inv\} \\ & body \\ & \{\text{POST } inv\} \end{array}\right\} \rightarrow \texttt{\# The invariant is maintained.} \tag{B.1}$$

and $inv \texttt{ and not } cond \Rrightarrow post$
`# The invariant and the negated condition`
`# imply the postcondition.`

See Ex. B.1.

B.4.2 Termination for `while`-loops

To check termination of `while` *cond*`:` *body*, we find must an integer-valued variant expression *expr* so that the loop body is guaranteed to decrease it, but not below zero. (Variants were the subject of Chp. 4.)

We have to show that

$\{\text{PRE } inv \texttt{ and } cond \texttt{ and } expr\texttt{==V}\}$
body
$\{\text{POST } \texttt{0<=}expr\texttt{<V}\}$

checks, where *expr* is an integer expression and `V` is an auxiliary variable (not otherwise used), and invariant *inv* checks as in Sec. B.4.1 just above.

For examples, see Chp. 4.

B.4.3 `while`-loops with `break` statements

Sometimes a loop can "terminate early", using a `break` statement. To check for example

$\{\text{PRE } pre\}$
`while` *cond*`1:`
  body`1`
  `{`*assertion*`}`
  `if` *cond*`2: break`
  body`2`
$\{\text{POST } post\}$,

that is when the *body* contains one or more `break` statements, the usual technique from Sec. B.4.1 is followed *except* that after the loop one can assume only the invariant and the negated guard *or* the assertions that held immediately before any of the `break` statements — *and they might not include the invariant.*

Thus (B.1) from above is modified so that the third check is

$$(inv \texttt{ and not } cond\texttt{1}) \texttt{ or } (assertion \texttt{ and } cond\texttt{2}) \Rrightarrow post$$

`# The invariant and the negated while-condition`
or `# the break condition(s)`
`# imply the postcondition.` (B.2)

Note that `break` statements need not decrease the variant.

B.4.4 Total correctness

Checking total correctness of a loop means checking its partial correctness and its termination. As we saw just above, that is usually done by checking partial correctness first, and then termination separately.

B.4.5 Partial correctness for `for`-loops

Particularly simple `while`-loops can be written as `for`-loops, with the initialisation, incrementing (or decrementing) and exiting the loop all "built in" — and the rules are as a result similar to the `while`-loop rules: in particular, you will need an invariant.

To check a (Python) `for`-loop of the form [3]

```
{PRE   inv("initial iterator") }
for e in "iterator":
  body
{POST  inv("complete iterator") }
```

we must check

```
{PRE   inv("iterator so far") }
body
{POST  inv("iterator so far and e as well") }
```

checks. A common example of an iterator is `range(N)` for some integer `N`.

Note however that (in Python) it is *not* true in general that *"the iterator is complete"* is true after the `for`-loop has ended. For example, if we are using `n` to iterate through the sequence `range(N)`, then we would have that

initial iterator corresponds to `n==0`, and

iterator so far to `n`, and

iterator so far and... as well to `n+1` and

complete iterator to `n==N`.

See Sec. 1.7 for some examples of `for`-loops. More detail about `for`-loops is given in App. G.2.

B.4.6 Termination for `for`-loops

A `for`-loop usually doesn't need a variant, because –as mentioned above– the exiting is built in. But we must be careful about the final value returned by the iterator. At the end of the (Python) `for`-loop described just above, we have `n == N-1`, not `n==N` as you might expect.

Even worse, if the range happened to be empty then the iterator might even be undefined once the loop has terminated (which it does immediately). Again, more detail about `for`-loops is given in App. G.2.

[3] Although `while`-loops are more or less the same in all languages, the `for`-loops might differ. See App. G.2.

B.5 Concurrency

Reasoning about concurrency (the subject of Part III) adds two ingredients to what we have already seen above: the `await` syatement, and checking *global* correctness.

B.5.1 The `await` statement

The `await` statement is used to make one thread wait until a condition is made true by some other thread. (If the condition is already true, then it does not have to wait, a special case.)

It is like an `else`-less `if`, but one that does not skip over its body if the condition is `False`: instead, it waits until the condition becomes `True`. The rule is that

{PRE *pre*} `await` *cond*: *prog* {POST *post*}

checks if

{PRE *pre* `and` *cond*} *prog* {POST *post*}

Just `await` *cond* is treated as `await` *cond*: `skip`. See Ex. B.4.

B.5.2 Global correctness and interference

In a concurrent setting, it must be checked that assertions in one thread are not *interfered with* by (assignment) statements in other threads. To check that `{` *assn1* `}` is not interfered with by `{` *assn2 prog* `}` in another thread, we check that the other thread cannot change *assn1* from `True` to `False`: we check

{PRE *assn1* `and` *assn2*}
prog
{POST *assn1*} ,

where the precondition *assn2* of the possibly interfering *prog* can be included in the check (if it helps).

As a convention, we indicate atomicity by whether several assignment statements are written on the same line: if they are, then assertions *between* them do not have to be checked for global correctness. Assertions between program components on separate lines *do* have to be checked.

Expressions on the right-hand side of an assignment, however, are usually *not* assumed to be atomic if they refer to two or more variables that can be changed by other threads. In our presentations we have not used special notations for that; rather we have pointed it out in the surrounding text, and in the final version of the program made sure that assignments refer to at most one non-local variable. See Ex. B.2.

B.6 Exotica: assertions, assumptions and specifications

There are three elementary programs that are used mainly in checking, but do not usually appear in the final code. They are used "temporarily" to make the program construction and its checking easier.

B.6.1 `assert`

The `assert` statement is used mainly, at least for us here, to express the connection between abstract and concrete datatypes (the subject of Part II), and to help to specify not-yet-written code. In particular, an `assert` is used to state a condition that "must be true at this point" if the program is to check (and thus work correctly). The rule is that

{PRE *pre*} `assert` *cond* {POST *post*}

checks just when

pre ⇛ *cond* `and` *post* .

`assert`-statements are sometimes available in programming languages, as executable code, where they are used for debugging and defensive programming. [4] In that case, an `assert` evaluates its condition, and halts the program (usually with a message) if the condition is not met.

B.6.2 `assume`

`assume`-statements are used only in specifying and checking: they are usually considered to be "non-compilable".

The statement `assume` *cond* "forces" the program to make *cond* hold at that point. If in fact the program does not do that, then it cannot be compiled to code: somehow the `assume` must be removed. In spite of that, it has a checking rule:

{PRE *pre*} `assume` *cond* {POST *post*}

checks just when

pre `and` *cond* ⇛ *post* .

See Ex. B.4.

The only way to remove an `assume` *cond* is to make sure (i.e. check carefully) that the preceding code makes *cond* true already.

B.6.3 Specification `x:` [*pre*, *post*]

A specification is like an `assert` followed by an `assume`, and is used to describe the effects of "not yet written" code in a way that the surrounding program can be checked now, rather only after that "not yet written" code *has been* written. The rule is

{PRE *pre*} `x:` [*pre1*, *post1*] {POST *post*}

checks just when

pre ⇛ *pre1* `and` (∀`x`· *post1* ⇒ *post*) ,

See Ex. B.3.

where an "empty ∀", that is no `x` given, can simply be removed.

A very common case is when *pre* and *pre1* are the same, and *post* and *post1*, in which case there is nothing to check. But –like `assume` statements– in general specifications cannot be compiled. The only way to "get rid" of them is to supply the missing code.

[4] Python has `assert`-statements.

B.7 *Exercises*

Exercise B.1 (p. 195) Explain why with Sec. B.2.2 above it's sufficient to be able to check `while`-loops with just one rule:

```
{PRE inv}
while cond:
  body
{POST inv and not cond}
```

if `{PRE` *cond* `and` *inv*`}` *body* `{POST` *inv*`}`.
For examples, see Chapters 2 and 3.

Exercise B.2 (p. 197) Why is it reasonable to assume that "only one non-local variable" on the right-hand side of an assignment allows the statement to be considered atomic? What if that non-local variable appears more than once on the right-hand side? What if it appears on the left- and on the right-hand side?

Exercise B.3 (p. 198) What is the difference between `x:[`*pre*`,`*post*`]` and

```
assert pre
assume post    ?
```

Exercise B.4 (p. 198) What is the difference between `assume` *cond* , that is with no variable `x`, and `await` *cond* , that is with no following statement?

Appendix C

Data refinement: the absolutely primitive rules for encapsulated data-types

The following collection of steps describes exactly what you can do to an encapsulated data-type while preserving its correctness. Put together in the right way, they describe replacing abstract- by concrete encapsulations, imposing a data-type invariant and transforming a whole program: all of three techniques we illustrated in Part II can be expressed in some combination of these steps.

C.1 Adding auxiliary variables

You can add a (new) variable `c`, say, and code that manipulates it, in any way you like — provided the old code's behaviour is not affected. In particular:

(a) The new code (concrete) terminates whenever the existing code (abstract) would.

(b) The new code does not assign to any already-there variables "`a`" in a way that that would introduce dependencies on `c`, and there can be no `c`-controlled conditionals or loops that contain assignments to `a`.

C.2 Imposing a (data type) invariant

You can impose any data-type invariant *inv* you like, and it may refer to any variables that only the encapsulation can alter. [1]

Do it by

(a) Adding `assume` *inv* to the end of the initialisation.

(b) Adding `assert` *inv* at the beginning of every externally accessible procedure or function.

(c) Adding `assume` *inv* at the end of every externally accessible procedure or just before every `return` of a function.

[1] In fact as a special case you can just add a data-type invariant anytime, without changing any variables.

C.3 Removing auxiliary variables

Auxiliary variables can be removed at any time *except* for `assume` statements that contain auxiliary variables — to remove those, you must show (by other means, using context as in Sec. C.4(e) below) that the assumed condition is identically `True`. But they must actually *be* auxiliary, occurring only in assignments and conditionals that do not affect other variables.

C.4 Other manipulations

In addition, there are other things you can do to your code at *any* time, i.e. not necessarily within an encapsulated data-type:

(a) You can weaken any assertion `assert`.

(b) You can strengthen any assumption `assume`.

(c) You can reduce nondeterminism, for example replacing `m:[l<h,l<=m<h]` by `m= (l+h)//2`.

(d) You can replace equals by equals (using context provided by assertions).

(e) You can add or remove whole assertions/assumptions provided context makes their conditions `True`.

C.5 An example: procedure `add` from `Set`

We return to the `add` example from Fig. 10.4. It is "syntactic sugar" for

```
class Set(N)
  local { N>=0 } ss= {}; assume |ss|<=N
  def add(s):
    assert |ss|<=N                                          (C.1)
    ss= ss ∪{s}
    assert |ss|<=N # ← See Exercise C.1.
    assume |ss|<=N    ,
```

where the blue statements are the de-sugaring of the data-type invariant `# INV |ss|<=N` in Fig. 10.4. The other statements, including the `assert |ss|<=N`, are explicitly part of the program.

See Ex. C.1.

We can immediately simplify the program to

```
class Set(N)
  local { N>=0 } ss= {}; assume |{}|<=N
  def add(s):
    assert |ss|<=N
    assert |ss ∪{s}|<=N
    ss= ss ∪{s}    ,
```

where we have first used the ←'ed `assert` to justify removing the `assume`, and then moved the `assert` before the assignment to `ss`, using substitution. We can then simplify it further by removing the `assume |{}|<=N`, which is where we use the class

precondition `N>=0`, and the `assert |ss|<=N` because it is weaker than the assert that immediately follows it. That gives

```
class Set(N)
  local { N>=0 } ss= {}

  def add(s):
    assert |ss ∪{s}|<=N
    ss= ss ∪{s}    .
```

We now add a auxiliary variables `ar` and `n` to get

```
class Set(N)
  local { N>=0 } ss= {}
  local ar,n= [0]*N,0

  def add(s):
    assert |ss ∪{s}|<=N
    ss= ss ∪{s}
    i= find(s)
    if i==n<N: ar[n],n= s,n+1

  local def find(s):  ···   ,
```

where the `<N` condition is necessary to make sure that the added code for the new auxiliaries is terminating: without it, there could be an index-out-of-bounds reference into `ar`. Note that the new (auxiliary) code does not affect in any way the "real" code that manipulates `ss`.

Now however we add a data-type invariant to link them together: it is `ss = set(ar[:n])` and, when distributed throughout the code (i.e. again "de-sugaring" the data-type invariant) we get

```
class Set(N)
  local { N>=0 } ss= {}
  local ar,n= [0]*N,0;
  assume ss == set(ar[:n])

  def add(s):
    assert ss = set(ar[:n])
    assert |ss ∪{s}|<=N
    ss= ss ∪{s}
    i= find(s)
    if i==n<N: ar[n],n= s,n+1
    assume ss = set(ar[:n])

  local def find(s):  ···   .
```

The first assumption can be removed, because the assignments just before reduce it to `assume {} == set(ar[:0])`, whose condition is identically `True`.

The extra test `n<N` can also be removed, because `n<N` is guaranteed by the postcondition of `find` (not shown here) and the two assertions at the beginning of `add`. That

gives

```
def add(s):
  assert ss = set(ar[:n])
  assert |ss ∪{s}|<=N
  ss= ss ∪{s}
  i= find(s)
  if i==n: ar[n],n= s,n+1
  assume ss = set(ar[:n])    ,
```

in which the final `assume` can be removed, since its condition is guaranteed to be `True` by the initial `assert`'s. Once they have allowed us to remove that `assume`, those `assert`'s themselves can be removed, because we are always allowed to remove `assert`'s. We now have

```
def add(s):
  ss= ss ∪{s}
  i= find(s)
  if i==n: ar[n],n= s,n+1    ,
```

in which the variable `ss` has become auxiliary. So –our final step– it can be removed as well, leaving

```
def add(s):
  i= find(s)                                    (C.2)
  if i==n: ar[n],n= s,n+1    ,
```

See Ex. C.2.

our concrete code. Notice that this *can* get a subscript-out-of-bounds error.

C.6 Getting your improved datatype to compile

Call the data-type invariant *dti*.

Recall from Sec. C.2(a) that you can add an `assume` *dti* at the *end* of the initialisation, but that (unlike (b) for procedures) you are not allowed to add a compensating `assert` *dti* at the *beginning* of the initialisation.

Any `assume`'s in your code (at the end of the procedures, or anywhere else in fact) cannot be compiled, however: the only way to remove them is to show (by reasoning) that their conditions are `True` whenever they are reached. Usually that is done by treating the `assert ··· assume` as a precondition-postcondition pair, i.e. a specification for the procedure body and checking that the procedure is correct. An alternative (but equivalent) approach is to distribute each `assume` forwards, through earlier code step-by-step, until it comes to rest just after some statement(s) that establish its condition trivially: and one such statement could be an `assert` of the very same condition, as we have seen in examples.

For the `assume` *dti* at the end of initialisation, the same applies *except* that there is (possibly) no helpful `assert` *dti* sitting there unless it was there already: in Sec. C.2(a) we were forbidden from adding it. And so if we find that that `assume` (or indeed any other) cannot be removed by reasoning, we have made our code uncompilable (but not incorrect).

The only way out of that impasse is that data-type precondition (which is an `assert`) might imply the `assume`'s condition: and then indeed you can remove the `assume`. But if the precondition needs to be strengthened, then *you need to ask*

permission to do so — it is a social issue, because strengthening the precondition of *any* code makes it less applicable than before. Strengthening the precondition of your data type might invalidate checks already done of other code that use it.

C.7 *Exercises*

Exercise C.1 (p. 202) What would happen if we left out the marked `assert |ss|<=N` in Prog. C.1?

Exercise C.2 (p. 204) The code in Prog. C.2 will get an index-out-of-bounds error if it is called when `n==N` and with an `s` that does not already occur in `ar`. How can such code be correct?

Appendix D

The arithmetic of conditions

D.1 Introduction and rationale

We have so far seen "conditions" in two places mainly: the first is as Boolean-valued, `True` or `False` expressions that control whether the `then`- or the `else`-branch of an `if` statement is taken, or whether a `while`-loop is (re-)entered. Readers who have got this far are very familiar with those uses already. Such conditions are executed as the program runs, and contribute to the answer the program finally gives.

But we have also seen conditions as assertions: preconditions, postconditions, invariants, or sometimes just "`#` —" -labelled comments between statements, or simply between braces "`{` — }". Those conditions are *not* executed as the program runs: their only (but crucially important) role is to help us to *understand* the answer the program finally gives — and to increase our confidence that the answer is correct. See Ex. D.1.

In this appendix we will make the connection between "conditions" in our programs and mathematical logic as it really is, i.e. concentrating on their second use: making sure that programs are correct; and helping to write them in the first place.

Sometimes these conditions –soon to be called called "formulae"– have to be extracted from the program text and thought about on their own, a bit like grabbing a bit of scrap paper to do a complicated bit of arithmetic on the side, perhaps arising out of your tax return. (That has happened in a number of exercises.) Your copied-out calculations are carried out in a different state of mind: you forget temporarily *why* you need to know what 15% of \$1234.56 is (but you continue to resent having to do it). Instead you are focussing only on getting the right answer. See Ex. 8.5. See Ex. 8.6. See Ex. 10.5. See Ex. 16.8. See Ex. 17.5.

When you do that with logic, you are in effect using scrap paper for conditions rather than for numbers, for programs rather than for tax returns. Again, it does not matter *why* you need to simplify that condition: what's important is that you do it right.

That is what we are studying here.

D.2 Why is my program correct?

We return to this well-known program fragment for swapping two variables, which we first saw at p. 58(h):

```
t= x
x= y
y= t    .
```

As we all know, the effect of that program is that `x` finally will have the value that `y` had initially, and similarly for `y`. But in what sense can we *prove* that to be true? And why do we bother?

