`print("Hello, Artworld!");`

Nicholas Rakita

January 2020

Contents

Introduction

Since the dawn of computing in the 1960's information processing systems have been utilized with great effect for the creation of art. This usage has necessitated the design and implementation of programs by artists in order to produce such works, and as a result programming is today regarded as an important tool in the arsenal of the contemporary artist. Regrettably these programs have largely been regarded merely as a means to an end, wherein the final art piece is considered the result of said programs and not the programs themselves. Simultaneously, code with little practical use has long been written by computer scientists, and these programs despite their obvious aesthetic value have also largely been regarded simply as curiosities or academic exercises. As the growing field of Software Studies suggests, programs rather than being mere feats of isolated engineering should instead be recognized as cultural artifacts deeply connected to the historical, social, and personal contexts in which they are created. Nick Montfort summarized this matter succinctly, writing that "code is a cultural resource, not trivial and only instrumental, but bound up in social change, aesthetic projects, and the relationship of people to computers" (Montfort et al 2013, p8). In light of this fact, it becomes not only possible, but indeed necessary to develop discourse regarding the aesthetic nature of code. The following text attempts to define and elucidate the properties of code-art, a fledgling medium consisting of programs understood to be complete artworks in their own right. Further, methodologies for the interpretation of code-art are described and utilized on a number of examples to facilitate the realization of programs as independent aesthetic objects. Comparison with more traditional mediums have also been addressed in order to illustrate the connections between code-art and the broader history and methodologies of the art world.

The foundation of programmatic expressivity lies in the existence of choice in the way a program may be written. Because of this, both approaches towards interpretation discussed in this text can be understood in regards to the choices a programmer has made in the construction of a given piece of code-art and the consequential properties these choices entail. One such set of properties relate to the syntactic nature of the program, being the literal manner in which the program was written. This interpretative paradigm acknowledges the fact that programs are essentially textual objects authored within the confines of strict formal grammars. Another set of properties worth considering are those relating to the semantic nature of the program, or in simpler terms, the function of the program and the manner in which it runs. In contrast, this paradigm emphasizes the duality of programs not simply as text, but also as representations of abstract machines whose architecture and purpose are of aesthetic interest. It is hard to frame either of these paradigms in terms of singular interpretive techniques. For example, a program's syntactic properties could be investigated on purely formal terms, or instead in regards to their conceptual commentary towards cultural practices within the industry. Because of this it is more accurate to conceptualize these paradigms as categorical umbrellas under which fall a

variety of methodologies for the investigation of code-art whose commonality is the target of their interest, being either syntactic or semantic. A plethora of examples demonstrating both syntactic and semantic aesthetics have been included in this text to highlight the incredible diversity of forms present in the medium and investigate the unique aesthetic properties they possess.

Perhaps the most important consideration to address before diving into the specifics of code-art are the motivations for creating and interpreting it in the first place. Just as the origins of code-art lie both in the worlds of fine art and computer science, the value of this medium can also be understood in relation to these two fields. For fine art the inclusion and investigation into aesthetic programming is relevant in that it represents an entirely new medium from which novel approaches to art making can be explored. The reality of a global society with an ever increasing utilization of computational technology necessitates the inclusion of this medium into contemporary art practices by virtue of its unrivaled ability to comment on, aestheticize, and provide understanding of the digital computer. Moreover, code-art also represents an important subject to the future of computer science and industrial programming. The integration of code and aesthetics offers substantial benefits to practical programming in much the same way that architecture and engineering have historically profited from influences within the art world (Fishwick 2002, p383). From the perspective of both the artist and the programmer code-art can be understood as a promising new source of innovation, inspiration, and novel perspectives.

On the Aesthetics of Syntax

"Computers are like Old Testament gods;
lots of rules and no mercy."

