The role of Aesthetics in the Understandings of Source code

Pierre Depaz
under the direction of Alexandre Gefen (Paris-3)
and Nick Montfort (MIT)

ED120 - THALIM

last updated - 03.11.2021

Chapter 1

Introduction

This thesis is an inquiry into the formal manifestations of source code, on how particular configurations of lines of code allow for aesthetic judgments and on the purposes that such configurations serve. The implications of this inquiry will lead us to consider the different ways in which people read and the different ways in which source code can be represented, depending on what it aims at accomplishing, and on the contexts in which it operates. This study on source code involves the different groups of people which read and write it, the purposes for which they write it, the programming languages they use to write it, and the natural language they use to speak about it. Most importantly, this thesis focuses on source code as a material and linguistic manifestation of a larger digital ecosystem of software and hardware to which it belongs. Since source code is only one component of code, as we will see below, this thesis also aims at studying the reality of written code, rather than its conceptual interpretations.

Starting from pieces of source code, this thesis will aim at assessing what programmers have to say about it, and attempt to identify how one or more specific *aesthetic fields* are used to refer to it. This aim depends on two facts: first, source code is a medium for expression, both to express the programmer's intent to the computer[**?**] and the programmer's intent to

another programmer[**?**]. Second, source code is a relatively new medium, compared to, say, paint or mechanics. As such, the development and solidification of aesthetic practices—that is, of ways of doing which do not find their immediate justification in a practical accomplishment—is an ongoing research project in computer science, software development and the digital humanities. Formal judgments of source code are therefore existing and well-documented, and are related to a need for expressiveness, as we will see in chapter 2, but their formalization is still an ongoing process.

Source code thus can be written in a way makes it subject to aesthetic judgments by programmers; that is, code *has* aesthetics, but it is unclear exactly *which* aesthetics. Indeed, these aesthetic judgments as they exist today rely on different domains to assess source code, as a means to grasp the cognitive object that is software. These draw from metaphors ranging from literature, architecture, mathematics and engineering. And yet, source code, while related to all of these, isn't exactly any of the them. Like the story of the seven blind men and the elephant[**?**], each of these domains touch on some specific aspect of the nature of code, but none of them are sufficient to entirely provide a solid basis for the aesthetic judgments of source code. It is at the crossroads of these domains that this thesis locates its work.

The examination of source code, and of the discourses around source code will integrate both the diversity of ways in which source code can exist, and the invariant aspects which underline all diverse approaches of source code. Particularly, we will see how each groups of practicionners tend to deploy references to one particular set of metaphorical references drawing from the domains above, but also how these references overlap across groups. The point of overlap, as we will demontrate, is that of *using a formal linguistic system to communicate the understanding of complex cognitive structures*. Through an interdisciplinary approach, we attempt to connect this formal symbol system to the broader role of aesthetics as a cognitive mechanism to deal with complexity.

The rest of this introduction will consist in establishing a more complete view of the context in which this research takes place, from computer science to digital humanities and science and technology studies. With this context at hand, we will proceed to highlight the specific problems which will be tackled regarding the current place of aesthetics in source code. After outlining our methodology and the theoretical frameworks which will be mobilized throughout this study, we will sketch out how the different chapters of this thesis will attempt at providing some responses to our research questions.

1.1 Context

1.1.1 The research territory: code

Most of our modern infrastructure depends, to a more or less dramatic extent, on software systems[**?**], from commercial spaces to classrooms, transport systems to cultural institutions, scientific production and entertainment products. Software regulates and automates the information stores, exchanges and creation which compose each of these domains of human activities. The complex processes are described in what is called source code, a vast and invisible set of texts. The number of lines of code involved in running these processes is hard to estimate; one can only rely on disclosures from companies, and publicly available repositories. To give an order of magnitude, all of Google's services amounted to over two billions source lines of code (SLOC)[**?**], while the 2005 release of the OSX operating system comprised 86 millions lines of code, and while the version 1.0 of the Linux kernel (an operating system which powers most of the internet and specialized computation) totalled over 175,000 SLOC, version 4.1 jumped to over 19.5 million lines of code in the span of twenty years[**?**].

Given such a large quantity of textual mass, one might wonder: who reads this code? To answer this question, we must start diving a bit deeper

into what source code really is.

Source code consists in a series of instructions, composed in a particular programming language, which is then processed by a computer in order to be executed. For instance, using the language called Python, the source code:

```
a = 4
b = 6

def compute(first, second):
    return (first * 2) + second

compute(a, b)
```

consists in telling the computer to store two numbers in what are called *variables*, then proceeds with describing the *procedure* for adding the double of the first terms to the second term, and concludes in actually executing the above procedure. Given this particular piece of source code, the computer will output the number `14` as the result of the operation `(4 * 2) + 6`. In this sense, then, source code is the requirement for software to exist: since computers are procedural machines, acting upon themselves and upon the world, they need a specification of what to do, and source code provides such a specification. In this sense, computers are the main "readership" of source code.

However, it is also a by-product of software, since it isn't no longer required once the computer has processed and stored it into a *binary* representation, a series of 0s and 1s which represent the successive states that the computer has to go through in order to perform the action that was described in the source code. *Binary code* is what most of the individuals who interact with computers deal with, in the form of packaged applications, such as a media player or a web browser. They (almost) never have to inquire about, or read, such source code. In this sense, then, source

code only matters until it gets processed by a computer, through which it realizes its intended function.

From another perspective, source code isn't just about telling computers what to do, but also a key component of a particular economy: that of software development. Software developers are the ones who write the source code and this process is first and foremost a collaborative endeavour. Software developers write code in successive steps, because they add features over time, or they fix errors that have shown up in their software, or they decide to rewrite parts of the source code based on new ideas, skills or preferences. In this case, source code is not used to communicate to the computer what it does, but it is used to communicated to other software developers what the *intent* of the software is. Source code is then the locus of human, collaborative work; it represents iterations of ideas, formalization of processes and approaches to problem-solving. As Harold Abelson puts it,

> "Programs must be written for people to read, and only incidentally for machines to execute."[**?**].

Official definitions of source code straddle this line between the first role of source code (as instructions to a computer) and the second role of source code (as indications to a programmer). For instance, a definition within the context of the Institute of Electrical and Electronics Engineering (IEEE) considers source code *any fully executable description of a software system, which therefore includes various representations of this description, from machine code to high-level languages and graphical representations using visual programming languages*[**?**]. This definition focuses on the ability of code to be processed by a machine, and mentions little about its readability (i.e. processability by other humans).

On the other hand, the definition of source code provided by the Linux Information Project[1] focuses on source code as *the version of software as*

[1] https://linfo.org/sourcecode.html

it is originally written (i.e. typed into a computer), by a human in plain text (i.e. human-readable, alphanumeric characters).[**?**]. The emphasis here is on source code as the support of human activity, as software developers need to understand the pieces of code that they are creating, or modifying. Source code thus has two kinds of readabilities: a computer one, which is geared towards the correct execution of the program, and a human one, which is geared towards the correct understanding of the program. In the lineage of this human-readability, we can point to the Free Software Foundation's equation of the free circulation and publication of source code with the free circulation of publication of ideas. Particularly, Freedom 1 (*The freedom to study how the program works, and adapt it to your needs*) and Freedom 2 (*The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.*) as stated in the FSF's definition of Free Software stipulates that access to source code is required to support these freedoms, a version of source code that is *not concealed*, i.e. readable by both human and machine[**?**].

In addition to this ability to communicate the ideas latent in it, source code, as an always potentially collaborative object, can be the locus of multiple subjectivities coming together. As Krysa and Sedek state in their definition, *source code is where change and influence can happen*, and where *intentionality and style are expressed*[**?**]. In their understanding, source code shares some features with natural languages as an intersubjective process[**?**], and as such is different from the machine language representation of a program, an object which they do not consider source code due to its unilaterality. The intelligibility of source code, they continue, facilitates its circulation and duplication among programmers. It is this aspect of a socio-technical object that we consider as important as its procedural effectiveness.

In this research, we build on these definitions to propose the following:

Source code is defined as one or more text files which are writ-

> ten by a human or by a machine in such a way that they elicit a meaningful response from a digital compiler or interpreter, and describe a software system. These text files are the starting point to produce an execution of the system described, whether the very first starting point, or an intermediate representation used for subsequent compilations. These files are collectively called *program texts*.

This definition takes into account a broad view of source code, including steps such as intermediate representations (transitory representations from one version of the source to another one), but also obfuscations (deliberately complicating the code to prevent human-readability while maintaining machine-readability) and minifications (reducing the amount of characters used in source code to its minimum). This will allow us to compare human-authorship of source code, machine-authorship, and hybrid modes, in which a human writes unreadable code with the help of tools. One aspect that is being more narrowly defined for the purpose of this study is the actual manifestation of code: while multiple media for source code exist, we exclude here all of those that are not written in the UTF-8 character set—i.e. textual representations. Since one of the questions of this study is to examine the literariness of source code aesthetics, other forms of source code, such as visual programming languages or biological computation, stand outside the scope of this study and should be investigated in subsequent work.

As for the term *program text*, it is chosen in order to highlight the dual nature of source code: that of a computational artefact to be formally processed and unambiguously understood[**?**], and that of a open-ended, multi-layered document, in the vein of Barthes' distinction between a text and a work[**?**].

1.1.2 Beautiful code

Under this definition of source code textually represented, we now turn to the existence of the aesthetics of such *program texts*. To contextualize this existence, we first need to touch upon the history and practice of software development. As an economic activity, software development came from a bottom-up dynamic, a *de facto* activity which was not expected in the early days of computing, where most of the work was divided between mathematics and engineering. Its earliest manifestation can be found in the physical rewiring process of mainframes in order to perform a specific computation, something more akin to firmware than to software. These rewiring tasks were done by mostly female assistants, under the direction of mostly male mathematicians[**?**], and consisted in a simple translation task which did not require any particular attention, or any particular skill. The recognition of software engineering as its own field came as its unique domain of expertise was required in larger engineering projects—for instance, the term *software engineering* was coined in the late 1960s by Margaret Hamilton and her team as they were working on the Apollo 11 Lunar Module software[**?**]. In the same decade, the first volume of *The Art of Computer Programming*, by Donald Knuth, addresses directly both the existence of programming as an activity separate from both mathematics and engineering, as well as an activity with an "artistic" dimension[**?**]. The first volume opens on the following paragraph:

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. This book is the first volume of a multi-volume set of books that has been designed to train the reader in the various skills that go into a programmer's craft.[**?**]

Considered one of the most canonical textbooks in the field, *The Art of Computer Programming* highlights two important aspects of programming for our purpose: that it can be an aesthetic experience and that it is the result of a craft, rather than of a highly-formalized systematic process, as we will see in chapter 1. Some of the aeshetic references related to source code are related to its writing and reading being a craft-like activity[**?**].

Craftsmanship is an essentially fleeting phenomenon, a practice rather than a theory, in the vein of Michel De Certeau's *tactics*, bottom-up actions informally designed and implemented by the users of a situation, product or technology as opposed to *strategies*[**?**], in which ways of doing are deliberately prescribed in a top-down fashion. Craft is hard to formalize, and the development of expertise in the field happens through practice as much as through formal education[**?**]. It is also one in which function and beauty exist in an intricate, embodied and implicit relationship, based on subjective qualitative standards rather than strictly external measurements, with the former rarely being explicitly stated[**?**].

Approaching programming (the activity of writing and reading code) as a craft[**?**] connects to the multiple testimonies of encountering beautiful code, some of which have made their ways into edited volumes or monographs[**?**, **?**, **?**]. Additionally, informal exchanges among programmers on forums, mailing lists, blog posts and code repositories[2] often mention beautiful code, either as a central discussion point or simply in passing. These testimonies constitute the first part of our corpus, as sources in which programmers comment on the aesthetic dimension of their practice. The second part of the corpus is composed of selected program texts, which we will examine in order to identify and formalize which aspects of the textual manifestation of software can elicit an aesthetic experience.

So the existence of something akin to art, something beautiful and pleasurable emerging from the reading and writing of source code has been

[2] See Annex for the list of collected corpus

acknowledged since the 1960s, in the early days of programming as a self-contained discipline. However, the formalization of an aesthetics of source code first requires a working definition of the concept of *aesthetics* as used in this study.

There is a long history of aesthetic philosphical inquiries in the Western tradition, from beauty as the imitation of nature[3], moral purification[4], cognitive perfection[5], sensible representations with emotional repercussions[6]. The common point of these definitions is that of *sensual manifestation*, that is the set of visible forms which can enable an *aesthetic experience*, a cognitive state of pleasure relying on, amongst others, an object, a sense of unity and of discovery[**?**].

The definition of aesthetics that we will use in this thesis starts from this requirement of sense perception, and then builds upon it using two theoretical frameworks: Nelson Goodman's theory of symbols[**?**] and Gérard Genette's distinction between fiction and diction[**?**]. The former provides us with an analysis of formal systems in aesthetic manifestations and their role in a cognitive process, while the second offers a broadened perspective on what qualifies as textual arts, or literature.

Goodman's view on aesthetics is an essentially communicative one: we use aesthetics to carry across more or less complex concepts. This communication process happens through various symbol systems (e.g. pictural systems, linguistic systems, musical systems, choreographic systems), the nature and organization of which can elicit an aesthetic experience. His conception of such an aesthetic experience isn't one of self-referential composition, or of purely emotional pleasure, but a cognitive one, one which belongs to the field of epistemology[**?**]. The symbol systems involved in the aesthetic judgment bear different kinds of relations to the worlds they refer to—such as denoting, representing, ressembling,

[3] Plato, Republic

[4] Aristotle, Poetics; Kant, Critique of the Power of Judgment

[5] Leibniz, Ars Combinatoria

[6] Baumgarten, Aesthetics

exemplifying— and their purpose is to communicate a truth about these worlds[**?**]. In Goodman's view, the arts and the sciences are, in the end, two sides of the same coin. They aim at providing conceptual clarity through formal, systematic means, and the arts can and should be approached with the same rigor as the sciences. In our case, programming, with its self-proclaimed craft-like status and its mathematical roots, stands equally across the arts and sciences.

His use of the term *languages* implies a broader set of linguistic systems than that of strictly verbal ones. This approach will support our initial conception of programming languages as verbal systems, but will allow us not to remain constrained by traditional literary aesthetics such as verse, rhyme or alliteration. To what extent is programming a linguistic activity is going to be one of the main inquiries of this thesis, and Goodman's extended, yet rigorous definition leaves us room to explore the semantic and syntactic dimension of source code as one of those languages of art. Programming languages as symbol systems will be explored further in Chapter 5.

With this analytical framework allowing us to analyze the matter at hand—program texts composed by a symbol system with an epistemic purpose—we turn to a more literary perspective on aesthetics. Genette's approach to literature, *the art of language*, results in the establishment of two dichotomies: fiction/diction, and constitutivity/conditionality. In his eponymous work[**?**], he extends previous conceptions of literature and poetics, from Aristotle to Jakobson, in order to broaden the scope of what can be considered literature, by questioning the conditions under which a text is given a literary status. As such, he establishes the existence of conditional literature alongside constitutive literature: the former gains its status of a literary text from the individual, subjective aesthetic judgment bestowed upon it, while the latter relies on pre-exisiting structures, themes and genres. This approach paves the way for an extending of the domain of literature[**?**], and a more subtle understanding of the aesthetic manifes-

tation in an array of textual works.

Genette also makes the distinction between fiction, with the focus being the potentiality of a text's object, its imaginative qualities and themes, and diction, with an emphasis on the formal characteristics of the text. Since code holds two existences, one as executed, and one as written, I propose to map Genette's concept of fiction on to source code when the latter is considered as a purely functional text—i.e. what the source code ultimately does in its domain of application. Because source code always holds software as a potential within its markings, its diction, waiting to be actualized through execution, one has to imagine what this code actually does. Written source code, then, could either be judged primarily on its fiction or on its diction. Since we focus on the written form of source code, and not on the type of its purpose, an attention to diction will be the entry point of this thesis.

A first approach to source code could be *constitutive*, in Genette's terms: a given program text could be considered aesthetically pleasing because the software it generates abides by some definitions of being aesthetically pleasing[7], or because the software itself is considered a piece of art, shown in exhibitions and sold in galleries. However, our empirical approach to source code aesthetics, by examining various program texts directly, and our inquiry into the possibility of multiple aesthetic fields co-existing within source code as a symbol system, asks us to forgo this constitutive definition of an aesthetic work as normative categories within software development. Our focus on sense perception within aesthetics starts from a conditional approach, in which programmers emit an aesthetic judgment on a program text, with an emphasis first on what the source code *is*, and only secondly on what it *does*[8].

Diction, then, focuses on the formal characteristics of the text. The

[7] For instance, Venustas, Firmitas, Utilitas; See Fishwisck, P. (éd), *Aesthetic Computing*

[8] As we've seen with Goodman, there is nonetheless a tight connection between those to states.

point here is not to assume an autotelic mode of existence for source code, but rather to acknowledge that there is a certain difference between the content of software and the form of its source—good software can be written poorly, and poor software can be written beautifully. This thesis chooses to focus on the formal aspects of code such as not to restrict ourselves to any specific kind, or genre, of program texts, leaving open the possibility for these categories to emerge after our analysis.

So, following Genette's re-asking of the Goodman's question of *When is art?* rather than the historical *What is art?*, we can now proceed with our understanding of aesthetics as a set of physical manifestations which can be grasped by the senses, akin to "the movement of a light, the brush a fabric, the splash of a color"[**?**], which aim at enabling a cognitive, communicative purpose, and which are not exclusively constituted by pre-existing categories.

This overview of the theoretical frameworks of this thesis already implicitly denotes the boundaries of this study. The domain we are investigating here is one that is delimited by both medium and purpose. First, the medium limitations is that of text, in its material sense, as mentioned above in our definition of source code. Second, the purpose limitation is that of computable code, rather than computed code: we are examining latent programs, with their reality as texts and their virtuality as actions, rather than the other way around. Executed software and its set of affordances (e.g. graphical user interfaces[**?**], real-time interactivity[**?**] and process-intensive developments[**?**]) differ from the literary and architectural ones that software, in its written form, is claimed to exhibit. However, executable and executed software, being to sides of the same coin, might suggest causal relationships—e.g. the aesthetics of source code affecting the aesthetics of software—and such an inquiry would be best reserved for a subsequent study.

Now that we've explicited our object of study—the formal manifestations of software under its textual form—we can turn to a review of the re-

search that has alredy been done on the subject, before highlighting some of the limitations. These relations between source code and aesthetics have been addressed by academic studies through different, separate dynamics.

1.1.3 Literature review

A literature review on this topic must address the dualistic nature of studies on source code, as research can be distinguished between the fields of computer science and engineering on one side, and that of the humanities on the other. This overview will provide us with a better sense of which aspects of code and aesthetics have been explored until now, and will invite us to address the remaining gaps.

We have seen that most technical literature, starting from *The Art of Computer Programming*, acknowledges the role that aesthetics have to play in the writing and reading of program texts. Along with the positions of Knuth and Djikstra regarding the importance of paying attention to the aspects of programming practice[**?**] which go beyond strictly mathematical and engineering requirements, Kerninghan and Plauer publish in 1978 their *Elements of Programming Style*[**?**]. In it, they focus on how code snippets with a given intent could be rewritten in order to keep the same intent but gain in quality—that is, in readability and understandability. For instance, the following program:

```
if(i == 0) c = '0'
if(i == 1) c = '1'
if(i == 2) c = '2'
if(i == 3) c = '3'
if(i == 4) c = '4'
if(i == 5) c = '5'
if(i == 6) c = '6'
if(i == 7) c = '7'
if(i == 8) c = '8'
```

```
if(i == 9) c = '9'
```

can be rewritten as:

```
if(i >= 0 && i < 10) c = '0' + i
```

which keeps the exact same functionality, but becomes much clearer. Why it becomes much clearer, though, is thought to be a given for the reader, and not explicited by the authors in terms of concepts such as cognitive surface, repleteness of a symbol system or representation of the main idea(s) at play (casting an integer to a character, rather than individually checking for each integer case). As the authors do employ terms which will form the basis of an aesthetics of software development, such as clarity, simplicity, or expressiveness, there are nonetheless no overarching principles deployed to systematize the manifestation of such principles, only examples are given.

While Kernighan and Plauer do not directly address the relationship of source code and aesthetics, this is something that Peter Molzberger undertakes five years later through an empirical, qualitative study aimed at highlighting the place aesthetics play in an expert programmer's practice[**?**]. Molzberger's study touches upon ideas of over-arching structure, tension between clarity and personality, and levels of expertise in aesthetic judgment. This short paper highlights multiple instances of code deemed beautiful which will be explored further in this thesis, without providing an answer as to *why* this might be the case. For instance, a conception of code as literature does not explain instances involving switch in scales and directions of reading, or a conception of code as mathematics does not explain the explicitly required need for a personal touch when writing source code[**?**].

In the context of formal academic research, such as the IEEE or the Association for Computing Machinery (ACM), subsequent research focuses on how to quantitatively assess a given quality of source code either

through a social perspective on the process of writing[**?**], a semantic perspective on the lexicon being used[**?**, **?**], an empirical study of programming style in the efficiency of software teams[**?**, **?**] or on the visual presentation of code in the comprehension process[**?**]. These focus on the connection of aesthetics with the performance of software development—beautiful code as being related to a good end-product. These methodologies are mostly quantitative, and do not take into account the ”artistry” and ”craft” component as laid out by Knuth and Molzberger, but are rather a big-data representation of Kernighan and Plauer's approach.

The development of software engineering as a profession has led to the publication of several books of specialized literature, taking a practical approach to writing good code, rather than a scientific one. Robert C. Martin's *Clean Code*'s audience belongs to the fields of business and professional trade, drawing on references from architecture, literature and craft in order to lay out the requirements of what he considers to be clean code. These specific mechanisms are highlighted in terms of how they will support a productive increase in the quality of software developed, as opposed as being satisfying in and of themselves. *Clean Code* was followed by a number of additional publications on the same topic and with the same approach[**?**, **?**, **?**]. Here, these provide an interesting counterpoint to academic research on quality code by relying on different traditions, such as the practical handbook, to explain why the way code is written is important.

Technical and engineering literature, then, establish the existence of and need for aesthetics, presented as formal properties which then constitute *quality code*. The methodology in these studies is either empirical, in the case of academic articles, looking at large corpora, more rarely interviewing programmers in order to draw conclusions regarding this relationship between formal properties and quality, while monographs and business literature draw on the experiences of their authors as a programmers to provide source code examples of specific principles, without extending on the rationale and coherence of these principles, let alone within a

source code-specific aesthetic framework. A particularly salient example is Greg Oram's edited volume *Beautiful Code*, in which expert programmers are invited to pick a piece of code and explain why they like it, sometimes commenting it line by line[**?**]. This very concrete, empirical inquiry into what makes source code beautiful does no, however, include a strong enough conclusion as to what *actually* makes code beautiful, but rather writing why they like the idea behind the code, or manifestoes such as Matz's *Code as an Essay*. As such, this monograph will be integrated in our corpus, as commentary rather than academic research. Another limitation to these studies is that they only address one specific group of programmers, and one specific type of software being written. In effect, those who write and read source code are far from being a homogeneous whole, and can be placed along distinct lines—e.g. academics, tinkerers or artists—with distinct practices and standards[**?**]. In none of these studies is it considered whether the conclusions established for one group would be valid for the others.

Before we move on to the perspective of the humanities, one should also note the specific field of philosophy of computer science, which inquires into the nature of computation, from ontological, epistemological and ethical points of view. These are useful both in the meta positioning they take regarding computer science as they well as how they show that issues of representation, interpretation and implementation are still unresolved in the field. Particularly, Rapaport's *Philosophy of Computer Science* provides an exhaustive literature review of the different fields which computer science is being compared to, from mathematics, engineering and art but—interestingly—few references to computer science as having any kind of relation with literature[**?**]. Another, more specific perspective is given by Richard P. Gabriel in his *Patterns of Software*, in which he looks at software as a similar endeavour as architecture, drawing on the works of Christopher Alexander. The focus is on its creative and relationship to patters, a subject we will investigate more in chapter 3. Finally, Brian Cantwell-

Smith's introduction to his upcoming *The Age of Significance: An Essay on the Origins of Computation and Intentionality* touches upon these similar ideas of intentionality by suggesting both that computation might be more productively studied from a humanities or artistic point of view than form a strictly scientifc point of view[**?**]. These philosophical inquiries into computation mention aesthetics mostly on the periphery, but nonetheless challenge the notion of computation as strictly functional, and suggest additional that perspectives on the topic are needed, including that of the arts.

From a humanities perspective, recent literature taking source code as the central object of their study covers fields as diverse as literature, science and technology studies, humanities and media studies and philosophy. Each of these monographs, edited volumes, catalog articles, book chapters or PhD theses, engage with code in its multiple intricacies. Software applications, source code excerpts, programming environments and languages are included in each of these works as primary sources, are considered as text to be read, examined and interpreted.

A first look at *Aesthetic Computing*, edited by Paul A. Fishwick allows us to higlight one of the important points of this thesis: the collection of essays in this collected volume focus more often on the graphical output of the software's work from the end-user's perspective than on the textual manifestations of their source (e.g. Nake and Grabowski's essay on the interface as aesthetic event)[**?**]. As for most studies of aesthetics within computer science, the main focus is on Human-Computer Interaction (HCI) as the art and science of presenting visually the output and affordances of a running program. While a vast and complex field, this is not the topic of this thesis which, rather than focusing on the aesthetics of the computable and executable, is limited to the aesthetics of the computed (texts).

The following works, because of their dealing with source code as text, and due to the background of their authors in literature and comparative

media studies, incorporate some aspect of literary theory and criticism, and authors such as N. Katherine Hayles, Maurice J. Black and Alan Sondheim rely on it as their principal perspective. Black, in his PhD dissertation *The Art of Code*[**?**] initiates the idea of a cross between programming and literature, and hypothesizes that writing source code is an act that is closer to modernism than postmodernism, as it relies on concepts of authorship, formal linguistic systems and, to some extent, self-reference. The aim of the study is to show how code functions with its own aesthetic, one which is distinct and yet closely related to a literary aesthetic. After highlighting how the socio-political structures of computing since the 1950s have affected the constitution of the idea of a code aesthetic both in professional and amateur programmers, Black moves towards the examination of code practices as aesthetic practices. Here, Black limits himself to the presentation of coding practices insofar as they are identified and referred to as aesthetic practices, but exclusively through a social, second-hand account, rather than formal, definition of a source code aesthetic.

Black establishes programming as literature, and vice-versa, he assumes that it is possible to write about literature through the lens of source code. However, the actual analysis of source code with the help of formal literary theories is almost entirely side-stepped, mentioning only Perl poetry as an overtly literary use of code. In summary, Black provides a first study in code as a textual object and as a textual practice whose manifestations programmers care deeply about, but does not address what makes code poetry different in its writing, reading and meaning-making than natural-language poetry.

N. Katherine Hayles, in *My Mother Was A Computer: Digital Subjects and Literary Texts*[**?**], and particularly in the *Speech, Writing, Code: Three Worldviews* essay temporarily removes code from its immediate social and historical situations and establishes it as a cognitive tool as significant in scale as those of orality and literacy[**?**], and attempts to qualify this worldview both in opposition to Saussure's *parole* and Derrida's *trace*, following

cybernetics and media studies thinkers such as Friedrich Kittler and Mark B. Hansen. Specifically, she introduces the idea of a Regime of Computation, which relies on the conceptual specificities of code-based expression (among which: depth, dynamism, fragmentation, etc.). Source-code specific contributions touch upon literary paradigms and cognitive effect in two ways. First, she highlights the way code recombines some traditional dialectics of literary theory, namely paradigmatic/syntagmatic, discreete/-continuous, compilation/interpretation, and flat/stacked languages, acting as such as clearly different mode of expression. Second, she draws on a comparison between two main programming paradigms, object-oriented programming and procedural programming, and on the syntax of programming languages, such as C++, in order to highlight a novel relationship between the structure and the meaning of programming texts, a structure which depends on its degree of similarity with natural languages.

While Hayles provides the basis for a much deeper analysis of source code's formal literary properties, she also maintains that source code studies should keep in mind the ever-underlying materiality that this very source code relies on; and then locates this materiality in the embodiment of users and readers, along with authors such as Mark Hansen[**?**], Bernadette Wegenstein[**?**] and Pierre Lévy[**?**]. Beyond the brief acknowledgment that she has of the political and economical conditions of software development and their impact on electronic texts, Hayles also stops short of considering programming languages in their varieties, and the material apparatuses which support them (documentation, architectures, compilers, tutorials, conferences and communities). Building on this approach, a conception of programming languages as a material seems like a possible avenue for looking into the formal possibilities they afford.

Alan Sondheim's essay *Codework*[**?**], as the introduction of the American Book Review issue dedicated to this specific form, provides another aspect of poetry which integrates source code as a creole language emerging from the interplay of natural and machine languages. Yet, this specific as-

pect of literary work scans the surface of code rather than with its structure and therefore provides more insight in the anthropology of how humans represent code through speech, rather than representing speech through code. This presents a somewhat postmodern view of programming languages, forcing them upon a relational, mutable conception of language as as series speech-acts, and leaving aside their structural and post-structural characteristics. Codework is essentially defined by its content and *milieu*, one which focuses on human exchanges and bypasses any involvement of machine-processing.

Another perspective on the relationship between speech and code is explored by Geoff Cox and Alex Mclean in *Speaking Code: Coding as Aesthetic and Political Expression*[**?**]. They establish reading, writing and executing source code as a speech-act, extending J.L. Austin's theory to a broader political application by including Arendt's approach of human activities and labor[**?**], from which coding is seen as the practice of producing laboring speech-acts.

They consider source code as a located, instantiated presence, understood as a politically semantic object affecting the multiple economic, social and discursive environments in which it lives. Focusing on speech particularly, this study doesn't quite address the syntactic specificities of codes, for example by looking at the use of loops, arrays, or other syntactical structures briefly touched upon by Hayles. Side-stepping the particular grammatical features of that speech, the authors nonetheless often illustrate the points they are working through, or begin developing those points, with snippets of code written by either McLean or established software artists, thus engaging with details of source code and taking a step away from the dangers of fetishizing code, or *sourcery*[**?**]. They include both deductive code (commenting existing source code) or inductive code (code written to act as an example to a point developed by the authors), in a show of the intertextuality of program texts and natural texts.

Away from the cultural relevance of code as developed by Cox and

McLean, Florian Cramer focuses on the cultural history of writing in computation, tying our contemporary fascination with source code into an older web of historical attempts at integrating combinatorial practices from Hebraic texts to Leibniz's universal languages[**?**]. It is in this space between magic and logic that Cramer locates today's experiments in source code (i.e. source code poetry, esoteric languages and codeworks), reminiscent of Simondon's definition of a technical object's essence[**?**]. By relocating it between magic and reality, code is no longer just arbitrary symbols, or machine instructions but also ideal execution, a set of discreete forms which relate to the totality of the world. Once formal execution is considered a cosmogonical force, it becomes synonymous with performative execution, through which it ties back to cultural practices throughout the ages, within both religious and scientific contexts.

Cramer extracts five axes along which to apprehend code-based works: totality/fragmentation, rationalization/occultation, hardware/software, syntax/semantics, artificial/natural language. While all these axes overlap each other, it is the *syntax/semantics* axis which aligns most with this research, givent that these thematical axes are all variations of one another. Towards the end of the book, his development of the concept of speculative programming is also particularly fruitful as the attempt to become a figure of thought and reflection in theory and artistic practice. Cramer states:

> *formalisms [...] have a cultural semantics of their own, even on the most primitive and basic level. With a cultural semantics, there inevitably is an aesthetics, subjectivity and politics in computing.*[**?**]

This points to the relationship between the formal disposition of source code within program texts and the cultural communities composed of the writers and readers of these program texts. As we've seen, code does have social components of varying natures, insofar as it operates as an expressive medium between varying subjects.

Adrian MacKenzie approaches source code, as part of a broader inquiry on the nature of software, through this social lens in *Cutting Code: Software and Sociality*[**?**]. The author focuses on a relational ontology of software: it is defined in how it acts upon, and how it is being acted upon by, external structures, from intellectual property frameworks to design philosophies in software architectures; it only provides an operational definition—software is what it does. His analysis of source code poetry focuses on famous Perl poems, Jodi's artworks and Alex McLean's `forkbomb.pl`, concerned with the executability of code as its dominant feature, dismissing Perl poetry as "*a relatively innocuous and inconsequential activity*"[**?**]. While software could indeed be a "patterning of social relations"[**?**], these social relations also take place through linguistic combinations in program texts. This tending to the material realities of software embedded within social and cultural networks and traditions is echoed in David M. Berry's *The Philosophy of Software: Computation and Mediation in the Digital Age*. His definition of materialities, however, focuses on the technical and social processes *around* code (e.g. build processes, specifications, test suites), rather than on the processes *within* code (i.e. texts, languages). While this former definition results in what he calls a *semiotic place*[**?**], a location in which those processes are organized meaningfully, such a semiotic sense of space could also apply, as we will see in chapter 2, to those intrinsic properties of source code.

Focusing specifically on the category of code poetry, Camille Paloque-Berges published, a couple of years later, *Poétique des codes sur le réseau informatique*[**?**]. This work deploys both linguistic and cultural studies theorists such as Barthes and De Certeau in order to explain these playful acts of source code poetry, along with works of esoteric languages and net.art. While the first chapter focuses on digital literature as the result of executed code in order to develop a heuristic to approach source code, and the third and last chapter focusing on the means of distribution of these works, particularly on the development of net.art, 1337 5p43k and codeworks, the

second chapter is the most relevant to our research focus. In it, Paloque-Berges provides an introduction of creative acts in source code on both a conceptual level (drawing from Hayles and Montfort) and on a technical, syntactical level. She looks at specific programming patterns and practices (`hello world`, quines), technical syntax (e.g. `$`, `@` as Perl tokens for expressing singular or plurals) and cultural paradigms (De Certeau's tactics and strategies), as she attempts to highlight the specificities of source code for aesthetic manifestation and invites further work to be done in this dual vein of close-reading and theoretical contextualization, beyond specific, heightened instances such as Perl poetry.

Honing on a minimal excerpt, *10 PRNT CHR$(205.5+RND(1)) : GOTO 10;*[**?**], is a collaborative work examining the cultural intertwinings of a single line of code, through hardware, language, syntax, outputs and themes. The whole endeavour is one rigorous close-reading of source code, in a deductive fashion, working from the words on the screen and elaborating the context within which these words exist, in order to establish the cultural relevance of source code, as related to the syntax, hardware and cultural context in which these words exist. While the study itself, being a close-reading of only one work, and particularly a *one-liner*, itself a specific genre, is restricted in terms of broad aesthetic statements, it does show how it is possible to talk about code not as an abstract construct but as a concrete reality. Particularly interesting is the section dedicated to the history of the BASIC programming language, and how particular languages afford particular statements and actions better in a given historical context, a point often glossed over in other studies.

