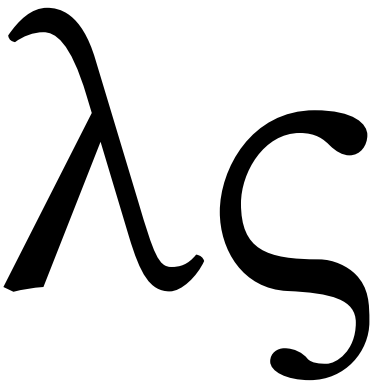

Lambdas and Logos

On Writing Elegant Code

Lukáš Hozda

“haskal”
– Alcuin @scheminglunatic

“The art of programming is the art of organizing complexity.”
– Edsger Dijkstra

“Humanity is impure yet contains the seeds of perfection”
– Dank Herbert @dank_herbert

“Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.”
– Alan Perlis

“A language that doesn’t affect the way you think about programming is not worth knowing.”
– also Alan Perlis

“Less is sometimes more”
– my grandma

Chapter Zero

Contents

CONTENTS

Chapter One

Preface

This book is for the junior or intermediate programmer, and for any other interested party. To be more specific, you will gain the most from this book if you fall into one or more of the following categories:

- You are a **university student**, or **recent graduate**. You were taught how to program at your school, and you have little experience writing software that interacts with the **real world**.
- You are **self-learned**, you write practical projects, but your programming knowledge is **almost completely practical**, driven by need of those projects, and you didn't delve much into the theory of things.
- You are a **fresh junior developer**, working in a company, and you are painfully discovering that maybe there is more to software development than you thought, and you now have to grapple with writing software that goes into production, is read and critiqued by others, and has to be maintained over a long period of time, and boy, you sure as hell have no experience doing that.
- You, like me, are an eternal **pursuer of beauty and elegance** in the things you do, and you consider programming to be a creative act that can have aesthetic merits

I think that I am getting ahead of myself with the last point, so let's take it back from the beginning.

My name is Lukáš Hozda, I am a programmer. I work in Braiins Systems s.r.o, starting as an Embedded SW Engineer, now a SWE Methodologist. This is a role that puts me somewhere between a software engineer, an educator/mentor, a public speaker, and a recruiter.[1] I first started programming when I was six or seven years old with Borland's Turbo Pascal version 5.5. By then, Pascal was already very much out of fashion, I was born in 2000, and got the Turbo Pascal books as discarded hand-me-downs from the library my mom works as.

Even before that, I apparently exhibited interest in computers and technology. This was to the point that a doctor has speculated that I might be addicted, and my parents should not feed my addiction. And it is true I was mesmerized by

[1] Because of the wide range of my activities, it is somewhat difficult to categorize me in the company structure. Right now, I am filed under HR, which lets me jokingly call myself the "most technologically competent HR in the world".

the sheer possibility and versatility of computers.[2]

I think we have forgotten what miracle it is. You hold a device, and it can do almost everything. Computers have become the cornerstone of our civilization. We made them smaller, we made them bigger, we made them more generalized, and more specialized, we have a tendency to replace analog machinery with them, because it perhaps requires less brains to program a rice cooker with a sensors, than to design a computer-less mechanism to drive all of its functionality.

Increasingly, we are putting computers in appliances, smart home devices and whatever else you can think of. Better yet, following the Inception[3] school of thought, we put computers in your computer. In your desktop or laptop, you may have a graphics card - that's a computer in your computer. It has a processing unit (the difference being it has a lot of somewhat specialized, weaker cores, as opposed to your CPU, which has fewer, stronger, more generalized cores), its own RAM, its own IO, and it can even execute code.

In your CPU, there is a tiny additional computer with its own memory, CPU and IO - for Intel, the Intel Management Engine, for AMD, the Platform Security Processor. Being a certified hater, and heavy classic ThinkPad fan, I am more familiar with turning off the IME.[4] This computer inside your CPU is running a Unix-like operating system called MINIX[5], and it can talk to the internet, and see absolutely everything going on in your computer.

But all of these computers are unified by one thing - they are running programs. That's why we invented computers in the first place, so that we can design algorithms, implement them, and run them.

And, despite the efforts of Large Language Models, someone still has to write those programs. That's the job of us, programmers. And I love programming, and I mean the act itself. To me, programming is one of the ultimate creative activities, and I love exploring it. Over the years, I have experimented and used

[2]Ironically, my parents' attempts to limit my computer access, and not buy me any then-current-gen electronics turned me into a MacGyver type character, and I would learn a lot being an online pirate and compensating for the shortcomings and lack of performance of the hardware I had access to.

[3]Great movie, by the way

[4]I consider it to be a backdoor

[5]Secretly making MINIX one of the most widespread desktop operating systems in the world.

with over a hundred programming languages, tried different approaches, and paradigms, all in pursuit of the perfect fit. Nowadays, I mostly write Rust (being an early adopter), Common Lisp and Scheme. By the end of this book, you will probably understand why. :)

In other words, I am on an eternal quest of finding the best ideas, and finding solutions tailored to my opinions, and striving to write the most elegant code possible. And I am a teacher, and I want to share what I have learned to you, dear reader, so that you may write code that is more beautiful.

This book is also a love letter to the many incredible programmers from the past, and their ideas. Ideas, which should be preserved and remain in working memory even today, which may be decades since their initial inception.

Lukáš Hozda,
rennaissance man

1.1 About how this book is written

This book is written as a series of topics that, I hope, flow freely from to another. We will start with a couple practical points to elaborate what's going on with elegant code, then talk about the importance of elegent code (and writing code elegantly – in a nice and ergonomic manner), and then we will explore both small scale and large scale practices that make programming more elegant. Naturally, we also explore what it being "elegant" means.

I, Lukáš Hozda, write in a meandering, exploratory manner. I change my voice often, depending on the subject matter, sometimes more serious, sometimes less.[6] Throughout the book, you will encounter a couple departures (that will always come back to the topic at hand), to provide you with additional context, either historical context, or context about alternatives that exist to what we are presenting.

In some cases, these tangents were short enough to be contained into footnotes, and so they are there. I encourage you to read the footnotes to grasp the whole idea of what am I trying to communicate.

This book also mentions and provides code examples in a number of programming languages and technologies, these include (in no particular order):

- Python
- C and C++
- Rust
- Common Lisp (including the Coalton language built on top of it by Robert Smith)
- Scheme
- Haskell
- Prolog
- Brainfuck
- Smalltalk
- Forth
- whatever else we decide to think of

Many more will be briefly mentioned in passing.

[6] Some parts of this book are also directly written by the co-author, Luukasa Pörtfors

You are not expected to know all of these languages. Hell, you are not expect to know most of these languages. Not knowing them should not stop you from picking up this book. It would be almost impossible to find people who "speak" Smalltalk, Lisp, Forth and Haskell at the same time anyway.

These languages to illustrate programming principles and open your eyes to new possibilites and philosophies that have shaped the history of programming and how we conceptualize what good programming is. We really want to broaden your horizons, and show you what's possible. That is, to see beyond the code and have basic understanding of the underlying ideas, so you can apply them elsewhere, too.

There is a parable with the notion of **monads**, you have probably heard about it, especially in the context of Haskell programming. A Haskeller comes up to you and says:

"A monad is just a monoid in the category of endofunctors, what's the problem?"

And you look at him as if he were an alien from a different dimension. Well, that's because they are terms that are very theoretical, and you probably don't need to know them for most everyday activities.

In reality, the concept of a monad isn't that difficult. It is just a wrapper over values that provides two actions:

1. You can put any value into this wrapper
2. You can chain operations that use wrapped values together

If you have used languages that utilize the following data types:

- Option or Maybe
- Result or Either (or both in case of Rust via the very popular `either` crate)
- Future or Promise
- List and Array[7]

[7] Some terms and conditions apply. A list is a monad if you have something like a `flat_map` operation, meaning you can map a list into a list of lists, and then concatenate these lists into one long list:

1. **[1, 2, 3, 4]** (the original list)
2. **[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]** (a list where we have **mapped** each element `N` into a list of integers from 1 to `N`)
3. **[1, 1, 2, 1, 2, 3, 1, 2, 3, 4]** (a list of lists that we have **flattened** into one contiguous list of integers)

Then you have used a monad. What is powerful about the idea of monads, from a theoretical standpoint is that now that we have defined something like this, we start seeing monads everywhere, and we can reason about operations in terms of monad.[8]

That is an example of how your horizon can be broadened by learning new concepts in one place (perhaps in the company of Haskellers), which you can recognize and deal with in other places then on.

Without a further ado, let's get into it.

[8] If you are curious about *somewhat* layman explanations of the other technical terms, then **monoid** means "something that has an identity element (like the empty list) and an associative binary operation (like list concatenation)". An **endofunctor** is something that can "map back to itself". If you know Rust or any other language with options, you know that you can `.map()` an `Option<T>` into an `Option<U>`. You will note that Either/Result does not have an identity element (Can you think of a default value if, you know neither the Ok/Left type or the Err/Right type?)

Chapter Two

The Art of Programming

Programming is a discipline that's almost a century old. Arguably much older if we count the efforts of Ada Lovelace, and much much older if we consider anything resembling an algorithm to be the origin of programming.

I think that it is fair to say that the notion of programming is much older than computers. We have a thousand different ways of describing algorithms, and in fact, whole programs, and the field has evolved immensely in the last century. We programmers seem to have a lot of opinions about how programming should be done, and have successfully turned the whole discipline into a question of clashing personal philosophies.

```
program HelloWorld;
begin
 WriteLn('Hello from Lambdas and Logos :)')
end.
```

Very similar to my first Pascal program.

Very often, we cannot objectively say, which solution to a given problem is the best one, although we can generally point out the very bad ones. Furthermore, none of us have the moral superiority of being a flawless programmer. Show me a programmer and I will show you someone who creates bugs.

Writing elegant code means writing code that makes it harder to create insidious bugs, by offering clarity and structure that make it easy to navigate, while still being an effective solution for the task at hand.

2.1 Lambdas and Logos

A programming language is a communication medium, just like a human language. It has a grammar, and a vocabulary, and just like you can convey a specific meaning by creating a story composed of sentences, you can solve an issue by creating a program composed of functions.

Who are we communicating with? The most obvious answer is with the computer. Unfortunately, the computer has no notion of humor, sarcasm, hyperbole, metaphor, implications, innuendos or any other departure from the most literal meaning of words, and so we have to be precised in what we say. You as a programmer might say "The computer isn't doing what it should!", but it does precisely what you told it to do.

If it doesn't do what you want it to do, then you need to phrase it correctly. This turns out to be difficult, especially if you are solving a difficult problem. But through grit, spit, and a whole lot of duct tape, we can do it.

What's worse is that we communicate not just with the computer, but with other programmers as well. You say: "This program is just for me, I wrote it by myself, for myself!" – you in 3 months is "other programmers".

We need to write programs that are:
- Understood by the computers
- Understood by the programmers

Here is a program in Brainfuck:

```
[(c) 2016 Daniel B. Cristofani
http://brainfuck.org/]

>>+>>>>>,[>+>>,]>+[--[+<<<-]<[<+>-]<[<[->[<<<+>>>>+<-]<<[>>+>[->]<<[<]
<-]>]>>>+<[[-]<[>+<-]<]>[[>>>]+<<<-<[<<[<<<]>>+>[>>>]<-]<<[<<<]>[>>[>>
>]<+<<[<<<]>-]]+<<<]+[->>>]>>]>>[.>>>]
```

The computer understands this program perfectly, how about you? I would have no idea what's going on.

Would it be better if we translated to a different language? Here is the same program in C:

```
void jonger(int beta[], int alpha, int omega) {
    int gamma[omega - alpha + 1];
    int theta = -1;
    gamma[++theta] = alpha;
    gamma[++theta] = omega;
    while (theta >= 0) {
        omega = gamma[theta--];
        alpha = gamma[theta--];
        int kappa = beta[omega];
        int lambda = (alpha - 1);
        for (int delta = alpha; delta <= omega - 1; delta++) {
            if (beta[delta] < kappa) {
                lambda++;
                int mu = beta[lambda];
                beta[lambda] = beta[delta];
                beta[delta] = mu;
            }
        }
        int mu = beta[lambda + 1];
        beta[lambda + 1] = beta[omega];
        beta[omega] = mu;
        int zeta = lambda + 1;
        if (zeta - 1 > alpha) {
            gamma[++theta] = alpha;
            gamma[++theta] = zeta - 1;
        }
        if (zeta + 1 < omega) {
            gamma[++theta] = zeta + 1;
            gamma[++theta] = omega;
        }
    }
}
```

I don't know how about you, but it still hard for me to understand what this program does. It is still something the the computer understands perfectly, but programmers, not so much. The issue is that the names of the variables and the function are very non-descriptive.

An adept programmer can now take a minute or a few to figure out that this is an iterative version of the **quicksort** algorithm. But the situation would be much improved if we used more useful names:

```
void quickSortIterative(int arr[], int low, int high) {
    int stack[high - low + 1];
    int top = -1;
    stack[++top] = low;
    stack[++top] = high;
    while (top >= 0) {
        high = stack[top--];
        low = stack[top--];
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        int pi = i + 1;
        if (pi - 1 > low) {
            stack[++top] = low;
            stack[++top] = pi - 1;
        }
        if (pi + 1 < high) {
            stack[++top] = pi + 1;
            stack[++top] = high;
        }
    }
}
```

This now resembles code that may be written by a student who is learning C for the first time. In a way, this is correct, but it is ugly. We have two problems to take care of:

- Code repetition and organization
- Visuals and documentation

For the first point, there are two instances, where all we are doing is swapping two values

```
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
//...
int temp = arr[i + 1];
```

```
arr[i + 1] = arr[high];
arr[high] = temp;
```

We can generalize it to a function called swap:

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

This clarifies the quicksort implementation by a good amount:

```
void quickSortIterative(int arr[], int low, int high) {
    int stack[high - low + 1];
    int top = -1;
    stack[++top] = low;
    stack[++top] = high;
    while (top >= 0) {
        high = stack[top--];
        low = stack[top--];
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(&arr[i], &arr[j]);
            }
        }
        swap(&arr[i + 1], &arr[high]);
        int pi = i + 1;
        if (pi - 1 > low) {
            stack[++top] = low;
            stack[++top] = pi - 1;
        }
        if (pi + 1 < high) {
            stack[++top] = pi + 1;
            stack[++top] = high;
        }
    }
}
```

Furthemore, if we recall the logic of **quicksort**, you will note that the step 3[9] is partitioning:

[9] Per Wikipedia, at least.

> Partition the range: reorder its elements, while determining a point of division, so that all elements with values less than the pivot come before the division, while all elements with values greater than the pivot come after it; elements that are equal to the pivot can go either way.

We have the partition right here:

```
int pivot = arr[high];
int i = (low - 1);
for (int j = low; j <= high - 1; j++) {
    if (arr[j] < pivot) {
        i++;
        swap(&arr[i], &arr[j]);
    }
}
swap(&arr[i + 1], &arr[high]);
int pi = i + 1;
```

This is the “divide” of the divide-and-conquer strategy **quicksort** employs. We can turn this into a function:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

Which improves how our quicksort function looks:

```
void quickSortIterative(int arr[], int low, int high) {
    int stack[high - low + 1];
    int top = -1;
    stack[++top] = low;
    stack[++top] = high;
    while (top >= 0) {
        // Pop high and low
        high = stack[top--];
        low = stack[top--];
        int pi = partition(arr, low, high);
        if (pi - 1 > low) {
```

```
            stack[++top] = low;
            stack[++top] = pi - 1;
        }
        if (pi + 1 < high) {
            stack[++top] = pi + 1;
            stack[++top] = high;
        }
    }
}
```

Which we can further improve by inserting some small comments and appropriate whitespace:[10][11]

```
void quickSortIterative(int arr[], int low, int high) {
    // create an auxiliary stack
    int stack[high - low + 1];

    // initialize top of stack
    int top = -1;

    // push initial values
    stack[++top] = low;
    stack[++top] = high;

    // keep popping from stack while it's not empty
    while (top >= 0) {
        // Pop high and low
        high = stack[top--];
        low = stack[top--];

        // get pivot position
        int pi = partition(arr, low, high);

        // if elements exist on left side of pivot
        if (pi - 1 > low) {
            stack[++top] = low;
            stack[++top] = pi - 1;
        }

        // if elements exist on right side of pivot
        if (pi + 1 < high) {
```

[10] It is arguable, how much commenting we need. Often, the answer I would provide is "as little as necessary". Overcommenting is a newbie mistake - we need to strike a balance. Formatting is very important also. We will discuss this in later on in this book

[11] Also, note that there is the stack data structure lurking around in this implementation. We should probably point it out and describe it, if we find more usecases for it in our program than just this simple quicksort

```
            stack[++top] = pi + 1;
            stack[++top] = high;
        }
    }
}
```

Using functions with descriptive names make your code more readable. The big idea is that we build up abstractions. These abstractions represent new actions that we can use to write program in a more descriptive manner, without having to worry about its implementation detail at every step of the way.

Let's take a slight theoretical detour by channeling our inner Dijkstra.[12] Unfortunately, I am unable to find this text, and so I am paraphrasing from memory, but Dijkstra essentially says that:

- Programs are processes composed of actions
- Action is a hopefully finite happening that has a defined effect
- Many happenings can be viewed as either a process or an action, depending on our interest in intermediate states
- Algorithms describe patterns of behavior using actions
- Algorithms are superior to simple step descriptions because they have connectives for **sequential**, **conditional** and **repetitive** composition of actions[13]
- The main strength of algorithms is that they can concisely express what many different happenings have in common. That is, you can describe how an infinite set of related scenarios are similar to one another

Building, or discovering abstractions is a very important part of every programmer's job. We are creating new primitive actions that we can compose into ever more complex processes. So, don't be shy to make functions and abstractions.

[12] He was a real one, no one could talk shit about programming languages (and programmers) quite like he did

[13] These correspond to code blocks, conditional statements and loops respectively. Which renditions of these are available in particular depends on your programming language of choice.

However, it is better to take a **more reactive than proactive approach** - you should create an abstraction because you identify something that is a general enough notion that it deserves to be described.

Preemptively creating abstractions that prove to be unnecessary increases development time, can harm performance[14], increase maintenance cost, and can increase cognitive load without adding value.

The last point is particularly important. Your abstractions should decrease cognitive load, not increase it. If you create an abstraction that is harder to understand than the unabstracted code, then it is a terrible abstraction.

Good programming follows "simplicity as a feature". The right amount of abstraction hides complexity when needed, but poor abstractions just add complexity. To paraphrase Einstein, **everything should be made as simple as possible, but no simpler.**

Simplicity also does not mean *stupidity*. The power of more elaborate programming languages lies in the fact that they let you design smarter abstractions that simplify programs effectively. Some programming languages presume that programmers are stupid[15], and take the power of creating generalized abstractions away from them.

This leads us to a very important point: **Programming languages matter.**[16]

Programming languages matter because they significantly influence how we model problems and design solutions. Different languages aren't just different syntaxes for expressing the same ideas - they embody different philosophies, different trade-offs, and different ways of conceptualizing computation.

Consider how differently you might approach a problem in C (thinking in terms of memory management and pointers), Haskell (thinking in terms of type transformations), Prolog (thinking in terms of logical relations), or APL (thinking in terms of array operations).

[14] Although for most usecases, you shouldn't sacrificity the clarity and readability of programs for performance. A clear and effective algorithm should always take precedence to microoptimizations.

[15] One such language's name rhymes with "No"

[16] From a certain point of view, that is.

This influence of language on thought reminds me of the **Sapir-Whorf hypothesis** from linguistics. Developed in the early 20th century by Edward Sapir and later expanded by his student Benjamin Lee Whorf, this hypothesis explores the relationship between language and cognition.

Whorf developed the idea while working as a chemical engineer and fire insurance inspector[17], where he noticed how language affected workers' perception of hazards. For instance, empty gasoline drums were treated carelessly because the word "empty" implied absence of danger, despite the explosive vapor they contained.

The hypothesis has two main variants. The strong version, **linguistic determinism**, claims that language completely determines thought, suggesting people cannot conceptualize ideas for which their language lacks words. Under this view, speakers of languages without future tense would struggle with long-term planning, or those without certain color terms couldn't perceive those distinctions. This strong version has been largely rejected by modern linguistics through empirical evidence showing people can think beyond the confines of their language.

