Wikilambda the Ultimate

The Wikimedia foundation’s search for the perfect language

Abstract

In 2020, the Wikimedia foundation launched its first new project in nearly a decade. The new project consists of two main parts: (1) Wikifunctions, a library of programming functions; and (2) Abstract Wikipedia, a language-agnostic Wikipedia that will be dynamically translated into the reader’s native tongue. Lying beneath both Wikifunctions and Abstract Wikipedia is a new system called Wikilambda, which can execute code in potentially any programming language, providing a massively flexible computing service drawing on Wikifunctions and powering Abstract Wikipedia. The entire system is designed to address a fundamental bias in Wikipedia, namely its bias towards majority languages like English and Spanish. In this paper, I present Wikilambda as an audacious attempt to realise a ‘perfect language’, as theorised by Umberto Eco (1995). Wikilambda provides a way of specifying functions that is supposed to transcend any particular ‘native’ language. In this way, it provides editors of Wikifunctions and Abstract Wikipedia with a way of contributing to the overall system no matter which ‘native’ programming languages they know. More broadly, Wikilambda aims to achieve the ‘democratization of programming’, by enabling any person to use any function without needing to know English or a particular programming language (Vrandečić 2021). To analyse the technical and ideological aspects of Wikilambda, I apply the techniques of Critical Code Studies (Marino 2020) to ‘the orchestrator’, the JavaScript application that instantiates Wikilambda’s new functional programming language. In the absence of a formal specification of the language, the Abstract Wikipedia team have gradually hacked Wikilambda out of JavaScript, leaving a fascinating public record of their attempt to realise their vision for a universal programming system.

The story of the search for the perfect language is the story of a dream and of a series of failures. (Eco 1995, 19)

## 1 Introduction: the risk

On New Year’s Day, 2023 an alarming headline appeared in *The Signpost*, Wikipedia’s community newsletter:

Wikimedia Foundation’s Abstract Wikipedia project “at substantial risk of failure” (Bayer 2023)

Abstract Wikipedia is a bold project conceived by Denny Vrandečić, the computer scientist behind the last major project of the Wikimedia Foundation: Wikidata. With Wikidata, Vrandečić had aimed to provide a machine-readable complement to Wikipedia. On Wikipedia, a human might read a person’s biography. On Wikidata, a computer can extract structured data for use in knowledge graphs and information retrieval. Vrandečić’s new project, Abstract Wikipedia, would combine the benefits of Wikipedia and Wikidata. In Abstract Wikipedia, human writers would compose articles in a machine-readable “template language.” A computer would “render” these articles into all the world’s languages. Ten years after they bet the farm on Vrandečić’s Wikidata project, the Wikimedia Foundation were ready for their next big gamble. In 2020, they re-hired Vrandečić from Google, launched Abstract Wikipedia as an official project, and in 2022-23, awarded the project US$1 million from the Wikimedia Endowment (“Annual Report 2022-23” 2023; Foundation 2023).

Three years in, and the project was “at substantial risk of failure.” What happened?

At the heart of the story is a programming language, or rather an entire programming *system*, called Wikilambda. Wikilambda is an extension to the MediaWiki software that runs Wikipedia. By default, MediaWiki allows contributors to read and write *articles*. The Wikilambda extension allows them to read and write *functions*. An article is to be *read*. A function can be *executed*. Wikilambda underlies the new Wikifunctions database, which one day will host the software that runs Abstract Wikipedia.[[1]](#footnote-20) Wikilambda is the reason for the “substantial risk of failure” reported in *The Signpost*. The idea of an “Abstract” Wikipedia, dynamically translated into all the world’s languages, is audacious enough. The idea of programming this new Abstract Wikipedia in a brand-new general-purpose programming system the like of which the world has never seen—is sublime.

Wikilambda provides a brand-new programming language and a brand-new environment for programming in it. The language has been designed to be maximally flexible and accessible. It will allow *any* person of *any* skill level to write *any* program in *any* language—that is, in any *natural* or *programming* language. If you wish to implement a function in Haskell, but use Hindi names for the function arguments, that’s fine! Wikilambda will ensure your Haskell function works with code written in C or Python, and will ensure that an Italian reader can read the function arguments in their own tongue. If you know what a function should do, but don’t know how to implement it, you can add the function to the database, write a description and perhaps some test cases to specify its behaviour, and wait for an experienced programmer to submit an implementation of the function.

Wikilambda will provide a common language for computer scientists and native language speakers, who will meet in Wikifunctions to build all the software for Abstract Wikipedia. This pooling of expertise is essential to achieve Abstract Wikipedia’s aims. The world’s native speakers are needed for their linguistic expertise, and the world’s programmers are needed for their coding prowess. Wikilambda gives these experts a language in which to communicate. Remarkably, this language has no name. The Abstract Wikipedia team sometimes call it the “Function Model,” and sometimes “the orchestrator” (“Wikifunctions:Function Model” 2024). At other times, they deny it is a language: “Wikifunctions is not a programming language, nor is trying to evangelise a particular language.” (“Wikifunctions:What Wikifunctions Is Not” 2024) But as the project has proceeded, the team have become embroiled in all the tricky problems of programming language design, and some observers have doubted their decisions.

