Thesis check-in - Role of aesthetics in understanding source code

Pierre Depaz

March 2021

1 Introduction

After establishing ground work on aesthetic manifestations in source code for software developers during the Spring 2020 semester, I have concluded with both an empirical manifestation of beautiful code, synthesized a typology of such manifestations (INCLUDE FINDINGS IN THREE.MD)—how code can be beautiful—and laid out a preliminary investigation as to why code could be beautiful. Aesthetic manifesations ("beauty") seem to occur whenever facilitate the clarity of intent of the writer, and the agency of the reader, are heightened. Beautiful code makes the underlying concepts clear and easily-graspable, and facilitates its modification by the reader by providing an error-free, and cognitively easy way to do so.

Furthermore, I outlined several directions for further research. These included the exploration of aesthetic standards for two additional categories of code writers: *source code poets* and *hackers*. Following discussions around this outcome, I have added three other directions: literary metaphors, architectural parallels and machine understanding. First, the place of literary metaphors is a response to the cognitive stake at play in reading and writing code, since code can be understood as a formal representation of mental models emerging from complex data structures and

their processing during execution time. Second, the parallels with architecture were suggested by a similar relationship to structure, planning and construction. These parallels are, as we will see below, claimed by software developers themselves, ranging from job titles to commercial best practices of software patterns; on a more theoretical level, the approaches to beauty in architecture will turn out to be productive lenses when thinking not just about executed code, but about source code as well. Third, when claiming that beautiful code facilitates understanding(s), it is important to clarify *whose* understanding of *what*. While previous work has focused on human understandings of human intentions, and human-made concepts, this document investigates to what extent do computers, as concrete machines, understand anything.

I will start by examining instances of source code poetry, defining it, contextualizing it, and analyzing it through close-readings. This will allow us to highlight specific aesthetic standards emerging from this corpus, namely *semantic layering* and *procedural rhetoric*. Source code poetry, with this clear emphasis on *poetry*, will then allow us to address the traditional relationship between literature and code, on an artistic level as well as on a linguistic one. The two concepts mentioned above will lead to an examination of the metaphor, from a literary and from a cognitive standpoint.

Particularly, the relationship that metaphors maintain to the process of knowing and understanding will be highlighted both in human texts and in program texts[1]. Connecting it to mental models will allow us to start thinking of these program texts in terms of *structure*, both surface-structure and deep-structure, and address how a theoretical framework of aesthetics might be connecting the two, including the place of imagination in acquiring knowledge and building understanding in these texts.

Mentioning structure will thus lead us into the overlap between architecture and software. After a short overview of how the two are usually related, I examine a particular set of aesthetic standards developed by Christopher

Alexander in his work on pattern languages. At the cursory level, these are tied to software patterns, techniques for developing better software that have emerged more out of practice rather than out of theory. At a deeper level, we will see that the standards of beauty—or, rather, of this *Quality Without a Name*—can be applied productively to better understand what qualities are exhibited by a program that is deemed beautiful. In particular, Richard P. Gabriel's work will further provide a connection between software, architecture and poetry.

One particular aspect of architecture—the folly, and to some extent large-scale installation artworks—will allow us to transition into our next corpus: hacking. Hacking, defined further as seemingly-exclusively functional code will further requalify the need for aesthetics in source code. We will see how this practice is focused much less on human understanding than on machine understanding, on producing code that is unreadable for the former, and yet crystal-clear for the latter—with an emphasis on human and machine performance. Despite a current lack of extensive research on hacking-related program texts, we will look into two instances of these: the one-liner and the demo to support our investigation in this domain.

This brings us to the broader question of human understanding and machine understandings. Starting from the distinction between syntax and semantics, I highlight discrepancies between semantics in natural languages and semantics in programming languages to define machine understanding as an autotelic one, completely enclosed within a formal description. Coming back to Goodman, we will see how such a formal system fits as a *language of art*, and yet remains ambiguous: is computation exclusively concerned with itself, or can it be said that it relates to the rest of the (non-computable) world? Additionally, the question of aesthetics within programming languages themselves will be approached in a dual approach: as linguistic constructs presenting affordances for creating program texts which exhibit aesthetic properties, and as objects with aesthetic proper-

ties themselves. Whether or not we can agree on machine understanding, the formalism of programming languages, and their aesthetic possibilities, provide an additional perspective on the communication of non-obvious concepts inherent to computing.

In conclusion, we will see that aesthetics in code is not exclusively a literary affair in the strict sense of the term, but is rather at the intersection of literature, architecture and problem-solving, insofar as they manifest through the (re-)presentation of complex concepts and multi-faceted uses, involving their writers and readers in semantics-heavy cognitive processes and mental structures.

Finally, I suggest further directions for research.

2 Computers and literary arts

This section focuses on source code poetry, as the closest use of "literary arts" involving code. We will see how this particular way of writing software, to an explicitly aesthetic end, rather than a functional one, summons specific claims to art and beauty. These claims maintain a complex relationship to the nature and purpose of code, in certain ways embracing the former, and moving away from the latter, but nonetheless allow us to more clearly define such a nature and such purposes. After an overview of the field, including delimitation of our corpus, I will highlight and analyze particular source code poems, chosen for their meaning-making affordances, and conclude on the aesthetic standards at play in their reading and writing, expanding on notions of *double-meaning* and *double-coding*.

2.1 Overview

Source code poetry is a distinct subset of electronic literature. 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 nonetheless 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, one of the distinctions that can be made in defining the elements of electronic literature which are included in our corpus is, in line with the framework of this research, the shift from output to input, for executable binary to latent source.

A large section of the works which fall within electronic literature focus on the result of an executed program, often effectively obfuscating one of the many chained acts of writing[1] which allow for the very existence of these works. For instance, the influence of *Colossal Cave Adventure*[2], the first work of interactive fiction, has been centered around on the playable output of the software, rather than on its source code. Written in FORTRAN 4 between 1975 1977, it exhibits several features which wouldn't fit within the typology we've previously established, particularly in terms of variable naming (e.g. variables such as 'KKKT', 'JSPK'; or 'GOTO' statements, whose harm has been considered at the same time this code was written[2]). *Colossal Cave Adventure*'s source code was indeed only examined due to the recognition of the cultural influence of the game, decades later, and not for its intrinsic properties.

A more contemporary example would be that of the Twine game engine, lowering the barrier to entry for writing interactive fictions in the age of the hyperlink. The result, while aesthetically satisfying, widely recognized and appreciated by the interactive fiction community, nonetheless consists in a single HTML document, comprising well-formatted and understandable HTML and CSS markups, along with three single lines of "uglified" JavaScript[3]. The explicit process of uglification[4] relies on the assumption

[1] See: Béatrice Fraenkel on chains of writing

[2] retrieved from: https://jerz.setonhill.edu/if/crowther/advf4.77-03-11

[3] For instance, the source code of https://pierredepaz.net/-/who/ consists of three lines of 52980 characters, and only 682 whitespace characters

[4] We could expand on this process of uglification, which consists of compacting humanly-

that no one would, or should, read the source code.