The weak version, **linguistic relativity**, suggests that language influences (but doesn't determine) thought and perception. It proposes that language makes certain distinctions easier to notice or express. This version has empirical support - for example, languages with different color term boundaries show slight differences in color recognition tasks, and languages that use absolute directions (north/south) rather than relative ones (left/right) affect how their speakers navigate space.

I believe something similar to the weak form applies to programming languages. The language you use influences which solutions you see first, which abstractions feel natural, which patterns you reach for instinctively, and how you decompose complex problems.

A programmer who only knows imperative languages will struggle to see elegant functional solutions. Someone trained only in class-based object-oriented programming might overuse inheritance where composition would be clearer. Unfortunately, many programmers tend to be narrow-minded hubristic creatures, who need to justify their investment into a particular technology. This has

[17] Some of the greatest ideas come from unexpected places, huh? :D

led to many snarking at and discounting programming languages that are too different to what they are already used to.

This is why I strongly recommend experiencing and immersing yourself in multiple **very different** programming languages. Each language teaches you new mental models that remain useful even when programming in other languages.

Learning Lisp makes you better at **symbolic programming** - treating code and data as the same underlying structure and manipulating programs themselves as data, and it lets you uncover something about the nature and implementation of programming languages[18]. Learning Rust makes you think more carefully about **ownership** and **lifetimes**, and everything that can possibly go wrong when it comes to memory management and concurrent code. Learning Prolog teaches you to think **declaratively** rather than procedurally.

The more diverse your language experience, the richer your conceptual toolkit becomes for solving problems elegantly in any language. Each paradigm teaches you to see computation from a different angle, and combining these perspectives leads to more creative and effective solutions. In the coming chapters, we will examine a number of these paradigms and observe even the most basic good practices of writing elegant code.

Going back to **quicksort**, this algorithm can be implemented (and most often is) in a recursive way also. In mathematics, recursion is very common, because a lot of numerical sequences are defined in terms of previous elements. Before Lisp popularized it, many programming languages did not support recursion at all.

This was the case for Fortran, which had no notion of recursion at all in its first version. The recursive version of quicksort is more elegant:

```
def quicksort(arr):
    # base case: arrays with 0 or 1 element are already sorted
    if len(arr) <= 1:
        return arr

    # choose pivot and partition around it (middle element)
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
```

[18] Lisp being the programmable programming language from outer space, of course

```
    # recursively sort subarrays and combine
    return quicksort(left) + middle + quicksort(right)
```

In comparison to the previous examples, this one is written in Python. Python is about as readable, as programming languages of the C (or broadly speaking, imperative) pedigree can get. Recursion is often discouraged, because most languages don't have tail-call optimizations[19], and even if they do, the most elegant representation of a particular problem recursively is not a tail call.

Quicksort is fine if we choose the appropriate pivot point. Usually, we go about `log2(N)` calls deep, and to reach the 1000 calls recursion limit Python imposes by default, we would need an array in the ballpark of 10^307 elements. We probably can't fit such an array into memory (or anywhere else) anyway, so this algorithm is fine to be represented recursively without paying much attention to the size of the input.

We can achieve even more readability by trying a functional programming-oriented language, where recursion is a prefered mechanism to solve problems requiring iteration:

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (first:rest) =
    let smaller = quicksort [a | a <- rest, a <= first]
        bigger  = quicksort [a | a <- rest, a > first]
    in smaller ++ [first] ++ bigger
```

This syntax may be a little unfamiliar to you, so let's go through it:

```
quicksort :: [Int] -> [Int]
```

First, we declare a function named `quicksort` that takes a list of integers and returns a list of integers.

```
quicksort [] = []
```

We define the base case: when given an empty list, return an empty list (an empty list is already sorted).

[19] TCO is when the compiler/interpreter rewrites your recursion into machine code that essentially corresponds to a loop. If you can resolve all expressions from the previous level of recursion, you don't pay for the nested recursive call. For example `return this_function(n - 1)` is a proper tail call. But `return 1 + this_function(x)` isn't, since the addition `(+)` operation remains unresolved until we finish recursing to the bottom-most call.

```
quicksort (first:rest) =
```

This pattern matches a non-empty list, splitting it into the first element `first` (our pivot) and the rest of the list `rest`. In languages related to Haskell, it is very common to name these bindings `(x:xs)`. However, if you aren't a Haskell programmer, I think `(first:rest)` tells you a little bit more about what's going on.

```
    let smaller = quicksort [a | a <- rest, a <= first]
```

We create and recursively sort a list containing only elements from `rest` that are less than or equal to the pivot.

```
        bigger  = quicksort [a | a <- rest, a > first]
```

Similarly, this creates and sorts a list of all elements greater than the pivot.

```
    in smaller ++ [first] ++ bigger
```

Finally, it concatenates the three parts: smaller elements, the pivot, and bigger elements. This solution is far more elegant, but it is vulnerable to potentially requiring a lot of nested calls, since we do not pick the middle element, but the first element as our starting pivot.

It is perhaps slightly less readable than the previous solution, but we can change to use a middle pivot:

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort [x] = [x]
quicksort elements =
    let pivot = elements !! (length elements `div` 2) -- Middle as pivot
        smaller = quicksort [a | a <- elements, a < pivot]
        equal = [a | a <- elements, a == pivot]  -- Handle duplicates properly
        bigger = quicksort [a | a <- elements, a > pivot]
    in smaller ++ equal ++ bigger
```

The somewhat weird `!!` operator just does list indexing, `elements !! 2` would retrieve the third element of the list `elements`.

Haskell is a very, very powerful language. It is perhaps the one pure functional programming lanuage that can be widely applied in practice. This means that we can express ideas fairly elegantly in it, because it gives us a lot of tools to our disposal.

On the other hand, learning Haskell takes a bit longer, and requires a bit of a paradigm shift if you are coming from languages where the imperative approach reigns supreme. The idea of functional programming is powerful enough that mainstream languages are now adopting its wisdom. However, they are largely impure, usually because they allow mutability[20] or make no effort to limit side-effects[21]. This is something the languages question do because functional programming is not the primary priority.

Here is a similar quicksort written in Rust:

```rust
fn partition<F>(arr: &[i32], pivot_idx: usize, pred: F) -> Vec<i32>
where
    F: Fn(i32) -> bool
{
    arr.iter()
       .enumerate()
       .filter(|&(i, &x)| i != pivot_idx && pred(x))
       .map(|(_, &x)| x)
       .collect()
}

fn quicksort(arr: &[i32]) -> Vec<i32> {
    // base case: empty or single-element slices are already sorted
    if arr.len() <= 1 {
        return arr.to_vec();
    }

    // choose middle element as pivot
    let pivot_idx = arr.len() / 2;
    let pivot = arr[pivot_idx];

    // Partition array using the helper function
    let smaller = partition(arr, pivot_idx, |x| x <= pivot);
    let greater = partition(arr, pivot_idx, |x| x > pivot);

    // recursively sort partitions and combine results
    let mut result = quicksort(&smaller);
    result.push(pivot);
    result.extend(quicksort(&greater));
```

[20] For functional programming, the biggest issue mutating values from the outside, that is, whatever what violates referential transparency – a situation where we can replace a function call with the result of said function call and the behavior of the program will not chagne

[21] Side-effects are once again a problem for referential transparency, and also the predictability of a program's execution. Haskell has solved the issue of side-effects with Monadic IO, where the

```
    result
}
```

Rust is a programming language that is fundamentally imperative, but has functional leanings. These show in two main characteristics. First, we have iterators and iterator operations as opposed to using loops[22]:

```
arr.iter()
   .enumerate()
   .filter(|&(i, &x)| i != pivot_idx && pred(x))
   .map(|(_, &x)| x)
   .collect()
```

And immutablity by default. We have to use the `mut` keyword for the only variable we modify in this example:

```
let mut result = quicksort(&smaller);
result.push(pivot);
result.extend(quicksort(&greater));
```

Functional programming is not the primary goal of Rust, but its features help towards its major goals: Control, explicitness and safety. Since different programming have different goals, we cannot say that a language is bad because it does not have full features of paradigm A, if it never intended to do so in the first place.

A programming language is good if it fulfills its goals effectively (or at all), if it is well implemented,[23] and if it is internally consistent.[24]. These are fairly difficult requirements, and generally, one can point out flaws in the design of any programming language. A lot of time, a programmer may reject a programming language as "bad" because its goals either aren't noble enough, or mean nothing to him. One may reject Python, because it is not functional enough, somebody else may reject Rust, because they never had a memory safety incident, or

[22]Which are still available, Rust has `loop`, `for`, `while`, and `while-let`. The `while-let` structures does not verify a boolean condition, but a pattern match.

[23]You would be surprised, but there have been times in history where we had struggled implementing grand ideas. PL/I was a fairly influential programming language created in 1966 by IBM, and it was about as massive as you would expect anything made by IBM to be. Many competing implementations were created, almost none of which implemented the language fully. Very quickly, we had several incompatible dialects out in the wild.

[24]Some languages are internally inconsistent intentionally, the chief among them being Perl. This is fine, since it has justification, although it may not be your (or my) cup of tea. On the other hand PHP is internally inconsistent because it is a patchwork language of dubious heritage.

programmed in a language where such issues show up. A Lisp enjoyer may reject everything that does not have metaprogramming and S-expressions.[25]

Sometimes, programming language make intentional sacrifices in their design that prove to be far too expensive for the general programmer population, which hampers the adoption of a programming language. Let's take a look at one last `quicksort` implementation, this time in Common Lisp, solved in the style of symbolic programming:

```
(define-sort-algorithm quicksort
  (sort (sequence)
        (if (null sequence)
            nil
            (let ((pivot (car sequence))
                  (rest  (cdr sequence)))
              (apply-rule 'combine
                          (apply-rule 'sort (apply-rule 'smaller pivot rest))
                          pivot
                          (apply-rule 'sort (apply-rule 'bigger pivot rest))))))

  (smaller (pivot rest)
           (remove-if-not (lambda (x) (<= x pivot)) rest))

  (bigger (pivot rest)
          (remove-if-not (lambda (x) (> x pivot)) rest))

  (combine (smaller pivot bigger)
           (append smaller (list pivot) bigger)))
```

If you haven't done any Lisp, you probably can't read what's going on. Lisp's syntax is incredibly simple, it only has two[26] syntactic elements:

- The **atom**, which is anything that is not a list, for example:

```
1234     ;; number
"hello"  ;; string
t        ;; true
nil      ;; false or empty or missing value
:green   ;; keyword
jeremy   ;; symbol
#\a      ;; char
```

- The **list**, which is a sequence in parentheses containing atoms or other lists:[27]

[25] Although in my experience, contemporary Lisp programmers have been exceptionally nice people.

[26] If you are a fellow experienced Lisper, shut the fuck up for now :)

```
((Heart and soul I fell in love with you)
 (Heart and soul the way a fool would do madly)
 (Because you held me tight)
 (And stole a kiss in the night))
```

This is the first verse of the song Heart and Soul, written as a list of bars. Each bar is a list of symbols representing the words.

The humble combination of atoms and lists is enough to represent the syntax of all of the concepts of a full-fledged programming language.[28], to call a function, use a macro or define something, you just write a list. Here is how to make a function:

```
(defun hello (name)
  ;; t means print to standard output,
  ;; ~A is printing a positional argumentfor display
  ;; ~% means newline... Lisp is quite old
  (format t "Hello, ~A!~%" name))
```

The form `(defun)`[29] has the folloding arguments:

- the name of the function -> `hello`
- a list of arguments -> `(name)`
- the body of the function, which can be N elements, in this case just a single call of the `(format)` function

And you call this function like this:

```
(hello "John") ;; prints out "Hello, John!"
```

Therefore, the form `(define-sort-algorithm)` takes five arguments:

- The name of the algorithm -> `quicksort`
- Four transformation rules that describe the algorithm - `sort`, `smaller`, `bigger`, and `combine`:

```
(sort (sequence)
    (if (null sequence)
        nil
```

[27] In Lisp (which originally stood for **LIS**t **P**rocessor), lists are heterogenous, each element can be a different type. Because lists can also contain lists, we can easily represent values of all sorts of nested data structures. In fact, the notion of user-defined types came quite late – we could just shove everything into lists.

[28] And largely also the state of the running programs written in it, more on that later. Homoiconicity is a scary word.

[29] We will clarify what that is in a moment!

```
        (let ((pivot (car sequence))
              (rest  (cdr sequence)))
          (apply-rule 'combine
                      (apply-rule 'sort (apply-rule 'smaller pivot rest))
                      pivot
                      (apply-rule 'sort (apply-rule 'bigger pivot rest))))))
(smaller (pivot rest)
         (remove-if-not (lambda (x) (<= x pivot)) rest))
(bigger (pivot rest)
        (remove-if-not (lambda (x) (> x pivot)) rest))
(combine (smaller pivot bigger)
         (append smaller (list pivot) bigger))
```

The rules can apply each other using the `(apply-rule ...)` form. So you ask me: "Common Lisp has a built-in syntax for describing sorting algorithms? That's awesome, can you give me a source so I can look into it?"

My source is that I made it the fuck up. I actually defined a macro, for this tiny Domain-Specific Language. That is something that we do very often in Common Lisp, in order to introduce new structures. Symbolic programming is about treating code and data as interchangeable. We can make "functions" (in common parlance macros), that take code and output other code. Or take data and output code. Or take code and output data.

This let's us think in terms of the relationships between data, and between code, and create the optimal tools to describe the problems we are solving. The curse of Lisp is that it uses parentheses for a syntax that's just lists, but it needs syntax to be lists, because Lisp is a language exceptionally suited for manipulating lists! And we want to be able to manipulate syntax as lists, so that we can create new syntax with meaning! So Lisp cannot make any other choice, or it would not be so good for syntax manipulation![30]

And for this reason, Lisp is not a mainstream programming language. The most mainstream Lisp-y language is Clojure, and Clojure made some sacrifices of "Lispness" by moving less towards symbolic programming and more towards functional programming. Oh well.

For the non-Lisper, macro definitions often look like nasal demons. Here is the definitionn of my `(define-sort-algorithm)`. It is perfectly fine and expected if

[30] As a matter of fact, there have been so many attempts to revolutionize the syntax of Lisp that I have lost track. People always end up gravitating back to the parenthetical S-expressions — `(something ...)` — in this case, there is immense power in simplicity.

you don't understand it, there is a lot of context and knowledge you are unlikely to have at this point, unless you have done Lisp before:

```
(defmacro define-sort-algorithm (name &body rules)
  `(defun ,name (sequence)
     ;; create a function to execute a rule by name
     (let ((rule-table (make-hash-table)))

       ;; function to apply a rule by name
       (flet ((apply-rule (rule-name &rest args)
                (let ((rule-fn (gethash rule-name rule-table)))
                  (unless rule-fn
                    (error "No rule named ~S found" rule-name))
                  (apply rule-fn args))))

         ;; define each rule with access to apply-rule
         ,@(mapcar (lambda (rule)
                     `(setf (gethash ',(car rule) rule-table)
                            (lambda ,(cadr rule)
                              ,@(cddr rule))))
                   rules)

         ;; start the algorithm
         (apply-rule 'sort sequence)))))
```

Macros are what make Lisp a language that can grow to meet your needs. Common Lisp was designed for the "programming in the large" era of the 1980s and 1990s, anticipating that programmers would build large systems over time. The ability to extend the language itself with new syntax constructs allows teams to build domain-specific languages tailored to their problem domains.[31]

This is symbolic programming at its finest - treating code as data that can be manipulated, transformed, and reasoned about. While many modern languages have adopted functional programming features, few have embraced this level of syntactic flexibility. Typically, mainstream languages only see the inclusion

[31]There is a tale of the two main Lisps - Common Lisp and Scheme - which has quite an interesting history. Common Lisp was designed to unite the many competing implementations of Lisp that popped up in the previous decades, whereas Scheme was designed as a small and tight language useful for illustrating concepts related to lambda calculus in a practical manner. Scheme is unfortunately still too small to have a widespread adoption, whereas Common Lisp is a large standardized language. The only language that's larger that I can think of is C++. However, Common Lisp has an incredibly stable standard, there hasn't been a new version since the final one published in 1994. This means that very old code works without issue and that sometimes, you will find libraries that are just "done", having no major development in 10+ years, but still being depended on regularly by new projects.

of basic macros at most. However, time is a flat circle and we see inclusion of stronger metaprogramming facilities in modern up-and-coming programming languages such as Rust or Nim.

The `define-sort-algorithm` macro allows us to describe sorting algorithms at a higher level of abstraction. Rather than focusing on implementation details, we express the essence of the algorithm as transformation rules. This approach makes the core logic more apparent:

1. If the sequence is empty, return empty
2. Otherwise, take the first element as pivot
3. Find elements smaller than the pivot
4. Find elements bigger than the pivot
5. Sort both partitions recursively
6. Combine the results

Different languages offer different tools for expressing these ideas. C lets us manipulate memory directly but requires explicit control flow. Python makes the algorithm more readable with list comprehensions. Haskell's pattern matching and type system enforce correctness. Rust combines safety with control. Lisp elevates the abstraction to manipulate the language itself.

Each approach represents a different balance in the eternal tension between what the computer understands and what humans understand. This tension is at the heart of programming as communication.

2.1.1 On Lambdas and Logos, refined

In the beginning, there was the λ-calculus.

Well, not quite the beginning. But when Alonzo Church formalized the λ-calculus in the 1930s, he created what would become the theoretical foundation for functional programming languages. This mathematical system for expressing computation using function abstraction and application showed that all computable functions could be expressed through these simple mechanisms.

The λ (lambda) symbol has since become emblematic of functional programming, representing the idea of anonymous functions that can be passed around, composed, and applied. When John McCarthy created Lisp in 1958, he directly implemented lambda expressions, bringing Church's mathematical abstraction into the realm of practical programming.[32]

Meanwhile, "logos"[33] comes to us from ancient Greek philosophy, where it represented discourse, reason, and the underlying principles that govern reality.[34] Heraclitus spoke of the logos as the universal principle according to which all things happen. For the Stoics, it was the divine reason that pervades everything. In the Gospel of John, "In the beginning was the Logos" - the Word, the fundamental ordering principle.

In our context, logos represents the communicative aspect of programming - how we express our ideas through code, how we reason about problems, and how we share that reasoning with others (including our future selves).

Programming languages sit at the intersection of these two concepts. They are formal systems with precise rules (lambda), yet they are also media for human expression and communication (logos). The elegance of a programming language comes from how well it balances these two aspects - how effectively it allows us to express human ideas in a form that computers can execute.

---

[32] Lisp was first invented as a "useful mathematical notation" for computer programs, McCarthy did not expect that someone would go and implement it: *"Steve Russell said, look, why don't I program this eval ... and I said to him, ho, ho, you're confusing theory with practice, this eval is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the eval in my paper into IBM 704 machine code, fixing bugs, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today ..."*

[33] Conveniently written as λόγος, which is where I got the λς on the title page

[34] And about a fifty other different things.

This brings us back to the Sapir-Whorf hypothesis. Just as human languages might influence how we perceive and categorize the world, programming languages influence how we decompose problems and construct solutions. A programmer fluent only in C sees the world in terms of procedures and memory management. A dedicated Haskell programmer sees it as type transformations and pure functions. A Lisp hacker sees code itself as just another data structure to manipulate, and the language as a malleable medium of communication. Conlanger's paradise

The true art of programming lies not in mastering any single language or paradigm[35], but in understanding the fundamental principles that underlie them. Each paradigm illuminates different aspects of computation:

- Imperative programming gives us direct control over the machine's state
- Functional programming gives us mathematical reasoning and composition
- Object-oriented programming gives us modeling through encapsulation and behavior
- Symbolic programming gives us code that can reason about and transform itself

By learning multiple paradigms, we expand our conceptual vocabulary. We become multilingual programmers, or programming linguists, able to choose the right language (or combination of languages) for the problem at hand.[36] We can communicate more clearly, not just with the computer, but with other programmers who will read and maintain our code.

The lambda gives us the formal tools to express computation. The logos gives us the purpose: to communicate ideas clearly and elegantly. Together, they represent the dual nature of programming as both science and art - a rigorous formal system that is also a medium of human expression.