To answer the second question first: for a program of this size, and indeed one you have written many times, we probably wouldn't bother with a proof. But for bigger programs, becomes much more valuable to be able to prove that bits of them are doing what they should; it's a double-check while coding that might save you hours of debugging later.

Now for the first question: we have seen that we can show that a program "works" by inserting carefully chosen comments that simply say (with a *condition*) what is supposed to be true at that point in the program. For the program above, we introduce (auxiliary) variables `X`,`Y` to stand for the initial values of `x`,`y`, and we add a comment, at the very beginning of the program, that shows clearly what `X` and `Y` are for; and then, at the end of the program, we insert another comment that says that those values have been swapped. Thus the comment at the beginning says "*Here*, let `x`,`y` be some `X`,`Y` respectively." And the comment at the end says "In that case, *here* we will find that they have been swapped, that now `x==Y and y==X`."

Note that the conditions in the assertions/comments are not "executed", that there is no assignment going on there. They merely state "what is true" (or what we *hope* is true) at that point in the program. It's the program's assignment statements that "do stuff", and the program's comments that "assert stuff". And if those comments are right (about the program), then indeed the program swaps those two variables.

Of course there will be other comments in your program (one hopes); and they might be of the form "This is why I did it this way." or "The following steps do more or less as follows." They are necessary, but do not contribute *directly* to proof: only the "This is true here." comments do that.

How do we know the comments are right? That's what we did earlier with the "how to check parts of your program" techniques we have explained already (Sec. 5.2 and Sec. B.1.1). We wrote

See Ex. D.2.

```
{PRE x==X and y==Y}
t= x
{ t==X and y==Y }
x= y
{ t==X and x==Y }
y= t
{POST y==X and x=Y}
```
(D.1)

Each comment-like assertion says what's true at that point. But how do we know the assertion is right? We'll look at just one step in the program, and apply a bit of magic:

```
⋮
{PRE t==X and y==Y}
x= y
{POST t==X and x==Y}
⋮
```

The "magic" is that you can be sure the comments are right if you get the initial comment by carrying out the assignment on the final comment: that is the `x` in `x==Y`

is replaced by y because of the assignment x= y, giving y==Y. (The other half of the condition has no x, so it is left alone.) Remember that that you "go backwards", from the after-comment to the before-comment. See Ex. D.3.

It's as simple as that.

D.3 How do I write my program in the first place?

Remember our exponential-calculating program that sets p to the value B**E, where B was for "base" and E was for "exponent"? A straightforward solution was Prog. 3.8, reproduced here:

```
# PRE 0<=E
p,e= 1,0
while e!=E: { Inv: p==B**e }
  p,e= p*B,e+1
  { p==B**e and e==E }
# POST p==B**E    .
```
(D.2)

But the much faster, logarithmic-time exponential program turned out to be Prog. 3.9, this one:

```
# PRE 0<=E
p,b,e= 1,B,E
while e!=0: { Inv: B**E == p*b**e }
  if e%2==0: b,e= b*b,e//2
  else:      p,e= p*b,e-1
{ B**E == p*b**e and e==0 }
# POST p==B**E    .
```
(D.3)

Do you remember how logical calculations helped us to *write* this program?

Forget that you have seen the solution Prog. D.3 already (just above). And suppose you have realised that in the special case where E is a power of 2, say E==2**N for some N, it would be enough to set p to B initially, and then to square it N times. This is a typical starting point for thinking about this problem (and where we all begin with writing a program to solve it). But getting the details right is tricky. And getting the details *wrong* consumes nights and weekends as you try to debug what was your best guess. Details like these...

- What do you do when E==0? There is no 2**N for that. Would your (not yet written) program just go into an infinite loop, dividing 0 by 2 again and again, forever?
- What do you do when E!=0 but still is not a power of 2? You'd have to fiddle something... But what? And how, exactly?

If you brushed those worries aside (temporarily), you might as your first step get as far as the incomplete program here:

```
p,e= B,E
while e!=1: # But what if E was zero?
  if e%2 == 0: p,e= p*p,e//2
  else: # And what do I do here?
```
(D.4)

Now what? This is where logic, and our "what's true here" comments, helped us to get our weekends back. Instead of guessing Prog. D.4 as just above, take your first step instead to a *different* incomplete program that at least handles `E` correctly even when it is zero:

```
# What do I do here, to make B**E = p*b**e true unconditionally?
while e!=0:
  if e%2 == 0:
    e= e//2
    # What do I do here, to make B**E = p*b**e true again?
  else:
    e= e-1
    # What do I do here, to make B**E = p*b**e true again?
  else:
```

In this program it's *already* clear that it will terminate (eventually `e` will reach zero) even if you're not sure what to do with `p` and `b`. So that's one problem solved: no more worries about `E==0` initially.

And now... we introduce a "what's true here" comment to identify the so-called invariant `B**E = p*b**e` of the loop. Remember that it's called "an invariant" because it is true just before the loop condition is checked, every single time and whether or not the loop is entered; and it remains true no matter how many times the loop iterates. That means it must be true the first time (which is therefore the job of the loop initialisation) and it is true the last time (and so describes whether the loop has accomplished).

So, finally... What then is the role of the formulae, and the "magic"? It's that, with them, you can use the substitution technique from Sec. 5.2 (or Sec. B.1.1) to check that your "what's true" comments are correct. And you can do it mechanically, almost without thinking. (And there are computer programs that can check them for you — mostly.)

Finding those comments can be hard, however, particularly for a tricky program (or a sneaky one — not quite the same thing). But if you use logic, it's much easier to be sure you have the right ones.

See Ex. D.2.

D.4 *Calculating* with conditions: let's start with sets

We begin our discussion of logic with a discussion of something much simpler: elementary set-theory.

Sets contain *elements*, and $x \in S$ is how we write that element x is contained in set S. Similarly we write $x \notin S$ for the opposite. Thus "$\in$" means "is an element of".

That's all there is to it — for example we have $2 \in \{1,2,3\}$ and $0 \notin \{1,2,3\}$.

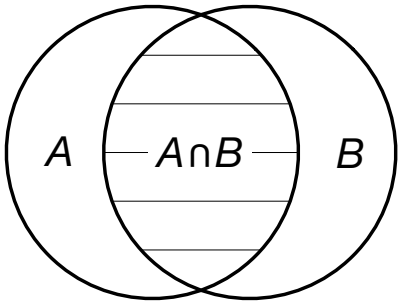
What more could there be?

Not too much, actually, at least for our purposes here. But sometimes we have to "calculate" with sets — usually to figure out when two sets written in *different* ways are actually the *same* set. It's more than just $\{1,2,3\} = \{3,2,1\}$ however; a better example is $A \cap B = B \cap A$, where "$\cap$" is set intersection, and is defined by

$$x \in A \cap B \quad \text{just when} \quad x \in A \text{ and } x \in B \ . \tag{D.5}$$

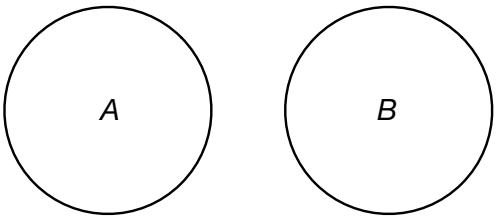
So we can easily see that $A \cap B = B \cap A$ holds, because "$x \in A$ and $x \in B$" means the same as "$x \in B$ and $x \in A$" . It's a "set fact". We can also see it from Fig. D.1, because

See Ex. D.3.



The intersection $A \cap B$ is the lens-shaped region in the middle above, filled with horizontal stripes $\equiv$. It is obvious that it lies inside of the circle of set A (and similarly B).

Figure D.1 Intersection of sets: a Venn diagram



Where is $A \cap B$ now?

Where do we put the $\equiv$ lines now? Since there is no picture corresponding to the intersection of A and B, we cannot "see" whether it is inside of A (or B).

Figure D.2 Intersection of *disjoint* sets, i.e. with no elements in common

the drawing is symmetric.

Here are two other common notations used with sets:

$(\cup)$	— union	$x \in A \cup B$	just when	$x{\in}A$ or $x{\in}B$
$(\subseteq)$	— containment	$A \subseteq B$	just when	$x{\in}A$ implies $x{\in}B$

Using those, we can write another "set fact" — it is $A \cap B \subseteq A$. But how would we prove this second fact (or indeed prove the first one)? Actually, *why* do we have to "prove" it? Isn't it obvious from the "Venn" diagram in Fig. D.1?

Maybe it isn't. What about when A and B have no elements in common? The corresponding figure (Fig. D.1 becomes Fig. D.2) is not much help now — and *that's* why we need proofs. [1]

But in this case the proof is simple: it is

$$
\begin{array}{ll}
 & x \in A \cap B \\
\equiv & \text{"by definition of } A \cap B\text{"} \\
 & x{\in}A \ \wedge \ x{\in}B \quad [2] \\
\Rrightarrow & \text{"since } \mathcal{A} \wedge \mathcal{B} \Rrightarrow \mathcal{A} \text{ for any propositions } \mathcal{A}, \mathcal{B}\text{"} \\
 & x{\in}A \ ,
\end{array}
$$

so that $A \cap B \subseteq A$ by definition of $(\subseteq)$ from (D.5) above. (For now, think of "$\equiv$" as "if and only if (iff)" and "$\Rrightarrow$" as "implies".)

But what are these "propositions" $\mathcal{A}$ and $\mathcal{B}$? And what do we mean by $\mathcal{A} \wedge \mathcal{B} \Rrightarrow \mathcal{A}$? And how do we know that the implication is valid?

In general –and that is the point of this introduction– you can usually find a set-like analogue for your reasoning steps in logic (as just above). And it does help with the intuition. But over time it becomes tiresome and indeed unreliable to think in terms of little circles and how they overlap, or don't: [3] you would rather be writing programs. And getting them right.

That's what logic is for.

D.5 Simple calculations in logic

Here we will use the usual symbols from logic, rather than their Python equivalents. Thus `not` is $\neg$ and `and` is $\wedge$ and `or` is $\vee$.

Logic is the arithmetic of Computer Science. In this section we just look at how it works, and how you can use it. (Later you can study *why* it works, if you want to.) It's a bit like differentiation $\mathrm{d}/\mathrm{d}x$. You don't need all those epsilons and deltas to do it: you just follow the rules. *Later* (and only if your interests incline that way) you can study *why* the differential calculus works.

Similarly, only if your interests incline that way do you need to know *why* logic works. Here we are just looking at what it is and how to "follow its rules".

Our main aim is to be able to work out whether logical statements are correct or incorrect: [4] it is quite easy (with some practice), and furthermore is fundamental to computer programming. This is a "fast forward" introduction help you get started.

Indeed you don't need to learn very much at this point: your skill will increase "by osmosis", as you get more and more used to reasoning carefully about your programs. Referring to a more comprehensive list will help; [5] but you don't have to learn them all by heart right from the start! Just knowing that they are there is enough.

We start with "propositional" logic, and we present it of course in the context of programming — where there is a "program state" containing "variables" that have "values", and there are "functions" that you can apply to them.

[1] EW Dijkstra once began a lecture by showing a slide with nothing on it at all. "This is a graph," he said. "It is the *empty* graph."
The two sets above have empty intersection: but where is it in the picture?

[2] The symbol "$\wedge$" means `and` ; the symbol "$\vee$" (not yet used here) means `or` .

[3] Think of an engineer who brings Cuisinaire Rods to work every day, to help him with his arithmetic, or –worse– counts using fingers.

[4] The technical terms in logic are "valid" and "invalid". We'll stick with correct and incorrect.

[5] There is one in App. E.

$$
\begin{array}{c}
0 \\
x \\
x{+}1 \\
\log x \\
\sin(\pi/2) \\
(a{+}b) \times 3!
\end{array}
$$

Figure D.3 Some terms

D.6 Terms in logic are like expressions in programming

Terms in logic are like what you find on the right-hand side of assignment statements in programs: they are built from variables, constants and functions. Thus x on its own is a term (it is a variable); and 1 is a term (it is a constant); and $x{+}1$ is a term (it is formed by applying the function $+$ to the two terms x and 1). [6]

The state, which maps variables to values, is what determines the values of terms: one speaks of a term "having some value" in a state. For example, in a state that maps x to three, the term x has the value three (trivially), and 0 has the value zero (in every state, in fact: that is why it is called a constant). And $x{+}1$ has the value four, because in that state x has value 3 and in *every* state (by convention) we know that "+" means addition and "1" means one.

Our *variables* will have short lower-case *italic* (maths font) names, drawn from the Roman alphabet. Our *constants* will have their usual mathematical symbols, like 0 and π. (The real number constants e and i will not cause trouble.) And $\mathbb{R}$ is a set-valued constant, meaning "all the real numbers".

Our *functions* will have their usual mathematical names too, like square root $\surd$, plus $+$, and factorial $!$. Some of those take one argument ($\surd$ and $!$), some take two ($+$), and the position of the arguments can vary: sometimes the function is written before its argument ($\surd$), sometimes between its arguments ($+$), and sometimes after its argument ($!$). The number of arguments a function takes is called its arity. See Ex. D.9.

We often need to introduce new functions, of our own, just for a particular problem. For those, the syntax is more regular: they will have short lower-case sans-serif names, in the Roman alphabet. Their arguments follow them, separated by spaces.

With all that, we can show how terms are constructed in general: a *term* is either

(a) a constant; or

(b) a variable; or

(c) a function applied to the correct number of other terms, depending on its arity.

Figure D.3 lists some terms.

[6] In the earlier parts of this twxt we would have used "programming font" here, so that `x` would be a variable (and a term too) and `x+1` a term (but not a variable). Here we will move to the more general mathematical conventions.

false
$1 < (a \div 2)$
$(x+1) = 7$
even 6
$\pi \in \mathbb{R}$

Figure D.4 Some simple formulae

D.7 Simple formulae are like (in)equalities in programming

Simple formulae [7] are built from terms and "predicate symbols". The best-known predicate symbols represent [8] the binary comparison relations from arithmetic: $(<)$, $(=)$, $(\leq)$ etc. Like functions, predicates have an arity; for binary relations, the arity is two. Again like functions, predicates are applied to terms.

Unlike functions, a predicate applied to (the correct number of) terms is not another term: it is a *simple formula*. Simple formulae do not have general values like terms; instead, they are either "True" or "False" in a particular state.

See Ex. D.8.

But here we must make a careful distinction for these Booleans. In our programs we have been writing `True` and `False` for the values that Booleans (or Boolean-valued expressions) can take: they are terms (as we saw in Sec. D.6 above), and we will continue to write them that way. But for the predicate versions of "true" and "false", the first selecting all states and the second selecting none, we will write true and false. That is, `True` and `False` are terms (denoting) the Boolean values that terms can take; but true and false are predicate symbols (of arity 0).

For conventional predicates (like binary relations) we use the usual notation. Predicate symbols that we introduce ourselves will be short Roman sans-serif and their arguments will follow them, separated by spaces (as for our introduced functions). Thus for example true and false are predicates of arity 0.

See Ex. D.9.

Figure D.4 lists some simple formulae. In any particular state (that assigns values to terms), a simple formula is either "True" or "False". To summarise so far: "`True`" is a Boolean value that a term can denote in a given state; "true" is a predicate symbol that holds for all states; and "True" is what a formula can be in a particular state. If you evaluate the formula in that state and get `True` as your answer, then that formula is True in that state. [9]

[7] They are called *atomic* formulae in the logic literature.
[8] Note the bit of pedantry here: the predicate symbols "represent", not "are" relations. The same applies to constants vs. constant symbols and functions vs. funcgtion symbols.
[9] An alternative to "yet another version of 'true' " would be to say that the formula "holds" in that state, or "does not hold". Experience shows however that however carefully you try to capture these nuances in different fonts, different words etc. it always takes a while before the ideas settle down.
In the end, though, it all becomes clear anyway, and it doesn't matter so much what words you use. Like riding a bicycle, once these distinctions are clear in you mind, you cannot imagine how they could ever have been confusing. And you never forget.

D.8 Propositions, and propositional formulae

Propositional formulae are built from simple formulae, using propositional connectives — that is, we regard simple formulae as *propositions*, and propositional formulae are made connecting them together. The connectives are $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implies), and $\Leftrightarrow$ (if and only if, or iff).

The expressions either side of `and` are called "conjuncts"; those either side of `or` are called "disjuncts".

(As nouns, they are conjunction, disjunction, negation, implication and equivalence. In our programming language they were `and`, `or`, `not`, (nothing) and `==`.) [10]

Except for $\neg$, all have two arguments, written on either side; the single argument of $\neg$ is written after it.

Like simple formulae, propositional formulae are either True or False, once given a state. If, for example, $\mathcal{A}$ and $\mathcal{B}$ are propositional formulae, then the propositional formula $\mathcal{A} \wedge \mathcal{B}$ is True in a state exactly when both $\mathcal{A}$ *and* $\mathcal{B}$ are True in that same state. That is summarised in this table:

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \wedge \mathcal{B}$
True	True	True
True	False	False
False	True	False
False	False	False

.

A complete set of "truth tables" for the five connectives is given in Figure D.5. In a formula $\mathcal{A} \Rightarrow \mathcal{B}$, the subformula $\mathcal{A}$ is the *antecedent*, and $\mathcal{B}$ is the *consequent*. See Ex. D.12. See Ex. D.11.

Following convention, we allow the abbreviation $a < b < c$ (and similar) for the propositional formula $a < b \wedge b < c$.

Figure D.6 gives some propositional formulae. See Ex. D.10.

