Software and Anarchy

Gnuxie Lulamoon and Hayley Patton

Draft of January 11, 2021

This book is dedicated, in respect and admiration,
to the spirit that lives on the computer.

Contents

Introduction

Preface

This book covers our interpretations of programming as it relates to *social ecology*, a philosophical theory that suggests many forms of dominion have the same causes and methods, and *peer production*, a self-organising mode of production, and how their application can support a radical programmer exceptionally well.

A *liberatory technology* provides an improvement in the work which can be done with some amount of effort and time. A community which utilises such a technology well can perform much more work independently. We believe a liberatory technology for programmers takes the form of a *dynamic environment*, which is a property of some programming languages and implementations that allows for many forms of modification and inspection at any time while a program is running.

We will also investigate the supposed disadvantages of a decentralised development strategy, and how they may instead be used to the advantage of a community. We also will present why it is desirable to distribute source materials, especially in software development, and how shunning distributing source materials as anti-capitalist "praxis" is nonsensical. Finally, we will discuss what an anti-hierarchical view of some communication networks can bring to light; especially with community moderation systems, the property relationships a user conjures with their environment, and how to remove as many constraints on the means of communication and presentation as possible.

We hold the belief that software peer production while rejecting the potential of dynamic environments is, at best, incoherent, and that a user of dynamic environments outside peer production may well have problems applying the environment as much as they could have. We try to achieve a few aims with this book: to introduce the reader to an idea of *software anarchy*, to investigate some common opposition to it, and to provide a wide selection of resources that a reader (whom can produce enough free time) can read to help develop their own approaches.

Beginning a constant analysis

Every piece of code you've ever written is a lie.

Every day we encounter issues with software, from flaws to bugs to minute nitpicks. Every day, we imagine how software could change, so that using it could be easier for everyone. Every day we experience more and more of the true nature of the software world. Every day we see that, in this world, things cannot be changed, and so eventually we learn to no longer imagine. Every day we contribute a million more lies, a million more arrows to fall back from the sky. Every day we bring ourselves to accept that this is how it is, how things are. Every day We say this is how it has to be.

Every piece of code you've ever written is a lie. Why? Why do we say this is how it has to be? Why do we bring ourselves to accept that this is how it is? Why can't we just change...

This despair in software production parallels the despair one might have with the political climate, the world at large. Anarchism is the process of constantly analysing all hierarchy and actively moving to dismantle them, and an anarchist framework should be applied to software. We should constantly analyse software: "Why is this so complicated?" "Why can't I just express what's in my head?" We should actively move to dismantle complicated software and develop new ways to express problems and ideas.

One might wonder about the practicality of undertaking such an analysis, as there are surely are some problems that simply cannot be solved; some things need to be completely rewritten, and some things just won't be extended in some ways. We have sunken so many resources into non-general solutions and made massive mountains of code, but it doesn't have to be this way. Another world is possible, even if it starts out a small one. We will describe a strategy for establishing *software anarchy*, our application of anarchism to software development.

While it would be inappropriate to claim any ownership or other significance in presenting these ideas, many of the other radical software development projects fall short in their analyses of hierarchies, relations between programmers, users and other users, and so on, and are certainly far from "what [we] had in mind". In contrast, we believe software anarchy provides a cohesive critique of the current state of software development, "radical" or otherwise, and would be integral to a praxis that actually liberates its practitioners from hierarchy established by software and its applications.

The last constant analysis

It appears radical programmers frequently lose their grounding on what forms of organisation are right or wrong while on the Internet. We hear one group establish and espouse the direct democracy they formed, and then go on to write about "the tyranny of the majority" elsewhere. We read someone cussing out intellectual property, and then see them express support for an ethical business model that promotes more information hiding. It is now common to suggest some ways of improving online discussion that push for more asymmetry and hierarchy which would be unthinkable in physical spaces. With the rapidly increasing overlap between cyberspace and the physical space, it should appear that a view on how to maintain an online society and a view on how to maintain an offline society should converge; but the opposite may well be occuring.

When an egalitarian community has continued to exist, it is always suggested that squashing the dynamics and independent experiments in the community, in favour of coercing the participants to use one set of techniques and produce one set of solutions. Depending on the social standing and marketing skills of the person who suggests the solution, it is sometimes almost taken seriously.

It is also in vogue to lose the control of, and lose abstract reasoning about a program. One programmer once wrote the words "The era of dynamic languages is over. There is currently a race to the bottom…" There is not a bit of resistance or disgust at that observation. Our plane is losing altitude quickly, and we simply couldn't give a shit. "Brace for impact. I'm not going to pull on the control wheel…" Such apathy is better than the average response, sadly. We are pushing ourselves to imagine how to design programs for smaller and less featureful machines than most of us will ever be tasked with programming, and basing our decisions for how to program larger machines on that. The target for what constitutes a *high level language* has moved so far that it is acceptable to have "abstractions" which will somehow get us out of performing partly-automated static analysis on our programs, because such itty bitty machines cannot support the runtime features that would otherwise be used. However, those abstractions cannot meaningfully be formed, and if they could be formed, they still would put their user in a worse position.

This form of meaningless and purposeless minimalism allows no useful work to be performed, and echoes the plan-everything-ahead design techniques we learnt and scoffed at in school, which we never saw or used again, because of how useless they were. We accept having to approximate clairvoyance, in order to micro-optimise a large system, and to validate the many interactions in them. With these costs, experimentation cannot possibly be done after the initial design is made, and the capability to make any progress beyond what has already been done is a joke. Bob Barton's infamous observation applies here: although there will be no priests, will radical programming become a low cult?

Acknowledgements

We would like to thank Robert Strandh and Selwyn Simsek, among many #sicl and ##symbolics2 participants, for reviewing earlier drafts of this book. We are currently sending out some drafts for proofreading; you can have your name here if you are reading this draft, and you like having your name in acknowledgement sections.

Chapter 1

Liberatory technology

Before we discuss how to support ourselves developing liberatory software, we will provide an intuition for what we could produce, which would be more likely to be liberatory than not. To do this, we will analyse the model poised by *social ecology* of a *liberatory technology*. It may appear counter-intuitive for technology to have a central part in an ecological process, but there are many purposes an appropriate technology would have; such as studying the local environment to decide how it should be complemented and best utilised by its residents, automating the recovery resources from scraps with techniques that few would do themselves, and generally relieving its users of toil, so that that they can do more interesting things. These uses coincide with our goal of relieving ourselves of toil, and automating processes we would rather not do manually.

The reader may or may not have an intuition for what a liberatory technology could be, so for the avoidance of doubt we will provide a definition. A liberatory technology is one which is thoroughly decentralised, can be adjusted and modified to fit its users' applications and environments, and is integrated in a way that it does not impede on how its users process things, instead complementing their processes. It is also designed to be understandable, usable and modifiable by a small community, allowing the community to support its use autonomously.

> We saw alternative technology as having great potential for decentralized, humanly scaled applications in the urban setting, and as lending itself to community control and directly democratic forms of decision making, thus providing a material base for the development of a decentralized, directly democratic society.
>
> Chodorkoff, 2010

A liberatory technology would also facilitate its use for many purposes, and can self-regulate to reduce the duties of its operator.

> The importance of machines with this kind of operational range can hardly be overestimated. They make it possible to produce a large variety of products in a single plant. A small or moderate-sized community using multi-purpose machines could satisfy many of its limited industrial needs without being burdened with underused industrial facilities. There would be less loss in scrapping tools and less need for single-purpose plants. The community's economy would be more compact and versatile, more rounded and self-contained, than anything we find in the communities of industrially advanced countries.
>
> Bookchin, 1971

This is fundamentally a book about programming, so the reader should know a computer is very much a multi-purpose machine, and a computer can self-regulate; one can compute almost anything that can be computed with a computer, and can do so faster than a human most of the time. Software on a computer can also protect itself if something goes wrong, and there are many techniques allowing software to continue after an exceptional situation. It would be pretty clear that computers are already integrated into the idea, as they are an important component of automated systems; but studying how to liberate the programmer and their programming *themselves* may need to be elaborated upon.

There are many ways to convince a computer to compute what you want it to, with varying qualities and peculiarities that convince people that some ways of conviction are usually more appropriate than others. The typical social ecologist and radical programmer have similar requirements for their technologies: it should be efficient, it should be understandable by another reader, the system should be modifiable by the user, and so on. Programmers often have problems fulfilling multiple of these requirements, by creating a slow program, or creating a fast but unreadable program, or producing an easily understood and performant program, which does not have any facilities for extension and modification, and so on. Fortunately, this situation is not innate to programming, and the qualities of a given programming environment can greatly affect the qualities of one's program. We believe that a *dynamic environment* frequently provides the requisite facilities to pursue the best approach, and the situations and criteria in which one does not are gradually decreasing with time.

Dynamic environments

A *dynamic environment* is roughly characterised by providing the programmer a way to have a conversation with the computer.[1] While the conversation is not held in English or any other natural language to humans, it is more like a conversation than the interactions made with a *static environment*, in which a programmer provides a *compiler* program with source code, which then produces a program that the programmer can hardly modify after compilation. A programmer can ask many things, such as what the result of evaluating some code is, and what the properties of some object are. The computer may reply with questions, such as how to proceed when an error occurs. This conversational approach has several advantages: there is much less latency than when compiling, running and debugging entire programs, a programmer can see their code running and inspect the data it uses instead of having to imagine both, and a dynamic system can be triv-

[1]An analogy that was once punned on by Alan Kay: "I had mentioned to someone that the *prose* of then current programming languages was lower than a cocktail party conversation, and that great progress would have been made if we could even get to the level of making *smalltalk*."

ially updated while it is still running,[2] which is crucial for programs that must run for a long time, such as servers and machine controllers.