In the case of visual poetry, one can see how the source code of works such as bpNichol's *First Screening*[5], is dictated exclusively by the desired output, with a by-product of visually pleasing artifacts throughout the code as foreshadowing the result to come[6]. It is a literal description of static, desired output, more akin to a cinematic timeline editor, in which there is a 1:1 relationship between the clips laid out and the final reel, and no room for unexpected developments. While computer-powered, such an example of visual poetry tend to side-step the *potentiality* of computing, of which source code is one of the descriptive symbol systems: each execution of the code is going to be exactly the same as the previous one, and the same as the next one[7]. While this might be a drastic example, in which unknowns are reduced to a minimum, visual poetry and interactive do rely heavily on the dynamic aspect of computer procedures to create aesthetic experiences[8]. The difference I am making here is that such aesthetic experience are claimed to take place in the realization of the computer-aided potentials of the work, rather than in the textual description of these potentials[9]. These examples, while far from being exhaustive, nonetheless show how little attention is paid to the source code of these works, since they are clearly—and rightly so—not their most important part.

Computer poetry, an artform 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

laid out source code into the small possible number of characters, usually for a production-ready build, optimized for loading times and dependency processing.

[5]https://www.vispo.com/bp/download/FirstScreeningBybpNichol.txt

[6]Still, a lovely artefact is the subroutine at line 1600, an "offscreen romance" only visible in the source.

[7]Barring any programmer-independent variables, such as hardware and software platform differences.

[8]For instance, see Text Rain, by Camille Utterbach and Romy Achituv

[9]Tellingly, the Smithsonian Museum, which acquired Text Rain, makes no mention of the source code of the piece.

the syntactical output of the program. Starting with Christopher Stratchey's love letters, 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, a fact highlighted by their ratio: a single rule for a seemingly-infinite amount of outputs.

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 many others[3], 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, it's programmer, an individual sometimes different from the poet). The poem exists through the code, but isn't exclusively limited to the humanly-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[10]. These do constitute a body of work centered around the concept of generative aesthetics[4], 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 musi-

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

cal works as well; still, this attention to the result make these works fall on the periphery of our current research.

The aspects of electronic literature examined so far still require computer execution in order to be fully realized as aesthetic experiences. We now turn to these works which still function as works of explicit aesthetic value primarily through the reading of their source. We will examine obfuscated code and code poetry (both at the surface level and at the deep level), to finally delimitate our corpus around the last one.

One of the earliest instances of computer source written exclusively to elicit a human emotional reaction, rather than fulfill any immediate, practical function, is perhaps the Apollo 11 Guidance Computer (AGC) code, written in 1969[11] in Assembler. Cultural references and jokes are peppered throughout the text as comments, asserting computer code as a means of expression beyond exclusively technical tasks[12], 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.

```
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

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

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

development of more expressive programming languages (e.g. Pascal in 1970, C in 1972), software development has become a larger field, growing exponentially[13], and fostering practices, communities and development styles and patterns[14]. 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[15] is the most popular and oldest organized production of such code, in which programmers submit 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.

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

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

[15] `https://ioccc.org`

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_
            _-_-_-_
}
```

[language=C, Caption="r.c, 1988"]

The above submission to the 1988 IOCCC[16] 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

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

traditional programming syntax[17], but rather on an intuitive, human-specific understanding[18].

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[19]. What 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.

2.2 Source code poetry

It is this 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 and obfuscated code, code poetry starts from this essential feature of computers to work with strictly defined

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

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

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

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[20], 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[5]; 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 languages. 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 make the whole communication process fail whenever a specific word, or a word order isn't understood.

The community of programmers writing in Perl[21] has been one of the

[20] See perl haikus in particular

[21] See: perlmonks, with the spiritual, devoted and communal undertones that such a name

most vibrant and productive communities when it comes to code poetry. Such a 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[6]. The poem *Black Perl*, submitted anonymously, is a representative example 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 without 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 proce-

implies.

dures 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.

The object of each of these predicates presents a different kind of ambiguity: earlier versions of Perl function in such a way that they ignore unknown tokens[22][23]. Each of the non-reserved keywords in the poem are therefore, to the Perl interpreter, potentially inexistant, allowing for a large latitude of creative freedom from the writer's part. Such a feature allows for a tension between the strict, untoucheable meaning of Perl's reserved keywords, and the almost infinite combination of variable and procedure names and regular expressions. This tension nonetheless happens within a certain rhythm, resulting from the programming syntax: `kill them, dump qualms, shift moralities`, here alternating the computer's lexicon and the poet's, both distinct and nonetheless intertwined to create a *Gestalt*, a whole which is more than the sum of its parts.

A clever use of Perl's handling of undefined variables and execution order allows the writer to use keywords for their human semantics, while subverting their actual computer function. For instance, the `die` function should raise an exception, but wrapped within the `exit ()` and `close` keywords, the command is not interpred and therefore never reaches the execution point, bypassing the abrupt interruption. The subversion here isn't purely semi-

[22] e.g. undefined variables do not cause a core dump.

[23] Which results in the poem having to be updated/ported, in this case by someone else than the original writer

otic, in the sense of what each individual word means, but rather in how the control flow of the program operates—technical skill is in this case required for artistic skill to be displayed.

Finally, the use of the `BEFOREHAND:` and `AFTERWARDS:` words mimick computing concepts which do not actually exist in Perl's implementation: the pre-processor and post-processor directives. Present in languages such a C, these specify code which is to be executed respectively before and after the main routine. In this poem, though, these patterns are co-opted to reminisce the reader of the prologue and epilogue sometimes present in literary texts. Again, these seem to be both valid in computer and human terms, and yet seem to come from different realms.

This instance of Perl poetry highlights a couple of concepts that are particularly present in code poetry. While it has technical knowledge of the language in common with obfuscation, it departs from obfuscated works, which operate through syntax compression, by highlighting the expressive power of semiotic ambiguity, giving new meaning to reserved keywords. Such an ambiguity is furthermore bi-directional: the computing keywords become imbued with natural language significance, bringing the lexicon of the machine into the realm of the poetic, while the human-defined variable and procedure names, and of the regular expressions, are chosen as to appear in line with the rhythm and structure of the language. Such a work highlights the co-existence of human and machine meaning inherent to any program text[24].

[24] Except perhaps those which deal exclusively with scientific and mathematical concepts

```
class Proc
                            def in_discomfort?; :me; end
                                                                    end
you_are = you =

   ->(you) do
       self.inspect until true
         until nil
               break you
                                                      end
           puts you.in_discomfort?
             you_are[you]
                                                  end

you[
          you_are
]
```

The poem above, written in Ruby by maca[25] in 2011 and titled `self_inspect.rb`, opens up an additional perspective on the relationship between aesthetics and expressivity in source code. Immediately, the layout of the poem is reminiscent both of obfuscated works and of free-verse poetry, such as E.E. Cummings' and Stéphane Mallarmé's works[26]. This particular layout highlights the ultimately arbitrary nature of whitespace use in source code formatting: `self_inspect.rb` breaks away from the implicit rhythm embraced in *Black Perl*, and links to the topics of the poem (introspection and *unheimlichkeit*) by abandoning what are, ultimately, social conventions, and reorganizing the layout to emphasize both keyword and topic, exemplified in the `end` keyword, pushed away at the end of their line.