The gravest doubters were the Google Fellows, a group of Google engineers seconded to the project. It was their damning evaluation of Wikilambda that triggered the story in *The Signpost*:

To summarize, creating a good programming language is hard, and having a good clear initial design is crucial. The Wikifunctions model ignores decades of programming language research and existing technology. Instead, it invents a completely new ad-hoc system, but that unfortunately does not seem to have good properties, and it is questionable whether it will be able to support [such] a large, complex software system, as Abstract Wikipedia. (Livneh et al. 2022)

Wikilambda is indeed remarkably “ad-hoc.” There is no specification of the language. There is no public procedure for agreeing on a specification. In this respect, Wikilamba makes a striking contrast with other free and open-source (FOSS) languages of the internet era, such as Python, Haskell or Common LISP, which have been designed collaboratively by online communities (Hudak 1989; Orlikowski and Yates 2002; Nand Sharma, Tony Roy Savarimuthu, and Stanger 2022). Unlike these carefully negotiated languages, Wikilamba appears to have been hacked together according to the whims and aspirations of its creators, who are fired by the uniqueness and morality of their enterprise. In their response to the Google Fellows’ evaluation, the Abstract Wikipedia team brushed aside the engineering concerns. The Google engineers might want the Abstract Wikipedia team to choose a single, proven, technical solution to their project, but this would defeat the aim of allowing “the community to co-create the necessary solutions for Abstract Wikipedia” (Blanton et al. 2022). Under-specification is not a failure of Wikilambda—it is the whole point.

In this essay, I seek to understand both the audacity and the ad-hockery of Wikilambda, the only piece of Abstract Wikipedia that currently exists. As a scholar of Eighteenth-Century literature, I feel in familiar territory. Wikilambda is only the latest of many proposals for a “perfect” or “universal” language, a language that will end the curse of Babel and unite the people of the world into a single community. Vrandečić (2021, 2020) himself is aware of this history, and tries to distinguish Abstract Wikipedia from the failed “universal language” of the seventeenth-century German philosopher Gottfried Leibniz. Vrandečić is in fact correct to distinguish Abstract Wikipedia and Wikifunctions from Leibniz’s *characteristica universalis*, but this is only because he has picked the wrong example. Leibniz aimed to create a language that would make formal reasoning more reliable. Vrandečić aims to create a language—or languages—in which speakers of any language can communicate. He aims to create a “polygraphy” (Eco 1995, chap. 9). Unfortunately for Vrandečić, every prior attempt at polygraphy has also failed.

The history of “perfect” and “universal” languages provides a rich context for understanding Wikilambda, and appreciating the particular variety of digital utopianism it expresses. In what follows, I apply Umberto Eco’s (1995) theory of perfect and universal languages to Wikilamba, unpacking the rhetoric and design decisions of the project. In [Section 2](#sec-architecture) I briefly describe the Abstract Wikipedia/Wikifunctions project, and Wikilambda’s place within it. In [Section 3](#sec-universality) and [Section 4](#sec-perfection), I consider first the “universal” and then the “perfect” aspects of Wikilamba, as expressed in the published writings of Vrandečić and his team. Eco (1995, 73) usefully distinguishes universality from perfection: a “universal” language is one “which everyone might, or ought to, speak;” a “perfect” language is one “capable of mirroring the true nature of objects.” I show that Vrandečić and his team vigorously defend the universality of Wikilambda, and make an altogether more paradoxical argument for its perfection. In [Section 5](#sec-code), I analyse the source code of “the orchestrator,” using the methods of Critical Code Studies (Marino 2020). The code itself records the struggle of the Abstract Wikipedia team to enact their vision. By studying the revision history of the code, we can see the Wikilambda ideology in the raw, as the team hew their utopia out of JavaScript.

Wikilambda is remarkable, both as a utopian proposition and as a digital platform. As a *utopia*, Wikilambda is communitarian, and implicitly critiques the entrepreneurial individualism of Silicon Valley and the “Californian Ideology” (Barbrook and Cameron 1996; Turner 2006). In this perspective, Wikilambda is an attempt to extend into computer programming the much-studied agonistic governance structures of Wikipedia (Tkacz 2015; Crawford and Gillespie 2016; Ford 2022)—despite the lack of such governance in the platform’s actual development. As a *platform*, Wikilambda attempts to realise something genuinely new: a platform whose own API is programmable by the community. Wikilambda thus moves beyond the three kinds of platform famously identified by Andreessen (2007; see also Helmond 2015). While it is hard to believe that Wikilambda will succeed, let alone the broader Abstract Wikipedia project, its audacity merits close attention.

## 2 Architecture: Abstract Wikipedia/Wikifunctions, and Wikilambda’s place within it

Abstract Wikipedia will not exist as a separate Wikipedia, the way [German Wikipedia](https://de.wikipedia.org) or [Wikipedia bahasa Indonesia](https://id.wikipedia.org) do. Instead, it will provide a mechanism for editors to add “abstract” pages to existing Wikipedias (“Abstract Wikipedia/Architecture” 2024; “Abstract Wikipedia/Components” 2024). The “abstract content” for the page will be stored in [Wikidata](https://wikidata.org). The “renderers” that translate the abstract content into natural language will be stored in [Wikifunctions](https://wikifunctions.org). When an abstract page is added to a particular Wikipedia, a new page will be generated, by fetching the abstract content from Wikidata, and feeding it into the relevant renderer from Wikifunctions.