> Traditionally, a program is thought of as describing objects to be created at run-time. [...] But, being a description, the program cannot be directly used to visualize the objects in a running program; the programmer must make a visual leap.
>
> Ungar, 1995

> When I looked at Java, I thought "my goodness, how could they possibly [...] survive all the changes, modifications, adaptations and interoperability requirements without a meta-system?" Without even, for instance, being able to load new things in while you're running? So, the fact that people adopted this as some great hope, is probably the most distressing to me personally, as I said, since MS-DOS.
>
> Kay, 1997

Note that designing and implementing a dynamic system may be more difficult than with a static system; for example, the representation of a global environment, which functions, variables, classes and so on reside in, may have to be made explicit, and data structures have to be made adaptable when class and type definitions change,[3] but those problems only have to be dealt with once by the implementation. The additional effort required to implement these mechanisms is quickly restored by the leverage a developer now has over the system, greatly broadening the range of programs an individual or a small community can maintain. This form of leverage provided by a liberatory technology pushes forward the "limited horizons of achievement" (Mumford, 1964) which would otherwise bind a community to more mundane techniques; a user is capable of performing the same task with little

[2]Well, not exactly trivial: in some cases, the programmer has to be careful to not provide a running program functions that do not immediately work with types already in the system and vice versa. But this is far better than nothing; *The long run* compares the qualities of this mutability with the immutable prior reasoning that is more common today.

[3]see `update-instance-for-redefined-class` in *Common Lisp*

effort, and with the same effort and time, the user can perform much larger tasks.

This ability may also serve to adapt software to conditions and aims the original producers did not anticipate; avoiding the assumed planning that authoritarian technics[4] require, where "one must not merely ask for nothing that the system does not provide, but likewise agree to take everything offered, [...] in the precise [forms] that the system, rather than the person, requires." With the larger, but more specialised and categorised software environments we see today, adaptability has become an aim of some developers. It is, however, not feasible to implement or take advantage of, without a dynamic system to empower the users and developers to perform anything more than superficial or cosmetic changes to software.

Always has been *malleable*

A concept of a *malleable system* has surfaced recently, which like most things we find ourselves writing about, was quite promising at first. A malleable system appears oddly similar to a dynamic system, allowing for arbitrary composition of its components, but is more focused on having users perform their own composition.

We found that a malleable system can be made of static components, and that some proponents of malleable systems assume static components in a malleable system. This assumption is a form of *futurism* which "[extends] the present into the future" (Bookchin, 1978), by establishing a hypothetical future where a malleable system can be produced, but somehow out of static components. It is scaling a structure which cannot be reasonably scaled, by vaguely suggesting how to adjust the relationship of its components; a kind of "technique" rightfully feared by programmers and social ecologists alike:

> Now, somebody could come along and look at a dog house and say, "Wow! If we could just expand that by a factor of a hundred we could make ourselves a cathedral."

[4]Note that Mumford uses the term *democratic technics*, which is a precursor of sorts to the *liberatory technology* of social ecology; and places an *authoritarian technics* in contrast to it.

> It's about three feet high. That would give us something thirty stories high, and that would be really impressive. We could get a lot of people in there. [...] [However,] when you blow something up [by] a factor of a hundred, it gets a factor of hundred weaker in its ability, and in fact, [...] it would just collapse into a pile of rubble.
>
> Kay, 1997

> Futurism is the present as it exists today, projected, one hundred years from now. That's what futurism is. If you have a population of X billions of people, how are you going to have food, how are you going to do this... nothing has changed. All they do is they make everything either bigger, or they change the size – you'll live in thirty story buildings, you'll live in sixty-story buildings. Frank Lloyd Wright was going to build an office building that was one mile high. That was futurism.
>
> Bookchin, 1978

A static malleable system attempts to scale a system that cannot be scaled; programs in static languages and static environments would be hopeless[5] at using objects and types they have not been programmed to use (without mangling them, perhaps by serializing objects, which would violate the notion of both an object and a type). We have already argued that a static system is harder to develop with, but such a malleable system made of static components would have significant drawbacks that would limit its malleability.

Some forms of *adaptors* would be used in a malleable system with static components; including a *dynamic linker*, used to swap the static components that are in use within the system, and a *embedded interpreter* for a dynamic language can be bundled with a static system to

[5]There was an argument about if dynamic *type systems* were any better at handling objects we don't know how to handle. At the implementation level, most optimisations that can only be done in static systems are some forms of inlining, possibly *monomorphizing* the generated code, so that it will be much more efficient with memory layouts it knows, at the cost of not being able to manipulate new types easily. At the language level, we would like for different protocols which "do the same thing" to cooperate, but this is not even possible in the *structural type systems* that were advocated for by static type proponents, where types are sets of applicable functions. So, yes, static languages and their implementations are hopeless.

provide glue code between static components. The language is also typically used to provide the extension mechanism intended to be used by a user. One example of a successful malleable system, as mentioned in the Malleable System catalogue,[6] is ole *GNU Emacs*, which we have happily used to type up this book. But working with its internals is not an enjoyable task; the *XEmacs* developers knew this well, and attributed it to the non-abstract representation of many Emacs objects:

> We want our implementation of keymaps to be used: we want them to be an abstract data type, not something like "if the third element of the alist is a cons whose car is a vector of length 7, then it represents an aliased indirection into the sixth element of the alist..."
>
> Zawinski, 2000

How could this be a problem with a static malleable system? Emacs isn't entirely written in Emacs Lisp, and there is a C core that does a lot of important stuff for it. Interfaces between the two languages, one bytecode-interpreted dynamic language, and one compiled static language, are likely to be difficult to work with, and effectively lead a programmer to use strange representations that only make the interface easier to implement.

The rest of implementing such an interface is also not easy:

> But basically, either you break the modularity so that you know what the module does with your objects and you have a maintenance nightmare, or you *copy objects*, use *smart pointers*, or use *reference counting* and you have a slow application.
>
> Robert Strandh on `#lisp`

Implementing an interface between static and dynamic components doesn't appear like it gets easier – never mind interfacing static components with other static components.

Another extension of this issue is that developers cannot always foresee the ways in which their programs could be extended; the GNU

[6] `https://malleable.systems/catalog/`

Emacs developers did not expect a client to introduce new "primitive"[7] types, or they did not immediately see why that would be useful.

As such, static systems cannot foster a complete "read-write culture" (as introduced in Chapter 2) which would be fundamental to malleable systems. In an attempt to facilitate this culture, the developers of a static system use adaptors, which are constrained to the capabilities of the components that provide their implementation. If a user needs to make changes to a static component that is used by the implementation of fundamental components of the system, they cannot use an adaptor to do so – they must have the knowledge that was used to create the system in order to change it, and so a user could not use the embedded programming environment bundled in a static system to modify these components. They would effectively have to become a developer of the system, by retrieving the source code of system, modifying it, and then rebuilding. This would violate read-write culture, as this is not something that can be expected of a user. We can also argue that this alone makes modification of static systems inherently inaccessible to those who need to program from another environment.

Proponents of malleable suggest that relinking and subclassing can be used to provide extensions to a static system, however there are many extensions that would in fact require direct patching in order to work. An example of modification that the programmer did not imagine before appears around 8:25 into Dan Ingalls's demonstration of an old Smalltalk system (Ingalls, 2017), in which Ingalls modifies the way text selection is visualised, by replacing a method in the class that handles drawing the selection. If Ingalls was using a static system, subclassing this class and re-implementing the method there would not update programs itself; programs would also have to be updated to use this new class. While it is entirely possible to provide a better styling mechanism, allowing the user to describe how graphical objects should be rendered (as many desktop environments do today), the general solution appears long after the problem and a specific solution do, and the user can only wait for the developer to patch their components

[7]In many Lisp systems, it is possible to define composite data types that are distinct from any in-built data types; *Common Lisp* provides `defstruct` and `defclass` to do so. Emacs Lisp is very unique compared to other Lisp systems, and not just with data representation.

should they not be able to modify the components; Ingalls was able to replace the method almost immediately.

A dynamic system makes it very possible for the user to inspect and modify the underlying system from any accessible environment. Additional problems that arise in making malleable systems simply do not exist in truly dynamic systems, as much of the host system can be reused in implementing the extension system. In many dynamic languages, it is possible to generate code at runtime, and introduce new bindings for functions, values, classes and so on, allowing a user to modify the program without the programmer having to specially allow them to. A domain specific language, should one be used,[8] can be compiled to the dynamic language, used interchangeably with components written directly in it, and use the provided, fast implementation of the language, as opposed to rolling an interpreter that is strictly slower than the host, and creating half a dynamic system to host the malleable system.

An entirely dynamic system such as a Smalltalk or Lisp machine obviates even more problems, including that of communicating (with serialisation and deserialisation of) complex objects between processes. Future systems such as CLOSOS (Strandh, 2013) will even eliminate sterilising objects to disk, as object storage will be persistent. As many issues with malleable systems are reduced with techniques performed by dynamic systems, we are well convinced a successful malleable system must be dynamic.

In case the utility of a dynamic system is still not apparent, we will try to contextualise the features of a dynamic system in a short “story”, in which their utility should be evident.

Techniques for building farming devices

Fred Armstrong is bored of farming, and wants to work on the telephone system of his town. However, he is somewhat too protective of his carrot farm and won’t give control of it to someone else to maintain instead.

[8]Though we would be surprised if said language wasn’t used already to write parts of the program!

Earlier, he had heard with a program a colleague told him about, which uses an *expert system* and readings from digital sensors and meters to create a to-do list of what should be done before he leaves to work with his new hobby. This would relieve him of, say, checking with the meteorologist if he did in fact hear rain while sleeping, as it would check the soil moisture (which is probably more accurate, anyway), or pull the weather records by itself. However, he had expressed concerns that he wouldn't be able to tell how the program had made its decisions, but his colleague said it would also explain how it came to a conclusion with a clear and logical description of its "thought process" in natural language.

Convinced that the program could help reduce the time he has to work on his farm, Fred downloads it on his laptop, and plays around with its simulation mode, allowing him to model various situations and observe how the system reacts, and inspect the rules it uses to decide what to do. He tries modelling the flood of the April of 2060, which he still recalls, having prompted him to work on a way of elevating the crops above water. Well, it wouldn't be too hard to spot a flood, but if it breaks somehow, it would probably break then. And it did not, so Fred is satisfied and calls up the local engineer for help integrating some sensors he found in a shed with the system.

Henry Babin, the local engineer, arrives at Fred's house the next day, and sits with him on the verandah as they discuss the system Fred wants to set up. Fred passes him one of the sensors, and Henry is visibly shocked by how old it looks. It sitting in a dusty shed did not help make it look new, but the technique it used to detect moisture and the concentration of some chemicals which are important for plant growth dated it to the mid 2020s at least. Nonetheless, he believed he could use it, as it would still function as well as newer sensors – it was just made with questionably sourced materials by a questionably compensated assembler.