— Joseph Campbell

A fundamental distinction between programming and natural languages can be found in the nature of their syntax. Natural languages being objects of cultural evolution can be seen to have developed incrementally over the course of human history. Because their primary function is to facilitate communication between humans, their grammars and vocabularies can be generally understood as sets of common elements to allow for meaning to be expressed between individuals. Ambiguity within natural languages is not only possible, but common. While this can result in occasional misunderstandings, it overall represents little issue for natural language speakers. In contrast programming languages are intentional linguistic constructions, which because of their function to specify computational mechanisms, necessitate unambiguous expression. The syntax of programming languages is therefore based on formal grammars borrowed from abstract mathematics which describe exactly how expressions may be formed. The rationale for specifying programming languages with formal grammars is twofold. Unambiguity is first an essential property if compilers and interpreters are to be written for the language, as without certainty about the meaning of an expression it would be impossible to correlate it with specific computational operations. Further, by adhering to such rigid grammars programming languages allow for communication of the delicate conceptual machinery specific to an algorithm between programmers with little risk of misunderstandings concerning their functioning. With all this being said, considering the importance of syntax within programming it is only natural that it forms an important avenue towards expressivity in code-art. Various uses of syntax can be found in code-art, such as using a language's syntax as a creative framework or abusing the limits of a syntax to purposefully thwart easy understanding of a program's functioning. The syntactic structure of programming languages can be seen as a unique form of textual expression complimentary to natural languages, and their introduction into the world of fine art therefore necessitates new literacies which can aid in their interpretation (Cox & McLean 2012, p21).

In fact, formal constraints such as code's syntax are hardly a new phenomena in the art world. An example of a strict methodological approach towards painting can be found in the works of the Dutch Neoplasticists. Using a rigid set of prescriptive rules towards colors and shapes the works of Mondrian and van Doesburg can be understood as exercises in combinatorial aesthetics (not unlike what is found in code-art), where the creative act is specifically manifested through the arrangement of predetermined forms. Another example of constraint systems in the art world can be found in the work of the French Oulipo group,

who utilized strict textual rules to explore novel structures in their poetry and literature. Some of the textual restrictions they experimented with include palindromes (texts which read the same backwards and forwards), lipograms (texts written with the exclusion of specific letters), and snowball poetry (in which each line of the poetry is one letter longer than the previous). Rather than seeing these restrictions as limitations on their work, Oulipo artists understood them instead to be tools to enhance the creativity of their writing. As a final example, the highly precise nature of code's textual features (such as indentation, alignment of elements, and capitalization) can also be seen to relate to the practice of concrete poetry, in which textual elements are arranged to evoke meanings beyond their literal definitions. Concrete poetry can be found in a vast array of artistic movements, for example the Futurist sound poem "Zang Tumb Tumb", or the various Dada poetry of Kurt Schwitters. The Letterist practice of hypergraphics can also be seen as an extension of this technique. A commonality between all concrete poetry and syntactically oriented code-art is the intentional acknowledgement of the visual nature of text and the exploration of its resulting aesthetics properties.

Code Poetry

Code poetry is the act of writing poems within the confines of a programming language's syntax. The earliest examples of this variety of code-art can be found to originate on Usenet during the early 1990's, mostly utilizing the Perl programming language. The challenge inherent in code poetry is the restriction that the poem must be valid source code, a constraint which entails the careful usage and placement of words and language symbols to avoid syntax errors. Frequently typical punctuation must be replaced by equivalent marks within the language, for example substituting periods with semicolons to terminate lines of the poem. Code poetry is not required to produce executable programs, and for this reason it can be seen as a particularly pure example of syntactic code-art. A notable early example of a code poem is 'Black Perl', a program shared frequently on early Perl message boards:

```
BEFOREHAND: close door, each window & exit; wait until time.
    open spellbook, study, read (scan, select, 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;
AFTERWORDS: tell nobody.
    wait, wait until time;
    wait until next year, next decade;
        sleep, sleep, die yourself,
        die at last
```

