up2020.bib
*eid+url+urldateeid*url*urlyearaccessed onverbacyan

Unix Philosophy 2020

Casper Ti. Vector <CasperVector@gmail.com>
https://gitea.com/CasperVector/up2020

2019-08/2019-09; version 0.1.4

Contents

Foreword

I began learning Linux in the end of 2008, and using it in the beginning of 2009; I have always used Linux as my main operating system thereafter. During the course of using Linux and interacting with the ecosystem of Unix-like operating systems, I got attracted to the notion of Unix philosophy, and have benefited enormously from it. (For more details on the origin of this document, see the Afterword.) This document is a compendium of my thoughts about the Unix philosophy, organised into three main parts, as are summarised below.

In the first part, we will first discuss the question "**what is the essence of the Unix philosophy**?", and I will argue that it is the minimisation of the cognitive complexity of the system while almost satisfying the requirements. These years, many people think the philosophy, which originated from legacy limitations on computational resources back in the 1970s, is no longer applicable in software development. So **is the Unix philosophy still relevant**, and will it still be relevant in the 2020s? My answer is "yes, and more relevant than when it was born". As a conclusion to this part, I will give my experience about **how the Unix philosophy can be followed in actual use and development**, to the advantage of the user / developer.

In the second part, we will move our focus out of the field of programming, and explore **the applicability of the Unix philosophy in science, technology and society, and the limits to that applicability**. After the discussion on science and technology, I will consider minimalism in philosophy, literature and arts, and attempt to find a cognitive basis for minimalism and its limits; the theory on the cognitive limits to minimalism will then be used to explain why society often does not work like Unix. As a conclusion, we will discuss **how minimalism can be adhered to, especially by the ordinary person**, to boost one's efficiency in everyday work.

In the third part, we will return to programming, but this time we will begin with a digression about Lisp and especially Scheme, which also encourage minimalism. In order to understand the minimalism of Lisp, we will examine the notion of homoiconicity, which I find as important as complexity. I will then try to answer "**what is the root of the antagonism between Unix and Lisp**?", and finally explore the benefits and feasibility of **a minimalist language framework which adopts the best from Lisp and C**, and which also avoids the weaknesses.

Before actually diving into the main parts, it should be noted that this document is meant to be read critically, for two reasons. First, what I write here is just my personal opinions about the Unix philosophy, and the latter is just one way, among the many ones, to model the world. Second, I still know little about unfamiliar research fields like philosophy, cognitive science, and programming languages, so the contents about these topics are highly tentative, and there may well be points that turn out to be amateurish or even ludicrous. I attempted to avoid making mistakes in this document, and please feel free to tell me if you find any.

This document is an extended transcript of my technical report, *Unix philosophy in the contemporary era*casper2018, on 10 November 2018 for the Linux Club of Peking University; I consider it as the culmination of my 10-year experience in the open-source community, and dedicate it to the 50th anniversary of Unixsalus2005. I enjoyed making the report as well as this document, and wish you could also have some fun reading through the document, whether you are a newcomer to the community, or an expert almost 3 times my age.

How to read this document

Due to feedback from multiple readers, it has been brought to my attention that the nature of this document as a compendium – a collection of thoughts about various subjects – can lead to some confusions: who are expected to read this document, and how should they read the document? Here I will briefly explain the logical relation between the parts / sections in this document, and then attempt to give a guide to readers with various backgrounds; please feel free to tell me if you have any advice on how to make this document more accessible.

01–07: modernising the Unix philosophy	**13–18:** minimalism outside of programming	**22–24:** Unix philosophy and Lisp
08–10: significance of the philosophy	**19:** minimalism from a cognitive viewpoint	**25–27:** reconciling Unix and Lisp
11–12: real-world Unix philosophy	**20–21:** minimalism for society and the individual	

Above is an outline of the logical organisation of this document. In my opinion, nowadays the Unix philosophy is often neglected, questioned or even distorted, due to misunderstanding about it. So in the first part, I will first present my attempt at a modernised formulation of the philosophy in Sections **??**–**??**. Using the formulation as the theoretical foundation, I will discuss significance of the Unix philosophy in the contemporary context in Sections **??**–**??**, and real-world applications of the philosophy in Sections **??**–**??**.

I disagree with some people's opinion that the Unix philosophy was now merely an aesthetic, because I believe it is rooted in human recognition, which results in its inherent importance. In the second part, using the above-mentioned formulation, I will first examine some analogues to the Unix philosophy outside of the field of programming in Sections **??**–**??**. After that, I will present my cognitive justification for the philosophy in Section **??**, and discuss its value for society and the individual in Sections **??**–**??**, similar to the last two sections in the first part.

In the recent years, it has gradually become my opinion that expertise from the Unix and Lisp circles can be joined to build systems that are more Unix-ish than was possible before. So in the third part, I will first discuss the relation between Lisp and the previously formulated Unix philosophy in Sections **??**–**??**. Finally, I will examine the benefits of the proposed Unix / Lisp reconciliation, and how this reconciliation can be achieved in Sections **??**–**??**.

It is generally recommended to read this document in the original order, and just skip parts / sections you are less interested in; however, Section **??** lays the theoretical foundation for this document, so you should at least skim through it. Quite a few examples from the "init war" are used in the first part, so people entangled in related events are advised to read that part. Those who like philosophical discussions might feel interested in the second part. Unix developers interested in programming languages, as well as researchers of these languages, will perhaps find the third part attractive. Footnotes are generally less essential contents, and often require deeper understanding of the subject matter.

01 An introduction to the history of Unix

CTSSwiki:ctss (first released in 1961), widely thought to be the first time-sharing operating system in history, was quite successful, and its success resulted in a much more ambitious project called Multicswiki:multics (with development starting in 1964). The project did not deliver a commercially usable system until 1969multicians:history despite joint efforts from MIT, GE and Bell Labs, as the system was too complicated for the human and computational resources available at that time[1]. Bell Labs withdrew from the project in 1969, a few months before the first commercial release of Multics; Ken Thompson and Dennis Ritchie, previosuly working on the project, went on to develop another operating system to fulfill their own needs, and the system would soon get the name "Unix"wiki:unixhist. In order to make the new system usable and manageable on a (sort of) spare PDP-7 minicomputer at Bell Labs, Thompson made major simplifications to the Multics design and only adopted certain key elements like the hierarchical file system and the shellseibel2009.

The 1970s saw the growth and spread of Unix (*cf.* wiki:unixhist for the details), and in my opinion the most important historical event during this period was the publication of *Lions' Commentary on Unix v6*wiki:lions[2] in 1976, which greatly spurred the propagation of Unix in universities. In the early 1980s, commercial versions of Unix by multiple vendors appeared, but AT&T, the owner of Bell Labs at that time, was barred from commercialising Unix due to antitrust restrictions. This changed in 1983, when the Bell System was broken up, and AT&T quickly commercialised Unix and restricted the distribution of its source code. The restriction on code exchange greatly contributed to the already looming fragmentation of Unix, resulting in what we now call the "Unix wars"raymond2003a; the wars, in combination with the negligence of 80x86 PCs' potentials by the Unix circle, led to the pitiful decline in popularity of Unix in the 1990s.

In 1991, Linus Torvalds started working on his own operating system kernel, which went on to become Linux; the combination of userspace tools from the GNU project (starting from 1985) and the Linux kernel achieved the goal of providing a self-hosting Unix-like environment that is free / open-source and low-cost[3], and kickstarted the GNU/Linux ecosystem, which is perhaps the most important frontier in the open-source movement. Currently, the most popular Unix-like systems are undisputedly Linux and BSD, while commercial systems like Solaris and QNX have minute market shares; an unpopular but important system, which I will introduce in Section **??**, is Plan 9 from Bell Labs (first released in 1992).

Before ending this section which has hitherto been mostly non-technical, I would like to emphasise the three techical considerations which "influenced the design of Unix" according to Thompson and Ritchie themselvesritchie1974, and all of these points will be discussed in later sections:

- **Friendliness to programmers**: Unix was designed to boost the productivity of the user as a programmer; on the other hand, we will also discuss the value of the Unix methodology from a user's perspective in Section **??**.
- **Simplicity**: the hardware limitations on machines accessible at Bell Labs around 1970 resulted in the pursuit of economy and elegance in Unix; as the limitations are no more, is simplicity now only an aesthetic? Let's see in Sections **??**–**??**.
- **Self-hosting**: even the earliest Unix systems were able to be maintained independent of machines running other systems; this requires self-bootstrapping, and its implications will be discussed in Sections **??** & **??**–**??**.

02 A taste of shell scripting

It was mentioned in last section that the shell was among the few design elements borrowed by Unix from Multics; in fact, as the main way that the user interacts with the systemritchie1974 (apart from the graphical interface), the shell is also a component where the design concepts of Unix are best reflected. As an example, we can consider the classic word frequency sorting problemrobbins2005 – write a program to output the n most frequent words along with their frequencies; this problem attracted answers from Donald Knuth, Doug McIlroy and David Hanson. The programs by Knuth and Hanson were written from scratch respectively in Pascal and C, and each took a few hours to write and debug; the program

[1] Actually, what the system required is no more than the hardware of a low-end home router now; in comparison, modern Linux systems can only run on the same hardware with massive tailorings to reduce the size. The Multicians Website has a pagemulticians:myths clarifying some common misunderstandings about Multics.

[2] For a modern port of Unix v6, *cf.* wiki:xv6.

[3] The 386BSD project also reached this goal, but its first release was in 1992; additionally, a lawsuit and some infighting at that timewiki:386bsd distracted the BSD people, and now it might be fair to say that, from then on, BSD never caught up with Linux in popularity.

by McIlroy was a shell script, which took only one or two minutes and worked correctly on the first run. This script, with minor modifications, is shown below:

```
#!/bin/sh
tr -cs A-Za-z\' '\n' | tr A-Z a-z |
sort | uniq -c |
sort -k1,1nr -k2 | sed "${1:-25}"q
```

Its first line tells Unix this is a program interpreted by `/bin/sh`, and the commands on the rest lines are separated by the **pipe** "`|`" into six steps:

- In the first step, the `tr` command converts all characters, except for (uppercase and lowercase) English letters and "`'`" (the `-c` option means to complement), into the newline character `\n`, and the `-s` option means to squeeze multiple newlines into only one.
- In the second step, the command converts all uppercase letters into corresponding lowercase letters; after this step, the input text is transformed into a form where each line contains a lowercase word.
- In the third step, the `sort` command sorts the lines according to the dictionary order, so that the same words would appear on adjacent output lines.
- In the fourth step, the `uniq` command replaces repeating adjacent lines with only one of them, and with the `-c` option prepends to the line the count of the repetition, which is also the frequency we want.
- In the fifth step, the `sort` command sorts the lines according to the first field (the frequency added in last step) in descending numerical order (the `-k1,1nr` option, where `n` defaults to the ascending order, and `r` reverses the order), and according to the second field (the word itself, with the `-k2` option) in dictionary order in case of identical frequencies.
- In the sixth step, the `sed` command only prints the first lines, with the actual number of lines specified by the first argument of the script on its execution, defaulting to 25 when the argument is empty.

Apart from being easy to write and debug, this script is also very maintainable (and therefore very customisable), because the input requirements and processing rules of the steps are very simple and clear, and we can easily replace the actual implementations of the steps: for instance, the word-splitting criterion used above is obviously primitive, with no consideration of issues like stemming (*eg.* regarding "look", "looking" and "looked" as the same word); if such requirements are to be implemented, we only need to replace the first two steps with some other word-splitting program (which probably needs to be written separately), and somehow make it use the same interface as before for input and output.

Like many Unix tools (*eg.* the ones used above), this script reads input from its **standard input** (defaults to the keyboard), and writes output to its **standard output** (defaults to the current terminal)[4]. Input/output from/to a specified file can be implemented with the **I/O redirection** mechanism in Unix, for example the following command in the shell (assuming the script above has the name `wf.sh` and has been granted execution permission)

```
/path/to/wf.sh 10 < input.txt > output.txt
```

outputs the 10 most frequent words, with their frequencies, from `input.txt` into `output.txt`. It is obvious that the pipe is also a redirection mechanism, which redirects the output of the command on its left-hand side to the input of the command on its right-hand side. From another perspective, if the commands connected by pipes are considered as filters, then each filter performs a relatively simple task; so the programming style in the script above is to decompose a complicated text-processing task into multiple filtering steps linked together with pipes, and then to implement the steps with relatively ready-made tools. Through the example above, we have taken a glimpse at the strong power that can be achieved by the combination of Unix tools; but other systems, like Windows, also have mechanisms similar to I/O redirection, pipes *etc* in Unix, so why don't we often see similar usage in these systems? Please read the next section.

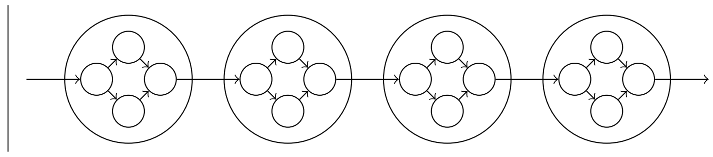
03 Cohesion and coupling in software engineering

Cohesion and coupling are extremely important notions in software engineering, and here we first try to understand what coupling is. Consider the interactions (*eg.* communication through text or binary byte streams, message passing using data packets or other media, calls between subroutines) between

[4] The `stdio.h` in C means standard I/O exactly.

modules in the two extreme cases from the figure in litt2014a. When some faults occur, how difficult will the debugging processes be with the two systems? When the requirements change, how difficult will the maintenance be with the two systems? I think the answer should be self-evident. In these two systems, similarly consisting of 16 modules, the sharp difference in the level of difficulty in debugging and maintenance is determined by the complexity of interactions between modules, and the **degree of coupling** can just be considered as a measure of the complexity of this kind of interactions. We obviously want the modules in a system to be as loosely coupled as reasonable, and the script from last section is easy to debug and maintain exactly because of the low coupling between the underlying commands.

Just like a system needs to be partitioned into modules (*eg.* Unix tools), modules often also need to be partitioned into submodules (*eg.* source files and function libraries), but the submodules are always much more tightly coupled than the tools themselves are, even with a most optimal design (*cf.* the picture below for an example). For this reason, when partitioning a system into modules, it is desirable to concentrate this kind of coupling inside the modules, instead of exposing them between the modules; I consider the **degree of cohesion** to be the measure of this kind of inherent coupling between submodules inside a module, and partitioning of a system according to the principle of high cohesion would naturally reduce the degree of coupling in the system. It may be said that coupling between (sub)modules somehow reflect the correlation between the nature of their tasks, so high cohesion comes with clear **separation of concerns** between modules, because the latter results in closely correlated submodules gathered into a same module. Low coupling is a common feature of traditional Unix tools, which is exactly due to the clear separation of concerns between them: as can be noticed from the script from last section, the tools not only have well-defined input/output interfaces, but also have clear processing rules from the input to the output, or in other words their behaviours are clearly guided; from the perspective of systems and modules, each Unix tool, when considered as a module, usually does a different unit operation, like character translation (`tr`), sorting (`sort`), deduplication / counting (`uniq`) and so on.



We have already seen that high cohesion and low coupling are desirable properties for software systems. You might ask, while there are many Windows programs that are loosely coupled (*eg.* Notepad and Microsoft Paint do not depend upon each other) and have somehow high cohesion (*eg.* Notepad is used for text editing while Microsoft Paint for drawing), why don't we often combine them as with Unix tools? The answer is actually obvious – because they are not designed to be composable; to put it more explicitly, they cannot use some simple yet general-purpose interface, like pipes, to collaborate, and therefore cannot be easily reused in automated tasks. In comparison, the power of Unix seen in last section comes exactly from its emphasis on reusability of user-accessible tools in automation, which results in the almost extreme realisation of the principle of high cohesion and low coupling in traditional Unix toolssalus1994. In conclusion, the requirements of cohesion and coupling must be considered with the background of **collaboration and reuse**, and the pursuit of collaboration and reuse naturally promotes designs with high cohesion and low coupling.

Until now, our examples have been relatively idealised or simplified, and here I give two more examples that are more realistic and relevant to hot topics in recent years. When Unix systems are started, the kernel will create a first process which in turn creates some other processes, and these processes manage the system services together; because of the important role of the first process in system initialisation, it is usually called "**init**"jdebp2015. systemd is the most popular init system as of now, and its init has very complex functionalities, while its mechanisms are poorly documented; furthermore, aside from the init program called `systemd` as well as related ancillary programs, systemd also has many other non-init modules, and the interactions between all these modules are complex and lack documentation (a fairly exaggerated depiction can be found at litt2014b). systemd obviously has low cohesion and high coupling, but this is unnecessary because the daemontools-ish design (represented in this document with s6) is much simpler than systemd, yet not weaker than systemd in functionalities.