The poem presents additional features which operate on another level, halfway between the surface and deep structures of the program text. First, the writer makes expressive use of the syntax of Ruby by involving data types. While *Black Perl* remained evasive about the computer semantics of the variables, such semantics take here an integral part. Two data types, the array and the symbol are used not just exclusively as syntactical necessities (since they don't immediately fulfill any essential purpose), but rather as semantic ones. The use of `:me` on line 2 is the only occurence of

[25] `https://github.com/maca`

[26] Particularly *Un coup de dés jamais n'abolira le hasard*.

the first-person pronoun, standing out in a poem littered with references to `you`. Symbols, unlike variable names, stand for variable or method names. While `you` refers to a (hypothetically-)defined value[27], a symbol refers to a variable name, a variable name which is here undefined. Such a reference to a first-person pronoun implies at the same time its ever elusiveness. It is here expressed through this specific syntactic use of this particular data type, while the second-person is referred to through regular variable names, possibly closer to an actual definition. It is a subtlety which doesn't have an immediate equivalent in natural language, and by relying on the concept of reference, hints at an essential *différance* between you and me.

Reinforcing this theme of the elusiveness of the self, maca plays with the ambiguity of the value and type of `you` and `you_are`, until they are revealed to be arrays. Arrays are basic data structures consisting of sequential values, and representing `you` as such suggests the concept of the multiplicity of the self, adding another dimension to the theme of elusiveness. The discomfort of the poem's voice comes from, finally, from this lack of clear definition of who `you` is. Using `you_are` as an index to select an element of an array, subverts the role suggested by the declarative syntax of *you are*. The index, here, doesn't define anything, and yet always refers to something, because of the assigment of its value to what the lambda expression `->` returns. This further complicates the poem's attempt at defining the self, returning the reverse expression `you_are[you]`. While such an expression might have clear, even simple, semantics when read out loud from a natural language perspective, knowledge of the programing language reveals that such a way to assign value contributes significantly to the poem's expressive abilities.

A final feature exhibited by the poem is the execution of the procedure. When running the code, the result is an endless output of print statements of "me", since Ruby interprets the last statement of a program as a return

[27] A variable name can represent a value and/or a memory address

value to be printed:

```
...
me
me
me
me
me
me
me
me
me
me
me
me
me
me
me
me
Traceback (most recent call last):
        11913: from poem.rb:16:in ‘<main>’
        11912: from poem.rb:13:in ‘block in <main>’
        11911: from poem.rb:13:in ‘block in <main>’
        11910: from poem.rb:13:in ‘block in <main>’
        11909: from poem.rb:13:in ‘block in <main>’
        11908: from poem.rb:13:in ‘block in <main>’
        11907: from poem.rb:13:in ‘block in <main>’
        11906: from poem.rb:13:in ‘block in <main>’
         ... 11901 levels...
            4: from poem.rb:13:in ‘block in <main>’
            3: from poem.rb:13:in ‘block in <main>’
            2: from poem.rb:12:in ‘block in <main>’
            1: from poem.rb:12:in ‘puts’
self_inspect.rb:12:in ‘puts’: stack level too deep (SystemStackError)
```

The computer execution of the poem provides an additional layer of meaning to our human interpretation. Through the assignment of `you_are` in an `until` loop, the result is an endless succession of the literal interpretation of the symbol `:me`, the actual result of being in discomfort. While we've seen that a symbol only refers to something *else*, the concrete[28] output of the poem evokes an insistence of the literal self, exhibiting a different tone than a source in which the presence of the pronoun *you* is clearly dominant. Such a duality of concepts is thus represented in the duality of a concise source and of an extensive output, and is punctuated by the ultimate impossibility of the machine to process the accumulation of these intertwined references to *me* and *you*, resulting in a stack overflow error.

[28] Both in terms of actual, and in terms of concrete poetry

The added depth of meaning from this code poem goes beyond the syntactic and semantic interplay immediately visible when reading the source, as the execution provides a result whose meaning depends on the co-existence of both source and output. Beyond keywords, variable names and data structures, it is also the procedure itself which gains expressive power: a poem initially about *you* results in a humanly infinite, but hardware-bounded, series of *me*[29].

```
Include corpus
```

2.3 Aesthetics of source code poetry

These analyses of program texts have highlighted some of the aesthetic features of source code which can elicit a poetic experience during both reading and execution. These can be further qualified through several concepts, which I introduce and extend here.

The first is **double-meaning**, taken from Camille Paloque-Bergès's work on networked texts, and her analysis of code poetics[7]. She defines it as the affordance provided by the English-like syntax of keywords reserved for programming to act as natural-language signifiers. As we've seen in *Black Perl*, the Perl functions can omdeed be interpreted as regular words when the source is read as a human text. Starting from her analysis of *codeworks*, a body of literature centered around a créole language halfway between humanspeak and computerspeak[30], it can be extended into the aesthetically productive overlap of syntactic realms.

Previous research by Philippe Bootz has also highlighted the concept of the *double-text* in the context of computer poetry, a text which exists both in its prototypal, virtual, imagined form, under its source manifestation, and which exists as an instantiated, realized one[8]. However, he asserts that, in its virtual form, "a work has no reality", specifically because it is

[29] Another productive comparison could be found in Stein's work, *Rose is a rose is a rose...*
[30] See in particular the work of Alan Sondheim and mezangelle

not realized. Here again, we encounter the dependence of the source on its realized output, indeed a defining feature of the generative aesthetics of computer poetry. As we've seen in the `self_inspect.rb` poem, a work of code poetry can very much exist in its prototypal form, with its output providing only additional meaning, further qualifying the themes laid out in source beforehand. Indeed, the output of that poem would have a drastically diminished semantic richness if the source isn't also read and understood. For this double-meaning to take place, we can say that the sitation is inverted: the output becomes the virtual, imagined text, while the source is the concrete instantiation of the poem.

Second, we draw on Geoff Cox and Alex McLean's concept of **double-coding**[9]. According to them, double-coding *"exemplifies the material aspects of code both on a functional and an expressive level"* (p.9). Cox and McLean's work, in a thorough exploration of source code as an expressive medium, focus on the political features of speaking through code, as a subversive praxis. They work on the broad social implications of written and spoken[31] code, rather than exclusively on the specific features of what makes source code expressive in the first place. Double-coding nonetheless helps us identify the unique structural features of programming languages which support this expressivity. As we've briefly investigated, notably through the use of data types such as symbols and arrays in source code poetry, programming languages and their syntax hold within them a specific kind of semantics which hold, for those who are familiar with them and understand them, expressive power, once the data type is understood both in its literal sense, and in its metaphorical one. The succint and relevant use of these linguistic features can thicken the meaning of a poem, bringing into the realm of the thinkable ways to approach metaphysical topics.

Finally, the tight coupling of the source code and the executed re-

[31] Which they conflate with the practice of live-coding

sult brings up Ian Bogost's concept of **procedural rhetoric**[10]. Bogost presents procedures as a novel means of persuasion, along verbal and visual rhetorics. Working within the realm of videogames, he outlines that the design and execution of processes afford particular stances which, in turn, influence a specific worldview, and therefore arguing for the validity of its existence. Further work has shown that source code examination can already represent these procedures, and hence construct a potential dynamic world from the source[11]. If procedures are expressive, if they can map to particular versions of a world which the player/reader experiences[32], then it can be said that their textual description can also already persuasive, and elicit both rational and emotional reactions due to their depiction of higher-order concepts (e.g. consumption, urbanism, identity, morality). As its prototypal version, source code acts as the pre-requisite for such a rhetoric, and part of its expressive power lies in the procedures it deploys (whether from value assignment, execution jumps or from its overall paradigms[33]). Manifested at the surface level through code, these procedures however run deeper into the conceptual structure of the program text, and such conceptual structures can nonetheless be echoed in the lived experiences of the reader.