Wikipedia editors will have control over how Abstract Wikipedia is integrated into their language. They may pick and choose which Abstract Wikipedia articles to import, or may link their Wikipedia to Abstract Wikipedia dynamically, automatically including all available Abstract articles for items that don’t already have articles in their language edition. This process relies on Wikidata to identify gaps in a given Wikipedia. Every article in Wikipedia is assumed to describe a single item, which is assigned an ID in Wikidata. Using Wikidata, it is possible to see which Wikipedias have articles about a given item. For instance, Joseph Furphy’s *Such is Life* has the Wikidata ID [Q7632777](https://www.wikidata.org/wiki/Q7632777). At the time of writing, this classic of Australian literature has articles only in the [English](https://en.wikipedia.org/wiki/Such_Is_Life_(novel)) and [Serbo-Croatian](https://sh.wikipedia.org/wiki/Such_Is_Life_(roman)) Wikipedias. Abstract Wikipedia could in principle quickly fill this gap in the 300+ other language editions, so long as there is some abstract content about the novel in Wikidata, and renderers available for each language in Wikifunctions.

Wikifunctions itself has a complex architecture, because it is a very unusual programming system. Programming systems normally work with text files. To write software, programmers simply open the files containing the source code and alter the text. Software can be distributed among programmers by sharing the source code files. Wikifunctions is an unusual programming system because it works with database records. Instead of storing software as text files in a folder system, Wikifunctions stores a complex amalgam of “functions” and their “implementations” in a vast database. Not only this, but because it is a Wikimedia project, Wikifunctions is a *wiki*, just like Wikipedia or Wikidata. Most large software projects use complex version control software to ensure that multiple programmers working on an application don’t “clobber” one another’s code. Wikifunctions will instead rely on Wikipedia’s normal editing tools.

The overall structure of the Wikilamba software is shown in [Figure 1](#fig-wikilambda-architecture). The part that implements a new programming language is the “function orchestrator,” whose source code I analyse in [Section 5](#sec-code). The orchestrator not only implements a new programming language, but serves as the main entry point for the Wikilamdba software.

|  |
| --- |
| Figure 1: High-level model of the Wikilambda Extension. Source: “Extension:WikiLambda” (2024). By jdforrester (Wikimedia Foundation) [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/deed.en). |

To understand the [Figure 1](#fig-wikilambda-architecture), consider an example. Imagine that an editor of German Wikipedia wishes to address a gaping hole in their encylopaedia, by importing the Abstract Wikipedia article for *Such is Life*. German Wikipedia would be running on a “MediaWiki app server.” When the German editor request the abstract article, a “Parser function request” would be sent to Wikifunctions. The function orchestrator would then execute the German renderer on the abstract content for *Such is Life*. To do this, it would need to make its own “Data fetch” requests to other “MediaWiki app servers,” for example, to request the abstract content from Wikidata, or to retrieve images from Wikimedia Commons. The process of rendering German text will undoubtedly be very complex, and the German renderer will draw together many different smaller functions stored in Wikifunctions. The “function evaluators” have the task of actually running each of these little functions when the “orchestrator” tells them to. Perhaps the function that conjugates German verbs is written in JavaScript, so the JavaScript evaluator will be used, while the function that pluralises German nouns is written in Python, so the Python evaluator will be used. Some functions will be written in the Wikilambda language itself, in which case the orchestrator can evaluate the function itself, without invoking another function evaluator. When all this work is complete, the rendered German text will be returned to German Wikipedia, and included in the encyclopaedia.

To try and improve the system’s performance, Wikifunctions will “memcache” all its work. It will essentially remember its answer to every request. If a second request to render *Such is Life* into German arrives, it will check to see if it has rendered it already. If a fresh enough German text is available in the “memcache,” it will simply return this cached version rather than going through the whole process of rendering the German text again. Similarly, when it is applying the German renderer to the abstract content of the article, it will check each time it uses a function whether it has used that function before. If the function has been run on the same inputs before, it will retrieve the answer from the cache. Only if there is a “cache miss” will a function evalutor be asked to re-evaluate the function ([Figure 1](#fig-wikilambda-architecture)).

At the time of writing, there are no renderers in the Wikifunctions database, and Wikidata has not yet been upgraded with the “template language” in which contributors will one day compose abstract articles.[[2]](#footnote-34) What does exist is an early version of the Wikilambda language, which anyone can use at https://wikifunctions.org.

The Wikilamdba language plays a glue-role in the broader Abstract Wikipedia/Wikifunctions project. Wikifunctions is a database of “functions.” Each function performs a single task, for example to [get the plural form of a word](https://www.wikifunctions.org/view/en/Z19260) or [to see which languages are available for a given “lexeme” in Wikidata](https://www.wikifunctions.org/view/en/Z19276). Contributors can provide “implementations” of these individual functions in a range of languages (currently JavaScript and Python); or they can implement a function by “composing” functions that already exist in the Wikifunctions database, using the Wikilambda language.[[3]](#footnote-37) It is this ability to glue or compose multiple functions together that turns Wikifunctions from a database of code snippets into a genuine “programming system” that can be “**collaboratively** grown and maintained,” and support the complex software required to realise Abstract Wikipedia (“Wikifunctions:Vision” 2023). Abstract Wikipedia’s renderers will need to pluralise words *and* see whether given words are available in the target language. These and many other functions will need to be composed together to build up the extremely complex Natural Language Generation (NLG) software that will render Abstract Wikipedia into human-readable reality.[[4]](#footnote-38)

This is a remarkable vision for the future of programming. In the following two sections, I evaluate the project’s claims for the [universality](#sec-universality) and [perfection](#sec-perfection) of Wikilambda. This analysis will provides essential context for the close analysis of the function orchestrator in [Section 5](#sec-code).