D.9 Operator precedence

Strictly speaking, a term like $2 + 3 \times 4$ is ambiguous: is its value fourteen or twenty? Such questions can be resolved by parentheses, that is $2 + (3 \times 4)$ vs. $(2 + 3) \times 4$, but they can be resolved also by general precedence rules. The usual rule from arithmetic is that $\times$ is done before $+$ and we say that $\times$ has *higher precedence*.

We adopt all the usual precedence rules from arithmetic, adding to them that functions have highest precedence of all: thus $\sqrt{4} + 5$ is seven, not three. When several functions are used, the rightmost is applied first: thus $\log \sin(\pi/2)$ is zero. [11] We do not require parentheses around function arguments; but note that $\sin \pi/2$ is zero, whereas $\sin(\pi/2)$ is one.

In propositional formulae, the precedence is (highest) $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ (lowest).

[10] We explain in Sec. D.10.1 below the difference between the "$\equiv$" and "$\Rrightarrow$" that we have been using earlier, and the "$\Leftrightarrow$" and "$\Rightarrow$" that we are using now.
[11] Without higher-order functions, the reverse does not make sense anyway.

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \wedge \mathcal{B}$
True	True	True
True	False	False
False	True	False
False	False	False

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \vee \mathcal{B}$
True	True	True
True	False	True
False	True	True
False	False	False

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \Rightarrow \mathcal{B}$
True	True	True
True	False	False
False	True	True
False	False	True

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A} \Leftrightarrow \mathcal{B}$
True	True	True
True	False	False
False	True	False
False	False	True

$\mathcal{A}$	$\neg\mathcal{A}$
True	False
False	True

Note the way that "True" and "False" are being used here. For example, the first line of the top-left truth table says that if some formula $\mathcal{A}$ is True in a particular state, and some (other) formula $\mathcal{B}$ is True in that same state, then the formula $\mathcal{A} \wedge \mathcal{B}$ is True in that same state.

Figure D.5 Truth tables for propositional connectives

$$\begin{gathered}
\mathsf{true} \\
x^2 = -1 \\
(x \leq y) \wedge (y \leq x + 1) \\
(x > 0) \Rightarrow (x + y \neq y) \\
(0 \leq p < q) \Rightarrow (0 < q) \\
(n! = n) \Leftrightarrow (n{=}1) \vee (n{=}2)
\end{gathered}$$

Figure D.6 Some propositional formulae

D.10 Calculation with logical formulae

D.10.1 Relations between formulae

The two (simple) formulae $x{=}y \Rightarrow x{\neq}z$ and $x{=}z \Rightarrow x{\neq}y$ are equivalent in this sense: in every state they are both True or both False together. In general, that two formulae $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* is written $\mathcal{A} \equiv \mathcal{B}$, and means

$$\text{In every state, } \mathcal{A} \text{ is True if and only if } \mathcal{B} \text{ is True} \quad .$$

And that is indeed the same as saying "In every state, $\mathcal{A} \Leftrightarrow \mathcal{B}$ is True." But there is an important difference between $\equiv$ and $\Leftrightarrow$. The first is a relation between formulae: $\mathcal{A} \equiv \mathcal{B}$ is a statement about the two separate formulae $\mathcal{A}$ and $\mathcal{B}$; it is *not* a formula itself. The second is a propositional connective: writing $\mathcal{A} \Leftrightarrow \mathcal{B}$ says nothing about any relationship between $\mathcal{A}$ and $\mathcal{B}$; rather it *is* a formula itself.

Here are two other relations between formulae. The statement $\mathcal{A} \Rrightarrow \mathcal{B}$ means

$$\text{In every state, if } \mathcal{A} \text{ is True then } \mathcal{B} \text{ is True} \quad .$$

That is the same as "In every state $\mathcal{A} \Rightarrow \mathcal{B}$ is True." And the statement $\mathcal{A} \Lleftarrow \mathcal{B}$ means

$$\text{In every state } \mathcal{A} \text{ is True if } \mathcal{B} \text{ is True} \quad .$$

It is the same as "In every state $\mathcal{B} \Rightarrow \mathcal{A}$ is True." Officially, the relation $\Rrightarrow$ between formulae is known as *entailment*, although in everyday speech we often say "implies" (and will continue to do so).

Those three relations are used to set out chains of reasoning like this one: for any formulae $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$,

$$\begin{array}{ll} & (\mathcal{A} \Rightarrow \mathcal{C}) \vee (\mathcal{B} \Rightarrow \mathcal{C}) \\ \equiv & \text{"writing implication as disjunction"} \\ & (\neg\mathcal{A} \vee \mathcal{C}) \vee (\neg\mathcal{B} \vee \mathcal{C}) \\ \equiv & \text{"associativity,commutativity of } \vee\text{"} \\ & (\neg\mathcal{A} \vee \neg\mathcal{B}) \vee (\mathcal{C} \vee \mathcal{C}) \\ \equiv & \text{"De Morgan, idempotence of } \vee\text{"} \\ & \neg(\mathcal{A} \wedge \mathcal{B}) \vee \mathcal{C} \\ \equiv & \text{"writing disjunction as implication"} \\ & \mathcal{A} \wedge \mathcal{B} \Rightarrow \mathcal{C} \ . \end{array}$$

Each formula is related to the one before it by the relation $\equiv$, $\Rrightarrow$, or $\Lleftarrow$. (Above however they are all three $\equiv$.) And each step between formulae carries a decoration, a "hint", suggesting why it is valid. The quotes "—" separate the hints from the proof itself. They are not part of the proof; they are *about* the proof.

See Ex. D.13.

The relation $\equiv$ between formulae is *transitive*, which means that whenever both $\mathcal{A} \equiv \mathcal{B}$ and $\mathcal{B} \equiv \mathcal{C}$, then we have $\mathcal{A} \equiv \mathcal{C}$ too. That is why the chain of equivalences above establishes overall that the first formula is equivalent to the last:

$$(\mathcal{A} \Rightarrow \mathcal{C}) \vee (\mathcal{B} \Rightarrow \mathcal{C}) \quad \equiv \quad \mathcal{A} \wedge \mathcal{B} \Rightarrow \mathcal{C} \ .$$

The other relations $\Rrightarrow$ and $\Lleftarrow$ are transitive as well, but not if mixed together. Either can be mixed with $\equiv$, however; thus from $\mathcal{A} \equiv \mathcal{B} \Rrightarrow \mathcal{C}$ we still have $\mathcal{A} \Rrightarrow \mathcal{C}$. Finally, writing just $\Rrightarrow \mathcal{A}$ on its own means that $\mathcal{A}$ is True in every state.

D.10.2 Rules for calculation

To reason as above requires some knowledge of the rules to which one can appeal, like "associativity, commutativity of $\vee$", that is "disjunction" as described in Sec. E.1.1. Each can be used to justify steps in a calculation, and often there are several that will do. One soon acquires favourites.

We do not present all those rules here; indeed, it will be some time before we need many of them. Where helpful, however, we refer to them directly. The reasoning above proved Rule E.40.[12] Here it is again, by numbers:

$$
\begin{array}{ll}
 & (\mathcal{A} \Rightarrow \mathcal{C}) \vee (\mathcal{B} \Rightarrow \mathcal{C}) \\
\equiv & \text{"Rule E.26 connecting } \Rightarrow \text{ with } \neg \text{ and } \vee\text{"} \\
 & (\neg\mathcal{A} \vee \mathcal{C}) \vee (\neg\mathcal{B} \vee \mathcal{C}) \\
\equiv & \text{"Rules E.3, E.5"} \\
 & (\neg\mathcal{A} \vee \neg\mathcal{B}) \vee (\mathcal{C} \vee \mathcal{C}) \\
\equiv & \text{"Rules E.22, E.1"} \\
 & \neg(\mathcal{A} \wedge \mathcal{B}) \vee \mathcal{C} \\
\equiv & \text{"Rule E.26"} \\
 & \mathcal{A} \wedge \mathcal{B} \Rightarrow \mathcal{C} \ .
\end{array}
$$

Note the use of equivalence to replace a *part* of a formula, leading to an equivalence for the *whole* formula. That is the usual rule in mathematics: we can substitute equals for equals. But some of our rules are entailments $\Rrightarrow$, not equivalences $\equiv$; their substitution within formulae can lead either to overall entailment or to its converse $\Lleftarrow$. But not always: entailment does distribute through quantification, conjunction, disjunction, and the consequent of implication; and it is reversed in negations and antecedents of implications. However it does not distribute at all through equivalence $\Leftrightarrow$.

Here is an example of distribution. Suppose we have $\mathcal{A} \Rrightarrow \mathcal{A}'$, $\mathcal{B} \Lleftarrow \mathcal{B}'$, and $\mathcal{C} \equiv \mathcal{C}'$. Then we can proceed as follows:

$$
\begin{array}{ll}
 & (\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow \mathcal{C} \\
\Rrightarrow & \text{"since } \mathcal{A} \Rrightarrow \mathcal{A}' \text{"} \\
 & (\mathcal{A}' \Rightarrow \mathcal{B}) \Rightarrow \mathcal{C} \\
\Rrightarrow & \text{"since } \mathcal{B} \Lleftarrow \mathcal{B}' \text{"} \\
 & (\mathcal{A}' \Rightarrow \mathcal{B}') \Rightarrow \mathcal{C} \\
\equiv & \text{"since } \mathcal{C} \equiv \mathcal{C}' \text{"} \\
 & (\mathcal{A}' \Rightarrow \mathcal{B}') \Rightarrow \mathcal{C}' \ .
\end{array}
$$

D.11 *Exercises* on propositions

Exercise D.1 (p. 207) What about `assert` and `assume` statements? Are they executed or not? Do they contribute to the program's answer, or instead to our confidence in the answer?

Exercise D.2 (p. 208) In Sec. D.2 it probably seems more natural that you should go *forwards* in checking Prog. D.1 — carry out the assignment on the initial comment, work your way through, and hope to end up with the final one.

Go on... Try it.

[12] The "Rule number" refers to the list in App. E; the actual number has no other significance.

Exercise D.3 (p. 210) Why does the set fact $A \cap B \subseteq A$ look so much like the propositional fact $\mathcal{A} \wedge \mathcal{B} \Rrightarrow \mathcal{A}$?

Exercise D.4 We've seen that intersection "$\cap$" between sets corresponds to "`and`" between propositions: you are an element of the *intersection* of two sets, say the set of women and the set of people taller than 1.75m, just if you are a woman *and* you are taller than 1.75m. Similarly "$\cup$" and "`or`" correspond.

In the same vein,

(a) What operator between sets corresponds to "$\Rrightarrow$" between propositions?

(b) What operator on sets corresponds to "$\neg$" on propositions?

(c) What operator between sets corresponds to "$\Rightarrow$" between propositions?

Exercise D.5 Which of these are terms?

(a) True See Ex. D.8.

(b) 17

(c) $\log^2 x$

(d) $\log \log x$

(e) $(\log x)^2$

(f) $\log x^2$

(g) $2x$

(h) $x < x{+}1$

Exercise D.6 Write terms for the following:

(a) The square root of the factorial of n.

(b) The factorial of the square root of n.

Exercise D.7 Which of these are propositional formulae?

(a) true

(b) *true*

(c) true

(d) True

(e) $x{<}y \Rrightarrow z$

(f) $x{<}y \Rightarrow z$

(g) $x{<}y \Rrightarrow y{<}z$

(h) $x{<}y \Rightarrow y{<}z$

(i) $x{<}y \Rrightarrow y{>}x$

Exercise D.8 (p. 214) Suppose your program contained a variable `b` of Boolean type, that is having value `True` or `False`. It could occur in several contexts:

(a) On the left of an assignment, as in `b= True`.

(b) On the right of an assignment, as in `b= not b`.

(c) In a conditional, as in `if b==True:`.

(d) In a conditional, as in `if b:`.

(e) In an assertion, as in `{ b }`.

(f) In an `assert` statement, as in `assert b`.

In each of those cases, is it a term — or is it a formula? Or something else?

Exercise D.9 (p. 214) What is another name for a *term* of arity 0?

Exercise D.10 (p. 215) Assuming that all variables denote natural numbers, which of these propositional formulae are True in all states?

(a) $x \geq 0$

(b) $x < y \Rightarrow x + 1 \leq y$

(c) $x \leq y \vee y \leq x$

(d) $x \leq y \wedge y \leq x \Rightarrow x = y$

(e) $x < y \wedge y < x \Rightarrow x = y$

(f) $x < y \wedge y < x \Rightarrow x \neq y$

(g) $x < y \vee y < x \Rightarrow x \neq y$

Exercise D.11 (p. 215) Use the truth tables of Figure D.5 to show that these formulae are True in all states:

(a) $\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A})$

(b) $(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C}))$

(c) $(\neg\mathcal{A} \Rightarrow \neg\mathcal{B}) \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A})$

Exercise D.12 (p. 215) We remarked that true and false are "0-adic" predicate symbols, being True respectively False in all states. How would you define 0-adic propositional connectives? What would their truth-tables be? Do we need them?

See Ex. D.11.

Exercise D.13 (p. 217) Show that $\mathcal{A} \Rrightarrow \mathcal{B} \Rightarrow \mathcal{A}$ is correct (valid). *Hint*: Recall the meaning of $\Rrightarrow$.

What about $\mathcal{A} \Rightarrow \mathcal{B} \Rrightarrow \mathcal{A}$?

D.12 Quantifiers

"Quantifiers" take us beyond purely propositional logic: they are how we express "for all" and "there exist". We haven't used them very much in this text; but for completeness we give here a brief description.

D.12.1 Universal quantification

A *universally quantified* formula is written

$$(\forall x \cdot \mathcal{A})\,, \quad {}^{13}$$

where x is a variable, called the *bound* variable, and $\mathcal{A}$ is some other formula, called the *body*. It is True in a state exactly when $\mathcal{A}$ is True for all values of x, where it is understood that we know the set from which those values of x are drawn (for example, the real numbers).

For now we assume that the possible values of x come from the state's entire collection of possible values; but the state does *not* map this x to any particular one of them.

We also allow a list of bound variables, as in $(\forall x, y \cdot \mathcal{A})$. There, the quantification is True exactly when the body is True for all values of those variables chosen independently. The order in the list does not affect the meaning.

Consider this parody of the distributive rule from arithmetic:

$$a + (b \times c) \quad = \quad (a + b) \times (a + c) \quad .$$

Although one would say informally "that is False", it is in fact True in some states. (Map all three variables to one-third.)

But the quantified formula

$$(\forall a, b, c \cdot a + (b \times c) = (a + b) \times (a + c)) \quad , \tag{D.6}$$

in which a, b, c are understood to range over the real numbers, is identically False in every state, because it is not the case that the body is true for *all* values of a, b, and c.

But now consider the similar formula

$$(\forall b, c \cdot a + (b \times c) = (a + b) \times (a + c)) \quad , \tag{D.7}$$

in which we have quantified only b and c. Its truth depends on the value the state assigns to a; and it is True when a is assigned 0, and False otherwise.

[13] It's a good idea to put parentheses around the outside of a quantification: they are like the $\{\cdots\}$ that indicate scope of local variables in programming languages like C.

D.12.2 Free and bound variables

Formula (D.7) depends on a, but not on b or c. Variable a is a *free* variable; variables b and c are not free, because they are bound by the quantifier $\forall$. In fact, variables b and c are just place-holders in that formula, indicating the positions at which all possible values in the state are to be considered. Changing their names does not affect the formula (provided the new names do not conflict with existing ones). Thus

$$(\forall d, e \cdot a + (d \times e) = (a + d) \times (a + e))$$

has the same meaning as (D.7). On the other hand, Formula (D.6) has no free variables, since a, b, c are bound; it does not depend on the value of any variable.

In general, *bound* variables are those bound by a quantifier, as is x in $(\forall x \cdot \mathcal{A})$; all free occurrences of x in $\mathcal{A}$ itself become bound occurrences in the larger $(\forall x \cdot \mathcal{A})$.

D.12.3 Existential quantification

Existential quantification is used to express "there exists". An *existentially quantified* formula is written

$$(\exists x \cdot \mathcal{A}) \quad ,$$

where x and $\mathcal{A}$ are as before. It is True exactly when there exists a value for x that makes $\mathcal{A}$ True. So the existentially quantified formula

$$(\exists a, b, c \cdot a + (b \times c) = (a + b) \times (a + c))$$

is True. Free occurrences of x in $\mathcal{A}$ are bound in $(\exists x \cdot \mathcal{A})$ just as they are in $(\forall x \cdot \mathcal{A})$.

D.12.4 Typed quantifications

A *typed quantification* indicates explicitly the set from which values for the bound variable are drawn. For example, if $\mathbb{Z}$ stands for the set of all integers, and $\mathbb{N}$ the for set of all natural numbers (non-negative integers). Then $(\exists x{:}\mathbb{Z} \cdot x < 0)$ is True, but $(\exists x{:}\mathbb{N} \cdot x < 0)$ is False (because 0 is the least natural number). In general, *typed* quantifications are written

$$(\forall x{:}T \cdot \mathcal{A}) \quad \text{and} \quad (\exists x{:}T \cdot \mathcal{A}) \ ,$$

where T denotes some set of values. The variable x then ranges over that set.

If we know beforehand the set from which values are drawn, we can use the simpler untyped quantifiers; the typing is then understood from context. But when several such sets are involved simultaneously, we use typed quantifiers.

D.13 *Exercises* on quantifiers

Exercise D.14 Assuming that the one-place predicates even, odd mean "is an even number", "is an odd number" respectively, write general formulae for the following:

(a) Every integer is either even or odd.

(b) Every odd natural number is one more than some even natural number.