A current synthesis of these approaches, Mark C. Marino's *Critical Code Studies*[**?**] and the eponymous research field it belongs to focuses on close-reading of source code as a method for interpreting it as discourse. Particularly, it is organized around cases studies: each with source code, annotations and commentary. This structure furthers the empirical approach we've seen in Cox and McLean's code, starting from lines of source

code in order in order to deduce cultural and social environments and intents through interpretation. This particular monograph, as is stated in the conclusion, offers a set of possible methodologies rather than conclusions in order to engage with code as its textual manifestations: the source code, viewed from different angles, can reveal more than its functional purpose. While Marino, with a background in the humanities, focuses mostly on the literary properties of code as a textual artifact, this thesis builds here on some of his methodologies, particularly reading how the form of the code complements its process and output, and searching the code for clever re-purposing or insight. However, while Marino mentions the aesthetics of code, he does not address the systematic composition of these aesthetics—focusing primarily on *what* the code means and only secondarily on *how* the code means it.

Taking a step back, Warren Sack's *The Software Arts*[**?**] historicizes software development as an epistemological practice, rather than as a strictly economic trade. Connecting some of the main components of software (language, algorithm, grammar), he demonstrates how these are rooted in a liberal arts conception of knowledge and practice, particularly visible as a parallel to Diderot and D'Alembert's encyclopedic attempt at formalizing craft practices. By examining this other, humanistic, tradition in parallel with its dominantly acknowledged scientific counterpart, Sack shows the multiple facets that code and software can support. Starting from the concept of "translation" as an updated version of Manovich's "transcoding", Sack analyzes what is being translated by computing, such as analyses, rhetoric and logic, byt doesn't however address the nature of the process in which these concepts are translated—algorithms as (liberal) ideas, but not as texts.

This activity of programming as craft, already acknowledged by programmers themselves, is further explored in Erik Pineiro's doctoral thesis[**?**]. In it, he examines the concrete, social and practical justifications for the existence of aesthetics within the software development community. Depart-

ing from specific, hand-picked examples such as those featured in Marino's study, his is more of an anthropological approach, revealing what role aesthetics play in a specific community of practicioners. Outlining references to ideas such as *cleanliness*, *simplicity*, *tightness*, *robustness*, amongst others, as aesthetic ideals that programmers aspire to, he does not however summon any specific aesthetic field (whether from literature, mathematics, craft or engineering), but rather frames it in terms of *intrsumental goodness*, with the aesthetics of code being an attempt to reach excellence in instrumental action. While he carefully lays out his argument by focusing on what (a certain group of) programmers actually say, instead of what they might be saying, there remains two limitations: it is not clear how source code as textual material can afford to reach such aesthetic ideals, and whether or not these aesthetic ideals apply to other groups of writers of code, such as the code poets mentioned in some of the works above.

This literature review allows us to have a better grasp of how the relationship between source code and aesthetics has been studied, both from a scientific and engineering perspective, as well as from a humanities perspective.

In the former approach, aesthetics are acknowledged as a component of reading and writing code, and assessed through practical examples, quantitiative analysis and, to a lesser extent, qualitative interviews. The research focus is on the effectiveness of aesthetics in code, rather than on unearthing a systematic approach to making code beautiful, even though issues of cognitive friction and understanding, as well as ideals of cleanliness, readability, simplicity and elegance do arise. As such, they form a good starting ground of varied, empirical investigations. On a more metaphysical level, works in the field of philosophy of computer science point at the fact that the nature of computing and software are themselves evasive, straddling different lines while not aligning clearly with either science, engineering or arts—pointing out that software is indeed something different.

As for the humanities, the focus is predominantly on literary heuristics of a restricted corpus or on socio-cultural dynamics, and the details and examples of the actual code syntax and semantics are often omitted even though the aesthetic aspects of a literary or cultural nature are being explored in source code, as a new kind of writing. There is a potential for beauty and art in source code,as made obvious by code poetry, but such a potential is not assessed through the same empirical lense as the former part of our literature review and only secondarily investigating which of intrinsic features of code can support aesthetic judgments.

Still, some recent studies, such as those by Paloque-Bergès, Montfort et. al, Cox and McLean and Marino, do engage directly with source code examples, and these constitute important landmarks for a code-specific aesthetic theory and methodology, whether it is as poetic language, speech-act, or critical commentary. Source code is taken as a unique literary device, but it remains unclear in exactly which aspects, besides its executability, it is different from both natural languages and low-level machine languages, and how this literary aspect relates to the effective, mathematical and craft-like nature of source code considered in the computer science and engineering literature.

1.2 The aesthetic specificities of source code

We can now turn to some of the gaps and questions left by this review, which can be grouped under three broad areas: dissonant aesthetic fields, lack of correspondance between empirical investigations and theoretical frameworks, and an absence of close-reading of program texts as expressive artifacts.

First, we can see that there are different aesthetic fields being summon when assessing aesthetics in source code. By aesthetic field, I mean the set of medium-specific symbol systems which operate coherently on

a stylistic level, as well as on a thematic level. The main aesthetic fields addressed in the context of source code are those of literature, architecture as well as craft and mathematics. Each of these domains have specific ways to structure the aesthetic experience of objects within that field. For instance, literature can operate in terms of plot, consonance or poetic metaphor, while architecture will mobilize concepts of function, structure or texture. While we will reserve a more exhaustive description of each of these aesthetic fields in chapter 3, the first gap I would like to highlight here is how the multiple aesthetic fields are used to frame the aesthetics of source code, without this plurality being explicitely addressed. Depending on which study one reads, one can see code as literature, as architecture, as mathematics or as craft, and there does not seem to be a consensus as to which of these maps closest to the essence of source code, with exhaustive studies often mentioning several, if not all of the above, fields[**?**].

Second, we can see a disconnect between empirical and theoretical work. The former, historically more present in computer science literature, but more recently finding its way into the humanities, aims at observing the realities of source code as a textual object, one which can be mined for semantic data analysis, or as a crafted object, one which is produced by programmers under specific conditions and replicated through examples and principles, rather than systems and theorems. Conversely, the theoretical approach to code, focusing on computation as a broad phenomenon encompassing engineering breakthroughs, social consequences and disruption of traditional understandings of textuality, rarely confronts such theoretical approaches with the concrete, physical manifestations of computation as source code[9], until recently. In consequence, there are theoretical frameworks that emerge to explain software (e.g. computation, procedurality, protocol), but no frameworks yet which tend to the aesthetics of source code. In the light of the history of aesthetic philosophy, lit-

[9]With exceptions of the recent works cited above.

erature studies and visual arts, defining such a precise framework seems like an elusive goal, but it is rather the constellation of conflicting and complementing frameworks which allow for a better grasp of their object of study. In the case of the particular object of this study, the establishment of such framework taking into account the specifically textual dimension of source code (as opposed to, say, McLean and Cox's attention to the speech dimension) is yet to be done. Following the software development and programming literature, such a framework could productively focus on the role and purpose that aesthetics play within source code, rather than on their autotelic nature as manifestations-for-themselves.

Finally, and related to the point above, we can identify a methodological gap. Due to reasons such as access and skill, close-reading of source code from a humanities perspective has been mostly absent, until the recent emergence of fields of software studies and critical code studies. The result is that many studies engaging with source code as a literary object did not provide code snippets to illustrate the points being made. While not necessary *per se*, I argue that if one establishes an interpretative framework related to the nature and specificity of software, such a framework should be reflected in an examination of one of the main components of software—source code. The way that this gap has been productively addressed in recent years has primarily been done through an understanding of code as a part of broader socio-technical artifacts[10], inscribing it within the phenomenon of computation. This focus on the context in which source code exists therefore leaves some room for similar approaches with respect to its textual qualities. Despite N. Katherine Hayles's call for medium-specificity when engaging with code[**?**], it seems that there hasn't yet been close-readings of a variety of program texts in order to assess them as specific aesthetic objects, in addition to their conceptual and socio-technical qualities.

[10]For instance, see the work done in the field of platform studies[**?**]

Having established an overview of the state of the research on this topic, and having identified some gaps remaining in this scholarship, we can now clarify some of the problems resulting from those gaps with the following questions.

What does source code have to say about itself?

The relative absence of empirical examination of its source component when discussing code does not seem to be consistent with a conception of source code as a literary object. As methodologies for examining the meanings of source code have recently flourished, the techniques of *close-reading*, as focusing first and foremost on "the words on the page"[**?**] have been applied for extrinsic means: extract what the lines of code have to say about the world, rather than what they have to say about themselves, about their particular organization as source files, as typographic objects or as symbol systems expressing concepts about the computational entities they describe. In this sense, it is still unclear how the possible combinations of control flow statements, function calls, function definitions, datatypes, variable declaration and variable naming, among other syntactic devices, enable program texts to be expressive. While close-reading will be a useful heuristic for investigating these problems, it will also be necessary to question the unicity of source code, and take into account how it varies across writers and readers and the social groups they constitute. This problem therefore has to be modulated with respect to the social environment in which it exists—it will then be possible to highlight to what extent the aesthetics of source code vary across these groups, and to what extent they don't.

How does source code relate to other aesthetic fields?

Multiple aesthetic fields are being mapped onto source code, allowing us to grasp such a novel object through more familiar lenses. However, the question remains of what it is about the nature of source code which

can act as common ground for approaches as diverse as literature, mathematics and architecture, or whether these references only touch on distinct aspects of source code. When one talks about structure in source code, do they refer to structure in an architectural sense, or in a literary sense? When one refers to *syntactic sugar* in a programming language, does this have implications in a mathematical sense? This question will involve inquiries into the relationship of syntax and structure, of formality and tacitness, of metaphor and conceptual mapping, and in understanding of how adjectives such as *clean*, *clear* and *simple* might have similar meanings across those different fields. Offering answers to these questions might allow us to move from a multi-faceted understanding of source towards a more specific one, as the meeting point for all these fields, source code might reveal deeper connections between each of those.

How do the aesthetics of source code relate to its functionality?

The final, and perhaps most important problem, concerns the status of aesthetics in source code not as an end, but as a means. A cursory investigation on the topic immediately reveals how aesthetics in source code can only be assessed only once the intended functionality of the software described has been verified. This stands in the way of a rather traditional opposition between beauty and functionality, and therefore begs further exploration. How do aesthetics support source code's functional purpose? And are aesthetics limited to supporting such purpose, or do they serve other purposes, beyond a strictly functional one? This paradox will relate to our first problem, regarding the meaning-making affordances of source code, and touch upon how the expressiveness of formal languages engage with different conceptions of use and function, therefore relating back to Goodman's concept of the languages of art, of which programming languages can be part of.

1.3 Methodology

To address such questions, we propose to proceed by looking at two kinds of texts: program texts and meta-texts. The core of our corpus will consist of the two categories, with additional texts and tools involved.

Our primary corpus is source code, taken as *program texts*. Due to the intricate relationship between source code and digital communication networks, vast amounts of source code are available online natively or have been digitized[11]. They range from a few lines to several thousands, date between 1969 and 2021, with a majority written by authors in Northern America or Western Europe. On one side, code snippets are short, meaningful extracts usually accompanied by a natural language comment in order to illustrate a point. On the other, extensive code bases are large ensembles of source files, often written in more than one language, and embedded in a build system[12]. Both can be written in a variety of programming languages, as long as these languages are composed in alphanumeric characters.

This lack of limitations on size, date or languages stems from our empirical approach. Since we intend to assess code conditionally, that is, based primarily on its own, intrinsic textual qualities, it would not follow that we should restrict to any specific genre of program text. As we carry on this study, distinctions will nonetheless arise in our corpus that align with some of the varieties amongst source—for instance, the aesthetic properties of a program text composed of one line of code might be different from those exhbited by a program text made up of thousands of lines code.

We also intend to use source code in both a deductive and an inductive manner. Through our close-reading of program texts, we will highlight some aesthetic features related to its textuality, taking existing source code

[11] While software was circulating freely on ARPANET and early networks, the application of the intellectual property regime on software in 1974 significantly reduced the open-availability of source code.

[12] A build system is a fairly complex series of code transformations intended to generate executable code.

as concrete proof of their existence. Conversely, we will also write our own source code snippets in order to illustrate the aesthetic features discussed in natural language. This use of source code snippets is widely spread among communities of programmers in order to qualify and strengthen their points in online discussions, and we intend to follow this weaving in of machine language and natural language in order to strengthen our argumentation. This approach will therefore oscillate between theory and practice, the concrete and the abstract, as it both extracts concepts from readings of source code and illustrates concepts by writing source code.

The case of programming languages is a particular one: they do not exclusively constitute program texts (unless they are considered strictly in their implementation details as lexers, interpreters and compilers, themselves described in program texts), but are a necessary, if artificial, condition for the existence of source code. They therefore have to be taken into account when assessing the aesthetic features of program text, as integral part of the affordances of source code. Rather than focusing on their context-free grammars or abstract notations, or on their implementation details, we will focus on the syntax and semantics that they allow the programmer to use. Still, programming languages are hybrid artefacts, and their intrinsic qualities are only assessed insofar as they relate to the aeshetic manifestations of source code written in those languages.

Meta-texts on source code make up our secondary corpus. Meta-texts are written by programmers, provide additional information, context and explanation for a given extract of source code, and is a significant part of the software ecosystem. Even though they are written in natural langauge, this ability to write comments has been a core feature of any programming language very early on in the history of computing, linking any program text with a potential commentary, whether directly among the source code lines (*inline commentary*) or in a separate block (*external commentary*)[13]. Exam-

[13] Such a distinction isn't a strict binary, and systems of inscription exist which couple code a commentary more tightly, such as WEB or Juptyer Notebook.

ples of external commentaries include user manuals, textbooks, documentation, journal articles, forums posts, blog posts or emails. The inclusion in our corpus of those meta-texts is due to two reasons: the practical reason of the high epistemological barrier to entry when it comes to assessing source code in linguistic or hardware environments which one isn't familiar with, and the theoretical reason of including the (aesthetic) judgment of programmers as it supports our conditional, rather than constitutive, approach.

While we intend to look at source through close-reading, favoring the role and essence of each line as a meaningful, structural element, rather than that of the whole, our interpretation of meta-texts will take place via discourse analysis. Building on Dijk and Kintsch's work on discourse comprehension[**?**], we intend to approach these texts at a higher level, in terms of the lexical field they use, as a marker of the aesthetic field they refer to, as well as at a lower level, noting which specific syntactic aspects of the code they refer to. This focus on both the micro-level (e.g. local coherence and proposition analysis) and on the macro-level (e.g. socio-cultural context, intended aim and lexical field usage) will allow us to link specific instances of written code with the broader semantic field that they exist in. This connection between micro- and macro- relies on the hypothesis that there is something fundamentally similar between a source code construct, its meaning and use at the micro-level, and the aesthetic field to which it is attached at a macro-level, a hypothesis we will address further when investigating the role of metaphor in source code.

In the end, this process will allow us to construct a framework from empirical observations. The last part of our methodology, after having completed this analysis of program-texts and their commentaries, is to cross-reference it with texts dealing with the manifestation of aesthetics in those peripheral fields. Literary theory, centered around the works of I.A. Richards, Roland Barthes and Paul Ricoeur can shed light on the attention to form, on the interplay of syntax and semantics, of open and closed texts,

and suggest productive avenues through the context of metaphor. Architecture theory will be involved through the two main approaches mentioned by software developers: functionalism as illustrated by the credo *form follows function* and works by Vitruvius, Louis Sullivan and the Bauhaus on one side, and pattern languages as initiated by the work of Christopher Alexander on the other. The aesthetic nature of the two remaining fields, mathematics and craft, have a thinner tradition of formalized aesthetics than literature and architecture, but we nonetheless include essays and monographs from practicioners in the field addressing those issues. This additional set of texts will allow us to operate comparatively when it comes to expliciting source code's aesthetics.

This study therefore aims at weaving in empirical observations, discourse analysis and external framing, in order to propose systematic approaches to source code's textuality. However, these will not unfold in a strictly linear sequence; rather, there will be a constant movement between practice and theory and between code-specific aesthetic references and broader ones: this interdisciplinary approach intends to reflect the multifaceted nature of software.

1.4 Roadmap

Our first step in this study is an empirical assessment of how programmers consider aesthetics with their practice or reading and writing it, first from a conceptual standpoint. After acknowledging and underlining the diversity of those practices, from software developers and scientists to artists and hackers, we will identify which concepts and references are being used the most when referring to beautiful code—concepts such as clarity, simplicity, cleanliness, and others. These concepts will then allow us to touch upon the field that are being referred to when considering the practice of programming: literature, architecture and mathematics as domains in them-

selves, and craft as a particular approach to these domains. Finally, we will how how the overlap of these concepts can be found in the process of *understanding*—communicating abstract ideas through concrete manifestations.

After establishing the role of aesthetics as a means for understanding source code, we will proceed to analyze further such a relationship between understanding, source code and aesthetics. We will see that one of the main features of source code is the elusiveness of its meaning, whether effective or intended. Beautiful code is often code that can be understood clearly, which raises the following question: how can a completely explicit and formal language allow ambiguity? The answer to this question will involve an analysis of the two audiences of source code: humans and machines.

Taking a step back towards textuality, we will then assess how the different fields that are being referred to when talking about source code have touched upon these issues of understanding, from rhetoric to literature, through architecture and mathematics. Thinking in terms of surface-structure and deep-structure, we will establish a first connection between program texts and literary text through their use of metaphors. Since metaphors aren't exclusively literary devices, looking at them from a cognitive perspective will also raise issues of modes of knowledge, between explicit, implicit and tacit. The understanding of beauty in architecture, based on the two traditions mentioned above, will provide an additional perspective by providing concepts of structure, function and usability. These will echo a final inquiry into mathematical beauty, drawing a direct link between idea and implementation, theorem and proof, and providing a deeper understanding of the concept of *elegance*.

With a firmer grasp on the stakes of source code as an understandable text, we can now turn to its effective manifestations, by close-reading program texts. Working through *structure*, *syntax* and *vocabulary*, we will be able to formalize a set of textual typologies involved in producing an

aesthetic experience through source code. Particularly, we will highlight where those tokens differ across communities of practice, and where they overlap, keeping in mind the conditionality of those aesthetic judgments, and attempt to trace connections between specific textual configurations of source code with the ideals summoned by the programmers. After this deductive consideration, we will move on to apply these typologies to several larger program texts—ranging from the LaTeX codebase, the Carnivore software artwork to several code poems. These will highlight a remaining component in the concrete manifestation of source code aesthetics: the place of programming languages.

At this point, we will have established aesthetics in source code as a way to address the inherent tensions of a program text's dual audience, computers and humans. Being understandable by both humans and machines is indeed the feat of programming languages, the symbol systems on which beautiful texts depend on. As we've elicited the intricacies of aesthetic manifestations in human to machine communication, we then investigate machine to machine communication. Deconstructing programming languages as formal grammars will show that there are very different conceptions of semantics and meanings expected from the computer than those expected from a human, even though a machine's perspective on beautiful code could still be based around concepts of effectiveness, simplicity and performance. *Contra* those, human use of programming languages reaches into the extreme of *esolangs*—an investigation into those will reveal that language is effectively considered as a material, one whose base elements can be recombined into unexpected puzzling structures.

Recognizing programming languages as the bridge between the two domains of programming—the human of the machine—will allow us to clarify how the different aesthetic fields (literature, architecture, mathematics) relate to programming. We will show how programming languages provide a gradual interface between different modes of being of source code: source code as text, source code as structure and source code as theory. The

need for aesthetics arises from the tradeoffs that need to be made when these different modes of being overlap[**?**].

We will then turn back to our research questions to suggest some possible answers. The reorganization of the source aesthetic fields inot a linear succession of the interpration or compilation process from high-level to low-level hints at a specifically spatial nature of program texts. Indeed, the specific aesthetics of source code are those of a constant doubling between the specificities of the human (such as natural handling of ambiguity, and intuitive understanding of the problem domain) and of the machine (such as speed of execution, and reliance on explicit formal grammars, which can also be seen as the tension between surface structure, one that is textual and readable, and deep structure, one that is made up of dynamic processes representing complex concepts, and yet devoid of any fluidity or ambiguity. It is this dynamism, both in terms of *where* and *when* code could be executed, which suggest the use of aesthetics in order to grasp more intuitively the topology and chronology, the state and behaviour of a program text.

Finally, we will relate the approaches of Goodman of art as cognitively effective symbol systems, and of Simondon's consideration of aesthetic thought as a link between technical thought and religious thought. Starting from a practical perspective on aesthetics taking from the field of craft—the thing well done—, aesthetics also highlight functionality on a cognitive level—the thing well thought. Beauty in source code seems to be dominantly what is useful and thoughtful, even when they are reflected in the distorting mirrors of hacks and esoteric languages, broadening our possible understandings of what aesthetics can do, and what functionality can be.

1.5 Implications and readership

This thesis fits within the field of software studies, and aims at clarifying what do we mean when we refer to code *code as*.... Code as literature, architecture or mathematics, code as philosophy or as craft, are metaphors which can be examined productively by looking at the texts themselves, an approach that has only been deployed in relatively recent work.

This relationship between practice, function and beauty is the broad, underlying question of this study. In the vein of the cognitive approach to art and aesthetics, this study is an attempt to show how aestethics play a communicative role, and how concrete manifestations can, through a metaphorical process, hint at broader ideas. In this sense, this study is not just about the relation of aesthetics and function, but also about the function of aesthetics. While this idea of aesthetics as a way of communicating ideas could be equally applied across artistic and non-artistic domains, another aim of this thesis is to highlight the relativity a aesthetic standards: using a similar medium, practices, uses and purposes determine as much, if not more, of the artistic worth of a given program text.

By examining the result of the practice of programmers at a close-level, this study hopes to contribute to a clarification of what exactly is programming, along with the consequences of the embedding of software in our social, economic and political practices. In order to address the question of whether algorithms are political in themselves, or if their use is political, it is important to define clearly what it is that we are talking about when discussing algorithms. A clarification of source code on a concrete level attempts to help clarify what this essential component of algorithms, and opens up potential for further work in terms of thinking no longer of the aesthetics of source code, but of its poetics, in the way source code, as a language of art, is also a way of worldmaking.

To this end, this thesis is aimed at a variety of readers and audience. From the humanities perspective, digital humanists and literary the-

orists interested in the concrete manifestations of source code as specific meaning-making techniques will be able to find the first steps of such an approach being laid out, and contrast these specific technique with the broader poetics of code studied by other scholars, or with the aesthetics of natural language texts.

Programmers and computer scientists will find an attempt at formalizing something they might have known implicitly ever since they started practicing writing and reading code, and the approach of languages as poetics and structure might help them think through these aspects in order to write perhaps more aesthetically pleasing, and thus perhaps better, code. Conversely, anyone engaged seriously in a craft activity could find here a rigorous study of what goes on into a specific craft, asking how their own practice engages with tools and modes of knowledge, and with a more explicit conception of beauty.

Finally, such a specific conception of beauty, then, will also be of interest to artists and art theorists. By investing aesthetics without a direct relation to the artwork, but rather within a functional purpose, this study suggests that one can think through beauty and artworks not as ends, but as ways to accomplish things that formal systems of explanation might not be able to achieve. An aesthetics of source code would therefore aim at highlighting the purpose of instrumental beauty within a textual environment.

Chapter 2

Aesthetic ideals in programming practices

The first step in our study of aesthetics in source code aims at identifying the aesthetic ideals that programmers ascribe to source code; that is, the qualifiers and semantic fields that they refer to when discussing program texts. To that end, we first start by clarifying whom we refer to by the term *programmers*, which reveals a multiplicity of practices and purposes, from *ad hoc*, one-line solutions, to printed code and massively-distributed codebases.

We then turn to the kinds of beauty that these programmers aspire to. After expliciting our methodology of discourse analysis, we engage in a review of the various kinds of publications and writings that programmers write, read and refer to when it comes to qualifying their practice. From this will result a clust of adjectives—e.g. *clean*, *simple*, *smelly*—which we argue are used in an aesthetic sense. These will provide a useful framework to inspect, in subsequent chapters, their formal manifestations as typed-out tokens.

From these, we can then move to a description of which aesthetic fields

are being referenced by programmers on a broader level, and consider how multiple kinds of beauties, from literary, to architectural and mathematical conceptions of beauty can overlap and be referred to by the same concrete medium.

Finally, we focus our attention on one of the points of overlap in these different references: the importance of function, craft and knowledge in the disposition and representation of code. We will show how this particular way of working plays a central role in an aesthetic approach to source code and results from the specificity of code as a cognitive material, a specificity we will inquire further in the next chapter.

2.1 The practice of programmers

The history of software development is that of a specific, reserved practice which was born in the aftermath of the second world war, which trickled down to broader and broader audiences at the eve of the twenty-first century. Through this development, multiple ways of doing, approaches and applications have been involved in producing software, resulting in different communities and types of programming. Each of these focus on the description of specific instructions to the computer, but do so with specific characteristics. To this end, we take a socio-historical stance on the field of programming, highlighting how diverse practices emerge at different moments in time, and how they are connected to contemporary technical and economic organizations.

Even though such types of reading and writing source code often overlap with one another, this section will highlight a diversity of more or less loose ways in which code is being written, notably in terms of references—what do they consider good?—, purposes—what do they write for?—and examples—how does their code look like?. First, we take a look at the software industry, to identify professional *software developers*, the large code bases they work on and the specific organizational practices within which they write it. They are responsible for the majority of source code written today, and do so in a professional and productive context, where maintainability, testability and reliability are the main concerns. Then, we turn to a parallel practice, one that is often exhibited by software developers, as they also take on the stance of *hackers*. Disambiguating the term reveals a set of practices where curiosity, cleverness, and idiosyncracy are central, finding unexpected solutions to complex problems, sometimes within artificial constraints. Finally, we look at *scientists* and *poets*. On one end, *scientists* embody a rather academic approach, focusing on abstract concepts such as simplicity, minimalism and elegance; they are often focused on theoretical issues, such as implementation of algorithms and mathemat-

ical models, as well as programming language design. On the other end, *poets* read and write code first and foremost for its textual and semantic qualities, publishing code poems online and in print, and engaging deeply with the range of metaphors allowed by a dynamic linguistic medium such as code.

While this overview encompasses most of the programming practices, we leave aside some approaches to code, mainly because they do not directly engage with the representation of source code as a textual matter. More and more, end-user applications provide the possibility to program in more or less rudimentary ways, something referred to as the "low-code" approach[**?**], and thus contributing to the blurring of boundaries between programmers and non-programmers[1].

2.1.1 Software developers

From local hardware to distributed software

As Niklaus Wirth puts it, *the history of software is the history of growth in complexity*[**?**], while paradoxically, lowering the barrier to entry. As computers' technical abilities in memory managment and processing power increased year on year since the 1950s, the nature of writing instructions shifted accordingly.

In his history of the software industry, Martin Campbell-Kelly traces the development of a discipline through both an economic and a technological lens, and he identifies three consecutive waves in the production of software[**?**]. During the first period, as soon as the 1950s, and continuing throughout the 1960s, software developers were contractors hired to en-

[1] For instance, Microsoft's Visual Basic for Applications, Ableton's Max For Live, MIT's Scrath or McNeel's Grasshopper are all programming frameworks which are not covered within the scope of this study. In the case of VBA and similar office-based high-level programming, it is because such a practice is a highly personal and *ad hoc* one, and therefore is less available for study.

gage directly with a specific computing machine. These computing, mainframes, were large, expensive, and rigid machines, requiring hardware-specific knowledge of the Assembler instruction set specific to each one, since they didn't feature an operating system which could facilitate some of the more basic memory allocation and input/output functions, and thus interoperable program-writing[2]. Two distinct groups of people were involved in the operationalization of such machine: electrical engineers, tasked with designing hardware, and programmers, tasked with implementing the software. While the former historically received the most attention[**?**], the latter was mostly composed of women and, as such, not considered essential in the process[**?**]. At this point, then, programming is closely tied to hardware.

The second period in software development starts in the 1960s, as hardware started to switch from vacuum tubes to transistors and from magnetic core memory to semiconductor memory, making them faster and more capable to handle complex operations. On the software side, the development of several programming languages, such as FORTRAN, LISP and COBOL, started to address the double issue of portability—having a program run unmodified on different machines with different instruction sets—and expressivity—allowing programmers to use high-level, English-like syntax, rather than assembler instruction codes. By then, programmers are no longer theoretically tied to a specific machine, and therefore acquire a certain autonomy, a recognition which culminates in the naming of the field of *software engineering* in 1968 at a NATO conference[3].

The third and final phase that Campbell-Kelly identifies is that of mass-market production: following the advent of the UNIX family of operating systems, the distribution of the C programming language, the wide availability of C compilers, and the appearance of personal computers such as the Commodore 64, Altair and Apple II, software could be effectively en-

[2] One of the first operating systems, MIT's Tape Director, would be only developped in 1956[**?**]

[3] source?

tirely decoupled from hardware[4]. And yet, software immediately enters a crisis, due to software development projects running over time and budget, being unreliable in production and unmaintainable in the long-run. What this highlighted is that the creation of software was no longer a corollary to the design of hardware, and that it would become the main focus of computing as a whole[**?**], and that it should therefore be addressed as such. It is at this time that discussions around best practices in writing source code started to emerge, once the activity of the programmer was no longer restricted to *tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment*[**?**].

This need for a more formal approach to the actual process of programming found one of its most important manifestations in Edsger Djikstra's *Notes on Structured Programming*[**?**]. In it, he argues for moving away from programming as a craft, and towards programming as an organized discipline, with its methodologies and systematization of program construction. Despite its laconic section titles[5], Djikstra nonetheless contributed to establish a more rigorous typology of the constructs required for reliable, provable programs—based on fundamentals such as sequence, selection, iteration and recursion—, and aimed at the formalization of the practice. Along with other developments (such as Hoare's contribution on proper data structuring[**?**], or the rise of object-oriented programming) programming would solidify its foundations as a profession:

> We knew how the nonprofessional programmer could write in an afternoon a three-page program that was supposed to satisfy his needs, but how would the professional programmer design a thirty-page program in such a way that he could really justify his design? What intellectual discipline would be needed? What properties could such a professional programmer demand with

[4]For a more detailed account of the personal computer revolution, see: Cerruzzi, P., A History of Modern Computing[**?**]

[5]See, for instance, Chapter 1: "*On our inability to do much*"

> justification from his programming language, from the formal tool he had to work with? [**?**]

As a result of such interrogations comes an industry-wide search for solutions to the intractable problem of programming: that it is *a technique to manage information which in turn produces information*. To address such a conundrum, a variety of tools, formal methods and management processes enter the market; they aim at acting as a *silver bullet*[**?**], addressing the cascade of potential risks[6] which emerge from large software applications. However, this growth in complexity is also accompanied by a diversification of complexity: as computers become more widely available, and as higher-level programming languages provide more flexibility in their expressive abilities, software engineering is being applied to a variety of domains, each of which might need a specific solution, rather than a generic process. Confronted with this diversity of applications, business literature on software practices flourishes[7], acknowledging that the complexity of software should be tackled at its bottleneck: the reading and writing of source code.

The most recent step in the history of software developers is the popularization of the Internet and of the World Wide Web. Even though the former had existed under as ArpaNet since 1969, the network was only standardized in 1982 and access to it was provided commercially in 1989. Built on top of the Internet, the latter popularized global information exchange, including technical resources to read and write code. Software could now be written by remote individual written on *cloud computing* platforms, shared through public repositories and deployed via containers with a lower barrier to entry than at the time of source code printed in magazines, of overnight batch processing and of non-time-sharing systems.

These software developers have written some of the largest codebases to this date, mainly because this type of activity represents the largest

[6] See `https://catless.ncl.ac.uk/Risks/` for such risks

[7] See Jackson, Principles of Program Design, or Martin, Clean Code, among others.

fraction of programmers. Due to its close ties to commercial distributors, however, source code written in this context often falls under the umbrella of proprietary software, thus made unvailable to the public. Some examples that we include in our corpus are either professional codebases that have been leaked[8], open-source projects that have come out of business environments, such as Soundcloud's Prometheus, Google's TensorFlow or Facebook's React, or large-scale open-source projects which nonetheless adhere to structured programming guidelines, such as Donald Knuth's TeX typesetting system or the Linux Foundation's Linux kernel.

Features of the field

The features of these codebases provide the backdrop to, and start to hint at, the qualities that software developers have come to ascribe to their object of practice. First, the program texts they write are large, much larger than any other codebase included in this study, they often feature multiple programming languages and are highly structured and standardized: each file follows a pre-established convention in programming style, which favors an authoring by multiple programmers without any obvious trace to a single individual authorship. These program texts stand the closest to a programming equivalent of engineering, with its formalisms, standards and usability. From this perspective, the IEEE's Software Engineering Body of Knoweldge (SWEBOK) provides a good starting point to survey the specificities of software developers as source code writers and readers[**?**]; the main features of which include the definition of requirements, design, construction testing and maintenance.

Software requirements are the acknowledgement of the importance of the *problem domain*, the domain to which the software takes its inputs from, and to which it applies its outputs. For instance, software written for a calculator has arithmetic as its problem domain; software written for a learning

[8] Such as the Microsoft Windows XP source code[**?**].

management system has students, faculty, education and courses as its problem domain; software written a banking institution has financial transactions, savings accounts, fraud prevention and credit lines as its problem domain. Requirements in software development aim at formalizing as best as possible the elements that must be used by the software in order to perform a successful computation, and an adequate formalization is a fundamental requirement for a successful software application.

Following the identification and codification of requirements, software design relates to the overall organization of the software components, considered not in their textual implementation, but in their conceptual agency. Usually represented through diagrams or modelling languages, it is concerned with *understanding how a system should be organized and designing the overall structure of that system*[**?**]. Of particular interest is the relationship that is established between software development and architecture. Considered a creative process rather than a strictly rational one, due to the important role of the contexts in which the software will exist (including the problem domain)[**?**], software architecture is considered essential from a top-down perspective, laying down an abstract blueprint for the implementation of a system, as well as form a bottom-up one, representing how the different components of an existing system interact. This apparent contradiction, and the role of architecture in the creative aspects of software development, will be further explored in chapter 2.

Software construction relates to the actual writing of software, and how to do so in the most reliable way possible. The SWEBOK emphasizes first and foremost the need to minimize complexity[9], in anticipation of likely changes and possible reuse by other software systems. Here, the emphasis on engineering is particularly salient: while most would refer to the

[9]Following C. Anthony Hoare's assessment in his Turing Award Lecture that *"there are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."*

creation of software as *writing* software, the IEEE document refers to it as *constructing* software[10]. Coding is only assessed as a practical consideration, one which should not take up the most attention, if the requirements, design and testing steps are satisfyingly implemented. Conversely, a whole field of business litterature[**?**, **?**, **?**, **?**] has focused specifically on the process of writing code, starting from the assumption that:

> We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such details that a machine can execute them is *programming*.[**?**]

As we see, the tension identified by Djikstra some thirty years before between craft and discipline is still alive and well at the beginning of the twenty-first century, even though the focus on code still relates to the need for reliability and maintainability in a maturing industry.

Software maintenance, finally, relates not to the planning or writing of software, but to its reading. Software is notoriously filled with bugs[11] and can, at the same time, be easily fixed while already being in a production environment through software update releases. This means that the lifecycle of a software doesn't stop when then first version is written, but rather when it does not run anymore, and this implies that the nature of software allows for it to be edited across time and space, by other programmers which might not have access to the original group of implementers: consequently, software should be first and foremost understandable—SWEBOK lists the first feature of coding as being *techniques for creating understandable source code*[**?**]. This requirement ties back to one of the main prob-

[10] The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging.[**?**].

[11] McConnell estimates that the industry average is about 15 - 50 errors per 1000 lines of delivered code.[**?**].

lems of software, which is its notorious cognitive complexity, one that remains at any stage of its development.

