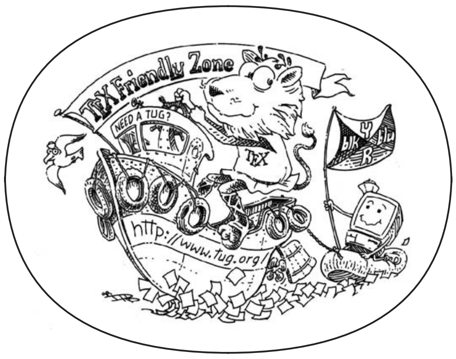
JOEL JAKUBOVIC

ACHIEVING SELF-SUSTAINABILITY IN INTERACTIVE GRAPHICAL PROGRAMMING SYSTEMS

ACHIEVING SELF-SUSTAINABILITY IN INTERACTIVE GRAPHICAL PROGRAMMING SYSTEMS

JOEL JAKUBOVIC



Doctor of Philosophy (PhD)
Computing, Engineering and Mathematical Sciences
School of Computing
University of Kent

September 2019–June 2023 – classicthesis v4.6

Joel Jakubovic: *Achieving Self-Sustainability in Interactive Graphical Programming Systems*, , © September 2019–June 2023

SUPERVISORS:
Tomas Petricek
Stefan Marr

LOCATION:
Canterbury, Kent

TIME FRAME:
September 2019–June 2023

ABSTRACT

This is my abstract, should be generated from MD.

PUBLICATIONS

This might come in handy for PhD theses: some ideas and figures have appeared previously in the following publications:

This is just an early – and currently ugly – test!

Jakubovic, Joel (2020). "What it takes to create with domain-appropriate tools. Reflections on implementing the "Id" system." In: Convivial Computing Salon 2020. Porto, Portugal.

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. URL: https://doi.org/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. URL: https://doi.org/10.1145/3563835.3568736.

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

Attention: This requires a separate run of `bibtex` for your `refsection`, e. g., `ClassicThesis1-blx` for this file. You might also use `biber` as the backend for `biblatex`. See also http://tex.stackexchange.com/questions/128196/problem-with-refsection.

ACKNOWLEDGMENTS

Acknowledgments go here, from MD

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

When we have an idea for some computer software, and try and 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. 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, but the text is too small for us to read. The designers have not included a feature for increasing the text size (perhaps it is a special message in a separate part of the app which is unaffected by the normal controls). A programmer would know that there is some API being called to render the text, and this API will be told the font size via some number in the app's memory. If we could just find this single number and change it, we might at least be able to read the text (even if the app is not prepared to lay out the larger text correctly in response to such a "surgical" intervention).

So, what does it take to find and change a number? The app itself provides no way to proceed through its surface interface to such internal details. This means we must face at least the accidental complexity of 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 font size. Such an expert task would take an extremely long time even for someone with the relevant experience. It does let us make a change to the running app, but not permanently; we would have to do this 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, even for programmers, on a novel codebase. 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; run-time tampering could be prevented by security policy (especially on a mobile device) and re-building from source cannot work without access *to* the source. Suffice to say, none of this is suitable for an average user. Even a seasoned programmer would consider it not worth the bother. Our task of changing a number, while technically possible, has a severely *disproportionate* accidental complexity cost.

In specific situations, software authors do have good reasons to restrict access to internals. For example, in a game it is important to enforce the rules and access to internals would enable arbitrary cheating. However, this consideration is not representative of most types of software. Despite this, we are unable to simply *choose* to build such "open" software. 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 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. Let's call this *naïve pokeability*:

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 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.

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 enforce their syntax and semantics without permitting *smaller-scale* adaptation, and variation in these respects is restricted to textual notations. Furthermore, some situations may call for more general *notations* or graphical interfaces that do not work like a language, but 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.

1.2 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 discussing the examples of the Web, HyperCard, and Smalltalk.

