ACHIEVING SELF-SUSTAINABILITY IN INTERACTIVE GRAPHICAL PROGRAMMING SYSTEMS

JOEL JAKUBOVIC

A Dissertation Submitted in Fulfillment of the Requirements for the Degree of
Doctor of Philosophy (PhD)
in Computer Science

to the School of Computing
Division of Computing, Engineering and Mathematical Sciences
University of Kent

Submitted June 2023

I dedicate this work to my parents Rachel and David and my sister Ruth, for supporting my quest for knowledge wherever it may lead.

ABSTRACT

Programming is fraught with accidental complexity. Software, including tools used for programming, is inflexible and hard to adapt to one's specific problem context. Programming tools do not support *Notational Freedom,* so programmers must waste cognitive effort expressing ideas in suboptimal notations. They must also work around problems caused by a reliance on plain text representations instead of *Explicit Structure.*

The idea of a *Self-Sustainable* programming system, open to adaptation by its users, promises a way out of these accidental complexities. However, the principles underlying such a property are poorly documented, as are methods for practically achieving it in harmony with Notational Freedom and Explicit Structure. We trace the causes of this difficulty and use them to inform our construction of a prototype self-sustainable system. By carefully reflecting on the steps involved in our specific case, we provide insight into how self-sustainability can be achieved in general, and thus how a motivated programmer can escape the aforementioned sources of accidental complexity.

PUBLICATIONS

Some ideas presented in this dissertation have appeared previously in the following publications:

Jakubovic, Joel (Dec. 2020a). "Programming "systems" deserve a theory too!" In: *Psychology of Programming Interest Group*.

– (2020b). "What It Takes to Create with Domain-Appropriate Tools. Reflections on implementing the "Id" system." In: *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. Programming '20. Porto, Portugal: Association for Computing Machinery, pp. 197–207. ISBN: 9781450375078. DOI: 10.1145/3397537.3397549.

Jakubovic, Joel, Jonathan Edwards, and Tomas Petricek (Feb. 2023). "Technical Dimensions of Programming Systems." In: *The Art, Science, and Engineering of Programming* 7.3. DOI: 10.22152/programming-journal.org/2023/7/13.

Jakubovic, Joel and Tomas Petricek (2022). "Ascending the Ladder to Self-Sustainability: Achieving Open Evolution in an Interactive Graphical System." In: *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2022. Auckland, New Zealand: Association for Computing Machinery, pp. 240–258. ISBN: 9781450399098. DOI: 10.1145/3563835.3568736.

Petricek, Tomas and Joel Jakubovic (2021). "Complementary science of interactive programming systems." In: *History and Philosophy of Computing*.

ACKNOWLEDGMENTS

---

1 https://www.hillside.net/plop/2021/index.php?nav=PLoP21
2 https://2021.programming-conference.org/track/programming-2021-conversation-starters
3 https://2022.programming-conference.org/home/mops-2022
4 https://futureofcoding.org/
5 https://vpri.org/
6 https://www.cemetech.net/forum/viewtopic.php?p=270092#270092

At this point, my mind is inexorably drawn to a crossed-out name in one of the dissertations in the School's collection. I shall restore balance to the cosmos by expressing my appreciation for Daria's love and support during my research, and for our continued friendship since going our separate ways. Let me also thank my family, office colleagues and anyone who has slipped my mind for their company along this most challenging of my accomplishments.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

DOM   Document Object Model

COLA   Combined Object Lambda Architecture

JS   JavaScript

DSL   Domain-Specific Language

MSL   Mood-Specific Language

GUI   Graphical User Interface

IDE   Integrated Development Environment

OS   Operating System

REPL   Read-Eval-Print Loop

RISC   Reduced Instruction Set Computing

CISC   Complex Instruction Set Computing

BL-ASM   BootstrapLab Assembler

INTRODUCTION

1

When we have an idea for some computer software, and try to make this idea a reality, we are forced to confront two types of complexity: the *essential* and the *accidental*. We know there is "no such thing as a free lunch", so we are able to accept the burden of whatever complexity is actually intrinsic to our idea. If we have a simple idea, we are prepared to do a little work; if it is more ambitious, we will accept having to do more work. Unfortunately, this *essential* complexity is often swamped by unwelcome incursions of tedious busy-work. Concepts that appear simple must be spelled out in great detail for a computer. This is the *accidental complexity* that is widespread in programming (Brooks 1978).

This is particularly egregious when the "idea" is merely to change or fix some small issue. Suppose we are using an app where the text is hard to read owing to a similar background colour. The designers have not included a feature for changing the colour of UI elements. A programmer would know that there must be some API being called to render the background. This API will receive the colour from a few numbers in the app's memory. If only we could find these numbers and change them, we would be able to read the text.

What, therefore, does it take to find and change a colour? The app itself provides no way to proceed through its surface interface to its internal mechanisms. Thus, the accidental complexity we must face includes working with some external tool that can open it up. We could attach an assembly-level debugger to the app process and stare at hex dumps for a long time, eventually figuring out which address holds the colour. Such an expert task would take an extremely long time even for someone with the relevant experience. It would only let us make a change to the *running* app; we would have to repeat the procedure every time we ran the app.

Alternatively, we could hope that the app is open-source, download the code, setup the build system, locate the relevant code, re-build the app, and re-install it. Each of these steps is also an expert task which would be incredibly lengthy on a novel codebase, even for experienced programmers. Furthermore, this approach entails destroying the running instance of the app and re-initialising it, possibly losing unsaved work.

In the worst case, both of these approaches could be blocked; runtime tampering could be prevented by the security policies of mobile devices, while re-building from source cannot work without access *to* that source. Suffice to say, none of this is suitable for an average user. Even a seasoned programmer would consider it not worth the trouble.

Our task of changing a colour, while technically possible, has a severely *disproportionate* accidental complexity cost.

In specific situations, software authors do have good reasons to restrict access to internals. In a game, it is important to enforce the rules; access to internals would enable arbitrary cheating. However, this is a *special case* not representative of most types of software. Despite this, we are unable to simply *choose* to build software that is "open". Even if *we* wrote the app and desire to support adaptation beyond what we anticipated, we face the fact that our tools can only create software that is "closed". The task of "supporting unanticipated modification" is itself a *feature* that we must somehow figure out and implement on top, and it is unclear how to achieve such a feature. Nevertheless, it is worth striving for a world where this accidental complexity is as reduced as possible. We might expect this to involve a mix of "demolition" work—that of removing barriers that have been placed in the way—and "construction" work of building tools that help us work more effectively.

1.1 A MORE COMPELLING EXAMPLE

The reason we have considered changing a colour is that this is an extremely small change that nevertheless has a plausible user story behind it (as opposed to, say, flipping a single bit in the app's memory just to see what happens). What we have established is that *even a tiny change* involves egregious accidental complexity, which does not provide much hope for more practical changes we might actually want to make. What is an example of a more realistic adaptation of one's software? There are many ways an end-user might want to adapt some software they are using, but a clearer example is a frequent activity of programmers: debugging.

The chief way in which programmers "adapt" existing software is by fixing bugs. This involves two stages: discovering the cause of the bug, and applying the fix. In order to discover the cause, it is helpful to change values in the program while it is running. In the default state of a program, however, this is not possible. Instead, the programmer must return to the source code and make modifications there (either to change values directly, or by writing code do so in response to specific run-time conditions) and add logging commands so they may see diagnostics at run time.

Alternatively, the programmer may run the program in tandem with a separate tool, a dedicated *debugger,* which fills in for this "missing" functionality of viewing (and possibly changing) run-time values as well as slowly stepping through the code as it executes. However, not even this will help with the second half of the job, that of changing the program to fix the bug. Realistically, the debugger only permits changing run-time *data* values, but let us grant for the sake of argument that it can also edit code. Even in this case, the changes will be limited to

the *running instance* of the program, and will be gone when it terminates. In order to fix the bug in a lasting way, the programmer must once again leave the *program* (the thing ostensibly being fixed) and go *outside* to the separate source code, and make the change there. In sum, this is a large amount of accidental complexity required for diagnosing and fixing a bug.

If only there were some way to have the program supply its *own* debugger, and provide access to its internal data and code through its running "surface", and finally persist any changes made that way, then the actual complexity of the enterprise could be brought closer to the essential complexity of the bug itself.

1.2 HOW SHOULD THINGS WORK?

Imagine a world where the average computer user can patch or improve their software the same way they might change a lightbulb or perform DIY in their home. This clearly relies on the ability to make small *piecemeal* changes to their home, without having to demolish the place and re-build it anew. We will call this *naïve pokeability*:[1]

Definition 1 (Naïve pokeability)**.** A change has *naïve pokeability* if it is possible to make the change while the software is *running* without having to consider the implications of restarting it.

Furthermore, the common-sense expectation is that the changes *persist* into the future:

Definition 2 (Persistent)**.** The result of a change is *persistent* if it remains until a future change overrides its effect.

Definition 3 (Transient)**.** A change is *transient* if, in the absence of special measures, it will be undone within a short timeframe and its effects will not last.

Additionally, the tools for making changes are of an appropriate scale and reside within the system. Changes are *self-supplied* and, if this covers all possible changes, we have a *self-sustainable* environment.

Definition 4 (Self-supplied)**.** A change is *self-supplied* by a piece of software if you can achieve the change by only using the software.

Definition 5 (Self-sustainable)**.** A software system is *self-sustainable* if arbitrary changes to it are self-supplied.

The ideal system functions like a workshop where new tools can be fashioned using existing tools as needed. They can be big or small, and this ensures that we can use the "right tool for the job", no matter the scale.

---

[1] By analogy to the "Naïve Realism" principle found in Boxer (diSessa 1985).

Definition 6 (Right Tool For The Job)**.** This is a principle in programming which acknowledges that tools have differing strengths and weaknesses for different tasks. To "Use The Right Tool For The Job" is an ideal that relies on either an *existing* range from which to select the best tool, or the capacity to design and build it on-demand.

Definition 7 (One-Size-Fits-All)**.** The opposite of "Use The Right Tool For The Job". This refers to using a single tool to do a wide range of tasks even though it may not be suited for some of them.

Definition 8 (Domain-Specific Adaptation)**.** This is a small or large part of a software system which provides its own custom interface for change.

In standard practice, a program is generated from *source code* and put into a running state. To change the program, one must change the source code, destroy the program and re-create it anew. These steps are accomplished with separate tools, meaning that changes tend not to be self-supplied (Definition 4).

There is a limited notion of "Use The Right Tool For The Job" in that there are different programming languages. However, languages tend to enforce their syntax and semantics without permitting *smaller-scale* adaptation, and variation in these respects is normally restricted to textual notations. Furthermore, some situations may call for more general *notations* or graphical interfaces that do not work like a language, but the fact that programming is optimised for languages makes using such notations more difficult.

Instead of the above, computer software should act as "Personal Dynamic Media" (Kay and Goldberg 1977). In this vision, a software system is *designed* to be adapted and modified by its users. By performing an explicit action (e. g. switching to "edit mode") the user can inspect the visible surface of the application to find the causes of its appearance in the form of code and data. They can also inspect a map of the non-visible implementation of the software's functionality and navigate to the relevant parts. There may be a common programming notation as a default, but where possible, parts of the implementation are presented in local notations or interfaces that are more easily understood. These interfaces can also be traced to *their* implementations and modified if desired. The user can then change any aspect of the software while it is running, without having to edit an external specification and destroy the running instance.

1.3   A FRAGMENTED VISION

Several pieces of this vision do exist, but not in an integrated whole. We can see some of the different characteristics we desire in software by examining the Web browser, HyperCard, and Smalltalk.
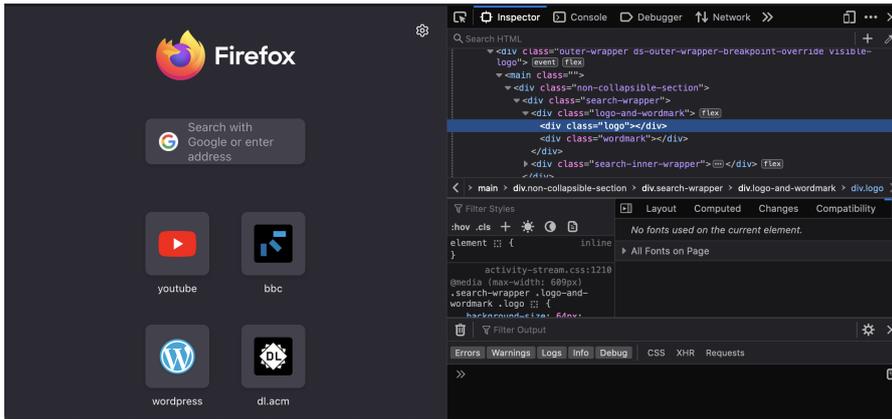
Figure 1.1: The Developer Tools available for any Web page or app in a modern browser.

1.3.1 *Web pages, web apps, and browsers*

The *web browser* has a powerful set of *developer tools* (Figure 1.1). This includes the "inspector" which can be used to edit the page's underlying elements in the Document Object Model (DOM). For example, an ad can be removed by locating and deleting its element. Some of this underlying "state" may have no visual effects (e. g. an element's ID attribute) and is thus hidden from an ordinary user. However, such state *is* visible from the inspector in the developer tools. This means that all of the "structural" state of a page is *potentially* visible, not to mention editable, in the web browser.

The above is worth contrasting with the case of the "behavioural" side of a web page oriented around the JavaScript (JS) programming language. Alongside the "structural" state of the web page, there is also the hidden state of JS objects. The JS console accepts commands which may read or change this state, but there is nothing like the element inspector for it.[2] What *is* visible in the dev tools is the *source code* of the scripts loaded by the page.

Many changes to the "behavioural" state can be accomplished in the console; for example, updating part of the state to a new object. However, this cannot be relied upon in the same way as DOM editing because JS language features prohibit many changes. For example, a variable declared in the source as const will not be changeable in the console. Moreover, fine-grained changes to code cannot be performed there either. The main unit of code organisation, the function, is an opaque object in the run-time environment; one cannot simply replace a particular line or expression within it. Instead, a complete new definition must be entered into the console to replace it wholesale. Yet even this will fail if the source declares the function const. In such cases, we

---

2 This may be because the DOM is a tree structure while the JS state is a general graph, and it's harder to build an editor for the latter.

lack *naïve pokeability* (Definition 1) and we have no choice but to edit the source files somehow. If the browser does not provide for local edits to be made to these files, a separate text editor must be used. The changes made in this way will only take effect once the scripts are reloaded by refreshing the page and losing its JS state.

Let us return to the "structural" state of the web page and note that we are free to make arbitrarily fine-grained changes using the element inspector. These changes take effect immediately without having to reset anything. There are, in fact, HTML text files backing the page structure, but they cannot restrict the inspector tool in the way JS files restrict the console. In short, the *state* of a web page has naïve pokeability while its dynamic *behaviour* does not reliably have this property.

There is one caveat: the HTML and JS files are the "ground truth", so any changes made via the inspector or the console will disappear when the page is closed or refreshed. Changes made in the browser are *transient* (Definition 3); only changes to the underlying files are *persistent* (Definition 2), and websites typically do not allow unknown individuals to change the files on their servers. All this is sad news for our user deleting their ad, as they will have to repeat it each time they access the page (or more realistically, use sophisticated programmatic middleware like an ad-blocking extension to make this automatic).

1.3.2   *HyperCard*

Before the Web, "hypertext" was regularly created and distributed by people in the form of HyperCard stacks. Alan Kay criticised the web for having a browser that neglects to include an *authoring* tool, immediately limiting the *creation* of web pages to people who could code in a text editor. By contrast, in HyperCard, the viewer and editor exist integrated together (Figure 1.2). Furthermore, there is an "edit" mode whereby a user can remix content from someone else, even re-programming the dynamic behaviour.

These aspects of HyperCard's design encouraged a community of producer-consumers for hypertext content. The web's higher cost of authoring led to a lower producer-to-consumer ratio, restricting the kind of medium that it would become. Note that the naïve pokeability of the element inspector does not amount to *authoring* a web page; such an interface is designed for fine-grained *change* rather than coarse-grained creation. It is also oriented towards programmers, being part of the "developer tools", compared to HyperCard's presentation of authoring as a primary use of the software.

1.3.3   *Smalltalk and COLA*

Smalltalk provides for behaviour editing at a finer granularity than the Web developer tools. Behaviour is separated first by class and then by
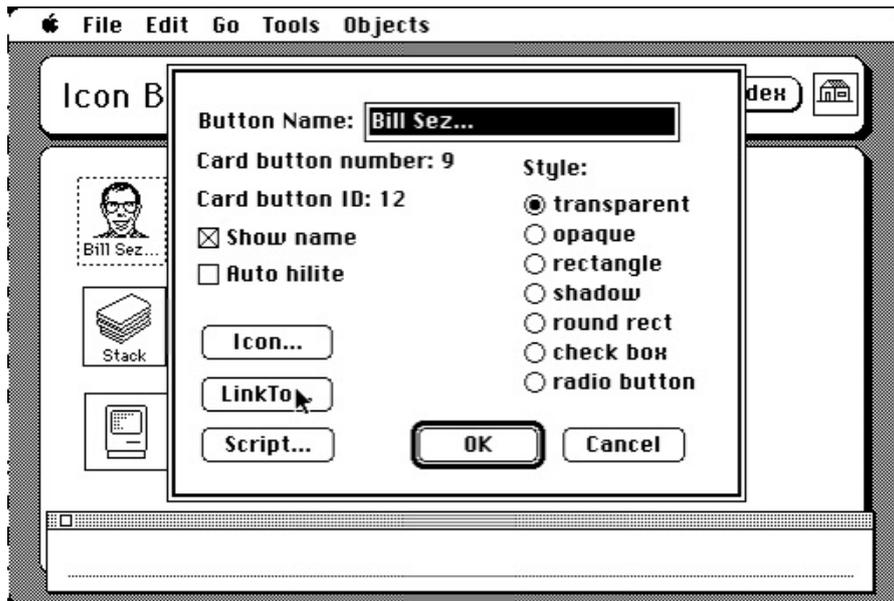
Figure 1.2: HyperCard, a pre-Web Hypertext system, included a direct authoring tool to complement the browser.

method; only then is a text editor presented for the code (Figures 1.3 and 1.4). More importantly, changes to this code take effect once committed, with no "restarting" of the system taking place. The state of the system is persisted by default to an "image" file. In short, Smalltalk provides persistent naïve pokeability for both code and data, and pioneers a large degree of *Self-Sustainability* (Definition 5).

That being said, Smalltalk systems tend to run on VMs that are implemented in a separate lower-level language like C++. Fundamental infrastructure such as object layout and memory management is available only as opaque primitives from the point of view of Smalltalk. Thus, to change these aspects one must still switch to a different programming system and re-compile.

The Combined Object Lambda Architecture or COLA (Piumarta 2006) seeks to take self-sustainability to the extreme, past the degree found in Smalltalk. COLA's minimal design makes said basic infrastructure self-supplied (Definition 4) so as to approximate a truly self-sustainable system. It is also designed to encourage domain-specific adaptations (Definition 8) down to a small scale of "Mood-Specific Languages" (MSLs) beyond the coarse-grained variation found with ordinary programming languages. However, the architecture as described does not have much to say about the user interface or graphics, taking place instead in the world of batch-mode transformations of streams.
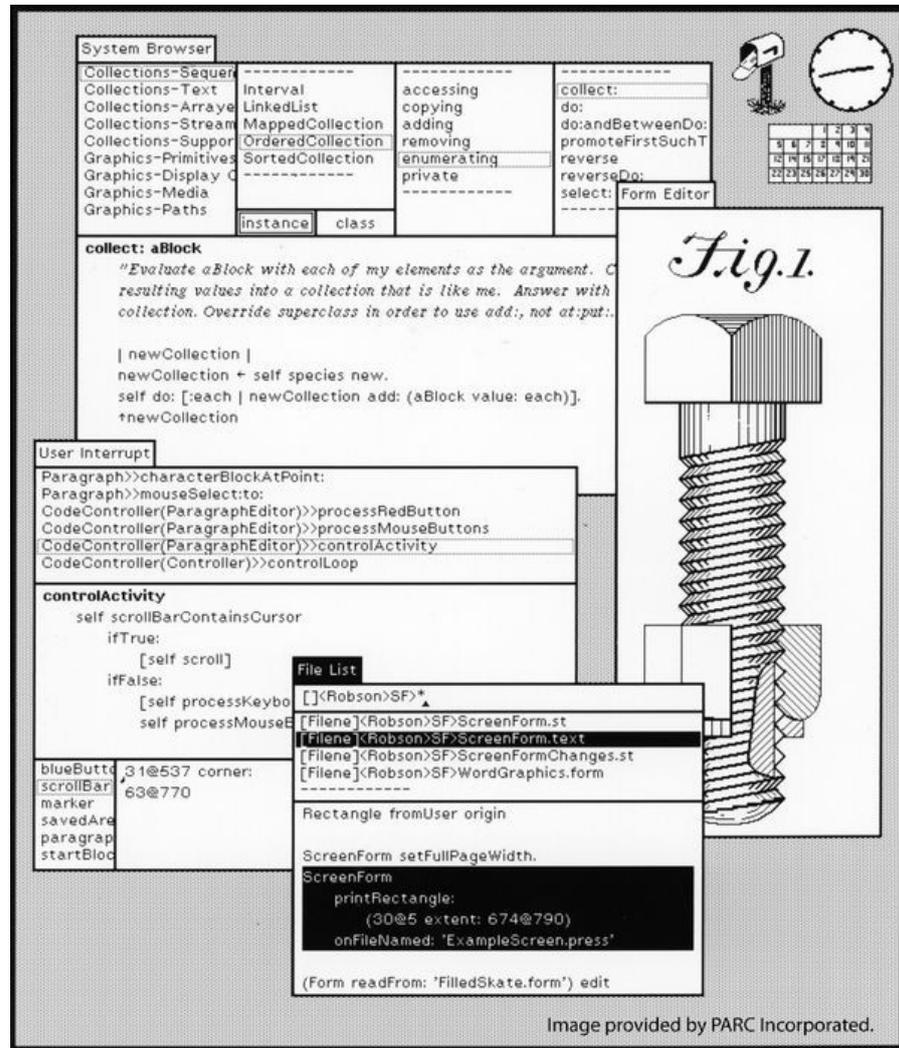
Figure 1.3: The interface of Smalltalk-80, which introduced the *class browser* visible at the top.
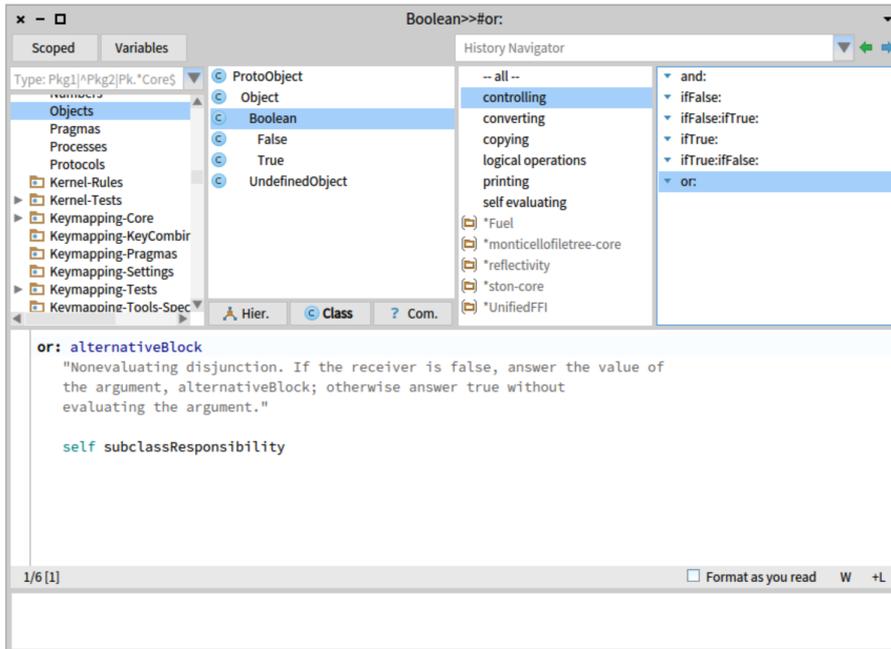
Figure 1.4: The class browser in the modern *Pharo* distribution of Smalltalk.

1.4 ACCIDENTAL COMPLEXITY BEYOND LANGUAGES

In our experience, the three most important sources of accidental complexity in programming are as follows:

1. In order to make even a small change to a program, we must go to the source code which may require an entirely different language and way of thinking. We lack *Self-Sustainability.*

2. We must describe graphical constructs with language in order to fit them into program code. This represents a lack of what we will soon define as *Notational Freedom.*

3. We have to avoid syntax errors, escape certain characters, and write code for parsing and serialising. This represents a lack of what we will soon define as *Explicit Structure.*

These are not quite observations about programming *languages.* Instead, they concern the wider environment of tools in which programming is performed, such as the editor interface and facilities for running the programs.

It is important not to conflate "coding" in a programming language with programming itself. In this dissertation, we see *programming* as the general act of making a computer do things by itself. By this definition, coding, visual programming, programming by example, and deep learning are some specific *means* by which to program. If we ignore this subtlety, we risk unwittingly limiting the scope of innovative ideas in the following ways:

Figure 1.5: A spreadsheet contains text, but is not a *syntax* or a *language*; the grid lines are intrinsically graphical.

- Instead of seeking the right notation, interface or representation for the job, we might seek the right *textual syntax* for the job. If we cannot find one, the real reason may simply be that text is not well-suited to the job (Figure 1.5). Yet if text is all we know, we will be under the false impression that it is an *intrinsically* hard job.

- Instead of being able to make changes to a *running* program, we are stuck changing its source code and re-creating the program. It is easy to make "closed" programs this way and hard to make programs open to "re-programming" while running.

- Instead of seeking a software *system* open to unanticipated changes as it runs, we might seek intricate *language* features that give flexibility only for *compiling* a program.

A key problem is that there is no established term for this scope of programming research, and hence no body of work in which we may situate it. This is the crux of the matter: we need a more general programming *systems* approach instead. We will discuss this further in Chapter 3 and use it in Chapter 5 to propose a systematic framework by which to analyse programming systems. This framework will include three properties that are central to the dissertation and develop them in detail. We will now familiarise the reader with the basic outline of these three properties.

1.5 THE THREE PROPERTIES

The goal at the end of Section 1.2 is much too ambitious a scope to achieve in this dissertation. However, from Definitions 1–8 and the above discussion, we distill three properties that help address the issues we identified. They are:

1. *Self-sustainability:* being able to evolve and re-program a system, using itself, while it is running. (This is a more intuitive definition that agrees with what we said earlier in Definition 5.)

2. *Notational Freedom:* being free to use any notation as desired to create any part of a program, at no additional cost beyond that required to implement the notation itself.

3. *Explicit Structure:* being able to work with data structures directly, unencumbered by the complexities of parsing and serialising strings.

1.5.1 *Importance of the Three Properties*

We are interested in exploring, developing, and achieving the Three Properties in programming systems. We will refine and expand these definitions in later chapters, but they are reasonable to start with. Each one brings its own advantages to a programming system:

1. Self-Sustainability reduces the accidental complexity of having to make changes using a separate, unfamiliar programming system. It also permits *innovation feedback:* anything helpful created using the system can benefit not only other programs sitting atop the system, but also the system's own development.

2. Notational Freedom makes it easier to use the "Right Tool For The Job" (Definition 6). Once a programmer has decided what the right tool is in their specific context, Notational Freedom means they can use such a tool more easily as a Domain-Specifc Adaptation (Definition 8). For example, if diagrams are desired, Notational Freedom removes the traditional limitation to use ASCII art. More generally, Notational Freedom removes the need to describe graphical constructs using language.

3. Explicit Structure avoids various pitfalls of strings, both in terms of correctness and convenience. Consumers of a structure benefit from an editor that can only save valid structures, and producers benefit by discovering errors early instead of later during consumption. Writing programs to use such structures is improved if one does not have to maintain code for parsing and serialising or think about escaping special characters.

These properties are exhibited occasionally in different systems, as we will mention in Chapters 2 and 3. However, it is rare to see two or all three present in the same system. This rarity suggests they are probably under-explored and under-developed, so we could stand to learn a lot by studying them. We do not doubt that these properties have drawbacks in addition to the above advantages, but we stand to gain from these advantages taking us closer to the ideal at the end of Section 1.2.

1.5.2   *The Three Properties in Combination*

It is worth exploring the Three Properties in *combination* because they complement each other in the following ways.

Suppose a system already has Notational Freedom. Self-Sustainability makes it easier to add new notations to it. In the converse case of a system lacking Notational Freedom, Self-Sustainability makes it easier to add Notational Freedom *itself* and lets the benefits flow into all aspects of the system's development; this is what we called *innovation feedback*.

Notational Freedom is impossible to achieve without Explicit Structure. In a world of parsed strings and text editors, we are limited to what we will term *syntactic* freedom in Section 3.3.2. Thus, Notational Freedom needs Explicit Structure as a necessary foundation.

Self-Sustainability also suffers without Explicit Structure. Self-Sustainability is vaguely understood by analogy to self-hosting compilers, as we will see in Section 2.3.1. The COLA work (Piumarta 2006) follows this approach, remaining unclear on how such a property can be achieved in interactive, graphical systems. Explicit Structure lets us study the other two Properties more purely, without getting confused by the accidental complexities of parsing and escaping (we will expand on this in Section 3.3.3.3).

We can prioritise the Three Properties based on the above interdependencies. Our primary goal is to explore Notational Freedom in interactive, graphical programming systems. To support this, we should achieve Self-Sustainability. To do both of these with minimal distraction, we should make sure to build on a foundation of Explicit Structure. We will not follow this order strictly, but it shows a logic as to how each property fits into the bigger picture. We see that the only way to discover how to achieve these goals is by *doing,* so we work to build a prototype programming system called *BootstrapLab* that makes progress on the Three Properties simultaneously.

1.6   THESIS STATEMENT AND CONTRIBUTIONS

The statement of our thesis is as follows:

> It is possible to add Notational Freedom to a given existing programming system, by embedding a Self-Sustainable system built on Explicit Structure.

We prove this by construction in the form of a prototype programming system called *BootstrapLab*, which is the topic of Chapter 4. Our main contribution is not BootstrapLab itself, but rather the necessary steps and principles that its construction led us to *discover.* We believe that it should be possible to build these Three Properties atop a wide variety of programming systems; the goal of Chapter 4 is to document

enough of a generalisable technique to make this feasible for the average programmer. It is as if we have developed the study of sorting by coming up with a prototype sorting algorithm—the new clarity is the important part, while the concrete program was just the vehicle that got us there.

Additionally, in order to assess how well BootstrapLab achieves the Three Properties, we propose a *technical dimensions* framework in Chapter 5 for analysing programming systems, which is our secondary contribution. BootstrapLab, being a programming system, is then evaluated in terms of dimensions constituting the Three Properties. We then review related work in Chapter 6. In Chapter 7 we acknowledge the limitations revealed by our evaluation, suggest future work for both of our contributions, and conclude with what we have learned and achieved.

1.7 SUPPORTING PUBLICATIONS

The following essay was adapted into Chapter 4:

Joel Jakubovic and Tomas Petricek (2022). "Ascending the Ladder to Self-Sustainability: Achieving Open Evolution in an Interactive Graphical System." In: *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2022. Auckland, New Zealand: Association for Computing Machinery, pp. 240–258. ISBN: 9781450399098. DOI: 10.1145/3563835.3568736

The following paper was adapted into Chapter 5 and part of Chapter 2:

Joel Jakubovic et al. (Feb. 2023). "Technical Dimensions of Programming Systems." In: *The Art, Science, and Engineering of Programming* 7.3. DOI: 10.22152/programming-journal.org/2023/7/13

BACKGROUND

The relevant background we will need to acquaint ourselves with falls into two halves: explaining the concept of a "programming system", and explaining how the Three Properties tie together existing concepts in programming. In Section 2.1, we define *programming systems* in contrast to programming languages and discuss why this is necessary. Then in Section 2.2, we illustrate this with landmark examples of programming systems from the past. Finally, in Section 2.3, we survey the existing patterns in programming that take us part of the way to the Three Properties.

2.1 PROGRAMMING SYSTEMS VS LANGUAGES

Many forms of software have been developed to enable programming. The classic form consists of a *programming language*, a text editor to enter source code, and a compiler to turn it into an executable program. Instances of this form are differentiated by the syntax and semantics of the language, along with the implementation techniques in the compiler or runtime environment. Since the advent of Graphical User Interfaces (GUIs), programming languages can be found embedded within graphical environments that increasingly define how programmers work with the language—for instance, by directly supporting debugging or refactoring. Beyond this, the rise of GUIs also permits diverse visual forms of programming, including visual languages and GUI-based end-user programming tools (we will survey these in Section 6.4).

The classic essay by Gabriel (2012) distinguishes the *languages* and *systems* paradigms in programming research. *Languages* are formal mathematical models of syntax and semantics; researchers might ask what an expression *means* and include code samples in papers. *Systems,* in contrast, are running pieces of software whose current state changes according to the effects of program code. Researchers studying systems are likely to be more concerned with what code *does* to a running system in a specific state instead of the more abstract language properties.

The topic of this dissertation, and many of the examples we will use to illustrate concepts, rely on understanding this distinction and only make sense within the systems paradigm. Therefore we shift our attention from *programming languages* to the more general notion of "software that enables programming"—in other words, *programming systems*.

Definition 9 (Programming System). A *programming system* is an integrated and complete set of tools sufficient for creating, modifying, and executing programs. These will include notations for structuring programs and data, facilities for running and debugging programs, and interfaces for performing all of these tasks. Facilities for testing, analysis, packaging, or version control may also be present. Notations include programming languages and interfaces include text editors, but are not limited to these.

A word about terminology: if we view languages in the sense of Gabriel's "languages paradigm", then it is a "type error" to include languages in the above definition. Abstract mathematical models of syntax and semantics are not the same as software. However, language *implementations* are software. We will use the term "language" to abbreviate "language implementation" since we do not use the other meaning in this dissertation.

With that said, our above notion of programming system covers classic programming languages together with their editors, debuggers, compilers, and other tools. Yet it is intentionally broad enough to *also* accommodate image-based programming environments like Smalltalk, operating systems like Unix, and hypermedia authoring systems like Hypercard, in addition to various other examples we will list presently.

2.2 EXAMPLES OF PROGRAMMING SYSTEMS

We illustrate the notion of a programming system through a number of example systems. We are not trying to exhaustively cover all possible systems, but simply give an impression based on major examples. We draw them from three broad reference classes:

- Software ecosystems built around a text-based programming *language*. They consist of a set of tools such as compilers, debuggers, and profilers. These tools may exist as separate command-line programs, or within an Integrated Development Environment (IDE).

- Those that resemble an Operating System (OS) in that they structure the execution environment and encompass the resources of an entire machine (physical or virtual). They provide a common interface for communication, both between the user and the computer, and between programs themselves.

- Programmable *applications*, typically optimised for a specific domain, offering a limited degree of programmability which may be increased with newer versions.

2.2.1 *Systems Based Around Languages*

Text-based programming languages sit within programming systems whose boundaries are not explicitly defined. To speak of a programming system we must include a language with, at minimum, an editor and a compiler or interpreter.

There is wiggle room in how we choose to circumscribe these elements. Do we mean a specific compiler version? Do we include common plugins or extensions? Still, we would expect these choices to have enough of a common overlap that we can proceed in analysis without worrying too much about the variations. We will revisit this point in Section 7.1.5.

JAVA WITH THE ECLIPSE ECOSYSTEM.   The Java language (Gosling et al. 2000) alone does not form a programming system, but it does if we consider it as embedded in an ecosystem of tools. A minimalistic delineation would consist of a text editor to write Java code and a command line compiler. A more realistic one is Java as embedded in the Eclipse IDE (desRivieres and Wiegand 2004). The programming systems view permits us to see whatever there may be beyond the textual code. In the case of Eclipse, this includes the debugger, refactoring tools, testing and modelling tools, GUI designers, and so on.

HASKELL TOOLS ECOSYSTEM.   Haskell is another language-focused programming system. It is used through the command-line *GHC* compiler (Marlow and Peyton-Jones 2012) and *GHCi* Read-Eval-Print Loop (REPL), alongside a text editor that provides features like syntax highlighting and auto-completion. Any editor that supports the Language Server Protocol (Microsoft 2022) will suffice to complete the programming system.

Haskell is mathematically rooted and relies on mathematical intuition for understanding many of its concepts. This background is also reflected in the notations it uses. In addition to the concrete language syntax for writing code, the ecosystem also uses an informal mathematical notation for writing about Haskell (e. g. in academic papers or on the whiteboard). This provides an additional tool for manipulating Haskell programs. Experiments on paper can provide a kind of rapid feedback that other systems may provide through live programming.

FROM REPLS TO COMPUTATIONAL NOTEBOOKS.   A different kind of developer ecosystem that evolved around a programming language is the Jupyter notebook platform (Kluyver et al. 2016). In Jupyter, data scientists write scripts divided into notebook cells, execute them interactively and see the resulting data and visualisations directly in the notebook itself. This brings together the REPL, which dates back to conversational implementations of Lisp in the 1960s, with literate

programming (Knuth 1984b) used in the late 1980s in Mathematica 1.0 (Wolfram 1991).

As a programming system representative of Computational Note-books (Lau et al. 2020), Jupyter has several interesting characteristics. The primary outcome of programming is the notebook itself, rather than a separate application to be compiled and run. The code lives in a document format, interleaved with other notations. Code is written in small parts that are executed quickly, offering the user more rapid feedback than in conventional programming. A notebook can be seen as a trace of how the result has been obtained, yet one often problematic feature of notebooks is that some allow the user to run code blocks out-of-order. The code manipulates mutable state that exists in a "kernel" running in the background. Thus, retracing one's steps in a notebook is more subtle than in, say, Common Lisp (Steele and Fahlman 1990), where the `dribble` function would directly record the user's session to a file.

2.2.2 *OS-Like Programming Systems*

"OS-likes" date from the 1960s when it became possible to interact one-on-one with a computer. At first, time-sharing systems enabled interactive shared use of a computer via a teletype; smaller computers such as the PDP-1 and PDP-8 provided similar direct interaction, while 1970s workstations such as the Alto and Lisp Machines added graphical displays and mouse input. These *OS-like* systems stand out as having the totalising scope of *operating systems*, whether or not they are ordinarily seen as taking this role.

MACLISP AND INTERLISP.   LISP 1.5 (McCarthy 1962) arrived before the rise of interactive computers, but the existence of an interpreter and the absence of declarations made it natural to use Lisp interactively, with the first such implementations appearing in the early 1960s. Two branches of the Lisp family (Steele and Gabriel 1993), MacLisp and the later Interlisp, embraced the interactive "conversational" way of working, first through a teletype and later using the screen and keyboard.

Both MacLisp and Interlisp adopted the idea of *persistent address space*. Both program code and program state were preserved when powering off the system, and could be accessed and modified interactively as well as programmatically using the *same means*. Lisp Machines embraced the idea that the machine runs continually and saves the state to disk when needed. Today, this *persistence* (Definition 2) is widely seen in cloud-based services like Google Docs and online IDEs. Another idea pioneered in MacLisp and Interlisp was the use of *structure editors*. These let programmers work with Lisp data structures not as sequences of characters, but as nested lists. In Interlisp, the programmer would use commands such as *P to print the current expression, or *(2 (X

Y)) to replace its second element with the argument (X Y). The PILOT system (Teitelman 1966) offered even more sophisticated conversational features. For typographical errors and other slips, it would offer an automatic fix for the user to interactively accept, modifying the program in memory and resuming execution. This is something that is only possible with Lisp's *naïve pokeability* (Definition 1).

SMALLTALK.   Smalltalk appeared in the 1970s with a distinct ambition of providing "dynamic media which can be used by human beings of all ages" (Kay and Goldberg 1977). The authors saw computers as *meta-media* that could become a range of other media for education, discourse, creative arts, simulation and other applications not yet invented. Smalltalk was designed for single-user workstations with a graphical display, and pioneered this display not just for applications but also for programming itself. In Smalltalk 72, one wrote code in the bottom half of the screen using a structure editor controlled by a mouse, and menus to edit definitions. In Smalltalk-76 and later, this had switched to text editing embedded in a *class browser* for navigating through classes and their methods.

Similarly to Lisp, Smalltalk adopts the persistent address space model of programming where data remains in memory, but based on *objects* and *message passing* instead of *lists*. Any changes made to the system state by programming or execution are preserved when the computer is turned off (this is *persistence* again, Definition 2). Lastly, the fact that much of the Smalltalk environment is implemented in itself makes it possible to extensively modify the system from within: Smalltalk exhibits Self-Sustainability.

An important descendant of Smalltalk is *ThingLab* (Borning 1981, Figure 2.1)—later succeeded by ThingLab II (Moloney et al. 1989)—which augments the class browser with a *picture pane.* ThingLab was a *constraint-based* programming system in the vein of Sketchpad (Sutherland 1963). ThingLab provided not only a library of pre-existing constraints and objects for use in electrical and geometric simulations, but was designed as a "kit-building kit" with the ability to create new types of objects and constraints via the Smalltalk language. The extent to which ThingLab and Sketchpad can define new graphical constructs, and give them semantics (as opposed to merely drawing a picture) qualifies for *Notational Freedom* as well as Self-Sustainability.

We include Lisp and Smalltalk in the OS-likes because they function as operating systems in many ways. On specialised machines, like the Xerox Alto and Lisp machines, the user started their machine directly in the Lisp or Smalltalk environment and was able to do everything they needed from *within* the system. Nowadays, however, this experience is associated with Unix and its descendants on a vast range of commodity machines.
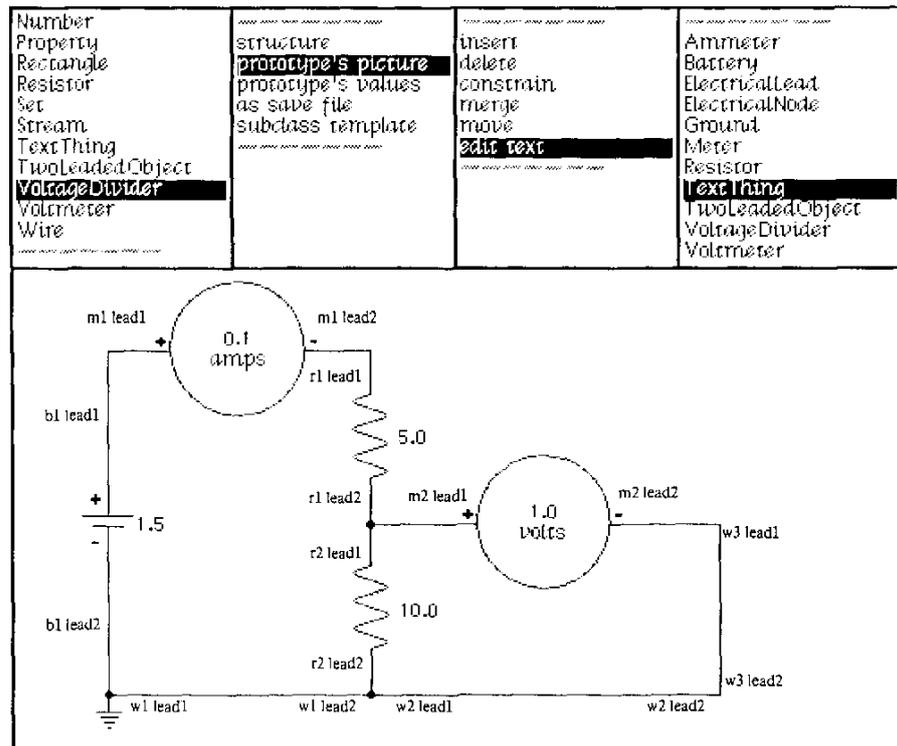
Figure 2.1: ThingLab could be used to explore constraint-based simulations such as electrical circuits, but could also leverage Smalltalk to create new objects and constraints that could be graphically specified and instantiated.

UNIX.   Unix fits our Definition 9 for programming systems and illustrates the ways that a system is shaped for its intended target audience. Built for computer hackers (Levy 1984), its abstractions and interface are close to the machine. Although historically linked to the C language, Unix developed a language-agnostic set of abstractions that make it possible to use multiple programming languages in a single system. While everything is an object in Smalltalk, the ontology of Unix consists of files, memory, executable programs, and running processes. Note the explicit "stage" distinction here: Unix distinguishes between volatile *memory* structures, which are lost when the system is shut down, and non-volatile *disk* structures that are preserved. This distinction between types of memory is considered, by Lisp and Smalltalk, to be an implementation detail to be abstracted over by their persistent address space. Still, this did not prevent the Unix ontology from supporting a pluralistic ecosystem of different languages and tools. Thus Unix is distinguished as a *meta*-programming system, supporting the creation and interaction of different programming systems within it. We will go into more detail on these points in Section 3.2.2.

EARLY AND MODERN WEB.   The Web evolved (Ankerson 2018) from a system for sharing and organising information to a *programming system*. Today, it consists of a wide range of server-side programming tools, JS and languages that compile to it, notations like HTML and CSS, and the sophisticated developer tools included in browsers. As a programming system, the "modern 2020s web" is reasonably distinct from the "early 1990s web". In the early web, JS code was distributed in a form that made it easy to copy and re-use existing scripts, which led to enthusiastic adoption by non-experts—recalling the birth of microcomputers like Commodore 64 with BASIC a decade earlier.

In the "modern web", multiple programming languages treat JS as a compilation target. JS is also used as a language on the server side. This web is no longer simple enough to encourage copy-and-paste remixing of code from different sites. However, as we observed in Section 1.3.1 it does come with advanced developer tools providing functionality resembling that of Lisp and Smalltalk. The DOM almost resembles the tree/graph model of Smalltalk and Lisp images, lacking the key *persistence* property. Such a limitation is being addressed by efforts like Webstrates (Klokmose et al. 2015), which synchronise the DOM between the server and clients. Thus if a client changes an element, this can be mirrored on the server side and saved as the ground truth of the web page.

COLAS.   The one system that directly influenced our work is the Combined Object Lambda Architecture, or COLA (Piumarta 2006): a small, expressive starting system designed for open evolution by its user. It is described as a mutually self-implementing pair of abstractions: a

```
+-------------+-------------+-----------------------+---------+-------------------------------------+
| 00 01 02 03 | 04 05 06 07 | 08 09 10 11 12 13 14 15 | 16 17 18 | 19 20 21 22 23 24 25 26 27 28 29 30 31 |
+-------------+-------------+-----------------------+---------+-------------------------------------+
|   version   |  headerSize |      typeOfService      |                    length                     |
+-------------+-------------+-----------------------+---------+-------------------------------------+
|               identification                |   flags  |                 offset                |
+---------------------------+-----------------------+---------+-------------------------------------+
|         timeToLive        |         protocol        |                   checksum                    |
+---------------------------+-----------------------+---------+-------------------------------------+
|                                       sourceAddress                                                |
+---------------------------------------------------------------------------------------------------+
|                                     destinationAddress                                             |
+---------------------------------------------------------------------------------------------------+
```
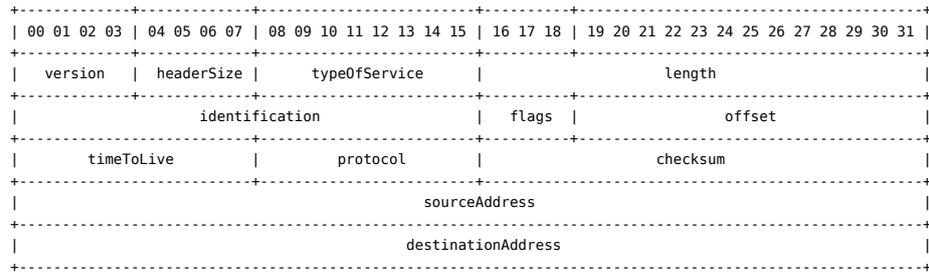
Figure 2.2: ASCII art diagram for an IP packet, in principle both human-readable and machine-parseable (Kay et al. 2007, p. 44).

structural object model (the "Object" in the name) and a behavioural Lisp-like language (the "Lambda"). COLA aims for maximal openness to modification, down to the basic semantics of object messaging and Lisp expressions.

The two remarkable features we see in the COLA idea are *self-sustainability* and a hint at *notational freedom*, which we will exhibit presently. COLA inherits self-sustainability from the Smalltalk tradition and attempts to amplify it. This provides for what the authors refer to as *internal evolution* as a means for implementing *MSLs:*

> Applying [internal evolution] locally provides scoped, domain-specific languages in which to express arbitrarily small parts of an application (these might be better called *mood-specific* languages). Implementing new syntax and semantics should be (and is) as simple as defining a new function or macro in a traditional language.

An example of a MSL is the one reported in Kay et al. (2008, p. 4) for concisely specifying how TCP packets should be processed. The report also notes:

> The header formats are parsed from the diagrams in the original specification documents, converting "ascii art" into code to manipulate the packet headers.

A representative diagram and the grammar for parsing it (itself written in a grammar-definition DSL) can be found in Kay et al. (2007, p. 44, Figures 2.2 and 2.3). They note:

> We can now define accessors for the fields of an IP packet header simply by drawing its structure. The following looks like documentation, but it's a valid *program.*

We see this as the extreme end of what a MSL is capable of. The ability for a programmer to express arbitrarily small parts of an application in a form they deem suitable is, in its fully *general* form, what we call

```
structure :=
    error  = ->[self error: ['"structure syntax error near: " , [self contents]]]
      eol  = '\r''\n'* | '\n''\r'*
    space  = [ \t]
  comment  = [-+] (!eol .)* eol
       ws  = (space | comment | eol)*
        _  = space*
   letter  = [a-zA-Z]
    digit  = [0-9]
identifier = id:$(letter (letter | digit)*) _   -> [id asSymbol]
   number  =                   num:$digit+ _   -> [Integer fromString: num base: '10]
  columns  =        '|'                        -> (structure-begin self)
               ( _ num:number                  -> [bitmap at: column
                                                         put: (set bitpos num)]
                   (num:number)* '|'           -> (let ()
                                                    (set bitpos num)
                                                    (set column [
                                                       [self readPosition] - anchor
                                                      ]))
               )+ eol ws                        -> [bitmap at: column
                                                         put: (set width [bitpos + '1])]
      row  =        ( n:number                  -> (set row n)
                     ) ? '|'                    -> (let ()
                                                    (set anchor [self readPosition])
                                                    (set column '0))
               _ ( id:identifier '|'           -> (structure-field self id)
                   _ )+ eol ws                  -> (set row [row + width])
     name  =        id:identifier (!eol .)* eol -> (structure-end id)
  diagram  =     ws columns row+ name | error
```

Figure 2.3: Grammar for parsing the ASCII art into a data structure definition
(Kay et al. 2007, p. 44). Message sends in square brackets [] and
Lisp-like expressions in parentheses () are visible; these respec-
tively relate to the "Object" and "Lambda" halves of the underlying
COLA.

Notational Freedom. With such a capability, code could be synthesised from *real* tabular diagrams[1] of packet headers, not just those rendered with ASCII characters.

We mentioned how these MSLs were supported by COLA's *internal evolution,* which we interpret as another term for Self-Sustainability (or, more precisely, its practice). This is provided for by the COLA architecture, composed of an object model for structure (Piumarta and Warth 2008) and a Lisp-like language for behaviour (Piumarta 2011).

The object model is a late-bound, Smalltalk-style objects and messaging environment called "Id", and we will describe it briefly here. An Id *object* is a block of state which can change as a result of messages received by it. Messaging (analogous to *method invocation* in Java-style OOP) works as follows:

- A message is sent by first *bind*-ing its name to its *method implementation*, which is specific code that gets run in the context of the receiver *R*. This binding is a dynamic operation that can use run-time conditions to make its decision—even create a method implementation on the fly. This *late binding* contrasts with *early binding* where the name is bound to the implementation statically and thus holds for the entire running lifetime of the system.

- This "bind" step is accomplished by sending a further message; this time, to the receiver's *vtable V(R)*. A vtable is another object that maps "message name" to "implementation code"—analogous to a "class" in Java-style OOP.

- Because this initial "bind" message is itself a message send, it triggers a similar "bind" to *its* vtable $V(V(R))$, and so on: recursing up the vtable chain, and terminating at a base case.

- The higher levels of the vtable chain mean that different kinds of vtables can be supported (as well as different kinds of "kinds of vtables", and so on). Each kind of vtable may implement the "bind" operation in its own way. For example, one kind of vtable could employ a simple dictionary mapping names to implementations. A different kind might build the code for the implementation just-in-time, without looking anything up in a data structure.

2.2.3   *Application-Focused Systems*

The previously discussed programming systems were either universal, not focusing on any particular kind of application, or targeted at broad fields, such as Artificial Intelligence and symbolic data manipulation in

---

1 We are thinking of vector graphics formats here, but computer vision techiques would allow someone to use raster screenshots if they so desired.

Lisp's case. In contrast, the following examples focus on narrower application domains. Many support programming based on rich interactions with specialised visual and textual notations.

SPREADSHEETS.   The first spreadsheets became available in 1979 in VisiCalc (Grad 2007; Zynda 2013) and helped analysts perform budget calculations. As programming systems, spreadsheets are notable for their two-dimensional grid substrate and their model of automatic re-evaluation. The programmability of spreadsheets developed over time, acquiring features that made them into powerful programming systems in a way VisiCalc was not. A major step was the 1993 inclusion of *macros* in Excel, later further extended with *Visual Basic for Applications* and more recently with *lambda functions* (Murray 2022).

HYPERCARD.   While spreadsheets were designed to solve problems in a specific application area, HyperCard (Michel 1989) was designed around a particular application format. Programs are "stacks of cards" containing multimedia components and controls such as buttons. These controls can be programmed with pre-defined operations like "navigate to another card", or via the HyperTalk scripting language for anything more sophisticated.

As a programming system, HyperCard is interesting for a couple of reasons. It effectively combines visual and textual notation. Programs appear the same way during editing as they do during execution. Most notably, HyperCard supports gradual progression from the "user" role to "developer": a user may first use stacks, then go on to edit the visual aspects or choose pre-defined logic until, eventually, they learn to program in HyperTalk.

GRAPHICAL LANGUAGES.   Efforts to support programming without relying on textual code are "languages" in a more metaphorical sense. In the "boxes-and-wires" style of LabView (Kodosky 2020) programs are made out of graphical structures. There are also the Programming-By-Example/Demonstration subgenres (Cypher 1993; Lieberman 2001) in which a general program is automatically generated by the user supplying sample behaviours and hints.[2]

2.3   PRECURSORS OF THE THREE PROPERTIES

In the next chapter, we will go on to develop the Three Properties in detail. However, they do not leap out of a vacuum, but are rather developments of concepts that already exist in programming. For each of the Three Properties, we will give a "glossary" of these existing concepts and finish with a "Conclusion" entry. In short:

---

2 Although there is nothing inherently graphical about programming by example or demonstration, the two definitive books cited show a strong graphical orientation.

- Self-Sustainability is foreshadowed by self-hosting compilers and reflection.

- Notational Freedom generalises Domain-Specific Languages (DSLs) and polyglot programming.

- Explicit Structure already exists widely in the form of data structures and various editors; programming is the exception.

2.3.1   *Precursors of Self-Sustainability*

2.3.1.1   *Self-Hosting*

This describes a compiler that can compile its own source code into a functionally identical compiler program. We can then change the language understood by the compiler by changing the source code, compiling it with the current version, and discarding this version in favour of the new one. We can then rewrite the compiler's source code to make use of the new language feature. In this way, a programming language can be evolved using itself. We can call the language "self-hosting" as a proxy for its compiler.