As is shown in the picture below, the init program of s6, `s6-svscan`, scans for subdirectories in a specified directory ("scan directory", like `/service`), and for each subdirectory ("service directory", like `/service/kmsg`) runs a `s6-supervise` process, which in turn runs the executable called `run` (like `/service/kmsg/run`) in the service directory to run the corresponding system service. The user can use s6's command line tools `s6-svc`/`s6-svscanctl` to interact with `s6-supervise`/`s6-svscan`, and can use ancillary files in service directories and the scan directory to modify the behaviours of `s6-supervise` and `s6-svscan`[5]. Only longrun services are managed by s6, while oneshot init scripts are managed by s6-rc, which also uses s6's tools to track the startup status of services in order to manage the dependency between them. It was mentioned above that this design is not weaker than systemd in functionalities, and here I give one example (*cf.* Section **??** for some in-depth examples): systemd supports service templates, for instance after a template called `getty@` is defined, the `getty@tty1` service would run the getty program on `tty1`; in s6/s6-rc, a similar functionality can be achieved by loading a 5-line library scriptgitea:srvrc in the `run` script.

```
s6-svscan                  Scans /service, controllable with s6-svscanctl
 ├ s6-supervise kmsg       Configured in /service/kmsg, controllable with s6-svc
 │  └ ucspilogd            exec()ed by /service/kmsg/run (cf. Section ??)
 ├ s6-supervise syslog     Configured in /service/syslog, controllable with s6-svc
 │  └ busybox syslogd      exec()ed by /service/syslog/run (cf. Section ??)
 ...
```

04 Do one thing and do it well

The Unix-ish design principle is often called the "**Unix philosophy**", and the most popular formulation of the latter is undoubtedly by Doug McIlroysalus1994:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

In connection with the discussion in last section, it is easy to notice the first point by McIlroy is an emphasis on high cohesion and low coupling[6], the second point is an emphasis on collaboration and reuse of programs, and the third point feels slightly unfamiliar: we did see the power from combination of Unix tools in text processing in Section **??**, but what is the underlying reason for calling text streams a universal interface? I believe that this can be explained by the friendliness of **human-computer interfaces** toward humans and computers (this will be involved again in Section **??**), where text streams are a compromise between binary data and graphical interfaces. Binary data are easily processed by computers, but are difficult to understand for humans, and the subtle differences between the ways binary data are processed by different processors also give rise to portability issues of encodings, with the endianness problem being a representative issue. Graphical interfaces are the most friendly to humans, but are much more difficult to write in comparison with textual interfaces, and do not compose easily even as of now[7]. Text streams are both easy for computers to process and fairly easy for humans to understand, and while it also involves issues with character encoding, the latter kind of issues are generally much simpler than similar issues with binary information.

McIlroy's formulation is not undisputed, and the most relevant disagreement is on whether text streams as a communication format are the best choice, which we will also discuss in Section **??**; in addition, while this formulation almost covers the factors that we have hitherto seen to make Unix powerful, I do not think it represents the Unix philosophy completely. It is worth noting that the appearance of pipes directly resulted in the pursuit of collaboration and reuse of command line programssalus1994; since McIlroy is the inventor of Unix pipes, his summary was probably based on shell scripting. Shell scripting is admittedly important, but it is far from the entirety of Unix: in the two following sections, we will see some relevant cases outside shell scripting that reflect the philosophy, and that cannot be covered by the classic formulation by McIlroy; and then in Section **??**, I will present what I regard as the essence of the Unix philosophy.

[5] This configuration method may seem unintuitive, but its rationale and benefits will be explained in Section **??** and Footnote **??**.

[6] By the way, this also shows that the principle of high cohesion and low coupling is not unique to objected-oriented programming. In fact, some people believechenhao2013 that every design pattern in OOP has some counterparts in Unix (*cf.* Section **??** for one of them).

[7] It is worth mentioning that I do not reject graphical interfaces, but just think that they are mainly necessary when the corresponding requirements are clumsy to implement with textual interfaces, and that the requirements of automation need consideration in their design; and speaking of the latter issue, as far as I know the automation of graphical interfaces is still a non-trivial subject. I currently find the AutoCAD design an interesting approach, where there is a command line along with the graphical interface, and operations on the graphical interface are automatically translated into commands and shown on the command line.

05 `fork()` and `exec()`

Processes are one of the most important notions in operating systems, so OS interfaces for process management are quite relevant; as each process has a set of state attributes, like the current working directory, handles to open files (called **file descriptors** in Unix, like the standard input and output used in Section **??**, and the **standard error output** which will be involved in Section **??**) *etc*, how do we create processes in specified states? In Windows, the creation of processes is implemented by the `CreateProcess()` family of functions, which usually needs about 10 arguments, some of which are structures representing multiple kinds of information, so we need to pass complicated state information when creating processes; and noticing that we need system interfaces to modify process states anyway, the code that does these modifications are nearly duplicated in `CreateProcess()`. In Unix, processes are created using the `fork()` function, which initiates a new child process with state attributes identical to the current process, and the child process can replace itself with other programs by calling the `exec()` family of functions; before `exec()`ing, the child process can modify its own state attributes, which are preserved during `exec()`. It is obvious that `fork()`/`exec()` requires very little information, and that Unix realised the decoupling between process creation and process state control through this mechanism; and considering that when creating processes in real applications, the child process often need to inherit most attributes from its parent, `fork()`/`exec()` actually also simplifies the user code greatly.

If you know some object-oriented programming, you should be easily able to notice that the `fork()`/`exec()` mechanism is exactly a realisation of the Prototype Pattern, and the same line of thought can also inspire us to think about the way processes are created for system services. With systemd, service processes are created by its init program `systemd`, which reads configuration file(s) for each service, runs the corresponding service program, and sets the process attributes for the service process according to the configuration. Under this design, all the code for process creation and process state control needs to be in init, or in other words what systemd does is like, in a conceptual sense, implementing service process creation in the style of `CreateProcess()` while using `fork()`/`exec()`. Borrowing the previous line of thought, we can completely decouple process state control from the init module: for example, with s6, `s6-supervise` almost does not modify any process attribute before `exec()`ing into the `run` program; the `run` program is almost always a script (an example is given below; *cf.* pollard2014 for some more examples), which sets its own attributes before `exec()`ing into the actual service program. The technique of implementing process state control with consecutive `exec()`s is expressively called **Bernstein chainloading**, because of Daniel J. Bernstein's extensive use of this technique in his software qmail (first released in 1996) and daemontools (first released in 2000). Laurent Bercot, the author of s6, pushed this technique further and implemented the unit operations in chainloading as a set of discrete toolsska:execline, with which we can implement some very interesting requirements (*cf.* Footnote **??** for one such example).

```
#!/bin/sh -e                       -e means to exit execution case of error
exec 2>&1                          Redirect the standard error to the standard output
cd /home/www                       Change the working directory to /home/www
exec \                             exec() the long command below; “\” joins lines
s6-softlimit -m 50000000 \         Limit the memory usage to 50M, then exec()
s6-setuidgid www \                 Use the user & group IDs of www, then exec()
emptyenv \                         Clear all environment variables, then exec()
busybox httpd -vv -f -p 8000       Finally exec() into a web server on port 8000
```

When creating service processes, chainloading is of course much more flexible than systemd's mechanism, because the modules of the former possess the excellent properties of high cohesion and low coupling, and are therefore easy to debug and maintain. In comparison, the mechanism in systemd is tightly coupled to other modules from the same version of systemd, so we cannot easily replace the malfunctioning modules when problems (*eg.* edge2017) arise. Because of the simple, clear interface of chainloading, when new process attributes emerge, we can easily implement the corresponding chainloader, and then integrate it into the system without upgrading: for instance, systemd's support for Linux's cgroups is often touted by systemd developers as one of its major selling pointspoettering2013, but the user interface is just operations on the `/sys/fs/cgroup` directory tree, which are easy to do in chainloading; now we already have some ready-made chainloaders availablepollard2019, so it can be said that the daemontools-ish design has natural advantages on support for cgroups. Additionally, the composability of chainloaders allow us to implement some operations that are hard to describe just using systemd's mechanisms: for example we can first set some environment variables to modify the behaviour of a later chainloader, and then clear the environment before finally `exec()`ing into the service program; *cf.* ska:syslogd for a more advanced example.

It is necessary to note that primitive forms of `fork()`/`exec()` appeared in operating systems earlier than Unixritchie1980, and Ken Thompson, Dennis Ritchie *et al* chose to implement process creation with this mechanism out of a pursuit of simplicity of the implementation, so the mechanism was not exactly an original idea in Unix; nevertheless we have also seen that based on `fork()`/`exec()` and its line of thought we can implement many complicated tasks in simple, clear ways, so the mechanism does satisfy the Unix design concepts in Section **??** in an intuitional sense. Now back to the subject of Unix philosophy: `fork()`/`exec()` conforms to the principle of high cohesion and low coupling, and facilitates collaboration and reuse of related interfaces, so we can regard it as roughly compliant to the first two points from Doug McIlroy's summary in last section despite the fact that it is not directly reflected in shell scripting; however, this mechanism does not involve the choice of textual interfaces, so it is not quite related to McIlroy's last point, and I find this a sign that `fork()`/`exec()` cannot be satisfactorily covered by McIlroy's formulation.

06 From Unix to Plan 9

Unix v7mcilroy1987 already had most notions seen currently (*eg.* files, pipes and environment variables) and many **system calls** (how the userspace requests services from the kernel, like `read()`/`write()` which perform read/write operations on files, and `fork()`/`exec()` mentioned in last section) that are still widely used now. To manipulate the special attributes of various hardware devices and avoid a crazy growth in number of system calls with the development of device support, this Unix version introduced the `ioctl()` system call, which is a "jack of all trades" that manipulates various device attributes according to its arguments, for instance

```
ioctl (fd, TIOCGWINSZ, &winSize);
```

saves the window size of the serial terminal corresponding to the file descriptor `fd` into the structure `winSize`. In Unixes up to this version (and even versions in few years to come), although there were different kinds of system resources like files, pipes, hardware devices *etc* to operate on, these operations were generally implemented through file interfaces (*eg.* read/write operations on `/dev/tty` are interpreted by the kernel as operations on the terminal), or in other words "everything is a file"; of course, as was mentioned just now, in order to manipulate the special attributes of hardware, an exception `ioctl()` was made. In comparison with today's Unix-like operating systems, Unixes of that age had two fundamental differences: they had no networking support, and no graphical interfaces; unfortunately, the addition of these two functionalities drove Unix increasingly further from the "everything is a file" design.

Berkeley socketswiki:sockets appeared in 1983 as the user interface for the TCP/IP networking protocal stack in 4.2BSD, and became the most mainstream interface to the internet in 1989 when the corresponding code was put into the public domain by its copyright holder. Coming with sockets was a series of new system calls like `send()`, `recv()`, `accept()` *et al.* Sockets have forms similar to files, but they expose too many protocol details, which make their operations much more complicated than those of files, and a typical example for this can be seen at pike2001; moreover, duplication began to appear between system calls, for example `send()` is similar to `write()`, and `getsockopt()`/`setsockopt()` are similar to the already ugly `ioctl()`. After that, the system calls began to grow constantly: for instance, Linux currently has more than 400 system callskernel:syscalls, while Unix v7 had only about 50wiki:unixv7; one direct consequence of this growth is the complication of the system interfaces as a whole, and the weakening of their uniformity, which led to an increase in difficulty of learning. The X Window Systemwiki:xwindow (usually called "X" or "X11" now), born in 1984, has problems similar to those of Berkeley sockets, and the problems are more severe: sockets are at least formally similar to files, while windows and other resources in X are not files at all; furthermore, although X did not introduce new system calls as with sockets, its number of basic operations, which can be roughly compared to system calls, is much larger than the number of system calls related to sockets, even when we only count the core modules of X without any extension.

After the analysis above, we would naturally wonder, how can we implement the support for networking and graphical interfaces in Unix, in a way that follows its design principles? Plan 9 from Bell Labs (often simply called "Plan 9") is, to a large extent, the product of the exploration on this subject by Unix pioneersraymond2003b. As was mentioned before, system calls like `ioctl()` and `setsockopt()` were born to handle operations on special attrbutes of system resources, which do not easily map to operations on the file system; but on a different perspective, operations on resource attributes are also done through communication between the userspace and the kernel, and the only difference is that the information passed in the communication is special data which represent operations on resource attributes. Following this idea, Plan 9 extensively employs **virtual file systems** to represent various system resources (*eg.* the network is represented by `/net`), in compliance with the "everything is a file" designpike1995; control files (*eg.* `/net/tcp/0/ctl`) corresponding to various resource files (*eg.* `/net/tcp/0/data`) implement operations on resource attributes, different **file servers** map file operations to various resource operations, and the traditional **mounting** operation binds directory trees to file servers. Because file servers use the network-transparent **9P protocol** to communicate, Plan 9 is naturally a distributed OS; in order to implement the relative independence between processes and between machines, each process in Plan 9 has its own **namespace** (*eg.* a pair of parent and child processes can see mutually independent `/env`, this way the independence of environment variables is implemented), so normal users can also perform mounting.

With the mechanisms mentioned above, we can perform many complicated tasks in Plan 9 in extremely easy ways using its roughly 50 system callsaiju:9syscalls: for instance, remote debugging can be done by mounting `/proc` from the remote system, and the requirements of VPN can be implemented by mounting a remote `/net`; another example is that the modification of users' networking permissions can be performed by setting permissions on `/net`, and that the restriction of user access to one's graphical interface can be done by setting permissions on `/dev/mouse`, `/dev/window` *etc.* Again back to the topic of Unix philosophy: on an intuitional level, the design of Plan 9 is indeed compliant to the philosophy; but even if `fork()`/`exec()`, analysed in last section, could be said to be roughly compliant to Doug McIlroy's formulation on the philosophy in Section **??**, I am afraid that the "everything is a file" design principle can hardly be covered by the same formulation; so the formulation was not very complete, and we need a better one.

07 Unix philosophy: minimising the system's complexity

In the two previous sections, we have already seen that Doug McIlroy's summary cannot satisfactorily cover the entirety of Unix philosophy; this summary (and especially its first point) can indeed be regarded as the most mainstream one, but there are also many summaries other than the one by McIlroywiki:unixphilo, for instance:

- Brian Kernighan and Rob Pike emphasised the design of software systems as multiple easily composable tools, each of which does one kind of simple tasks in relative separation, and they in combination can do complex tasks.
- Mike Gancarz[8] summarised the Unix philosophy as 9 rules.
- Eric S. Raymond gave 17 rules in *The Art of Unix Programming.*

I believe that the various formulations of the philosophy are all of some value as references, but they themselves also need to be summarised: just like how Plan 9, as was mentioned in last section, implemented requirements which need several hundreds of system calls elsewhere, with only about 50 system calls by using virtual file systems, the 9P protocol and namespaces. In Sections **??** & **??**–**??**, our intuitive criteria for a system's conformance to the Unix philosophy were all that the system used a small number of simple mechanisms and tools to implement requirements that were more difficult to implement in other ways, or in other words they reduced the complexity of the system. Based on this observation, I believe the essence of the Unix philosophy is **the minimisation of the cognitive complexity of the system while almost satisfying the requirements**[9], and the three restrictions in my formulation are explained below:

- The prerequisite is to **almost** satisfy the requirements, because said requirements can often be classified into essentials (*eg.* support for networking and graphical interfaces) and extras (*eg.* support for Berkeley sockets and the X Window System), and some extras can be discarded or implemented in better ways.
- We need to consider the total complexity of the **system**, because there are interactions between the modules, and only examining some of the modules would result in omission of the effects on

[8] He is, curiously, one of the designers of the X Window System (*cf.* Section **??**).

[9] Some people noted that while this formulation can be used to compare existing software systems and practical plans for such systems, it does not directly tell us how to design and implement minimal software systems. However, the formulation does cover existing summaries for the Unix philosophy, which are more practical; it also indirectly leads to the practical ways to foster minimalist habits (and creativity) in Sections **??**–**??**. I consider the relation between this formulation and practical ways to minimal software systems to be similar to the relation between the principle of least action and the calculus of variations, as well as to the relation between the Ockham's razor and the theories about the minimal message length and the minimal description length (*cf.* Section **??**).

their behaviours by their dependencies: for example, if a requirement could be implemented in both ways in the figure from litt2014a, and both implementations use the same user interface, we surely cannot say they would be equally compliant to the Unix philosophy just because of the identical interfaces.

- It is stipulated that we are discussing about the **cognitive** complexity, because as was shown in the comparison above (a more realistic comparison can be found at github:acmetiny/gitea:emca), the quality of a system is not only determined by its code size; we also have to consider the cohesion and coupling of its modules, and the latter are essentially attributes oriented toward humans, not computers, which I will further discuss in Section **??**.

```
init                           init                              init
├ ···               ──────→    ├ ···                 ──────→     ├ ···
│ └ sh (1)                     │ └ sh                            │ └ sh (1)
│   └ srv --fork (2)           │   └ srv --fork (2)              ├ srv --fork (3)
···                            ···   └ srv --fork (3)            ···
```

(1) The user's shell, which waits for another command after the child process `srv --fork` exits.
(2) Service program run by the user, which exits after `fork()`ing a child.
(3) Child process `fork()`ed by the service program, reparented to `init` after the former's parent exited.

Let's see a fairly recent example. In some init systems (represented by sysvinit), longrun system services detach themselves from the control of the user's shell through `fork()`ing, in order to implement backgroundinggollatoka2011 (as is shown in the example above): when the service program is run by the user from the shell, it `fork()`s a child process, and then the user-run (parent) process exits; now the shell waits for another command from the user because the parent process has exited, while the child process is reparented to init upon exiting of the parent process, and no longer under the user shell's control. However, in order to control the state of a process, its process ID should be known, but the above-mentioned child has no better way to pass its PID other than saving it into a "PID file", which is an ugly mechanism: if the service process crashes, the PID file would become invalid, and the init system cannot get a real-time notification[10]; moreover, the original PID could be occupied by a later process, in which case the init system might mistake a wrong process for the service process.

With s6 (*cf.* Section **??**; other daemontools-ish systems and systemd behave similarly), the service process is a child of `s6-supervise`, and when it exits the kernel will instantly notify `s6-supervise` about this; the user can use s6's tools to tell `s6-supervise` to change the state of the service, so the service process is completely independent of the user's shell, and no longer needs to background by `fork()`ing. This mechanism in s6 is called **process supervision**, and from the analysis above we see that using this mechanism the init system can track service states in real time, without worrying about the pack of problems with PID files. In addition, since the service process is only restarted by its parent (*eg.* `s6-supervise`) after exiting with supervision, in comparison with the sysvinit mechanism where the service process is created by a close relative of init on system startup but created by a user's shell when restarting, the former mechanism has much better reproducibility of service environments. An apparent problem with supervision is that services cannot notify the init system about its readiness by exiting of the init script as with sysvinit, and instead has to use some other mechanism; the readiness notification mechanism of s6ska:notify is very simple, and can emulate systemd's mechanism using some toolska:sdnwrap.