What does this look like in practice, then? Ideals of clarity, reusability and reliability—and their opposites—can be found in some of the available code bases of professional software. The paradox to be noted here is that, even though software developers write the most code, it is the least accessible online and, as such, the following excerpts are covering the range of commercial software (Microsoft Windows), licensed and publicly available software (Kirby), and an open-source software (Prometheus).

The first excerpts come from the source code for Microsoft Windows 2000, which was made public in 2004. The program text contains 28,655 files, the largest of our corpus, by multiple orders of magnitude, with 13,468,327 combined lines and including more than 10 different file extensions. Taking a closer look at some of these files allow us to identify some of the specific features of code written by software developers, and how they specifically relate to architectural choices, collaborative writing and verbosity.

First, the most striking visual feature of the code is its sheer size. Representing such a versatile and low-level system such as an operating system manifest themselves in files that are often above 2000 lines of code. In order to allow abstraction techniques at a higher-level for the end-developer, the operating system needs to do a significant amount of ”grunt” work, relating directly to the concrete reality of the hardware platform which needs to be operated. For instance, the initialization of strings of text for the namespaces (a technique directly related to the compartmentalization) is necessary, repetitive work which can be represented using a rhythmic visual pattern, such as in `cmdatini.c`:

```
{
  ULONG i;
```

```
    RtlInitUnicodeString( &CmRegistryRootName,
                          CmpRegistryRootString );

    RtlInitUnicodeString( &CmRegistryMachineName,
                          CmpRegistryMachineString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareName,
                          CmpRegistryMachineHardwareString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDescriptionName,
                          CmpRegistryMachineHardwareDescriptionString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDescriptionSystemName,
                          CmpRegistryMachineHardwareDescriptionSystemString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDeviceMapName,
                          CmpRegistryMachineHardwareDeviceMapString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareResourceMapName,
                          CmpRegistryMachineHardwareResourceMapString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareOwnerMapName,
                          CmpRegistryMachineHardwareOwnerMapString );

    RtlInitUnicodeString( &CmRegistryMachineSystemName,
                          CmpRegistryMachineSystemString );

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSet,
                          CmpRegistryMachineSystemCurrentControlSetString);

    RtlInitUnicodeString( &CmRegistryUserName,
                          CmpRegistryUserString );

    RtlInitUnicodeString( &CmRegistrySystemCloneName,
                          CmpRegistrySystemCloneString );

    RtlInitUnicodeString( &CmpSystemFileName,
                          CmpRegistrySystemFileNameString );

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetEnumName,
                          CmpRegistryMachineSystemCurrentControlSetEnumString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetEnumRootName,
                          CmpRegistryMachineSystemCurrentControlSetEnumRootString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetServices,
                          CmpRegistryMachineSystemCurrentControlSetServicesString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetHardwareProfilesCurrent,
                          CmpRegistryMachineSystemCurrentControlSetHardwareProfilesCurrentString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlClass,
                          CmpRegistryMachineSystemCurrentControlSetControlClassString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlSafeBoot,
                          CmpRegistryMachineSystemCurrentControlSetControlSafeBootString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlSessionManagerMemoryManagement,
                          CmpRegistryMachineSystemCurrentControlSetControlSessionManagerMemoryManagementString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlBootLog,
```

```
                        CmpRegistryMachineSystemCurrentControlSetControlBootLogString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetServicesEventLog,
                        CmpRegistryMachineSystemCurrentControlSetServicesEventLogString);

    RtlInitUnicodeString( &CmSymbolicLinkValueName,
                        CmpSymbolicLinkValueName);

#ifdef _WANT_MACHINE_IDENTIFICATION
    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlBiosInfo,
                        CmpRegistryMachineSystemCurrentControlSetControlBiosInfoString);
#endif

    //
    // Initialize the type names for the hardware tree.
    //

    for (i = 0; i <= MaximumType; i++) {

        RtlInitUnicodeString( &(CmTypeName[i]),
                            CmTypeString[i] );

    }

    //
    // Initialize the class names for the hardware tree.
    //

    for (i = 0; i <= MaximumClass; i++) {

        RtlInitUnicodeString( &(CmClassName[i]),
                            CmClassString[i] );

    }

    return;
}
```

The repetition of the `RtlInitUnicodeString` call in the first part of this listing stands at odds with today's industry-standard practices of not repeating oneself; these would rather point towards the second part of the code, the two `for()` statements. While this practice would only be formalized in Andy Hunt's *The Pragmatic Programmer* in 1999[**?**], the longevity of the windows 2000 operating system and its update cycle would nonetheless have affected how this code is written. The reason why such a repetition applies is the requirement of registering each string with the kernel. Dealing with a different problem domain (kernel instructions) leads to a different kind of expected aesthetics[12].

[12]Effectively, references to **`RtlInitUnicodeString()`** happen 1580 times across 336

Verbosity, the act of explicitly writing out statements which could be functionally equivalent in a compacted form, is a significant feature of the Windows 2000 codebase, and also relies on a particular semantic environment: that of using the C programming language. As mentioned above, the development of C and UNIX in the 1970s have led to wide adoption of the former, and to some extent of the later (even though Windows is a notable exception since it is an operation system not based on the UNIX tradition). What we see in this listing is the consequence of this development: using a verbose language inevitably leads to a verbose program text, something we will see in the following section on hacker's code and will explore much further in chapter 5.

Another significant aesthetic feature of the Windows 2000 program text is its use of comments, and specifically how those comments representing the multiple, layered authorship. This particular source code is one that is written across individuals and across time, each with presumably its own writing style. Yet, writing source code within a formal organization leads to the adoption of coding styles, with the intent that *all code in any codebase should look like a single person typed it, no matter how many people contributed*[**?**]. For instance, the excerpt in **??** from `jdhuff.c` is a example of such overlapping of styles:

Here, we see three different writings of comments `//`, `/* */` as well as different kinds of capitalizations. Those comments are ignored at compile time: that is, they are not meaningful to the machine, and are only expected to be read by other programmers, primarily programmers belonging to one's organization. This hints at the various origins of the authors, or at the very least at the different moments, and possible mental states of the potential single-author: irregularity in comment writing can connect to irregularities in semantic content of the comments. This irregularity becomes suspicious, and leads to ascribing a different epistemological value to them. If comments aren't procedurally guaranteed to be reflected in the execu-

files

```
  no_more_data:
                  // There should be enough bits still left in the data
                       segment;
                  // if so, just break out of the outer while loop.
                  if (bits_left >= nbits)
                          break;
                        /* Uh-oh.  Report corrupted data to user and
                            stuff zeroes into
                         * the data stream, so that we can produce some
                              kind of image.
                         * Note that this code will be repeated for
                             each byte demanded
                         * for the rest of the segment.  We use a
                             nonvolatile flag to ensure
                         * that only one warning message appears.
                         */
                  if (! *(state->printed_eod_ptr))
                  {
                          WARNMS(state->cinfo, JWRN_HIT_MARKER);
                          *(state->printed_eod_ptr) = TRUE;
                  }
                  c = 0;                        // insert a zero byte
                      into bit buffer
      }
    }

    /* OK, load c into get_buffer */
    get_buffer = (get_buffer << 8) | c;
    bits_left += 8;
  }

  /* Unload the local registers */
  state->next_input_byte = next_input_byte;
  state->bytes_in_buffer = bytes_in_buffer;
  state->get_buffer = get_buffer;
  state->bits_left = bits_left;

  return TRUE;
}
```

Listing 2.1: buffer.c

tion, and outcome, of the program, then one tend to rely on the fact that "The only document that describes your code completely and correctly is the code itself" ([**?**]). This excerpt highlights the constant tension between source code as the canonical origin of knowledge of what the program does and how it does it, while comments refelect the idiosyncratic dimension of all natural-language expressions of human programmers.

And yet, this chronological and interpersonal spread of the program text, as well as organizational practices require the use of comments in order to maintain aesthetic and cognitive coherence in the program, if only by the use of comment headers, which locate a specific file within the greater architectural organization of the program text. For instance, in `pnpenum.c`:

This highlights both the multiple authorship (here, we have one original author and one revisor) as well as the evolution in time of the file: comments are the only manifestation of this layering of revisions which ultimately results in the "final" software[13].

A complementary example is the Kirby CMS[14]. With development starting in 2011, a first release in 2012 and having developed a steady user-base, correlated in Google Trends analytics[15], consistent forum posts[16] and commit history[17], it exhibits a particular set of features, in both its purpose and its source code alike. Kirby is an open-source, developer-first (meaning that it affords direct engagement of other developers with its architecture through modification, extension or partial replacement), single-purpose project. As such, it stands at the other end of the commercial, efficient software spectrum than Microsoft Windows 2000.

The Kirby source code is entirely available online, and the following snip-

[13] The term "final" is in quotes, since the Windows 2000 source contains the mention **BUGBUG** 7436 times across 2263 files, a testatment to the constant state of unfinishedness that some software might remain in.

[14] Allgeier, Bastian et. al., https://github.com/getkirby/kirby, 2011, consulted in 2022

[15] https://trends.google.com/trends/explore?date=all&q=kirby%20cms

[16] https://forum.getkirby.com

[17] https://github.com/getkirby/kirby

```
  /*++

Copyright (c) 1996 Microsoft Corporation

Module Name:

    enum.c

Abstract:

    This module contains routines to perform device enumeration

Author:

    Shie-Lin Tzong (shielint) Sept. 5, 1996.

Revision History:

    James Cavalaris (t-jcaval) July 29, 1997.
    Added IopProcessCriticalDeviceRoutine.

--*/
```

Listing 2.2: enum.c

```
 /**
 * Enables distinct select clauses.
 *
 * @param bool $distinct
 * @return \Kirby\Database\Query
 */
public function distinct(bool $distinct = true)
{
    $this->distinct = $distinct;
    return $this;
}
```

Listing 2.3: Query.php

pets hint at another set of formal values—conciseness, expliciteness and delimitation. Conciseness can be seen in the lengths of the various components of the code base. For instance, the core of Kirby consists in 248 files, with the longest being `src/Database/Query.php` at 1065 lines, and the shortest being `src/Http/Exceptions/NextRouteException.php`, for an average of 250 lines per file. At the third level, the length of the function bodies is also minimal, ranging from XXXX to XXXX*calculate*. Compared to the leading project in the field, Wordpress.org, which has respectively XXXX *insert analysis here*.

If we look at a typical function declaration within Kirby, we found one such as the distinct setter for Kirby's database:

Out of these 11 lines, the actual functionality of the function is focused on one line, `$this->distinct = $distinct;`. Around it are machine-readable comment snippets, and a function wrapper around the simple variable setting. The textual overhead then comes from the wrapping itself: the actual semantic task of deciding whether a query should be able to include distinct select clauses (as opposed to only allowing join clauses), is now

decoupled from its actual implementation (one could describe to the computer such an ability to generate distinct clauses by assigning it a boolean value, or an integer value, or passing it as an argument for each query, etc.). The quality of this writing, at first verbose, actually lies in its conciseness in relation to the possibilities for extension that such a form of writing allows: the `distinct()` function could, under other circumstances, be implemented differently, and still behave similarly from the perspective of the rest of the program. Additionally, this wrapping enables the setting of default values (here, `true`), a minimal way to catch bugs by always providing a fallback case.

Kirby's source code is also interestingly explicit in comments, and succint in code. Let us take, for instance, from the `Http\\Route` class (**??**).

The 9 lines above the function declaration are machine-readable documentation. It can be parsed by a programmatic system and used as input to generate more classical, human-readable documentation[18]. This is noticeable due to the highly formalized syntax `param string name_of_var`, rather than writing out "this function takes a parameter of type string named `name_of_var`". This does compensate for the tendency of comments to drift out of synchronicity with the code that they are supposed to comment, by tying them back to some computational system to verify its semantic contents, while providing information about the inputs and outputs of the function.

Beyond expliciting inputs and outputs, the second aspect of these comments is targeted at the *how* of the function, helping the reader understand the rationale behind the programmatic process. Comments here aren't cautionary notes on specific edge-cases, as seen in fig. XX above, but rather natural language renderings of the overall rationale of the process. The implication here is to provide a broader, and more explicit understanding of the process of the function, in order to allow for further maintenance,

[18] See, for instance, JavaDocs, or ReadTheDocs

```
 /**
 * Tries to match the path with the regular expression and
 * extracts all arguments for the Route action
 *
 * @param string $pattern
 * @param string $path
 * @return array|false
 */
public function parse(string $pattern, string $path)
{
    // check for direct matches
    if ($pattern === $path) {
        return $this->arguments = [];
    }

    // We only need to check routes with regular expression since all
        others
    // would have been able to be matched by the search for literal
        matches
    // we just did before we started searching.
    if (strpos($pattern, '(') === false) {
        return false;
    }

    // If we have a match we'll return all results
    // from the preg without the full first match.
    if (preg_match('#^' . $this->regex($pattern) . '$#u', $path,
        $parameters)) {
        return $this->arguments = array_slice($parameters, 1);
    }

    return false;
}
```

Listing 2.4: Route.php

extension or modification.

Finally, let us look at a subset of the function, the clause of the third `if statement: (preg_match('#^' . $this->regex($pattern). '$#u', $path, $parameters))`. Without comments, one must realize on cognitive gymnastics and knowledge of the PHP syntax in order to render this as an extraction of all route parameters, implying the removal of the first element of the array. In this sense, then, Kirby's code for parsing an HTTP route is both verbose—in comments—and parsimonious—in code.

What these aesthetic features (small number of files, short file length, short function length) imply is an immediate feeling of *building blocks*. Short, graspable, (re-)usable (conceptual) blocks are made available to the developer directly, as the Kirby ecosystem, like many other open-source projects, relies on contributions from individuals who are not expected to have any other encounter with the project other than, at the bare minimum, the source code itself.

These two examples, Microsoft Windows 2000 and Kirby CMS, highlight some of the presentations of source code—repetition, verbosity, commenting and conciseness—within socio-technical ecosystems made up of hardware, institutional practices ranging from corporate guidelines to open-source contribution, with efficiency and usability remaining at the forefront, both at the result level (the software) and at the process level (the code).

Software developers are a large group of practitioners whose focus on producing effective, reliable and sustainable software, leads them to writing in more-or-less codified manner. Before diving into how such a manner of writing relates to references from architecture and engineering in order to foster simplicity and understandability, in section 2.2, we acknowledge that the bondary between groups of practicioners isn't a clear-cut one, and so we turn to another practice closely linked to professional development—hacking.

2.1.2 Hackers

Popular description of hackers tend to veer towards dishelvement, obsession and esoteric involvement with the machine, accompanied by seemingly-radical value systems and political beliefs. They are often depicted as lonely, obsessed programmers, hyperfocused on the task at hand and able to switch altered mental states as they dive into computational problems, as described by Joseph Weizenbaum in 1976[19]. While some of it is true—for instance, the gender, the compulsive behaviour and the embodied connection to the machine—, hackers nonetheless designate a wider group of people, one which writes code driven by curiosity, cleverness and freedom. Such a group has had a significant influence in the culture of programming, with which it overlaps with the aforementionned values of intellectual challenges.

To hack, in the broadest sense, is to enthusiastically inquire about the possibilities of exploitation of technical systems[20] and, as such, predates the advent of the computer[21]. Computer hacking specifically came to proeminence as early computers started to become available in north-american universities, and coalesced around the Massachussets Institute of Technology's Tech Model Railroad Club[**?**]. Computer hackers were

[19]" Wherever computer centers have become established, that is to say, in countless places in the United States, as well as in virtually all other industrial regions of the world, bright young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the buttons and keys on which their attention seems to be as riveted as a gambler's on the rolling dice. When not so transfixed, they often sit at tables strewn with computer printouts over which they pore like possessed students of a cabalistic text." ([**?**])

[20]"HACKER [originally, someone who makes furniture with an axe] n. 1. A person who enjoys learning the details of programming systems and how to stretch their capabilities, as opposed to most users who prefer to learn only the minimum necessary. 2. One who programs enthusiastically, or who enjoys programming rather than just theorizing about programming." ([**?**])

[21]See Rosenbaum's report in the October 1971 issue of Esquire for an account of phreaking, computer hacking's immediate predecessor[**?**].

skilled and highly-passionate individuals, with an autotelic inclination to computer systems: these systems mattered most when they referenced themselves, instead of interfacing with a given problem domain. Early hackers were often self-taught, learning to tinker with computers while still in high-school[**?**], and as such tend to exhibit a radical position towards expertise: skill and knowledge aren't derived from academic degrees or credentials, but rather from concrete ability and practical efficacy[22].

The histories of hacking and of software development are deeply intertwined: some of the early hackers worked on software engineering projects—such as the graduate students who wrote the Apollo Guidance Computer routines under Margaret Hamilton—, and then went on to profoundly shape computer infrastructure. Particularly, the development of the UNIX operating system by Dennis Ritchie and Ken Thompson is a key link in connecting hacker practices and professional ones. Developed from 1969 at Bell Labs, AT&T's research division, UNIX was "very close to being the first system under which a programmer could sit down directly at a machine and compose programs on the fly, exploring possibilities and testing while composing" ([**?**]), and was "was brainstormed by three people and implemented by Ken Thompson in two days — on an obsolete machine that had been designed to be a graphics terminal for a 'real' computer." ([**?**]). This was a system which was supporting the free exploration of a system's boundaries central to the hacker culture, and which relied on sharing and circulating source code in order to allow anyone to improve it—in effect, AT&T's inexpensive licensing model until the 1980s, and the use of the C programming language starting from 1977 made it widely available within university settings[23]. UNIX, then, was spreading its design philosophy of clear, modular, simple and transparent design across programming com-

[22] A meritocratic stance which has been analyzed in further in [**?**]

[23] "Unix has become well entrenched in the nation's colleges and universities due to Western Electric's extensive, inexpensive licensing of the system. As a result, many of today's graduating computer scientists are familiar with it." ([**?**])

munities.

The next step in the evolution of hacker culture was to build on this tenet to share source code, and hence to make written software understandable from its textual manifestation. The switch identified in the previous section from hardware being the most important component of a computing system to software had lead manufacturers to stop distributing source code, making proprietary software the norm. Until then, executable software was the consequence of running the source code through a compilation process; around the 1980s, executable software was distributed directly as a binary file, its exact contents an unreadable series of 0s and 1s. As a result to licensing changes of the UNIX system, the GNU project was created, and in its wake the Free Software Foundation, which established the ethical requirement to access the source code of any software.

In the meantime, personal microcomputers came to the market and opened up this ability to tinker and explore computer systems beyond the realms of academic-licensed large mainframes and operating systems. Starting with models such as the Altair 8800, the Apple II and the Commodore 64, as well as with easier, interpreted computer languages such as BASIC, whose first version for such micro-computers was written by Bill Gates, Paul Allen and Monte Davidoff[**?**]. While not considered ”proper” programming by other programmers, the microcomputer revolution allowed for new groups of individuals to explore the interactivity of source code due to their small size when published as type-in listings.

In the wake of the larger free software movement, emerged its less radical counterpart, the open-source movement, as well as its more illegal counterpart, security hacking. The former are usually the types of individuals depicted in mainstream news outlets when they reference hackers: programmers indulging breaching private systems, sometimes in order to cause illegal financial, intelligence or material harm. Security hackers, sometimes called crackers, form a community of practice of their own, with

ideas of superior intelligence, subversion, adventure and stealth[24]. These practices do refer to the original conception of hacking—getting something done quickly, but not well—and include such a practical, efficient appoach into its own set of values and ideals, which are in turn represented in the kinds of program texts[25].

Meanwhile, the open-source movement took the tenets of hacking culture and adapted it to make it more compatible to the requirements of businesses. Open-source can indeed be seen as a compromise between the software industry development practices and the efficacy of free software development. Indeed, beyond the broad values of intellectual curiosity and skillful exploration, free software projects such as the Linux kernel, the Apache server or the OpenSSL project are highly efficient, and used in both commercial, non-commercial, critical and non-critical environments[**?**]. Such an approach sidesteps the political and ethical values held in previous iterations of the hacker ethos in order to focus exclusively on the sharing of source code and open collaboration while remaining within an inquisitive and productive mindframe. With the advent of corporate *hackathons*—short instances of intense collaboration in order to create new software, or new features on a software system—are a particularly salient example of this overlap between industry practices and hacker practices[**?**][26].

Hackers are programmers which, while overlapping with industry-

[24] For a lyrical account of this perception of the hacker ethos, see *The Conscience of a Hacker*, published in Phrack Magazine: " This is our world now... the world of the electron and the switch, the beauty of the baud. We make use of a service already existing without paying for what could be dirt-cheap if it wasn't run by profiteering gluttons, and you call us criminals. We explore... and you call us criminals. We seek after knowledge... and you call us criminals." ([**?**])

[25] Those program texts include computer viruses, worms, trojan horses and injections, amongst others.

[26] Along with the address of the software corporate giant Meta's headquarters: 1, Hacker Way, Menlo Park, CA 94025, U.S.A.

embedded software developers, hold a set of values and ideals regarding the purpose and state of software. Whether academic hackers, amateurs, security hackers or open-source contributors, all are centered around the object of source code as a vehicle for communicating the knowledge held within the software, bypassing auxiliary resources such as natural-language documentation. Those political and ethical values often overlap with aesthetic values associated to how the code exists in its textual manifestation.

Sharp and clever

> To hack is, according to the dictionary, ”to cut irregularly, without skill or definite purpose; to mangle by or as if by repeated strokes of a cutting instrument”. I have already said that the compulsive programmer, or hacker as he calls himself, is usually a superb technician. It seems therefore that he is not ”without skill” as the definition will have it. But the definition fits in the deeper sense that the hacker is ”without definite purpose”: he cannot set before him a clearly defined long-term goal and a plan for achieving it, for he has only technique, not knowledge. He has nothing he can analyze or synthesize; in short, he has nothing to form theories about. His skill is therefore aimless, even disembodied. It is simply not connected with anything other than the instrument on which it may be exercised. His skill is that of a monastic copyist who, though illiterate, is a first rate calligrapher.[**?**]

While he looks down on hackers, perhaps unfairly, from the perspective of a computer scientist whose theoretical work can be achieved only through thought, pen and paper—an approach to programming which we will address in the next section—, the point still remains: hackers are first and foremost technical experts who can get lost into technics for their

own sake. From a broad perspective, hackers therefore seem to exhibit an attitude of *direct engagement*, *subverted use* and *technical excellence*. Gabriella Coleman, in her anthropological study of hackers, *Coding Freedom: The Ethics and Aesthetics of Hacking*, highlights that hackers value both semantic ingenuity[27] and technical wittiness, even though source code written by hackers can take multiple shapes, from one-liners, to whole operating systems, to deliberate decisions to subvert best practices in crucial moments

The *one-liner* is a piece of source code which fits on one line, and is usually intepreted immediately by the operating system. They are terse, concise, and eminently functional: they accomplish one task, and one task only. This binary requirement of functionality (in the strict sense of: "does it do what it's supposed to do, or not?") actually finds a parallel in a different kind of one-liners, the humoristic ones in jokes and stand-up comedy. In this context, the one-liner also exhibits the features of conciseness and impact, with the setup conflated with the punch line, within the same sentence. One-liners are therefore self-contained, whole semantic statements which, through this syntactic compression, appear to be clever—in a similar way that a good joke is labelled clever.

In programming, one-liners have their roots in the philosophy of the UNIX operating system, as well as in the early diffusion of computer programs for personal computer hobbyists[**?**]. On the one side, the Unix philosophy is fundamentally about building simple tools, which all do one thing well, in order to manipulate text streams[**?**]. Each of these tools can then be piped (directing one output of a program-tool into the input of the next program-tool) in order to produce complex results—reminiscing of the orthogonality feature of programming languages. Sometimes openly acknowledged by language designers—such as those of AWK—the goal is to write short programs which shouldn't be longer than one line. Given that

[27] Hackers themselves tend to favor puns—the free software GNU project is a recursive acronym for *GNU's Not UNIX*.

constraint, a hacker's response would then be: how short can you make it?

If writing one-line programs is within the reach of any medium-skilled programmer, writing the shortest of all programs does become a matter of skill, coupled with a compulsivity to reach the most syntactically compressed version. For instance, Guy Steele[28] recalls:

> This may seem like a terrible waste of my effort, but one of the most satisfying moments of my career was when I realized that I had found a way to shave one word off an 11-word program that [Bill] Gosper had written. It was at the expense of a very small amount of execution time, measured in fractions of a machine cycle, but I actually found a way to shorten his code by 1 word and it had only taken me 20 years to do it[**?**].

This sort of compulsive behaviour is also manifested in the practice of *code golf*, challenges in which programmers must solve problems by using the least possible amount of characters—here, the equivalent of *par* in golf would be Kolmogorov complexity[29]. So minimizing program length in relation to the problem complexity is a definite feature of one-liners, since choosing the right programming language for the right taks can lead to a drastic reduction of syntax, while keeping the same expressive and effective power. Tasked with parsing a text file to find which lines had a numerical value greater than 6, Brian Kernighan writes the code in C**??**[30]:

[28] Influential langugage designer, who worked on Scheme, ECMAScript and Java, among others.

[29] See: `https://en.wikipedia.org/wiki/Kolmogorov_complexity`

[30] From Succesful Language Design, Brian Kernighan at the University of Nottingham, `https://www.youtube.com/watch?v=Sg4U4r_AgJU`

The equivalent in AWK, a language he designed, and which he actually refers to in the comment on line 15, presumably as a heuristic as he is writing the function, is see in **??**

The difference is obvious, not just in terms of formal clarity and cleanliness of the surface structure, but also in terms of matching the problem domain: this obviously prints every line in which the third field is greater than 6. The AWK one-liner is more efficient, more understandable because more intuitive, and therefore more beautiful. On the other hand, however, one-liners can be so condensed that they loose all sense of clarity for someone who doesn't have a deep knowledge in the specific language in which it is written. Here is Conway's game of life implemented in one line of APL**??**.

```
[caption={gol.apl}, label={gol.apl}, float, floatplacement=H]
  life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ∘.⌽ ⊂⍵}
```

The obscurity of such a line—due to its highly-unusual character notation, and despite the pre-existing knowledge of the expected output—shows why one-liners are usually highly discouraged for any sort of code which needs to be *worked on* by other programmers. Cleverness in programming indeed tends to be seen as a display of the relationship between the programmer and the machine, rather than between different programmers, and only tangentially about the machine. On the other hand, though, the nature of one-liners makes them highly portable and shareable, infusing them with what one could call *social beauty*. Popular with early personal computer adopters, at a time during which the source code of programs were printed in hobbyist magazines and needed to be input by hand, and during which the potential of computation wasn't as widely distributed amongst society, being able to type just one line in, say, a BASIC interpreter, and resulting in unexpected graphical patterns created a sense of magic and wonder in first-time users—how can so little do so much?[31].

[31]For an example fo such one-liner, see for instance: `https://www.youtube.com/watch?v=0yKwJJw6Abs`

Another example of beautiful code written by hackers is the UNIX operating system, whose inception was an informal side-project spearheaded by Ken Thompson and Dennis Ritchie in the 1970s. As the first portable operating system, UNIX's influence in modern computing was significant, e.g. in showing the viability and efficiency of text-based processing, hierarchical file-system, shell scripting and regular expressions, amongst others. UNIX is also one of the few pieces of production software which has been carefully studied and documented by other developers. One of the most famous examples is *Lions' Commentary on UNIX 6th Edition, with Source Code* by John Lions, an annotated edition of the UNIX source code, which was circulated illegaly in classrooms for twenty years before its official publication was authorized by the copyright owners[**?**]. Coming back to the relationship between architecture and software development, Christopher Alexander asks, in the preface of Richard P. Gabriel's *Patterns of Software*[**?**],

> *For a programmer, what is a comparable goal? What is the Chartres of programming? What task is at a high enough level to inspire people writing programs, to reach for the stars?*

And UNIX might be one of the answers to that question, both by its functionality, and by its conciseness, if not alone by its availability. Another program which qualifies as beautiful hacker code, due both to its technical excellence, unusual solution and open-source availability is the function to compute the inverse square root of a number, a calculation that is particularly necessary in any kind of rendering application (which heavily involves vector arithmetic). It was found in the *Quake* source code, listed in **??** verbatim[32].

[32] The Quake developers aren't the authors of that function—the merit of which goes to Greg Walsh—but are very much the authors of the comments.

What we see here is indeed a combination of the understanding of the problem domain (what's the acceptable result I need to maintain a high-framerate with complex graphics), and how the specific knowledge of computers (i.e. bit-shifting of a float cast as an integer) and the snappiness and wonder of the the comments[33]. The use of `0x5f3759df` is what programmers call a *magic number*, a literal value whose role in the code isn't made clearer by a descriptive variable name. Usually bad practice and highly-discouraged, the magic number here is exactly that: it does makes the magic happen.

Further examples of such intimate knowledge of both the language and the machine can be found in the works of the *demoscene*. Starting in Europe in the 1980s, demos were first short audio-visual programs which were distributed along with *crackware* (pirated software), and to which the names of the people having cracked the software were prepended, in the form of a short animation[**?**]. Due to this very concrete constraint—there was only so much memory left on a pirated disk to fit such a demo—programmers had to work with these limitations in order to produce the most awe-inspiring effects. Indeed, one notable feature of the demoscene is that the output should be as impressive as possible, as an immediate, phenomenological appreciation of the code which could make this happen[34]. Indeed, the `comp.sys.ibm.pc.demos` news group states in their FAQ:

> A Demo is a program that displays a sound, music, and light show, usually in 3D. Demos are very fun to watch, because they seemingly do things that aren't possible on the machine they were programmed on.
>
> Essentially, demos "show off". They do so in usually one, two, or all three of three following methods:

[33] *what the fuck?* indeed

[34] For an example, see *Elevated*, programmed by iq, for a total program size of 4 kilobytes: `https://www.youtube.com/watch?v=jB0vBmiTr6o`

- They show off the computer's hardware abilities (3D objects, multi-channel sound, etc.)
- They show off the creative abilities of the demo group (artists, musicians)
- They show off the programmer's abilities (fast 3D shaded polygons, complex motion, etc.)[**?**]

This showing off, however, does not happen through immediate engagement with the code from the reader's part, but rather in the thorough explanation of the minute functionalities of the demo by its writer. Because of these constraints of size, the demos are usually written in C, openGL, Assembly, or the native language of the targeted hardware. Source code listings of demos also make extensive use of shortcuts and tricks, and little attention is paid to whether or not other humans would directly read the source—the only intended recipient is a very specific machine (e.g. Commodore 64, Amiga VCS, etc.). The release of demos, usually in demoparties, are sometimes accompanied by documentation, write-ups or presentations[35]. However, this presentation format acknowledges a kind of individual, artistic feat, rather than the *egoless programming* lauded by Brooks in professional software development[36].

Pushing the boundaries of how much can be done in how little code, here is a 256-bytes demo resulting in a minute-long music video[**?**] on the Commodore 64. It is firsted listed as a hexademical dump by its author (see **??**)

Even with knowledge of how hexadecimal instructions map to the instruction set of the specific chip of of the Commodore 64 (in this case, the SID 8580), the practical use of these instructions takes productive advan-

[35] You can find *Elevated*'s technical presentation here: `https://www.iquilezles.org/www/material/function2009/function2009.pdf`

[36] In architecture, such technical and artistic feat for its own sake, devoid of any reliable social use, is the pavillion, or the folly.

```
0000000 0801 080d d3ff 329e 3232 0035 0000 4119
0000010 d01c dc00 0000 d011 0be0 3310 610e f590
0000020 0007 1fff 4114 24d5 2515 5315 6115 29d5
0000030 0f1b 13e6 13e6 02d0 20e6 61a9 1c85 20a7
0000040 3fe0 08f0 0c90 114e 6cd0 fffc 6da0 2284
0000050 d784 4b4a a81c 13a5 3029 02d0 1cc6 2fe0
0000060 11f0 02b0 02a2 10c9 09f0 298a aa03 f3b5
0000070 0a85 ab2d b000 b711 b622 9521 a500 4b13
0000080 aa0e f8cb cc86 0749 0b85 13a5 0f29 0fd0
0000090 b8a9 1447 0290 1485 0729 b5aa 85f7 a012
00000a0 b708 910d 880f f910 b7a8 9109 8803 f9d0
00000b0 7e4c 78ea 868e 8e02 d021 4420 a2e5 bdfd
00000c0 0802 0295 d0ca 8ef8 0315 cc4c a900 8d50
00000d0 d011 ad58 dc04 c3a0 1c0d 48d4 044b 30a0
00000e0 188c 71d0 e6cb 71cb 6acb 2005 58a0 d505
00000f0 cb91 dfd0 aa2b 6202 1800 2026 2412 1013
```

Listing 2.5: A Mind is Born

tage of ambivalence and side-effects. In the words of the author, Linus Akesson (emphasis mine):

> We need to tell the VIC chip to look for the video matrix at address $0c00 and the font at $0000. This is done by writing $30 into the bank register ($d018). But this will be done from within the loop, as doing so allows us to use the value $30 for two things. *An important property of this particular bank configuration is that the system stack page becomes part of the font definition*.

Demosceners therefore tend to write beautiful, deliberate code which is hardly understandable by other programmers without explanation, and yet hand-optimized for the machine. This presents a different perspective of the relationship between aesthetics and understanding, in which aesthetics do not support and enable understanding, but rather become a proof of the mastery and skill required to input such a concise input for such an overwhelming output. This shows in an extreme way that one does need a degree of expert knowledge in order to appreciate it—in this sense, aesthetics in programming are shown to be almost often dependent on pre-existing knowledge.

Hackers are then programmers who write code within a variety of settings, from academia to hobbyists through professional software development. Yet, some patterns emerge. First, one can see the emphasis on the *ad hoc*, insofar as choosing "the right tool for the right job"[37] is a requirement for hacker code to be valued positively. This requirement thus involves an awareness of which tool will be the most efficient at getting the task at hand done, with a minimum of effort and minimum of overhead, usually at the expense of sustaining or maintaining the software beyond any immediate needs, making it available or comprehensible neither across time

[37] find citation

nor across individuals, a flavour of *locality*. Second, this need for knowing and understanding one's tools hints at a sort of "materiality" of code, whether instructions land in actual physical memory registers, staying away from abstraction and remaining in "concrete reality" by using magic numbers, or sacrificing semantic clarity in order to *"shave off"* a character or two.

The ideals at play in the writing and reading of source code for hackers is thus centered around specific means of knowledge: knowledge of the hardware, knowledge of the programming language used and knowledge of the tradeoffs acceptable all the while exhibiting an air of playfulness—how far can one go pushing a system's boundaries before it breaks down entirely? How little effort can one put in order to get a maximum outcome? Yet, one aspect that seems to elude hackers in their conception of code is that of conceptual soundness. If code is considered beautiful by attaining previously unthought achievements of purposes with the least amount of resources, rationalization as to why, whether *a prior* or *a posteriori*, does not seem to be a central value. Hackers exhibit tendencies to both *get the job done* and *do it for the sake of doing it*. This behaviour is unlike that of computer and data scientists, and towards whom we turn to next.

If hacking can be considered a practice which deals with the practical intricacies of programming, involving concrete knowledge of the hardware and the language, our third group tends towards the opposite. Programming scientists (of which computer scientists are a subset) engage with progamming first and foremost at the conceptual level, with different locii of implementation: either as a *theory*, or as a *model*.