(c) There is an even integer that is not one more than any odd natural number.

(d) Zero is the least natural number.

(e) There is no least integer.

(f) Given any positive real number, there is another real number strictly between it and zero.

Exercise D.15 Recall that $(\exists x \cdot \mathcal{A})$ means "there is *at least* one x such that $\mathcal{A}$". Write another formula that means "there is *at most* one x such that $\mathcal{A}$".

Exercise D.16 (Recall Exercise D.15.) Write a formula that means "there is *exactly* one x such that $\mathcal{A}$".

Exercise D.17 Show that this is correct, using rules from App. E:

$$(\exists x \cdot (\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\neg\mathcal{A} \Rightarrow \mathcal{C})) \quad \equiv \quad (\exists x \cdot \mathcal{A} \wedge \mathcal{B}) \vee (\exists x \cdot \neg\mathcal{A} \wedge \mathcal{C})\,.$$

Exercise D.18 Suppose $\mathcal{N}$ contains no free x. Prove correctness of this:

$$(\exists x \cdot (\mathcal{N} \Rightarrow \mathcal{A}) \wedge (\neg\mathcal{N} \Rightarrow \mathcal{B})) \quad \equiv \quad (\mathcal{N} \Rightarrow (\exists x \cdot \mathcal{A})) \wedge (\neg\mathcal{N} \Rightarrow (\exists x \cdot \mathcal{B}))\ .$$

Hint: Recall Exercise D.17.

Exercise D.19 Prove correctness of this, for any formula $\mathcal{A}$:

$$(\exists a \cdot (\forall b \cdot \mathcal{A})) \quad \Rrightarrow \quad (\forall b \cdot (\exists a \cdot \mathcal{A}))\,.$$

Is the converse true? See Ex. E.5.

Exercise D.20 Show that $(\exists x, y \cdot x \neq y) \equiv (\forall x \cdot (\exists y \cdot x \neq y))$ is correct.
Hint: To show $\mathcal{A} \equiv \mathcal{B}$, show $\mathcal{A} \Rrightarrow \mathcal{B} \Rrightarrow \mathcal{A}$.

D.14 (General) formulae

Now we can draw together all the above. A *formula* is any one of the following:

(a) A simple formula.

(b) $\neg\mathcal{A}$, where $\mathcal{A}$ is a formula.

(c) $\mathcal{A} \wedge \mathcal{B}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A} \Rightarrow \mathcal{B}$, or $\mathcal{A} \Leftrightarrow \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are formulae.

(d) $(\forall x{:}T \cdot \mathcal{A})$ or $(\exists x{:}T \cdot \mathcal{A})$, where x is a list of variables, T denotes a set, and $\mathcal{A}$ is a formula.

That definition allows nested quantifications, such as

$$(\forall a{:}\mathbb{R} \cdot (\exists b, c{:}\mathbb{R} \cdot a + (b \times c) = (a + b) \times (a + c)))$$

(which is True in all states), and the application of propositional operators to quantifications, such as

$$x \neq 0 \Rightarrow (\exists y{:}\mathbb{Z} \cdot 0 \leq y \wedge y < x) \quad ,$$

True in all states in which x is a natural number.

Figure D.7 gives some general formulae.

$$\mathsf{true}$$
$$x{\neq}3$$
$$y{>}0 \Rightarrow y{\neq}0$$
$$\big(\forall x{:}\mathbb{R} \cdot \big(\exists y{:}\mathbb{C} \cdot y^2{=}x\big)\big)$$
$$a \div b = c \;\Leftrightarrow\; (\exists r \cdot 0{\leq}r{<}b \;\wedge\; a = b{\times}c + r)$$

The real numbers are $\mathbb{R}$; the complex numbers are $\mathbb{C}$.

Figure D.7 Some general formulae

Appendix E

Some helpful logical identities [1]

Here we will use the usual symbolx from logic, rather than their Python equivalents. Thus `not` is $\neg$ and `and` is $\wedge$ and `or` is $\vee$.

E.1 Some basic propositional rules

Throughout this Sec. E.1 the symbols $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ stand for formulae of predicate calculus. The rules however are *propositional* because they do not deal with the quantification or substitution of variables: those topics are covered in Sec. E.2.

E.1.1 Conjunction and disjunction

The propositional connectives for conjunction, $\wedge$, and disjunction, $\vee$, are idempotent, commutative, associative and absorptive, and they distribute through each other. (In conditions within programs they would be written `and` and `or`.)

Conjunction and disjunction are *idempotent* connectives:

$$\mathcal{A} \wedge \mathcal{A} \quad \equiv \quad \mathcal{A} \quad \equiv \quad \mathcal{A} \vee \mathcal{A} \quad . \tag{E.1}$$

Conjunction and disjunction are *commutative* connectives:

$$\mathcal{A} \wedge \mathcal{B} \quad \equiv \quad \mathcal{B} \wedge \mathcal{A} \tag{E.2}$$
$$\mathcal{A} \vee \mathcal{B} \quad \equiv \quad \mathcal{B} \vee \mathcal{A} \quad . \tag{E.3}$$

Conjunction and disjunction are *associative* connectives:

$$\mathcal{A} \wedge (\mathcal{B} \wedge \mathcal{C}) \quad \equiv \quad (\mathcal{A} \wedge \mathcal{B}) \wedge \mathcal{C} \tag{E.4}$$
$$\mathcal{A} \vee (\mathcal{B} \vee \mathcal{C}) \quad \equiv \quad (\mathcal{A} \vee \mathcal{B}) \vee \mathcal{C} \quad . \tag{E.5}$$

Rules E.1 to E.5 mean that we can ignore duplication, order and bracketing in conjunctions $\mathcal{A} \wedge \mathcal{B} \wedge \cdots \wedge \mathcal{C}$ and disjunctions $\mathcal{A} \vee \mathcal{B} \vee \cdots \vee \mathcal{C}$.

Sometimes terms can be removed immediately from expressions involving both conjunctions and disjunctions. This is *absorption*:

$$\mathcal{A} \wedge (\mathcal{A} \vee \mathcal{B}) \quad \equiv \quad \mathcal{A} \quad \equiv \quad \mathcal{A} \vee (\mathcal{A} \wedge \mathcal{B}) \; . \tag{E.6}$$

[1] These rules were collected originally in *Laws of the logical calculi*, Technical Monograph PRG-78, Carroll Morgan and JW Sanders, 1989.

The *distribution* of $\wedge$ through $\vee$ is similar to the distribution of multiplication over addition in arithmetic. But in logic distribution goes both ways, so that $\vee$ also distributes through $\wedge$:

$$\mathcal{A} \wedge (\mathcal{B} \vee \mathcal{C}) \equiv (\mathcal{A} \wedge \mathcal{B}) \vee (\mathcal{A} \wedge \mathcal{C}) \tag{E.7}$$
$$\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C}) \equiv (\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C}) \quad . \tag{E.8}$$

And finally, we have

$$\mathcal{A} \wedge \mathcal{B} \Rrightarrow \mathcal{A} \tag{E.9}$$
$$\mathcal{A} \wedge \mathcal{B} \Rrightarrow \mathcal{B} \tag{E.10}$$
$$\mathcal{A} \Rrightarrow \mathcal{A} \vee \mathcal{B} \tag{E.11}$$
$$\mathcal{B} \Rrightarrow \mathcal{A} \vee \mathcal{B} \tag{E.12}$$

E.1.2 Constants and negation

In ordinary multiplication, $a \times 1 = a$ and $a \times 0 = 0$. We say therefore that 1 is a *unit* and 0 a *zero* of multiplication. Similarly, the predicate constant true is the unit of $\wedge$ and the zero of $\vee$:

$$\mathcal{A} \wedge \mathsf{true} \equiv \mathcal{A} \tag{E.13}$$
$$\mathcal{A} \vee \mathsf{true} \equiv \mathsf{true} \quad . \tag{E.14}$$

The constant false is the unit of $\vee$ and the zero of $\wedge$:

$$\mathcal{A} \wedge \mathsf{false} \equiv \mathsf{false} \tag{E.15}$$
$$\mathcal{A} \vee \mathsf{false} \equiv \mathcal{A} \quad . \tag{E.16}$$

Negation $\neg$ acts as a *complement*:

$$\neg\mathsf{true} \equiv \mathsf{false} \tag{E.17}$$
$$\neg\mathsf{false} \equiv \mathsf{true} \tag{E.18}$$
$$\mathcal{A} \wedge \neg\mathcal{A} \equiv \mathsf{false} \tag{E.19}$$
$$\mathcal{A} \vee \neg\mathcal{A} \equiv \mathsf{true} \quad . \tag{E.20}$$

Furthermore it is an *involution*:

$$\neg\neg\mathcal{A} \equiv \mathcal{A} \quad . \tag{E.21}$$

And it satisfies De Morgan's laws:

$$\neg(\mathcal{A} \wedge \mathcal{B}) \equiv \neg\mathcal{A} \vee \neg\mathcal{B} \tag{E.22}$$
$$\neg(\mathcal{A} \vee \mathcal{B}) \equiv \neg\mathcal{A} \wedge \neg\mathcal{B} \quad . \tag{E.23}$$

With negation, we have two more absorptive rules:

$$\mathcal{A} \vee (\neg\mathcal{A} \wedge \mathcal{B}) \equiv \mathcal{A} \vee \mathcal{B} \tag{E.24}$$
$$\mathcal{A} \wedge (\neg\mathcal{A} \vee \mathcal{B}) \equiv \mathcal{A} \wedge \mathcal{B} \quad . \tag{E.25}$$

E.1.3 Normal forms

A formula is in *disjunctive normal form* if it is a finite disjunction of other formulae each of which is, in turn, a *conjunction* of simple formulae. *Conjunctive normal form* is defined complementarily.

Rules E.7, E.8, E.22 and E.23 allow us to convert any proposition to either disjunctive or conjunctive normal form, as we choose, and rules E.19 and E.20 serve to remove adjacent complementary formulae. For example,

$$
\begin{array}{ll}
 & \mathcal{A} \wedge \neg(\mathcal{B} \wedge \mathcal{C} \wedge \mathcal{A}) \\
\equiv & \text{“Rule E.22”} \\
 & \mathcal{A} \wedge (\neg\mathcal{B} \vee \neg\mathcal{C} \vee \neg\mathcal{A}) \\
\equiv & \text{“Rule E.7”} \\
 & (\mathcal{A} \wedge \neg\mathcal{B}) \vee (\mathcal{A} \wedge \neg\mathcal{C}) \vee (\mathcal{A} \wedge \neg\mathcal{A}) \\
\equiv & \text{“Rule E.19”} \\
 & (\mathcal{A} \wedge \neg\mathcal{B}) \vee (\mathcal{A} \wedge \neg\mathcal{C}) \vee \mathsf{false} \\
\equiv & \text{“Rule E.16”} \\
 & (\mathcal{A} \wedge \neg\mathcal{B}) \vee (\mathcal{A} \wedge \neg\mathcal{C}) \quad .
\end{array}
$$

The second formula above is in conjunctive normal form and the third, fourth, and fifth are in disjunctive normal form.

E.1.4 Implication

Implication $\Rightarrow$ satisfies the law

$$\mathcal{A} \Rightarrow \mathcal{B} \quad \equiv \quad \neg\mathcal{A} \vee \mathcal{B} \ , \tag{E.26}$$

and that leads on to these rules:

$$
\begin{array}{rcll}
\mathcal{A} \Rightarrow \mathcal{A} & \equiv & \mathsf{true} & \text{(E.27)} \\
\mathcal{A} \Rightarrow \mathcal{B} & \equiv & \neg(\mathcal{A} \wedge \neg\mathcal{B}) & \text{(E.28)} \\
\neg(\mathcal{A} \Rightarrow \mathcal{B}) & \equiv & \mathcal{A} \wedge \neg\mathcal{B} & \text{(E.29)} \\
\mathcal{A} \Rightarrow \mathcal{B} & \equiv & \neg\mathcal{B} \Rightarrow \neg\mathcal{A} \quad . & \text{(E.30)}
\end{array}
$$

The last above is called the *contrapositive law*. Useful special cases of those are

$$
\begin{array}{rcll}
\mathcal{A} \Rightarrow \mathsf{true} & \equiv & \mathsf{true} & \text{(E.31)} \\
\mathsf{true} \Rightarrow \mathcal{A} & \equiv & \mathcal{A} & \text{(E.32)} \\
\mathcal{A} \Rightarrow \mathsf{false} & \equiv & \neg\mathcal{A} & \text{(E.33)} \\
\mathsf{false} \Rightarrow \mathcal{A} & \equiv & \mathsf{true} & \text{(E.34)} \\
\mathcal{A} \Rightarrow \neg\mathcal{A} & \equiv & \neg\mathcal{A} & \text{(E.35)} \\
\neg\mathcal{A} \Rightarrow \mathcal{A} & \equiv & \mathcal{A} \quad . & \text{(E.36)}
\end{array}
$$

These next two rules distribute implication $\Rightarrow$ through conjunction and disjunction:

$$
\begin{array}{rcll}
\mathcal{C} \Rightarrow (\mathcal{A} \wedge \mathcal{B}) & \equiv & (\mathcal{C} \Rightarrow \mathcal{A}) \wedge (\mathcal{C} \Rightarrow \mathcal{B}) & \text{(E.37)} \\
(\mathcal{A} \vee \mathcal{B}) \Rightarrow \mathcal{C} & \equiv & (\mathcal{A} \Rightarrow \mathcal{C}) \wedge (\mathcal{B} \Rightarrow \mathcal{C}) & \text{(E.38)} \\
\mathcal{C} \Rightarrow (\mathcal{A} \vee \mathcal{B}) & \equiv & (\mathcal{C} \Rightarrow \mathcal{A}) \vee (\mathcal{C} \Rightarrow \mathcal{B}) & \text{(E.39)} \\
(\mathcal{A} \wedge \mathcal{B}) \Rightarrow \mathcal{C} & \equiv & (\mathcal{A} \Rightarrow \mathcal{C}) \vee (\mathcal{B} \Rightarrow \mathcal{C}) \quad . & \text{(E.40)}
\end{array}
$$

The following extra rules are useful in showing that successive hypotheses may be conjoined or even reversed:

$$\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C}) \quad \equiv \quad (\mathcal{A} \wedge \mathcal{B}) \Rightarrow \mathcal{C} \quad \equiv \quad \mathcal{B} \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C}) \quad . \tag{E.41}$$

And the next law is the basis of definition by cases:

$$(\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\neg\mathcal{A} \Rightarrow \mathcal{C}) \quad \equiv \quad (\mathcal{A} \wedge \mathcal{B}) \vee (\neg\mathcal{A} \wedge \mathcal{C}) \quad . \tag{E.42}$$

E.1.5 Equivalence

Equivalence satisfies this law:

$$\begin{aligned} \mathcal{A} \Leftrightarrow \mathcal{B} \quad &\equiv \quad (\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\mathcal{B} \Rightarrow \mathcal{A}) && \text{(E.43)} \\ &\equiv \quad (\mathcal{A} \wedge \mathcal{B}) \vee \neg(\mathcal{A} \vee \mathcal{B}) && \text{(E.44)} \\ &\equiv \quad \neg\mathcal{A} \Leftrightarrow \neg\mathcal{B} \quad . && \text{(E.45)} \end{aligned}$$

Also we have these:

$$\begin{aligned} \mathcal{A} \Leftrightarrow \mathcal{A} \quad &\equiv \quad \mathsf{true} && \text{(E.46)} \\ \mathcal{A} \Leftrightarrow \neg\mathcal{A} \quad &\equiv \quad \mathsf{false} && \text{(E.47)} \\ \mathcal{A} \Leftrightarrow \mathsf{true} \quad &\equiv \quad \mathcal{A} && \text{(E.48)} \\ \mathcal{A} \Leftrightarrow \mathsf{false} \quad &\equiv \quad \neg\mathcal{A} && \text{(E.49)} \\ \mathcal{A} \Rightarrow \mathcal{B} \quad &\equiv \quad \mathcal{A} \Leftrightarrow (\mathcal{A} \wedge \mathcal{B}) && \text{(E.50)} \\ \mathcal{B} \Rightarrow \mathcal{A} \quad &\equiv \quad \mathcal{A} \Leftrightarrow (\mathcal{A} \vee \mathcal{B}) && \text{(E.51)} \\ \mathcal{A} \vee (\mathcal{B} \Leftrightarrow \mathcal{C}) \quad &\equiv \quad (\mathcal{A} \vee \mathcal{B}) \Leftrightarrow (\mathcal{A} \vee \mathcal{C}) \quad . && \text{(E.52)} \end{aligned}$$

Equivalence is commutative and associative

$$\begin{aligned} \mathcal{A} \Leftrightarrow \mathcal{B} \quad &\equiv \quad \mathcal{B} \Leftrightarrow \mathcal{A} && \text{(E.53)} \\ \mathcal{A} \Leftrightarrow (\mathcal{B} \Leftrightarrow \mathcal{C}) \quad &\equiv \quad (\mathcal{A} \Leftrightarrow \mathcal{B}) \Leftrightarrow \mathcal{C} \ , && \text{(E.54)} \end{aligned}$$

and, from Rules E.50 and E.51, it satisfies EW Dijkstra's *Golden Rule*:[2]

$$\Rrightarrow \quad \mathcal{A} \wedge \mathcal{B} \Leftrightarrow \mathcal{A} \Leftrightarrow \mathcal{B} \Leftrightarrow \mathcal{A} \vee \mathcal{B} \quad , \tag{E.55}$$

where the $\Rrightarrow$ alone on the left means "*unconditionally* correct" (universally valid), equivalently "True in *all* states".