We've seen through this section that the poetic expressivity of source code poems rely on several aesthetic mechanisms, which can be combined for further expressive effect. From layout and syntactic obfuscation, to double-meaning through variables and procedure names, double-coding and the integration of data types and functional code into a program text and a rhetoric of procedures in their written form, all of these activate the connection between programming concepts and human concepts to bring the unthinkable within the reach of the thinkable. The next section will explore this connection further, in terms of mental models, literary metaphors and cognitive structures.

[32] Versions of worlds can be explored further through Goodman's *Ways of Worldmaking*

[33] e.g. declarative, imperative, functional

3 Metaphors and Mental Models

This section focuses on metaphors and mental models in both programming and literature, and how they're related to understanding and cognition, both in broad terms and in specific texts. The evokative power of the excerpts seen above make ample use of the multiple facets of understanding, switching from one frame to the other. This switch relates to the most commonly used definition of metaphor: that of labeling one thing in terms of another, thereby granting additional meaning to the subject at hand. Our approach here will bypass some of the more minute distinctions made between metonymy (in which the two things mentioned are already conceptually closely related), comparison (explicitly assessing differences and similarities between two things, often from a value-based perspective) and synechdoche (representing a whole by a subset), as they all relate to a larger, more contemporary definition of the concept.

This part of the thesis relies especially on the works of George Lakoff and Mark Johnson, and of Paul Ricoeur, due to their requalification of the nature and role of metaphor in the 20th century. While Lakoff and Johnson's approach to the conceptual metaphor will serve a basis to explore metaphors in the broad sense across software and narrative, I also argue that Ricoeur's focus on the tension of the *statement* rather than primarily on the *word* will help us better understand some of the aesthetic manifestations of software metaphors, without being limited to tokens. Following a brief overview of their contributions, I examine the various uses of metaphor in software and in literature, touch upon the cognitive turn in literary studies, and conclude the section by the ambiguity of a cognitive account of programming.

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[12], 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, to complement his broadening of the metaphorical field, we turn to Paul Ricoeur's assessment of it.

Writing in *The Rule of Metaphor*, Ricoeur operates two shifts which will help us better assess not just the inherent complexity of program texts, but the ambivalence of programming languages as well. His first shift regards the locus of the metaphor, which he saw as being limited to the single word—a semiotic element—to the whole sentence—a semantic element[13]. This operates in parallel with his attention to the *lived* feature of the metaphor, insofar it exists in a broader, vital, experienced context. Approaching the metaphor while limiting it to words is counterproductive because words refer back to "contextually missing parts"—they are eminently overdetermined, polysemic, and belong to a wider network meaning

than a single, one-to-one relationship[34]. Looking at it from the perspective of the sentence brings this rich network of potential meanings and broadens the scope for interpretation. As we've briefly touched upon in the previous section when reading `self_inspect.rb`, all of the evocative meaning of the poem isn't contained exclusively in each token, and the power of the whole is greater than the sum of its parts.

Secondly, Ricoeur inspects a defining aspect of a metaphor by the *tensions* it creates. His analysis builds from the polarities he identifies in discourse between event (time-bound) and meaning (timeless), between individual (subjective, located) and universal (applicable to all) and between sense (definite) and reference (indefinite)[35]. The creative power of the metaphor is its ability to both create and resolve these tensions, to maintain a balance between a literal interpretation, and a metaphorical one—between the immediate and the potential, so to speak. Tying it to the need for language to be fully realized in the lived experience, he poses metaphor as a means to creatively redescribe reality. As we will be approaching the topic of syntax and semantics in programming languages, we will see that these tensions can be a fertile ground for poetic creation through aesthetic manifestations.

3.1 In literature

If the conceptual turn initiated by Lakoff and Johnson's analysis of the metaphor broadens the horizon of their applicability beyond the strict domain of literature, it is nonetheless clear that metaphors appear and operate in particular ways in literary works, from fiction to poetry. We look at such specificity here in anticipation of identifying which features of poetic metaphors could be mapped to the program texts of our corpus—whether

[34] As he sees it in the traditional, Aristotelician sense of the term.

[35] For the extent to which source code can be considered discourse has been discussed, see: Cox and McLean, *Speaking Code*.

explicitly poetic, as in source code poetry, or not, as in regular source code.

So while Lakoff bases poetic metaphors on the broader metaphors of the everyday life, he also operates the distinction that, contrary to conventional metaphors which are so widely accepted that they go unnoticed, the poetic metaphor is *non-obvious*. Which is not to say that it is convoluted, but rather that it is new, unexpected, that it brings something previously not thought of into the company of broad, conventional metaphors—concepts we can all relate to because of the conceptual structures we are already carry with us, or are able to easily integrate. This echoes our mention of Flusser's analysis of poetry as that which brings ideas into the realm of the thinkable.

It does so along four different axes, in terms of how the source domain affects the target domain that is connected to. First, a source domain can *extend* its target counterpart: it pushes it in an already expected direction, but does so even further, sometimes creating a dramatic effect by this movement from conventional to poetic. For instance, a conventional metaphor would be saying that *"Juliet is radiant"*, while a poetic one might extend the attribution of positivity associated with brightness by saying *"Juliet is the sun*[36].

Poetic metaphors can also *elaborate*, by adding more dimensions to the target domain, while nonetheless being related to its original dimension. Here, dimensions are themselves categories within which the target domain usually falls (e.g. the sun has an astral dimension, and a sensual dimension). Naming oneself as *The Sun-King* brings forth the additional dimension of hierarchy, along with a specific role within that hierarchy—the sun being at the center of the then-known universe.

Metaphors gain poetic value when they *put into question* the conventional approaches of reasoning about, and with, a certain target domain. Here is perhaps the most obvious manifestation of the *non-obvious* re-

[36] From *Romeo and Juliet*, Act 2, Scene 2

quirement, since it quite literally proposes something that is unexpected from a conventional standpoint. When Camus describes Tipasa's countryside as being *blackened from the sun*[37], it subverts our pre-conceptions about what the countryside is, what the sun does, and hints at a semantic depth which would go on to support a whole philosophical thought (*la pensée de midi*). Interestingly, the re-edition of L'Étranger for its 70th anniversary can itself be seen as a form of poetic metaphor, since it was published under Gallimard's *Futuropolis* collection. While the actual *Futuropolis* doesn't claim to focus on any sort of science-fiction publications, and rather on illustrations, the very name of the collection applies onto the work of Camus, and of the others published alongside him, can elicit in the reader a sense of a kind of avant-gardism that is still present today.

Finally, poetic metaphors *compose* multiple metaphors into one, drawing from different source domains in order to extend, elaborate, or question the original understanding of the target domain. Such a technique of superimposition creates semantic depth by layering these different approaches. It is particularly at this point that literary criticism and hermeneutics appear to be necessary to expose some of the threads pointed out by this process. As an example, the metaphor of Charles Bovary's cap, a drawn-out metaphorin Flaubert's work which ends up depicting something which clearly isn't a cap, operates by extending the literal understanding of how a cap is constructed, elaborating on the different components of a hat in such a rich and lush manner that it leads the reader to question whether we are still talking about a hat. This metaphorical composition can be interpreted as standing for the orientalist stance which Flaubert takes vis-à-vis his protagonists, or for the absurdity of material pursuit and ornament[38], or for the novel itself, whose structure itself is composed of complex layers, under the guise of banal appearances. Composed metaphors highlight how they exist along *degrees of meanings*, from the conventional to the poetic,

[37] *"A certaines heures, la campagne est noire de soleil"*, from *Noces à Tipasa*

[38] Which ultimately leads Emma to her demise.

and further to the non-sensical. This difference of degree, rather than of kind, is one I ascribe to when it comes to delimiting corpus of the present research in different ways of writing and reading code—writing code as poetry, as tool, as a hack or as research aren't absolutely siloed off from each other.