2.1.3 Scientists

Historically, then, programming emerged as a distinct practice from computing sciences: not all programmers are computer scientists, and not all computer scientists are programmers. Nonetheless, scientists engage with programming and source code in two distinct ways, and as such open up the landscape of the type of code which can be written, and of the standards which support the evaluation of good code. First, we will look at code being written in support of non-specifically computer science research activities and, through it, examine how the specific needs of usability, replicability and data structuring link back to standards of software development. Second, we will inquire specifically into code written by computer scientists, such as programming language designers, and develop how computer implementation exists between the dual scientific pillars of theorization and experimentation[**?**].

Computation as a means

Scientific computing, defined as the use of computation in order to solve non-computer science tasks, started as early as the 1940s and 1950s in the United States, aiding in the design of the first nuclear weapons[**?**]. Essentially, calculations necessary to the verification of theories in disciplines such as physics, chemistry or mathematics were handed over to the computing machines of the time. Beyond the military applications of early computer technology, one can point in particular to Harlow and Fromm's article on *Computer Experiments in Fluid Dynamics*, published in 1965, focusing on how the advent of computing technology would prove to be of great assistance in physics and engineering:

> The fundamental behavior of fluids has traditionally been studied in tanks and wind tunnels. The capacities of the modern computer make it possible to do subtler experiments on the

computer alone.[**?**]

In general, then, computation and computers are perceived as promising automated aids in processing data at a much faster rates than human scientists[**?**]. The remaining issue, then, is to make computers more accessible to scientists which did not have direct exposure to computers. Beyond the unaffordable price point of university mainframes before the personal computer revolution, another vector for simplification and accessibility is the development of adequate programming languages[38]. Developed in 1964 at Dartmouth College, BASIC (Beginners' All-purpose Symbolic Instruction Code) aims at addressing this hurdle by designing "*the world's first user-friendly programming language*"[39]. The intent is to provide non-computer scientists with easy means to instruct the computer on how to instruct the computer to perform computations relevant to their work. Still, computing in the academia will only pick up with the distribution of the multiple versions of the UNIX timesharing system, which allowed multiple users to use a given machine at the same time, and the performance boost provided by the C programming language in the late 1970s.

By the dawn of the 21st century, scientific computing had increased in the scope of its applications (extending beyond engineering and experimental, so-called "hard" sciences, to social sciences and the humanities) as well as in the time spent developing and using software[**?**][**?**], with the main programming languages used being MATLAB, C/C++ and Python. While C and C++'s use can be attributed to their historical standing, popularity amongst computer scientists, efficiency for systems programming and speed of execution, MATLAB and Python offer different perspectives. MATLAB, originally a matrix calculator from the 1970s, became popular with the academic community by providing features such as a reliable way to do

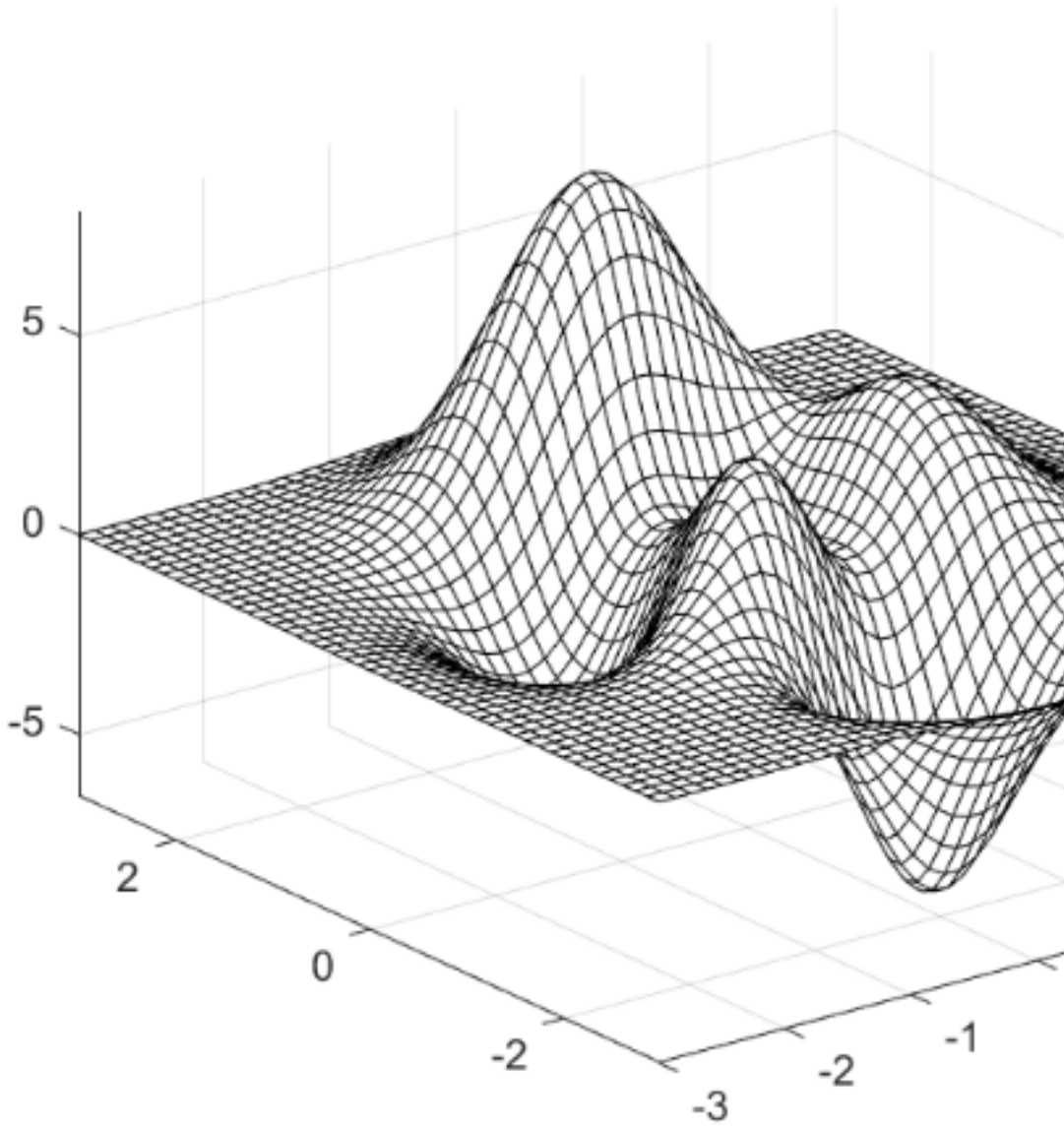
[38] See Chapter 4 for a more complete discussion on programming languages.

[39] `https://web.archive.org/web/20190611180750/https://granitegeek.concordmonitor.com/2019/06/11/finally-a-historical-marker-that-talks-about-something-important/`

```
X = (-3:1/8:3)*ones(49,1);
Y = X';
Z = 3*(1-X).^2.*exp(-(X.^2) - (Y+1).^2) \
- 10*(X/5 - X.^3 - Y.^5).*exp(-X.^2-Y.^2) \
- 1/3*exp(-(X+1).^2 - Y.^2);
mesh(X,Y,Z)
```

Listing 2.6: mesh.m

floating-point arithmetic and a graphical user interface (GUI). Along with its powerful array-manipulation features, the ability to visualize large series of data and plot it on a display contributed to MATLAB's popularity[**?**], features shared with RStudio, a GUI to the R programming language. In **??**, one can see how concise the plotting of a three-dimensional plane is in MATLAB, requiring only one call to `mesh`.



In parallel to MATLAB and R, Python represents the advent of the so-called scripting languages. Scripting languages are programming languages which offer readability and versatility, along with decoupling from

the actual operating system that it is being executed on. System languages, such as C, are designed and used in order to interact directly with the computer hardware, and to constitute data structures from the ground up[**?**]. On the other hand, scripting languages were designed and used in order to connect existing software systems or data sources together, most notably in the early days of shell scripting (such as `Bash`, `sed` or `awk`). Starting with the late 1990s, and the appearance of languages such as Perl[40] and Python[41], scripting languages became more widely used by non-programmers who already had data to work with and needed tools to exploit it. In the following decades, the development of additional scientific libraries such as *SciKit*, *NumPy* for mathematics and numerical work or *NLTK* for language processing and social sciences in Python complemented the language's ease of use by providing manipulation of complex scientifc concepts[**?**], a phenomenon of user-extension which has also been observed in R and MATLAB's ecosystems[**?**].

This steady rise of scientific computing has nonetheless highlighted the apparent lack of quality standards in academic software, and how the lack of value judgments on the software written might impact the reliability of the scientific output. Perhaps the most well-known example of such a lack is the one revealed by the leak of the source code of the Climate Research Unit from the University of East Anglia in 2009[**?**]. In the leak, inline comments of the authors, as well as code reviews of external software developers point out to the CRU leak as being a symptom of the state of academic software. As Professor Darrel Ince stated to the UK Parliamentary Committee in February 2010:

> There is enough evidence for us to regard a lot of scientific software with worry. For example Professor Les Hatton, an international expert in software testing resident in the Universities of Kent and Kingston, carried out an extensive analysis of several

[40] First version developed in 1987 by Larry Wall

[41] First official release in 1991 by Guido Van Rossum

> million lines of scientific code. He showed that the software had an unacceptably high level of detectable inconsistencies.[**?**]

As a response to this realization, the beginning of the 2000s has seen the desire to re-integrate the best practices of software engineering in order to correct scientific software's lack of accuracy[**?**]. Indeed, software engineering, as we've seen above, had developed on their own since its establishment as an independent academic discipline and professional field. Such a split, described by Diane Kelly as a "*chasm*"[**?**] then had to face the different standards to which commercial software and scientific software are subject to. For instance, commercial software must be extensible and performant, two qualities that do not necessarily translate to an academic setting, in which software might be written within a specific, time-constrained, research project, or in which access to computing resources (i.e. supercomputers) might be less of a problem.

Within Landau et. al's conception of the scientific process as the progression from problem to theory, followed by the establishment of a model, the devising of a method, and then on to implemementation and finally to assessment[**?**], code written as academic software is now involved in the latter two stages of method and implementation. Within those two stages, software has to abide by the processes and requirements of scientific research. First and foremost, reproducibility is a core requirement of scientific research in general[42] and bugs in a scientific software system can lead to radically different ouptuts given slightly different input data, while concealing the origin of this radical difference. Good academic code, then, is one which defends actively against these, perhaps to the expense of performance and maintainability. This can be addressed by reliable error-handling, regular assertions of the state of the processed data and extensive unit testing[**?**].

[42] This requirement dates back the 1600s with Robert Boyle and the Invisible College in England[**?**]

Furthermore, a unique aspect of scientific software comes from the lack of clear upfront requirements. Such requirements, in software development, are usually provided ahead of the programming process, and should be as complete as possible. As the activity of scientists is defined by an incomplete understanding of the application domain, requirements tend to emerge as further knowledge is developed and acquired[**?**]. As a result, efforts have been made to familiarize scientists with software development best practices, so that they can implement quality software by themselves. Along with field-specific textbook[43] the most prominent initiative in the field is *Software Carpentry*, a collection of self-learning and teaching resources which aims at implementing software best practices across academia, for scientists and by scientists. Founded by Greg Wilson, the co-editor of *Beautiful Code*, the organization's title refers directly to equivalents in the field of software development[44].

We conclude here on a convergence of quality standards of broad academic software towards the quality standards of commercial software development[45]. And yet, this convergence is due to, as we've seen, a past divergence between computation and science, as computer science worked towards asserting and pursuing its own field of research. As a subset of science, computer science nonetheless holds specific standards, taking software not as a means to an end, but as the end itself.

[43] See *Effective Computation in Physics*[**?**] or *A Primer for Computational Biology*[**?**] as textbooks covering similar software-oriented material from different academic perspectives.

[44] See 1.1.2 *Features of the field* above fo a discussion of the literature on writing good code for software developers)

[45] See Graphbrain at `https://github.com/graphbrain/graphbrain` for such an example. The code's organization and formal features are congruent and on par with commercial software.

Computation as an end

Computer scientists are scientists whose work focuses on computation as a means, rather than as a tool. As such, they study the phenomenon of computation, investigating its nature and effects through th development of a theoretical frameworks around it. Originally derived from computability theory, as a branch of formal mathematical logic, computation emerged as an autonomous field from work in mechanical design and configuration (Ada Lovelace and Charles Babbage), work on circuit and language design (C. S. Pierce, Konrad Zuse and John Von Neumann), work on mathematical foundations (Alan Turing and Alonzo Church), information theory (Claude Shannon), systems theory (Norbert Wiener) and expert systems (John McCarthy and Marvin Minsky)[**?**]. In the middle of such a constellation ranging from mathematical theory to practical electronics, computer science establishes institutional grounding with the inauguration of the first dedicated academic deparment at Purdue University in 1962.

From this multifaceted heritage and academic interdisciplinarity, computer scientists have established the some of the foundations of the field, identifying key areas such as data structures, algorithms and language design as the foundations of the discipline[**?**]. Through the process, the tracing of the "roots" of computation remained a constant debate as to whether computer science exists within the realm of mathematics, of engineering or as a part of the natural sciences. The logico-mathematical model of computer science contends that one can do computer science without a computer, solely armed of a pen and a paper, while the engineering approach of computer science tends to put more practical matters, such as architecture, language design and systems programming at the core of the discipline; both being a way to generate and process information as natural phenomenon[**?**].

The broad difference we can see between these different conceptions of computer science is that of *episteme* and *techne*. On the theoretical and

scientific side, computer science is concerned with the primacy of ideas, rather than of implementation. The quality of a given program is thus deduced from its formal (in the mathematical sense) properties, rather than its formal (in the aesthetic sense) properties. The first manifestations of such a theoretical focus can be found in the Information Processing Language (1956), which was designed and developed originally to prove Bertrand Russell's *Principia Mathematica*. While the IPL, as one of the very first programming languages, influenced the development of multiple subsequent languages, not least of all being LISP, some later languages came to be known as logic programming languages, based on a formal logic syntax of facts, rules and clauses about a given domain and whose correctness can be easily proven (see **??** below for an example of the *Prolog* logic programming language).

Due to its Turing-completeness, one can write programs such as language processing, web applications, cryptography or database programming (using the *Datalog* variant of *Prolog*), but its use remains limited outside of theoretical circles in 2021[46]. Another programming language shares this feature of theoretical soundness faced with a limited range of actual use in production environments, Lisp—*LISt Processor*—designed to process lists. It was developed in 1958, the year of the Dartmouth workshop, on Artificial Intelligence by its organizator, John McCarthy. Inheriting from IPL, it retained the core idea that programs should separate the knowledge of the problem (input data) and ways to solve it (internal rules), assuming the rules are independent to a specific problem.

The base structural elements of LISP are not symbols, but lists (of symbols, of lists, of nothing), and they themselves act as symbols (e.g. the empty list). By manipulating those lists recursively—that it, processing something in terms of itself—Lisp highlights even further this tendency to separate itself from the problem domain, and to exhibit autotelic tenden-

[46] See the Stackoverflow Developer survey `https://insights.stackoverflow.com/survey/2021`

```
  % induce(E,H) <- H is inductive explanation of E
induce(E,H):-induce(E,[],H).

induce(true,H,H):-!.
induce((A,B),H0,H):-!,
        induce(A,H0,H1),
        induce(B,H1,H).
induce(A,H0,H):-
        /* not A=true, not A=(_,_) */
        clause(A,B),
        induce(B,H0,H).
induce(A,H0,H):-
        element((A:-B),H0),     % already assumed
        induce(B,H0,H).         % proceed with body of rule
induce(A,H0,[(A:-B)|H]):-       % A:-B can be added to H
        inducible((A:-B)),      % if it's inducible, and
        not element((A:-B),H0), % if it's not already there
        induce(B,H0,H).         % proceed with body of rule
```

Listing 2.7: Prolog sample source

```
(define (eval-expr env)
(lambda (expr env)
  pmatch expr
    [,x (guard (symbol? x))
      (env x)]
    [(lambda (,x) ,body)
      (lambda (arg)
        (eval-expr body (lambda (y)
                                          (if (eq? x y)
                                                arg
                                                (env y))))))]
      [(,rator ,rand)
        ((eval-expr rator env)
          (eval-expr rand env))]))
```

Listing 2.8: Scheme interpreter written in Scheme

cies. This is facilitated by its atomistic and relational structure: in order to solve what it has do, it evaluates each symbol and traverses a tree-structure in order to find a terminal symbol. Building on these features, Willam Byrd, computer scientst at the University of Utah, describes the following lines of Scheme (a LISP dialect) as "the most beautiful program ever written"[**?**], a Scheme interpreter written in Scheme (**??**):

The beauty of such a program, for Byrd, is the abilty of these fourteen lines to reveal powerful and complex ideas about the nature and process of computation. As an interpreter, this program can take any valid Scheme input and evaluate it correctly. It does so by showing and using ideas of recursion (with calls to `eval-expr`), environment (with the evaluation of the `body`) and lambda functions, as used throughout the program. Following Alan Kay, creator of the Smalltalk programming language, Byrd equates the feelings he experiences in witnessing and pondering the program above to those suggested by Maxwell's equations, which constitute the foundation

of classical electromagnetism ((**??**))[**?**]. In both cases, then, the quality ascribed to those inscriptions come from the simplicity and conciseness of their base elements—making it easy to understand what the symbols mean and how we can compute relevant outputs—all the while implying complex consequences for both, respectively, computer science and electromagnetism.

$$\frac{\partial \mathcal{D}}{\partial t} = \nabla \times \mathcal{H}\frac{\partial \mathcal{B}}{\partial t} = -\nabla \times \mathcal{E}\nabla \cdot \mathcal{B} = 0\nabla \cdot \mathcal{D} = 0 \tag{2.1}$$

With this direct manipulation of symbolic units upon which logic operations can be executed, Lisp became the language of AI, an intelligence conceived first and foremost as abstractly logical, if not outright algebraic. Lisp-based AI was thus working on what Seymour Papert has called "toy problems"—self-referential theorems, children's stories, or simple puzzles or games. In these, the problem and the hardware are reduced from their complexity and multi-consequential relationships to a finite, discreete set of concepts and situations. Confronted to the real world—that is, to commercial exploitation—Lisp's model of symbol manipulation, which proved somewhat successful in those early academic scenarios, started to be applied to issues of natural language understanding and generation in broader applications. Despite disappointing reviews from government reports regarding the effectiveness of these AI techniques, commercial applications flourished, with companies such as Lisp Machines, Inc. and Symbolics offering Lisp-based development and support. Yet, in the 1980s, over-promising and under-delivering of Lisp-based AI applications, which often came from the combinatorial explosion deriving from the list- and tree-based representations, met a dead-end.

"*By making concrete what was formerly abstract, the code for our Lisp interpreter gives us a new way of understanding how Lisp works*", notes Michael Nielsen in his analysis of Lisp, pointing at how, across from the

episteme of computational truths stands the *techne* of implementation[**?**]. The alternative to such abstract, high-level language, is to consider computer science as an engineering discipline, a shift between theoretical programming and practical programming is Edsger Dijkstra's *Notes on Structured Programming*. In it, he points out the limitation of considering programming only as a concrete, bottom-up activity, and the need to formalize it in order to conform to the standards of mathematical logical soundness. Djikstra argues for the superiority of formal methods through the need for a sound theoretical basis when writing software, at a time when the software industry is confronted with its first crisis[47].

Within the software engineering debates, the theory and practice vocabulary had slightly different tones, with terms like "art" and "science" labeling two different mindsets concerning programming[**?**]. As mentioned by Djikstra's example, software engineering suffered from an earlier image of programming as an inherently unmanageable, unsystematic, and artistic activity. There again, many saw programming essentially as an art or craft[**?**], rather than an exact science. Beyond theoretical soundness, computer science engineering concerns itself with efficiency and sustainability, with measurements such as the *O()* notation for program execution complexity. It's not so much about whether it is possible to express an algorithm in a programming language, but whether it is possible to run it effectively, in the contingent environments of hardware, humans and problem domains[48].

This approach, halfway between science and art, is perhaps best seen in Donald Knuth's magnum opus, *The Art of Computer Programming*. In it, Knuth summarizes the findings and achievements of the field of computer science in terms of algorithm design and implementation, in order to "*to organize and summarize what is known about the fast subject of computer methods and to give it firm mathematical and historical foundations.*"[**?**].

[47] See section above.

[48] Notably, algorithms in textbooks tend to be erroneous when used in production; only in five out of twenty are they correct[**?**].

The art of computer programming, according to Knuth, is therefore based on mathematics, but nonetheless different from it insofar as it has to deal with effectiveness, implementation and contingency[49]. In so doing, Knuth takes on an empirical approach to programming, inspecting source code and running software to assess their performance, an approach he first inaugurated for FORTRAN programs when reporting on their concrete effectiveness for the United States Department of Defense[**?**].

Another influential academic textbook dealing not just with computation as a an autotelic phenomenon is *Structure and Interpretation of Computer Programs*, in which the authors insist that source code is *"must be written for people to read, and only incidentally for machines to execute"*[**?**]. Still, even when confronted with implementation and the plurality of contingencies of non-mathematical elements which accompany it, the aesthetic standard in this engineering approach to computer science is the proportionality between the number of lines of code written and the complexity of the idea explained, as we can see in the series *Beautiful Julia Algorithms*[**?**]. For instance, **??** implements the Bubble Sort sorting algorithm in one loop rather than the usual two loops in C, but the simplicity of scientific algorithms is expressed even further in **??** the one-line implementation of a procedure for finding a given element's nearest neighbor, a crucial component of classification systems, including AI systems.

According to Tedre, computer science itself was split in a struggle between correctness and productivity, between theory and implementation, and between formal provability and intuitive art. In the early developments of the field, when machine time was expensive and every instruction cycle counted, efficiency ruled over elegance, but in the end he assesses elegance prevailed, as we will see with the evolution of craft within programming in section 1.4.1 below.

In closing, one should note that the *Art* in the title of the book does not,

[49] *The Art of Computer Programming* involves a hypothetical computer, called MIX, to implement the algorithms discussed.

```
function bubble_sort!(X)
for i in 1:length(X), j in 1:length(X)-i
    if X[j] > X[j+1]
        (X[j+1], X[j]) = (X[j], X[j+1])
    end
end
end
```

Listing 2.9: Bubble Sort implementation in Julia

```
function nearest_neighbor(x', phi, D, dist)
  D[argmin([dist(phi(x), phi(x')) for (x,y) in D])][end]
end
```

Listing 2.10: Nearest neighbor implementation in Julia

however, refer to art as a fine art, or a purely aesthetic object. In a 1974 talk at the ACM, Knuth goes back to its Latin roots, where we find *ars*, *artis* meaning "skill.", noting that the equivalent in Greek being τεχνη, the root of both "technology" and "technique.". This semantic proximity helps him reconcile computation as both a science and an art, the first due to its roots in mathematics and logic, and the second

> because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the state of the Art.[**?**]

When written within an academic and scientific context, we can see how source code tends to align with the aesthetic standards of software development, valuing clarity, reabability, sustainability, in particular through Greg Wilson's work on the development of software development principles through the Software Carpentry and Data Carpentry initiatives. This alignment can also be seen in a conception of computer science as a kind of engineering, as an empirical practice which can and should be formalized in order to become more efficient. There, one can turn to Donald Knuth's *Art of Computer Programming* to see the connections between the academia's best practices and the industry's best practices. And yet, a practical conception of computation, a conception of computation as engineering isn't the only conception of computer science. Within a consideration of computer science as a theoretical and abstract object of study, source code becomes a means of providing insights into more complex abstract concepts, such as the Lisp interpreter, or one-line algorithms implementing foundational algorithms in computer science. It is this relation to a conception of beauty traditionally associated with mathematics and physics

which we will investigate further. But, first, we complete our overview of code practitioners by turning to the software artists, who engage most directly with source code as a written material through source code poetry.

2.1.4 Poets

Source code poetry is a distinct subset of both electronic literature, and software art. One the one hand, electronic literature is a broad field encompassing natural language texts taking full advantage of the dynamic feature of computing to redefine the concept of text, authorship and readership. It encompasses a variety of approaches, including generative literature, interactive fiction, visual poetry, source code poetry and esoteric programming languages, as well as certain aspects of software art. However, we focus here only on the elements of electronic literature which shift their focus from output to input, from executable binary with transformed natural language as a result, to static, latent source.

On the other hand, software art is an umbrella term regrouping artistic practices which engage with the computer on a somewhat direct, material level, whether through hardware[50] or software[51]. This space for artistic experimentation flourished at the dawn of the 20th century, with initiatives such as the *Transmediale* festival's' introduction of a *software art* award between 2001 and 2004, or the *Run_me* festival, from 2002 to 2004. In both of these, the focus is on projects which incorporate standalone programmes or script-based applications which aren not merely functional tools, but in themselves a artistic creation, as decided by the artist, jury and public. These works often bring the normally hidden, basic materials from which digital works are made (e.g. code, circuits and data structures) into the foreground[**?**]. Code poetry is therefore a form a software art whose execution is only secondary to the work's meaning.

Computer poetry, a form based on the playful *détournement* of the computer's constraints, gets closer to our topic insofar as the poems generated represent a more direct application of the rule-based paradigm to the syntactical output of the program. Starting with Christopher Stratchey's love

[50] See Alexei Shuglin's *386 DX* (1998-2013)

[51] See Netochka Nezanova's *Nebula.M81* (1999)

letters (1953), generated (and signed!) by MUC, the Manchester Univac Computer, computer poems are generated by algorithmic processes, and as such rely essentially on this particular feature of programming: laying out rules in order to synthesize syntactically and semantically sound natural language poems. Here, the rules themselves matter as much as the output, as seen by their ratio: a single rule for a seemingly-infinite amount of outputs, these being the only thing shown to the public.

These works and their authors build on a longer tradition of rule-based composition, from Hebrew to the Oulipo and John Cage's indeterministic composition, amongst others[**?**], a tradition in which creativity and beauty can emerge from within a strict framework of formal rules. Nonetheless, the source code to these works is rarely released in conjunction with their output, hinting again at their lesser importance in terms of their overall artistic values. If computer poetry is composed of two texts, a natural-language output and a computer-language source, only the former is actually considered to be poetry, often leaving the latter in its shadow (as well as, sometimes, its programmer, an individual sometimes different from the poet). The poem exists through the code, but isn't exclusively limited to the human-readable version of the code, as it only comes to life and can be fully appreciated, under the poet's terms, once interpreted or compiled. While much has been written on computer poetry, few of those commentaries focus on the soundness and the beauty of the source as an essential component of the work, and only in recent times have we seen the emergence of close-readings of the source of some of these works for their own sake[52]. These do constitute a body of work centered around the concept of generative aesthetics[**?**], in which beauty comes from the unpredictable and somewhat complex interplay of rule-based systems, and whose manifestations encompass not only written works, but games, visual and musical works as well.

[52] See the publications in the field of Critical Code studies, Software studies and Platform studies.

And yet, the approach of code poets is more specific than broad generative aesthetics: it is a matter of exploring the expressive affordances of static source code, and the overlap of machine-meaning and human-meaning essential to the correct functioning of code which acts as a vector for artistic communication. Such an overlap of meaning which appears as a specific feature of source code poetry. In a broad sense, code poetry conflates classical poetry (as strict syntactical and phonetical form, along with poetic expressivity) with computer code, but it is primarily defined by the fact that it does not require the code to be executed, but only to be read by a human. Following the threads laid out by computer poetry, code poetry starts from this essential feature of computers to work with strictly defined formal rules, but departs from it in terms of utility. Source code poems are only functional insofar as they are accepted by the intepreter or compiler of the language in which they are written. To the computer, they are indeed functional, in that they are legal and can be parsed; but they do not do anything of *use*. Such formal compliance is only a pre-requisite, a creative constraint, for their human writers.

Within this reliance on creative constraints provided by a computing environment, the emphasis here is on the act of reading, rather than on the act of deciphering, as we've seen with obfuscated code (and in functional code in general). Source code poems are often easy to read[53], and have an expressive power which operates beyond the common use of programming. Starting from Flusser's approach, I consider poetry as a means to bring concepts into the thinkable, and to crystallize thoughts which are not immediately available to us[**?**]; through various linguistic techniques, poetry allows us to formulate new concepts and ideas, and to shift perspectives.

In their different manifestations, code poems make the boundary between computer meaning and human meaning thinner and thinner, a feature often afforded by the existence and use of higher-level programming

[53] See perl haikus in particular

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

Listing 2.11: japh.pl

languages. Starting with the development of FLOWMATIC in 1955 by Grace Hopper, it was shown that an English-like syntactical system could be used to communicate concepts for the computer to process. From there, programming languages could be described along a gradient, with binary at the lowest end, and natural language (in an overwheling majority, English) at the highest end. This implies that they could be written and read similarly to English, including word order, pronouncation and interpretation, similar to the error-tolerance of human laguages, which doesn't cause the whole communication process to fail whenever a specific word, or a word order isn't understood.

The community of programmers writing in Perl[54] has been one of the most vibrant and productive communities when it comes to code poetry. this use of Perl started in 1990, when the language creator Larry Wall shared some of the poems written in the language, and it gained further exposition through the work of Shannon Hopkins[**?**]. The first Perl poem is considered to have been written by Larry Wall, the creator of the language, in 1990, reproduced in **??**.

Hopkins analyzes the ability of the poem to enable dual understandings of the source: human and machine. Yet, departing from the previous conceptions of source that we've looked at, code poetry does not aim at expressing the same thing to the machine and to the human. The value of a good poem comes from its ability to evoke different concepts for both

[54] See: *perlmonks*, `https://perlmonks.org/`, with the spiritual, devoted and communal undertones that such a name implies.

readers of the source code. As Hopkins puts it:

> In this poem, the q operator causes the next character (in this case a newline) to be taken as a single quote, with the next occurrence of that delimiter taken as the closing quote. Thus, the single-quoted line 'Just another Perl hacker' is printed to STDOUT. In Perl, the "unless $spring" line is mostly filler, since $spring is undefined. In poetical terms, however, "$spring" is very important: haiku poetry is supposed to specify (directly or indirectly) the season of the year. As for the q operator, that reads in English as the word "queue", which makes perfect sense in the context of the poem. [**?**]

In this spirit, additional communities around code poetry have formed, whether in university settings[55], or as independent intiatives[56]. Beyond collections such as threads and hashtags on Twitter[57], code poetry also features artistic publications, such as printed anthologies of code poetry in book form[**?**][**?**]

Yet, code poems from the 20th century aren't the first time where a part of the source code is written exclusively to elicit a human reaction, without any machinic side-effects. One of the earliest of those instances is perhaps the Apollo 11 Guidance Computer (AGC) code, written in 1969[58] in Assembly. Cultural references and jokes are peppered throughout the text as comments, asserting computer code as a means of expression beyond exclusively technical tasks[59], and independent from a single writer's preferences, since they passed multiple checks and review processes to end up in the final, submitted and executed document.

[55] Such as Stanford's Code Poetry Slam, which ran between 2014 and 2016, `https://web.archive.org/web/20161024152353/http://stanford.edu/%7Emkagen/codepoetryslam/`

[56] See the Source Code Poetry event, `https://www.sourcecodepoetry.com/`

[57] See `#SongsInCode` at `https://twitter.com/search?q=%2523SongsInCode`

[58] Hamilton et. al., 1969, retrieved from https://github.com/chrislgarry/Apollo-11

[59] See also: "Crank that wheel", "Burn Baby Burn"

```
663  STODL  CG
664      TTF/8
665  DMP*  VXSC
666            GAINBRAK,1  # NUMERO MYSTERIOSO
667        ANGTERM
668      VAD
669                LAND
670     VSU     RTB
```

Code comments allow a programmer to write in their mother tongue, rather than in the computer's, enabling more syntactic and semantic flexibility, and thus reveal a burgeoning desire for programmers to express themselves within their medium of choice.

At the turn of the 1980s, following the transition to programming from an annex practice to full-fledged discipline and profession, along with the development of more expressive programming languages (e.g. Pascal in 1970, C in 1972), software development has become a larger field, growing exponentially[60], and fostering practices, communities and development styles and patterns[61]. Source code becomes recognized as a text in its own, which can hold qualities and defects of its own, and to which engineering and artistic attention must be paid. No longer a transitional state from formula to binary, it becomes a semantic material, whose layout, organization and syntax are important to the eyes of its writers and readers. Pushing further into the direction of the visual layout of the code, such an endeavour becomes pursued for its own sake, equally important to the need for a program to be functional.

The Obfuscated C Code Contest[62], starting in 1984, is the most popular and oldest organized production of such code, in which programmers sub-

[60] Source: `https://insights.stackoverflow.com/survey/2019#developer-profile-_-years-since-learning-to-code`

[61] From Djikstra's Notes on Structured Programming to Knuth's Literate Programming and Martin's Clean Code

[62] `https://www.ioccc.org`

mit code that is functional and visually meaningful beyond the exclusive standards of well-formatted code. If the source code's meaning was previously entirely subsumed into the output in computer poetry, and if such a meaning existed in parallel in the comments of the AGC routines, pointing at the overlay of computer-related semantics (e.g. line numbers) and human-related semantics (e.g. number of the beast), obfuscated code is a first foray into closely intertwining these separate meanings in the source code itself, making completely transparent, or completely opaque what the code does just by glancing at it.

The above submission to the 1988 IOCCC[63] is a procedure which does exactly what it shows: it deals with a circle. More precisely, it estimates the value of PI by computing its own circumference. While the process is far from being straightforward, relying mainly on bitwise arithmetic operations and a convoluted preprocessor definition, the result is nonetheless very intuitive—the same way that PI is intuitively related to PI. The layout of the code, carefully crafted by introducing whitespace at the necessary locations, doesn't follow any programming practice of indentation, and would probably be useless in any other context, but nonetheless represents another aspect of the *concept* behind the procedure described, not relying on traditional programming syntax[64], but rather on an intuitive, human-specific understanding[65].

Obfuscating practices, beyond their technical necessities (for security and efficiency), are traditionally tied to hacking practices, prominently with one-liners. As such, they rely on the brain-teasing process of deciphering, and on the pleasurable, aesthetic experience of resolving and uniting two parallel meanings: what we see in the code, and what it does[66]. What

[63] Source: `https://web.archive.org/web/20131022114748/http://www0.us.ioccc.org/1988/westley.c`

[64] For such a program, see for instance: `https://crypto.stanford.edu/pbc/notes/pi/code.html`

[65] Concrete poetry also makes such a use of visual cues in traditional literary works.

[66] Also known informally as the "Aha!" moment, crucial in puzzle design.

we focus on here is the aspect of obfuscation which plays with the different layers of meaning: meaning to the computer, meaning to the human, and different ways of representing and communicating this meaning (from uglifying, to consistent formatting, to depicting a circle with dashes and underscores). While the aesthetics at play in hacking will be further explored below, we focus on the fact that obfuscating code practices, beyond hiding the meaning and the intent of the program, also manifest an attempt to represent such a meaning in different ways, leaving aside traditional code-writing practices and suggesting the meaning of the program by challenging the abilities of human interpretation at play in the process of deciphering programs.

Code poetry values code which, while being functional, expresses more than what it does, by entering into *Sprachspiele*, where pronounciation, syntax and semantics are playfully composed in order to match a human poetic form, such as the haiku, or to constitute a linguistic puzzle. Relying on the inherent tendency of source code to remain opaque, obfuscated code contests go a step further by seeing how far can such an opacity be sustained, often involving creative ways.