See Ex. E.1.

E.2 Some basic predicate rules

Now we consider rules concerning the universal and existential quantifiers, $\forall$ and $\exists$. Although for most practical purposes we wish the quantification to be *typed*

$$\begin{array}{l} (\forall x{:}T \cdot \mathcal{A}) \\ (\exists x{:}T \cdot \mathcal{A}) \end{array} \quad ,$$

where T denotes a type and $\mathcal{A}$ is a formula, for simplicity we state our rules using untyped quantifications:

$$\begin{array}{l} (\forall x \cdot \mathcal{A}) \\ (\exists x \cdot \mathcal{A}) \end{array} \quad .$$

[2] In some texts the Golden Rule is written $\mathcal{A} \wedge \mathcal{B} \equiv \mathcal{A} \equiv \mathcal{B} \equiv \mathcal{A} \vee \mathcal{B}$, in which case the "$\equiv$" is being used as a propositional connective. Here we are using "$\equiv$" as a relation between formulae, and not as a propositional connective; and that is why we write (E.55) with "$\Leftrightarrow$".

Each can be converted to a rule for typed quantification by uniform addition of type information, *provided the type is non-empty.* These rules enable us to convert between the two styles:

$$(\forall x{:}T \cdot \mathcal{A}) \quad \equiv \quad (\forall x \cdot x \in T \Rightarrow \mathcal{A}) \tag{E.56}$$
$$(\exists x{:}T \cdot \mathcal{A}) \quad \equiv \quad (\exists x \cdot x \in T \wedge \mathcal{A}) \quad , \tag{E.57}$$

where the simple formula $x{\in}T$ as usual means 'x is in the set T'.

For more general constraints than typing, we have these abbreviations as well, which include a range formula $\mathcal{R}$:

$$(\forall x{:}T \mid \mathcal{R} \cdot \mathcal{A}) \quad \equiv \quad (\forall x \cdot x \in T \wedge \mathcal{R} \Rightarrow \mathcal{A}) \tag{E.58}$$
$$(\exists x{:}T \mid \mathcal{R} \cdot \mathcal{A}) \quad \equiv \quad (\exists x \cdot x \in T \wedge \mathcal{R} \wedge \mathcal{A}) \quad , \tag{E.59}$$

Note that E.56 and E.58 introduce implication, but E.57 and E.59 introduce conjunction.

E.2.1 Substitution and instantiation

We write substitution of a term E for a variable x in a formula $\mathcal{A}$ as

$$\mathcal{A}[x\backslash E] \quad ,$$

and we write the multiple substitution of terms E and F for variables x and y respectively as

$$\mathcal{A}[x, y\backslash E, F] \quad .$$

Unless quantifiers are present, substitution $\mathcal{A}[x\backslash E]$ means *literally* replace all occurrences of x by E, so that things like

$$(\mathcal{A} \Rightarrow \mathcal{B})[x\backslash E] \quad \equiv \quad \mathcal{A}[x\backslash E] \Rightarrow \mathcal{B}[x\backslash E]$$

are true by definition (of substitution).

Substitution is used for example in the rule for checking assignments (Secs. 5.2 and B.1.1); but it is also used to explain the basic properties of the quantifiers: we have

$$(\forall x \cdot \mathcal{A}) \quad \Rrightarrow \quad \mathcal{A}[x\backslash E] \qquad \text{for any term } E \tag{E.60}$$
$$\mathcal{A}[x\backslash E] \quad \Rrightarrow \quad (\exists x \cdot \mathcal{A}) \qquad \text{for any term } E \quad , \tag{E.61}$$

that is that $\forall x$ means "holds for *any* term E you substitute for x", and that $\exists x$ follows from "holds if it held for *any* term you substituted for x". And in simple cases, substitutions just replace the variable by the term.

But if the variable being replaced (for example x) occurs *within* the scope of a quantifier, then we must take account of whether variables are free or bound. Suppose, for example, that $\mathcal{A}$ is the formula $(\exists x \cdot x \neq y) \wedge x = y$; then

$$\begin{array}{rrl} & \mathcal{A}[x\backslash y] & \text{is} \quad (\exists x \cdot x \neq y) \wedge y = y \ , \\ \text{but} & \mathcal{A}[y\backslash x] & \text{is} \quad (\exists z \cdot z \neq x) \wedge x = x \quad . \end{array}$$

The variable z is *fresh*, not appearing in $\mathcal{A}$. In the first case, $x \neq y$ is unaffected because *that* occurrence of x is bound by $\exists x$. Indeed, since we could have used any other letter (except y) without affecting the meaning of the formula –and it would not have been replaced in that case– we do not replace it in this case either. The occurrence of x in $x = y$ is free, however, and the substitution occurs.

In the second case, since both occurrences of y are free, both are replaced by x. But on the left we must not "accidentally" quantify over the newly introduced x – $(\exists x \cdot x \neq x)$ would be wrong– so we change (before the substitution) the bound x to a fresh variable z. Such an "accidental quantification" is called *variable capture*.

Finally, note that multiple substitution can differ from successive substitution:

$$\begin{array}{rlll} & \mathcal{A}[y\backslash x][x\backslash y] & \text{is} & (\exists z \cdot z \neq y) \wedge y = y \\ \text{but} & \mathcal{A}[y, x\backslash x, y] & \text{is} & (\exists z \cdot z \neq x) \wedge y = x \quad . \end{array}$$

E.2.2 The one-point rules

These rules allow quantifiers to be eliminated in many cases. They are called "one-point" because the bound variable is constrained to take one value exactly. If x does not occur (free) in the term E, then

$$(\forall x \cdot x = E \Rightarrow \mathcal{A}) \quad \equiv \quad \mathcal{A}[x\backslash E] \quad \equiv \quad (\exists x \cdot x = E \wedge \mathcal{A}) \quad . \tag{E.62}$$

If the type T in Rules E.56 and E.57 is finite, say $\{a, b\}$, we have the similar

$$(\forall x{:}\{a, b\} \cdot \mathcal{A}) \quad \equiv \quad \mathcal{A}[x\backslash a] \wedge \mathcal{A}[x\backslash b] \tag{E.63}$$

$$(\exists x{:}\{a, b\} \cdot \mathcal{A}) \quad \equiv \quad \mathcal{A}[x\backslash a] \vee \mathcal{A}[x\backslash b] \quad . \tag{E.64}$$

Those can be extended to larger (but still finite) types $\{a, b, \ldots, z\}$. We are led to think, informally, of universal and existential quantification as infinite conjunction and disjunction respectively over all the constants of our logic:

$$\begin{array}{lll} (\forall x{:}\mathbb{N} \cdot \mathcal{A}) & \text{represents} & \mathcal{A}(0) \wedge \mathcal{A}(1) \cdots \\ (\exists x{:}\mathbb{N} \cdot \mathcal{A}) & \text{represents} & \mathcal{A}(0) \vee \mathcal{A}(1) \cdots \end{array}$$

E.2.3 Quantifiers alone

Quantification is idempotent:

$$(\forall x \cdot (\forall x \cdot \mathcal{A})) \quad \equiv \quad (\forall x \cdot \mathcal{A}) \tag{E.65}$$

$$(\exists x \cdot (\exists x \cdot \mathcal{A})) \quad \equiv \quad (\exists x \cdot \mathcal{A}) \quad . \tag{E.66}$$

Extending De Morgan's laws E.22 and E.23, we have

$$\neg\,(\forall x \cdot \mathcal{A}) \quad \equiv \quad (\exists x \cdot \neg\mathcal{A}) \tag{E.67}$$

$$\neg\,(\exists x \cdot \mathcal{A}) \quad \equiv \quad (\forall x \cdot \neg\mathcal{A}) \quad . \tag{E.68}$$

E.2.4 Extending the commutative rules

These rules extend the commutativity of $\wedge$ and $\vee$:

$$(\forall x \cdot (\forall y \cdot \mathcal{A})) \quad \equiv \quad (\forall x, y \cdot \mathcal{A}) \quad \equiv \quad (\forall y \cdot (\forall x \cdot \mathcal{A})) \tag{E.69}$$

$$(\exists x \cdot (\exists y \cdot \mathcal{A})) \quad \equiv \quad (\exists x, y \cdot \mathcal{A}) \quad \equiv \quad (\exists y \cdot (\exists x \cdot \mathcal{A})) \quad . \tag{E.70}$$

E.2.5 Quantifiers accompanied

Extending the associative and previous rules,

$$(\forall x \cdot \mathcal{A} \wedge \mathcal{B}) \quad \equiv \quad (\forall x \cdot \mathcal{A}) \wedge (\forall x \cdot \mathcal{B}) \tag{E.71}$$

$$(\exists x \cdot \mathcal{A} \vee \mathcal{B}) \quad \equiv \quad (\exists x \cdot \mathcal{A}) \vee (\exists x \cdot \mathcal{B}) \tag{E.72}$$

$$(\exists x \cdot \mathcal{A} \Rightarrow \mathcal{B}) \quad \equiv \quad (\forall x \cdot \mathcal{A}) \Rightarrow (\exists x \cdot \mathcal{B}) \quad . \tag{E.73}$$

Here are weaker rules (using $\Rrightarrow$ rather than $\equiv$) which are nonetheless useful:

$$(\forall x \cdot \mathcal{A}) \Rrightarrow (\exists x \cdot \mathcal{A}) \tag{E.74}$$
$$(\forall x \cdot \mathcal{A}) \vee (\forall x \cdot \mathcal{B}) \Rrightarrow (\forall x \cdot \mathcal{A} \vee \mathcal{B}) \tag{E.75}$$
$$(\forall x \cdot \mathcal{A} \Rightarrow \mathcal{B}) \Rrightarrow (\forall x \cdot \mathcal{A}) \Rightarrow (\forall x \cdot \mathcal{B}) \tag{E.76}$$
$$(\exists x \cdot \mathcal{A} \wedge \mathcal{B}) \Rrightarrow (\exists x \cdot \mathcal{A}) \wedge (\exists x \cdot \mathcal{B}) \tag{E.77}$$
$$(\exists x \cdot \mathcal{A}) \Rightarrow (\exists x \cdot \mathcal{B}) \Rrightarrow (\exists x \cdot \mathcal{A} \Rightarrow \mathcal{B}) \tag{E.78}$$
$$(\exists y \cdot (\forall x \cdot \mathcal{A})) \Rrightarrow (\forall x \cdot (\exists y \cdot \mathcal{A})) \quad . \tag{E.79}$$

See Ex. E.3.

E.2.6 Manipulation of quantifiers

If a variable has no free occurrences, its quantification is superfluous:

$$(\forall x \cdot \mathcal{A}) \equiv \mathcal{A} \qquad \text{if } x \text{ is not free in } \mathcal{A} \tag{E.80}$$
$$(\exists x \cdot \mathcal{A}) \equiv \mathcal{A} \qquad \text{if } x \text{ is not free in } \mathcal{A} \quad . \tag{E.81}$$

Other useful rules of this kind are the following, many of which are specialisations of Rules E.71 to E.73. In each case, variable x must not be free in the formula $\mathcal{N}$:

$$(\forall x \cdot \mathcal{N} \wedge \mathcal{B}) \equiv \mathcal{N} \wedge (\forall x \cdot \mathcal{B}) \tag{E.82}$$
$$(\forall x \cdot \mathcal{N} \vee \mathcal{B}) \equiv \mathcal{N} \vee (\forall x \cdot \mathcal{B}) \tag{E.83}$$
$$(\forall x \cdot \mathcal{N} \Rightarrow \mathcal{B}) \equiv \mathcal{N} \Rightarrow (\forall x \cdot \mathcal{B}) \tag{E.84}$$
$$(\forall x \cdot \mathcal{A} \Rightarrow \mathcal{N}) \equiv (\exists x \cdot \mathcal{A}) \Rightarrow \mathcal{N} \tag{E.85}$$
$$(\exists x \cdot \mathcal{N} \wedge \mathcal{B}) \equiv \mathcal{N} \wedge (\exists x \cdot \mathcal{B}) \tag{E.86}$$
$$(\exists x \cdot \mathcal{N} \vee \mathcal{B}) \equiv \mathcal{N} \vee (\exists x \cdot \mathcal{B}) \tag{E.87}$$
$$(\exists x \cdot \mathcal{N} \Rightarrow \mathcal{B}) \equiv \mathcal{N} \Rightarrow (\exists x \cdot \mathcal{B}) \tag{E.88}$$
$$(\exists x \cdot \mathcal{A} \Rightarrow \mathcal{N}) \equiv (\forall x \cdot \mathcal{A}) \Rightarrow \mathcal{N} \quad . \tag{E.89}$$

Bound variables can be renamed, as long as the new name does not conflict with existing names:

$$(\forall x \cdot \mathcal{A}) \equiv (\forall y \cdot \mathcal{A}[x\backslash y]) \qquad \text{if } y \text{ is not free in } \mathcal{A} \tag{E.90}$$
$$(\exists x \cdot \mathcal{A}) \equiv (\exists y \cdot \mathcal{A}[x\backslash y]) \qquad \text{if } y \text{ is not free in } \mathcal{A} \quad . \tag{E.91}$$

Finally, we have for any term E,

$$(\forall x \cdot \mathcal{A}) \Rrightarrow \mathcal{A}[x\backslash E] \tag{E.92}$$
$$\mathcal{A}[x\backslash E] \Rrightarrow (\exists x \cdot \mathcal{A}) \quad . \tag{E.93}$$

If $\mathcal{A}$ is True for all x, then it is True for E in particular; and if $\mathcal{A}$ is True for E, then certainly it is True for some x.

E.3 *Exercises* on rules for logic

See Ex. E.2.

Exercise E.1 (p. 228) Why does $\mathsf{true} \Rrightarrow \mathcal{A}$ mean "$\mathcal{A}$ is True in all states"? Given that it does, it suggests that it would be convenient to make true the default left-hand side of $\Rrightarrow$. Can you suggest more concrete justifications for that?

What should the right-hand default of $\Rrightarrow$ be, and why?

See Ex. E.1.

Exercise E.2 If $\mathcal{A} \Rrightarrow \mathcal{B}$ is correct, does it follow that $\neg\mathcal{B} \Rrightarrow \neg\mathcal{A}$ is also correct? If so, can you apply De Morgan's Law to your answer to Ex. E.1?

See Ex. E.4.

Exercise E.3 (p. 231) Explain in words why Rule E.78 is correct. Couldn't the two x's that make $\mathcal{A}$ and $\mathcal{B}$ True on the left be *different* x's?

See Ex. E.3.

Exercise E.4 Prove Rule E.78 *without* the hand-waving that you were deliberately encouraged to do in Ex. E.3. That is, produce a line-by-line proof in the style of Sec. E.1.3.

Hint: The line-by-line proof of E.78 is only three lines, i.e. two steps: use E.88 and E.61. On the other hand, the in-English "model" answer given in Ex. E.3 involved a case analysis *and* two full paragraphs of prose.

See Ex. D.19.

Exercise E.5 Try adapting your answer to Ex. D.19 so that you prove also the converse:

$$(\exists a \cdot (\forall b \cdot \mathcal{A})) \quad \Leftleftarrows \quad (\forall b \cdot (\exists a \cdot \mathcal{A}))\,.$$

What goes wrong?

Exercise E.6 Revisit the programming exercises in which we discovered a bit of condition-arithmetic was needed... and see whether you can use the rules in this appendix to *do* that arithmetic. They were Exercises 8.5, 8.6, 10.5, 16.8 and 17.5.

Check your answers with truth tables (Fig. D.5).

E.4 Epilogue on notation and terminology [3]

This whole text –its point– is *not* intended to study Logic for its own sake. Still, it's useful to make explicit connections with the conventional treatments. In particular, there are many (different) opinions about whether to use ($\rightarrow$) or ($\Rightarrow$) for implication, or ($\equiv$) or ($\Leftrightarrow$) for equivalence, and so on...

What's important is to be consistent within a single text. Below are our conventions for here:

implication in English comes in many forms. In everyday English it's "If *this*, then *that*." or equivalently "*This* implies *that*." Note however that "*This* if *that*." means the exact opposite of "implies". It's "*That* implies *this*."

An alternative to "implies" is "only if", so that "*This* only if *that*." is the same as "*This* implies *that*."

[3] This section is based on *A Mathematical Introduction to Logic*, Herbert B Enderton, Academic Press, 1972. But any conventional book devoted to Formal Logic would do.

reverse implication in English From just above, we see that "*This* is implied by *that*." means "*This* if *that*."

bi-implication in English is implication both ways. It can be written "if and only if", or "just when" or (often by mathenmaticians and logicians) just "iff". It can also be written "is equivalent to".

well formed formulae Logic has very strict rules about syntax, i.e. about what symbol-strings are formulae and what are not. (The same strictness applies to programs in a programming language but, of course, not the same rules exactly.) A well formed formula is called a *wff* (and pronounced somewhere between "wiff" and "woof").

implication as a logical connective is *in this text* written $\Rightarrow$. In some presentations of logic however (especially text books on logic itself) it is written $\rightarrow$ instead. No matter how it is written, placed between two *wff*'s, it makes another (bigger) *wff* whose left-hand side is the *antecedent* and right-hand side is the *consequent*.

The single-line "implies" brings with it $\leftarrow$ for "is implied by" and $\leftrightarrow$ for "iff". (In this text we use $\Leftarrow$ and $\Leftrightarrow$.)

implication as "entails" is in this text written $\Rrightarrow$. In some (probably most) presentations of logic it's written $\models$ or $\vdash$ and these two *are* different (as we explain below) — in this case it's not just a matter of whether you prefer a single- or a double line.

