1
Introduction

We forget how incredible the computer is. The modern computer executes billions of operations *each second*, every one of which must work perfectly – accurately performing the right operation at each and every cycle. How is this even possible? How can we, mere humans with our cognitive limitations, manage to build devices that work at this pace with this level of fidelity? Each of the billions of instructions executed per second on a modern computer is another detail to be managed. How can we gain control over this mass of detail?

In Jorge Luis Borges's 1944 short story *Funes the Memorious*, the protaganist, Ireneo Funes, experiences what it is like to perceive the world at this level of streaming detail. After being thrown from a wild horse and severely crippled, he develops a prodigious memory. He recalls, perfectly and instantaneously, every moment of his life.

> He knew by heart the forms of the southern clouds at dawn on the 30th of April, 1882, and could compare them in his memory with the mottled streaks on a book in Spanish binding he had only seen once.... Two or three times he had reconstructed a whole day; he never hesitated, but each reconstruction had required a whole day. (Borges, 1962)

Yet, each of his memories was individual, disconnected, divorced of any higher structural patterns. Borges relates,

> With no effort, he had learned English, French, Portuguese and Latin. I suspect, however, that he was not very capable of thought. To think is to forget differences, generalize, make abstractions. In the teeming world of Funes, there were only details, almost immediate in their presence.

Without abstraction, there are only details. And it is through abstraction – forgetting differences, generalizing – that we can get control of the sheer daunting complexity of controlling a computer.

What is abstraction? ABSTRACTION is the process of *viewing a set of apparently dissimilar things as instantiating an underlying identity.* Funes sees a field of flowers, hundreds of blooms. To him, they are each

individuals, but to the botanist, these apparently dissimilar individuals are all instances of a type, the genus Tulipa, the tulips. By capturing innumerable individual plants into a hierarchy of abstract families, genera, and species, the bewildering complexity of plant life on the planet becomes more manageable.

Programming computers is a battle against the sheer daunting complexity of the task. The chief weapon in the battle is abstraction. The first objective of this book is to introduce you to a broad variety of abstraction mechanisms and their uses, providing you with an appropriate armamentarium. The second objective is to open your eyes to the beauty that computer programming can manifest when those tools are elegantly applied.

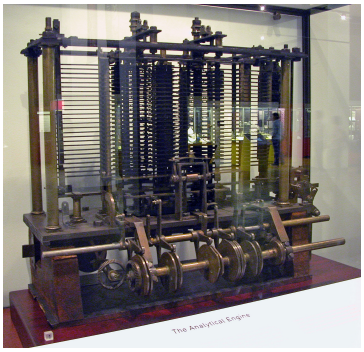


Figure 1.1: A model of a part of Charles Babbage's analytical engine, intended for the calculation of tables of mathematical functions such as the trigonometric functions like sine and cosine, the Bernoulli numbers, or logarithms, as in Figure 1.2 below.

You are already familiar with some of the primary abstraction mechanisms used in programming computers. (I assume throughout this book that you've had some experience programming computers using an imperative programming language, of the sort, for instance, acquired in Harvard's CS50 or CS50x course.)

Let's take as an example the problem of generating a table of logarithms. The choice is not random. The building of tables of mathematical functions like the logarithm was the motivating task for the earliest computer designs, those of Charles Babbage in the 1820s and 1830s (Figure 1.1). In the margin (Figure 1.2) is the beginning of such a table.

x	$log_2 x$
1	0.0000
2	1.0000
3	1.5850
4	2.0000
	...

Figure 1.2: A small table of logarithms

A program to print out this kind of table might look like this:

```
printf "1 0.0000\n";
printf "2 1.0000\n";
printf "3 1.5850\n";
printf "4 2.0000\n" ;;
```

and when the program is executed, it prints the table:

```
# printf "1 0.0000\n";
# printf "2 1.0000\n";
# printf "3 1.5850\n";
# printf "4 2.0000\n" ;;
1 0.0000
2 1.0000
3 1.5850
4 2.0000
- : unit = ()
```

(For the moment, the details of the language in which this computation is written are immaterial. We'll get to all that in a bit. The idea is just to get the gist of the argument. In the meantime, you can just let the code waft over you like a warm summer breeze.)