In this section, we've seen how the set of individuals who write and read code isn't homogeneous. Instead, we can see a significant degree of variation between source code written within the context of software engineering, hacking, scientific research and artistic activity. While none of these areas are exclusive of the others—a software developer by day can hack on the weekend and participate in code poetry events—, they do convey different perspectives on how the code is written, and on how it is evaluated. This cursory introduction to each approach has shown that, for instance, software engineers prefer code which is modular, modifiable, sustainable and understandable by the largest audience of possible contributors, while hackers would favor conciseness over expressivity, and tolerate idiosyncracy for the purpose of immediate, functional efficiency. On

the other hand, scientific programming favors ease of use, accuracy and reproducibility, sometimes overlapping with software engineering, while code poets explore the semantic tension between a human interpretation and the machine interpretation of a given source code.

What we see here are strands of similarity within apparent diversity. The code snippets in this section show that there is a tendency to prefer readability, conciseness, clarity, expressivity and functionality, even though different types of the aforementionned practices would put a different emphasis on each of those aspects. The question we turn to next, then, is to what extent do these different practices of code writing and reading share common judgments regarding their formal properties? Do hackers and poets agree on some value judgment, and how? To start this investigation, we first analyze programmers' discourses in the following section in order to identify concrete categories of formal properties which might enable a source code to be positively valued for its appearance, before we turn to the aesthetic registers code practicionners refer to when discussing beautiful code.

2.2 Ideals of beauty

With this overview of the varieties of practices at play amongst those who read and write source code, we will analyze more thoroughly what are the aesthetic standards most value by those different groups. The aim here is to formalize our understanding of which source code is considered beautiful, and to do so at multiple levels. The goal here is to capture both the specific manifestations of beautiful code as specified and enunciated by programmers, as well as the semantic contexts from which these enunciations originate. What we will see is that, while a set of aesthetic values and a set of aesthetic manifestations can be pinpointed precisely, the domains that are mobilized to justify these values are clearly distinct. To do so, we will introduce the framework of discourse analysis, complemented by a a medium-specific reading through critical code studies and rhetorical code studies on one hand, and the work done by conceptual metaphors on the other.

2.2.1 Introduction to the Methodology

Discourse consists of text, talk and media, which express ways of knowing the world, of experiencing and valuing the world. This work builds on Kintsch and Van Dijk's work on providing tools to analyze an instance of discourse, centered around what constitutes good source code. While discourse analysis is also used critically by unearthing which value judgments occur in power relationships[67], we focus here on aesthetic value judgments, as their are first expressed through language. Of all the different approaches to discourse, the one we focus on here is that of the *pragmatics*. We find this approach particularly fitting through its implication of the *cooperative principle*, in which utterances are ultimately related to one another through communicative cooperation to reveal the intent of

[67] See Diana Mullet on Critical Discourse Analysis[**?**]

the speaker[**?**]. Practically, this means that we assume the position of programmers talking to programmers is cooperative insofar as both speaker and listener want to achieve a similar goal: writing good code. This double understanding—focusing first and foremost on utterances, and then re-examining them within a broader context—will ultimately lead us to examine the influence of the production media (blog post, forums, conferences, text books), of the cultural background (software practices as outlined above as well as additional factors such as skill levels. Our comprehension of those texts, then, will be set in motion by a a dual movement between local, micro-units of meaning and broader, theoretical macro-structure of the text, and linked by acts of co-reference[**?**].

Particular attention will be paid to the difference between intentional and extensional meaning[**?**]. As we will see, some of the texts in our corpus tend to address a particular problem (e.g. on forums, social media or question & answer platforms), or to discuss broader concepts around well-written code. Particularly,

> Figures of speech may attract attention to important concepts, provide more cues for local and global coherence, suggest plausible pragmatic interpretations (e.g., a promise versus a threat), and will in general assign more structure to elements of the semantic representation, so that [meaning] retrieval is easier.[**?**]

Following this idea, we will proceed by examining syntactic markers to deduce overarching concepts at the semantic level. Among those syntactic markers, we include single propositions as explicit predicates regarding source code, lexical fields used in those predicates in order to identify their connotations and denotations, as well as for the tone of the enunciations to identify value judgments. At the semantic level, we will examine the socio-cultural references, the *a priori* knowledge assumed from the audience, as well as the cognitive units which compose the theme of the discourse at hand.

And yet, the discourses we will examine aren't exclusively composed of natural language, but also of source code extracts, resulting in a hybrid between natural and machine syntax within the same discursive artifact.

In line with John Cayley's analytic framework of structure, syntax and vocabulary[**?**], we can nonetheless echo discourse analysis as applied to natural languages. Cayley's framework highlights essential aspect of analysis which applies both to natural languages and source code: that of scales at which aesthetic judgment operates. It also provides a bridge with literature and literary studies without imposing too rigid of a grid. While it does not immediately acknowledge more traditional literary concepts such as fiction, authorship, literarity, etc., it does leave room for these concepts to be taken into account. Particularly, we will see that the concept of authorship—who writes to whom—will be useful in the future.

Finally, our intepretation of the macrostructures described by Kintsch and Van Dijk will rely extensively on the work done by metaphors as the conceptual level, rather than at the strictly linguistic one. Lakoff and Johsnon's seminal work develops a theory of conceptual metaphors by highlighting their essential dependence on pre-existing cognitive structures, which we associate with already-understood concepts. The metaphor maps a source domain (made up of cognitive structure(s)) to a target domain. In the process, they extend the field of applicability of metaphors from the strictly literary to the broadly cultural: metaphors work because each of us has some conception of those domains involved in the metaphorical process. Metaphors rely in part on a static understanding, resulting in a fixed meaning from the application of a given source to a given target. Some of these sources are called *schemas*, and are defined enough to not be mistaken for something else, but broad enough to allow for multiple variants of itself to be applied to various targets[**?**], providing both diversity and reliability. As we will see below, their approach allows us to focus not just on textual objects, but on the vast range of metaphors used specifically in computing-related environments. Given that the source of the metaphor

should be grounded, with as little invariablity as possible, in order to qualify a potentially ill-defined target domain, this provides us with a first foray into the inherent elusiveness and instability of computing when presented to a broader audience.

Going beyond the role of metaphors manifested in expressions such as *the desktop*, *the mouse*, or *the cloud*, we will explore Lakoff's understanding of the specifically poetic metaphor further below as preliminary work to assess the linguistic component of computing—source code. For now, we will pay close attention to what programmers are saying about (beautiful) source code, which metaphors they employ to support these value judgments, and why—focusing first on the metaphors *of* source code, before moving, in the section, to the metaphors *in* source code.

The corpus studied here consists of texts ranging from textbooks and trade manuals to blog posts and online forum discussions. The rationale behind such a broad approach is to constitute a lexical basis that is not just empirical (i.e. taking into account what practicing programmers consider when assessing good code, expressed in the everyday interactions of online forums and blog posts), but also prescriptive. The inclusion of more authoritative sources, such as canonical textbooks or widely-read blog posts from technology investors will allow us to introduce a normative dimension to our research. As this section highlights, there are *specific* ways to write good code, which are echoed both from bottom-up and from top-down perspectives.

2.2.2 Lexical Field in Programmer Discourse

In terms of existing studies of the lexical field programmers use, Erik Pineiro has done significant work in his doctoral thesis. In it, he argues that aesthetics exist from a programmers perspective, decoupled from the final, executable form of the software. While this current study draws on his work, and confirms his findings, it also departs from it in several aspects. First,

Pineiro focuses on a narrower corpus, that of the Slashdot.org[68] forums[**?**] (p. 51). Second, he examines aesthetic judgment from a private perspective of software engineers, separate from other possible aesthetic fields which might enter in dialogue with beautiful code[**?**] (p.52). Finally, his discussion of aesthetics takes place in a broader context of business management and productivity„ while this current study situates itself within aesthetic philosphy, and its implications within how things are considered beautiful.

Clean is the first adjective which stands out as a requirement when assessing beauty in code. It is featured in the title of a series of best-selling trade manuals written by Robert C. Martin and published by Prentice Hall from 2009 to 2021, the full titles of which clearly enunciate their normative aim[69]. Cleanliness, in Martin's terms, is defined by circumlocutions. After asking leading programmers what clean code means to them, he carries on in the volume by providing examples of *how* to achieve clean code, rather that by defining what it is. Nonetheless, some hints can be glimpsed from Ward Cunningham's answer:

> You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.[**?**] p.10

along with Grady Brooch's:

> Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent

[68] `https://slashdot.org`

[69] *Clean Code: A Handbook of Agile Software Craftsmanship*, *The Clean Coder: A Code Of Conduct For Professional Programmers*, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, *Clean Agile: Back to Basics*, *Clean Craftsmanship: Disciplines, Standards, and Ethics*.

> but rather is full of crisp abstractions and straightforward lines of control.[**?**]p.11

Cleanliness is thus tied to expressiveness: clean code is devoid of any extraneous syntactic and semantic symbols (e.g. it does one thing, and one thing well), in order to let the problem at hand appear, with all its implications. Instead, the tool (code, and programming languages) disappear at the syntactic level, to gain expressiveness at the semantic level. Cleanliness is mostly a definition by negation: it states that something is clean if it is free from impurities, blemish, error, etc. As an alternative to this definition by negation, In the spirit of defining by example, trade manuals such as *Clean Code* provide examples on how to move from bad code, to clean code through specific, practical guidelines regarding naming, spacing, class delimitation, etc..

Martin echoes Hunt when he advocates for such a definition of clean as lack of additional information:

> Don't spoil a perfectly good program by overembellishment and over-refinement.[**?**]

This advice to programmers denotes a conception of clean that is not just about removing as much syntactic form as possible, but which also implies a balance. *Overembellishment* implies excess addition, while *over-refinement* implies, on the contrary, excess removal. This normative approach finds its echo in the numerous quotations of Antoine de Saint-Exupéry's comment on aircraft design:

> Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. [**?**][70]

[70] *In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away, when a body has been stripped down to its nakedness.*, translated by Lewis Galantière in the 1939 edition of *Wind, Sand and Stars*

This balance between too much and too little is found in another dichotomy stated by programmers: between simple and clever. Simplicity, argues Jeremy Gibbons, is not only a restraint on the quantity of syntactic tokens (as one could achieve by keeping names short, or aligning indentations), but also a semantic equilibrium at the level of abstracted ideas[**?**]. The balance between breadth and depth of the task of the code, between the precision of a use-case and its generalization, and its leveraging of existing library—i.e. supposedly reliable—code is summed up in a quote by Ralph Waldo Emerson concluding his column:

> We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes.[**?**]

In another paper published by the ACM, Kristiina Karvonen argues for simplicity not just as a design goal, as leveraged by human-computer interface designers, but as a term with a longer history within the tradition of aesthetic philosophy, especially the work of Johann Joachim Winckelmann[**?**]. In particular, she stresses the difficulty "to create significant, that is beautiful works of art with simple means"[**?**]. Here, her correlation between significance and beauty hints at the semantic role of simplicity, as a means to communicate ideas (i.e. *signify*) to an audience.

Precisely, simplicity is correlated with clarity (of meaning); if the former refers mainly to the syntactic component (fewer tokens), it enables the non-obfuscated framing of the ideas at play. One example is given by Dave Bush in a post titled *15 Ways to Write Beautiful Code*:

```
  void SomeMethod(){
    if(x != y){
        //-- stuff
    }
}
```

```
void SomeClearerMethod(){
    if(x == y) return;
    //-- do stuff
}
```

Here, the strive for simplicity leads to removing the brackets, and flipping the boolean check in the if-statement to add a `return`. Even though it is, strictly speaking, more chararcters than the brackets and newline (six characters compared to four), the program becomes clearer by separating the two branching cases inherent to the use of conditional logic, under the form of an if-statement. In the second version, it is made clear that, if a condition *is*, the execution should stop, and any subsequent statement can entirely disregard the existence of the if-statement; in the first version, the condition that *is not* is entangled with code that should be executed, since the existence of the if-statement has to be kept in mind until the closing bracket[**?**].

As a corollary to clarity stands obfuscation. It is the act, either intentional or un-intentional, to complicate the understanding of what a program does by leading the reader astray through a combination of syntactic techniques. In its most widely applied sense, obfuscation is used for practical production purposes: reducing the size of code, and preventing the leak of proprietary information regarding how a system behaves. For instance, the JavaScript source code in **??** is obfuscated through a process called *minification* into the source code in **??**

This process of obfuscation has very clear, quantitative assessment criterias, such as the size of the source code file and cryptographic complexity[**?**]. Nonetheless. obfuscation can also be valued as a positive aesthetic standard, of which the IOCCC is the most institutionalized guarantor, since 1984. These kinds of bofuscations, as Mateas and Montfort analyze, involve the playful exploration of the intertwinings of syntax and semantics, seeing how much one can bend the former without affecting the latter. These textual manipulations, they argue, possess an inherently

```
 import { ref, onMounted, reactive } from 'vue';
import Footer from './components/Footer.vue';
import Header from './components/Header.vue';
import { SyllabusType } from './js/types';

const msg = ref("")
const HOST = import.meta.env.DEV ? "http://localhost:3046" : ""
const syllabi = new Array<SyllabusType>()

let start = () => {
    window.location.href = '/cartridge.html'
}

onMounted(() => {
    fetch(`${HOST}/syllabi/`,
        {
            method: 'GET'
        })
        .then(res => {
            return res.json()
        })
        .then(data => {
            Object.assign(syllabi, JSON.parse(data))
            console.log(syllabi);
            if (syllabi.length == 0)
                msg.value = "No syllabi :("
            else
                msg.value = `There are ${syllabi.length} syllabi.`

        })
        .catch(err => {
            console.error(err)
            msg.value = "Network error :|"
        })
})
```

Listing 2.12: home.js (before minification)

```
import{_ as p,f as m,r as v,g as f,o as l,c as n,a as c,h as e,t as r
    ,b as u,i as b,u as _,F as y,H as g,e as w}from"./Header.js";
    const H={class:"container p-3"},N=e("h1",null,"Home",-1),k={class
    :"syllabi"},x=["href"],B={class:"cta"},F=m({setup(S){const s=v
    (""),d="http://localhost:3046",o=new Array;let h=()=>{window.
    location.href="/cartridge.html"};return f(()=>{fetch(`${d}/
    syllabi/`,{method:"GET"}).then(t=>t.json()).then(t=>{Object.
    assign(o,JSON.parse(t)),console.log(o),o.length==0?s.value="No
    syllabi :(":s.value=`There are ${o.length} syllabi.`}).catch(t=>{
    console.error(t),s.value="Network error :|"})}),(t,i)=>(l(),n(u,
    null,[c(g),e("main",H,[N,e("div",k,[e("div",null,r(s.value),1),e
    ("ul",null,[(l(!0),n(u,null,b(_(o),a=>(l(),n("li",null,[e("div",
    null,[e("a",{href:"/syllabi/"+a.ID},r(a.title),9,x)]),e("div",
    null,r(a.description),1)]))),256))])])]),e("div",B,[e("button",{id
    :"cta-upload",class:"btn btn-primary mb-4 cc-btn",onClick:i[0]||(
    i[0]=a=>_(h)())},"Upload yours!")])]),c(y)],64))}});var O=p(F,[["
    __file","/home/pierre/code/commonsyllabi/viewer/www/src/Home.vue
    "]]);w(O).mount("#app");
```

Listing 2.13: home.js (after minification)

literary quality:

> Obfuscation and weird languages invite us to join programming contexts to the literary contexts that must obviously be considered when evaluating literary code. They also suggest that coding can resist clarity and elegance to strive instead for complexity, can make the familiar unfamiliar, and can wrestle with the language in which it is written, just as much contemporary literature does.[**?**]

An example of such literary connection is given by Noël Arnaud's work *Poèmes Algol*[**?**], in which he uses the constructs of the language Algol 68 in order to evoke in the reader something different than what the program actually does (i.e. fail to execute anything meaningful).

Another insight on simplicity and programming regarding the communication of ideas is hinted at by Richard P. Gabriel in his use of the concept of *compression* in both poetry and programming. In an interview with Janice J. Jeiss, he states:

> I'm thinking about things like simplicity – how easy is it going to be for someone to look at it later? How well is it fulfilling the overall design that I have in mind? How well does it fit into the architecture? If I were writing a very long poem with many parts, I would be thinking, "Okay, how does this piece fit in with the other pieces? How is it part of the bigger picture?". When coding, I'm doing similar things, and if you look at the source code of extremely talented programmers, there's beauty in it. There's a lot of attention to compression, using the underlying programming language in a way that's easy to penetrate. Yes, writing code and writing poetry are similar. [**?**]

Simplicity in programming is presented here as akin to compression in poetry: in the increasing of semantic charge (or significance, in Karvonen's

terms) all the while reducing the syntactic load (or the quantitity of formal tokens in the source code). One of those extraneous loads is explanation:

> When it requires a lot of explanation like that, it's not "beautiful code," but "a clever hack."[**?**]

This answer by Mason Wheeler, posted on the software engineering *Stack Exchange* forum, in response to the question "How can you explain "beautiful code" to a non-programmer?"[**?**], highlights simplicity's opposite, cleverness.

Cleverness is often found, and sometimes derided, in examples of code written by hackers, since it unsettles the balance between precision and generality. Clever code would tend towards exploiting particularities of knowledge of the medium (the code) rather than the goal (the problem). Hillel Wayne presents this snippet of C code as an example of bad clever code:

```
def is_unique(_list):
  return len(set(_list)) == len(_list)
```

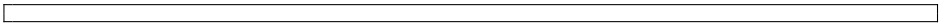
Here, the knowledge of how the `set()` function in Python behaves, is required in order to understand that the `is_unique()` function returns whether all the elements of the given list are unique. A programmer without familiarity with Python would be unable to do so without consulting the Python documentation (i.e. requiring extraneous explanation).

Hillel elaborates on the difference between "bad" clever code[71], which is essentiall read-only due to its idiosyncracy and reliance on tacit knowledge, and "good" clever code, and such distinction corroborates our previous observations regarding beautiful code as a means for expression of the problem domain. His example is that the problem of sorting the roughly

[71] See, for instance, Duff's device, an idiosyncratic and language-specific way to speed up loop unrolling in C. The author himself feels "a combination of pride and revulsion at this discovery"[**?**]

300 million U.S. american citizens by birthdate can be made considerably more efficient by cleverly considering that no U.S. american citizen is older than 120 years.

Meanwhile, cleverness is a valued attribute in the context of hacker code, putting more emphasis on the technical solution than on the problem domain. A salient is example was the 1994 `smr.c` entry to the IOCCC, which aimed at being the smallest self-reproducing program[**?**]. Here is an exact reproduction of the source code:



Consisting of a zero bytes file, `smr.c` provides both a clever understanding and reduction of the problem domain, and a clever understanding of what C compilers would effectively accept or not as a valid program-text[**?**]. Because it has since been banned under the rules of the IOCCC, this clever source code entirely renounces any claim to a more general application, and finds its aesthetic value only within a specific community.

Simplicity, then is the ability to provide code that fits the problem exactly: without being too precise, or too generic, displaying an understanding of and a focus on the application domain, rather than the applied tools, as William J. Mitchell sums it up in his introductory textbook for graphics programming:

> Complex statements have a zen-like reverence for perfect simplicity of expression.[**?**]

We see here the idea of reaching a conceptual revelation through the reduction of complex styntactical assemblages. This strive towards attaining an inverse relationship between the complexity of an idea and the means to express it is contiguous to another related criteria for beautiful source code present in programmers' discourse: elegance.

Chad Perrin, in his article *ITLOG Import: Elegance*, approaches the concept as a negation of the gratuitous, a means to reduce as much as possible

the syntactic footprint while keeping the conceptual footprint intact:

> In pursuing elegance, it is more important to be concise than merely brief. In a general sense, however, brevity of code does account for a decent quick and dirty measure of the potential elegance that can be eked out of a programming language, with length measured in number of distinct syntactic elements rather than the number of bytes of code: don't confuse the number of keystrokes in a variable assignment with the syntactic elements required to accomplish a variable assignment.[**?**]

He also hints at the additional meaningufness of elegance, as he compares it to other aesthetic properties, such as simplciity, complexity or symmetry. If simplicity inhabits a range between too specific and too general, he describes an elegant system as exactly appropriate for the task at hand. While he touches at length on the influence of progamming languages in the possibility to write elegant source code—a question to which we will come back in Chapter 4. Elegance, he says, relies on underlying principles, but is nonetheless subject to its manifestation through a particular, linguistic interface.

This underlying aspect is also present in Bruce McLennan's discussion of the concept. As he approaches it through the dual lens of structural engineering, this indicates that he also considers elegance as a more profound concept which can manifest itself across disciplines, both as a way of making, and as a way of thinking[**?**]. He defines his *Elegance Principle* as:

> Confine your attention to designs that *look* good because they *are* good.[**?**]

Such a definition relies heavily on the sensual component of elegance: while an underlying property of, at least, human activities, it must nonetheless be manifested in some perceptible way. On *Stackexchange*, user

asoundmove corroborates this conception of achieving a simple and clean system where any subsequent modification would lead to a decrease in quality:

> However to me beautiful code must not only be necessary, sufficient and self-explanatory, but it must also subjectively feel perfect & light.[**?**]

Once again connecting simplicity (under the guise of necessity and sufficiency), the perception of elegance is also related to a subjective feeling of adequacy. Paul DiLascia, writing of the Microsoft Developer Network Magazine, illustrates his conception of elegance—as a combination of simplicity, efficiency and brilliance—with recursion[**?**]:

```
int factorial(int n)
{
  return n==0 ? 1 : n * factorial(n-1);
}
```

Recursion, or the technique of defining something in terms of itself, is a positively valued feature of programming[**?**]. In so doing, it minimizes the number of elements at play and constrains the problem domain into a smaller set of moveable pieces. Another example, provided in the same *Stackexchange* discussion is the `quicksort` algorithm, which can be implemented recursively or iteratively, with the former being significanly shorter:

```
  // https://stackoverflow.com/a/12553314/4665412
  public static void recursiveQsort(int[] arr,Integer start, Integer
      end) {
    if (end - start < 2) return; //stop clause
    int p = start + ((end-start)/2);
    p = partition(arr,p,start,end);
    recursiveQsort(arr, start, p);
    recursiveQsort(arr, p+1, end);
}
```

```
public static void iterativeQsort(int[] arr) {
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(0);
    stack.push(arr.length);
    while (!stack.isEmpty()) {
        int end = stack.pop();
        int start = stack.pop();
        if (end - start < 2) continue;
        int p = start + ((end-start)/2);
        p = partition(arr,p,start,end);

        stack.push(p+1);
        stack.push(end);

        stack.push(start);
        stack.push(p);
    }
}
```

To conclude this brief survey on how programmers perceive elegance in source code, we can follow Mahmoud Efatmaneshik and Michael J. Ryan who, in the IEEE Systems journal, offer a defintion of elegance which relies both on a romantic perception—including subjective perception, ”gracefulness”, ”appropriateness” and ”usability”—and practical assessment with terms such as ”simple”, ”neat”, ”parsimonious” or ”efficient”[**?**]. In doing so, they ground source code aesthetics as a resolutely dualistic norm, between subjectivity and objectivity, qualitative and quantitative[72].

And yet, rather than subjectivity and objectivity being opposites, one could also consider them as contingent. Due to the interchangeability in the use of the some of the terms we’ve seen by programmers, both qualitative—in terms of the language used—and quantitative—in terms of the syntax/semantics ration—assessments of source seem to be comple-

[72] A duality we will investigate further through the prism of human and machine understanding in section XXX

mentary in considering it elegant.

Another way to understand what programmers mean when they talk about beautiful code is to look beyond the positive terms used to qualify it (clarity, simplicity, elegance, etc.), and shift our attention to how other terms are used negatively. We have already touched up qualifiers such as clever, or obfuscating, which have ambiguous statuses depending on the community that they're being used in—specifically hackers and artists.

- smelly

- entangled, spaghetti, interdependency (chandra)

- verbose

2.3 Aesthetic domains - 6000

Now that we've done some empirical work, we can try to abstract away a bit and then look into how this relates to existing frameworks of aesthetics.

Now that *we look at proofs, through discourse*, what kind of beauty can we be dealing with?

2.3.1 Literary Beauty

chandra, geek sublime

matz, code as an essay

This second approach contrasts with the functional component of the first one, but nonetheless stands in relationship with it. the creative beauty, by defying traditional beauty standards, does help us highlight, through deviance, what the norm is. These texts on "creative beauty" include the classical perl poetry, code poems, IOCC, code poetry contest, etc.

The poem *Black Perl*, submitted anonymously, is a representative example of the richness of the productions of this community:

```
#!/usr/bin perl
no warnings;

BEFOREHAND: close door, each window & exit; wait until time.
  open spellbook, study, read (scan, $elect, tell us);
write it, print the hex while each watches,
  reverse its, length, write, again;
kill spiders, pop them, chop, split, kill them.
  unlink arms, shift, wait & listen (listening, wait),
  sort the flock (then, warn "the goats" & kill "the sheep");
kill them, dump qualms, shift moralities,
  values aside, each one;
die sheep? die to : reverse { the => system
  ( you accept (reject, respect) ) };
next step,
  kill `the next sacrifice`, each sacrifice,
  wait, redo ritual until "all the spirits are pleased";
do { it => "as they say" }.
  do { it => (*everyone***must***participate***in***forbidden**s*e*
      x*)
+ }.
  return last victim; package body;
exit crypt (time, times & "half a time") & close it,
```

```
    select (quickly) & warn your (next victim);
AFTERWARDS: tell nobody.
  wait, wait until time;
wait until next year, next decade;
sleep, sleep, die yourself,
  die @last
```

The most obvious feature of this code poem is that it can be read by anyone, including by readers with no previous programming experience: each word is valid both as English and as Perl. A second feature is the abundant use of verbs. Perl belongs to a family of programming languages grouped under the *imperative* paradigm, which matches a grammatical mood of natural languages, the *imperative mood*. Such mood emphasizes actions to be take rather than, for instance, descriptions of situations, and thus sets a clear tone for the poem. The fact that Perl is based on stating procedures to be executed and states to be changed creates this feeling of relentless urgency when reading through the poem, a constant need to be taking actions, for things to be changed. Here, the native constraints of the programming language interacts directly with the poetic suggestion of the work in a first way: the nature of Perl is that of giving orders, resulting in a poem which addresses *someone* to execute *something*. Still, Perl's flexibility leaves us wondering as to who and what are concerned by these orders. Is the poem directing its words to itself? To the reader? Is Perl just ever talking exclusively to the computer? This ambiguity of the adressee adds to the ominousness of each verse.

- minimalism - https://vimeo.com/47364930 → concise code is code as litterature, because he says one of the issues is that there are just too many lines of code that one can wrap its head around. so there's a need for shrinking down content

2.3.2 Mathematical beauty

Mostly elegance, could be a good place to work on the distinction between proof and theorem, concept and appearance.

the program is perfect in the mind, and mistakes come when translation occurs into chunks/statements (kinda like a mathematical work: the theorem is there first, and the proof comes laboriously second cf. **"Mathematical Invention", Poincaré, 1914"** → **"The Art of Creation", Arthur Koesler, 1960**)

- Syntactic simplicity, or elegance, measures the number and conciseness of the theory's basic principles. Ontological simplicity, or parsimony, measures the number of kinds of entities postulated by the theory. code is syntactic simplicity because wrngles together complex concepts (e.g. perl, one liners), or code is ontological simplicity, because all is within computation (e.g. lisp)

2.3.3 Architectural beauty

This allows a segue into everyday aesthetics and environmental aesthetics

Completing his definition of elegance, McLennan touches upon the way to assess elegance through visual means in another publication on the more general need for aesthetics[73]. The ability to distinguish an elegant system from an ugly one, then, comes through *practice*, the implications of which we will examine in the last section of this chapter[**?**].

[73] Along with elegance, he includes efficiency and economy as the core aesthetic values of technological activities.

2.4 Craft and beauty - 10p

Paul Graham, LISP programmer, co-founder of the Y Combinator startup accelerator and widely-read blogger[74], highlights the status of programming languages as a medium, in its essay *Hackers and Painters*[**?**]. Particularly, he stresses the materiality of code, depicting hackers as people who:

> are trying to write interesting software, and for whom computers are just a medium of expression, as concrete is for architects or paint for painters.

the ethical aspect of writing beautiful code: well-written, good code is value aesthetically and morally

also combination of hard and soft (cf. green coding guidelines, hun pragmatic programmer)

Now that we've seen how the aesthetic ideals of code borrow from different registers, we need to think about practice, or ways. all of the above can be seen through the prism of craft

2.4.1 Functional beauty

again, pineiro as instrumental goodness

This first approach, by comparing both source and comment at the same time (taking texts which are explicitly described as being beautiful), explicitly highlights the requirements for source code to be beautiful.

There is also an emerging development in aesthetics of integrating function as a criteria for an aesthetic experience.

2.4.2 Embodiment

also note the place and role of tools (IDEs, teletypes, fast compiling, etc.)

[74] And even achieving god-like status among certain circles[**?**]

2.4.3 Tacit knowledge

→ this one actually goes to the next chapter

Chapter 3

Aesthetic ideals in programming practices

The first step in our study of aesthetics in source code aims at identifying the aesthetic ideals that programmers ascribe to source code; that is, the qualifiers and semantic fields that they refer to when discussing program texts. To that end, we first start by clarifying whom we refer to by the term *programmers*, which reveals a multiplicity of practices and purposes, from *ad hoc*, one-line solutions, to printed code and massively-distributed codebases.

We then turn to the kinds of beauty that these programmers aspire to. After expliciting our methodology of discourse analysis, we engage in a review of the various kinds of publications and writings that programmers write, read and refer to when it comes to qualifying their practice. From this will result a clust of adjectives—e.g. *clean*, *simple*, *smelly*—which we argue are used in an aesthetic sense. These will provide a useful framework to inspect, in subsequent chapters, their formal manifestations as typed-out tokens.

From these, we can then move to a description of which aesthetic fields

are being referenced by programmers on a broader level, and consider how multiple kinds of beauties, from literary, to architectural and mathematical conceptions of beauty can overlap and be referred to by the same concrete medium.

Finally, we focus our attention on one of the points of overlap in these different references: the importance of function, craft and knowledge in the disposition and representation of code. We will show how this particular way of working plays a central role in an aesthetic approach to source code and results from the specificity of code as a cognitive material, a specificity we will inquire further in the next chapter.

3.1 The practice of programmers

The history of software development is that of a specific, reserved practice which was born in the aftermath of the second world war, which trickled down to broader and broader audiences at the eve of the twenty-first century. Through this development, multiple ways of doing, approaches and applications have been involved in producing software, resulting in different communities and types of programming. Each of these focus on the description of specific instructions to the computer, but do so with specific characteristics. To this end, we take a socio-historical stance on the field of programming, highlighting how diverse practices emerge at different moments in time, and how they are connected to contemporary technical and economic organizations.

Even though such types of reading and writing source code often overlap with one another, this section will highlight a diversity of more or less loose ways in which code is being written, notably in terms of references—what do they consider good?—, purposes—what do they write for?—and examples—how does their code look like?. First, we take a look at the software industry, to identify professional *software developers*, the large code bases they work on and the specific organizational practices within which they write it. They are responsible for the majority of source code written today, and do so in a professional and productive context, where maintainability, testability and reliability are the main concerns. Then, we turn to a parallel practice, one that is often exhibited by software developers, as they also take on the stance of *hackers*. Disambiguating the term reveals a set of practices where curiosity, cleverness, and idiosyncracy are central, finding unexpected solutions to complex problems, sometimes within artificial constraints. Finally, we look at *scientists* and *poets*. On one end, *scientists* embody a rather academic approach, focusing on abstract concepts such as simplicity, minimalism and elegance; they are often focused on theoretical issues, such as implementation of algorithms and mathemat-

ical models, as well as programming language design. On the other end, *poets* read and write code first and foremost for its textual and semantic qualities, publishing code poems online and in print, and engaging deeply with the range of metaphors allowed by a dynamic linguistic medium such as code.

While this overview encompasses most of the programming practices, we leave aside some approaches to code, mainly because they do not directly engage with the representation of source code as a textual matter. More and more, end-user applications provide the possibility to program in more or less rudimentary ways, something referred to as the "low-code" approach[**?**], and thus contributing to the blurring of boundaries between programmers and non-programmers[1].

3.1.1 Software developers

From local hardware to distributed software

As Niklaus Wirth puts it, *the history of software is the history of growth in complexity*[**?**], while paradoxically, lowering the barrier to entry. As computers' technical abilities in memory managment and processing power increased year on year since the 1950s, the nature of writing instructions shifted accordingly.

In his history of the software industry, Martin Campbell-Kelly traces the development of a discipline through both an economic and a technological lens, and he identifies three consecutive waves in the production of software[**?**]. During the first period, as soon as the 1950s, and continuing throughout the 1960s, software developers were contractors hired to en-

[1] For instance, Microsoft's Visual Basic for Applications, Ableton's Max For Live, MIT's Scrath or McNeel's Grasshopper are all programming frameworks which are not covered within the scope of this study. In the case of VBA and similar office-based high-level programming, it is because such a practice is a highly personal and *ad hoc* one, and therefore is less available for study.

gage directly with a specific computing machine. These computing, mainframes, were large, expensive, and rigid machines, requiring hardware-specific knowledge of the Assembler instruction set specific to each one, since they didn't feature an operating system which could facilitate some of the more basic memory allocation and input/output functions, and thus interoperable program-writing[2]. Two distinct groups of people were involved in the operationalization of such machine: electrical engineers, tasked with designing hardware, and programmers, tasked with implementing the software. While the former historically received the most attention[**?**], the latter was mostly composed of women and, as such, not considered essential in the process[**?**]. At this point, then, programming is closely tied to hardware.

The second period in software development starts in the 1960s, as hardware started to switch from vacuum tubes to transistors and from magnetic core memory to semiconductor memory, making them faster and more capable to handle complex operations. On the software side, the development of several programming languages, such as FORTRAN, LISP and COBOL, started to address the double issue of portability—having a program run unmodified on different machines with different instruction sets—and expressivity—allowing programmers to use high-level, English-like syntax, rather than assembler instruction codes. By then, programmers are no longer theoretically tied to a specific machine, and therefore acquire a certain autonomy, a recognition which culminates in the naming of the field of *software engineering* in 1968 at a NATO conference[3].

The third and final phase that Campbell-Kelly identifies is that of mass-market production: following the advent of the UNIX family of operating systems, the distribution of the C programming language, the wide availability of C compilers, and the appearance of personal computers such as the Commodore 64, Altair and Apple II, software could be effectively en-

[2] One of the first operating systems, MIT's Tape Director, would be only developped in 1956[**?**]

[3] source?

tirely decoupled from hardware[4]. And yet, software immediately enters a crisis, due to software development projects running over time and budget, being unreliable in production and unmaintainable in the long-run. What this highlighted is that the creation of software was no longer a corollary to the design of hardware, and that it would become the main focus of computing as a whole[**?**], and that it should therefore be addressed as such. It is at this time that discussions around best practices in writing source code started to emerge, once the activity of the programmer was no longer restricted to *tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment*[**?**].