A bigger advantage of process supervision is the management of system logs. With the sysvinit mechanism, in order to detach itself from the user shell's control, the service process has to redirect its file descriptors, which were inherited from the shell through `exec()` and previously pointed to the user's terminal, to somewhere else (usually `/dev/null`), so its logs have to be saved by alternative means when not directly written to the disk. This is the origin of the syslog mechanism, which lets service processes output logs to `/dev/log` which is listened on by a system logger; so all system logs would be mixed up before being classified and filtered[11], and the latter operations can become a performance bottleneck when the log volume is huge, due to the string matching procedures involved. With supervision, we can assign one logger process for each service processska:s6log[12], and redirect the standard error output of

[10] In fact, init would be notified upon death of its children, but using this to monitor `fork()`ing service programs creates more complexity, and does not solve the problem cleanly (*eg.* what if the program crashes before writing the PID file?); all other attempts to "fix" PID files are riddled with similar issues, which do not exist when using process supervision.

[11] As a matter of fact, because `/dev/log` is a socket (to be precise, it needs to be a `SOCK_STREAM` socketbercot2015d), in principle the logger can do limited identification of the log source and then somehow group the log streams, and this is not hard to implement with tools by Laurent Bercotska:syslogd, vector2019b.

[12] systemd unfortunately does not do this, and instead mixes up all logs before processing. Incidentally, here the different

the latter to the standard input of the former by chainloading, so service processes only need to write to the standard error in order to transfer logs; because each logger only needs to do classification and filtering of logs from the corresponding service (instead of the whole system), the resource consumption of these operations can be minimised. Furthermore, using the technique of **fd holding**ska:fdhold (which, incidentally, can be used to implement so-called "socket activation"), we can construct highly fault-tolerant logging channels that ensure the log is not lost when either the service or the logger crashes and restarts.

From the analyses above, we can see that process supervision can greatly simplify the management of system services and their logs, and one typical example for this is the enormous simplification of the management of MySQL servicespollard2017. Because this mechanism can, in a simple and clear way (minimising the cognitive complexity of the system), implement the requirements for managing system services and their logs (and cleanly implement new requirements that are troublesome to do with the old mechanism), I find it highly conformant to the Unix philosophy.

08 Unix philosophy and software quality

It was mentioned in Section **??** that the resource limitations when Unix was born resulted in its pursuit of economy and elegance, and nowadays many people consider the Unix philosophy outdated exactly because of this reason. I think this issue can be analysed from the viewpoint of software quality, or in other words whether the conformance to the philosophy is correlated with the quality of software systems. There are many definitions for software quality, and one of themwiki:quality divides it into five aspects: **reliability**, **maintainability**, **security**, **efficiency** and **size**; it is easy to notice that the last two aspects are mainly oriented toward machines, and the first three mainly toward humans. Since the limits on available hardware resources was the main cause for the origination of the Unix philosophy, let's first examine the two mainly machine-oriented aspects: with today's hardware resources multiple orders of magnitude richer than the era when Unix was born, speaking of perceived software efficiency and size, is the philosophy no longer so relevant? I am inclined to give a negative conclusion, and I use the most common requirement for most users, web browsing, as an example.

With the constant upgrade of hardware, it appears that our browsing experience should become increasingly smooth, but this is often not what we usually feel in fact: although the speed of file downloading and the resolution of videos we watch grow continuously, the loading speed of pages we experience on many websites does not seem to grow at the same pace; this observation might be a little subjective, but frameworks like Google's "Accelerated Mobile Pages" and Facebook's "Instant Articles" can perhaps attest to the existence of the phenomenon. Moreover, the memory consumption problem of web browsers has still not disappeared over time, which to some extent shows that aside from the efficiency issue, the size issue is not satisfactorily solved over time either. This is a general problem in the realm of software, and one classic summary iswiki:wirth:

> Software efficiency halves every 18 months, compensating Moore's law.

It is my opinion that if we were to be satisfied with writing "just usable" software in terms of efficiency and size, it would perhaps be unnecessary to consider the Unix philosophy; and that if we however want to write software whose efficiency and size do not deteriorate with the release of new versions, the philosophy will still be of its value.

Now we consider the three aspects that are largely human-oriented, and since security is to be specifically discussed in the next section, here we mainly focus on reliability and maintainability. It cannot be denied that today's programmer resources and programming tools are almost imaginable at the inception of Unix, which is why mainstream Unix-like systems in this era can be much more complicated than Multics (*cf.* Footnote **??**). On the other hand, I believe that the improvements in these areas are far from able to counter the rule summarised by Tony Hoare in his Turing Award lecturehoare1981 (and many computer scientists think similarly):

> Almost anything in software can be implemented, sold, and even used, given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way, and that is reliability. **The price of reliability is the pursuit of the utmost simplicity.**

loggers can be run as different low-privilege users, therefore implementing a high level of privilege separation. In addition, by a moderate amount of modification to the logger program, a feature that prevents log tamperingmarson2013 can be implemented, and the latter is often boasted by systemd proponents as one of its exclusive features.

It is a price which the very rich find most hard to pay.

Hoare focused on reliability, but I believe maintainability is, to a large extent, also governed by the rule, and one example for the correspondence between complexity and maintainability (which I regard development cost as a part of) can be found at rbrander2017. In the following I will demonstrate the relation between complexity and reliability / maintainability, using s6 and systemd as examples.

As was noted in Section **??**, init is the first process after a Unix system is started, and in fact it is also the root node of the whole process tree, whose crash (exiting) would result in a kernel panic[13], so it must be extremely reliable; and since init has root permissions, it also has to be highly secure. Again as was mentioned before, contrary to the design of s6, the init module of systemd is overly complex, and has too much, too complex interaction with other modules, which makes the behaviours of systemd's init difficult to control in a satisfactory manner, for instance ayer2016, edge2017 are examples for actual problems this has led to. Similarly, the systemd architecture, which has low cohesion and high coupling, makes other modules difficult to debug and maintain just like the init module: the number of unresolved bugs with systemd grows incessantly over time, still without any sign of leveling off (let alone sustainably decreasing)waw:systemd. In comparison, with s6/s6-rc and related packages, the fix for any bug (there have been very few) almost always arrives within one week of the bug report, and even if we also count other projects with functionalities emulated by systemd, the number of bugs still does not grow in systemd's fashion[14].

From the perspective of the user, systemd's behaviours are too complex, and consequently its documentation can only describe most typical application scenarios, and many use cases unconsidered by its developers become actual "corner cases" (*eg.* dbiii2016; for a very detailed analysis on the origin of this kind of problems, *cf.* vr2015) where the behaviours of systemd are very difficult to deduce from the documentation. Sometimes, even if a requirement happens to be successfully implemented with systemd, the corresponding configuration lacks reproducibility because there are too many factors affecting systemd's behaviours (*eg.* fitzcarraldo2018/zlogic2019). On the contrary, a user familiar with shell scriping and basic notions about processes can read through the core s6/s6-rc documentation comfortably in 2–3 hours, and then will be able to implement the desired configuration with s6/s6-rc; a vast majority of problems that can arise will be easily traceable to the causes, and problems with s6/s6-rc themselves are very raregitea:slewman. Furthermore, the behaviours of systemd change too fasthyperion.2019, which further complicates problems considering that these behaviours are already very complex; in comparison, clear notices are given when those rare backward-incompatible changes occur in s6/s6-rc and related packages, which in combination with the well-defined behaviours of related tools minimises the unpredictability of upgrading.

systemd has nearly two orders of magnitude more developers than s6, and uses fairly advanced development methods like coverage tests and fuzzing, but even so its quality is far from that of s6, which adequately demonstrates that the increase in human resources and the progress in programming tools are far from substitutes for the pursuit of simplicity in software. Even if it could be said that the Unix philosophy is not as relevant as before in terms of software efficiency and size, from the analysis above I believe we can conclude that in terms of reliability and maintainability, **the Unix philosophy has never become outdated, and is more relevant than when it was born**: the disregard of simplicity due to the disappearance of resource limitations contributed to the spread of low-quality software, and systemd is just its extreme manifestation in the realm of system programmingska:systemd; programmers used to be forced into pursuing simplicity by resource limitations, and now we largely need to adhere to the philosophy by self-discipline, which is harder[15].

[13] But init can `exec()`, which enables mechanisms like `switch_root`; in addition, it is exactly using `exec()` by init that s6 realised the decoupling between the main submodules of the init system and code related to the beginning phase of booting and the final phase of shutdownska:pid1.

[14] We can also compare systemd with the Linux kernel, which is of a huge size and is developed rapidly: by periodically pausing the addition of new features (**feature freeze**) and concentrating on fixing bugs discovered in the current period (**bug converge**), the latter effectively controlled its bug growth; systemd developers do not do the same thing, and nor do they control its bug growth by other means of project management, which shows that they lack proper planning in software development (of course, this may be because they do not even feel they were able to effectively fix bugs without creating new problems).

[15] Similar phenomena are not unique to programming, for example the appearance of software like Microsoft Publisher in the 1990s enabled the ordinary person to do basic typesettingkadavy2019, which however also contributed to people's negligence of basic typographical principlessuiseiseki2011.

09 Unix philosophy and software security

After the disclosure of the PRISM project in the US by Edward Snowden, information security became a topic that attracts continued attention, so an entire section in this document is dedicated to the relation between the Unix philosophy and software security. Assuming very few software bugs are injected by malicious actors, then security vulnerabilities, like other defects, are usually introduced inadvertently by developers. Due to this I think it may be assumed that the cognitive complexity of a software system determines its number of defects, because programming is after all a task similar to other kinds of mental labour, and the same kind of products by the same person that consumed the same amount of energy should naturally contain similar numbers of defects. Defects (also including security vulnerabilities) in software systems come with code changes, and go with analyses and debugging, while the degree of difficulty in analyses and debugging obviously depends on the size of the codebase as well as the degree of cohesion and coupling, or in other words the cognitive complexity of the software system. Therefore we can see that **complexity is a crucial factor that governs the creation and annihilation of defects, including security vulnerabilities, in software** (which probably also explains why the number of unresolved bugs in systemd increases constantly), so the Unix philosophy, which pursues simplicity, is extremely important to software security.

The root cause of many software defects is the fundamental weaknesses in the design of these software systems, and the counterpart to these weaknesses in information security is, to a large extent, weaknesses in cryptographic protocols; accordingly, I give two examples for the purely theoretical analysis above, one about cryptographic protocols and the other about implementation of such protocols. Due to the strongly mathematical nature of cryptographic protocols, they can be mathematically analysed, and it is generally thought in the field of information security that cryptographic protocols without enough theoretical analyses lack practical valueschneier2015. Nevertheless, even with this background, some widely used cryptographic protocols are so complicated that they are very hard to analyse, and one typical example is the IP security protocols represented by IPsec[16]. After analysing IPsec, Niels Ferguson and Bruce Schneier remarkedferguson2003 that

> On the one hand, IPsec is far better than any IP security protocol that has come before: Microsoft PPTP, L2TP, *etc.* On the other hand, we do not believe that it will ever result in a secure operational system. It is far too complex, and the complexity has lead to a large number of ambiguities, contradictions, inefficiencies, and weaknesses. It has been very hard work to perform any kind of security analysis; we do not feel that we fully understand the system, let alone have fully analyzed it.

And they gave the following rule:

> **Security's worst enemy is complexity.**

Similarly, when discussing how to prevent security vulnerabilities similar to the notorious Heartbleed (which originated from the homemade memory allocator in OpenSSL concealing the buffer overread issue in the codebase) from occuring again, David A. Wheeler remarkedwheeler2014 that

> I think **the most important approach for developing secure software is to simplify the code so it is obviously correct**, including avoiding common weaknesses, and then limit privileges to reduce potential damage.

With the rapid development on the Internet of Things, the number of Internet-connected devices is growing steadily, and the 2020s may become the decade of IOT, which creates at least two problems. First, security vulnerabilities on these ubiquitous devices would not only give rise to unprecedentedly large botnets, but also possibly result in very realistic damages to the security of the physical world due to the actual purposes of these devices, and for this very reason security is a first subject the IoT must tackle. Second, these connected devices often have highly limited hardware resources, so the efficiency and size of software will inevitably become relevant factors in IoT development. As such, I believe that **the Unix philosophy will still manifest its strong relevance in the 2020s**.

[16] I find the recent cjdns (and later derivatives like Yggdrasil) to be, protocol-wise, a scheme with perhaps great potentials, because it is a compulsorily end-to-end encrypted (preventing surveillance and manipulation, and simplifying upper-level protocols) mesh network (simplifying routing, and rendering NAT unnecessary) which uses IPv6 addresses generated from public keys as network identifiers (eliminating IP address spoofing) and is quite simple. And I need to clarify that I detest the current implementation of cjdns, which seems too bloated from the build system to the actual codebase; I even suspect that if an almost identical protocol was implemented by Laurent Bercot, the codebase might have been less than 1/10 of its current size.

> Sysadmin: `login` appears backdoored; I'll recompile it from clean source.
> Compiler: `login` code detected; insert the backdoor.
> Sysadmin: the compiler appears backdoored, too; I'll recompile it from clean source.
> Compiler: compiler code detected; add code that can insert the `login` backdoor.
> Sysadmin: (now what?)

Before concluding this section, I would like to digress a little and examine the issue of compiler backdoors, which makes the compiler automatically injects malicious code when processing certain programs (as is shown in the example above): when suspecting the compiler after noticing abnormalities, people would certainly think of generating the compiler itself from a clean copy of its source code; however, if the source code is processed by the dirty compiler (*eg.* most C compilers are written in C, so they can compile themselves, or in other words **self-bootstrap**[17]), which automatically inserts the above-mentioned injector, what should we do? This extremely covert kind of backdoors are called **Trusting Trust** backdoors, which became well-known through the Turing Award lecturethompson1984 by Ken Thompson[18], and got its name from the title of the lecture. A universal method against Trusting Trust is "Diverse Double-Compiling"wheeler2009, which compiles the clean source code of the suspected compiler with another compiler, and compares the product from the latter with the product from self-compilation of the former to determine whether there is a Trusting Trust backdoor. One alternative method avoids self-bootstrapping, and instead constructs the compiler step by step from the low-level machine codenieuwenhuizen2018, and I will discuss this method in detail in Section **??**–**??**.

10 Unix philosophy and free / open-source software

"Free software"wiki:free and "open-source software"wiki:oss are notions that are very similar in extensions but clearly different in intensions: the former stresses the **freedom to (run,) study, redistribute and improve software**, while the latter stresses the **convenience to use, modify and redistribute source code of software**. In this document, I do not intend to further analyse the similarities and differences between them, and instead will develop our discussion based on one aspect of their common requirements on software. It is evident that both demands that the user has the rights to study and improve the source code, in order to modify the software's behaviours, to a reasonable extent, to fulfill his/her requirements. Correspondingly, the key point I would like to express in this section is that the grant of these rights does not imply the user has sufficient control over the behaviours of the software, which in extreme conditions allows the existence of **software projects that are formally free / open-source but actually close to proprietary / closed-source**. Of course not every user is able to study and improve the source code, so all comparisons of software projects involved in this section are done from the viewpoint of a same user with an appropriate background in computer science and software engineering.

We know software that only has obfuscated source code released is ineligible to be called free / open-source software, because obfuscation makes the source code difficult to understand, or in other words increases the cognitive complexity of the source code. On the other hand, from the analyses before, we know that software systems with low cohesion and high coupling also have high cognitive complexities, and some FOSS projects are also affected by this, like Chromium and Firefox, the currently mainstream open-source browsers, and systemd, which has been mentioned multiple times. It can be noticed that the user's control over the behaviours of these software systems has been significantly undermined: for Chromium and Firefox, the typical sign of this problem is that when there are updates that defy user requirements (*eg.* beauhd2019, namelessvoice2018), the user has few choices other than pleading to the development teams[19]; for systemd, the typical sign is that when various "corner cases" (*eg.* ratagupt2017) which should not have even existed in the first place (*cf.* Section **??**) are encountered, the user has to work around the problems by all means before the developers are able to somehow fix them, and the developers may even

[17] "Booting" in system startup is exactly short for bootstrapping, and here the compiler bootstraps itself.
[18] Who is, incidentally, a chess enthusiast; do you notice the pattern of "predicting the enemy's move"?
[19] There are Firefox replacements like Waterfox and Pale Moon, but these replacements lag behind Firefox in terms of security updates *etc*, due to limitations in human resourceshoffman2018.

plainly refuse to consider the requests (*eg.* akcaagac2013/junta2017[20] and freedesktop:sepusr[21]). In fact, the user's control over these software is not even comparable with some source-available software with restrictions on redistribution, like old versions of Chez Scheme which will be mentioned in Section **??**–**??**. From the above, we can see that from software that offers strong control (like s6) and software that allows sufficient control (like old Chez Scheme), to software that only offers highly limited control (like systemd) and traditional proprietary / closed-source software, **in terms of the user's control over software behaviours, the boundary between FOSS and PCSS is already blurring**.

The analysis above is from a purely technical perspective, and the weakening of user control over FOSS is indeed mainly because of the architectures of these software systems, which have low cohesion and high coupling, but I think there is one important exception: in the following, through comparison to proprietary software, I will argue that **systemd is the first open-source project to promote its software using proprietary practices**. Both proponents and opponents of systemd generally agree that the most important milestone during the course where systemd became the default init system in mainstream Linux distributions was when it became the default in the "jessie" release of Debiansfcrazy2014, paski2014[22], and agree that the latter was because GNOME 3 began to depend on interfaces provided by `logind` from systemdbugaev2016. However, although the only dependency was nominally the `logind` interfacesvitters2013, systemd developers soon made it explicit that `logind` was intended to be tied to systemd in the first placepoettering2013, which led to the *de facto* dependency of GNOME 3 on systemd; on the other hand, I have never seen any credible analysis of advantages of systemd `logind` over elogind (appeared in 2015), so systemd developers tied `logind` to systemd deliberately without any known technical advantage.