Now of course this code is hopelessly written. Why? Because it treats each line of the table as an individual specimen, missing the abstract view. The first step in viewing the lines abstractly is to note

that they are actually instances of an underlying uniformity: Each string is of the form of an integer (call it x) and the log (with base 2) of x. They are instances of the underlying pattern

```
printf "%2d %2.4f\n" x (log2 x);
```

for each of several values of the variable x. (Again, the details of the language being used are postponed, but you hopefully get the idea.) This mechanism, the STATE VARIABLE, is thus a mechanism for abstraction – for making apparently dissimilar computations manifest an underlying identity. To take full advantage of this type of variable, we'll need to specify the sequential values, 1 through 4 say, that the variable takes on.

```
for x = 1 to 4 do
  printf "%2d %2.4f\n" x (log2 x)
done
```

Like Monsieur Jourdain, who discovered he'd been speaking prose his whole life, you've been using abstraction mechanisms without realizing it. Without them, programming is impossible.

This particular style of programming, imperative programming, is undoubtedly most familiar to you. Its most basic abstraction mechanisms are the state variable and the loop. It is the style seen in some of the earliest, most influential programming languages, from FORTRAN to the ALGOL family of languages, to C, to Python, and beyond. And it is the style of programming captured by the first universal model of computation, the TURING MACHINE of Alan Turing (Figure 1.3).

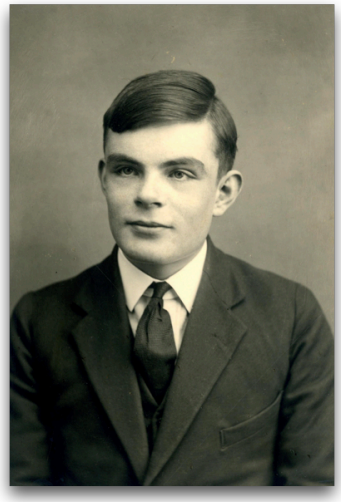


Figure 1.3: Alan Turing (1912–1954), whose Turing machine provided the first universal model of computation, based on imperative programming notions of state and state change. Turing is rightfully credited with fundamental contributions to essentially all areas of computer science: the theory of computing, hardware, software, artificial intelligence, computational biology, and much more. His premature death by suicide at 41 after undergoing "therapy" at the hands of the British government following his conviction for the "crime" of homosexuality is certainly one of the great intellectual tragedies of the twentieth century. (The British government got around to apologizing for his treatment some 50 years later.) The highest award in computing, the Turing Award, is appropriately named after him.

But there are many other abstraction mechanisms than state variables and loops, underpinning many other programming paradigms than imperative programming, and allowing many other ways of designing computations. It is the goal of this book to introduce several such abstraction mechanisms, provide practice in their use and application, and thereby open up a broad range of programming possibilities not otherwise available.

An especially important abstraction mechanism is the function, a mapping from inputs to outputs. The idea of the function gives its name to the paradigm of *functional programming*, and we will begin with functions and functional programming ideas. But functional programming is only one of several paradigms that we will discuss.

1.1 An extended example: greatest common divisor

By way of example of the distinction between imperative and functional programming, consider the very practical question of tiling a bathroom floor of size 28 by 20 units. We can tile such a floor with tiles

that are 2 by 2, since both 28 and 20 are evenly divisible by 2, but 3 by 3 tiles don't work, since neither 28 nor 20 are divisible by 3. If we want to use the fewest tiles, it would be useful to know the largest number that divides both dimensions evenly, their GREATEST COMMON DIVISOR (GCD).

Here is how we might program a calculation of GCD in an imperative style:

```
let gcd_down a b =
  let guess = ref (min a b) in
  while (a mod !guess <> 0) || (b mod !guess <> 0) do
    guess := !guess - 1
  done;
  !guess ;;
```

This procedure works by counting down from the smaller of the two numbers, one by one, until a common divisor is found. Since the search for the common divisor is from the largest to the smallest possibility, the greatest common divisor is found.