This need for a more formal approach to the actual process of programming found one of its most important manifestations in Edsger Djikstra's *Notes on Structured Programming*[**?**]. In it, he argues for moving away from programming as a craft, and towards programming as an organized discipline, with its methodologies and systematization of program construction. Despite its laconic section titles[5], Djikstra nonetheless contributed to establish a more rigorous typology of the constructs required for reliable, provable programs—based on fundamentals such as sequence, selection, iteration and recursion—, and aimed at the formalization of the practice. Along with other developments (such as Hoare's contribution on proper data structuring[**?**], or the rise of object-oriented programming) programming would solidify its foundations as a profession:

> We knew how the nonprofessional programmer could write in an afternoon a three-page program that was supposed to satisfy his needs, but how would the professional programmer design a thirty-page program in such a way that he could really justify his design? What intellectual discipline would be needed? What properties could such a professional programmer demand with

[4]For a more detailed account of the personal computer revolution, see: Cerruzzi, P., A History of Modern Computing[**?**]

[5]See, for instance, Chapter 1: "*On our inability to do much*"

> justification from his programming language, from the formal tool he had to work with? [**?**]

As a result of such interrogations comes an industry-wide search for solutions to the intractable problem of programming: that it is *a technique to manage information which in turn produces information*. To address such a conundrum, a variety of tools, formal methods and management processes enter the market; they aim at acting as a *silver bullet*[**?**], addressing the cascade of potential risks[6] which emerge from large software applications. However, this growth in complexity is also accompanied by a diversification of complexity: as computers become more widely available, and as higher-level programming languages provide more flexibility in their expressive abilities, software engineering is being applied to a variety of domains, each of which might need a specific solution, rather than a generic process. Confronted with this diversity of applications, business literature on software practices flourishes[7], acknowledging that the complexity of software should be tackled at its bottleneck: the reading and writing of source code.

The most recent step in the history of software developers is the popularization of the Internet and of the World Wide Web. Even though the former had existed under as ArpaNet since 1969, the network was only standardized in 1982 and access to it was provided commercially in 1989. Built on top of the Internet, the latter popularized global information exchange, including technical resources to read and write code. Software could now be written by remote individual written on *cloud computing* platforms, shared through public repositories and deployed via containers with a lower barrier to entry than at the time of source code printed in magazines, of overnight batch processing and of non-time-sharing systems.

These software developers have written some of the largest codebases to this date, mainly because this type of activity represents the largest

[6] See `https://catless.ncl.ac.uk/Risks/` for such risks

[7] See Jackson, Principles of Program Design, or Martin, Clean Code, among others.

fraction of programmers. Due to its close ties to commercial distributors, however, source code written in this context often falls under the umbrella of proprietary software, thus made unvailable to the public. Some examples that we include in our corpus are either professional codebases that have been leaked[8], open-source projects that have come out of business environments, such as Soundcloud's Prometheus, Google's TensorFlow or Facebook's React, or large-scale open-source projects which nonetheless adhere to structured programming guidelines, such as Donald Knuth's TeX typesetting system or the Linux Foundation's Linux kernel.

Features of the field

The features of these codebases provide the backdrop to, and start to hint at, the qualities that software developers have come to ascribe to their object of practice. First, the program texts they write are large, much larger than any other codebase included in this study, they often feature multiple programming languages and are highly structured and standardized: each file follows a pre-established convention in programming style, which favors an authoring by multiple programmers without any obvious trace to a single individual authorship. These program texts stand the closest to a programming equivalent of engineering, with its formalisms, standards and usability. From this perspective, the IEEE's Software Engineering Body of Knoweldge (SWEBOK) provides a good starting point to survey the specificities of software developers as source code writers and readers[**?**]; the main features of which include the definition of requirements, design, construction testing and maintenance.

Software requirements are the acknowledgement of the importance of the *problem domain*, the domain to which the software takes its inputs from, and to which it applies its outputs. For instance, software written for a calculator has arithmetic as its problem domain; software written for a learning

[8] Such as the Microsoft Windows XP source code[**?**].

management system has students, faculty, education and courses as its problem domain; software written a banking institution has financial transactions, savings accounts, fraud prevention and credit lines as its problem domain. Requirements in software development aim at formalizing as best as possible the elements that must be used by the software in order to perform a successful computation, and an adequate formalization is a fundamental requirement for a successful software application.

Following the identification and codification of requirements, software design relates to the overall organization of the software components, considered not in their textual implementation, but in their conceptual agency. Usually represented through diagrams or modelling languages, it is concerned with *understanding how a system should be organized and designing the overall structure of that system*[**?**]. Of particular interest is the relationship that is established between software development and architecture. Considered a creative process rather than a strictly rational one, due to the important role of the contexts in which the software will exist (including the problem domain)[**?**], software architecture is considered essential from a top-down perspective, laying down an abstract blueprint for the implementation of a system, as well as form a bottom-up one, representing how the different components of an existing system interact. This apparent contradiction, and the role of architecture in the creative aspects of software development, will be further explored in chapter 2.

Software construction relates to the actual writing of software, and how to do so in the most reliable way possible. The SWEBOK emphasizes first and foremost the need to minimize complexity[9], in anticipation of likely changes and possible reuse by other software systems. Here, the emphasis on engineering is particularly salient: while most would refer to the

[9]Following C. Anthony Hoare's assessment in his Turing Award Lecture that *"there are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."*

creation of software as *writing* software, the IEEE document refers to it as *constructing* software[10]. Coding is only assessed as a practical consideration, one which should not take up the most attention, if the requirements, design and testing steps are satisfyingly implemented. Conversely, a whole field of business litterature[**?**, **?**, **?**, **?**] has focused specifically on the process of writing code, starting from the assumption that:

> We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such details that a machine can execute them is *programming*.[**?**]

As we see, the tension identified by Djikstra some thirty years before between craft and discipline is still alive and well at the beginning of the twenty-first century, even though the focus on code still relates to the need for reliability and maintainability in a maturing industry.

Software maintenance, finally, relates not to the planning or writing of software, but to its reading. Software is notoriously filled with bugs[11] and can, at the same time, be easily fixed while already being in a production environment through software update releases. This means that the lifecycle of a software doesn't stop when then first version is written, but rather when it does not run anymore, and this implies that the nature of software allows for it to be edited across time and space, by other programmers which might not have access to the original group of implementers: consequently, software should be first and foremost understandable—SWEBOK lists the first feature of coding as being *techniques for creating understandable source code*[**?**]. This requirement ties back to one of the main prob-

[10] The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging.[**?**].

[11] McConnell estimates that the industry average is about 15 - 50 errors per 1000 lines of delivered code.[**?**].

lems of software, which is its notorious cognitive complexity, one that remains at any stage of its development.

What does this look like in practice, then? Ideals of clarity, reusability and reliability—and their opposites—can be found in some of the available code bases of professional software. The paradox to be noted here is that, even though software developers write the most code, it is the least accessible online and, as such, the following excerpts are covering the range of commercial software (Microsoft Windows), licensed and publicly available software (Kirby), and an open-source software (Prometheus).

The first excerpts come from the source code for Microsoft Windows 2000, which was made public in 2004. The program text contains 28,655 files, the largest of our corpus, by multiple orders of magnitude, with 13,468,327 combined lines and including more than 10 different file extensions. Taking a closer look at some of these files allow us to identify some of the specific features of code written by software developers, and how they specifically relate to architectural choices, collaborative writing and verbosity.

First, the most striking visual feature of the code is its sheer size. Representing such a versatile and low-level system such as an operating system manifest themselves in files that are often above 2000 lines of code. In order to allow abstraction techniques at a higher-level for the end-developer, the operating system needs to do a significant amount of ”grunt” work, relating directly to the concrete reality of the hardware platform which needs to be operated. For instance, the initialization of strings of text for the namespaces (a technique directly related to the compartmentalization) is necessary, repetitive work which can be represented using a rhythmic visual pattern, such as in `cmdatini.c`:

```
{
  ULONG i;
```

```
    RtlInitUnicodeString( &CmRegistryRootName,
                          CmpRegistryRootString );

    RtlInitUnicodeString( &CmRegistryMachineName,
                          CmpRegistryMachineString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareName,
                          CmpRegistryMachineHardwareString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDescriptionName,
                          CmpRegistryMachineHardwareDescriptionString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDescriptionSystemName,
                          CmpRegistryMachineHardwareDescriptionSystemString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareDeviceMapName,
                          CmpRegistryMachineHardwareDeviceMapString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareResourceMapName,
                          CmpRegistryMachineHardwareResourceMapString );

    RtlInitUnicodeString( &CmRegistryMachineHardwareOwnerMapName,
                          CmpRegistryMachineHardwareOwnerMapString );

    RtlInitUnicodeString( &CmRegistryMachineSystemName,
                          CmpRegistryMachineSystemString );

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSet,
                          CmpRegistryMachineSystemCurrentControlSetString);

    RtlInitUnicodeString( &CmRegistryUserName,
                          CmpRegistryUserString );

    RtlInitUnicodeString( &CmRegistrySystemCloneName,
                          CmpRegistrySystemCloneString );

    RtlInitUnicodeString( &CmpSystemFileName,
                          CmpRegistrySystemFileNameString );

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetEnumName,
                          CmpRegistryMachineSystemCurrentControlSetEnumString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetEnumRootName,
                          CmpRegistryMachineSystemCurrentControlSetEnumRootString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetServices,
                          CmpRegistryMachineSystemCurrentControlSetServicesString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetHardwareProfilesCurrent,
                          CmpRegistryMachineSystemCurrentControlSetHardwareProfilesCurrentString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlClass,
                          CmpRegistryMachineSystemCurrentControlSetControlClassString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlSafeBoot,
                          CmpRegistryMachineSystemCurrentControlSetControlSafeBootString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlSessionManagerMemoryManagement,
                          CmpRegistryMachineSystemCurrentControlSetControlSessionManagerMemoryManagementString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlBootLog,
```

```
                        CmpRegistryMachineSystemCurrentControlSetControlBootLogString);

    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetServicesEventLog,
                        CmpRegistryMachineSystemCurrentControlSetServicesEventLogString);

    RtlInitUnicodeString( &CmSymbolicLinkValueName,
                        CmpSymbolicLinkValueName);

#ifdef _WANT_MACHINE_IDENTIFICATION
    RtlInitUnicodeString( &CmRegistryMachineSystemCurrentControlSetControlBiosInfo,
                        CmpRegistryMachineSystemCurrentControlSetControlBiosInfoString);
#endif

    //
    // Initialize the type names for the hardware tree.
    //

    for (i = 0; i <= MaximumType; i++) {

        RtlInitUnicodeString( &(CmTypeName[i]),
                            CmTypeString[i] );

    }

    //
    // Initialize the class names for the hardware tree.
    //

    for (i = 0; i <= MaximumClass; i++) {

        RtlInitUnicodeString( &(CmClassName[i]),
                            CmClassString[i] );

    }

    return;
}
```

The repetition of the `RtlInitUnicodeString` call in the first part of this listing stands at odds with today's industry-standard practices of not repeating oneself; these would rather point towards the second part of the code, the two `for()` statements. While this practice would only be formalized in Andy Hunt's *The Pragmatic Programmer* in 1999[**?**], the longevity of the windows 2000 operating system and its update cycle would nonetheless have affected how this code is written. The reason why such a repetition applies is the requirement of registering each string with the kernel. Dealing with a different problem domain (kernel instructions) leads to a different kind of expected aesthetics[12].

[12]Effectively, references to `RtlInitUnicodeString()` happen 1580 times across 336

Verbosity, the act of explicitly writing out statements which could be functionally equivalent in a compacted form, is a significant feature of the Windows 2000 codebase, and also relies on a particular semantic environment: that of using the C programming language. As mentioned above, the development of C and UNIX in the 1970s have led to wide adoption of the former, and to some extent of the later (even though Windows is a notable exception since it is an operation system not based on the UNIX tradition). What we see in this listing is the consequence of this development: using a verbose language inevitably leads to a verbose program text, something we will see in the following section on hacker's code and will explore much further in chapter 5.

Another significant aesthetic feature of the Windows 2000 program text is its use of comments, and specifically how those comments representing the multiple, layered authorship. This particular source code is one that is written across individuals and across time, each with presumably its own writing style. Yet, writing source code within a formal organization leads to the adoption of coding styles, with the intent that *all code in any codebase should look like a single person typed it, no matter how many people contributed*[**?**]. For instance, the excerpt in **??** from `jdhuff.c` is a example of such overlapping of styles:

Here, we see three different writings of comments `//`, `/* */` as well as different kinds of capitalizations. Those comments are ignored at compile time: that is, they are not meaningful to the machine, and are only expected to be read by other programmers, primarily programmers belonging to one's organization. This hints at the various origins of the authors, or at the very least at the different moments, and possible mental states of the potential single-author: irregularity in comment writing can connect to irregularities in semantic content of the comments. This irregularity becomes suspicious, and leads to ascribing a different epistemological value to them. If comments aren't procedurally guaranteed to be reflected in the execu-

files

```
  no_more_data:
                  // There should be enough bits still left in the data
                       segment;
                  // if so, just break out of the outer while loop.
                  if (bits_left >= nbits)
                          break;
                        /* Uh-oh.  Report corrupted data to user and
                            stuff zeroes into
                         * the data stream, so that we can produce some
                              kind of image.
                         * Note that this code will be repeated for
                            each byte demanded
                         * for the rest of the segment.  We use a
                            nonvolatile flag to ensure
                         * that only one warning message appears.
                         */
                  if (! *(state->printed_eod_ptr))
                  {
                          WARNMS(state->cinfo, JWRN_HIT_MARKER);
                          *(state->printed_eod_ptr) = TRUE;
                  }
                  c = 0;                        // insert a zero byte
                      into bit buffer
      }
    }

    /* OK, load c into get_buffer */
    get_buffer = (get_buffer << 8) | c;
    bits_left += 8;
  }

  /* Unload the local registers */
  state->next_input_byte = next_input_byte;
  state->bytes_in_buffer = bytes_in_buffer;
  state->get_buffer = get_buffer;
  state->bits_left = bits_left;

  return TRUE;
}
```

Listing 3.1: buffer.c

tion, and outcome, of the program, then one tend to rely on the fact that "The only document that describes your code completely and correctly is the code itself" ([**?**]). This excerpt highlights the constant tension between source code as the canonical origin of knowledge of what the program does and how it does it, while comments refelect the idiosyncratic dimension of all natural-language expressions of human programmers.

And yet, this chronological and interpersonal spread of the program text, as well as organizational practices require the use of comments in order to maintain aesthetic and cognitive coherence in the program, if only by the use of comment headers, which locate a specific file within the greater architectural organization of the program text. For instance, in `pnpenum.c`:

This highlights both the multiple authorship (here, we have one original author and one revisor) as well as the evolution in time of the file: comments are the only manifestation of this layering of revisions which ultimately results in the "final" software[13].

A complementary example is the Kirby CMS[14]. With development starting in 2011, a first release in 2012 and having developed a steady user-base, correlated in Google Trends analytics[15], consistent forum posts[16] and commit history[17], it exhibits a particular set of features, in both its purpose and its source code alike. Kirby is an open-source, developer-first (meaning that it affords direct engagement of other developers with its architecture through modification, extension or partial replacement), single-purpose project. As such, it stands at the other end of the commercial, efficient software spectrum than Microsoft Windows 2000.

The Kirby source code is entirely available online, and the following snip-

[13] The term "final" is in quotes, since the Windows 2000 source contains the mention BUGBUG 7436 times across 2263 files, a testatment to the constant state of unfinishedness that some software might remain in.

[14] Allgeier, Bastian et. al., https://github.com/getkirby/kirby, 2011, consulted in 2022

[15] https://trends.google.com/trends/explore?date=all&q=kirby%20cms

[16] https://forum.getkirby.com

[17] https://github.com/getkirby/kirby

```
  /*++

Copyright (c) 1996 Microsoft Corporation

Module Name:

    enum.c

Abstract:

    This module contains routines to perform device enumeration

Author:

    Shie-Lin Tzong (shielint) Sept. 5, 1996.

Revision History:

    James Cavalaris (t-jcaval) July 29, 1997.
    Added IopProcessCriticalDeviceRoutine.

--*/
```

Listing 3.2: enum.c

```
 /**
 * Enables distinct select clauses.
 *
 * @param bool $distinct
 * @return \Kirby\Database\Query
 */
public function distinct(bool $distinct = true)
{
    $this->distinct = $distinct;
    return $this;
}
```

Listing 3.3: Query.php

pets hint at another set of formal values—conciseness, expliciteness and delimitation. Conciseness can be seen in the lengths of the various components of the code base. For instance, the core of Kirby consists in 248 files, with the longest being `src/Database/Query.php` at 1065 lines, and the shortest being `src/Http/Exceptions/NextRouteException.php`, for an average of 250 lines per file. At the third level, the length of the function bodies is also minimal, ranging from XXXX to XXXX*calculate*. Compared to the leading project in the field, Wordpress.org, which has respectively XXXX *insert analysis here*.

If we look at a typical function declaration within Kirby, we found one such as the distinct setter for Kirby's database:

Out of these 11 lines, the actual functionality of the function is focused on one line, `$this->distinct = $distinct;`. Around it are machine-readable comment snippets, and a function wrapper around the simple variable setting. The textual overhead then comes from the wrapping itself: the actual semantic task of deciding whether a query should be able to include distinct select clauses (as opposed to only allowing join clauses), is now

decoupled from its actual implementation (one could describe to the computer such an ability to generate distinct clauses by assigning it a boolean value, or an integer value, or passing it as an argument for each query, etc.). The quality of this writing, at first verbose, actually lies in its conciseness in relation to the possibilities for extension that such a form of writing allows: the `distinct()` function could, under other circumstances, be implemented differently, and still behave similarly from the perspective of the rest of the program. Additionally, this wrapping enables the setting of default values (here, `true`), a minimal way to catch bugs by always providing a fallback case.

Kirby's source code is also interestingly explicit in comments, and succint in code. Let us take, for instance, from the `Http\\Route` class (**??**).

The 9 lines above the function declaration are machine-readable documentation. It can be parsed by a programmatic system and used as input to generate more classical, human-readable documentation[18]. This is noticeable due to the highly formalized syntax `param string name_of_var`, rather than writing out "this function takes a parameter of type string named `name_of_var`". This does compensate for the tendency of comments to drift out of synchronicity with the code that they are supposed to comment, by tying them back to some computational system to verify its semantic contents, while providing information about the inputs and outputs of the function.

Beyond expliciting inputs and outputs, the second aspect of these comments is targeted at the *how* of the function, helping the reader understand the rationale behind the programmatic process. Comments here aren't cautionary notes on specific edge-cases, as seen in fig. XX above, but rather natural language renderings of the overall rationale of the process. The implication here is to provide a broader, and more explicit understanding of the process of the function, in order to allow for further maintenance,

[18] See, for instance, JavaDocs, or ReadTheDocs

```
 /**
 * Tries to match the path with the regular expression and
 * extracts all arguments for the Route action
 *
 * @param string $pattern
 * @param string $path
 * @return array|false
 */
public function parse(string $pattern, string $path)
{
    // check for direct matches
    if ($pattern === $path) {
        return $this->arguments = [];
    }

    // We only need to check routes with regular expression since all
        others
    // would have been able to be matched by the search for literal
        matches
    // we just did before we started searching.
    if (strpos($pattern, '(') === false) {
        return false;
    }

    // If we have a match we'll return all results
    // from the preg without the full first match.
    if (preg_match('#^' . $this->regex($pattern) . '$#u', $path,
        $parameters)) {
        return $this->arguments = array_slice($parameters, 1);
    }

    return false;
}
```

Listing 3.4: Route.php

extension or modification.

Finally, let us look at a subset of the function, the clause of the third `if statement: (preg_match('#^' . $this->regex($pattern). '$#u', $path, $parameters))`. Without comments, one must realize on cognitive gymnastics and knowledge of the PHP syntax in order to render this as an extraction of all route parameters, implying the removal of the first element of the array. In this sense, then, Kirby's code for parsing an HTTP route is both verbose—in comments—and parsimonious—in code.

What these aesthetic features (small number of files, short file length, short function length) imply is an immediate feeling of *building blocks*. Short, graspable, (re-)usable (conceptual) blocks are made available to the developer directly, as the Kirby ecosystem, like many other open-source projects, relies on contributions from individuals who are not expected to have any other encounter with the project other than, at the bare minimum, the source code itself.

These two examples, Microsoft Windows 2000 and Kirby CMS, highlight some of the presentations of source code—repetition, verbosity, commenting and conciseness—within socio-technical ecosystems made up of hardware, institutional practices ranging from corporate guidelines to open-source contribution, with efficiency and usability remaining at the forefront, both at the result level (the software) and at the process level (the code).

Software developers are a large group of practitioners whose focus on producing effective, reliable and sustainable software, leads them to writing in more-or-less codified manner. Before diving into how such a manner of writing relates to references from architecture and engineering in order to foster simplicity and understandability, in section 2.2, we acknowledge that the bondary between groups of practicioners isn't a clear-cut one, and so we turn to another practice closely linked to professional development—hacking.

3.1.2 Hackers

Popular description of hackers tend to veer towards dishelvement, obsession and esoteric involvement with the machine, accompanied by seemingly-radical value systems and political beliefs. They are often depicted as lonely, obsessed programmers, hyperfocused on the task at hand and able to switch altered mental states as they dive into computational problems, as described by Joseph Weizenbaum in 1976[19]. While some of it is true—for instance, the gender, the compulsive behaviour and the embodied connection to the machine—, hackers nonetheless designate a wider group of people, one which writes code driven by curiosity, cleverness and freedom. Such a group has had a significant influence in the culture of programming, with which it overlaps with the aforementionned values of intellectual challenges.

To hack, in the broadest sense, is to enthusiastically inquire about the possibilities of exploitation of technical systems[20] and, as such, predates the advent of the computer[21]. Computer hacking specifically came to proeminence as early computers started to become available in north-american universities, and coalesced around the Massachussets Institute of Technology's Tech Model Railroad Club[**?**]. Computer hackers were

[19]" Wherever computer centers have become established, that is to say, in countless places in the United States, as well as in virtually all other industrial regions of the world, bright young men of disheveled appearance, often with sunken glowing eyes, can be seen sitting at computer consoles, their arms tensed and waiting to fire their fingers, already poised to strike, at the buttons and keys on which their attention seems to be as riveted as a gambler's on the rolling dice. When not so transfixed, they often sit at tables strewn with computer printouts over which they pore like possessed students of a cabalistic text." ([**?**])

[20]"HACKER [originally, someone who makes furniture with an axe] n. 1. A person who enjoys learning the details of programming systems and how to stretch their capabilities, as opposed to most users who prefer to learn only the minimum necessary. 2. One who programs enthusiastically, or who enjoys programming rather than just theorizing about programming." ([**?**])

[21]See Rosenbaum's report in the October 1971 issue of Esquire for an account of phreaking, computer hacking's immediate predecessor[**?**].

skilled and highly-passionate individuals, with an autotelic inclination to computer systems: these systems mattered most when they referenced themselves, instead of interfacing with a given problem domain. Early hackers were often self-taught, learning to tinker with computers while still in high-school[**?**], and as such tend to exhibit a radical position towards expertise: skill and knowledge aren't derived from academic degrees or credentials, but rather from concrete ability and practical efficacy[22].

The histories of hacking and of software development are deeply intertwined: some of the early hackers worked on software engineering projects—such as the graduate students who wrote the Apollo Guidance Computer routines under Margaret Hamilton—, and then went on to profoundly shape computer infrastructure. Particularly, the development of the UNIX operating system by Dennis Ritchie and Ken Thompson is a key link in connecting hacker practices and professional ones. Developed from 1969 at Bell Labs, AT&T's research division, UNIX was "very close to being the first system under which a programmer could sit down directly at a machine and compose programs on the fly, exploring possibilities and testing while composing" ([**?**]), and was "was brainstormed by three people and implemented by Ken Thompson in two days — on an obsolete machine that had been designed to be a graphics terminal for a 'real' computer." ([**?**]). This was a system which was supporting the free exploration of a system's boundaries central to the hacker culture, and which relied on sharing and circulating source code in order to allow anyone to improve it—in effect, AT&T's inexpensive licensing model until the 1980s, and the use of the C programming language starting from 1977 made it widely available within university settings[23]. UNIX, then, was spreading its design philosophy of clear, modular, simple and transparent design across programming com-

[22] A meritocratic stance which has been analyzed in further in [**?**]

[23] "Unix has become well entrenched in the nation's colleges and universities due to Western Electric's extensive, inexpensive licensing of the system. As a result, many of today's graduating computer scientists are familiar with it." ([**?**])

munities.

The next step in the evolution of hacker culture was to build on this tenet to share source code, and hence to make written software understandable from its textual manifestation. The switch identified in the previous section from hardware being the most important component of a computing system to software had lead manufacturers to stop distributing source code, making proprietary software the norm. Until then, executable software was the consequence of running the source code through a compilation process; around the 1980s, executable software was distributed directly as a binary file, its exact contents an unreadable series of 0s and 1s. As a result to licensing changes of the UNIX system, the GNU project was created, and in its wake the Free Software Foundation, which established the ethical requirement to access the source code of any software.

In the meantime, personal microcomputers came to the market and opened up this ability to tinker and explore computer systems beyond the realms of academic-licensed large mainframes and operating systems. Starting with models such as the Altair 8800, the Apple II and the Commodore 64, as well as with easier, interpreted computer languages such as BASIC, whose first version for such micro-computers was written by Bill Gates, Paul Allen and Monte Davidoff[**?**]. While not considered ”proper” programming by other programmers, the microcomputer revolution allowed for new groups of individuals to explore the interactivity of source code due to their small size when published as type-in listings.

In the wake of the larger free software movement, emerged its less radical counterpart, the open-source movement, as well as its more illegal counterpart, security hacking. The former are usually the types of individuals depicted in mainstream news outlets when they reference hackers: programmers indulging breaching private systems, sometimes in order to cause illegal financial, intelligence or material harm. Security hackers, sometimes called crackers, form a community of practice of their own, with

ideas of superior intelligence, subversion, adventure and stealth[24]. These practices do refer to the original conception of hacking—getting something done quickly, but not well—and include such a practical, efficient appoach into its own set of values and ideals, which are in turn represented in the kinds of program texts[25].

Meanwhile, the open-source movement took the tenets of hacking culture and adapted it to make it more compatible to the requirements of businesses. Open-source can indeed be seen as a compromise between the software industry development practices and the efficacy of free software development. Indeed, beyond the broad values of intellectual curiosity and skillful exploration, free software projects such as the Linux kernel, the Apache server or the OpenSSL project are highly efficient, and used in both commercial, non-commercial, critical and non-critical environments[**?**]. Such an approach sidesteps the political and ethical values held in previous iterations of the hacker ethos in order to focus exclusively on the sharing of source code and open collaboration while remaining within an inquisitive and productive mindframe. With the advent of corporate *hackathons*—short instances of intense collaboration in order to create new software, or new features on a software system—are a particularly salient example of this overlap between industry practices and hacker practices[**?**][26].

Hackers are programmers which, while overlapping with industry-

[24] For a lyrical account of this perception of the hacker ethos, see *The Conscience of a Hacker*, published in Phrack Magazine: " This is our world now... the world of the electron and the switch, the beauty of the baud. We make use of a service already existing without paying for what could be dirt-cheap if it wasn't run by profiteering gluttons, and you call us criminals. We explore... and you call us criminals. We seek after knowledge... and you call us criminals." ([**?**])

[25] Those program texts include computer viruses, worms, trojan horses and injections, amongst others.

[26] Along with the address of the software corporate giant Meta's headquarters: 1, Hacker Way, Menlo Park, CA 94025, U.S.A.

embedded software developers, hold a set of values and ideals regarding the purpose and state of software. Whether academic hackers, amateurs, security hackers or open-source contributors, all are centered around the object of source code as a vehicle for communicating the knowledge held within the software, bypassing auxiliary resources such as natural-language documentation. Those political and ethical values often overlap with aesthetic values associated to how the code exists in its textual manifestation.

Sharp and clever

> To hack is, according to the dictionary, ”to cut irregularly, without skill or definite purpose; to mangle by or as if by repeated strokes of a cutting instrument”. I have already said that the compulsive programmer, or hacker as he calls himself, is usually a superb technician. It seems therefore that he is not ”without skill” as the definition will have it. But the definition fits in the deeper sense that the hacker is ”without definite purpose”: he cannot set before him a clearly defined long-term goal and a plan for achieving it, for he has only technique, not knowledge. He has nothing he can analyze or synthesize; in short, he has nothing to form theories about. His skill is therefore aimless, even disembodied. It is simply not connected with anything other than the instrument on which it may be exercised. His skill is that of a monastic copyist who, though illiterate, is a first rate calligrapher.[**?**]

While he looks down on hackers, perhaps unfairly, from the perspective of a computer scientist whose theoretical work can be achieved only through thought, pen and paper—an approach to programming which we will address in the next section—, the point still remains: hackers are first and foremost technical experts who can get lost into technics for their

own sake. From a broad perspective, hackers therefore seem to exhibit an attitude of *direct engagement*, *subverted use* and *technical excellence*. Gabriella Coleman, in her anthropological study of hackers, *Coding Freedom: The Ethics and Aesthetics of Hacking*, highlights that hackers value both semantic ingenuity[27] and technical wittiness, even though source code written by hackers can take multiple shapes, from one-liners, to whole operating systems, to deliberate decisions to subvert best practices in crucial moments

The *one-liner* is a piece of source code which fits on one line, and is usually intepreted immediately by the operating system. They are terse, concise, and eminently functional: they accomplish one task, and one task only. This binary requirement of functionality (in the strict sense of: "does it do what it's supposed to do, or not?") actually finds a parallel in a different kind of one-liners, the humoristic ones in jokes and stand-up comedy. In this context, the one-liner also exhibits the features of conciseness and impact, with the setup conflated with the punch line, within the same sentence. One-liners are therefore self-contained, whole semantic statements which, through this syntactic compression, appear to be clever—in a similar way that a good joke is labelled clever.

In programming, one-liners have their roots in the philosophy of the UNIX operating system, as well as in the early diffusion of computer programs for personal computer hobbyists[**?**]. On the one side, the Unix philosophy is fundamentally about building simple tools, which all do one thing well, in order to manipulate text streams[**?**]. Each of these tools can then be piped (directing one output of a program-tool into the input of the next program-tool) in order to produce complex results—reminiscing of the orthogonality feature of programming languages. Sometimes openly acknowledged by language designers—such as those of AWK—the goal is to write short programs which shouldn't be longer than one line. Given that

[27] Hackers themselves tend to favor puns—the free software GNU project is a recursive acronym for *GNU's Not UNIX*.

constraint, a hacker's response would then be: how short can you make it?

If writing one-line programs is within the reach of any medium-skilled programmer, writing the shortest of all programs does become a matter of skill, coupled with a compulsivity to reach the most syntactically compressed version. For instance, Guy Steele[28] recalls:

> This may seem like a terrible waste of my effort, but one of the most satisfying moments of my career was when I realized that I had found a way to shave one word off an 11-word program that [Bill] Gosper had written. It was at the expense of a very small amount of execution time, measured in fractions of a machine cycle, but I actually found a way to shorten his code by 1 word and it had only taken me 20 years to do it[**?**].

This sort of compulsive behaviour is also manifested in the practice of *code golf*, challenges in which programmers must solve problems by using the least possible amount of characters—here, the equivalent of *par* in golf would be Kolmogorov complexity[29]. So minimizing program length in relation to the problem complexity is a definite feature of one-liners, since choosing the right programming language for the right taks can lead to a drastic reduction of syntax, while keeping the same expressive and effective power. Tasked with parsing a text file to find which lines had a numerical value greater than 6, Brian Kernighan writes the code in C**??**[30]:

[28] Influential langugage designer, who worked on Scheme, ECMAScript and Java, among others.

[29] See: `https://en.wikipedia.org/wiki/Kolmogorov_complexity`

[30] From Succesful Language Design, Brian Kernighan at the University of Nottingham, `https://www.youtube.com/watch?v=Sg4U4r_AgJU`

The equivalent in AWK, a language he designed, and which he actually refers to in the comment on line 15, presumably as a heuristic as he is writing the function, is see in **??**

The difference is obvious, not just in terms of formal clarity and cleanliness of the surface structure, but also in terms of matching the problem domain: this obviously prints every line in which the third field is greater than 6. The AWK one-liner is more efficient, more understandable because more intuitive, and therefore more beautiful. On the other hand, however, one-liners can be so condensed that they loose all sense of clarity for someone who doesn't have a deep knowledge in the specific language in which it is written. Here is Conway's game of life implemented in one line of APL**??**.

```
[caption={gol.apl}, label={gol.apl}, float, floatplacement=H]
  life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ∘.⌽ ⊂⍵}
```

The obscurity of such a line—due to its highly-unusual character notation, and despite the pre-existing knowledge of the expected output—shows why one-liners are usually highly discouraged for any sort of code which needs to be *worked on* by other programmers. Cleverness in programming indeed tends to be seen as a display of the relationship between the programmer and the machine, rather than between different programmers, and only tangentially about the machine. On the other hand, though, the nature of one-liners makes them highly portable and shareable, infusing them with what one could call *social beauty*. Popular with early personal computer adopters, at a time during which the source code of programs were printed in hobbyist magazines and needed to be input by hand, and during which the potential of computation wasn't as widely distributed amongst society, being able to type just one line in, say, a BASIC interpreter, and resulting in unexpected graphical patterns created a sense of magic and wonder in first-time users—how can so little do so much?[31].

[31]For an example fo such one-liner, see for instance: `https://www.youtube.com/watch?v=0yKwJJw6Abs`

Another example of beautiful code written by hackers is the UNIX operating system, whose inception was an informal side-project spearheaded by Ken Thompson and Dennis Ritchie in the 1970s. As the first portable operating system, UNIX's influence in modern computing was significant, e.g. in showing the viability and efficiency of text-based processing, hierarchical file-system, shell scripting and regular expressions, amongst others. UNIX is also one of the few pieces of production software which has been carefully studied and documented by other developers. One of the most famous examples is *Lions' Commentary on UNIX 6th Edition, with Source Code* by John Lions, an annotated edition of the UNIX source code, which was circulated illegaly in classrooms for twenty years before its official publication was authorized by the copyright owners[**?**]. Coming back to the relationship between architecture and software development, Christopher Alexander asks, in the preface of Richard P. Gabriel's *Patterns of Software*[**?**],

> *For a programmer, what is a comparable goal? What is the Chartres of programming? What task is at a high enough level to inspire people writing programs, to reach for the stars?*

And UNIX might be one of the answers to that question, both by its functionality, and by its conciseness, if not alone by its availability. Another program which qualifies as beautiful hacker code, due both to its technical excellence, unusual solution and open-source availability is the function to compute the inverse square root of a number, a calculation that is particularly necessary in any kind of rendering application (which heavily involves vector arithmetic). It was found in the *Quake* source code, listed in **??** verbatim[32].

[32] The Quake developers aren't the authors of that function—the merit of which goes to Greg Walsh—but are very much the authors of the comments.

What we see here is indeed a combination of the understanding of the problem domain (what's the acceptable result I need to maintain a high-framerate with complex graphics), and how the specific knowledge of computers (i.e. bit-shifting of a float cast as an integer) and the snappiness and wonder of the the comments[33]. The use of `0x5f3759df` is what programmers call a *magic number*, a literal value whose role in the code isn't made clearer by a descriptive variable name. Usually bad practice and highly-discouraged, the magic number here is exactly that: it does makes the magic happen.