Through these, Lakoff and Johnson highlight how metaphors *function*, and how they can be identified. Another issue they address is that of the *role* they fulfill in our everyday experiences as well as in our aesthetic experiences. Granted a propensity to structure, to adapt, to reason and to induce value judgment, metaphors are ultimately seen as a means to comprehend the world. By importing structure from the source, the metaphor in turn creates structure in our lives, in our understandings (and thus have power over us). Our understanding grasps these structures through their features and attributes (one might even call them affordances, following Gibson[14]), and integrates them as a given—in what Ricoeur would call a *dead* metaphor. This is one of their key contribution, that metaphors have a function which goes beyond an exclusive, disinterested, self-referential, artistic role. If metaphors are ornament, it is far from being a crime, because these are ornaments which, in combining imagination and truth, expand our conceptions of the world by making things *fit* in new ways.

This approach of beauty as means to understand however predates Lakoff and Johnson. Through his contribution to aesthetic philosophy, Monroe Beardsley's started touching upon metaphor from a semantic perspective. Published alongside his inquiries into the aesthetic character of an experience, and taken later on by Ricoeur as a basis for his study, *The Metaphorical Twist* implies that semantics and aesthetics might be connected through the structuring operation of the metaphor—that which elicits an aesthetic experience can do so through the creation of unexpected, or previously unattainable meaning. Ricoeur's theory of the metaphor indeed builds on Beardsley's conception that metaphor can have a designative role (the primary subject) which adds a *"local texture of irrelevance"*, a

"foreign component", whose semantic richness might over-reach and obfuscate the intended meaning, as well as a connotative one (the secondary subject), in which meaning is peripheral. For Ricoeur, it is indeed literary criticism, beyond logical grammar and linguistics, which hold the key to understanding metaphors. Through an analysis of Beardsley's work, he highlights the metaphor-induced tension, between central and periphery, between illuminating and obfuscating, between evidence and irrelevance.

As Beardsley inquiries into the features necessary for an aesthetic experience, of which the metaphor is part, he lists five criteria to distinguish the character of such an experience. Besides object-directedness, felt-freedom, detached-affect and wholeness, is the criteria of *active discovery*, which is

> "a sense of actively exercising the constructive powers of the mind, of being challenged by a variety of potentially conflicting stimuli to try and make them cohere; exhilaration in seeing connections between percepts and meanings; a sense of intelligibility"[39]

As such, Beardsley highlights the possibility of an aesthetic experience to make understandable, to unlock new knowledge in the beholder, and he considers metaphors as a way to do so. The stages he lists go from (1) the word exhibiting properties, to (2) those properties being made into meaning, and finally into (3) a staple of the object, consolidating into (or dying from becoming) a commonplace. This interplay of a metaphor being integrated into our everyday mental structures, of poetry bringing forth into the thinkable, and in metaphor creating a tension for such bringing-forth to happen, makes the case for at least one of the consequences of an aesthetic experience, and therefore one of its functions: making sense of the complex concepts of world.

[39] The Aesthetic Experience, in The Aesthetic Point of View[15].

More recent work in aesthetics and literary research have continued in this vein. Building on the focus on conceptual structures, the attention has shifted to the relationship between literature (as part of aesthetic work and eliciting aesthetic experiences) and cognition. This move starts from the limitation of explaing "art for art's sake", and inscribing it into the real, lived experiences of everyday life mentioned above, perhaps best illustrated by the question posed in Jean-Marie Schaeffer's eponymous work—*Why fiction?*. Indeed, if literary and aesthetic criticism are to be rooted in the everyday, and in the conventional conceptual metaphors which structure our lives, our brains seem to be the lowest common denominator, and thus a good starting point for a new contribution to understanding the arts. A similar approach can be seen in Michael Polanyi's work on tacit knowledge, in which that which the scientist knows isn't entirely and absolutely formal and abstracted, but rather embodied, implicit, experiential. This limitation of codified, rigorous language when it comes to communicating knowledge, opens up the door for an investigation of how literature and art can help with this communication, while keeping in mind the essential role of the senses and lived experience in knowledge acquisition (i.e. integration of new conceptual structures)[16].

Some of the cognitive benefits of art aren't too dis-similar to those posed by Beardsley, but shift their rationale from hermeneutics and criticism to cognitive science. These benefits can be pleasure, emotion, or understanding. Terrence Cave focuses on the latter when he says that literature *"allows us to think things that are difficult to think otherwise*. Such a possibility has been examined from two perspectives: in terms of the role of imagination, and in terms of the role of the sense.

Harris posits that literature is an object of knowledge, a creator of knowledge, and that it does so through the interplay between rational thought and imaginative though, between the "counterfactual imagination" and our daily lives and experiences. Through this tension, this suspension of disbelief, but nonetheless accompanied by an epistemic awareness. Working on

literary allusions, Ziva Ben-Porat shows this simultaneous activation of two texts is influenced by several factors. First, the form of the linguistic token itself has a large influence over the understanding of what it alludes to. Its aesthetic manifestation, then, is modulating the conceptual structures which will be acquired by the reader. Second, the context in which the alluding token(s) appears also influences the correct interpretation of such an allusion, and thus the overall understanding of the text. This contextual approach, once again hints at the change of scale that Ricoeur points in his shift from the word to the sentence. Finally, a third factor is the personal bagage (a personal encyclopedia) brought by the reader. Such a bagage consists of varying experience levels, of quality of the know-how that is to be activated during the reading process, and of the cognitive schemas that readers carry with them. Imagination in literary interpretation, then, seems to be relying on various aspects, from the very concrete form and choice of the words used, to the unspoken knowledge structures held in the reader's mind, themselves depending on varied experience levels.

The work of imagination relies on how the written word can elicit the recall of sensations. This takes place through the recreation, the evokation of sensory phenomena in linguistic terms, such as the *perceptual modeling*[40] of literary works, which she defineds as simulations relying on the senses to communicate situations, concepts, and potential realities. As such, literature unleashes our imaginary by recreating sensual experiences—Lakoff even goes as far as saying that we can only imagine abstract concepts if we can represent them in space[41]. It seems that recreating sensations is a way to trigger the imaginative process, and suggests the fitness of the schemas depicted. By depicting situations that, while fictional, nonetheless are pos-

[40] Elane Scarry's expression

[41] Geoff Hinton, pioneer of modern deep-learning, has reportedly said that, to visualize 100-dimensional spaces, one should first visualize a 3-dimensional, and then "shout 100 really really loud, over and over again", source: `https://medium.com/artists-and-machine-intelligence/a-journey-through-multiple-dimensions-and-transformations-in-space-the-final-frontier-d8435d81ca51`

sible in a reality often very similar to the one we live in, it is easy for the reader to connect and understand the point being made by the author. So if literature is an object of knowledge, both sensual and conceptual, offering an interplay between rational and imaginative thought, it still relies on the depiction of mostly familiar situations (the protagonists physiologies, the rules of gravity, the fundamental social norms are rarely challenged). A first issue that we encounter here, in trying to connect source code and computing to this school of thought, is that code has close to no sensual existence. In trying to communicate concepts, states and processes related to code and computing, and unable to depict them by their own material and sensual properties, we once again resort to metaphor.