Fig 1: *The "Black Perl" program*

Code poetry exemplifies the fact that code is not only functional, but indeed can posses expressive or literary qualities as well (Fuller et al. 2008, p208). Moreover, code poetry provides insight into the fact that constructs within programming languages, far from being mere signifiers for abstract computations, also serve as metaphors for the personified actions they entail. In particular the tendency in code poems to creatively use built in keywords (such as 'kill', 'open', or 'warn' in the above example) highlights the fact that programming languages enable the encoding of abstract computational machinery in a manner akin to the narrative formats used in traditional forms of literature. A study in 2006 utilizing automatic textual analysis on programming documentation found substantial evidence for the idea that programmers tend to personify their code, thinking of it in terms of computational entities performing actions. Concerning this study Alex McLean remarked that "rather than finding metaphors supporting a mechanical, mathematical or logical approach as you might expect, components were instead described as actors with beliefs and intentions, being social entities acting as proxies for their developers." (McLean 2011, p33). This perspective enables an alternative understanding of the programmer not simply as an engineer, but instead as a digital playwright.

Obfuscation

Regarding the textual structuring of a program, technically the only limiting factors of what may be expressed relate to the rules of the formal grammar defining its syntax (that is, what the compiler or interpreter accept as correct programs). Despite this, many additional rules of structuring can be found in programming derived not from implicit syntax, but rather from socially constructed consensuses regarding 'good form'. An example of these rules might be indentation, which in many languages represents an entirely optional syntactic construct. Nonetheless, in general programmers adhere to conventions concerning indentation, such as the rule that nested blocks of code must always be indented further than the code they are embedded in. Conventions such as these exist because programs are not written for the sake of the machine which executes them, but rather for the sake of other humans. By formatting code in a manner which conforms with agreed upon standards (so called *clean code*) programmers ensure that their work can be understood and used by others. In direct opposition to this concept is the practice of writing *obfuscated code*, in which code is authored in a fashion meant to intentionally prevent others from understanding it.

Obfuscated programs abandon usual programming conventions such as meaningful variable names, standard indentation or whitespace, and typical line ordering in favor of structures specifically designed to obscure the means by which a program functions. The practice of writing obfuscated programs is possible in all programming languages, but a community of special notoriety in this area is the *International Obfuscated C Contest*, an annual competition started in 1984 where users of the C programming language vie to see who can submit the most notably bizarre programs. The below example from Volker Diels-Grabsch is one of the winners from 2019, and implements a compression algorithm (a fact particularly humorous due to the compressed nature of the program itself):

```
#include<stdio.h>
int main(){int a=0,b=a;long long c[178819],d=8,e=257,f,g,
h,i=d-9;for(;a<178819;){c[a++]=i;}for(a*=53;a;a>>=8)putc\
har(a);if((f=getchar())<0)return 0;for(;(g=getchar())>=0;
){h=i=g<<8^f;g+=f<<8;a=e<(512<<a%8|(a<7))||f>256?a:a>6?15
:a+1;for(;c[i]>-1&&c[i]>>16!=g;)i+=i+h<69000?h+1:h-69000;
h=c[i]<0;b|=h*f<<d;for(d+=h*(a%8+9);d>15;d-=8)putchar(b=b
>>8);f=h?g-f*256:c[i]%65536L;if(a<8*h){c[i]=g*65536L|e++;
}}b|=f<<d;for(d+=a%8;d>-1;d-=8)putchar(b>>=8);return!53;}
```

Fig 2: *Diels-Grabsch's Winning Entry*

The common factor of all obfuscation techniques is their exploration of the play possible in programming, that is, all the available means of expressing a program in a given language (Mateas & Montfort 2005, p147-148). By doing

so programmers writing obfuscated programs extend the traditional characterizations of beautiful code (elegance, simplicity, clarity of expression) with new ideals such as complexity, obscurantism, and visual appearance. In the tradition of avant-garde art obfuscated programs push the boundaries of conventional aesthetics and attempt to map the unexplored areas of their medium. Obfuscated programs are written to confuse, intimidate, and amuse their viewers, however their most important capacity is to highlight the infinity of expressive possibilities by subverting convention.