2.3.1.2   *Bootstrapping*

This is the process of getting a language into a self-hosting state (Evans 2001). Suppose we design a novel language *NovLang* and we are happy to use C++ to build its compiler. Bootstrapping it consists of the following steps; see Figure 2.4 for a *tombstone diagram* (Wickerson and Brunet 2012):

1. We write a NovLang compiler in NovLang, but we cannot run it yet.

2. We translate this by hand to C++ and build a temporary NovLang compiler.

3. We run this to compile the NovLang source code from step 1.

4. We obtain a runnable compiler for NovLang, which was written in NovLang and is now self-hosting.

5. We can now discard the C++ code.

2.3.1.3   *Meta-Circular*

This describes an interpreter that is written in its own language (C2 Contributors 2012). This was first introduced in Lisp, in which one can write Lisp code to walk nodes in a data structure and treat them
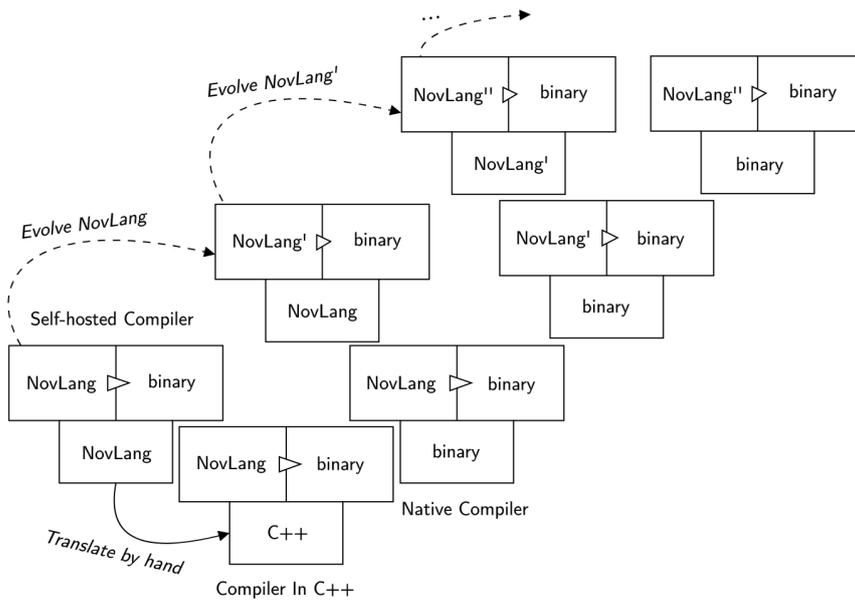
Figure 2.4: In a *tombstone diagram*, each "tombstone" represents a program translating the language on its left to the one on its right, where this program is written in the language at the base of the tombstone. In this example, C++ is used to implement a temporary compiler for NovLang, which then compiles a self-hosted compiler to a runnable binary. This binary, plus its original NovLang source code, can then be mutually used to evolve NovLang in the manner shown.

as Lisp expressions. If such code is compiled, it results in an ordinary Lisp interpreter. Alternatively, if we feed such code into an existing Lisp interpreter, this new inner interpreter is now meta-circular. We could change the code to add a new language feature, in which case the inner interpreter would understand this slightly improved language. However, this approach does not scale, as each improvement would nest a further interpreter within the previous ones, multiplying the overhead to impractical levels.

2.3.1.4   *Reflection*

This is the capacity for a system to display, explain or affect its own computational behaviour during run time (Maes 1987). It is sometimes explained with the word "aboutness": an ordinary program is "about" its domain (say, calculations), while a reflective program is also "about" its own computation. One test of this is the ability to make the tacit explicit (Smith 1982): entities that are normally implicit and unaddressable (such as the stack frame or variable binding environment) can be made so by an explicit command to reflect. An ordinary meta-circular interpreter cannot name its outer interpreter's data structures, but a reflective one can (and may be able to change how its outer interpreter works, and thus how it itself works). This is developed exhaustively for Lisp-like languages in Smith (1982). While reflection originates as a property of languages, the Self environment (Ungar and Smith 2007) provides an example in an interactive context. Any object on screen can call up an "outliner" object with a description of its prototype, private state and methods. This outliner is an object and can have the same operation applied to it (Bystroushaak 2019).

2.3.1.5   *Conclusion*

These concepts (self-hosting, bootstrapping, meta-circularity, and reflection) seem related but it is not obvious how. We interpret all of these as different manifestations of self-sustainability in special contexts, such as compilers or interpreters. In Chapter 3 we will make this more precise by delving into the differences between compilers, interpreters, and interactive programming systems.

2.3.2   *Precursors of Notational Freedom*

2.3.2.1   *Use The Right Tool For The Job*

This is a widespread maxim in programming (C2 Contributors 2014c) that we encountered in Definition 6. The ideal conditions capturing the spirit of this idea are as follows.

- *Subjective Preference:* What is "right" is subjectively determined by the programmer, even on a whim. Ordinarily, when proposing a change to a language (e. g. Python), every user of the language is forced to confront the change, which invites debate about what is "right" for everyone. This could be sidestepped if each programmer could use what is right, for their own context, without this being forced on others. This is meant even in a collaborative context: ideally, each collaborator may use their chosen tool while a common infrastructure or format allows their efforts to cohere.

- *Metaphorical Tool:* The "tool" might be an entire programming system or language, a design approach, or simply a notation in which to express a component.

- *Range of Scopes:* The "job" can be large (an entire project), small (a single expression) or anything in-between.

Even though these are ideal conditions that we do not inhabit, being aware of them lets us get a sense of how applicable this principle is and opens us to any low-hanging fruit in this area. We will now review the limited extent to which we can apply the principle in our actual environment of programming.

2.3.2.2 *Polyglot Programming*

This is the practice of using multiple languages in a single project (Ford 2006). Modules implemented in different languages need to share data and invoke each other's functions, for which there are several approaches:

- If the modules are separate programs, standard inter-process communication mechanisms like interchange file formats (JSON, XML), socket protocols, and Remote Procedure Calls are available.

- Polyglot programming *within* a shared process address space is trickier; the classic approach is to have different languages (e. g. C, Pascal) compile to a common *object file* format understood by the linker. This method is restricted to compile time; for run-time sharing, languages use Foreign Function Interfaces (FFIs). This practice has been critiqued by Kell (2009) who advocates the use of "integration domains" instead.

- Polyglot programming within a single *file* is rare and restricted to fixed combinations. Perl and JS support regexes written with a local syntax. PHP embeds HTML, which in turn includes JS and CSS. C# supports Language INtegrated Queries (LINQ) which are a C# syntax adaptation of SQL queries. Unlike the freedom to choose *any* combination of existing languages for an entire project, these instances only permit use of a *pre-approved* set decided by the language designers.

2.3.2.3 *Domain-Specific Languages*

These go beyond Polyglot Programming by encouraging *custom* languages designed by the programmer for their problem domain. Where Polyglot Programming is about making the best use of *existing* languages designed by someone else, DSLs come closer to a *freedom* to use what one subjectively determines to be the best tool for one's job. Jet-Brains' MPS (Voelter and Pech 2012) is an interactive programming system that encourages DSLs. "Reader Macros" in Lisp allow a programmer to use custom syntax for parts of the code. The COLA design (Piumarta 2006) supports "Mood-Specific Languages" intended to span a range of scopes down to individual expressions, and the related OMeta (Warth 2009) project is a platform for custom DSLs. The Eco editor (Diekmann and Tratt 2014) also supports MSLs, as does JetBrains' MPS (Voelter and Pech 2012).

2.3.2.4 *Conclusion*

"Use The Right Tool For The Job" is the basic intuition behind Notational Freedom. In practice, we see a restricted version: use the right *pre-existing* tool for the job, as long as the job is no smaller than a single file. Occasionally, a pre-approved set of different languages are available within a single file. In the rare cases that support the use of custom "tools" within a file, we risk being restricted to *languages* rather than general *notations.* We will list the rare examples that provide for the latter when we give a full definition of Notational Freedom in the next chapter (Section 3.3.2).

2.3.3 *Precursors of Explicit Structure*

2.3.3.1 *Structure*

This is the relation of parts to wholes. An entire data structure is made up of smaller parts which reference each other. From one perspective, a data structure is a graph of memory blocks connected by numerical pointers. At a more human-friendly level, it is a graph of dictionaries with named entries, some of which point at other dictionaries. Both of these models can be visualised as boxes with arrows.

2.3.3.2 *Binary files*

This is a very general term referring to any file that cannot be treated as plain text. We see binary files as *compacted* data structures. Unlike data structures in memory, which can be sparsely spread across large regions and intermingled with each other, a binary file will often contain the parts densely packed together without any unrelated data. If not, the

binary file serves as an uncompacted "image" of a region or regions of memory.

2.3.3.3   *Syntax*

In programming, this refers to the "look and feel" of a language's textual source code. Formally, syntax is the set of rules defining legal and illegal symbol sequences. This idea can be metaphorically extended to non-sequential structures. For example, we can think of C `struct` definitions as setting out the valid "shape" of the parts of a data structure. The same applies to binary file formats.

2.3.3.4   *Quantitative Syntax*

This is a term introduced by Hall (2017, p. 13) for the pattern of prefixing a block with its length and using numerical pointers to link structures. A simple example is the Pascal String which begins with a length byte and continues for that many characters. Binary files rely primarily on quantitative syntax.

2.3.3.5   *Qualitative Syntax*

This, in contrast to Quantitative Syntax, relies on special delimiters. For example, the C String begins right away with its characters, relying on a null byte to show up at some point and signal the end.

2.3.3.6   *Text files, i. e. Strings*

These are lists of plain text characters. In programming, they are often containers for machine-readable structures as an alternative to binary files. Like the latter, they compact the structure, but in a way subject to the constraints of plain text and qualitative syntax. In practice, the structure in question is a tree, in which case the string is built as an *in-order* traversal.[3] This means that special qualitative syntax (usually the matched bracket characters `([{<`, occasionally `begin`/`end` or whitespace as indentation) is employed *around* substrings to encode the structure.

2.3.3.7   *Parsing and serialising*

These convert between strings and structures. Parsing *recovers* structure implicit in a string, while serialising spins out the structure into a string.

---

3 The alternative pre-order and post-order traversals result in machine-friendly prefix and postfix serialisations, which can build the structure back up via a stack. However, these are not easily readable or writeable for humans.
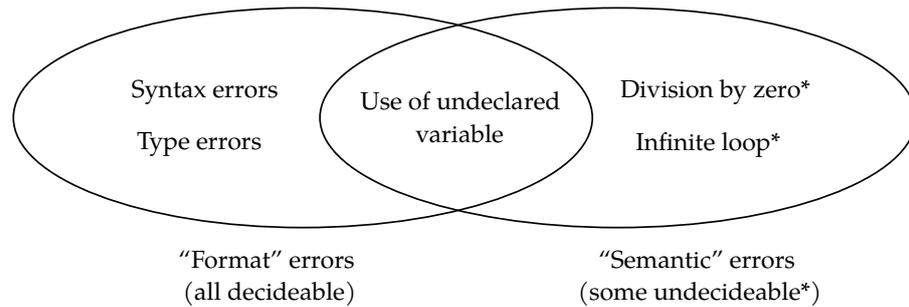
Figure 2.5: Format errors include syntax errors, type errors and some "semantic" errors as long as they are decideable.

2.3.3.8 *Format error*

This is a part of a data structure that violates a decidable expectation of its consumer. For example, a syntactically valid file containing program source code might violate static typing rules or use a name that was not declared. Undecidable "semantic" rules like prohibiting division by zero are excluded from this term (see Figure 2.5).

2.3.3.9 *Syntax error*

This is a specific type of format error where part of a text string violates a grammatical expectation of its consumer.

2.3.3.10 *Editors*

These are programs for creating various data structures in the form of files. Editors for 3D models, vector graphics, raster images, audio, and video understand the file formats and strive to save only valid files. It is usually not possible to even *express* a structure in the editor that contains a format error. Such cases are exceptional: for example, a 3D scene might open without errors in another 3D editor, but cause errors in a game engine according to the latter's additional requirements—perhaps it expects specific objects in the scene named `Player`, `Exit`, and so on. Nevertheless, for most editors and most use cases, the consumer-side validity rules are in harmony with the producer-side rules.

2.3.3.11 *Text Editors*

These are a type of editor for plain text files. However, they are widely used to write code in programming languages, which have extra syntax rules beyond the plain text format. Unlike most editors, text editors *can* save files that are invalid from the perspective of their consumers under realistic use cases. These syntax errors are then discovered at the point of consumption.

2.3.3.12 *Conclusion*

The basic intuition behind Explicit Structure is the *directness* experienced in creation and programming. Almost every data structure in computing has an editor with which one can manipulate the structure directly, and when programming we can act as if data structures have named parts that we can simply reference. This directness is interrupted by the standalone exception of text editors (on the creation side) and strings with machine-readable[4] implicit content (on the programming side).

2.4 REVIEW AND NEXT STEPS

We have established the frame of *programming systems* through Definition 9 and numerous examples. This prepared us to survey the various *precursors* of our Three Properties, which are more common than the Properties themselves, as aspects of different systems without limiting ourselves to languages. Now that we have this context, we will go deeper into our model of programming systems and finally give full definitions of the Three Properties, which will lay the groundwork for our contribution in Chapter 4.

---

4 We are unconcerned with strings that contain natural language simply to be echoed out to the user (e. g. error messages). However, our ideas about Explicit Structure could be applicable to cases where software must parse and interpret natural language too.

ANALYSIS

This dissertation is about building programming systems. In Section 2.1 we defined this concept and explained how it is related to that of a programming language. Here, we will distinguish the general concepts of *state* and *change* in programming systems (Section 3.1). We will illustrate this by distinguishing the *low-level binary* and *minimally human-friendly* levels of abstraction. We will then offer our interpretation of three major sets of conventions in which programming systems have existed, which we call *paradigms* (Section 3.2). With these in mind we will go on to define our Three Properties in more detail (Section 3.3). We will conclude by reviewing the limitations of existing work in achieving the Three Properties in the way we desire.

3.1 TWO FUNDAMENTALS: STATE AND CHANGE

For the present work, our most general model of a programming system is like a physical system in the sense of analytical mechanics (Sussman and Wisdom 2001). There is always a current *state* of the system, and this will necessarily *change over time*. We stress that this is the case regardless of whether the underlying programming metaphor is imperative, purely functional, logic-based, or otherwise eschews a notion of "state" in its conceptual model.

To see how this is inevitable, consider the following. In working with a declarative or functional programming system, the expression you are currently editing or the output you are seeing at a given moment is, by definition, a single state. This state changes whether you interact or simply wait for progress. In other words, anything to which this view is not applicable will not be interactive or interesting.

In such a model, we include both the visible interface and the "hidden" internal state of the system (e. g. heap data structures) as part of "state". Such an all-encompassing "state", of course, is not comprehensible atomically but is always broken down into substructures: on the interface side, this is usually various types of rectangles, while internally we see byte lists, object graphs, trees and so on. Likewise, the actual *change* from one state to another usually does not involve all of the state but only a small part of it. In the limit, there is usually some smallest unit of state (a byte, dictionary entry, tree node) and this gives rise naturally to primitive *instructions* describing a change to such a small unit. Different choices for how to represent the instructions have implications for where it is possible to take the evolution of a system.

We will see in Chapter 4 that some choices are more appropriate than others for ensuring a system can be made self-sustainable.

3.1.1   *The Low-Level Binary World*

While human beings think in terms of names, computer hardware works on numerical bit patterns. This shows through to the lowest level of the software stack, which we call the *low-level*, *machine-level* or *binary world.* Here, state consists of one long line of bytes while change is achieved via machine instructions interpreted by the hardware. There are three noteworthy features of state here for a programmer's mental model:

- *Flatness:* if there is not enough space to insert something, we have to physically move things to make room. For example, if a list of integers is represented by a contiguous array, inserting at the beginning requires moving every later entry one place forward. If there is not enough spare capacity in the array, it needs copying somewhere else with more space.

- *Absoluteness:* while various instruction sets support relative addresses, data structures are typically established through absolute pointers. Thus when a data structure is moved, any internal pointers need relocating based on the new start address.

- *Numerical meaninglessness:* addresses are numerical, but no number has an inherent meaning. This interchangeability means it is unlikely that two parties will happen to coordinate on the same number for the same purpose. Instead, they must communicate beforehand and agree on which numerical addresses hold which things or correspond to which names.

3.1.2   *The Minimally Human-Friendly World*

Key technologies were developed to allow humans to create software using *names* instead of numbers: symbolic assemblers, high-level languages, and so on. These free us from the cognitive difficulties associated with numerical labels in the binary world. For example, programs in the C programming language use names for variables, functions, and the parts of data structures. However, C's nature as "portable assembly" (Kernighan and Ritchie 1989) means that the flatness and absoluteness of memory must remain in our awareness when using the language. Even though data structures can be designed to grow by containing slots for pointers to newly allocated blocks, competence at C still requires an understanding of the real nature of low-level memory.

A language like JS, in contrast, does away with this: state in JS is a graph of dictionaries with named entries. One does not allocate so many bytes of memory and receive an absolute pointer, but instead creates an

empty dictionary and receives the dictionary itself.[1] We assert that this is at least the *minimally* human-friendly model of state that is possible, even though improvements could easily be suggested. To be explicit, the aforementioned three low-level aspects get negated as follows:

- *Dynamic and nestable* instead of *flat:* one can simply add new entries, insert items in collections and create new objects.

- *References* instead of *absolute pointers:* there is no distinction between pointers and values. References are implicit and automatic. One only needs to be aware how side effects work for mutable objects versus immutable values like strings. The underlying system can lay out these data structures in memory however it wishes and even move them without the programmer needing to update the references.

- *Names* instead of *numbers:* objects can be identified by a root-level name or by a multi-name path. This provides for logical nesting relationships and makes it easier for parties to coordinate (agreeing on an already-used name is easier than finding an available number and agreeing on it).

3.1.3   *Let Us Avoid The Low-Level Binary World*

In the comparison of programming abstractions, there are what we could call *industrial* virtues: precise control, efficiency, performance, and so on. There are also *leisurely* virtues: simplicity, convenience, and lack of responsibilities. The industrial and leisurely virtues are somewhat in opposition and correlate with a model's perception as "low-level" or "high-level". Lower-level abstractions are said to embody more of the "industrial" virtues at the expense of the "leisurely", and vice versa for high-level abstractions.

We say all this to note that we agree with the general pattern, but with one important exception: we do not think it is worth working *directly* in the low-level binary world outside of narrow special cases. It was historically necessary to begin there, and it still underlies all of the software in which we do our work, but we do not see any benefits to working within that model for building programming systems.

To be clear, we are only explicitly stating a preference that is implicit but widely agreed upon: namely, that there is not usually a good reason to program in machine code when assembler or higher is available, or to use numeric literals when named constants are available. Anything involving relocating pointers, resizing structures or mapping names to numbers should be handled automatically and not occupy any of our

---

1 Of course, this is a description of the human experience; all the layers of representation including bytes and physical hardware are present, but do not demand the programmer's attention.

cognitive resources as programmers. There may be exceptions, but our point is precisely that they are not the common case.

We will see in Chapter 4 that building a self-sustainable system follows a path similar to the historical development of programming: we begin at a low level and build up more advanced features and conveniences. However, we do not think it is necessary to begin at quite the same low level as the historical case: in all our work, we will take the minimally human-friendly level as our pre-existing starting point without concern as to how it is implemented.

3.2   PARADIGMS OF PROGRAMS AND PROGRAMMING

The concept of a scientific "paradigm" was popularised by Kuhn (1970). It refers to the set of norms and conventions in which scientific questions are pursued and results are interpreted. In computing, a "programming paradigm" is a set of norms around the concepts and style in which programs are built.

In this section, we wish to broaden the scope and consider the foundational assumptions of programming itself, and the effects this has on how it is performed. A paradigm here is a set of norms and conventions centred around an idea of what a "program" is, what programs are *for*, and what is technologically feasible in the current environment.

We observe that certain periods of computing history and influential programming systems embody distinct paradigms in this way. We propose three: Batch Mode, Unix, and Interactive. Each one accommodates our Three Properties differently.

It is important to understand the Interactive Paradigm, whose assumptions pose the least resistance to achieving the Three Properties. Equally important is to be clear on the Unix Paradigm we currently inhabit, because we can only realistically achieve our goals from within it. However, the Unix Paradigm is best understood as inheriting from its predecessor, Batch-Mode, which we will introduce first.

Our goal in this section is to contrast the basic paradigmatic assumptions, and their consequences, between our current paradigm (Unix) and the one we would prefer for the work of this dissertation (Interactive). Because we see the Unix Paradigm as inheriting its key assumptions and consequences from its Batch-Mode predecessor, we will find it easier to introduce them in the latter's context. Thus we will explicitly describe the assumptions and consequences of Batch-Mode, continue with a discussion of how Unix augments Batch-Mode while retaining the same assumptions and consequences, and finally describe the assumptions and consequences of the Interactive Paradigm by comparison to those of Batch-Mode.

3.2.1 *The Batch-Mode Paradigm*

Computer programs originated as "batch-mode" processes. This is the manner in which a calculation proceeds and delivers a result at the end, or a compiler passes over source code and outputs a machine-level program.

In this paradigm, a program starts, runs, and then stops. The effect or "behaviour" of a program is its *output;* to change its behaviour, we need to change the executable program. But we cannot change the program binary directly; most nontrivial changes would invalidate various binary offsets, which would then have to be discovered and adjusted. Instead, we change its *source code*, and then *re-generate* the program as another batch-mode operation.

This poses no problem because any important effects of the program are outside it, whether on a paper printout or saved to magnetic tape. Any data structures the program creates occur within its "working memory", a temporary scratchpad internal to the program and discarded when it's done its job. This reflects the technological fact that storage comes in two varieties: one is fast but volatile (it loses its contents without a steady power supply), while the other is non-volatile but slow. Both are expensive, as is processing power.

3.2.1.1 *Assumptions and Consequences in Batch-Mode*

The above points can be distilled into the following assumptions:

1. *Program Outputs Result.* The point of a program is to output a result, such as a numerical calculation or data retrieved from records.

2. *Resource Scarcity.* Processing speed and storage (fast and slow) are scarce resources that we can only afford for essential tasks.

3. *Delete By Default.* There are only one or two data items we care about long-term (e. g. the result). Any intermediate steps taken to create the result are unimportant or uninteresting, so their working data structures should be discarded to free resources.

The three key consequences of these assumptions are:

1. *Run Time Is Volatile.* We implicitly design programs to run in the fast/volatile storage. This ensures performance and that the uninteresting intermediate state is not wastefully persisted.

2. *Few Things To Save.* For the one or two exceptional data items that we do care about (such as the output), we have to remember to write code to move these out of volatile memory and into non-volatile storage. This is not tricky, because there are only a few items we care about!
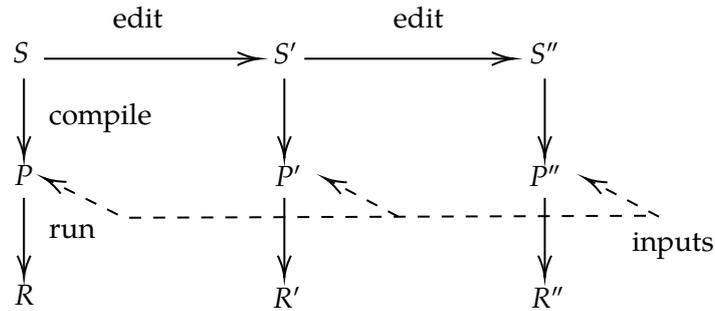
Figure 3.1: Change By Re-Creation: source code *S* is compiled into a program *P* which is run on some inputs to produce a result *R* signifying the observable "behaviour" of a program in the Batch-Mode paradigm. To change this behaviour under the same inputs, we must trace up the arrows to the source code and edit it into *S′*. From this, a new program *P′* is compiled, which is run to produce a new result *R′*, and so on.

3. *Run Time As Obstacle.* The time during which the program is running is an *obstacle* to us getting the result; a better program is one that terminates sooner.

3.2.1.2   *Compilers and Change By Re-Creation*

In Batch Mode, the programmer writes code in a high-level language and passes this through a *compiler* which outputs an executable program. The programmer can then make changes to the source code and compile a new executable. Strictly speaking, the old program executable is *replaced* with the newly generated one. Still, the continuity between the versions means "the program" as envisioned by the programmer is changing. In other words, Batch Mode facilitates *change by re-creation:* in order to change something, we trace its history to the process that generated it, change the source input, and then re-generate everything from that point onwards (Figure 3.1). This applies even if the change is small; consider how a single-character typo in one's LaTeX-typeset thesis is fixed by re-running LaTeX and re-typesetting the entire document. Splitting a project into multiple files, compiled separately, can help mitigate this, but only to a point; having one function per file or similar would clash with established programming and editing workflows.

3.2.1.3   *Static Commitment in Batch-Mode*

The stages of *compile time* and *run time* are of comparable importance in this paradigm. Properties that are *static*, or *early-bound*, are invariant over the entire run time of a program and are "baked in" at compile time. Those that are *dynamic* or *late-bound* may vary over the run time. Since a program has a specific answer to work out (Program Outputs Result), then the only properties that *need* to be dynamic are those that
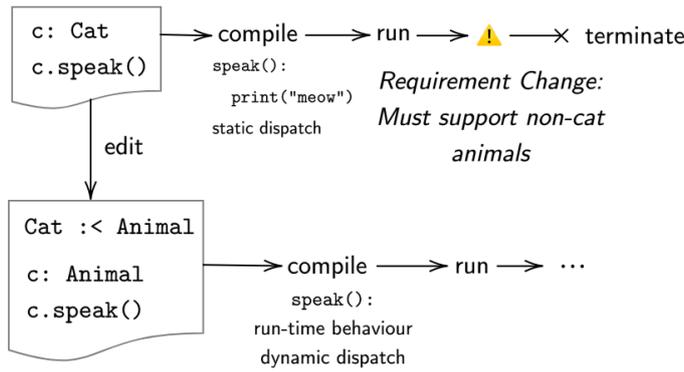
Figure 3.2: Source code is compiled and optimised under the promise that the
`speak` method always has the same behaviour (i. e. static dispatch).
If requirements change and invalidate this promise, the program
remains permanently committed to the promised condition. How-
ever, this is not a problem in the Batch-Mode Paradigm; the running
instance is left or forced to terminate and the program is changed
by re-creation (Section 3.2.1.2). A new program is compiled, this
time without commitment to such a promise; the behaviour of
`speak` now depends on which specific subclass of `Animal` the object
happens to have *at run time.*

directly pertain to such a process. All other properties are candidates
for *static commitment:* the compiler (suitably informed, e. g. by type
annotations) can assume they will never change, and hence can avoid
generating code to deal with the consequences of such changes.

   Beyond optimising the program's performance, this also gives pro-
grammers an opportunity to make mathematical guarantees about
certain properties. This all follows from "Run Time As Obstacle": a
program's destiny is to terminate with an answer, and a better program
terminates sooner. Requirements may change so that some formerly
static property now needs to be dynamic; for example, a behaviour that
was previously the same for all objects might now need to be dispatched
on the type of the object (Figure 3.2). In such a situation, the program
is re-written and re-compiled, ready to run under the new conditions.
In the event that the old version of the program happens to be running,
it can be deleted once it terminates.

3.2.1.4   *Key Legacies of Batch-Mode*

The batch-mode paradigm can be considered an appropriate adapta-
tion to the early conditions of computing: specialised, industrial or
research institution-scale use cases, and extreme scarcity in storage and
processing speed. Its legacy survives today in the following forms:

1. The *volatility split:* at all times, a programmer must remain aware
   of, and specify, whether their data is to be stored in volatile or

non-volatile storage. An example is the question "Does this belong in a class, or in the database?".[2] Variables and classes are easy to express in code, but are presumed to be transient; file and database access grants persistence, but is more complex to express.

2. The *change by re-creation* model: in order to change something, trace its history to the process that generated it, change the source input, and then re-generate everything from that point onwards.

3. Encouragement of *static commitment:* many innovations in programming take the form of improved ways to enforce static properties over a program's run time, such as type system features (Pierce 2002).

3.2.2 *The Unix Paradigm*

Unix is a family of Operating Systems dating from the 1970s. For this dissertation, we are not interested in the differences between Unix versions or descendants. Instead, we refer to the common set of concepts and conventions that form ordinary programming practice today, most importantly *files* and *processes.*

Here we have a *continuously* running master-program (the *kernel*) which allocates computation and storage to batch-mode programs under its supervision. Unix calls these *processes;* each one a sort of virtual processor plus working memory, which it calls "core". As in Batch-Mode, core is just a means for the process to do its job quickly, and is discarded upon termination. The other world, of "things that matter", is a hierarchical tree of *files* shared between processes and persisted as they come and go. Files and "pipes" serve as important inter-process communication mechanisms, and this *composability* of processes is an important part of the Unix philosophy.

Unix was created before the rise of GUIs and naturally preserved the batch-mode norm from its surroundings. Still, it contains early signs of what we will call the "interactive" paradigm in Section 3.2.3. Users can access the current state of processes and files via the command line *terminal* or *shell*. This runs as a process but is, like the kernel, continuously running, passing input and output between the user and the kernel.

There is one further small innovation worth noting. As an Operating System, Unix goes some of the way towards hiding the Volatility Split by *paging* core to disk (Babaoğlu et al. 1979), prioritising fast storage for processes that need it (the others are paused waiting for some event). However, Unix still sees processes as temporary scaffolding to be discarded when complete (Delete By Default). What this means is that

---

2 There do exist technologies like Object-Relational Mappers (ORM) which bridge the split, but this is optional infrastructure with its own costs. Its existence serves to highlight the significance of the Volatility Split.

even though Unix is clearly capable of persisting a process' data while it is still active, it will still be discarded when the process completes.

We see that the Volatility Split, Change By Re-creation, and Static Commitment were preserved from the Batch-Mode era in the processes and files within Unix. Let us summarise all this as the *Unix Paradigm:* a compromise between the batch-mode and interactive paradigms (described soon in Section 3.2.3) where limited interaction is used to organise batch-mode computation. For this reason, we consider the assumptions, consequences, and legacies of Batch-Mode (Sections 3.2.1.1 and 3.2.1.4) to be inherited by Unix as its assumptions, consequences and legacies.

3.2.2.1   *Unix as a Programming System*

We already gave a cursory analysis of Unix as a programming system in Section 2.2.2, on which we will now go into further detail. We note that Unix is an overarching system managing many smaller processes. To the extent that *other* programming systems are implemented as Unix processes, Unix functions as a meta-system supporting subordinate programming systems. Thus there are always *two* levels to programming in the Unix Paradigm: the "large" *inter-process* scope comprising processes and their communication, and the "small" *intra-process* scope within each process. In both of these, *state* consists of both core and files thanks to the Volatility Split. However, the two levels emphasise these to different degrees.

At the inter-process scope, it seems appropriate to describe the filesystem as the primary "state" (Section 3.1) of the Unix system. There is, of course, nontrivial state in core such as the currently running processes and bookkeeping information for them, which will be lost if the system is restarted. Nevertheless, from its own perspective, the "important data" for users all lives in the filesystem; what happens to be running at any given time in core is merely a means to an end.[3] The agent of *change* at the inter-process scope is the kernel, and the primitive "instructions" are the system calls used to write to files and change the file tree.

Meanwhile, at the small scope of a *process* embodying a programming system nested within Unix, the manifestation of state and change will depend on the particular system itself. Yet because we know it is running as a Unix process, we can at least be certain that the lowest level of "state" will be split between core and files, and "change" will occur through the execution of machine instructions.

---

3 "Few Things To Save" reminds us: if a data structure was important, the programmer would have written code to save it to a file!

3.2.3   *The Interactive Paradigm*

We define the *interactive paradigm* as the programming model that emerges from conditions of speed and storage abundance. Such abundance frees us from having to start with the question "what purposes can computers currently cope with?" and instead ask "what are computers for?" with the answer being roughly equivalent to "anything". This generality means that we should be careful to avoid embedding limiting assumptions in the infrastructure that supports programs and programming.

3.2.3.1   *Assumptions and Consequences of the Interactive Paradigm*

The Interactive paradigm relaxes or rejects the basic axioms of batch-mode programming:

1. "Program Outputs Result" is rejected. The point of a program is to simulate a piece of the world somehow useful to a human, but in a manner that is free of the constraints of physical media substances like paper. Producing an output is only one of many such effects useful to humans.

2. "Resource Scarcity" is rejected. Processing and storage are (or will be[4]) sufficiently abundant that we can use them generously in service of higher goals.

3. "Few Items To Save" is rejected. We do not know *a priori* and in full generality which data items we care about long-term, because the space of human purposes is large. For some tasks, the journey is more important than the destination.[5]

And the consequences:

1. Instead of "Delete By Default", we have "Persist Intermediate Data". We cannot *commit* on principle to saving some data and discarding or hiding other data. We persist everything by default and provide means to free up resources explicitly. We reluctantly abstain from this only where necessary for performance, treating this as a temporary optimisation to relax in future. But we cannot decide for the user what they will be interested in.

2. The Volatility Split is replaced with Volatility Obliviousness: code can simply create and manipulate data structures without the programmer needing to keep in mind what type of storage they live in.

---

4 This was the attitude at Xerox PARC summed up in the principle "design for the hardware of tomorrow".

5 An example of this is the Event Sourcing pattern (Fowler 2005; Orchard and Matthews 2008), where the history of state changes is recorded and accessible as a sequence.

3. Instead of Change By Re-creation, we have "Change by Chang-ing". It ought to be possible to change the state or behaviour of the system directly, as opposed to changing some upstream specifica-tion and re-creating the system from that. This is important both for performance and for avoiding premature deletion of data.

4. Instead of "Run Time As Obstacle", we have "Run Time Is Valu-able". It is no longer the case that a good program terminates quickly. There may not even be any reason for it to terminate at all. Where previously the run time was an inconvenient delay to getting the result, now the run time may be the *raison d'être* of the software, such as is the case with any interactive graphical application.

This paradigm is embodied in Lisp and Smalltalk, both for different reasons. Lisp originated before Unix as a language for mathematical and logical symbol manipulation in AI research; it makes sense that such a tool had little need to import batch-mode, industrial-scale computation as its primary concept. Smalltalk, on the other hand, deliberately re-jected this convention to serve its goal of shifting computing out of the industrial mode and into the personal. In neither do we find a manda-tory separation between volatile and non-volatile storage, nor between "large objects" (files and processes) and "small objects" (variables and code). Instead we find a graph of data structures, called "expressions" in Lisp and "objects" in Smalltalk.

3.2.4 *Batch-Mode Anachronisms*

The relaxed assumptions make sense in today's computing environment with plentiful processing and storage. However, the Unix Paradigm preserves the assumptions and consequences of Batch Mode. From this perspective, the three Batch Mode legacies feel obsolete.

3.2.4.1 *Volatility Split*

The Volatility Split treats what ought to be an *implementation detail* as a major design concern. The question of whether the variable x should live in the physical form of RAM cells, magnetic domains, or flash storage cells may not even be knowable by the programmer. It resembles the question of at which precise address a new string buffer should live in memory; a function like `malloc()` removes the need for the programmer to "manually" decide such an implementation detail, and this is a good thing.[6] Similarly, the storage medium of a variable should surely be

---

6 Even though this is known as "manual" memory management, the "manual" refers to the management of allocation *lifetime*, in contrast to *garbage collection* which auto-matically decides when to free memory. The actual allocation of memory via a call to `malloc()` is automatic by the very fact that it is a subroutine being run on a computer.

deferred to some automatic mechanism; compare the common adage that warns against prematurely optimising one's code: "the compiler is smarter than you are (in this regard)!" The critical fact at work here is that a human programmer is limited to the following two options:

1. Proclaim a blanket policy for the entirety of run time (the data named x shall reside in volatile memory, full stop)

2. Instruct the program to make the decision based on run-time conditions (an algorithm to cache x in fast memory but also persist it to disk)

Many languages in the Unix Paradigm have semantics such that declaring a variable, with convenient syntax, serves as the programmer taking the first option. In the cases of compiler optimisations and the choice of allocated memory *location,* it is recognised that the *second* option should be expected to deliver better results. The same logic argues for the *type* of storage to be treated the same way, making the Volatility Split something the programmer never has to think about.

3.2.4.2   *Change By Re-creation*

Re-creation seems like an unnecessarily convoluted path to achieve a simple change. Suppose we want the field foo of object greeter to change to "Hello". The system knows the object referred to by the name greeter and knows how the field foo is stored. It seems odd to have to dig through code and re-generate the object along with everything it touched, a practice Bret Victor dubbed "destroy-the-world programming" (Victor 2012); surely we ought be able to just type greeter.foo = "Hello" and have the property update accordingly in the running system.

Change By Re-creation may be justifiably retained as an *implementation* strategy hidden from the user's concern, so long as the data loss from terminating the current process is mitigated. An example of this is the method proposed by Basman et al. (2016) under the name "Queen of Sheba Adaptation", in which data is persisted and the overhead of restarting is hidden from observers.

3.2.4.3   *Static Commitment*

Static commitment mechanisms, such as type systems, now play a much less helpful role—even an obstructive one—compared to the Batch-Mode world. If a program's value derives from its behaviour as an *interactive system,* it may well be running for a long time or even (ideally) forever. This intensifies any disadvantages of static commitment mechanisms like type systems.

The immediate problem is that if requirements change and a commitment must be relaxed to vary at run time, we are forced to terminate

the running system and re-generate it. Unlike the Batch-Mode case (Section 3.2.1.3), we cannot simply wait for the program to run its course and terminate, and if we do force it to terminate we risk losing important transient state.

The more fundamental problem is that a mandatory commitment might be clearly *premature* from the perspective of the user. A type commitment like "Type X shall always be a subtype of Y" may be too restrictive, too soon. Consider a game in which the class `Goblin` is a subclass of `Enemy`. In a language like C++, class relationships are statically enforced. This prevents the player from befriending Goblins later in the game, as we cannot write code to change a `Goblin` instance to inherit from `Friend` at an appropriate point during run time. This means that the natural means of expressing relationships in the language must be abandoned in favour of an ad-hoc replacement with dynamic capabilities and no syntactic sugar. We will discuss this further in Section 6.3.

It must be stressed that there is still a role for commitments and optimisations, but it is *piecemeal* at smaller scales *during* run time. As the duration of the program's run time increases, the set of properties one might wish to statically commit to shrinks correspondingly, for the simple reason that the likelihood of a change in requirements increases. In the extreme, consider a system which is meant to run forever (or "indefinitely"); we would want to be very careful about any commitments we allow the compiler to embed into the fabric of the program, since if they were unintentional or turned out to be mistaken, we could only correct the problem by terminating the system.

For an analogy, consider a single program with a specific purpose in a Unix environment. If its requirements change, the program is re-compiled, while the rest of the system *does not need to be affected.* This is because any "static" commitments with which it was compiled apply to the run time of the individual Unix *process.* Now suppose that instead, the *entire* Unix environment was assembled incorporating static commitments about this program, where these commitments were scoped to the lifetime of the *entire environment.* Then, when requirements for the program change, the whole Unix environment must be re-compiled and *re-installed*; it will not do to merely replace the program, because the rest of the environment was compiled and optimised with the now-invalidated commitments in mind.

This is an extreme hypothetical, but it is analogous to what can happen when ordinary process-level static commitment is employed for a long-running, interactive or open-ended process such as a game. Any property whose change during run time cannot be *ruled out* would be unwise to enforce as a process-level static commitment. Technologies like hot-swapping or Dynamic Code Evolution (Würthinger et al. 2013) bring the benefits of commitment and optimisation to such systems at a scale that is more appropriate for them.

3.2.5 *Conclusion*

As examined by Gabriel (1991) and Kell (2013), Unix "won" in a way
that Lisp and Smalltalk did not, firmly establishing the Unix Paradigm
as ubiquitous. Where Lisp and Smalltalk exist, they are processes sitting
within Unix and saving to "image" files. For implementors of novel pro-
gramming systems, the tenacious Volatility Split, Change By Re-creation
model and Static Commitments clash with the Interactive Paradigm
natural to the enterprise. The takeaway for this dissertation is that if we
wish to build our system in the Unix paradigm, we must be mindful of
its shortcomings relative to the ideal Interactive paradigm and expect
part of our work to involve mitigating them.

3.3 THE THREE PROPERTIES IN MORE DETAIL

In Section 1.5 we gave introductory definitions for Self-Sustainability,
Notational Freedom, and Explicit Structure. Now it is time to go into
more detail and examine them in light of the paradigms we identified.

3.3.1 *Self-Sustainability*

Self-sustainability involves being able to evolve and re-program a sys-
tem, using itself, while it is running. At the upper limit of this would re-
side "stem cell"-like systems: those which can be progressively evolved
to arbitrary behaviour without having to step outside of the system to
a lower implementation level. Any difference between these systems
would be merely a difference in current state, since any could be turned
into any other.

The lower limit, of minimal self-sustainability, looks something like
the following: beyond the transient run-time state changes that make up
the user level of any piece of software, the user cannot change anything
without dropping down to the implementation level. This would resem-
ble a traditional end-user "application" focused on a narrow domain
with no means to do anything else.

At a sufficiently large scope, self-sustainability is inevitable. By anal-
ogy, while any nation's economy might be dependent on other na-
tions, the world economy is a closed system that provides its own in-
puts.[7] Similarly, the ecosystem of software as a whole is necessarily self-
sustainable. Even an *individual* Unix system is largely self-sustainable at
its inter-process scope, but it is notable that we lose self-sustainability
on the way down to the intra-process scope. We will discuss these two
scopes as they relate to self-sustainability, after which we will distin-
guish the *user level* and *implementation level* of programming systems.

---

7 Disregarding energy from the Sun, chemical elements contributed from past asteroid
impacts, etc.

Then we will conclude with a definition of *innovation feedback,* the key advantage a self-sustainable system has over others. For further information and motivation on Self-Sustainability beyond our own analysis here, we recommend the introductory sections in Piumarta (2006) and Piumarta and Warth (2008) and the vision presented by Cook (2018).

3.3.1.1   *Self-Sustainability at the Inter-Process Scope*

At the inter-process scope of Unix, we have individual processes—text editors, compilers, interpreters, debuggers—which change the large-scale system state (files), such as by creating new programs. Some of these processes run *shell scripts* to coordinate this activity, this being the de-facto programming language[8] at the inter-process scope. In this way, a Unix system is evolved and re-programmed using itself, while it is running. Hence, Unix (i. e. programming-in-the-large) is self-sustainable, which is congruent with the origins of Unix as a system for programmers.

The matter is complicated by the fact that a small minority of special changes require restarting the system to take effect. However, the spirit of the "while it is running" condition is that the system does not need to be *destroyed and rebuilt from scratch*. Because the "state" of the inter-process scope is mainly files, the destructive operation here is not so much "restart" as perhaps "reset" or "reinstall".

In contrast, for a programming system that exists as a process *within* Unix, its data structures in volatile memory will be permanently lost if it is restarted, and these data structures may well be an important part of its state as a continuously running interactive programming system. Indeed, when we turn our attention to the intra-process scope of the Unix Paradigm, we mostly do not see self-sustainability.

3.3.1.2   *Self-Sustainability at the Intra-Process Scope*

Compiled programming languages like C++ or Java are used via several different Unix processes. These include interactive ones like text editors, and batch-mode ones like the compiler. Self-sustainability ought to be the analogue of the properties in 2.3.1 for interactive programming systems. To recap, these were self-hosting compilers, the bootstrapping thereof, meta-circular interpreters, and reflection.

The self-hosting compiler is how self-sustainability manifests in the batch-mode world, where the memory used by the process is disposable and unimportant, hence Change by Re-creation does not cause too many problems. For interactive programming systems, this is often inappropriate or at least highly inconvenient, so it is hard to base full self-sustainability on self-hosting.

---

8 Technically, the various shell *dialects* (bash, csh, etc.) form multiple de-facto *languages,* but this is not important for the point.

In this respect, interactive systems more closely resemble *interpreters* than compilers. The job of a compiler is to generate a new program, which could (in principle) be a *replacement* for the old compiler. Moreover, both the new program and its source code live in the filesystem, i. e. non-volatile storage. The job of an interpreter is simply to *execute* a program, and by default any changes we make to its state as a result of the code we feed it will not survive process termination. Unlike interpreters, interactive processes like REPLs or programming systems are meant to run as long as the user wishes and contain a lot of important state in memory.

Traditionally, the Volatility Split was just forwarded into the user's mental model, making no guarantees about whether work would be saved in the event of a crash and recommending the user to save regularly. End-user applications did eventually implement "auto-save", but this is a feature programmed in to preserve specific user data. For programmers, "auto-save" of run-time state involves infrastructure that takes considerable work to implement, owing to "Delete By Default". Therefore, re-creating the system risks important data loss. Even if such infrastructure is present, restarting the system to make a change may interrupt the user experience with an inconvenient delay.

Perhaps instead of the above compiler-related precursors, we can adapt the interpreter-related precursors, meta-circularity and reflection. Meta-circularity will only aid us if the inner interpreter is causally connected to the outer one via reflection. Therefore, out of the precursors we have identified, reflection is the most suitable one to develop into self-sustainability; particularly as it manifests in Self (Bracha and Ungar 2004; Ungar and Smith 2007) rather than, say, Java. This could be achieved if reflection is scaled up to encompass as much of the system as possible and if reflected data can be changed rather than simply observed. It would also be necessary to protect any changes achieved through reflection from being lost upon process termination. In short, self-sustainability takes much from reflection in interpreters but combines this with the persistent evolution of the self-hosting compiler.

3.3.1.3   *User vs. Implementation Levels*

For any piece of software, there are two levels:

- The *user level* is where software is used for its intended purpose by its target audience. For example, the user level of Firefox involves browsing websites.

- The *implementation level* is where the software is created and changed in ways unavailable at the user level. As a trivial example, by taking all of the source code of Firefox and replacing it with the code for Hello World, a programmer can change Firefox into a Hello World program.
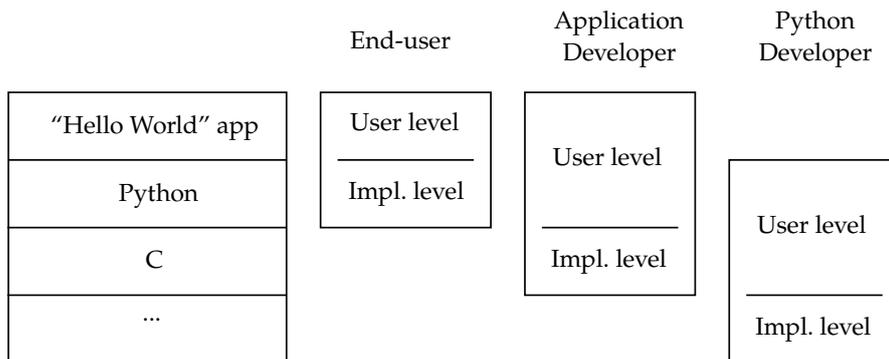
Figure 3.3: Relativity of user versus implementation level depending on one's role.

If the software is a programming system, then this can get confusing: both levels involve programming! Consider this example situation: someone is using Python to write a Hello World program, and the programming system (Python interpreter plus editor) is written in C. We can view this situation in three ways depending on the "user" (Figure 3.3):

1. We can focus on the user of the Hello World program. The user level is Hello World, while the implementation level involves Python.

2. We can focus on the programmer as the user of Python. The user level involves Python *and* the Hello World program (for testing, debugging, and so on) while the implementation level involves C.

3. We can focus on a different programmer who works on the Python implementation. The user level involves C *and* Python (for testing, debugging, and so on). Even though there are further implementation levels below, we short-cut the analysis here and leave them unspecified.

In this dissertation, we are interested in *building* programming systems with the Three Properties: Self-Sustainability, Notational Freedom and Explicit Structure. This means we occupy the third viewpoint, where we are detached from any particular end-user program that might get created. We see our situation as follows:

- We are using some already-existing programming system (in our example above, this was C). We did not create it and we do not expect to be able to change it. We call this the *platform*.

- The programming system we create using the platform is called the *product system* or simply "the system" (Python, in our example).

3.3.1.4   *Platforms and Substrates*

Because we seek to build a product system that is Self-Sustainable, the picture becomes more complicated. The point of Self-Sustainability is to blur the distinction between the implementation and user levels. Not only can the system be used to create ordinary programs, but it can also be used to change itself. We use the term *in-system* to refer to changes made within the product system, by using it as a programming system at its user level.

In the Platonic ideal self-sustainable system, there is no distinction between the two levels at all. In practice, the best we can do is attempt to *minimise* the implementation level to a tiny core:[9] everything else can be changed in-system. In this case, we call this tiny core the *substrate*. Our task as implementors is to use an established platform to write a minimal substrate that can then support a self-sustainable system. Almost all aspects of the system can then be changed in-system, and the rest must be changed in the substrate using the platform.

To summarise the picture in the self-sustainable case (Figure 3.4):

- The *platform* is the already-existing programming system that we accept as-is and do not expect to have much control over: for example, the C programming language.

- The *substrate* consists of the code we write in the platform. We exercise control over it, but we do not expect it to be accessible in-system, which is why we seek to minimise it. An example would be a C Virtual Machine for Smalltalk.

- The *product system* is the programming system supported by the substrate, such as Smalltalk. Ideally, a tiny part of it is implemented in the substrate while most of it is implemented in itself. In other words, most changes to the system can be made using the running system (they are *self-supplied*, Definition 4) while only a few may require modifying the substrate and restarting or re-compiling the running system.

For ordinary software, there is no reason to distinguish the platform and the substrate; the two of them together constitute the implementation level. For a self-sustainable system this distinction is necessary, because the "implementation level" extends into the product system itself (ideally being concentrated there).

---

9   In the limit, we will leave the world of software only to hit the non-malleable world of physical hardware. Still, search "FPGA" in (Piumarta 2006) for its speculations on pushing this as far as it can go (see the captions for Figure 3 and Figure 14 and page 23.)
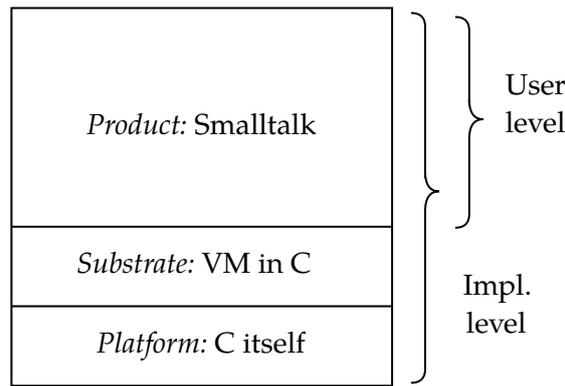
Figure 3.4: Example platform (C) supporting a substrate (Smalltalk VM) for a self-sustainable product system (Smalltalk). Because the product is self-sustainable, the user and implementation levels are no longer disjoint, so the platform/substrate/product distinction is a more helpful alternative.

3.3.1.5   *The Key Benefit: Innovation Feedback*

A system that is self-sustainable has an advantage over those that are not: we call this *innovation feedback*. Innovations programmed using the system—useful functions, notations, or tools—can benefit their *own* development as well as the rest of the system. In contrast, for a system whose internals cannot be affected by any program within it, innovations can only be exploited for the system's development by *duplicating* the work at the system's implementation level.

For example, at the inter-process scope of Unix, a text editor may be used to improve its own source code, which can then be compiled into an even better text editor. This improved editor can then edit its own source code and begin a new cycle of self-improvement. The shell interface, graphical interfaces and all tools are just *programs* which can be replaced with newly compiled improvements, all using other Unix programs. Therefore, a Unix system is not limited to improving detached *separate* distributions of programs destined for a different user, but naturally improves itself as well.

When put this way, it sounds obvious; of course computer software is used to improve computer software. Yet how different it is at the intra-process scope: the very same text editor *on its own* cannot compile its improved replacement. A Python interpreter written in C may empower us to create useful Python functions, but to improve the interpreter itself we need to inhabit the world of C, in which our Python functions are unavailable. Figure 3.5 contrasts this situation with innovation feedback in Unix.

From our presentation, it may appear that self-sustainability is a peculiar special case that is naturally hard to achieve. Yet the world of software as a whole is self-sustainable, as is an individual Unix system. We conjecture that self-sustainability is a natural default that
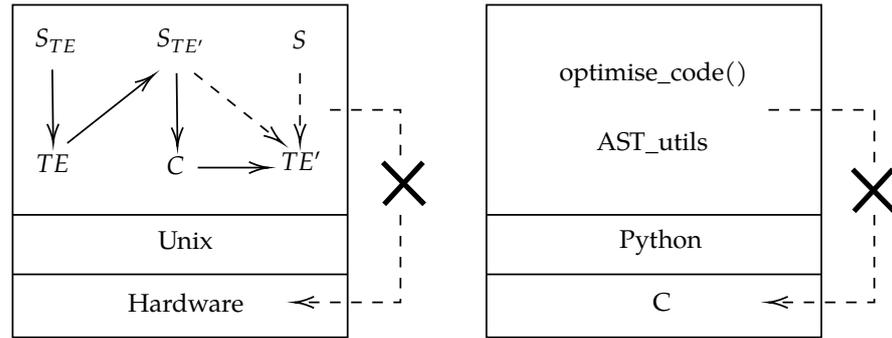
Figure 3.5: Software innovations within a Unix system (left) cannot feed back into its hardware platform. However, the software innovations can feed into each other: a text editor *TE* edits its source code S_{TE}. This new source $S_{TE'}$ is put through the compiler *C* to create an improved text editor TE', which can edit not only the source code *S* of other programs but also its own. In a Python system (right), Python innovations can assist in the Python world but cannot feed back to assist with the C implementation of Python.

was *prevented* by the historical contingency of Unix enforcing Batch-Mode assumptions on intra-process programming. Nevertheless, we accept the Unix Paradigm as a given, and accomplish the tasks of this dissertation on that basis.

3.3.2 *Notational Freedom*

In Section 2.3.2 we mentioned the maxim "Use The Right Tool For The Job". This is a noble aspiration, but one that is not currently fulfilled. At the "large scope" of the Unix paradigm, there is a vast array of programming languages specialised for different jobs. Yet we saw some barriers to Polyglot Programming, such as the need for inter-process communication. As we scale down to *within* a single process and forego inter-process communication, data sharing between languages gets thornier (Section 2.3.2.2).

The ideal is where a component at any scale can be expressed in a *notation* that is particularly suited to it. Here, we will define the lesser stages of *syntactic* and *linguistic* freedom before arriving at full notational freedom.

3.3.2.1 *Caveat on Subjective Value Judgements*

Before proceeding, it is important to stress the fact that Notational Freedom is about supporting *subjectively* appropriate notations. To explain the concept, we will give some examples of notations and judge them as better, worse, convenient, unwieldy, and so on. When we make these judgements, we are not claiming them as objective facts; nor are we even claiming them subjectively across all possible use cases. We are simply

giving concrete examples of *possible* preferences and using these to illustrate our points. That being said, we have not chosen these examples at random: they are in line with our preferences and plausibly shared by others. If the reader does not share the preferences in these examples, a hypothetical reading may be useful ("*supposing that* a programmer had such a preference…"). With this in mind, we will turn to the first step of *syntactic freedom.*

3.3.2.2   *Syntactic Freedom*

We noted in Section 2.3.2 that, within a single file, combinations of different languages are occasionally possible. Yet these are fixed sets, pre-approved by language designers and set in stone for all contexts and users. We could call this property *syntactic[10] plurality*, where a language named A additionally permits syntaxes B, C, and D in certain situations. This goes a small way towards "Use The Right Tool For The Job", but what is conspicuously missing is the ability for the *programmer* to decide which syntaxes to use, and where, based on their local context.

This would be syntactic *freedom* beyond pre-approved plurality. It is not unreasonable to expect that a single function, or perhaps even a single line of code, might be best expressed in a different language to the rest of the code, in a way that the language designer cannot anticipate.

We will use a running example as we build up through the stages to full Notational Freedom. A common occurrence in scientific or graphics programming is a few lines of mathematical operations. It would be convenient to bring these closer to familiar mathematical notation instead of an unwieldy ASCII approximation. Suppose we start with the following notation for a formula:

```
vec_a.mul(cos(ang_b/2))
    .add(vec_b.mul(cos(ang_a/2)))
    .add(vec_a.cross(vec_b))
```

Syntactic freedom would mean being able to see this situation and specify a local syntax that lets us rewrite it as:

```
cos(ang_b/2) vec_a + cos(ang_a/2) vec_b + vec_a × vec_b
```

Languages provide specific, limited freedoms in this regard; consider C++'s operator overloading or Haskell's arbitrary infix operators. However, these specific facilities do not amount to full syntactic freedom.

We are aware of true syntactic freedom in three places: COLA's Mood-Specific Languages (MSLs), with the related OMeta framework of Warth (2009); Lisp's Reader Macros; and Haskell's quasiquotes (Mainland

---

10 Programming languages differ by semantics as well, but for Notational Freedom we are only interested in the surface notational aspects. Infrastructure for supporting this would already be close to supporting freedom of semantics, as shown by COLA's mood-specific languages.

2007). These are to be commended for a significant step in the right direction, but the syntax of strings as defined by formal grammars has limitations. It can only see two directions (left and right) and has no notion of display variations like typefaces, weights, sizes, colours and so on. This encourages us to edit expressions as lists of characters without support for nested boxes or other interfaces that can be useful. In other words, it keeps us in the text editor interface with its Implicit Structure and associated problems (see Section 3.3.3 below). If we go on to lift these restrictions, we arrive at *linguistic* freedom.