3.2 In Software

It is interesting to consider that the first metaphor in computing might be concommitant with the first instance of modern computing—the Turing *machine*. While Turing machines are widely understood as being manifested into what we call computers (laptops, tablets, smartphones, etc.), and thus definitely within the realm of machines, the Turing machine isn't strictly a machine *per se*. Rather, it is more accurately defined as a mathematical model which in turn defines an abstract machine. Humans can be considered Turing machines (and, in fact, one of the implicit requirements of the Turing machine is that, given enough time and resources, a human should be able to compute anything that the Turing machine can compute), and non-humans can also be considered Turing machines[42]. Debates in computer science related to the nature of computing[17] have shown that computation is far from being easily reduced to a simple mechnical concern, and the complexity of the concept is perhaps why we ultimately revert to metaphors in order to better grasp them.

[42] See research in biological computing, using DNA and protein to perform computational tasks

Jumping ahead to the 1980s, these uses of metaphors became more widespread and entered public discourse once personal computing became available to ever larger audiences. With the release of the XEROX Star features of the computer which were until then described as data processing were given a new life in entering the public discourse. The Star included technological innovations such as a bitmapped display, a two-button mouse, a window-based display including icons and folders. For instance, the desktop metaphor relies on previous understanding on what a desktop is, and what it is used for in the context of physical office-work; since early personal computers were marketed for business applications (such as the Start), these metaphors built on the broad cognitive structures of the user-base in order to help them make sense of this new tool. Paul DuGay, in his cultural study of the Sony Walkman, makes a similar statement when he describes the Sony Walkman, a never-before-seen compound of technological innovations, in terms of pre-existing, and well-established technologies[18]. The icon of a floppy disk for writing data to disk, the sound of wrinkled paper for removing data from disk, the designation of a broad network of satellite, underground and undersea communications as a cloud, these are all metaphors which help us make sense of the broad possibilities brought forth by the computing revolution.

The work of metaphors takes on an additional dimension when we introduce the concept of interfaces. As permeable membranes which enable (inter)actions between the human and the machine, they are essential insofar as they allow for different kinds of agency, based on different degrees of understanding. Departing from the physically passive posture of the reader towards an active engagement with a dynamic system, interfaces highlight even further the cognitive role of the metaphor. These depictions of things-as-other-things influence the mental model which we build of the computer system we engage in. For instance, the prevalent windows metaphor of our contemporary desktop and laptop environments obfuscates the very concrete action of the CPU (or CPUs, in the case of multi-core architecture) of

executing one thing at a time, except at speeds which cannot be intuitively grasped by human perception. Alexander Galloway 's work on interfaces as metaphorical representations suggests a similar concern since it is based on Jameson's theory of cognitive mapping. While Jameson uses it in a political and historical context, the heuristic is nonetheless useful here: cognitive mapping is the process by which the individual subject situates himself within a vaster, unrepresentable totality, a process that corresponds to the workings of ideology. Substituting ideology with the computer[43], we can see how such a process helps make sense of the unthinkable, of that which is too complex to put into symbols (words, icons, sounds, etc.).

Moving away from userland, in which most of these metaphors exist, we now turn to examine the kinds of metaphors that are used by programmers and computer scientists themselves. Since the sensual reality of the computer is that it is a high-frequency vibration of electricity, one of the first steps taken to productively engage with computers is that of abstraction. The word computer itself can be considered as an abstraction: originally used to designate the women manually inputting the algorithms in room-scale mainframes, the distinction between the machine and its operator was considered to be unnecessary. The relation between metaphor and abstraction is a complex one, but we can say that metaphorical thought requires abstraction, and that the process of abstraction ultimately implies designating one thing by the name of another (a woman by a machine's, or a machine by a woman's), being able to use it interchangeably, and therefore lowering the cognitive friction inherent to the process of specification.

This need to get away from the specificities of the machines has been one of the essential drives in the development of programming languages. Since we cannot easily and intuitively deal with binary notation to represent complex concepts, programming helps us deal with this hurdle by present-

[43]The relation between which has been explored by Galloway, Chun, Holmes and others, and is particularly apparent in how an operating system is designated in French: *système d'exploitation*.

ing things in terms of other things. Most importantly, we represent binary signs in terms of English language. This is by no means a metaphorical process, but rather an encoding process, in which tokens are being separated and parsed into specific values, which are then processed by the CPU as binary signs. Nonetheless, as the abstraction layer offered by programming languages allowed us to focus on *what* we want to do, rather than on *how* to do it. This is where we can see a metaphor come in, as the possibility to deal with more complex concepts required us to grasp them. Allen Newell and Herbert A. Simon, in their 1975 Turing Award lecture, offer a good example of symbolic (i.e. conceptual) manipulation relates inherently to understanding:

> In none of [Turing and Church's] systems is there, on the surface, a concept of the symbol as something that *designates*.

The complement to what he calls the work of Turing and Church as automatic formal symbol manipulation is to be completed by the process of *interpretation*, which is defined simply as the ability of a system to designate an expression and to execute it. We encounter here one of the essential qualities of programming languages: the ambivalence of the term *interpretation*. A machine interpretation is clearly different from a human interpretation: in fact, most people understand binary as the system comprised of two numbers, 0 and 1, when really it should be intepreted as a system of two distinct signs (red and blue, Alex and Max, hot and cold, etc.). To assist in the process of human interpretation, I argue that metaphors have played a part in helping programmers construct accurate mental representations related to computing. These metaphors can go both ways: helping humans understand computing concepts, and to a certain extent, helping computers understand human concepts.

Perhaps one of the first metaphors a programmer encounters when learning about the discipline is that which states that the function is like a kitchen recipe. You specify a series of instructions which, given some

input ingredients (arguments), result in an output result (return value). Explaining the need for a *void* keyword to individuals with limited experience and knowledge of how programming works is a good example of the non-straightforwardness of computing concepts. Similarly, the use of the term *server* is conventionally associated and represented as a machine sending back data when asked for it, when really it is nothing but an executed script or process running on said machine.

Another instance of synbolic use relying on metaphorical interpretation can be found in the word *stream*. Originally designating a flow of water within its bed, it has been gradually accepted as designating a continuous flow of contingent binary signs. *Memory* stands for record, and is stripped down of its essentially partial, subjective and fantasized aspects usually highlighted in literary works (perhaps *volatile memory* gets closer to that point). *Objects*, which came to prominence with the rise of object-oriented programming, have little to do with the physical properties of objects, with no affordance for being traded, for acting as social symbols, for gaining intrinsic value, but rather the word is used as such for highlighting its boundedness.