Further examples of such intimate knowledge of both the language and the machine can be found in the works of the *demoscene*. Starting in Europe in the 1980s, demos were first short audio-visual programs which were distributed along with *crackware* (pirated software), and to which the names of the people having cracked the software were prepended, in the form of a short animation[**?**]. Due to this very concrete constraint—there was only so much memory left on a pirated disk to fit such a demo—programmers had to work with these limitations in order to produce the most awe-inspiring effects. Indeed, one notable feature of the demoscene is that the output should be as impressive as possible, as an immediate, phenomenological appreciation of the code which could make this happen[34]. Indeed, the `comp.sys.ibm.pc.demos` news group states in their FAQ:

> A Demo is a program that displays a sound, music, and light show, usually in 3D. Demos are very fun to watch, because they seemingly do things that aren't possible on the machine they were programmed on.
>
> Essentially, demos "show off". They do so in usually one, two, or all three of three following methods:

[33] *what the fuck?* indeed

[34] For an example, see *Elevated*, programmed by iq, for a total program size of 4 kilobytes: `https://www.youtube.com/watch?v=jB0vBmiTr6o`

- They show off the computer's hardware abilities (3D objects, multi-channel sound, etc.)
- They show off the creative abilities of the demo group (artists, musicians)
- They show off the programmer's abilities (fast 3D shaded polygons, complex motion, etc.)[**?**]

This showing off, however, does not happen through immediate engagement with the code from the reader's part, but rather in the thorough explanation of the minute functionalities of the demo by its writer. Because of these constraints of size, the demos are usually written in C, openGL, Assembly, or the native language of the targeted hardware. Source code listings of demos also make extensive use of shortcuts and tricks, and little attention is paid to whether or not other humans would directly read the source—the only intended recipient is a very specific machine (e.g. Commodore 64, Amiga VCS, etc.). The release of demos, usually in demoparties, are sometimes accompanied by documentation, write-ups or presentations[35]. However, this presentation format acknowledges a kind of individual, artistic feat, rather than the *egoless programming* lauded by Brooks in professional software development[36].

Pushing the boundaries of how much can be done in how little code, here is a 256-bytes demo resulting in a minute-long music video[**?**] on the Commodore 64. It is firsted listed as a hexademical dump by its author (see **??**)

Even with knowledge of how hexadecimal instructions map to the instruction set of the specific chip of of the Commodore 64 (in this case, the SID 8580), the practical use of these instructions takes productive advan-

[35] You can find *Elevated*'s technical presentation here: `https://www.iquilezles.org/www/material/function2009/function2009.pdf`

[36] In architecture, such technical and artistic feat for its own sake, devoid of any reliable social use, is the pavillion, or the folly.

```
0000000 0801 080d d3ff 329e 3232 0035 0000 4119
0000010 d01c dc00 0000 d011 0be0 3310 610e f590
0000020 0007 1fff 4114 24d5 2515 5315 6115 29d5
0000030 0f1b 13e6 13e6 02d0 20e6 61a9 1c85 20a7
0000040 3fe0 08f0 0c90 114e 6cd0 fffc 6da0 2284
0000050 d784 4b4a a81c 13a5 3029 02d0 1cc6 2fe0
0000060 11f0 02b0 02a2 10c9 09f0 298a aa03 f3b5
0000070 0a85 ab2d b000 b711 b622 9521 a500 4b13
0000080 aa0e f8cb cc86 0749 0b85 13a5 0f29 0fd0
0000090 b8a9 1447 0290 1485 0729 b5aa 85f7 a012
00000a0 b708 910d 880f f910 b7a8 9109 8803 f9d0
00000b0 7e4c 78ea 868e 8e02 d021 4420 a2e5 bdfd
00000c0 0802 0295 d0ca 8ef8 0315 cc4c a900 8d50
00000d0 d011 ad58 dc04 c3a0 1c0d 48d4 044b 30a0
00000e0 188c 71d0 e6cb 71cb 6acb 2005 58a0 d505
00000f0 cb91 dfd0 aa2b 6202 1800 2026 2412 1013
```

Listing 3.5: A Mind is Born

tage of ambivalence and side-effects. In the words of the author, Linus Akesson (emphasis mine):

> We need to tell the VIC chip to look for the video matrix at address $0c00 and the font at $0000. This is done by writing $30 into the bank register ($d018). But this will be done from within the loop, as doing so allows us to use the value $30 for two things. *An important property of this particular bank configuration is that the system stack page becomes part of the font definition*.

Demosceners therefore tend to write beautiful, deliberate code which is hardly understandable by other programmers without explanation, and yet hand-optimized for the machine. This presents a different perspective of the relationship between aesthetics and understanding, in which aesthetics do not support and enable understanding, but rather become a proof of the mastery and skill required to input such a concise input for such an overwhelming output. This shows in an extreme way that one does need a degree of expert knowledge in order to appreciate it—in this sense, aesthetics in programming are shown to be almost often dependent on pre-existing knowledge.

Hackers are then programmers who write code within a variety of settings, from academia to hobbyists through professional software development. Yet, some patterns emerge. First, one can see the emphasis on the *ad hoc*, insofar as choosing "the right tool for the right job"[37] is a requirement for hacker code to be valued positively. This requirement thus involves an awareness of which tool will be the most efficient at getting the task at hand done, with a minimum of effort and minimum of overhead, usually at the expense of sustaining or maintaining the software beyond any immediate needs, making it available or comprehensible neither across time

[37] find citation

nor across individuals, a flavour of *locality*. Second, this need for knowing and understanding one's tools hints at a sort of "materiality" of code, whether instructions land in actual physical memory registers, staying away from abstraction and remaining in "concrete reality" by using magic numbers, or sacrificing semantic clarity in order to *"shave off"* a character or two.

The ideals at play in the writing and reading of source code for hackers is thus centered around specific means of knowledge: knowledge of the hardware, knowledge of the programming language used and knowledge of the tradeoffs acceptable all the while exhibiting an air of playfulness—how far can one go pushing a system's boundaries before it breaks down entirely? How little effort can one put in order to get a maximum outcome? Yet, one aspect that seems to elude hackers in their conception of code is that of conceptual soundness. If code is considered beautiful by attaining previously unthought achievements of purposes with the least amount of resources, rationalization as to why, whether *a prior* or *a posteriori*, does not seem to be a central value. Hackers exhibit tendencies to both *get the job done* and *do it for the sake of doing it*. This behaviour is unlike that of computer and data scientists, and towards whom we turn to next.

If hacking can be considered a practice which deals with the practical intricacies of programming, involving concrete knowledge of the hardware and the language, our third group tends towards the opposite. Programming scientists (of which computer scientists are a subset) engage with progamming first and foremost at the conceptual level, with different locii of implementation: either as a *theory*, or as a *model*.

3.1.3 Scientists

Historically, then, programming emerged as a distinct practice from computing sciences: not all programmers are computer scientists, and not all computer scientists are programmers. Nonetheless, scientists engage with programming and source code in two distinct ways, and as such open up the landscape of the type of code which can be written, and of the standards which support the evaluation of good code. First, we will look at code being written in support of non-specifically computer science research activities and, through it, examine how the specific needs of usability, replicability and data structuring link back to standards of software development. Second, we will inquire specifically into code written by computer scientists, such as programming language designers, and develop how computer implementation exists between the dual scientific pillars of theorization and experimentation[**?**].

Computation as a means

Scientific computing, defined as the use of computation in order to solve non-computer science tasks, started as early as the 1940s and 1950s in the United States, aiding in the design of the first nuclear weapons[**?**]. Essentially, calculations necessary to the verification of theories in disciplines such as physics, chemistry or mathematics were handed over to the computing machines of the time. Beyond the military applications of early computer technology, one can point in particular to Harlow and Fromm's article on *Computer Experiments in Fluid Dynamics*, published in 1965, focusing on how the advent of computing technology would prove to be of great assistance in physics and engineering:

> The fundamental behavior of fluids has traditionally been studied in tanks and wind tunnels. The capacities of the modern computer make it possible to do subtler experiments on the

computer alone.[**?**]

In general, then, computation and computers are perceived as promising automated aids in processing data at a much faster rates than human scientists[**?**]. The remaining issue, then, is to make computers more accessible to scientists which did not have direct exposure to computers. Beyond the unaffordable price point of university mainframes before the personal computer revolution, another vector for simplification and accessibility is the development of adequate programming languages[38]. Developed in 1964 at Dartmouth College, BASIC (Beginners' All-purpose Symbolic Instruction Code) aims at addressing this hurdle by designing "*the world's first user-friendly programming language*"[39]. The intent is to provide non-computer scientists with easy means to instruct the computer on how to instruct the computer to perform computations relevant to their work. Still, computing in the academia will only pick up with the distribution of the multiple versions of the UNIX timesharing system, which allowed multiple users to use a given machine at the same time, and the performance boost provided by the C programming language in the late 1970s.

By the dawn of the 21st century, scientific computing had increased in the scope of its applications (extending beyond engineering and experimental, so-called "hard" sciences, to social sciences and the humanities) as well as in the time spent developing and using software[**?**][**?**], with the main programming languages used being MATLAB, C/C++ and Python. While C and C++'s use can be attributed to their historical standing, popularity amongst computer scientists, efficiency for systems programming and speed of execution, MATLAB and Python offer different perspectives. MATLAB, originally a matrix calculator from the 1970s, became popular with the academic community by providing features such as a reliable way to do

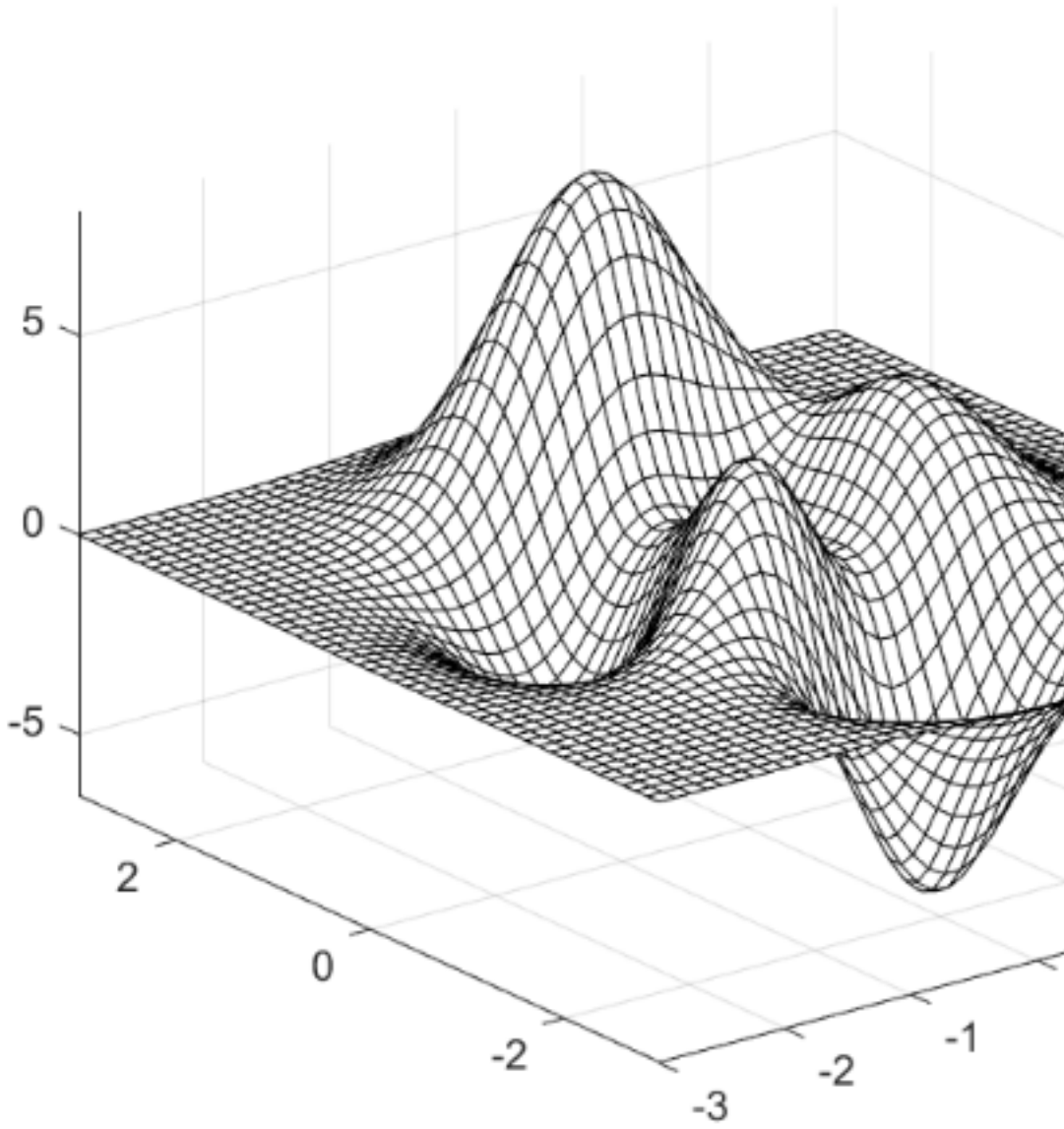
[38] See Chapter 4 for a more complete discussion on programming languages.

[39] `https://web.archive.org/web/20190611180750/https://granitegeek.concordmonitor.com/2019/06/11/finally-a-historical-marker-that-talks-about-something-important/`

```
X = (-3:1/8:3)*ones(49,1);
Y = X';
Z = 3*(1-X).^2.*exp(-(X.^2) - (Y+1).^2) \
- 10*(X/5 - X.^3 - Y.^5).*exp(-X.^2-Y.^2) \
- 1/3*exp(-(X+1).^2 - Y.^2);
mesh(X,Y,Z)
```

Listing 3.6: mesh.m

floating-point arithmetic and a graphical user interface (GUI). Along with its powerful array-manipulation features, the ability to visualize large series of data and plot it on a display contributed to MATLAB's popularity[**?**], features shared with RStudio, a GUI to the R programming language. In **??**, one can see how concise the plotting of a three-dimensional plane is in MATLAB, requiring only one call to `mesh`.



In parallel to MATLAB and R, Python represents the advent of the so-called scripting languages. Scripting languages are programming languages which offer readability and versatility, along with decoupling from

the actual operating system that it is being executed on. System languages, such as C, are designed and used in order to interact directly with the computer hardware, and to constitute data structures from the ground up[**?**]. On the other hand, scripting languages were designed and used in order to connect existing software systems or data sources together, most notably in the early days of shell scripting (such as `Bash`, `sed` or `awk`). Starting with the late 1990s, and the appearance of languages such as Perl[40] and Python[41], scripting languages became more widely used by non-programmers who already had data to work with and needed tools to exploit it. In the following decades, the development of additional scientific libraries such as *SciKit*, *NumPy* for mathematics and numerical work or *NLTK* for language processing and social sciences in Python complemented the language's ease of use by providing manipulation of complex scientifc concepts[**?**], a phenomenon of user-extension which has also been observed in R and MATLAB's ecosystems[**?**].

This steady rise of scientific computing has nonetheless highlighted the apparent lack of quality standards in academic software, and how the lack of value judgments on the software written might impact the reliability of the scientific output. Perhaps the most well-known example of such a lack is the one revealed by the leak of the source code of the Climate Research Unit from the University of East Anglia in 2009[**?**]. In the leak, inline comments of the authors, as well as code reviews of external software developers point out to the CRU leak as being a symptom of the state of academic software. As Professor Darrel Ince stated to the UK Parliamentary Committee in February 2010:

> There is enough evidence for us to regard a lot of scientific software with worry. For example Professor Les Hatton, an international expert in software testing resident in the Universities of Kent and Kingston, carried out an extensive analysis of several

[40] First version developed in 1987 by Larry Wall

[41] First official release in 1991 by Guido Van Rossum

> million lines of scientific code. He showed that the software had an unacceptably high level of detectable inconsistencies.[**?**]

As a response to this realization, the beginning of the 2000s has seen the desire to re-integrate the best practices of software engineering in order to correct scientific software's lack of accuracy[**?**]. Indeed, software engineering, as we've seen above, had developed on their own since its establishment as an independent academic discipline and professional field. Such a split, described by Diane Kelly as a "*chasm*"[**?**] then had to face the different standards to which commercial software and scientific software are subject to. For instance, commercial software must be extensible and performant, two qualities that do not necessarily translate to an academic setting, in which software might be written within a specific, time-constrained, research project, or in which access to computing resources (i.e. supercomputers) might be less of a problem.

Within Landau et. al's conception of the scientific process as the progression from problem to theory, followed by the establishment of a model, the devising of a method, and then on to implemementation and finally to assessment[**?**], code written as academic software is now involved in the latter two stages of method and implementation. Within those two stages, software has to abide by the processes and requirements of scientific research. First and foremost, reproducibility is a core requirement of scientific research in general[42] and bugs in a scientific software system can lead to radically different ouptuts given slightly different input data, while concealing the origin of this radical difference. Good academic code, then, is one which defends actively against these, perhaps to the expense of performance and maintainability. This can be addressed by reliable error-handling, regular assertions of the state of the processed data and extensive unit testing[**?**].

[42] This requirement dates back the 1600s with Robert Boyle and the Invisible College in England[**?**]

Furthermore, a unique aspect of scientific software comes from the lack of clear upfront requirements. Such requirements, in software development, are usually provided ahead of the programming process, and should be as complete as possible. As the activity of scientists is defined by an incomplete understanding of the application domain, requirements tend to emerge as further knowledge is developed and acquired[**?**]. As a result, efforts have been made to familiarize scientists with software development best practices, so that they can implement quality software by themselves. Along with field-specific textbook[43] the most prominent initiative in the field is *Software Carpentry*, a collection of self-learning and teaching resources which aims at implementing software best practices across academia, for scientists and by scientists. Founded by Greg Wilson, the co-editor of *Beautiful Code*, the organization's title refers directly to equivalents in the field of software development[44].

We conclude here on a convergence of quality standards of broad academic software towards the quality standards of commercial software development[45]. And yet, this convergence is due to, as we've seen, a past divergence between computation and science, as computer science worked towards asserting and pursuing its own field of research. As a subset of science, computer science nonetheless holds specific standards, taking software not as a means to an end, but as the end itself.

[43] See *Effective Computation in Physics*[**?**] or *A Primer for Computational Biology*[**?**] as textbooks covering similar software-oriented material from different academic perspectives.

[44] See 1.1.2 *Features of the field* above fo a discussion of the literature on writing good code for software developers)

[45] See Graphbrain at `https://github.com/graphbrain/graphbrain` for such an example. The code's organization and formal features are congruent and on par with commercial software.

Computation as an end

Computer scientists are scientists whose work focuses on computation as a means, rather than as a tool. As such, they study the phenomenon of computation, investigating its nature and effects through th development of a theoretical frameworks around it. Originally derived from computability theory, as a branch of formal mathematical logic, computation emerged as an autonomous field from work in mechanical design and configuration (Ada Lovelace and Charles Babbage), work on circuit and language design (C. S. Pierce, Konrad Zuse and John Von Neumann), work on mathematical foundations (Alan Turing and Alonzo Church), information theory (Claude Shannon), systems theory (Norbert Wiener) and expert systems (John McCarthy and Marvin Minsky)[**?**]. In the middle of such a constellation ranging from mathematical theory to practical electronics, computer science establishes institutional grounding with the inauguration of the first dedicated academic deparment at Purdue University in 1962.

From this multifaceted heritage and academic interdisciplinarity, computer scientists have established the some of the foundations of the field, identifying key areas such as data structures, algorithms and language design as the foundations of the discipline[**?**]. Through the process, the tracing of the "roots" of computation remained a constant debate as to whether computer science exists within the realm of mathematics, of engineering or as a part of the natural sciences. The logico-mathematical model of computer science contends that one can do computer science without a computer, solely armed of a pen and a paper, while the engineering approach of computer science tends to put more practical matters, such as architecture, language design and systems programming at the core of the discipline; both being a way to generate and process information as natural phenomenon[**?**].

The broad difference we can see between these different conceptions of computer science is that of *episteme* and *techne*. On the theoretical and

scientific side, computer science is concerned with the primacy of ideas, rather than of implementation. The quality of a given program is thus deduced from its formal (in the mathematical sense) properties, rather than its formal (in the aesthetic sense) properties. The first manifestations of such a theoretical focus can be found in the Information Processing Language (1956), which was designed and developed originally to prove Bertrand Russell's *Principia Mathematica*. While the IPL, as one of the very first programming languages, influenced the development of multiple subsequent languages, not least of all being LISP, some later languages came to be known as logic programming languages, based on a formal logic syntax of facts, rules and clauses about a given domain and whose correctness can be easily proven (see **??** below for an example of the *Prolog* logic programming language).

Due to its Turing-completeness, one can write programs such as language processing, web applications, cryptography or database programming (using the *Datalog* variant of *Prolog*), but its use remains limited outside of theoretical circles in 2021[46]. Another programming language shares this feature of theoretical soundness faced with a limited range of actual use in production environments, Lisp—*LISt Processor*—designed to process lists. It was developed in 1958, the year of the Dartmouth workshop, on Artificial Intelligence by its organizator, John McCarthy. Inheriting from IPL, it retained the core idea that programs should separate the knowledge of the problem (input data) and ways to solve it (internal rules), assuming the rules are independent to a specific problem.

The base structural elements of LISP are not symbols, but lists (of symbols, of lists, of nothing), and they themselves act as symbols (e.g. the empty list). By manipulating those lists recursively—that it, processing something in terms of itself—Lisp highlights even further this tendency to separate itself from the problem domain, and to exhibit autotelic tenden-

[46] See the Stackoverflow Developer survey `https://insights.stackoverflow.com/survey/2021`

```
  % induce(E,H) <- H is inductive explanation of E
induce(E,H):-induce(E,[],H).

induce(true,H,H):-!.
induce((A,B),H0,H):-!,
        induce(A,H0,H1),
        induce(B,H1,H).
induce(A,H0,H):-
        /* not A=true, not A=(_,_) */
        clause(A,B),
        induce(B,H0,H).
induce(A,H0,H):-
        element((A:-B),H0),     % already assumed
        induce(B,H0,H).         % proceed with body of rule
induce(A,H0,[(A:-B)|H]):-       % A:-B can be added to H
        inducible((A:-B)),      % if it's inducible, and
        not element((A:-B),H0), % if it's not already there
        induce(B,H0,H).         % proceed with body of rule
```

Listing 3.7: Prolog sample source

```
(define (eval-expr env)
(lambda (expr env)
  pmatch expr
    [,x (guard (symbol? x))
      (env x)]
    [(lambda (,x) ,body)
      (lambda (arg)
        (eval-expr body (lambda (y)
                                          (if (eq? x y)
                                                arg
                                                (env y))))))]
      [(,rator ,rand)
        ((eval-expr rator env)
          (eval-expr rand env))]))
```

Listing 3.8: Scheme interpreter written in Scheme

cies. This is facilitated by its atomistic and relational structure: in order to solve what it has do, it evaluates each symbol and traverses a tree-structure in order to find a terminal symbol. Building on these features, Willam Byrd, computer scientst at the University of Utah, describes the following lines of Scheme (a LISP dialect) as "the most beautiful program ever written"[**?**], a Scheme interpreter written in Scheme (**??**):

The beauty of such a program, for Byrd, is the abilty of these fourteen lines to reveal powerful and complex ideas about the nature and process of computation. As an interpreter, this program can take any valid Scheme input and evaluate it correctly. It does so by showing and using ideas of recursion (with calls to `eval-expr`), environment (with the evaluation of the `body`) and lambda functions, as used throughout the program. Following Alan Kay, creator of the Smalltalk programming language, Byrd equates the feelings he experiences in witnessing and pondering the program above to those suggested by Maxwell's equations, which constitute the foundation

of classical electromagnetism ((**??**))[**?**]. In both cases, then, the quality ascribed to those inscriptions come from the simplicity and conciseness of their base elements—making it easy to understand what the symbols mean and how we can compute relevant outputs—all the while implying complex consequences for both, respectively, computer science and electromagnetism.

$$\frac{\partial \mathcal{D}}{\partial t} = \nabla \times \mathcal{H}\frac{\partial \mathcal{B}}{\partial t} = -\nabla \times \mathcal{E}\nabla \cdot \mathcal{B} = 0\nabla \cdot \mathcal{D} = 0 \tag{3.1}$$

With this direct manipulation of symbolic units upon which logic operations can be executed, Lisp became the language of AI, an intelligence conceived first and foremost as abstractly logical, if not outright algebraic. Lisp-based AI was thus working on what Seymour Papert has called "toy problems"—self-referential theorems, children's stories, or simple puzzles or games. In these, the problem and the hardware are reduced from their complexity and multi-consequential relationships to a finite, discreete set of concepts and situations. Confronted to the real world—that is, to commercial exploitation—Lisp's model of symbol manipulation, which proved somewhat successful in those early academic scenarios, started to be applied to issues of natural language understanding and generation in broader applications. Despite disappointing reviews from government reports regarding the effectiveness of these AI techniques, commercial applications flourished, with companies such as Lisp Machines, Inc. and Symbolics offering Lisp-based development and support. Yet, in the 1980s, over-promising and under-delivering of Lisp-based AI applications, which often came from the combinatorial explosion deriving from the list- and tree-based representations, met a dead-end.

"*By making concrete what was formerly abstract, the code for our Lisp interpreter gives us a new way of understanding how Lisp works*", notes Michael Nielsen in his analysis of Lisp, pointing at how, across from the

episteme of computational truths stands the *techne* of implementation[**?**]. The alternative to such abstract, high-level language, is to consider computer science as an engineering discipline, a shift between theoretical programming and practical programming is Edsger Dijkstra's *Notes on Structured Programming*. In it, he points out the limitation of considering programming only as a concrete, bottom-up activity, and the need to formalize it in order to conform to the standards of mathematical logical soundness. Djikstra argues for the superiority of formal methods through the need for a sound theoretical basis when writing software, at a time when the software industry is confronted with its first crisis[47].

Within the software engineering debates, the theory and practice vocabulary had slightly different tones, with terms like "art" and "science" labeling two different mindsets concerning programming[**?**]. As mentioned by Djikstra's example, software engineering suffered from an earlier image of programming as an inherently unmanageable, unsystematic, and artistic activity. There again, many saw programming essentially as an art or craft[**?**], rather than an exact science. Beyond theoretical soundness, computer science engineering concerns itself with efficiency and sustainability, with measurements such as the *O()* notation for program execution complexity. It's not so much about whether it is possible to express an algorithm in a programming language, but whether it is possible to run it effectively, in the contingent environments of hardware, humans and problem domains[48].

This approach, halfway between science and art, is perhaps best seen in Donald Knuth's magnum opus, *The Art of Computer Programming*. In it, Knuth summarizes the findings and achievements of the field of computer science in terms of algorithm design and implementation, in order to "*to organize and summarize what is known about the fast subject of computer methods and to give it firm mathematical and historical foundations.*"[**?**].

[47] See section above.

[48] Notably, algorithms in textbooks tend to be erroneous when used in production; only in five out of twenty are they correct[**?**].

The art of computer programming, according to Knuth, is therefore based on mathematics, but nonetheless different from it insofar as it has to deal with effectiveness, implementation and contingency[49]. In so doing, Knuth takes on an empirical approach to programming, inspecting source code and running software to assess their performance, an approach he first inaugurated for FORTRAN programs when reporting on their concrete effectiveness for the United States Department of Defense[**?**].

Another influential academic textbook dealing not just with computation as a an autotelic phenomenon is *Structure and Interpretation of Computer Programs*, in which the authors insist that source code is *"must be written for people to read, and only incidentally for machines to execute"*[**?**]. Still, even when confronted with implementation and the plurality of contingencies of non-mathematical elements which accompany it, the aesthetic standard in this engineering approach to computer science is the proportionality between the number of lines of code written and the complexity of the idea explained, as we can see in the series *Beautiful Julia Algorithms*[**?**]. For instance, **??** implements the Bubble Sort sorting algorithm in one loop rather than the usual two loops in C, but the simplicity of scientific algorithms is expressed even further in **??** the one-line implementation of a procedure for finding a given element's nearest neighbor, a crucial component of classification systems, including AI systems.

According to Tedre, computer science itself was split in a struggle between correctness and productivity, between theory and implementation, and between formal provability and intuitive art. In the early developments of the field, when machine time was expensive and every instruction cycle counted, efficiency ruled over elegance, but in the end he assesses elegance prevailed, as we will see with the evolution of craft within programming in section 1.4.1 below.

In closing, one should note that the *Art* in the title of the book does not,

[49] *The Art of Computer Programming* involves a hypothetical computer, called MIX, to implement the algorithms discussed.

```
function bubble_sort!(X)
for i in 1:length(X), j in 1:length(X)-i
    if X[j] > X[j+1]
        (X[j+1], X[j]) = (X[j], X[j+1])
    end
end
end
```

Listing 3.9: Bubble Sort implementation in Julia

```
function nearest_neighbor(x', phi, D, dist)
  D[argmin([dist(phi(x), phi(x')) for (x,y) in D])][end]
end
```

Listing 3.10: Nearest neighbor implementation in Julia

however, refer to art as a fine art, or a purely aesthetic object. In a 1974 talk at the ACM, Knuth goes back to its Latin roots, where we find *ars*, *artis* meaning "skill.", noting that the equivalent in Greek being τεχνη, the root of both "technology" and "technique.". This semantic proximity helps him reconcile computation as both a science and an art, the first due to its roots in mathematics and logic, and the second

> because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the state of the Art.[**?**]

When written within an academic and scientific context, we can see how source code tends to align with the aesthetic standards of software development, valuing clarity, reabability, sustainability, in particular through Greg Wilson's work on the development of software development principles through the Software Carpentry and Data Carpentry initiatives. This alignment can also be seen in a conception of computer science as a kind of engineering, as an empirical practice which can and should be formalized in order to become more efficient. There, one can turn to Donald Knuth's *Art of Computer Programming* to see the connections between the academia's best practices and the industry's best practices. And yet, a practical conception of computation, a conception of computation as engineering isn't the only conception of computer science. Within a consideration of computer science as a theoretical and abstract object of study, source code becomes a means of providing insights into more complex abstract concepts, such as the Lisp interpreter, or one-line algorithms implementing foundational algorithms in computer science. It is this relation to a conception of beauty traditionally associated with mathematics and physics

which we will investigate further. But, first, we complete our overview of code practitioners by turning to the software artists, who engage most directly with source code as a written material through source code poetry.

3.1.4 Poets

Source code poetry is a distinct subset of both electronic literature, and software art. One the one hand, electronic literature is a broad field encompassing natural language texts taking full advantage of the dynamic feature of computing to redefine the concept of text, authorship and readership. It encompasses a variety of approaches, including generative literature, interactive fiction, visual poetry, source code poetry and esoteric programming languages, as well as certain aspects of software art. However, we focus here only on the elements of electronic literature which shift their focus from output to input, from executable binary with transformed natural language as a result, to static, latent source.

On the other hand, software art is an umbrella term regrouping artistic practices which engage with the computer on a somewhat direct, material level, whether through hardware[50] or software[51]. This space for artistic experimentation flourished at the dawn of the 20th century, with initiatives such as the *Transmediale* festival's' introduction of a *software art* award between 2001 and 2004, or the *Run_me* festival, from 2002 to 2004. In both of these, the focus is on projects which incorporate standalone programmes or script-based applications which aren not merely functional tools, but in themselves a artistic creation, as decided by the artist, jury and public. These works often bring the normally hidden, basic materials from which digital works are made (e.g. code, circuits and data structures) into the foreground[**?**]. Code poetry is therefore a form a software art whose execution is only secondary to the work's meaning.

Computer poetry, a form based on the playful *détournement* of the computer's constraints, gets closer to our topic insofar as the poems generated represent a more direct application of the rule-based paradigm to the syntactical output of the program. Starting with Christopher Stratchey's love

[50] See Alexei Shuglin's *386 DX* (1998-2013)

[51] See Netochka Nezanova's *Nebula.M81* (1999)

letters (1953), generated (and signed!) by MUC, the Manchester Univac Computer, computer poems are generated by algorithmic processes, and as such rely essentially on this particular feature of programming: laying out rules in order to synthesize syntactically and semantically sound natural language poems. Here, the rules themselves matter as much as the output, as seen by their ratio: a single rule for a seemingly-infinite amount of outputs, these being the only thing shown to the public.

These works and their authors build on a longer tradition of rule-based composition, from Hebrew to the Oulipo and John Cage's indeterministic composition, amongst others[**?**], a tradition in which creativity and beauty can emerge from within a strict framework of formal rules. Nonetheless, the source code to these works is rarely released in conjunction with their output, hinting again at their lesser importance in terms of their overall artistic values. If computer poetry is composed of two texts, a natural-language output and a computer-language source, only the former is actually considered to be poetry, often leaving the latter in its shadow (as well as, sometimes, its programmer, an individual sometimes different from the poet). The poem exists through the code, but isn't exclusively limited to the human-readable version of the code, as it only comes to life and can be fully appreciated, under the poet's terms, once interpreted or compiled. While much has been written on computer poetry, few of those commentaries focus on the soundness and the beauty of the source as an essential component of the work, and only in recent times have we seen the emergence of close-readings of the source of some of these works for their own sake[52]. These do constitute a body of work centered around the concept of generative aesthetics[**?**], in which beauty comes from the unpredictable and somewhat complex interplay of rule-based systems, and whose manifestations encompass not only written works, but games, visual and musical works as well.

[52] See the publications in the field of Critical Code studies, Software studies and Platform studies.

And yet, the approach of code poets is more specific than broad generative aesthetics: it is a matter of exploring the expressive affordances of static source code, and the overlap of machine-meaning and human-meaning essential to the correct functioning of code which acts as a vector for artistic communication. Such an overlap of meaning which appears as a specific feature of source code poetry. In a broad sense, code poetry conflates classical poetry (as strict syntactical and phonetical form, along with poetic expressivity) with computer code, but it is primarily defined by the fact that it does not require the code to be executed, but only to be read by a human. Following the threads laid out by computer poetry, code poetry starts from this essential feature of computers to work with strictly defined formal rules, but departs from it in terms of utility. Source code poems are only functional insofar as they are accepted by the intepreter or compiler of the language in which they are written. To the computer, they are indeed functional, in that they are legal and can be parsed; but they do not do anything of *use*. Such formal compliance is only a pre-requisite, a creative constraint, for their human writers.

Within this reliance on creative constraints provided by a computing environment, the emphasis here is on the act of reading, rather than on the act of deciphering, as we've seen with obfuscated code (and in functional code in general). Source code poems are often easy to read[53], and have an expressive power which operates beyond the common use of programming. Starting from Flusser's approach, I consider poetry as a means to bring concepts into the thinkable, and to crystallize thoughts which are not immediately available to us[**?**]; through various linguistic techniques, poetry allows us to formulate new concepts and ideas, and to shift perspectives.

In their different manifestations, code poems make the boundary between computer meaning and human meaning thinner and thinner, a feature often afforded by the existence and use of higher-level programming

[53] See perl haikus in particular

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

Listing 3.11: japh.pl

languages. Starting with the development of FLOWMATIC in 1955 by Grace Hopper, it was shown that an English-like syntactical system could be used to communicate concepts for the computer to process. From there, programming languages could be described along a gradient, with binary at the lowest end, and natural language (in an overwheling majority, English) at the highest end. This implies that they could be written and read similarly to English, including word order, pronouncation and interpretation, similar to the error-tolerance of human laguages, which doesn't cause the whole communication process to fail whenever a specific word, or a word order isn't understood.

The community of programmers writing in Perl[54] has been one of the most vibrant and productive communities when it comes to code poetry. this use of Perl started in 1990, when the language creator Larry Wall shared some of the poems written in the language, and it gained further exposition through the work of Shannon Hopkins[**?**]. The first Perl poem is considered to have been written by Larry Wall, the creator of the language, in 1990, reproduced in **??**.