In the functional style, this same "countdown" algorithm might be coded like this:

```
let gcd_func a b =
  let rec downfrom guess =
    if (a mod guess <> 0) || (b mod guess <> 0) then
      downfrom (guess - 1)
    else guess in
  downfrom (min a b) ;;
```

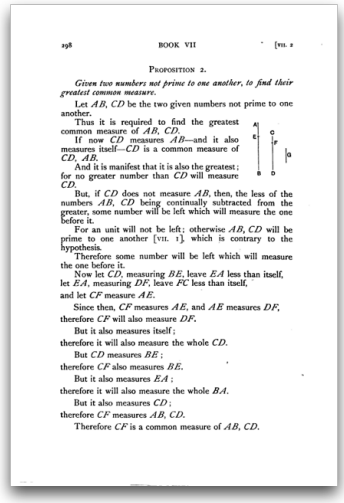
Here, in the context of calculating the GCD of `a` and `b`, a new function `downfrom` is introduced to check a particular guess of the GCD of the two numbers. The `downfrom` function takes an input `guess` and checks whether it is the GCD of `a` and `b`. If so, the output value of the function is the guess `guess` itself, but if not, a one-smaller guess is tried. Having defined this counting-down function, the calculation of the GCD itself proceeds just by guessing the minimum of the two numbers.

You may find unusual some of the properties of this latter implementation of what is essentially the identical algorithm – counting down one by one from the minimum of the two numbers until a common divisor is found. First, there are no overt loops, and no assignments to variables that change the state of the computation by changing the value of a variable. It's just functions and their application. Second, the function `downfrom` defined in the code appeals to `downfrom` itself as part of the calculation of its output. It is defined by RECURSION, that is, in terms of itself. Such functions are *recursive*, and when they invoke themselves for a computation are said to *recur*.[1] You may wonder whether this is quite kosher. Isn't defining something in terms of itself a bad idea? But in this case at least, the definition works

[1] Not *recurse* please. To recurse is to curse again, not the kind of thing a program – or a person – should be doing.

fine, because the value of `downfrom guess` depends not on the value of `downfrom guess` itself but of `downfrom (guess - 1)`, a different value. This may itself depend on `downfrom (guess - 2)`, and so on, but eventually one of the inputs to `downfrom` will be a common divisor, and in that case, the output value of `downfrom` does not depend on `downfrom` itself. The recursion "bottoms out" and the GCD is returned.

This style of programming – by defining and applying functions – has a certain elegance, which can be seen already in the distinction between the two versions of the GCD computation already provided. But as it turns out, the algorithm underlying both of these implementations is a truly bad one. Counting down is just not the right way to calculate the GCD of two numbers. As far back as 300 BCE, Euclid of Alexandria provided a far better algorithm in Proposition 1 of Book 7 (Figure 1.4) of his treatise on mathematics, *Elements*. Euclid's algorithm for GCD is based on the following insight: Any square tiling of a 20 by 28 area will tile both a 20 by 20 square and the 8 by 20 remainder. More generally, any square tiling of an a by b area (where a is greater than b) will tile both a b by b square and the b by $a - b$ remainder. Thus, to calculate the GCD of a and b, it suffices to calculate the GCD of b and $a - b$. Eventually, we'll be looking for the GCD of two instances of the same number (that is, a and b will be the same; we'll be looking to tile a square area) in which case we know the GCD; it is a (or b) itself. Figure 1.5 shows the succession of smaller and smaller rectangles explored by Euclid's algorithm for the 20 by 28 case.



298 BOOK VII [VII. 2

PROPOSITION 2.

Given two numbers not prime to one another, to find their greatest common measure.

Let AB, CD be the two given numbers not prime to one another.

Thus it is required to find the greatest common measure of AB, CD.

If now CD measures AB—and it also measures itself—CD is a common measure of CD, AB.

And it is manifest that it is also the greatest; for no greater number than CD will measure CD.

But, if CD does not measure AB, then, the less of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it.

For an unit will not be left; otherwise AB, CD will be prime to one another [VII. 1], which is contrary to the hypothesis.

Therefore some number will be left which will measure the one before it.