In the chapters that follow, we'll explore how to put these principles into practice. We'll examine patterns of elegant code across paradigms, and we'll learn how to structure our programs to communicate their intent clearly. Whether you're writing a quicksort algorithm or a complex enterprise system, the funda-

[35] Although mastering any of them certainly helps, the big idea is to never become narrow-minded in your approach

[36] Unfortunately in the real world, the choice of language is often made for you. In that case, your multilingual skills help you recognize how to write better code, and how to apply wisdom of different worlds in this one.

mental challenge remains the same: to express your ideas in a way that both computers and humans can understand.

That is, submit to no one, and bend the world to your will.

2.2 Coding != Programming

In our modern technological landscape, the terms “coding” and “programming” are often used interchangeably, as if they were perfect synonyms. Maybe to some they are, but not to me. I view it as linguistic laziness of the highest degree.

This linguistic laziness obscures an important distinction that lies at the heart of our discipline. While related, these terms represent fundamentally different activities and mindsets, a distinction worth exploring if we wish to elevate our craft.[37]

Let me present to you my conception of these terms.

Coding refers to the mechanical process of writing instructions in a programming language. It’s about syntax, about translating already-formed ideas into code that a machine can execute. At its most basic level, coding is a transcription task – taking a solution that exists in some form and rendering it in a formal language. This is not to diminish its difficulty; good coding requires attention to detail, knowledge of language features, and technical skill. But coding, in isolation, is merely implementation.

Maybe, as a junior developer employed in a company, you will be doing a great deal of coding, because it takes a while to gain experience and penetrate both the domain the product you are working is situated in, and its implementation. This is fine, but you shouldn’t have the false impression that this is all there is to it, and that you shouldn’t be thinking when writing code, even if someone already did all the planning for you, and all you are presented with is a task in the form of “In class X, add method Y, taking parameters Z, which you will call in class A, method B”.[38]

Programming, on the other hand, encompasses a far broader intellectual territory. Programming is the art of computational thinking, of dissecting problems into their essential components, of discovering or inventing abstractions that make complexity manageable. It involves architecture and design, algorithm selection, data structure consideration, and deep engagement with the problem domain. Programming happens away from the keyboard as often as at it – in

[37] In the past, I have been more cynical and accused the mainstream media and business people of using the word coding to devalue the prestige of our discipline.

[38] Feel free to reimagine this sentence in your favorite paradigm

conversations, on whiteboards, during walks, in the shower, or while falling asleep.[39]

When I tell people I'm a programmer, they often imagine me sitting at a computer typing frantically for hours, producing line after line of obscure symbols. This Hollywood-perpetuated image misses the essence of what I actually do. Most of my time is spent thinking, reading, discussing, arguing with idiots on the internet, sketching, and understanding. The actual typing of code might represent only a fraction of my working day, especially now that I am no longer working as a software engineer, but take a more educational role. As the legendary computer scientist Donald Knuth once observed, "Programming is the art of telling another human what one wants the computer to do."

Consider the evolution of our tools. Early programmers used punch cards, where each card represented a single line of code.[40] This physical constraint forced programmers to think carefully before committing an instruction, as mistakes were costly to correct. Today, we can type code rapidly and undo mistakes with a keystroke, but this ease has sometimes disconnected us from the deliberation that preceded implementation. The best programmers maintain that deliberative mindset even with modern tools – they think deeply before they code.

A programmer places understanding at the apex of priorities. Without a thorough grasp of the problem, even the most elegant code is merely an attractive wrong answer. I have made a lot of attractive wrong answers in my life. This understanding is multi-layered: understanding the stated requirements, the unstated expectations, the users' actual needs (which may differ from what they say they want)[41], the constraints of the system, and the implications of different approaches. A programmer recognizes that the hardest part of building software isn't the "coding" – it's figuring out what to build, and how to build it, and especially how to build it in a way that is robust enough for a given usecase.

[39] My best programming is done on long walks through nature or old Prague. I find that the repetitive motion of walking, and the sounds of outside help me eliminate distractions, and naturally lead me into a deep thinking state. On the comparatively rarer occassions that I wear earphones, walks a

[40] If you have ever been wondering where the practice of "80 characters per line of code max" comes from, guess how many characters you could fit on a punchcard, and how many characters could horizontally fit on early terminals.

[41] Often, the user is completely wrong about what they want, and their needs have to be taken with a grain of salt and ideally, signed in blood.

As a result, a programmer's code should be refined, clear, and purposeful – a crystallization of their thinking process. After all, the code you write is the reflection of your thought process. If your thinking about a given problem is disorganized, so will be the code you write. Just as good writing isn't merely grammatically correct but also clear, and persuasive, and properly utilizes the language you are writing your text in, good programming isn't merely syntactically valid but also elegant and comprehensible. The code we write is a communication medium, not just to the computer but to other programmers (including our future selves)[42]. As Robert C. Martin puts it, "Clean code always looks like it was written by someone who cares."[43]

The distinction extends further when we consider professional roles. A **software engineer** applies programming principles to solve real-world problems within practical constraints. Engineers must bridge multiple domains – they need to understand not just computation but also the specific field where they're applying it. A financial software engineer needs to grasp accounting principles. A medical software engineer needs to understand healthcare workflows. This cross-domain expertise is what enables them to translate messy human systems into computational models that actually serve their intended purpose.

As a software engineer, you are the lord of compromises. You need to design and implement a system that fullfills a task as well as possible, you have to do it in reasonable time, and you generally have to make some sacrifices in the name of integrating the project with the rest of the company ecosystem[44]

Meanwhile, a **researcher**[45] in programming explores the theoretical foundations, develops new paradigms, creates programming languages, or investigates computational limits. They may work on problems that won't have practical applications for decades, if ever, but their work expands our understanding of what's possible and pushes the boundaries of our field.

[42] If you take one thing from this book, let it be this.

[43] Credit where credit is due, but I am not a huge fan of the Clean Code book, but that is for another day. It is mostly just that it is very old-Java-centric.

[44] You can't just say "Oh, we have Python everywhere, and our company is mostly Python develepers, so I will write this in a purely functional Haskell, which I happen to know, and it will have monads, and blackjack and hookers!". What you can do, however, is integrate elements of good functional style into the architecture and implementation of the project in Python, granted that these elements create a cohesive structure.

[45] *Computer scientist* also feels appropriate

In my free time, I like to guide my programming activities according to the following mantra:

> *Program in such a way that any practical appliaction of your code is purely coincidental*

This is great for having fun, and for learning a lot. It is important to make a distinction, which a lot of programmers of all skill levels sometimes fail to make, and that is that free-time, open-source, and commercial programming are all different disciplines[46]

It is also worth noting that many people who aren't programmers write code. Scientists use scripting languages to analyze data.[47] Accountants create Excel formulas. System administrators write automation scripts. With the advent of large language models and AI assistants, the number of people who can produce functional code without deep programming knowledge will increase dramatically.

These tools democratize access to coding, which is can be positive,[48] but they cannot substitute for the thinking process at the heart of programming. An LLM can help you express an idea in code, but it cannot (yet) tell you which idea is worth expressing, and for each idea, how it should be expressed such that it fits into a greater context. At the time of this writing, LLMs are really bad at higher-level architecture. An AI can implement a solution, but it cannot tell you if you're solving the right problem. It can optimize code, but it cannot tell you if your entire approach should be reconsidered. The language model might write syntactically perfect code that's conceptually misguided because it mirrors the user's incomplete understanding.

This is why the role of the programmer remains critical: we are not merely code producers but computational thinkers who understand problems deeply enough to model them effectively. While an LLM might help a doctor write a Python

[46] And don't get me started on the needs and conventions of different fields. Commercial web-development is a completely different world from programming in the automotive industry, and not just because of the technologies used, but how they are used.

[47] Or go the exact opposite directions and raw-dog Fortran, or alternatively use Julia. Jupyter Notebooks are also very popular among scientists.

[48] What I mean is that LLMs are timesavers - you can ask them for small changes, minor refactors, and looking up information. I have benefitted from this, although probably arguably less than someone who uses more conventional programming languages and technologies.

script to analyze patient data, it cannot replace the programmer who designs the hospital's entire electronic health record system with an understanding of security, data integrity, workflow, scalability, and regulatory compliance, and the perhaps pessimistic understanding of the possibility of human error at every step of the way.[49]

If you're reading this book, you should think of yourself as a programmer (or a programmer-in-training, if you want to), not just a coder. Abandon any imposter syndrome that might make you think otherwise. You are engaging with a discipline that requires creative thinking, problem-solving, and deep understanding – you're not just learning syntax.

However, in claiming the title of programmer, hold yourself to the standards it implies. Make understanding your priority. Refine your thinking before you refine your code. Think, Mark, think! Recognize that clear code comes from clear thought, and confused code usually reflects confused thinking. Be willing to restart when you realize your approach is fundamentally flawed – as painful as that can be. In your hobby programming, you have a luxury of throwing things away, and trying different approaches, not being bound by severe time constraints which some fields of commercial programming otherwise have.[50]

Programming is inherently creative. We, as programmers, build digital worlds from nothing but thought, giving form to ideas and solving problems that often have no precedent. In what other field can you create something so complex, yet very mutable and alive, with nothing more than a computer and your mind? That's crazy, dude. The barrier of entry into the world of programming is very low, and the sky is your limit. There's a particular joy in seeing your thoughts externalized and animated, in watching a computer dance to the tune you've composed. When we refer to "elegant" code, we're making an aesthetic judgment not unlike how we might evaluate a poem or a painting. I sometimes say that elegant code "tastes good", it seems to be my particular form of synesthesia.

[49] At least at the time of this writing, the so-called vibe coding has been a path to hell, and understanding code remains a neccessary skill. This also applies reflexively - the better programmer you are, the better you can describe problems, and the better you can utilize AI in a productive way.

[50] Of course, in commercial products, you just can't keep restarting, or even restart in the first place. However, you should not feel guilty about trying to justify refactors, so long as you still keep advancing the product

Programming should be fun – not always in the moment (debugging can be frustrating, but when you figure it out feels great - Have you ever killed a difficult Dark Souls boss?), but in the larger sense of providing intellectual satisfaction and creative fulfillment. It should engage your curiosity, challenge your mind, and reward your efforts with the distinct pleasure of seeing abstract ideas become concrete reality.

The distinction between coding and programming isn't about establishing a hierarchy where programmers look down on "mere coders." After all, I call myself a teacher, it would be foolish to look down on people who know less than I do, or be hubristic enough to think poorly of people who know more than I do. Rather, it's about recognizing the full scope of what programming entails and aspiring to practice it in its complete form.[51] Coding is an essential component of programming, but programming is more than coding – it's a mode of thinking, a way of approaching problems, and a creative discipline that happens to produce code as its artifact.

As you progress through this book and your career, strive to be more than someone who writes code.[52] Think clearly about problems, design elegant solutions, who communicate effectively through code, and find joy in the creative process of programming. The code you produce will be better for it, and so will your experience of creating it. And I suppose that's the gist of that

[51] And ideally, share it!

[52] And never thing there isn't any further milestone you can aim for!

2.3 Programming should be fun

I would like to now expand on one of the last thoughts from the previous section - that programming should be fun. I would like to paraphrase Gerald Jay Sussman, one of the creators of the Scheme programming language. A couple years ago, he had a talk called "Programming (is) should be fun" for the ACM SIGPLAN Scheme conference, which resonated with me deeply.[53] I will therefore try to relay Gerald's ideas here and provide commentary on them.

Sussman and his colleague Harold Abelson began their seminal book "Structure and Interpretation of Computer Programs" (commonly abbreviated as SICP)[54] with a quote from Alan Perlis that sets the tone for their approach to computing:

> I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun.

In his talk, Sussman argues that programming has lost much of this original joy. It has become industrialized, over-complicated, and burdened with processes that strip away the creative aspects that make it intellectually stimulating. He

[53] Scheme is a very elegant language, in that for how minimalistic it is, it is quite powerful, and a lot of programming ideas can be expressed quite clearly. The vast majority of its syntactic forms can be expressed in terms of only a handful special forms. You can built up many control structures with macros and those forms. Particularly the idea of a closure - a lambda/anonymous function that captures things from its environment is quite powerful - powerful enough that it is present in Common Lisp too, which often practices dynamic scope unlike Scheme and most other languages, as the *let over lambda* pattern. An excellent, although a bit too enthusiastic, and very hardcore (in the author's own words) book with this title has been written by Doug Hoyte.

[54] SICP is probably the seminal text for showcasing programming concepts via Scheme. It is a fairly old book, but a timeless classic. There is a newer version created by perverse minds that replaces Scheme with JavaScript. It does make kinda sense that JS would be the one language flexible enough to replace Scheme, seeing as JavaScript originally **was** essentially Scheme (business people, who famously hated all fun, told Brendan Eich to replace his Scheme in browser with something that looks more like Java, which was a very marketable buzzword, given Java's novelty and popularity at the time).

observes that modern software development has morphed from an exploratory, creative endeavor into something resembling factory work - where programmers are expected to plug components together without necessarily understanding how they function.

To me, seeing how things function is one of my favorite activities within the whole of IT. Computer Science, or IT at large, is a field that can be described as an infinite series of Plato's cave allegories. It is very foolish to stop at one point and think "I know enough about the nature and utilization of computers". In the parlance of my generation, we refer to this as "L take, bozo". Alternatively, IT could also be described as a rabbit hole that never ends, but I think the level design of Plato's caves is more telling.

Sussman points to several developments that have contributed to this shift:

First, the proliferation of massive, complex frameworks that nobody fully understands. Modern software is built on towering stacks of abstractions - operating systems, libraries, frameworks, middleware, virtual machines, and more.[55] Each layer adds complexity that obscures the underlying principles. When something goes wrong, most programmers lack the deeper understanding required to diagnose and fix the issue meaningfully. Instead, they resort to workarounds and band-aid solutions.[56]

Second, the changing nature of programming education. What was once a discipline focused on understanding computation from first principles has increasingly become vocational training. Students learn specific technologies and tools rather than fundamental concepts. They're taught to use frameworks and libraries without understanding how they work internally. This approach might produce programmers who can quickly build applications using current tools, but it fails to develop the deep thinking necessary for innovation.

Today, this is not true in all universities. Some universities offer courses such as theoretical computer science, which are far less about vocational training. The shift toward more vocational training is understandable, seeing as in recent decades, companies have wanted to hire more and more programmers, as soft-

[55] The idea here is that you should know these things exist and how they work in principle. It is infeasible that you would know how all of these things work in-depth in concrete terms.

[56] For an example of this, see any Microsoft source code leak ever.

ware takes an increasingly important role in running our society.[57]

Third, the growing complexity of software ecosystems has made it nearly impossible for any single person to truly understand the entirety of a system. This compartmentalization leads to a sense of alienation - programmers become cogs in a machine rather than craftspeople who take pride in their work.[58]

As an antidote to these trends, Sussman advocates for a return to programming as intellectual exploration. He suggests we should build systems from first principles, understanding each component thoroughly. Rather than treating complex systems as black boxes, we should strive to understand them "all the way down" - from high-level abstractions to the hardware that executes our code.[59]

What is most interesting to me, Sussman takes a contrarian perspective on bugs and errors that many professional environments would find heretical.[60] Instead of viewing bugs as failures to be eliminated, he frames them as opportunities for learning. When something goes wrong, it often reveals gaps in our understanding. These moments, while frustrating, provide chances to deepen our knowledge of systems.

This view doesn't mean Sussman encourages sloppy programming. As a matter of fact, sloppy programming is to be avoided at all costs! Rather, he suggests that the process of finding and fixing bugs can be intellectually rewarding. It's through this exploration - building something, seeing it fail, understanding why, and improving it - that we develop genuine expertise. The joy comes not just from creating something that works, but from truly understanding how and why it works.

As a matter of fact, the presence of bugs can often reveal information about the problem that originally wasn't available to you. By discovering and fixing bugs,

[57] Can you imagine a govermnent where all the operations are up to people, and no software is involved? It doesn't even work **with** the software, let alone without.

[58] This also makes it increasingly more difficult to write good code, as you have to watch out for interactions with the rest of the ecosystem, which may be non-trivial. Part of the success of the Rust programming language is that it is low-level enough to support systems programming and strict and explicit enough to alert programmers to potentially problematic interactions, and encourage them to handle them by default. Part of the issues with programs written in C and C++ is that the safety features are opt-in, not opt-out, and programmers tend to have an inflated sense of their own skill and infallibility.

[59] This should be def

[60] Many programmers in general would, have you ever heard the term "skill issue"?

you learn more about the problem you are solving. You may discover edge cases or inputs that you previously didn't think could occur.

Sussman is particularly critical of the trend toward "programming by coincidence" - where developers copy-paste code from Stack Overflow or other sources without fully understanding it.[61] This approach might produce working software in the short term, but it creates brittle systems that resist modification and improvement. True mastery comes from building a deep mental model of how systems work, which allows for creative problem-solving rather than rote application of patterns.[62]

He also laments the loss of playful experimentation in programming. In the early days of computing, programmers had more freedom to explore and create for the sake of learning. Today's focus on productivity metrics, deadlines, and commercial concerns has diminished this aspect of the discipline. Sussman argues that time spent in seemingly unproductive exploration often leads to insights that prove valuable later - but this process can't be easily quantified or scheduled.

The diminishing role of elegance in programming particularly concerns Sussman. Elegant code - concise, clear, and powerful - emerges from deep understanding. Yet modern development processes often prioritize immediate functionality over thoughtful design. The result is bloated, complex systems that become increasingly difficult to maintain and extend.

This is typically done in the name of meeting deadlines and generating profits, and boy, I, as an evil capitalist, have no problem with the notion of generating profits. However, rushing too much leads to the creation of significant amount of technical debt. Technical debt is costly, and the longer it exists and the more is your product scaling the costliest it is. Eventually, it may happen that the dam breaks, and the only thing that can save a project is a complete rewrite – which is costly and takes time.

Sussman points to Lisp and its descendants (like his own Scheme) as languages that embody the principles he values. These languages are built on a small

[61] I suppose this could be considered, the earlier, more involved form of vibe coding. Nowadays, Stack Overflow traffic is decreasing, while LLMs have revolutionized programming. I know many people who now have models integrated into their editors.

[62]

set of powerful abstractions that can be combined in countless ways.[63] They encourage thinking about programming in terms of transformations and compositions rather than step-by-step procedures. This approach fosters the kind of deep understanding that makes programming both intellectually stimulating and personally rewarding.

He also emphasizes the importance of building mental models when programming. Rather than memorizing libraries and APIs, programmers should develop frameworks for understanding how systems work. These mental models allow us to reason about code, predict its behavior, and design solutions that address fundamental issues rather than symptoms. The joy in programming comes partly from refining these mental models through experience.

Another point Sussman makes is about the relationship between the programmer and the machine. He suggests we should view computers not as tools to be used, but as collaborators in the creative process. Programming isn't just about telling the computer what to do; it's about expressing ideas in a form that both humans and computers can understand. This dual nature of programming - as both a technical and communicative act - is what makes it uniquely challenging and rewarding.

In essence, Sussman argues for a return to seeing programming as an intellectual adventure. It should involve exploration, discovery, and the joy of understanding complex systems. While acknowledging the practical realities of commercial software development, he suggests that by reconnecting with the fun and creativity of programming, we not only make our work more personally fulfilling but also become better problem-solvers.

And I couldn't have said it better myself.

[63] Lisp was the first programming language that espoused functional programming ideals. However, later languages were more theoretically rigorous, whereas the main strength of Lisp, that makes it unique to other languages became symbolic computation and its metaprogramming features.

2.4 It's not just the code

Understanding of code comes primarily from how you write the actual lines of code, as in, the text in them. However, that is not all. There are other things that are important to help your understanding, and to proliferate the understanding of others.

For you, one thing that is important, and that may be often discounted is how you look at the code. Do you have an IDE, or a properly set up editor, that provides hints, Language Server Protocol actions, or REPL integration, editor that has nice colorful syntax highlighting that is appropriately smart[64], and other bells and whistles that help navigating and understanding codebases.