Whichever you use, that is $\Rrightarrow$ or $\models$ or $\vdash$, it's written *between* two *wff*'s, and it does not make a new *wff*. Instead it indicates a kind of implication between the left-hand *wff* (still the antecedent) and the right-hand *wff* (still the consequent). Sometimes however one speaks of "assumptions" and "conclusions" instead of left-hand side or antecedent etc.

semantic entailment is what we are principally concerned with here, that is that $\mathcal{A} \Rrightarrow \mathcal{B}$ or $\mathcal{A} \models \mathcal{B}$ (whichever way you write it) means "In every state, if $\mathcal{A}$ is True then so is $\mathcal{B}$." There are reversed versions as well.

"logical" entailment is what is mean by $\mathcal{A} \vdash \mathcal{B}$, that you can *prove by following syntactic rules* that $\mathcal{A} \models \mathcal{B}$. The difference is a bit like using a truth table (which would be $\models$) or using the rules in this App. E (which would be $\vdash$).

In this text however we are not making a distinction between the two, and so we use $\Rrightarrow$ for both.

soundness is a property of logic itself (rather than being a property of something the logic is "about"). It is that if $\mathcal{A} \vdash \mathcal{B}$ holds then so does $\mathcal{A} \models \mathcal{B}$, i.e. that if we use our rules to reason, a truth-table would agree with our conclusion.

Usually, in presentations of logic, the number of rules is kept to a minimum: for propositional logic you need only three rules to be able to prove everything — provided you also allow "If you have proved $\mathcal{A}$ and you have proved $\mathcal{A} \Rightarrow \mathcal{B}$ then you have (are deemed to have) proved $\mathcal{B}$ as well." (It's called "Modus Ponens".) The reason for keeping the number of rules small is that it makes it easier to prove things *about* the logic, like its soundness.

completeness is also a property of logic itself, but goes the other way: it is that if $\mathcal{A} \models \mathcal{B}$ holds then so does $\mathcal{A} \vdash \mathcal{B}$, i.e. if a truth-table shows something to be

correct (brute force), then you could show the same thing by using the right logical rules (cleverness).

implication (reprise) In logic (i.e. in life) there are at least five kinds of implication, all meaning more or less "if... then..." But they apply to different things, and keeping them clearly separated is one of the crucial things that makes logic –and therefore checking program correctness– possible. Here they are:

1: Ordinary "implies" is used in everyday speech.

2: The symbol $\Rightarrow$, or $\rightarrow$ makes a bigger *wff* from two smaller ones, and that bigger *wff* is True *in a particular state* if the antecedent's being True in that state implies the consequent's being True in that same state.

3: The symbol $\models$ relates two *wff*'s, meaning that *in every state* if the antecedent *wff* is True in that state then so will the consequent *wff* be True in that same state.

4: The symbol $\vdash$ relates two *wff*'s, meaning that you can prove, by following pre-defined (and presumably sound) logical rules, that *in every state* if the antecedent *wff* is True in that state then so will the consequent *wff* be True in that same state.

The key thing here is the assertion that there is a proof.

5: A horizontal line, with some $\vdash$'s above and some below, asserts that if you can prove the ones above, you are deemed to have proved the one below. Here is an example of Modus Ponens written that way:

$$\frac{\vdash \mathcal{A} \qquad \vdash \mathcal{A} \Rightarrow \mathcal{B}}{\vdash \mathcal{B}} \ .$$

If you read $\vdash \mathcal{A}$ as "$\mathcal{A}$ is a theorem", then Modus Ponens is in effect a rule about how to infer other theorems from theorems you already have, and so it's called a "rule of inference".

Appendix F

Illustration of heap behaviour during garbage collection

F.1 Mark-and-Sweep garbage collection

The illustrations in Fig. F.1(a)–(g) show a successful mark-and-sweep garbage collection as described in Sec. 17.4: it begins with the heap in a "random" state, where there are not only reachable nodes coloured white, but also unreachable nodes coloured black. The small numbers in the first illustration show the reference counts as well, so that we can see the pair at bottom left would never be collected if we used that approach. (Pointers to null are omitted in the diagrams.)

For mark-sweep without interference, the program is paused, "reachability" is propagated from the roots, and then the unreachable nodes are collected. The program is then resumed.

In (h) we show what can go wrong when the "Mutator" program is *not* paused, and it interferes with the scanning.

F.2 Woodger's scenario

"Woodger's Scenario", mentioned in Sec. 17.11, is shown in Fig. F.2 adapted to our current black/white with black-counting scheme.

See Ex. 17.11.

(1) Initially `Root1` points to `Null`, `Root2` points to `T` and `T` points to `Null`. A scan has completed; all nodes are black; and there is no garbage.

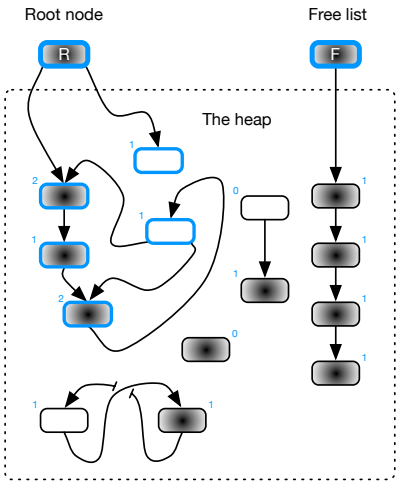
(2) The Mutator decides that it will swing `Root1` from `Null` to `T`, and as its first step (in the incorrect order) it has blackened `T` (which was however already black).

Its second step (6) will be to complete the swing of `Root1` from `N` to `T`; but before it can do that...

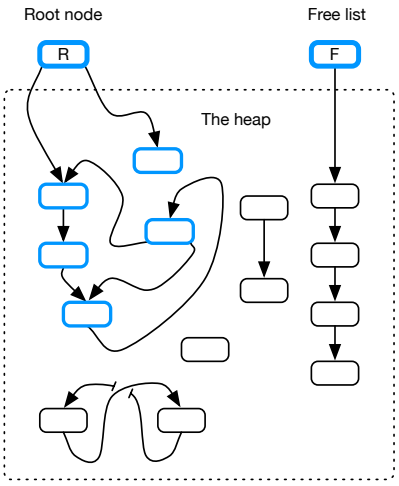
(3) The Mutator is suspended in State (2), and the Scanner-Collector makes a complete cycle: again all nodes are already black; there is no garbage.

It then begins a second cycle: its first step is to whiten all nodes (including `T`).

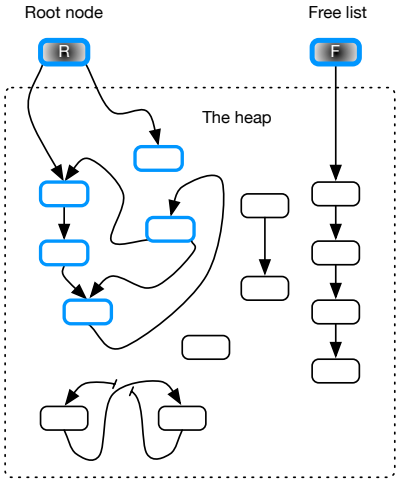
(4) But then it blackens the roots `R1`, `R2` and `N`. Node `T` however remains white.



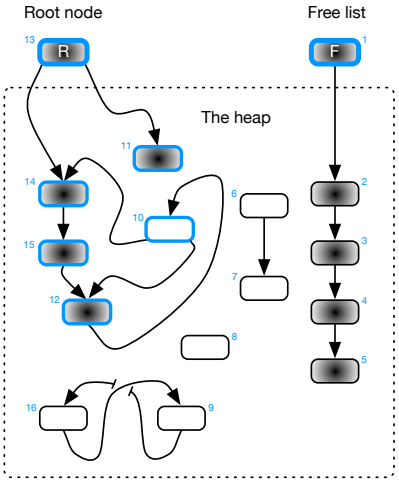
(a) Initially the colours are "randomly" assigned, and don't correspond properly to reachability (except for the two roots R and F).
(The node-pair at bottom left would never be collected using reference counting.)



(b) The first step in mark-sweep is to whiten every node: all the blacks in (a) are now white.
(We have removed the reference counts.)

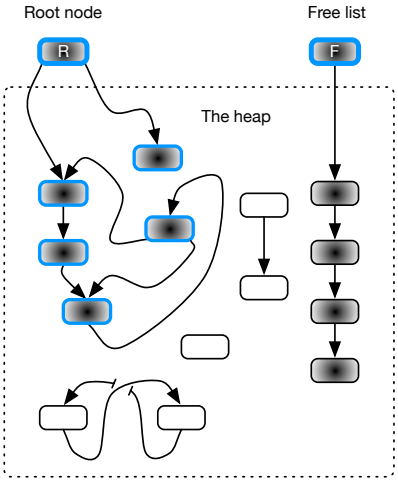


(c) Then the roots are blackened. (Here there are only two.)
Then the nodes are examined in arbitrary order, in this example from top to bottom within right to left, numbered 1–16.

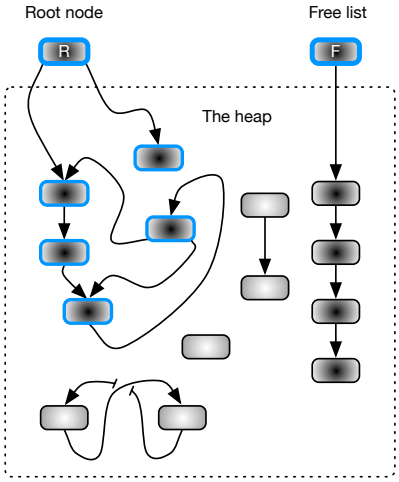


(d) Any black node propagates black to its "children". Here we have scanned all the nodes once; but Node 10 will be blackened (from Node 12) only on the *next* scan.
Then the third scan will blacken no more nodes, and the overall scanning loop will terminate.

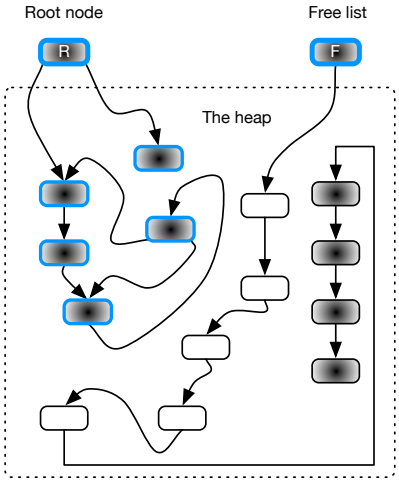
Figure F.1 Illustration of mark-sweep (App. F.1, and continued below)



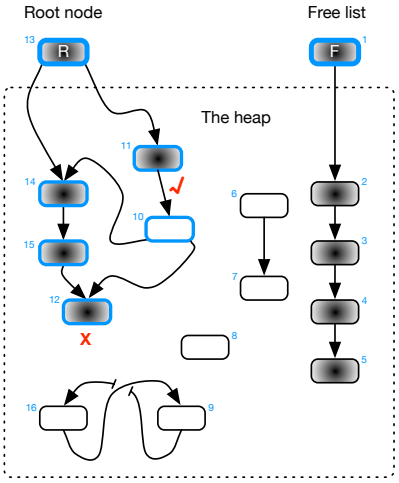
(e) The (top to bottom, left to right here) scanning and propagation finishes when no more propagations occur.



(f) Once that is done, any nodes that are still white must be garbage.



(g) The garbage nodes are added to the free list. (Here we added them at the front.



(h) Here we illustrate possible interference from the Mutator in Step (d).

The collection of garbage has completed successfully in Step (g).

However in (h) we show what interference can do, i.e. if the user's program is not paused while the scan is going on. After the Scanner has examined Node 11 in (d), but before it examines Node 12, the Mutator uses Node 12's pointer to find Node 10, and changes Node 11's null pointer to that. Then the Mutator sets Node 12's pointer to null.
The Scanner resumes, and completes in (d) –no nodes blackened– but Node 10 is reachable and white: instead of (e) we will end up with (h).

Figure F.1 Illustration of mark-sweep (continued)

(5) The Scanner continues, but gets only as far as checking `R1` before it is suspended. Since `R1` already points to black `N`, the black-count does not increase. But then...

(6) ... the Mutator finally (and disastrously) completes its swing, not noticing that `Root1` now points to the white `T`. *It does not blacken* `T`, however, because it has already done that (2).

(7) The Mutator also begins a swing of `Root2` from `T` to `Null`, first colouring `N` black (which it is already), then then...

(8) ... completes the swing of `Root2` from `T` to `Null`.

(9) The Scanner completes, scanning `Root2`, `N` and `T`. But because `R1` has already been scanned, its black is not propagated to `T`; and because of the Mutator's swing (8), the black at `R2` is not propagated either.

The scan completes, because the number of blacks has not increased: it is still 3.

(10) And then the Collector collects `T`, because it is white. But it is pointed to by `R1`, and so is not garbage.

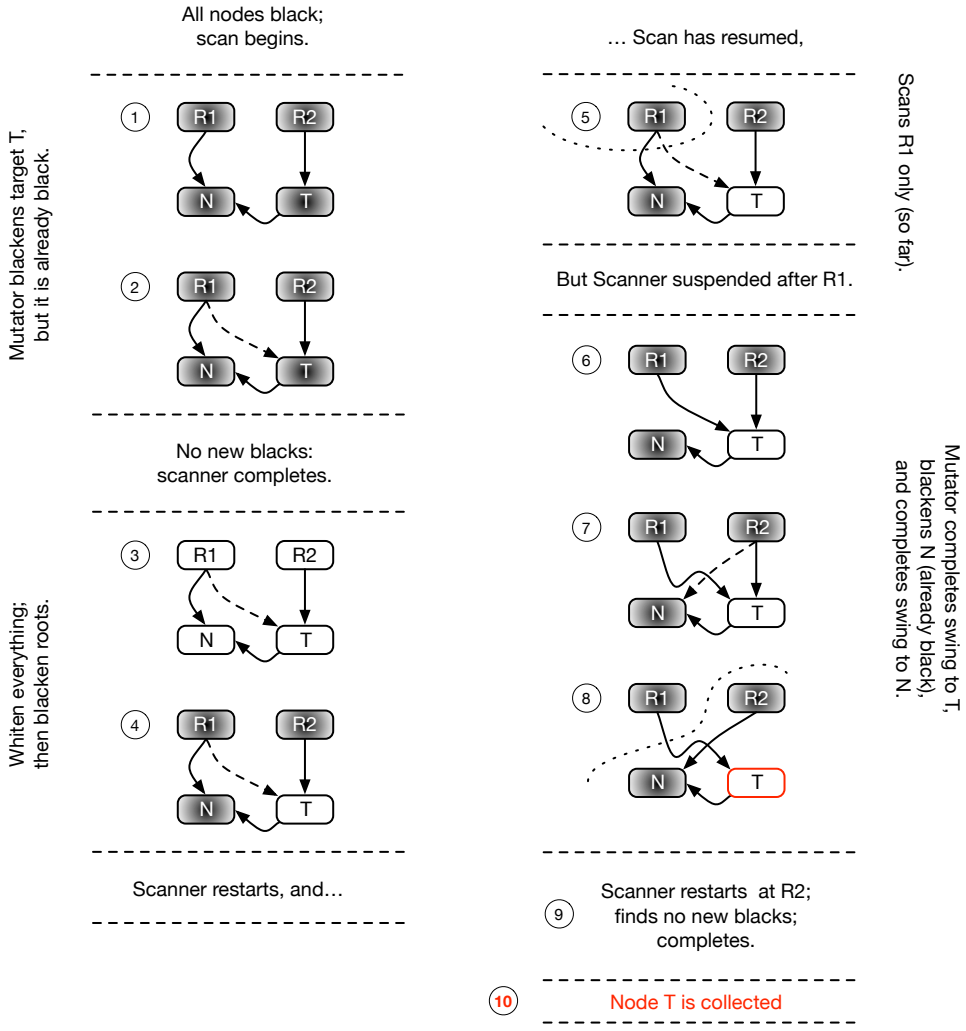


Figure F.2 Woodger's scenario (App. F.2)

Appendix G

Python-specific issues

G.1 Block structure

In Python, control structures are as usual indicated explicitly by a keyword at their beginning (with `if`, `while` etc.) but the *body* of the structure does not have an explicit keyword to indicate its end: instead, indentation is used. A slight disadvantage of that convention is that postconditions, say *after* a loop rather than after the last statement of the loop body, are identified only because they are "outdented". Thus for example in Prog. 3.2 (partly repeated here)

```
n,r= 0, Fun([])
while n!=N:
  ...
  n= n+1
  { r == Fun(A[:n]) }
{ n==N and r == Fun(A[:n]) }    ,
```

the assertion `r == Fun(A[:n])` is *within* the loop, at the end of the body (indicating in fact that the loop invariant is restored). But the assertion `n==N and r == Fun(A[:n])` is *outside* the whole loop, in fact the loop's postcondition: its invariant *and* the negation of its condition.

The only indication of the difference is that the second is outdented, left-aligned with the `while` above.

G.2 `for`-loops vs. `while`-loops in Python

In Python, a typical `for`-loop is written

```
for n in range(start,end,increment): body
```

and –like a `while`-loop– it executes the loop body for varying values of the variable `n`, usually the values

```
start
start+increment
start+2*increment
⋮
```

until (but not including) the first value that goes to `end` *or beyond.*

and so on. Indeed `for`-loops seem (and sometimes are) an attractive alternative to `while`-loops, because

(a) You don't need to initialise `n` explicitly — the loop executes `n= start` automatically for you, before it begins.

(b) You don't need to increment `n` explicitly — the loop executes `n= n+increment` automatically for you, at the end of (almost) every iteration.