Most of these designations, stating a thing in terms of another aren't metaphors in the full-blown, poetic sense, but they do hint at the need to represent complex concepts into humanly-graspable terms. The need for these is only semantic insofar as it allows for an intended interaction with the computer to be carried out successfully—one has an intuitive understanding that interrupting a stream is an action which might result in incompleteness of the whole. Furthermore, this process of linguistic abstraction doesn't actually require clear definitions for the concepts involved. The example of the terminology in modern so-called cloud computing uses a variety of terms stacked up to each other in what might seem to have no clear *denotative* meaning (e.g. Google Cloud Platform offers *Virtual machine compute instances*), but nonetheless have a clear *operative* meaning (e.g. the thing on which my code runs). This further qualifies the complexity

of the sense-making process in dealing with computers: we don't actually need to truly understand what is precisely meant by a particular word, as long as we use it in a way which results in the expected outcome[44].

The reverse process also brings forth issues of conceptual representation through formal symbolic means. The work of early artifical intelligence researchers consists not just in making machines perform intelligent tasks, but also implies that intelligence itself should be clearly and inambiguously represented. The work of Terry Winograd, for instance, was concerned with language processing (intepretation and generation)[19]. Through his inquiry, he touches on the different ways to represent the concept of language in machine-operational terms, and highlights two possible represenations which would allow a computer to interact meaningfully with language. He considers a *procedural* representation of language, one which is based on algorithms and rules to follow in order generate an accurate linguistic model, and a *declarative* representation of language, which relies on data structures which are then populated in order to create valid sentences. At the beginning of his exposé, he introduces the historically successive metaphors which we have used to build an accurate mental representation of language (language as law, language as biology, language as chemistry, language as mathematics).

Metaphors are implicitly known not to be true in their most literal sense. Max Black in *Models and Metaphors* argues that metaphors are too loose to be useful in analytic philosophy, and therefore too lose for programming languages, heavily based on the analytic tradition. Yet, they still rely heavily on models in order to make human concepts graspable and operation to the computer. These tools deployed during the representational process differ from conventional or public metaphors insofar as they can be logically operated upon and therefore empirically verifiable or falsifiable. These models are means through which we aim at taking the conceptual

[44] See the famous comment in the UNIX source: *You are not expected to understand this*.

structures on which metaphors operate, and explicit them in formal symbol systems[45].

Abstraction, metaphors and synmbolic representations are thus abundant when it comes to computing, both in terms of trying to represent to ourselves what it is that a computer can and effectively does, but also in terms of explaining to the computer what it is we're trying to operate on (from an integer, to a non-ASCII word, to a renewable phone subscription or to human language).

3.3 Computers and psychology

So metaphors work in software because they do not exist just within literature, but yet remain too vague for a strict computer interpretation. Such a computer interpretation, in turn, is too complex and fine-grained for most individuals interacting with them (from end-users to most programmers and computer scientists) to be useful. The conclusion we establish here, is about how connections between mental models relate to the process of understanding, at the overlap between human understanding and computer understanding, and how aesthetic experience can affect this encounter.

The mental model offers a good starting point for exploring this overlap. A mental model, as a kind of internal symbolic representation of external reality, is a more rigorous and formal conceptual structure than a metaphor—which only offers a broad direction, rather than an actionable basis. They are usually related to knowledge, since the construction of accurate and useful mental models through the process of understanding underpins knowledge acquisition. However, mental models need not be correlated with empirical truth, but extensive enough to be described by logical means. Mental models can be constructed or further qualified by the use of metaphors, but they are nonetheless more precise than the

[45]For a further inquiry of models and theories, see Weizenbaum in *Computer Power and Human Reason*

cognitive structures on which metaphors rely—a mental model can be seen as a more specific instance of a conceptual structure. The term *schema*, used in cognition-influenced literary studies, again following Lakoff, offers a point of entry into the psychology of computer programming.

Francoise Détienne, in her study of how computer programmers design and understand programs, defines the activity of designing programs in activating schemas, mental representations that are abstract enough to encompass a wide use (web servers all share a common schema in terms of dealing with requests and responses), but nonetheless specific enough to be useful (requests and responses are qualitatively different subsets of the broader concept of inputs and outputs). This flexibility is useful when one needs to deal with two aspects of working within a programming environment. The computer's actions and responses are comprised of the prescriptive (what the computer should do) to the effective (what the computer actually does), one of the tensions at the heart of computer programming.

Within a given context—which include goals and heuristics—, elements are being perceived, processed through existing knowledge schemas in order to extract meaning. Starting from Kintsch and Van Dijk's approach of understanding text[46], she nonetheless highlights some differences. In program texts, there is an entanglement of the plan, of the arc, of the tension, which does not happen so often in most of the traditional narrative text. They are also dynamic, procedural texts, which exhibit complex causal relations between states and events. Finally, the understanding of program text is first a general one, which only subsequently applies to a particular situation (a fix or an extension needing to be written), while narrative texts tend to focus on specific instances of protagonists, scenes and descriptions.

A similarity in understanding program texts and narrative texts is that

[46] to expand

the sources of information for understanding either are: the text itself, the individual experience and the broader environment in which the text is located (technical, social). Building on Chomsky, the activity of understanding in programming can be seen as understanding the *deep structure* of a text throuh its *surface structure*. One of the heuristics deployed to achieve such a goal is looking out for what she calls *beacons*, as thematic organizers which structure the reading and understanding process. However, one of the questions that isn't answered specifically, and which is the aim of this thesis, is to highlight how does the surface structure in programming result in the understanding of the deep structure.

INCLUDE MIT STUDY

Going back to research in contemporary literary studies can start laying out threads of an answer. Jérôme Pelletier uses Carl Plantinga to define emotional responses in the face of aesthetic objects as dual: either one has an emotional response to the artefact itself (surface), of an emotional response to what it represents (deep). In the context of reading fiction, the reader is helped in their understanding by looking out for *guides* or *props*[47], which are similar to the *beacons* emphasized by Detienne. A notable difference is that the guides are suggested, implied, left as traces for the reader to subtly construct (as in the case of the cap metaphor in *Madame Bovary*), rather than explicitly stated throughout the program text (most obviously in the form of comments). However, we've seen previously that the use of comments is, by most programmers, not considered to be an aesthetic feature of an inspected source code, hinting at the fact that subtlety (useful subtlety), might be a desired attribute of beautiful code.

Programming is then fiction, in that the pinpointing of its source of existence is difficult, and in that it affords the experience of imagining contents of which one is not the source, and of which the certainty of which isn't defined. Furthermore, both programming and fiction suggest surface-

[47] Currie, 1990

level guiding points helping the process of constructing mental models and conceptual representations. It is also non-fiction, in that it deals with concrete issues and problems (more often than not, a pestering bug), and that it provides a pragmatic frame for processing representations, in which assumptions stemming from burgeoning mental models can be easily verified or falsified. It might then be appropriate to treat it as such, simultanesouly fiction and non-fiction.

To conclude this section, then, we can turn to Jerome Bruner, who considers that art allows us to *"reading in others' minds*, to anticipate what a writer has been intending for us to understand through their text, eithe program or narrative. This intent component relates to the interpretation issue mentioned above: the interpretation of the machine is different from the interpretation of the human, and therefore what also needs to be intepreted is the intent of the author. Reading is then akin to constructing a *cognitive cartography*, allowing for an experience to be made intelligible, sensible. The repeated implication of spatial and visual components of metaphors and mental models allows us to consider metaphors as an architecture of thought. The next section is therefore dedicated to examining more closely the parallels between software architecture and physical architecture, and examining how the aesthetic standards of the latter could apply to the aesthetic standards of the former.