Henry connects the sensor to his computer, and retrieves the *data sheet* for the sensor, and the manual for the farming program. As well as a comprehensive description of the user-facing portion of the program, the manual contains instructions for how to interface new hardware with the expert system, cross-referencing the *protocol classes* and *methods* that the system requires, in order to read measurements off the sensors.

After the manual agreed with him that he had correctly implemented as much as could be tested without performing a test with the sensor, Henry connects a sensor that he knew to be accurate and already supported by the system, and plants them both in his garden. He then starts a program that the manual provided in order to check if the values read were correct, but the concentrations plotted appear to be about a magnitude too low.

Henry grumbles at the computer, and remembers, much like Fred's program, his programming environment has a mode where it tracks how values came to be in his program.[9] He uses the inspector of his programming environment to find that his interface produced the values by scaling the raw values read to the limits of the sensor... of which he had entered the maximum concentration with one too many zeroes. As soon as Henry replaces the maximum concentration he had provided, the values of the old sensor shoot back up to those of the new sensor, and Henry is satisfied with his efforts for the afternoon, and goes to take a nap before returning to Fred's house with the sensor and working program.

Henry visits Fred again, returns his sensor and installs the updated program on Fred's laptop. Fred installs the sensors, and the pair proceed to drink tea, as the program has to take a few samples before producing its first report. The program states that the soil is too dry, so it must be watered, but looking up at the dark clouds suggests very clearly that it is going to rain soon. Henry offers to modify the program to consider the weather forecast for the day, but Fred, eager to show off what he learned by messing with the simulation, connects a forecast source from the server of the meteorologist, and enters a rule to put off watering if it is early into the day, and it is likely to rain; partly imitating the rule that estimates soil moisture from rainfall records which he observed in the simulator. Henry barely holds in a giggle, thinking that Fred is fumbling around with the program, but is pleased to see him pick up rule programming so quickly.

[9]This is something like a *time travelling debugger*, which records how a program is executed, so that the user can step through it at their own pace after execution. We suppose, for simple functional expressions, we could just present the function calls made.

Who's doing the computation now?

Unfortunately, static systems still exist, and even more unfortunately, they may well be on the rise again. Dynamic optimisations and compilers have been applied to languages such as Lisp, APL, Smalltalk and Self, Java, Julia and so on for almost fifty years, and high level languages can frequently perform faster than static languages in symbolic processing.[10] But that is not a factor that could fit into the world view of a programmer fooled by *Ousterhout's fallacy*, the false dichotomy between "scripting" languages (which are interpreted and have sub-optimal data representations) and "system" languages (which are compiled and have optimal data representations), who could not believe that a dynamic environment can be just as fast and provide guarantees like a static environment. Thus the solution must be to produce static systems with better guarantees, and work around problems (such as *memory safety*) that simply do not exist in dynamic languages, and not having the benefits of late binding, introspection and so on.

We had previously written an essay about static environments, entitled *Masochist programming* (Patton, 2019). This odd description of a programmer that advocates for static environments relates to how they appear to want more work to do, and to have a much more "painful" experience programming. We wanted to know **what excuses does one have for bonding themselves to a static environment?** So you can write code for microcontrollers? Like that web server we heard of that falls over with 2,000 connections (which was called a denial of service to save face), which is clearly going to run on a microcontroller soon? To make micro-optimisations to drop levels of indirection that access to a compiler could do for you, and then optimise out some more? Is it more "real time" (whatever that means to you)? Are you now free from watching your consing and control flow to produce consistent execution times? And you're designing for sub-millisecond latency? All of the above? Most people we ask are really doing none.

While we're here: expressing concerns about the safety of C and C++ programs has not and is not limited to proponents of other "sys-

[10]One example is solving the n-queens problem in OCaml, a functional high level language, and in C++, an imperative low level language. The C++ program ran up to about 10× slower than the OCaml program: `http://flyingfrogblog.blogspot.co.uk/2011/01/boosts-sharedptr-up-to-10-slower-than.html`

tems" languages,[11] contrary to popular belief. For example, the iconic Scheme textbook *Structure and Interpretation of Computer Programs* makes note of the unsafe-by-default code C and C++ compilers generate:

> Compilers for popular languages, such as C and C++, put hardly any error-checking operations into running code, so as to make things run as fast as possible. As a result, it falls to programmers to explicitly provide error checking. Unfortunately, people often neglect to do this, even in critical applications where speed is not a constraint. Their programs lead fast and dangerous lives. For example, the notorious "Worm" that paralyzed the Internet in 1988 exploited the Unix operating system's failure to check whether the input buffer has overflowed in the `finger` daemon.
>
> Abelson and Sussman, 1996

This "radical" desire of more toil reminds us of old Soviet propaganda posters; the kind where some proles are farming, metalworking, loading guns, that kind of thing, with some motivational caption that we can't read. In that time in Russia, industrialisation was highly sought out, in an "age burdened by scarcity, when the achievement of socialism entailed *sacrifices* and a *transition period* to an economy of material abundance" (Bookchin, 1971) and working to advance the industry as such was a noble thing to do with one's time. That was, of course, ninety or so years ago, and now with the possibility to greatly reduce toil, and capitalists already ordering production greater than what can be consumed (and then to burn off the surplus), it would be absurd to tell people to work more. Apparently not so for static programming environments!

[11] We don't really think systems languages exist; as previously mentioned, machines that had operating systems written in Lisp and Smalltalk had some success, and were more advanced than Unix at the time; the Unix-Haters Handbook (Garfinkel et al., 1994) provides some commentary from users of those machines who were forced to migrate to Unix systems.

The long run

But efficiency isn't the only desire of a static programmer; many proponents suggest that they are able to verify their behaviour at compile-time (summarised in the mantra "if it compiles, it works"). Dynamic systems are usually designed to facilitate working with moving targets, so verification may not be desirable for many users, who could not produce a specification that could be verified before it is changed.

> Market pressure encourages the development and deployment of systems with large numbers of complex features. Often systems are deployed before the interaction between such features is well understood. During the lifetime of a system, the feature set will probably be changed and extended in many ways.
>
> Armstrong, 2003

Some systems even assume the opposite is inevitable, that there are errors in user programs, and attempt to behave correctly in the presence of software errors. Such systems may also have advantages for handling hardware (and other external) errors, as those are just another type of failure that the system can accommodate for. A dynamic approach is better suited for maintaining long-running processes that undergo changes, yet by the superficial appeal of being able to "prove" behaviour, and delivering the possibly false belief that one is safe from programming errors, it is mostly non-present in mainstream programming.

(There are some cases where a dynamic environment simply couldn't work, such as deploying programs for constrained machines like microcontrollers and unmodifiable programs such as *smart contracts*, but again the times an average programmer works with those are probably very overstated. However, many theorem prover programs today, such as Coq and ACL2, are very interactive, yet produce arguably the most static things programmers make. It would not be hard to assume that making testing environments more interactive would allow for greater confidence in the correctness of a project in less time.)

> But with C++ and Java, the dynamic thinking fostered by object-oriented languages was nearly fatally assaulted

> by the theology of static thinking inherited from our mathematical heritage and the assumptions built into our views of computing by Charles Babbage whose factory-building worldview was dominated by omniscience and omnipotence.
>
> And as a result we find that object-oriented languages have succumbed to static thinkers who worship perfect planning over runtime adaptability, early decisions over late ones, and the wisdom of compilers over the cleverness of failure detection and repair.

Gabriel, 2002

The languages which may be considered dynamic or object-oriented today, like Python and Java, are in effect the worst of both worlds: unable to be adapted and migrated to new specifications like classic dynamic systems, and unable to be analysed and statically proven like static systems.

The trend towards static monoliths with dubious claims to "efficiency" and "correctness" should be terrifying; prototyping and experimentation, which may become the means by which people adapt and integrate technics and our software into their contexts and use-cases, are going to become much harder by producing increasingly more static structures. The conversational and malleable aspects of dynamic systems make them almost always ideal in reducing toil for programmers, and a free society that requires programming certainly should educate and support its programmers in using such systems. Overstating the cases in which this is not the case is comparable to overstating the minimum toil a free society would need to support itself.

Chapter 2

Peer production

Peer production could be described as "coincidental" organisation, where individuals organise to achieve a common goal, with no formalities and usually no other reasons to associate. This mode of production has appeared with the proliferation of the Internet, as it makes collaboration between strangers with common interests quite possible. It can be said that peer production maximises the agency of an individual producer, while promoting a free flow of information. Such a flow of information is arguably harder to prevent than allow in modern society, especially with programs and their source code; there are many websites that a developer can upload source code to, and be provided with issue and patch submission tracking at the very least. A hobbyist is very likely to release their source code by default when presented with these facilities.

Infrastructure for hosting source materials is crucial to facilitating peer production, but it is not the only crucial component. (Kleiner, 2010) goes very far into how to provide financial and material support for producers, proposing a *venture commune* to decentrally manage materials. We would only reiterate the points and strategies presented in the manifesto, so we will not discuss it further in this book. Instead, we will discuss how to organise communities to engage in peer production, while preserving their decentral nature, without extinguishing but rather *utilising* the plurality and dissensus of a decentral community.

Non-hierarchical people organisation

Static forms of organisation may hinder the creative and innovative processes of their participants. Binding ourselves to projects and individual manifestations of our line of thought is not sustainable, as we find ourselves frustrated with what we cannot change, and what we should have done, as we progress.

> If we use our creative power for ourselves, it destroys us. If we sacrifice ourselves in order to serve the creative power, then it creates us.
>
> Bennett, 2009

It is also difficult to find resources and funding when one is investigating very new ideas, without a benefactor (such as a multi-billion dollar printer company, or a government or military contract) that trusts one to find something interesting, or has enough resources to throw at anyone.

> True innovation also involves questioning the assumptions that almost everyone agrees with. This can sometimes make those of us engaged in research feel a bit like thought criminals. [...] Of course, no one is going to send inquisitors to our homes to persecute us for disagreeing with the mainstream. However, you can run out of funding very fast.
>
> Bracha, 2013

By organising a group around the development of a project, the participants may deny themselves the potential of producing a new project that better supports their ideas. Peer production may allow its participants to diverge and follow their lines of thought more gracefully than a rigid collective or cooperative form; it is likely, should one find themselves traversing a new idea, that the people that one wants to collaborate with are outside a collective, and so at that point the collective is more or less reduced to an accounting service.