The Lisp language has a reputation for being unreadable. This is only partially true, but is very often an artefact from early days, where many unlucky students had to contend with Lisp that had no syntax highlighting, barely matching parentheses highlighting, and possibly improper formatting. Let's consider the macro I showed you for defining **quicksort** in terms of transformation rules.

This is how it looked:

```
(defmacro define-sort-algorithm (name &body rules)
  `(defun ,name (sequence)
     ;; create a function to execute a rule by name
     (let ((rule-table (make-hash-table)))

       ;; function to apply a rule by name
       (flet ((apply-rule (rule-name &rest args)
                (let ((rule-fn (gethash rule-name rule-table)))
                  (unless rule-fn
                    (error "No rule named ~S found" rule-name))
                  (apply rule-fn args))))

         ;; define each rule with access to apply-rule
         ,@(mapcar (lambda (rule)
                     `(setf (gethash ',(car rule) rule-table)
                            (lambda ,(cadr rule)
```

[64] One thing I do often in my Emacs setup is that I define additional highlighting for the programming languages I use extensively for things that are not distinguished by default. In Scheme, for instance I highlight different naming conventions with different colors - predicate functions typically end with a question mark (like `string?`), mutating functions end with a exclamation mark (such as `set!` - I highlight those in red, we hate mutation in our beautiful functional world! Grrr), type conversion functions often have an arrow in them `->`.

```
                                ,@(cddr rule))))
                    rules)

          ;; start the algorithm
          (apply-rule 'sort sequence)))))
```

This is very colorful, maybe too colorful depending on your tastes (and what color scheme I end up going with for the examples). Let's take a look at it completely deprived of colors:

```
(defmacro define-sort-algorithm (name &body rules)
  `(defun ,name (sequence)
     ;; create a function to execute a rule by name
     (let ((rule-table (make-hash-table)))

       ;; function to apply a rule by name
       (flet ((apply-rule (rule-name &rest args)
                (let ((rule-fn (gethash rule-name rule-table)))
                  (unless rule-fn
                    (error "No rule named ~S found" rule-name))
                  (apply rule-fn args))))

         ;; define each rule with access to apply-rule
         ,@(mapcar (lambda (rule)
                     `(setf (gethash ',(car rule) rule-table)
                            (lambda ,(cadr rule)
                              ,@(cddr rule))))
                   rules)

         ;; start the algorithm
         (apply-rule 'sort sequence)))))
```

To me, this is far less readable. The syntax highlighting helped my brain visually distinguish different categories of elements. It is important that you find a good color scheme and level of syntax highlighting.[65] This may mean you have to switch editor, if you are using something that is not very configurable. I use Emacs, which is the extreme when it comes to configurability, and it may be far too configurable for most people. However, even in VS Code, you should be able to affect how things are syntax highlighted by, for instance, installing better language extensions (like **Rust Analyzer** for Rust).

Let's see how the previous example looks in my editor in light mode. I use an amalgamation of themes related to the default colors of the Acme editor. The

[65] There is just so many of them that you can be sure that there is something that will speak to you.

Acme editor famously had no syntax highlighting, and had properties which make it even more different from the mainstream than Emacs and Vim. Here's the picture:

```
254
255
256 (defmacro define-sort-algorithm (name &body rules)
257   `(defun ,name (sequence)
258      ;; Create a function to execute a rule by name
259      (let ((rule-table (make-hash-table)))
260
261        ;; Function to apply a rule by name
262        (flet ((apply-rule (rule-name &rest args)
263                 (let ((rule-fn (gethash rule-name rule-table)))
264                   (unless rule-fn
265                     (error "No rule named ~S found" rule-name))
266                   (apply rule-fn args))))
267
268          ;; Define each rule with access to apply-rule
269          ,@(mapcar (λ (rule)
270                      `(setf (gethash ',(car rule) rule-table)
271                             (λ ,(cadr rule)
272                               ,@(cddr rule))))
273                    rules)
274
275          ;; Start the algorithm
276          (apply-rule 'sort sequence)))))
277
278
```

As you can see, I am highlighting a whole bunch of stuff, and most importantly the matching parentheses that my cursor is next to. This is very important in Lisp, but may be less important other languages. In general, syntax highlighting is a good thing, however, some languages are better at revealing their structure than others. Typically, these are languages that use special characters and clear formatting and naming conventions to distinguish different elements in the source code. This is similar to how in German, nouns are written with a capital letter.[66]

Many programming language use the convention of naming types using a capital letter. This is the case in Rust:

```
struct Point {
    x: f64,
    y: f64,
}
```

[66] For instance, many C-like languages use semicolons and different brace types, []{}()<>

```rust
fn main() {
    let origin = Point { x: 0.0, y: 0.0 };
    let point = Point { x: 3.5, y: -2.1 };
}
```

In dark theme, the highlighting I use for Lisp in my editor is quite similar:

```lisp
(defmacro define-sort-algorithm (name &body rules)
  `(defun ,name (sequence)
     ;; Create a function to execute a rule by name
     (let ((rule-table (make-hash-table)))

       ;; Function to apply a rule by name
       (flet ((apply-rule (rule-name &rest args)
                (let ((rule-fn (gethash rule-name rule-table))
                      (unless rule-fn
                        (error "No rule named ~S found" rule-name))
                      (apply rule-fn args)))))

         ;; Define each rule with access to apply-rule
         ,@(mapcar (λ (rule)
                     `(setf (gethash ',(car rule) rule-table)
                            (λ ,(cadr rule)
                              ,@(cddr rule))))
                   rules)

         ;; Start the algorithm
         (apply-rule 'sort sequence))))))
```

To further the configuration point: You'll often hear people joke that "Linux is free if you don't value your time" – implying that the time spent configuring your system isn't worth it. I strongly disagree with this sentiment. A well-configured development environment is an investment that pays continuous dividends. Every minute shaved off your daily workflow, every eye strain prevented, every wrist pain avoided, is a return on that investment.

Time spent understanding and optimizing your tools isn't wasted – it's multiplied across every hour you spend programming for the rest of your career. This is why professional mechanics own their own tools, why surgeons have preferences for specific instruments, and why programmers should care deeply about their environment.

For example, learning keyboard shortcuts in your editor or IDE might take days to internalize but will save you weeks of cumulative time over years. Setting

up snippets, templates, and custom macros might feel like procrastination, but these accelerators make the mechanics of coding disappear so you can focus on the problem-solving aspect.

Let's broaden the scope. Physical comfort significantly impacts your ability to maintain focus and write elegant code. A good monitor reduces eye strain and lets you see more context at once. A comfortable chair prevents back pain during long coding sessions. And perhaps most importantly, a good keyboard can prevent repetitive strain injuries that plague many programmers.

I personally use a Corne v3 split keyboard, which keeps my wrists at a comortable angle and reduces the distance my fingers need to travel. Split keyboards look strange to the uninitiated, but they're designed around human anatomy rather than manufacturing convenience. Your tools should adapt to you, not the other way around.

A decent computer is also an essential investment. Waiting for compilation, having your editor lag when opening large files, or experiencing freezes during debugging all break your concentration and cognitive flow. The mental context switch caused by these interruptions is far more costly than the seconds you actually wait.

Perhaps the most insidious enemy of elegant code is the constant barrage of distractions we subject ourselves to. Our generation faces unprecedented challenges to sustained focus, and the same device we program on is usually connected to a world of interruptions.

My productivity and concentration improved after I:

- Disabled push notifications for most applications
- Unsubscribed from dozens of email newsletters and promotional content
- Stopped using Instagram entirely
- Used browser extensions on desktop and alternative clients on Android to completely remove every mention of YouTube Shorts

These attention-harvesting mechanisms are specifically engineered to hijack your focus. Each notification triggers a dopamine response that makes deep work more difficult. When writing elegant code requires deep thought about structure, relationships, and abstractions, these interruptions are poison to quality.

Learning a more advanced editor like Emacs, Vim, Neovim, or Helix may seem difficult or pointless at first, but in my opinion, it pays incredible dividends. These tools were designed by programmers for programmers, with the specific goal of making text manipulation (which is what programming is, at a mechanical level) as efficient as possible.[67]

The modal editing of Vim means you spend less time reaching for arrow keys or the mouse. The extensibility of Emacs means you can create custom workflows that match exactly how you think.[68] The modern amenities of Neovim and Helix bring these powerful paradigms into the present with sensible defaults and better performance.

Even if you stick with an IDE, learning its keyboard shortcuts and advanced features is worth your time.[69] The goal is to reduce the friction between your thoughts and their expression in code.

Of course, not everyone can afford top-of-the-line equipment.[70] Economic realities differ, and it's important to acknowledge that. If you're programming on a budget laptop with a basic keyboard, that doesn't make you any less of a programmer. Many brilliant systems were written on modest hardware.[71]

So, focus on the things you can do for free: learning your tools deeply, eliminating digital distractions, and creating the best environment possible within

[67] There has been a very interesting editing model, apart from the aforementioned editors called **structural editing**. The idea of structural editing is that the editor is able to build a syntactic tree out of the source code (which is very difficult, since most of the time, when you are actively editing code, you have invalid syntax issues – "of course there's a missing semicolon and unclosed parenthesis! I didn't finish writing it yet!"), and let's you maniputa the syntactic tree directly. This can be extremely effective, since the editor truly understands the source code. However, it is hard to implement and get used to. Structural editing is fairly popular in the Lisp world because of how easy it is to parse. Check out the packages/functionality of **paredit** and **parinfer**.

[68] I have a modal editing setup in my Emacs that's fairly similar to Kakoune's/Helix's modal editing model. Emacs let's you override completely everything. You could make it behave like VS Code, if you wanted to. This book was written in Emacs, too.

[69] Just imagine how much time you waste navigating context menus when you could press a 3-key shortcut.

[70] As a matter of fact, I learned a lot when I was a child precisely because I didn't have proper equipment, and have to make do with what I had.

[71] There is an argument to be made. Some people advocate for development on poor hardware because it will encourage you to make faster and leaner programs (so that you can comfortably run them on what you have). This is a fair point, however, I think that thinking about making your programs effective is enough, and not wasting time is more important.

your constraints. A quiet room with decent lighting and a comfortable chair from a thrift store will serve you better than an expensive setup in a distracting environment.

Second-hand ThinkPads, for instance, offer excellent keyboards and reliability at reasonable prices.[72] Free software like Linux can breathe new life into aging hardware.[73] Community-built mechanical keyboards can be more affordable than you might think. Prioritize what matters most for your specific work.

Let's now go to the "understanding of others" side, which may, once again, include a future you, we have documentation, tests (which happen to be useful small examples for things that may be unclear!), bigger examples, comments in the source code (however, keep in mind that too many comments reduce visual clarity), and so on.

Good code is about communication, as I have said before and will say again. When others (or your future self) read your code, they should understand not just what it does, but why it does it that way. This requires thoughtful presentation:

- Consistent formatting makes patterns more apparent
- Meaningful variable and function names communicate purpose
- Judicious comments explain the "why" when code can only show the "how"
- Documentation provides context and user instructions
- Tests demonstrate expected behavior in concrete terms
- Version control messages explain the evolution of the code[74]

Remember that code is read far more often than it's written. The time you invest in making it presentable pays off every time someone needs to understand, modify, or build upon your work.

---

[72] They are also durable and spare parts are readily available. Look for the P or T series. Avoid the X1 Carbons, if you want user serviceability, anything with Yoga in the name, and especially avoid the E series (cheaply made).

[73] I have been using an ancient Dell workstation as a home server for many years. I only stopped because I gave it away to a student.

[74] I wouldn't actually consider version control to be that important as a medium of communication for users of a library, framework, program, whatever. However, I have had some instances, where a library changed unexpectedly (and the author didn't use proper semantig versioning !!!!), and being able to track down changes with their justification has been immensely helpful. Version control is also a good record of how work has evolved, when you need to get back in the groove after returning to a project you have not actively developed for a long while.

The big idea is that for elegant programs and proper understanding, it helps to view the code in a way that helps you the most, and when distributing it, you should aid the understanding of others. The environment, both physical and digital, in which you write code is as important as the code itself. By optimizing this environment and presenting your code clearly, you make elegance more achievable.

2.5 Elegant code and the cost of inelegant code

Let's reiterate the main idea:

Elegant code provides tangible benefits that extend far beyond aesthetic satisfaction. Code that clearly communicates its intent requires less mental overhead to understand, modify, and debug. This translates directly to reduced maintenance costs, fewer bugs, and increased development velocity over time. When confronted with a complex problem or unexpected behavior, elegant code offers clarity where obscure implementations force developers to untangle nested logic and hidden assumptions.

Conversely, inelegant code – even in small, seemingly isolated components – accumulates as technical debt. This debt compounds interest in the form of bugs that are difficult to isolate, features that become increasingly complex to implement, and onboarding processes that grow more painful with each new team member. What begins as a "temporary shortcut" or "just getting it working" often calcifies into permanent architecture that constrains future development. The cost of inelegance is rarely paid upfront but extracted slowly through countless hours of confusion and frustration.

While programmers may disagree on specific elements that constitute elegant code, the pursuit of elegance itself should be universal. More important than any particular style choice is consistency throughout a codebase. A team that establishes and adheres to clear conventions, regardless of what those conventions are, will produce more maintainable software than one where each developer follows different practices. The most valuable patterns are those applied uniformly—they become the reliable grammar through which your code communicates its purpose to present and future maintainers.

Chapter Three

Programming in the small

It is very easy to take the features of modern programming language for granted and as essential. You couldn't imagine a programming language not having these features. I am talking about features like proper if statements, functions, recursion, proper loops (of all shapes and sizes), access visibility modiifiers (so you can decide what is public, what is private and everything in-between), proper module systems, so that whole programs don't share a single namespace with their libraries, user-definable data structures, so that you can assign more meaning to the data you are working with, and so on.

All of these things are concepts that did not exist in programming languages, at some point. Better yet, we have been doing programming for decades before each of them has been introduced. We are introducing more more and more concepts all the time, some of them are user-facing, some of them simplify or improve the implementation of programming languages themselves. They are either born of theory, or of necesssity.

At one moment in history, we had to start thinking about how to organize programs. This was at both a small scale, and a large scale. The large scale has previously had much less attention because truth to be told, our programs just weren't that big. And there weren't as many engineers working on them. And the programs did not have to deal with interactions with many other things. Very often, they did not have to deal with interactions with operating systems and other programs running on the same machine. And if they did, the situation was far simpler because we often only had a single-core CPU, and the CPU was far more stupid and less magic about the way it executed code.[75]

A pivotal moment, which inspired the name of the two main parts of the book was when we formally started talking about "programming in the small", and "programming in the large". This breakthrough of recognizing the two distinct scales of programming came with the publication of a seminal paper by Frank DeRemer and Hans Kron in 1976, titled **"Programming-in-the-Large Versus Programming-in-the-Small"** in IEEE Transactions on Software Engineering.[76] This paper explicitly distinguished between the concerns of writing individual modules and algorithms (programming in the small) versus the challenges of

[75]You think the code that you write makes it into the binary? And that the operating system will not be creative about scheduling it? And that the CPU will not do dynamic reordering, speculations and branch predictions, and do black magic with its caches to run it as fast as possible? Parallel programming is quite difficult, when you get down to it.

[76]You should pirate this paper if you don't have institutional acccess. IEEE wants like 45$ for it.

organizing complete systems composed of many interconnected modules (programming in the large).

DeRemer and Kron observed that the programming languages of their era were primarily designed for the "small" - they excelled at expressing algorithms, control structures, and local data manipulations. But they were woefully inadequate when it came to expressing the structure of large systems. Their thesis was that we needed specialized "Module Interconnection Languages" (MILs) that would focus exclusively on describing how components fit together.

Perhaps what's most interesting about this paper is that it was published in 1976, when software systems were minuscule by today's standards. If DeRemer and Kron were concerned about programming in the large in the 1970s, imagine what they would think of modern systems with millions of lines of code, distributed across thousands of modules, running on multiple machines, written by hundreds of engineers!

Their paper described **programming in the small** as the activity we were all familiar with - writing the individual components that perform specific tasks. It's about local reasoning, algorithms, and data structures. It's when we focus on the implementation details and behaviors within a single module. The tools we use for this are programming languages, algorithms, and data structures, and our main concerns are correctness, efficiency, and readability.

Programming in the large, on the other hand, deals with organizing systems composed of many modules. It's about managing relationships between components, establishing interfaces, and controlling dependencies. The focus shifts from how a specific computation is performed to how different parts of the system interact and communicate. The paper argued that these concerns required different tools and approaches.

You can see attempts to address these concerns in languages like C and C++, with their header files and implementation files. The idea was to separate interfaces from implementations, allowing programmers to understand how to use a module without delving into its internal workings.

Consider the following example of a simple linked list implementation in C. We have a header representing the user-facing functionality:

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H
```

```c
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct {
    Node* head;
    int size;
} LinkedList;

void list_init(LinkedList* list);

int list_prepend(LinkedList* list, int value);

int list_append(LinkedList* list, int value);

int list_remove(LinkedList* list, int value);

int list_contains(const LinkedList* list, int value);

int list_size(const LinkedList* list);

void list_clear(LinkedList* list);

#endif /* LINKED_LIST_H */
```

The C compiler actually does not really understand the notion of a header. The preprocessor, which resolves the directives starting with the `#` character does understand the `#include` statement has two possible sources for where a header might come from. We also have to wrap the whole file in a `#ifndef`, otherwise, the file could get duplicated in the long source for when all headers are finally resolved[77].

What ends up happening is that all the headers you reference in your `.c` file are pasted into that file, creating one very long file that the compiler can then process as a whole, going through it top to bottom. This was a very pragmatic solution at the time, since it was very cheap to implement, was very simple, fairly flexible and worked quite fast.

The syntax of the headers also isn't special in any way, and only represents forward declarations for the actual functions in the implementation file:

[77]The preprocessor is a bit smarter to ensure that diagnostics from your C/C++ compiler remain useful, but we can still consider it essentially text copy-pasta.

```c
#include "linked_list.h"
#include <stdlib.h>
#include <assert.h>

// a private helper function to create a new node
static Node* create_node(int value) {
    // you can try implementing the data structure on your own :)
}

void list_init(LinkedList* list) {
    // implementation omitted..
}

int list_prepend(LinkedList* list, int value) {
    // implementation omitted..
}

int list_append(LinkedList* list, int value) {
    // implementation omitted..
}

int list_remove(LinkedList* list, int value) {
    // implementation omitted..
}

int list_contains(const LinkedList* list, int value) {
    // implementation omitted..
}

int list_size(const LinkedList* list) {
    // implementation omitted..
}

void list_clear(LinkedList* list) {
    // implementation omitted..
}
```

The C/C++ approach falls short of DeRemer and Kron's vision of MILs in several ways. Headers are merely textual inclusions rather than formal module boundaries. They don't provide true namespace isolation (C++ namespaces came much later and are still not perfect). There's no formal mechanism for explicit import and export controls until C++20's module system. Headers are preprocessor-based rather than being a fundamental language-level concept.

Modern languages have increasingly incorporated features to address the "programming in the large" challenges. Java has its package system, later enhanced

with the module system in Java 9. ML and OCaml have sophisticated module systems with signatures. Rust has its crate and module system. Python has packages and imports. Common Lisp has packages and systems. R6RS Scheme has libraries that everybody hates.[78] The list goes on.

Interestingly, the idea of having separate interface files did not really catch on. Apart from C/C++, there is only a couple programming languages that do something like that[79], for instance Ada, or OCaml. The rest of the languages typically does not utilize two separate file types. And the compilers decide what is the interface and what isn't based on the contents of the one implementation file. Here is an example in Ada:

```
-- stack.ads
package Stack is
   procedure Push (Item : in Integer);
   function Pop return Integer;
   Stack_Empty : exception;
end Stack;
```

For modules in Ada programs and libraries, you have a specification file, as shown above (the **s** in **ads** stands for **specification**), which only contains the signatures, and a body file, which contains the actual implementation:

```
-- stack.adb (body)
package body Stack is
   type Table is array (Positive range <>) of Integer;
   Space : Table (1 .. 100);
   Index : Natural := 0;

   procedure Push (Item : in Integer) is
   begin
      Index := Index + 1;
      Space (Index) := Item;
   end Push;

   function Pop return Integer is
      Result : Integer;
   begin
      if Index = 0 then
         raise Stack_Empty;
      end if;
      Result := Space (Index);
```

[78] And if you are curious, Emacs Lisp has nothing. Lmao.
[79] Typically better than C/C++'s minimalistic solution.