3.3.2.3 *Linguistic Freedom*

By this, we mean freedom to represent expressions as arbitrary written *language* rather than restricted *syntax.* In the physical world, written language takes many forms which are hard to digitise in their full detail—we don't expect our personal handwriting to be adopted as an internationally-recognised font. Yet even in computing, written language takes a variety of forms and supports a variety of display characteristics. In programming, these are stripped away and we generally only have formatless *syntax* to work with.[11]

In our running mathematical example, there is a very good illustration of the step up to linguistic freedom in the form of publishing-standard mathematical notation (and conveniently for this document, a core capability of LaTeX):

$$\cos\left(\frac{b}{2}\right)\mathbf{a} + \cos\left(\frac{a}{2}\right)\mathbf{b} + \mathbf{a} \times \mathbf{b}$$

This exhibits the following improvements (from the perspective of established norms in mathematical notation) that do not fit into the "syntax" technologies:

- Vertical layout of fractions

- Replacing the `vec_a` and `ang_a` with **a** and *a*, distinguished by weight

- No restriction to fixed-width characters or spaces

Beyond these display characteristics, it also leaves open the possibility of an editing interface not restricted to character-by-character string operations. For LaTeX mathematics, we must often write verbose textual source in the manner of Section 3.3.2.2's example and render it

---

11 Technically, syntax highlighters make keywords bold and add various colours, but these are fixed rules applying to display only. The point is that it is not possible to create a "bold variable" or a "red function".

into the better notation. Yet there do exist interfaces for editing mathematics more directly, such as those in MS Word, Mathcha[12], Desmos[13], and Wolfram Alpha[14]. Linguistic freedom permits such "structured" or "projectional" editing as an option. This brings to mind JetBrains' MPS (Voelter and Pech 2012) and Eco (Diekmann and Tratt 2014) as the only systems of which we are aware that are designed to specifically offer Linguistic Freedom.

For our running example, we have plausibly reached the Right Tool For The Job at this point. In general, however, we still regard "language" as too narrow of a constraint. By this, we simply mean notations that consist of repeated glyph shapes laid out in a (mostly) linear manner. This is different to other uses of the term: "visual language" may not include text at all, but uses the "language" term for the arbitrary arrangement of elements. Under this latter meaning, the "mood-specific language" idea has all the generality it deserves. Nevertheless, in programming, we think the term "language" runs the risk of mentally excluding graphical or interactive possibilities. To ensure they remain, our use of the word "language" will stick to denoting mostly-linear renderings of glyphs and we shall use words like "notation" or "interface" for the fully general extension.

3.3.2.4 *Full Notational Freedom*

Here, we wish to have unrestricted support for graphics and interaction. In our mathematical example, full notational freedom would support an interactive 3D visualisation of the vectors involved if that was what the programmer desired. Language isn't everything—diagrams and pictures are sometimes the form in which a problem or solution is delivered, and programmers ought not to be forced to describe pictures using words. We should emphasise that we are using the terms "notation" and "interface" interchangeably; we are not just talking about static pictures but dynamic entities on a screen. The "text editor" interface is one such example. One could use a text editor to work on the hex code `0xff00ff` representing the colour magenta. Alternatively, one could use a colour picker interface.

Programming systems that embody full Notational Freedom are rare, and tend to leverage Self-Sustainability in its service. Contrast the cases of the Eco multi-language editor (Diekmann and Tratt 2014) and the Glamorous Toolkit (Chiş et al. 2015). In the former case, the ability to insert graphics into the editor does exist, but as an experimental capability that is not a typical use case, and the system is not self-sustainable. In contrast, the Glamorous Toolkit was deliberately designed for "Moldable Development", part of which involves support-

---

12 https://www.mathcha.io/
13 https://www.desmos.com/
14 https://www.wolframalpha.com/
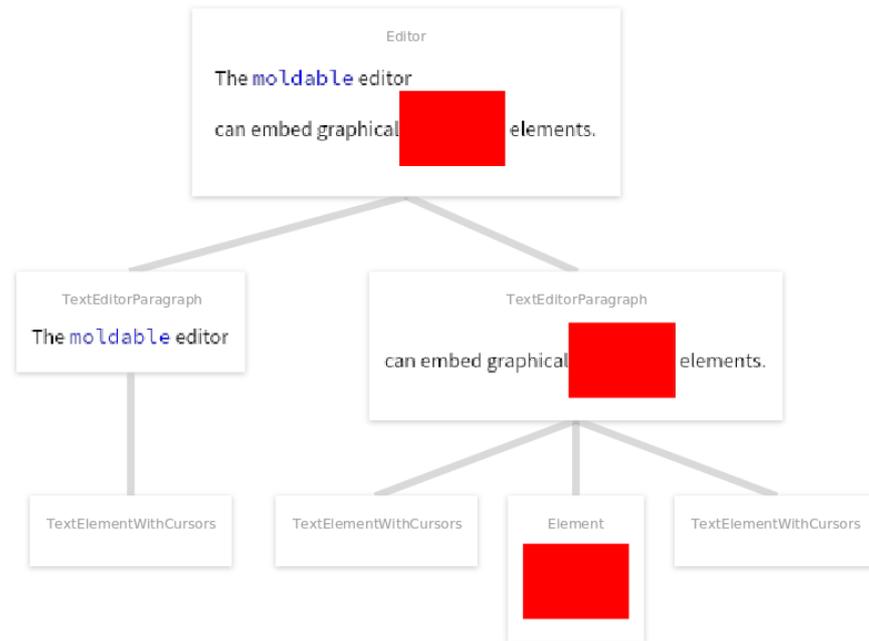
Figure 3.6: The Pharo Smalltalk-based Glamorous Toolkit supports Notational Freedom in a deliberate manner, as evidenced by this picture from its documentation.[15]

ing Notational Freedom (Figure 3.6); no doubt, its descent from Pharo Smalltalk was instrumental for this. Similarly, ThingLab I (Borning 1981) and II (Moloney et al. 1989) clearly support arbitrary diagrammatic notations, though without an emphasis on integration with the Smalltalk code editor. Babylonian Programming (Rauch et al. 2019) shows a plurality of notations integrated into the code editor for the purpose of live examples, and is built on the self-sustainable Lively4 system (Lincke et al. 2017). These non-Eco examples lend credence to the pragmatic necessity of Self-Sustainability if one wants arbitrary Notational Freedom. Even in the limited context of *syntactic* freedom, this connection is evident in *Helvetia*, a polyglot framework for Smalltalk (Renggli and Gîrba 2009), in addition to the previously-cited OMeta (Warth 2009) and COLA (Piumarta 2006).

The most important point to keep in mind is that this property is called Notational *Freedom*, rather than something like "Optimal Notation". We recognise that different notations suit different purposes and respect the art of developing them as a separate area of expertise, which we do not claim for ourselves. The idea is to support the *subjective* productivity of the programmer, who we assume is best equipped to judge the appropriateness of notations for herself. This property is about *supporting* the usage of different notations for different contexts.

---

15 https://book.gtoolkit.com/graphical-stack-ejn67l1cykgfjrl9aklgriubv
    Retrieved 02/2024.

3.3.2.5 *What It Means to "Support" Local Notations*

It is true that there is no such thing as a free lunch; we do not go so far as to suggest that the system should turn a natural language description into a working interface (recent advances in AI notwithstanding). So long as the programmer is willing to do the necessary work to program a new notation—such as writing a grammar, specifying the layout of symbols, or implementing the rendering and input handling for an interface—this should suffice to use it in harmony with all the other available notations.

However, the Unix Paradigm and text editors impose an *additional* "tax" in terms of effort beyond this reasonable standard. In the best case, there may be some plugin architecture, while at worst it may require forking the editor's source code. The actual work involved in *creating* the notation might be very small, but it will be dwarfed by the task of getting the editor to accept it. Notational Freedom does not come by default, and most editors and programming systems are not designed with it in mind.

A system with Notational Freedom is designed *without* the assumption that there will only be one notation and lacks these taxes to the extent possible. In short, by "support" we mainly mean the removal of artificial *barriers* that stand in the way of mood-specific notations in ordinary programming systems. Thus, it is natural that Notational Freedom and Self-Sustainability go hand-in-hand.

3.3.3 *Explicit Structure*

As we remarked in Section 2.3.3, Explicit Structure refers to the sense of working with data *directly* rather than through some other medium. To go into more detail, it will be useful to split the life-cycle of a data structure into two halves:

- On the *producer side*, the data structure is created or edited using some interface.

- On the *consumer side*, a programmer is writing code that uses the data structure.

Explicit Structure is hard to define positively because it is the default state of affairs across much of computing, with programming being the notable exception. On the producer side, explicit structure is exhibited by a vector graphics editor like Inkscape: one simply draws a diagram with shapes and saves it as a file. On the consumer side, Explicit Structure looks like a programmer navigating through named parts of the diagram structure:

```
svg.root_nodes[1].children[2].fill_color = '#ff00ff';
```

Explicit Structure is perhaps easier to define negatively, as a *lack* of Implicit Structure. Implicit Structure is present when we use *plain text* (or Qualitative Syntax more generally, as in null-terminated C strings) as a communication or storage medium: the structure can only be navigated after parsing the string. The problem with plain text is that it fuses together two independent concepts: what we could call *presentation* (how one reads and modifies the data) and *representation* (how the data is stored in memory or on disk). For example, the AST of a program *could* be stored in its tree form and *presented* as indented text. But what we do instead is serialise the text, store that, and parse it back before we are able to work with it.

In our work, we approach the default Implicit Structure of programming with skepticism. We direct the reader to the fuller arguments from the authors of Subtext (Edwards 2005) and Infra (Hall 2017), but we will summarise the most important points here and offer some of our own.

3.3.3.1   *Language Can Be Stored Differently to a Character List*

Language always contains *structure*: English paragraphs contain sentences, containing clauses, containing words. Natural language, like English, is typically entered into a digital medium only to be poured out again at some other end, like photo uploads; normally, the computer does not need to dive into the structure at all. On the other hand, for programming languages, the Abstract Syntax Tree structure is the entire point.

Despite this, programming language source code is universally stored as a sequence of characters, rather than as the tree or graph that it represents. This has the downside that every program that consumes or transforms the code must recover (parse) this structure out, discovering any mistakes only at this point of consumption (since these were just recorded with the other characters). This is comparable to storing vector graphics diagrams as arrays of pixels and using Computer Vision to haphazardly recognise shapes and lines: unnecessary work to recover information that was thrown away at creation time.

More unfortunate work results from having to "escape" characters that have been reserved to denote structure instead of their literal selves. In the worst case, the storage of language as character lists is largely responsible for the class of attacks known as SQL injection. This would not be possible with SQL commands represented as trees containing holes to be filled with user-submitted strings.

Consider the common practice of embedding SQL commands in the source code of various languages. In C#, these are forced into the syntax of the host language as "embedded queries", yet a programmer may prefer to use SQL syntax directly as part of the source. The traditional path-of-least-resistance to achieving the latter was to have SQL code in-

side program strings, which created significant security risks. This is by no means intrinsic to having SQL source be what the programmer *types* or *sees*; it is entirely possible to combine a text editing user experience with an explicitly structured in-memory or disk representation.

3.3.3.2 *Digital "plain text" is not inherently human-readable*

This argument is made best in Hall 2017, p. 14 which deserves quoting here:

> The critical observation is that software infrastructure is heavily involved in supporting the human-readability of text. It is not the case that the bit sequences of UTF8 or any other text encodings are somehow intrinsically understandable to a human. An application interprets the bytes as character codes as per a known standard, which are mapped to glyphs in a font, and rendered to a grid of pixels. This chain of interpretation and transformation starts with clusters of electrons and ends with clusters of photons before the human nervous system takes over. The point being that there is still a necessary software layer performing a transformation in the middle.
>
> electrons → bits → charactercodes → glyphs → pixels → photons
>
> "Human readability" just colloquially implies that it is a standard encoding understood by most text editors. The sense of inherent readability merely comes from the ready availability of tools that render ASCII and Unicode. One can assume that a text editor or some text rendering infrastructure exists in the target system. Therefore, any encoding could technically achieve the same 'human readable' status as ASCII if it and its editors were general-purpose enough to warrant an equally ubiquitous install base. Thus, there is an opportunity to expand or upgrade the realm of what can be considered human readable.

It can be tempting to defend text files as human-readable in contrast to binary files which are not. However, as Hall's argument shows, this is mere *status quo* bias: all of the "human-readability" of a text file is due to the wide availability of text editors. Few can read text through a binary or hexadecimal rendering of character codes, which is the basis on which a fair comparison with "binary files" must be made. By the same criteria, binary PDF files are human-readable owing to the ubiquity of PDF readers. In other words, the difference between plain text and "binary" data is not so much an essential difference of two kinds but simply a difference in tool availability.

3.3.3.3 *We Study the Spherical Cow*

Text usually functions as a *medium* or rendering of something that is not inherently text. In the first place, text was invented to record speech made by human beings. In programming, text is used as a *proxy* for a nested tree-like structure, but is *not the structure itself.*

To study a programming idea like self-sustainability, it is unfortunate to have the accidental complexities of text representation "getting in the way" of studying the idea itself. Even though a real-world programming system may use text, we wish to avoid this obscuring layer for much the same reason as a physicist studies a frictionless sphere in a vacuum instead of a cow in a field. We want to not be distracted with air resistance and complex shapes, so as to focus on the property we are interested in; future work can then add the practical complexities back in again for a more realistic model. In short, we study Self-Sustainability and Notational Freedom *directly* as properties of interactive graphical systems, and Explicit Structure is necessary for this.

3.4   CONCLUSIONS

There are systems that exhibit one or two of our Three Properties, but each has shortcomings. Infra (Hall 2017) and Subtext (Edwards 2005) thoroughly exploit Explicit Structure; the former develops the other two properties to a small extent, while the latter leaves them out of scope. Eco (Diekmann and Tratt 2014) supports Notational Freedom, but as an editor, does not form a complete programming system and lacks self-sustainability. Meanwhile, Smalltalk-based systems like the Glamorous Toolkit and ThingLab show strong potential for Notational Freedom as a result of their self-sustainability. JetBrains' MPS (Voelter and Pech 2012) possibly goes the furthest in promising Linguistic Freedom atop a base of Explicit Structure, and is even partly developed using itself as evidenced by its GitHub language metrics.[16]

However, MPS has the considerable resources of a software company behind it and is designed to be industrially viable. Similarly, Smalltalk descendants are fully-functional complex systems designed for utility in research and industry. Our goal is *not* to compete with these existing systems; rather, we think our Three Properties are, in essence, small enough to be realisable by an individual for personal use. This is why we pursue a general technique for *adding* these properties to a programming system that lacks them (Chapter 4). Such a contribution could suit a wider range of contexts than a requirement to use a specific existing system like MPS or Smalltalk.

COLA, by far the most important influence on the work in this dissertation, promises Self-Sustainability and restricted Notational Freedom

---

16 The "languages" section at https://github.com/JetBrains/MPS shows MPS covering 78.5% of the code as of March 2024.

in the form of MSLs. However, MSL support is presented in the paper (Piumarta 2006) as a pipeline of traditional *batch-mode transformations* such as parsing, analysis, and code generation. Similarly, it presents its bootstrapping process in terms of batch-mode transformations of various source code files. This entrenches it in the Implicit Structure of the Unix Paradigm (Section 3.2.2) which obscures the essential ideas relevant to our purposes (recall Section 3.3.3.3).

We would rather have the ability to gradually *sculpt* a system into a self-sustainable state, interactively, through a combination of manual actions and automatic code. This requires that the system should be conceived primarily through a graphical interface, yet the COLA design does not provide guidance in this respect. It is possible to see how a COLA's various languages and components work together as a sort of self-sustainable command-line REPL, but less easy to see how its text-centric approach may apply to arbitrary graphical interfaces.

COLA sketches a way to escape the intra-process scope of the Unix Paradigm in a vehicle comprehensible to a single individual. Yet it only does so in the framing of batch-mode stream transformations, which limits its potential. We will refer to COLA repeatedly for inspiration and comparison and we will offer some analysis in terms of our own concepts. However, we must adapt its approach to a basis of Explicit Structure while retaining the essential ideas. This is the topic of the next chapter.

BOOTSTRAPLAB: THE THREE PROPERTIES IN THE WEB BROWSER

We now arrive at the main contribution of this work: the construction of a programming system with our Three Properties, which we call *BootstrapLab*.[1] On its own, the existence of such a system acts as a constructive proof of our Thesis Statement (Section 1.6). However, we are less interested in the fact that such a thing *is* possible than we are in *how* it was done and what we can learn from the process. We have discussed our Three Properties in previous chapters to the extent feasible *while remaining in the theoretical realm,* but since they are meant to be properties of programming systems, it is essential that we give practical experience an opportunity to teach us something about them that we could not learn otherwise. We would expect not only to learn how to *achieve* such properties, but also to gain clarity on their nature.

Thus, in this chapter[2] we take the self-sustainable COLA as our inspiration and seek to build up to its Lisp-like "behavioural" half by means of code with Explicit Structure. We critically analyse its development process and identify ideas that may apply more generally. This will lay the foundations for creating new self-sustainable programming systems. We will have to wait until Chapter 5 to evaluate BootstrapLab in terms of our Three Properties, but we will summarise it in terms of Olsen's criteria for user interface systems (Olsen 2007) at the end of this chapter.

What is presented here is not necessarily a chronologically accurate account, but a *rational reconstruction* of the steps involved. For each step, we describe the general task at hand, illustrate this with concrete decisions made in the implementation of BootstrapLab and, where appropriate, sketch possible alternative decisions and their likely consequences. In other words, it is a depth-first exploration of the process with some alternative branches suggested along the way. It can be understood on two levels:

1. It tells the development story of a concrete system. BootstrapLab is a novel programming system, based on explicit structure and built on top of the web platform, which exhibits a minimal demonstration of Self-Sustainability and Notational Freedom.

2. It presents a rational reconstruction of the logical steps needed to bootstrap a *general* self-sustainable programming system (by

---

1 https://github.com/jdjakub/BootstrapLab/tree/phd-thesis/orom/computation
2 This chapter was adapted from our 2022 *Onward!* Essay entitled "Ascending the Ladder to Self-Sustainability" (Jakubovic and Petricek 2022)

taking BootstrapLab to be representative of the important parts of such a task). It highlights design *forces* and *heuristics* for resolving them which can be used by designers of future self-sustainable systems.

4.1 METHODOLOGY

We follow in the spirit of COLA, but we aim to bootstrap a graphical and interactive self-sustainable system instead of a textual one based on batch-mode transformations. The system should not have barriers in the way of using custom notations. We also want to work with an *interactive* system, meaning that the user should be able to modify the state of the running system through manual gestures and not just programmatically.

This approach can better exploit the graphical and interactive capabilities of modern computing, but it also sidesteps the tedious accidental complexities of parsing and serialising text. Similarly, making the system interactive will allow the user to better understand the consequences of individual small changes and will, in turn, support a virtuous cycle of self-improvement.

Our desire is to make an interactive, structured "port" of the COLA approach. This is unexplored territory. It must be emphasised that finding the right "final design" upfront should not be necessary and would defeat the spirit of the enterprise. The point is to build an initial kernel which is then sufficient for evolving and improving itself.

Unlike COLA, we do not write an initial object system in a language like C++. In order to support interactivity, structure, and graphics, we begin with a platform that already conveniently supports those features. This forms a suitable "blank slate" from which to gradually develop the system into a self-sustainable state. At each stage, we take stock of which changes can feasibly be achieved at the user level within the system, versus those that can only be achieved at the implementation level (recall Section 3.3.1.3). We then ask ourselves: how can we imbue the user level with control over some of these aspects?

The following sections propose key steps for evolving self-sustainability in this way, informed by our actual experience applying them in BootstrapLab. We will point out the *forces* that shape the design and the *heuristics* by which we resolve competing forces. As we proceed through the development journey, we will reflect on the heuristics in light of actual practice and compare our choices for BootstrapLab with the corresponding stages of three other systems:

- COLA, since it is our main inspiration whose approach we are adapting;

- The Altair 8800 of the 1970s, to stand in for low-level or "hardware" platforms;

Product: BootstrapLab	In-system
Substrate: JS code	
Platform: Web, JS	Producer
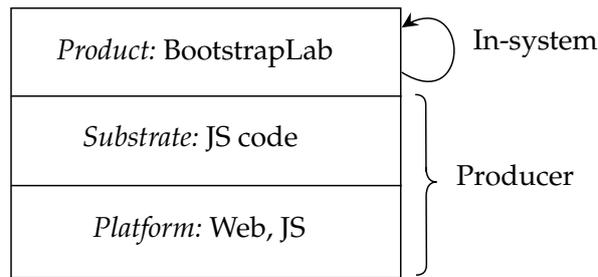
Figure 4.1: The producer comprises the JS platform in the context of the web browser, along with the substrate written in JS. Working in-system means using the product (BootstrapLab) to work on itself.

- Squeak Smalltalk, as a well-established and practical programming system developed in a similar way (Ingalls et al. 1997).

4.2 CONCEPTS AND TERMINOLOGY

Following the terminology in Section 3.3.1.4, we use the term *product system* (or simply "the system") to refer to the programming system that we evolve towards self-sustainability. We use the *producer system* (or simply "producer") to bootstrap the product system. The producer is divided into two layers: the *platform* consists of all the pre-existing capabilities of the producer, while the *substrate* is the basis for the product system that we have to build. We use the term *in-system* to refer to changes made within the product system, by using it as a programming system at its user level. These terms are summarised in Figure 4.1. We also model the product system as having a *state* that *changes over time* as we introduced in Section 3.1.

Recall the definition of *bootstrapping* we gave in Section 2.3.1. Our task is to explore the question: how do we bootstrap *interactive graphical* self-sustainable systems? Note that by definition, what we do with a self-sustainable system is open-ended. This chapter is solely concerned with *getting there* from the ordinary world, which is why we will spend so much discussion on the design of the substrate. This determines how the product system can evolve, how soon can it become self-sustainable and to what extent.

4.3 JOURNEY ITINERARY

The rest of the chapter documents the steps involved in designing a self-sustainable system. Be advised that the sequence is a *rational reconstruction.* The implementation of BootstrapLab followed a more meandering path, but the following steps gesture at the Platonically optimal pathway for bootstrapping a self-sustainable programming system:

1. *Choose a starting platform.* The platform is a *pre-existing* programming system that we use to create and run the product system. The platform cannot be re-programmed, let alone to become self-sustainable, but it allows us to build a self-sustainable product system. To choose a platform, we consider its distance from desired substrate features and personal preference.

2. *Design a substrate.* The substrate defines the basic infrastructure supporting the product system: how the state is *represented* and *changed*. The design of a substrate re-uses parts of the platform where possible and extends it where necessary. We must decide which platform capabilities to use to represent the state and how to expose graphics and interaction. We design a minimal *instruction set,* BL-ASM, which describes changes to the state, and can be represented using the state. We then use the platform to implement an engine that executes these instructions.

3. *Implement temporary infrastructure.* Use the platform to implement tools for working within the substrate, most importantly a *state viewer* or editor. These tools constitute a "ladder" that we will pull up behind us once we have ascended to in-system implementations of these tools.

4. *Implement a high-level language.* The substrate's instruction set (BL-ASM) is cumbersome, so ensure programs can be expressed in-system via high-level constructs. Decide how to represent expressions as structured state and whether to *interpret* or *compile* them into BL-ASM. Ideally, develop such an engine in BL-ASM gradually and interactively. Alternatively, implement it at the platform level and later *port* it to BL-ASM or the high-level language itself.

5. *Pay off outstanding substrate debt.* Port all remaining temporary infrastructure into the system, taking advantage of the infrastructure itself and the high-level language. The result is a *self-sustainable* programming system.

6. *Provide for domain-specific notations.* Use the self-sustaining state editor to construct a more convenient interface for editing high-level expressions. Add *novel notations and interfaces* as needed. Use these not just for programming new end-user applications, but also to improve the product system itself.

What we have here looks like a Waterfall development plan, each step strictly following from the completion of the previous. This presentation is convenient as a summary, but in practice, the sequencing here need not be so rigid. Adjacent steps may overlap, or we may need to prototype and revise a previous step in light of the result.

The general outline also resembles the discredited "recapitulation theory" in biology, where in order for an embryo to develop into a full

organism, it passes through the evolutionary history of its ancestors. In other words, for a *particular* cell to develop into an animal, it needs to fast-track its ancestors' evolution from a cell in the distant past. While this has since been rejected in biology, it is a good summary of what is going on in our project here.

For us, the bootstrapping of a *particular* self-sustainable system fast-tracks the historical development of computing's abstractions. It begins at the low level and ascends through to higher-level languages, each time building the next stage in the current one. This journey could be seen as an attempt to *reconstruct* programming on top of a more structured, graphical substrate than the byte arrays we all had to use the first time around. With that in mind, let us now proceed to the first stage.

4.4 CHOOSE A STARTING PLATFORM

> The platform is a *pre-existing* programming system that we use to create and run the product system. The platform cannot be re-programmed, let alone to become self-sustainable, but it allows us to bootstrap a self-sustainable product system. To choose a platform, we consider its distance from desired substrate features and personal preference.

The first step is to choose the platform that we will use as the basis for the product system. This could be any existing high-level or low-level programming system. An important factor is simply personal familiarity or preference for a particular platform. This plays a role during bootstrapping, but is destined to become irrelevant once self-sustainability is achieved.

The other major consideration is the primitives provided by the platform. They influence how we can design the substrate on top of it in the next step. If we begin with a high-level platform with many convenient features (e. g. graphics and audio capabilities), then we will have to regard them as black boxes. We may expose them as primitives in the product system, but we will not be able to *re-program* them in-system since we cannot re-program the platform.

Alternatively, such imported convenient high-level features could later be *re-implemented* in the product using more basic primitives. However, this would delay the point from which we can work fully in-system. This foreshadows a coming design tension in the substrate (Section 4.5.3).

In the Altair 8800, the *platform* comprised linear memory (state) and native CPU instructions (state change). The platform did not provide other tooling aside from switches to manually set memory values (Figure 4.2).

Figure 4.2: The Altair 8800 microcomputer and its front panel of switches. *Image credit: (Colegrove 2020).*

In COLA, the platform is C (Piumarta and Warth 2008) or C++ (Piumarta 2006) and the Unix command-line environment; in other words, it is the Unix programming system (Section 3.2.2).

According to Ingalls et al. (1997), the platform for Squeak was clearly Smalltalk, even though it was later compiled to C. We could make the distinction that Smalltalk was the *development platform* while C was the *deployment platform.* However, we will continue to use the term *platform* to mean the *development* platform, because our concern is how to *build* such systems in the first place. Also, despite our general characterisation of Smalltalk as self-sustainable, the versions available to the Squeak developers were not so, or not enough (Ingalls et al. 1997):

> While Smalltalk met the technical desiderata, none of the available implementations gave us the kind of control we wanted over graphics, sound, and the Smalltalk engine itself (…) It became clear that the best way to get what we all wanted was to build a new Smalltalk with these goals and to share it with this wider community.

Even had they been equipped with a sufficiently self-sustainable platform, other issues such as licensing and portability would have still prevented them from being content with it as a vehicle for their goals. This reminds us that one may have all sorts of reasons to create a *new* self-sustainable programming system, even if one is already available.

However, our work here proceeds from what we consider to be a more common premise: that the platforms that are available for us to use (incorporating factors like skills, familiarity, and preference) are *not* self-sustainable, and need to be made so by our own augmentation. Accordingly, for BootstrapLab, we chose JS and the Web platform. This programming system is the one we are most familiar and comfortable with, and it also provides us with built-in Web technologies, libraries (including graphics), and the browser developer tools. In other words,

such a platform provides a range of convenient tools to assist bootstrapping. However, because of their large scope, we will only be able to expose a carefully selected subset of such features as primitives to the product system.

What can be changed at the user level? At this point, there is no product system to speak of yet. This means that nothing can be changed in-system. The platform can, in principle, be modified, but by assumption this is so unfamiliar, uneconomical, or inappropriate that the developer has opted to make a (new) self-sustainable system instead.

4.5 DESIGN A SUBSTRATE

> The substrate defines the basic infrastructure supporting the product system: how the state is *represented* and *changed*. The design of a substrate re-uses parts of the platform where possible and extends it where necessary. We must decide which platform capabilities to use to represent the state and how to expose graphics and interaction. We design a minimal *instruction set,* BL-ASM, which describes changes to the state, and can be represented using the state. We then use the platform to implement an engine that executes these instructions.

With the *platform* defined as the already-existing programming system that we start from, we define the *substrate* as the basic infrastructure, implemented via the platform, necessary for the product system. This substrate is the part of the system which we have control over (being programmed *by us*, unlike the platform itself) yet which we do not expect to expose from within the system. In other words, the substrate is the small non-self-sustainable core that supports the self-sustainable product on top of it.

In short, our division is as follows:

	Created by us?	Self-sustainable?
Platform	No	No
Substrate	Yes, atop Platform	No
Product	Yes, atop Substrate	Yes

The design of the substrate can be considered along two axes (Table 4.1). The first dimension follows the distinction between data and code, or *state* and state *change* (Section 3.1). We must first decide how the *state* of the system will be represented. Often, this is a matter of choosing an appropriate subset of what the platform already provides. Then, we decide how primitive *changes* to that state can be described and define the instruction set.

Table 4.1: The conceptual divisions of the substrate.

Domain \ Agent	Human (Manual)	Computer (Automatic)
State	User Interface	Data structures
Change	UI Controls	Instructions

The second dimension follows the division between the *computer* and *human* actors. The full state of the system will be an internal data structure, but a *part* of the state—comprising the state of the user interface—can be directly seen by the user. Similarly, *change* can be performed automatically or manually. There must be a way to run instructions automatically at a high speed, but the user interface must also provide controls for a human to make changes at their own pace.

While the foregoing model applies to programming systems generally, a special condition is required for those that are self-sustainable. We must represent instructions as pieces of state, as opposed to having "two types of things"—ordinary data, and code—which must be viewed and edited using completely different tools. This property of *Code As Data*[3] means instructions can be generated and manipulated just like ordinary state, whether programmatically or manually. Only if this is possible can higher-level abstractions be built up, in-system, from the low level.

Requirement 1 (Code As Data). Instructions must be readable and writeable as ordinary state.

The fact that we call this a *requirement* is more out of a desire to save trouble than as something that is *strictly* necessary from the beginning. If code is not ordinary data, Code As Data can be trivially, if inefficiently, bolted on after the fact. We would simply write an interpreter (for the same language, or a new one) in the existing code, which reads instructions from somewhere in the "data" memory and executes them. We would then forget all about the original interpreter and work entirely within the new language. In fact, as we will detail in Section 4.5.5 shortly, this is exactly what we should expect to have to do as a key part of designing a substrate. The point is that we should not *repeat* any such "mistake" of the platform and have to fix it a second time due to our own mis-design; we might as well avoid the overhead of an inner interpreter by simply having Code As Data from the beginning in our substrate.

---

3 In Lisp, this idea is called *homoiconicity*; elsewhere it is known as the *von Neumann* architecture. We are unsure of any essential differences between these three terms but use the more direct "code as data" to avoid quibbles.

4.5.1   *Substrates in Squeak and Altair*

The path laid out for Squeak in Ingalls et al. (1997) is very clear: the Virtual Machine (VM) is developed in Smalltalk (the development platform) and compiled to C (the deployment platform). In C, the VM conforms to our conception of a substrate: it is not modifiable from within Squeak. On the other hand, during development, it is unclear whether this is still the case, owing to the high self-sustainability of Smalltalk. However, we presume that it remains off-limits from within the system, on the basis that any VM-altering capabilities present in the Smalltalk implementation would break if straightforwardly compiled to C. Since Squeak is itself a version of Smalltalk, the substrate's conception of *state* is a graph of *objects* and its *change* takes the form of VM bytecode instructions; extensive details can be found in Ingalls et al. (1997).

As for the Altair 8800, we are using it as an example of a hardware platform; unlike our other example systems, the "programming system" it would support has been left open-ended. However, for most things one would want to achieve with such a machine, one would need basic software facilities like memory and time slice management, device interfacing, and so on. These are precisely the facilities of an Operating System (OS) and we would regard such a basic software layer as a substrate for such a system. Similarly, we would regard the Unix OS (or at least the kernel) as the substrate in Unix systems—even the Altair, supposing it had the requisite hardware resources and capabilities. As we pointed out in Section 3.2.2.1, the Unix substrate is cleaved into two layers: *inter*-process, in which state comprises the filesystem which is changed by system calls, and *intra*-process, in which state comprises raw memory changed by machine instructions (Section 3.1.1). In the latter case, state and change are more or less inherited from the hardware platform, while the former is built as infrastructure on top of this.

4.5.2   *COLA's Low-Level Byte Arrays*

As an example of the intra-process notions of state and change, we can look at COLA, a programming system conceived as a process within Unix. As such, its substrate is quite minimal and the majority is inherited "for free" from the low-level runtime environment that makes up a process.

At the lowest level, state in COLA consists of an array of bytes, addressed numerically. Some structure is imposed on this via C's standard memory allocation routines, refining the model of state to a graph of fixed-size memory blocks and the stack. Changes to this state are represented as machine instructions encoded as bytes. This is the basic state model of a C program; the sample code for COLA's object model (Piumarta and Warth 2008, Appendix B) embellishes this with little more

than a way to associate objects to their vtables[4] and a cache for method lookups.

This bare-bones, low-level substrate does not require much development on top of the platform and so it is quicker to complete. The ontology of state is copied from the platform, and in this case the machine instructions can be inherited too.[5] Completing the substrate more quickly means we can start working in-system sooner, but there is a downside: it may be cumbersome to work with such minimal functionality. The unfortunate effect would be that we speed through a primitive substrate, only to suffer slow progress at the beginning of in-system development.

Building back up from machine-code level may be an impressive hacker achievement or useful for pedagogy (Agaram 2020). But it is clearly not optimal, speed-wise, when we already have a higher-level platform to program with.

In the other direction, there is no limit to how fancy we could make the substrate in terms of high-level abstractions and convenient features. However, these would take much longer to implement and delay in-system development. Moreover, this risks doing a lot of work that can never benefit from in-system innovation feedback (recall Section 3.3.1.5); the substrate's implementation will not be modifiable from within the system it supports.

4.5.3   *The Major Design Conflict*

We clearly have two opposing tendencies here, which we will formalise as follows:

Force 1 (Avoid Boilerplate)**.** Push complex features into the substrate to avoid wasting in-system development time on them.

Force 2 (Escape The Platform)**.** Push complex features in-system to avoid delaying in-system development and to have them benefit from innovation feedback.

We will refer to these throughout the journey. They conflict over where the implementation of convenient functionality should reside. In any specific design, they will resolve in some compromise. It is helpful to consider the extreme points of this.

Force 2 wants to get the substrate over with as quickly as possible, eager to escape the (real) limitations of the platform and get working

---

4 A *vtable* specifies object behaviour by supplying runnable code for a requested method name. It is separate from the object "instance" so that multiple objects can share the same behaviour. In the sample code, an object is an arbitrary memory block preceded by a vtable pointer.

5 In general, the internal representation of code in the platform will be unavailable to us when programming with it, so we expect not to be able to inherit it. This low-level platform is a special case, where we do have access to code if we are willing to write instructions using their numerical codes.

in a system that *can* be arbitrarily changed. Force 2, left unchecked, will guide us to adopt a substrate resembling a Turing machine:[6] have a tape for the state; instructions for manually shifting left and right, reacting to the current symbol, and writing a new one; have a user interface in which to do these things manually. Such a substrate is so simple it could be coded in an hour or two. Yet our first duties in-system will be to implement extremely basic features, like data addressing and arithmetic, in an extremely tedious way. The endpoint of Force 2 is the Turing Tarpit (Perlis 1982).

On the other hand, if we follow Force 1 unchecked, we spend much time and effort working with the platform to produce, in effect, a *complete* novel programming system. Any feature we would find useful in-system, we would have already implemented outside it. Yet this means that all the important functionality could not be changed except by going back to the source code in the platform; we'd have created a boring old *non*-self-sustainable programming system. The endpoint of Force 1 is programming-as-usual.

A symptom of the latter failure mode would be that we never felt comfortable leaving the platform behind and continuing development from within the system. Self-sustaining systems are meant to be *grown* from a small enough starting point; we shouldn't need to come up with a flawless design ahead of time. This will only be possible if we artfully balance Forces 1 and 2 so that the in-system programming experience becomes tolerable in a reasonable timeframe.

REFLECTIONS ON THE MACHINE-LEVEL APPROACH.   We experienced something like the "Escape The Platform" (Force 2) absurdity for COLA when following the sample C implementation of its object system (Jakubovic 2020b). The code was easy enough to comprehend and compile, but what we were left with was a system living entirely in memory lacking even a command-line interface. In order to develop the system, it seemed necessary to run it in a machine-level debugger.

Even if we had stayed with it, we would still be stuck in the low-level binary world which is unfriendly for humans to work with, as we explained in Section 3.1.1. Instead of names, we only have numbers for addressing things. The state is flat and we cannot insert or grow something without physically moving other content to make space. Any structure, such as trees or graphs, has to be *faked* as memory blocks pointing to each other.

Slightly better is the model found in VM designs. For example, in the Java VM (Lindholm et al. 2023), bytecodes address "local no. #3", and "constant no. #4" instead of "memory location 952A" and "memory location 3B84". However, this is still far from human-friendly. Low-level substrates are better than the Turing machine, and were a historical

---

6 Or, equivalently, a lambda calculus evaluator, or Gödel's recursive functions, or Conway's Game of Life, etc.

necessity in the early days of computing. Nowadays, however, we have an opportunity to leave them behind, and instead build new systems on top of a "low level" that is nevertheless *minimally*[7] human-friendly (Section 3.1.2).

Heuristic 1 (Minimally human-friendly low-level)**.** Ensure the substrate natively supports *string names* and *substructures.* This is a minimal concession to "Avoid Boilerplate" (Force 1) that still keeps the substrate simple enough and thus does not strongly conflict with "Escape The Platform" (Force 2).

We can unpack the rationale behind this heuristic as a consequence of an assumption, or even a *bet:* because string names and substructures are so primitive, simple, or "unremarkable", we are *betting* that we will be content with the way they work long into the system's lifetime, and not feel the need to change them. This would be an unwise assumption for complex functionality, such as a high-level language; it is highly likely that, at some point, we will want to experiment with it or change how it works for some reason; therefore, the implementation of a high-level language must reside in-system (we will cover this in Section 4.7). This seems far less inevitable in the case of extremely basic infrastructure, so implementing it in the substrate is at least a reasonably safe decision.

Clearly, our conception of a *substrate* has much in common with that of a VM. However, we wish to avoid the established connotations of such a term and allow for degrees of freedom that might not be associated with a "virtual machine". For example, a substrate might base the design on names and dictionary structures (an approach that we adopt and will discuss presently); conversely, a VM design would probably pre-optimise these to more resemble the Low-Level Binary model (Section 3.1.1). Another example is that one could choose to embed a high-level language in a substrate, whereas a virtual "machine" would specify low-level bytecodes in which a high-level language could be implemented. We will actually take the VM approach in this respect, but the point is to show how a "substrate" need not necessarily resemble a "machine" and thus why we use the former term to generalise the latter.

4.5.4 *BootstrapLab's Simple, Structured State Model*

For the design of BootstrapLab, we chose the Web platform and JS, out of a combination of personal preference and conveniences like graphics. This imposed a number of design decisions on the substrate, due to

---

7 We recognise that there is something *inherently* human-unfriendly about programming in "low-level" code; we are simply observing that there is accidental complexity above and beyond what seems to be necessary, which could be fixed. In other words, we have our own reasons to program in terms of small instructions, but we see no redeeming value in working with flat, numerical state.

a tendency for earlier choices to determine which later ones will feel "natural" or "fitting".

In our high-level platform language JS, state is a graph of plain JS objects acting as property dictionaries. Suppose we *still* chose a low-level binary substrate like that of COLA. This would no doubt be possible: declare one giant JS array called `state`, design numerical instruction encodings which overwrite numbers at certain indexes, etc. Yet this would feel like a perverse *waste* of something the platform was giving us for free.

JS already provides the basic human affordances of naming and sub-structure, so why would we throw them away and force ourselves to implement them in-system? The low-level COLA substrate does plausibly follow from its base C platform; our choice of JS as the platform encourages us to preserve *its* own state model in the substrate we design.

Similarly, it would make no sense to represent instructions as numbers or strings. While in the binary world, machine instructions are byte sequences with bitfields for opcodes and operands, in a dictionary substrate inherited from JS, it makes sense to have explicit fields for this data:

```
{
  operation: 'copy',
      from: [ alice, 'age' ],
        to: [   bob, 'age' ]
}
```

As this example shows, addresses in a dictionary-based state model consist of an object reference and a key name.

This "preservation" incentive pervades the journey from platform to product system. The substrate should leverage representations made possible by the platform, while the instruction representation should leverage the structuring of state provided by the substrate. This will also apply to further subdomains expressed in the state model, such as graphics and high-level programming expressions. We formalise this as the following:

Force 3 (Alignment)**.** Everything should *fit:* instructions, high-level expressions, and graphics expressions should all fit the substrate, and the substrate should fit the platform.

It should be noted that while Alignment *constrains* the set of acceptable designs, it is not so powerful as to *determine* a single one. All it does is set a minimum "floor" of important functionality in the substrate, excluding designs that fall below it. We continue to face further choices in terms of how far to *augment* this bare minimum.

In the end, our substrate for BootstrapLab largely *inherits* its state model, only making simplifications. For example, JS objects have proto-typal inheritance, meaning that a simple "read" operation of a property

requires potentially traversing a chain of objects. Our substrate here omits this, so reads are quite simple. Additionally, JS includes a special `Array` object type. We omitted this, opting to represent lists as maps[8] with numerical keys. This unification means that the state model only has one type of composite entity, the *map;* a fact we will exploit later for the high-level language.

We also noticed that we would not get very far if all our progress in-system could be wiped clean by losing our browser tab. Our platform, sitting within the Unix paradigm (Section 3.2.2), does not provide *persistence* out of the box, so we had to implement a mechanism in the substrate. We walk the state graph from the root node and discover a spanning tree, specially marking cyclic or double-parent references. We then serialise this into a JSON file which we can load by undoing the process. This is reminiscent of the image-based persistence in Smalltalk, though it is frustratingly manual. Nevertheless, it was critical to patch this unfortunate aspect of the platform and this was enough to do so.

JS is a "high-level" language, but this is *relative* to the conventional "ground" reference point of the *hardware* as the lowest level. However, the "lowest level" of interest to us is *our* platform, which in this case is not hardware but JS. Therefore, measuring from JS as the reference point, we consider this substrate low-level *relative* to it. In other words, the gap between our substrate and JS is very small. It could have been otherwise; "Avoid Boilerplate" (Force 1) gave us several ideas for convenient features of a smarter state model, but "Escape The Platform" (Force 2) urged us to press ahead without them and see if we needed them later. Appendix C contains these details.

4.5.5 *Designing the Instruction Set*

While the "data" half of the substrate may be easy to inherit from the platform, the "code" half is typically not. Simply including an interpreter for source code in JS is not an option; this would embed a reliance on a strings and parsing in the core of the system, against our desire for Explicit Structure.

Slightly better would be an interpreter for the JS abstract syntax tree. However, "Escape The Platform" (Force 2) still pushes against this. A high-level language interpreter is nontrivial even without parsing and would delay our ability to work in-system. Also, an interpreter is a computer program; this program, or parts of it, might be best expressed or debugged via particular notations; by having it in the substrate, we'd restrict ourselves to the interface of JS in our text editor.

Instead, consider what it takes to implement the interpreter for Assembler, a.k.a. the Fetch-Decode-Execute cycle. We fetch the next small change to make to the state (an instruction). We do a simple case-split on the opcode field; we carry out some small change to the state; rinse

---

8 We refer to our substrate's basic dictionary structure as the *map* for brevity.

and repeat. With this, we can surely mirror the real-world development of higher-level languages from lower ones.

Heuristic 2 (Use Imperative Assembler)**.** Begin from imperative assembler, as this allows us to make arbitrary changes to the state using a minimal interpreter that is quick to implement. "Escape The Platform" (Force 2) outweighs "Avoid Boilerplate" (Force 1) here.

It is important to stress that this "assembler" is relative to the form of the substrate. If the substrate is binary memory, "assembler" will refer to machine instructions. But in our case of a minimally human-friendly low level (Heuristic 1), there is nothing binary about them. The instructions express operations on structured objects with names and are, themselves, represented as structured objects with names. Similarly, "imperative" just refers to the fact that the instructions are arranged in a sequence from the point of view of the interpreter, because it is easier to implement a fetch-execute cycle than, say, a resolver for a dependency graph. The above considerations lead us to Heuristic 3.

Heuristic 3 (Simple Assembler)**.** Prefer fewer instruction types (RISC) over more (CISC). This reduces the size of the interpreter and will be quickest to implement. It will make programs longer, but this can be mitigated by a high-level programming language. "Escape The Platform" (Force 2) outweighs "Avoid Boilerplate" (Force 1) here too.

Right away, we know there will have to be a special piece of state for the *instruction pointer*. This could indicate the *current* instruction or the *next*; we chose the latter for BootstrapLab and called it `next_instruction`.

The value of this "register" is determined by how exactly we fetch the next instruction. Perhaps each one has a `next` field which we can simply follow. In this case, the `next_instruction` will simply be the instruction itself. This also gives us convenient conditionals (e. g. fields called `if_true` and `if_false`) but means that instruction *sequences* will have a nesting structure. This latter consequence may be inconvenient for presentation in a tree view. For BootstrapLab, we chose the alternative of numerically indexed lists of instructions which easily display in a column. This choice determined `next_instruction` to instead hold an *address*, called `ref`, made of container map and key name:

```
next_instruction: {
  ref: {
    map: // the instruction list
    key: 1 // i.e. first instruction in the list
  },
  value: // the fetched instruction itself
}
```

Here, the "fetch" step involves dereferencing the address, incrementing the key name, and updating the `value` key to the instruction itself.

Next, we turn to what types of instructions we need. Alignment (Force 3) means that, given a state model, obtaining an instruction set should be more of a "derivation" than a hard design problem. This is because some choices are obviously inappropriate and others clearly fitting to the state model. For example, in a tree-structured state model, it would be foolish to have instructions that can only see the root level:

```
{ op: 'copy', from: 'source_key', to: 'dest_key' }
```

Without the ability to read or write keys *within* an arbitrary tree node, as far as programmatic change is concerned, the state becomes a *de facto* flat list instead of a tree. Therefore, it is critical that anywhere in the state can be accessed or modified by an appropriate instruction sequence.

We suspect that the following basic functions suffice to make an instruction set Turing-complete:

1. Copy from one location to another (an "immediate" or "literal" is just copied from the instruction itself)

2. Treat a value as an address and follow the reference

3. Unconditional jump (copy a value to the instruction pointer)

4. Conditional jump (take a path based on a run-time condition)

"Avoid Boilerplate [in-system]" (Force 1) may push us to include basic boilerplate in the substrate, like arithmetic or an operand stack. Furthermore, it is advisable to have an "escape hatch" into the platform if possible. In BootstrapLab, our platform language JS provides the `eval()` function to execute a string of JS code. We exposed this as a `js` instruction. This allows us to use and store JS code in the *running* system instead of having to edit the source file, contributing a little to self-sustainability.

The resulting instruction set for BootstrapLab was derived from these considerations, as well as extreme application of "Simple Assembler" (Heuristic 3). It uses the top-level map as a set of "registers" whose contents are *immediately* accessible. The main registers are:

- `focus`: an "accumulator" register used by most instructions. Holds the key for map operations.

- `map`: holds the map to be updated or navigated by map instructions.

- `source`: holds the new value when updating a map entry.

- `next_instruction`: program counter; holds the address of the next instruction and the instruction itself.

State that is "further away" than the top level is accessed by following key paths from there, or from existing map references. Beside the few special registers used by instructions, other names in the top level are available as local variables in user code. The instructions are as follows (see Section B.2 for full details):

- `load` fills the `focus` register with a literal value.

- `deref` treats the value in `focus` as naming a register and copies the register's value into `focus`.

- `index` expects a map in the `map` register and a key name in `focus`. It looks up this key in `map` and replaces `map` with the value.

- `store` copies the value in `focus` to a named register. Alternatively, if no register is present, it copies the value in the `source` register to the destination identified by `map` and `focus`, as with the `index` instruction.

An instruction is represented as a map with an `op` field for its name and other fields for parameters. For example:

```
{ op: 'store', register: 'source' }
```

It is remarkable that these few operations really are sufficient even for conditional and unconditional jumps. A jump is achieved by overwriting `next_instruction`, and this can be conditionalised by `index`ing a map of code paths based on a selector. We made the decision that `index`, if accessing a key not present in the map, will try and retrieve the special key `_` instead. This supports a generic "else" or "otherwise" clause for conditionals. We provide a worked example in Section C.1.2 and also show how an arbitrary path-to-path copy reduces to these primitive operations. This lends support to the Turing-completeness suspicion: since we can effect the basic operations[9] found in real-world, practical, non-minimal instruction sets, our set is at least as Turing-complete as they are.

The minimal, microcode-like instruction set here was an experiment in extreme parsimony; see Section C.1 for the gory details. Although it was interesting, certain basic operations (such as jumps) are extremely verbose, taking many instructions. Although it was quick to implement these instructions in JS, it was too tedious to work with them in-system. In retrospect, it looks like we went too far with "Escape The Platform" (Force 2) here and fell into its associated Turing Tarpit trap. We thus consider an *extreme* interpretation of "Simple Assembler" (Heuristic 3) refuted for the purposes of working in-system sooner. We recommend achieving a better balance by including direct path arguments in instructions (e.g. "copy `a.b.c` to `x.y.z`" as a single instruction), as well as separate (un)conditional jump instructions.

---

[9] See the discussion in Section B.2.4 for why we do not consider arithmetic as among these basic operations.

Nevertheless, at this point we have a workable instruction set for BootstrapLab; we will subsequently refer to it as BL-ASM.

4.5.6   *Graphics and Interaction*

Now that we have covered the computer-oriented part of the substrate, we turn to the human-oriented user interface state and change aspects. One way we wish to distinguish BootstrapLab from COLA is that graphical interfaces are present from the beginning and not just an afterthought. There are two factors here: how graphics are represented in the substrate, and how they are actually displayed.

It may be useful to see this as a microcosm of the entire journey. First we must select a graphics library available for our platform (i.e. the *graphics platform*). Then we must decide how graphics are represented in our substrate (a graphics *sub*-substrate) and how these graphics actually end up on our screen.

In BootstrapLab, we chose to build off the THREE.js 3D graphics library as our platform. As for the substrate, we face an immediate choice between so-called "immediate mode" and "retained mode" conventions. In immediate mode, we draw and change graphics by issuing commands; a "code-like" approach. In retained mode, the state of the scene is represented as some structured arrangement of state. When we want to change something, we simply change the relevant part of the state and the display should automatically update.

Immediate-mode in this case could be realised by, say, exposing all the relevant THREE.js functions as special instruction types. In actuality, however, this sounded far too tedious to work with; "Avoid Boilerplate" (Force 1) won out and we opted for retained mode instead. The rest of the design then fell out via Alignment (Force 3).

Consider the low-level binary substrate in which microcomputer games were programmed. In this world there is a special region of memory, the *framebuffer*, which is treated as the ground truth of pixels displaying on the screen. To draw, programs rasterise shapes into pixels and write to the framebuffer.[10] The framebuffer has a flat structure—two-dimensional, yet not by any means nested—aligning with the substrate it sits within. This suggests that a natural choice for retained-mode graphics representation can be found by inspecting the substrate. In BootstrapLab's case, the natural choice is not a flat "framebuffer" but a tree structure of data describing shapes and text—vector graphics.[11]

Heuristic 4 (In-state graphics). Make graphical interfaces expressible as ordinary state in a special location. Having graphics built into the substrate responds to "Avoid Boilerplate" (Force 1) while Force 3 directs us to use a representation that fits the state model.

---

10 In Commodore 64 BASIC, this would be accomplished with commands like POKE 1024,1.

11 Further rationale for this approach can be seen in (Hague 2010).

```
-scene:
 +camera:
 -shapes:
  text: Hello World
  -position:
   basis: world
   right: 0
   up: 0
   forward: -100
  -children:
   -yellow_shape:
    color: 0x999900
    width: 0.5000
    height: 0.5000
    -center:
     basis: shapes
     right: 0
     up: 0.5000
     forward: -1
   -blue_shape:
    color: 0x009999
```
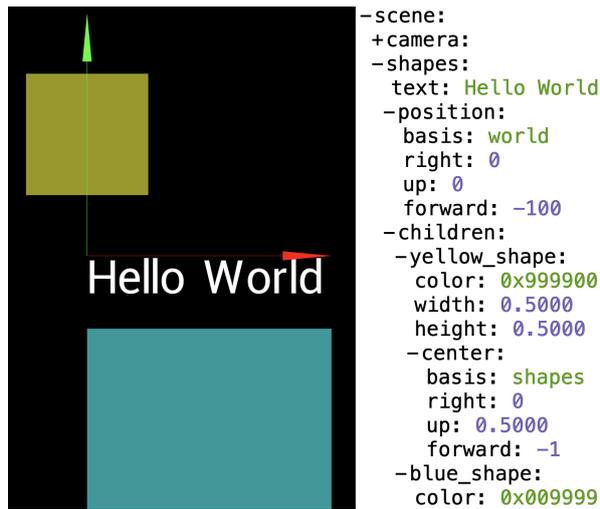
Figure 4.3: Example of how nested tree fields are represented (right) vs. the rendered output (left). The right-hand half is the temporary state view discussed in Section 4.6, where curly braces and commas have been dropped from the notation. The +/- signs are UI controls for expanding and collapsing tree nodes, respectively.

In BootstrapLab, this is a sub*tree* of the state under the top-level name `scene` (Figure 4.3). There are several special keys (e. g. `text`, `position`, `color`, `children`) which have graphical consequences. Other keys may be used as ordinary state.

For interaction, we need to expose the platform's ability to listen for user input. In BootstrapLab, we would execute a named code sequence in the substrate from JS event handlers, which now function as "device drivers" (Figure 4.4).

This is a basic sketch with some issues elided that a complete account would cover. For example, what happens when an input event occurs during the handling of a previous event? Possibilities include ignoring the extra event or providing some sort of stack analogue[12] for nested handlers. Such a data structure may also be necessary for saving and restoring the instruction pointer along with other context. These concerns have analogues in interrupt handling for operating system design, which we would consult for future work. However, at this early stage, we found our existing approach to be adequate and progressed to the next step.

4.5.7   *BootstrapLab Substrate Summary*

Computer state is a graph of maps; lists are just maps with numerical keys. Instructions are `load`, `deref`, `store`, `index`, `js`. Special top-level

---

12 Of course, in a structured substrate, there is room for improvement on the linear form of the low-level machine stack; see Section 4.7.4 for how we did this for the high-level language.

```
window.onkeydown = e => {
  state.set('input', 'type', 'keydown');
  state.set('input', 'key', e.key);
  let input_handler_code = state.get('input', 'handler');
  save_context();
  state.set('next_instruction', new state.Map({
    map: input_handler_code, key: 1
  }));
  asm.execute_till_completion();
  restore_context();
};
```

Figure 4.4: Sketch of a "Device driver" triggering a generic event handler sequence in-system. We have experimented with running in-system code in response to input, but as of the time of writing, BootstrapLab's event handlers are less sophisticated than this sketch (see Section B.2.6).

keys are `focus`, `map`, `source` and `next_instruction`. User Interface state is controlled via the special `scene` subtree of state. Each node may use special keys like `text`, `width`, `height`, `color`, `position`, and `children`, as well as arbitrary other keys for user data. For a full reference, see Appendix B.

What can be changed at the user level? System state can be modified and instructions can be executed, but only using the cumbersome capabilities of the platform. In BootstrapLab, this means using the JS debugging console to edit state (Section B.1.3) and to call a function to execute a certain number of instructions (Section B.2.5).

4.6   IMPLEMENT TEMPORARY INFRASTRUCTURE

> Use the platform to implement tools for working within the substrate, most importantly a *state viewer* or editor. These tools constitute a "ladder" that we will pull up behind us once we have ascended to in-system implementations of these tools.

In most cases, the base platform will provide some way of viewing and modifying state, but this is typically inconvenient to use. The next step in bootstrapping a self-sustainable system involves implementing temporary infrastructure that lets us work with state more conveniently.

4.6.1   *Early Computing, Squeak, and COLA*

Temporary infrastructure to support in-system development can be found in the evolution of many self-sustainable systems. A historical ex-

ample is the Teletype loader for the Altair 8800. Here, the base platform was the Altair hardware with its memory and native CPU instructions. The only way to modify state in the platform was through the use of hardware switches at the front of the computer (Figure 4.2), which could be used to read and set values in a given range of memory.

Here, programming *in-system* looked like the tedious setting of switches to poke numerical instructions to memory. To make entering programs easier, the recommended first step when using the Altair 8800 was to manually input instructions for a *boot loader* that communicated over the serial port. When finished, this could be run to load instructions from a paper tape. From here, programmers could write instructions more conveniently using a Teletype terminal and have them loaded into the Altair memory.