1.2.1 *Web pages, web apps, and browsers*

The *web browser* has a powerful set of *developer tools*. This includes the "element inspector" which can be used to edit the web page's underlying elements. For example, an ad can be removed by locating and deleting its element. Note that the "state" here is mostly visible, since it is directly responsible for visual elements appearing on the page. Some of the 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 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 centered around the JavaScript programming language. Alongside the "structural" state of the web page, there is also the hidden state of JavaScript objects. The JavaScript console accepts commands which may read or change this state, but there is nothing like the element inspector for it.[1] 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 at the console; for example, updating part of the state to a new object. However, this will not work if the source declares the variable as `const`, and more importantly, this does not work for fine-grained changes to code. The main unit of code organisation, the `function`, is an opaque object in the runtime 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. However, replacing a function can be prohibited by the source just like the redefinition of `const` variables. In these cases, there is 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 do not take effect until the scripts are reloaded by refreshing the page.

Compare the above situation with the "structural" state of the web page; we are free to make arbitrarily fine-grained changes using the element inspector, and our changes take effect immediately without having to reset anything. There are, in fact, HTML text files backing the page structure, but they are irrelevant given the inspector tool. In

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

short, the *state* of a web page has *naïve pokeability* (Definition 1) while its dynamic *behaviour* does not reliably have this property.

There is one caveat: since the HTML and JavaScript files are the "ground truth", any changes made via the inspector or the console will disappear when the page is closed or refreshed. Only changes to the underlying files are *persistent*, and websites typically do not allow random 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 use sophisticated programmatic middleware, such as an ad-blocking extension, to do this automatically).

1.2.2 *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 doesn't include an *authoring* tool, instantly limiting the *creation* of web pages to people who can code in a text editor. In HyperCard, the viewer and editor exist integrated together. Furthermore, there is an "edit" mode whereby a user can remix content from someone else, even reprogramming 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.

1.2.3 *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 method; only then is a text editor presented for the code. 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.

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.

Going further in the same direction as Smalltalk is the Combined Object Lambda Architecture or COLA Ian Piumarta 2006. COLA 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 down to a small scale of "mood-specific languages" 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.

1.3 THE MISSING SYNTHESIS

The situation is that we can pick at most two from:

1. GUIs with the potential for domain-specific graphical adaptation (Web, Smalltalk)
2. Reliable, persistent naïve pokeability of state and behaviour (Smalltalk, COLA)
3. Full self-sustainability with domain-specific languages (COLA)

Why has a synthesis of all three not been achieved? Part of the difficulty is that programming is framed in terms of *languages* with a focus on parsed syntax and batch-mode transformations. This makes it an uphill battle to achieve even one of these three properties.

It's unfortunate that we conflate "coding" in a programming language with programming itself. This makes it hard to talk about the more general concept that contains alternative possibilities. We see *programming* as the ill-defined act of making a computer do things by itself. Coding, visual programming, programming by example and deep learning are some specific *means* by which to program.

If we see programming as coding, then we unwittingly limit the scope of innovative ideas.

- Instead of seeking the right notation, interface or representation for the job—we seek the right *textual syntax* for the job. If we can't find one, the real reason may simply be that text is not well-suited to the job! Yet if text is all we know, it looks like 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 it. It is easy to make closed programs this way but hard to make open ones.
- 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.

This last point is the crux of the matter: we need a more general programming *systems* approach instead. In Chapters **??** and **??** we will discuss this and 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 proceed to familiarise the reader with the basic outline of these three properties.

1.4 THE THREE PROPERTIES

The goal at the end of Section 1.1 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 underlie a good proportion of 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.

We are interested in exploring, developing, and achieving these 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". Once a programmer has decided what constitutes the latter 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 parsing/serialising code or think about escaping.

These properties are exhibited occasionally in different systems, as we will mention in Chapters 2 and **??**. 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 1.1.