Now let CD, measuring BE, leave EA less than itself, let EA, measuring DF, leave FC less than itself, and let CF measure AE.

Since then, CF measures AE, and AE measures DF, therefore CF will also measure DF.

But it also measures itself; therefore it will also measure the whole CD.

But CD measures BE; therefore CF also measures BE.

But it also measures EA; therefore it will also measure the whole BA.

But it also measures CD; therefore CF measures AB, CD.

Therefore CF is a common measure of AB, CD.

Figure 1.4: Proposition 1 of Book 7 of Euclid's *Elements*, providing his algorithm for calculating the greatest common divisor of two numbers.

An initial presentation of Euclid's algorithm is this:

```
let rec gcd_euclid a b =
  if a < b then gcd_euclid b a
  else if a = b then a
  else gcd_euclid b (a - b) ;;
```

Now, in the case that $a = b$, were we to continue on one more round of checking the GCD of b and $a - b$, the difference $a - b$ would simply be 0. Thus, we can check for this condition instead.

```
let rec gcd_euclid a b =
  if a < b then gcd_euclid b a
  else if b = 0 then a
  else gcd_euclid b (a - b) ;;
```

We can simplify further. When subtracting off b from a, the remainder may still be greater than b, in which case, we'll want to subtract b again, continuing to subtract b until, eventually, the remainder is less than b. Thus, instead of using the difference $a - b$ as the new second argument of the recursive call, we can use the remainder a mod b.

```
let rec gcd_euclid a b =
  if a < b then gcd_euclid b a
```

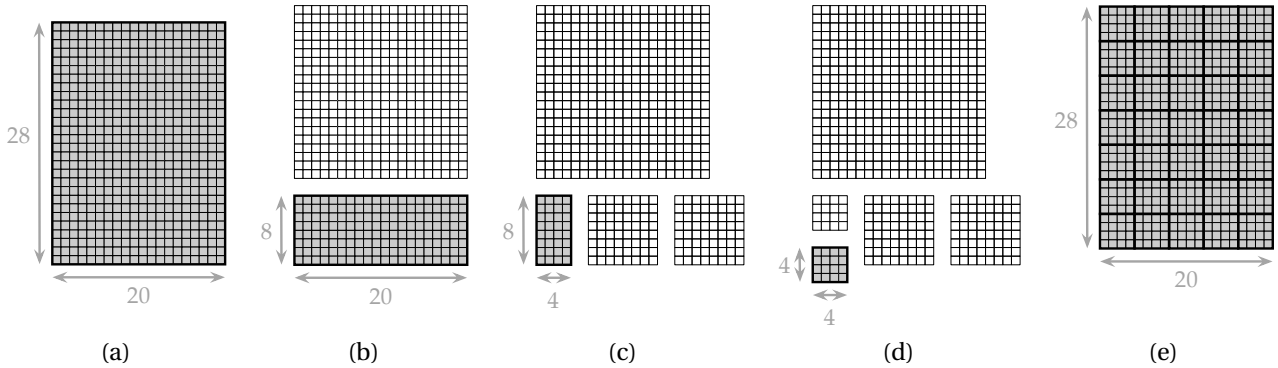


Figure 1.5: Euclid's algorithm for GCD starting (a) with a 20 × 28 rectangle to be tiled. Removing the 20 × 20 square (b) leaves a 20 × 8 remainder to be tiled. From that rectangle, we remove, successively, two 8 × 8 squares (c), leaving a 4 × 8 remainder. Finally, removing a 4 × 4 square (d) leaves a 4 × 4 square, the largest square that can tile the whole (e).

```
  else if b = 0 then a
  else gcd_euclid b (a mod b) ;;
```

Finally, notice that if $a < b$, then the values b and a mod b are just b and a, respectively – exactly the values we want to use for the recursive call in that case. We can therefore drop the test for $a < b$ entirely.

```
let rec gcd_euclid a b =
  if b = 0 then a
  else gcd_euclid b (a mod b) ;;
```

This is EUCLID'S ALGORITHM. Compare it to the countdown algorithm above. The difference is stark. Euclid's method is beautiful in its simplicity.

