λς

Lambdas and Logos

On Writing Elegant Code

Lukáš Hozda

Contents

1 Preface

This book is for the junior or intermediate programmer, and to 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.

Lukáš Hozda,
rennaissance man

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.

```pascal
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:

```c
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

```c
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[6] is partitioning:

[6] 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:[7][8]

```
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) {
```

[7] 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

[8] 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.[9] 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[10]
- 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.

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

[10] 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[11], 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[12], and take the power of creating generalized abstractions away from them.

This leads us to a very important point: **Programming languages matter.**[13]

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).

[11] 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.

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

[13] From a certain point

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[14], 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

[14] 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[15]. 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]
```

[15] 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, 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).

```
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[16] or make no effort to limit side-effects[17]. 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));

    result
}
```

[16] 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

[17] 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

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[18]:

```
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 its fulfills it goals effectively (or at all), if it is well implemented,[19] and if it is internally consistent.[20]. These are fairly difficult requirements, and generally, one can point out flaws in the design of any programming language.

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

[18] 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.

[19] 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 things 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.

[20] 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.

`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[21] 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:[22]

```
((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))
```

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

[22] 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.

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.[23], 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)`[24] 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
        (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))
```

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

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

```
(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![25]

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 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:

[25] 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.

```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 (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.[26]

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 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.

[26] 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.

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.[27]

[27] 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,*

Meanwhile, “logos”[28] comes to us from ancient Greek philosophy, where it represented discourse, reason, and the underlying principles that govern reality.[29] 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

The true art of programming lies not in mastering any single language or paradigm[30], 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

---

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 …”

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

[29] And about a fifty other different things.

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

- 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.[31] 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.

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.[32]

[31]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.

[32]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.

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".[33]

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.[34]

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

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

[34] 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

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.[35] 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)[36], 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 communication medium, not just to the computer but to other programmers (including our future selves)[37]. As Robert C. Martin puts it, "Clean code always looks like it was written by someone who cares."[38]

[35] 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.

[36] 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.

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

[38] 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.

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[39]

Meanwhile, a **researcher**[40] 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, and that is that free-time, open-source, and commercial programming are all different disciplines[41]

---

[39] 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.

[40] *Computer scientist* also feels appropriate

[41] 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.

It is also worth noting that many people who aren't programmers write code. Scientists use scripting languages to analyze data.[42] 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,[43] 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.

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

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

[43] 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.

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.

Remember that programming is inherently creative. We 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 so mutable and alive, with nothing more than a computer and your mind? 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. Rather, it's about recognizing the full scope of what programming entails and aspiring to practice it in its complete form. 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. Be someone who thinks clearly about problems, who designs elegant solutions, who communicates effectively through code, and who finds joy in the creative process of programming. The code you produce will be better for it, and so will your experience of creating it.

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.[44] 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)[45] with a quote from Alan Perlis that sets the tone for their approach to computing:

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

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

[44]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.

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

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

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

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

First, the proliferation of massive, complex frameworks that nobody fully understands. Modern software is built on towering stacks of abstractions - operating systems, libraries, frameworks, middleware, virtual machines, and more. 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.[46]

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

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.

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

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

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.

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

This view doesn't mean Sussman encourages sloppy programming. 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.

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. 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.

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.

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. 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.

2.4 It's not just the code

2.5 Elegant code and the cost of inelegant code

3 Programming in the small

3.1 Line lengths and whitespace

3.2 Source code files

3.3 Naming things

3.4 Documenting code

3.5 Taming your hubris

3.6 Object-Oriented Programming

3.7 Functional Programming

3.8 Symbolic Programming

3.9 Optimizations en route to hell

3.10 Design patterns

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