Furthermore, 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*). Despite this, both Notational Freedom and Self-Sustainability suffer without Explicit Structure. Notational Freedom is impossible to achieve in a world of parsed strings and text editors (this being merely what we term *syntactic* freedom in Section **??**) so it needs Explicit Structure as a necessary foundation. Self-Sustainability is currently best understood as a vague analogy to self-hosting compilers, as we will see in Section 2.3.1. The COLA work follows this view, being unclear how such a property can be achieved in interactive, graphical systems. Explicit Structure lets us study these other two properties more purely, without getting confused by the accidental complexities of parsing and escaping.

Due to this last point, we see Explicit Structure as a necessary foundation for the work to follow. Thus, out of the six possible ways we could prioritise the Three Properties, we are left with two: explore self-sustainability with the aid of notational freedom, or vice versa. Our work in Chapter **??** will take the former path and Chapter **??** will fit the latter.

1.5 THESIS STATEMENT AND CONTRIBUTIONS

The statement of our thesis is as follows:

> It is possible to add Notational Freedom to the web browser programming system by embedding a Self-Sustainable system built on Explicit Structure.

We prove this by constructing such a system which we call *BootstrapLab*. It is evaluated according to *technical dimensions* that we derive from the Three Properties. These dimensions fit within the methodological framework that we propose for studying programming systems in Chapter **??**.

BootstrapLab itself is a contribution, but we also contribute 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; our hope is that in Chapter **??** we have documented enough of a generalisable technique to make this feasible for the average programmer.

Instead of seeking to master the ins-and-outs of Smalltalk, Unix or indeed BootstrapLab, what is needed is to steal the best ideas and synthesise them into something fresh—to have our cake and eat it too. 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.

1.6 SUPPORTING PAPERS

The following publications form chapters in this thesis:

Joel Jakubovic, 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`. URL: `https://doi.org/10.22152/programming-journal.org/2023/7/13`

This won the journal's Editors' Choice Award and was adapted into Chapter **??**.

Joel Jakubovic (2020). "What it takes to create with domain-appropriate tools. Reflections on implementing the "Id" system." In: Convivial Computing Salon 2020. Porto, Portugal

This forms Chapter **??**.

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`. URL: `https://doi.org/10.1145/3563835.3568736`

This forms Chapter **??**.

2 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. Subsequently, 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.

The classic essay by Gabriel 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 might 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 thesis, 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 have no use for 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 mention next.

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. Therefore, 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 with analysis without worrying too much about the variations.

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* REPL, alongside a text editor that provides features like syntax highlighting and autocompletion. 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. n.d. 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 1984 used in the late 1980s in Mathematica 1.0 Wolfram 1991.

As a programming system representative of Computational Notebooks Lau et al. 2020, Jupyter has a number of 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 G. 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” begin 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 G. L. 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 **??**) 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 systems, 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*, Definition **??**). Lastly, the fact that much of the Smalltalk environment is implemented in itself makes it possible to extensively modify the system from within.

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.

UNIX. Unix fits Definition 9 and illustrates the fact that many aspects of programming systems are shaped by their 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 **??**.

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, JavaScript 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, JavaScript 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 JavaScript as a compilation target, and JavaScript 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 1.2.1 it does come with advanced developer tools providing functionality resembling that of Lisp and Smalltalk. The Document Object Model (DOM) almost resembles the tree/graph model of Smalltalk and Lisp images, lacking the key persistence property (Definition **??**). Such a limitation can be surpassed 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 Ian 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 structural object model (the "Object" in the acronym) 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 say more about in Section 2.3. 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 *mood-specific languages:*

> 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 mood-specific language is the one reported in Kay, I. Piumarta, 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.

This machine interpretation of ASCII art diagrams is another example of a mood-specific language, and we see this as the extreme end of what they are 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 Notational Freedom. With such a capability, code could be synthesised from *real* tabular diagrams of packet headers, not just those rendered with ASCII characters.

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 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.*

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 is also the Programming-By-Example Lieberman 2001 subset in which a general program is automatically generated by the user supplying sample behaviours and hints.