The Lisp “Curse” redemption arc

When a developer has introspective tools and some time to poke around, they are in a very good position to analyse difficult codebases, which are particularly large or are written in a way that is unnatural to the reader. However, the supposed secondary and tertiary effects of providing a programmer with sufficient power over their programming environment may be significant enough to deter cooperation according to an essay titled “The Lisp Curse”,[1] which is frequently used as an excuse to avoid confrontation over social issues affecting the *Common Lisp* community.[2] The main points made are that technical issues in sufficiently powerful languages and environments become social issues, and that having such power reduces some natural cooperative force between programmers, causing them to part ways easily and thus not achieve anything significant without external discipline. This would spell disaster for our peer production model, if it weren't that the centralised models put in contrast to a dynamic-decentralised development model can only be worse at producing stagnation and removing agency from the user; which would greatly slow any experimentation and the progress of the community. In short, the apparent incoherence of peer production should be embraced instead of lamented, as we may stand to learn a lot from incomplete prototypes when trying to produce some sort of grand unified product.

There are two apparent “solutions” that avoid this curse that we will explore. The first solution is to add the extension to the system via the implementation, forcing the community to adopt this extension, removing the agency of the user and setting them up to be screwed if the solution becomes a problem. The second is to ensure that any task is too large to tackle without cooperation, by reducing the power and efficiency of each individual user, and in doing so, eliminating all facilities for the individual creative process.

Neither solution is particularly appealing. If the provided extension has flaws that require it to be replaced, fixing the problem will affect many more clients, as opposed to if the client had more options. For example, *JavaScript* used to use a *callback* system (which is really a form of *continuation-passing style*) for interfacing with the outside world. Writing in a continuation-passing style manually was regarded by some as difficult to read and write,[3] so a *promise*-based system and

[1] `http://www.winestockwebdesign.com/Essays/Lisp_Curse.html`

[2] It has been suggested many times that one could substitute the name *Lisp* for some other name, like *JavaScript* or *Python*, and most of the article wouldn't look too wrong.

[3] Having too many callbacks in one function is often called *callback hell*.

some syntactic sugar (`async` and `await`) were introduced to make using it look like normal, synchronous code. Fortunately, promises are still compatible with continuation-passing style code, so it did not require any code to be replaced, but it still cuts a program into blocking and non-blocking parts.

> You still can't use them with exception handling or other control flow statements. You still can't call a function that returns a future from synchronous code.
>
> You've still divided your entire world into asynchronous and synchronous halves and all of the misery that entails. So, even if your language features promises or futures, its face looks an awful lot like the one on my strawman.
>
> Nystrom, 2015

Features such as asynchronous programming are very difficult to handle without getting it right the first time. The way to go forward while how to implement features is being debated is to provide a construct that subsumes it, such as providing access to implicit continuations like *Scheme* or using another unique combination of syntax and constructs that provide something like continuations, such as *monads* and the `do`-notation present in *Haskell* and *F#*, allowing a programmer to wire the continuations into their asynchronous backend; or by providing many green threads, such as in *Erlang*, and have the backend unblock the green threads when they are ready to continue. The latter two techniques facilitate composing various implementations, as they implement a common protocol which allows them to compose easily, where both implementations can be used in the same project if necessary. Thus, we are in a much safer position with *more* power to reproduce special language constructs if they become problematic, and then to unify and compose multiple implementations, allowing participants to work without a consensus. Kay (1997) succinctly states this view as "the more the language can see its own structures, the more liberated you can be from the tyranny of a single implementation."

Disempowering an individual discourages experimentation, whether it be in attempting to implement a new concept, or modifying existing code to improve it. If the goal is to develop new concepts, then having many partial implementations is better than one complete implementation; as they reflect on different views of the concept. (An implementation is never really complete, and a concept is never really finished, anyway.) One complete implementation is prone to be wrong, and a linear progression of products provides less information to work with, compared to many experiments. If the goal is to produce stable software, knowledge of such experiments and prior work is very useful. Kay had frequently called various tenents of programming a "pop culture", where participants have no knowledge of their history. Without such experiments, we have no history to investigate, even in the near future. It would thus be ideal to experiment as much as possible, and use the lessons learnt to produce a comprehensive theory and then a comprehensive implementation; and such software may not require replacement as immediately as if it were developed in a centralised or linear manner.

Disempowering a community has negative effects for creating one "unified" product, too. While inducing difficulty to go ahead with any decision that isn't unanimous leads to *a* consensus, it is an entirely arbitrary consensus, which can be as terrible as it can be good. The resulting structure may be good at giving orders and techniques to its participants, but "although everyone marches in step, the orders are usually wrong, especially when events begin to move rapidly and take unexpected turns." (Bookchin, 1971) Suppose a sufficiently large group of scientists, say all the physicists in Europe, were all told to perform the same experiment with the same hypothesis, the administration in charge would be laughed at, as it would be hugely redundant and inefficient to investigate only one problem and one hypothesis with as many physicists. However, such a situation is presented as an ideal for software design, when groups pursuing their own hypotheses and theories are considered "uncoordinated", or called "lone wolves".[4] Disempowerment also precludes the group from attempting another strategy, without another unanimous decision on which to attempt.

The most viable option is to go forward with multiple experiments, and provide participants with more power, so they may late bind and ignore the "social problems" produced by the diverse environment, in

[4]"But what about producing a baseline implementation that is usually acceptable in any case?" Such a library is only really determined after more analysis, and what constitutes the baseline is often what is required for the baseline for some other concept, which leads to an unpredictable situation. As we will soon discuss, a baseline is usually decided upon anyway.

turn provided by having reasonable control over the environment. Measuring progress by the usage and completion of one implementation of a concept is an inherently useless measure; it would not consider the progress made in another implementation or design. Such a measure "subjects people to itself" (Stirner, 1845) and inhibits their creative processes. "You're imagining big things and painting for yourself [. . .] a haunted realm to which you are called, an ideal that beckons to you. You have a fixed idea!" Progress on producing should be measured in how much of the *design space* has (or can be easily) traversed, as opposed to the completion of one product; a poor design choice could entail a final product being unfit for its purpose, but a failed prototype is always useful for further designs to avoid. With that metric, a decentral development model is greatly surperior to a centralised model.

Community effects

Peer production can better support decentralised development that doesn't come upon a consensus for whatever reason. This is a natural effect of developing sufficiently broad concepts and concepts which the theory of is frequently changing; where there are many ways to implement the concept which are not always better or worse than one another. Examples of this theme are hash tables and pattern matching, for which there are many implementations with varying performance characteristics, and for which research appears to generate a new technique or optimisation every few years.

Empirically, the Common Lisp community has not delivered on the prediction the author made, of many mutually incomprehensible incomplete implementations of any new concept. Pattern matching, again, was a concept that the author of the article believed would be implemented many times when functional programming becomes further put into the mainstream. Functional programming did indeed become fairly mainstream,[5] and since then, there has been one pattern matching library widely used. In fact, the number of common pattern matching libraries has been reduced, with *Optima* being deprecated

[5]Of course, the reader – pardon me – **Real Hackers** should be aware that functional techniques are not necessarily a replacement for object-oriented techniques, and thus can't be compared or "moved on from" to another.

in favour of the now common *Trivia*, which uses a newer compilation technique. Furthermore, there have also only been one popular lazy evaluation library (*CLAZY*), and one static type system implementation (*Coalton*). So, empirically, there has not been an explosion of poor attempts at any functional programming libraries; or we have not heard of them.

We may not have heard of any other of these genres of libraries prior to writing this book due to *network effects*, which filter out many bad libraries when there are many options; with no disposition otherwise, one is most likely to follow the design choices that appear popular. For example, there are many *testing frameworks* in Common Lisp (and it has become a joke to write another framework), but we can count most of the frameworks mentioned publicly per week on one hand. This has a sort of smoothing effect, where there are a small number of usually-good choices that are frequently recommended, greatly reducing the perceived entropy of the environment.[6]

A culture that encourages experimentation, but allows the community to settle on usually-good defaults, can remain cooperative and cohesive without risk of stagnation; the many forms of communication which do not require formal arrangements, and the rapid network effects produced by online communication can support both qualities. The community should also be aware of duplication of code when its prototypes converge, and make a goal of reducing such duplication and improving code quality; which is hard to "sell" as code quality is hard to quantify and concisely describe, but is of course necessary to support further development.

> Since execution performance is readily quantified, it is most often measured and optimized–even when increased performance is of marginal value [...] Quality and reliability of software is much more difficult to measure[7] and is

[6]However, this can also lead to *shared misery*, where users recommend what they have always used, even if they know there is a better alternative: `http://metamodular.com/Psychology/shared-misery.html`
Again, when it is difficult to change one's decision, and one is now dependent on external factors, it may become very unfortunate that clustering occurs. For example, Fediverse server usage can be seen to follow *Zipf's law*, where few servers support many users: `https://social.coop/@Stoori/100542845444542605`

[7]There are now some misleading metrics, such as the "nine nines" of uptime that an Erlang-based telephony system achieved with 14 machines, and the code quality *badges*

> therefore rarely optimized. It is human nature to "look for one's keys under the street-lamp, because that is where the light is brightest."
>
> It should be a high priority among computer scientists to provide models and methods for quantifying the quality and reliability of software, so that these can be optimized by engineers, as well as performance.
>
> Baker, 1991

Beyond our commentary on development models, there are some other issues with the article that should be pointed out. The fatalism of the article is self-fulfilling, which is not helpful for attempting to rid Lisp of its "curse". If one takes the advice and avoids powerful languages and environments, then they will appear unpopular and its espousers can use it as evidence for their babbling, with a statement like "Look, no one uses Lisp still! The Lisp Curse was right!" Assuming this development model is a problem, is the aim to actually relieve the community of this issue, or is it to badmouth the community? It also supposes that the goals then, such as Lisp machines and operating systems, are the goals now. As suggested near the end of *Always has been "malleable"*, we do want a Lisp operating system, but the special machines are less necessary with clever compilation strategies today; but more importantly, this metric conveniently ignores that the choices in "normal" operating systems almost collapsed to Windows or some Unix, equally universally affecting other vendors, regardless of technology choices; instead opting to implicitly put the blame on Lisp users for the loss of diversity of operating systems. The foreword to (Garfinkel et al., 1994) also names *TOPS-20*, the *Incompatible Time System* and *Cedar* as victims of this collapse, which were not written in Lisp or a dynamic language, but were still disposed of around the same time.