The history of Squeak (Ingalls et al. 1997) mentions temporary infrastructure via Smalltalk, such as a throwaway memory allocator:

> By following this plan, facilities became available just as they were needed. For example, the interpreter and object memory were debugged using a temporary memory allocator that had no way to reclaim garbage. However, since the system only executed a few byte codes, it never got far enough to run out of memory, Likewise, while the translator was being prepared, most of the bugs in the interpreter and object memory were found and fixed by running them in Smalltalk.

Furthermore, Smalltalk is a programming *system,* famous for its interface including the class/method browser. Thus it is likely that the Squeak developers did not need to invest as much work in viewing/debugging infrastructure as we would expect for most alternative platforms.

In the COLA architecture, there is a four-step process, the first three of which appear to be throwaway (Piumarta 2006, Section 6.1 "Bootstrapping"). This includes a compiler for their state model in C++. This is aptly "jettisoned without remorse" once it has been re-implemented in itself, though it is unclear how a state model can perform computation (only after this do they implement the "behavioural layer"). Regardless, this clearly echoes the bootstrapping process for programming languages (Section 4.2).

The problem with these steps is that they are hard to port to a context involving structured, graphical notation and interactive system evolution. Our task is to get the system into a state where the platform, in a sense, can be "jettisoned" in terms of our attention, even though the platform-implemented substrate will be running in the background.
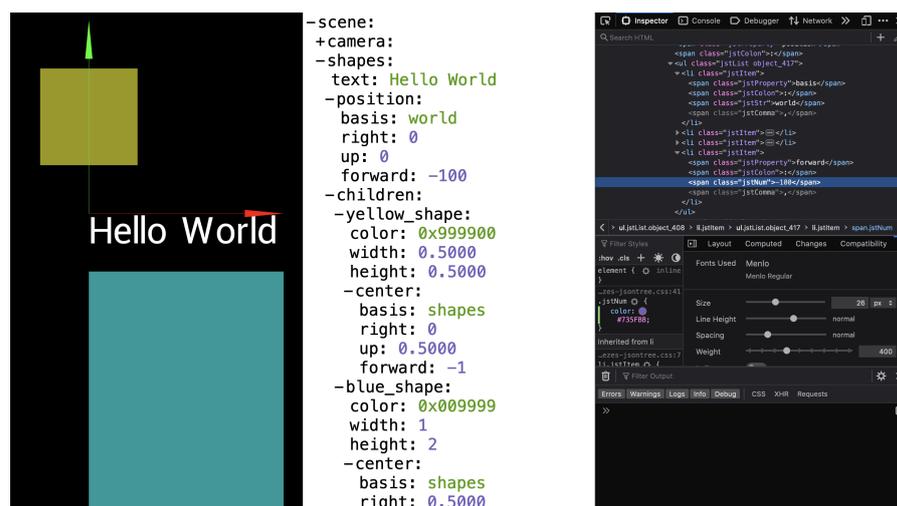
Figure 4.5: The full BootstrapLab interface. From the left: graphics window, temporary HTML state viewer, and browser developer tools. The +/- signs in the state viewer are UI controls for expanding and collapsing tree nodes, respectively.

4.6.2   *Temporary Infrastructure in BootstrapLab*

On its own, our chosen platform for BootstrapLab only has one way to view parts of the state: issuing JS commands via the developer console to poll a current value. This is almost as tedious as toggling switches on the Altair. Being able to see a live view of all of the state would be a highly useful facility early on; recall Section 1.3.1 in which we praised the browser's Element Inspector. In this case, "Avoid Boilerplate" (Force 1) won relative to "Escape The Platform" (Force 2); we capture this as Heuristic 5. We implemented a tree view in the substrate based on an existing JS library. State editing can continue to be done via the console (see Figure 4.5).

Heuristic 5 (Platform editor). As soon as possible, use the platform to implement a temporary state viewer and/or editor. This temporary infrastructure will later be discarded, but given a capable enough platform, it is very easy to implement. For this reason, it is valuable for simplifying further in-system development. Again, "Avoid Boilerplate" (Force 1) outweighs "Escape The Platform" (Force 2).

The JS tree view is a complex set of functionality set to work and display in one specific way, and all control over this resides at the substrate level. The infrastructure cannot be modified from within the system. Therefore, we regard this situation as *temporary*. It is a ladder that we climb to end up in a state where we can implement a (better) state editor in-system. Once a suitable in-system editor exists, we can pull up the ladder (or if you like, "jettison it without remorse").

At this point of the bootstrapping process, BootstrapLab's interface consists of three sections (Figure 4.5). On the right, we have the browser

console, inherited through from the platform's interface. In the middle, we have the output of the platform's main graphics technology, the DOM, displaying the temporary state viewer.[13] Because we have not chosen to expose DOM control from within the system, the system only affects this area indirectly through ordinary state changes. Finally, on the left, we have the THREE.js-backed graphics window[14], where we will later build a state editor whose behaviour (including appearance) *will* be controlled from within the system.

Ideally, we would have actually supported interactive state *editing* in the temporary infrastructure, not just viewing. In our case, however, we tolerated state editing through console commands until implementing a state editor in terms of the left-hand graphics window (see Section 4.8).

Another example of temporary infrastructure is zoom-and-pan in the graphics window. Working within a small box is very restrictive if we want to use it for viewing and editing the entirety of the system state. The finite region can be opened up into an infinite workspace by adding the ability to pan and zoom the camera with the mouse. This was important to have early on for BootstrapLab, so once again "Avoid Boilerplate" (Force 1) overrode "Escape The Platform" (Force 2) and we implemented this in JS.

What can be changed at the user level? The basic activities of viewing or editing state should be made easier by the temporary infrastructure. For the Altair 8800, instruction entry was improved. In Squeak, debugging was made possible by a throwaway allocator. For COLA, the basic state model was made available in the first place. For BootstrapLab, we targeted state visibility.

4.7 IMPLEMENT A HIGH-LEVEL LANGUAGE

The substrate's instruction set (BL-ASM) is cumbersome, so ensure programs can be expressed in-system via high-level constructs. Decide how to represent expressions as structured state and whether to *interpret* or *compile* them into BL-ASM. Ideally, develop such an engine in BL-ASM gradually and interactively. Alternatively, implement it at the platform level and later *port* it to BL-ASM or the high-level language itself.

The temporary infrastructure created in the preceding step may be enough to allow limited development in-system. However, it does not yet provide the barely tolerable programming experience we would need in order to feel comfortable ditching the platform. For this, an additional step is needed.

---

13 Adapted from JSON Tree (https://github.com/lmenezes/json-tree).

14 Utilising THREE Mesh UI (https://github.com/felixmariotto/three-mesh-ui) for text, on top of THREE.js itself (https://threejs.org/).

To make programming in-system pleasant enough, we need a high-level programming language that executes on top of the system substrate. This means that programs and all their necessary run-time state will be stored in the system state, while the execution will be done either by a compiler to the substrate's instruction set or an interpreter.

4.7.1   *Shortcuts for Low-Level Substrates*

For a programming system built atop a primitive platform, such as hardware, the temporary infrastructure may be the best tool that is available for programming. In that case, we would write the compiler or interpreter directly using the instruction set. However, as long as the platform has higher capabilities or one has access to alternative platforms, this may not be optimal. When Paul Allen and Bill Gates wrote the famous BASIC programming language for the Altair 8800, they did not do this *on* the Altair, but instead used an Intel 8080 CPU emulator written and running on Harvard's PDP-10. The high-level language was thus developed *outside the system.*

For Squeak, there was no need to write the Smalltalk interpreter in low-level VM bytecode when the language was already available (Ingalls et al. 1997):

> The interpreter is structured as a single class that gets translated to C along with the Object Memory and BitBlt classes. In addition, a subclass (Interpreter Simulator) runs all the same code from within a Smalltalk environment by supporting basic mouse, file, and display operations. This subclass was the basis for debugging the Squeak system into existence. (…) Having an interpreter that runs within Smalltalk is invaluable for studying the virtual machine. Any operation can be stopped and inspected, or it can be instrumented to gather timing profiles, exact method counts, and other statistics.

Having an existing platform that already does everything one wants is certainly the optimal situation to be in:

> We attribute the speed with which this initial work was accomplished to the Squeak philosophy: do everything in Smalltalk so that each improvement makes everything smaller, faster, and better. It has been a pleasant revelation to work on such low-level system facilities as real-time garbage collection and FM music synthesis from within the comfort and convenience of the Smalltalk- language and environment.

However, we must bear in mind that the platform from which Squeak begins is our *end-goal.* In our project we deliberately target the much

```
(define fac
  (lambda (n)
    (if (= n 0)
      1
      (* n (fac (decr n)))))))
(fac 3)
```

Figure 4.6: The factorial function in Lisp, built around lists.

more difficult task of achieving our dream programming system from a lesser starting point, and in the process uncover the general story one can expect to re-trace in any similar project.

In COLA, it is unclear how the Lisp-like programming language is built beyond the broad outlines. What is clear is that the bootstrapping process is carried out by means of source code files written in some text editor. In other words, it wisely takes advantage of the affordances of its Unix platform (at its *inter*-process layer), avoiding the Turing Tarpit failure mode described in Section 4.5.3.

4.7.2 *High-Level Language for BootstrapLab*

If we take JS, and strip away the concrete syntax, we get a resulting tree structure of function definitions, object literals, and imperative statements. A similar structure with similar semantics would be obtained from other dynamic languages. In fact, this would largely resemble Lisp S-expressions under Lisp semantics; hardly surprising considering Lisp's famously minimal syntax of expression trees. Furthermore, the evaluation procedure for Lisp is simple and well-established. For these reasons, in addition to COLA's influence, we designed a Lisp-like tree language in the substrate. This way, we provide high-level constructs (if/else, loops, functions, recursion, and so on) for in-system programming.

We call it Lisp-*like* because Alignment (Force 3) encouraged us to *revisit* Lisp's design to better fit with our substrate. Ordinary Lisp is based on lists whose elements have *implicit* meanings based on their positions; this aligns with its substrate made of S-expressions. However, our *map-based* substrate comes with named labels and suggests a language based around maps whose entries are explicitly *named*, so we called it *Masp*.[15].

Figure 4.6 shows a Lisp definition of the factorial function followed by a call, while Figures 4.7 and 4.8 show versions of the Masp equivalent at different levels of notational sugar. In terms of *internal* representation, the final, most verbose version in Figure 4.8 (no. 5) would be the one

---

15 This is not too hard to come up with, but we would like to credit the origin of the name to (C2 Contributors 2014b) and related discussion.

1. Human-Optimised   **2. De-Sugared**   **3. Explicit Keys**

apply: name:
define `fac` as: $n \Rightarrow$ define `fac` as: define `fac` as:
apply: arg_names:
  match n with: lambda n to: lambda [n] body:
    $0 \rightarrow 1$ to n, apply: to: n, apply:
    $\_ \rightarrow n \cdot \mathtt{fac}(n-1)$ $0:1$ 0: 1
  `fac` 3 $\_ : n *$ apply `fac` with $n =$ _: apply: `mul`
      apply `decr` to: n *1:* n
  apply `fac` with $n = 3$ *2:* apply: `fac`, n:
      apply: `decr`, 1: n
  apply: `fac`, n: 3
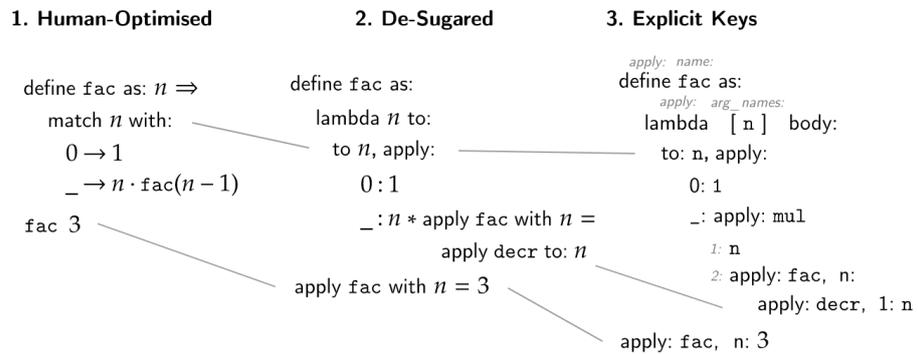
Figure 4.7: The factorial function in Masp, built around maps. Notational Freedom would allow us to present Masp in something like the first, heavily sugared form; the way in which the underlying expressions are built out of maps would only become apparent after user-configurable de-sugaring.

4. Inline Keys   **5. Verbose Syntax**

apply: `define`, name: `fac`, as: { apply: `define`,   name: `fac`,   as: {
  apply: `lambda`, arg_names: [1: n] , body: apply: `lambda`,   arg_names: { 1: n },
    to: n, apply: body: {
      0: 1   to: n, apply: {
      _: apply: `mul`, 1: n, 2:   0: 1, _: {
        apply: `fac`, n: apply: `mul`, 1: n, 2: {
          apply: `decr`, 1: n   apply: `fac`, n: {
apply: `fac`, n: 3   apply: `decr`, 1: n }}}}}}}
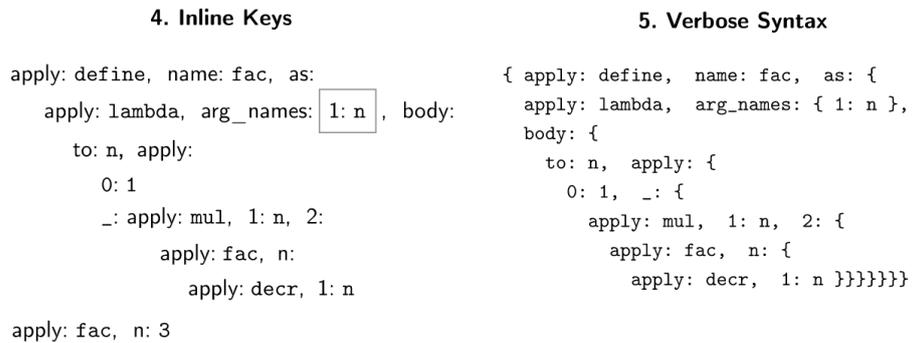
Figure 4.8: The final steps of de-sugaring the Masp factorial function. A technical apples-to-apples comparison with Lisp might involve the final stage, but this would omit the ways its verbosity could be mitigated by Notational Freedom.

to compare to Lisp. It contains strictly *more* information than the Lisp code (explicit named arguments), but given our key goal of Notational Freedom, this should not imply a burdensome interface. We would prefer to interact with such programs in the first notation (Figure 4.7, no. 1), which uses infix operators, different typefaces, and a pattern-match construct that does not exist under that name internally.[16] Yet the underlying data structures need be no less machine-friendly than those of Lisp, preserving the powerful metaprogramming potential that makes it so valuable. Thanks to Notational Freedom and Explicit Structure, there is no need to compromise one of these for the sake of the other; there is hope to get the best of both worlds here.

A fuller description of Masp is given in Appendix D. The diagrams here are an "artist's impression" of a plausible notation design, but BootstrapLab is in far too primitive a state to support this at present. The actual rendering of Masp in the system (see Figure 4.15) is even more verbose than no. 5, because different map entries live on separate lines and are not inlined to save space. However, it is still worth understanding what all this work we are doing will ultimately enable.

4.7.3 *Choosing an Appropriate Implementation*

What would it mean to "implement" a high-level language like Masp? First, we have the binary choice of whether to implement an *interpreter* or a *compiler.* Additionally, we are faced with a choice of implementation *platform.* In BootstrapLab, this could be JS (i.e. the platform for BootstrapLab itself) or BL-ASM. Once one of these is working, we could even choose to use Masp itself (as a self-hosting compiler or meta-circular interpreter). Some of the possibilities could be automatically generated from others; the six combinations fit together in a neat dependency structure, which we can visualise using *tombstone diagrams* (Figure 4.9).

The key advantage of a compiler is that it can create programs and thus save us translation work. However, at this early and crude stage in the system, we will undoubtedly need to carefully step through the execution of Masp code. Since the experience of debugging a Masp program compiled to BL-ASM would be very tedious and unpleasant, we cannot content ourselves with having a compiler alone. Moreover, we find it strictly easier to make an interpreter, which simply executes the effects of the code, than a compiler, where we must additionally think about how to achieve those effects in the output language. So we need some sort of interpreter at minimum; the question is, implemented in which platform?

- BL-ASM would have the advantage that the interpreter exists in-system. However, it would be difficult to write by hand, and could

---

16 Similarly to how we handled conditionals in BL-ASM (Section 4.5.5), if/else is a pattern match in Masp, which is implemented as applying a *map* (here seen as a function "literal") to the value. See Section D.2.1 for more details.
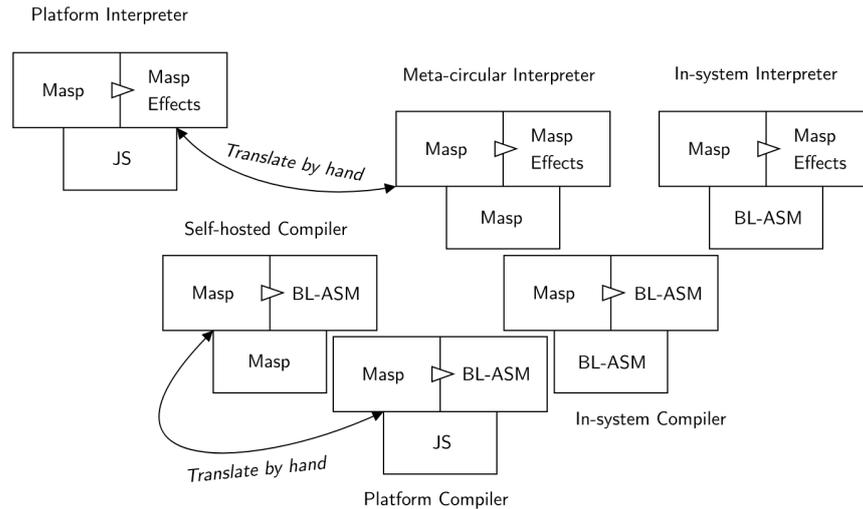
Figure 4.9: The six variations on "implementing Masp" linked as tombstone diagrams. For example, a *platform compiler,* running in JS, takes Masp code as input and outputs the corresponding BL-ASM. If the Masp code in question does the same thing as this JS, then the platform compiler will generate an *in-system* compiler that runs in BL-ASM. Interpreters can only be leaf nodes; their outputs are immediate "effects" in the state instead of code that will achieve the same thing.

be generated automatically at a later stage (Figure 4.9). Even if we succeeded, it would be hard to maintain owing to the lack of a high-level description.

- Masp itself could work, as long as we were prepared to hand-translate it into BL-ASM or JS to bootstrap it off the ground (recall Section 2.3.1.2). Writing in Masp would only be worthwhile if the language were beyond the prototype stage and had decent editing infrastructure. However, owing to its verbose structure in the tools available to us at this early stage (i. e. our platform text editor), this would be a mistaken choice.

- JS, unlike BL-ASM, is a high-level language. Unlike Masp, it is very well-optimised for our existing programming tools (e. g. the text editor). Therefore, it is the most sensible choice.

Realistically, according to the dependency structure in Figure 4.9, we would have to begin with JS even in the unlikely case where it was hand-compiled from Masp. Because we could better code in JS than Masp at this early stage, we began with JS with a view towards hand-translating to Masp at a later stage. Additionally, because we wanted to experiment with Masp and refine it as soon as possible, we began with an *interpreter* and deferred compilation to a later stage. In other words, our projected future for BootstrapLab traces the following path through Figure 4.9:

1. Create a *platform interpreter* and experiment with Masp.

2. Depending on how long it takes for Masp to be more appealing than JS, write either a *platform compiler* or a *self-hosted compiler* first; then hand-translate it into the "other" language (this is the bi-directional arrow at the bottom of Figure 4.9).

3. Run the platform compiler on the self-hosted compiler to obtain an *in-system compiler.* At this point, the platform compiler could be "jettisoned", i. e. deleted from the substrate code.

4. Hand-translate the platform interpreter into a *meta-circular interpreter.*

5. Run the in-system compiler on the meta-circular interpreter to obtain an *in-system interpreter*. At this point, the platform interpreter could also be jettisoned.

This is the "bootstrapping" process for languages (Section 2.3.1.2) applied in the explicitly-structured world of BootstrapLab. While this does not yet resemble "scuplting" the Masp language bit-by-bit, it does lay a foundation to get there; perhaps the size of the cycle could be shrunk from "entire Masp program" down to individual expressions.

One such way to make the process more incremental is by treating State and Change separately. When implementing a processor (by which we abbreviate "interpreter or compiler") in-system, all its internal state will also be stored in-system. However, *state* can still live in-system even if the processor's *code* lives on the platform. This takes advantage of the platform's high-level language for *code* while leveraging the product system for debugging and visualisation of *state.* In this way, a later port to in-system implementation is simplified, because half of it (the state) is done already.

The transition from platform to in-system implementation can be even more gradual; once the internal state is stored in-system, it becomes possible to port *parts* of the interpreter code piecemeal to in-system instructions, invoking them from the remaining parts running outside. We codify this as Heuristic 6:

Heuristic 6 (In-state operation)**.** Store high-level-language processing *state* in-system, even if the language processor *code* remains running on the platform. This will ease porting the processor to in-system implementation and support a gradual transition ("Escape The Platform", Force 2).

4.7.4 *Implementing Masp for BootstrapLab*

To summarise the above logic, "Escape The Platform" (Force 2) encouraged us to start executing Masp expressions early to get experience with

```
- expr:
     apply: fac
     n: 1
```

Figure 4.10: The `expr` part of the Masp context contains the current expression being evaluated. This represents the initial state for applying the factorial function with parameter `n` bound to 1.

the language. We chose to implement a *platform interpreter* for Masp using JS as this was the easiest way to achieve that.

A naïve approach would simply implement the standard Lisp interpreter routines (`eval` and `apply`) as recursive JS functions. However, this would miss an opportunity for visualisation and debugging that is already present in our substrate. Instead, we followed Heuristic 6 and had intermediate interpreter *state* reside in-system. This made a later in-system port easier by doing half of the work now.

Lisp evaluation is done by walking over the expression tree. At any point, we are looking at a subtree and will evaluate it until reaching a primitive value. Ordinarily, the "current subexpression" is an argument to `eval` at the top of the stack, where the stack records our path from the original top-level expression. Since we already had a tree visualisation, we used that instead of a stack. We did, however, need to maintain references to parent tree nodes (see Section C.3) in order to backtrack towards the next unevaluated subexpression once the current one has been evaluated. Furthermore, instead of *destructively* replacing tree nodes with their "more-evaluated" versions, we "annotate" the tree instead. This design choice follows Subtext (Edwards 2005), making it possible to trace provenance and enable novel programming experiences. Figures 4.10–4.13 show some examples. For further details, see Section D.2.

What can be changed at the user level? Depending on the particular path taken through Figure 4.9, the semantics of the language may or may not (yet) be modifiable from the user level. The user is almost able to use the high-level language in-system for convenient programming ... but may be unable to enter the expressions conveniently in the first place. This matter will be addressed shortly.

4.8 PAY OFF OUTSTANDING SUBSTRATE DEBT

> Port all remaining temporary infrastructure into the system, taking advantage of the infrastructure itself and the high-level language. The result is a *self-sustainable* programming system.

If we had developed both the state editor and the high-level programming language in-system, we would already have an elementary

```
                            - expr:
function closure            - apply:
                              expr: fac
    for fac                 - value:
       - - - - - - - - - - -  - arg_names:
                                1: n
                              - body:
                                to: n
                              - apply:
    match n with:              0: 1
                              - _:
        0 → 1                    apply: mul
        _ → n · fac(n − 1)      1: n
                              - 2:
                                  apply: fac
                                - n:
                                    apply: decr
                                    to: n
       - - - - - - - - - - -  env: () => defining_env
                            - n:
                              expr: 1
                              value: 1
```
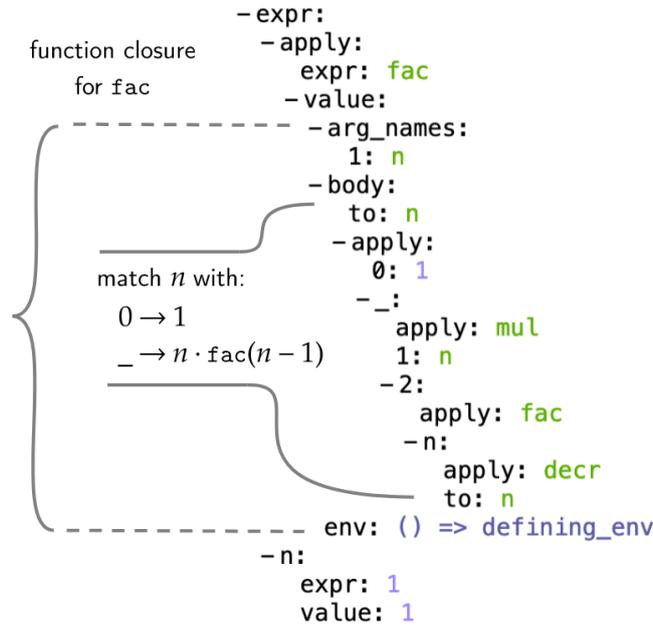
Figure 4.11: After some evaluation steps, both the original expression (the name fac) and its value (its function closure) are visible. Similarly, the literal expression 1 has evaluated to itself. Part of Figure 4.7 has been included to help deceipher the verbose notation displayed by the state view.

```
                            - expr:
                            - to:
                              expr: n
                              value: 1
match n with:               - apply:
                            + expr:
                            - value:
                              - literal:
        0 → 1                 0: 1
        _ → n · fac(n − 1)   - _:
                                apply: mul
                                1: n
                              - 2:
                                  apply: fac
                                - n:
                                    apply: decr
                                    to: n
                              env: () => defining_env
                            - env:
                            - entries:
                                n: 1
                            + parent:
```
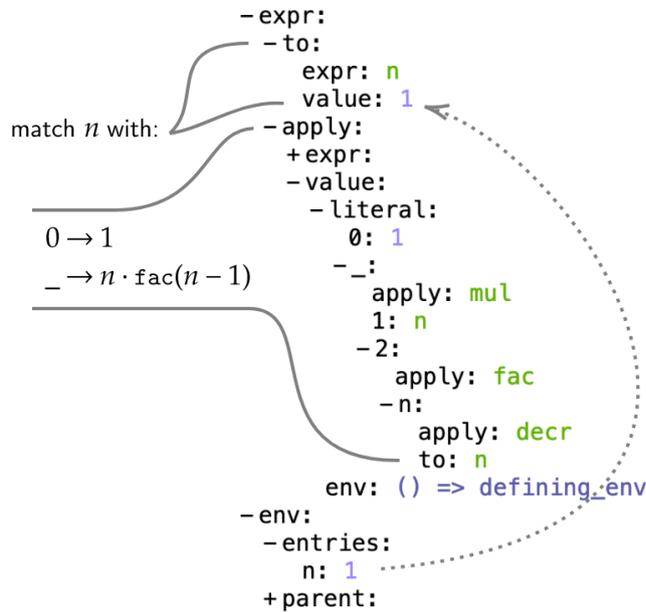
Figure 4.12: The next step of evaluation, read as: "To the value 1 (which came from the expression n), apply this function literal in an environment where n is bound to 1".

```
-expr:
  -apply:
    expr: mul
    -value:
      body: (c, args) => { upd(c, 'value',
      args[1]*args[2]); return true; }
  -1:
    expr: n
    value: 1
  -2:
    apply: fac
    -n:
      apply: decr
      to: n
-env:
  -entries:
    n: 1
```

Figure 4.13: Some steps later, we have an application of a built-in multiplication function whose JS code is visible (see Section D.2.2.2 for how JS primitives work). The second operand is an as-yet unevaluated recursive application of fac.

self-sustainable system at this point. This would have been our only option if we had been somehow stuck with only a primitive platform, as was the case at the dawn of computing in the 1940s. With a richer platform available, one can choose to implement a state viewer, editor and high-level programming language on it following Heuristic 5. Since these will now run outside of the product system, they will be functionally part of the substrate—yet they do not belong there. This *substrate debt*, incurred due to "Escape The Platform" (Force 2), now needs to be paid off.

4.8.1   *Substrate Debt in Squeak*

The closest analogue of this concept that we can see in Squeak is in the difference between its development and deployment platforms. Unlike our BootstrapLab, Squeak was not developed in its deployment platform; all the "hard" work was done taking advantage of Smalltalk. Thus, the main "debt" to pay off was to port it all to C, which was accommplished by automatic translation (Ingalls et al. 1997):

Compile the interpreter to make it practical:

- Design a translator from a subset of Smalltalk-80 to C.
- Implement this translator.
- Translate the virtual machine to C and compile it.
- Write a small C interface to the Mac OS.
- Run the compiled interpreter with the new image.

In our case, unlike Squeak, there is no need for any such compilation to JS; our development and deployment platforms are the same.

However, JS was not nearly as full-featured as Smalltalk, which meant we accumulated a different type of substrate debt via the temporary infrastructure we had to build.

Squeak exemplifies the ideal development journey, quickly and straight-forwardly ending up with a self-sustainable system:

> It was easy to stay motivated, because the virtual machine, running inside Apple Smalltalk, was actually simulating the byte codes of the transformed image just five weeks into the project. A week later, we could type "3 + 4" on the screen, compile it, and print the result, and the week after that the entire user interface was working, albeit in slow motion. We were writing the C translator in parallel on a commercial Smalltalk, and by the eighth week, the first translated interpreter displayed a window on the screen. Ten weeks into the project, we "crossed the bridge" and were able to use Squeak to evolve itself, no longer needing to port images forward from Apple Smalltalk. About six weeks later, Squeak's performance had improved to the point that it could simulate its own interpreter and run the C translator, and Squeak became entirely self-supporting.

4.8.2 *Substrate Debt in BootstrapLab*

In such an ideal development journey, we would have a high-level programming language and a basic state editor in-system by now. This did not happen for BootstrapLab.

The Masp interpreter we developed used in-system state, but controlled it from JS. Our state viewer was also fully implemented in JS. Editing took place through the browser development console. The alternative, creating a Masp interpreter and state editor in-system using the low-level BL-ASM instructions, had been technically possible but prohibitively tedious. The in-system tooling was far from supplanting the existing platform interface of JS in the text editor. Continuing to use the latter was, therefore, the only sensible choice to make progress.

Nevertheless, to make the high-level language and editor part of a self-sustainable programming system, they ultimately need to be implemented in-system. Thus we incurred a *substrate debt* which we owe to "Escape The Platform" (Force 2) and now need to pay off. The advantage of delaying this work and ending up with Masp is that we can at least port JS to Masp, which is more convenient than porting to BL-ASM (recall Figure 4.9). Generally, such substrate debt should be paid off as soon as the indebted implementation is complete. In total, we had three parts of it to pay off:

1. Replace the temporary JS state *viewer* with an in-system JS state *editor*
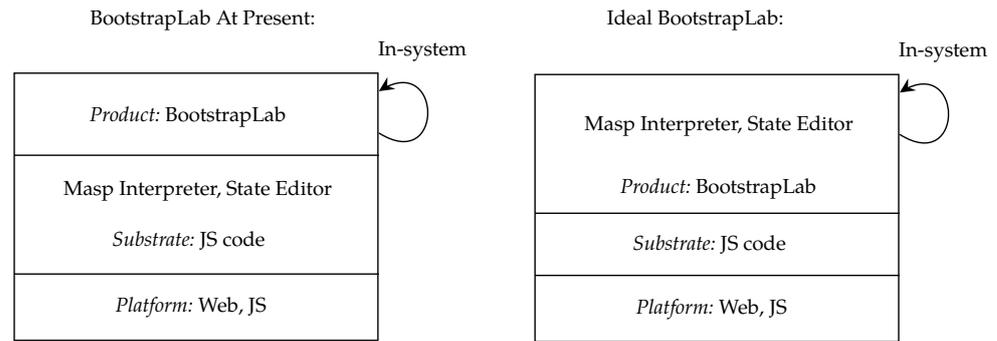
Figure 4.14: The Masp interpreter and state editor currently live in the substrate, but they belong in the product system. This is *substrate debt.*

2. Port this JS state editor to Masp

3. Port the Masp interpreter from JS to BL-ASM

In BootstrapLab, we split the task of supplanting the temporary state viewer into two halves. We first replaced the temporary viewer, which exists fully outside of the system, with an editor that uses in-system state and graphics, but is controlled from JS (recall Heuristic 6). We then started to port the editor code from the platform to in-system Masp, which is where we are at the time of writing. Figure 4.14 shows the situation in terms of platform/substrate/product.

4.8.3   *Supplanting the Temporary State Viewer*

Once we could run Masp programs in the substrate, we needed a better way of entering and editing them. We desired a state editor in the graphics window to make the existing state view obsolete. Considering the proof-of-concept nature of this work, we created a rudimentary tree editor that nevertheless surpassed the existing practice of issuing commands in the JS console.

To edit state in JS, we needed to either address its parent with a full path from the top level, or to use a reference previously obtained this way. To set a primitive value, we would type a JS command including the key name and the value. For example, the following console command could be issued to make a shape red:

```
upd(ctx, 'scene', 'shapes', 'children',
    'yellow_shape', 'color', '0xff0000')
```

This was not a high bar to clear. Evidently, we could greatly improve the experience by simply *clicking* on the relevant key name and typing.

We implemented a basic tree view in the graphics window (Figure 4.15). Nodes can be expanded and collapsed, and entries can be changed by clicking and typing. The display is "on-demand" and breadth-first: map entries are read upon expanding a node. This means

Figure 4.15: Left: tree editor in graphics window. Right: temporary state viewer
in the DOM. The +/- controls for expanding and collapsing nodes
are not present in the editor, but these operations can still be
performed by double-clicking.

that cycles in our graph substrate do not pose a fatal problem, as they
did in the temporary state view (see Section C.3). The basic CRUD
operations are accounted for as follows:

- Update (primitive): The Tab key commits the value and selects
  the next entry.

- Create: If the above runs past the end of the map, special "new
  entry" fields for entering a new key and value are created. These
  disappear if abandoned without committing.

- Update (composite): Enter commits a new, empty map and selects
  the "new entry" field within it.

- Delete: Backspace on an empty value will delete the entry. If it
  was the only entry, it will be replaced with the "new entry" field.

- Read: The display of the entry in the graphics window provides
  this.

It is worth noting that Alignment (Force 3) applies here too: the
structure of the substrate clearly has implications for the structure of
the editing interface. If our substrate consisted of low-level bytes, the
traditional hex editor interface would be an immediate requirement.

Such an interface could plausibly be simpler to implement than the complex nested tree editing we needed for BootstrapLab. This suggests a potential feedback into the choice of substrate: *a more complex substrate will require a more complex editor.*

We might even be tempted to conclude that it only makes sense to use a low-level substrate, since we can complete a basic editor sooner and subsequently work in-system. This neglects the fact, however, that the higher-level structures of our substrate would inevitably be required at some point. Thus we would have to do the same work anyway, but only once we had suffered through the human-unfriendly low-level substrate.

It is also remarkable that, in this restricted interaction domain, we finally *did* manage to surpass our default JS text editor. There is a cost to typing out concrete syntax like : and {} for JS map structures, as well as ensuring indentation is correct.[17] For entering state structures, we found the structured editing style to be quicker. As a result, where previously we might have added new persistent state in our JS startup code, we now directly entered it into the system and persisted it manually.

There is a caveat to all this. The whole exercise was in the service of paying our substrate debt from earlier—pulling up the state viewer "ladder" that had got us to this point. Ideally, we would have built up its replacement in-system. Yet as pointed out, JS was still the most appealing way to program at this stage, so we used it for this editor as well. In other words, we took on a new debt in order to pay off the first one! To resolve it, we would port the JS to Masp—a process which is underway at the time of writing for both the Masp interpreter and the state editor.

In general, at the end of this stage the substrate should not contain anything that we wanted to be modifiable in-system. Thus:

What can be changed at the user level? The structural "syntax" and semantics of the high-level language can be changed. The graphical interface of the system can also be changed, including the concrete notation for programs and data, which we turn to next.

4.9   PROVIDE FOR DOMAIN-SPECIFIC NOTATIONS

> Use the self-sustaining state editor to construct a more convenient interface for editing high-level expressions. Add *novel notations and interfaces* as needed. Use these not just for programming new end-user applications, but also to improve the product system itself.

Because BootstrapLab is currently in the middle of the previous stage, this section describes our plans for when this is complete. At such a

---

17   Although JS will run fine with or without indentations, source code is meant for *human* eyes—and ours regarded proper indentation as a requirement for legibility.

point in the journey, the editor implementation would now be part of the product system, so we could modify it from within to our heart's content.

We admitted earlier how, in BootstrapLab, we had not managed to bring the system interface up to a level where it became more effective than JS. With the implementation of a state editor, we came closer. Indeed, for entering general state structures, it is not obvious how to improve on it. Yet when it comes specifically to Masp expression structures, we must enter their verbose details even though they are highly regular and could be captured through fewer interactions. If we streamline this *subdomain* of the BootstrapLab interface, it would make Masp programming just as convenient as typing JS, if not more so—and we could finally escape the text editor entirely.

4.9.1 *A Taster*

First, we propose a restricted proof-of-concept of notational variation from within the system. We choose to target a small part of the problem: the Masp `apply` node, a frequent enough occurence that a small improvement will be helpful.

In the general state editor, one must type each of these key-value pairs for a function application:

```
apply: setColor
red: 11
green: 22
blue: 33
```

Instead, we desire something like autocomplete for parameters. Instead of typing `apply`, we press `a` and enter `setColor`. Subsequent tabbing should fill in the parameter names automatically and let us type the arguments. Furthermore, as a small notational difference we will omit the word `apply`:

```
setColor
red: _
green: _
blue: _
```

The underscores represent unfilled fields right after this structure gets created.

To reprogram the editor to work like this, we would do the following from within the editor. Navigate to the Masp code structures for the editor that synthesise the graphics structures to display a given state node. Enter Masp code that checks for the key `apply` in the given node and, if present, only renders the value of the key instead of the key

itself.[18] Then, navigate to the code that handles key input. Add code that, when `a` is pressed, will insert a new map containing the `apply` key, render this to the graphics, and send text input to its value text box. Finally, navigate to the code that commits an entry on a `Tab` keypress. Change this to detect if it is for an `apply` key and, if so, to look up the symbol in the value node and treat it as a Masp function closure. For each entry in the `arg_names` field, add an entry to the map with a dummy value, render this to graphics, and then proceed with the default behaviour (highlight the next entry in the map). Depending on the precise implementation, it may be the case that only subsequent edits will be rendered this way. Otherwise, care may be necessary to refresh and re-render the entire editor state.

4.9.2   *A More Ambitious Novel Interface*

The above "taster" is a simple example of an interface that could be plausibly implemented early in BootstrapLab's self-sustaining lifetime. Beyond this, it points to a more general class of extensions which would support *projectional editing.* Projectional editors (Steimann et al. 2017) are a class of programming interfaces that provide domain-specific interfaces for certain program subexpressions, such as LATEX-style mathematical expressions to replace ASCII renderings (recall Section 3.3.2.3). We would do well to import such ideas into BootstrapLab. We proceed to sketch how such an interface would be added to the system, and how its ramifications are different from ordinary non-self-sustainable projectional editors.

As an example, suppose we want to program some fancy graphics. Fancy graphics require sophisticated vector mathematical formulae. In textual programming languages, these are expressed as ASCII with limited infix notation. The Gezira/Nile project (Amelang 2012a,b) attempted to improve on this with Unicode mathematical syntax. An extreme endpoint would be LATEX. All we have at the moment is something worse than all of these: verbose, explicit tree views spanning multiple lines.

We think ahead with a view towards making the fancy graphics programming more pleasant. Suppose we decide that we would ideally like to implement them with the aid of concise mathematical notation, as opposed to our current state of verbose trees. How could we achieve this?

The broad approach would be similar to our previous taster example. We would have to start, again, at the code that renders state into graphics. Add a condition that checks for a `math` key, which we would use as a tag to hint at this display preference. Enter code to translate operator names

---

18 Admittedly, this will display all structures with an `apply` key this way, but further discretion is just as achievable with further programming. The point is that this can be changed at the user level.

to Unicode symbols, place them at infix positions, place parentheses appropriately, and render the whole thing to a single line in the tree view (ideally keeping the tree structure of the expressions in the graphics state). Then, modify the input handling and tree navigation code to appropriately work on this *inline* tree structure. And so on.

The above points are, of course, a high-level sketch, but it is *programming* all the same and is plausible to achieve with a high-level language. Techniques from the literature would be helpful, such as Hazel's calculus for editing structures with holes (Omar et al. 2019), or bi-directional synchronisation between the rendered graphics and the state's ground truth (Hempel et al. 2019).

4.9.3 *Real Example: Colour Preview*

While the above speculation has value, it is only fair that we show a real example. As a very modest proof-of-concept: instead of a hex string for colours, let us see a rectangle with that colour instead.

Some Masp code (Figure 4.16) lives under the global register called `render_map_entry`. It is invoked from the JS function for rendering map entries in the tree editor. It checks if a map entry is named "color", and if so, returns an appropriately coloured box with a grey border. Figure 4.17 shows the difference.

These results so far could have been achieved without going to the trouble of implementing the hook in Masp. JS would have sufficed. However, having this code in Masp lets us do something not possible with the equivalent JS. The system has access to the explicitly structured Masp code and can choose how to display it.

As this Masp code is evaluated on its own source tree, it encounters the hex constant `0xaaaaaa` representing the grey border and displays this *with the very notation it implements.* See Figure 4.18 for the difference. This is a minimal demonstration of Innovation Feedback (Section 3.3.1.5): there is no artificial barrier to innovations (in this case, displaying colour previews) applying to their own implementation code.

The major caveat is that, unlike the hex strings, this new notation is *read-only.* If we try to use the tree editor on this entry, it will most likely break; we would have to revert back to the old notation in order to change the colour. The obvious next step would be to go from *notation* to more of an *interface;* to turn the colour *preview* into a colour *picker* and ensure the tree editor is not fragile with respect to it. However, this was not necessary to demonstrate our main point that Innovation Feedback is now feasible.

```
match key_name with:                      to: key_name,  apply:
                                            _: { apply: quote,  to: unhandled }
     _ → "unhandled"                        color:
  color →                                     apply: block
    · local box  =          (interior)        1:
        ┌─────────────────────────────┐         apply: local,  name: box,  is:
        │ width: 0.45, height: 0.2     │           width: 0.45,  height: 0.2
        │ center: (0.875, −0.1, −0.9)  │           center: { right: 0.875,  up: -0.1,
        │ children:        (border)    │                     forward: -0.9 }
        │   ┌───────────────────────┐  │           children:
        │   │ width: 0.5, height: 0.25│ │             1:
        │   │ center: (0, 0, −1)      │ │               width: 0.5,  height: 0.25
        │   │ color: 0xaaaaaa         │ │               center: { right: 0,  up: 0,
        │   └───────────────────────┘  │                         forward: -1 }
        └─────────────────────────────┘           color: 0xaaaaaa
    · box.color := value                   2: { apply: set,  map: box,  key: color,
    · return box                                         to: value }
                                           3: box
```

Figure 4.16: This Masp code checks if a map entry is named "color". If so, it returns an appropriately coloured box with a grey border. Otherwise, it returns the string unhandled. The left is how we would prefer to read it; the right is slightly tidier than what is actually visible in the state viewer.
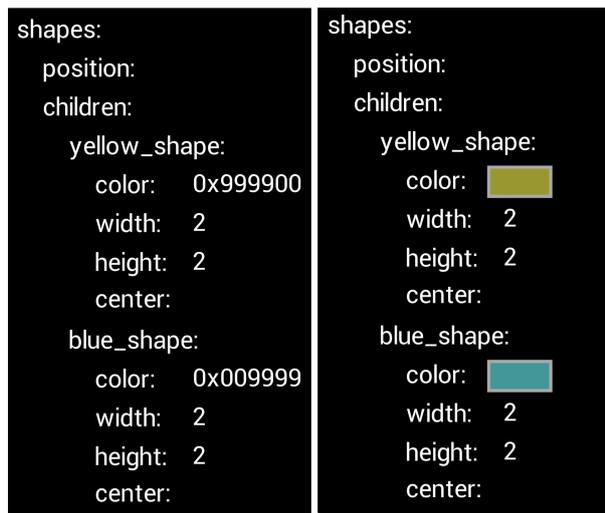
Figure 4.17: Before (left) and after (right) activating the Masp rendering hook.

Figure 4.18: The grey colour constant displays as the hex string `0xaaaaaa` in the right-hand HTML tree view. However, in the left-hand tree editor, this code runs on its own source representation, turning the colour constant into a grey box instead. This is a minimal example of Innovation Feedback.

4.9.4   *The Key Takeaway*

In the non-self-sustainable world, a projectional editor is implemented in some traditional programming language and interface; say, Java. The domain-specific notations can benefit a wide variety of programs created using the editor. Yet, this range of beneficiaries nevertheless forms a "light cone" emanating out from the editor, never including the editor itself. For example, any vector formulae used to render the interface of the editor will remain as verbose Java expressions, along with any code for new additions to the editor. The tragedy of non-self-sustainable programming is that it can never benefit from its own innovations.

Conversely, in BootstrapLab, the benefits of the new notation spread across the whole system; the "light cone" *includes* the editor implementation itself. If we previously had to squint and parse verbose maths trees in the implementation of the maths rendering, we could now open up the code again and see it rendered in the more readable way that it itself implements!

In COLA, notational variation appears to be limited to variation in concrete syntax. Our uncompromising insistence on *explicit, non-parsed structure* at the core of BootstrapLab, while costly in terms of interface implementation, was precisely in order to be free of such a restriction in the end. While one *could* implement a multiline text field with syntax highlighting in BootstrapLab, it is at least crystal-clear that a vast array of other interfaces are possible, unimpeded by any privileging of text strings.

4.10   SITUATION, TASK, USER, IMPORTANCE

At this point, it would be nice to be able to evaluate BootstrapLab as a programming system, particularly with regard to our official desiderata of embodying the Three Properties. However, they were novel concepts that we introduced earlier in the dissertation, and the task of evaluating programming *systems* in general is a novel and uncertain area (Edwards et al. 2019a). In the worst-case scenario, we might be tempted to write a long verbal argument along the lines of "on the one hand, we have Notational Freedom in *this* sense, but not in *that* sense..." but this would be rather anticlimactic. Fortunately, we developed a more systematic means to evaluate BootstrapLab and systems more generally, but its size means it must constitute the next chapter.

Still, as a stop-gap, we turn to an existing methodology that can let us summarise what we have done while indicating the specific points on which it could be judged. We close this chapter with a description of BootstrapLab expressed in the STUI (Situation, Task, User, Importance) framework of Olsen (2007), summarised by Hempel (2018). The premise is that a system is aimed at some audience of *Users* performing

some set of *Tasks* in certain *Situations*. The system claims to address some problem of *Importance* related to these areas of attention. Different types of claims are codified with names in the framework, for example "Problem Not Previously Solved" (targeting a novel STU context) or "Empowering New Participants" (expanding the set of Users).

It is worth separately applying this approach to (a) BootstrapLab itself, (b) the technique we have presented as a sequence of steps, and (c) the "ideal BootstrapLab" that we would develop with more time and resources.

Firstly, BootstrapLab itself is made to help the author (User) discover how to interactively achieve self-sustainability and explore its effects (Task) for research (Situation). The claim to Importance is that such a system did not previously exist (Problem Not Previously Solved).

The technique was developed to help individual programmers (User) escape the limitations of their go-to programming environments for general programming tasks (Task) where they are able and willing to put in this investment (Situation). Such an investment of time and work may not be possible or appropriate in some situations. However, we see it as existing on the same continuum of existing programming investments, such as writing a utility function or library to ultimately pay itself off in productivity gains. This contribution falls under "Generality", applying to all sorts of programming systems taken as the platform. It also qualifies for "Empowering New Participants" in the sense that the benefits of the Three Properties (especially Self-Sustainability) need no longer be confined to specific programming systems like Smalltalk; one *ought* to be able to have them "bolted on" to one's own preferred platform.

Finally, the "ideal BootstrapLab" would be an example of this process applied to our preferred platform, the Web browser (User, Situation). It would function as a Smalltalk-like "personal dynamic medium" for both exploring problem spaces and implementing solutions (Task), but one that neatly slots into our familiar programming practices (e. g. does not require installing and learning Smalltalk). While it is necessarily lifted up by programmers, the availability of mood-specific notations could make it of use to non-programmers (Empowering New Participants).

While the STUI structure is helpful for summarising what we have done, it is quite broad and does not include the Three Properties that we intended BootstrapLab to embody. As promised, our remaining task in this dissertation is to examine this issue of evaluating BootstrapLab and programming systems more generally.

TECHNICAL DIMENSIONS OF PROGRAMMING SYSTEMS

We introduced the concept of a *programming system* in Section 2.1. Not only is such a concept necessary for framing our work in Chapter 4, there is also a growing interest in programming systems in both research and industry. Yet while programming *languages* are a well-established concept, analysed and compared with a common vocabulary, no similar foundation exists for the wider range of programming *systems.* In this chapter,[1] we will examine this problem and propose a framework of "Technical Dimensions" to kickstart systematic research on programming systems. We will then make our Three Properties (Section 1.5) more precise as sets of dimensions under this framework. We will then use these dimensions to assess how BootstrapLab fulfils the Three Properties and evaluate it on that basis.[2]

5.1 BARRIERS TO PROGRAMMING SYSTEMS RESEARCH

Researchers are studying topics such as *programming experience* and *live programming* that require considering not just the *language*, but further aspects of a given system (as evidenced by workshops dedicated to these topics: LIVE (2023) and PX (2023)). At the same time, companies are building new programming environments like Replit (repl.it 2022) or "low-code" tools like Dark (Dark Language Team 2022) and Glide (Glide 2022).

However, the academic research on programming suffers from a lack of common vocabulary. While we may thoroughly assess programming *languages*, as soon as we add interaction or graphics into the picture, evaluation beyond subjective "coolness" becomes fraught with difficulty.[3] Comparisons make the most sense when comparing "like for

---

1 Adapted from our paper for Programming 2023 (Jakubovic et al. 2023), which grew out of an earlier proposal (Jakubovic 2020a).

2 It might seem appropriate to also perform an evaluation via the Cognitive Dimensions of Notation (Green and Petre 1996). However, this would not actually tell us anything interesting; the novel contribution of this system is not its notation. The interface is minimal and unpolished for reasons of expediency. The point is not that we have come up with a new notation or UI that will improve programming; the notation is something that each user should fit to themselves according to subjective preference. The important goal is that the system *supports* the usage of different notations for different contexts. Notations in BootstrapLab should be a free parameter, so it does not make sense to apply Cognitive Dimensions to BootstrapLab *itself*, and it does not provide any value to analyse the placeholder interface in this way.

3 The same difficulty in the context of user interface systems has been analysed by Olsen (2007). Interesting future work would be a detailed analysis of publications on pro-

like", yet graphical programming systems may be so varied that it is unclear what the stable points of comparison should be. Moreover, when designing new systems, inspiration is often drawn from the same few standalone sources of ideas. These might be influential past systems like Smalltalk, programmable end-user applications like spreadsheets, or motivational illustrations like those of Victor (2012).

Instead of forming a solid body of work, the ideas that emerge are difficult to relate to each other. The research methods used to study programming *systems*, as we defined them in Definition 9 lack the rigorous structure of programming *language* research methods. They tend to rely on singleton examples, which demonstrate their author's ideas, but are inadequate methods for comparing new ideas with the work of others. This makes it hard to build on top and thereby advance the state of the art.

Studying *programming systems* is not merely about taking a programming language and looking at the tools that surround it. It presents a *paradigm shift* (Kuhn 1970) to a perspective that is *incommensurable* with that of languages. When studying programming languages, what matters is in the program code; when studying programming systems, what matters is in the behaviour of the system. As documented by Gabriel (2012), looking at a *system* from a *language* perspective makes it impossible to think about concepts that arise from interaction with a system which are not reflected in the language.

5.2   OUR PROPOSAL

We propose a common language as an initial step towards more progressive research on programming systems. Our set of *technical dimensions* seeks to break down the holistic view of systems along various specific "axes". The dimensions identify a range of possible design choices, characterised by two extreme points in the design space. They are not fully quantitative, but they do allow comparison by locating systems on a common axis. We do not intend for the extreme points to represent "good" or "bad" designs; we expect any position to be a result of design trade-offs. At this early stage in the life of such a framework, we encourage agreement on descriptions of systems first in order to settle any normative judgements later.

The set of dimensions can be understood as a map of the design space of programming systems. Past and present systems will serve as landmarks, and with enough of them, we may reveal unexplored or overlooked possibilities. In the absence of such a map, the field has not

---

gramming systems to understand this issue in depth. One notable characteristic is that publications tend to present (parts of) new systems. This is the case for 5/6 and 6/7 papers in the LIVE 2020 and 2021 workshops respectively (Hempel and Lau 2021; Hempel and Perera 2020). In contrast, publications in the field of programming *languages* often address specific issues of interest to a greater number of languages.

been able to establish a virtuous cycle of feedback; it is hard for practitioners to situate their work in the context of others' so that subsequent work can improve on it. Our aim is to provide foundations for the study of programming systems that would allow such development.

In short, while there is a theory for programming languages, programming *systems* deserve a theory too (Jakubovic 2020a). It should apply from the vast scale of operating systems to the comparatively small scale of language implementations. It should be possible to analyse the common and unique features of different systems, to reveal new possibilities, and to build on past work in an effective manner. In Kuhnian terms (Kuhn 1970), it should enable a body of "normal science": filling in the map of the space of possible systems , thereby forming a knowledge repository for future designers.

We will develop *self-sustainability, notational freedom,* and *explicit structure* as Technical Dimensions, following on from the discussion in Section 3.3. For each one, we will give examples that illustrate the range of values it spans. Then we will apply them to BootstrapLab. The rest of our extensive catalogue of dimensions can be found in Appendix E, organised into related clusters: *interaction*, *notation*, *conceptual structure*, *customisability*, *complexity*, *errors*, and *adoptability*.

5.3 DIMENSIONS, QUALITATIVE AND QUANTITATIVE

There is a problem where the most easily *measurable* properties are not necessarily very interesting, while the interesting properties are not straightforwardly measurable. We have discussed our Three Properties intuitively and qualitatively in Chapters 2 and 3. However, as they stand there is too much ambiguity for anything resembling an objective, plausible consensus on how much they are present in a given system. Therefore, in the next section we will break them down into narrower dimensions that we can apply for evaluating BootstrapLab. A few of these will be boolean (asking whether something is possible or present in a system) but most will be quantitative *penalty* dimensions. This means that maximising the value of one of our Three Properties (e. g. Self-Sustainability) will correspond to *minimising* its constituent penalty dimensions.

5.3.1 *How We Define and Apply the Dimensions*

We say a penalty dimension is "quantitative" in that its definition intuitively describes an amount of which there can be more or less, even if we leave the question of how to actually measure it numerically as future work. We do not have the scope to compare the various ways these quantities could be defined for measurement, and it would be misguided to pick one simply for the sake of having numbers. Where a relevant quantity does already exist (e. g. lines of code) we may propose

it as a measure for the dimension. Otherwise, we will use a variation on the Likert scale used in psychology (Likert 1932). Instead of "strongly agree" to "strongly disagree", we will assign the scores "minimal", "low", "moderate", "high" and "infinite".

With such a scale, it would be possible to rigorously measure a system against the dimensions by means of a questionnaire and analysing the distribution of responses (expecting consensus around a single score, and perhaps re-working the dimensions for which this is not the case). However, such an approach is beyond the scope of this work. Instead, when we apply these dimensions to evaluate BootstrapLab in Section 5.5, we will give our own personal assessment of each score along with its justification. We think that the narrow focus of the dimensions makes it likely that such judgements would be aligned with those of the reader. Even in the case of serious disagreement, this narrow focus would make it easier to productively reach agreement in a way that would be much harder for the complex, qualitative definitions of the Three Properties from which they derive (Section 3.3). Therefore, while we agree that giving our personal assessment in terms of the Three Properties directly would be hard to judge objectively, we believe that doing so on the finer scale of the dimensions is appropriate. Further discussion of these issues and suggestions for future work will be found in Section 7.1.

5.3.2 *Aggregation and Simplification*

It is worth noting that even once we have broken down a high-level concept into several low-level dimensions, the high-level concept can still be considered a *dimension* if we define some suitable aggregation of the scores of its constituent dimensions (this could be a simple sum, a weighted average, or something more sophisticated). We will not practice this, but it is worth keeping in mind as we encounter complications and decide how to respond to them.