It is also, as it turns out, much more efficient. This can be determined analytically or experienced empirically.

1.2 Programming as design

Euclid's algorithm for GCD shows us that there is more than one way to solve a problem, and *some ways are better than others.* The dimensions along which programmed solutions can be better or worse are manifold. They include

- succinctness,
- efficiency,
- readability,
- maintainability,
- provability,
- testability,

and, most importantly but ineffably,

- beauty.

Computer programming is not the only practice where practitioners may generate multiple ways of satisfying a goal, which can be evaluated along multiple independent and perhaps conflicting metrics. Architects, engineers, illustrators, industrial designers may generate wildly different plans in response to a client's constraints and desires. All live in a space of possibilities from which they choose solutions that vary along multiple, often competing, criteria.

What all of these practices have in common is DESIGN – the navigation of a space of options, generated by applicable tools, in search of the good, as measured along multiple dimensions. In the case of computer programming, the tools are exactly the abstraction mechanisms provided by a programming language.

A crucial consideration in teaching programming from this perspective is what abstraction mechanisms to concentrate on, as these define the space of options within which we can navigate. As discussed above, the most important of these abstraction mechanisms is the function. In addition to being a fantastic method for abstracting computation (which will become clear some time around Chapter 8), functions also serve as a platform upon which many other abstraction mechanisms can be deployed and combined. It may be difficult at first to see the incredible utility of the function as a unifying abstraction mechanism, but hopefully, as you see more and more examples of their use in combination with other techniques, you will come to appreciate the function's centrality in the design of programs.

Figure 1.6: Princeton professor Alonzo Church (1903–1995), inventor of the lambda calculus, the foundation of all functional programming languages; PhD adviser of Alan Turing.

Indeed, functions and their application are such a powerful computational tool that they constitute, by themselves, a complete universal computational mechanism. The Princeton mathematician and logician Alonzo Church (1936) developed a "calculus" of functions alone, the so-called LAMBDA CALCULUS (see Section B.1.4), a logical system that included functions and their application and *literally nothing else* – no data objects or data structures of any kind, neither atomic (like integers) nor composite (like lists); no mutable state (like variable assignment); no control structures (like conditionals or loops). Astoundingly, Turing (1937) was then able to show that anything that can be computed by his universal model of computation, the Turing machine, can also be computed in Church's lambda calculus. Thus, the lambda calculus – comprised only of functions and their applications remember – is itself a universal model of computation. This argument for the universality of Turing's and Church's computation models is now known as the CHURCH-TURING THESIS. (The close connection

between the lambda calculus and the Turing machine mirrors the close relationship between Church and Turing; Church was Turing's PhD adviser at Princeton.)

In this book, we concentrate on the following abstraction mechanisms, listed with the style of programming they are associated with:

Abstraction	*Programming paradigm*
functions	functional programming
algebraic data types	structure-driven programming
polymorphism	generic programming
abstract data types	modular programming
mutable state	imperative programming
loops	procedural programming
lazy evaluation	programming with infinite data structures
object dispatch	object-oriented programming
concurrency	concurrent programming

Table 1.1: Some abstraction mechanisms and the programming paradigms they allow.

Of course, there are many other abstraction mechanisms and programming paradigms, but these should both give you a good sense of the importance of a variety of abstractions and provide an excellent base on which to build.

As with any design practice, computer programming is best learned by seeing a range of examples of the space of options – examples that are better or worse along one dimension or another – with attention paid to the process of developing, modifying, and improving such solutions. For that reason, we will often show computer programs being built up in stages and being modified to demonstrate alternative designs (as we did with the GCD example above), and programming problems will be revisited as new abstraction mechanisms open further parts of the design space. You may find the multiple variations on a theme redundant – as indeed they are – but we know of no better way to get across the idea of programming as a design practice than the careful development and exploration of a significant program design space.

Figure 1.7: Robin Milner (1934–2010), developer of the ML programming language, from which OCaml derives, the first functional language with type inference. He received the Turing Award in 1991 for his work on ML and other innovations.