(c) You don't need to test `n` explicitly to exit the loop.

So for example instead of

```
n= 0
while n!=N:
  body
  n= n+1
```

you might instead write the more concise `for n in range(N): body` .

But you must be careful about the *last* value the variable `n` takes. And so we give below an example of what programmers should bear in mind when they use `for`-loops. Always check the precise definition for the language you are using.

In Python, the loop

```
for n in range(N):
  body
{ n == N-1 }
```
(G.2)

establishes `n==N-1` at loop end, provided `N>=1`. But the "corresponding" loop

```
n= 0
while n!=N:
  body
{ n==N }
```

instead establishes `n==N` at loop end, provided `N>=0`.
That means that although the two loops execute the loop bodies the same number of times, and with the same values of `n`, the two loop forms are *not* equivalent. (G.2)

The `while`-loop equivalent (in Python) of Prog. G.2 is actually

```
if N>0:
  n= 0
  while True:
    body
    if n<N-1: break
    n= n+1    .
```

Note that if `N==0` in the fragment above, then `n` is not assigned to at all (not even set to 0), and that otherwise, on loop end variable `n` will be `N-1`, not `N`.

See Ex. 1.10.
See Ex. 3.10.

G.3 Object orientation in Python, and in general

In Python, encapsulated data types are made using "classes" and "instances" of them; and the procedures and/or functions used to access them are called "methods". The variables the instances contain are called "attributes."

What we studied in Part II were the initial simple approaches (historically) to encapsulation, so that we could see where the rules for checking more modern program-structuring tools come from, and why they are the way they are. The closest modern equivalent of what we have been discussing is probably the "static" class of Java (and its equivalent in other languages). The idea of the coupling invariant however is (and remains) the key.

But *maintenance* of coupling invariants – all of those ideas that were set out in Chp. 10– are in modern object-oriented programming languages sometimes difficult to enforce by compile-time checks. For example, in Python an encapsulated data type would be made by declaring a "class" and then making "instances" of it; and in general (unless you take special measures) the procedures, functions and variables inside any of those instances can freely be called (procedures and functions) and read and updated (variables) from outside the instance, blatantly violating the principles we set out in (b)(i) of Sec. 10.3. And if that is done, the invariant of the data type (i.e. of the instance) will possibly not be maintained. See Ex. G.1.

To address the encapsulation issue (at least partially), each object-oriented language has its own collection of "access modifiers" that control how the encapsulation "boundary" is allowed to be broken and by whom: the modifiers have names like "private", "protected", "public" and "package"; and each object-oriented language has its own rules for what they mean and what they guarantee.

G.4 Exercises

Exercise G.1 (p. 243) In Prog. 10.1 we saw an abstract class implementing a set of maximum size `N`, and in Prog. 10.2 we saw a concrete version of that class, using an array to store the set's values. Both `makeEmpty` and `add` were "public" procedures accessible from outside the class. (The procedure `find` was annotated "local", i.e. not accessible from outside the class.)

What problem might you encounter if one public procedure in a class called another public procedure in the same class, either directly or via other intermediate procedures possibly defined elsewhere?

Exercise G.2 Python has a `continue` statement that returns control immediately to the beginning of the enclosing `while` loop. What is the connection between that practice and the potential problem mentioned in Ex. G.1?

Does a similar problem arise in the use of `break`? See Ex. G.1.

Appendix H

Exercise answers suppressed in this copy

Appendix I

Drill answers suppressed in this copy

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, 217
$\Rightarrow$ is "implies", **8**
$\Rrightarrow$ is "implies" as well (for now), **8**
  but is actually entailment, 217
`//` is (integer) quotient, **16**
`%` is remainder, **16**
`+` is sequence concatenation, **73**
`+` is sequence concatenation, **19**
`**` means "to the power of", **38**
`[x]*r` makes a sequence of `r` `x`'s, **41**, 94∘
$\equiv$ is "iff" (for now), **59**
  but is actually equivalence,
  and does not operate between Booleans, **217**
$\in$ membership
  of bag, **106**
  of set, **73**
`-` difference
  of two bags, **106**
  of two sets, **74**
$\cup$ union
  of two sets, **73**
`{}` is the empty set, **73**
`+` sums two bags, **106**
$\bot$ is "bottom"
  meaning "no thread", **122**, 135
`x:[`*pre*`,`*post*`]`, specification, **111**
`$b` as "lock variable" for `b`, **122**
$\wedge$ is conjunction, corresponding to `and` between Booleans in a program, 212, **225**
$\vee$ is disjunction, corresponding to `or` between Booleans in a program, **215**, 218, 225
$\Leftrightarrow$ is iff-and-only-if, corresponding to `==` in a program, **215**
$\neg$ is negation, corresponding to `not` applied to a Boolean in a program, **215**
$\forall$ for all, *see* quantification
$\exists$ there exists, *see* quantification
$\rightarrow$ is implication too (but here we write it $\Rightarrow$), **233**
$\models$ is semantic entailment (but here we write it $\Rrightarrow$), **233**
$\vdash$ is logical entailment (but here we write it $\Rrightarrow$ as well), **233**

`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`, 139
  `A[1:]` is the tail of `A`, 139
  `A[n:n]` is the empty sequence, 45, 47
J-R **Abrial**, 179
absorption, 225
abstract data-type, 77–78
abstract/concrete hybrid, *qv*
access modifier, *see* object orientation
accreditation

of Engineers, 65
of programmers, 65
`acquire()` in Python, 121↓
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`, 133
left-hand side of implication, **8**, 215
anti-symmetric, **52**
arithmetic, of Booleans, *qv*
arity of operator, function or predicate, **213**
"arrays" in Python, 94
`assert` statement, 95, **112**
at checking time, 93
at run time, 93
how to check, **198**
if it fails, allows unpredictable behaviour, 101
in Python, 93↓
assertion `{···}`, **9**, 7–9, 15, **106**
brackets vs. comment, 20, 115
checked from end towards beginning, 59, 61, 177
compile-time vs. run-time, 174
"housekeeping", 18
how to check, **198**
intermediate, 18, 58, 177
pre- and postconditions are special cases of, 15
several in a row, 13, 59, 60
used as "scaffolding", 28
vertical vs. horizontal format, 9
assignment statement
checked by textual substitution, 8, 13, 16, 17, *see* exercises, 32, 33, 57–59, 62, **191**, 229
examples of, 57
multiple, 10, 84
not available in *C*, 58↓
associativity
of ∧, 225
of ∨, 217, 225
of longest common prefix, 46
of multiplication, 85
`assume` statement
how to check, **198**
with frame, **112**
without frame, **112**
ATM, 127
automated teller machine, 120
in Imperial currency, 51
invariant for, 120
many active concurrently, 120
atomic action, **121**, 120–121, 123, 127, 141
written on a single line, 130, 141, 153
atomicity
implicit if two or more statements are on the same line, 197
removing it, 161–163
auxiliary variable, 79–82, 86–87, 132∘, 132, 195
adding, 79
does not appear in running code, 133
label-like, 141, 142
removing, 80, 86
use in postcondition, 58
for variants, 50, 58, 61
`await` statement, 131–133
how to check, **132**, **197**
implemented with lock, 134

`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**, 175
binary search, 36–37, 45–47, 61, 72, 74, 81, 91, 100, 102, 110, 175–177
interpreting postcondition, 45
overflow bug, 175↓, 178
trees, 92
black nodes, *see* garbage collection
blame
finding who is responsible for an error, 95
body
of a quantification, **221**
Boolean
arithmetic, 82
expression, 3

honorary, 140, 152↓
Boolean arithmetic, 82
bound variable, 221, **222**, 229
bounds checking, *see* "everyday" errors
`break` to exit loop, 21, 64, 195, 243, 244
bubble sort, 63
is inefficient, 63
bugs
found using `assert` statements, 198
in specification, 107
introduced by carelessness, 161
never-ending `while`-loop, 14
their main source, 177
building
a garden shed, an office block, an aeroplane, 65
vocational skill, 65↓

C programming language, 3↓, 23↓, 146, 148, 193↓
has no built-in sets, 73↓
but Python does, 73↓
has no multiple assignments, 58↓, 88↓, 88, 89
but Python does, 84
pointers, 146
CAS, *see* compare and swap
cascading invariants, 42–44, 84
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, 112
automatically with a verifier, 162
conditionals, 62
effort can double with each new variable added, 66
in isolation, 60
loop bodies, 13
circular reasoning, 130
not allowed, 129
only apparent, 129, 130
circular structure
made from pointers, 148, 149∘, 166
class definition
data-type invariants, 93
global constants, 101
global variables (references to), 101
local
variables, 77
overall precondition, 93
parameter, 93
procedure precondition, 93
coding
reducing mistakes, v
what coding really is, iii
coffee cups, disappearing, 145, 147
Collector, 147, **148**
comments, v
good comments, 15
some comments are not very good, 5
"What does this bit of code do?" vs. "What's true here?", 67
common sense is built on sound engineering principles, 109
commutative, 225
commutativity
of ∨, 217
of quantifiers, 230
compare and swap, abbreviated *CAS*, **123**, 131, 133, 137
complement, 226
completeness, **234**
concatenation, *see* sequence
concrete data-type, 79–81
concurrency
atomic action
for compare and swap, 123
atomicity, 121
critical section, **123**, 123, 131–135, **137**
implemented with lock, 123
implemented with queue, 139
thread cannot remain indefinitely within it, 139, 142
deadlock, 133–135, 139, 142
fair scheduling, 135
global variables, 137
"in the large" vs. "in the small", 145
in the world generally, 120
interleaving, 119
livelock, 133–135
liveness, 133–135, 139
locks, 121–123

mutual exclusion, **131**, 131–133, **137**
implemented with queue, 139
of *ATM*'s, 120
purpose and examples, 119–121
safety, 133–135, 139, 142, 143
sharing variables, 119
simplest example, 119
starvation, 133–135, 139, 142, 144
thread, **119**, 121, 131
within an operating system, 145
condition
used for checking programs, 3
written using Python syntax, 3
conditional (`if`)
examples of checking, 59–61
how to check
with `elif`, 62
with `else`, 62, **194**
without `else`, **193**
conditional strict majority `csm`, 39, 47
conjunct, 11↓, **32**
splitting two, 31
splitting more than two, 32
expression next to "`and`" or ∧, **215**
conjunction
written "`and`" or ∧, 11↓, 32, 215, 225
conjunctive normal form, **227**
consequent
right-hand side of implication, **8**, 215
constant symbols in logic
are as usual, 213
written in sans-serif, 213
`continue` to restart loop, 21, 243
continued fraction, *see* program exercises
contrapositive, 227
counting black nodes, *see* garbage collection
coupling invariants, **74**, 73–82, 84, 86, 89, 91–93, 99, 101, 103, 107, 108, 112–114, 135, 243
explain how data-type operations cooperate, 74
for implementing mutual exclusion, 140
global variables (references to), 101
keeping only the last six digits, 89
linking matrices to scalars, 88
course assignments
a sample of, vii
critical section, *see* concurrency
`csm`, *see* conditional strict majority

Dafny, 174–177
data refinement, *qv*
data structures, 71–72
triples, 71
data types
basic, **71**
primitive, **71**
structured, **71**
their encapsulation, 72, 74, 76
data-type encapsulation, 91–103
as a "locked box", 88, 113
case studies
pocket calculator, 105–108
set (data type), 91–98
global variables (references to), 101
hides internal state, 88
local
procedures, 94∘, 94, 96
variables, 77, 96
more advanced techniques, 100–101
rules for
justification of, 98–100
summary of, 95–98
data-type invariant, **93**, 91–95
always results in a refinement, 100, 103
ensures consistency, 91
established by class initialisation, 93
implicit pre- and postcondition of every class procedure (method), 93
maintained by every class procedure (method), 93
might not be maintained, 243
sometimes allowed to be broken, 102
data-type representation
vs. encapsulation, 92
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, **226**
deadlock, *see* concurrency

debugging the programming *process*, v, 4
defensive programming, 95, 97, 198
determinant of empty matrix, *see* exercises
dictionary order, 52
EW **Dijkstra**, 149↓, 161↓, 179, 212↓
  Golden Rule, 228↓, 228
`DiM()`, days in month, 31
disasters and their causes, 106
disjunct
  expression next to "`or`" or ∨, **215**
disjunction
  written "`or`" or ∨, 215, 218, 225
disjunctive normal form, **227**
distribution, 226
distributive rule
  from arithmetic, 221
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, 82, 87
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, 47
encapsulated data type, *see* data-type encapsulation
entailment (⇛), **8**, **217**
  different from implication ⇒, 215↓, 217
  distribution of, 218
equivalence (≡)
  chain of, 217
  does not operate between Booleans, 217
  is not a propositional connective, 217, 228↓
Euclid's algorithm `gcd`, 50
"everyday" errors, 17
  index out of bounds, 4, 17, 23, 43, 47, 143
    is sometimes acceptable, 95
  uninitialised variable, 47
exchange sort, 64
  is inefficient, 64
executing code "in your head", 78
exercises (assorted)
  `assert` statement, 101
  assertion `{···}`
    brackets vs. comment, 115
  assignment-statement checking, 17
  binary search (degenerate), 81
  breaking data-type invariants, 102
  delete `del()` from the Mean Calculator, 108
  determinant of empty matrix, 19
  facts about sequences and ∑, 17
  linear search, 81, 82
  loop invariants, 81
  minimum vs. minimal, 56
  postcondition, 81
  precondition, 81
  `remove()` element from set, 101
  requirements, 81
  set membership, 81
  specification used as program text, 114
  summing a sequence, 19
    if empty, 19
  `undo()` procedure for Mean Calculator, 108
  variants, 81
existential quantification, *qv*
(there) exists ∃, *see* quantification
expressions
  how to check for undefined, **193**
"eyeballing" an implication, 19, 21

`F`, free chain start, *see* garbage collection
false
  unit of ∨, 226
  zero of ∧, 226
false describes no states, **214**
"false", *see* "true"
Fibonacci numbers, **83**, 83–89
  in logarithmic time, 86–87
finite sets of integers, 56
`floor` function, 46

flowchart, 10, 13
  checking *between* the boxes vs. checking *inside* the boxes, 67
  very helpful intuitively, 67
flowchart, making one, 21
RW **Floyd**, **10**, 179
for all ∀, *see* quantification
`for`-loop
  assertions for, 47
  cannot execute forever, 14
  how to check, **196**
  vs. `while`-loop, 47, 101, 196, 241
    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, **214**
  general, **223**
  propositional, 215
frame of a specification, 111
`free()`, `malloc()`, 146 ff
free chain `F`, *see* garbage collection
free variable, **222**, 229
"fresh" means "not already used for something else", **123**
frozen music player, 4
function symbols
  are as usual, 213
function symbols in logic
  written in sans-serif, 213
function, higher-order, **215**

garbage
  caused by pointers, 146
  is in the heap, 146
  sometimes called "space leak", 147
  what garbage is, 146–147
  why garbage needs collection, 147
garbage collection, 145–169
  Collector, **147**, 147
  complete vs. safe, 166
  `F`, free chain start, **151**
  free chain, 147
    starts at `F`, **151**
  global correctness, 164
  label-like auxiliary variable, 162
  mark-sweep, 147
  Mutator, **148**
    might be more than one, 168
  Mutator-Collector interference is not possible, 148
  Mutator-Scanner interference, 148, 150○, 151, 153, 154○, 154, 156○, 159
  `N`, null pointer target, **151**
  nodes, **149**
    black, **151**
    counting black nodes, 159–161
    white, **151**
  null pointer target `N`, **151**
  "on the fly", **148**
  reachable node, 147, **149**
  reference counting, 148
  root node, **149**
  safe vs. complete, 166
  Scanner, **147**, 151–153, 155–158
    global correctness, 153–155, 157–158, 160, 163, 167
    invariants for, 152, 153, 155, 161, 163, 164, 166
    local correctness, 152–153, 155–157, 159–160, 162–164, 167
    variants for, 152, 155, 166
    Version 1 (no), 152, 155
    Version 2 (ok, but), 157, 158
    Version 3 (ok, but), 159, 160
    Version 4 (yes), 166
    Version 4 (yes, finally), 165
  Scanner-Collector interference, 167
  swinging a pointer, **149**
garden shed, building, *qv*
GC, *see* global correcteness
`gcd`, greatest common divisor, **50**
global
  postcondition, *qv*
  precondition, *qv*
global correctness, 124, **129**, 129, 167, 197
  abbreviated *GC*, 129
global invariance, 128–129
global invariants, **128**, 140, 142
  compare globally correct, stable, 144
global variables, 101, 137
globally correct
  compare stable, global invariants,