In this sense, The Lisp "Curse" is only real because we make it real. We set up metrics and constraints that promote conformist and centralist development strategies, then pat ourselves on the back for

that frequently appear on open source project documentation. The former is a product of fault-tolerant hardware design as much as it is fault-tolerant software design, and the latter won't tell off the programmer for larger design problems, say, using a poor choice of data structures.

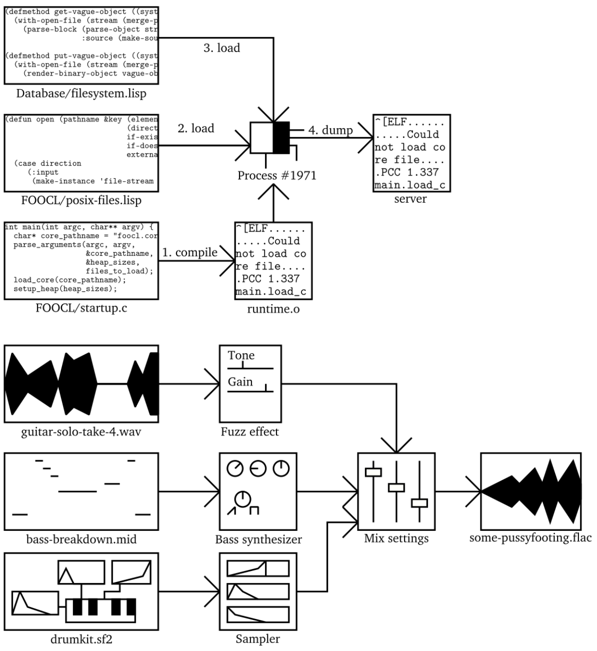
having made it impossible for anyone to test new ideas on their own. These sorts of metrics and organisational methods "treat [...] all twists and turns in the process of theorizing as more or less equivalently suspect, equivalently random flights of fancy," (Gillis, 2015) which have no real purpose. It would be interesting to see if consciously attempting to avoid this centralism would produce better quality software; allowing developers to go off on any interesting tangent, and breaking the illusion that there is one way to achieve the aim. This state should not pose any issues with sufficient communication; even if it does not appear very coordinated, or the group of diverging programmers appears uncooperative, they are much more likely to find the right approach and base for their product. The environment can also be made more condusive to finding new approaches, by opening communication channels, and publishing the *source materials* required to implement a product.

Source materials

There is a concept of a *read-write culture*, also known as *remix culture*, where "consumers" of some product also modify and reproduce the product, making them also producers. Kleiner (2010) provides a critique of the "Creative Anti-Commons", explaining that the *Creative Commons* provide many options to enforce intellectual property and impede on derivative works, while its author suggests it would support such a culture.

> It is assumed that, as an author-producer, everything you make and everything you say is your property. The right of the consumer is not mentioned, nor is the distinction between producers and consumers of culture disputed.

Such a read-write culture is fundamentally impossible without sharing *source materials*. These source materials may consist of source code, building instructions, original media, and so on. The transformations that one can make on many products without sources are very limited and of very low quality. For example, remixing a song is really only achievable by adding post-processing to the the mastered audio, such as cutting the audio and adding effects, which is difficult to get


```
(defmethod get-vague-object ((syst
  (with-open-file (stream (merge-p
    (parse-block (parse-object str
                 :source (make-sou

(defmethod put-vague-object ((syst
  (with-open-file (stream (merge-p
    (render-binary-object vague-ob
```

```
(defun open (pathname &key (elemen
                           (direct
                           if-exis
                           if-does
                           externa
  (case direction
     (:input
       (make-instance 'file-stream
```

```
int main(int argc, char** argv) {
  char* core_pathname = "foocl.cor
  parse_arguments(argc, argv,
                  &core_pathname,
                  &heap_sizes,
                  files_to_load);
  load_core(core_pathname);
  setup_heap(heap_sizes);
```

```
^[ELF......
.....Could
not load co
re file....
.PCC 1.337
main.load_c
```

Figure 2.1: Many digital products are produced in complex ways, in which the source materials are almost impossible to recover.

right, and imprecise editing can confuse the listener. Modifying the behaviour a program with only its executable binary is basically impossible, and a consumer of those examples is likely to want to do both. To achieve that, the consumer would need to have the appropriate source materials. The breadth of what can be required is illustrated in Figure 2.1.

This read-write culture is also crucial for anarchist software: our previously mentioned liberatory technology would only become stale and unusable, without constant adaptation and reciprocation of information regarding it. When our friend Jaidyn Levesque found that the Peer Production License did not have protections for preserving

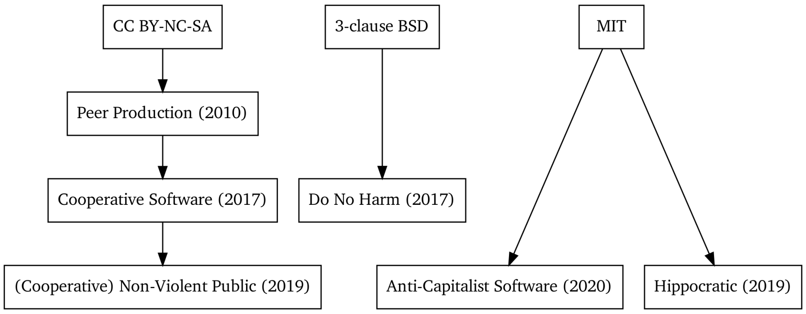

CC BY-NC-SA
Peer Production (2010)
Cooperative Software (2017)
(Cooperative) Non-Violent Public (2019)
3-clause BSD
Do No Harm (2017)
MIT
Anti-Capitalist Software (2020)
Hippocratic (2019)

Figure 2.2: A “family tree” of some licenses considered to be ethical and/or cooperative.

source materials, she modified the license to add them, producing a *Cooperative Software License* that had source protections like the *Affero General Public License,* while keeping the anti-capitalist stipulations in the Peer Production License; which may better facilitate a read-write culture.

Ethical licenses, such as the *Non-Violent Public License* (based on the Cooperative Software License), potentially provide another option for developers who are concerned that their products may be used for very bad things, instead of living with the possibility, or avoiding developing what they want. These ambiguous situations come up more frequently than one might think; frequently when a product appears politically inert, but is used in a larger context, which can be very polarized. For example, a database may be used to track the finances and resources of a collective, or it may be used to comb through data collected from mass surveillance. A developer can now begin to distinguish between the uses they believe are moral, and the uses they believe are immoral, with such a license. While such licenses have always existed, recent observations of problems in free software culture have encouraged the creation of newer ethical and cooperative licenses; some of which are shown in Figure 2.2.

An interesting question is, assuming that these licenses can be upheld in a court (which is not even clear for the *GNU General Public License*, a relatively more permissive license), are there any actors that can violate them anyway? It is unlikely states are going to be held properly accountable for violent acts they commit soon, and more unlikely that license violations will be part of hearings on such acts, but there is not much one could do to prevent *some* actors from using your products other than to not produce them. A similar thing can be said for sufficiently large corporations which no court would want to try to control. Reasoning like this is often used to avoid these licenses, and we cannot deny that it is entirely possible that they could be worked around; but these licenses can still serve as deterrents to malicious adoption. For example, Google has banned use of any software licensed under the Affero General Public License,[8] as the "virality" of the license could require them to release sources for their internal code, which would threaten their proprietary surveillance model. This threat has been a crucial tactic for keeping copyleft code out of proprietary products for years, and we can adopt these tactics, but with a better sensibility for what software freedom may be.

Another view on *copyleft* licenses is that they may implicitly threaten state violence. For example, the *Centre for a Stateless Society*, a publisher and think-tank for left-wing market anarchism, explicitly shuns both intellectual property and copyleft licensing.[9] However, the alternative (of legally permitting such uses) may, again, lead to state and corporate violence. It may be difficult to avoid using the threat of the state to reduce the possible violent acts done; but the immediate threat of viral licensing deters many actors without having to do much oneself. (Using state violence to possibly prohibit itself is at least very amusing to some users of such licenses.)

Ethical and cooperative licenses have had some success in the last five or so years, as they provide a very practical way to allow a programmer to effectively "practise what they preach". However, this idea was not understood by the authors of a so-called "Anti-Capitalist Software License" who appear to only care for the survival of a programming firm as anti-capitalism, and not any form of cooperation or collaboration, and put out statements such as:

[8] "Code licensed under the GNU Affero General Public License (AGPL) MUST NOT be used at Google." `https://opensource.google/docs/using/agpl-policy/`

[9] "C4SS loathes intellectual property and finds copyleft licenses troubling in their implicit threat of state violence." `https://c4ss.org/about`

> ACSL recognizes that the copyleft requirement of open source can be a drain on limited resources, can expose sensitive or secure information, and can put software at risk of theft.
>
> The availability of source code is less important than the organization of software labor.
>
> `https://anticapitalist.software/`

The former contradicts all three of basic anti-capitalist, peer production and software development theories: code itself should not be "sensitive information" that can be used to harm itself – *security by obscurity* is widely acknowledged as a generally bad idea, and we should not need to elaborate on what may happen when making that assumption. Publishing and reusing free source code is an excellent way to get more resources,[10] and the existence of "theft" implies support of private property and that the author somehow did not consent to publishing their own source code! We are generally convinced the authors believe an anti-capitalist software firm is somehow a closed organisation that survives off hiding information from its consumers. Users of that license may be able to stay afloat by hiding source code and enforcing private property, but they have done nothing that can remotely be called anti-capitalist. The form of organisation suggested by such a license is nothing more than plain old capitalism with its intellectual property and information hoarding; we could draw a comparison to *socialism for the rich* and call it *socialism for the producer*, but it is not even that, as it paints collaboration between producers as a disadvantage! This attempt to avoid capitalism has somehow become so morbid, that it has wrapped around and reproduced capitalism again, and it did not even require a year or two for the Anti-Capitalist Software Party to turn in on its supporters; this license was truly dead on arrival.

[10]This all began because someone realised open source code was a free replacement for labour for capitalists; suggesting that one has limited resources with open source code is to fully reject the premise one has started with!

Chapter 3

Applied language

We have now discussed how to produce liberatory software and how to form cooperative networks to support production, but we need to properly liberate and decentralise the subjects of our products now. A liberatory technology is fundamentally a product of its *society*; even if it is developed in a dynamic manner, and its authors are organised, one cannot seriously suggest that it is an improvement on anything if it provides leverage for power imbalances and abuse.