After this, systemd developers attempted to tie udev to systemdpoettering2012, and then attempted to increase the development cost of the eudev project when independently implementing udev-compatible interfacespoettering2014 by pushing the merger of kdbus into the Linux kernel[23]. Considering the obvious renege of previous promises by systemd developerspoettering2011a, sievers2012, and their push of kdbus with total disregard of projects like eudev and mdev when they already knew kdbus lacked technical advantagescox2012, I find it completely fair to establish that systemd developers deliberately practised the typical "**embrace, extend and extinguish**"wiki:eee scheme, which resulted in unnecessary **vendor lock-in**:

- Developing facilities in their project that can be used by downstream projects, which might extend currently available facilities.
- Lobbying downstream projects into using said facilities (perhaps making false promises about low coupling during the process); when the facilities extend the current ones, pushing for the adoption of these extensions, which creates compatibility issues for competing projects.
- After their own project becomes the *de facto* standard, tying said facilities to the project knowingly without technical advantages, supplanting other "incompatible" projects.

It is undeniable that the open-source community is not an undisturbed utopia, and that it is instead full of controversies or even so-called "holy wars", but as far as I know there has never been a controversy where the developers involved adopt practices like EEE as blatantly as systemd developers do[24]. I am of the opinion that FOSS developers should have higher moral standards than PCSS developers do, so these

[20] With the chainloading technique (*cf.* Section **??**) which has existed since the beginning of this century, we can completely avoid the binary logging format used by `journald` in systemd, and yet implement most user requirements doable with `journald` in simpler and more reliable ways. Moreover, even if we do not consider the superfluity of `journald`, I have never seen any technical advantage for mandating the forwarding of logging information through `journald` when we just want syslog, and adding the feature that allows syslog services to directly listen to logs does not even seem difficult to systemd developers: they only need to allow configuring `journald` to not listen on `/dev/log`.

[21] However, it can be said that systemd developers' reasons for not supporting separate mounting of `/usr` without using an initramfs are very untenablesaellaven2019a.

[22] The appropriateness of criteria for the results from both votes in the original contexts were disputeddasein2015, coward2017, but anyway the injustice of the behaviours of the systemd developers themselves is unaffected regardless of the goodness of Debian's decisions. Similarly, the existence of elogind does not disprove the fact that systemd developers expected `logind` to be tied to systemd, so it does not invalidate the above-mentioned injustice, and similar conclusions also hold for events mentioned in the following related to udev and kdbus.

[23] After questions by other kernel developers (*eg.* lutomirski2015), the technical reasonshartman2014 for the merger was gradually shown to be untenable, and finally kdbus never made it into the kernel.

[24] For instance, GNU software is often criticised as too bloated, and supplanting simpler workalikes due to its feature creep, but most GNU software systems can be replaced quite easily. GCC might be one of their software systems with most hard downstream dependencies, but on one hand its feature set does not seem to be easily implementable with low coupling (*eg.* LLVM, which competes with GCC, does not seem to have an architecture much better than GCC), and on the other hand we do not have strong evidences that GCC developers added tight-coupled new features knowingly without technical advantages.

proprietary practices are, although compatible with mainstream open-source licences[25], more outrageous than similar practices in PCSS communities to me, or as Laurent Bercot put itbercot2015a, bercot2015b (I call it **free / open-source mal-bloatware**):

> systemd is as proprietary as open source can be.

Although the systemd debacle has not yet reach its end (I will discuss how to speed up this process in Section **??**–**??**), we can already examine the issues it highlights: what is the root of this debacle, and how can we avoid similar debacles in the future? As was noted in Section **??**, I think the technical root is the negligence of software simplicity resulted by the disappearance of limitations on hardware resources, and the attendant low cohesion and high coupling in critical system software was "innovatively" combined by its developers with the EEE ploy to create the current status of vendor lock-in. To avoid this kind of debacle from occuring again, we need to realise that **free / open-source software should embrace the Unix philosophy**, because only in this way can we block the way EEE uses to infect the open-source community through low cohesion and high coupling. One opinion (*eg.* bugaev2016) thinks that systemd and its modules were actually compliant to the philosophy, but now we should clearly know that it is wrong, and the lesson we should learn from it is that when discussing the Unix philosophy, we should keep it firmly in mind that what we discuss is the **total complexity of the system**: in comparison with the previous system which fully reuses existent tools, after seeing systemd's architecture with low cohesion and high coupling, its complex external dependenciesgithub:sdreadme (increasing the external coupling of systemd), and its reimplementations of functionalities from tools already existing in the systemwosd:arguments[26] (ignoring collaboration and reuse), which would we consider systemd to be, "small, minimal, lightweight" or "big, chaotic, redundant, resource intensive"poettering2011b?

In the end of this section, I want to stress that while the systemd debacle is a huge challenge to the open-source community, it is meanwhile an important opportunity: as was noted above, it lets us clearly see the severe consequences from the blind omission of the Unix philosophy, and the result of not learning the lesson from it would inevitably be to "make later generations lament the later generations"; systemd is destined to be introduced into the hall of shame in the open-source community, but on the other hand it also urges us to re-examine those excellent software systems (including "unpopular" software like Plan 9 and daemontools), and to learn from them how to apply the Unix philosophy in actual work. More details on this will be discussed in Sections **??**–**??**, and here I only give two additional comments regarding FOSS. First, the pursuit of simplicity can **save time and energy for volunteers** (which exist in large numbers, do not have constant cash support, and have made huge contributions), making it easier for them to concentrate on the most meaningful projects, which is particularly relevant nowadays where new requirements continuously emerge with the advancement of technology. Second, simple, clear code naturally encourages the user to participate in development, which **increases the effective review of FOSS** and therefore contributes to the improvement in software quality, and this may be a way, from the origin, to avoid the next Heartbleed catastrophe and realise Linus's Lawwiki:eyeball

> Given enough eyeballs, all bugs are shallow.

11 Practical minimalism: a developer's perspective

In the end of last section, I mentioned that the pursuit of simplicity can save time and energy, which was in fact a simplified expression. In order to make software simple, the developer needs to invest quite a large amount of time and energy in architecture design, so the tangible production in the short term may seem smaller than that by a developer who throws quick and dirty solutions at problems. Nevertheless, once the same requirements are implemented, in comparison with bloated and obscure code, simple and clear code will possess higher reliability, maintainability and security, and therefore, in the long run, will save the time and energy consumed by the developer during the whole lifecycle of the software. In commercial development, sometimes in order to seize the market and rapidly release new features, it may be necessary to put simplicity on a secondary position to realise "move fast and break things" as with Facebook, but I believe pursuing simplicity should still be the norm in the long run, for two reasons. First, seizing the market is a fairly rare requirement in open-source development because in principle

[25] Incidentally, tivoisationwiki:tivo is also compatible with most open-source licences.

[26] Few of the reimplementations do better than the originals, and some of them (*eg.* wouters2016, david2018) can even be called disasters; the stock replies from systemd proponents are "these functionalities can be disabled" (completely ignoring the issue of whether they were technically better) and "do it if you can" (totally oblivious of the rule that "who caused the problem fixes it"torvalds2014).

naturally excellent software projects can succeed by their quality[27], so projects that often have this need invariably remind me of low software quality and dirty practices represented by "embrace, extend and extinguish". Second, most features used to seize the market will not be discarded soon, so in order to keep the project maintainable, refactoring will happen sooner or later. From this we can see that in software development, and in particular open-source development, the Unix philosophy, which pursues simplicity, is a principle worth following, and in this section we will discuss how we can actually follow this principle.

Before delving into the details, we need to have a general principle for actual practices: since we pursue simplicity, we should keep in mind that simplicity is something we can be proud of, and use the implementation of specified requirements as minimal viable programsarmstrong2014 as one standard for programming capabilities, instead of only looking at the amount of code one produces. To achieve this, we should firmly remember what Ken Thompson said, **take pride in producing negative code**wiki:negcode**, and often consider refactoring**:

> One of my most productive days was throwing away 1000 lines of code.

Merely having the attitude for pursuing simplicity is of course not enough, and we also need actual methods to improve our abilities to write simple code; among them, **studying available excellent software projects and related discussions** is an important way to achieve such improvement. I personally recommend beginning the study from projects by Laurent Bercotska:software, the Plan 9 projectwiki:plan9, the suckless family of projectssuckless:home and discussions on the cat-v websitecatv:hsoft[28]. The key to the design of simple systems based largely sound foundations (*eg.* Unix) is **clever reuse of existing mechanisms**, which requires deep understanding of the underlying nature of these mechanisms, and following are some typical examples that are particularly worth studying:

- It was noted in Section **??** that the operations on attributes of system resources can be thought as communication between the userspace and the kernel that passes special data, and the communication can be mapped to operations on control files, avoiding the `ioctl()` family of system calls.
- qmailbernstein2007 implemented its access control using the user-permission mechanism of Unix, its mailbox-aliasing mechanism using the file system, and its separation between code for the transport and application layers following the idea in `inetd` (and in fact, the UCSPIbernstein1996).
- LMDBwiki:lmdb implemented a key-value store with excellent properties and outstanding performance in a few thousands of code using mechanisms like `mmap()` and copy-on-write, and eliminated garbage collection by using a page-tracking mechanism based on B+ trees.
- When discussing possible implementation strategies for message passing in the publication–subscription (bus) style, Laurent Bercot notedbercot2016 that the data transfered need garbage collection based on reference counting, and that this requirement can be implemented using file descriptors in Unix.

It was noted just now that the key to designing simple systems based on largely sound systems is clever reuse, but what if unsound parts are encountered? In fact, today's mainstream Unix-like systems have many unsound parts from the lower levels to the upper levels, and insightful people do not ignore them, for instance:

- As was noted in Section **??**, the BSD socket mechanism and the X Window System were landmarks in Unix's evolution toward a bloated system, and Plan 9 was the result of Unix pioneers' in-depth thinking on issues represented by them; similarly, as was noted in Section **??**, process supervision was the result of reflections on the management of system services and their logs by Daniel J. Bernstein *et al.*
- The currently mainstream standard C library interfaces are far from idealska:djblegacy, and POSIX, the portable standard, is just a unification of existent behaviours in Unix-like systems, whether these behaviours are sound or not. Bernstein carefully inspected these interfaces when writing software like qmail, and defined a set of interfaces with much higher quality by catious en-

[27] Here I say "in principle" because there are some subtle "exceptions", and I use Plan 9 as an example. Plan 9 has four formal releases hithertowiki:plan9, where the first release (1992) was only available for universities, the second (1995) only for non-commercial purposes, and only the third and fourth releases (2000 and 2002, respectively) were truly open-source, so it completely missed the best opportunities to gain influence through free distribution. On the other hand, after Plan 9 was made open-source in 2000, due to the quite aggressive minimalism (you can have a glimpse of it at catv:hsoft) of people in its circle, they flatly refused to implement requirements like web browsers supporting JavaScript; I concur that these requirements are ugly, but not implementing them naturally made it very hard for users to adapt, since after all non-smooth transition is a general difficulty in the realm of software. In addition, the increasingly severe disregard of simplicity in upper-level development (if you have once compiled the Bazel program used to build TensorFlow, you would perhaps have an intuitive experience for this; despite the employment of Ken Thompson and Rob Pike *et al* at Google, these software systems are so strikingly bloated, which leaves me puzzled) also contributed to the divergence between Plan 9 and "modern" requirements, and one important goal of this document is to raise the awareness for minimalism.

[28] I find the cat-v site significantly aggressive, and therefore suggest critical reading of its contents.

capsulation of system calls (*eg.* `read()`/`write()`) and not-too-bad library functions (*eg.* `malloc()`). The latter interfaces were taken out from qmail *etc* by other developers, and now have multiple descendants, among which I think the skalibs library is the best[29].

- The Bourne shell (`/bin/sh`, and descendants like `bash`, `zsh` *etc*) widely used in Unix-like systems has quite weird interfacesska:diesh, for instance the value of a variable whose name is stored in `$str` usually has to be accessed through the dangerous `eval`, because `$$str` and `${$str}` cannot be used; nevertheless, these pecularities are not inherent weaknesses in all shells, for example the `rc` shell from Plan 9 avoided most pitfalls in the Bourne shellvector2016b.

Similarly, we should get used to **looking upon existing software critically**: for instance the GNU project, which spreads software freedom, produced many mediocre software systemsbercot2015c; OpenBSD, which values security extremely, has support for POSIX even worse than that of macOSbercot2017; Apache, once the "killer application" in the open source community, is implemented in a bloated and inefficient way. These problems do not affect the relevance of these projects, but we should not be complacent with the *status quo*; furtherly, we should treat the current achievements and trends calmly, and **focus on the essence instead of the phenomena**: for instance when reading here, you should already have some rough understanding of the essence of many features in systemd, and how necessarily (or not) correlated they are with the architecture of systemd; in Section **??**, I will present my understanding of the notion of "language features", which might help you to think about some currently popular languages from a new viewpoint.

It is necessary to note that people mentioned in this document that have made great contributions also make mistakes: for example, the root cause of the "silly qmail syndrome"simpson2008 was Bernstein's incorrect implementation of the SPOOLing system composed of the mail queue, mail sorting (the writer) and mail transfer (the reader), and SPOOLing is a fairly useful notion in operating systems; Tony Hoare, in his Turing Award lecturehoare1981, recommended compilers to use the single-pass scheme, but we will see in Section **??** that multi-pass processing can be simpler, clearer and more reliable than single-pass processing; the designers of Plan 9 adopted a networking architecture with multiple terminals connecting to a few central servers at Bell Labs due to concerns about hardware prices, and noted that although Plan 9 can be used on personal computers they rejected that kind of usagepike1995, but this design is evidently not quite applicable nowadays with cheap hardware and people increasingly distrusting centralised servers. Thus when anyone can make mistakes, who should we trust? I find two key points:

- **Analyse specific issues specifically**: examine the proof and its bases, and note whether the bases are still applicable in the current scenario[30]. For instance it is well known that `goto` statements should be avoided in C programming, but many projects, including the Linux kernel, employ "`goto` chains"shirley2009 because `goto` is avoided to prevent complicated back-and-forth jumping and avoid spaghetti code; on the contrary, `goto` chains not only do no harm to the readability of the code, but also have better readability than all other equivalent ways to write the code. Similarly, the goal for summarising the essence of the Unix philosophy as a complexity issue in Section **??** was not to make it pretty, but to correctly understand its spirit when determining the quality of systems.
- **Investigate extensively and listen broadly**: listen to opinions from multiple sides, in order to have a view of the problem scope that is as complete as reasonable. For example the "success" of systemd for the time being was largely because most people in the Linux circle only knew sysvinit, systemd, Upstart and OpenRC, and almost ignored the daemontools-ish design and its potentials; after adequately understanding the designs of various init systems, and the comparison between them by proponents and opponents of these systems, you would naturally know which init system to choose. Similarly, some systemd proponents will inevitably fight the points in this document with teeth and nails, and I just hope you to read enough of this document and its references, and then form your own conclusion by combining opinions you know from multiple sides.

Before concluding this section, I would like to discuss some additional issues about systemd. First, the systemd debacle has not yet finished, and we can accelerate this process only by speeding up the perfection

[29] Its author notedska:libskarnet that under many circumstances, static-linked executables of programs written with skalibs are one order of magnitude smaller than those of workalikes written with the standard library or other utility libraries. By the way, dynamic linking was originally created to work around the high space consumption of the bloated X Window System, and similar issues have already been largely alleviated with the progress in hardware resources, so the various troubles with dynamic linking make simplicity-minded developers even more inclined to use static linking than beforecatv:dynlink.

[30] Similarly, this document intentionally deviates from the academic convention that avoids citing Wikipedia, which is why I list only "References", and not a "Bibliography"; the reason is that I want to provide the reader with materials that are (more or less) more vivid and are constantly updated.

of its substitutes and making them close to complete in terms of actually implemented requirements, so I appeal all developers disappointed by the terrible properties of systemd to take part in, or keep an eye on the development of these substitutes. Second, although there are projects like eudev and elogind, the code from their mother projects is of mediocre quality (or otherwise they would not have been easily tied to systemd), so the race on their development against the systemd project, which has abundant human resources, would naturally be disadvantageous; on the other hand, we should focus more on supporting alternative projects like mdevd and skabus, which start from scratch and pursue simple, clear code, in order to let "the good money drive out the bad". Finally, "**do not foist what you dislike upon others**", when participating in unfamiliar open-source projects, we should pay due respect to their traditions, and avoid the tendency to impose our opinions on others like that of systemd developers, which is also relevant with projects unrelated to init: for instance, due to influence from multiple factors, different people pursue simplicity to different degrees, and this personal choice needs to be respected, as long as it is made after sufficient investigation and careful consideration, and does not infringe upon other people's choices.

12 Practical minimalism: a user's perspective

It was noted in last section that for developers, pursuing simplicity might be disadvantageous in the short term, but would save time and energy in the long run, and a similar statement also holds for the ordinary user: for example, Windows admittedly has a "user-friendly" graphical interface, but once some harder tasks are involved and there is no ready-made tool, in Windows we would need its tools similar to Unix shells (batch scripts, VBscript, PowerShell, or cross-platform languages like Python) anyway. From this we can see that if you want a self-sufficient experience, you would need to learn those "user-unfriendly" tools sooner or later, and since we have already seen the great power from combining Unix tools with the shell in a quite intuitive way in Section **??**, this learning process would really be rewarding. I believe the shell is not hard to learn, and the key is to understand its essential usage instead of the eccentricities of the Bourne shell mentioned in the last section. Here I suggest carefully learning `rc`github:rc after a preliminary walkthrough of the Bourne shell, because the former concisely represents the main ideas in shell scripting, so when coming back to the Bourne shell after this, it would be easier to know which parts are less important.