Secondary Notation

Within all programming languages there exist syntactic constructs that allow for meaningful information to be conveyed between programmers, but that are ignored entirely by computers. These constructs, for example spatial layout, comments, or variable names, are referred to as secondary notation (McLean 2011, p27). The existence of secondary notation within programming languages reinforces that idea that languages exist and are designed to accommodate programmers and not the machines which execute them. Stripped of these human aspects, code remains only esoteric strings of syntax, lacking in any value to programmers or broader human understanding of the mechanisms by which they function (Cox & McLean 2012, p23). In recognition of the importance of secondary notation a notable genera of code-art are programs which comment directly on the anthropic components of programming and the cultural values they represent.

```
/*****************************************************************
* TODO: check if debug version is used env['CACHED_JPEG'] = 1
*****************************************************************/

/**********************
* TODO: Write tests!
**********************/

/*********************************************************************
* TODO: a supprimer plus tard car pose des problemes de securite
*********************************************************************/

/***********************************************************
* TODO: use usbutils functions (need to be externalised!)
***********************************************************/

/*************************************************************
* TODO: verify if the codec is supported for video writting.
*************************************************************/
```

Fig 3: *Excerpt from "Much TODO"*

One amusing example from this genre is the program "Much TODO" written by Steven Read. A feature of nearly all programming dialects is the ability to write comments in natural language which are embedded within the code and used to indicate the function of a specific section or to record potential issues the author encountered. Another common use for these comments is to indicate pieces of the code which have yet to be written, usually denoted by the prefix 'TODO:'. To create "Much TODO" Read algorithmically collected roughly 100,000 such comments from code posted online and compiled them into a single program. "Much TODO" is indeed a syntactically valid program, however because it contains only comments its function remains purely virtual. In regards to the work's purely hypothetical functionality Read playfully claims that the program "does everything, and nothing". "Much TODO" can be understood as commentary towards the fact that programs encode meaning which exists in the space of their secondary notation and which is not limited to the confines of their mechanical components.

On the Aesthetics of Semantics

"At the heart of stored-program computing
lies the Faustian substitution of word for action."

— Wendy Hui Kyong Chun

As mentioned in the prelude of the previous chapter, the formal nature of programming languages' syntax exists necessarily so that a given program can also be executed by a computer. This section thus addresses code-art which derives its aesthetic value not from the manner in which it was written, but rather what happens when it is run, that is, its semantics. An important distinction must first be made between two varieties of semantics, one of which is relevant to code-art and one which is not. The first is mechanical-semantics, which entails the precise operations performed by a computer when a program is executed. The second is conceptual-semantics which refers to the ways in which we understand the function of these operations. The mechanical-semantics of a program are unambiguous and correspond directly to the rules of execution encoded in a programming language's compiler or interpreter. In contrast, the conceptual-semantics of a program are a product of human interpretation of its mechanical-semantics, and as such are ambiguous and open to a variety of perspectives (for the sake of brevity, in the rest of the chapter the term semantics will be used to denote the conceptual-semantics of a program, unless otherwise specified). For an example of this dichotomy, consider a program which processes a list of numbers by adding a fixed value to each of them. To the computer evaluating the program there is only one possible interpretation of this program, which is precisely equal to the function we have described (the mechanical-semantics). From the human perspective however, the function of this program depends entirely on how we interpret the list of numbers given to the program. If we take the input to be a text document (stored in computers by lists of numbers representing the ASCII codes for the letters of the text), then the program performs a Caesar cipher, a primitive encryption algorithm. If instead, however, we take the input to be a digital image (a list of numbers specifying each pixel's color), then the program can be interpreted as one which increases the brightness of an image. In the discourse of code-art the mechanical-semantics are relevant only in so much as they enable us to discuss and engage with the multitude of interpretable conceptual-semantics associated with a piece of code. The ambiguity of conceptual-semantics can be compared with and integrated into the broader characterization in post-modernism of an art piece's interpretation as a function of the viewer and not the author. Moreover, the multiplicity found in a program's conceptual-semantics allow us to approach software with paradigms borrowed from the humanities, opening the door to new understandings of these objects.

