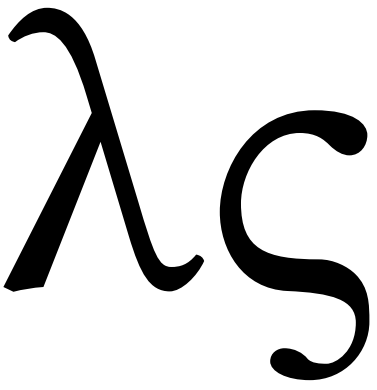

Lambdas and Logos

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

Contents

1 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

[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".

my parents should not feed my addiction. And it is true I was mesmerized by 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

[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.

activities, and I love exploring it. Over the years, I have experimented and used 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.

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

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.

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]

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.

[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)

[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?)

2 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.

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

[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.

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

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 fundamental 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.

[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.

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

[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

[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

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.

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

[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.

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.

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,

[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

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 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]

---

[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.

[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.

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.

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.

[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

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

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:

[51] And ideally, share it!

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

[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).

> 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 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 in-

[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.

creasingly 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 software 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.

---

[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"?

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, 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

[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]

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 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.

[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.

And I couldn't have said it better myself.

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)
```

[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 `->`.

```
                    `(setf (gethash ',(car rule) rule-table)
                           (lambda ,(cadr rule)
                             ,@(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).

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

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 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:

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

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

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]

---

[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.

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

[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.

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.

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.

3 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**

[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.

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 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.

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

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

```c
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

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

[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.

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:

```
#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;
```

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

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

[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

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 programmers 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

---

his Eiffel programming language. Eiffel is mostly just a research language, but very interesting to check out!

[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.

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 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]

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

[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.

[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.

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.

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" &&
```

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

```
    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)) {
        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

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

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;
→   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 ...)))
```

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

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 entire source files. A source file, much like a chapter in a book, should have a clear purpose and structure.

Long source files can be problematic. When a file grows beyond a few hundred lines, 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 generally get uncomfortable when files approach 1,000 lines. 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. 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.

Let's start at the top of a file. After any language-required headers (like the shebang line in shell scripts or the package declaration in Java), imports or includes typically come first. These statements 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
import pandas as pd
import requests

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

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, prefer explicit imports like `import numpy as np`. This makes it clear where each function or class is coming from:

```
# Poor practice - wildcard import
from numpy import *
result = sqrt(array([1, 2, 3]))  # Where do sqrt and array come from?

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

There are exceptions to this rule. Some libraries are 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 customary to place constants and global variables. These definitions establish the foundational values that the rest of the file will work with:

```
#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), 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:

```
// 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
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):

```
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
    }

    // 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.

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:

```
;; 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))))
```

```lisp
;; 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. This feature, available in most modern editors, allows you to collapse sections of code to focus on what's relevant. Some languages provide explicit support for this with region markers:

```
// 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.

Speaking of Forth, an interesting stack-based language from the 1970s, it took a unique approach to file organization. Forth's philosophy was extreme simplicity and directness, with a focus on small, composable "words" (functions). A well-written Forth file resembles a progressive revealing of a vocabulary, with each new word building on previously defined ones in an explicit bottom-up fashion. This approach naturally leads to good file organization.

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.

Prolog, a logic programming language, illustrates the importance of file organization in a unique way. 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 minutiae, but they compound to significantly affect readability and maintainability. A well-organized source file reduces the cognitive load on readers, allowing them to find what they need quickly and understand the code's structure without unnecessary effort. Just as a well-organized book with clear chapter divisions and a logical progression helps readers navigate complex topics, thoughtfully structured source files help programmers navigate complex codebases.

4 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

4.6 The Expression Problem

4.7 Technical Debt - Now or Never

4.8 Code reviews

5 Conclusion

5.1 No silver bullet

5.2 Aesthetics are an acquired skill, and an acquired taste