Furtherly, just like the shell in connection with graphical interfaces, simple software systems have similar relation to complex ones, and here I use RHEL/CentOS and Alpine Linux[31] as examples. Red Hat's system is big and "comprehensive" just like Windows, and usually works as expected when used exactly in ways intended by its developers, but the system has too much additional and underdocumented encapsulation in order to make the "intended ways" more "user-friendly"; the attendant problem is that when faults occur the system is hard to debug because of its high complexity, and the user has to work around the encapsulation by all means when special requirements arise, while worrying about potential effects by the workarounds on the behaviours of other system componentssaellaven2019b. Exactly the opposite, Alpine Linux uses simple components, like musl and BusyBox, to construct its base system, and tries hard to avoid unnecessary encapsulation and dependencies, which makes it, while certainly less "user-friendly" than Red Hat's system, highly stable and reliable, and easily to debug and customise. This of course reminds us of the comparison between systemd and s6/s6-rc in Section **??**, so it can be intuitively felt that the complexity of software not only affects developers, but also has apparent effects on users, as was remarked by Joe Armstrong, the designer of the Erlang language[32]:

> The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

As was mentioned in Section **??**, I believe that the effect of software complexity on user experience is particularly relevant for free / open-source software systems, because their internals are open to users, which gives the user the possibility of debugging and customisation by themselves.

Due to the reasons above, I believe that **even for the ordinary user, complexity should be taken into consideration when choosing software systems**, and simple software like musl, Busy-

[31] Incidentally, if most distributions should be called "Some Name GNU/Linux"instgentoo:interj, then should Alpine be called "Alpine BusyBox/musl/Linux"?

[32] I am very aware that the original remark was meant for object-oriented programming, and as a matter of fact, implicit states are factors that introduce coupling just like global variables. So OO systems that lack separation between the states of objects have the problem of high coupling reminiscent of the similar problem in complex software systems, and this problem is particularly severe when there are multiple levels of encapsulation and inheritance.

Box, s6/s6-rc, Alpine Linux, as well as `rc`, vis, abduco/dvtm, are to be preferred. When choosing between simple but nascent software (like BearSSL) and complex but relatively well-reviewed software (like LibreSSL/OpenSSL), prefer the former as long as its author has good track records and considers it sufficiently practical for use. When having to choose between multiple bloated software systems, prefer the relatively simpler and better-tested, for instance when using systemd-based systems, prefer the generic tools and avoid the reimplementations in systemd. On the other hand, even when lightweight alternatives cannot adequately implement our requirements, we can still support or pay attention to them, to be able to migrate as early as reasonable after they become adequately feature-complete: for example, I first noticed the s6 project one year before the first release of s6-rc, and I began to prepare the migration of my systems to s6/s6-rc after the release (so that the systems would support service dependencies and oneshot init scripts), eventually realising the migration one year later. s6-related software were admittedly not sufficient for the requirements of the average user when systemd became the default init system in Debian, which was one technical reason systemd could successfully achieve its domination. However, s6/s6-rc is now very close to the standard of fulfilling the requirements of the average uservector2018a, vector2019a[33], and it also possesses excellent properties that systemd developers alleged but failed to realisepoettering2011b, so I request all interested users to actively watch and participate in the development of s6/s6-rc and related software.

Sometimes the user's choice of software systems is constrained by factors like requirements at work, and I think that in this scenario the user can use the system in a way that is as simple, clear as reasonable under the current constraints, and remember to leave room for more ideal solutions, in order to **minimise the cost of possible migration in the future**. For instance, most Linux distributions write their system maintenance scripts in the Bourne shell instead of `rc`, while the user may need to customise some of them (which is also why it was not suggested in the above to only learn `rc` and skip the Bourne shell), and the user can write the customised parts in a form that is simple, clear and close to the `rc` way as far as is reasonable. Another example is that I have to interact with the systemd-based CentOS 7 due to work requirements, and that my choice is to do most work on a machine controlled by myself, and try to use CentOS 7 in a simplest way in virtual machines, in order to minimise the dependence on it.

If you are using systemd relucantly because of software systems (Linux distribution or desktop environment) you are used to, I suggest you to actively try out simple software, gradually get rid of software strongly coupled with systemd, and henthforth bear firmly in mind the lesson from the systemd debacle: in this nonpeaceful open-source community, **it is only by mastering the secret weapon of simplicity that we can master our own destiny when encountering dangerous traps like systemd** and avoid the catastrophe instead of getting trapped[34]. To achieve this, I consider it necessary for the user to be especially wary of bloated components, like GNOME, in the system, and when it shows the tendency to be closely coupled to suspicious projects, consider lightweight alternatives like Fluxbox, Openbox, awesome and i3 in time: after those bloated projects get tied to mal-bloatware, even assuming their developers are totally well-intentioned, they would not necessarily be able to afford cleaning up the infections in their own projectsvitters2013, which is a natural result of the high complexity of these projects. Furtherly, I think the user should keep a necessary degree of caution against those overly bloated software projects with major developers funded by a same commercial corporation like Red Hatsaellaven2019a[35]: from the examples of Chromium and Firefox (*cf.* Section **??**), we know that companies in the open source community can also sacrifice user interests in favour of commercial interests, so we would get attacked on multiple fronts if the developers of these projects are asked to conspire in "embrace, extend and extinguish"; and even if we take a step back and assume that EEE is not a direct result of abetting from the employer, employers that connive at employees who use filthy practices to monopolise the market, which creates conflicts of interests between the companies and the open-source community, are still very despicable, and we should vote with our package managers to express our contempt of these companies.

You may notice that while the section title says "a user's perspective", the simplicity of some software mentioned in this section can only be sufficiently appreciated when the user has a certain level of

[33] Besides, I guess all systemd functionalities, that are meaningful enough in practice, can be implemented in an infrastructure based on s6/s6-rc, and the codebase of many among them will be smaller than 1/5 those of their systemd counterparts.

[34] Even in Gentoo, which compiles everything from the source code and is very customisable, users who want to use GNOME 3 without systemd used to have to jump through multiple hoopsdantrell2019, and those who would like to separately mount `/usr` without using an initramfs still have to maintain a patchset by themselvesstevel2011 because of developers blindly following systemd practicessaellaven2013.

[35] Furtherly, it is also necessary to be cautious with complex standards driven by commercial companies, like HTML5 which is now driven by big browser vendors and the companies behind them. In addition, I would like to express my sincere esteem for the BusyBox developers that, although working at Red Hat, have the courage to stand up against systemd developersvlasenko2011.

background in it; actually, it was mentioned in Section **??** that Unix was designed as a system to boost the efficiency of programmers, and we also see that Unix works best when its internal mechanisms are adequately understood by the user. However, most users have after all only limited energy to understand the mechanisms of their systems they use, so does this imply only programmers could make good use of Unix-like operating systems? I do not think so, and instead I believe a user without related backgrounds previously just needs to overcome his/her repulsion of programming, and **start with good tools and tutorials**, which would not consume too much energy: for instance, assuming one carefully reads the documentation of `rc` after roughly learning the Bourne shell, and then comes back to review the latter, even a newbie would be able to understand basic shell scripting within one or two days, provided that enough exercises are done; if another one or two days are invested in studying basic notions related to Unix processes, the user would be able to start learning s6/s6-rc. After studying the basics, the user would save a lot of energy during the learning process if the following points are kept in mind:

- **Remember to assess the importance of knowledge**: often think about which knowlege that you learn is more **common** and more **useful**, as was noted in dodson1991 (we will further discuss this statement in Section **??**):

  > Self-adjoint operators are very common and very useful (hence very important).

- **Dare to ask for help, and be smart when asking**: when facing problems that you cannot solve, make full use of the power of the open-source community; but remember to express the problem in a concise and reproducible manner, and also present your own efforts at solving it.
- **Avoid over-programming**: as was noted in last section, take pride in simplicity, and pursue problem solving with simple, clear methods; avoid over-engineering in everyday work, and consider manual completion of small-scaled tasks.

Before ending this part, I would like to quote the famous sentence by Dennis Ritchie:

> Unix is very simple, it just needs a genius to understand its simplicity.

This sentence may be called both right and wrong: it is right because making good use of Unix requires deep understanding of its mechanisms, while it is wrong because this requirement is not difficult to achieve, with proper guidance, for those who are eager to learn. I believe that the essence of efficient Unix learning is **minimisation of the total complexity of the learning process required for the completion of specified practical tasks**, and that the key to it is fostering the habit of minimising the total complexity; this habit connects one's work and life to the Unix philosophy, and I will elaborate on its relevance in Section **??**.

13 Hilbert's 24th problem: minimalism in mathematics

In 1900, David Hilbert published a list of 23 unsolved mathematical problems, and these problems, to be called "Hilbert's problems", influenced mathematicians throughout the 20th century, and still continue to have major influences; in 2000, one additional problem was discovered in Hilbert's notes that was somehow excluded from the original publication, and we begin this part with this 24th problemwiki:hilbert. What Hilbert's 24th problem asks for is essentially a formal standard for simplicity of mathematical proofs, and a method to demonstrate that a certain proof is the simplest for a theorem. From this we can already see the pursuit of simplicity also exists in mathematics, and this pursuit is in fact shared by other great mathematicians. For example, Paul Erdős often referred to "THE BOOK"[36], an imaginary book where God keeps the simplest proofs, and said in a lecture in 1985 that

> You don't have to believe in God, but you should believe in THE BOOK.

It also seems that the pursuit of simplicity is not an exclusive for masters, because Edsger W. Dijkstra once said:

> How do we convince people that in programming simplicity and clarity – in short: what mathematicians call "elegance" – are not a dispensable luxury, but a crucial matter that decides between success and failure?

From the quotation we know that mathematicians, in a sense, appear more minimalist than programmers; in Section **??**, we will see an extreme example for minimalism in mathematics.

In Hilbert's original note on the 24th problem, he attempted to reduce a certain class of proofs into a class of sequences mainly consisting of elements of a certain type, and judge the simplicity by lengths of the sequences. Nowadays, from the viewpoint of programming languages, this can be generalised quite easily[37]: proofs (like programs) are after all based on axioms (like primitive operations), and by embedding proofs for all theorems (like implementations of library functions) a proof involves, said proof can be reduced into a sequence of applications of the axioms, which can be counted. With this correspondence in mind, we can **"refactor" proofs and propositions**, just as with programs, to make them shorter and clearer, for example

> For discrete probability measures constrained by a set of moment conditions, if a probability measure exists with the (*) form, it must be unique and must be the maximal-entropy distribution; and if the maximum-entropy distribution exists, it must have the form of (*) and therefore must be unique.

can be refactored into

> For discrete probability measures constrained by a set of moment conditions, the maximal-entropy distribution exists if and only if a probability measure exists with the (*) form, in which case they must be identical and unique.

Following the line of thought above, we should be able to construct the formal standard for simplicity desired by Hilbert; however, what about the part about showing a proof to be the simplest? I guess there probably exist proofs that cannot be demonstrated to be the simplest, and here I explain my guess with the notion of Kolmogorov complexity, which will be further discussed in Section **??**. To show a proof to be the simplest, the minimal complexity of possible proofs probably needs to be sought, which seems to be closely related to the Kolmogorov complexity. However, even notwithstanding the uncomputability of the complexity, Chaitin's incompleteness theorem shows that there exists an upper bound for it that proofs could be shown to have or surpass. So what if the complexity of the theorem to be proven is already higher than this upper bound? Under this circumstance, even the minimal complexity of possible proofs cannot be demonstrated, which does not bode well for what Hilbert wanted.

14 Boltzmann's formula: an axiom or a theorem?

Back in an interview during the application for the postgraduate program I got enrolled to eventually, I was asked the question "is Boltzmann's entropy formula, $S = k_\mathrm{B} \ln \Omega$, an axiom or a theorem in

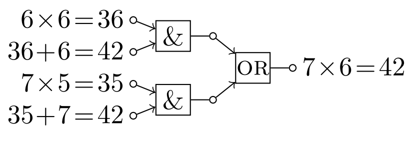
[36] A real-world "approximation", *Proofs from THE BOOK*wiki:thebook, was dedicated in 1998 to Erdős, who gave many advices during the preparation of the book but deceased before the publication.

[37] The correspondence can be formalised through **Gödel numbering**wiki:godelnum, which maps different programs into different integers; similar operations can also be done on proofs, so every proof is formally equivalent to a program. The converse is not true, because a proof has to finish in finite steps, while a program does not have to; nevertheless, since algorithms also need to end in finite steps, every algorithm is formally equivalent to a proof.

statistical physics?”, by one of the interviewers. My answer, “it can be both”, was quite a surprise for the interviewers, judging by their reactions; I then explained what I meant was that one set of propositions can be based on multiple alternative basic sets of axioms, and that a proposition can be an axiom with some bases and meanwhile a theorem with other bases. Perhaps because the full answer was still quite unexpected to the interviewers, which were mainly physicists, they did not ask further; but the question should obviously not stop at that point, and I will examine it closely in this section.

An immediate follow-up question to my answer could be “then which choice of axioms would be better?”, because **all choices of axioms are not equal**, although they can be “rebased” onto each other: for example, a given set of axioms can be extended with a known theorem, and the resulting set of axioms would trivially be, although redundant, as consistent as the previous one. Moreover, even among mutually equivalent minimal choices, some can be clumsy to use[38]: metaphorically, although many Turing-equivalent models of computation are available, computer scientists would most often use Turing machines, lambda calculus and a few other models for generic discussions, instead of cellular automata or even `sed`.

$$\begin{tikzcd}
 & \text{Rolle's theorem} \arrow[dl, shift right] \arrow[dr, shift left] & \\
\text{Lagrange's mean value theorem} \arrow[ur, shift right] & & \text{Cauchy's mean value theorem} \arrow[ul, shift left] \arrow[ll]
\end{tikzcd}$$



I approach the latter problem with a model based on graphs, which I call “**deduction graphs**”, with propositions as their nodes and deductions as their edges, so the graphs are closely related to dependency graphs yet very different from the latter: first, there are often “equivalent” propositions that imply each other, so the graphs usually contain loops, unlike dependency graphs which are acyclic (*cf.* the relation between the three mean value theorems shown in the picture above on the left); second, a proposition may well be proven in multiple ways, so disjunctions, or in other words “virtual nodes” (*eg.* the “$7 \times 6 = 42$” node in the picture above on the right), need to be included; third, some proofs are more complex than the others, so the edges are weighted. Given this model, we can compare choices of axioms based on the total path weights of the spanning trees from the different bases, which correspond to the intuitive notion of how complex it is to generate all the propositions.

Deduction graphs are probably impractical for quantitative use in real-world applications, in large part due to the formal systems being infinite; but they are very instructive in a conceptual sense, and we will refer to it multiple times in this part. For example, the model helps to explain the criterion for importance of knowledge quoted in Section **??**: by mastering notions that are common (linked to many other parts in the graph) and useful (simplifying otherwise complicated deductions), we would be able to understand the subject matter much more efficiently. And before concluding this section, let’s briefly come back to the original question on Boltamann’s formula: I guess that it will be a theorem with almost all reasonable minimal choices of axioms, as long as entropy S is a derived, instead of primitive, notion in statistical physics.

15 Kolmogorov complexity and knowledge organisation

In previous sections, we have scratched the surface of the issue of lower bounds for complexities in certain scenarios multiple times: for instance, when considering how to demonstrate the simplicity of a proof for a given theorem (*cf.* Section **??**), I examined the notion of minimal complexity of possible proofs; and earlier when discussing cohesion (*cf.* Section **??**), I defined it as the inherent coupling between submodules, which also correlates with some sort of inevitable complexity. When considering the intrinsic complexities of objects, the **Kolmogorov complexity**wiki:kolmogorov is often used, and in this section we will examine it closely.

In a nutshell, the Kolmogorov complexity of an object, usually encoded in a string, is the smallest length of computer programs that can be used to produce it, and its value has very limited dependence on the programming language used: the difference between complexity values from two languages has an upper bound that depends only on the pair of languages, and not on the string in question. The

[38] When I was a PhD candidate, out of curiosity about functional programming, I attended a highly introductory report on category theory, and the reporter mentioned some mathematicians’ attempts to rebase mathematics from set theory onto category theory. While the effort can be academically interesting, I sort of doubt its practical significance: from the extremely limited parts I know, category-theoretical formulations of fundamental mathematical notions appear more complicated than their set-theoretical counterparts, so the former do not seem superior to the latter according to the model in this section.

Kolmogorov complexity is an **uncomputable** function, or in other words it is provably impossible to write a program to compute it for all inputs; furthermore, Chaitin's incompleteness theorem states that there exists an upper bound for provable lower bounds for the complexity, which only depends the language and the axiomatic system used for proofs, so we cannot even prove a string is more complex than this upper bound.

You would certainly wonder, given such "bad" properties of the Kolmogorov complexity, is it still meaningful? To my knowledge, it is indeed rarely used in practical measurements of complexity, and instead used mainly in proofs for impossibilites, like the uncomputability of certain functions. Nevertheless, this does mean one thing: as can be easily seen above, the limit to information compression[39] is the Kolmogorov complexity, so the uncomputability of the complexity and the unprovability of its lower bound under most circumstances imply that there will never be a formal end to the research on information compression. Similar conclusions exist for other research fields, as the term "full employment theoremwiki:employ" summarises; on a slightly philosophical level, while the "bad" properties show the limitations on formal theories available in these fields, they also mean that **human efforts will not be easily replaced by computer programs**.

It is necessary to notice that the Kolmogorov complexity measures the intrinsic complexity of a known object, and does not easily extends to comparison between multiple possible but unknown objects, like the possible proofs of a theorem. Furthermore, it is even less applicable to problems that are closely related to information compression but are less formal, like the field of **knowledge organisation** in library science; I believe that the latter in its general sense is the real-world counterpart to the minimisation of system complexity in programming, and therefore has to be more or less minimalist. However, just like deduction graphs, even though the Kolmogorov complexity is not quantitatively applicable to practical problems (not exactly in fact, and we will see some indirect applications in Section **??**), it is still conceptually instructive because of its nature as the **descriptive complexity**.

16 Minimalism in science and technology

So far in this part, we have been mainly discussing minimalism in mathematics and hypothetically axiomatic physics, but minimalism can also be observed in other fields of science and technology, and especially in those with **considerable and tangible costs of complexity**, including both financial and mental costs. Traditional engineering is quite a good example for this, because the financial cost of complexity are still obvious even with the advancement of technology: as long as an engineering project is under a somewhat tight budget, a simple way to do same task would usually be preferred over a complex way, provided that the simple way does not imply some serious trade-off. For instance, Jon Bentley wrote in *Programming Pearls*[40] thatbentley1999

> General Chuck Yeager (the first person to fly faster than sound) praised an airplane's engine system with the words "simple, few parts, easy to maintain, very strong".