The aesthetics of code-art which utilize semantics as an expressive substrate are typically derived from a relation to the abstract notion of computation and

the formal or cultural properties associated with it. Semantic code-art involves a level of self-awareness on the part of the artist towards the relation between their programs and the mechanical processes which they entail. In the consideration of these processes emphasis is placed not simply on a program's input and output, but the ways in which it connects one to the other. Semantic code-art has the capacity to comment on a myriad of topics such as the ability for code to affect or modify its own operation, the hidden properties of computations orchestrated by the operating system, or even the very limits of what is actually computable.

The mechanical and pre-performative nature of semantic code-art can be compared to the techniques of several other artists or groups. One notable genre which bears resemblance to the medium are event scores, popularized by the Fluxus movement. Just as semantic code art is suggestive of computational enactment, event scores are themselves merely objects which suggest a performance. In this capacity they derive value from their hypothetical execution in much the same way that semantic code-art derives value from its computational implications. Art which embeds aesthetics in mechanical systems can also be related to semantic code-art. An example of this style could be the kinetic sculptures of Jean Tinguely, which embrace the beauty of mechanistic structure and automation. In particular his auto-destructive works and other pieces from this field can be seen as important conceptual precursors to many of the examples in this chapter. Semantic code-art can also be compared more broadly with the field of process art, wherein emphasis is placed on the aesthetic value of the execution or performance of an action rather than the result of this operation. The interpretation of this genre along with that of semantic code-art depends on the acknowledgement that the manner in which an operation is performed can, in itself, be beautiful.

Quines

Self-referentiality has in recent decades become a heavily utilized concept in both postmodern art and literature. Consequently, it comes as no surprise then that self-referentiality can also be found in code-art. A notable variety of programs which embody this concept are quines. The word quine was first coined by Douglas Hofstadter in his 1979 novel "Gödel, Escher, Bach" and refers to a program which when run, outputs its own source code. Another way of conceptualizing this is to view quines as self-replicating programs whose mechanical operation and the output of this operation coincide. Quines can be understood as exercises in quoting, wherein a program is able to embed the specification of its operation as a value within itself. A more complicated variation on this idea are quine-relays, which are programs written in one language which output a program in another language, which when run is capable of producing the original program. This process can be extended to any number of intermediate languages, leading to quine-relays whose execution cycles through

entire chains of self-replicating programs written in different languages. Perhaps the most impressive example of this is the "128-Language Orborous Quine":