## 3 Universality: functions for all!

The primary aim of Wikilambda is to enable wiki-programming. Just as Wikipedia is the encyclopaedia anyone can edit, Wikifunctions will be the *library*, and Abstract Wikipedia the *application*, that anyone can program. The Wikilambda developers identify two barriers to their aim: (1) most programming languages use English vocabulary, but most people don’t speak English; (2) most programming systems require you to be a skilled programmer to contribute, but most people aren’t skilled programmers. The developers promise to crash these barriers, and create the first truly universal programming system.

### 3.1 All languages in one

Imagine a programming system that truly is **accessible**, one that is open not only to speakers of English but to billions of people more who will not have to learn English first in order to learn how to program. (“Wikifunctions:Vision” 2023)

Wikilambda is designed to support programmers who speak all the world’s languages. Nearly all programming languages, complain the Wikifunctions developers, are based on English. Even languages such as Python, Ruby and Lua use English as a source of keywords, despite the fact that their inventors respectively speak Dutch, Japanese and Portuguese as their first language. Programmers learning these languages need to use (apparently) English words such as function, table, and if to write their software. According to the Wikifunctions team, the Anglocentrism of programming makes it inaccessible. To learn these languages, I need to “learn English first.” Otherwise, how could I understand what if does?

To support programmers who don’t speak English, Wikifunctions stores all functions and their implementations under Z-codes. if, for example, is object [Z802](https://wikifunctions.org/wiki/Z802) in the database. When a user visits this page in Wikifunctions, the Z-code is automatically transliterated into their preferred language. An English-speaking visitor will see if, while a Bangla-speaking visitor will see যদি . Not only this, but all the component parts of the function are translated as well. An English speaker will see that the if function requires three inputs, called condition, then and else.[[5]](#footnote-41) A Bangla speaker will see these three inputs as শর্ত, যদি সত্য তাহলে and যদি সত্য না তাহলে ([Figure 2](#fig-bangla-if)).

|  |
| --- |
| Figure 2: Function Z802 in Wikifunctions in Bangla. Visitors can run the function by filling in the three inputs in the large box in the right of the figure, then pressing the blue button. Source: https://www.wikifunctions.org/wiki/Z802?uselang=bn&oldid=132257 |

In their response to the Google Fellows’ Evaluation, the Wikilambda developers spell out the political consequences of Anglocentric programming languages.

A solution designed by a small group of Westerners is likely to produce a system that replicates the trends of an imperialist English-focused Western-thinking industry. Existing tools tell a one-voice story; they are built by the same people (in a socio-cultural sense) and produce the same outcome, which is crafted to the needs of their creators (or by the limits of their understanding). Tools are then used to build tools, which will be again used to build more tools; step by step, every level of architectural decision-making limits more and more the space that can be benefitted by these efforts. (Blanton et al. 2022)

The Wikilambda developers echo common arguments of postcolonial theory. Language is not only a “means of communication,” but also a “carrier of culture” (Thiong’o 1986, 13). When a speaker of a local language is forced to adopt an imperial tongue such as English, it results in a kind of “colonial alienation”: in everyday life, the local speaker uses their mother tongue, but the language of learning and technology is the imperial tongue (Thiong’o 1986, 16). The imperial tongue may give access to new knowledge, but it also carries its own culture, and forces the local speaker to adopt the imperial point of view. The Wikilambda developers extend this argument into the world of software. Software is rooted in the English language, and so imposes Anglo-American culture on the world. This limits the “space” in which programmers can dream up new software. This is a problem in general, but also a problem specifically for Abstract Wikipedia, which will require new and highly innovative software to succeed. The necessary ideas may be lurking in the mind of a Gikuyu or Tok Pisin speaker, who hithterto could not express their ideas in any conventional programming language.

Wikifunctions is not the first programming system that contests the dominance of English. Marino (2020, 151–54) describes several prior efforts. When Grace Murray Hopper designed FLOW-MATIC in the 1950s, she wrote it to run using English, French or German vocabulary. More recently, Ramsey Nasser has designed قلب (or ’alb), a variant of Scheme that runs right-to-left and uses Arabic characters, while in Hawai’i, the ’Anu’u project is translating C# into into Hawai’ian. Marino (2020, 132–33) agrees with the Wikilambda developers that English vocabulary can have a “colonizing effect” on programmers.