Minimalism is also an obvious pursuit by theoretical physicists: Fred Brooks, the author of *The Mythical Man-Month*, wrote in *No Silver Bullet*brooks1987 that

> Einstein repeatedly argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.

This attitude reminds us of the reduction of system calls in Plan 9 (*cf.* Section **??**); similar attitudes were also evident in other prominent figures in theoretical physics, such as Issac Newton (noticing his *Principia Mathematica*) and James Clerk Maxwell (noticing his equations). Minimalism in the form of elegant deductions, comparable to elegant implementations of requirements, can also be observed: one of my friends, who was major in mechanics, once mentioned that while many textbooks in fluid dynamics spend lots (sometimes one or two chapters) of text analysing gravity waves[41], Lev Landau and Evgeny Lifshitz managed the exposition in just one section of around 10 pages in their *Course of Theoretical Physics*, which even included a few pages of exercises with answers.

The tradition to derive systematic theories from a small set of rules can surely be traced to earlier origins, and the *Elements* by Euclid is usually considered as one among those most important original materials. It is almost undisputed that geometry became an indispensable part of Western education

[39] Incidentally, the Hutter Prizewiki:hutter, basically a contest in lossless compression of textual human knowledge, bridges the fields of information compression, knowledge organisation and artificial intelligence.

[40] His column under the same name in *Communications of the ACM* was where the word frequency sorting problem (*cf.* Section **??**) was originally discussed.

[41] Not to be confused with gravitational waves in general relativity.

exactly because of the *Elements*, but the book had a problem with algebra: the algebraic propositions and deductions were all written as sentences, not formulae, perhaps due to the awkward number system at that time. In the 20th century, the Bourbaki group of mathematicians[42] went the opposite way during the course of constructing foundations of mathematics in a purely axiomatic way based on set theory: although their works were indeed highly influential and productive, the contents were written in an extremely austere and abstract way, with few explanatory remarks, a minimal number of examples, almost no pictures, and zero real-world application. While this style did not affect the logical correctness of Bourbaki's works, it created problem for human readers attempting to understand the contents; an in-depth analysis of this issue will be carried out in Section **??**, and here I would just like to discuss one more thing that is perhaps worth some consideration in the end of this section.

When discussing the possible outcomes in case the current foundations of mathematics were proven inconsistentgaillard2010[43], it was noted that most researchers would be able to move on with alternative foundations. After all, set theory was born in the 1870s well after many other fields, and the set-theoretical presentations of many results are basically restatements in a different mathematical language: for example, V. I. Arnold was able to teach group theory, up to a topological proof for the insolubility of general quintic equations in radicals in half a year, to secondary school students in 1963–1964 without using any axiomaticsarnold1998, alekseev2004[44]. Similar observations also exist in physics, like the relative independence of results in phenomenological quantum mechanics from the actually underlying mechanisms, and that of results in macroscopic thermodynamics from basic principles of statistical physics. All these prove the existence of **relative separation of abstraction layers** in formal and semiformal systems, as with software systems: from the perspective of deduction graphs, we can convert axioms in a deduction graph to theorems in a new graph by adding more primitive axioms, and even though propositions that are already theorems may gain new proofs, the overall structure of the entire graph will not change very much.

17 Minimalism in scientific and non-scientific theories

In this part, I have already mentioned multiple scientific theories, each of which based on a minimal set of fundamental hypotheses, which when formalised become axioms. In last section, I noted that this tradition is closely related to the *Elements* by Euclid; I did not restrict the tradition to science and technology, because it was actually also followed by many philosophers and even some theologians. One prominent example that lies between these fields is the answer by Pierre-Simon Laplace when he was asked by Napoleon Bonaparte about the absence of God from his five-volume monumental work, *Mécanique Céleste*: "I had no need of that hypothesis". This tradition is summarised as the princple called **Ockham's razor**wiki:ockham, usually phrased as "entities should not be multiplied without necessity". It should be noted the principle is a criterion about the choice between multiple theories when they are all consistent with available observations and make the same predictions, and does not imply that the simplest theory was most likely correct under all circumstances.

Multiple attempts have been made to mathematically justify Ockham's razor using formulations related to probability, for example the theories about the minimal message lengthwiki:mml and the minimal description lengthwiki:mdl. Both MML and MDL took inspiration from the Kolmogorov complexity discussed in Section **??**, but unlike the Kolmogorov complexity which is hard to use in real-world applications, MML and MDL can be practically used in the choice of statistical models. The MML and MDL approaches differ from each other quite subtly, but the general ideas remain highly similar: somehow represent a model, and concatenate it with the representation of observed data encoded using the model, and it can be proven that **the model with the shortest combined representation is most likely to be correct**. From this we can see a strong mathematical correlation with information compression, so MML and MDL are associated with the Kolmogorov complexity; and if we consider available observations as the data, and theories as the models, the simplest theory that efficiently explains the observations wins according to the result above, which is consistent with the intuitive principle of Ockham's razor. The result is also easily reminiscent of the Unix philosophy in the sense of minimising the total complexity of

[42] Whose use of the bending sign in warnings was the direct inspiration for Donald Knuth's "☡" sign.

[43] In case you wonder why this is even a problem, note that **Gödel's incompleteness theorems**wiki:godelthm (actually the second theorem) show that these foundations cannot prove themselves to be consistent, so the theoretical problem does have some ground.

[44] By the way, quite a few books by Dan Friedman, like *The Little Schemer* and *The Little Prover*, achieved similar goals.

the system, if we compare the intended applications to the data, and the implementations of supporting interfaces to the models.

18 Minimalism as an aesthetic in literature and arts

Pursuing simplicity is also an import aesthetic in literature and arts, and in this section we will briefly examine this aesthetic, mainly through literature. *The Elements of Style*, the prestigious American English style guide, was listed by the *Times* magazine in 2011 as one of the 100 most influential English books since 1923; William Strunk Jr., its author, strongly emphasised "**omit needless words!**", and made the following recommendation on writing styles in the book[45]:

> Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all details and treat his subject only in outline, but that every word tell.

The Elements of Style did not only concern literary works, so what it reflected was a common aesthetic in English writing, and a similar aesthetic already existed in China at least as early as about the 5th century. *The Literary Mind and the Carving of Dragons*, a book by Xie Liu who lived in the Southern and Northern Dynasties period, was the first systematic book in China on literary critique, and likewise considered works in all genres as literature. Ao Li summarised the focus of the book as two points in his *Complete Highlights of Masterpieces in Chinese Literature*:

> One of them was to argue against the pompous style devoid of practicality, and the other was to advocate for the pragmatic style that "crucial issues in military and politics be considered in writing".

Shi-Jin Mou, the first secretary general of Chinese Society of *The Literary Mind*, remarked on the fundamental stances in the book thatmoushijin1995[46]

> There should be rich contents to accompany the refined forms – this is the absolute rule in literary creation, and is the supreme standard in literary critiques by Xie Liu. This fundamental stance was followed throughout *The Literary Mind*.

When evaluating literary works, phrases like "piling up flowery language", "flashy and without substance", "devoid of contents" *etc* are never compliments, which shows that the aesthetic above is well accepted in literature both inside and outside China. On the other hand, also confirming this acceptance is the general appreciation of depictive devices that express rich connotations with subtle details, like the practice of exhibiting dynamics or emotions of characters with just a few strokes in painting, by both Chinese and Western people in literature and arts. Another example is the advocation of "distilling characters" and the attention to "eyes of poems" in Chinese literature, for instance the following line from a poem

is widely praised exactly because the character " " provoked the imagination that the silence in the evening was interrupted by the sound of knocking, and meanwhile suggested (in comparison with the character " ") that the person outside the door was a visiting guest instead of a returning host. I believe that similar examples generally exist in various civilisations, and that the underlying aesthetic was, to a large extent, not transmitted from other civilisations. Just like base-10 numbers emerged independently in multiple civilisations probably because each human has 10 fingers ("digits"), **the independent origination of minimalist aesthetics in disparate civilisations should be due to some deep reasons**, and I will explore the reasons in the next section.

19 Minimalism from a cognitive perspective

In Section **??**, I asked, "is simplicity now only an aesthetic?", and when reading here you should already know it is not, because there are plenty of practical reasons for pursuing simplicity. In Section **??**, I noted

[45] Qiao Dong translated this 63-word maxim into 63 Chinese charactersdongqiao1999: " "

[46] It was noted in the same reference that *The Literary Mind and the Carving of Dragons* was the result of the author's reaction to the pompous style prevalent at that time, but isn't this document actually similar?

that the independent origination of minimalist aesthetics in multiple civilisations should be attributed to some deep origin; actually, I believe that this aesthetic has its cognitive root, which is closely correlated with those practical causes, and is independent of the particular civilisation. In this section, I will discuss the cognitive basis of minimalism, and also examine some apparent "exceptions" to this basis; prepared with these analyses, we will able to proceed to the next and final two sections in this part. I base my argument in this section on the basic hypothesis that **most (if not all) ethics and aesthetics are results of practical requirements in real life**. Why do we dislike people that jump the queue when there are many people waiting? Because this behaviour sacrifices many people's time and energy for the benefits of a few people, and incites people to just wait in a crowded fashion, which reduces the throughput and may lead to safety risks. Why do designers dislike Comic Sans? Because this font, when used with anti-aliasing (which is the absolute norm today), has poor legibilitykadavy2019; this reason might seem subtle, but would appear more intuitive if you recall that when you first learned to write, the teacher would almost instinctly discourage illegible handwriting.

Paul Graham noted that when mathematicians and good programmers work on problems, they hold the problems in their heads in order to fully explore the problem scopes, so simplicity definitely matters in problem solvinggraham2007; he also noted that ordinary programmers working in typical office conditions rarely enter the state where their programs were fully loaded into their heads. Considering that the latter kind of programmers can still produce mediocre programs that more or less achieve the stated goals, but that mathematicians do not often enjoy such relaxation in requirements, this might provide an explanation for the observation in Section **??** that mathematicians seem more minimalist than programmers. Furtherly, I argue that the need to hold the problem in one's head when solving problems is not unique to mathematicians and programmers, because **human are bad at multitasking**: borrowing notions from computer science, we can say that the cost for "context switching" in human mind is huge, so in order to avoid context switching and achieve optimal performance, all relevant subproblems of the problem has to be loaded into the head. With simple yet powerful tools, we will be able to make many subproblems easier to understand, which increases our ability to hold larger problems in our heads, and I believe this is the cognitive root of our appreciation for these tools. V. I. Arnold recalled his instinctive reaction when taught about the intrinsic connections between seemingly unrelated mathematical notionsarnold1998:

> *[... a more impressive example, omitted here for the sake of brevity ...]* Jacobi noted the most fascinating property of mathematics, that in it one and the same function controls both the presentations of an integer as a sum of four squares and the real movement of a pendulum. These discoveries of connections between heterogeneous mathematical objects can be compared with the discovery of the connection between electricity and magnetism in physics or with the discovery of the similarity in the geology of the east coast of America and the west coast of Africa.

Similarly, I was totally amazed when I read about the origin of Aboriginal Linuxlandley2017:

> The original motivation of Aboriginal Linux was that back around 2002 Knoppix was the only Linux distro paying attention to the desktop, and Knoppix's 700 megabyte live CD included the same 100 megabytes of packages Linux From Scratch built, which provided a roughly equivalent command line experience as the 1.7 megabytes tomsrtbt which was based on busybox and uClibc.

V. I. Arnold's comment was meant in comparison with the style by the Bourbaki group (*cf.* Section **??**), and you would probably wonder, if simplicity is so desirable, why would untrained readers feel uncomfortable reading mathematical books written in that style? I base my answer on the analysis of difference between humans and machines in Section **??**, which on a deeper level shows that humans and machines are respectively good at abstract and intuitive tasks. Moreover, in order to fully understand the subject matter, humans also need repeated stimulation that covers its different aspects (which may also help to explain why humans are good at intuition), and I guess this might be a product of the natural adaption to better recognition of important knowledge, as was discussed using deduction graphs. All these tell us that although formal systems only need the abstract notations of propositions and deductions, human additionally require remarks, examples, pictures, and real-life applications to fully understand them; for this very reason, **formal abstractions and informal pictures are mutually complementary and equally important to us**. For example, consider the ϵ-δ definition of limits in calculus and its informal version, "the output can be made arbitrarily close to the limit by setting the input sufficiently close to the specified value", which also captures the underlying topological nature.

There is one more issue to consider at the end of this section: in the above, I noted that mathematicians are often not allowed to produce “roughly passable” results just as programmers often do, but why? Because mathematicians usually need to rigorously prove their results, or at least present conjectures which they strongly suspect to be true; in comparison, programmers consider software as engineering products, and are often willing to sacrifice provable correctness in exhange for lower cost in development and maintenance due to the inherent complexity of software systems[47]. So when formal proofs are too expensive, we have to resort to reasoning by ourselves, and as Tony Hoarehoare1981 noted:

> There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

So we know that while machines only need to know executable instructions corresponding to programs, humans also need to somehow ensure the **correctness of these programs**; and whether the method to ensure correctness is formal proofs or human reasoning, clear understandings of the internal mechanisms of programs will be necessary. Therefore when evaluating a software system, we should not only measure its size, but also examine the cohesion and coupling in the system, which is exactly why I stressed the term “cognitive complexity” in Section **??**.

20 Limits to application of Unix philosophy in society

As we have seen in this part, apart from the Unix philosophy in programming, minimalism also has its relevance in science, technology, philosophy, literature and arts; and this is not coincidental, because minimalism is rooted in human cognition, probably as a result of high cost for human multitasking. A natural question after this is “is the Unix philosophy somehow applicable to society?”, since society is in many aspects similar to a machine, and thus can more or less be analysed from the Unix viewpoint. One important reason for regarding society as a machine is that due to the division of labour, the former can be conceptually divided into interacting groups of people, each group doing a set of different tasks, and groups can again be divided into subgroups doing different subtasks. Following this line of thought, society can be roughly compared to a system composed of interacting modules, and therefore the Unix philosophy seems applicable.

Given this background, it is certainly beneficial to somehow induce **a proper level of separation of concerns** between different groups (and similarly between subgroups inside a single group) in society, and it can also be tempting to suggest extremely fine-grained division of labour such that each person performs a very small and well-defined job mechanically. However, as was graphically shown in Charlie Chaplin’s masterpiece *Modern Times*, making people work like dumb machines would actually be against the human nature, and would result in much more damages than benefits to society. I believe this is one fundamental reason why society often does not work like Unix, and this can be explained by the human need for diverse stimulation as was discussed in last section: if all you did throughout the day was some mechanical and monotonous job, you mind would get full of that job, and would become increasingly estranged from other kinds of human acts. Therefore while it is desirable to produce minimalist works, it is also necessary to **respect the human nature** and encourage people to often try something different and mentally challenging; from another perspective, diverse stimulation also helps to boost people’s creativity, and therefore contribute positively to the vibrancy of the whole society.

As was emphasised just now, humans are not completely comparable to machines, due to the human need for diverse stimulation; similarly, groups of humans are not completely comparable to modules in computer systems, and one main reason for this is that **social groups, once formed, would pursue their interests**, unlike software/hardware modules which just work mechanically. For instance, as was observed by people like Laurent Bercotska:systemd, companies naturally tend to produce mediocre software, and Paul Graham attributedgraham2007 this to companies wanting developers to be replaceable; this tendency (even if it is an inadvertent effect) is destined to be more serious for open-source companies that rely on selling support for revenues, because it is more profitable to produce software that is hard to maintain independently, and I seriously suspect this has been a contributing factor in the systemd debacle (*cf.* Section **??**). From this we know that the interactions between social groups are not purely cooperative, and just like modules in Unix-ish high-security software systemsdjb:qmailsec, social groups

[47] As a matter of fact, **formal verification** is quite widely used in the hardware industry (perhaps due to the relatively fixed sets of requirements for hardware), but still sees very few large-scale applications in software. A well-known example is the formal proofs for the seL4 microkernelwiki:sel4, which is one order of magnitude larger than the codebase itself, and the latter is already very small and well-designed.

should have limited mutual trust; on this aspect, Unix and society have reached some kind of agreement, although in different ways.

21 Minimalism for the average person

I wrote in Section **??** about the essence of efficient Unix learning, and how to realise this goal; after reading all previous sections in this part, you should already know that these are in fact not specific to Unix, and are instead general principles for all kinds of learning processes. So what is the actual role of Unix in learning? In my opinion, the Unix philosophy naturally fosters the minimalist habit, which is the key to efficient learning; in this section, I will discuss how this habit can be better nurtured, and how it can actually benefit the average person.

As I wrote before, efficient learning is basically about reducing the total complexity of learning, and we already have a conceptual model for it – deduction graphs (*cf.* Section **??**); using this model, I analysed the criterion in Section **??** for importance of knowledge, and here I have some additions to the analysis. We know that instead of learning by rote[48], it is usually much better to remember the important points and deduce the rest from them on demand; this means we can "compress" a deduction graph into the common and useful notions therein, and then walk from these notions to the demanded parts. However, it must be stressed that **what we remember is not only the important notions**, but also a rough impression on how they are related to other notions, and this impression is usually (if not always) based on prior knowledge: for example, when learning the multiplication table, acquaintance with addition and subtraction is very useful because it helps the learner to associate unfamiliar results with familiar ones (*eg.* $7 \times 6 = 7 \times 7 - 7 = 49 - 7 = 42$). So for efficient learning, the prior knowledge depended on by the above-mentioned impression is preferably something deeply rooted in our mind, which I believe can be simply called intuition: as was analysed in Section **??**, in order to fully understand something, we need diverse stimulation, and intuition (*eg.* natural numbers and their basic operations) is a result of prior stimulation, which can help to reduce the amount of stimulation needed for learning; this also supports the point in the same section that informal pictures are as important as formal abstractions.