We will examine the stratification of social relationships established on “decentralised” only in software networks; including how social standing is reflected in a social network (which we may as well call our society for the sake of argument), and the ways in which a mediating moderator can make interactions more difficult than they could be. We will then examine the effects of the simplification of the means of presentation and how we can communicate on social networks, and the effects of making protocol extension part of a protocol itself.

Digital feudalism and social capitalism

<Aurora_v_kosmose> Making any and all storage dependent on having dcoin also has the issue of locking out anyone with limited monetary means. If only decentralized systems were easy.
<no-defun-allowed> Indeed, it just perpetuates hierarchy.
<aeth> I mean, that's basically the point of Urbit in particular... I'm surprised you didn't mention (or notice?) the neo-feudalism of it
<no-defun-allowed> I did notice it, and I didn't say anything because I expected it.
<aeth> heh

A conversation in #lispcafe about cryptocurrencies

Many software projects, including free-and-open-source projects and even soon-to-be "ethically licensed" projects, create hierarchies that have no need to exist. Two examples of types of project appear to continuously perpetuate hierarchies: cryptocurrencies, for self-evident reasons;[1] and discussion sites of various forms, including micro-blogging sites (including Mastodon, Twitter, and so on), and online forums (Lemmy, Lobsters, Raddle/Postmill, Reddit, and so on).

Terms like "digital feudalism" or "social capitalism" are often used to describe hierarchies in socialisation. While some will object to misusing terminology like that, we will continue to use it as such, as our readers will immediately acknowledge that these are probably not good systems to keep around, and that they are good analogies of the power structures they describe. We will provide our interpretations of these words in the context of *the Fediverse*, a network of interconnected micro-blogging servers that typically run some software that supports the *ActivityPub* messaging protocol, often using *Mastodon* or *Pleroma* – but our commentary is not limited to any specific software and protocol used. It is all to do with how this software handles moderation, and how people on this user handle interacting with each other. Perhaps the more radical projects which end up recreating these forms of

[1]Although there are some currencies that attempt to disrupt the implicit capitalism in them, such as Faircoin, by using a more direct consensus system instead of proof-of-work or proof-of-stake.

hierarchy are the most insidious, as one expects them to have made an improvement.

Digital feudalism

The first system we identified was one of digital feudalism; in which moderators, who usually would act as if they are serving their serfs, have total and essentially unaccountable control over them. On the Fediverse, this analogy is pushed even further than usual, as servers and their hosts act as the lords of this system; as it is quite difficult to transfer an account between servers, and that notion of "transferring" loses identity anyway.

Though we have not found any proponents of feudalism fortunately, attempts at suggesting this hierarchy promotes "autonomy" sound as ridiculous as proponents of "libertarian" capitalism (some union of propertarians, American libertarians, "anarcho"-capitalists, and so on).

As hinted at before, excusal of hierarchies when software is involved is not exclusive to capitalist apologists! No followers of an ideology or lack thereof are particularly more aware of the kind of power dynamics they partake in than others, despite any claims otherwise; our fellow radicals appear to forget what they learned about *the People's Stick* and all those other analogies and critiques for power, when they believe they can do self-contradictory things like forming "an organisational model and governance that puts marginalised voices first"[2] and provide autonomy for their users by strengthening moderation and centralising power into one in-group. In the case that a server operator must step in, they must have a damned good reason, their actions should almost always be reversible, and if not, they better be able to be held accountable if they screw up. This naïve trust in "moderators doing moderation" lead one of our colleagues to write a corollary to Bakunin:

> When the people are being beaten with a stick, they are not much happier if it is a particuarly efficient stick, that allows many people to be beaten at once.
>
> Dlorah, 2020

[2]Clipped from `https://parast.at/`

The efficient stick is the norm for Fediverse moderation, as improving the efficiency of moderation is generally considered a good idea, when moderators can only be trusted to do the right thing; a relationship that may or may not be acceptable in the long term (our experience leads us to think it is not usually acceptable though).

But unlike lords of feudal society, the powers associated with moderation have significant toil attached to them! While most cases a moderator may have to scan through are benign, some cases can require time to research what has happened, if it has happened before, in order to reach what they believe is the fairest solution, and some other cases are so blatantly horrific that moderators looking at them are severely disturbed.

We have heard about both kinds of cases from our friends, but as we trend towards larger servers with fewer moderators per user, moderation can even become traumatic: a 2018 lawsuit concluded recently with Facebook paying out $52 million to moderators whom "suffered psychological trauma and symptoms of post-traumatic stress disorder from repeatedly reviewing violent images" (Wong, 2020). While we are some magnitudes of growth away from, say, having to filter through violent and gruesome imagery frequently, a solution appear that would relieve moderators of most of their current activities may be well received, and we could agree that they would distribute their powers to the community in return; with no doubt that the distribution of stress from some of the community would greatly improve their mood, resulting in fewer incidents to moderate! *Collaborative filtering*, succinctly describable as copying actions taken by users that take actions similar to a user, would be an ideal mechanism for distributed moderation; it has been used on *Usenet* to recommend good articles (Syst et al., 1997), and moderation is basically the reverse of that. Using a distributed technique "should be more than enough to effectively automate most of the role of [a moderator]" (Dlorah, 2020).

In short, it is very hard to say who wins in digital feudalism; many users are at the whim of tyrannical moderators, or cannot contribute without any good moderators, and the magnitude and quality of the work moderators must do cannot be healthy for them in the long term. Providing users with the ability to collaborate and filter their own environment, and reducing the stress of moderators would universally improve the subjective quality of the Fediverse.

Social capitalism

Even outside the realm of moderation, there is also an imposition of some kind of "ownership" of discourse, and huge variations in social standing, which appear to directly affect how one communicates with others. We call this *social capitalism*, because just like actual capitalism, one with enough capital can do almost anything, regardless of harm caused, and escape punishment just by having *social capital*. The words uttered when one gets on another's nerves are no longer "fuck off" or "get lost", they now try to show force by indicating ownership of the conversation! "Get off my timeline!" "Fuck outta of my mentions!"[3] Of course, boundaries must be understood, and some topics simply shall not be discussed, but when one's reasoning is no longer rooted in any form of respect and is a threat in itself, then their reasoning is completely bogus.

The belief that a conversation somehow belongs to someone, and that they have some authority over it even, is highly erroneous. If we were to analyse conversation as if it were a commodity, it is clear that any notion of value is produced by whoever continues and reads the conversation. The role of a host of any shape is greatly overstated; Kleiner notes that "the real value of [sites that share community-created value are] not created by the developers of the site; rather, it is created by the people who upload [content] to the site." (Kleiner, 2010)

If we were to analyse it as a normal conversation, starting a conversation obviously does not give one any control of it. This assumption leads to obviously hierarchical and asymmetric dynamics, including a suggestion we recently read that "[we need] posts that can only be replied to by mutuals but public and shareable freely". While this admittedly sounds very nice, it is trivial to abuse, and is a clear continuation of this myth of social property. And of course, indicating that this would produce a *safe space* is also inaccurate; as with politeness outside the Internet, one who publishes publicly is expected to treat their readers as they would like to be treated themself, and there are no such guarantees in an asymmetrical environment.

These examples are quite nasty, and we may sound like we can only imagine the worst by highlighting only those. Humourous comments

[3]Mentions do force one to read something they don't want to, though; but it is possible to mute conversations and not hear more of it.

such as "Reply to win *a valuable thing*", and "*Blatantly wrong statement*: change my view" where the reader is unable to reply, are less grim examples of this imbalance which demonstrate why it can be problematic, without causing significant harm.

Both social capitalism and digital feudalism are forms of *asymmetrically* in power, which is anathematic to socialisation and establishing any kind of relationship or mutual trust, allowing one to write whatever and repress anyone seeking to hold them accountable. It is no less important to abolish social asymmetry than it is to abolish asymmetry in computer systems, and a decentralised computer system cannot try to enforce the former with the latter. While the demands ten years ago may have been to establish decentralised networks, we now demand social decentralisation of the resulting networks which are only decentralised in terms of software.

The means of presentation and extension

The inverse of a "radical" proprietary software-producing firm as imagined in *Source materials* has appeared many times before, with "free software" projects that are run like proprietary projects, such as the *Signal* messaging program. The Signal developers have many arbitrary and unenforceable constraints on their users, like disallowing users from using their own modified clients[4] because it somehow slows down making changes. Long ago, we wrote about how this is the opposite of what has actually happened in the Common Lisp community, how network effects would shift a userbase to unanimously using or not using a feature, and how some abstract thinking can relieve implementers of having to deal with updates:

> Other standards like *Bordeaux threads* and *Gray streams* have seen widespread implementation [in implementations of Common Lisp], despite not being in the ANSI Common Lisp standard, or other authoritative documentation. De-

[4]See this rule in action in `https://github.com/LibreSignal/LibreSignal/issues/37` and a blog post about how it must feel to be one of those puny decentralised protocols that can't do anything at `https://signal.org/blog/the-ecosystem-is-moving/`. We strongly advise against reading either sober.

> facto standards such as these are very popular simply because everyone can use them, creating a kind of network effect in a userbase.
>
> [Moxie] attributes his issues to “XEPs that aren’t consistently applied anywhere”, but if someone wants to run a server or entry point into a federated network, they should be up-to-date on new extensions; not doing so repels users. Thanks to the network effect previously mentioned, users of inferior servers can move to better ones, probably the ones that their connections use.
>
> With a layer of indirection, you can just push out “code” (well, it can be Turing complete if you want) to add new features and types to a program.
>
> Patton, 2018

It can easily be said that this was a poor attempt at arguing whatever point was being made then; but when “adding features” is part of one’s protocol, then one only has to implement that protocol once. This line of thinking lead to the *replicated object system* design of *Netfarm*, one of our experiments in producing what we would consider anarchist software, and the implications of such a design may be very interesting.

The *means of presentation* are the forms in which a communication platform allows one to express or present an idea. For example, a microblogging site allows expression in text with fewer than some number of characters (often allowing uploading other files, displaying images inline), a technical journal allows expression in articles, a forum allows expression primarily in posts and comments, and so on.

Diverse information demands diverse representations, and forcing information across many services that each handle a specific means of presentation and representation creates fragmentation and hampers discoverability.