Despite this, there is a problem at the heart of postcolonial language design. The fact is that “words in code simply do not behave the way they do in spoken or written language” (Marino 2020, 145–47). For example, the function mapcar in the Scheme programming language has nothing to do with either maps or cars. The mapcar function in fact *applies the passed operator to the first item of each list in a list of lists*. Similarly, many languages have a print function, which has nothing to do with printing, but instead causes text to appear on screen. This reflects a deeper point. It may be true that programming languages such as Python and Lua use English vocabulary, but Python and Lua are not, in fact, English. They are themselves languages, cherished by particular communities of programmers, with their own idioms and styles and values. It may be true that you need to learn English before you can learn Python or Lua, but this is not because of the English vocabulary of those languages. It is because most of the books and tutorials and documentation are in English, and the international community of programmers use English as a *lingua franca*. The tricky part of learning to program is *learning to program*. When do you need to use the if function? Being able to write যদি instead of if won’t help. What *would* help is a good book, teacher or documentation in your native language.

Wikilambda does provide facilities for multilingual documentation. Functions can be described in any language, and the visitor will automatically see the description in their own language. But it unlikely that Wikifunctions will ever free programmers from the dominance of English. First, it is likely that Wikifunctions’ discussion pages and policy documents will be dominated by English, as they are on Wikidata (Ford and Iliadis 2023). Second, the creators intend for the platform to mostly rely on “native language” implementations of functions in the database, by which they mean implementations in existing programming languages such as Python and JavaScript. Since these are the very languages whose Anglocentrism they decry, the Wikilamda’s teams critiques apply to their own system too.

### 3.2 Functional literacy

As well as linguistic universality, the Wikilambda team also aim for a kind of programming universality. Their

Most languages invented to democratise programming have not been “purely functional,” but have been either imperative languages like BASIC and Scratch, or object-oriented languages like Smalltalk and Processing (Vee 2017; Blackwell 2024). Imperative and object-oriented languages are good for beginners because they provide intuitive metaphors. Imperative languages allow you to command the computer to do things. Object-oriented languages allow you to simulate the real world by defining “objects” that represent actual entities. In both the imperative and object-oriented cases, democratic languages have typically focussed on highly aesthetic aspects of programming, such as drawing graphics, manipulating text, or controlling robots (Montfort 2013). In such aesthetic situations, the beginning programmer can literally see, hear, or feel the effects of their code. This multisensory experience provides both feedback and encouragement to the learner.

Purely functional languages present programmers with a highly abstract metaphor for the computer. Rather than commanding the computer to do things, or simulating a world of familiar objects, the functional programmer defines individual mathematical functions, and then “composes” them into complex wholes (Hudak 1989). While it is possible to teach purely functional programming in an aesthetic context (e.g. Hudak 2000), most functional languages designed for beginners have not been *purely* functional languages. Examples include LOGO and Racket, both of which are “functional” in a broad sense, but are *im*pure because they allow “side-effects,” and thus can support the more familiar imperative and object-oriented styles of programming.

The Wikilambda is *purely* functional for technical reasons, but the Wikilambda team have nonetheless made a universalist case for this aspect of the language. As the Wikilambda team explain, function “purity” is essential if you want to store functions in a database, and execute them individually on demand; since “pure” functions *only* depend upon the inputs that are given to them, they have the useful property of “determinism” (“Wikifunctions:Determinism” 2024). In their evaluation of the system, the Google Fellows do not dispute this, but raise a different objection: programming with pure functions is “less intuitive for the average programmer,” and would be especially inappropriate for the kind of software needed for Abstract Wikipedia (Livneh et al. 2022). How can this system be universal and inclusive if it uses a counter-intuitive and inappropriate programming paradigm?

The Wikilambda team’s rebuttal is striking. Functional programming is only hard because programmers have “already absorbed the imperative programming paradigm” from their education. “For people who have no prior relationship with programming, a functional paradigm might be as intuitive as an imperative one, if not more so.” (Blanton et al. 2022) This argument is hard to credit from a commonsense perspective. If purely functional programming is so intuitive, why have computer literacy advocates never adopted it before? From an ideological perspective, the argument is more interesting. By establishing a brand-new language in an unusual paradigm, the Wikilambda team imply, Wikilambda will sweep away existing inequalities of skill and knowledge. In this new world, everyone will be a beginner. The dominant “imperative” paradigm will go the same way as the dominant “English” language.

### 3.3 *The Lord of the Rings*

Twice in there response to the Google Fellows, the Wikilambda team use a nerdy metaphor. The Wikilamdba team do not want a “One Ring solution” (Blanton et al. 2022). Instead, they want to allow contributors to experiment with many solutions. The “One Ring” is of course the ring of power forged by the villanous Sauron in J.R.R. Tolkein’s *The Lord of the Rings* (*LOTR*). The influence of *LOTR* has been immense in the history of computing. In the early days of the internet, there were so many computers called “Frodo” that *LOTR* became a vivid illustration of the need for a domain name system (Abbate 1999). *LOTR* remains one of the most-read and discussed books among software engineers (Dillon and Schaffer-Goddard 2023).

The Wikilambda team position themselves as a band of hobbits opposing the mighty Saurons of Big Tech and the English language. But in their choice of reference, they unwittingly reveal a contradiction at the heart of their universalist credo. Wikilambda *is* a “One Ring solution.” It is a single, global, unified programming system, that will encompass potentially all languages. *LOTR* itself is the opposite of Wikilambda. It depicts a world of extreme linguistic and ethnic diversity. As is well known, Tolkien wrote the book by first inventing several fictional languages, and then writing fictional histories to explain the development and diversification of these languages. By contrast, Wikilambda is a single language that attempts to unite many diverse ones. To resolve this contradiction, the Wikilambda team make a paradoxical argument for its perfection. This language is so simple that it is in fact no language at all.