We know from the above that in order to learn efficiently, we should not only focus on the important notions, but also try to find connetions between them and the notions that we have already been acquainted with; we can often begin the latter with actively **identifying similarities between notions**, like "`fork()`/`exec()` is like the Prototype Pattern", and "musical intervals between the 12 notes are similar to time intervals between the 12 hours on a clock". With this practice, we would be able to use prior knowledge to minimise the amount of information we need to remember, achieving the effect called "compressing the book into a booklet" (of the essentials); besides, when new connections between notions are discovered, understanding of the old notions involved are also deepened, and as was noted in Section **??**, profound understanding about the subject matter is often the key to elegant inventions. On a conceptual level, after the discovery of structural similarities between different deduction graphs, deductions from each graph can often offer invaluable insights into previously overlooked deductions from others; on this aspect, the analogy between bouncing macroscopic fluid droplets and microscopic particles governed by quantum mechanics, as well as Chinese Academician Wan-Xie Zhong's work on the computational analogy between classical dynamics and structural mechanics, serve as two good examples.

So we have examined the correlation between multiple deduction graphs, and considered how prior knowledge can help us to master new notions; as was noted in the above, **new knowledge can also help us to consider prior notions in new ways**: in other words, "reviewing the old reveals the new"; this is exactly why we should foster the habit of revisiting old knowledge, and is why I mentioned in Section **??** that we should often consider refactoring. Now that the issue of habits is mentioned again, I find it appropriate to note that both intuition and habits are results of repeated stimulation, and their difference is that the former mainly concerns "what something is" (*eg.* lemons are sour) while the latter mainly concerns "how to do something" (*eg.* how to use some tableware). **Good intuition is highly rewarding, but often requires a certain amount of training to acquire**: for example, a person with adequate training in organic chemistry would immediately recognise the elegance in the synthesis route for "orzane" (through "cyclocatene") in zxhxy2018[49], but the training would not be trivial even for the most intelligent student with the most excellent teacher. The same holds for habits as well, which

[48] Sometimes, learning by rote can be inevitable, and in this case the learner can consider using a flashcard program, which can often dramatically increase the efficiency of memorisation. Mnemosyne and Anki are two open-source flashcard programs, but unfortunately both feel quite bloated.

[49] This is obviously not the original source, but I really could not find one; as far as I know, this synthesis route can be at least traced to 2014, when I drew its final step on a T-shirt after seeing it.

also serves as a warning to us: as was noted in Section **??**, we should respect the human nature, but meanwhile it would be unwise to indulge ourselves in habits that results in more damages than benefits in the long run; as a metaphor, while slouching may feel more comfortable, it usually leads to myopia, hunchback and pigeon chest, all of which I can attest to be unhealthy.

In the end of this part, I would like to consider an example in classical music before presenting the final points: it is perhaps not quite unfair to say most musicians agree that J. S. Bach is the greatest composer in the history of Western music, while L. van Beethoven and W. A. Mozart contend for the second place; however, almost nobody would dispute Mozart's talent being much bigger than Bach's, but why?[50] I think the answer is implied in an observation by one of my schoolmates: "Mozart always had new musical ideas, while Bach constantly explored for new possibilities in existent ideas"; it is perhaps due to this very reason that elements in Bach's works are often called prototypes for musical styles that would appear as late as the 20th century. The point I would like to make from this example is that **while "geniuses" can easily think in depth, they are often not quite motivated to think very hard or rethink old ideas**; the reason is that there are so many sufficiently meaningful problems for them to solve, which however leaves some obscure yet important corners untouched. In programming, an extremely important part of these corners is the simplification of software systems, which is often accessible even to the average programmer as long as he/she is determined enough; in practice this simplification is more relevant for us, the average programmers, so in my opinion we should feel more motivated to do it than "geniuses" are. In addition, as was analysed above, the minimalist habit lets us concentrate on the essence of problems, which naturally gives rise to deep thinking and therefore can often provide profound insights that are the key to difficult problems. Sometimes, these insights are exactly the difference between great programmers and us, and this is perhaps another explanation for the Dennis Ritchie quotation in Section **??** – by adhering to minimalism, even the ordinary programmer can become a "genius"; because of this very reason, I conclude this part with the following quotation:

> Common sense is instinct, and enough of it is genius.

[50] A similar comparison can perhaps be made between Albert Einstein and John von Neumann, although the judgement on their greatness would be much more controversial.

22 From Lisp to Scheme

After the discussions about minimalism outside programming in last part, now we come back to the subject of programming; but before examining Unix itself, let's first discuss something outside the Unix circle – the Lisp family of programming languageswiki:lisp. Born in 1958, Lisp is one of the oldest programming languages, along with Fortran (1957), Algol (1958) and Cobol (1959), and is perhaps the only one of them still constantly attracting new interest as of now: Fortran finds little (and still diminishing) use outside of numerical computation in science and engineering, few young programmers are interested in maintaining existent (stale and also diminishing) Cobol codebases, and it is fair to say that Algol as a language has been dead for long. So what is the reason for the continued vitality of Lisp? The answer is Lisp's inherent simplicity, and its strong **expressiveness** despite the simplicity; in this section, we will briefly examine the simplicity of Lisp.

Lisp was designed by John McCarthy[51], who showed that a practically usable Turing-complete language can be built with a handful of primitive functions in addition to conditional expressions and recursive function definitionsmccarthy1960. Steve Russell noted, to McCarthy's surprise, that the `eval` function in Lisp could be directly translated into assembly code, and the result was the first Lisp interpreter, running on an IBM 704 mainframe, compiled manually by Russell. Alan Kay, the designer of Smalltalk, made the following commentfeldman2004 about the idea:

> *[...]* that was the big revelation to me when I was in graduate school – when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manualmccarthy1962 was Lisp in itself. These were "Maxwell's Equations of Software!" This is the whole world of programming in a few lines that I can put my hand over.

Although Lisp can be reduced to a minimal core, not all languages in the Lisp family are indeed minimalist: for example, now the unqualified name "Lisp" often means Common Lisp, which is fairly big even in terms of core functionalities; nevertheless, proper abstractions to compress boilerplate code are still strongly encouraged by Common Lisp experts, like Paul Grahamgraham1993. However, there does exist a truly minimalist Lisp, and it is Scheme.

Scheme was not intended to be minimalist from the beginning, according to its designers, Gerald Jay Sussman and Guy L. Steele Jr.sussman1998 (note how similar this is to Unix's conscious pursuit of simplicity after the inception of pipes – *cf.* Section **??**):

> We were actually trying to build something complicated and discovered, serendipitously, that we had accidentally designed something that met all our goals but was much simpler than we had intended. *[...]* we realized that the **λ-calculus** – a small, simple formalism – could serve as the core of a powerful and expressive programming language.

Anyway, minimalism became a defining property of Scheme, as can be seen from the first sentence from the introduction to its specifications, *Revised^3 Report on the Algorithmic Language Scheme* and onward (usually called *R^{3}RS*, *R^{4}RS etc*)scmreports:home[52]:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

This attitude clearly resembles the Unix philosophy, and it makes Scheme the most expressive programming language in my opinion; for this reason, in all following text in this document I will use Scheme to represent Lisp.

The reason for this digression about Lisp is its strong expressiveness despite the simplicity, and this expressiveness largely comes from the way Lisp code is written – **S-expressions** (short for symbolic expressions)[53]; for example, the factorial function can be written in Scheme and Python respectively

[51] By the way, he was also one of the pioneers in time-sharing operating systems (*cf.* Section **??**).

[52] The specifications up to *R^{5}RS* were all a few scores (usually less than 60) pages of text, including examples and rationales; *R^{6}RS* significantly expanded both the core and standard librariesweinholt2019, in order to standardise some (more or less legitimately) demanded features present in mainstream languages. *R^{6}RS* was highly controversial, most importantly because certain features are less useful for educators, hobbyists *etc* who use just a small subset of the functionalitiesscmreports:position; as a result, *R^{7}RS* was intentionally separated into a "small" specification and a "large" one. Incidentally, the specification of Go is also rough 50 pages, but Go is much less expressive than Scheme (*cf.* Section **??** for an example).

[53] In the original paper for Lispmccarthy1960, McCarthy actually intended to use another form called M-expressions (short for meta expressions) for source code, and let the computer translate them to S-expressions; however, programmers at that time generally preferred to use S-expression directly. M-expressions are homoiconic just like S-expressions, and a format closely related to the former is used by Mathematica, the computational software.

as[54]:

```
(define (f i)
  (if (> i 1)
      (* (f (- i 1)) i)
      1))
```

```
def f(i):
  if i > 1:
    return i * f(i - 1)
  return 1
```

S-expressions would probably appear unintuitive to a newbie, but they are in fact much easier for computers to process than most alternative source code formats, and still quite readable to the programmer after a small degree of familiarisation. You may note that this is very similar to why text streams are used as the interface for traditional Unix tools (*cf.* Section **??**), so why are S-expressions more machine-friendly, and what technical advantages do they bring us? Please read the next section.

[54] They are not exactly equivalent: first, Scheme does not have a `return` primitive, and instead uses expressions in **tail positions** as well as **continuations** (*cf.* Footnote **??**) to return values; second, `(define (fn (args ...)) ...)` is a shorthand for `(define fn (lambda (args ...) ...))` in Scheme, while Python's `lambda` is severely limited (*cf.* Section **??** for an example).

23 S-expressions and homoiconicity

To understand the machine-friendliness of S-expressions, we first need to know how source code for a program is represented internally by the computer: whatever language you use, the source code will be converted (**parsed**) into a data structure that represents its syntax elements necessary for the computer to understand the program, and this data structure is called the **abstract syntax tree**wiki:ast. For instance, the AST corresponding to the expression $4 \times 2 - (a + 3)$ and the S-expression equivalent to the latter are shown below:

```
      −
    /   \
   ×      +
  / \    / \
 4   2  a   3
```

```
(-
  (* 4 2)
  (+ a 3))
```

S-expressions can be regarded as representations for nested lists, and as we can see from the example above, nested lists map to tree structures (including ASTs) naturally; additionally, although S-expressions do not eliminate the need for parsing, they are technically much easier to parse than other formats, which makes them a most convenient representation for ASTs. Therefore in combination with the strong capability for list manipulation in Lisp (originally "LISt Processor"), S-expressions enable Lisp to almost effortlessly process Lisp code just like regular data.

When a language can modify source code written in itself just like other data, it is called a **homoiconic** languagewiki:homoiconic; the ability to treat code as data has extremely profound implications, as can be seen from the beginning of the half-joking document *The Gospel of Tux*[55], which is why I consider homoiconicity a notion as profound as complexity:

> In the beginning Turing created the Machine. And the Machine was crufty and bogacious, existing in theory only. And von Neumann looked upon the Machine, and saw that it was crufty. He divided the Machine into two Abstractions, **the Data and the Code, and yet the two were one Architecture**. This is a great Mystery, and the beginning of wisdom.

It was mentioned in Section **??** that all Turing-equivalent models of computation are not equally wieldy for generic discussions, and similarly all homoiconic languages are not equally wieldy for practical "code as data" usage[56]: for example, it might be said that all languages which support text processing are in a sense homoiconic, because the source code is text; however, most **code transformations**, that are interesting to implementers of interpreters and compilers, would be very cumbersome to implement only with text processing mechanisms. For this reason, in all following text in this document, when discussing homoiconicity I will specifically mean it on the **AST level**, or in other words the ability to easily modify the AST of source code just like regular data.

In a language with AST-level homoiconicity, the macro system can be implemented to use functions, that transform ASTs, as macros; with this kind of macros, new syntaxes can be easily defined, and many **language features** that are hardcoded in non-homoiconic languages can be implemented as elegant extensions to the core language. For example, goroutines, which are usually considered one of the main features of Go, are essentially **syntactic sugar** for interfaces to lightweight coroutines in the libthreadlucent2002/libtaskswtch:libtask model[57], and they are hardcoded in Go exactly because new syntaxes cannot be easily added to Go. The example about goroutines might seem slightly abstract because we have not seen code transformations in action, and here I give an example that is both more concrete and more advanced.

Anonymous functions, essentially λ-expressions, are very useful when used as function arguments, as is shown in the Python code below:

```
sorted(l, key = lambda e: e[0])
```

However, the function body of `lambda` in Python must be an expression, so assignment cannot be used, for instance the following function on the left cannot be directly transformed to a `lambda`. But if we regard assignment as a kind of filter that converts a tuple of input values (*eg.* (x, y)) into a modified tuple

[55] Apart from the von Neumann Architecturewiki:neumann, the profundity of the "code is data" idea can be traced to earlier origins, for instance Gödel numberingwiki:godelnum and the halting problemwiki:halt.

[56] Another example is XML, a descendant of SGML; the latter is in turn a document markup language, which makes the XML markup quite verbose. Although XML naturally models a tree structure, its use outside of document markup often results in files with more markups than contents; in addition, the complex specification of XML makes it nontrivial to parse, and therefore unsuitable as source code for programs.

[57] Another model is based on the **continuation-passing style**wiki:cps.

of values (*eg.* (x, y')), the function can be transformed into the composition of multiple `lambda`s, as is shown on the right.

```
def f1(x, y):
  y = y / (1 - x)
  x = x + y
  return x * x + y
```

```
f2 = lambda x, y:
  ((lambda v: v[0] * v[0] + v[1])
  ((lambda v: (v[0] + v[1], v[1]))
  ((x, y / (1 - x)))))
```

The definition of `f2` is evidently ugly, and part of the ugliness lies in the explicit value tuple `v`; another source of ugliness is the reversed order of filters – the last `lambda` to be applied is written first, because it is the outermost. In Scheme, the `let` form has its input values written before its body, so we can use it to compose filters in the desired order, as is shown below on the left.

```
(define f3
  (lambda (x y)
    (let ((y (/ y (- 1 x))))
    (let ((x (+ x y)))
    (+ (* x x) y)))))
```

```
f3 = lambda x, y: \
  (lambda y:
    (lambda x: x * x + y)
    (x + y)) \
  (y / (1 - x))
```

As you might have already guessed, `let` is just syntactic sugar for `lambda`, and `(let ((a` e_1`) (b` e_2`) ...) ...)` is a shorthand for `((lambda (a b ...) ...)` e_1 e_2 `...)`, so `f3` is equivalent to the Python function above on the right. You should have also noticed `v` is absent from `f3`, and this works because arguments of inner `lambda`s (*eg.* the `x` in `lambda x: x * x + y`) shadows outer variables with the same names (*eg.* the argument `x` of `f3`).

```
f4 = lambda x, y: check(
  divide(y, 1 - x),
  (lambda y:
    (lambda x: x * x + y)
    (x + y)))
```

```
check = lambda arg, fn: \
  None if arg == None else fn(arg)

divide = lambda a, b: \
  None if b == 0 else a / b
```

Nevertheless, this is not the end of the story, because $1 - x$ might be zero, and I want the function to return `None` in such cases; but considering that functions are only evaluated when applied to some arguments, `f3` can be transformed into the function above on the left, with ancillary functions on the right. Moreover, I did not tell you x and y might be `None` instead of numbers, but following the line of thought above, we can further transform `f4` into[58]

```
f5 = lambda x, y: check(
  x, (lambda x: check(
    y, (lambda y: check(
      divide(y, 1 - x), (lambda y: check(
        x + y, (lambda x: x * x + y))))))))
```

Using this technique, we actually implemented exceptions in a purely functional way[59]; if you are familiar with Haskell, you might realise this essentially using a monad. A Haskell counterpart of `f5` can be written in the form below on the left, which is equivalent to the form on the right using syntactic sugar hardcoded in Haskell[60].

```
f5 = \x y ->
  (x >>= \x ->
    (y >>= \y ->
      (divide y (1 - x) >>= \y ->
        (Just (x + y) >>= \x ->
          Just (x * x + y)))))
```

[58] Incidentally, this is reminiscent of the continuation-passing style (*cf.* Footnote ??), which is in fact not a coincidencetroelskn2009.

[59] Conversely, we can also see that, as a matter of fact, exceptions have a ground in type theory. Besides, I find it necessary to note that unlike Haskell, Lisp does not pursue purely functional programming, although the functional style is certainly preferred.

[60] `return` in Haskell is just a function, and the two `return`s on the right correspond exactly to the two `Just`s on the left, which are needed due to type-theoretical requirements in Haskell.

```
f5 = \x y -> do
  x <- x
  y <- y
```

```
y <- divide y (1 - x)
x <- return (x + y)
return (x * x + y)
```

So what is the role of Scheme and S-expressions here? As was mentioned just now, the syntactic sugar above is hardcoded in Haskell; in Scheme, it can be implemented as a syntactic extension.

In the end of this section, I find it necessary to note that in the operating system, **directory trees resemble S-expressions** in that they are essentially tree structures; the "everything is a file" principle of Unix (*cf.* Section **??**) is a direct result of this idea, although most important contributors to Unix probably did not realise the similar potential of S-expressions. For example Daniel J. Bernstein noted, in the context of security, that parsing and quoting of textual data significantly complicate software systemsdjb:qmailsec; instead, he extensively used directory trees with very simple formats to configure his software, and this practice is also adopted by some programmers with close relation to his software, as can be seen in most daemontools-ish systems.

The complexity of parsing could have been greatly alleviated had we used homoiconic formats, like S-expressions, when processing data with complicated structures; the implications of homoiconicity in the context of Unix philosophy will be further examined in the rest of this document, and here I instead want to consider a little more about directory trees: given the similarity between directory trees and S-expressions, why don't we simply replace most configuration directories with files containing S-expressions? In my opinion, an S-expression file is indeed easier to work with than a directory tree is, but it is also more rigid and monolithic; in comparison, with a directory tree we can update parts of the configuration atomically, and can assign different permissions to different parts. Both of the latter are much harder to achieve with a single file, which I believe shows the advantage of directory trees as a configuration interface.