1.3 *The OCaml programming language*

In order that we can introduce multiple abstraction mechanisms and programming paradigms with a minimum of programming language detail, we use a multi-paradigm programming language called OCAML. (The examples above were written in OCaml.) OCaml is a member of the ML family of programming languages first developed

at University of Edinburgh by Robin Milner (Figure 1.7) in the 1970's. The OCaml dialect of ML itself was developed at the French national research lab Institut National de Recherche en Informatique et en Automatique (INRIA), where it continues to be developed and maintained. OCaml is a multi-paradigm programming language in that it provides support not only for functional programming, but also imperative programming, object-oriented programming, and all the other mechanisms and paradigms listed in Table 1.1.

OCaml is especially attractive from a pedagogical standpoint because it provides these capabilities on the basis of a relatively small foundation of well-designed orthogonal primitive language constructs, so that programming concepts can be introduced and experimented with, without the need for learning a huge set of syntactic idiosyncrasies. Nonetheless, as with learning any new programming language, it will take a bit of getting used to the ideas and notations of OCaml, and in fact getting practice with learning new notations is a useful skill in its own right.

I emphasize that this book is not a book about OCaml programming. (For instance, this book doesn't pretend to present the language comprehensively, instead covering only those parts of the language needed to present the principles being taught. For that reason, you will want to get at least a bit familiar with the reference documentation of the language.) Rather, it is a book about the role of abstraction in the design of software, which uses the OCaml language as the medium in which to express these ideas. But in order to get these ideas across, we need *some* language, and it turns out that OCaml is an ideal language for this pedagogical purpose. Of course, we'll have to spend some time going over the particularities of the OCaml language, which may seem odd mostly because of their unfamiliarity. The text may have a bit of a disjointed quality to it, bouncing back and forth between details of OCaml and higher-level concepts. But the time spent learning the details of the language isn't time wasted. It pays off in lessons that generalize to any programming you will do in the future. You may discover, like many do, that once you've gained some proficiency with OCaml, you find its charms irresistible, and continue to use it (or its close derivatives like Microsoft's F#, Facebook's Reason, Apple's Swift, or Mozilla's Rust) when appropriate – as many companies including those mentioned do. But whether you continue to program in OCaml or not, the patterns of thinking and the sophistication of your understanding will be the payoff of this process, translatable to any programming you'll do in the future. In fact, the market for software developers reflects this payoff as well. As shown in Figure 1.8, the market rewards software developers fluent in the kinds of technologies and ideas fea-

tured in this book; their salaries are substantially higher on average. Although such pecuniary benefits aren't the point of this book, they certainly don't hurt.

1.4 Tools and skills for design

The space of design options available to you is enabled by the palette of abstraction mechanisms that you can fluently deploy. Navigating the design space to find the best solutions is facilitated by a set of skills and analytic tools, which we will also introduce throughout the following chapters as they become pertinent. These include more precise notions of the syntax and semantics of programming languages, facility with notations, sensitivity to programming style (see especially Appendix C), programming interface design, unit testing, tools (big-O notation, recurrence equations) for analyzing efficiency of code. Having these tools and skills at your disposal will add to your computational tool-box and stretch your thinking about what it means to write good code. I expect, based on my own experiences, that learning to develop, analyze, and express your software ideas with precision will also benefit your abilities to develop, analyze, and express ideas more generally.

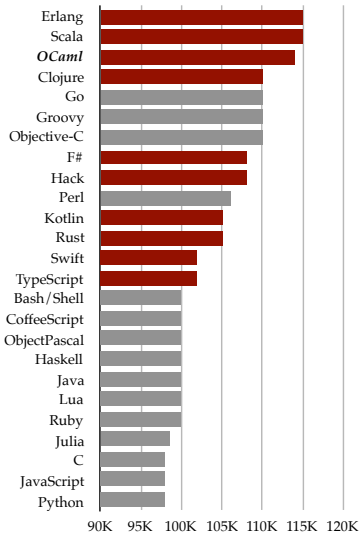


Figure 1.8: United States average salary by technology, from StackOverflow Developer Survey 2018. Highlighted bars correspond to technologies in the typed functional family.