## 4 Perfection: the language that is not a language

But it is perhaps nothing more than our ‘democratic’ illusion to imagine that perfection must imply universality. (Eco 1995, 190)

What would it mean for a programming language to be ‘perfect’? The perfect languages surveyed by Eco (1995) attempt to align their vocabulary and syntax with nature. The vocabulary of a perfect language should include exactly one word for each natural kind. The grammar of a perfect language should all the speaker or writer to combine these words in exactly the way that these natural kinds are combined in nature. The language itself ought to be a mirror of the world, such that any fact about the world can be expressed concisely and unambiguously. Of course the whole project relies on the idea that there are a finite set of natural kinds, and that the rules for their combination are known.

Programming languages are obviously more limited than natural languages, because they exist for a specific purpose. A programming language is designed to program a computer. Thus to be perfect, a programming language only needs one word for each natural kind *of computation*, and grammatical rules that allow these natural kinds to be combined *as a computer is able to combine things*. What are the natural kinds of computation? What are the primitive elements of all computer programs, and what are the ways they can be combined? In this section, I try to see how Wikilambda answers these questions, by comparing its language to Scheme, a language developed by AI researchers in the 1970s that resembles the Wikilambda language in crucial respects.

There have been many attempts to answer these questions, and to create a universally valid model of computation. The two most famous attempts are Alan Turing’s universal machine (1936), and Alonzo Church’s λ-calculus (1932). The “universality” of Turing’s machine is different to the “universality” discussed above in [Section 3](#sec-universality). Turing’s machine is “universal” because it is theoretically able to perform any computation. Church’s λ-calculus has the same property (Turing 1937). Wikilambda invokes Turing and Church to stake its own claim for perfection. The software is named after Church’s λ-calculus. The Wikifunctions database incorporates a stylized lambda into its logo ([Figure 3](#fig-wikifunctions-logo)). The Google Fellows report that “Turing-completeness” is an explicit requirement of the system (Livneh et al. 2022), though it is Church’s λ-calculus that has more directly inspired the design of the Wikilambda language.

|  |
| --- |
| Figure 3: The Wikifunctions logo, incorporating lambda. Based upon the submissions of NGC 54, Jon Harald Søby and Stevenliuyi. [CC BY-SA 4.0.](https://commons.wikimedia.org/w/index.php?curid=112952316) |

### 4.1 Wikilambda’s claim to perfection

The Wikilambda team do not simply claim that their language is Turing-complete. Many languages are Turing-complete, including deliberately unusable languages like Brainfuck (Mateas and Montfort 2005), and languages not intended for programming, such as the typesetting system TeX. Indeed, philosophers have debated whether there is *any* physical object which is *not* Turing-complete (Chalmers 1996). The Wikilambda team make a stronger claim than Turing-completeness. Not only is their language able to express any computation, but its syntax mimics the natural form of computation itself: “… we do not really think of function composition as a new programming language, but rather as an abstract syntax tree to compose function calls together …” (Blanton et al. 2022). Taken at its strongest, this statement implies that the Wikilambda language is perfect. Conventional programming languages are parsed into “abstract syntax trees,” which are then converted into machine code. Wikilambda strips away everything on top of the abstract syntax tree, giving the user direct access to the program as the computer sees it. It is not really a “language,” because it is immediate. Nothing stands between the programmer and the actual structure of their program. When the user manipulates this language, they manipulate the abstract syntax tree itself.

The Wikilambda team make this case with considerable uncertainty, and hedge their arguments. They wonder whether the difference between a “language” and an “abstract syntax tree” is merely “terminological,” and then ironise their argument with an emoticon: “… the function model in the ZObject system merely allows nesting function calls and express[ing] data. How hard can that be? ;)” (Blanton et al. 2022). On the page “What Wikifunctions is Not,” they insist again that this is “Not a Programming Language,” stressing that Wikilambda’s function composition “will be extremely limited compared to most other programming languages, since we only allow for nested function calls and that’s it.” (“Wikifunctions:What Wikifunctions Is Not” 2024) Elsewhere they do, however, admit that they are involved in a tricky process of language design. In the README for the “function orchestrator,” they admit that “the orchestrator kind of implements a functional programming language,” and apologise for the resulting “complexity” of the code (Massaro et al. 2024b). This uncertainty in the documentation suggests that the Wikilambda team are unsure what exactly they are making. Are they “kind of” making a very “limited” language that “merely” or “only” allows nesting of functions? Or is that just an empty “terminological” question?

In fact the Wikilambda language is not an “abstract syntax tree,” and there is nothing “mere” or “limited” about it. The supposed “limitations” of the language are actually deliberate decisions about what features to implement within it. Comparing the Wikilambda language with Scheme reveals some of these important decisions.