For example, it might be objected that a dimension cannot simply apply to a system as a whole, but actually takes different scores for different *parts* of a system. We can answer this objection with an interpretation of the dimension as precisely such an aggregate of its application across different parts of the system. Ultimately, programming systems are complex and any property we speak of may apply at multiple levels. For practical purposes including those of our evaluation of BootstrapLab, we must make simplifications and apply our dimensions to an entire system as best we can. Later in Section 7.1, we will return to the complexities we have elided here.

5.4 THE THREE PROPERTIES AS DIMENSIONS

We will now proceed to break down each of our Three Properties into dimensions (or, in the boolean case, "criteria", but we will stick to the general term).

5.4.1 *Dimensions Constituting Self-Sustainability*

In light of the points in Section 3.3.1, we can discover some key dimensions of self-sustainability with the help of an existing programming system that is not self-sustainable. Using the terminology from Section 3.3.1.3, let us cast the *Web browser* as the *product system* (i. e. that which we wish to make self-sustainable) and C++ as the *platform* (i. e. the system we use to implement the product). What would it take to make the browser self-sustainable?

5.4.1.1 *Minimise The Substrate Size*

Recall from Section 3.3.1.4 that the *substrate* is the portion of the product system not accessible from its user level. In the case of the web browser, it is the C++ code constituting its implementation. To get a self-sustainable system, the substrate must be minimised by shifting implementation out of it and into the programming capabilities of the product system. In this case, most of the named entities in the C++ code are stuck at the implementation level, inaccessible at the user level of JS, so we must move the former into the latter.

To sketch how this process could be carried out systematically, we can begin with the graphical surface of the product system. For each graphical element, we inquire into the causes of its display; this will include graphical rendering code, but also the data that is being rendered and the code that generated it. By tracing backwards in this way we discover the web of causes that produced the shape on the screen. This will often go through the user level (JS), but if we keep tracing back, we will hit the implementation level. Each time this happens, we port the code from the implementation level to the user level.

We continue this until it is no longer feasible; for example, there will ultimately have to be some native machine-code interpreter for JS in the running system. In practice, there would need to be the usual investments in JIT compilation and optimisation technology as seen in VMs for Smalltalk and other languages.

These ideas suggest a dimension of *substrate size* as a penalty for self-sustainability. In other words, a self-sustainable system minimises this dimension. We already compared the strategies of doing this minimisation earlier or later in Section 4.5.3. A reasonable measure of substrate

size does exist as the number of lines of code[4] that implement it, so we will use this measure in our evaluations later.

5.4.1.2   *Minimise Persistence Effort to Fix "Delete By Default"*

As we discussed from Section 3.2.1.1 onward, the activities of a running process under Unix are considered disposable. In order for a system to be self-sustainable, it has to be able to preserve developments of its state through process termination. The standard VM solution is to have most of the system state saved in an "image" file and concentrate the substrate in a runnable binary that need not be changed. However this is accomplished, *persistence* of run-time changes is necessary to encourage indefinite evolution of the system. This applies to the whole browser, but could also be a concern for individual tabs or web pages that can be closed or refreshed.

This suggests another penalty dimension of *persistence effort.* To illustrate the range of values, we offer the following examples:

- Any system which automatically persists to an "image" (Lisp, Smalltalk) or otherwise (Webstrates; Klokmose et al. 2015) causes *minimal* persistence effort on the part of the user.[5]

- A system with a manual "save" button that persists all state would have almost-minimal persistence effort. This comprises both the need to remember to save and the act of pressing the button.

- A system where one must repeat a manual procedure over different parts of the state to persist all of it would have *moderate* persistence effort.

- A typical programming language in a "vanilla" state (e. g. excluding third-party libraries) has *high* persistence effort for its runtime data structures, owing to the "Delete By Default" policy of the Unix Paradigm (Section 3.2.2). With the use of a specific third-party library or framework (such as an Object-Relational Mapper) this persistence effort may be reduced. In the absence of such a framework, the programmer would have a lot of work to do in order to persist all state (wrap every variable and stack frame in code for loading and saving its value).

- We could ascribe *infinite*[6] persistence effort where it is impossible to persist state. This is easier to imagine in the case of an end-user

---

4 Under realistic conditions, it matters less whether these are "significant" lines of code or contain whitespace and comments, than that this convention is *consistent* between two systems being so compared.

5 Arguably, this effort is zero, since the user does not have to think about it. However, we will stick to the term "minimal" for consistency with our stated scoring terminology.

6 An infinite score can be interpreted as saying: it would take less effort to duplicate the source code of the system and add persistence at its implementation level, than it would to persist state using user-level functionality.

application with no scripting capability; if the developers failed to persist something (e. g. the position or sizing of a window) then the user cannot do anything about it. In the case of a programming system, hard barriers to persistence include inaccessible state (e. g. in JS, one cannot refer to stack frames or read their state) or a lack of enumerability (e. g. there is no way to traverse all objects in the system and thereby persist them).

It may be objected that this measure should be considered on a piece-meal basis *per piece of state* instead of on the system as a whole. For example, a system could have infinite persistence effort with respect to some state (e. g. stack frames; Basman et al. 2016) but low persistence effort with respect to everything else (this being the effort invested to set up an Object-Relational Mapper for the rest of the state). As mentioned in Section 5.3.2, given such a fine-grained application of this measure and a method of weighting each contribution, we could derive a convenient aggregate measure of persistence effort for the whole system. However, this is too complicated for the scope of our work here, so we will give an overall impression of the property without systematically going into finer detail.

5.4.1.3   *Support Code Viewing and Editing*

The browser's JS console makes it possible to make some changes expressible as JS commands, modulo the caveats in Section 1.3.1; these would need mitigating. The source code can be viewed but not edited; we would need to make a small change so that the source code viewer could also be used to make persistent edits to code. These points suggest boolean dimensions of *code viewing* and *code editing.* An example of a system that has both is Smalltalk with its class browser.

5.4.1.4   *Support the Manipulation of Code as Data*

Once we can type text inside the system, we will be able to write code. However, this code will be inert unless the system can interpret data structures as programs and actually execute them. This is the case whether these data structures were created manually or by a program. If this is not possible, re-programming the system will not be possible (beyond selecting from a predefined list of behaviours). The browser does already satisfy this criterion since JS has an `eval()` function that can execute a string of JS code. This suggests a boolean dimension of *data execution.*

Any system with an `eval` function has this property, such as Lisp. In the low-level binary world (Section 3.1.1) the fact that the *instruction pointer* can be pointed at bytes in memory and interpret them as instructions also qualifies. A negative example exists in a language like

C, where there is no `eval` function.[7] In such a case, one may employ the workaround of defining a mini-language (whether textual, or a binary bytecode) and an interpreter C function. It is important to be clear on which level the property would be thus established: what we called the *product system* (the program being implemented by the C code) would have data execution but the *platform* (the C language itself) would remain without it.

5.4.2   *Dimensions Constituting Notational Freedom*

In Section 3.3.2 we mentioned the salient stages prior to notational freedom, namely syntactic and linguistic freedom. Recall that *syntactic* freedom involves specifying grammars for textual languages, while *linguistic* freedom adds custom layout, rendering and editing of textual symbols. In Section 3.3.2.5 we framed the issue as one of *removing* artificial barriers to using local notations, while respecting the essential complexity of implementing a notation itself. This suggests three penalty dimensions for the effort involved in using custom syntax, linguistic forms, and general graphical notations.

One complication is that we defined these stages as successive generalisations, i.e. notational freedom includes and implies linguistic freedom which includes syntactic freedom. It would not be very helpful to observe that a system has high syntactic freedom and then claim this gives it high linguistic freedom by virtue of the former being contained within the latter. That is not quite what we intend by the term "linguistic freedom". Instead, we would be concerned with linguistic freedom *above* the syntactic and notational freedom *above* the linguistic.

Therefore, our dimensions (all penalties) are as follows. They are minimised if a custom syntax, language, or notation can be "slotted in" once it exists, with no resistance from the system:

CUSTOM SYNTAX EFFORT.   The work required to use a custom syntax, not counting that required to specify the syntax itself (e.g. as a grammar). COLA (Piumarta 2006) and OMeta (Warth 2009) score low on this, since they are specifically designed for this purpose. Most programming languages have *infinite* custom syntax effort, because their parsers are separate programs that adhere to a fixed grammar that cannot be changed by statements in the language. This includes JS despite its inclusion of a regex sub-syntax, HTML despite its inclusion of JS and CSS, and C# despite its LINQ sub-language for queries; these examples may exhibit syntactic *plurality*, but there is no way to include arbitrary[8] user-supplied syntax for use in the source code.

---

7 Modulo "smashing the stack" (Aleph One 1996), which is platform-dependent and abnormal C programming style.

8 There is always the option of including string literals in the code containing custom syntax, but the available syntax will always be restricted by the escaping rules of the

CUSTOM LANGUAGE EFFORT.   The work required to use custom language-like notation beyond syntax, not counting that required to implement the rendering and interaction. Most programming languages, COLA, and OMeta get an infinite score here, while MPS (Voelter and Pech 2012) and Eco score low.

CUSTOM NOTATION EFFORT.   The work required to use custom graphical notation beyond what we called language in Section 3.3.2.3, not counting that required to implement the rendering and interaction. To relate the examples from Section 3.3.2.4, non-infinite scores are rare, only exhibited in a few systems. Eco, owing to a screenshot showing inclusion of a picture, scores non-infinite on this dimension. From their discussion in Section 9.2 of the paper (Diekmann and Tratt 2014), it is likely to score High or Moderate rather than Low because arbitrary graphical notations are a novel unexplored use case for the system for which it has not been optimised. On the other hand, the Glamorous Toolkit (Chiş et al. 2015) appears built to support arbitrary graphics in its text editor interfaces, suggesting a lower value on this dimension. ThingLab I (Borning 1981) and II (Moloney et al. 1989) would similarly score low, at least insofar as diagrammatic notations are concerned (as opposed to integration with the editing of Smalltalk code). Babylonian Programming (Rauch et al. 2019), built on the self-sustainable Lively4 (Lincke et al. 2017), is also likely to have a lower score on this penalty dimension.

5.4.3   *Dimensions Constituting Explicit Structure*

The best way we have found to detect Explicit Structure is as a lack of Implicit Structure, which we break down into *producer* and *consumer* concerns. On the producer side, we have an editor with an interface creating and changing a data structure. This is saved and passed onto consumers, which can be collaborators using editors or a programmer writing code to use the data structure.

It is tempting to define Implicit Structure in terms of the producer's editing interface: a text editor has lots of it, while a structured or projectional editor lacks it. But this is incompatible with our desire for Notational Freedom; if someone wishes to use a text editor *interface* to type their data structures into existence, they should be free to do so.

Equally tempting is to locate Implicit Structure in the interchange file format, such as whether it is a text file. Yet as long as the system handles the loading and saving for this file format, it makes no difference from a consumer's point of view and they do not *experience* the downsides associated with Implicit Structure.

---

host language. Moreover, such an approach would be seen as an "abuse" of the host language, compared to a language that was designed for syntactic freedom.

So if Implicit Structure is not about the interface, or how the data is really stored, what is it? The definition we are interested in is about how much users or programmers must be *aware* of it and devote cognitive resources to working with it. On the producer side, this manifests as which types of syntax errors or more general *format* errors they are able to save and pass on to consumers (Section 2.3.3). On the consumer side, Implicit Structure is revealed by the amount of code we have to write to deal with parsing, serialising, escaping, loading and saving, and so on. Therefore we declare two penalty dimensions:

FORMAT ERRORS.   How many different types of format errors can be introduced, saved as invalid structures, and passed to consumers, such that they will halt with an error? For example, text editors allow all possible syntax errors to be saved, along with several format errors (e. g. type mismatches and use of undeclared names). However, a text editor interface that refused to save invalid files could form part of a system with Explicit Structure. Block or structure editors may prevent all format errors from being saved, which would constitute the minimal value of this dimension.

The approach of Hazel (Omar et al. 2019) is unique in having language semantics that *tolerates* fully typed "holes" representing missing expressions, which obviates concerns about persisting or transferring such programs. This highlights the fact that Explicit Structure is about *mismatches* between the producer and consumer of data, rather than producer leniency or consumer intolerance *per se.*

STRING WRANGLING EFFORT.   How much code has to be written to convert between Implicit and Explicit Structure? Explicit Structure implies a minimal value for this and would look something like the following:

```
data = load('filename')
```

Here, there are zero lines of string wrangling. Only one line, translating between the filesystem and the internal system namespaces, is required to prepare the data structure for use.

If such a `load` function is already present, then users experience no string wrangling effort for the use cases of this function, i. e. the file formats it supports. If the function does not exist, and a user must write string wrangling code on an ad-hoc basis, this dimension is correspondingly high relative to that format. Suppose the user factors this ad-hoc string wrangling into their *own* implementation of the `load` function; this implementation effort would count towards the dimension, but would pay for itself in the reduced string wrangling effort thereafter; this situation would lie somewhere between the previous two.

These considerations all establish scores for this dimension *relative* to a particular file format or string syntax. These could be aggregated to form a score for a particular program which uses several such formats. However, if we are trying to assess the *programming system* along

this dimension, we would have to somehow aggregate across all possible programs one could create with the system, including the various different formats they are likely to include.[9]

Recognising that different programming systems are targeted at different goals and have differing strengths and weaknesses, the possibility space could be refined into all *likely* programs or use cases of the programming system, weighted by the probability of a user of the system wanting to create such a program. This opens up further decisions about this user and whether we should additionally aggregate across possible (or likely) users of the system. We could go further, but we think the complexity is clear; as mentioned in Section 5.3, we will simply give our judgement about how BootstrapLab as a whole scores on this dimension and leave more sophisticated approaches to future work (Section 7.1).

5.5 EVALUATING BOOTSTRAPLAB

Having finally distilled the Three Properties into Technical Dimensions, we will now apply them to BootstrapLab to gauge how far we succeeded at our goals.

5.5.1 *Measures of Self-Sustainability*

Substrate Size: 1550 LoC.

BootstrapLab's homogeneity of state contributes to a smaller substrate than a design where system registers and user data lived in two separate partitions of state. There is deliberately only one system namespace: the state graph rooted at the top-level registers. Some of these names have special functions in the low-level BL-ASM, but otherwise this namespace is free for user additions. These can be added manually in the in-system editor or in code by the primitive `store` instruction.

The present graphical state of the system lives entirely in a special part of the system state: the `scene` tree. Therefore, at any given moment, it is possible to change what the graphics window will display. However, there are two limitations:

1. The range of these changes is constrained to the range of graphical primitives currently understood by the substrate which it passes on to THREE.js. Currently these are limited to axis-aligned flat-coloured rectangles and basic text of a uniform size, style, colour, etc (see Section B.1.2).

---

9 There is a large variety of *existing* data storage formats (e. g. JSON and XML) and an infinite variety of potential *custom* formats that could be created on an ad-hoc basis (e. g. chat messages containing special escape sequences).

2. The behaviour that affects the graphics currently lives in JS. This means that the logic according to which the tree editor renders map entries is inaccessible to in-system code.

Indeed, there are about 1550 lines of JS off-limits to the actions of the system. At least 33% of this, however, constitutes our substrate debt: the Masp interpreter and tree editor (Section 4.8.2). In a further-developed version, these could be moved out. Even so, in such a further-developed version, the substrate may be larger anyway by exposing more types of graphical primitives. This suggests that capabilities of the platform provide a lower bound on the substrate size: if the platform provides a way to draw a circle, but the substrate does not expose this to the system, then we have reason to interpret this as an incomplete programming system. On the other hand, the substrate may expose a more general set of graphics operations that allow the system to draw circles itself, say to a pixel surface.

Persistence Effort: Moderate. Part or all of the state graph can be manually persisted via the `export_state()` function in the browser console. This means that in-system progress can be saved, even though it would be better for the user experience to have this done automatically. It is clear that indefinite evolution is *permitted* but perhaps not quite *encouraged*.

Code Editing: present. Code editing is crude but feasible via the in-system tree editor for most use cases. In rare cases, the JS console must be used.

Data Execution: Present. Because of Alignment (Force 3), low-level instructions that change state are represented as ordinary maps with certain format constraints. The instruction set is sufficient for constructing arbitrary graph structures in the state, including programs composed of instructions. The `next_instruction` register can be pointed at such a list and execution can be started using `run_and_render()` in the JS console. The analogous properties hold for high-level Masp code which is also represented as maps.

VERDICT.   BootstrapLab's practical capacity to change its implementation is limited by its substrate debt, while its manual persistence adds friction to working within the system. Still, there are no hard barriers to self-sustainability.

5.5.2   *Measures of Notational Freedom*

Custom syntax effort: moderate. Because of substrate debt, it may not be possible to make changes at the user level such that a string can be "executed" according to custom syntax and semantics via a click or key combination. However, it is possible to use the `js` "escape hatch" instruction to embed arbitrary JS code to do the appropriate process-

ing. In the absence of substrate debt, it would be possible to edit the relevant parts of the system to support custom syntaxes—both textual, as strings, but also "structural" with different map structures to those that Masp expects. This direct editing of the system could still be costly, and adopting the techniques in the Lisp half of COLA (Piumarta 2011) or OMeta (Warth 2009) could bring custom syntaxes closer to being "slotted in" without difficulty.

Custom language effort: moderate. This follows similar considerations, except the substrate debt related to graphical capabilities and the lack of exposure of certain platform graphical primitives are also relevant here. However, implementing language-like notations may be aided by the existing layout capabilities of the tree editor.

Custom notation effort: moderate. Again the reasoning is similar, but in this case the tree editor layout capabilities may not be directly helpful. Here, the lack of exposure of platform graphics primitives limits what can be achieved. Still, as shown in Section 4.9.3, use of custom notations is *feasible* as long as the appropriate "hook point" is available, which we added to the substrate specifically for the proof-of-concept.

VERDICT.   BootstrapLab in its current state *permits* notational freedom but at a moderate cost. This is still an improvement on the norm of *infinite* cost in programming systems (recall the examples in Section 5.4.2).

5.5.3 *Measures of Explicit Structure*

Format errors: low. The structure editing interface of the tree editor eliminates the existence of syntax errors for data, Masp code, and instructions. Within these structures, certain format errors are possible (e. g. failing to supply required arguments to an instruction).

String wrangling effort: low. Because all data, including instructions and Masp code, is embedded in map data structures edited structurally, there is little need for the user to write parsing or serialising code. The sole exceptions are with hexadecimal colour codes, where the initial # character may need stripping, and rendered map entries, where the colon : needs attaching and stripping. Strings are of course present, but as primitive values without substructure (i. e. names).

VERDICT.   BootstrapLab succeeds at our goal of being based on explicit structure. All "code" or "language" structures are represented directly instead of as text strings.

5.6 CONCLUSIONS

BootstrapLab embodies our Three Properties to a satisfactory extent; it is strongest on Explicit Structure and weaker on Notational Freedom and Self-Sustainability. In a system developed without attention to these

properties, the default practices of programming would end up raising barriers to their realisation (Section 1.4). We deliberately designed BootstrapLab to support these properties and avoid hard barriers to them, so there is significant potential for improvement from further development efforts. We will outline this future work in Section 7.2.

Beyond this dissertation, there is interest in developing new programming systems. Such systems go beyond the simple model of code written in a programming language using a more or less sophisticated text editor. They combine textual and visual notations, create programs through rich graphical interactions, and challenge accepted assumptions about program editing, execution and debugging. Despite the growing number of novel programming systems, it remains difficult to evaluate the design of programming systems and to see how they improve over work done in the past. To address the issue, we proposed a framework of "technical dimensions" that captures essential characteristics of programming systems in a systematic fashion.

This framework puts the vast variety of programming systems, past and present, on a common footing of commensurability. As more and more systems are assessed in the framework, a picture of the space of possibilities will gradually emerge. Some regions will be conspicuously empty, indicating unrealised possibilities that could be worth trying; this is how we regard BootstrapLab. In this way, a domain of "normal science" is created for the design space. Designers of the next generation of programming systems can then build upon the successes and lessons of those that came before.

RELATED WORK

In the preceding chapters, we referenced programming systems and research literature that were directly relevant to the topics we were discussing. At this point, having presented BootstrapLab and evaluated it according to the Technical Dimensions, we can situate our contributions in the setting of more general related work. We will cover the research group from which many of our influences originate, work relating to each of the Three Properties, and the study of programming systems. Note that some references may be repeated from earlier for the sake of completeness.

6.1 STEPS AND THE LEGACY OF VPRI

The COLA system design (Piumarta 2006), from which we have drawn the most in this work, emerged from the now-retired Viewpoints Research Institute (VPRI). VPRI aimed at creating "fundamentally new computing technologies", which is particularly visible throughout the 6-year project known as "STEPS towards a new computing" (Amelang et al. 2011, 2012; Kay et al. 2006, 2007, 2008, 2009; VPRI 2010). The aim was to fully replicate a familiar graphical end-user operating system, with applications, in under 20,000 total lines of code. Such an ambitious goal provided the constraint needed to force innovation in distinguishing essential and accidental complexity and ways to reduce the latter. Innovations included the widespread use of domain-specific languages supported by OMeta (Warth 2009) and investment in highly flexible core abstractions as evidenced by COLA's object and composition models (Piumarta 2011; Piumarta and Warth 2008).

Two of our Three Properties, Self-Sustainability and Notational Freedom, recur as themes in the STEPS work.[1] Self-Sustainability is exhibited by COLA. Mood-Specific Languages in COLA and those supported by OMeta demonstrate what we called *syntactic freedom* in Section 3.3.2.2. The Gezira (Amelang 2012a) and Nile (Amelang 2012b) projects utilise a custom mathematical syntax, taking advantage of Unicode characters for expressing graphics code that is cumbersome in ordinary languages. While this does not amount to Syntactic Freedom, it is a good example of the sort of thing that Syntactic Freedom enables; we can expect more innovations like this only if it is not too difficult to deploy a custom syntax once one has been designed.

---

[1] Explicit Structure is absent, but this is to be expected owing to its niche status; see Section 6.5.

6.2   SELF-SUSTAINABILITY AND ITS THEORY

The only name we are aware of for the concept we called Self-Sustainability is "self-sustaining", seen in the two workshops on such systems (Rose and Hirschfeld 2008; Rose et al. 2010) in which parts of the COLA design were published. We derived our term "Self-Sustainability" to be able to refer to a property that can be present or absent in programming systems. We have referred to Self-Sustain*able* systems rather than Self-Sustain*ing* systems for consistency with this.

Self-Sustainability appears in related work as Smalltalk variants. Much of the STEPS work took place via the Squeak variant (Ingalls et al. 1997), of which Pharo (Bergel et al. 2013) is a descendant. Glamorous Toolkit (Chiş et al. 2015) is a "moldable development environment" built in Pharo. The Lively Kernel (Ingalls 2008) is a Web implementation of a Smalltalk-like environment; Fizzygum² is similar.

The problem with all of these, as regards our goals in this dissertation, is that they are all complicated software systems, with their own histories, made to be practically useful to researchers or industry. As such, knowledge of the principles and tricks for *implementing* these systems is sequestered away in the practical experience of their developers and not written down in a discoverable location. Furthermore, it would be difficult to separate knowledge about the property we are interested in (Self-Sustainability) from the various other aspects of the implementation of these systems (such as useful libraries or optimisations).

These facts made it clear to us that we would be best equipped to understand Self-Sustainability by trying to achieve it ourselves in a minimal context with minimal distractions. The related systems are self-sustainable in order to be useful for certain communities; BootstrapLab aims for Self-Sustainability to better understand it (along with the other two Properties).

We are only aware of a few sources that aim at a similar goal of understanding. The "Meta-Helix" approach of Polito et al. (2015) is intended to reduce confusion when implementing meta-circular Meta-Object-Protocols. As we mentioned in Section 2.3.1, meta-circularity is a specific manifestation of Self-Sustainability. The exhaustive development of Procedural Reflection for Lisp-like languages by Smith (1982) is helpful for its philosophical rigour, e. g. the use-mention distinction and careful precision of terminology. The process described by Evans (2001) is a parallel of what we did with BootstrapLab in the restricted context of batch-mode interpreters of text strings. The introductory sections of Piumarta (2006) and Piumarta and Warth (2008) constitute a good explanation of Self-Sustainability and why it is desirable, as does Cook (2018).

---

2 http://fizzygum.org/

6.3 VIDEO GAMES

Video games and game engines are led in the *direction* of Self-Sustainability by their nature as highly dynamic, long-lived virtual environments. They are an example of the issues we covered in Section 3.2.4.3. The creative world-building nature of games means that requirements change more often than other types of software; development is partly a process of discovery of what the final product should be. Accordingly, it is important to rapidly prototype and iterate ideas, especially by artists or other specialists who may not be expert programmers. This incentivises ways to try out new ideas without the costly operation of restarting a large, resource intensive process or the even more costly operation of recompiling the underlying program. It also incentivises editing tools (for game levels, internal scripting languages, or configuration) to be part of the game software itself. After development, these internal "developer tools" may either be stripped from the version shipped to customers, or hidden (in which case, determined hackers will eventually uncover them sooner or later).

Since games also have strong requirements for real-time performance (responsiveness to input, rendering of complex scenes, simulating physics, synchronising a shared world across the internet, and managing worlds too large to fit into memory all at once) languages like C++ are a standard choice for implementation. However, the default data structuring mechanisms of these languages (such as C++ classes) must regrettably be *avoided* for directly modelling highly dynamic relationships between objects in the simulated world. A C++ class promises a *static commitment* to *always* contain its listed member variables of the specified types, member functions of the specified signatures, and to *always* remain in any inheritance relationships with other classes. This rules out a majority of the dynamic change that is inherent to the behaviour of a game and its development process. For this reason, standard game programming patterns (Nystrom 2014) build infrastructure to work around this and support the modelling of objects whose relationships and contained properties may *change* during run time. "Entity Component Systems" (Ehrlich 2013) are a widely used architecture for this purpose. The general pattern of working around static commitment is known as "Greenspun's Tenth Rule" (C2 Contributors 2014a):

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug ridden, slow implementation of half of CommonLisp.

6.4 NOVEL NOTATIONS VERSUS NOTATIONAL FREEDOM

"Visual Programming" contains many examples of custom notations for program code and data. Sketchpad (Sutherland 1963) is an early,

influential example of diagrammatic notation augmented with the dynamic capabilities of computation. DesignScript (Aish 2012) is a visual language for computer-aided design and computation. The Apparatus editor[3] is a Web-based editor for dynamic graphics influenced by Sketchpad. Bret Victor's presentations (Victor 2013) demonstrate programmatic graphics based on direct manipulation instead of textual code. Data is represented in Boxer (diSessa and Abelson 1986) and Forms/3 (Judith Hays 1995) as named, nested boxes; in Boxer, programs reside in textual code boxes. Programming By Example and Programming By Demonstration (Cypher 1993; Lieberman 2001) involve custom notations designed for either representing program structures or for supplying example input-output pairs from which to infer general behaviour. Sketch-n-Sketch (Hempel et al. 2019) uses a textual and graphical notation that are synchronised with each other.

There are many more examples of custom programming notations and interfaces (Edwards 2017; Nickerson 1994; Reese 2022). Despite the abundance of earnest attempts at general-purpose or special-purpose notations, programming is still mostly performed via plain text. This is understandable given that much of the cited research is experimental and that programming infrastructure (editors, compilers, version control etc.) only supports plain text. There is also a seductive failure mode of imposing a single notation for all purposes, in denial of the fact that different people have preferences about the tools they work with, and that notations necessarily have different suitabilities for different purposes.

Thus, while we respect the effort invested in Visual Programming and custom notations and wish these efforts success, we avoid the conclusion that there is an optimal notation or set of notations (across users and situations) waiting to be found. Instead, we see as-yet unrealised gains in supporting and encouraging the *ad-hoc* use of custom notations on an opportunistic basis, wherever the user judges them to be most helpful. We observe much effort expended over the years on developing custom notations but comparatively little on enabling them to be used together *à la carte*, which is why we focus on Notational *Freedom.*

We see an effort towards Notational Freedom in Hempel and Chugh (2020), where *tiny structure editors* are derived from the code of `toString` functions. This highlights the fact that by "notation" we mean a dynamic, two-way *interface* and not just a rendered picture. Other efforts include the Mood-Specific Languages of COLA (Piumarta 2006) and OMeta (Warth 2009), the Eco editor (Diekmann and Tratt 2014), Jet-Brain's MPS (Voelter and Pech 2012), and the Glamorous Toolkit (Chiş et al. 2015). Our issue with the latter two is similar to what we said about Smalltalk-like systems in Section 6.2: they are impressive and useful as industry tools in which to get things done, but their size and complexity make it hard to learn the essential aspects of supporting

---

3 http://aprt.us/

Notational Freedom. Eco has the advantage of being a research project whose paper *does* cover its design and implementation, so its approach deserves a place in future work on BootstrapLab (see Section 7.2.5). The same goes for any insights or documentation for other systems which emerge during, or after, the writing of this dissertation.

6.5 STRUCTURE EDITING AND ITS VARIATIONS

Explicit Structure has precedent in structure editors, projectional editors and block-based languages (Chuchem 2023; Omar et al. 2019), but these approaches have met many difficulties in terms of widespread adoption. Text editing and plain text formats are still entrenched as the *de facto* standard in programming. *Outside* of programming, we observe the opposite situation, which allows us to point there for intuition about why Explicit Structure is a sensible concept and could be beneficial.

For example, photo editing and vector graphics programs exist and are optimised for the types of interactions involved in those domains. Photo and vector graphics files are not required to be readable in a text editor (i. e. we are free to distribute graphics in forms other than ASCII art) and so these domains do not suffer from the accidental complexities of text formats. If we observe that the textual syntax of programs is really a *proxy* for tree and graph structures, then this invites the investigation of the costs and benefits of treating programming structures the same way we do other types of files.

We are aware of two projects especially concerned with Explicit Structure: the Subtext programming system (Edwards 2005) and the Infra data interchange format (Hall 2017). Subtext explores novel programming ideas that are only feasible from a basis of Explicit Structure, while Infra is proposed as a common format unifying text and binary data. We find these sources particularly valuable for explaining Explicit Structure and arguing its benefits.

6.6 PROGRAMMING SYSTEMS AND THEIR ANALYSIS

Our "programming systems" approach lies between a narrow focus on programming languages and a broad focus on programming as a socio-political and cultural subject. The concept of a programming system is technical in scope, although we acknowledge the technical side often has important social implications as in the case of the "Adoptability" dimensions (Section E.7). This contrasts with the more socio-political focus found in Tchernavskij (2019) or in software studies (Fuller et al. 2008). It overlaps with Kell's conceptualisation of Unix, Smalltalk, and Operating Systems generally (Kell 2013).

The distinction between more narrow *programming languages* and broader *programming systems* is more subtle. Richard Gabriel noted an invisible paradigm shift from the study of "systems" to the study of

"languages" in computer science during the 1990s (Gabriel 2012), and this observation informs our distinction here. One consequence of the change is that a *language* is often formally specified apart from any specific implementations, while *systems* resist formal specification and are often *defined by* an implementation. We recognise typical programming language implementations (e. g. including an ordinary compiler and text editor) as a *small region* of the space of possible systems, at least as far as interaction and notations might go. Our attention is drawn to *interactive programming system* aspects of languages, such as text editing and command-line workflows.

6.6.1 *Programming Systems Research*

There is renewed interest in programming systems in the form of recent non-traditional programming tools:

- Computational notebooks such as Jupyter (Kluyver et al. 2016) facilitate data analysis by combining code snippets with text and visual output, in a manner reminiscent of Literate Programming (Knuth 1984a). They are backed by stateful "kernels" and used interactively.

- "Low code" end-user programming systems allow application development (mostly) through a GUI. One example is Coda (Coda 2022), which combines tables, formulas, and scripts to enable non-technical people to build "applications as documents".

- Domain-specific programming systems such as Dark (Dark Language Team 2022), which claims a "holistic" programming experience for cloud API services. This includes a language, a direct manipulation editor, and near-instantaneous building and deployment. We analyse Dark via the Technical Dimensions in Section E.8.

- Even for general purpose programming with conventional tools, systems like Replit (repl.it 2022) have demonstrated the benefits of integrating all needed languages, tools, and user interfaces into a seamless experience, available from the browser, that requires no setup.

Research that follows the programming systems perspective can be found in a number of research venues. Those include Human-Computer Interaction conferences such as UIST[4] and VL/HCC[5]. However, work in those often emphasises the user experience over technical description. Programming systems are often presented in workshops such as LIVE

---

4 ACM Symposium on User Interface Software and Technology
5 IEEE Symposium on Visual Languages and Human-Centric Computing

and PX[6]. However, work in those venues is often limited to the authors' individual perspectives and suffers from the aforementioned difficulty of comparing to other systems.

Concrete examples of systems were given in Section 2.2. Recent systems which motivated some of our dimensions include Subtext (Edwards 2005), which combines code with its live execution in a single editable representation; Sketch-n-sketch (Hempel et al. 2019), which can synthesise code by direct manipulation of its outputs; Hazel (Omar et al. 2019), a live functional programming environment with typed holes to enable execution of incomplete or ill-typed programs; and Webstrates (Klokmose et al. 2015), which extends Web pages with real-time sharing of state.

6.6.2 *Already-Known Characteristics*

There are several existing projects identifying characteristics of programming systems. Some revolve around a single one, such as levels of liveness (Tanimoto 2013), or plurality and communicativity (Kell 2017). Others propose an entire collection. *Memory Models of Programming Languages* (Sitaker 2016) identifies the "everything is an X" metaphors underlying many programming systems; for example, the "everything is a file" of Unix and the "everything is an object" of Smalltalk. The *Design Principles of Smalltalk* (Ingalls 1981) documents the philosophical goals and dicta used in the design of Smalltalk; the "Gang of Four" *Design Patterns* (Gamma et al. 1995) catalogues specific implementation tactics; and the *Cognitive Dimensions of Notations* (Green and Petre 1996) introduces a common vocabulary for software's *notational surface* and for identifying their trade-offs.

The latter two directly influence our Technical Dimensions framework. Firstly, the Cognitive Dimensions are a set of qualitative properties which can be used to analyse *notations*. We have extended this approach to the "rest" of a system, beyond its notation, with the *Technical* Dimensions. Secondly, our individual dimensions naturally fall under larger *clusters* that we present in a regular format, similar to the presentation of the classic Design Patterns. As for characteristics identified by others, part of our contribution is to integrate them under a common umbrella: the existing concepts of liveness, pluralism, and uniformity metaphors ("everything is an X") are included in our framework (Sections E.1.2 and E.3.1).

6.6.2.1 *Methodology*

We follow the attitude of *Evaluating Programming Systems* (Edwards et al. 2019b) in distinguishing our work from HCI methods and empirical evaluation. We are generally concerned with characteristics that are

---

6 Programming eXperience

not obviously amenable to statistical analysis (e. g. mining software repositories) or experimental methods like controlled user studies, so numerical quantities are generally not featured.

Similar development seems to be taking place in HCI research focused on user interfaces. The UIST guidelines (Kumar and Nebeling 2021) instruct authors to evaluate system contributions holistically, and the community has developed heuristics for such evaluation, such as *Evaluating User Interface Systems Research* (Olsen 2007). Our set of dimensions offers similar heuristics for identifying interesting aspects of programming systems, though they focus more on underlying technical properties than the surface interface.

Finally, we believe that the aforementioned paradigm shift from programming systems to programming languages has hidden many ideas about programming that are worth recovering and developing further (Petricek and Jakubovic 2021). Thus our approach is related to the idea of *complementary science* developed by Chang (2004) in the context of history and philosophy of science. Chang argues that even in disciplines like physics, superseded or falsified theories may still contain interesting ideas worth documenting. In the field of programming, where the goal is not necessarily finding out what is *true* but exploring what is *possible*, past systems are discarded or forgotten for all sorts of contingent and unprincipled reasons. Therefore, Chang's *complementary science* approach seems particularly suitable for such a field.

FUTURE WORK AND CONCLUSIONS

Having followed the development of BootstrapLab in Chapter 4 and its evaluation along the Technical Dimensions from Chapter 5, we now turn to the limitations of these two contributions and sketch the future work that they suggest. Some of the forthcoming points have already been introduced by necessity as part of earlier chapters, but here we have an opportunity to expand on them in more detail. We first address our Technical Dimensions framework, acknowledging the pragmatic simplifications we had to make and exploring the challenges of making it more rigorous. Subsequently, we acknowledge BootstrapLab's state as a work-in-progress, suggesting how to continue its development and apply its approach to other domains. Finally, we bring this work to a close by reviewing what we have presented and how it relates to the broader vision from Section 1.2.

7.1 IMPROVING THE TECHNICAL DIMENSIONS

In Chapter 5, we proposed a systematic approach to analysing programming systems that go beyond languages. We took our complex and qualitative Three Properties and derived quantitative *dimensions* to act as a proxy for them. These dimensions were narrow enough to apply to BootstrapLab. However, we were limited by scope to *argue* for the scores we gave, which is far from the everyday process of self-evident "measurement" that we ideally desire to have.

In this discussion, we will take the approach of prioritising *conceptual* clarity. We will put practical issues to one side and inquire about what a rigorously perfected Technical Dimensions would look like, even if the answer would be practically infeasible to work with. Then we will add the practical concerns back in. In the end, future work consists both of improving the theoretical concepts *and* developing practical methods of using them. We note that some of the following issues were acknowledged in the Appendix of our Technical Dimensions paper (Jakubovic et al. 2023). However, the discussion here should be taken to be a more updated version which supersedes that of the paper wherever they overlap.

7.1.1 *Scoping The Dimensions*

Even if some property fails to hold for an entire programming system, it may well still hold for some part of the system. For example, take our *persistence effort* dimension from Section 5.4.1 and imagine a Smalltalk-

like system where everything is automatically persisted except for a single, special global called x. It would be unhelpful to characterise this system as having infinite persistence effort simply because it is technically impossible to persist the entire state. Informally, we see that it has *mostly* minimal persistence effort with the sole exception of x, for which it is infinite. This example was deliberately extreme, but a more realistic one is the Web platform whose JS stack cannot be referenced or traversed by JS code.

The true scope of a dimension like "persistence effort" is more of a *field* in the physical sense, defined at every atomic piece of state in the system (the *field points*). Similarly, "custom syntax effort" from Section 5.4.2 is defined for individual syntaxes that a user may wish to use in the system, whether they already exist or merely potentially could exist. This highlights the additional difficulty that we may wish to characterise a system not only by how it happens to be right now, but by how it would perform across many *potential* use cases. We mentioned the complexity of considering "actual" vs. "potential" field points when defining *string wrangling effort* in Section 5.4.3; this dimension properly applies per "situation" consisting of a specific user, programming against a specific string format, in a specific program. These terms (user, string format, program created with the programming system) all invite further definition.

Making all this more precise, and establishing the minimal scope of the other dimensions we have proposed, is an open problem. As the next best option, we gave scores for BootstrapLab as a whole in Section 5.5 while elaborating on any relevant complications in prose.

7.1.2 *Aggregation Functions and Weights*

Having realised that the dimensions apply as more of a "field", we could recover the simpler coarser-level score that we want as some sort of *aggregation* of the finer-level scores. We mentioned this in Section 5.3.2 to pre-empt any concerns about our simplified approach to the dimensions: in the absence of a rigorous treatment of the "scoping problem" just discussed, we were forced to do the aggregation *intuitively* based on our understanding of BootstrapLab.

Future work would consider what the aggregation function should be: a simple addition (even an integral) over scores, an average, or something else. Where *potential* field points (users, programs, notations, etc.) are concerned, the infinite possibilities mean some sort of weighted aggregation would be necessary. We might compress the infinite range into a finite number of categories, one of which is a catch-all "other" category, and assign a weight for the probability or relevance of each category. This introduces a further question of how these weights are established or justified; intuitively, we know that programming systems have strengths and weaknesses and are built to cater to different prob-

lem domains or types of user (Orchard 2011), but how could this be made more rigorous? We must leave this, and the full development of the other ideas we have sketched here, as open questions.

7.1.3 *Defining Quantitative Measures or Resolution Criteria*

So far, our concerns have been conceptual; we have been talking about hypothetical "scores" and "weights". In our evaluation of BootstrapLab, we remained at this "almost-quantitative" level: we scored using the terms "minimal", "moderate", and so on (or "present" / "absent" for boolean criteria). We justified these scores by means of argument and intuition. This is suboptimal from a research perspective; a fully developed dimensions framework should enable researchers to agree or at least productively disagree (perhaps leading to new dimensions or definitions on which they *do* agree).

One improvement would be to further define *resolution criteria* for our score terms to the point that two parties, following the same definitions, would independently converge on the same scores. Alternatively, we might pursue *real* quantitative scores with concrete numbers. The challenge here would be avoiding the trap of properties that are easily quantifiable yet irrelevant or uninteresting. A good quantification (or set of quantified dimensions) should feel like a strict *improvement* on the qualitative description, rather than something that has lost an important feature of the qualitative description. In its absence, we would err on the side of staying with the qualitative and holding out for a future quantitative definition, instead of committing to the suboptimal quantitive definition for the sake of having numbers at all.

7.1.4 *Obtaining Consensus on Scores*

Even with narrow, precise definitions of how a dimension should be scored, it is a further task to establish consensus on *what* the score is for a given system. A perfectly crisp definition would be followed the same way by all parties and lead to the same conclusion, but we would not expect this in practice. Variations could arise from researchers interpreting terms in the definitions differently, aggregating differently over field points, or circumscribing systems differently (see the following Section 7.1.5).

The point of the Technical Dimensions framework is to edge towards an *objective* analysis of programming systems, on which different researchers can readily agree. A situation where there is an implicit "subject" parameter (system S scores X along dimension D *according to person P*) may be a necessary evil in the short term, but researchers should strive to debug their disagreements and improve the dimensions to the point where the "person" can be dropped.

7.1.5 *The Circumscription Problem of Systems*

One issue that has been present throughout this dissertation, and touched on in Section 2.2.1, concerns what exactly we refer to with the names "Smalltalk", "Lisp", "Java", and so on. Definition 9 certainly helps, but there is still a lot of freedom in how we draw the boundary around named programming systems.

For example, there are different distributions and implementations of Smalltalk, different versions of each one, and different concrete running instances used by people including different libraries and personal tweaks. We have implicitly taken "Smalltalk" to be some suitable aggregation over these, evoking what is common to all of them and smoothing over rare variations that would complicate our analyses. If one Smalltalk user modified their system to no longer be self-sustainable, this does not change the fact that the "typical" Smalltalk is self-sustainable. On the other hand, suppose we discovered a widely-used Smalltalk distribution that was deliberately diminished in this property; perhaps this should force us to drop the term "Smalltalk" and split our analysis into two systems instead. Making these points explicit and rigorous would involve similar work to the aforementioned issues with dimensions and their scoring.

We see this issue as no *worse* than the parallel in programming languages, where people routinely talk about "C++" or "Python" even though these have different implementations, versions and individual installations. However, it is slightly easier to point to the "essence" of a programming language due to its definition in terms of formal syntax and semantics, or at least an official specification by a standards body. In contrast, programming systems have more of a *de facto* existence as running software (Gabriel 2012) which invites appeals to popularity, community size, or influence as a substitute for formal or official definitions.

7.2 IMPROVING BOOTSTRAPLAB

In Chapter 4 we closely and carefully followed the construction of our prototype programming system, BootstrapLab. We then evaluated it against our Three Properties (as sets of dimensions) in Section 5.5. While it shows promise in demonstrating the technical feasibility of custom notations and innovation feedback, its capabilities are not as impressive as we would like, owing to its early stage of development. However, this was for a worthy cause. If we had sped ahead with its development, we would have fallen into the trap we noted in Section 6.2, adding another impressive programming system to the list without any transferrable knowledge. Instead, by taking our slower approach—reflecting on how we made design and implementation decisions and making them explicit—we have contributed a *method* that is easier to

understand than BootstrapLab's source code or commit history. From a position of being satisfied with this tradeoff, we can set out the next steps for BootstrapLab.

7.2.1 *Pay Off Substrate Debt*

BootstrapLab currently sits between the final two steps of the journey, described in Sections 4.8–4.9. We provided isolated examples of Notational Freedom (Section 4.9.3) and of Self-Sustainability (evidenced by the Innovation Feedback in Figure 4.18). This does succeed at establishing BootstrapLab as a proof-of-concept, but to go further we would need to finish paying the "substrate debts" we incurred. These consist of porting the Masp interpreter to BL-ASM and the Tree Editor to Masp. Additionally, the system will inevitably need access to more and more of the functionality available in the platform such as audio, networking, and threading. These could be exposed through the substrate, as we did for graphics via the scene tree.

7.2.2 *Make BL-ASM More Usable*

With the benefit of hindsight, we would recommend going for an instruction set that is convenient enough to *use* such that immediately building programs in-system is a worthwhile endeavour. As we admitted at the end of Section 4.5.5, our own wild adventure in minimality was a mistake in this regard, causing us to stay in JS, implement the high-level language there and port it later. It would be interesting to see the process of gradually building each component of a high-level language engine interactively in-system. Out of the possibilities in Section 4.7.3, we began with the *platform interpreter*, so exploring the others would be illuminating—particularly the *platform compiler*, which could self-host relatively quickly.

7.2.3 *Alternative Implementation Strategies*

It would be interesting to forego any temporary infrastructure (Section 4.6) at all, or build up entirely in-system without using platform tools. This would require more careful substrate design to get this process going effectively. While it could give some insight or appreciation for the hardships of early computing, its practical value in the modern environment is unclear and may be best considered a challenge for hacker wizardry.

7.2.4   *Make the System Less Fragile*

Because self-sustainability by definition exposes core infrastructure to potential user modification, the risks from mistakes or bugs are magnified. Smalltalk is famously capable of executing the code `true become: false` which results in it breaking. We encountered an instance of this class of issue in BootstrapLab. In the process of replacing a keyboard handler in-system, we typed a small change which immediately took effect. This edit was supposed to be only a part of a larger change, which in hindsight should have been committed to the system atomically. Because it was applied immediately, the new keyboard handler effectively became an incomplete function and typing was no longer possible.

This highlights the need, in any practical realisation of self-sustainability, for "guardrails" securing accidental changes to core infrastructure or "versioning" that allows changes to be directed at "the next version of the system" and applied atomically. This complements the COLA authors' recommendation (Piumarta 2006, p. 23) for stable "points of reference" in a system in which everything is flexible and homogenous, which would likely be disorienting for a newcomer accustomed to traditional programming.

7.2.5   *Import From Related Work*

To reduce the custom syntax, language and notation effort from its present score of "moderate", we would seek to learn from the approaches of OMeta (Warth 2009), Eco (Diekmann and Tratt 2014), MPS (Voelter and Pech 2012), and the Glamorous Toolkit (Chiş et al. 2015), particularly as regards implementation (since we already agree that their end results are desirable). Similarly for self-sustainability, we would like to build COLA's object model (Piumarta and Warth 2008) in the graphical substrate of BootstrapLab[1] and see if we can obtain a full COLA that way.

7.2.6   *Bootstrap on Other Platforms and Substrates*

At every step of the development journey, there were choice points where we naturally could only move forward with one of the options. Future work could explore the other branches. We cannot provide an exhaustive listing here, but will give some examples.

At the first step (Choose A Platform), all sorts of other platforms could be chosen. While COLA built on top of one "slice" of Unix—files, build tools and process memory—we see another possibility in focusing on the hierarchical *file system* as a state model to inherit through to a

---

[1] We already built the object model in a different system (Jakubovic 2020b), but its shortcomings motivated the approach we took in BootstrapLab instead.

substrate. This is one obvious *structured* substrate lurking within Unix and some of our work here is no doubt applicable to it: directories act as maps, filenames as keys and file contents as values. Symlinks could add graph structure to this tree where needed. Similar ideas can found in the Hull design (Sústrik 2019).

We acknowledge that it might feel perverse to have files contain "primitive" values, such as a single number, or to represent instructions as directory trees, since files are normally used as "large" objects. However, it must be noted that there is precedent for using them more generally for data large and small, such as in Plan 9 (Pike et al. 1993) and `procfs` (Killian 1984). If this was still too much to stomach, the default option for "code" (shell scripts) could simply be inherited on the understanding that this would impose a dependency on implicitly-structured text at the core of the system. What is most unclear is how graphics would be displayed and interacted with—possibly requiring a special binary as part of the substrate, for opening and synchronising a main window.

Supposing we keep our chosen web-based platform, we could still consider alternative substrates. One possibility is inheriting the DOM as the state model. This is the choice made by Webstrates (Klokmose et al. 2015), which stores textual JS code for programmatic change. Following our approach, we might want a lower-level and structured instruction set instead. This would, at the very least, need to be capable of changing parent/child/sibling relationships, node attributes, and inner textual content. One warning is that the rest of the DOM API that would need to be exposed, in order to be able to produce a functional modern web page or web app, is somewhat daunting in scope. It would also be necessary to have some way of listening for changes to DOM nodes so that any constraints could be maintained and dependencies could be updated. Webstrates does provide synchronisation between networked clients on the same page, so perhaps its methods could be adapted.

7.3 REVIEW

We began in Chapter 1 with a vision of open software that can be adapted by its users without expending disproportionate work on accidental complexity. We introduced the key concept of a *programming system* along with Three Properties that would contribute to this vision: Self-Sustainability, Notational Freedom, and Explicit Structure. Our thesis was that it is possible to build a programming system with these properties on top of one's chosen starting platform (in our case, the Web browser), with an emphasis on the prize of *Notational Freedom* being facilitated by the other two Properties.

We then went on to establish the terms and concepts in which we would frame our work fulfilling this claim. In Chapter 2 we defined programming systems as a generalisation of languages, giving examples

of the diverse types of systems we are interested in. We then showed how the Three Properties have precedent in existing patterns and concepts. These "precursor" properties are well-adapted to a certain set of assumptions about how programming works, but do not tell us much about how to achieve the Properties in interactive, graphical systems with Explicit Structure. We categorised different sets of assumptions about how programming works as "paradigms" in Chapter 3 and explained why the Batch-Mode assumptions, as inherited through Unix, make our goals more difficult. We also introduced ideas that would help us understand our task, such as the differences between low-level and minimally human-friendly state models and the basic structure of a self-sustainable system as *platform*, *substrate* and *product*.

With these important concepts understood, we presented our proof of the thesis statement in Chapter 4: an account of the design forces and decisions involved in creating *BootstrapLab*, whose development steps are sufficiently general to act as a template for alternative paths through the design space (we provide a summary in Appendix A). We then proposed *technical dimensions* in Chapter 5 as a means to verify the extent of our Three Properties, plus more generally other properties of programming systems, and evaluated BootstrapLab using this framework.

7.4 CONCLUSIONS

Our efforts taught us that the process of developing a self-sustainable system roughly mirrors the historical development of programming that shaped much of how we do things today. Technologies like the assembler and the compiler were born from a truly impoverished platform of flat memory, numerical instructions, printed output, and rows of switches. Self-sustainable systems like Unix were gradually raised up from this primordial world, yet it still has a tendency to show through and force human minds to wrestle with it.

This work can be interpreted as a sketch of how we might build similar infrastructure on the back of *modern* computing environments with explicitly structured representation of data and graphical interfaces. In other words, we have opened an investigation into what programming could look like if it were *re*-bootstrapped today, not on top of flat memory, but on a richer base platform—such as, in our case, the web browser.

Of course, we have focused so much on Self-Sustainability because we see it as essential infrastructure for supporting our terminal goal of *Notational Freedom,* and it was the most demanding of the Three Properties to achieve. In retrospect, Explicit Structure was fairly easy to maintain by committing to it from the very beginning; the only discipline needed was to resist the occasional temptation to use text for core infrastructure. We would have further developed Notational Freedom if our roadmap

to Self-Sustainability had already existed; instead, we were satisfied once we established a minimal proof-of-concept that it was now possible (Section 4.9.3). Because Self-Sustainability was such an unclear concept in our domain of interactive graphical systems, it deserved to be made legible via the slow, reflective process we documented.

In his 1997 OOPSLA keynote "The Computer Revolution Hasn't Happened Yet" (Kay 2000), Alan Kay hoped that future users of Squeak would use it to start a virtuous cycle of innovation: "Think of how you can obsolete the damn thing by using its own mechanisms for getting the next version of itself." Twenty-six years later, the self-sustainability pioneered by Smalltalk remains as elusive as ever outside of its communities. Our hope is that through our contribution here, we have increased the range of its potential beneficiaries. We wish to empower programmers to add self-sustainability to their *own* preferred systems by following the steps we developed. In addition to the other benefits of such a property, this unlocks the full promise of "Use The Right Tool For The Job" in the form of Notational Freedom, advancing us that much further in the struggle against accidental complexity.

A

BOOTSTRAPLAB JOURNEY SUMMARY

Here, we provide a succinct reference for the major journey steps in Chapter 4, along with the requirements, forces and heuristics that we encountered in our experience.

CHOOSE A STARTING PLATFORM.   Select an existing programming system on which to build.

DESIGN A SUBSTRATE.   Architect the basic form of *state* and *change* in the system, as implemented in the platform.

- Requirement 1: *Code As Data.* The smallest units of change (instructions) must be readable and writeable as ordinary data.

- Force 1: *Avoid Boilerplate.* Prevent the *Turing Tarpit* failure mode by making the substrate more advanced.

- Force 2: *Escape The Platform.* Work in-system as soon as possible by tolerating a *less* advanced substrate.

- Heuristic 1: *Minimally Human-Friendly Low Level.* Ensure that the substrate supports *string names* and *substructures* (Force 1 > Force 2).

- Force 3: *Alignment.* Everything should *fit:* instructions, high-level expressions, and graphics expressions should all fit the substrate, and the substrate should fit the platform.

- Heuristic 2: *Use Imperative Assembler.* The fetch-execute cycle is the easiest "language processor" to implement, so begin with that (Force 2 > Force 1).

- Heuristic 3: *Simple Assembler.* Design a RISC rather than CISC instruction set (Force 2 > Force 1). *Warning:* our extreme interpretation was counterproductive (Section C.1).

- Heuristic 4: *In-State Graphics.* Make graphical interfaces expressible as ordinary state in a special location (Forces 1 and 3).

IMPLEMENT TEMPORARY INFRASTRUCTURE.   Use the platform to implement tools for working within the substrate, most importantly a state viewer or editor; these will be discarded once in-system versions are available.

Heuristic 5: *Platform Editor.* As soon as possible, use the platform to implement a temporary state viewer and/or editor (Force 1 > Force 2).

IMPLEMENT A HIGH-LEVEL LANGUAGE.   The substrate's instruction set is cumbersome, so ensure programs can be expressed in-system via high-level constructs.

Heuristic 6: *In-State Operation.* Store high-level-language processing *state* in-system, even if the language processor *code* remains running on the platform (Force 2).

PAY OFF OUTSTANDING SUBSTRATE DEBT.   Port all remaining temporary infrastructure into the system to arrive at a *self-sustainable* system.

PROVIDE FOR DOMAIN-SPECIFIC NOTATIONS.   Use the self-sustaining state editor to construct a more convenient interface for editing high-level expressions. Add novel notations and interfaces as needed.

BOOTSTRAPLAB SUBSTRATE REFERENCE

This chapter serves as a more complete description of the BootstrapLab substrate than the one given in Section 4.5. Following Section 3.1, we divide it into a discussion of *State* and *Change*.

B.1 STATE IN BOOTSTRAPLAB

State is a graph of *maps* (key-value dictionaries) pointing to each other, along with JS *primitives:* numbers, booleans, strings, `null`, `undefined`, and ordinary JS objects.

Maps contain *entries* or *fields* consisting of a *key* and a *value*. We also refer to maps and primitives together as *values*. Starting from a given map `m`, we write a *path* `m.foo.bar.baz` to denote the value arrived at by following the keys `foo`, `bar`, and `baz`. For this to be defined, `m.foo` and `m.foo.bar` must be maps. A *list* is just a map with numerical keys, but as in JS, it may have additional non-numerical keys if convenient.[1]

The substrate inherits the reference semantics of JS; if one wants to insulate a map from side effects originating from other references, one must perform a copy to some level of depth and operate on the copy. Other JS semantics, such as prototype chains, were considered but postponed for the sake of practicality; see Section C.2 for an outline of what was left out of the substrate.

There is a *root* for the state, from which everything is discoverable; all absolute paths begin here, and every value that "exists" necessarily has some path from root, although it may have more than one owing to the graph structure. The root is not addressable from user code, but in JS it is a global called `ctx`. An "address" consists of a map and a key; analogously to paths, a piece of state may have more than one "address" if it is referenced from multiple maps.

B.1.1 *Registers*

Because values at the root level are accessible from only a single key, they are the "first port of call" for instructions and are known as *registers*. Some specific *substrate* registers are reserved for use by instructions. The main substrate registers are:

---

1 In other words, the "list" concept is contained within the "map" concept, but where the system expects a list (e. g. when fetching instructions) it will ignore any extra user-defined keys.