> The first one is the Marxist notion of a general intellect. With today’s platforms, we are not facing such a phenomenon. Our use of contemporary digital platforms is extremely fragmented and there is no such thing as progress of the collective intelligence of the entire working class or society. Citizens are facing relentless efforts deployed by

digital capitalists to fragment, standardise, and 'taskify' their activities and their very existences.

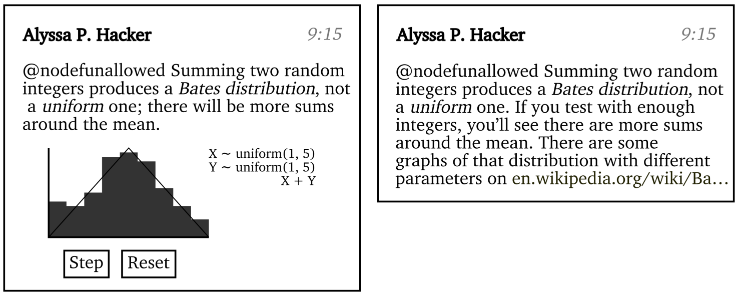
Casilli and Marsili, 2018

Opening up a platform to accept any means of presentation would annihilate any gimmicks or distinguishing features of it, but that may be the most interesting approach possible, as such a platform could present any information in the most appropriate way. Even with the media that common platforms support, support is limited to a lousy subset, usually prohibiting typesetting of mathematical equations, referencing, and sometimes even basic formatting.[5] While, say, Mastodon and LaTeX both transmit text in some form, the former is evidently more suitable for near-real-time communication, whereas the latter is more suitable for long-form writings, such as this book. It would not be hard to give the former the capabilities of the latter: some servers already provide mathematical typesetting and formatting options.

Beyond that, there are many more advanced means of presentation which can be immediately seen to have uses, that are not even close to implementable on the common platforms of today, such as viewing three-dimensional objects, dynamic and randomised mediums like soundscapes, and simulations of natural phenomena (such as the demonstration of approximating a *Bates distribution* in Figure 3.1). Should our dreams of casual programming (as mentioned in Chapter 1) come true, it would not be hard to believe sharing programs directly would be commonplace.

However, it should not be necessary to modify a server to provide these new means of presentation. To modify the behaviour of the platform without modifying the platform itself, the platform will have to communicate in programs and/or objects that describe and present themselves, instead of text and/or plain media formats. Implementing such a platform may be very difficult to begin with, but it is much more tedious to incrementally extend a platform, such as the Web or some protocol residing on it such as ActivityPub, that merely display documents and texts. For reference, the *Chromium* web browser con-

[5]It may be argued that opening up the means of presentation may make it inaccessible, as some formats are difficult to interpret by some users. However, alternative presentations can be recovered at the least, which is not the case for workarounds for less expressive means of presentation.



Alyssa P. Hacker *9:15*

@nodefunallowed Summing two random integers produces a *Bates distribution*, not a *uniform* one; there will be more sums around the mean.

Alyssa P. Hacker *9:15*

@nodefunallowed Summing two random integers produces a *Bates distribution*, not a *uniform* one. If you test with enough integers, you'll see there are more sums around the mean. There are some graphs of that distribution with different parameters on en.wikipedia.org/wiki/Ba…

Figure 3.1: Providing a simulation that the reader can run can be much more illustrative than using a static medium to make a point.

tains about 34 million lines of code, and a complete *Squeak* Smalltalk environment contains only about 5 million; 4 million lines in *OpenSmalltalk*, a just-in-time compiling virtual machine, and 1 million lines in the Smalltalk environment, including a graphical environment, bytecode compiler, debugger and class browser, and some other components that do not appear in a browser, including an email client, graphics framework, and package manager.[6]

Such a total lines-of-code count could be misleading as to how much work an implementer has to do; much of the complexity would be reusable with almost no modification per implementation, as it would make up the distributed "image" the client retrieves. Furthermore, a large subset of the functionality of modern Squeak could likely be significantly smaller; the first implementation of the Squeak virtual machine was written in only about eight thousand lines of code (Ingalls et al., 1997).

This is still magnitudes more code than the more common reaction to the complexity of the Web would require, in which protocols are proposed that are intended to only display "documents". As we have mentioned, what constitutes presenting a text document is very dubious, yet most suggestions provide very little to work with when tasked with book-making. *Gemini* is one example of this reaction, accompa-

[6]There are quite a few debugging tools in a browser; many provide debuggers, profilers and whatnot, but they do not usually contain complete programming environments.

nied with some vague, nostalgic association with the “essence of the Web”. It is supposed you can write a Gemini client in less than a few hundred lines of code,[7] yet the end result is sending uninteractive text with minimal formatting across a network. For what it can produce, the result is hardly an advancement over the printing press! We don’t need a computer to publish text documents; pen and paper, and either patience or a photocopier would suffice. And, of course, you can draw and format the text however you like with pen and paper. We shall conclude with part of a much more interesting suggestion for a better book:

> We do not feel that technology is a *necessary* constituent for this process any more than is the book. It may, however, provide us with a better “book”, one which is active [...] rather than passive. It may be something with the attention grabbing powers of TV, but controllable by [the user] rather than the networks. It can be like a piano: a product of technology, yes, but one which can be a tool, a toy, a medium of expression, a source of unending pleasure and delight... and as with most gadgets in unenlightened hands, a terrible drudge!
>
> Kay, 1972

With some consideration of protocol self-extension, we can happily intermingle many different modes of expression and representation. With the normalisation of protocol self-extension, we could greatly reduce the toil protocol implementers must perform, while not compromising on the expression of the protocol. Messing with a protocol may become an emerging mode of expression in itself, like Dadaists had done for poetry and art, and demoscene programmers do with the obscure features (optionally to be read in air-quotes) of computer hardware. In any case, an appropriate meta-means of presentation can vastly expand the information that can be communicated on one protocol and one set of implementations.

[7]“Experiments suggest that a very basic interactive client takes more like a minimum of 100 lines of code, and a comfortable fit and moderate feature completeness need more like 200 lines.” `https://gemini.circumlunar.space/docs/faq.html`

Conclusion

This book describes some of the things we can do with dynamic systems and peer production, and what we should be able to do in the future with that support and an analysis of eliminating hierarchies, in all the forms that they may appear in our software.

We hope for the annihilation of odd constraints and the decentralisation of all things, and only then could we have properly liberated computing. To achieve this, we must dissolve hierarchies in our software, our means of production, and our social applications of software. What would be produced would be formless and incomparable to any computing systems that exist now; but what could emerge from such formlessness and adaptability would be absolutely beautiful!

Appendix A

Cooperative Software License

This book is licensed under the Cooperative Software License, written by Jaidyn Levesque whom noticed the Peer Production License (as written in an appendix of Kleiner, 2010) did not offer protections for source material, as discussed in Chapter 2.

You are very welcome to get the LaTeX source for this book from `https://gitlab.com/cal-coop/software-anarchy/` and modify it for your needs; but we ask that if you change the contents in the book (as opposed to, say, changing the layout to fit your preferred printing medium), you make it clear you have modified it.

The License is quite long; if it would be preferable not to print it, you are also welcome to replace the following copy of the license with a reference to a copy online. We have provided comments at the end of the main source file for this book (`software-anarchy.tex`) which may be used to replace this text with a reference to the License.

Cooperative Software License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS COOPERATIVE SOFTWARE LICENSE (*LICENSE*). THE WORK IS PROTECTED BY COPYRIGHT AND ALL OTHER APPLICABLE LAWS.

ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED. BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED IN THIS LICENSE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN AS CONSIDERATION FOR ACCEPTING THE TERMS AND CONDITIONS OF THIS LICENSE AND FOR AGREEING TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS LICENSE.

1. Definitions

a. *Adaptation* means a work based upon the *Work*, or upon the *Work* and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the *Work* may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an *Adaptation* for the purpose of this License. For the avoidance of doubt, where the *Work* is a musical work, performance or phonogram, the synchronization of the *Work* in timed-relation with a moving image (*synching*) will be considered an *Adaptation* for the purpose of this License.

b. *Collection* means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the *Work* is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a *Collection* will not be considered an *Adaptation* (as defined above) for the purposes of this License.

c. *Distribute* means to make available to the public the original and copies of the Work or *Adaptation*, as appropriate, through sale, gift or any other transfer of possession or ownership.

d. *Licensor* means the individual, individuals, entity or entities that offer(s) the *Work* under the terms of this License.

e. *Original Author* means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the *Work* or if no individual or entity can be identified, the publisher; and in addition:

   (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore;

   (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and,

   (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. *Work* means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. *You* means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the *Work*, or who has received express permission from the *Licensor* to exercise rights under this License despite a previous violation.

h. *Publicly Perform* means to perform public recitations of the *Work* and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public *Work*s in such a way that members of the public may access these *Work*s from a place and at a place individually chosen by them; to perform the *Work* to the public by any means or process and the communication to the public of the performances of the *Work*, including by public digital performance; to broadcast and rebroadcast the *Work* by any means including signs, sounds or images.

i. *Reproduce* means to make copies of the *Work* by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the *Work*, including storage of a protected performance or phonogram in digital form or other electronic medium.

j. *Software* means any digital *Work* which, through use of a third-party piece of Software or through the direct usage of itself on a computer system, the memory of the computer is modified dynamically or semi-dynamically. *Software*, secondly, processes or interprets information.

k. *Source Code* means the human-readable form of *Software* through which the *Original Author* and/or *Distributor* originally created, derived, and/or modified it.

l. *Network Service* means the use of a piece of *Software* to interpret or modify information that is subsequently and directly served to users over a computer network.

2. Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant

Subject to the terms and conditions of this License, *Licensor* hereby grants *You* a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the *Work* as stated below:

a. to *Reproduce* the *Work*, to incorporate the *Work* into one or more *Collections*, and to *Reproduce* the *Work* as incorporated in the *Collections*;

b. to create and *Reproduce Adaptations* provided that any such *Adaptation*, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original *Work*. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to *Distribute* and *Publicly Perform* the *Work* including as incorporated in *Collections*; and,

d. to *Distribute* and *Publicly Perform Adaptations*. The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(g), all rights not expressly granted by *Licensor* are hereby reserved, including but not limited to the rights set forth in Section 4(h).