```
      Index := Index - 1;
      return Result;
   end Pop;
end Stack;
```

The Ada tangent is a very interesting one, and worthy of exploring. Ada is a significant programming language developed in the late 1970s at the direction of the U.S. Department of Defense for mission-critical systems. Unlike many languages that evolved through academic research or community development, Ada was created through a formal requirements process focused on reliability and safety.

The language incorporated features that were considered advanced for its time: strong typing, comprehensive modularity through packages, built-in concurrency, generics, and exception handling. At the moment, a lot of these features are considered essential for modern programming languages.

What distinguishes Ada is its design philosophy that emphasizes catching errors at compile time rather than runtime. It has a very strict type system,[80] a clear separation between interface and implementation in its package system, and a robust concurrency model through tasks. Ada uses a somewhat verbose syntax that reduces ambiguity and includes features like preconditions and postconditions for functions.[81]

The language continues to be used in domains where software failures could have severe consequences: air traffic control, railway systems, spacecraft, defense systems, and medical devices. This demonstrates that while some pro-

[80] Strong strict typing is pretty much essential in mission-critical code as it helps catch type errors at compile time rather than runtime.

[81] The pre- and post-conditions were a relatively late addition to Ada (2012), and they represent form of a development pattern called **Design by Contract**. Design by Contract is a programming approach where components interact based on formal agreements: preconditions specify what a function requires to work correctly, postconditions declare what it guarantees upon completion, and invariants define conditions that must always hold true. It establishes clear responsibilities:

- callers must satisfy preconditions,
- implementations must ensure postconditions

This creates a framework that treats software interfaces like legal contracts with mutual obligations. Design by contracct was invented several decades earlier by Bertrand Meyer with his Eiffel programming language. Eiffel is mostly just a research language, but very interesting to check out!

grammers might find Ada's strict rules limiting, these same constraints provide necessary safeguards in safety-critical applications.

Unfortunately, Ada didn't end up taking the programming world by a storm, and never reached a mainstream status. Its ideas however influenced an emphasis on safety in later languages, such as Rust.

The distinction between programming in the small and programming in the large remains very relevant today. As systems grow more complex, the challenges of organizing and coordinating large codebases have only become more pressing.

The growth of microservices architecture could even be seen as another approach to managing the complexity of programming in the large - decomposing systems into smaller, more manageable pieces that can be developed and deployed independently. However, using microservices adds complexity that is not justified for the scale of most projects. As with everything else, we need to think, and figure out the least wasteful way to tackle a problem.[82]

In the chapters that follow, we'll explore both programming in the small and programming in the large. We'll start with the small - how to write clear, efficient, and elegant code at the level of individual functions, classes, and modules. Then we'll expand to the large - how to organize these components into coherent systems that can grow and evolve over time. But even as we examine these concepts separately, remember that they are intimately connected. The decisions you make at the small scale affect what's possible at the large scale, and the structures you establish at the large scale influence how you approach the small.[83]

Keep in mind, this book will not give you complete solutions, we will discuss issues that are faced by developers all the damn time, and what we can do to ameliorate them. We will give you context, a lot of context, that will help you

[82] Microservices were quite the fad a few years ago, but now, most experienced programmers take a more measure approach. If you don't need to scale horizontally using microservices, then they are a lot of extra work. Microservices are, however, useful when you are combining several different technologies, and are more malleable if you decide to make a rewrite from one language into another (or other major refactors), since it is *relatively easy* to rewrite services one by one, incerementally replacing the old ones with new ones in a living system.

[83] Another way to think about is that if you create an intractable mess at the smallest scale, you don't have the clarity of mind to think about the highest level abstractions and architectural decisions you could make.

figure out where to go next, if you want to explore any solution in-depth. A book about the merits of object oriented programming will probably not give you fair and unbiased overview, of how you could alternatively model applications under a functional paradigm, that approaches programming under a completely different philosophy.

For now, let's focus on the fundamentals of programming in the small - the practices that lead to clear, maintainable code at the level of individual components. After all, even the largest cathedrals are built one stone at a time.[84]

[84] If you have suggestions for any cathedrals or churches I should visit in Europe, please let me know at **me@mag.wiki**. I like taking pictures of them.

3.1 Line lengths and whitespace

Let's start with something very small, and manageable, line lengths and whitespace. I have already previously mentioned the recommended line length limit of 80 characters.

This specific limitation of 80 characters comes from the early days of computing. IBM punch cards, introduced in the late 19th century and widely used until the 1970s, could hold exactly 80 characters per card. Later, when text terminals replaced punch cards, manufacturers naturally adopted the same 80-column width as the standard. VT100 terminals, which became a de facto standard, displayed 80 columns by 24 rows of text. Early programmers grew accustomed to this constraint, and it's remarkable how this technological limitation from the era of physical media continues to influence our digital practices today.

Now, obviously, we aren't limited by punch cards or VT100 terminals anymore. Our modern monitors can display far more than 80 characters horizontally, especially with the prevalence of widescreen and ultra-wide displays. I personally work on a 28-inch 4K monitor on the right side and a UW-WQHD 34 inch curved monitor on the left side[85] most days, which could theoretically display hundreds of characters per line at a readable size. So why do we still care about line length?

While the 80-character limit might seem arbitrary and outdated, keeping lines of reasonable length serves several practical purposes beyond tradition. For my own code, I generally try to stay under 120 characters, though I'm not militantly strict about it. This provides a good balance between using available screen space and maintaining readability. It is also something you get a feel for naturally, and you don't even have to worry about it

Long lines are simply harder to read – our eyes have to travel farther, and it becomes easier to lose track of where we are. This is the same reason why newspapers and magazines use columns instead of stretching text across the entire page width. The typographical principle that 60-70 characters per line is optimal for reading prose applies similarly to code.

[85] Both ThinkVision, both a little more expensive than they should have been, both, however, gentle to my eyes.

We don't have to aim for such a small limit because we count whitespace into that limit, so it can be way more than 80 even, just not an extreme amount.

But there's a deeper reason why many style guides, linters, and programmers with strong opinions (such as myself) enforce line length limits. Long lines often indicate problematic code structure. Consider this example:

```
if (user.isAuthenticated() && user.hasPermission("edit") && !
document.isLocked() && document.owner == user.id && document.status !=
"archived" && system.allowsEditing()) {
    // Allow editing
    document.makeEditable();
}
```

This sprawling condition is difficult to understand at a glance and adds cognitive load. It could be improved in several ways. One approach is simply breaking it into multiple lines with proper indentation:

```
if (user.isAuthenticated() &&
    user.hasPermission("edit") &&
    !document.isLocked() &&
    document.owner == user.id &&
    document.status != "archived" &&
    system.allowsEditing()) {
    // Allow editing
    document.makeEditable();
}
```

Where you put the **&&** is up to you, some people prefer to do the opposite way:

```
if (user.isAuthenticated()
    && user.hasPermission("edit")
    && !document.isLocked()
    && document.owner == user.id
    && document.status != "archived"
    && system.allowsEditing()) {
    // Allow editing
    document.makeEditable();
}
```

For very long conditions, where indentation of the condition is the same as the body, you may opt to place the opening brace of the condition on the next line, to visually separate the body of the control structure from its condition:

```
if (user.isAuthenticated()
    && user.hasPermission("edit")
```

```
    && !document.isLocked()
    && document.owner == user.id
    && document.status != "archived"
    && system.allowsEditing())
{
    // Allow editing
    document.makeEditable();
}
```

If you are using a language that has a strong opinion about it, just follow whatever is the suggestion. Your formatting shouldn't stand out.

Better, but still complex. A more elegant approach might be inverting conditions for early termination:

```
if (!user.isAuthenticated()) return;
if (!user.hasPermission("edit")) return;
if (document.isLocked()) return;
if (document.owner != user.id) return;
if (document.status == "archived") return;
if (!system.allowsEditing()) return;

// If we got here, all conditions are met
document.makeEditable();
```

Or abstracting the complex condition into a well-named function:

```
if (canUserEditDocument(user, document)) {
    document.makeEditable();
}
```

If you don't want to create a function, you can assign the test of the conditions to a variable.

Each of these transformations was essentially forced by a line length constraint. Without such a constraint, it's tempting to just keep adding conditions to a single line, creating what some call "freight train" code that's difficult to modify or debug.

Similarly, deeply nested blocks often produce lines that exceed reasonable length limits:

```
function processOrder(order) {
    if (order.isValid()) {
        if (order.items.length > 0) {
            if (checkInventory(order.items)) {
```

```
                if (order.paymentMethod) {
                    if (processPayment(order.paymentMethod,
order.totalAmount)) {
                        if (updateInventory(order.items)) {
                            return generateOrderConfirmation(order);
                        } else {
                            return "Error updating inventory";
                        }
                    } else {
                        return "Payment processing failed";
                    }
                } else {
                    return "No payment method specified";
                }
            } else {
                return "Items out of stock";
            }
        } else {
            return "Order contains no items";
        }
    } else {
        return "Invalid order";
    }
}
```

This "pyramid of doom"[86] is a maintenance nightmare. Line length limits would force you to refactor into something more manageable:

```
function processOrder(order) {
    if (!order.isValid()) {
        return "Invalid order";
    }

    if (order.items.length === 0) {
        return "Order contains no items";
    }

    if (!checkInventory(order.items)) {
        return "Items out of stock";
    }

    if (!order.paymentMethod) {
        return "No payment method specified";
    }

    if (!processPayment(order.paymentMethod, order.totalAmount)) {
```

[86] Which may even overflow on the pages of the book.

```
        return "Payment processing failed";
    }

    if (!updateInventory(order.items)) {
        return "Error updating inventory";
    }

    return generateOrderConfirmation(order);
}
```

When it comes to formatting and whitespace more broadly, you should generally follow the established conventions of your language community. Some languages have very clear formatting expectations – Go has gofmt, Rust has rustfmt, Python has PEP 8. Others like C and C++ have multiple competing styles. If you're working on a team or an open-source project, conform to their existing style. If you're starting from scratch, pick a style that's common in the ecosystem, document it, and be consistent.

I strongly recommend using an automated formatter to enforce whatever style you choose. We humans are remarkably bad at maintaining perfect consistency, and discussions about code formatting are perhaps the least productive debates programmers can have. Let the machines handle it so you can focus on the substance of your code.

One particular issue that deserves special mention: never mix tabs and spaces for indentation. This creates code that looks properly aligned in your editor but breaks in any editor with different tab width settings. Most modern editors can be configured to automatically convert tabs to spaces or vice versa, and many do this by default based on language. Some editors also offer the option to make whitespace visible – showing dots for spaces and arrows for tabs – which can be helpful for identifying inconsistencies.[87]

That said, there is one situation where mixing tabs and spaces makes sense: using tabs for indentation and spaces for alignment. This approach lets each developer set their preferred indentation width while maintaining proper alignment of code elements:

```
// Using tabs for indentation, spaces for alignment
function calculateTotal(items) {
→   let total     = 0;
```

[87] Ideally your color scheme or editor highlights the whitespace visualizing characters in a subdued manner, you don't want whitespace in bright colors.

```
→   let taxRate   = 0.07;
→   let shipping  = 5.99;
→
→   for (const item of items) {
→   →   total += item.price * item.quantity;
→   }
→
→   return total + (total * taxRate) + shipping;
}
```

This strategy is particularly useful in languages where alignment can significantly enhance readability, which wasn't the case the previous example. In Common Lisp, for instance, aligning the names and values in a `let` form makes the structure clearer:

```
(defun process-customer (customer)
  (let ((name      (customer-name customer))
        (address   (customer-address customer))
        (orders    (customer-orders customer))
        (balance   (customer-balance customer)))
    (process-the-data ...)
    (and-return-something ...)))
```

The `let` form creates a lexical scope where variables are bound to specific values, making those bindings available only within the body of the expression, allowing for local variable definitions that don't affect the surrounding environment.

If you are unfamiliar with this term **lexical scope**, then hear ye, hear ye.

There are multiple ways that languages deal with scope. Lexical scope is the option that you are probably used to. Common Lisp also supports dynamic scope, where variables are pushed on a stack, and whatever is bound to that name closest to the top of the stack is what is used.

Here is an example:

```
;; define a special (dynamically scoped) variable
(defvar *multiplier* 10)

;; function that uses the dynamic variable
(defun multiply (n)
  (* n *multiplier*))

;; normal call uses the global value
(multiply 5)   ; => 50 (5 * 10)
```

```
;; temporarily override the dynamic variable
;; the *multiplier* on the top of the stack is now 2
(let ((*multiplier* 2))
  (multiply 5))  ; => 10 (5 * 2)

;; back to using the global value
(multiply 5)  ; => 50 again
```

This is a double-edged sword. Improper usage of dynamic scope can lead to very confusing errors. More you know :)

Now, the same alignment applies for keyword arguments and class slot definitions:

```
(defclass product ()
  ((name        :initarg :name        :accessor product-name)
   (price       :initarg :price       :accessor product-price)
   (stock       :initarg :stock       :accessor product-stock)
   (category    :initarg :category    :accessor product-category))
  (:documentation "Represents a product in the inventory system."))
```

This is how you define a class in Common Lisp. The **slot** is just the lisp slang for a class field. If you are curious, this is how you can make an instance of it:

```
(make-instance 'product :name     "A really old thinkpad"
                       :price    "Priceless, bro"
                       :stock    "One and only, homie"
                       :category "Bitchin")
```

Finally, use blank lines judiciously to separate logical sections of code. Always put an empty line between function definitions – this is standard practice in virtually every language. Within functions, use blank lines to separate logical groups of operations. There's no rigid rule for this – it's about making the structure of your code visually apparent.

For instance, you might group variable declarations together, followed by a blank line, then the main processing logic, another blank line, and finally the return statement:

```
def analyze_data(raw_data):
    # Initialize variables
    processed_data = []
    error_count = 0
    total_items = len(raw_data)
```

```
    # Process each data point
    for item in raw_data:
        try:
            result = transform_item(item)
            processed_data.append(result)
        except ValueError:
            error_count += 1

    # Return analysis results
    return {
        "processed": processed_data,
        "err_rate": error_count / total_items if total_items > 0 else 0,
        "total_processed": total_items - error_count
    }
```

Or you might add a blank line after a complex operation that uses several temporary variables that aren't needed later:

```
int calculateOptimalRoute(Graph* graph, Node* start, Node* end) {
    // Initialize data structures
    PriorityQueue* queue = createPriorityQueue();
    HashMap* distances = createHashMap();

    // Populate initial distances
    for (int i = 0; i < graph->nodeCount; i++) {
        setHashMap(distances, graph->nodes[i], INT_MAX);
    }
    setHashMap(distances, start, 0);

    // Run Dijkstra's algorithm
    insertPriorityQueue(queue, start, 0);
    while (!isEmptyPriorityQueue(queue)) {
        Node* current = extractMinPriorityQueue(queue);
        int currentDist = getHashMap(distances, current);

        // Process all neighbors
        for (int i = 0; i < current->neighborCount; i++) {
            Node* neighbor = current->neighbors[i];
            int weight = current->weights[i];
            int tentative = currentDist + weight;

            if (tentative < getHashMap(distances, neighbor)) {
                setHashMap(distances, neighbor, tentative);
                insertPriorityQueue(queue, neighbor, tentative);
            }
        }
    }
```

```
    // Clean up and return result
    int result = getHashMap(distances, end);
    destroyPriorityQueue(queue);
    destroyHashMap(distances);

    return result;
}
```

This is a good compromise. However, I tend to be even more liberal with my whitespace usage, and flank all control structures, unless it is the last one in a block, with an empty line. So I would put an empty line here too:

```
    for (int i = 0; i < graph->nodeCount; i++) {
        setHashMap(distances, graph->nodes[i], INT_MAX);
    }
                                                    // <- new empty line
    setHashMap(distances, start, 0);
```

These blank lines aren't just arbitrary – they reflect the logical structure of the algorithm and help readers understand the code's organization at a glance.

Ultimately, line length limits and whitespace usage are small details, but programming is an activity where small details accumulate to create either clarity or confusion. By being thoughtful about these aspects of your code, you reduce friction for everyone who needs to read, understand, and modify it – including your future self.

One final note is that my recommendation also goes against the pursuit of making solution in the least amount of lines. You may have heard people say things like: "This window manager only has 2000 lines of code (LOC)!"

Well, my suggestions go against that. If you want a similar, yet still useful metric, either count non-empty lines, or semicolons, in the languages that have them.

3.2 Source code files

Now that we've discussed line lengths and whitespace within the code, let's take a broader view and consider the organization of whole source files. Every file in your project, should have a clear purpose and structure.

Long source files can be problematic. When a file grows beyond a few hundred lines (or, let's arbitrarily say, more than 1500), it becomes difficult to navigate and understand as a cohesive unit. The upper limit depends somewhat on the language and the nature of the code, but I think you can develop a feel for it. There are exceptions, of course – generated code, certain types of data-heavy files, or code in languages where the class-per-file convention doesn't apply might legitimately be longer.[88] But if you're regularly creating multi-thousand-line source files by hand, it's worth considering whether they could be split into more focused, single-responsibility components.

The history of file size limitations is interesting. In early computing, physical constraints like memory limited file sizes. The CP/M operating system, popular in the late 1970s, limited files to 8 megabytes, which seemed enormous at the time. Today, while our computers can handle massive files, our human cognitive limitations remain unchanged. We simply can't hold thousands of lines of code in our working memory at once. If you ever manage to reach a source file that has more than a megabyte, and isn't generated from anything, please email me.

Anyways, now that we have the file length down, let's consider the actual contents. After any language-required headers (like the shebang line in shell scripts or the package declaration in Java), imports or includes typically come first. We declare dependencies and establish the context for the rest of the file. It's good practice to organize them logically, usually in groups separated by blank lines:

```
# Standard library imports
import os
import sys
import json
from datetime import datetime

# Third-party library imports
import numpy as np
```

[88] There is utility a whole projects being in the same file. This makes it more portable/practical, since a single file is easier to distribute. Particularly in C, there is a trend of single file (usually single header) libraries.

```
import pandas as pd
import requests

# Local application imports
from myapp.models import User
from myapp.utils.formatting import format_currency
```

(I wouldn't add these comments, this is just to illustrate the logical grouping)

This organization immediately communicates the file's dependencies to readers. They can see at a glance what standard facilities are being used, what external libraries are required, and what local components are involved.

Avoid using wildcard or asterisk imports when possible. Instead of `from numpy import *`, which pulls all symbols from numpy into your namespace potentially causing naming conflicts,[89] prefer explicit imports like `import numpy as np`. This makes it clear where each function or class is coming from:

```
# wildcard import
from numpy import *
result = sqrt(array([1, 2, 3]))

# explicit import
import numpy as np
result = np.sqrt(np.array([1, 2, 3]))
```

There are exceptions to this rule. Maybe this file is `numpy` heavy, and it does not depend on much beyond that, and typing `np.` everywhere would be much too redundant. Some libraries are also specifically designed to be imported with wildcards. Parser combinator libraries in functional languages often work this way, as do many ORM query builders. Rust acknowledges this pattern formally with "prelude" modules specifically designed for wildcard importing:

```
// Importing a prelude is an accepted practice in Rust
use sqlx::prelude::*;
```

After imports, it's a fairly good practice to place constants and global variables. These definitions establish the bindings that the rest of the file will work with, and that may be exported as public API:

[89] By doing reckless asterisk imports, you are making your code more fragile to changes of the libraries you are importing from. A symbol that could have been previously disambiguated can be in a conflict, because now two libraries may be exporting it.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_BUFFER_SIZE 1024
#define DEFAULT_TIMEOUT 30000  // milliseconds

static const char* CONFIG_FILE_PATH = "/etc/myapp/config.json";
static int global_error_count = 0;

// Functions follow...
```

The rest of the file should follow a logical flow, generally defining building blocks before they're used. While modern compilers and interpreters don't typically require this ordering (C and C++ have forward declarations, and many languages do multiple passes)[90], it makes the code more readable for humans who process information sequentially.