Hopkins analyzes the ability of the poem to enable dual understandings of the source: human and machine. Yet, departing from the previous conceptions of source that we've looked at, code poetry does not aim at expressing the same thing to the machine and to the human. The value of a good poem comes from its ability to evoke different concepts for both

[54] See: *perlmonks*, `https://perlmonks.org/`, with the spiritual, devoted and communal undertones that such a name implies.

readers of the source code. As Hopkins puts it:

> In this poem, the q operator causes the next character (in this case a newline) to be taken as a single quote, with the next occurrence of that delimiter taken as the closing quote. Thus, the single-quoted line 'Just another Perl hacker' is printed to STDOUT. In Perl, the "unless $spring" line is mostly filler, since $spring is undefined. In poetical terms, however, "$spring" is very important: haiku poetry is supposed to specify (directly or indirectly) the season of the year. As for the q operator, that reads in English as the word "queue", which makes perfect sense in the context of the poem. [**?**]

In this spirit, additional communities around code poetry have formed, whether in university settings[55], or as independent intiatives[56]. Beyond collections such as threads and hashtags on Twitter[57], code poetry also features artistic publications, such as printed anthologies of code poetry in book form[**?**][**?**]

Yet, code poems from the 20th century aren't the first time where a part of the source code is written exclusively to elicit a human reaction, without any machinic side-effects. One of the earliest of those instances is perhaps the Apollo 11 Guidance Computer (AGC) code, written in 1969[58] in Assembly. Cultural references and jokes are peppered throughout the text as comments, asserting computer code as a means of expression beyond exclusively technical tasks[59], and independent from a single writer's preferences, since they passed multiple checks and review processes to end up in the final, submitted and executed document.

[55] Such as Stanford's Code Poetry Slam, which ran between 2014 and 2016, `https://web.archive.org/web/20161024152353/http://stanford.edu/%7Emkagen/codepoetryslam/`

[56] See the Source Code Poetry event, `https://www.sourcecodepoetry.com/`

[57] See `#SongsInCode` at `https://twitter.com/search?q=%2523SongsInCode`

[58] Hamilton et. al., 1969, retrieved from https://github.com/chrislgarry/Apollo-11

[59] See also: "Crank that wheel", "Burn Baby Burn"

```
663  STODL  CG
664      TTF/8
665  DMP*  VXSC
666            GAINBRAK,1  # NUMERO MYSTERIOSO
667        ANGTERM
668      VAD
669                LAND
670     VSU     RTB
```

Code comments allow a programmer to write in their mother tongue, rather than in the computer's, enabling more syntactic and semantic flexibility, and thus reveal a burgeoning desire for programmers to express themselves within their medium of choice.

At the turn of the 1980s, following the transition to programming from an annex practice to full-fledged discipline and profession, along with the development of more expressive programming languages (e.g. Pascal in 1970, C in 1972), software development has become a larger field, growing exponentially[60], and fostering practices, communities and development styles and patterns[61]. Source code becomes recognized as a text in its own, which can hold qualities and defects of its own, and to which engineering and artistic attention must be paid. No longer a transitional state from formula to binary, it becomes a semantic material, whose layout, organization and syntax are important to the eyes of its writers and readers. Pushing further into the direction of the visual layout of the code, such an endeavour becomes pursued for its own sake, equally important to the need for a program to be functional.

The Obfuscated C Code Contest[62], starting in 1984, is the most popular and oldest organized production of such code, in which programmers sub-

[60] Source: `https://insights.stackoverflow.com/survey/2019#developer-profile-_-years-since-learning-to-code`

[61] From Djikstra's Notes on Structured Programming to Knuth's Literate Programming and Martin's Clean Code

[62] `https://www.ioccc.org`

mit code that is functional and visually meaningful beyond the exclusive standards of well-formatted code. If the source code's meaning was previously entirely subsumed into the output in computer poetry, and if such a meaning existed in parallel in the comments of the AGC routines, pointing at the overlay of computer-related semantics (e.g. line numbers) and human-related semantics (e.g. number of the beast), obfuscated code is a first foray into closely intertwining these separate meanings in the source code itself, making completely transparent, or completely opaque what the code does just by glancing at it.

The above submission to the 1988 IOCCC[63] is a procedure which does exactly what it shows: it deals with a circle. More precisely, it estimates the value of PI by computing its own circumference. While the process is far from being straightforward, relying mainly on bitwise arithmetic operations and a convoluted preprocessor definition, the result is nonetheless very intuitive—the same way that PI is intuitively related to PI. The layout of the code, carefully crafted by introducing whitespace at the necessary locations, doesn't follow any programming practice of indentation, and would probably be useless in any other context, but nonetheless represents another aspect of the *concept* behind the procedure described, not relying on traditional programming syntax[64], but rather on an intuitive, human-specific understanding[65].

Obfuscating practices, beyond their technical necessities (for security and efficiency), are traditionally tied to hacking practices, prominently with one-liners. As such, they rely on the brain-teasing process of deciphering, and on the pleasurable, aesthetic experience of resolving and uniting two parallel meanings: what we see in the code, and what it does[66]. What

[63] Source: `https://web.archive.org/web/20131022114748/http://www0.us.ioccc.org/1988/westley.c`

[64] For such a program, see for instance: `https://crypto.stanford.edu/pbc/notes/pi/code.html`

[65] Concrete poetry also makes such a use of visual cues in traditional literary works.

[66] Also known informally as the "Aha!" moment, crucial in puzzle design.

we focus on here is the aspect of obfuscation which plays with the different layers of meaning: meaning to the computer, meaning to the human, and different ways of representing and communicating this meaning (from uglifying, to consistent formatting, to depicting a circle with dashes and underscores). While the aesthetics at play in hacking will be further explored below, we focus on the fact that obfuscating code practices, beyond hiding the meaning and the intent of the program, also manifest an attempt to represent such a meaning in different ways, leaving aside traditional code-writing practices and suggesting the meaning of the program by challenging the abilities of human interpretation at play in the process of deciphering programs.

Code poetry values code which, while being functional, expresses more than what it does, by entering into *Sprachspiele*, where pronounciation, syntax and semantics are playfully composed in order to match a human poetic form, such as the haiku, or to constitute a linguistic puzzle. Relying on the inherent tendency of source code to remain opaque, obfuscated code contests go a step further by seeing how far can such an opacity be sustained, often involving creative ways.

In this section, we've seen how the set of individuals who write and read code isn't homogeneous. Instead, we can see a significant degree of variation between source code written within the context of software engineering, hacking, scientific research and artistic activity. While none of these areas are exclusive of the others—a software developer by day can hack on the weekend and participate in code poetry events—, they do convey different perspectives on how the code is written, and on how it is evaluated. This cursory introduction to each approach has shown that, for instance, software engineers prefer code which is modular, modifiable, sustainable and understandable by the largest audience of possible contributors, while hackers would favor conciseness over expressivity, and tolerate idiosyncracy for the purpose of immediate, functional efficiency. On

the other hand, scientific programming favors ease of use, accuracy and reproducibility, sometimes overlapping with software engineering, while code poets explore the semantic tension between a human interpretation and the machine interpretation of a given source code.

What we see here are strands of similarity within apparent diversity. The code snippets in this section show that there is a tendency to prefer readability, conciseness, clarity, expressivity and functionality, even though different types of the aforementionned practices would put a different emphasis on each of those aspects. The question we turn to next, then, is to what extent do these different practices of code writing and reading share common judgments regarding their formal properties? Do hackers and poets agree on some value judgment, and how? To start this investigation, we first analyze programmers' discourses in the following section in order to identify concrete categories of formal properties which might enable a source code to be positively valued for its appearance, before we turn to the aesthetic registers code practicionners refer to when discussing beautiful code.

3.2 Ideals of beauty

With this overview of the varieties of practices at play amongst those who read and write source code, we will analyze more thoroughly what are the aesthetic standards most value by those different groups. The aim here is to formalize our understanding of which source code is considered beautiful, and to do so at multiple levels. The goal here is to capture both the specific manifestations of beautiful code as specified and enunciated by programmers, as well as the semantic contexts from which these enunciations originate. What we will see is that, while a set of aesthetic values and a set of aesthetic manifestations can be pinpointed precisely, the domains that are mobilized to justify these values are clearly distinct. To do so, we will introduce the framework of discourse analysis, complemented by a a medium-specific reading through critical code studies and rhetorical code studies on one hand, and the work done by conceptual metaphors on the other.

3.2.1 Introduction to the Methodology

Discourse consists of text, talk and media, which express ways of knowing the world, of experiencing and valuing the world. This work builds on Kintsch and Van Dijk's work on providing tools to analyze an instance of discourse, centered around what constitutes good source code. While discourse analysis is also used critically by unearthing which value judgments occur in power relationships[67], we focus here on aesthetic value judgments, as their are first expressed through language. Of all the different approaches to discourse, the one we focus on here is that of the *pragmatics*. We find this approach particularly fitting through its implication of the *cooperative principle*, in which utterances are ultimately related to one another through communicative cooperation to reveal the intent of

[67] See Diana Mullet on Critical Discourse Analysis[**?**]

the speaker[**?**]. Practically, this means that we assume the position of programmers talking to programmers is cooperative insofar as both speaker and listener want to achieve a similar goal: writing good code. This double understanding—focusing first and foremost on utterances, and then re-examining them within a broader context—will ultimately lead us to examine the influence of the production media (blog post, forums, conferences, text books), of the cultural background (software practices as outlined above as well as additional factors such as skill levels. Our comprehension of those texts, then, will be set in motion by a a dual movement between local, micro-units of meaning and broader, theoretical macro-structure of the text, and linked by acts of co-reference[**?**].

Particular attention will be paid to the difference between intentional and extensional meaning[**?**]. As we will see, some of the texts in our corpus tend to address a particular problem (e.g. on forums, social media or question & answer platforms), or to discuss broader concepts around well-written code. Particularly,

> Figures of speech may attract attention to important concepts, provide more cues for local and global coherence, suggest plausible pragmatic interpretations (e.g., a promise versus a threat), and will in general assign more structure to elements of the semantic representation, so that [meaning] retrieval is easier.[**?**]

Following this idea, we will proceed by examining syntactic markers to deduce overarching concepts at the semantic level. Among those syntactic markers, we include single propositions as explicit predicates regarding source code, lexical fields used in those predicates in order to identify their connotations and denotations, as well as for the tone of the enunciations to identify value judgments. At the semantic level, we will examine the socio-cultural references, the *a priori* knowledge assumed from the audience, as well as the cognitive units which compose the theme of the discourse at hand.

And yet, the discourses we will examine aren't exclusively composed of natural language, but also of source code extracts, resulting in a hybrid between natural and machine syntax within the same discursive artifact.

In line with John Cayley's analytic framework of structure, syntax and vocabulary[**?**], we can nonetheless echo discourse analysis as applied to natural languages. Cayley's framework highlights essential aspect of analysis which applies both to natural languages and source code: that of scales at which aesthetic judgment operates. It also provides a bridge with literature and literary studies without imposing too rigid of a grid. While it does not immediately acknowledge more traditional literary concepts such as fiction, authorship, literarity, etc., it does leave room for these concepts to be taken into account. Particularly, we will see that the concept of authorship—who writes to whom—will be useful in the future.

Finally, our intepretation of the macrostructures described by Kintsch and Van Dijk will rely extensively on the work done by metaphors as the conceptual level, rather than at the strictly linguistic one. Lakoff and Johsnon's seminal work develops a theory of conceptual metaphors by highlighting their essential dependence on pre-existing cognitive structures, which we associate with already-understood concepts. The metaphor maps a source domain (made up of cognitive structure(s)) to a target domain. In the process, they extend the field of applicability of metaphors from the strictly literary to the broadly cultural: metaphors work because each of us has some conception of those domains involved in the metaphorical process. Metaphors rely in part on a static understanding, resulting in a fixed meaning from the application of a given source to a given target. Some of these sources are called *schemas*, and are defined enough to not be mistaken for something else, but broad enough to allow for multiple variants of itself to be applied to various targets[**?**], providing both diversity and reliability. As we will see below, their approach allows us to focus not just on textual objects, but on the vast range of metaphors used specifically in computing-related environments. Given that the source of the metaphor

should be grounded, with as little invariablity as possible, in order to qualify a potentially ill-defined target domain, this provides us with a first foray into the inherent elusiveness and instability of computing when presented to a broader audience.

Going beyond the role of metaphors manifested in expressions such as *the desktop*, *the mouse*, or *the cloud*, we will explore Lakoff's understanding of the specifically poetic metaphor further below as preliminary work to assess the linguistic component of computing—source code. For now, we will pay close attention to what programmers are saying about (beautiful) source code, which metaphors they employ to support these value judgments, and why—focusing first on the metaphors *of* source code, before moving, in the section, to the metaphors *in* source code.

The corpus studied here consists of texts ranging from textbooks and trade manuals to blog posts and online forum discussions. The rationale behind such a broad approach is to constitute a lexical basis that is not just empirical (i.e. taking into account what practicing programmers consider when assessing good code, expressed in the everyday interactions of online forums and blog posts), but also prescriptive. The inclusion of more authoritative sources, such as canonical textbooks or widely-read blog posts from technology investors will allow us to introduce a normative dimension to our research. As this section highlights, there are *specific* ways to write good code, which are echoed both from bottom-up and from top-down perspectives.

3.2.2 Lexical Field in Programmer Discourse

In terms of existing studies of the lexical field programmers use, Erik Pineiro has done significant work in his doctoral thesis. In it, he argues that aesthetics exist from a programmers perspective, decoupled from the final, executable form of the software. While this current study draws on his work, and confirms his findings, it also departs from it in several aspects. First,

Pineiro focuses on a narrower corpus, that of the Slashdot.org[68] forums[**?**] (p. 51). Second, he examines aesthetic judgment from a private perspective of software engineers, separate from other possible aesthetic fields which might enter in dialogue with beautiful code[**?**] (p.52). Finally, his discussion of aesthetics takes place in a broader context of business management and productivity„ while this current study situates itself within aesthetic philosphy, and its implications within how things are considered beautiful.

Clean is the first adjective which stands out as a requirement when assessing beauty in code. It is featured in the title of a series of best-selling trade manuals written by Robert C. Martin and published by Prentice Hall from 2009 to 2021, the full titles of which clearly enunciate their normative aim[69]. Cleanliness, in Martin's terms, is defined by circumlocutions. After asking leading programmers what clean code means to them, he carries on in the volume by providing examples of *how* to achieve clean code, rather that by defining what it is. Nonetheless, some hints can be glimpsed from Ward Cunningham's answer:

> You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.[**?**] p.10

along with Grady Brooch's:

> Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent

[68] `https://slashdot.org`

[69] *Clean Code: A Handbook of Agile Software Craftsmanship*, *The Clean Coder: A Code Of Conduct For Professional Programmers*, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, *Clean Agile: Back to Basics*, *Clean Craftsmanship: Disciplines, Standards, and Ethics*.

> but rather is full of crisp abstractions and straightforward lines of control.[**?**]p.11

Cleanliness is thus tied to expressiveness: clean code is devoid of any extraneous syntactic and semantic symbols (e.g. it does one thing, and one thing well), in order to let the problem at hand appear, with all its implications. Instead, the tool (code, and programming languages) disappear at the syntactic level, to gain expressiveness at the semantic level. Cleanliness is mostly a definition by negation: it states that something is clean if it is free from impurities, blemish, error, etc. As an alternative to this definition by negation, In the spirit of defining by example, trade manuals such as *Clean Code* provide examples on how to move from bad code, to clean code through specific, practical guidelines regarding naming, spacing, class delimitation, etc..

Martin echoes Hunt when he advocates for such a definition of clean as lack of additional information:

> Don't spoil a perfectly good program by overembellishment and over-refinement.[**?**]

This advice to programmers denotes a conception of clean that is not just about removing as much syntactic form as possible, but which also implies a balance. *Overembellishment* implies excess addition, while *over-refinement* implies, on the contrary, excess removal. This normative approach finds its echo in the numerous quotations of Antoine de Saint-Exupéry's comment on aircraft design:

> Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher. [**?**][70]

[70] *In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away, when a body has been stripped down to its nakedness.*, translated by Lewis Galantière in the 1939 edition of *Wind, Sand and Stars*

This balance between too much and too little is found in another dichotomy stated by programmers: between simple and clever. Simplicity, argues Jeremy Gibbons, is not only a restraint on the quantity of syntactic tokens (as one could achieve by keeping names short, or aligning indentations), but also a semantic equilibrium at the level of abstracted ideas[**?**]. The balance between breadth and depth of the task of the code, between the precision of a use-case and its generalization, and its leveraging of existing library—i.e. supposedly reliable—code is summed up in a quote by Ralph Waldo Emerson concluding his column:

> We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes.[**?**]

In another paper published by the ACM, Kristiina Karvonen argues for simplicity not just as a design goal, as leveraged by human-computer interface designers, but as a term with a longer history within the tradition of aesthetic philosophy, especially the work of Johann Joachim Winckelmann[**?**]. In particular, she stresses the difficulty "to create significant, that is beautiful works of art with simple means"[**?**]. Here, her correlation between significance and beauty hints at the semantic role of simplicity, as a means to communicate ideas (i.e. *signify*) to an audience.

Precisely, simplicity is correlated with clarity (of meaning); if the former refers mainly to the syntactic component (fewer tokens), it enables the non-obfuscated framing of the ideas at play. One example is given by Dave Bush in a post titled *15 Ways to Write Beautiful Code*:

```
  void SomeMethod(){
    if(x != y){
        //-- stuff
    }
}
```

```
void SomeClearerMethod(){
    if(x == y) return;
    //-- do stuff
}
```

Here, the strive for simplicity leads to removing the brackets, and flipping the boolean check in the if-statement to add a `return`. Even though it is, strictly speaking, more chararcters than the brackets and newline (six characters compared to four), the program becomes clearer by separating the two branching cases inherent to the use of conditional logic, under the form of an if-statement. In the second version, it is made clear that, if a condition *is*, the execution should stop, and any subsequent statement can entirely disregard the existence of the if-statement; in the first version, the condition that *is not* is entangled with code that should be executed, since the existence of the if-statement has to be kept in mind until the closing bracket[**?**].

As a corollary to clarity stands obfuscation. It is the act, either intentional or un-intentional, to complicate the understanding of what a program does by leading the reader astray through a combination of syntactic techniques. In its most widely applied sense, obfuscation is used for practical production purposes: reducing the size of code, and preventing the leak of proprietary information regarding how a system behaves. For instance, the JavaScript source code in **??** is obfuscated through a process called *minification* into the source code in **??**

This process of obfuscation has very clear, quantitative assessment criterias, such as the size of the source code file and cryptographic complexity[**?**]. Nonetheless. obfuscation can also be valued as a positive aesthetic standard, of which the IOCCC is the most institutionalized guarantor, since 1984. These kinds of bofuscations, as Mateas and Montfort analyze, involve the playful exploration of the intertwinings of syntax and semantics, seeing how much one can bend the former without affecting the latter. These textual manipulations, they argue, possess an inherently

```
 import { ref, onMounted, reactive } from 'vue';
import Footer from './components/Footer.vue';
import Header from './components/Header.vue';
import { SyllabusType } from './js/types';

const msg = ref("")
const HOST = import.meta.env.DEV ? "http://localhost:3046" : ""
const syllabi = new Array<SyllabusType>()

let start = () => {
    window.location.href = '/cartridge.html'
}

onMounted(() => {
    fetch(`${HOST}/syllabi/`,
        {
            method: 'GET'
        })
        .then(res => {
            return res.json()
        })
        .then(data => {
            Object.assign(syllabi, JSON.parse(data))
            console.log(syllabi);
            if (syllabi.length == 0)
                msg.value = "No syllabi :("
            else
                msg.value = `There are ${syllabi.length} syllabi.`

        })
        .catch(err => {
            console.error(err)
            msg.value = "Network error :|"
        })
})
```

Listing 3.12: home.js (before minification)

```
import{_ as p,f as m,r as v,g as f,o as l,c as n,a as c,h as e,t as r
    ,b as u,i as b,u as _,F as y,H as g,e as w}from"./Header.js";
    const H={class:"container p-3"},N=e("h1",null,"Home",-1),k={class
    :"syllabi"},x=["href"],B={class:"cta"},F=m({setup(S){const s=v
    (""),d="http://localhost:3046",o=new Array;let h=()=>{window.
    location.href="/cartridge.html"};return f(()=>{fetch(`${d}/
    syllabi/`,{method:"GET"}).then(t=>t.json()).then(t=>{Object.
    assign(o,JSON.parse(t)),console.log(o),o.length==0?s.value="No
    syllabi :(":s.value=`There are ${o.length} syllabi.`}).catch(t=>{
    console.error(t),s.value="Network error :|"})}),(t,i)=>(l(),n(u,
    null,[c(g),e("main",H,[N,e("div",k,[e("div",null,r(s.value),1),e
    ("ul",null,[(l(!0),n(u,null,b(_(o),a=>(l(),n("li",null,[e("div",
    null,[e("a",{href:"/syllabi/"+a.ID},r(a.title),9,x)]),e("div",
    null,r(a.description),1)]))),256))])])]),e("div",B,[e("button",{id
    :"cta-upload",class:"btn btn-primary mb-4 cc-btn",onClick:i[0]||(
    i[0]=a=>_(h)())},"Upload yours!")])]),c(y)],64))}});var O=p(F,[["
    __file","/home/pierre/code/commonsyllabi/viewer/www/src/Home.vue
    "]]);w(O).mount("#app");
```

Listing 3.13: home.js (after minification)

literary quality:

> Obfuscation and weird languages invite us to join programming contexts to the literary contexts that must obviously be considered when evaluating literary code. They also suggest that coding can resist clarity and elegance to strive instead for complexity, can make the familiar unfamiliar, and can wrestle with the language in which it is written, just as much contemporary literature does.[**?**]

An example of such literary connection is given by Noël Arnaud's work *Poèmes Algol*[**?**], in which he uses the constructs of the language Algol 68 in order to evoke in the reader something different than what the program actually does (i.e. fail to execute anything meaningful).

Another insight on simplicity and programming regarding the communication of ideas is hinted at by Richard P. Gabriel in his use of the concept of *compression* in both poetry and programming. In an interview with Janice J. Jeiss, he states:

> I'm thinking about things like simplicity – how easy is it going to be for someone to look at it later? How well is it fulfilling the overall design that I have in mind? How well does it fit into the architecture? If I were writing a very long poem with many parts, I would be thinking, "Okay, how does this piece fit in with the other pieces? How is it part of the bigger picture?". When coding, I'm doing similar things, and if you look at the source code of extremely talented programmers, there's beauty in it. There's a lot of attention to compression, using the underlying programming language in a way that's easy to penetrate. Yes, writing code and writing poetry are similar. [**?**]

Simplicity in programming is presented here as akin to compression in poetry: in the increasing of semantic charge (or significance, in Karvonen's

terms) all the while reducing the syntactic load (or the quantitity of formal tokens in the source code). One of those extraneous loads is explanation:

> When it requires a lot of explanation like that, it's not "beautiful code," but "a clever hack."[**?**]

This answer by Mason Wheeler, posted on the software engineering *Stack Exchange* forum, in response to the question "How can you explain "beautiful code" to a non-programmer?"[**?**], highlights simplicity's opposite, cleverness.

Cleverness is often found, and sometimes derided, in examples of code written by hackers, since it unsettles the balance between precision and generality. Clever code would tend towards exploiting particularities of knowledge of the medium (the code) rather than the goal (the problem). Hillel Wayne presents this snippet of C code as an example of bad clever code:

```
def is_unique(_list):
  return len(set(_list)) == len(_list)
```

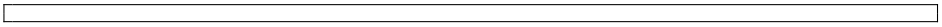
Here, the knowledge of how the `set()` function in Python behaves, is required in order to understand that the `is_unique()` function returns whether all the elements of the given list are unique. A programmer without familiarity with Python would be unable to do so without consulting the Python documentation (i.e. requiring extraneous explanation).

Hillel elaborates on the difference between "bad" clever code[71], which is essentiall read-only due to its idiosyncracy and reliance on tacit knowledge, and "good" clever code, and such distinction corroborates our previous observations regarding beautiful code as a means for expression of the problem domain. His example is that the problem of sorting the roughly

[71] See, for instance, Duff's device, an idiosyncratic and language-specific way to speed up loop unrolling in C. The author himself feels "a combination of pride and revulsion at this discovery"[**?**]

300 million U.S. american citizens by birthdate can be made considerably more efficient by cleverly considering that no U.S. american citizen is older than 120 years.

Meanwhile, cleverness is a valued attribute in the context of hacker code, putting more emphasis on the technical solution than on the problem domain. A salient is example was the 1994 `smr.c` entry to the IOCCC, which aimed at being the smallest self-reproducing program[**?**]. Here is an exact reproduction of the source code:



Consisting of a zero bytes file, `smr.c` provides both a clever understanding and reduction of the problem domain, and a clever understanding of what C compilers would effectively accept or not as a valid program-text[**?**]. Because it has since been banned under the rules of the IOCCC, this clever source code entirely renounces any claim to a more general application, and finds its aesthetic value only within a specific community.

Simplicity, then is the ability to provide code that fits the problem exactly: without being too precise, or too generic, displaying an understanding of and a focus on the application domain, rather than the applied tools, as William J. Mitchell sums it up in his introductory textbook for graphics programming:

> Complex statements have a zen-like reverence for perfect simplicity of expression.[**?**]

We see here the idea of reaching a conceptual revelation through the reduction of complex styntactical assemblages. This strive towards attaining an inverse relationship between the complexity of an idea and the means to express it is contiguous to another related criteria for beautiful source code present in programmers' discourse: elegance.

Chad Perrin, in his article *ITLOG Import: Elegance*, approaches the concept as a negation of the gratuitous, a means to reduce as much as possible

the syntactic footprint while keeping the conceptual footprint intact:

> In pursuing elegance, it is more important to be concise than merely brief. In a general sense, however, brevity of code does account for a decent quick and dirty measure of the potential elegance that can be eked out of a programming language, with length measured in number of distinct syntactic elements rather than the number of bytes of code: don't confuse the number of keystrokes in a variable assignment with the syntactic elements required to accomplish a variable assignment.[**?**]

He also hints at the additional meaningufness of elegance, as he compares it to other aesthetic properties, such as simplciity, complexity or symmetry. If simplicity inhabits a range between too specific and too general, he describes an elegant system as exactly appropriate for the task at hand. While he touches at length on the influence of progamming languages in the possibility to write elegant source code—a question to which we will come back in Chapter 4. Elegance, he says, relies on underlying principles, but is nonetheless subject to its manifestation through a particular, linguistic interface.

This underlying aspect is also present in Bruce McLennan's discussion of the concept. As he approaches it through the dual lens of structural engineering, this indicates that he also considers elegance as a more profound concept which can manifest itself across disciplines, both as a way of making, and as a way of thinking[**?**]. He defines his *Elegance Principle* as:

> Confine your attention to designs that *look* good because they *are* good.[**?**]

Such a definition relies heavily on the sensual component of elegance: while an underlying property of, at least, human activities, it must nonetheless be manifested in some perceptible way. On *Stackexchange*, user

asoundmove corroborates this conception of achieving a simple and clean system where any subsequent modification would lead to a decrease in quality:

> However to me beautiful code must not only be necessary, sufficient and self-explanatory, but it must also subjectively feel perfect & light.[**?**]

Once again connecting simplicity (under the guise of necessity and sufficiency), the perception of elegance is also related to a subjective feeling of adequacy. Paul DiLascia, writing of the Microsoft Developer Network Magazine, illustrates his conception of elegance—as a combination of simplicity, efficiency and brilliance—with recursion[**?**]:

```
int factorial(int n)
{
  return n==0 ? 1 : n * factorial(n-1);
}
```

Recursion, or the technique of defining something in terms of itself, is a positively valued feature of programming[**?**]. In so doing, it minimizes the number of elements at play and constrains the problem domain into a smaller set of moveable pieces. Another example, provided in the same *Stackexchange* discussion is the `quicksort` algorithm, which can be implemented recursively or iteratively, with the former being significanly shorter:

```
  // https://stackoverflow.com/a/12553314/4665412
  public static void recursiveQsort(int[] arr,Integer start, Integer
      end) {
    if (end - start < 2) return; //stop clause
    int p = start + ((end-start)/2);
    p = partition(arr,p,start,end);
    recursiveQsort(arr, start, p);
    recursiveQsort(arr, p+1, end);
}
```

```
public static void iterativeQsort(int[] arr) {
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(0);
    stack.push(arr.length);
    while (!stack.isEmpty()) {
        int end = stack.pop();
        int start = stack.pop();
        if (end - start < 2) continue;
        int p = start + ((end-start)/2);
        p = partition(arr,p,start,end);

        stack.push(p+1);
        stack.push(end);

        stack.push(start);
        stack.push(p);
    }
}
```

To conclude this brief survey on how programmers perceive elegance in source code, we can follow Mahmoud Efatmaneshik and Michael J. Ryan who, in the IEEE Systems journal, offer a defintion of elegance which relies both on a romantic perception—including subjective perception, ”gracefulness”, ”appropriateness” and ”usability”—and practical assessment with terms such as ”simple”, ”neat”, ”parsimonious” or ”efficient”[**?**]. In doing so, they ground source code aesthetics as a resolutely dualistic norm, between subjectivity and objectivity, qualitative and quantitative[72].

And yet, rather than subjectivity and objectivity being opposites, one could also consider them as contingent. Due to the interchangeability in the use of the some of the terms we’ve seen by programmers, both qualitative—in terms of the language used—and quantitative—in terms of the syntax/semantics ration—assessments of source seem to be comple-

[72] A duality we will investigate further through the prism of human and machine understanding in section XXX

mentary in considering it elegant.

Another way to understand what programmers mean when they talk about beautiful code is to look beyond the positive terms used to qualify it (clarity, simplicity, elegance, etc.), and shift our attention to how other terms are used negatively. We have already touched up qualifiers such as clever, or obfuscating, which have ambiguous statuses depending on the community that they're being used in—specifically hackers and artists.

- smelly

- entangled, spaghetti, interdependency (chandra)

- verbose

3.3 Aesthetic domains - 6000

Now that we've done some empirical work, we can try to abstract away a bit and then look into how this relates to existing frameworks of aesthetics.

Now that *we look at proofs, through discourse*, what kind of beauty can we be dealing with?

3.3.1 Literary Beauty

chandra, geek sublime

matz, code as an essay

This second approach contrasts with the functional component of the first one, but nonetheless stands in relationship with it. the creative beauty, by defying traditional beauty standards, does help us highlight, through deviance, what the norm is. These texts on "creative beauty" include the classical perl poetry, code poems, IOCC, code poetry contest, etc.

The poem *Black Perl*, submitted anonymously, is a representative example of the richness of the productions of this community:

```
#!/usr/bin perl
no warnings;

BEFOREHAND: close door, each window & exit; wait until time.
  open spellbook, study, read (scan, $elect, tell us);
write it, print the hex while each watches,
  reverse its, length, write, again;
kill spiders, pop them, chop, split, kill them.
  unlink arms, shift, wait & listen (listening, wait),
  sort the flock (then, warn "the goats" & kill "the sheep");
kill them, dump qualms, shift moralities,
  values aside, each one;
die sheep? die to : reverse { the => system
  ( you accept (reject, respect) ) };
next step,
  kill `the next sacrifice`, each sacrifice,
  wait, redo ritual until "all the spirits are pleased";
do { it => "as they say" }.
  do { it => (*everyone***must***participate***in***forbidden**s*e*
       x*)
+ }.
  return last victim; package body;
exit crypt (time, times & "half a time") & close it,
```

```
    select (quickly) & warn your (next victim);
  AFTERWARDS: tell nobody.
    wait, wait until time;
  wait until next year, next decade;
  sleep, sleep, die yourself,
    die @last
```

The most obvious feature of this code poem is that it can be read by anyone, including by readers with no previous programming experience: each word is valid both as English and as Perl. A second feature is the abundant use of verbs. Perl belongs to a family of programming languages grouped under the *imperative* paradigm, which matches a grammatical mood of natural languages, the *imperative mood*. Such mood emphasizes actions to be take rather than, for instance, descriptions of situations, and thus sets a clear tone for the poem. The fact that Perl is based on stating procedures to be executed and states to be changed creates this feeling of relentless urgency when reading through the poem, a constant need to be taking actions, for things to be changed. Here, the native constraints of the programming language interacts directly with the poetic suggestion of the work in a first way: the nature of Perl is that of giving orders, resulting in a poem which addresses *someone* to execute *something*. Still, Perl's flexibility leaves us wondering as to who and what are concerned by these orders. Is the poem directing its words to itself? To the reader? Is Perl just ever talking exclusively to the computer? This ambiguity of the adressee adds to the ominousness of each verse.

- minimalism - https://vimeo.com/47364930 → concise code is code as litterature, because he says one of the issues is that there are just too many lines of code that one can wrap its head around. so there's a need for shrinking down content

3.3.2 Mathematical beauty

Mostly elegance, could be a good place to work on the distinction between proof and theorem, concept and appearance.

the program is perfect in the mind, and mistakes come when translation occurs into chunks/statements (kinda like a mathematical work: the theorem is there first, and the proof comes laboriously second cf. **"Mathematical Invention", Poincaré, 1914"** → **"The Art of Creation", Arthur Koesler, 1960**)

- Syntactic simplicity, or elegance, measures the number and conciseness of the theory's basic principles. Ontological simplicity, or parsimony, measures the number of kinds of entities postulated by the theory. code is syntactic simplicity because wrngles together complex concepts (e.g. perl, one liners), or code is ontological simplicity, because all is within computation (e.g. lisp)

3.3.3 Architectural beauty

This allows a segue into everyday aesthetics and environmental aesthetics

Completing his definition of elegance, McLennan touches upon the way to assess elegance through visual means in another publication on the more general need for aesthetics[73]. The ability to distinguish an elegant system from an ugly one, then, comes through *practice*, the implications of which we will examine in the last section of this chapter[**?**].

[73] Along with elegance, he includes efficiency and economy as the core aesthetic values of technological activities.

3.4 Craft and beauty - 10p

Paul Graham, LISP programmer, co-founder of the Y Combinator startup accelerator and widely-read blogger[74], highlights the status of programming languages as a medium, in its essay *Hackers and Painters*[**?**]. Particularly, he stresses the materiality of code, depicting hackers as people who:

> are trying to write interesting software, and for whom computers are just a medium of expression, as concrete is for architects or paint for painters.

the ethical aspect of writing beautiful code: well-written, good code is value aesthetically and morally

also combination of hard and soft (cf. green coding guidelines, hun pragmatic programmer)

Now that we've seen how the aesthetic ideals of code borrow from different registers, we need to think about practice, or ways. all of the above can be seen through the prism of craft

3.4.1 Functional beauty

again, pineiro as instrumental goodness

This first approach, by comparing both source and comment at the same time (taking texts which are explicitly described as being beautiful), explicitly highlights the requirements for source code to be beautiful.

There is also an emerging development in aesthetics of integrating function as a criteria for an aesthetic experience.

3.4.2 Embodiment

also note the place and role of tools (IDEs, teletypes, fast compiling, etc.)

[74] And even achieving god-like status among certain circles[**?**]

3.4.3 Tacit knowledge

→ this one actually goes to the next chapter