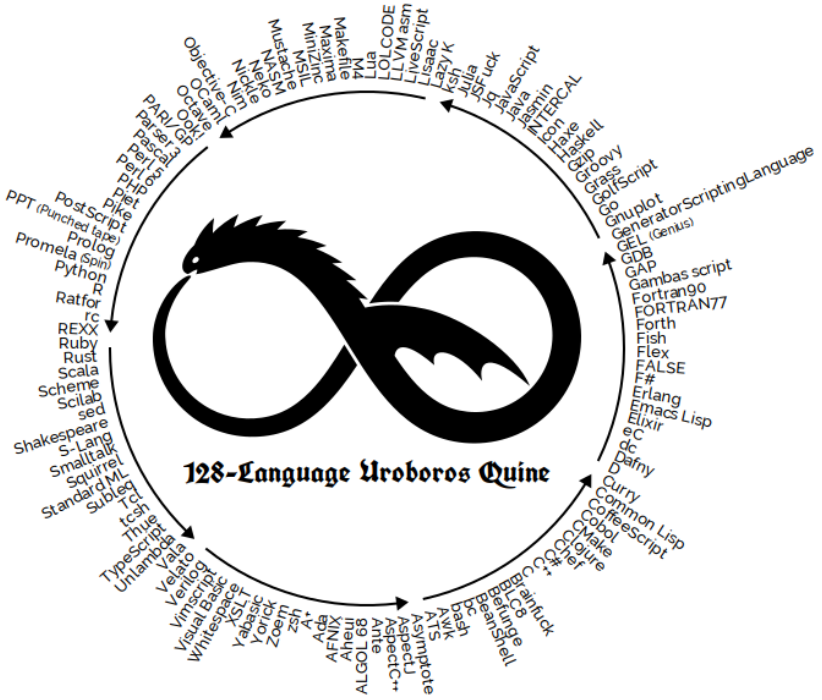

Fig 4: *Language cycle for the "128-Language Orborous Quine"*

Quines for the most part can be interpreted from a formal point of view. They serve as demonstrations of a programmers skill by necessitating clever solutions to the nontrivial problem of embedding of a program within itself. Quine-relays in particular demand a high degree of technical expertise to produce because of their essential polyglottal nature. The tradition of writing quines beyond demonstrating skill, also serves to expose the value placed in semantic cleverness within the coding community. A beautiful example of this is a winning quine entry in the 1994 IOCCC written by Szymon Rusinkiewicz which consisted of a single empty text file. The genius of Rusinkiewicz's program lies in his realization that many C compilers accept this empty program as valid and compile it into a

program which likewise produces no output, meaning it represents the smallest possible quine. This program deservedly won in the "Worst Abuse of the Rules" category.

Quines can also be interpreted as a kind of inverted analog to the field of auto-destructive art. They represent conceptual machines which, rather than destroying themselves, are able to perform a kind of recursive auto-poiesis. Moreover, the self-encoding nature of Quines highlights a powerful concept in regards to the abilities of computers, meta-programming. This term refers to the capacity for programs to operate on and manipulate other programs or even themselves. This potential underlies one of the most profound results in computer science, the halting problem, which states the impossibility of writing a program which can determine if other programs will eventually stop or keep running forever. This result, when considered philosophically has implications and analogs beyond the realm of computer science. Implications of the halting problem indicate underlying epistemological instabilities within our systems of reasoning about computation. The acknowledgement of these instabilities in computer science can be correlated with similar rejections of total or universalist paradigms in postmodernity.

Computation as a Medium

Certain pieces of code-art can be understood in regards to their ability to express or demonstrate properties of their own evaluation process. These programs seek to manifest the internal mechanistic structures of their computation, performing in essence an exorcism on the spectre of calculation which haunts our motherboards. These programs can be seen in some ways as an outgrowth of the practically oriented debugging techniques utilized by programmers for decades to elucidate and find errors in the semantics of their code. Now more than ever, computers can be seen to operate as black-boxes, wherein the internal execution of programs is hidden away from the prying eyes of users. This is partially due to the exponential increase in the number of software layers between users and the execution of code by the machine, however it is also due to intentional efforts to obscure computation for the sake of security, performance, or protection of intellectual property. Code-art represents a powerful tool in reestablishing direct observation between humans and the execution of the code they write.

A potent example of this variety of code-art is a program written by Alex McLean called "forkbomb.pl", a winning piece of code-art in the 2002 Transmediale software art exhibition. This program is an example of a fork-bomb, a species of suicidal program which endlessly spawns new copies of itself which run in parallel, representing a kind of aggressive colonization which eventually crashes the system on which they're run. McLean's program is interesting specifically because it consists of two nested infinite loops to spawn copies of itself, one which prints a '0' to the terminal and the other which prints a '1'. Like most other fork-bombs this program puts heavy stress on the computer, and

consequently the system becomes highly sensitive to subtle scheduling behaviors of the underlying operating system. These minute variations in timing and state cause the program to run indeterminately, meaning the program displays a unique portrait of the user's computer under stress each time it is run (Cox & McLean 2012, p44). The output of this program can therefore be understood as a fingerprint expressive of the precise computational context in which it is run.

```
#!/usr/bin/perl

no warnings;

my $strength = $ARGV[0] + 1;
while (not fork) {
  exit unless --$strength;
  print 0;
  twist: while (fork) {
    exit unless --$strength;
    print 1;
  }
}
goto 'twist' if --$strength;
```

Fig 5: *"forkbomb.pl"*

Esoteric Programming Languages

One of the most interesting kinds of semantic code-art are *esoteric programming languages*. These works are interpreters, compilers, or simply theoretical specifications of programming languages specifically designed to be bizarre in their syntax or execution. Esoteric programming languages provide the unique capability to provide meta-commentary towards the very act of programming. Michael Mateas and Nick Monfort elegantly summarized the value of esoteric programming languages, saying "In the field of weird languages, also known as esoteric languages, the programmer moves up a level to exploit not just the play of a particular language, but the play that is possible in programming language design itself." (Mateas & Montfort 2005, p147-148). In this regard they can be compared with or conceptualize as anti-art, in that they transgress commonly held conceptions towards how programming can be conducted in order to explore the essential nature of code.