24 The New Jersey and MIT / Stanford styles

In last section, we have seen the unrivaled power from homoiconcity, which allows the macro system to easily manipulate ASTs in order to implement new language features; furthermore, because the macro expansion is recursive, macros in later parts of a program can access all language features defined earlier. When writing high-performance interpreters and optimising compilers, homoiconicity enables us to not only write transformations on the processed source code succinctly by using these **structural macros**, but also implement the transformations as multiple simple, clear and reliable passes, similar to traditional Unix tools connected by pipes. The prime example for this **multi-pass processing** scheme[61] is the nanopass frameworkandersen2016[62], which is used by Chez Schemewiki:chez, the prestigious Scheme compiler that became open-source in 2016; Chez Scheme often produces executables that are even faster than workalikes written in C, yet it has a codebase more than one order of magnitude smaller than that of GCC, and self-compiles in seconds while GCC requires thousands of seconds.

As can already be seen above, in comparison with Lisp, C is in many aspects an inferior languagegraham2002, to put it bluntly, and the pros and cons of both will be further examined in the next section; this is a result of effects from multiple historical and methodological factors, and I summarise them as the following. Due to hardware limitations, C basically began as a portable assembly language, reasonably simple to implement with acceptable performance on computers at Bell Labs; due to the success of Unix, continued efforts are made to create stronger optimising C compilers, so it is still one of the most performant languages; and while it certainly has multiple weaknesses in comparison with other languages, people usually either do not find the weaknesses a major drawback, or simply use alternative languages. In simple words, C was chosen by Unix initially because the former was simple to implement and fairly easy to use, and later mainly because of inertia; the first reason highlights some kind of pragmatism present throughout the development of Unix, and a well-known summary of it is Richard P. Gabriel's "worse is better"dreamsongs:wib[63].

[61] Noticing the analogy between directory trees and S-expressions in Section **??**, we can also implement multi-pass preprocessors for configuration directoriesgitea:slewman; furthermore, from the viewpoint of the Prototype Pattern, `fork()/exec()` and Bernstein chainloading can also be regarded as similar designs.

[62] Incidentally, the name of one main author of nanopass, when abbreviated, becomes "AWK".

[63] As you can see from the page, Richard P. Gabriel had still not decided which style was better. Additionally, the complexity issue of interruptible system calls in Unix can be bypassed by userspace wrappers that simply retry until the system calls return without interruption; however, current standards curiously do not require such wrappers, which is one reason why Daniel J. Bernstein created his own interface set (*cf.* Section **??**).

Gabriel compared Lisp with Unix and concluded that, basically, when simplicity of the interface and simplicity of the implementation conflict[64], Lisp would choose the former ("**the right thing**" or the MIT / Stanford style), while Unix would choose the latter ("**worse is better**" or the New Jersey style). Apart from C, another main manifestation of "worse is better" in Unix is the preference of textual interfaces by traditional Unix tools[65] (*cf.* Section **??**); and just like C, textual interfaces are often criticised in the Lisp circle, because plain text is unstructured unlike homoiconic formats (*eg.* S-expressions). However, from the perspective of minimising the system's total complexity which covers both the interface and the implementation, I believe that **the choice between plain text and S-expressions is not a black-or-white problem**: when processing data with simple structures, using plain text is usually better because the interface is still easy enough to use; in contrast, when processing data with complicated structures, the decrease in interfacial complexity of using S-expressions often outweighs the increase in implementational complexity. When dealing with languages, extensive processing of data with deeply nested and often self-referencing structures are necessary, so S-expressions are undoubtedly preferable.

Another common criticism against Unix from the Lisp circle is that the former relentlessly exposes low-level details to the user, and that instead these details should be hidden from the user by means of encapsulation[66]; however, a common problem with encapsulation is **leaky abstractions**, where customisation often has to be achieved by working around the encapsulation. systemd and Linux distributions from Red Hat, examined in the first part of this document, are the more severe examples; actually I believe there are few abstractions that are satisfactorily airtight, and among them are certain well-designed programming languages as well as kernels of Unix-like operating systems. Even when we distinguish between "normal" and "advanced" users, providing encapsulation to the former without interfering with the latter's convenience in customising and debugging does not seem like a solved problem.

25 Benefits of reconciling Unix and Lisp

As was noted in last section, C is an inferior language to Lisp, most importantly due to its lack of expressiveness: for example, had it used a macro system (like that in Dalegithub:dale) that could modify ASTs, then features like object-oriented programming would be implementable using macros, and C++ would perhaps have been unnecessary. Another prominent weakness of C, widely noted both inside and outside the Lisp circle, is its memory unsafety which results in buffer overflows, null-pointer dereferences *etc*; alternative C libraries like skalibs (*cf.* Section **??**) surely help to reduce this kind of bugs, but are far from able to reducing them to the minimum, and I will mention a possible strategy for this in the next section.

However, to be fair, Lisp also has its weaknesses, among which a best-known is the performance cost of dynamic type checking, especially in scenarios like numerical computation; automatic type inference, even in its advanced form in *eg.* Chez Scheme, is not a universal solution to this problem. Another main problem is the necessity of garbage collection in Lisp implementations, which makes compiled executables larger in size and often unsuitable for use in real-time environments; furthermore, advanced requirements for low-level development, for instance cryptography (*cf.* NaCl and BearSSL) which often requires both constant-time execution and high performance (often involving assembly), seem largely ignored in the Lisp circle. To summarise them up, from what I know, I feel that while Lisp is good at implementing complicated application requirements through abstractions, it is less involved in low-level development, where it has big potentials in fact.

Something interesting lies in the observations above – the pros and cons of Lisp and C seem to be complementary; this naturally leads to the guess that if we somehow manage to design **a minimalist homoiconic language that combines their strengths while avoiding their weaknesses**, it would be, in Tony Hoare's wordshoare1981[67],

[64] Richard P. Gabriel later noted in *Worse is Better is Worse* that people often ignore this central premise, and instead just blindly aim to produce software that only implements 50% of the requirements. Another sign of ignoring the premise, I believe, is the objection to refactoring when attempting to fulfill the complete requirements, with "worse is better" as an excusechiusano2014.

[65] In addition to the plain text *vs.* S-expressions debate, there is also the text *vs.* binary debate; in my opinion, from cases like the use of CDB in qmail and s6-rc, we can see that binary files are not always avoided, but only when there are simple ways to use text files without compromising the requirements (*eg.* performance and security).

[66] From this criticism, we can also understand why Lisp focuses more on the interfacial complexity.

[67] A most important lesson we have to learn from the speech, as Tony Hoare put it, is that simplicity must be taken as the first priority when pursuing an ultimate language.

> *[...]* a language to end all languages, designed to meet the needs of all computer applications, both commercial and scientific, *[...]*

But is that achievable? If yes, how to do it, and what would we get? If no, would the attempt be worthwhile? My guess for the first question is "yes", while the second and the fourth will be respectively explored in the next section and Section **??**; in the rest of this section, I will give some examples for the third question.

As was mentioned in Section **??**, Laurent Bercot implemented the unit operations in chainloading as a set of command-line tools; in order to implement unit operations involving two commands (*eg.* loops and pipelines), he designed a special quoting syntax for command line arguments, and wrapped it up as "blocks"ska:elblocks in his language called execline[68]. Nevertheless, chainloading does not necessarily imply `exec()`, and instead can be done in a shell-like languagevector2016a (*cf.* shell commands like `cd` and `umask`); based on the model of scshscsh:home, the execline language can be replaced with scsh-like Scheme, with chainloaders replaced by scsh-like Scheme programs analogous to the following shell reimplementation of the `cd` chainloader provided by execlinevector2018c:

```
#!/bin/rc -e
cd $1; shift; exec $*
```

Following this line of thought, it is not unimaginable to replace traditional Unix tools with Scheme programs as well; considering the expressiveness of Scheme and the existence of elegant Scheme compilers like Chez Scheme, this would surely help to reduce the total complexity of the system. So now we see that by migrating to Lisp for system programming, even these simple tools can be further simplified; and as was noted before, Lisp is probably even more powerful for application programming, so I find it undoubted that **by embracing homoiconicity, we will be able to build systems that are more Unix-ish than was possible before**.

Now that we have seen the power of homoiconicity in system programming, it is appropriate to revisit the Trusting Trust issue in Section **??**; at that time I noted that one method against Trusting Trust is to construct the compiler from the machine code in multiple steps, and it should be self-evident that the auditability of this procedure directly depends on the latter's total complexity. It was also noted in Section **??** that Steve Russell implemented the first Lisp interpreter in assembly by hand, which in combination with the elegance of Chez Scheme tells us that **homoiconicity will dramatically reduce the total complexity of bootstrapping from machine code**, and therefore strengthen our defence against Trusting Trust[69].

26 Lisp / C reconciliation: how to?

As was emphasised in Section **??**, we should analyse specific issues specifically, and while the disappearance of resource limitations does not imply simplicity was no longer important, it surely motivates us to rethink previous compromises made due to these limitations. We are well past the era where garbage collection was prohibitively expensive for regular computers, and now for devices with severely limited resource we generally prefer cross-compilation over native compilation. Given this observation while considering the issue of big executables and real-time requirements in last section, the proposed language (I give it the codename "Nil", for uNIx plus Lisp) should be able to use GC for its interpreter and compiler, but **executables produced by the compiler might need to avoid using GC whenever reasonable**[70].

In last section, we revisited the issue of bootstrapping, and I hinted about a procedure where a minimal interpreter is somehow built, and then in a later step a production-ready compiler is produced; a first point to note is that this deviates from today's common practice of using a compiler to build an interpreter, and instead constructs an interpreter first. However, as can be guessed, Nil interpreters would probably be easier to write in assembly than Nil compilers are, just as with Lisp which Nil is

[68] Its design is similar to the initial shell in Unix, the Thompson shellwiki:thompson, which was substituted by the Bourne shell in Unix v7 due to its inadequacy for more complex programming.

[69] Another potential problem is hardware backdoors, and I think a way to deter them is to leave adequate room for variation in every step of the bootstrapping procedure: because the semantics of low-level code is difficult to analyse (which results in the importance of open source), any attempt at mechanically injecting malicious code may trigger false positives, which may expose the backdoor.

[70] And the programmer should be aware of how to avoid GC when implementing special requirements, or alternatively we can perhaps consider (optional and configurable) real-time GCbernstein2013. Furtherly, considering the inherent pros and cons of different methods for memory management (GC, RAII, manual management *etc*), we probably need a minimal mechanism where multiple memory-management methods can be used at the same time without conflicts.

modeled upon, so from the viewpoint of complexity it would be better to generate the compiler from a primitive interpreter[71]. Similarly, noticing the expressiveness of Nil, it would be natural to use a self-bootstrappable Nil compiler as the base compiler in production systems, from which other compilers (if still necessary, and perhaps including one for C) are produced.

Hitherto in this section, we have been examining the Lisp-like aspects of Nil, but what about the C-like aspects? I consider Dale, mentioned in last section, a possible prototype for a C-like layer, upon which a Lisp-like layer would be implemented using macros, and many other languages (*eg.* shell and Makefile) would be emulated in turn on the Lisp-like layer, accessible to the user via something like LaTeX's **macro packages**. Another approach I have thought of, which I give the codename Nil-powered assembly, is based on the notion of a portable assembly[72] (*cf.* Section **??**): the C-like layer would be implemented based on the Lisp-like layer as most other languages would be, and it would be a markup-like "reverse-macro" system which provides primitive procedures that when called emit assembly instructions to the output.

In addition, as was noted in last section, Nil needs a type system stronger than that of Lisp, for which I find Yin Wang's design base on **contracts**wangyin2013a, wangyin2013b a very thought-provoking reference; it is worth noting that this design helps to boost memory safety of the C-like layer, and even facilitates the formal verification of written programs (*cf.* Footnote **??**). The foreign function interface between the Lisp-like and C-like layers will also be an issue, which I am not yet knowledgeable enough to comment on. Finally, before ending this section, I need to stress the non-technical issue of inertia, which troubles Scheme (*cf.* Footnote **??**), Plan 9 (*cf.* Footnote **??**) as well as qmail and other excellent (or not) software projects: apart from the technical challenges Nil will inevitably encounter, it will also have to attract some attention from both academia and industry to gain enough momentum (there is surely the possibility that the latter simply does not care, but see the next section); and even if Nil might have some success, getting rid of historical burdens while adapting to new requirements[73] would not be easy for it, as is for other projects.

27 Toward a "theory of everything"

As was noted in last section and Section **??**, Nil might be a language that is impossible even in the theoretical sense, or might not attract adequate attention from the industry, so would the exploration on Nil still be worthwhile in such cases? In this final section, I will give my answer to this question based on the analogy I made in vector2018c:

> I guess reconciling Lisp and Unix would be much easier than reconciling quantum mechanics and general relativity; and it would be, in a perhaps exaggerated sense, as meaningful.

Theoretical physicists and high-energy physicists have long pursued a theory of everything that can reconcile quantum field theory and general relativity, because the theory would provide a unified basis for all low-level physical phenomena. To my knowledge, there does not seem to be a firm guarantee such a theory can be successfully achieved by humans: for example, the best-known candidate, string theory, still appears far from testable, even though it does agree with observed facts. On the other hand, there is not a proof that such a theory is unachievable, either; and apart from the theoretical promises, research on it has always been giving rise to exciting advancement in diverse fields ranging from pure mathematics (profoundly connected to theoretical physics) to civil engineering (involved in the construction of large facilities for physical experiments). For these two reasons, a ToE is still the lifelong pursuit to many physicists.

Similarly, although the efforts at Lisp / C reconciliation are not guaranteed to really result in an ultimate language, it would surely be rewarding to explore **the best way the two languages can collaborate**, so that the system's complexity is minimised and the programmer's productivity is maximised: as has already been seen in Section **??**, the reconciliation, even in a very primitive form, can already contribute greatly to the reduction in the complexity of Unix systems; I believe that further simplification

[71] Probably in multiple steps, involving more and more powerful compilers, interpreters and even assemblers, and the assemblers would probably somehow resemble the proposed Nil-powered assembly; to facilitate auditing, a smaller number of intermediate steps should probably be preferred, as long as the total size of the codebase involved is close to the minimum. Besides, on a more theoretical level, I find the notion of Futamura projections (essentially **partial evaluation**)wiki:parteval highly interesting.

[72] Since the C-like layer would essentially be a machine code generator, similar generators like Chez Scheme's built-in assembler and Daniel J. Bernstein's qhasm might be highly instructive references.

[73] For programming languages, one of its most important aspects is the (careful) standardisation of additional language features and libraries, which is in my opinion done terribly by the Scheme community.

of these systems will attract more people to explore the fundamental aspects of computer systems, just like how the Raspberry Pi reignited the enthusiasm for low-level development. And even if our efforts might not result in production-ready Nil-based systems finally, I am completely convinced that insightful communication between the Unix and Lisp circles will, as was explained in Section **??**, definitely result in a lot of **byproducts that are both interesting and meaningful**, therefore I conclude this document with two quotations respectively from Rob Pikepike2000[74] and Hal Abelsonabelson2008:

> Narrowness of experience leads to narrowness of imagination.

> If you *[carefully study the example interpreters in this book]*, you will change your view of your programming, and your view of yourself as a programmer. You'll come to see yourself as a designer of languages rather than only a user of languages, as a person who chooses the rules by which languages are put together, rather than only a follower of rules that other people have chosen.

[74] The description is, ironically, applicable to himself as well regarding programming languages.

[heading = bibintoc, title = References]

Afterword

Soon after beginning to learn Linux, I bought the book *Classic Shell Scripting*, which initiated my first enthusiastic experience in formal programming. My interest was exactly because of the word frequency program mentioned in Section **??**, used by the book as an example for the power of the Unix philosophy. (Note how this is similar to V. I. Arnold's reaction to L. A. Tumarkin's teaching practice – *cf.* Section **??**.) Although the Unix philosophy (under the name "Software Tools philosophy") was followed throughout the book, I did not intentionally follow the philosophy until I attended the *Linux Programming Environment* course in Autumn 2009 by Mr. Dong-Gang Cao at my university, where the notion was formally given and emphasised.

My life with Linux was relatively peaceful, until systemd came into the picture: in 2011 and 2012, the developers of systemd pushed for the addition of logind as a dependency of GNOME 3poettering2011a, and merged udev into systemdsievers2012; vehement controversies and flamewars beginning from then only began to wane in the last two yearsslashdot:systemd, and it is my explicit wish that this document contributes to the demise of systemd and the proprietary practices associated with it. "Every cloud has a silver lining": in the search for a systemd substitute I found the s6 project in Summer 2014, and studying the design and implementation of software in the style by Daniel J. Bernstein proved to be an invaluable practice in application of the Unix philosophy. During the process, the questions "is the Unix philosophy now outdated?" and "what is the social value of the Unix philosophy?" became recurring issues in my mind, which resulted in the first two main parts of this document.

In Spring 2018, I envisioned a somehow C-like language based on S-expressions when considering the inexpressiveness of Cvector2018b, roughly one year after stumbling upon scshangelbeats2018, although I became aware of the elegance of Lisp perhaps as early as 2012, mainly from the writings by Yin Wang (while he is highly controversial in the Chinese open-source community, he often makes very interesting points). The inspiration for said C-like language can actually be traced earlier, when I began thinking about the possibility of a shell-like language that also supports Bernstein chainloadingvector2016a in the beginning of 2016. I finally came up with a practical minimalist plan for such a languagevector2018c in Spring 2018, and meanwhile came across the idea of "Lisp / C reconciliation", which grew into the third part of this document through a discussion threadstevel2018 in the Gentoo forums.

Before ending this document, I would like to thank the Linux Club of Peking University and the skarnet software community, without which I would probably be yet another highly amateurish dabbler in Linux, and this document would have never come into existence. I would also like to thank the general open-source community, including the Unix circle, the Lisp circle and more, encompassing people pro-systemd and anti-systemd, for providing rich and diverse technical materials and opinions, which is a priceless offering. Inspired by the musical settings in *Thunderstorm* by Yu Cao, and the appendices to *The TEXbook* and *The METAFONTbook* by Donald Knuth, the structure of this document was modeled after the *Mass in B Minor* by J. S. Bach.