- `focus`: an "accumulator" register used by most instructions.

- `map`: holds the map to be updated or navigated by map instructions.

- `source`: holds the new value when updating a map entry.

- `next_instruction`: program counter; holds the address of the next instruction and the instruction itself.

- `scene`: container for the scene tree affecting the graphics window (discussed shortly).

- `addend`, `factor`, `basis`: operands for mathematical instructions.

All other keys at the root level are general-purpose *user registers* available for programs to use.

B.1.2 *Graphics State: The `scene` Tree*


```
-yellow_shape:
  color: 0x999900
  width: 2
  height: 2
 -center:
  basis: shapes
  right: 2
  up: 1.750
  forward: -1
```

Figure B.1: A rectangle inferred from the presence of `color`, `width`, `height`, and `center`. As can be seen, `color` expects a hex string.

While state in general can be graph-structured, it is tree-structured[2] under the `scene` register. The `scene` itself is a list of *graphics maps*. The substrate recognises a map as a graphics map if it contains at least one *graphics property:* any of the keys `width`, `height`, `color`[3], `opacity`, `position`, `center`, `top_left`, `zoom`, `text`, or `children`. The presence of these keys causes the substrate to connect their values to shapes in the graphics window and maintain synchronisation. Any graphics map can have a list of `children` in the scene hierarchy.

At present, only crude graphics are possible via two shapes. A rectangle has a `center`, `color`, `width`, and `height` (Figure B.1), while a text label has a `top_left`, `text`, and `opacity` (Figure B.2).

More interesting is the protocol of vector properties like `center` and `top_left`. Firstly, the naming of these properties themselves follows a design principle which could be called "Naïve Honesty".[4]

---

2 See Section C.3 for how this was accomplished.

3 We comply with the fact that, for better or worse, American spelling conventions are the *de facto* standard for internal identifiers and code more generally.

4 By analogy to Boxer's "Naïve Realism" (diSessa and Abelson 1986).

```
                        —mytb:
  Hello World             text: Hello World
                        —top_left:
                          right: 1.500
                          up: 0.5000
```
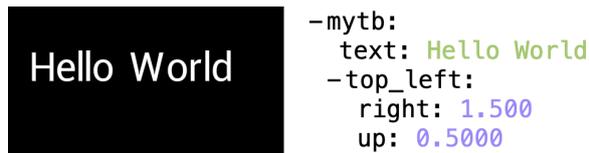
Figure B.2: A text label inferred from the presence of `text` and `top_left`. The `opacity` property, not shown, takes a number between 0 and 1.

```
          —camera:
            zoom: 0.4403
           —position:
             basis: world
             right: 1.451
             up: −0.5981
             forward: 10
```

Figure B.3: The camera 'zoom' and 'position' are bidirectionally synchronised between the state and the graphics window.

Heuristic 7 (Naïve Honesty)**.** If a more abstract term has only one intended meaning in a particular API, use the honest concrete concept instead.

For example, the term "position", widely used in graphics APIs, is not very self-documenting when it comes to shapes. In practice, it always means "the centre of the shape" or "the top left-hand corner" or some such meaningful function of the shape itself. Therefore, why not just be explicit about this, and save the user the effort of figuring out what it actually means? The co-ordinates are specified according to the same principle: instead of the ever-ambiguous x, y, and z, we have `right`, `up`, and `forward` (that is, into the screen from your position). In an ideal substrate, one could specify co-ordinates via `left`, `down`, `backward`, or any combination, and the system would automatically flip the signs as necessary for its platform graphics API calls. However, this functionality does not yet exist in BootstrapLab.

Third, a vector property may have a `basis` key naming a registered co-ordinate frame; built-in examples are `world` (the root THREE.js basis) and `screen` (pixel co-ordinates from the top left). This key can be seen in the special *camera* graphics map (Figure B.3), linked to the zoomable/pannable view in the graphics window. If left unspecified, the basis is assumed to be that of the parent node in the tree. However, the user can set an explicit `basis` to express co-ordinates as most convenient to them, while the substrate will convert between frames under the hood.

This is another application of Naïve Honesty in regard to our frustrations about graphics programming: the co-ordinate basis of a vector is often left as *implicit* information for the programmer to carry around in his head, who must also have a notebook handy to write the relevant

matrix equations to transform things properly. We believe this tedium is exactly what should be handled automatically by the substrate (on this theme, see Geisler et al. (2020)). As it stands, basis frames are registered by the key name of the graphics map into a flat list, and are thus vulnerable to name collisions and synchronisation issues. However, what we have is a promising start.

Naïve Honesty can be seen as a response to a design requirement called "No Guessing":

Requirement 2 (No Guessing). The user of an API should never have to uncover the meaning of a concept through trial-and-error experimentation. In graphics programming, the user should never have to create a throwaway object and transform it to resolve ambiguity about sign conventions, axis conventions, unit conventions, basis conventions, etc.

A reasonable reply to No Guessing might be to just write better documentation. However, this is still not a solution to implicit basis frames in user code, and Naïve Honesty attacks the root of the problem in the poor conventions themselves.

B.1.3 *Manually Updating State*

In order to update a piece of state *and ensure* that all relevant UI also updates to reflect this, the upd() function is used in JS code and the console. For example, to change the colour of the shape in Figure B.1 to red, one would issue the following command in the console:

```
upd(ctx, 'scene', 'shapes', 'children',
      'yellow_shape', 'color', '0xff0000')
```

In order to obtain a JS reference to a value, one gives a path to map_get():

```
v = map_get(m, 'children', '1', 'color')
upd(ctx, 'my_value', v)
```

B.1.4 *Persisting State*

Calling the function export_state(map, filename) in the console will walk the state graph from the given map and download it as a JSON file with the filename; if unspecified, it will default to bl-state.json. Calling import_state(filename) in the console or the code will yield a JS Promise that will resolve to the same map, which can then be slotted in via upd(). For example, the following is how we load the Masp code in Section 4.9.3 during initialisation:

```
import_state('misc/render.json').then(x => {
  upd(ctx, 'render_map_entry', x);
});
```

In keeping with Alignment (Force 3), the smallest units of change ought to just "fall out" of the structure of the State:

- Change a map entry to a new value (our `store` instruction)

- Create a new map (our `load` instruction)

- Actions necessary to support the above (`index` and `deref`)

- Inheritance of platform changes, i. e. mutate JS state and call any API (the `js` escape hatch)

The result is our instruction set, which we call BL-ASM. We will elaborate on these shortly, but first we will specify how instructions are represented as state, and how we will notate them in shorthand and diagrams. We will mostly stick to reference material here; for more in-depth design rationale, see Section C.1.

B.2.1   *Instruction Encoding in State, Text, and Diagrams*

An instruction is a map with an `op` field serving as the opcode, along with any parameters as further entries. Some examples:

```
{ op: load, value: my_reg },
{ op: deref },
{ op: store, register: my_dest }
```

Because this notation is so verbose, and no instruction has more than one parameter (see Section C.1), we use an inline textual notation with unnamed parameters. We would write the above like so:

```
load my_reg ; deref ; store my_dest
```

In circumstances where there are many instructions and we need to be even more concise, we only use the first letters of the names. The above example would reduce to:

```
l my_reg ; d ; s my_dest
```

In the next section, we will specify the semantics of the instructions. In line with the spirit of *Notational Freedom* (Section 3.3.2), we will use box-and-arrow diagrams, since we judge this more suitable for our substrate than traditional formal semantics notations. However, we will take some cues from the latter; for example, we show the state before and after the instruction. We also use symbols to stand for abstract values. Specifically, *K* denotes any string (or "key"), *M* denotes a map, and *V* denotes an arbitrary value. We additionally employ a grey spot to highlight the part of the state that was mutated. Figure B.4 shows the general form.
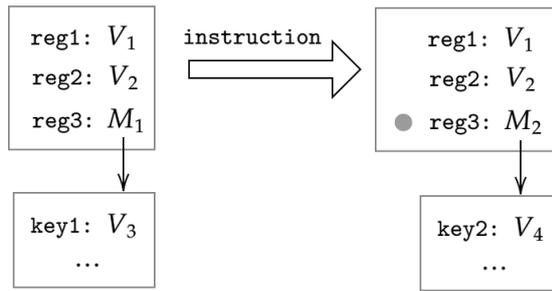
Figure B.4: On the left, we have registers with their values, one of which references a map M_1 with some entries. On the right, we have the state after execution of the instruction; the register reg3 has been changed (highlighted by the grey spot) to reference a new map M_2. Because instructions were designed to be "minimal" changes, the grey spot represents the only change and all other state remains untouched.

B.2.2   *Change Map Entry and Supporting Instructions*

We will now proceed to explain the semantics of store, index, deref, and load.

B.2.2.1   *Parameterless Store: Change Map Entry*

The store instruction, with no parameters, expects a map M in the map register, a key string K in the focus register, and a value V in the source register. After execution, the entry K of M will have the value V.
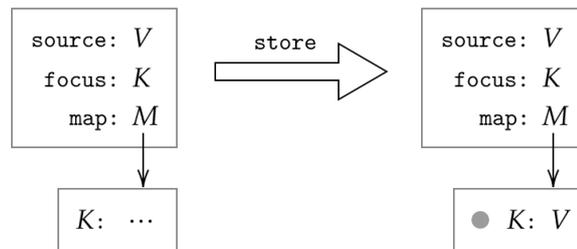
B.2.2.2   *Index: Follow Key in Map*

The index instruction, like the store to which it is dual[5], takes a map *M* in map and a key string *K* in focus. After execution, map contains the value *M.K*, unless this is undefined. In that case, it will try the special key _ as a failsafe and map will contain *M._* instead, which could still be undefined.
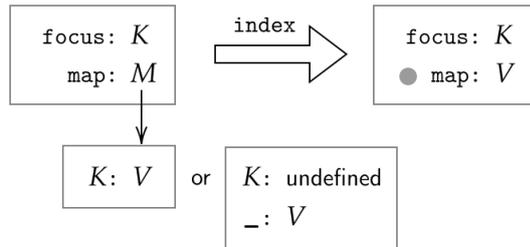
B.2.2.3   *Store To Register: Change Root Entry*

There are alternate semantics[6] when store is executed with a parameter register of string value *K*. In this case, given a value *V* in focus, the root-level entry *K* will have value *V*.

B.2.2.4   *Deref: Follow Key in Root*

With a string *K* in focus and register *K* holding the value *V*, an execution of deref will place *V* in focus.

---
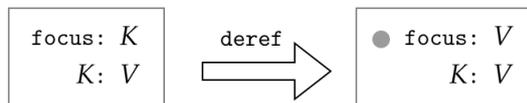
5 We mean this in an informal sense, but it points to some interesting analysis which we have not undertaken. Compare also deref and the register version of store, leaving load curiously on its own.

6 This "overloading" of an instruction is straightforward in a map substrate, as compared to a flat binary one (Section 3.1.3), although an argument could be made for it to be a separate instruction called store-reg.

B.2.2.5   *Load: Instantiate a Literal*

Finally, `load` takes a parameter `value` with a value V. After execution, the `focus` register contains V.
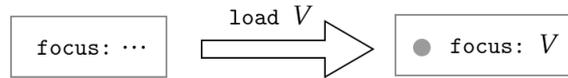
Actually, there is a subtlety if V is a map: the `focus` register then contains a *copy* of V. This is to preserve the intended use of the `value` parameter as a "literal"[7], especially when the map is empty (see Section B.2.3 shortly).
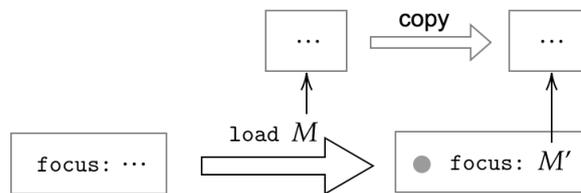
At the time of writing, the copy performed is a deep copy, but this is almost certainly wrong, as would be always using a shallow copy. The purpose of having a `load` instruction with a map literal in the code is to *instantiate* a new map with that structure, so the literal in the code must remain unaffected. The basic intuition (Figure B.5) is that we want things that *present as* part of the literal in the code to be fully (deep) copied; in other words, an inner map that is fully "nested" (and not referred to elsewhere) should be duplicated. Meanwhile, any references to data structures that already exist elsewhere (which thus should not present as "fully nested" in the map literal) should be preserved. However, the substrate does not yet distinguish between these two cases.[8]

B.2.3   *Create New Map*

Because of the fact that `load` makes a copy of its `value`, creating a new map is simple: `load` M, where M is an empty map. After execution, `focus` contains a new empty map that can be written to, *without* affecting the empty map in the instruction itself, which will stay empty (unless, of course, the instruction is deliberately modified "as data" by separate

---

7 We use "literal" by analogy to "string literal", "number literal", etc. In other words, an entity presented in its entirety in the source code, rather than loaded from an external source at run time or built up from separate pieces.

8 It could be argued that the generator of the code (whether user or compiler) knows best, hence `load` should just take an optional flag for whichever case is not the default.
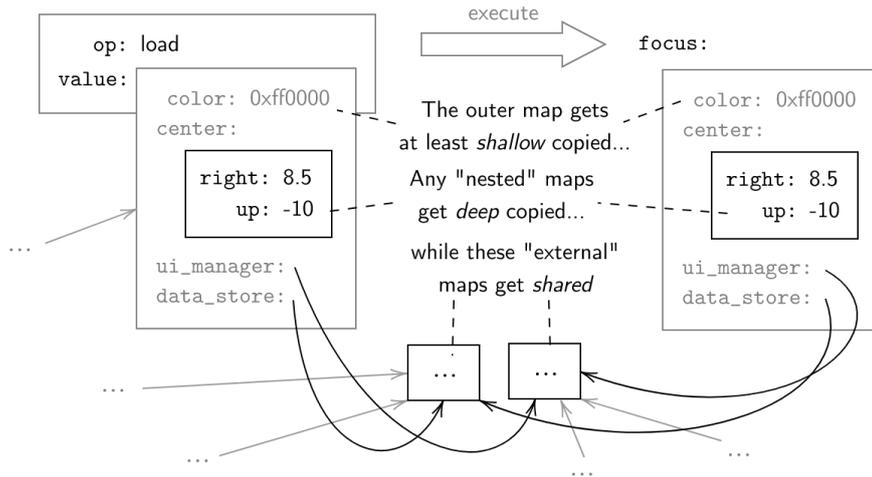
Figure B.5: On the left, we have a `load` instruction for a "literal" map, meant to be used as-is. Its effect is to fill the `focus` register with *some sort of* copy of the `value`. We would present "child" data (i.e. with no other parents/referrers) as graphically nested within its parent, and these maps should get duplicated. However, data that is clearly meant to be shared, as evidenced by other existing references, ought to remain that way.

code). If `load` did not perform a copy, it would be necessary to use a *different* `load` instruction each time one wanted to create a new map.[9] However, arguments could be made to the effect that `load` should just have special semantics for the empty map, or that there should be a `load-new` instruction, and so on.

B.2.4 *Inheritance of JS-level Change*

The `js` instruction takes a `func` parameter and calls it as a JS function. This functions as an all-purpose "escape hatch" into the JS platform, analogous to `asm` blocks in C code.

```
{ op: js, func: () => {
    alert("Hello, World!");
}}
```

Some features of the platform were common enough to be worth implementing as their own instructions, as an optimisation:

- `add` adds the `addend` register to `focus`.

- `mul` multiplies `focus` by the `factor` register.

---

9 This situation is the same Python's notorious "mutable default arguments" for functions (Python Guide 2016). Default arguments are evaluated the one time a function gets *defined*, instead of per call, so subsequent calls to the function will see any successive changes to the same default argument object.

- `sign` replaces `focus` with its mathematical sign ($+1$, 0, or -1).

- `typeof` replaces `focus` with its result under the JS `typeof` operator.

- `basis` rewrites the vector in `focus` to be the same vector, but in the basis named by the `basis` register.

However, these remain experimental rather than practical and we did not get around to implementing an operand stack.

We could argue in favour of such instructions in terms of Alignment (Force 3). Supposing our basic state-prodding instructions were Turing-complete, we could certainly implement arithmetic, etc. using Church numerals encoded as maps, but this would be a waste of the vastly more efficient capabilities provided by the platform. Similarly, in a real-world instruction set (e. g. x86), any special-purpose silicon in the *hardware* platform ought to be accessible from some instruction; we view arithmetic and logic instructions as *access points* to the heavily optimised ALU, as opposed to "intrinsic" requirements of the instruction set.

Tentatively, we conjecture that instructions are either *intrinsic*, i. e. necessary for Turing-completeness, or *gateways* to platform optimisations. Any instructions for accessing e. g. video memory could be considered intrinsic if we subsume video memory as just a special part of the overall state. Furthermore, any special instructions that affect the outside world (e. g. refresh the screen, or send network packets) could be re-interpreted as writing to special parts of state that have side effects; see also Self's behaviour-as-state (Ungar and Smith 2007) or Plan 9's "control files" (Pike et al. 1993).

B.2.5   *The Fetch-Execute Cycle*

The `next_instruction` register holds a map with keys `ref` and `value`. The `value` entry holds the next instruction itself, while `ref` contains entries `map` and `key` as the address of the instruction, where `key` is an integer. During the fetch-execute cycle, `key` is incremented. In this way, a list of instructions can serve as a "basic block". Furthermore, if incrementing the key "runs off the end" of a list, the substrate will follow any `continue_to` entry in the list and continue execution at key 1 of the associated map.

The fetch-execute cycle is activated via the JS console function called `run_and_render(n)`, taking a number n of instructions to execute (warning: this takes place on the main JS thread, so if there are many instructions, the UI will stall). If n is omitted, it defaults to 1, effectively acting as a single-step command. Finally, any instruction can have a `break` parameter; if set to a value JS considers "truthy", the fetch-execute cycle will halt after executing the instruction.

B.2.6   *Input Handling*

Currently, input handling is entirely a substrate capability, as opposed to an in-system one. While we have successfully experimented with running in-system code as "device drivers" in response to input (Section 4.5.6), this is not how things work at present. As a leftover from these experiments, there is a register called `pointer` containing a boolean `is_dragging` and vector-valued `pressed_at` and `released_at`. The JS `mousedown`/`mouseup` event handlers ensure `is_dragging` reflects whether the left mouse button is currently down over the graphics window, and log co-ordinates in `pressed_at` and `released_at` as appropriate. However, there is not currently any functionality in-system to "listen" for changes on these entries. Furthermore, because `run_and_render()` blocks the JS thread until completion, not even a polling-based approach would work (since the events would queue up until after the thread unblocks).

These handlers, along with `mousemove` and `onwheel`, update the camera `position` and `zoom` properties to allow the user to zoom and pan around the graphics scene. The click handlers and `keydown` handler are full of code for selecting and editing entries in the Tree Editor (Section 4.8.3). When eventually ported to in-system code (whether Masp or BL-ASM), these event handlers will more resemble the "device driver" pattern that we mentioned.

BOOTSTRAPLAB TRIVIA

Appendix B was meant as a reference, but many aspects of the substrate deserve further elaboration in terms of design rationale or unrealised potential. We will discuss these matters here.

C.1 THE MINIMAL RANDOM-ACCESS INSTRUCTION SET (AND ITS PERILS)

In this section, we will explain the design rationale that led us to the peculiarities of the instruction set. Recall Heuristic 3 which instructed us to pursue an *easy-to-implement* instruction set. We pursued this goal to the extreme out of curiosity for what was possible. Of course, it turned out that the corresponding explosion in the number of instructions necessary to do a simple thing outweighed any implementation advantage...

We did this by breaking down higher-level instructions to their component operations until we felt we could go no further. This led to a sort of "microcode" level where each instruction's implementation corresponded to some single-line JS operation. In other words, the platform itself blocked any further decomposition.

C.1.1 *Deconstruction of a Path-to-Path Copy*

Our method for achieving this can be illustrated if we start with a hypothetical complex instruction, e.g. `copy a.b.c to x.y.z`. The actual *work* involved in executing this in JS would involve three steps:

1. Traverse the path `a`, `b`, `c` and save the value in a local variable

2. Traverse the path `x`, `y` and save the (map) value too

3. Set the key `z` in the map to the saved value.

If we score *strictly by JS implementation size* (a mistake, in hindsight), we could improve by simply splitting up these steps into instructions of their own. Any other "complex" instructions that used some of the same steps (e.g. path traversal) will also be covered by these, and the total JS would be reduced.

For the first path traversal, we start at the root map (or more generally, any given starting map) and follow each of the keys in turn. We have only one step here (follow key) repeated three times. That's another micro-instruction!

At this stage, we have this very simple instruction: `follow-key k`. It clearly relies on some implicit state register for the current map, and takes a single parameter. We pushed the limits of sanity by going further, factoring the parameter *out* into another state register, so the resulting instruction is just `follow-key` (we called it `index`). In other words, we applied the following heuristic:

Heuristic 8 (Registers for parameters). Factor out instruction "parameters" into special state registers where possible.

The motivation for this is a vague intuition about sharing parameter values. Under a parameter scheme, copying the same thing to multiple destinations will duplicate the "source" parameter many times, even though the only thing that's changing is the destination. The converse is true for operations with the same destination—maybe not overwriting copies, but arithmetic or other accumulating operations. By breaking these parameters into state, we set a source or destination once only. This has a subjective aesthetic appeal from the point of view of minimality, and an even more dubious efficiency value. We emphasise that it was an experiment and advise against it for the purposes of implementing a system quickly.

C.1.2   *Copying and Jumping*

Combinations of the instructions express the expected copying and jumping operations. For example, `load source-reg`; `deref`; `store dest-reg` copies the value in root-level `source-reg` to `dest-reg`. The first instruction loads the literal string `source-reg` into the `focus`; the second replaces `focus` with the contents of its named register; the third copies the `focus` to the named destination.[1]

The `copy a.b.c to x.y.z` from earlier would decompose as follows (recall the ultra-concise notation from Section B.2.1):

1. `l a; d; s map; l b; i; l c; i; l map; d; s source`

2. `l x; d; s map; l y; i`

3. `l z; s`

(Recall that `index` replaces `map` with the result of following its key named by `focus`, and `store` without any arguments copies from `source` to the `focus` key entry within `map`.)

A jump is accomplished by overwriting the address in the `ref` field of `next_instruction`. The `map` or the `key` of this address can be overwritten

---

[1] It turns out that, if you extract the destination parameter from `store`, you meet an infinite regress and will be unable to store to any top-level register. For example, if we extract the parameter to `dest-reg`, we have to somehow give it the value it previously took in the instruction—but this is precisely a `store` operation and we're already in the middle of one. However, see Section C.1.3 for a way around this.

in a single instruction, but if an entirely new address is required, this needs to be built up separately and overwritten atomically. In other words, we cannot overwrite the map and then overwrite the key. The ugly reality is, after overwriting the map, it will have jumped to a different instruction somewhere else!

A *conditional* jump is sneaked in by *indexing* into a map to obtain the new list of instructions (which is the map that will overwrite the map under `next_instruction.ref`). For example, in the following register snapshot:

```
...
weather: stormy
map: {
    sunny: { ... sunny code sequence ... }
    rainy: { ... rainy code sequence ... }
    _:     { ... other code sequence ... }
}
```

One of the three code paths will be selected according to whatever happens to be in the `weather` register via the following instructions: `load` weather; `deref`; `store` focus; `index`. The map register will hold the result, in this case the "other" code sequence (recall that the special key _ is used as an "else" clause for lookups). What remains is then to copy this within `next_instruction.ref`.

This example proved that we can do strict equality matching, but what about comparisons? We may observe how conditional jumps are traditionally based on a relation to zero, e. g. "jump if less than zero". A condition like $3 < 7$ is rearranged into $3 - 7 < 0$ and the "jump if less than zero" instruction is called with the value -4. In BootstrapLab, we go one step further: if we apply the mathematical `sign` instruction to the value -4, we get -1. We can then simply use the trick we already described and `index` into a map containing keys `-1`, `0` and `1`. All of the relevant conditions can be expressed this way: for example, "greater than or equal to zero" can be achieved by pointing the keys `0` and `1` to the same code sequence.

c.1.3   *A Meta-Circular Inelegance*

From the perspective of self-sustainable systems, it is worth checking that there are no hard barriers to implementing a BL-ASM interpreter in itself. We noticed such a problem with the register `store` instruction.

Firstly, let us call our existing BL-ASM the "base" level, where the hypothetical interpreter code lives. Next, we call the code being interpreted the "virtual" level. Suppose that the virtual register file is a map under the base-level user register `virtual` and the current virtual instruction lives in `instr` (Figure C.1). In this case, upon seeing `op: store` the following base-level code will do the trick:
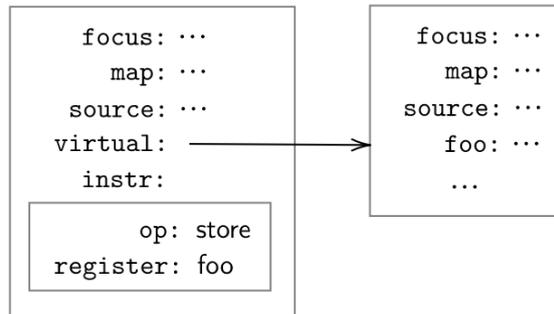
Figure C.1: Separate base and virtual registers.

```
l virtual; d; s map; l focus; i; l map; d; s source
// now, source = virtual.focus
l instr; d; s map; l register; i; l map; d; s tmp
// now, tmp = instr.register ("foo")
l virtual; d; s map; l tmp; d; s
// now, virtual[instr.register] = virtual.focus
```
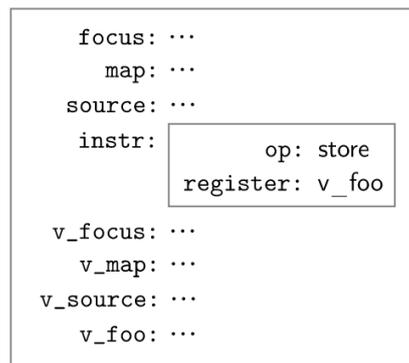

Figure C.2: Sibling base and virtual registers.

So far, so good; however, the same is *not* possible if we choose the virtual registers to live *alongside* the base registers (whether or not this is a good idea); call this the "virtual sibling registers" model. We are now considering the picture in Figure C.2 instead (assume for the sake of simplicity that virtual code plays nice and only names virtual registers). This time, we get stuck:

```
l instr; d; s map; l register; i; l map; d; s tmp
// now, tmp = instr.register ("v_foo")
// goal: copy value of v_focus to register named by tmp
???
```

The obvious way this ought to work, via the `store` instruction, has the wrong "addressing mode": it takes a literal string from the instruction, instead of indirecting through something to get a dynamically determined destination, which is what we need. The only way we can achieve this is through self-modifying code, by copying `instr.register` to the `register` field of a new `store` instruction and performing the relevant jumps before and after. This is very inelegant.

The essential JS logic that implements these semantics is as follows:

```
inst = ... // The current base−level instruction
root[inst.register] = root.focus;
```

It is natural to ask what instruction, if any, could perform this operation in-system. Under *Registers For Parameters* (Heuristic 8), the `inst.register` value, which is being used "indirect", should be factored out into a new register. We refuse to call it the `register` register, but `destination` could work. Since we will be left with a parameterless `store`, we should rename it to `store-reg` to avoid conflict with the map-related `store`.
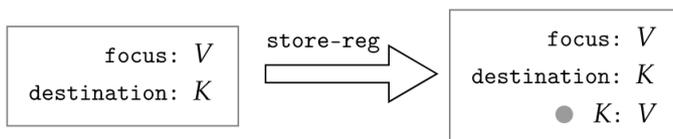
So far, where previously we had `store reg1`, we now have `load reg1` followed by some way of storing `focus` to `destination`. We add a `dest` instruction that does exactly this.
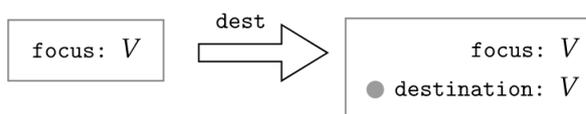
Now, we can translate the old `load myValue; store reg1` to `load reg1; dest; load myValue; store-reg`. The inelegance that caused our re-design can be codified as a new heuristic:

Heuristic 9 (Interpreter Alignment)**.** Ensure the platform interpreter can be translated into the substrate language.

Under the new instruction set, we *can* implement virtual `dest` and `store-reg` (`sr`) under the "virtual sibling registers" model. For `dest`, we wish to fill `v_destination` with the value in `v_focus`:

```
l v_destination; dest; l v_focus; d; sr
```

For `store-reg`, we wish to copy the value in `v_focus` to the register *named* by `v_destination`:

```
l v_destination; d; dest; l v_focus; d; sr
```

Notice that this is identical to the implementation of dest except for the insertion of a deref after the first load.

We can summarise the modified architecture, at the register level, as having a "read head" focus and a "write head" destination. The only way a value in the state gets to the register file is in through focus and out through destination, coupled by the dest instruction. Beyond the register level, we could analyse map as a combined read-write head. However, we will leave further analysis for subsequent work. Interested readers may consult Jakubovic (2021) for some further intuitions and design philosophy.

C.2   THE CUTTING ROOM FLOOR

Escape The Platform (Force 2) directed us to do without several advanced substrate features we were tempted to include. For example, it would be useful to attach state change listeners to keep parts of the state in sync with others. We could go even further and include constraint-based programming features.

On another note, our substrate is based on "maps" without a predefined ordering of the entries. However, there is always some order in which they will be displayed:

```
{ red: 100, green: 255, blue: 0 }
```

Thus it might be nice to be able to set this on a per-map basis. A convenient way to expose this in-system would be via another map, or "order map" which would be a list map of key names:

```
{ 1: red, 2: green, 3: blue }
```

A practical use of this is for enabling iteration through a map's keys or entries. If we wish to be rigorous, the order map itself would have an order map, which would (by default) be the same for all order maps:

```
{ 1: 1, 2: 2, 3: 3 }
```

Of course, with such a conceptually infinite sequence of order maps, care must be taken to implement it in a finite, on-demand way. Perhaps some clever circular reference would work, as COLA does for its *vtable* relation (Piumarta and Warth 2008). This raises the question of how to obtain an order map in-system. If we make it an ordinary key on all maps, we must be careful to render it only on-demand and to exclude it from ordinary iteration through keys. Plus, would we want the visual clutter of always displaying it? It might be better to make it accessible through a special instruction order-map.

We then face a further *synchronisation* problem, where we must alter the display order whenever the order map is changed, and insert or remove entries from the order map to match its source.

Other thoughts along these lines included *parent* maps for delegating lookups (similar to JS's prototype system), *inverse* maps, and *meta* maps for possibly collecting all of these (drawing inspiration from Lua's metatables). Of these, we will only discuss inverse maps in more detail.

Inverse maps come from the view of a map as a mathematical function from key names to values. Often in advanced data structures (such as those for graphical diagrams) it is essential to know "who points to me" via some key. For example, the question "Is this node the `source` of anyone else?" is a natural one, but normally it is impossible to answer based on ordinary dictionary keys. In ordinary programming languages, this information needs to be kept track of separately; say, in a manually synchronised list called `sources` that lives on the node. It is frustrating that the "forward" question is trivially answered by just following a map entry, yet the "backward" question has to be hacked around like this.[2]

An inverse map would somehow collect all references to a map from other ones. A user-level "map" would be implemented by two dictionary structures, the forward and backward halves, which are automatically kept in sync by the substrate implementation. The previously mentioned issues of access, mutation and others also rear their ugly heads here, so we can be forgiven for discarding the idea for the sake of making progress. Still, a properly worked out implementation would provide a valuable service for a high-level substrate.

C.3   GRAPHS VS. TREES

A classic debate in the world of explicit structure is whether to use restricted *tree* structures or to allow arbitrary *graphs*. A tree has the advantage that every node has a single parent, which is a useful canonical answer to the question "what context am I in?". On the other hand, many practical problems do not fit inside a tree structure; either because they are DAGs, and a node can have multiple parents, or because they involve cyclic relations. Because we did not know what sort of things we would require in BootstrapLab, we erred on the side of freedom and supported full graph relations. This bit back at us in two ways, both involving the graphics domain.

Firstly, cyclic structures need to be rendered with care; a naïve depth-first search will never terminate. For a long time, we did not have any cyclic structures and got away with a depth-first approach to DOM generation in the temporary state view (Section 4.6.2).

---

2 Norvig's "Relation" pattern (Norvig 1996) for dynamic languages is relevant to this sort of concern.

Secondly, while this was the case, the graphics sub-region of state needed to be a tree. Spatial containment and other visual nesting (e.g. for the tree editor) is a tree structure, as is the underlying parent-child relationship of THREE.js objects. Many aspects of rendering the tree editor required the ability to ask "what context am I in?" but this is unanswered by default in a graph substrate. Providing a "parent" key for each node would not do—this would be a cyclic reference. Instead, we kludged it: the first map to reference another map becomes its "parent", and this lasts until the reference is deleted. This parent property is available from JS; as we port the tree editor to Masp, we will have to decide how to expose it in-system (probably through a special instruction).

Of course, we eventually did require cyclic structures—for the tree editor! Each graphics node in the editor has a `source` key providing a way for edits to propagate back to the source state node. All edit nodes live in the graphics tree, including the one corresponding to the root node of the state. In this case, the `source` points all the way upwards to this root node. This cycle broke our state view and there was much gnashing of teeth to hack around this. Eventually, we bit the bullet and improved the state view JS to cope with cycles—having previously hoped we were done with this temporary infrastructure.

Let this be a warning that Alignment (Force 3) will come for you in the end. If your substrate allows cycles, your state view must tolerate them!

MASP REFERENCE

The objective of *Masp* is to get all the benefits of a Lisp-like expression language in a way that is Aligned (Force 3) with our map-based substrate (see Appendix B). We want to preserve the dynamic, functional, and metaprogramming facilities of Lisp, while leaving behind the reliance on positional parameters and making full use of map affordances.

Lisp is built around the "linked list" data structure as implemented by "cons" cells. Expressions to be evaluated take the form of lists. However, lists can also represent "data" lists that are not meant to be evaluated. Similarly, Masp includes both "data" maps and maps representing an expression to evaluate. Accordingly, we will divide our description of Masp into *State* and *Change*.

D.1 STATE IN MASP

State is inherited from the substrate (see Section B.1); in particular, lists continue to exist as maps with numerical keys. However, there is one important Masp structure that needs encoding *within* this substrate: the variable binding environments (or *envs* for short). An env is basically a tree of dictionaries, where children delegate failed lookups to their parents. In Lisp this is traditionally encoded as a tree of linked list cells, where lookups are accomplished by linear search. In Masp, the map substrate already provides the dictionary structure for free, with lookup as a primitive operation (no linear scans!) and all that must be done is to add extra structure for delegation.

Delegation could be as simple as adding a `parent` key to a map, but this "in-band" encoding would essentially reserve the word `parent` and preclude it from "user"-level Masp code. We could call it `__parent` or something equally strange and perhaps get away with it, but for elegance's sake we took the alternative route. An *env* is a map with two entries, called `parent` and `entries`. The latter holds the local entries (which can have whatever keys we want), while `parent` points to the parent env if it has one.

D.2 CHANGE IN MASP

The fundamental distinction a Lisp/Masp interpreter needs to make is between expressions that *need* evaluating, and those that do not. If Lisp is asked to evaluate a list, it will assume the first element is a "function

expression" and evaluate it to obtain a function.[1] It will then *apply* this function to the rest of the list as arguments, evaluating them or leaving them alone as the function specifies.

In Masp, a function application is *explicitly* marked by the presence of an `apply` key. When Masp evaluates a map, it looks for this key and treats its value as a function expression. The rest of the keys in the map are its arguments. True, this *in-band* encoding means that no function may have a parameter called `apply`, but we made this decision for simplicity and readability of Masp code, given the primitive notations available to us at the time. Consider the following illustrative function application:

```
apply: drawLine, from: { x: 10, y: 20 }, to: { x: 50, y: 60 },
                color: { apply: quote, to: 0xff0000 }
```

The evaluator looks at the `apply` key, sees it is a JS string, and looks up "drawLine" in the current env to get a closure. Suppose the closure lacks a `dont_eval_args` key; the evaluator goes through each remaining entry[2] and evaluates it. The `from` and `to` values are both data maps lacking an `apply` key, so they remain as they are. The `color` argument is a further application[3], which evaluates to the string `0xff0000` (simply putting the string itself would cause a variable lookup).

This illustrates that, as far as JS primitives are concerned, nearly all are treated as "terminal" values that need no evaluation. The sole exception is a string, which denotes a variable to be looked up in the current env.

By convention, single-argument functions like `quote` name the argument `to` in order to provide the reading "*apply* function *to* argument". Some functions, such as `add`, `mul`, `sub`, etc. number their arguments instead of using their official names:[4] `apply: add, 1: 10, 2: 20`.

As in Lisp, an individual function parameter can be "extensional" (evaluated before use) or "intensional" (used in its un-evaluated form). For example, the `if` function used as (`if cond then else`) has `cond` in extensional position; the behaviour of `if` depends entirely on whether `cond` evaluates to `true` or `false` and not on the make-up of the `cond` expression itself (Smith 1982, p. 360). Meanwhile, the `then` and `else` code are used *intensionally* because one of them should not be evaluated (otherwise, the "wrong" branch would still get to cause its side effects).

---

1 By Lisp, we abbreviate "the version(s) of Lisp that have the desirable features we wish to see in Masp"; any versions of Lisp that do not do what we mention are not germane to the discussion.

2 In a nondeterministic order—which is worth fixing in subsequent work. Perhaps Order Maps (Section C.2) have their place here.

3 The special meaning of the `apply` key means that a "data" map in argument position to such a function cannot have such a key; it would be seen as an application and evaluated. This would not be a problem, however, for a function with `dont_eval_args` set, provided the function specifically left the given argument unevaluated.

4 Despite our tolerance of verbosity with the promise of future Notational Freedom, we could not stomach having `addend`, `multiplicand`, `subtrahend`, etc. in Masp arithmetic.

For simplicity, functions in Masp are assumed all-extensional by default; this can be disabled by setting the `dont_eval_args` entry on the function closure, in which case it is up to the function itself to perform any evaluation.

A Masp *closure* is a map with entries for the function `body`, local `env`, a list of `arg_names`, and optionally a `dont_eval_args` flag. If the `body` is a JS function, the closure represents a primitive function run in the platform. Otherwise, the `body` is treated as a Masp expression to evaluate in the context of the local arguments.

Interestingly, `arg_names` is never actually used by anything, not even the evaluator. It is necessary *in Lisp,* as it links the positional arguments passed to the internal names used in the body; in Masp, arguments are already passed with names, so no such linking is necessary.[5]

D.2.1   *Maps as Functions*

We find it useful to view maps abstractly as "extensional[6] functions": mathematical functions taking inputs to outputs, whose mappings are explicitly listed instead of via some computation. Therefore it is important that maps must be usable as functions in Masp; we must be able to "apply" a map to a value. This gives us a special case of pattern matching for free (i.e. strict equality matching). It may be worth considering how augmenting the definition of maps-as-functions could support more advanced pattern matching (e.g. inequalities, more complex conditions) but this is beyond the scope of Masp as-is. We retain the failsafe match named _ from the substrate's `index` instruction (Section B.2.2.2).

For example, the pattern match in the Factorial definition (Figures 4.7 and 4.8) could be extended like so (using $...$ to denote syntactic sugar):

```
to: n, apply:
  0: 1
  1: 1
  2: 2
  3: 6
  _: $ n * fac(n-1) $
```

This situation can be read imperatively as "to n, apply the map", or indicatively as "pushing n through the map".

---

5 We kept `arg_names` to serve as an order map (see Section C.2) to enable positional parameters as a concise notation in the future. One addition suggested by the Lisp usage is an `arg_renames` to connect the preposition-like names used in calls (`to`, `is`, `as`, etc.) to more descriptive internal names, allowing function bodies to refer to `theValue` instead of `as` while retaining the natural-language-esque rhythm on the calling side.

6 Extensional in the sense of "listing out the entries"; a different meaning to the "extensional" of "evaluates all arguments"! Both of these philosophical concepts are relevant to Masp and it is unfortunate that they share the same word.
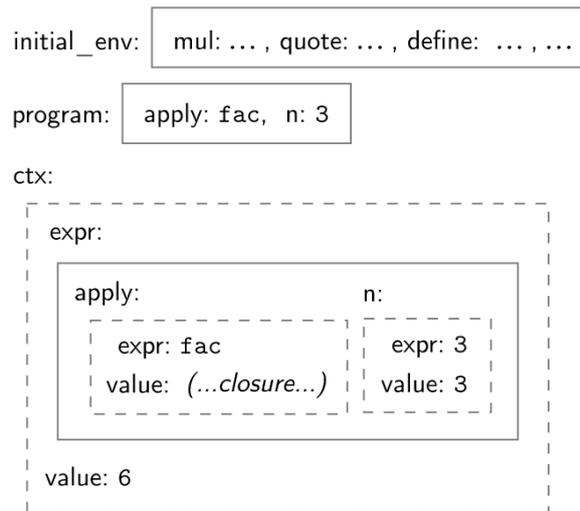
Figure D.1: The `masp` register contains an `initial_env` with basic primitives, a `program`, and the current evaluation context `ctx`. When fully evaluated, the program has its tree structure expanded into a tree of local contexts, each with an `expr` and a `value` (envs have been omitted for brevity). Solid rectangles enclose ordinary maps; dashed rectangles draw attention to local contexts.

D.2.2 *Masp Tree-Based Evaluation*

As shown in Section 4.7.4, Masp evaluates expressions incrementally and additively in a subtree of the state. This subtree lives under the `masp` top-level register. Inside it lie `initial_env`, the global variable-binding env of last resort; `program`, the program being evaluated; and `ctx`, the current evaluation context somewhere within `program`.

The JS interpreter function `masp_step()` performs a small evaluation step, e. g. resolving a variable binding (this is the "incremental" aspect of evaluation). As a JS function, it is "re-entrant" in the sense that all of the information it needs lives in the Masp tree, not the JS stack. Thus, every step, it must re-learn where it is in terms of evaluating the current expression and leave appropriate markers in the tree to direct its next run.

A *local evaluation context,* or *context* for short, contains a local `env`, an `expr`, and eventually a `value` (this is the "additive" aspect of evaluation; one can still see the expression that the value came from). If `masp_step` sees that there is no `value`, its goal is to make one. Otherwise, its goal is to point the `ctx` back up to the context for the parent expression, or pass the value up there. At all times, the original `program` tree is unaffected; after full evaluation, every expression will have been duplicated and wrapped in a context, forming a tree of contexts collectively containing a copy of the `program` (Figure D.1).

D.2.2.1 *Beginning the Walkthrough*

We will walk through the process from the perspective of `masp_step`. If the `expr` is a string, we add a `value` obtained from looking up the `expr` in the local `env`. If `expr` is a map with no `apply` entry, i. e. a "literal" map, we set the `value` to a special closure in which `literal` is set to the `expr` and the current `env` is included. Otherwise, if the `apply` node has not yet been evaluated, we wrap it in a new context and enter (i. e. we make it the new `ctx`). Once there is a value for the function closure, evaluation proceeds to the arguments unless `dont_eval_args` is set on the closure. A tracking variable in the context, `arg_i`, helps us discover the next unevaluated argument, as determined by the order of map keys from JS' `Object.keys()`.

D.2.2.2 *Protocol Between JS Primitives and Masp*

After processing the arguments, we inspect the `body` of the function closure. If it is a JS function, we call this "primitive" with the context and argument values (for the full list of primitives, see Section D.2.3). The protocol for Masp primitives is that returning `true` (on the JS stack) means that it has returned a value (in Masp); otherwise, it is evaluating a subexpression and needs to be run again afterwards. For example, here is the JS code for the `decr` (decrement) primitive:

```
'decr': { body: (c, args) =>
  { upd(c, 'value', args.to-1); return true; }}
```

It sets the value in the context and returns `true`, signalling that it has also returned in Masp. A more complicated *intensional* primitive is `define` (its usage reads "define *name: … as: …*"):

```
'define': { body: (c, args) => {
  if (typeof args.as === 'object') {
    const val = map_get(args.as, 'value');
    if (val !== undefined) {
      upd(masp, 'initial_env', 'entries', args.name, val);
      upd(c, 'value', null); return true;
    }
  }
  masp_enter('as');
}, dont_eval_args: true }
```

Because `dont_eval_args` is set, `define` will receive the `name` argument as an unevaluated string. It needs to evaluate the `as` argument, and if it has already done so, it binds it to the `name` in the global `initial_env`, sets the Masp return value as `null` (i. e. no return value, entirely side-effects) and returns `true` to return in Masp. Otherwise, it creates and enters a new context for `as`, and implicitly returns `undefined` in JS, which is falsy, telling the evaluator that `define` has not finished evaluating.

D.2.2.3 *Evaluating a Non-Primitive Body*

Back to inspecting the `body`. If a `body` is absent but we have a `literal` map instead, we push the single argument (named `to`) through the map and treat the result as the `body` (going via the `_` key if this fails). This facilitates pattern-matching for the purpose of obtaining *further code to execute* beyond obtaining just a value straight away.

At this point, we have a `body` made of Masp code to evaluate in an appropriate context. We create a new context, duplicate the body code so we can destructively replace expressions with contexts down the line, and set that as the `value` of the `apply` context. We fill the local `env` with the processed arguments, and enter this new context (i.e. set it as the `ctx`).

Finally, if we have been through all the preceding conditions in `masp_step` but did not "do" anything, and the current context already has a value, we must have finished evaluating this subexpression and re-enter the parent context to continue there.

D.2.3 *List of Primitives*

Here, we will describe the primitives contained in the Masp `initial_env`. Be aware that this list is not yet "complete" owing to the in-progress state of Masp. For space, the `apply` before each function name is abbreviated to `ap`. The primitives are:

- `ap: quote, to:`V evaluates to V. It is most commonly used to represent a literal string value, since in many contexts a string will be treated as a variable reference and evaluated as such. It could also be used to represent a data map that contains an `apply` key, "protecting" it from evaluation.

- `ap: mul, 1:`x`, 2:`y multiplies the numbers x and y. It is present to support the factorial example.

- `ap: decr, to:`x decrements x, i.e. evaluates to $x - 1$. We included it instead of a subtraction function to ease the number of evaluation steps when debugging the factorial example.

- `ap: fun, arg_names:`L`, body:`B creates a function closure with the list L of argument names, the body code B, and the lexical env at the point of definition. Re-naming of Lisp `lambda`.

- `ap: define, name:`S`, as:`E sets the key S in the `initial_env` to the result of evaluating E, imperatively setting a global variable.

- `ap: local, name:`S`, is:`E sets the key S in the local env to the result of evaluating E, imperatively setting a local variable.

- `ap: block, 1:`E_1`, 2:`E_2`, ...` imperatively executes the expressions E_n and evaluates to the value of the last one.

- `ap: get, map:`M`, key:`S evaluates to the value of key S in map M (or `null` if undefined).

- `ap: set, map:`M`, key:`S`, to:`E updates key S of map M to the value of E (updating any graphics, because it uses `upd` internally).

- `null` is a variable bound to the JS value `null`. Workaround for the fact that the tree editor can only input strings and numbers, not arbitrary JS values.

- `undefined`, `true`, and `false` are variables bound for the same reason as `null`.

D.3   IMPORTANT RELATED WORK

Procedural Reflection in Programming Languages (Smith 1982) was of great inspiration while we designed Masp, and remains of serious interest. The work builds up to the design of a fully, elegantly *reflective* version of Lisp called 3-LISP. In order to do this, it explains the unique features and subtleties of ordinary "1-LISP" while critiquing inelegant aspects of its design, and applies philosophical rigour to address these issues. The intermediate 2-LISP is possessed of strong "Category Alignment" regarding the different types of data structures available (e. g. splitting 1-LISP's overloaded "list" into the applicative "pair" and the data "rail"), and methodically refines 1-LISP's `eval`/`apply` into a philosophically rigorous `normalise`/`reduce`. For reasons of time and resource availability, we designed Masp mostly as an analogue of 1-LISP, inheriting its flaws. However, we would welcome efforts to "port" Smith's methodology regarding 2-LISP and 3-LISP to map-based substrates.

Another approach worth developing is found in (Piumarta 2011). *Open, Extensible Composition Models* inserts a level of indirection into the Lisp evaluator, such that user-defined *evaluators* and *applicators* can be supplied to support "novel composition mechanisms". These are also interesting ideas which, regrettably, we did not have the opportunity to try out. We welcome further work to port the ideas to maps and see if they are compatible with the 2-LISP or 3-LISP architectures.

E

TECHNICAL DIMENSIONS CATALOGUE

Here, we present our proposed technical dimensions in detail. While they do contain some novel ideas of our own, they also integrate a wide range of existing concepts under a common umbrella. Note that the material here represents an update of our publication (Jakubovic et al. 2023), mainly in regards to our Three Properties. Their dimensions in Section 5.4 represent an evolution of some of the concepts in the paper, specifically:

- Our definition of Explicit Structure (Section 3.3.3) supersedes what was previously "Surface / internal notations".

- Notational Freedom (Section 3.3.2) came from some aspects of the *Notation* dimensions (Section E.2), particularly *uniformity of notations*.

- Our description of Self-Sustainability (Section 3.3.1) is a fully developed version of the old dimension of the same name.

Therefore, in order to make this reference catalogue self-contained, we have rewritten the relevant sections here with adaptations of the material in Chapters 3 and 5.

The intention of this catalogue is to provide a *reference* to be looked up and *used* as needed, not something that should be read from start to finish. We recommend skimming through the catalogue for anything particularly interesting before proceeding to Section E.8. There, we will reference several dimensions in the context of a specific example, at which point it may be helpful to come back for more detail. For a quick overview, we include a concise reference sheet on the next page, though it may make more sense after reading the relevant sections.

We present the dimensions grouped under *clusters*. These may be regarded as "topics of interest" or "areas of inquiry" when studying a given system, grouping together related dimensions against which to measure it.

Each cluster is named and opens with an indented *summary*, followed by a longer *discussion*, and closes with a list of any *relations* to other clusters along with any *references* if applicable. Within the main discussion, individual *dimensions* are listed, although our Three Properties are first introduced as *concepts*. Sometimes, a particular value along a dimension (or a combination of values along several dimensions) can be recognised as a familiar pattern—perhaps with a name already established in the literature. These are marked as *examples*. Finally, interspersed discussion is introduced as a *remark*.

Dimension (CLUSTER)	Summary	Range of key examples
INTERACTION	How do users manifest their ideas, evaluate the result, and generate new ideas in response?	
Feedback Loops	How wide are the various gulfs of *execution* and *evaluation* and how are they related?	Immediate Feedback (short) vs. batch mode (long) gulf of evaluation
Modes of Interaction	Which sets of feedback loops only occur together?	Setup vs. editing vs. debugging
Abstraction Construction	How do we go from abstractions to concrete examples and vice versa?	Programming by Example vs. first principles
Implicit vs. Explicit Structure	How much time must users spend accounting for syntax or format errors?	Zero (explicit structure) vs. lots (implicit structure)
NOTATION	How are the different textual / visual programming notations related?	
Notational Structure	What notations are used to program the system and how are they related?	Notations overlap and need sync vs. complement each other
Primary / Secondary Notations	Is one notation more important than others?	Secondary build scripts vs. visual editor and code on equal footing in Flash
Expression Geography	Do similar expressions encode similar programs?	Concise yet error-prone vs. explicit yet verbose
Uniformity of Notations	Does the notation use a small or a large number of basic concepts?	Lisp S-expressions vs. English-like textual notations
Notational Freedom	How easily can the system integrate novel notations that the user has implemented?	Encourages custom notations vs. enforces a single one
CONCEPTUAL STRUCTURE	How is meaning constructed? How are internal and external incentives balanced?	
Conceptual Integrity vs. Openness	Does the system present as elegantly *designed* or pragmatically *improvised*?	Integrity (Everything is a X) vs. openness (compatible mixtures)
Composability	What are the primitives? How can they be combined to achieve novel behaviors?	Sequence, selection, repetition, function abstraction, recursion, logical connectives
Convenience	Which wheels do users not need to reinvent?	Small vs. expansive standard libraries
Commonality	How much is *common structure* explicitly marked as such?	Common structure is redundantly flattened vs. factored out
CUSTOMIZABILITY	Once a program exists in the system, how can it be extended and modified?	
Staging of Customization	Must we customize *running* programs differently to *inert* ones? Do these changes last beyond termination?	Source code vs. config files, Developer Tools tab, auto image-based persistence, scripting language
Externalizability	Which portions of the system's state can be referenced and transferred to/from it?	None (state is private) vs. all state exposed as human-legible, CSS-like addressing
Additive Authoring	How far can the system's behavior be changed by *adding* expressions?	None (requires power to change original) vs. full (anything can be overridden repeatedly)
Self-Sustainability	How far can the system's behavior be changed from within?	None (rely on external tools) vs. self-sufficient (contains everything needed)
COMPLEXITY	How does the system structure complexity and what level of detail is required?	
Factoring of Complexity	What programming details are hidden in reusable components and how?	Domain-specific (more hiding) vs. general-purpose (less hiding)
Level of Automation	What part of program logic does not need to be explicitly specified?	Garbage collection (low-tech) vs. Prolog engine (hi-tech)
ERRORS	What does the system consider to be an *error*? How are they prevented and handled?	
Error Detection	What errors can be detected in which feedback loops, and how?	Human inspection in live coding vs. partial automation in static typing
Error Response	How does the system respond when an error is detected?	Does it stop, recover automatically, ignore the error or ask the user how to continue?
ADOPTABILITY	How does the system facilitate or obstruct adoption by both individuals and communities?	
Learnability	What is the attitude towards the *learning curve* and what is the target audience?	HyperCard for the general public vs. FORTRAN for scientists
Sociability	What are the social and economic factors that make the system the way it is?	Cathedral vs. Bazaar model

> How do users manifest their ideas, evaluate the result, and generate new ideas in response?

An essential aspect of programming systems is how the user interacts with them when creating programs. Take the standard form of statically typed, compiled languages with straightforward library linking: here, programmers write their code in a text editor, invoke the compiler, and read through error messages they get. After fixing the code to pass compilation, a similar process might happen with runtime errors.

Other forms are yet possible. On the one hand, some typical interactions like compilation or execution of a program may not be perceptible at all. On the other hand, the system may provide various interfaces to support the plethora of other interactions that are often important in programming, such as looking up documentation, managing dependencies, refactoring or pair programming.

We focus on the situations where a programmer interacts with the system to construct a program with a desired behaviour. To analyse them, we use the concepts of *gulf of execution* and *gulf of evaluation* from *The Design of Everyday Things* (Norman 2002).

E.1.1   *Dimension: feedback loops*

In using a system, one first has some idea and attempts to make it exist in the software; the gap between the user's goal and the means to execute the goal is known as the *gulf of execution*. Then, one compares the result actually achieved to the original goal in mind; this crosses the *gulf of evaluation*. These two activities comprise the *feedback loop* through which a user gradually realises their desires in the imagination, or refines those desires to find out "what they actually want".

A system must contain at least one such feedback loop, but may contain several at different levels or specialised to certain domains. For each of them, we can separate the gulf of execution and evaluation as independent legs of the journey, with possibly different manners and speeds of crossing them.

For example, we can analyse statically checked *programming languages* (e.g. Java, Haskell) into several feedback loops (Figure E.1):

1. Programmers often think about design details and calculations on a whiteboard or notebook, even before writing code. This *supplementary medium* has its own feedback loop, even though this is often not automatic.

2. The code is written and is then put through the static checker. An error sends the user back to writing code. In the case of success, they are "allowed" to run the program, leading into cycle 3.

Feedback loop =
Gulf of Execution +
Gulf of Evaluation

Supplementary medium =
e.g. paper notebook for
working out the code design

Cycle 1: Supplementary medium
Repeats until code ready to submit

Cycle 2: Static checks
Repeats until new code is
statically valid

Cycle 3: Runtime observation
Repeats until program "works
well enough"

execute

evaluate
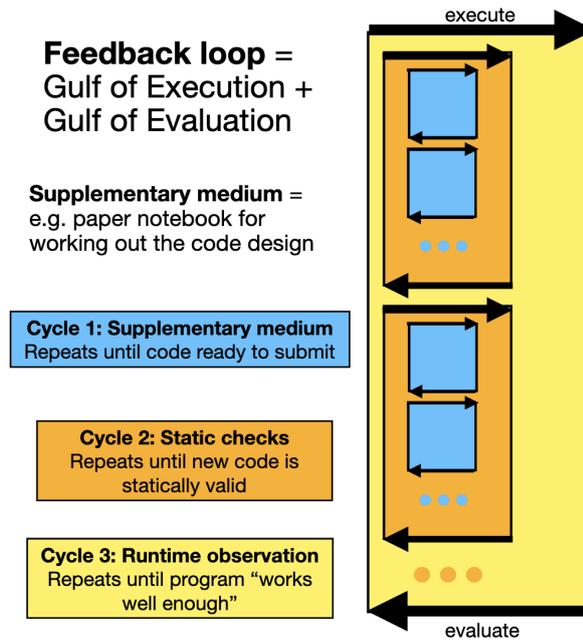
Figure E.1: The nested feedback loops of a statically-checked programming
language.