4 Architecture

This section is mostly based on the work of *patterns*, both in softdev and in architecture, and how they can be considered both functional and beautiful; furthermore I will develop on how these two relate (through habitability, QWaN).

4.1 General software architecture

Emergence of the field, planning vs. non planning.

Planning as a consequence of the structured revolution.

Non-planning as a consequence of low-barrier to contribution.

4.2 Beauty in architecture

Highlight how uncertain, elusive it is. And then focus a big part on Alexander's work.

Movement of people is also some kind of structure

4.3 Applying architectural beauty in software

A lot of Gabriel's work, his connection to poetry, and also looking at how it applies well (softdev), and not so well (code poetry → because the functionality of code poems isn't quite so obvious as the functional use of a building, or of a program)

Transition with the case of the case of hacking: code that is not meant to be read.

Patterns as Software Design Cannon, Paul Taylor: "The time and effort invested inmastering software design alternatives is generally appreciated, but by contrast, architects of the physical worldassume that they can understand and digest canonical works from two-dimensional magazine images and brief commentaries from eloquent observers."

Downton (1998): ”This last point is particularly important—buildings and design are often judged from artisticperspectives that bear no relation to how the building’s occupants perceive or occupy the building.”

Desgin patterns: one subset of these is the aesthetic, and the one i need to drill on. Not all patterns are beautiful, but according to alexander, they help make something ”good” (intrinsically linked to the beautiful)

5 Hacking

Is there beauty in inscrutability? Particularly, this redirects to the understanding of the machine (e.g. trying to reduce character counts for one-liners).

5.1 History of hacking

And a disambiguation of the term.

5.2 Unreadability and aesthetics

This shows you need a degree of expert knowledge in order to appreciate it.

Example of the one-liners.

Example of the demoscene.

5.3 Hacking and/or the lack of beauty

Why is there ne beauty in hacking? Does there need to be beauty anywhere?

There doesn’t seem to be so much because it’s so much focused on what the machine understands; the real beauty is how they make the humans *realize* what the machine *really* understands.

6 Machine understandings

This is about whether or not the machine really understands anything.

UNIX → you are not expected to understand this

6.1 Computation

What is a computer? what is computation? There are differring approaches, and I should highlight what is mine (that of the formal, symbolic system). Cantell-Smith can be good here, if alone to explain that things are complicated to explain, but nonetheless intuitive to any serious practitioner of CS.

6.2 Programming languages

First, by what makes a PL (a bit of history, a bit of theory)

Second, what makes a good PL. Also: the idiom!

Third, esoteric languages.

Also, moods! Imperative programming creates a mood: `https://en.wikipedia.org/wiki/Imperative_mood`

6.3 Formal systems

Start by what is a formal system, then show two consequences: first, it fits with goodman and second, it actually seems to have trouble with meaning.

7 Programming Semantics

Answer the question as to whether they have semantics (they do according to the specs, but it's just another *décalage*)

7.1 Semantics in PL

Approach it first in a very practical way (parse trees, tokens, environments), and then in a more theoretical way (what are we even trying to communicate?)

7.2 Concepts of PL

The issue actually comes from the fact that some concepts might be very foreign and hard to communicate (which is why programming can be hard, and how the whole section has shown relative limitations of doing so).

And so this is why we need aesthetics: to communicate both easy and complicated things, to reduce friction as much as possible. Example: Alan Perlis's Epigrams are poem-like sentences.

Another example if thread concurrency. The book on parallel programming mentions an example of parallel thread processing which looks beautiful but is ugly on the inside: https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e2.pdf p. 105

8 Conclusion

Sum up a bit all that has been said.

aesthetics matter, even in such a highly formal, syntactical, autotelic system as a computer. they connect a surface-structure with a deep structure.

how does it contribute to the world? by showing that there is no separate domain of aesthetics, but also that they're not essential, but a mark of high-quality and, again *that they allow us to understand*.

Recap on some of the concepts:

- semantic compression - spatio-visual problem solving

Finishing on the MIT study and Goodman:

> The emphasis placed on the symbolic, cognitive, planning aspects of the arts leads us to give value to the role played by problem-solving, seeing there a model in terms of which the moment-to-moment artist's behavior at work can be described. "An analysis of behavior as a sequence of problem- solving and planning activities seems to be most promising [...]" (goodman)

8.1 Next steps

Close-reading of more source code.

Gathering of more hacking resources and computer science/abstract resources.

Actually tie in the analysis of source code poems more tightly with aesthetics (denotation, depictions, etc.)

> A work of art has a character and a content, including formal (balance and unity), aesthetic (gracefulness, garrishness), expressive (melancholy, cheerfulness), representational (a woman, a data structure), semantic (meaning, metaphor), and symbolic (death, life, disintegration) (budd, aesthetic essence).

References

[1] Francoise Detienne. *Software Design – Cognitive Aspect*. Springer Science & Business Media, December 2012.

[2] Will Crowthers. *Colossal Cave Adventure*. 1977.

[3] Florian Cramer. *Words Made Flesh*. Piet Zwart Institute, 2003.

[4] Olga Goriunova and Alexei Shulgin. *Read Me: Software Art & Cultures*. Aarhus University Press, Aarhus, 2004th edition edition, December 2005.

[5] VILÉM FLUSSER and Rodrigo Maltez Novaes. *On Doubt*. University of Minnesota Press, 2014.

[6] Sharon Hopkins. Camels and Needles: Computer Poetry Meets the Perl Programming Language. *Usenix Technical Conference*, 1992.

[7] Camille Paloque-Bergès. *Poétique des codes sur le réseau informatique*. Archives contemporaines, 2009.

[8] Philippe Bootz. *The Problem of Form Transitoire Observable, A Laboratory For Emergent Programmed Art*. 2005.

[9] Geoff Cox and Christopher Alex McLean. *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press, 2013.

[10] I. Bogost. *The Rhetoric of Video Games*. 2007.

[11] Jeremy Tirrell. Dumb People, Smart Objects: The Sims and The Distributed Self. In *The Philosophy of Computer Games Conference*, 2012.

[12] George Lakoff. *Metaphors We Live By*. University of Chicago Press, 1980.

[13] Paul Ricoeur. *The Rule of Metaphor: The Creation of Meaning in Language*. Psychology Press, 2003.

[14] James Jerome Gibson. *The Ecological Approach to Visual Perception*. Psychology Press, 1986. Google-Books-ID: DrhCCWmJpWUC.

[15] Monroe C. Beardsley. The Aesthetic Point of View*. *Metaphilosophy*, 1(1):39–58, 1970. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-9973.1970.tb00784.x.

[16] Michael Polanyi and Amartya Sen. *The Tacit Dimension*. University of Chicago Press, Chicago ; London, revised ed. edition edition, May 2009.

[17] William J. Rapaport. Philosophy of Computer Science: An Introductory Course. *Teaching Philosophy*, 28(4):319–341, 2005.

[18] Paul du Gay, Stuart Hall, Linda Janes, Anders Koed Madsen, Hugh Mackay, and Keith Negus. *Doing Cultural Studies: The Story of the Sony Walkman*. SAGE Publications Ltd, Los Angeles, CA, second edition edition, June 2013.

[19] Terry Winograd. *Language As a Cognitive Process: Syntax*. Addison-Wesley, Reading, Mass, May 1982.