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. Here, we will give a glossary of these existing concepts. In short:

- Self-Sustainability is foreshadowed by self-hosting compilers and reflection.
- Notational Freedom generalises domain-specific languages and polyglot programming.
- Explicit Structure already exists in the form of data structures and various editors.

2.3.1 *Precursors of Self-Sustainability*

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 and we can call the language "self-hosting" (as a proxy for its compiler.)

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:

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.

META-CIRCULAR. This describes an interpreter that is written in its own language C. 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 as Lisp expressions. If such code was compiled, it would result 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.

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 Bystroushaak 2019 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.

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 **??** 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*

USE THE RIGHT TOOL FOR THE JOB. This is a widespread maxim in programming C. W. Contributors 2014. 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 and 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.

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 runtime sharing, languages use Foreign Function Interfaces (FFIs). This practice has been critiqued by Kell 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 JavaScript support regexes written

with a local syntax. HTML files include not only HTML but also JavaScript 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.

DOMAIN-SPECIFIC LANGUAGES (DSLS). 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. JetBrains' 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 Ian 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 mood-specific languages and even more general notations.

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.* Only MPS and Eco, as far as we are aware, go "all the way" in this regard. We will continue this discussion in more detail in Chapter **??**.

2.3.3 *Precursors of Explicit Structure*

2.3.3.1 *Representation*

STRUCTURE. This is the relation of parts to wholes. An entire data structure is made up of smaller parts which reference each other. From a machine's point of view, a data structure is a graph of memory blocks connected by numerical pointers. For humans, 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.

BINARY FILES. These are *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.

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.

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.

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.

2.3.3.2 *Formats*

TEXT FILES, A.K.A. 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 usually a tree, in which case the string is built as an *in-order* traversal. This means that special qualitative syntax (usually the matched bracket characters `(`, `[`, `{`, or `<`) is employed *around* substrings to encode the structure.

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.

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 2.1).

SYNTAX ERROR. This is a special case of format error where part of a text string violates a grammatical expectation of its consumer.

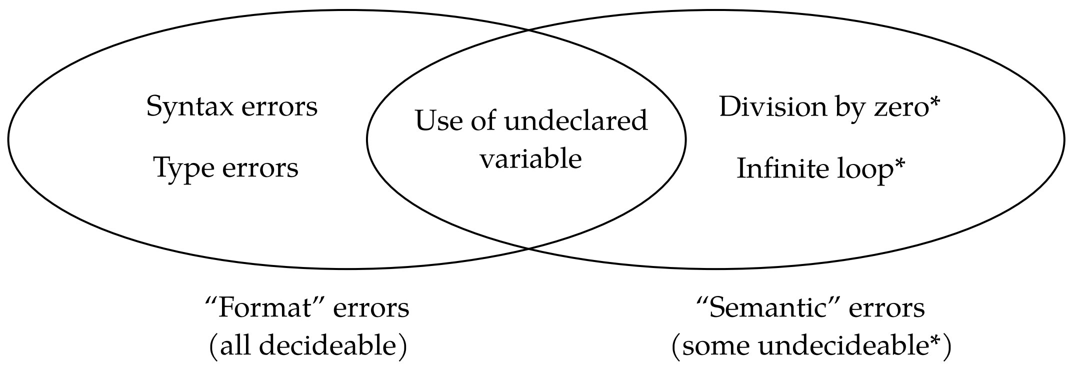

Syntax errors
Type errors
Use of undeclared variable
Division by zero*
Infinite loop*
"Format" errors (all decideable)
"Semantic" errors (some undecideable*)

Figure 2.1: Format errors include syntax errors, type errors and some "semantic" errors as long as they are decideable.

2.3.3.3 *Editors*

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.

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.

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[1] implicit content (on the programming side).

1 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 may be applicable to cases where software must parse and interpret natural language too.

A
APPENDIX TEST

Lorem ipsum at nusquam appellantur his, ut eos erant homero concludaturque. Albucius appellantur deterruisset id eam, vivendum partiendo dissentiet ei ius. Vis melius facilisis ea, sea id convenire referrentur, takimata adolescens ex duo. Ei harum argumentum per. Eam vidit exerci appetere ad, ut vel zzril intellegam interpretaris.