- The execution gulf comprises multiple cycles of the supplementary medium, plus whatever overhead is needed to invoke the compiler (such as build systems).

- The evaluation gulf is essentially the waiting period before static errors or a successful termination are observed. Hence this is bounded by some function of the length of the code (the same cannot be said for the following cycle 3).

3. With a runnable program, the user now evaluates the *runtime* behaviour. Runtime errors can send the user back to writing code to be checked, or to tweak dynamically loaded data files in a similar cycle.

- The execution gulf here may include multiple iterations of cycle 2, each with its own nested cycle 1.

- The *evaluation* gulf here is theoretically unbounded; one may have to wait a very long time, or create very specific conditions, to rule out certain bugs (like race conditions) or simply to consider the program as fit for purpose.

- By imposing *static checks*, some bugs can be pushed earlier to the evaluation stage of cycle 2, reducing the likely size of the cycle 3 *evaluation* gulf.

- On the other hand, this can make it harder to write statically valid code, which may increase the number of level-2 cycles, thus increasing the total *execution* gulf at level 3.

- Depending on how these balance out, the total top-level feedback loop may grow longer or shorter.

E.1.2 *Example: immediate feedback*

The specific case where the *evaluation* gulf is minimised to be imperceptible is known as *immediate feedback*. Once the user has caused some change to the system, its effects (including errors) are immediately visible. This is a key ingredient of *liveness*, though it is not sufficient on its own. (See *Relations.*)

The ease of achieving immediate feedback is obviously constrained by the computational load of the user's effects on the system, and the system's performance on such tasks. However, such "loading time" is not the only way feedback can be delayed: a common situation is where the user has to manually ask for (or "poll") the relevant state of the system after their actions, even if the system finished the task quickly. Here, the feedback could be described as *immediate upon demand* yet not *automatically demanded*. For convenience, we choose to include the latter criterion—automatic demand of result—in our definition of immediate feedback.

In a *REPL* or *shell*, there is a *main* cycle of typing commands and seeing their output, and a *secondary* cycle of typing and checking the command line itself. The output of commands can be immediate, but usually reflects only part of the total effects or even none at all. The user must manually issue further commands afterwards, to check the relevant state bit by bit. The secondary cycle, like all typing, provides immediate feedback in the form of character "echo", but things like syntax errors generally only get reported *after* the entire line is submitted. This evaluation gulf has been reduced in the JavaScript console of web browsers, where the line is "run" in a limited manner on every keystroke. Simple commands without side-effects,[1] such as calls to pure functions, can give instantly previewed results—though partially typed expressions and syntax errors will not trigger previews.

E.1.3 *Example: direct manipulation*

Direct manipulation (Shneiderman 1983) is a special case of an immediate feedback loop. The user sees and interacts with an artefact in a way that is as similar as possible to real life; this typically includes dragging with a cursor or finger in order to physically move a visual item, and is limited by the particular haptic technology in use.

---

1 Of course, these are detected via some conservative over-approximation which excludes expressions that *might* side-effect.

Naturally, because moving real things with one's hands does not involve any waiting for the object to "catch up",[2] direct manipulation is necessarily an immediate-feedback cycle. If, on the other hand, one were to move a figure on screen by typing new co-ordinates in a text box, then this could still give *immediate feedback* (if the update appears instant and automatic) but would *not* be an example of direct manipulation.

Spreadsheets contain a feedback loop for direct manipulation of values and formatting, as in any other WYSIWYG application. Here, there is feedback for every character typed and every change of style. This is not the case in the other loop for formula editing and formula invocation. There, we see a larger execution gulf for designing and typing formulas, where feedback is only given upon committing the formula by pressing enter. This makes it an "immediate feedback" loop only *on-demand*, as defined above.

E.1.4   *Dimension: modes of interaction*

The possible interactions in a programming system are typically structured so that interactions, and the associated feedback loops, are only available in certain *modes*. For example, when creating a new project, the user may be able to configure the project through a conversational interface like `npm init` in modern JavaScript. Such interactions are no longer available once the project is created. This idea of interaction modes goes beyond just programming systems, appearing in software engineering methodologies. In particular, having a separate *implementation* and *maintenance* phase would be an example of two modes.

Editing vs debugging. A good example is the distinction between *editing* and *debugging* mode. When debugging a program, the user can modify the program state and get (more) immediate feedback on what individual operations do. In some systems, one can even modify the program itself during debugging. Such feedback loops are not available outside of debugging mode.

Lisp systems sometimes distinguish between *interpreted* and *compiled* mode. The two modes do not differ just in the efficiency of code execution, but also in the interactions they enable. In the interpreted mode, code can be tested interactively and errors may be corrected during code execution (see *Error response*). In compiled mode, the program can only be tested as a whole. The same two modes also exist, for example, in some Haskell systems where the REPL uses an interpreter (GHCi) distinct from the compiler (GHC).

---

2 In some situations, such as steering a boat with a rudder, there is a delay between input and effect. But on closer inspection, this delay is between the rudder and the boat; we do not see the hand pass through the wheel like a hologram, followed by the wheel turning a second later. In real life, objects touched directly give immediate feedback; objects controlled further down the line might not!

Jupyter notebooks. A programming system may also unify modes that are typically distinct. The Jupyter notebook environment does not have a distinct debugging mode; the user runs blocks of code and receives the result. The single mode can be used to quickly try things out, and to generate the final result, partly playing the role of both debugging and editing modes. However, even Jupyter notebooks distinguish between editing a document and running code.

E.1.5   *Dimension: abstraction construction*

A necessary activity in programming is going between abstract schemas and concrete instances. Abstractions can be constructed from concrete examples, first principles or through other methods. A part of the process may happen in the programmer's mind: they think of concrete cases and come up with an abstract concept, which they then directly encode in the system. Alternatively, a system can support these different methods directly.

One option is to construct abstractions *from first principles*. Here, the programmer starts by defining an abstract entity such as an interface in object-oriented programming languages. To do this, they have to think what the required abstraction will be (in the mind) and then encode it (in the system).

Another option is to construct abstractions *from concrete cases*. Here, the programmer uses the system to solve one or more concrete problems and, when they are satisfied, the system guides them in creating an abstraction based on their concrete case(s). In a programming language IDE this manifests as the "extract function" refactor, whereas in other systems we see approaches like macro recording.

Pygmalion. In Pygmalion (Smith 1975), all programming is done by manipulating concrete icons that represent concrete things. To create an abstraction, one can use "Remember mode", which records the operations done on icons and makes it possible to bind this recording to a new icon.

Jupyter notebook. In Jupyter notebooks, one is inclined to work with concrete things, because one sees previews after individual cells. This discourages creating abstractions, because then one would not be able to look inside at such a fine grained level.

Spreadsheets. Up until the recent introduction of lambda expressions into Excel (Murray 2022), spreadsheets have been relentlessly concrete, without any way to abstract and reuse patterns of computation other than copy-and-paste.

E.1.6   *Concept: implicit vs. explicit structure*

Explicit Structure refers to the sense of working with data *directly* rather than through some other medium. It is a property of the interaction

between users, interfaces, and underlying data structures. We split the
life-cycle of a data structure into two halves:

- On the *producer side*, the data structure is created or edited using
  some interface.

- On the *consumer side*, a programmer is writing code that uses the
  data structure.

Explicit Structure is hard to define positively because it is the default
state of affairs across much of computing, with programming being the
notable exception. On the producer side, explicit structure is exhibited
by a vector graphics editor like Inkscape: one simply draws a diagram
with shapes and saves it as a file. On the consumer side, Explicit Struc-
ture looks like a programmer navigating through named parts of the
diagram structure:

```
svg.root_nodes[1].children[2].fill_color = '#ff00ff';
```

The best way we have found to model Explicit Structure is as a lack
of Implicit Structure, where we follow the split between *producer* and
consumer concerns. On the producer side, we have an editor with an
interface creating and changing a data structure. This is saved and
passed onto consumers, which can be collaborators using editors or
a programmer writing code to use the data structure. The definition
of Implicit Structure we are interested in is about how much users or
programmers must be *aware* of it and devote cognitive resources to
working with it. On the producer side, this manifests as which types of
syntax errors or more general *format* errors they are able to save and pass
on to consumers. On the consumer side, Implicit Structure is revealed
by the amount of code we have to write to deal with parsing, serialising,
escaping, loading and saving, and so on. Therefore, we suggest the
following two dimensions which add up to Implicit Structure.

E.1.7 *Dimension: format errors*

How many different types of format errors can be introduced, saved as
invalid structures, and passed to consumers, such that they will halt
with an error? For example, text editors allow all possible syntax errors
to be saved and several format errors (e. g. type mismatches and use
of undeclared names). However, a text editor interface that refused to
save invalid files could form part of a system with Explicit Structure.

Block or structure editors may prevent all format errors from being
saved, which would constitute the minimal value of this dimension.
The approach of Hazel (Omar et al. 2019) is unique in having language
semantics that *tolerates* fully typed "holes" representing missing ex-
pressions, which obviates concerns about persisting or transferring
such programs. This highlights the fact that Implicit Structure is about

mismatches between the producer and consumer of data, rather than producer leniency or consumer intolerance *per se.*

Arguably, the maximal value of Implicit Structure would be found in a hex editor: any data at all may be saved and passed on. This can violate even the basic rules of text files through non-printable characters, while any and all internal rules of a binary file can also be easily broken.

E.1.8 *Dimension: string wrangling effort*

How much code has to be written to convert between Implicit and Explicit Structure? Explicit Structure implies a minimal value for this and would look something like the following:

```
data = load('filename')
data.foo.bar = 'baz';
```

Here, there are zero lines of string wrangling. Only one line, translating between the filesystem and the internal system namespaces, is required to prepare the data structure for use.

If such a load function is already present, then users experience no string wrangling effort for the use cases of this function, i. e. the file formats it supports. If the function does not exist, and a user must write string wrangling code on an ad-hoc basis, this dimension is correspondingly high relative to that format. Suppose the user factors this ad-hoc string wrangling into their *own* implementation of the load function; this implementation effort would count towards the dimension, but would pay for itself in the reduced string wrangling effort thereafter; this situation would lie somewhere between the previous two.

These considerations all establish scores for this dimension *relative* to a particular file format or string syntax. These could be aggregated to form a score for a particular program which uses several such formats. However, if we are trying to assess the *programming system* along this dimension, we would have to somehow aggregate across all possible programs one could create with the system, including the various different formats they are likely to include.[3]

Recognising that different programming systems are targeted at different goals and have differing strengths and weaknesses, the possibility space could be refined into all *likely* programs or use cases of the programming system, weighted by the probability of a user of the system wanting to create such a program. This opens up further decisions about this user and whether we should additionally aggregate across possible (or likely) users of the system. We defer such complications to future work.

---

[3] There is a large variety of *existing* data storage formats (e. g. JSON and XML) and an infinite variety of potential *custom* formats that could be created on an ad-hoc basis (e. g. chat messages containing special escape sequences).

E.1.9   *Relations*

- *Errors* (Section E.6) A longer evaluation gulf delays the detection of errors. A longer execution gulf can increase the *likelihood* of errors (e.g. writing a lot of code or taking a long time to write it). By turning runtime bugs into statically detected bugs, the combined evaluation gulfs can be reduced.

- *Adoptability* (Section E.7): The *execution* gulf is concerned with software using and programming in general. The time taken to realise an idea in software is affected by the user's familiarity and the system's *learnability*.

- *Notation* (Section E.2): Feedback loops are related to *notational structures*. In a system with multiple notations, each notation may have different associated feedback loops. The motto "The thing on the screen is supposed to be the actual thing" (Pawson 2004), adopted in the context of live programming, relates *liveness* to a direct connection between surface notation and internal data. The idea is that interactable objects should be equipped with faithful behaviour, instead of being intangible shadows cast by the hidden *real* object.

E.2   NOTATION

How are the different textual / visual programming notations related?

Programming is always done through some form of notation. We consider notations in the most general sense and include any structured gesture using textual or visual notation. Textual notations primarily include programming languages, but also things like configuration files. Visual notations include graphical programming languages. Other kinds of structured gestures include user interfaces for constructing visual elements used in the system.

E.2.1   *Dimension: notational structure*

In practice, most programming systems use multiple notations. Different notations can play different roles in the system. On the one hand, multiple *overlapping notations* can be provided as different ways of programming the same aspects of the system. In this case, each notation may be more suitable to certain kinds of users, but may have limitations (for example, a visual notation may have a limited expressive power). On the other hand, multiple *complementing notations* may be used as the means for programming different aspects of the system. In this case, programming the system requires using multiple notations, but the

notations may be specialised for the task at hand; think of how HTML describes document structure while JavaScript specifies its behaviour.

E.2.2 *Example: overlapping notations*

A programming system may provide multiple notations for programming the same aspect of the system. This is typically motivated by an attempt to offer easy ways of completing different tasks: say, a textual notation for defining abstractions and a visual notation for specifying concrete structures. The crucial issue in this kind of arrangement is *synchronising* the different notations; if they have different characteristics, this may not be a straightforward mapping. For example, source code may allow more elaborate abstraction mechanisms like loops, which might appear as visible repetition in the visual notation. What should such a system do when the user edits a single object that resulted from such repetition? Similarly, textual notation may allow incomplete expressions that do not have an equivalent in the visual notation. For programming systems that use *overlapping notations*, we need to describe how the notations are synchronised.

Sketch-n-Sketch (Hempel et al. 2019) employs overlapping notations for creating and editing SVG and HTML documents. The user edits documents in an interface with a split-screen structure that shows source code on the left and displayed visual output on the right. They can edit both of these and changes are propagated to the other view. The code can use abstraction mechanisms (such as functions) which are not completely visible in the visual editor (an issue we return to in *expression geography* below). Sketch-n-Sketch can be seen as an example of a *projectional editor*.[4]

UML Round-tripping. Another example of a programming system that utilises the *overlapping notations* structure are UML design tools that display the program both as source code and as a UML diagram. Edits in one result in automatic update of the other. An example is the Together/J[5] system. To solve the issue of notation synchronisation, such systems often need to store additional information in the textual notation, typically using a special kind of code comment. In this example, after the user re-arranges classes in UML diagrams, the new locations need to be updated in the code.

E.2.3 *Example: complementing notations*

A programming system may also provide multiple complementing notations for programming different aspects of its world. Again, this is typically motivated by the aim to make specifying certain aspects

---

4 Technically, traditional projectional editors usually work more directly with the abstract syntax tree of a programming language.

5 https://www.mindprod.com/jgloss/togetherj.html

of programming easier, but it is more suitable when the different aspects can be more clearly separated. The key issue for systems with complementing notations is how the different notations are connected. The user may need to use both notations at the same time, or they may need to progress from one to the next level when solving increasingly complex problems. In the latter case, the learnability of progressing from one level to the next is a major concern.

Spreadsheets and HyperCard. In Excel, there are three different complementing notations that allow users to specify aspects of increasing complexity: (i) the visual grid, (ii) formula language and (iii) a macro language such as Visual Basic for Applications. The notations are largely independent and have different degrees of expressive power. Entering values in a grid cannot be used for specifying new computations, but it can be used to adapt or run a computation, for example when entering different alternatives in What-If Scenario Analysis. More complex tasks can be achieved using formulas and macros. A user gradually learns more advanced notations, but experience with a previous notation does not help with mastering the next one. The approach optimises for easy learnability at one level, but introduces a hurdle for users to surmount in order to get to the second level. The notational structure of *Hyper-Card* is similar and consists of (i) visual design of cards, (ii) visual programming (via the GUI) with a limited number of operations, and (iii) HyperTalk for arbitrary scripting.

Boxer and Jupyter. Boxer (diSessa and Abelson 1986) uses *complementing notations* in that it combines a visual notation (the layout of the document and the boxes of which it consists) with textual notation (the code in the boxes). Here, the textual notation is always nested within the visual. The case of Jupyter notebooks is similar. The document structure is graphical; code and visual outputs are nested as editable cells in the document. This arrangement is common in many other systems such as Flash or Visual Basic, which both combine visual notation with textual code, although one is not nested in the other.

E.2.4   *Dimension: primary and secondary notations*

In practice, most programming systems use multiple notations. Even in systems based on traditional programming languages, the *primary notation* of the language is often supported by *secondary notations* such as annotations encoded in comments and build tool configuration files. However, it is possible for multiple notations to be primary, especially if they are *overlapping* as defined earlier.

Programming languages. Programming systems built around traditional programming languages typically have further notations or structured gestures associated with them. The primary notation in Unix is the C programming language. Yet this is enclosed in a programming *system* providing a multi-step mechanism for running C code via the

terminal, assisted by secondary notations such as shell scripts. Some programming systems attempt to integrate tools that normally rely on secondary notations into the system itself, reducing the number of secondary notations that the programmer needs to master. For example, in the Smalltalk descendant Pharo, versioning and package management is done from within Pharo, removing the need for secondary notation such as `git` commands and dependency configuration files.[6]

Haskell. In Haskell, the primary notation is the programming language, but there are also a number of secondary notations. Those include package managers (e.g. the `cabal.project` file) or configuration files for Haskell build tools. More interestingly, there is also an informal mathematical notation associated with Haskell that is used when programmers discuss programs on a whiteboard or in academic publications (Bird and Moor 1997). The idea of having such a mathematical notation dates back to the *Report on Algol 58* (Perlis and Samelson 1958), which explicitly defined a "publication language" for "stating and communicating problems" using Greek letters and subscripts.

E.2.5 *Dimension: expression geography*

A crucial feature of a notation is the relationship between the structure of the notation and the structure of the behaviour it encodes. Most importantly, do *similar expressions* in a particular notation represent *similar behaviour*?[7] Visual notations may provide a mapping that is more or less direct. On the one hand, similar-looking code in a block language may mean very different things. On the other hand, the similar looking design of two HyperCard cards will result in similar looking cards—the mapping between the notation and the logic is much more direct.

C/C++ expression language. In textual notations, this may easily not be the case. Consider the two C conditionals:

- `if (x==1) { ... }` evaluates the Boolean expression x==1 to determine whether x equals 1, running the code block if the condition holds.

- `if (x=1) { ... }` *assigns* 1 to the variable x. In C, assignment is an expression *returning* the assigned value, so the result 1 is interpreted as `true` and the block of code is *always* executed.

A notation can be designed to map better to the logic behind it, for example, by requiring the user to write `1==x`. This solves the above problem as 1 is a literal rather than a variable, so it cannot be assigned to (`1=x` is a compile error).

---

6 The tool for versioning and package management in Pharo can still be seen as an *internal* domain-specific language and thus as a secondary notation, but its basic structure is *shared* with other notations in the Pharo system.

7 See the similar discussion of "density" in (Basman 2016).

E.2.6   *Dimension: uniformity of notations*

One common concern with notations is the extent to which they are uniform. A uniform notation can express a wide range of things using just a small number of concepts. The primary example here is S-expressions from Lisp.

Lisp systems. In Lisp, source code is represented in memory as S-expressions, which can be manipulated by Lisp primitives. Lisp defines an S-expression as either an atom, or a pair of S-expressions written `(s1 . s2)`. By convention, an S-expression `(s1 . (s2 . (s3 . nil)))` represents a list, written as `(s1 s2 s3)`. Notational uniformity in Lisp is closely linked to uniformity of representation.[8] In the idealised model of LISP 1.5, the data structures represented by an S-expression are what exists in memory. In real-world Lisp systems, the representation in memory is more complex. A programming system can also take a very different approach and fully separate the notation from the in-memory representation.

The S-expression basis of Lisp gives such systems robust macro processing as part of their semantics: expanding a macro revises the list structure of the code that uses the macro. Combining these makes it possible to define extensions to the system in Lisp, with syntax indistinguishable from Lisp. Moreover, it is possible to write a program that constructs another Lisp program and not only run it interpretively (using the `eval` function) but compile it at runtime (using the `compile` function) and execute it. Many domain-specific languages, as well as prototypes of new programming languages (such as Scheme), were implemented this way. Lisp the language is, in this sense, a "programmable programming language" (Felleisen et al. 2018; Foderaro 1991).

E.2.7   *Concept: notational freedom*

Notational Freedom is the full realisation of the maxim "Use The Right Tool For The Job". The ideal is where a component at any scale can be expressed in a *notation* that is particularly suited to it. There are several related concepts of which Notational Freedom is the full generalisation: *syntactic* freedom, *linguistic* freedom, and syntactic/linguistic/notational *plurality.*

Syntactic Freedom is where custom *syntaxes,* or languages restricted to the conventions of code (linear display, plain text without formatting), may be defined and used as desired by the programmer. An established instance of this is Domain-Specific Languages. Syntactic Freedom allows a programmer, if dissatisfied with the following:

```
vec_a.mul(cos(ang_b/2))
```

---

8 Notations generally are closely linked to representation in that the notation may mirror the structures used for program representation. Basman et al. (2016) refer to this as a distinction between "dead" notation and "live" representation forms).

```
.add(vec_b.mul(cos(ang_a/2)))
.add(vec_a.cross(vec_b))
```

to define a local syntax that allows him to rewrite it as the following (note the Unicode infix operators and juxtaposition-as-multiplication):

```
cos(ang_b/2) vec_a + cos(ang_a/2) vec_b + vec_a × vec_b
```

Linguistic Freedom is a relaxation on the constraints of syntactic freedom to support language-like notations with non-linear display and different text styles. Standard mathematical notation is an example of such a notation; Linguistic Freedom would permit the following development of the previous example:

$$\cos\left(\frac{b}{2}\right)\mathbf{a} + \cos\left(\frac{a}{2}\right)\mathbf{b} + \mathbf{a} \times \mathbf{b}$$

Syntactic/linguistic/notational *plurality* is where different notations of the respective class are available to the programmer, but only as a *pre-determined* set of options; the programmer does not have *freedom* to define his own and use them wherever he wants. Examples are the inclusion of *regex* syntax in Perl and JavaScript; the three-way collaboration of HTML, JavaScript and CSS in web development; LINQ in C#.

Full *Notational* Freedom includes syntactic and linguistic freedom (which themselves include plurality) but goes beyond, allowing arbitrary graphical notations. An example of what one could do with this freedom is to replace a hex string with a colour picker interface.

The way we measure Notational Freedom is as the minimisation of several *penalty* dimensions which we turn to presently. The work involved in creating the syntax, language or notation is excluded; the dimensions only measure the amount of *extra* work it takes to get the programming system to accept them in the contexts they are intended for. They are minimised if the custom syntax, language, or notation can be "slotted in" once it exists, with no resistance from the system.

E.2.8   *Dimension: custom syntax effort*

The work required to use a custom syntax, not counting that required to specify the syntax itself (e. g. as a grammar). COLA (Piumarta 2006) and OMeta (Warth 2009) score low on this, since they are specifically designed for this purpose. Most programming languages have *infinite* custom syntax effort, because their parsers are separate programs that adhere to a fixed grammar that cannot be changed by statements in the language. This includes JS despite its inclusion of a regex sub-syntax, HTML despite its inclusion of JS and CSS, and C# despite its LINQ sub-language for queries; these examples may exhibit syntactic *plurality*, but

there is no way to include a user-supplied syntax for use in the source code.

E.2.9   *Dimension: custom language effort*

The work required to use custom language-like notation beyond syntax, not counting that required to implement the rendering and interaction. Most programming languages, COLA, and OMeta get an infinite score here, while MPS (Voelter and Pech 2012) and Eco (Diekmann and Tratt 2014) score low.

E.2.10   *Dimension: custom notation effort*

The work required to use custom graphical notation beyond what we called "language-like" notations, not counting that required to implement the rendering and interaction. Non-infinite scores here are rare, only exhibited in a few systems. Eco, owing to a screenshot showing inclusion of a picture, scores non-infinite on this dimension. From their discussion in Section 9.2 of the paper (Diekmann and Tratt 2014), it is likely to score High or Moderate rather than Low because arbitrary graphical notations are a novel unexplored use case for the system for which it has not been optimised. On the other hand, the Glamorous Toolkit (Chiş et al. 2015) appears built to support arbitrary graphics in its text editor interfaces, suggesting a lower value on this dimension. ThingLab I (Borning 1981) and II (Moloney et al. 1989) would similarly score low, at least insofar as diagrammatic notations are concerned (as opposed to integration with the editing of Smalltalk code). Baby-lonian Programming (Rauch et al. 2019), built on the self-sustainable Lively4 (Lincke et al. 2017), is also likely to have a lower score on this penalty dimension.

E.2.11   *References*

Cognitive Dimensions of Notations (Green and Petre 1996) provide a comprehensive framework for analysing individual notations, while our focus here is on how multiple notations are related and how they are structured. It is worth noting that the Cognitive Dimensions also define *secondary notation*, but in a different sense to ours. For them, secondary notation refers to whether a notation allows including redundant information such as colour or comments for readability purposes.

The importance of notations in the practice of science, more generally, has been studied by (Klein 2003) as "paper tools". These are formula-like entities which can be manipulated by humans in lieu of experimentation, such as the aforementioned mathematical notation in Haskell: a "paper tool" for experimentation on a whiteboard. Program-

ming notations are similar, but they are a way of communicating with a machine; the experimentation does not happen on paper alone.

E.2.12 *Relations*

- *Interaction* (Section E.1): The feedback loops that exist in a programming system are typically associated with individual notations. Different notations may also have different feedback loops.

- *Adoptability* (Section E.7): Notational structure can affect learnability. In particular, complementing notations may require (possibly different) users to master multiple notations. Overlapping notations may improve learnability by allowing the user to edit the program in one way (perhaps visually) and see the effect in the other notation (such as code).

- *Errors* (Section E.6). A process that merely records user actions in a sequence (such as text editing) will, in particular, record any *errors* the user makes and defer their handling to later use of the data, keeping the errors *latent*. A process which instead treats user actions as edits to a structure, with constraints and correctness rules, will be able to catch errors at the moment they are introduced and ensure the data coming out is error-free.

E.3 CONCEPTUAL STRUCTURE

How is meaning constructed? How are internal and external incentives balanced?

E.3.1 *Dimension: conceptual integrity vs. openness*

The evolution of programming systems has led away from *conceptual integrity* towards an intricate ecosystem of specialised technologies and industry standards. Any attempt to unify parts of this ecosystem into a coherent whole will create *incompatibility* with the remaining parts, which becomes a major barrier to adoption. Designers seeking adoption are pushed to focus on localised incremental improvements that stay within the boundaries established by existing practice. This creates a tension between how highly they can afford to value conceptual elegance, and how open they are to the pressures imposed by society. We will turn to both of these opposite ends—*integrity* and *openness*—in more detail.

E.3.2   *Example: conceptual integrity*

> I will contend that Conceptual Integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. (Fred Brooks, *Aristocracy, Democracy and System Design*; Brooks (1995)

> Conceptual integrity arises not (simply) from one mind or from a small number of agreeing resonant minds, but from sometimes hidden co-authors and the thing designed itself. (Richard Gabriel, *Designed As Designer*; Gabriel (2008)

Conceptual integrity strives to reduce complexity at the source; it employs *unified concepts* that may *compose orthogonally* to generate diversity. Perhaps the apotheosis of this approach can be found in early Smalltalk and Lisp environments, which were complete programming systems built around a single language. They incorporated capabilities commonly provided *outside* the programming language by operating systems and databases. Everything was done in one language, and so everything was represented with the datatypes of that language. Likewise, the libraries and idioms of the language were applicable in all contexts. Having a *lingua franca* avoided much of the friction and impedance mismatches inherent to multi-language systems. A similar drive exists in the Python programming language, which follows the principle that "There should be one—and preferably only one—obvious way to do it" in order to promote community consensus on a single coherent style.

In addition to Smalltalk and Lisp, many programming languages focus on one kind of data structure (Sitaker 2016):

- In COBOL, data consists of nested records as in a business form.

- In Fortran, data consists of parallel arrays.

- In SQL, data is a set of relations with key constraints.

- In scripting languages like Python, Ruby, Lua, and JavaScript, much data takes the form of string-indexed hash tables.

Finally, many languages are *imperative*, staying close to the hardware model of addressable memory, lightly abstracted into primitive values and references into mutable arrays and structures. On the other hand, *functional* languages hide references and treat everything as immutable structured values. This conceptual simplification benefits certain kinds of programming, but can be counterproductive when an imperative approach is more natural, such as in external input/output.

E.3.3   *Example: conceptual openness*

Perl, contra Python. In contrast to Python's outlook, Perl proclaims "There is more than one way to do it" and considers itself "the first postmodern programming language" (Wall 1999). "Perl doesn't have any agenda at all, other than to be maximally useful to the maximal number of people. To be the duct tape of the Internet, and of everything else." The Perl way is to accept the status quo of evolved chaos and build upon it using duct tape and ingenuity. Taken to the extreme, a programming system becomes no longer a *system*, properly speaking, but rather a *toolkit for improvising* assemblages of *found* software. Perl can be seen as championing the values of *pluralism*, *compatibility*, or *conceptual openness* over conceptual integrity. This philosophy has been called *Postmodern Programming* (Noble and Biddle 2004).

C++, contra Smalltalk. Another case is that of C++, which added to C the Object-Oriented concepts developed by Smalltalk while remaining 100% compatible with C, down to the level of ABI and performance. This strategy was enormously successful for adoption, but came with the tradeoff of enormous complexity compared to languages designed from scratch for OO, like Smalltalk, Ruby, and Java.

Worse, contra Better. Richard Gabriel first described this dilemma in his influential 1991 essay *Worse is Better* (Gabriel 1991) analysing the defeat of Lisp by Unix and C. Because Unix and C were so easy to port to new hardware, they were "the ultimate computer viruses" despite providing only "about 50%–80% of what you want from an operating system and programming language". Their conceptual openness meant that they adapted easily to the evolving conditions of the external world. The tradeoff was decreased conceptual integrity, such as the undefined behaviours of C, the junkyard of working directories, and the proliferation of special purpose programming languages to provide a complete development environment.

Unix and Files. Many programming languages and systems impose structure at a "fine granularity": that of individual variables and other data and code structures. Conversely, systems like Unix and the Web impose fewer restrictions on how programmers represent things. Unix insists only on a basic infrastructure of "large objects" (Kell 2013), delegating all fine-grained structure to client programs. This scores many points for conceptual openness. *Files* provide a universal API for reading and writing byte streams, a low-level construct containing so many degrees of freedom that it can support a wide variety of formats and ecosystems. *Processes* similarly provide a thin abstraction over machine-level memory and processors.

Concepual integrity is necessarily sacrificed for such openness; while "everything is a file" gestures at integrity, in the vein of Smalltalk's "everything is an object", exceptions proliferate. Directories are special kinds of files with special operations, hardware device files require

special `ioctl` operations[9], and many commands expect files containing newline separators. Additionally, because client programs must supply their *own* structure for fine-grained data and code, they are given little in the way of mutual compatibility. As a result, they tend to evolve into competing silos of duplicated infrastructure (Kell 2009, 2013).

The Web. Web HTTP endpoints, meanwhile, have proven to be an even more adaptable and viral abstraction than Unix files. They operate at a similar level of abstraction as files, but support richer content and encompass internet-wide interactions between autonomous systems. In a sense, HTTP GET and PUT have become the "subroutine calls" of an internet-scale programming system. Perhaps the most salient thing about the Web is that its usefulness came as such a surprise to everyone involved in designing or competing with it. It is likely that, by staying close to the existing practice of transferring files, the Web gained a competitive edge over more ambitious and less familiar hypertext projects like Xanadu (Nelson 1965).

The choice between compatibility and integrity correlates with the personality traits of *pragmatism* and *idealism*. It is pragmatic to accept the status quo of technology and make the best of it. Conversely, idealists are willing to fight convention and risk rejection in order to attain higher goals. We can wonder which came first: the design decision or the personality trait? Do Lisp and Haskell teach people to think more abstractly and coherently, or do they filter for those with a pre-existing condition? Likewise, we can provocatively speculate whether introverted developers prefer the cloisters of Smalltalk or Lisp to the adventurous "Wild West" of the Web.

E.3.4   *Dimension: composability*

In short, *you can get anywhere by putting together a number of smaller steps.* There exist building blocks which span a range of useful combinations. Composability is, in a sense, key to the notion of "programmability" and every programmable system will have some level of composability (e.g. in the scripting language.)

Unix shell commands are a standard example of composability. The base set of primitive commands can be augmented by programming command executables in other languages. Given some primitives, one can "pipe" one's output to another's input (`|`), sequence (`;` or `&&`), select via conditions, and repeat with loop constructs, enabling full imperative programming. Furthermore, command compositions can be packaged into a named "script" which follows the same interface as primitive commands, and named subprograms within a script can also be defined.

---

9 The `ioctl` interface is a way for userspace programs to communicate device-specific request codes to device drivers; for example, to eject a CD.

In *HyperCard*, the *Authoring Environment* is *non*-composable for programming buttons: there is simply a set of predefined behaviours to choose from. Full scriptability is available only in the *Programming Environment*.

The *Haskell type system*, as well as that of other functional programming languages, exhibits high composability. New types can be defined in terms of existing ones in several ways. These include records, discriminated unions, function types and recursive constructs (e.g. to define a `List` as either a `Nil` or a combination of element plus other list). The C programming language also has some means of composing types that are analogous in some ways, such as structs, unions, enums and indeed even function pointers. For every type, there is also a corresponding "pointer" type. It lacks, however, the recursive constructs permitted in Haskell types.

E.3.5 *Dimension: convenience*

In short, *you can get to X, Y or Z via one single step.* There are ready-made solutions to specific problems, not necessarily generalisable or composable. Convenience often manifests as "canonical" solutions and utilities in the form of an expansive standard library.

Composability without convenience is a set of atoms or gears; theoretically, anything one wants could be built out of them, but one must do that work. This situation has been criticised as the *Lisp Curse* (Winestock 2011).

Composability *with* convenience is a set of convenient specific tools *along with* enough components to construct new ones. The specific tools themselves could be transparently composed of these building blocks, but this is not essential. They save users the time and effort it would take to "roll their own" solutions to common tasks.

For example, let us turn to a convenience factor of *Unix* shell commands, having already discussed their composability above. Observe that it would be possible, in principle, to pass all information to a program via standard input. Yet in actual practice, for convenience, there is a standard interface of *command-line arguments* instead, separate from anything the program takes through standard input. Most programming systems similarly exhibit both composability and convenience, providing templates, standard libraries, or otherwise pre-packaged solutions, which can nevertheless be used programmatically as part of larger operations.

E.3.6 *Dimension: commonality*

Humans can see Arrays, Strings, Dicts and Sets all have a "size", but the software needs to be *told* that they are the "same". Commonality like this can be factored out into an explicit structure (a "Collection"

class), analogous to database *normalisation*. This way, an entity's size can be queried without reference to its particular details: if c is declared to be a Collection, then one can straightforwardly access `c.size`.

Alternatively, it can be left implicit. This is less upfront work, but permits instances to *diverge*, analogous to *redundancy* in databases. For example, Arrays and Strings might end up with "length", while Dict and Set call it "size". This means that, to query the size of an entity, it is necessary to perform a case split according to its concrete type, solely to funnel the diverging paths back to the commonality they represent:

```
if (entity is Array or String)  size := entity.length
else if (entity is Dict or Set) size := entity.size
```

E.3.7 *Examples: flattening and factoring of commonality*

Data structures usually have several "moving parts" that can vary independently. For example, a simple pair of "vehicle type" and "colour" might have all combinations of (Car, Van, Train) and (Red, Blue). In this *factored* representation, we can programmatically change the colour directly: `pair.second = Red` or `vehicle.colour = Red`.

In some contexts, such as class names, a system might only permit such multi-dimensional structure as an *exhaustive enumeration*: RedCar, BlueCar, RedVan, BlueVan, RedTrain, BlueTrain, etc. The system sees a flat list of atoms, even though a human can see the sub-structure encoded in the string. In this world, we cannot simply "change the colour to Red" programmatically; we would need to case-split as follows:

```
if (type is BlueCar) type := RedCar
else if (type is BlueVan) type := RedVan
else if (type is BlueTrain) type := RedTrain
...
```

The *commonality* between RedCar, RedVan, BlueCar, and so on has been *flattened*. There is implicit structure here that remains *un-factored*, similar to how numbers can be expressed as singular expressions (16) or as factor products (2,2,2,2). *Factoring* this commonality gives us the original design, where there is a pair of values from different sets.

In *relational databases*, there is an opposition between *normalisation* and *redundancy*. In order to fit multi-table data into a *flat* table structure, data needs to be duplicated into redundant copies. When data is *factored* into small tables as much as possible, such that there is only one place each piece of data "lives", the database is in *normal form* or *normalised*. Redundancy is useful for read-only processes, because there is no need to join different tables together based on common keys. Writing, however, becomes risky; in order to modify one thing, it must be synchronised to the multiple places it is stored. This makes highly normalised databases optimised for writes over reads.

E.3.8   *Remark: the end of history?*

Today we live in a highly developed world of software technology. It is estimated that 41,000 person years have been invested into Linux. We describe software development technologies in terms of *stacks* of specialised tools, each of which might capitalise over 100 person-years of development. Programming systems have become programming *eco*systems: not designed, but evolved. How can we noticeably improve programming in the face of the overwhelming edifice of existing technology? There are strong incentives to focus on localised incremental improvements that don't cross the established boundaries.

The history of computing is one of cycles of evolution and revolution. Successive cycles were dominated in turn by mainframes, minicomputers, workstations, personal computers, and the Web. Each transition built a whole new technology ecosystem replacing or on top of the previous. The last revolution, the Web, was 25 years ago, with the result that many people have never experienced a disruptive platform transition. Has history stopped, or are we just stuck in a long cycle, with increasingly pent-up pressures for change? If it is the latter, then incompatible ideas now spurned may yet flourish.

E.3.9   *References*

- How to Design a Good API and Why it Matters (Bloch 2007)

E.4   CUSTOMIZABILITY

> Once a program exists in the system, how can it be extended
> and modified?

Programming is a gradual process. We start either from nothing, or from an existing program, and gradually extend and refine it until it serves a given purpose. Programs created using different programming systems can be refined to different extents, in different ways, at different stages of their existence.

Consider three examples. First, a program in a conventional programming language like Java can be refined only by modifying its source code. However, you may be able to do so by just *adding* new code, such as a new interface implementation. Second, a spreadsheet can be modified at any time by modifying the formulas or data it contains. There is no separate programming phase. However, you have to modify the formulas directly in the cell—there is no way of modifying it by specifying a change in a way that is external to the cell. Third, a *self-sustaining* programming system, such as Smalltalk, does not make an explicit distinction between "programming" and "using" phases, and it can be modified and extended via itself. It gives developers the power to

experiment with the system and, in principle, replace it with a better system from within.

E.4.1   *Dimension: staging of customisation*

For systems that distinguish between different stages, such as writing source code versus running a program, customisation methods may be different for each stage. In traditional programming languages, customisation is done by modifying or adding source code at the programming stage, but there is no (automatically provided) way of customising the created programs once they are running.

There are a number of interesting questions related to staging of customisation. First, what is the notation used for customisation? This may be the notation in which a program was initially created, but a system may also use a secondary notation for customisation (consider Emacs using Emacs Lisp). For systems with a stage distinction, an important question is whether such changes are *persistent*.

Smalltalk, Interlisp and similar. In image-based programming systems, there is generally no strict distinction between stages and so a program can be customised during execution in the same way as during development. The program image includes the programming environment. Users of a program can open this, navigate to a suitable object or a class (which serve as the *addressable extension points*) and modify that. Lisp-based systems such as *Interlisp* follow a similar model. Changes made directly to the image are persistent. The PILOT system for Lisp (Teitelman 1966) offers an interactive way of correcting errors when a program fails during execution. Such corrections are then applied to the image and are thus persistent.

Document Object Model (DOM) and Webstrates: In the context of Web programming, there is traditionally a stage distinction between programming (writing the code and markup) and running (displaying a page). However, the DOM can also be modified by browser Developer Tools—either manually, by running scripts in a console, or by using a userscript manager such as Greasemonkey. Such changes are not persistent in the default browser state, but are made so by Webstrates (Klokmose et al. 2015) which synchronise the DOM between the server and the client. This makes the DOM collaborative, but not (automatically) *live* because of the complexities this implies for event handling.

E.4.2   *Dimension: addressing and externalisability*

Programs in all programming systems have a representation that may be exposed through notation such as source code. When customising a program, an interesting question is whether a customisation needs to

be done by modifying the original representation, or whether it can be done by *adding* something alongside the original structure.

In order to support customisation through addition, a programming system needs a number of characteristics introduced by Basman et al. (2016, 2018). First, the system needs to support *addressing*: the ability to refer to a part of the program representation from the outside. Next, *externalisability* means that a piece of addressed state can be exhaustively transferred between the system and the outside world. Finally, *additive authoring* requires that system behaviours can be *changed* by simply *adding* a new expression containing addresses—in other words, anything can be *overridden* without having to touch its specification. Of particular importance is how addresses are specified and what extension points in the program they can refer to. The system may offer an automatic mechanism that makes certain parts of a program addressable, or this task may be delegated to the programmer.

Cascading Style Sheets (*CSS*): CSS is a prime example of additive authoring within the Web programming system. It provides rich addressability mechanisms that are partly automatic (when referring to tag names) and partly manual (when using element IDs and class names). Given a web page, it is possible to modify almost any aspect of its appearance by simply *adding* additional rules to a CSS file. The Infusion project (Basman 2021) offers similar customisability mechanisms, but for behaviour rather than just styling. There is also the recent programming system Varv (Borowski et al. 2022), which embodies additive authoring as a core principle.

Object Oriented Programming (*OOP*) *and Aspect Oriented Programming* (*AOP*): in conventional programming languages, customisation is done by modifying the code itself. OOP and AOP make it possible to do so by adding code independently of existing program code. In OOP, this requires manual definition of extension points, i.e. interfaces and abstract methods. Functionality can then be added to a system by defining a new class (although injecting the new class into existing code without modification requires some form of configuration such as a dependency injection container). AOP systems such as AspectJ (Kiczales et al. 2001) provides a richer addressing mechanism. In particular, it makes it possible to add functionality to the invocation of a specific method (among other options) by using the *method call pointcut:* a specification for running code before, after, or around all calls to a given method, no matter how scattered around the code. This functionality is similar to *advising* in Pilot (Teitelman 1966).

E.4.3 *Concept: self-sustainability*

For most programming languages, programming systems, and ordinary software applications, if one wants to customise beyond a certain point, one must go beyond the facilities provided in the system itself.

Most programming systems maintain a clear distinction between the *user level*, where the system is used, and *implementation level*, where the source code of the system itself resides. If the user level does not expose control over some property or feature, then one is forced to go to the implementation level. In the common case this will be a completely different language or system, with an associated learning cost. It is also likely to be lower-level—lacking expressive functions, features or abstractions of the user level—which makes for a more tedious programming experience.

It is possible, however, to carefully design systems to expose deeper aspects of their implementation *at the user level*, relaxing the formerly strict division between these levels. For example, in the research system *3-LISP* (Smith 1982), ordinarily built-in functions like the conditional `if` and error handling `catch` are implemented in 3-LISP code at the user level.

The degree to which a system's inner workings are accessible to the user level, we call *self-sustainability*. At the maximal degree of this dimension would reside "stem cell"-like systems in a state of *zero commitment:* they could be progressively evolved to arbitrary behaviour without having to "step outside" of the system to a lower implementation level. In a sense, any difference between these systems would be merely a difference in initial state, since any could be turned into any other. Furthermore, such a system could be turned into any *existing* programming system by making commitments and thus *subtracting* flexibility.

The other end, of minimal self-sustainability, corresponds to minimal customisability: beyond the transient run-time state changes that make up the user level of any piece of software, the user cannot change anything without dropping down to the means of implementation of the system. This would resemble a traditional end-user "application" focused on a narrow domain with no means to do anything else.

The examples that we see of high self-sustainability all tend to be *Operating System-like*. Unix is widely established as an operating system, while Smalltalk and Lisp have been branded differently. Nevertheless, all three have shipped as the operating systems of custom hardware, and have similar responsibilities. Specifically: they support the execution of "programs"; they define an interface for accessing and modifying state; they provide standard libraries of common functionality; they define how programs can communicate with each other; they provide a user interface.

Unix: Self-sustainability of Unix is owed to the combination of two factors. First, the system is implemented in binary files (via ELF[10]) and text files (for configuration). Second, these files are part of the user-facing filesystem, so users can replace and modify parts of the system using Unix file interfaces.

---

10   Executable and Linkable Format.

Smalltalk and COLAs: Self-sustainability in Smalltalk is similar to Unix, but at a finer granularity and with less emphasis on whether things reside in volatile (process) or non-volatile (file) storage. The analogous points are that (1) the system is implemented as objects with methods containing Smalltalk code, and (2) these are modifiable using the class browser and code editor. Combined Object Lambda Architectures, or COLAs (Piumarta 2006), are a theoretical system design to improve on the self-sustainability of Smalltalk. This is achieved by generalising the object model to support relationships beyond classes.

We will illustrate the key dimensions of self-sustainability with the help of an existing programming system that is not self-sustainable. We cast the *Web browser* as the *product system* (i. e. that which we hypothetically wish to make self-sustainable) and C++ as the *platform* (i. e. the system used to implement it). What would it take to make the browser self-sustainable?

E.4.4   *Dimension: substrate size*

The *substrate* is the portion of a system not accessible from its user level. In the case of the web browser, it is the C++ code constituting its implementation. To get a self-sustainable system, the substrate must be minimised by shifting implementation out of it and into the programming capabilities of the product system. In this case, most of the named entities in the C++ code are stuck at the implementation level, inaccessible at the user level of JS, so we must move the former into the latter.

To sketch how this process could be carried out systematically, we can begin with the graphical surface of the product system. For each graphical element, we inquire into the causes of its display; this will include graphical rendering code, but also the data that is being rendered and the code that generated it. By tracing backwards in this way we discover the web of causes that produced the shape on the screen. This will often go through the user level (JS), but if we keep tracing back, we will hit the implementation level. Each time this happens, we port the code from the implementation level to the user level.

We continue this until it is no longer feasible; for example, there will ultimately have to be some native machine-code interpreter for JS in the running system. In practice, there would need to be the usual investments in JIT compilation and optimisation technology as seen in VMs for Smalltalk and other languages.

These ideas suggest a dimension of *substrate size* as a penalty for self-sustainability. In other words, a self-sustainable system *minimises* this dimension. A reasonable measure of substrate size could be the number of lines of code that implement it.

E.4.5   *Dimension: persistence effort*

In order for a system to be self-sustainable, it has to be able to preserve developments of its state through process termination. The standard VM solution is to have most of the system state saved in an "image" file and concentrate the substrate in a runnable binary that need not be changed. However this is accomplished, *persistence* of run-time changes is necessary to encourage indefinite evolution of the system. In the case of the Web browser, this applies to the whole browser, but could also be a concern for individual tabs or web pages that can be closed or refreshed.

This suggests another penalty dimension of *persistence effort.* To illustrate the range of values, we offer the following examples:

- Any system which automatically persists to an "image" (Lisp, Smalltalk) or otherwise (Webstrates; Klokmose et al. 2015) causes *minimal* persistence effort on the part of the user.[11]

- A system with a manual "save" button that persists all state would have almost-minimal persistence effort. This comprises both the need to remember to save and the act of pressing the button.

- A system where one must repeat a manual procedure over different parts of the state to persist all of it would have *moderate* persistence effort.

- A typical programming language in a "vanilla" state (e. g. excluding third-party libraries) has *high* persistence effort for its runtime data structures. With the use of a specific third-party library or framework (such as an Object-Relational Mapper) this persistence effort may be reduced. In the absence of such a framework, the programmer would have a lot of work to do in order to persist all state (wrap every variable and stack frame in code for loading and saving its value).

- We could ascribe *infinite*[12] persistence effort where it is impossible to persist state. This is easier to imagine in the case of an end-user application with no scripting capability; if the developers failed to persist something (e. g. the position or sizing of a window) then the user cannot do anything about it. In the case of a programming system, hard barriers to persistence include inaccessible state (e. g. in JS, one cannot refer to stack frames or read their state) or a lack of enumerability (e. g. there is no way to traverse all objects in the system and thereby persist them).

---

11 Arguably, this effort is zero, since the user does not have to think about it.

12 An infinite score can be interpreted as saying: it would take less effort to duplicate the source code of the system and add persistence at its implementation level, than it would to persist state using user-level functionality.

It may be objected that this measure should be considered on a piece-meal basis *per piece of state* instead of on the system as a whole. For example, a system could have infinite persistence effort with respect to some state (e. g. stack frames; Basman et al. 2016) but low persistence effort with respect to everything else (this being the effort invested to set up an Object-Relational Mapper for the rest of the state). Given such a fine-grained application of this measure and a method of weighting each contribution, one could derive a convenient aggregate measure of persistence effort for the whole system. However, this is too complicated for the scope of our work here, so we defer it to further development of the framework.

E.4.6 *Dimensions: code viewing and editing*

The browser's JS console makes it possible to make some changes expressible as JS commands, but some caveats would need mitigating. For example, the source code can be viewed but not edited; we would need to make a small change so that the source code viewer could also be used to make persistent edits to code. These points suggest *boolean* dimensions of *code viewing* and *code editing.* An example of a system that has both is Smalltalk with its class browser.

E.4.7 *Dimension: data execution*

Once we can type text inside the system, we will be able to write code. However, this code will be inert unless the system can interpret data structures as programs and actually execute them. This is the case whether these data structures were created manually or by a program. If this is not possible, re-programming the system will not be possible (beyond selecting from a predefined list of behaviours). The browser does already satisfy this criterion since JS has an `eval()` function that can execute a string of JS code. This suggests a boolean dimension of *data execution* (or Code As Data).

Any system with an `eval` function has this property, such as Lisp. In the low-level binary layer of programming, the fact that the *instruction pointer* can be pointed at bytes in memory and interpret them as instructions also qualifies. A negative example exists in a language like C, where there is no `eval` function. In such a case, one may employ the workaround of defining a mini-language (whether textual, or a binary bytecode) and an interpreter C function. It is important to be clear on which level the property would be thus established: what we called the *product system* (the program being implemented by the C code) would have data execution but the *platform* (the C language itself) would remain without it.

E.4.8 *References*

In addition to the examples discussed above, the proceedings of self-sustaining systems workshops (Rose and Hirschfeld 2008; Rose et al. 2010) provide numerous examples of systems and languages that are able to bootstrap, implement, modify, and maintain themselves; Gabriel's analysis of programming language revolutions (Gabriel 2012) uses *advising* in PILOT, related Lisp mechanisms, and "mixins" in OOP to illustrate the difference between the "languages" and "systems" paradigms.

E.4.9 *Relations*

- *Flattening and factoring* (Section E.3.7): related in that "customizability" is a form of creating new programs from existing ones; factoring repetitive aspects into a reusable standard component library facilitates the same thing.

- *Interaction* (Section E.1): this determines whether there are separate stages for running and writing programs and may thus influence what kind of customisation is possible.

E.5 COMPLEXITY

How does the system structure complexity and what level of detail is required?

There is a massive gap between the level of detail required by a computer, which executes a sequence of low-level instructions, and the human description of a program in higher-level terms. To bridge this gap, a programming system needs to deal with the complexity inherent in going from a high-level description to low-level instructions.

Ever since the 1940s, programmers have envisioned that "automatic programming" would allow higher-level programming. This did not necessarily mean full automation. In fact, the first "automatic programming" systems referred to higher-level programming languages with a compiler (or an interpreter) that expanded the high-level code into detailed instructions.

Most programming systems use *factoring of complexity* and encapsulate some of the details that need to be specified into components that can be reused by the programmer. The details may be encapsulated in a library, or filled in by a compiler or interpreter. Such factoring may also be reflected in the conceptual structure of the system (Section E.3.7). However, a system may also fully *automate* some aspects of programming. In those cases, a general-purpose algorithm solves a whole class of problems, which then do not need to be coded explicitly. Think

of planning the execution of SQL queries, or of the inference engine supporting a logic programming language like Prolog.

E.5.1 *Remark: notations*

Even when working at a high level, programming involves manipulating some program notation. In high-level functional or imperative programming languages, the programmer writes code that typically has clear operational meaning, even when some of the complexity is relegated to a library implementation or a runtime. When using declarative programming systems like SQL, Prolog or Datalog, the meaning of a program is still unambiguous, but it is not defined operationally—there is a (more or less deterministic) inference engine that solves the problem based on the provided description. Finally, systems based on *programming by example* step even further away from having clear operational meaning—the program may be simply a collection of sample inputs and outputs, from which a (possibly non-deterministic) engine infers the concrete steps of execution.

E.5.2 *Dimension: factoring of complexity*

The basic mechanism for dealing with complexity is *factoring* it. Given a program, the more domain-specific aspects of the logic are specified explicitly, whereas the more mundane and technical aspects of the logic are left to a reusable component. Often, this reusable component is just a library. Yet in the case of higher-level programming languages, the reusable component may include a part of a language runtime such as a memory allocator or a garbage collector. In the case of declarative languages or programming by example, the reusable component is a general purpose inference engine.

E.5.3 *Dimension: level of automation*

Factoring of complexity shields the programmer from some details, but those details still need to be explicitly programmed. Depending on the customizability of the system, this programming may or may not be accessible, but it is always there. For example, a function used in a spreadsheet formula is implemented in the spreadsheet system.

A programming system with higher *level of automation* requires more than simply factoring code into reusable components. It uses a mechanism where some details of the operational meaning of a program are never explicitly specified, but are inferred automatically by the system. This is the approach of *programming by example* and *machine learning*, where behaviour is specified through examples. In some cases, deciding whether a feature is *automation* or merely *factoring of complexity*

is less clear: garbage collection can be seen as either a simple case of automation, or a sophisticated case of factoring complexity.

There is also an interesting (and perhaps inevitable) trade-off. The higher the level of automation, the less explicit the operational meaning of a program. This has a wide range of implications. Smaragdakis (2019) notes, for example, that this means the implementation can significantly change the performance of a program.

e.5.4   *Example: domain-specific languages*

Domain-specific languages (Fowler 2010) provide an example of factoring of complexity that does not involve automation. In this case, programming is done at two levels. At the lower level, an (often more experienced) programmer develops a domain-specific language, which lets a (typically less experienced) programmer easily solve problems in a particular domain: say, modelling of financial contracts, or specifying interactive user interfaces.

The domain-specific language provides primitives that can be composed, but each primitive and each form of composition has explicitly programmed and unambiguous operational meaning. The user of the domain-specific language can think in the higher-level concepts it provides, and this conceptual structure can be analysed using the dimensions in Section E.3. As long as these concepts are clear, the user does not need to be concerned with the details of how exactly the resulting programs run.

e.5.5   *Example: programming by example*

An interesting case of automation is *programming by example* (Lieberman 2001). In this case, the user does not provide even a declarative specification of the program behaviour, but instead specifies sample inputs and outputs. A more or less sophisticated algorithm then attempts to infer the relationship between the inputs and the outputs. This may, for example, be done through program synthesis where an algorithm composes a transformation using a (small) number of pre-defined operations. Programming by example is often very accessible and has been used in spreadsheet applications (Gulwani et al. 2012).

e.5.6   *Example: next-level automation*

Throughout history, programmers have always hoped for the next level of "automatic programming". As observed by Parnas (1985), "automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer".

We may speculate whether Deep Learning will enable the next step of automation. However, this would not be different in principle from existing developments. We can see any level of automation as using *artificial intelligence* methods. This is the case for declarative languages or constraint-based languages—where the inference engine implements a traditional AI method (GOFAI, i.e., Good Old Fashioned AI).

E.5.7 *Relations*

- *Conceptual structure* (Section E.3): In many cases, the factoring of complexity follows the conceptual structure of the programming system.

- *Flattening and factoring* (Section E.3.7): One typically automates the thing at the lowest level in one's factoring (by making the lowest level a thing that exists outside of the program—in a system or a library)

E.6 ERRORS

What does the system consider to be an *error*? How are they prevented and handled?

A computer system is not aware of human intentions. There will always be human mistakes that the system cannot recognise as errors. Despite this, there are many that it *can* recognise, and its design will determine *which* human mistakes can become detectable program errors. This revolves around several questions: What can cause an error? Which ones can be prevented from happening? How should the system react to errors?