144
KF **Gödel**, 177↓
Golden Rule, *see* EW Dijkstra
Good subsegment, 29
greatest common divisor `gcd`, 50
    variant, 50
D **Gries**, 149↓, 179

handle, turning it, 10
head-scratching
    "Use the assertions, Luke.", 36, 153
heap, *see* garbage (collection)
hint in proof, 217
CAR **Hoare**, 144, 179, 187–189
"housekeeping" assertions, *qv*
how to check programs, summary and definitions, **191**
hybrid of abstract- and concrete, 79

idempotent, 217, 225
`if ··· else: skip`, 61
`if`-branches
    reason about each `if`-case separately, 37, 41
IFM, acronym for "(In-)Formal Methods", vii
"Imperial" currency: pounds, shillings and pence, 51
implication
    does not usually have an equivalent symbol for Booleans in a program, 215
    five different meanings of *??*, 234
    written (⇒), **8**
      different from entailment ⇛, 217
implicit increment, *see* `for`-loop
(In-)Formal Methods, iii
    some of its benefits, 66
    what it is, v
indexing is from zero, 4
"indexitis", 190↓
infinite loop, *see* loop variants
infinitely descending chain, **52**
infinity ∞
    getting rid of, 48
    used as value in program, 48
inlining, *see* procedure calls
insertion sort, 64
    is inefficient, 64
interference, **124**, 124, 129, 139, 141, 197
    between Mutator and Scanner, *see* garbage collection
interleaving, 121, 127–128, 135
    between lines of code, 127↓
intermediate assertion, 18
internal state
    hidden by encapsulation, 88
internet, program "fixes", 4
interruption
    without, 132
interrupts, 124
    controlling them for mutual exclusion, 137
introduce a new variable, 33–37, 40
introductory course, vii
invalid, *see* valid
invariants (and finding them), 21, 31–48
    associated with locks, 122
    cascading, 42–44, 84
    choose them *before* coding, 31
    coupling, *qv*
    data-type, *qv*
    even simple ones are helpful, 38, 42
    for recursion, 83, 88
    global, *qv*
    helps to design loops, 27
    introduce new variable, 33–37
    must usually be supplied by the programmer, 177
    simplify postcondition, 38–41
    split conjunct, 31–33
involution, **226**
isolation, checking components in, *qv*
iterating
    down, 45, 58, 61
    towards the middle, 36–37
    up, 33, 43, 58, 84, 152

Jack
    "be nimble", 3
    should be more careful, 4, 32, 49, 155
Java programming language, 243

DE **Knuth**, 175
König's Lemma, *see* termination

label-like auxiliary variable, *qv*
`lad`, *see* largest absolute difference
L **Lamport**, 149↓

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, 114
length of shortest run sr, 48
lexicographic variant, **51**, *see also* termination, 55
its underlying order, 54
sometimes can be reduced to simple, 55
sometimes can't be reduced to simple, 55
lgs(n), longest *Good* subsegment, **27**
linear search, 72, 74, 75, 77, 81, 82, 110
sometimes too inefficient, 71
linear-time exponential, 38
list of variables, *qv*
livelock, *see* concurrency
liveness, *see* concurrency
local
correctness, 124, 167
abbreviated *LC*, 128
procedures in encapsulation, 94∘, 94, 96
variables in encapsulation, 77, 96, 106
`local` is not a declaration in Python, 77↓
`local def`, 94∘
`find(s)`, 110
lock variable
variable `$b` associated with lock `b`, 131
lock-free implementation, 144
"locked box", *see* data-type encapsulation
locks
adding them, 120–121
implemented with *CAS*, 123
`lock` statement, 121
lock variable
variable `$b` associated with lock `b`, 122
`unlock` statement, 121
what locks guarantee, 122
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, 66
early exit with `break`, 35
exit, 13
(avoiding) infinite, 22, *see* loop variants
invariants, 10–13, 23–48, 91
unconventional thought processes for its design, 66
unrolling
preserves correctness, 35
variants, 14–15, 49, 53, 58, 61, 64, 195
for spinlock, 125
"The Lost Art", iv↓, 173, 179
lr, longest run, **48**

majority voting csm, 47
`malloc()`, `free()`, 146 ff, 151
AJ **Martin**, 149↓
mathematics all the way down, iv, 172–179
matrix multiplication
used to calculate Fibonacci numbers, 84
`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, 62
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, 47, 48
specification, 42
maximum up to here muh, **42**
The Mean Calculator, 105–108
mes, *see* maximum end sum
method

get- and set- , 96
of a class, 79
minimal has no others less than it, 52, 56
minimum is less than all others, 52, 56
J **Misra**
and Peterson's algorithm, 139
modularity and re-use, v
Modus Ponens, **233**
is a rule of inference, 234
motorcycle
tuning your own, 65
`mss`, maximum segment sum, **42**
`muh`, *see* maximum up to here
multi-set (also bag), 105, **106**
multiple assertions, how to check
on one line, **192**
on successive lines, **192**
multiple assignment, *qv*
music player, frozen, 4
Mutator, **148**
mutual exclusion, 131, 132∘

`N`, null-pointer target, *see* garbage collection
negation, written "`not`" or ¬, 215
nodes, *see* garbage collection
non-empty-segment sum, 48
null pointer `N`, *see* garbage collection

OB abbreviates "other business", 131
object orientation, 243
What's it for?, 109
. . . is not discussed generally in this text, 109↓
access modifier, 243
package, 243
private, 243
protected, 243
public, 243
class, 243
attribute, 243
instance, 243
method, 243
static, 243
in `Python`, 243
one public procedure calling another, 243
office block, *see* building
one-point laws, **230**
other business, abbreviated *OB*, 131
outside-in design, *see* program
overflow bug, *see* binary search
S **Owicki**, 149↓, 179
Owicki-Gries Method, 124, 127–135

partial correctness, **49**
partial order, **52**
`pass`, *see* `skip`
pence, *see* Imperial
GL **Peterson**, 137–144
his mutual-exclusion algorithm, 139–143
"poised" between atomic statements, 141
cannot deadlock, 142
complete code for, 142∘
is safe, 142
label-like auxiliary variable, 141
prevents starvation, 142
pointers, *see C* programming language
null, swinging, *see* garbage collection
"poised" between atomic statements, *see* Peterson's algorithm
polygons puzzle, 56
POST indicates postcondition, 6
postcondition, *see also* program, 6, 15, 21
assertion labelled POST, 6
at the end of the program, 5
global, 127, 128
interpreting, 45
that depends on the precondition, 50, 58
use of auxiliary variable, 50, 58
`#` 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, **215**
precondition, *see also* program, 6, 15, 21
assertion labelled PRE, 6
default is `True`, 7, 8, 57, 62
global, 127, 128
`#` PRE `···`, 7
predicate
in logic, **214**
vs. predicate symbol, 214

predicate symbols
written in sans-serif, 214
primitive data types, *qv*
procedure calls
inlining, 78
program
checking
dividing a big programming problem into a tree of smaller ones, v, 8, 16, 17, 67
. . . 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, 47, 102
program components
`for`-loop vs. `while`-loop, 11, 241
assignment statement `x=` *exp*, 57–59
`break`, 21, 26∘, 45, 47, 51, 64, 74, 80, 134, 143, 152↓, 243
its interaction with invariants, 21, 243
conditional (`if`), 59–61
`continue`, 21, 243
its interaction with invariants, 243
`for`-loop, 11
`skip` (do nothing), 59
`while`-loop, 11
program examples
ATM
in Imperial currency, 51
many active concurrently, 120
binary search, 36–37, 47, 61, 72, 74, 91, 100, 102, 175
bubble sort, 63
circular structure
made from pointers, 166
critical section, 131, 135
date
calculating "today" from day number, 3, 21, 22, 32
calculation has no variant?, 15
Euclid's algorithm `gcd`, 50
exchange sort, 64
Fibonacci numbers, 83, 84
in log-time, 85–87
recursive version is inefficient, 83
`for`-loop vs. `while`-loop, 19, 242
garbage collection, 145–169
Good subsegment, 23
caterpillar analogy, 26
greatest common divisor `gcd`, 50
insertion sort, 64
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`, 48
linear search, 72, 74, 75, 77
linear-time exponential, 38
log-time exponential, 38, 84, 85
log-time Fibonacci, 83–89
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
maximum
of three values, 16
of four values, 62
of whole sequence, 20
maximum segment sum `mss`
in $O(N^3)$, 42
in $O(N^2)$, 43
in $O(N)$, 44, 47, 48
mutual exclusion, 131, 132∘
Peterson's mutual-exclusion algorithm, *see* concurrency, *qv*
quotient and remainder (finding), 16, 54
rectangular-matrix puzzle, 53
search
binary, *qv*

linear, *qv*
set (concrete)
represented by fixed-size array, 94∘
sorting
three variables, 59, 62
four variables, 63
summing a sequence, 4, 5, 11, 14, 18, 50
swap two variables, 61
train-carriages puzzle, 54
unrolling a loop, 7, 9, 10
vertical format for assertions, 9
`while`-loop vs. `for`-loop, 19
program exercises
binary search, 45–81
Boolean arithmetic, 82
bubble sort, 63
checking conditionals, 62
circular reasoning, 130
continued fraction, 20
coupling invariants, 89, 108, 114, 135
critical section, 125, 135
date
calculating "today" from day number, 22, 45
calculation has no variant?, 21
empty sequence or subsegment
sum of, 47
"everyday" errors, 17
exchange sort, 64
flowchart, making one, 21
`for`-loop, assertions for, 47
global correctness, 167
index out of bounds, 17
infinity ∞ used as value in program, 48
insertion sort, 64
intermediate assertion, 18
iterating
down, 45
iterating down, 18
length of longest run `lr`, 48
local
correctness, 167
lock-free implementation, 144
maximum
of empty sequence, 20
of three values, 16
of four values, 62
of whole sequence, 16, 20
maximum segment product `xsp`, 48
non-empty-segment sum, 48
program algebra, 47, 102
quadratic complexity, 48
rounding up or -down, 46
safety, 143
sequential composition of programs
how to check, 17, 61
shortest run `sr`, 48
sorting, 45, 63
three variables, 62
four variables, 63
a whole sequence, 63, 64
spin lock, 124
stable condition, 144
summing a sequence, 18
swap two variables, 61
termination, 56
dictionary order, 55
lexicographic variant, 55
lexicographic variant sometimes can be reduced to simple, 55
multi-set ordering, **56**
polygons puzzle, 56
quotient and remainder (finding), 54
social-distance puzzle, 53
train-carriages puzzle, 54
well founded sets, 55, 56
train-carriages puzzle, *see also* termination
uninitialised variable, 47
variants (and finding them), 166
decreasing, 61
for infinite loop, 125
increasing, 22
program review, 4
proof by induction, 112↓
proof, hint in, *qv*
propositional connective, 215
prototype
"quick and dirty" refined later to a more sophisticated implementation, 91
pseudocode
does not allow rigorously reasoning, 78
more than, 77–78, 92, 105, 110
Python, 3

caveats — take care, 241
`font`, **3**
mentioned mainly in footnotes, 3↓
used in this text, v, 241
uses reference counting for garbage collection, 148↓

quadratic complexity, 48
quantification
body of, **221**
existential (∃), **222**
nested, **223**
typed, **222**
universal (∀), **221**
untyped, **222**
"quick and dirty", *see* prototype
quick fixes
avoid them, 35
quotient and remainder (finding), 54
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, 53
recursion, 83
refinement
data-, **100**
equality is a special case, 100
guaranteed by adding data-type invariants, 100
of code, **100**
proper, **100**
reflexive, **52**
`release()` in Python, 121↓
`remove()` element from set, 101
`repeat`-loop, 152↓
invariant need not be true on initialisation, 152, 155
requirements, 5, 7, **15**, 21
examples, 24
indicated by "`# -`", 5
should not be too detailed or technical, 15, 105, 106
vs. specification, 16
whether to buy a thing, vs. how to use it, 106
rolling-up a loop, 11
root node, **147**
rounding up or -down, 46
Rubik's Cube, 53
rule of inference, *see* Modus Ponens

safety, *see* concurrency
satisfaction modulo theories (*SMT*), 177↓
Scanner, **147**
CS **Scholten**, 149↓
search
binary (logarithmic), *qv*
linear, *see* program examples
(sub-)segment, *see* sequence
sentinel, **75**, 75–77
sequence concatenation `+`, **19**, **73**, 74
sequence or subsegment
`A[0:N]`, 4
`A[low:high]` is inclusive/exclusive, **4**, 29, 36, 45
empty, *qv*
is `A[n:n]`, 45, 47
"falling off the end", 75
sequential composition of programs
how to check, 17, 61, **193**
in Python, in *C*, 193↓
sequential programs (not concurrent), 128
serial access (to a class), 123
serial use (in a concurrent system), 120, 121
set (concrete)
represented by fixed-size array, 94∘
set data-type
abstract, 77
compared with concrete, 97
as primitive
difference `-`, **74**
membership ∈, **74**
union ∪, **73**
concrete
represented by fixed-size array, 92, 94∘
represented by sequence, 73–75, 79, 91
empty `{}`
written `set()` in Python, 73
shillings, *see* Imperial
shortest run `sr`, **48**, *see also* program exercises, 244
$\sum$ means "the sum of", **5**
simple formula, *see* formula, atomic
simplify postcondition, 38–41

`skip`
called `pass` in Python, 60↓, 192↓
checking by implication, 59, 61, **192**
default `else` of conditional, 61, 121
does nothing, 59
SMT, *see* satisfaction modulo theories
JLA **van de Schnepscheut**, 149↓
social-distance puzzle, *see also* termination, 53
sorting, 45, 63
three variables, 59, 62
four variables, 63
a whole sequence, 63, 64
bubble sort, 63
exchange sort, 64
insertion sort, 64
inefficient, *see* bubble sort, exchange sort, insertion sort
soundness, **233**
space leak, *see* garbage
specification, *see also* program, 5–9, 15, 40↓, 159
as contract, 7
examples, 24
how to check, **198**
implementing, 167
satisfying, 15
used as program text, 110–112, 114, 159∘, 159, 167
vs. `assert` then `assume`, 114
vs. requirements, 16
written `x:[`*pre*`,`*post*`]`, **111**
written in mathematical language, iv
spin lock, 124, 133, 134
split a conjunct, 31–33, 40, 153
spot (·) in quantification, **221**
`sr`, shortest run, **48**
stable condition, 143, 144
compare globally correct, global invariants, 144
starvation, *see* concurrency
state space of program
maps variables to values, 213
static class, *see* object orientation
EFM **Steffens**, 149↓
N **Stenning**, 161↓
structural variant
sometimes can be reduced to simple, *see also* termination, 56
structured datatypes, 52
subscript out of range, *see* "everyday" errors
substitution
explains quantifiers, 229
for checking assignment statements, *qv*
into formula (notation for), 229
into quantified formulae, 230
kept separate from reasoning about assertions themselves, 57
variable capture, 230
sum of no terms is zero, 32
summing a sequence, 4, 5, 11, 14, 18
loop variant for, 50
swap two variables, 58, 61
swinging a pointer, *see* garbage collection
synchronised methods, 123
syntax errors, iii

term, in logic, **213**
termination, 27, 49–53, 56, 114, 153, 155, 157, 160, 195, 196
dictionary order, 55
lexicographic variant, 55
sometimes can be reduced to simple, 55
multi-set (also bag) ordering, **56**
polygons puzzle, 56
quotient and remainder (finding), 54
rectangular-matrix puzzle, 53
social-distance puzzle, 53
structural variant
sometimes can be reduced to simple, 56
train-carriages puzzle, 54
well founded sets, 55, 56
testing is necessary, v, 173
there exists ∃, *see* quantification
thread, **119**, 120–121
guarantees execution order of its components, 121, 127
time complexity (efficiency)
logarithmic $O(\log N)$, 38, 39, 84
linear $O(N)$, 39, 40, 43, 44, 47, 83
$O(N \log N)$, 40
quadratic $O(N^2)$, 43, 47, 48
cubic $O(N^3)$, 42

exponential $O(c^N)$, 83, **88**
today, *see* date
total correctness, **49**
total order, **52**
train-carriages puzzle, 54
transitive, **52**, 217
dividing a big programming problem into a *tree* of smaller ones, 9, 16
True
written True as value of logical formula in a state, **214**
"true"
used as English word, 208
written `true` as a logical constant symbol, **214**
written `True` as Boolean in Python, **207**, 214
written True as value of logical formula in a state, **214**, 220
`true` describes all states, **214**
unit of $\wedge$, 226
zero of $\vee$, 226
`True` as default precondition, *qv*
Trustworthy Systems Group, vii
truth table, 215
"tweaking", *see* efficiency and neatness for correctness (really?), 24

`undo()` procedure for Mean Calculator, 108
uninitialised variable, 47
unit of an operator, 226
universal quantification ($\forall$), *qv*
universally valid, 228
The University of New South Wales, vii
unlock
twice in a row, 122
unrolling a loop, 7, 9, 10, *qv*
ur-polygon, *qv*

"valid" means correct, in logic, 212↓
variable
bound, *qv*, **222**
free, *qv*
in logic written x, 213
in programs written `x`, 213
list of, in quantification, 221
variable capture, *see* substitution
variants (and finding them), **14**, 49–53, 166
bounded above, 22
bounded below, 32
can never be negative, 14
decreasing, 61
dictionary order, 51, 55
for infinite loop, 125
for recursion, 83, 88
increasing, 14, 22, 50, 81
vs. decreasing, 50
lexicographic, **51**, 51–52
multi-set ordering, 56
must strictly decrease, 14, 32, 50
simple, 49–54
structural, 52
use auxiliary variable with, to check decrease, 58, 61, 64
well founded, 52
when there can't be one, 15
verifier, *see* checking programs automatically
vertical format for assertions, 9
vocational skill, 65↓

well formed formula (*wff*), **233**
well founded sets, **52**, *see also* termination, 55, 56
wff, well formed formula, **233**
What does this do? comment, *qv*
"What happens?" vs. "What is true?", 127
What's true here? comment, *qv*
`while`-loop
vs. `for`-loop, 241
how to check partial correctness, **194**
how to check termination, **195**
how to check total correctness, **196**
with `break` statements, **195**
`while`-loop vs. `for`-loop, 19
white nodes, *see* garbage collection
M **Woodger**, 161↓
"working" program, 15

`x:[`*pre*`,`*post*`]`, specification, **111**
used as program text, 111

`yield()`, 124

zero of an operator, *see* unit, 226
Zune, *see* program examples (leap-year bug)