This leads to a natural organization where helper functions come before the functions that use them, and the main entry point (if applicable) comes toward the end of the file:

```c
// First, define utility functions
static void log_error(const char* message) {
    fprintf(stderr, "ERROR: %s\n", message);
    global_error_count++;
}

static char* read_file_contents(const char* path) {
    FILE* file = fopen(path, "r");
    if (!file) {
        log_error("Could not open file");
        return NULL;
    }

    // File reading implementation...
    fclose(file);
    return buffer;
}

// Then define main business logic functions
```

[90] Forward declarations are an interesting concept as well. They are often present in languages that were at one point single-pass. Meaning that the compiler essentially only went through the file once, and it wouldn't do any cross-referencing after the fact. In practice, all contemporary C and C++ compilers are multi-pass, but we need to preserve the original semantics, or far too much would break.

```
void process_config() {
    char* config = read_file_contents(CONFIG_FILE_PATH);
    if (!config) {
        log_error("Failed to read configuration");
        return;
    }

    // Process configuration...
    free(config);
}

// Finally, the main function
int main(int argc, char* argv[]) {
    process_config();

    if (global_error_count > 0) {
        printf("Completed with %d errors\n", global_error_count);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Within class definitions, similar principles apply. In object-oriented languages, it's common to place fields and properties at the top of the class, followed by methods. Constructors and destructors often either come first (showing initialization) or last (showing the full lifecycle):[91]

```
class Customer {
private:
    // Fields first
    std::string name_;
    std::string email_;
    int customer_id_;
    std::vector<Order> orders_;

public:
    // Constructor
    Customer(std::string name, std::string email)
        : name_(std::move(name)), email_(std::move(email)),
customer_id_(0) {}

    // Destructor
    ~Customer() {
        // Cleanup if needed
    }
```

[91] Whichever way you prefer, just be consistent

```
    // Accessors
    const std::string& name() const { return name_; }
    const std::string& email() const { return email_; }
    int customer_id() const { return customer_id_; }

    // Business logic methods
    void place_order(const Order& order) {
        orders_.push_back(order);
    }

    double calculate_total_spend() const {
        double total = 0.0;
        for (const auto& order : orders_) {
            total += order.total_amount();
        }
        return total;
    }
};
```

In C++, where access modifiers create distinct sections within a class, the conventional order is often public members first (showing the interface), followed by protected members (for inheritance), and finally private implementation details. Some coding standards reverse this, putting private members first to emphasize data hiding. Either approach is fine, but be consistent within a project.

Smalltalk, one of the earliest object-oriented languages, had an interesting approach to file organization. Instead of text files, Smalltalk code lived in an "image" – a snapshot of the entire system state. Classes and methods were organized in a browser that showed hierarchical relationships, making the concept of "file organization" quite different. Modern environments have returned to some of these ideas with sophisticated IDE navigation tools.

We will focus on Smalltalk a bit more when we discuss Object-Oriented Programming, as Smalltalk is both its purest form, and simultaneously what appears to be a genetic dead-end. I suppose, in some way, it was too good for us.

Beyond these general principles, place related items near each other. If two functions work closely together or one calls the other, keep them adjacent in the file. This proximity creates a cohesive narrative flow through the code:[92]

[92]This is another important point of what I am trying to convey. Our programs are best served as narratives. Humans are wired for narratives. We frame almost everything all the time into narratives. Mnemonic tools often involve creating narratives for the things you want to remember.

```lisp
;; Group related functions together
(defun parse-customer-record (record)
  (let ((fields (split-string record ",")))
    (make-customer :name  (first fields)
                   :email (second fields)
                   :id    (parse-integer (third fields)))))

(defun validate-customer (customer)
  (and (valid-name-p       (customer-name customer))
       (valid-email-p      (customer-email customer))
       (positive-integer-p (customer-id customer))))

;; These functions operate on different entities, so they're separated
(defun parse-product-record (record)
  (let ((fields (split-string record ",")))
    (make-product :name  (first fields)
                  :price (parse-float (second fields))
                  :stock (parse-integer (third fields)))))
```

For files that grow larger despite your best efforts at decomposition, code folding becomes invaluable. Code folding is available in most modern editors, and it allows you to collapse sections of code to focus on what's relevant. Some languages provide explicit support for this with region markers, as does C#, for instance:

```csharp
// In C#, you can use #region to define foldable sections
#region Customer Management Functions

public Customer CreateCustomer(string name, string email) {
    // Implementation...
}

public bool UpdateCustomer(int id, Customer updatedInfo) {
    // Implementation...
}

public bool DeleteCustomer(int id) {
    // Implementation...
}

#endregion

#region Order Processing Functions

public Order CreateOrder(int customerId, List<OrderItem> items) {
    // Implementation...
}
```

```
// More order functions...

#endregion
```

Even in languages without built-in folding support, you can usually achieve similar results with comments that your editor recognizes. Emacs users like me often employ `outline-minor-mode`, which can fold sections based on comment patterns or indentation. Vim, Neovim, Helix, and Kakoune all offer folding capabilities that can be configured to work with your language of choice. Every single IDE you can think of has it.

Another good practice is to always separate top-level items (functions, classes, type definitions) with empty lines. This visual breathing room helps readers identify the boundaries between major components:

```
struct Customer {
    name: String,
    email: String,
    id: u32,
}

impl Customer {
    fn new(name: String, email: String) -> Self {
        Self {
            name,
            email,
            id: 0,
        }
    }

    fn validate(&self) -> bool {
        // Validation logic...
        true
    }
}

struct Order {
    items: Vec<OrderItem>,
    customer_id: u32,
    total: f64,
}

// Functions continue...
```

Finally, it's helpful to occasionally view your file from a "bird's eye view" to assess its organization. Many editors offer a command to collapse all foldable sections, giving you a structural overview. In VS Code, you might use "Fold All" (Ctrl+K Ctrl+0); in Emacs, `outline-hide-body`; in Vim, `zM`. Alternatively, tools like **ctags** can generate an index of all definitions in a file, providing a similar overview.

As one last example for this section, let's consider Prolog, a logic programming language, where file organization is fairly critical.[93] In Prolog, the order of clauses can affect program behavior due to its backtracking search strategy. A well-organized Prolog file groups related predicates and orders clauses to minimize backtracking, showing how even the structure of a source file can impact program efficiency. We will discuss Prolog more later on in this book, but this is how a file might look, where ordering matters:

```
% base case first - immediately handles terminating condition
factorial(0, 1) :- !.

% recursive case second - only tried when base case doesn't match
factorial(N, Result) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, SubResult),
    Result is N * SubResult.
```

If we reversed the order of the clauses, Prolog would needlessly try the recursive clause first.

All these considerations may seem like tinny details, but they compound to significantly affect readability and maintainability. A well-organized source file reduces the mental load on readers, allowing them to find what they need quickly and understand the code's structure without unnecessary effort. And once again, you are a reader too! So make your job easier, too.

[93] Prolog is a very interesting and extremely influential language, although often, there is a lot of intermediary languages between "a language you know" and Prolog. You are probably familiar with **SQL**, the lingua franca of relational databases. The **SQL** language is heavily inspired by **datalog**. And **datalog** is a language heavily, heavily inspired by Prolog. It just has a different evaluation model.

3.3 Naming things

Naming things is one of the genuinely difficult problems in programming. As Phil Karlton famously quipped, "There are only two hard things in Computer Science: cache invalidation and naming things." A good name communicates the purpose of a variable, function, or type, making code more readable and maintainable. Poor naming, on the other hand, can mislead readers and introduce subtle bugs.[94]

The primary goal of a name is to communicate the purpose of the symbol it represents.[95] When someone reads your function name along with its parameters and return type, they should be able to make a reasonable guess about what the function does, even if they don't see the implementation.[96] Consider this function signature:

```
int calculate(int a, int b);
```

This tells us almost nothing. Compare it to:

```
int add_integers(int first_number, int second_number);
```

The latter immediately communicates its purpose. We would be shocked if this function did anything other than return the sum of the two parameters.[97]

One of the first considerations when naming things is to follow the existing conventions of the language you're using. Each programming language has developed its own naming traditions over time. C++ uses camelCase for meth-

[94] The "Safe Code" subject at the Czech Technical University actually lists misleading naming among issues (documentation). However, they have hungarian notation in their examples (explained later on in this chapter), which still makes them moderately deplorable.

[95] By symbol I mean class, variable, struct, constant, function, method, macro, module name, package name, interface, trait, typeclass and whatever else you can think of a as a logical named unit in a given programming language.

[96] It is generally a sign of a badly executed library if its users have to constantly refer to its implementation, as opposed to being good with what the LSP/Editor suggests in autocomplete plus documentation.

[97] This function could have been called just `add`, and in many languages, it would have been. However, C generics are a bit problematic and bordering on functionally non-existent, and C also has no notion of methods, or modules, so `add()` is very likely to polute the namespace of bigger projects. Remember that `add()` can stand for adding up to numbers (and here we have the problem of numerous number tupes), but also for adding an element to a collection (e.g. a vector or a linked list – although semantically, you could have a different names for it – `push` or `append`)

ods, snake_case for variables,[98] and PascalCase for classes. Java uses camelCase for methods and variables, and PascalCase for classes. Python generally uses snake_case for almost everything except classes, which use PascalCase.

If you're contributing to an existing project that has established naming conventions, follow them even if they differ from the language norms or your personal preferences. Consistency is more valuable than personal expression when it comes to code readability. If you believe the project's conventions need improvement, that's a discussion to have with the team rather than a unilateral decision to make in your next commit.[99]

The history of naming conventions includes some interesting diversions. Hungarian notation,[100] popularized at Microsoft in the 1980s, prefixed variable names with information about their type. For example, `szName` indicated a zero-terminated string, while `iCount` indicated an integer.[101] This made sense in early programming environments where type information wasn't easily available and using the wrong type could lead to subtle bugs.[102]

Today, Hungarian notation should be avoided at all costs. Modern languages have strong type systems, and development environments can show you the type of any variable on demand. Hungarian notation now just adds visual clutter without providing value. The original form was actually about semantic

[98] Often, but not always. C and C++ are both very inconsistent in naming conventions. There are many style guides with different ideas about how you should do things. Generally, the style guides are at their core the reflection of the time and culture they were created in. For C++, I used to generally try and follow the LLVM project style guide, since if someone is a deeply-knowledgeable authority on that language, its probably the guys developing a major compiler for it.

[99] I don't think this is really neccessary to say, but it helps to have justification for the changes you suggest. Just barging in and saying "your naming conventions are shit" probably won't garner you much support in your endeavor.

[100] It is called "hungarian" because it was invented by Charles Simonyi, who is hungarian. He was a programmer at Xerox PARC, one of the legendary places where major advances in computing were being made in the last century. He later joined Microsoft. There is a an alleged second justification for "hungarian", and that is "because it made programs look like they were written in some inscrutable foreign language".

[101] There are actually two types of the hungarian notation, Systems hungarian and Apps hungarian. In the first variant, we prefix with the actual data type, in the latter, we prefix with a more logical data type. `iCount` is Systems, `szName` is Apps – because there is no zero-terminated string type present in the language.

[102] And when we also had shittier editors, and very often, didn't even write code on a computer.

information (what the variable is for) rather than just the type, but most people implemented it incorrectly anyway.[103]

I also consider problematic is the "C" prefix some C++ programmers use for class names, like `CProduct` or `CCustomer`. This convention emerged from early Windows programming and Microsoft's MFC library, but it really adds no value. The "C" merely indicates that the symbol is a class, which is already obvious from context and potentially from PascalCase naming. Modern C++ code has abandoned this practice in favor of cleaner names like simply `Product` or `Customer`.[104]

That said, some prefixes remain useful in certain contexts. In C++, using an underscore prefix (`_name`) or suffix (`name_`) for class member variables can help distinguish them from local variables in methods, particularly in languages that don't require explicit `this.` or `self.` references. The `m_` prefix (like `m_name`) is less elegant but serves the same purpose.[105] In Rust, an underscore prefix is used to mark variables that are intentionally unused, suppressing compiler warnings. For example:

```
fn process_result(result: Result<String, Error>) {
    let _unused = result.expect("This just asserts, we don't use the
value");
    // or:
    let _ = some_function_with_side_effects();
}
```

It's important that a name communicates the purpose of what it's naming, not redundant information about what kind of symbol it is. Don't add `Class` to class names, `function` to function names, or `var` to variable names. This is stating the obvious and adds noise without value. A class representing a user should be `User`, not `UserClass`. A class storing configuration should be called `Configuration`, not `ConfigurationManager`.[106]

The importance of a name scales with the scope of the symbol. Variables used across an entire class or module deserve descriptive names. Conversely, vari-

[103] vUsing adjHungarian nnotation vmakes nreading ncode adjdifficult. –Mark Stock

[104] Of all the bad naming conventions, I hate this one the most.

[105] I don't like the `m_` prefix, since it stands for **member**, which registers the same way as the `C` prefix for **class**, but I am willing to begrudgingly accept its existence.

[106] I think I stole this from Mark Rendle, but I am not sure which of his talks it was. He is a great guy, always on the verge of mental breakdown.

ables with tiny scopes can be shorter without losing clarity. In a small closure/lambda function in Rust, short parameter names are perfectly clear:

```rust
let parsed_values = raw_data.iter()
    .filter_map(|s| s.parse::<i32>().ok())
    .collect::<Vec<_>>();
```

Here, `s` is fine because its scope is just a few characters long and the context makes its purpose obvious.

Some languages have developed distinctive naming conventions. Scheme, a Lisp dialect, has particularly well-thought-out conventions. Predicates (functions that return true/false) end with a question mark, like `string?` or `null?`. Mutating functions that change their arguments end with an exclamation mark, like `set!` or `vector-set!`. Type conversion functions often contain an arrow, like `string->number`. Here's how these conventions look in actual Scheme code:

```scheme
;; Predicate functions with ? suffix
(define (empty? lst)
  (null? lst))

;; Mutation functions with ! suffix
(define (reverse-in-place! vec)
  (let ((len (vector-length vec)))
    (do ((i 0 (+ i 1))
         (j (- len 1) (- j 1)))
        ((>= i j))
      (let ((temp (vector-ref vec i)))
        (vector-set! vec i (vector-ref vec j))
        (vector-set! vec j temp)))))

;; Conversion with -> notation
(define (string->integer str)
  (call-with-input-string str read))
```

These conventions make Scheme code quite self-documenting. Just by looking at a function name, you can tell if it's a predicate, if it mutates its arguments, or if it's performing a type conversion. It is very helpful, since readability in the Lisps largely depends on the identifiers and formatting, we have very little punctuation to help us out.

Common Lisp, interestingly, doesn't follow these conventions as strictly. Its

predicates are less consistent, sometimes using `-p` suffixes (like `stringp`)[107] and sometimes using other patterns. However, it does have some unique conventions. "Earmuffs" (asterisks surrounding a name) mark special variables with dynamic scope, warning programmers about their unusual binding behavior:

```
(defvar *global-database* nil)

(let ((*global-database* (connect-to-database)))
  ;; Code using the temporarily bound special variable
  )
```

Constants in Common Lisp often use "plusmuffs" (plus signs around a name):

```
(defconstant +max-connections+ 100)
```

The Lisp family is somewhat unusual in allowing characters like `-`, `?`, `!`, and `+` in identifiers. Most languages restrict identifiers to alphanumeric characters and underscores to simplify parsing, especially when using infix operators. This is a fundamental distinction that's worth understanding.

In languages with infix operators, the parser needs to disambiguate between operators and identifiers. If `-` can be both a subtraction operator (as in `a - b`) and part of an identifier (as in `user-name`), the parser needs complex rules to determine which is which in any context. This is why most infix languages prohibit symbols like `-` in identifiers.

Lisp's prefix notation elegantly sidesteps this problem. Since operators are just functions in the first position of a list, like `(+ a b)`, there's never ambiguity about whether a symbol is an operator or part of an identifier. This allows Lisp dialects to use a much richer character set for identifiers, enabling the expressive naming conventions we see in Scheme and Common Lisp.

This explains why most mainstream languages have more restrictive naming rules - it's a direct consequence of their infix syntax, not an arbitrary limitation.

The language Nim takes a unique approach to naming flexibility. Nim is case-insensitive and underscore-insensitive, meaning these identifiers are all considered identical:

[107] There are also some rules about whether the suffix should be `p` or `-p`. If the identifier is a single word with no dashes in it, then `p` like `stringp`, if it is multiple, like `user-subscribed`, then `user-subscribed-p`. You will find things that break this convention.

```
proc addTwoNumbers(a, b: int): int =
  return a + b

# These calls are all identical to Nim
echo add_two_numbers(3, 4)   # Works
echo addTwoNumbers(5, 6)     # Also works
echo addtwoNUMBERS(7, 8)     # Still works
```

This eliminates whole categories of naming debates but might be confusing to programmers coming from other languages.[108]

In the early days of computing, when memory and display space were limited, there was pressure to keep identifiers short. C's standard library reflects this history with functions like `strcpy` (string copy) and `printf` (print formatted). There's no honor in continuing this tradition of omitting vowels or creating cryptic acronyms. Modern computers have no practical limit on identifier length, and modern editors have autocompletion.

This C code from the 1970s:

```
char *strcpy(char *dst, const char *src);
```

Would be more readable today as:

```
char *string_copy(char *destination, const char *source);
```

Unfortunately, we can't change C's standard library without breaking millions of programs, but we shouldn't perpetuate this style in new code.

When choosing a casing convention in a language without strong norms, I like to use PascalCase (also called UpperCamelCase) for types, traits, interfaces, and similar constructs, while using snake_case for functions, methods, and variables. This visually distinguishes types from values, making code easier to parse at a glance. Many languages use ALL_CAPS for constants, which is another useful visual distinction.

These clear conventions can be leveraged by syntax highlighting to provide additional visual cues. When your editor can identify types, functions, and

[108] Nim has an interesting heritage, drawing inspiration from several languages considered somewhat obscure today, including Oberon, Modula-2, and Modula-3, all from the Pascal family. These languages were designed by Niklaus Wirth and his colleagues with a focus on simplicity, safety, and efficiency. Nim also takes ideas from Python's syntax and Ada's type system, creating a unique blend of influences from both mainstream and academic languages.

variables based partly on their naming patterns, it can color them differently, further enhancing readability.[109]

Since we're discussing C idioms, it's worth mentioning a common mistake: adding the suffix `_t` to type names. This suffix is reserved by POSIX for standard type definitions. If you create your own type like `my_struct_t`, you risk conflicts with future standards. Instead, use no suffix at all.

Modern programming languages have features that allow more concise naming without sacrificing clarity. With methods, generics, or typeclasses, we can have a single `.length()` method that works on vectors, arrays, strings, or any other collection, rather than needing specific names like `vector_length()`, `array_length()`, and so on. This is part of why object-oriented and functional programming can lead to more readable code:[110]

```
// Each collection type has its own .len() method
let string_length = my_string.len();
let vector_length = my_vector.len();
let map_length = my_hashmap.len();
```

Many languages have semantic naming conventions that go beyond syntax. In Rust, there's a strong convention that constructors are named `new`:[111]

```
let my_string = String::new();
let my_vector = Vec::new();
```

If you're creating a new type in Rust, using any other name for the basic constructor like `create()` or `make_instance()` would violate user expectations. Similarly, conversion functions often use `from` and `into`:

```
let s = String::from("hello");
let v: Vec<u8> = s.into();
```

Following these semantic conventions makes your code more predictable for others. The most common operations should have the most conventional names.

[109] Some languages, like Haskell, actually require this distinction - types must start with an uppercase letter, while values and functions must start with lowercase.

[110] In C, we cannot really do this. This is partly what makes it incredibly verbose, and part of the original appeal of C++, and also of Object Oriented Programming at large.

[111] Rust actually doesn't have constructor methods. We are simply using functions that return a new instance, and are functionally no different from other functions.

Sometimes context affects naming priorities. Consider this Python code from my friend Kaden Bilyeu (aka Bikatr7), written for the Advent of Code programming competition:

```
from aocd import get_data,submit

def is_saf(lvl):
    for i in range(len(lvl)-1):
        if(lvl[i] == lvl[i+1]):
            return False

    dif = [lvl[i+1]-lvl[i] for i in range(len(lvl)-1)]
    inc = all(d > 0 for d in dif)
    dec = all(d < 0 for d in dif)

    if(not (inc or dec)):
        return False

    return all(1 <= abs(d) <= 3 for d in dif)

def chk_damp(lvl):
    if(is_saf(lvl)):
        return True

    for i in range(len(lvl)):
        tmp = lvl[:i] + lvl[i+1:]
        if(is_saf(tmp)):
            return True
    return False
```

By normal standards, this naming is terrible. `is_saf` presumably means "is safe," `chk_damp` might be "check damping," and `lvl` is likely "level." In production code, this would be unacceptable. But in a programming competition where typing speed matters, these shortcuts can make sense. The code was written to be produced quickly, not maintained for years. Context matters.

Single-letter variable names do have legitimate uses. In mathematical code, using `x`, `y`, and `z` for coordinates or `i`, `j`, and `k` for loop indices is often clearer than longer names because they match established mathematical conventions. For example:

```
def quadratic_formula(a, b, c):
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        return None  # No real solutions
    x1 = (-b + math.sqrt(discriminant)) / (2*a)
```

```
    x2 = (-b - math.sqrt(discriminant)) / (2*a)
    return (x1, x2)
```

Here, `a`, `b`, and `c` are more appropriate than `coefficient_a`, `coefficient_b`, and `coefficient_c` because they directly reference the standard form of a quadratic equation: $ax^2 + bx + c = 0$.

In Haskell, single-letter variables are common in function definitions, but for a different reason. Haskell's focus on composition and partial application means many functions are written in a point-free style where the arguments aren't explicitly named:

```
-- Double every element and keep only the even results
processNumbers :: [Int] -> [Int]
processNumbers = filter even . map (*2)
```

This style treats functions as transformations to be composed rather than operations on explicit variables. When variables are needed, they're often given meaningful names for complex values but single letters for simple parameters, following mathematical tradition.

The essence of good naming is that it makes code easier to understand for humans. A computer doesn't care what you call your variables – you could name everything `x1`, `x2`, `x3` and the program would run the same.[112] But you and your collaborators need to read, understand, and modify that code. Good names are documentation, explaining the purpose and intent of each piece of code.

In any case, be consistent. Consistency within a codebase is more important than following any particular naming convention. If half your functions use `snake_case` and half use `camelCase`, readers will be constantly distracted by the inconsistency.[113] Pick conventions that work for your project and stick to them religiously.

And since we mentioned the word documentation, let's look at it more broadly.

[112] Have you seend YandereDev's code? You would be surprised what people are capable of creating that somehow still kinda runs.

[113] I actually get physically annoyed at inconsistent code.

3.4 Documenting code

The importance of documentation in software development is something I'll assume you already know.

If you've ever had to work with an undocumented library or tried to modify code that has no documentation, you know the pain of trying to decipher what the original author was thinking.[114] Documentation is critical for both programming in the small (helping understand individual functions, types etc.), and programming in the large (providing a roadmap for navigating larger systems).

Documentation exists in several forms. There's **user-facing documentation**, which explains how to use a product from a business perspective. This is what most people think of when they hear "documentation": manuals, help files, or knowledge bases. But there's also **programmer-facing documentation**, which itself splits into two categories: documentation for those **using your code** as a library or API, and documentation for those **contributing to your codebase**.

In this chapter, we will concern ourselves with the programmer-facing documentation. User documentation is beyond the scope of this book.

Let's talk about documentation for library users first. This typically lives outside the source code itself, often in dedicated documentation files, websites, or READMEs. However, some languages have integrated documentation systems that extract documentation directly from source code. Rust is particularly good at this, with its documentation comments that can include executable code examples:

```
/// Adds two numbers together.
///
/// # Examples
///
/// ```
/// let result = my_crate::add(2, 3);
/// assert_eq!(result, 5);
/// ```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

[114] And the shame if the original author is you.

When you run `cargo doc`, Rust generates beautiful HTML documentation that includes these examples. But more importantly, when you run `cargo test`, Rust will actually execute the code in your documentation examples as tests. These "doctests" ensure that your documentation stays in sync with your code. If you change the behavior of a function without updating its documentation, the tests will fail.

This is a brilliant solution to the age-old problem of **documentation drift**, where documentation becomes increasingly inaccurate as code evolves. If your language doesn't have built-in support for this, it's worth investigating if there are third-party tools that offer similar functionality.

Not that many languages are as tightly integrated with documentation as Rust is, but many of them have tools in their ecosystems that can parse and generate documentation from specially formatted comments. Let's consider this completelely undocummented example of Scala:[115]

```
def calculateDiscount(items: List[Map[String, Any]]): Map[String, Double] = {
  val eligibleItems = items.filter(item =>
    item.contains("price") &&
    item.contains("category") &&
    item("price").toString.toDouble >= 20.0
  )

  val categoryTotals = eligibleItems.groupBy(item => item("category"))
    .map { case (category, items) =>
      val total = items.map(i => i("price").toString.toDouble).sum
      (category.toString, total * (if (total > 100) 0.15 else 0.1))
    }

  categoryTotals
}
```

We can add a comment that is parsed by the **scaladoc** tools that will document this function:

```
/**
 * Calculates category-based discounts for items meeting price thresholds.
 *
 * @param  items List of items with their details
 * @return  Map of category names to calculated discount amounts
```

[115] Scala is one of the two languages for the JVM that I respect, the other being Clojure. Scala is a language with a very difficult job - it has to be able to integrate with the JVM ecosystem at large, it has to be functional, but because it interacts with Java, there has to be some Object Oriented Programming features going on. It is not as pure a language as Haskell, but it is a step above Kotlin.

```
*/
def calculateDiscount(items: List[Map[String, Any]]): Map[String, Double] = {
  val eligibleItems = items.filter(item =>
    item.contains("price") &&
    item.contains("category") &&
    item("price").toString.toDouble >= 20.0
  )

  val categoryTotals = eligibleItems.groupBy(item => item("category"))
    .map { case (category, items) =>
      val total = items.map(i => i("price").toString.toDouble).sum
      val discountRate = if (total > 100) 0.15 else 0.1
      (category.toString, total * discountRate)
    }

  categoryTotals
}
```

Documentation for library users should primarily explain what things do, not how they're implemented internally. Users of your API typically don't care about the clever algorithm you used,[116] they just want to know what your function does, what inputs it accepts, what outputs it produces, and what errors might occur. There are exceptions to this rule, of course. If there's something about the implementation that users should know, like the choice of hashing algorithm in a security library – that should be included. It's often also worthwhile to include performance characteristics if they might affect how users should use your API.[117]

On the other side of the fence is documentation for contributors to your project. This consists of several layers. First is the code itself, which should be clear and self-documenting as we've discussed in previous chapters, and will discuss in future chapters. Second is the user documentation – contributors need to understand what the code is supposed to do from a user's perspective. Third is special documentation specifically for contributors.[118]

This contributor-specific documentation should help new people orient themselves in your codebase. Where should they start looking? What are the main components and how do they fit together? It should also justify any unusual design decisions. If you've done something in a non-standard way, explain why.

[116] Unless what you are providing **is** some general purpose algorithm.

[117] What I mean is that some operations provided by your library may be very costly, and you want to let the users know that they should not be using them often, and maybe utilize some buffering, batching.

[118] For instance, us Rust bros have the **rustc book**, which provides some hints into how the Rust compiler looks internally, and how to develop it.

Finally, it should document conventions specific to your project – coding style, branch naming conventions, testing requirements, and so on.

The fourth layer of contributor documentation is comments within the code. These shouldn't be excessive, but should explain things that might not be immediately clear or obvious from the code itself. The fifth layer is often overlooked but quite valuable: the history of the code as recorded in version control systems like Git.[119]

Git commits serve as documentation of how and why the code has evolved over time.[120] For this reason, it's important to write good commit messages. If you name your commit "Update file", I will kill you. That tells future developers nothing about what changed or why. Commits should be focused on a single logical change – it's generally a bad idea to have one commit changing two distinct components for unrelated reasons. They shouldn't contain sneaky unrelated changes that aren't mentioned in the commit message.

A good commit message follows conventions. Many projects use prefixes like `feat:`, `fix:`, or `chore:` to categorize changes. The message should be written in the imperative mood – "Add feature" rather than "Added feature" or "Adds feature". It should have a title line of less than 72 characters, which is the maximum length before Git will insert a line break when displaying messages. If your project is already wrangled into some agile framework with Jira or another instrument of pain,[121] it's helpful to cross-reference the relevant ticket in your commit message. This creates a link between business decisions and technical implementations, giving context to changes.[122]

When it comes to comments in the code itself, finding the right balance is tricky. Too many comments can clutter the code and make it harder to read, while too few can leave important context unexplained. Comments should explain things that may be less clear, may seem odd, or otherwise require justification.

---

[119] At the moment, Git is the version control system with the biggest market share. The things we describe here generally are applicable to all the versioning systems, such as Mercurial, pijul or fossil.

[120] This book is in a git repository. And because I am literally only using it as a file storage, I don't follow any of the commit message suggestions that I offer in this chapter. I just proverbially punch the keyboard every time, hehe. Probably don't do this for programming projects.

[121] I have a hate-hate relationship with Jira, where I hate it for being a buggy and sluggish software, and I hate that there isn't really much alternative to it.

[122] It is also helpful to have this information when you are tracking down where an issue was introduced.

Overcommenting is a common newbie mistake. You don't need comments like this:

```
# Increment counter by one
counter += 1
```

The code already clearly states what's happening. The code examples I provide in this book are probably often overcommented compared to what production code would have – but that's fine for educational purposes.

Undercommenting is also a mistake, particularly for complex algorithms or business logic. If a piece of code implements a specific business rule or edge case, a comment explaining the requirement can save future developers from mistakenly "fixing" what appears to be a strange implementation but is actually a deliberate design choice.[123]

```
# Orders over $100 get free shipping, but this doesn't apply to
# international orders due to regulatory requirements as of 2023
if order_total > 100 and order.country in DOMESTIC_COUNTRIES:
    order.shipping_cost = 0
```

Striking a good balance with comments is a matter of taste and experience. It takes time to develop a sense for what needs explanation and what doesn't.

In a collaborative company situation with proprietary code, when you're working as a team, or if you have a similarly tightly knit open-source project, it can be a good idea to sign comments that might be questioned. This allows a colleague looking at the code to know who to ask about it without having to dig through `git blame`. For example:

```
/*
 * Using a bubble sort here because the data set is guaranteed to be
 * small (<10 elements) and nearly sorted already. Benchmarking showed
 * no benefit from more complex algorithms in this specific case.
 * - LH, 1989-06-15
 */
```

[123] There is a bit of discussion to be had here. If there is a piece of code that looks wrong, and fixing it would break something else, doesn't that seem kind of weird? It seems like an improperly designed system, or in more general terms, an instance of technical debt. Clearly, there are two components of your system here interacting in an unintentional way. If it is within your reach and power, consider redesigning the implementation such that neither component "looks odd".

This practice isn't universally followed, and whether you do it depends on your team's preferences and workflow.

Documentation is an important part of elegant code, so don't skimp out on it. Ideally, you should be creating is you are creating the project. Just like writing integration tests, it helps you get some insight into whatever it is that you are doing.

3.5 Taming your hubris

If you've spent any time in the programming world, you've likely encountered what I call the "hubris problem." Many programmers, especially those early in their career, suffer from an overabundance of confidence that manifests in various ways. They believe their chosen solution is unquestionably the best, that their preferred technologies are superior to all alternatives, that they can build incredibly ambitious systems without breaking a sweat, and most dangerously, that they never make mistakes.

This hubris isn't entirely negative. The belief that you can build something grand and ambitious is actually essential for innovation. Without it, we wouldn't have many of the software systems we rely on today. Nobody would start building an operating system, a web browser, or a new programming language if they had a completely realistic assessment of the challenges involved. Sometimes a healthy dose of overconfidence is necessary to begin ambitious projects that change the world.

The problem arises when this confidence isn't tempered with appropriate caution and humility. At the small scale, this often manifests as overly clever code. We've all seen it - dense, cryptic one-liners that the author was clearly proud of, but that make maintenance a nightmare, or the opposite, a hundred lines that could have been 10 lines.

Consider this scenario: you need to find all prime numbers up to a certain limit. Here's an overly clever solution using the Sieve of Atkin:

```
def sieve_of_atkin(limit):
    # Initialize sieve
    sieve = [False] * (limit + 1)
    # Put in candidate primes based on quadratic forms
    for x in range(1, int(limit**0.5) + 1):
        for y in range(1, int(limit**0.5) + 1):
            # First quadratic form: 4x² + y² = n
            n = 4 * x**2 + y**2
            if n <= limit and (n % 12 == 1 or n % 12 == 5):
                sieve[n] = not sieve[n]

            # Second quadratic form: 3x² + y² = n
            n = 3 * x**2 + y**2
            if n <= limit and n % 12 == 7:
                sieve[n] = not sieve[n]
```

```
            # Third quadratic form: 3x² - y² = n (when x > y)
            n = 3 * x**2 - y**2
            if x > y and n <= limit and n % 12 == 11:
                sieve[n] = not sieve[n]

    # Eliminate composites by sieving
    for x in range(5, int(limit**0.5) + 1):
        if sieve[x]:
            for y in range(x**2, limit + 1, x**2):
                sieve[y] = False

    # Generate primes
    primes = [2, 3]
    primes.extend([x for x in range(5, limit + 1) if sieve[x]])
    return primes
```

Versus a more straightforward approach using the Sieve of Eratosthenes:

```
def sieve_of_eratosthenes(limit):
    # Create a boolean array and initialize all entries as true
    prime = [True for i in range(limit + 1)]
    p = 2

    while p * p <= limit:
        # If prime[p] is not changed, then it is a prime
        if prime[p]:
            # Update all multiples of p
            for i in range(p * p, limit + 1, p):
                prime[i] = False
        p += 1

    # Generate primes
    primes = [p for p in range(2, limit + 1) if prime[p]]
    return primes
```

And for truly small limits, sometimes the simplest solution is just a hardcoded table:

```
def small_primes(limit):
    # Hardcoded list of first few primes
    all_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47]
    # Return only primes up to the limit
    return [p for p in all_primes if p <= limit]
```

The Sieve of Atkin is theoretically more efficient for very large numbers, but it's complex, difficult to understand at a glance, and easy to implement incorrectly.

The Sieve of Eratosthenes is straightforward, well-understood, and sufficient for most practical applications. And for small ranges, a simple lookup table might be the most readable and efficient solution of all.

This exemplifies the KISS principle - Keep It Simple, Stupid. Often attributed to Kelly Johnson of Lockheed Skunk Works, this principle reminds us that simplicity should be a key goal in design. Systems work best when they're simple rather than complex.

Complex code is more prone to bugs, harder to test, and creates a higher cognitive load for everyone who has to work with it. There's a time and place for clever solutions, but it's usually when performance constraints genuinely demand it, not when you're trying to impress your colleagues or future readers of your code.

The history of programming languages offers interesting case studies in hubris and its interaction with ambition. Let's look at two contrasting examples: Rust and Lisp.

Rust emerged partly as a response to a particular kind of hubris - the belief that it was feasible for large teams to write safe, secure software in C++. Many talented programmers held this belief, including those at Mozilla working on the Firefox browser. They implemented best practices, used static analyzers, conducted code reviews, and followed all the established advice for writing secure C++. Yet memory safety bugs continued to plague their codebase.

This wasn't because the Mozilla engineers were incompetent - quite the opposite. They were among the best in the industry. The problem was systemic. C++ gives you tremendous power and flexibility, but it puts the entire burden of memory safety on the programmer. You might enforce rigorous standards on your own code, but what about your dependencies? What about code written by team members who joined recently? What happens when deadline pressure forces shortcuts?

The Mozilla engineers who conceived Rust realized that a different approach was needed. Instead of making safety opt-in through careful coding practices, they designed a language where safety was the default and unsafety required explicit opt-out. This was an ambitious goal, and there was certainly hubris in thinking they could create a systems programming language that was both safe and fast. But in this case, the ambition was justified, and Rust has successfully

carved out a niche as a language for systems programming where safety is paramount.

We shouldn't criticize C++ too harshly for its safety issues. It was designed in a different era, with different constraints and priorities. Compatibility with C was a primary goal, which necessarily limited how safe it could be. Every language makes trade-offs, and C++ prioritized backward compatibility and performance over safety guarantees.

Lisp represents a different kind of ambitious hubris. When John McCarthy created Lisp in the late 1950s, most programming was done in assembly language or early compiled languages like FORTRAN. These languages were closely tied to the machine architecture - the Turing machine model of computation. McCarthy was working on artificial intelligence and needed a language that could manipulate symbolic expressions more naturally.

Rather than starting from the Turing machine model, McCarthy based Lisp on lambda calculus, a mathematical formalism for describing computation developed by Alonzo Church in the 1930s.[124]

Lambda calculus offers a completely different way of thinking about computation, focusing on functions and their applications rather than state machines. It treats functions as first-class values that can be passed around, returned from other functions, and created dynamically. This was an extremely abstract concept for a programming language in 1958.

Lambda calculus and Turing machines represent two fundamentally different but equally powerful models of computation that emerged in the 1930s. While both can compute the same set of functions (as demonstrated by the Church-Turing thesis), they offer radically different perspectives on what computation is.

Turing machines model computation as a sequence of state changes on a memory tape - a mechanical process of reading, writing, and moving according to a finite table of rules. This naturally maps to imperative programming languages where you explicitly manipulate state.

[124]This is partly what inspired the title of this book, "Lambdas and Logos." Lambda represents this alternative model of computation based on functions rather than state machines, while Logos represents the idea of programming as a form of communication and reasoning.

Lambda calculus, by contrast, models computation as a process of function application and substitution, with no concept of state or time - just pure transformation of expressions according to simple rules. This corresponds more naturally to functional programming, where you think in terms of transformations rather than steps.

Most mainstream programming languages are built on the Turing machine model, with their assignment statements, loops, and mutable data structures. This isn't because the Turing model is inherently superior, but because early computers were physical machines with explicit state, making the Turing approach a more intuitive fit for hardware. As we've moved toward higher-level abstractions and parallel computing, many of lambda calculus's ideas have proven increasingly valuable.[125]

The hubris of McCarthy and the early Lisp pioneers was thinking they could create a practical programming language based on these abstract mathematical principles. Yet they succeeded, and Lisp became not just a usable language but an extraordinarily influential one. The concepts pioneered in Lisp - garbage collection, dynamic typing, first-class functions, and homoiconicity (code as data) - have influenced virtually every modern programming language.[126]

Common Lisp, which emerged in the 1980s as a standardization of various Lisp dialects, took this ambition even further. It aimed to be the ultimate pro-

[125] There is another, third model of computation rearing its head in recent years – interaction nets and interaction combinators. Introduced by Yves Lafont in the early 1990s, interaction nets provide a graph-based model of computation where transformations occur through local interactions between nodes in a network. Interaction combinators distill this model to just three primitive operations (typically called gamma, delta, and epsilon) that together form a complete computational system. What makes this approach particularly compelling is its inherent parallelism and optimal reduction properties - computations naturally share results and can execute concurrently without complex coordination. These theoretical advantages have remained largely academic until recently, when Victor Taelin began implementing them in practical systems. His Higher-Order Virtual Machine (HVM) and the newer HVM2, along with the Bend programming language built on these principles, represent one of the first serious attempts to bring interaction combinators from mathematical theory into practical programming. While still experimental, this approach promises a computing model that might better utilize the massively parallel hardware we're increasingly building, potentially offering significant efficiency gains for certain problems compared to both imperative and traditional functional approaches. Bend is particularly interesting because it looks and feels like a normal functional language (well it looks like Python), despite its radically different underlying execution model.

[126] Lisp actually introduced even more fundamental things, such as the `if` statement, where each branch had a body. Before, all we would have are conditional jumps in the note of `IF CONDITION GO TO SOMEWHERE`

grammable programming language, with a powerful macro system that allowed programmers to extend the language itself. This quote from Kent Pitman captures the philosophical difference:

```
;; C programmers say:
(printf "Hello, world!\n")

;; Pascal programmers say:
(writeln '|Hello, world!|)

;; Lisp programmers say:
(defun hello-world ()
  (format t "Hello, world!~%"))

;; And then they say:
(defmacro greet (name)
  `(format t "Hello, ~A!~%" ,name))
```

In most languages, you adapt your problem to the language. In Lisp, you can adapt the language to your problem.