More dummy text.

A.1 APPENDIX SECTION TEST

Test: Table A.1 (This reference should have a lowercase, small caps A if the option `floatperchapter` is activated, just as in the table itself → however, this does not work at the moment.)

LABITUR BONORUM PRI NO	QUE VISTA	HUMAN
fastidii ea ius	germano	demonstratea
suscipit instructior	titulo	personas
quaestio philosophia	facto	demonstrated

Table A.1: Autem usu id.

A.2 ANOTHER APPENDIX SECTION TEST

Equidem detraxit cu nam, vix eu delenit periculis. Eos ut vero constituto, no vidit propriae complectitur sea. Diceret nonummy in has, no qui eligendi recteque consetetur. Mel eu dictas suscipiantur, et sed placerat oporteat. At ipsum electram mei, ad aeque atomorum mea. There is also a useless Pascal listing below: Listing A.1.

Listing A.1: A floating example (`listings` manual)

```
for i:=maxint downto 0 do
begin
{ do nothing }
end;
```

Ankerson, Megan Sapnar (2018). *Dot-Com Design: The Rise of a Usable, Social, Commercial Web*. NYU Press. ISBN: 1479892904.

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`.

Contributors, C2 (2012). *Meta-circular Evaluator*. URL: `https://wiki.c2.com/?MetaCircularEvaluator`.

Contributors, C2 Wiki (2014). *Pick The Right Tool For The Job*. URL: `https://wiki.c2.com/?PickTheRightToolForTheJob`.

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/`.

Evans, Edmund Grimley (2001). *Bootstrapping A Simple Compiler From Nothing*. URL: `https://web.archive.org/web/20061108010907/http://www.rano.org/bcompiler.html`.

Ford, Neal (2006). *Polyglot Programming*. URL: `http://memeagora.blogspot.com/2006/12/polyglot-programming.html`.

Gabriel, Richard P. (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`.

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`.

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`.

Jakubovic, Joel (2020). "What it takes to create with domain-appropriate tools. Reflections on implementing the "Id" system." In: Convivial Computing Salon 2020. Porto, Portugal.

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. URL: https://doi.org/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. URL: https://doi.org/10.1145/3563835.3568736.

Kay, A. and A. Goldberg (1977). "Personal Dynamic Media." In: *Computer* 10.3, pp. 31–41. DOI: 10.1109/C-M.1977.217672.

Kay, A., I. Piumarta, et al. (2008). *STEPS Toward The Reinvention of Programming, 2008 Progress Report*. URL: http://www.vpri.org/pdf/tr2008004_steps08.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*.

Klokmose, Clemens N., James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon (2015). "Webstrates: Shareable Dynamic 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. (n.d.). "Jupyter Notebooks—a publishing format for reproducible computational workflows." In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (), p. 87.

Knuth, Donald Ervin (1984). "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.

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.

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.

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.

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/.

Piumarta, Ian (2006). *Accessible Language-Based Environments of Recursive Theories*. URL: http://www.vpri.org/pdf/rn2006001a_colaswp.pdf.

Smith, Brian Cantwell (1982). "Procedural Reflection in Programming Languages." PhD thesis. Massachusetts Institute of Technology. URL: https://dspace.mit.edu/handle/1721.1/15961.

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.

Teitelman, Warren (1966). "PILOT: A Step Toward Man-Computer Symbiosis." PhD thesis.

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.

Warth, Alessandro (2009). "Experimenting with Programming Languages." PhD thesis. Los Angeles, CA: University of California. URL: http://www.vpri.org/pdf/tr2008003_experimenting.pdf.

Wolfram, Stephen (1991). *Mathematica: A System for Doing Mathematics by Computer (2nd Ed.)* USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0201515075.

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.

DECLARATION

Put your declaration here.

Canterbury, Kent, September 2019–June 2023

Joel Jakubovic

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LYX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.