One example of an esoteric programming language which provides interesting meta-commentary on the usual syntax of languages is *Whitespace*, a programming language whose syntactic constructs consist entirely of spaces, tabs, and line-ending characters. Because of this, programs written in Whitespace appear essentially invisible when opened in a text editor. Whitespace subverts and

reverses the typical distinction between syntactic constructs and secondary-notation in typical programming languages. It can be understood mainly as tongue-in-cheek commentary on the often ignored importance of spacing within code, implementing this by hyperbolically reducing the act of programming to hitting the space, tab, and enter keys.

The oldest esoteric programming language *INTERCAL* was created by Don Woods and James M. Lyon in 1972. Programming in INTERCAL was designed to be a difficult and perverse undertaking, which was achieved by integrating many of the more difficult features of other languages with an obtuse syntax and set of operations (Mateas & Montfort 2005, p149). Many of these design decisions are quite satirical, for example statements in the language may be prefixed with a 'PLEASE' keyword; programs containing too few 'PLEASE's were rejected by the compiler for being too rude, and likewise programs with too many were also rejected for being too simpering (Temkin 2017, p86). INTERCAL transforms the usually straightforward process of transcribing an algorithm into a puzzling battle to satisfy the arcane demands of an incomprehensible compiler.

A final example of the value within esoteric programming languages and their interpretation could be *Compute*, a language designed by a programmer know only by the pseudonym 'Orange'. Compute is an instance of what is called a *conceptual language*, that is, a programming language which is specified purely hypothetically, lacking a concrete implementation either because one is unnecessary or impossible to write. Compute accepts as its syntax any and all natural human languages (as such this document actually represents a valid Compute program). For example, the language allows for a program which calculates all prime numbers to be encoded as "Hey program, find all the prime numbers". The only catch is that Compute lacks any semantics for encoding input and output, meaning that such a program could never produce output which proved it had completed the computation. Compute cheekily emphasizes the importance of the relationship between computations and the humans who initiate them by demonstrating the uselessness of programs which cannot interact with their creators. The semantics of code are ultimately irrelevant without people to give them meaning.

Closing Statements

Just as visual art and music provides us with novel perspectives into the nature of our senses of sight and hearing respectively, code-art can provide us with new understandings towards the way we think. Code is fundamentally a means of encoding the instructions to perform an operation in a textual format. Moreover, as discussed throughout this text, programming languages are constructs designed by humans for humans. As such code's linguistic and semantic components are distinctly relevant to understanding the ways in which we conceptualize the processes essential to our existence. Independent from its ability to be executed by a machine, code can and must be interpreted into something meaningful by humans (Chun 2008, p313). The significance of Code goes beyond its function to instruct a computer to perform a set of operations, it is also a personal and cultural product which allows for the expression of values, criticisms, or paradigms just as all other artforms do.

The artistic community would do itself a great injustice in ignoring the value of code-art. This medium represents a new alternative to artforms of the past, with a special relevance to the nature of computation and thought. We must equip ourselves with new techniques for finding meaning in the placement of a semicolon or the cleverness of an algorithm if art is to move forward. Not only can this medium serve to enrich and innovate the field of fine art, but the integration of aesthetics and code will allow for better understandings of purely practical programming too. Software development as a discipline must follow in the footsteps of architecture by acknowledging the importance of aesthetics to the construction of human relevant utilities. As this text has demonstrated, Code can be beautiful in many ways, and thus it is not only possible, but imperative, that we must develop ways of articulating and interpreting these aesthetic properties for the sake of both art and engineering.

```c
#include <stdio.h>

void main(int argc, string[] argv) {
    printf("I am more than a text file.\n");
    printf("I am more than just instructions.\n");
    printf("I am culture.\n");
    printf("Hello, Artworld!\n");
    return 0;
}
```