4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. *You* may *Distribute* or *Publicly Perform* the *Work* only under the terms of this License. *You* must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the *Work You Distribute* or *Publicly Perform*. *You* may not offer or impose any terms on the *Work* that restrict the terms of this License or the ability of the recipient of the *Work* to exercise the rights granted to that recipient under the terms of the License. *You* may not sublicense the *Work*. *You* must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the *Work You Distribute* or *Publicly Perform*. When *You Distribute* or Publicly Perform the *Work*, *You* may not impose any effective technological measures on the *Work* that restrict the ability of a recipient of the *Work* from *You* to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the *Work* as incorporated in a *Collection*, but this does not require the *Collection* apart from the *Work* itself to be made subject to the terms of this License. If *You* create a *Collection*, upon notice from any *Licensor You* must, to the extent practicable, remove from the *Collection* any credit as required by Section 4(f), as requested. If *You* create an *Adaptation*, upon notice from any *Licensor You* must, to the extent practicable, remove from the *Adaptation* any credit as required by Section 4(f), as requested.

b. Subject to the exception in Section 4(e), you may not exercise any of the rights granted to *You* in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the *Work* for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

c. If the *Work* meets the definition of *Software*, *You* may exercise the rights granted in Section 3 only if *You* provide a copy of the corresponding *Source Code* from which the *Work* was derived in digital form, or *You* provide a URI for the corresponding *Source Code* of the *Work*, to any recipients upon request.

d. If the *Work* is used as or for a *Network Service*, *You* may exercise the rights granted in Section 3 only if *You* provide a copy of the corresponding *Source Code* from which the *Work* was derived in digital form, or *You* provide a URI for the corresponding *Source Code* to the *Work*, to any recipients of the data served or modified by the Network Service.

e. *You* may exercise the rights granted in Section 3 for commercial purposes only if:

    i. *You* are a worker-owned business or worker-owned collective; and

    ii. after tax, all financial gain, surplus, profits and benefits produced by the business or collective are distributed among the worker-owners; and

    iii. *You* are not using such rights on behalf of a business other than those specified in 4(e.i) and elaborated upon in 4(e.ii), nor are using such rights as a proxy on behalf of a business with the intent to circumvent the aforementioned restrictions on such a business.

f. Any use by a business that is privately owned and managed, and that seeks to generate profit from the labor of employees paid by salary or other wages, is not permitted under this license.

g. If *You Distribute*, or *Publicly Perform* the *Work* or any *Adaptations* or *Collections*, *You* must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the *Work* and provide, reasonable to the medium or means *You* are utilizing:

    i. the name of the *Original Author* (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or *Licensor* designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution (*Attribution Parties*) in *Licensor*'s copyright notice, terms of service or by other reasonable means, the name of such party or parties;

ii. the title of the *Work* if supplied;

iii. to the extent reasonably practicable, the URI, if any, that *Licensor* specifies to be associated with the *Work*, unless such URI does not refer to the copyright notice or licensing information for the *Work*; and,

iv. consistent with Section 3(b), in the case of an *Adaptation*, a credit identifying the use of the *Work* in the *Adaptation* (e.g., "French translation of the *Work* by Original Author," or "Screenplay based on original *Work* by Original Author").

The credit required by this Section 4(g) may be implemented in any reasonable manner; provided, however, that in the case of a *Adaptation* or *Collection*, at a minimum such credit will appear, if a credit for all contributing authors of the *Adaptation* or *Collection* appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, *You* may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising *You*r rights under this License, *You* may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the *Original Author*, *Licensor* and/or Attribution Parties, as appropriate, of *You* or *You*r use of the *Work*, without the separate, express prior written permission of the Original Author, *Licensor* and/or Attribution Parties.

h. For the avoidance of doubt:

   i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the *Licensor* reserves the exclusive right to collect such royalties for any exercise by *You* of the rights granted under this License;

   ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the *Licensor* reserves the exclusive right to collect such royalties for any exercise by *You* of the rights granted under this License if *You*r exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,
   iii. Voluntary License Schemes. The *Licensor* reserves the right to collect royalties, whether individually or, in the event that the *Licensor* is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by *You* of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b).

i. Except as otherwise agreed in writing by the *Licensor* or as may be otherwise permitted by applicable law, if *You Reproduce*, *Distribute* or *Publicly Perform* the *Work* either by itself or as part of any *Adaptations* or *Collections*, *You* must not distort, mutilate, modify or take other derogatory action in relation to the *Work* which would be prejudicial to the *Original Author*'s honor or reputation. *Licensor* agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make *Adaptations*) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the *Original Author*'s honor and reputation, the *Licensor* will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable *You* to reasonably exercise *You*r right under Section 3(b) of this License (right to make *Adaptations*) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO

REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by *You* of the terms of this License. Individuals or entities who have received *Adaptations* or *Collections* from *You* under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the *Work*). Notwithstanding the above, *Licensor* reserves the right to release the *Work* under different license terms or to stop distributing the *Work* at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time *You Distribute* or *Publicly Perform* the *Work* or a *Collection,* the *Licensor* offers to the recipient a license to the *Work* on the same terms and conditions as the license granted to *You* under this License.

b. Each time *You Distribute* or *Publicly Perform* an *Adaptation, Licensor* offers to the recipient a license to the original *Work* on the same terms and conditions as the license granted to *You* under this License.

c. If the *Work* is classified as *Software,* each time *You Distribute* or *Publicly Perform* an *Adaptation, Licensor* offers to the recipient a copy and/or URI of the corresponding *Source Code* on the same terms and conditions as the license granted to *You* under this License.

d. If the *Work* is used as a *Network Service,* each time *You Distribute* or *Publicly Perform* an *Adaptation,* or serve data derived from the *Software,* the *Licensor* offers to any recipients of the data a copy and/or URI of the corresponding *Source Code* on the same terms and conditions as the license granted to *You* under this License.

e. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

f. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

g. This License constitutes the entire agreement between the parties with respect to the *Work* licensed here. There are no understandings, agreements or representations with respect to the *Work* not specified here. *Licensor* shall not be bound by any additional provisions that may appear in any communication from *You*. This License may not be modified without the mutual written agreement of the *Licensor* and *You*.

h. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Appendix B

Bibliography

Bibliography

Abelson, H., & Sussman, G. (1996). *Structure and interpretation of computer programs*. MIT Press. http://mitpress.mit.edu/sicp/

Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors* (Doctoral dissertation). KTH Royal Institute of Technology. http://erlang.org/download/armstrong_thesis_2003.pdf

Baker, H. (1991). Dubious achievement. *Communications of the ACM*. https://web.archive.org/web/20160321151425/www.pipeline.com/~hbaker1/letters/CACM-DubiousAchievement.html

Bennett, J. G. (2009). *Talks on Beelzebub's tales*. Bennett Books.

Bookchin, M. (1971). *Post scarcity anarchism*. Ramparts Press. https://theanarchistlibrary.org/library/murray-bookchin-post-scarcity-anarchism-1

Bookchin, M. (1978). Why doing the impossible is the most rational thing we can do. https://theanarchistlibrary.org/library/murray-bookchin-why-doing-the-impossible-is-the-most-rational-thing-we-can-do

Bracha, G. (2013). Does thought crime pay? *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming and Applicaitons: Software for Humanity*.

Casilli, A., & Marsili, L. (2018). Earn money online: The politics of microwork and machines. *Green European Journal*. http://www.casilli.fr/2018/05/18/we-need-a-political-subject-capable-to-think-an-alternative-to-digital-labor-interview-green-european-journal-vol-17-2018/

Chodorkoff, D. (2010). Alternative technology and urban reconstruction. https://libcom.org/library/alternative-technology-urban-reconstruction

Dlorah, N. (2020). A Parastatal problem. http://noslebadlorah.altervista.org/parastatal.html

Gabriel, R. (2002). Objects have failed. https://dreamsongs.com/ObjectsHaveFailedNarrative.html

Garfinkel, S., Weise, D., & Strassmann, S. (1994). *The Unix-Haters handbook*. International Data Group. http://simson.net/ref/ugh.pdf

Gillis, W. (2015). Science as radicalism. https://theanarchistlibrary.org/library/science-as-radicalism-william-gillis

Ingalls, D. (2017). Yesterday's computer of tomorrow: The Xerox Alto. https://www.youtube.com/watch?v=NqKyHEJe9_w

Ingalls, D., Kaehler, T., Maloney, Wallace, & Kay, A. (1997). Back to the future: The story of Squeak, a practical Smalltalk written in itself. *OOPSLA*. http://ftp.squeak.org/docs/OOPSLA.Squeak.html

Kay, A. (1972). A personal computer for children of all ages. http://www.vpri.org/pdf/hc_pers_comp_for_children.pdf

Kay, A. (1997). The computer revolution hasn't happened yet [We mostly used the transcription from https://moryton.blogspot.com/2007/12/computer-revolution-hasnt-happened-yet.html.]. https://www.youtube.com/watch?v=oKg1hTOQXoY

Kleiner, D. (2010). *The Telekommunist manifesto*. Network Notebooks 03. http://media.telekommunisten.net/manifesto.pdf

Mumford, L. (1964). Authoritarian and democratic technics. https://theanarchistlibrary.org/library/lewis-mumford-authoritarian-and-democratic-technics

Nystrom, B. (2015). What colour is your function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Patton, H. (2018). Reflections on "Reflections: The ecosystem is moving". https://gitlab.com/cal-coop/netfarm/deprecated/netfarm-docs/-/blob/master/general/reflections-on-reflections.md

Patton, H. (2019). Masochist programming, or who's doing the computation now? http://txti.es/masochistprogramming

Stirner, M. (1845). The unique and its property. https://theanarchistlibrary.org/library/max-stirner-the-unique-and-its-property

Strandh, R. (2013). CLOSOS: Specification of a Lisp operating system. http://metamodular.com/closos.pdf

Syst, R., Konstan, J., Miller, B., Maltz, D., Herlocker, J., Gordon, L., & Riedl, J. (1997). Collaborative filtering to Usenet news. *Communications of The ACM - CACM*.

Ungar, D. (1995). Annotating objects for transport to other worlds. https://bluishcoder.co.nz/self/transporter.pdf

Wong, Q. (2020). Facebook reaches $52m settlement with ex-content moderators over PTSD. https://www.cnet.com/news/facebook-to-pay-52-million-in-settlement-with-former-content-moderators-suffering-from-ptsd/

Zawinski, J. (2000). The Lemacs/FSFmacs schism. https://www.jwz.org/doc/lemacs.html