### 4.2 Wikilambda and Scheme compared

Scheme is an influential dialect of the LISP programming language. The Wikilambda team also consider their language to be a dialect of LISP: “If there was a TLDR of the project it would probably be ‘something like LISP in JSON’.” (“Wikifunctions:Function Model” 2024) Like Scheme, the Wikilambda language is very simple. In this simplicity, Scheme and Wikilambda differ from other dialects of LISP, such as Common LISP and Clojure, which include additional features to support common programming tasks (Steele and Gabriel 1996; Hickey 2020).

The inventors of Scheme present it as an extension of the λ-calculus , and claim a certain kind of perfection for the language. Scheme extends the λ-calculus with just enough features to make it “convenient,” while retaining the power of the λ-calculus to represent the inner structure of a computation. Thus Scheme is not merely a programming language, but a powerful “meta-language” that can be used to study the nature of computation itself (Sussman and Steele Jr. 1975; Steele Jr. and Sussman 1976). In a series of famous memos, known as the “Lambda Papers,” Scheme’s creators show how the language can easily simulate many different programming paradigms, including continuation-passing, message-passing, imperative and structured programming (Steele Jr. 1976, 1977; Steele Jr. and Sussman 1976). Thus Scheme’s perfection is twofold: it exposes the fundamentals of computation by resembling the λ-calculus, while also having the ability to morph into many other programming languages.

The Wikilamba language is even simpler than Scheme. This simplicity has led its creators into some difficult design choices. We can see the crucial differences by comparing a short program in each language. I have chosen the factorial function, a simple function often used to demonstrate recursive programming to beginners. In mathematical notation, “factorial of n” is written . To compute the factorial of a number, you multiply all the natural numbers up to . For example, to compute the factorial of 6:

It is possible to describe the factorial function using two simple rules, which can then be used to implement it in a programming language. To find these two rules, we first observe a pattern in the function:

From this, we can derive the first rule

This is a “recursive” rule, because the factorial function is defined in terms of itself. The factorial of is equal to *the factorial* of multiplied by itself. This recursion could go on forever, so we need a second rule to stop subtracting from . The required rule is simple: . When is equal to we stop subtracting , and we can multiply all the terms together to get the final answer.[[6]](#footnote-57)

[Listing 1](#lst-scheme-factorial) shows how to write the factorial function in Scheme, using these two rules.

|  |
| --- |
| Listing 1: A recursive implementation of factorial in Scheme  (define (factorial n)  (if (= n 0) ; condition  1 ; consequent  (\* n (factorial (- n 1))))) ; alternative |

On line 1, we define a new function called factorial, which takes one input, called n. On line 2, we decide which rule to apply. If n is equal to zero, then the first option will be chosen (the *consequent*), and the function will simply output the value 1. If n is equal to any other number, then the second option will be chosen (the *alternative*). In this case, n will be multiplied by the output of the factorial function applied to n - 1. In Scheme, as in many programming languages, \* is the multiplication symbol. Thus the expression (\* n (factorial (- n 1))) is exactly equivalent to .

[Listing 2](#lst-wikilambda-factorial) is the current Wikilambda implementation of the factorial function in the Wikifunctions database (“Factorial, Composition (Z13863)” 2024). This is its representation in “canonical form,” which is the format used within the system to represent Wikilambda code. Users are not exposed to the code directly. Instead, they see a visual representation of the code, with the various Z-codes translated into their preferred language ([Figure 4](#fig-wikilambda-factorial)).

|  |
| --- |
| Listing 2: A recursive implementation of factorial in Wikilambda  {  "Z1K1": "Z2",  "Z2K1": {  "Z1K1": "Z6",  "Z6K1": "Z13863"  },  "Z2K2": {  "Z1K1": "Z14",  "Z14K1": "Z13667",  "Z14K2": {  "Z1K1": "Z7",  "Z7K1": "Z802",  "Z802K1": {  "Z1K1": "Z7",  "Z7K1": "Z13522",  "Z13522K1": {  "Z1K1": "Z18",  "Z18K1": "Z13667K1"  },  "Z13522K2": {  "Z1K1": "Z13518",  "Z13518K1": "0"  }  },  "Z802K2": {  "Z1K1": "Z7",  "Z7K1": "Z13636",  "Z13636K1": {  "Z1K1": "Z13518",  "Z13518K1": "1"  }  },  "Z802K3": {  "Z1K1": "Z7",  "Z7K1": "Z13539",  "Z13539K1": {  "Z1K1": "Z18",  "Z18K1": "Z13667K1"  },  "Z13539K2": {  "Z1K1": "Z7",  "Z7K1": "Z13667",  "Z13667K1": {  "Z1K1": "Z7",  "Z7K1": "Z13582",  "Z13582K1": {  "Z1K1": "Z13518",  "Z13518K1": {  "Z1K1": "Z18",  "Z18K1": "Z13667K1"  }  }  }  }  }  }  },  "Z2K3": {  "Z1K1": "Z12",  "Z12K1": [  "Z11",  {  "Z1K1": "Z11",  "Z11K1": "Z1002",  "Z11K2": "factorial, composition"  }  ]  },  "Z2K4": {  "Z1K1": "Z32",  "Z32K1": [  "Z31"  ]  },  "Z2K5": {  "Z1K1": "Z12",  "Z12K1": [  "Z11"  ]  } } |

|  |
| --- |
| Figure 4: The Wikilambda factorial function in the user interface |

The Wikilambda code is extremely verbose, partly because of the extreme simplicity of the language. In what follows, I analyse selected lines of [Listing 2](#lst-wikilambda-factorial) to illustrate some similarities and differences with [Listing 1](#lst-scheme-factorial).