Sources

Figures

Fig 1 - "Black Perl" taken from: *https://boingboing.net/2013/09/24/black-perl-a-poem-in-perl-3.html*

Fig 2 - Winning IOCCC entry by Volker Diels-Grabsch taken from: *http://www.ioccc.org/2019/diels-grabsch1/prog.c*

Fig 3 - Excerpt from "Much TODO" taken from: *http://stevenread.com/project/much-todo*

Fig 4 - Language diagram for the "128-Language Ouroboros Quine" taken from: https://github.com/mame/quine-relay

Fig 5 - "forkbomb.pl" taken from: https://slab.org/forkbomb-pl/

References

Cox, G. and McLean, A. (2013). *Speaking code*. Cambridge, Mass.: The MIT Press.

Chun, W. (2008). On "Sourcery," or Code as Fetish. *Configurations*, 16(3), pp.299-324.

Fishwick, P. (2002). Aesthetic Programming: Crafting Personalized Software. *Leonardo*, 35(4), pp.383-390.

Fuller, M. (2008). *Software Studies: A Lexicon*. Cambridge, Mass.: MIT Press.

Mateas, M. and Montfort, N. (2005). A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics. *Proceedings of the 6th Digital Arts and Culture Conference*, pp.144-153.

McLean, A. (2011). *Artist-Programmers and Programming Languages for the Arts*. Ph.D. University of London.

Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglas, J., Marino, M., Mateas, M., Reas, C., Sample, M. and Vawter, N. (2013). *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. Cambridge, Massachusetts: The MIT Press.

Temkin, D. (2017). Language Without Code: Intentionally Unusable, Uncomputable, or Conceptual Programming Languages. *Journal of Science and Technology of the Arts*, 9(3), p.83.

Bibliography

Bertran, I. (2012). *code {poems}*. [online] Continent. Available at: http://continentcontinent.cc/index.php/continent/article/viewArticle/97 [Accessed 13 Jan. 2020].

Broeckmann, A. (2006). *Software Art*. [online] Mikro.in-berlin.de. Available at: http://www.mikro.in-berlin.de/wiki/tiki-index.php?page=Software+Art [Accessed 13 Jan. 2020].

Chun, W. (2008). On "Sourcery," or Code as Fetish. *Configurations*, 16(3), pp.299-324.

Cox, G. and McLean, A. (2013). *Speaking code*. Cambridge, Mass.: The MIT Press.

Fishwick, P. (2002). Aesthetic Programming: Crafting Personalized Software. *Leonardo*, 35(4), pp.383-390.

Fuller, M. (2008). *Software Studies: A Lexicon*. Cambridge, Mass.: MIT Press.

Mackenzie, A. (2006). *Cutting Code: Software and Sociality*. New York: Peter Lang Publishing.

Mateas, M. and Montfort, N. (2005). A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics. *Proceedings of the 6th Digital Arts and Culture Conference*, pp.144-153.

McLean, A. (2011). *Artist-Programmers and Programming Languages for the Arts*. Ph.D. University of London.

Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglas, J., Marino, M., Mateas, M., Reas, C., Sample, M. and Vawter, N. (2013). *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. Cambridge, Massachusetts: The MIT Press.

Orwant, J. (2001). *The Perl Poetry Contest*. [online] Dr. Dobb's. Available at: https://www.drdobbs.com/the-perl-poetry-contest/199101825 [Accessed 13 Jan. 2020].

Temkin, D. (2015). *Compute*. [online] esoteric.codes. Available at: https://esoteric.codes/blog/does-compute [Accessed 13 Jan. 2020].

Temkin, D. (2017). Language Without Code: Intentionally Unusable, Uncomputable, or Conceptual Programming Languages. *Journal of Science and Technology of the Arts*, 9(3), p.83.

Wall, L. (1999). *Perl, the first postmodern computer language*. [online] Wall.org. Available at: http://www.wall.org/~larry/pm.html [Accessed 13 Jan. 2020].