Following the standard literature on errors (Reason 1990), we distinguish four kinds of errors: slips, lapses, mistakes and failures. A *slip* is an error caused by transient human attention failure, such as a typo in the source code. A *lapse* is similar but caused by memory failure, such as an incorrectly remembered method name. A *mistake* is a logical error such as bad design of an algorithm. Finally, a *failure* is a system error caused by the system itself that the programmer has no control over, e.g. a hardware or a virtual machine failure.

E.6.1 *Dimensions: error detection*

Errors can be identified in any of the *feedback loops* that the system implements. This can be done either by a human or the system itself, depending on the nature of the feedback loop.

Consider three examples. First, in live programming systems, the programmer immediately sees the result of their code changes. Error

detection is done by a human and the system can assist this by visualising as many consequences of a code change as possible. Second, in a system with a static checking feedback loop (such as syntax checks or static type systems), potential errors are reported as the result of the analysis. Third, errors can be detected when the developed software is run, either when it is tested by the programmer (manually or through automated testing) or when it is run by a user.

Error detection in different feedback loops is suitable for detecting different kinds of errors. Many slips and lapses can be detected by the static checking feedback loop, although this is not always the case. For example, consider a "compact" *expression geography* where small changes in code may result in large changes of behaviour.[13] This makes it easier for slips and lapses to produce hard to detect errors; for example, a typo might coincidentally overlap with an existing name in scope, which would not trigger an expected "undeclared name" error. Mistakes are easier to detect through a live feedback loop, but they can also be partly detected by more advanced static checking.

E.6.2   *Example: static typing*

In statically typed programming languages like Haskell and Java, types are used to capture some information about the intent of the programmer. The type checker ensures code matches the lightweight specification given using types. In such systems, types and implementation serve as two descriptions of programmer's intent that need to align; what varies is the extent to which types can capture intent and the way in which the two are constructed; that is, which of the two comes first.

E.6.3   *Examples: TDD, REPL and live coding*

Whereas static typing aims to detect errors without executing code, approaches based on immediate feedback typically aim to execute (a portion of) the code and let the programmer see the error immediately. This can be done in a variety of ways.

In case of *test-driven development*, tests play the role of a specification (much like types) against which the implementation is checked. Such systems may provide more or less immediate feedback, depending on when tests are executed (automatically in the background, or manually). Systems equipped with a read-eval-print loop (REPL) let programmers run code on-the-fly and inspect results. For successful error detection, the results need to be easily observable: a printed output is more helpful than a hidden change of system state. Finally, in live coding systems, code is executed immediately and the programmer's ability to recognise

---

13 For example, the one-character difference between the Unix command `rm -rf ./*`, which recursively deletes all items in the current directory, and `rm -rf /*`, which does the same thing from the *root of the filesystem*.

errors depends on the extent to which the system state is observable. For example, in live-coded music (Aaron and Blackwell 2013), you *hear* that your code is not what you wanted, providing an easy-to-use immediate error detection mechanism.

E.6.4   *Remark: eliminating latent errors*

A common aim of error detection is to prevent *latent errors*, i.e. errors that occurred at some *earlier* point during execution, but only manifest themselves through an unexpected behaviour later on. For example, we might dereference the wrong memory address and store a junk value to a database; we will only find out upon accessing the database. Latent errors can be prevented differently in different feedback loops. In a live feedback loop, this can be done by visualising effects that would normally remain hidden. When running software, latent errors can be prevented through a mechanism that detects errors as early as possible (e.g. initialising pointers to null and stopping if they are dereferenced.)

Elm and time-travel debugging. One notable mechanism for identifying latent errors is the concept of *time-travel debugging* popularised by the Elm programming language. In time-travel debugging, the programmer is able to step back through time and see what execution steps were taken prior to a certain point. This makes it possible to break execution when a latent error manifests, but then retrace the execution back to the actual source of the error.

E.6.5   *Dimension: error response*

When an error is detected, there are a number of typical ways in which the system can respond. The following applies to systems that provide some kind of error detection during execution.

- It may attempt to automatically recover from the error as best as possible. This may be feasible for simpler errors (slips and lapses), but also for certain mistakes (a mistake in an algorithm's concurrency logic may often be resolved by restarting the code).

- It may proceed as if the error did not happen. This can eliminate expensive checks, but may lead to latent errors later.

- It may ask a human how to resolve the issue. This can be done interactively, by entering into a mode where the code can be corrected, or non-interactively by stopping the system.

Orthogonally to the above options, a system may also have a way to recover from latent errors by tracing back through the execution in order to find the root cause. It may also have a mechanism for undoing all actions that occurred in the meantime, e.g. through transactional processing.

Interlisp and Do What I Mean (DWIM). Interlisp's DWIM facility attempts to automatically correct slips and lapses, especially misspellings and unbalanced parentheses. When Interlisp encounters an error, such as a reference to an undefined symbol, it invokes DWIM. In this case, DWIM then searches for similarly named symbols frequently used by the current user. If it finds one, it invokes the symbol automatically, corrects the source code and notifies the user. In more complex cases where DWIM cannot correct the error automatically, it starts an interaction with the user and lets them correct it manually.

LATEX. When a run of LATEX encounters an error, it prompts the user with several options. One of them is to ignore the error and proceed, in the hopes that it is inconsequential. Another is to edit the input stream at the location of the error and resume. If all else fails, the run can be halted. These options echo the "Abort, Retry, Ignore" dialog from MS-DOS.

E.6.6   *Relations*

- *Feedback loops*: Error detection always happens as part of an individual feedback loop. The feedback loops thus determine the structure in which error detection can happen.

- *Automation:* A semi-automatic error recovery system (such as DWIM) implements a form of automation. The concept of antifragile software (Monperrus 2017) is a more sophisticated example of error recovery through automation.

- *Expression geography:* In an expression geography where small changes in notation lead to valid but differently behaved programs, a slip or lapse is more likely to lead to an error that is difficult to detect through standard mechanisms.

E.6.7   *References*

The most common error handling mechanism in conventional programming languages is exception handling. The modern form of exception handling has been described by Goodenough (1975), while Ryder et al. (2005) documents the history and influences of Software Engineering on exception handling. The concept of *antifragile software* (Monperrus 2017) goes further by suggesting that software could improve in response to errors. Work on Chaos Engineering (Chang et al. 2015) is a step in this direction.

Reason (1990) analyses errors in the context of human errors and develops a classification of errors that we adopt. In the context of computing, errors and *miscomputation* have been analysed from a philosophical perspective (Floridi et al. 2015; Fresco and Primiero 2013). Notably,

attitudes and approaches to errors also differ for different programming subcultures (Petricek 2017).

E.7   ADOPTABILITY

> How does the system facilitate or obstruct adoption by both individuals and communities?

We consider adoption by individuals as the dimension of *Learnability*, and adoption by communities as the dimension of *Sociability*.

E.7.1   *Dimension: learnability*

Mainstream software development technologies require substantial effort to learn. Systems can be made easier to learn in several ways:

- Specialising to a specific application domain.

- Specialising to simple small-scale needs.

- Leveraging the background knowledge, skills, and terminologies of specific communities.

- Supporting learning with staged levels of complexity and assistive development tools (Fry 1997). Better *Feedback Loops* can help (Section E.1).

- Collapsing heterogeneous technology stacks into simpler unified systems. This relates to the dimensions under *Conceptual Structure* (Section E.3).

FORTRAN was a breakthrough in programming because it specialised to scientific computing and leveraged the background knowledge of scientists about mathematical formulas. COBOL instead specialised to business data processing and embraced the business community by eschewing mathematics in favor of plain English.

LOGO was the first language explicitly designed for teaching children. Later BASIC and Pascal were designed for teaching then-standard programming concepts at the University level. BASIC and Pascal had second careers on micropocessors in the 90's. These microprocessor programming systems were notable for being complete solutions integrating everything necessary, and so became home schools for a generation of programmers. More recently languages like Racket (Felleisen et al. 2015), Pyret (Pyret Crew 2016), and Grace (Black et al. 2012) have supported learning by revealing progressive levels of complexity in stages. Scratch returned to Logo's vision of teaching children with a graphical programming environment emphasising playfulness rather than generality.

Some programming languages have consciously prioritised the programmer's experience of learning and using them. Ruby calls itself *a programmer's best friend* by focusing on simplicity and elegance. Elm (Czaplicki and Chong 2013) targets the more specialised but still fairly broad domain of web applications while focusing on simplicity and programmer-friendliness. It forgoes capabilities that would lead to run-time crashes. It also tries hard to make error messages clear and actionable.

If we look beyond programming languages *per se*, we find programmable systems with better learnability. The best example is spreadsheets, which offer a specialised computing environment that is simpler and more intuitive. The visual metaphor of a grid leverages human perceptual skills. Moving all programming into declarative formulas and attributes greatly simplifies both creation and understanding. Research on Live Programming (Hancock and Resnick 2003; Victor 2012) has sought to incorporate these benefits into general purpose programming, but with limited success to date.

HyperCard and Flash were both programming systems that found widespread adoption by non-experts. Like spreadsheets they had an organising visual metaphor (cards and timelines respectively). They both made it easy for beginners to get started. Hypercard had layers of complexity intended to facilitate gradual mastery.

Smalltalk and Lisp machines were complex but unified. After overcoming the initial learning curve, their environments provided a complete solution for building entire application systems of arbitrary complexity without having to learn other technologies. Boxer (diSessa 1985) is notable for providing a general-purpose programming environment—albeit for small-scale applications—along with an organising visual metaphor like that of spreadsheets.

E.7.2   *Dimension: sociability*

Over time, especially in the internet era, social issues have come to dominate programming. Much programming technology is now developed by open-source communities, and all programming technologies are now embedded in social media communities of their users. Therefore, technical decisions that impact sociabilty can be decisive (Meyerovich and Rabkin 2012). These include:

- Compatibility: easy integration into standard technology stacks, allowing incremental adoption, and also easy exit if needed. This dynamic was discussed in the classic essay *Worse is Better* (Gabriel 1991) about how Unix beat Lisp.

- Developing with an open source methodology reaps volunteer labor and fosters a user community of enthusiasts. The technical advantages of open source development were first popularised in the essay *The Cathedral and the Bazaar* (Raymond and Young 2001),

which observed that "given enough eyeballs, all bugs are shallow". Open source has become the standard for software development tools, even those developed within large corporations.

- Easy sharing of code via package repositories or open exchanges. Prior to the open-source era, commercial marketplaces were important, like VBX components for VisualBasic. Sharing is impeded when languages lack standard libraries, leading to competing dialects like Scheme (Winestock 2011).

- Dedicated social media communities can be fostered by using them to provide technical support. Volunteer technical support, like volunteer code contributions, can multiply the impact of core developers. In some cases, social media like Stack Exchange has even come to replace documentation.

One could argue that sociabilty is not purely a *technical* dimension, as it includes aspects of product management. Rather, we believe that sociability is a pervasive cross-cutting concern that cannot be *separated* from the technical.

The tenor of the online community around a programming system can be its most public attribute. Even before social media, Flash developed a vibrant community of amateurs sharing code and tips. The Elm language invested much effort in creating a welcoming community from the outset (Czaplicki 2018). Attempts to reform older communities have introduced Codes of Conduct, but not without controversy.

On the other hand, a cloistered community that turns its back on the wider world can give its members strong feelings of belonging and purpose. Examples are Smalltalk, Racket, Clojure, and Haskell. These cohesive communities believe in the philosophy of their leaders and strive to follow it in the code they write.

The economic sustainability of a programming system can be even more important than strictly social and technical issues. Adopting a technology is a costly investment in terms of time, money, and foregone opportunities. Everyone feels safer investing in a technology backed by large corporations that are not going away, or in technologies that have such widespread adoption that they are guaranteed to persist. A vibrant and mature open-source community backing a technology also makes it safer.

Unfortunately, sociability is often in conflict with learnability. Compatibility leads to ever increasing historical baggage for new learners to master. Large internet corporations have invested mainly in technologies relevant to their expert staff and high-end needs. Open-source communities have mainly flourished around technologies for expert programmers "scratching their own itch". While there has been a flow of venture funding into "no-code" and "low-code" programming systems, it is not clear how they can become economically and socially
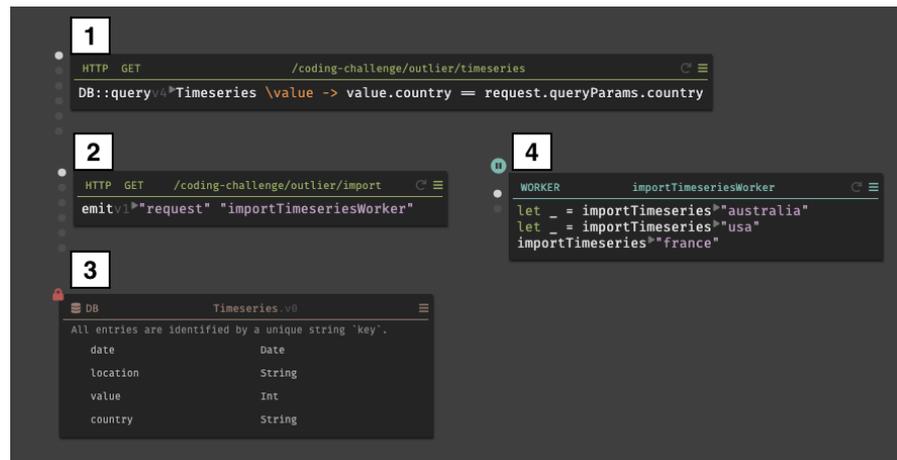
Figure E.2: A simple web service in Dark consisting of two HTTP endpoints (1, 2), a database (3), and a worker (4).

sustainable. By and large, the internet era has seen the ascendancy of expert programmers and the eclipsing of programming systems for "the rest of us".

E.8   EVALUATING THE DARK PROGRAMMING SYSTEM

This section demonstrates using the framework to analyse the recent programming system *Dark* (Dark Language Team 2022), explaining how it relates to past work and how it contributes to the state of the art.

Dark is a programming system for building "serverless backends", i.e. services that are used by web and mobile applications. It aims to make building such services easier by "removing accidental complexity"[14] resulting from the large number of systems typically involved in their deployment and operation. This includes infrastructure for orchestration, scaling, logging, monitoring and versioning. Dark provides integrated tooling (Figure E.2) for development and is described as *deployless*, meaning that deploying code to production is instantaneous.

Dark illustrates the need for the broader perspective of programming systems. Of course, it contains a programming language, which is inspired by OCaml and F#. But Dark's distinguishing feature is that it eliminates the many secondary systems needed for deployment of modern cloud-based services. Those exist outside of a typical programming language, yet form a major part of the complexity of the overall development process.

With technical dimensions, we can go beyond the "sales pitch", look behind the scenes, and better understand the interesting technical aspects of Dark as a programming system. Tables E.1 and E.2 summarise the more detailed analysis that follows.

---

14 https://roadmap.darklang.com/goals-of-dark-v2.html

Table E.1: Summary of where Dark lies on some of the dimensions.

Dimension (*CLUSTER*)	Summary
INTERACTION	
Modes of Interaction	Single integrated mode comprises development, debugging and operation ("deployless")
Feedback Loops	Code editing is triggered either by user or by unsupported HTTP request and changes are deployed automatically, allowing for *immediate feedback*
ERRORS	
Error Response	When an unsupported HTTP request is received, programmer can write handler code using data from the request in the process
CONCEPTUAL STRUCTURE	
Conceptual Integrity vs. Openness	Abstractions at the domain specific high-level and the functional low-level are both carefully designed for conceptual integrity.
Composability	User applications are composed from high-level primitives; the low-level uses composable functional abstractions (records, pipelines).
Convenience	Powerful high-level domain-specific abstractions are provided (HTTP, database, workers); core functional libraries exist for the low-level.
ADOPTABILITY	
Learnability	High-level concepts will be immediately familiar to the target audience; low-level language has the usual learning curve of basic functional programming

Table E.2: Summary of where Dark lies on some of the dimensions.

Dimension (*CLUSTER*)	Summary
NOTATION	
Notational Structure	Graphical notation for high-level concepts is complemented by structure editor for low-level code
Uniformity	Common notational structures are used for database and code, enabling the same editing construct for sequential structures (records, pipelines, tables)
COMPLEXITY	
Factoring of Complexity	Cloud infrastructure (deployment, orchestration, etc.) is provided by the Dark platform that is invisible to the programmer, but also cannot be modified
Level of Automation	Current implementation provides basic infrastructure, but a higher degree of automation in the platform can be provided in the future, e.g. for scalability
CUSTOMISABILITY	
Staging of Customisation	System can be modified while running and changes are persisted, but they have to be made in the Dark editor, which is distinct from the running service
Self-Sustainability	Clear separation between user and implementation levels; system is for making web services, not open-ended exploration of programming.

E.8.1   *Dimensional analysis of Dark*

MODES OF INTERACTION AND FEEDBACK LOOPS.   Conventional *modes of interaction* (E.1.4) include running, editing and debugging. For modern web services, running refers to operation in a cloud-based environment that typically comes with further kinds of feedback (logging and monitoring). The key design decision of Dark is to integrate all these different modes of interaction into a single one. This tight integration allows Dark to provide a more immediate *feedback loop* (E.1.1) where code changes become immediately available not just to the developer, but also to external users. The integrated mode of interaction is reminiscent of the image-based environment in Smalltalk; Dark advances the state of art by using this model in a multi-user, cloud-based context.

FEEDBACK LOOPS AND ERROR RESPONSE.   The integration of development and operation also makes it possible to use *errors* occurring during operation to drive development. Specifically, when a Dark service receives a request that is not supported, the user can build a handler (Chisa 2020) to provide a response—taking advantage of the live data that was sent as part of the request. In terms of our dimensions, this is a kind of *error response* (Section E.6.5) that was pioneered by the PILOT system for Lisp (Teitelman 1966). Dark does this not just to respond to errors, but also as the primary development mechanism, which we might call *Error-Driven Development.* This way, Dark users can construct programs with respect to sample input values.

CONCEPTUAL STRUCTURE AND LEARNABILITY.   Dark programs are expressed using high-level concepts that are specific to the domain of server-side web programming: HTTP request handlers, databases, workers and scheduled jobs. These are designed to reduce accidental complexity and aim for high *conceptual integrity* (Section E.3.1). At the level of code, Dark uses a general-purpose functional language that emphasises certain concepts, especially records and pipelines. The high-level concepts contribute to *learnability* (Section E.7.1) of the system, because they are highly domain-specific and will already be familiar to its intended users.

NOTATIONAL STRUCTURE AND UNIFORMITY.   Dark uses a combination of graphical editor and code. The two aspects of the notation follow the *complementing notations* (Section E.2.1) pattern. The windowed interface is used to work with the high-level concepts and code is used for working with low-level concepts. At the high level, code is structured in freely positionable boxes on a 2D surface. Unlike Boxer (diSessa and Abelson 1986), these boxes do not nest and the space cannot be used for other content (say, for comments, architectural illustrations or other media). Code at the low level is manipulated using a syntax-aware

structure editor (high *explicit structure,* Section E.1.6) showing inferred types and computed live values for pure functions. It also provides special editing support for records and pipelines, allowing users to add fields and steps respectively.

FACTORING OF COMPLEXITY AND AUTOMATION.   One of the advertised goals of Dark is to remove accidental complexity. This is achieved by collapsing the heterogeneous stack of technologies that are typically required for development, cloud deployment, orchestration and operation. Dark hides this via *factoring of complexity* (Section E.5.2). The advanced infrastructure is provided by the Dark platform and is hidden from the user. The infrastructure is programmed explicitly and there is no need for sophisticated automation (Section E.5.3). This factoring of functionality that was previously coded manually follows a similar pattern to the development of garbage collection in high-level programming languages.

CUSTOMISABILITY.   The Dark platform makes a clear distinction between the platform itself and the user application, so *self-sustainability* (Section E.4.3) is not an objective. The strict division between the platform and user (related to its aforementioned *factoring of complexity*) means that changes to Dark require modifying the platform source code itself, which is available under a license that solely allows using it for the purpose of contributing. Similarly, applications themselves are developed by modifying and adding code, requiring destructive access to it—so *additive authoring* (Section E.4.2) is not exhibited at either level. Thanks to the integration of execution and development, persistent changes may be made during execution (c.f. *staging of customisation*, Section E.4.1) but this is done through the Dark editor, which is separate from the running service.

E.8.2   *Technical Innovations of Dark*

This analysis reveals a number of interesting aspects of the Dark programming system. The first is the tight integration of different *modes of interaction* which collapses a heterogeneous stack of technologies, makes Dark *learnable*, and allows quick feedback from deployed services. The second is the use of *error response* to guide the development of HTTP handlers. Thanks to the technical dimensions framework, each of these can be more precisely described. It is also possible to see how they may be supported in other programming systems. The framework also points to possible alternatives (and perhaps improvements) such as building a more self-sustainable system that has similar characteristics to Dark, but allows greater flexibility in modifying the platform from within itself.

BIBLIOGRAPHY

Aaron, Samuel and Alan F. Blackwell (2013). "From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages." In: *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*. FARM '13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 35–46. ISBN: 9781450323864. DOI: 10.1145/2505341.2505346. URL: https://doi.org/10.1145/2505341.2505346.

Agaram, Kartik (2020). "Bicycles for the Mind Have to Be See-Through." In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. <Programming> '20. Porto, Portugal: Association for Computing Machinery, pp. 173–186. ISBN: 9781450375078. DOI: 10.1145/3397537.3397547. URL: https://doi.org/10.1145/3397537.3397547.

Aish, Robert (2012). "DesignScript: Origins, Explanation, Illustration." In: *Computational Design Modelling*. Ed. by Christoph Gengnagel, Axel Kilian, Norbert Palz, and Fabian Scheurer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–8. ISBN: 978-3-642-23435-4.

Aleph One (1996). "Smashing The Stack for Fun and Profit." In: *Phrack magazine* 7.49, pp. 14–16.

Amelang, Dan (2012a). *Gezira*. URL: https://github.com/damelang/gezira.

– (2012b). *The Nile Programming Language. Declarative Stream Processing for Media Applications*. URL: https://github.com/damelang/nile.

Amelang, Dan et al. (2011). *STEPS Toward Expressive Programming Systems, 2011 Progress Report*. URL: http://www.vpri.org/pdf/tr2011004_steps11.pdf.

– (2012). *STEPS Toward the Reinvention of Programming, 2012 Final Report*. URL: http://www.vpri.org/pdf/tr2012001_steps.pdf.

Ankerson, Megan Sapnar (2018). *Dot-Com Design: The Rise of a Usable, Social, Commercial Web*. NYU Press. ISBN: 1479892904.

Babaoğlu, Özalp, William Joy, and Juan Porcar (1979). "Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX Operating System." In: *Department of Electrical Engineering and Computer Science, University of California, Berkeley*.

Basman, Antranig (2016). "Building Software is Not (Yet) a Craft." In: *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2016, Cambridge, UK, September 7-10, 2016*. Ed. by Luke Church. Psychology of Programming Interest Group, p. 32. URL: http://ppig.org/library/paper/building-software-not-yet-craft.

Basman, Antranig (2021). *Infusion Framework and Components*. URL: `https://fluidproject.org/infusion.html`.

Basman, Antranig, L. Church, C. Klokmose, and Colin B. D. Clark (2016). "Software and How it Lives On: Embedding Live Programs in the World Around Them." In: *PPIG*. URL: `http://www.klokmose.net/clemens/wp-content/uploads/2016/10/ppig-2016.pdf`.

Basman, Antranig, Clayton Lewis, and Colin Clark (2018). "The Open Authorial Principle: Supporting Networks of Authors in Creating Externalisable Designs." In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 29–43.

Bergel, Alexandre, Damien Cassou, Stéphane Ducasse, and Jannik Laval (2013). *Deep into Pharo*. en. Morrisville, NC: Lulu.com.

Bird, Richard and Oege de Moor (1997). *Algebra of programming*. USA: Prentice-Hall, Inc. ISBN: 013507245X.

Black, Andrew P., Kim B. Bruce, Michael Homer, and James Noble (2012). "Grace: the absence of (inessential) difficulty." In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012. Tucson, Arizona, USA: Association for Computing Machinery, pp. 85–98. ISBN: 9781450315623. DOI: `10.1145/2384592.2384601`. URL: `https://doi.org/10.1145/2384592.2384601`.

Bloch, Joshua (2007). *How to Design a Good API and Why it Matters*. URL: `http://www.cs.bc.edu/~muller/teaching/cs102/s06/lib/pdf/api-design`.

Borning, Alan (Oct. 1981). "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory." In: *ACM Trans. Program. Lang. Syst.* 3.4, pp. 353–387. ISSN: 0164-0925. DOI: `10.1145/357146.357147`. URL: `https://doi.org/10.1145/357146.357147`.

Borowski, Marcel, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose (2022). "Varv: Reprogrammable Interactive Software as a Declarative Data Structure." In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery. ISBN: 9781450391573. DOI: `10.1145/3491102.3502064`. URL: `https://doi.org/10.1145/3491102.3502064`.

Bracha, Gilad and David Ungar (Oct. 2004). "Mirrors: design principles for meta-level facilities of object-oriented programming languages." In: *SIGPLAN Not.* 39.10, pp. 331–344. ISSN: 0362-1340. DOI: `10.1145/1035292.1029004`. URL: `https://doi.org/10.1145/1035292.1029004`.

Brooks, FP (1995). "Aristocracy, Democracy and System Design." In: *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley.

Brooks, Frederick P. (1978). *The Mythical Man-Month: Essays on Softw*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201006502.

Bystroushaak, Anon (2019). *Environment and the Programming Language Self (part one; environment)*. URL: https://blog.rfox.eu/en/Programming/Series_about_Self/Environment_and_the_programming_language_Self_part_one_environment.html.

C2 Contributors (2012). *Meta-circular Evaluator*. URL: https://wiki.c2.com/?MetaCircularEvaluator.

– (2014a). *Greenspun's Tenth Rule*. URL: http://wiki.c2.com/?GreenspunsTenthRuleOfProgramming.

– (2014b). *Masp Brainstorming*. URL: https://wiki.c2.com/?MaspBrainstorming.

– (2014c). *Pick The Right Tool For The Job*. URL: https://wiki.c2.com/?PickTheRightToolForTheJob.

Chang, Hasok (2004). *Inventing temperature: Measurement and scientific progress*. Oxford: Oxford University Press.

Chang, Michael Alan, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever (2015). "Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction." In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIG-COMM '15. London, United Kingdom: Association for Computing Machinery, pp. 371–372. ISBN: 9781450335423. DOI: 10.1145/2785956.2790038. URL: https://doi.org/10.1145/2785956.2790038.

Chiş, Andrei, Oscar Nierstrasz, and Tudor Gîrba (2015). "Towards moldable development tools." In: *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU 2015. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 25–26. ISBN: 9781450339070. DOI: 10.1145/2846680.2846684. URL: https://doi.org/10.1145/2846680.2846684.

Chisa, Ellen (2020). *Introduction: Error Rail and Match with `DB::get`*. URL: https://youtu.be/NRMmy9ZzA-o.

Chuchem, Yair (2023). *Structural Code Editor Projects*. URL: https://github.com/yairchu/awesome-structure-editors/.

Coda (May 23, 2022). *Coda: The doc that brings it all together*. URL: https://coda.io.

Colegrove, Tim (2020). *Own work, CC BY-SA 4.0*. URL: https://commons.wikimedia.org/w/index.php?curid=89430810.

Cook, Dan (2018). *Self-Defined Object System*. URL: https://www.cemetech.net/forum/viewtopic.php?p=270092#270092.

Cypher, Allen, ed. (1993). *Watch What I Do. Programming by Demonstration*. The MIT Press. URL: http://acypher.com/wwid/.

Czaplicki, Evan (2018). URL: https://www.youtube.com/watch?v=uGlzRt-FYto.

Czaplicki, Evan and Stephen Chong (2013). "Asynchronous functional reactive programming for GUIs." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, pp. 411–422. ISBN: 9781450320146. DOI: 10.1145/2491956.2462161. URL: https://doi.org/10.1145/2491956.2462161.

Dark Language Team (May 23, 2022). *Dark Lang*. URL: https://darklang.com.

desRivieres, J. and J. Wiegand (2004). "Eclipse: A platform for integrating development tools." In: *IBM Systems Journal* 43.2, pp. 371–383. DOI: 10.1147/sj.432.0371.

Diekmann, Lukas and Laurence Tratt (Sept. 2014). "Eco: A language composition editor." In: *Software Language Engineering (SLE)*. Springer, pp. 82–101. DOI: 10.1007/978-3-319-11245-9_5. URL: https://soft-dev.org/pubs/html/diekmann_tratt__eco_a_language_composition_editor/.

diSessa, Andrea A. (1985). "A Principled Design for an Integrated Computational Environment." In: *Human–Computer Interaction* 1.1, pp. 1–47. DOI: 10.1207/s15327051hci0101_1. URL: https://doi.org/10.1207/s15327051hci0101_1.

diSessa, Andrea A. and H. Abelson (Sept. 1986). "Boxer: A Reconstructible Computational Medium." In: *Commun. ACM* 29.9, pp. 859–868. ISSN: 0001-0782. DOI: 10.1145/6592.6595. URL: https://doi.org/10.1145/6592.6595.

Edwards, Jonathan (2005). "Subtext: Uncovering the Simplicity of Programming." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, pp. 505–518. ISBN: 1595930310. DOI: 10.1145/1094811.1094851. URL: https://doi.org/10.1145/1094811.1094851.

– (2017). *Gallery of Programming UIs*. URL: https://alarmingdevelopment.org/?p=1068.

Edwards, Jonathan, Stephen Kell, Tomas Petricek, and Luke Church (2019a). "Evaluating programming systems design." In: *Proceedings of 30th Annual Workshop of Psychology of Programming Interest Group*. PPIG 2019. Newcastle, UK.

– (2019b). "Evaluating programming systems design." In: *Proceedings of 30th Annual Workshop of Psychology of Programming Interest Group*. PPIG 2019. Newcastle, UK.

Ehrlich, Justin (2013). "The Component Entity System for Virtual Environments." In: *Proceedings of the International Conference on Computer Graphics and Virtual Reality CGVR'13*. Ed. by Hamid R. Arabnia, Leonidas Deligiannidis, and Ashu M. G. Solo. URL: http://worldcomp-proceedings.com/proc/p2013/CGV3033.pdf.

Evans, Edmund Grimley (2001). *Bootstrapping A Simple Compiler From Nothing*. URL: https://web.archive.org/web/20061108010907/http://www.rano.org/bcompiler.html.

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt (2015). "The Racket Manifesto." In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Mor-

riset. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 113–128. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.113. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2015.113.

– (Feb. 2018). "A Programmable Programming Language." In: *Commun. ACM* 61.3, pp. 62–71. ISSN: 0001-0782. DOI: 10.1145/3127323. URL: https://doi.org/10.1145/3127323.

Floridi, Luciano, Nir Fresco, and Giuseppe Primiero (2015). "On malfunctioning software." In: *Synthese* 192.4, pp. 1199–1220.

Foderaro, John (Sept. 1991). "LISP: Introduction." In: *Commun. ACM* 34.9, p. 27. ISSN: 0001-0782. DOI: 10.1145/114669.114670. URL: https://doi.org/10.1145/114669.114670.

Ford, Neal (2006). *Polyglot Programming*. URL: http://memeagora.blogspot.com/2006/12/polyglot-programming.html.

Fowler, Martin (2005). *Event Sourcing*. URL: https://martinfowler.com/eaaDev/EventSourcing.html.

– (2010). *Domain-specific languages*. Pearson Education.

Fresco, Nir and Giuseppe Primiero (2013). "Miscomputation." In: *Philosophy & Technology* 26.3, pp. 253–272.

Fry, Christopher (Apr. 1997). "Programming on an Already Full Brain." In: *Commun. ACM* 40.4, pp. 55–64. ISSN: 0001-0782. DOI: 10.1145/248448.248459. URL: https://doi.org/10.1145/248448.248459.

Fuller, Matthew et al. (2008). *Software studies: A lexicon*. Mit Press.

Gabriel, Richard P. (1991). *Worse Is Better*. URL: https://www.dreamsongs.com/WorseIsBetter.html.

– (2008). "Designed as Designer." In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: Association for Computing Machinery, pp. 617–632. ISBN: 9781605582153. DOI: 10.1145/1449764.1449813. URL: https://doi.org/10.1145/1449764.1449813.

– (2012). "The Structure of a Programming Language Revolution." In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012. Tucson, Arizona, USA: Association for Computing Machinery, pp. 195–214. ISBN: 9781450315623. DOI: 10.1145/2384592.2384611. URL: https://doi.org/10.1145/2384592.2384611.

Gamma, Erich, Richard Helm, Ralph E. Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass: Addison-Wesley.

Geisler, Dietrich, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson (Nov. 2020). "Geometry types for graphics programming." In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428241. URL: https://doi.org/10.1145/3428241.

Glide (May 23, 2022). *Glide: Create apps and websites without code*. URL: https://www.glideapps.com.

Goodenough, John B. (1975). "Exception Handling: Issues and a Proposed Notation." In: *Commun. ACM* 18.12, pp. 683–696. DOI: 10.1145/361227.361230. URL: https://doi.org/10.1145/361227.361230.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha (2000). *The Java language specification*. Addison-Wesley Professional.

Grad, Burton (2007). "The Creation and the Demise of VisiCalc." In: *IEEE Annals of the History of Computing* 29.3, pp. 20–31. DOI: 10.1109/MAHC.2007.4338439.

Green, T. R. G. and M. Petre (1996). "Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework." In: *JOURNAL OF VISUAL LANGUAGES AND COMPUTING* 7, pp. 131–174.

Gulwani, Sumit, William R Harris, and Rishabh Singh (2012). "Spreadsheet data manipulation using examples." In: *Communications of the ACM* 55.8, pp. 97–105.

Hague, James (2010). *Living Inside Your Own Black Box*. URL: https://prog21.dadgum.com/66.html.

Hall, Christopher K. (2017). "A New Human-Readability Infrastructure for Computing." PhD thesis. Santa Barbara, CA: University of California. URL: http://www.christopherkhall.com/Dissertation.pdf.

Hancock, C. and M. Resnick (2003). "Real-time programming and the big ideas of computational literacy." PhD thesis. Massachusetts Institute of Technology. URL: https://dspace.mit.edu/handle/1721.1/61549.

Hempel, Brian (2018). *Evaluating User Interface Systems Research (Summary / Cheat Sheet)*. URL: https://people.cs.uchicago.edu/~brianhempel/Evaluating%20User%20Interface%20Systems%20Research%20-%20Graphical%20Summary.pdf.

Hempel, Brian and Ravi Chugh (2020). "Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)." In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–5. DOI: 10.1109/VL/HCC50065.2020.9127256.

Hempel, Brian and Sam Lau (2021). *LIVE Workshop 2021*. URL: https://liveprog.org/live-2021/.

Hempel, Brian, Justin Lubin, and Ravi Chugh (2019). "Sketch-n-Sketch: Output-Directed Programming for SVG." In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: Association for Computing Machinery, pp. 281–292. ISBN: 9781450368162. DOI: 10.1145/3332165.3347925. URL: https://doi.org/10.1145/3332165.3347925.

Hempel, Brian and Roly Perera (2020). *LIVE Workshop 2020*. URL: https://liveprog.org/live-2020/.

Ingalls, Dan (2008). "The Lively Kernel: just for fun, let's take JavaScript seriously." In: *Proceedings of the 2008 Symposium on Dynamic Languages*.

DLS '08. Paphos, Cyprus: Association for Computing Machinery. ISBN: 9781605582702. DOI: 10.1145/1408681.1408690. URL: https://doi.org/10.1145/1408681.1408690.

Ingalls, Dan, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay (1997). "Back to the future: the story of Squeak, a practical Smalltalk written in itself." In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '97. Atlanta, Georgia, USA: Association for Computing Machinery, pp. 318–326. ISBN: 0897919084. DOI: 10.1145/263698.263754. URL: https://doi.org/10.1145/263698.263754.

Ingalls, Daniel (1981). *Design Principles Behind Smalltalk*. URL: https://archive.org/details/byte-magazine-1981-08/page/n299/mode/2up.

Jakubovic, Joel (Dec. 2020a). "Programming "systems" deserve a theory too!" In: *Psychology of Programming Interest Group*.

– (2020b). "What It Takes to Create with Domain-Appropriate Tools. Reflections on implementing the "Id" system." In: *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. Programming '20. Porto, Portugal: Association for Computing Machinery, pp. 197–207. ISBN: 9781450375078. DOI: 10.1145/3397537.3397549.

– (2021). *Primitive Instructions for Nested Trees*. URL: https://programmingmadecomplicated.wordpress.com/2021/11/03/primitive-instructions-for-nested-trees/.

Jakubovic, Joel, Jonathan Edwards, and Tomas Petricek (Feb. 2023). "Technical Dimensions of Programming Systems." In: *The Art, Science, and Engineering of Programming* 7.3. DOI: 10.22152/programming-journal.org/2023/7/13.

Jakubovic, Joel and Tomas Petricek (2022). "Ascending the Ladder to Self-Sustainability: Achieving Open Evolution in an Interactive Graphical System." In: *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2022. Auckland, New Zealand: Association for Computing Machinery, pp. 240–258. ISBN: 9781450399098. DOI: 10.1145/3563835.3568736.

Judith Hays, Margaret Burnett (1995). *A Guided Tour of Forms/3*. URL: http://web.engr.oregonstate.edu/~burnett/Forms3/Tour/tour.html.

Kay, Alan (2000). "The Computer Revolution Hasn't Happened yet (Keynote Session)." In: *Proceedings of the Eighth ACM International Conference on Multimedia*. MULTIMEDIA '00. Marina del Rey, California, USA: Association for Computing Machinery, p. 1. ISBN: 1581131984. DOI: 10.1145/354384.354390. URL: https://doi.org/10.1145/354384.354390.

Kay, Alan and Adele Goldberg (1977). "Personal Dynamic Media." In: *Computer* 10.3, pp. 31–41. DOI: 10.1109/C-M.1977.217672.

Kay, Alan, Dan Ingalls, Yoshiki Oshima, Ian Piumarta, and Andreas Raab (2006). *Proposal to NSF*. URL: http://www.vpri.org/pdf/rn2006002_nsfprop.pdf.

Kay, Alan, Ian Piumarta, Kim Rose, Dan Ingalls, Dan Amelang, Ted Kaehler, Yoshiki Ohshima, Scott Wallace, Alessandro Warth, and Takashi Yamamiya (2007). *STEPS Toward The Reinvention of Programming, First Year Progress Report*. URL: ttp://www.vpri.org/pdf/tr2007008_steps.pdf.

Kay, Alan et al. (2008). *STEPS Toward The Reinvention of Programming, 2008 Progress Report*. URL: http://www.vpri.org/pdf/tr2008004_steps08.pdf.

Kay, Alan et al. (2009). *STEPS Toward The Reinvention of Programming, 2009 Progress Report*. URL: http://www.vpri.org/pdf/tr2009016_steps09.pdf.

Kell, Stephen (2009). "The mythical matched modules: overcoming the tyranny of inflexible software construction. Overcoming the tyranny of inflexible software construction." In: *OOPSLA Companion*.

– (2013). "The Operating System: Should There Be One?" In: *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*. PLOS '13. Farmington, Pennsylvania: Association for Computing Machinery. ISBN: 9781450324601. DOI: 10.1145/2525528.2525534. URL: https://doi.org/10.1145/2525528.2525534.

– (2017). "Some Were Meant for C: The Endurance of an Unmanageable Language." In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: Association for Computing Machinery, pp. 229–245. ISBN: 9781450355308. DOI: 10.1145/3133850.3133867. URL: https://doi.org/10.1145/3133850.3133867.

Kernighan, Brian W. and Dennis M. Ritchie (1989). *The C Programming Language*. USA: Prentice Hall Press. ISBN: 0131103628.

Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold (2001). "An Overview of AspectJ." In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, pp. 327–353. DOI: 10.1007/3-540-45337-7_18. URL: https://doi.org/10.1007/3-540-45337-7%5C_18.

Killian, Tom (1984). "Processes as Files, USENIX Summer Conf." In: *Proc., Salt Lake City*.

Klein, Ursula (2003). *Experiments, Models, Paper Tools: Cultures of Organic Chemistry in the Nineteenth Century*. Stanford, CA: Stanford University Press. ISBN: 9780804743594. URL: http://www.sup.org/books/title/?id=1917.

Klokmose, Clemens N., James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon (2015). "Webstrates: Shareable Dy-

namic Media." In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST '15. Charlotte, NC, USA: Association for Computing Machinery, pp. 280–290. ISBN: 9781450337793. DOI: 10.1145/2807442.2807446. URL: https://doi.org/10.1145/2807442.2807446.

Kluyver, Thomas, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. (2016). "Jupyter Notebooks—a publishing format for reproducible computational workflows." In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, p. 87.

Knuth, D. E. (Jan. 1984a). "Literate Programming." In: *The Computer Journal* 27.2, pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. eprint: https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf. URL: https://doi.org/10.1093/comjnl/27.2.97.

Knuth, Donald Ervin (1984b). "Literate programming." In: *The computer journal* 27.2, pp. 97–111.

Kodosky, Jeffrey (June 2020). "LabVIEW." In: *Proc. ACM Program. Lang.* 4.HOPL. DOI: 10.1145/3386328. URL: https://doi.org/10.1145/3386328.

Kuhn, Thomas S. (1970). University of Chicago Press.

Kumar, Ranjitha and Michael Nebeling (2021). *UIST 2021 - Author Guide*. URL: https://uist.acm.org/uist2021/author-guide.html.

Lau, Sam, Ian Drosos, Julia M. Markel, and Philip J. Guo (2020). "The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry." In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201.

Levy, Steven (1984). *Hackers: Heroes of the Computer Revolution*. USA: Doubleday. ISBN: 0385191952.

Lieberman, Henry, ed. (2001). *Your Wish Is My Command. Programming By Example*. Morgan Kaufmann. URL: https://www.sciencedirect.com/book/9781558606883/your-wish-is-my-command.

Likert, Rensis (1932). "A technique for the measurement of attitudes." In: *Archives of Psychology* 22.140, p. 55.

Lincke, Jens, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff (2017). "Designing a live development experience for web-components." In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*. PX/17.2. Vancouver, BC, Canada: Association for Computing Machinery, pp. 28–35. ISBN: 9781450355223. DOI: 10.1145/3167109. URL: https://doi.org/10.1145/3167109.

Lindholm, Tim, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith (2023). "The Java® Virtual Machine Specification. Java SE 21

Edition." In: URL: https://docs.oracle.com/javase/specs/jvms/se21/html/index.html.

LIVE (2023). *LIVE Workshops*. URL: https://liveprog.org/.

Maes, Pattie (Dec. 1987). "Concepts and Experiments in Computational Reflection." In: *SIGPLAN Not.* 22.12, pp. 147–155. ISSN: 0362-1340. DOI: 10.1145/38807.38821. URL: https://doi.org/10.1145/38807.38821.

Mainland, Geoffrey (2007). "Why It's Nice to be Quoted: Quasiquoting for Haskell." In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: Association for Computing Machinery, pp. 73–82. ISBN: 9781595936745. DOI: 10.1145/1291201.1291211. URL: https://doi.org/10.1145/1291201.1291211.

Marlow, Simon and Simon Peyton-Jones (2012). *The Glasgow Haskell Compiler*. Ed. by A. Brown and G. Wilson. The Architecture of Open Source Applications. CreativeCommons. Chap. 5. ISBN: 9781105571817. URL: http://www.aosabook.org.

McCarthy, John (1962). *LISP 1.5 Programmer's Manual*. The MIT Press. ISBN: 0262130114.

Meyerovich, Leo A. and Ariel S. Rabkin (2012). "Socio-PLT: Principles for Programming Language Adoption." In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012. Tucson, Arizona, USA: Association for Computing Machinery, pp. 39–54. ISBN: 9781450315623. DOI: 10.1145/2384592.2384597. URL: https://doi.org/10.1145/2384592.2384597.

Michel, Stephen L. (1989). *Hypercard: The Complete Reference*. Berkeley: Osborne McGraw-Hill.

Microsoft (May 23, 2022). *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/.

Moloney, J., Alan Borning, and B. Freeman-Benson (1989). "Constraint technology for user-interface construction in ThingLab II." In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '89. New Orleans, Louisiana, USA: Association for Computing Machinery, pp. 381–388. ISBN: 0897913337. DOI: 10.1145/74877.74917. URL: https://doi.org/10.1145/74877.74917.

Monperrus, Martin (2017). "Principles of Antifragile Software." In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. Programming '17. Brussels, Belgium: Association for Computing Machinery. ISBN: 9781450348362. DOI: 10.1145/3079368.3079412. URL: https://doi.org/10.1145/3079368.3079412.

Murray, Alan (2022). "LET, LAMBDA, and the Helper Functions." In: *Advanced Excel Formulas: Unleashing Brilliance with Excel Formulas*. Berkeley, CA: Apress, pp. 751–803. ISBN: 978-1-4842-7125-4. DOI: 10.

1007/978-1-4842-7125-4_15. URL: https://doi.org/10.1007/978-1-4842-7125-4_15.

Nelson, T. H. (1965). "Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate." In: *Proceedings of the 1965 20th National Conference*. ACM '65. Cleveland, Ohio, USA: Association for Computing Machinery, pp. 84–100. ISBN: 9781450374958. DOI: 10.1145/800197.806036. URL: https://doi.org/10.1145/800197.806036.

Nickerson, Jeffrey Vernon (1994). *Visual programming*. New York University. URL: https://web.stevens.edu/jnickerson/ch2.pdf.

Noble, James and Robert Biddle (Dec. 2004). "Notes on Notes on Postmodern Programming." In: *SIGPLAN Not.* 39.12, pp. 40–56. ISSN: 0362-1340. DOI: 10.1145/1052883.1052890. URL: https://doi.org/10.1145/1052883.1052890.

Norman, Donald A. (2002). *The Design of Everyday Things*. USA: Basic Books, Inc. ISBN: 9780465067107.

Norvig, Peter (1996). "Design patterns in dynamic programming." Object World, Boston, MA. URL: https://norvig.com/design-patterns/.

Nystrom, Bob (2014). *Game Programming Patterns*. URL: https://gameprogrammingpatterns.com/.

Olsen, Dan R. (2007). "Evaluating User Interface Systems Research." In: *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. UIST '07. Newport, Rhode Island, USA: Association for Computing Machinery, pp. 251–258. ISBN: 9781595936790. DOI: 10.1145/1294211.1294256. URL: https://doi.org/10.1145/1294211.1294256.

Omar, Cyrus, Ian Voysey, Ravi Chugh, and Matthew A. Hammer (Jan. 2019). "Live Functional Programming with Typed Holes." In: *Proc. ACM Program. Lang.* 3.POPL. DOI: 10.1145/3290327. URL: https://doi.org/10.1145/3290327.

Orchard, Dominic (2011). "The four Rs of programming language design." In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2011. Portland, Oregon, USA: Association for Computing Machinery, pp. 157–162. ISBN: 9781450309417. DOI: 10.1145/2089131.2089138. URL: https://doi.org/10.1145/2089131.2089138.

Orchard, Dominic A and Steve Matthews (2008). "Integrating Lucid's Declarative Dataflow Paradigm into Object-Orientation." In: *Mathematics in Computer Science* 2, pp. 103–122.

Parnas, David Lorge (1985). "Software Aspects of Strategic Defense Systems." In: URL: http://web.stanford.edu/class/cs99r/readings/parnas1.pdf.

Pawson, Richard (2004). "Naked Objects." PhD thesis. Trinity College, University of Dublin.

Perlis, Alan J. (Sept. 1982). "Special Feature: Epigrams on programming." In: *SIGPLAN Not.* 17.9, pp. 7–13. ISSN: 0362-1340. DOI: 10.1145/947955.1083808. URL: https://doi.org/10.1145/947955.1083808.

Perlis, Alan J. and K. Samelson (Dec. 1958). "Preliminary Report: International Algebraic Language." In: *Commun. ACM* 1.12, pp. 8–22. ISSN: 0001-0782. DOI: 10.1145/377924.594925. URL: https://doi.org/10.1145/377924.594925.

Petricek, Tomas (2017). "Miscomputation in software: Learning to live with errors." In: *Art Sci. Eng. Program.* 1.2, p. 14. DOI: 10.22152/programming-journal.org/2017/1/14. URL: https://doi.org/10.22152/programming-journal.org/2017/1/14.

Petricek, Tomas and Joel Jakubovic (2021). "Complementary science of interactive programming systems." In: *History and Philosophy of Computing*.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0262162091.

Pike, Rob, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom (Apr. 1993). "The Use of Name Spaces in Plan 9." In: *SIGOPS Oper. Syst. Rev.* 27.2, pp. 72–76. ISSN: 0163-5980. DOI: 10.1145/155848.155861. URL: https://doi.org/10.1145/155848.155861.

Piumarta, Ian (2006). *Accessible Language-Based Environments of Recursive Theories*. URL: http://www.vpri.org/pdf/rn2006001a_colaswp.pdf.

– (2011). "Open, Extensible Composition Models." In: *Proceedings of the 1st International Workshop on Free Composition*. FREECO '11. Lancaster, United Kingdom: Association for Computing Machinery. ISBN: 9781450308922. DOI: 10.1145/2068776.2068778. URL: http://www.vpri.org/pdf/tr2011002_oecm.pdf.

Piumarta, Ian and Alessandro Warth (2008). "Open, Extensible Object Models." In: *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, pp. 1–30. ISBN: 9783540892748. DOI: 10.1007/978-3-540-89275-5_1. URL: http://www.vpri.org/pdf/tr2006003a_objmod.pdf.

Polito, Guillermo, Stéphane Ducasse, Noury Bouraqadi, and Luc Fabresse (2015). "A Bootstrapping Infrastructure to Build and Extend Pharo-like Languages." In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 183–196. ISBN: 9781450336888. DOI: 10.1145/2814228.2814236. URL: https://doi.org/10.1145/2814228.2814236.

PX (2023). *Programming Experience (PX) Workshops*. URL: http://programming-experience.org/.

Pyret Crew (2016). *They Pyret Code; or A Rationale for the Pyret Programming Language*. URL: https://pyret.org/pyret-code/index.html.

Python Guide (2016). *The Hitchiker's Guide To Python*. URL: https://docs.python-guide.org/writing/gotchas/#mutable-default-arguments.

Rauch, David, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld (2019). "Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code." In: *CoRR* abs/1902.00549. arXiv: 1902.00549. URL: http://arxiv.org/abs/1902.00549.

Raymond, Eric S. and Bob Young (2001). *The Cathedral & The Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly.

Reason, James (1990). *Human error*. Cambridge university press.

Reese, Ivan (2022). *Visual Programming Codex*. URL: https://github.com/ivanreese/visual-programming-codex.

Renggli, Lukas and Tudor Gîrba (2009). "Why Smalltalk wins the host languages shootout." In: *Proceedings of the International Workshop on Smalltalk Technologies*. IWST '09. Brest, France: Association for Computing Machinery, pp. 107–113. ISBN: 9781605588995. DOI: 10.1145/1735935.1735954. URL: https://doi.org/10.1145/1735935.1735954.

repl.it (May 23, 2022). *Replit: The collaborative browser-based IDE*. URL: https://replit.com.

Rose, Kim and Robert Hirschfeld, eds. (2008). *Self-Sustaining Systems, First Workshop*. Vol. 5146. Lecture Notes in Computer Science. Potsdam, Germany: Springer. ISBN: 978-3-540-89274-8. DOI: 10.1007/978-3-540-89275-5. URL: https://doi.org/10.1007/978-3-540-89275-5.

Rose, Kim, Robert Hirschfeld, and Hidehiko Masuhara, eds. (2010). *Workshop on Self-Sustaining Systems*. Tokyo, Japan: ACM. ISBN: 978-1-4503-0491-7. DOI: 10.1145/1942793. URL: https://doi.org/10.1145/1942793.

Ryder, Barbara G., Mary Lou Soffa, and Margaret Burnett (Oct. 2005). "The Impact of Software Engineering Research on Modern Programming Languages." In: *ACM Trans. Softw. Eng. Methodol.* 14.4, pp. 431–477. ISSN: 1049-331X. DOI: 10.1145/1101815.1101818. URL: https://doi.org/10.1145/1101815.1101818.

Shneiderman (1983). "Direct Manipulation: A Step Beyond Programming Languages." In: *Computer* 16.8, pp. 57–69. DOI: 10.1109/MC.1983.1654471.

Sitaker, Kragen Javier (2016). *The Memory Models That Underlie Programming Languages*. URL: http://canonical.org/~kragen/memory-models/.

Smaragdakis, Yannis (2019). "Next-Paradigm Programming Languages: What Will They Look like and What Changes Will They Bring?" In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery,

pp. 187–197. ISBN: 9781450369954. DOI: 10.1145/3359591.3359739. URL: https://doi.org/10.1145/3359591.3359739.

Smith, Brian Cantwell (1982). "Procedural Reflection in Programming Languages." PhD thesis. Massachusetts Institute of Technology. URL: https://dspace.mit.edu/handle/1721.1/15961.

Smith, D. C. (1975). "Pygmalion: a creative programming environment." In.

Steele, G. and S.E. Fahlman (1990). *Common LISP: The Language*. HP Technologies. Elsevier Science. ISBN: 9781555580414. URL: https://books.google.cz/books?id=FYoOIWuoXUIC.

Steele, Guy L. and Richard P. Gabriel (1993). "The Evolution of Lisp." In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: Association for Computing Machinery, pp. 231–270. ISBN: 0897915704. DOI: 10.1145/154766.155373. URL: https://doi.org/10.1145/154766.155373.

Steimann, Friedrich, Marcus Frenkel, and Markus Voelter (2017). "Robust projectional editing." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, pp. 79–90. ISBN: 9781450355254. DOI: 10.1145/3136014.3136034. URL: https://doi.org/10.1145/3136014.3136034.

Sussman, Gerald Jay and Jack Wisdom (2001). *Structure and Interpretation of Classical Mechanics*. Cambridge, MA, USA: MIT Press. ISBN: 0262194554.

Sústrik, Martin (2019). *Hull: An alternative to shell that I'll never have time to implement*. URL: https://250bpm.com/blog:153/.

Sutherland, Ivan Edward (1963). "Sketchpad: A man-machine graphical communication system." PhD thesis. Massachusetts Institute of Technology. DOI: https://doi.org/10.1145/1461551.1461591. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf.

Tanimoto, Steven L. (2013). "A Perspective on the Evolution of Live Programming." In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE '13. San Francisco, California: IEEE Press, pp. 31–34. ISBN: 9781467362658.

Tchernavskij, Philip (2019). "Designing and Programming Malleable Software." PhD thesis. Université Paris-Saclay, École doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC).

Teitelman, Warren (1966). "PILOT: A Step Toward Man-Computer Symbiosis." PhD thesis.

Ungar, David and Randall B. Smith (2007). "Self." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, pp. 9–1–9–50. ISBN: 9781595937667. DOI: 10.1145/1238844.1238853. URL: https://doi.org/10.1145/1238844.1238853.

Victor, Bret (2012). *Learnable Programming*. URL: http://worrydream.com/#!/LearnableProgramming.

– (2013). *Drawing Dynamic Visualisations*. URL: http://worrydream.com/#!/DrawingDynamicVisualizationsTalk.

ViewPoints Research Institute (2010). *STEPS Toward Expressive Programming Systems, 2010 Progress Report*. URL: http://www.vpri.org/pdf/tr2010004_steps10.pdf.

Voelter, Markus and Vaclav Pech (2012). "Language Modularity with the MPS Language Workbench." In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, pp. 1449–1450. ISBN: 9781467310673.

Wall, Larry (1999). *Perl, the first postmodern computer language*. URL: http://www.wall.org//~larry/pm.html.

Warth, Alessandro (2009). "Experimenting with Programming Languages." PhD thesis. Los Angeles, CA: University of California. URL: http://www.vpri.org/pdf/tr2008003_experimenting.pdf.

Wickerson, John and Paul Brunet (2012). "Pearl: Diagrams for Composing Compilers." In: URL: https://johnwickerson.github.io/papers/jdiagrams.pdf.

Winestock, Rudolf (2011). *The Lisp Curse*. URL: http://www.winestockwebdesign.com/Essays/Lisp_Curse.html.

Wolfram, Stephen (1991). *Mathematica: A System for Doing Mathematics by Computer (2nd Ed.)* USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0201515075.

Würthinger, Thomas, Christian Wimmer, and Lukas Stadler (2013). "Unrestricted and safe dynamic code evolution for Java." In: *Science of Computer Programming* 78.5. Special section: Principles and Practice of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination, pp. 481–498. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2011.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0167642311001456.

Zynda, Melissa Rodriguez (Sept. 2013). "The First Killer App: A History of Spreadsheets." In: *Interactions* 20.5, pp. 68–72. ISSN: 1072-5520. DOI: 10.1145/2509224. URL: https://doi.org/10.1145/2509224.