Both Rust and Lisp represent cases where ambitious hubris led to genuine innovation. But for every such success story, there are countless projects where hubris led to failure - overengineered solutions that collapsed under their own complexity, reinvented wheels that turned out lopsided, or clever hacks that became maintenance nightmares.

Learning to program is a never-ending series of what we might call, borrowing from Plato, cave experiences. You start in darkness, seeing only shadows. You learn a language, a framework, a paradigm, and think "Now I understand programming." Then you step outside and realize you were just seeing shadows on the wall of a much larger cave. This process repeats endlessly.

It's foolish to believe you've reached the end of this journey - that you know all there is to know about programming and have nothing more to learn. Even programmers with decades of experience continue to encounter new ideas, techniques, and paradigms that change how they think about code. You can learn from programmers less experienced than you, who might see problems from fresh perspectives. You can learn from languages and tools you've never used, which might approach problems in ways you never considered.

The Smalltalk community has a saying that captures this: "Simple things should be simple, complex things should be possible." This philosophy led them to

create one of the first object-oriented languages and the first modern IDE with a graphical user interface in the 1970s. Despite Smalltalk's influence on modern programming, many programmers have never used it or studied its ideas directly. This is just one example of how limiting yourself to what you already know can prevent you from discovering valuable insights.

To wrap up, it's perfectly fine - even necessary - to have big ideas and ambitious goals. Don't let imposter syndrome or lack of confidence hold you back from attempting significant projects. But approach your craft with humility and an awareness of your own limitations. Recognize that you will make mistakes, that your first solution might not be optimal, and that there's always more to learn. Balance your ambition with a willingness to question your assumptions and listen to feedback.

The best programmers aren't those who never make mistakes or always choose the perfect solution on the first try. They're those who can recognize their mistakes, adapt their thinking, and continually improve their craft through a combination of ambition and humility.

3.6 Paradigms

So far, we have been mentioning a lot of languages under a lot of programming paradigms. Most of these languages are multiparadigmatic, meaning that they support multiple paradigms. Often, we can put together a fairly extensive list because some paradigms are subsets of others.[127]

The introduction of concretely defined programming paradigms aim to solve both problems of programming in the small and programming in the large. We will talk about them here, although their implications reach into the scope of programming in the large as well.

In this book, we will talk about object-oriented programming, since that is the paradigm that most people are familiar with, about functional programming, since that is the other paradigm most mentioned, and then symbolic programming, which is a transcendental paradigm, that rears its (not ugly, beautiful) head in all languages that have a notion of meta-programming, and is the best description for what Lisp aims for.

The concept of programming paradigms wasn't formally articulated until 1978, when Robert Floyd used the term "paradigms of programming" in his Turing Award lecture. Floyd borrowed the term from Thomas Kuhn's influential book "The Structure of Scientific Revolutions," which described how scientific fields evolve through paradigm shifts. Floyd argued that programming was undergoing similar transformations as new approaches to problem-solving emerged.

Prior to this formal recognition, programmers had already developed distinct approaches to programming, largely influenced by the hardware and theoretical models available to them. The earliest electronic computers were programmed in machine code or assembly language, where the focus was necessarily on manipulating the machine's state through sequences of instructions. This approach naturally led to what we now call imperative programming - giving the computer explicit commands to execute in sequence.

[127] I suppose the actual relationship can be a bit confusing. If we think of paradigms as imposing constraints and making sacrifices, which is an idea I present in a couple chapters, then it would make sense to say that e.g. OOP is more constrained than pure procedural programming. It would then make sense to say that OOP is a subset of procedural. If we however consider OOP as being a paradigm where more named concepts are available than in procedural programming, then we would say that procedural is a subset of OOP. I prefer the first approach.

Each programming paradigm represents a set of voluntary constraints that programmers impose upon themselves. You may ask me: why would limiting our options be beneficial? Well because when we choose and name a set of axioms and approaches to follow, we can design a language that can model them effectively. Many C programmers have prior to C++[128] manually implemented an object system, but it isn't fun. You want a programming language that has one for you.

The most fundamental division in programming paradigms is between imperative and declarative approaches. This split emerged gradually as computing evolved from its origins toward more abstract models. We can blame Fortran and Lisp.

Imperative programming, which includes procedural programming and object-oriented programming, focuses on describing how a program operates step by step. It's characterized by statements that change a program's state, with the programmer explicitly specifying the exact sequence of operations. This approach directly reflects the Von Neumann architecture of most computers, where instructions execute sequentially and operate on memory.

Early languages like FORTRAN (1957), ALGOL (1958), and COBOL (1959) were primarily imperative. They provided abstractions above assembly language but still required thinking in terms of sequential operations and state changes. Here's a simple example in FORTRAN:

```
C Calculate the sum of numbers from 1 to 10
      INTEGER SUM, I
      SUM = 0
      DO 10 I = 1, 10
        SUM = SUM + I
10    CONTINUE
      PRINT *, 'The sum is:', SUM
      END
```

Declarative programming, which emerged almost simultaneously but took longer to gain mainstream adoption, focuses instead on what a program should

[128] And still after it... one instance being the GObject library. GObject provides a portable object system for C, and is the cornerstone of the GNOME project. You interact with programs that use this library all the time (GTK, Pango, gstreamer, and many other tools and libraries spawned by GNOME use it). There was an idea to build a programming language on top of GObject - Vala. It looks somewhere between Java and C# and never reached widespread adoption.

accomplish without specifying the exact steps. The programmer describes the desired result, and the implementation details are handled by the language runtime. This approach creates a higher level of abstraction that can be easier to reason about for certain problems.

A pivotal moment in the evolution of programming paradigms came with the structured programming movement of the late 1960s and early 1970s. Edsger Dijkstra's 1968 letter "Go To Statement Considered Harmful" criticized the unrestricted use of goto statements, arguing they made programs difficult to understand and analyze:

"The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. It is like the match: useful in skilled hands, disastrous in the hands of a child."

This critique sparked a movement toward structured control flow using if-then-else constructs, loops, and subroutines. Languages like ALGOL 68 and later Pascal embraced these principles, demonstrating how deliberately constraining programmer freedom (by discouraging or eliminating goto) could lead to more reliable and maintainable code.

This period also saw the development of several major paradigms that would shape programming for decades to come:

Object-oriented programming emerged from Simula 67 and gained prominence with Smalltalk in the 1970s. While we'll explore OOP in depth in the next chapter, it's worth noting that Alan Kay, one of Smalltalk's creators, originally conceived of objects not as the class hierarchies that later dominated OOP implementations, but as autonomous computational entities communicating through messages. Kay was inspired by biological cells and the ARPANET (the precursor to the Internet), envisioning systems of independent objects coordinating through messages.

The early vision of OOP was transformative - it offered a way to manage complexity by encapsulating state and behavior into discrete units that communicated through well-defined interfaces. This approach proved particularly valuable for modeling real-world systems and building user interfaces. We'll explore the evolution and principles of OOP more thoroughly in its dedicated chapter.

Functional programming traces its roots to Alonzo Church's lambda calculus, a formal system developed in the 1930s as a way to investigate computability. LISP, created by John McCarthy in 1958, was the first programming language to incorporate these ideas, though it wasn't purely functional. More strict functional languages like ML (1973) and later Haskell (1990) emerged to explore the benefits of a more purely functional approach.

Functional programming treats computation as the evaluation of mathematical functions and avoids state and mutable data. This leads to referential transparency - the property that a function's result depends only on its inputs, not on any external state. This constraint makes programs easier to reason about, test, and parallelize. We'll delve deeper into functional programming in its dedicated chapter.

Logic programming represents perhaps the most purely declarative approach to programming. Instead of describing a sequence of operations or defining functions, logic programming involves stating facts and rules, then making queries about what can be deduced from them. The language implementation handles the details of how to search for solutions.

Prolog, created in 1972 by Alain Colmerauer and Philippe Roussel, is the most widely known logic programming language. Let's examine our earlier Prolog example in more detail:

```
% Facts about family relationships
parent(john, bob).    % John is a parent of Bob
parent(john, lisa).   % John is a parent of Lisa
parent(mary, bob).    % Mary is a parent of Bob
parent(mary, lisa).   % Mary is a parent of Lisa
parent(bob, ann).     % Bob is a parent of Ann
parent(bob, jim).     % Bob is a parent of Jim

% Rules defining other relationships
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

In this code, we first define a set of facts using the `parent` predicate. Each fact states a relationship - for example, `parent(john, bob)` asserts that "John is a parent of Bob." These facts form our knowledge base.

Next, we define rules that allow us to derive new information. The rule `sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.` should be read as "X is

a sibling of Y if there exists some Z who is a parent of X and also a parent of Y, and X is not the same as Y." The variables X, Y, and Z are automatically bound to values that satisfy all the conditions.

Similarly, `grandparent(X, Z) :- parent(X, Y), parent(Y, Z).` defines the grandparent relationship: "X is a grandparent of Z if X is a parent of some Y and Y is a parent of Z."

When we make a query like `sibling(ann, jim).`, Prolog tries to determine whether this statement can be proven true given our facts and rules. Here's how:

1. It looks for a rule for `sibling` and finds `sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.`
2. It substitutes `ann` for X and `jim` for Y, giving us `sibling(ann, jim) :- parent(Z, ann), parent(Z, jim), ann \= jim.`
3. It then tries to satisfy each condition:
   - `parent(Z, ann)`: It looks for facts matching this pattern and finds `parent(bob, ann)`, so Z = bob.
   - `parent(bob, jim)`: It checks if this fact exists, and indeed finds `parent(bob, jim).`
   - `ann \= jim`: It verifies that ann and jim are different individuals.
4. Since all conditions are satisfied, Prolog confirms that `sibling(ann, jim)` is true.

When we make an open query like `grandparent(X, ann).`, Prolog finds all possible values of X that make the statement true:

1. It looks for a rule for `grandparent` and finds `grandparent(X, Z) :- parent(X, Y), parent(Y, Z).`
2. It substitutes `ann` for Z, giving us `grandparent(X, ann) :- parent(X, Y), parent(Y, ann).`
3. It then tries to find values for X and Y that satisfy both conditions:
   - `parent(Y, ann)`: It finds `parent(bob, ann)`, so Y = bob.
   - `parent(X, bob)`: It finds `parent(john, bob)` and `parent(mary, bob)`, so X could be either john or mary.
4. Prolog returns both solutions: X = john and X = mary.

This paradigm is particularly powerful for problems involving complex relationships and constraints. The programmer specifies what properties a solution should have, not how to compute it. Prolog's execution model uses unification

(pattern matching) and backtracking (systematically trying different possibilities) to find solutions.

Logic programming excels in areas like expert systems, constraint satisfaction problems, natural language processing, and certain types of artificial intelligence. It allows complex logical relationships to be expressed concisely and problems to be solved declaratively.

The constraint of having to express problems in terms of logical relations forces a different kind of thinking than imperative or functional programming. It encourages breaking problems down into facts and rules, which can often lead to elegant solutions for certain classes of problems.

Stack-based programming represents a radically different approach to program structure. In this paradigm, operations primarily manipulate values on a stack rather than in named variables. Forth, created by Charles Moore in the early 1970s, is the canonical stack-based language.

Let's examine our earlier Forth example in more detail:

```
: SQUARE ( n -- n^2 )  DUP * ;
: SUM-OF-SQUARES ( a b -- c )  SQUARE SWAP SQUARE + ;

5 7 SUM-OF-SQUARES .  \ Prints 74
```

In Forth, programs are built from "words" (similar to functions in other languages) that manipulate a data stack. Reading this example:

- `: SQUARE ( n -- n^2 ) DUP * ;` defines a word called SQUARE. The comment `( n -- n^2 )` is a stack effect diagram showing that SQUARE takes one value from the stack and replaces it with its square. The implementation uses `DUP` to duplicate the top value on the stack, and `*` to multiply the top two values, replacing them with their product.

- `: SUM-OF-SQUARES ( a b -- c ) SQUARE SWAP SQUARE + ;` defines a word that computes the sum of squares of two numbers. The stack effect diagram shows it takes two values and returns one. The implementation:
  - `SQUARE` squares the top value (b)
  - `SWAP` exchanges the top two stack values, bringing a to the top
  - `SQUARE` squares a
  - `+` adds the two squared values

- `5 7 SUM-OF-SQUARES .` puts 5 and 7 on the stack, calls SUM-OF-SQUARES, and then prints the result using `.` (which removes and displays the top stack value).

Here's the step-by-step execution:

1. `5` pushes 5 onto the stack. Stack: [5]
2. `7` pushes 7 onto the stack. Stack: [5 7]
3. `SUM-OF-SQUARES` executes:
   - `SQUARE` executes on 7:
     - `DUP` duplicates 7. Stack: [5 7 7]
     - `*` multiplies the top two values. Stack: [5 49]
   - `SWAP` exchanges 5 and 49. Stack: [49 5]
   - `SQUARE` executes on 5:
     - `DUP` duplicates 5. Stack: [49 5 5]
     - `*` multiplies the top two values. Stack: [49 25]
   - `+` adds 49 and 25. Stack: [74]
4. `.` prints and removes the top value, 74.

Forth's approach is postfix (also called Reverse Polish Notation), where operators follow their operands. This eliminates the need for parentheses and operator precedence rules, resulting in a simple syntax that's easy to parse and compile.

The philosophy behind Forth emphasizes extreme simplicity, composability, and efficiency. A complete Forth system (compiler, interpreter, and core library) can be implemented in just a few kilobytes. This minimalism was crucial in the resource-constrained environment where Forth was developed - Moore initially implemented it on a minicomputer with just 8KB of memory.

The constraint of working primarily with a stack forces programmers to break problems down into small, composable operations. This often leads to solutions that are remarkably concise and efficient. However, it also requires a different mental model than variable-based programming, as you must keep track of what's on the stack at each point in your program.

Stack-based programming has found niches in embedded systems, where its small footprint is valuable, and in certain specialized domains like graphics and virtual machines. The PostScript language, used for describing printed page layouts, is stack-based, as is the bytecode interpreter for the Java Virtual Machine.

Array-based programming offers yet another perspective, treating arrays (or more generally, collections of data) as primary objects of manipulation rather than individual elements. This approach enables concise, powerful operations on entire data sets without explicit loops or iteration.

APL (A Programming Language), created by Kenneth Iverson in the 1960s, is the quintessential array-oriented language. It's famous for its concise notation using a special set of symbols, each representing a powerful operation on arrays. Let's examine our earlier APL examples in more detail:

```
⍝ Generate a 10×10 multiplication table
table ← (⍳10) ∘.× ⍳10

⍝ Find all prime numbers up to 100
sieve ← {(⍳⍵) ~((⍳⍵) ∘.| ⍳⍵)/⍨(⍳⍵) ≠ ⊤⍨(⍳⍵)}
sieve 100
```

In the first example:

- `⍳10` generates an array containing integers from 1 to 10
- `∘.×` is the "outer product" operator with multiplication
- `(⍳10) ∘.× ⍳10` computes the outer product of the vector [1,2,...,10] with itself using multiplication, resulting in a 10×10 matrix where each element is the product of the corresponding row and column indices.

This single line creates an entire multiplication table that would require nested loops in most other languages.

The prime number sieve is more complex but showcases APL's power. Breaking it down:

- `⍳⍵` generates integers from 1 to the input value (⍵)
- `(⍳⍵) ∘.| ⍳⍵` creates a table of remainders when each number is divided by each number
- `(⍳⍵) ≠ ⊤⍨(⍳⍵)` identifies numbers that aren't equal to themselves modulo each number
- `/⍨` filters the original array based on this condition
- `~` removes certain elements, helping identify the primes
- The entire expression efficiently implements a prime number sieve in a single, dense line of code

Modern array-oriented languages like BQN and Uiua continue this tradition while attempting to make the notation more accessible. For instance, here's a BQN implementation of the multiplication table:

```
mult_table ← (↕10) +˝ × (↕10)
```

Array-oriented programming excels at numerical and data processing tasks. Its focus on whole-array operations aligns well with how modern hardware works, where operations on contiguous blocks of memory can be highly optimized. This paradigm has influenced features in many mainstream languages - NumPy in Python, LINQ in C#, and array programming facilities in Julia and R all draw inspiration from the array-oriented approach.

The constraint of thinking in terms of operations on entire arrays rather than individual elements forces a higher level of abstraction. This can lead to solutions that are not only more concise but also more efficient, as they can take advantage of vectorized operations and parallelism.

We can therefore say that different paradigms represent different ways of modeling computation:

- Imperative programming models computation as a sequence of steps that modify state
- Functional programming models computation as the composition and evaluation of mathematical functions
- Logic programming models computation as deduction from facts and rules
- Stack-based programming models computation as operations on a stack
- Array-based programming models computation as operations on entire collections of data

Each paradigm makes certain types of problems easier to solve and makes certain types of errors harder to introduce. For example:

- Functional programming's emphasis on immutability eliminates entire categories of bugs related to shared mutable state
- Logic programming's declarative nature allows complex relationships to be expressed more concisely
- Object-oriented programming's encapsulation helps manage complexity in large systems

- Array-based programming's focus on whole-collection operations can eliminate off-by-one errors common in explicit loops

These paradigms have influenced each other over time. For instance, many imperative languages now incorporate functional features, and many functional languages have adopted type systems influenced by object-oriented concepts. The boundaries between paradigms are increasingly blurred as languages adopt successful ideas from multiple approaches.

The analogy to natural language families is apt but imperfect. Just as Romance languages share vocabulary and grammar derived from Latin, programming language families often share syntax, semantics, or conceptual models. However, programming languages evolve both through inheritance and through conscious design decisions, creating more complex relationships than natural language families.

For example, C++ directly descends from C, inheriting much of its syntax and semantics while adding object-oriented and generic programming features. Python, though often grouped with C-like languages due to some syntactic similarities, has a very different underlying model influenced by ABC, Modula-3, and various functional languages.

The Lisp family (Common Lisp, Scheme, Clojure) shares the distinctive S-expression syntax and symbolic processing capabilities, though they differ significantly in their approach to evaluation, typing, and standard libraries. The ML family (Standard ML, OCaml, F#) shares a heritage of strong static typing and pattern matching, though they vary in their purity and platform integration.

An interesting recent example is the Bend language, which superficially resembles Python syntactically but internally represents programs as interaction nets based on a graph reduction model. This creates a situation where the language looks familiar to Python programmers but operates on fundamentally different principles, demonstrating how surface-level similarities can mask deep conceptual differences.

Most modern mainstream languages are multiparadigmatic, incorporating elements from several approaches. Java, though primarily object-oriented, has added functional features like lambdas and streams. Python supports procedural, object-oriented, and functional styles. Rust combines elements of imperative, functional, and generic programming with its unique ownership system.

This multiparadigmatic nature reflects a practical recognition that different problems are best solved with different approaches. A graphical user interface might be naturally expressed using objects, while a data transformation pipeline might be clearer in a functional style. The ability to select the right paradigm for each component of a system is a mark of experienced programmers.

Each paradigm offers a distinct perspective on what programs fundamentally are. To an imperative programmer, a program is a sequence of operations. To an object-oriented programmer, it's a network of communicating entities. To a functional programmer, it's a composition of mathematical functions. To a logic programmer, it's a set of facts and rules for deduction.

Understanding these different perspectives, even ones you don't regularly use, provides valuable mental models and problem-solving approaches. A Java programmer might never write production Prolog code, but understanding logical reasoning can influence how they model complex business rules. A Python programmer might never write APL, but exposure to array-oriented thinking could improve how they use NumPy.

In the following chapters, we'll explore object-oriented, functional, and symbolic programming in greater depth. Each offers valuable tools for creating elegant, maintainable code, and understanding when and how to apply each paradigm is an essential skill for modern programmers.

3.8 Functional Programming

3.10 Optimizations en route to hell

3.11 Design patterns

Chapter Four

Programming in the large

4.1 Preparation and agility

4.2 A goodly home for programs

4.3 Stratification

4.4 Separation of Concern

4.5 Locality of Behavior

4.7 Technical Debt - Now or Never

4.8 Code reviews

Chapter Five

Conclusion

CONCLUSION

5.1 No silver bullet

5.2 Aesthetics are an acquired skill, and an acquired taste