The first crucial similarity is that both Scheme and Wikilambda use prefix notation.[[7]](#footnote-65) Ordinary mathematical notation uses *in*fix notation, with optional parentheses. Consider the expression . The and symbols go *in between* the numerals, and are therefore *in*fixes. Parentheses are optional because there is a rule that tells us to perform the multiplication before the addition. There is no need to write . In both Scheme and Wikilambda, the mathematical operator goes *before* the numerals, and parentheses are compulsory. Here is the multiplication in [Listing 2](#lst-wikilambda-factorial):

{  
 "Z1K1": "Z7",  
 "Z7K1": "Z13539",  
 "Z13539K1": { ... },  
 "Z13539K2": { ... }  
}

Line 2

A [Z7](https://www.wikifunctions.org/wiki/Z7) is a “Function Call.” The next few lines will tell Wikilambda which function to “call,” and what its inputs are.

Line 3

First comes the function. [Z13539](https://www.wikifunctions.org/wiki/Z13539) is the function “multiply two natural numbers.”

Line 4

Next comes the first number, also called the “multiplicand.” The label has two parts, “Z13539,” referring to the multiplication function, and “K1,” meaning “the first input.”

Line 5

Finally comes the second number, also called the “multiplier.” The label again has two parts, “Z13539” for the function, and “K2” for “second input.”

Line 6

The expression is fully parenthised. Because Wikilamdba uses JSON as its file format, it uses curly braces {} rather than the traditional parentheses ().

This excerpt is far more verbose than the equivalent section of [Listing 1](#lst-scheme-factorial):

(\* ... ...)

The verbosity is due to Wikilambda’s extreme simplicity. Scheme has some rudimentary syntactical rules that are absent from Wikilamda. For example, in Wikilambda, the multiplication is explicitly labelled as a [Z7:Function Call](https://www.wikifunctions.org/wiki/Z7), so that the system knows that the user intends to call a function. In Scheme, there is no need to label the expression as a function call: it is simply assumed that the first symbol in the parentheses is a function, which will be applied to whatever symbols come after it. In Wikilambda, each part of the expression is explicitly labelled. We can see from the labels which is the first input (“K1”) and which is the second input (“K2”) to the function. In Scheme, this labelling is handled by the programming system. The writer of code can simply write the two inputs to the function, and the Scheme interpreter will do the work of labelling them as “input 1” and “input 2.”

This explicit labelling does justify the idea that programming Wikilambda allows users to directly manipulate the “abstract syntax tree.” Work that is usually done by the program interpreting the code is instead being done by the human programmer. The labelling also means that the order of parts of the program is strictly speaking irrelevant. Although the code is presented in prefix notation, with the function first and it inputs in their proper order second, the function orchestrator actually uses the provided labels to determine what the function and its inputs are. It would still evaluate the code correctly if the lines were jumbled up. In Scheme, however, the interpreter relies on the symbols being in their proper order.

There are many other differences between Scheme and Wikilambda, but one further contrast will serve to illustrate the tricky problems of language design faced by the Wikilambda programmers. The crucial contrast lies in the if function, which enables the factorial function to be recursive. Recall that if is necessary in order to select which rule to apply: if , one rule applies, else the other applies.

{  
 "Z1K1": "Z7",  
 "Z7K1": "Z802",  
 "Z802K1": { ... },  
 "Z802K2": { ... },  
 "Z802K3": { ... }  
}

Lines 2-3

This is the most important aspect of this expression: the “If” function ([Z802](https://www.wikifunctions.org/wiki/Z802)) is just an ordinary function like any other, invoked by a “Function Call” ([Z7](https://www.wikifunctions.org/wiki/Z7)). As we will see, when the Wikilambda team opted to treat all control strcutres as simple functions, they obliged themselves to implement “lazy evaluation,” which has significant consequences for the entire system (“Wikifunctions:Function Model” 2024; “Abstract Wikipedia/Semantics of Wikifunctions” 2024).

Line 4

The first input (“K1”) is the *condition*. In [Listing 2](#lst-wikilambda-factorial), the condition checks whether .

Line 5

The second input (“K2”) is the *consequent* or *then* expression: what the function will do if the “condition” is true.

Line 6

The third input (“K3”) is the *alternative* or *else* expression: what the function will do if the “condition” is false.

To understand the need for “lazy evaluation” in Wikilambda, we can again contrast it with Scheme. In Scheme, if is not a normal function, but rather a “special form” which behaves differently to an ordinary function. When an ordinary function is called in Scheme, all its inputs are evaluated before the function is then applied to them. This is called “applicative order” evaluation. For example, imagine we wanted to calculate

In Scheme, that is written:

(\* (+ 3 1) (/ 4 2))

We start at \*, the multiplication operator. Before we apply this operator, we first “evaluate the inputs,” which means “calculate (+ 3 1) and (/ 4 2)”. Then the expression becomes:

(\* 4 2)

Once the values of the inputs are known, we perform the multiplication:

8

Now imagine that we wanted to calculate . To calculate using the code in [Listing 1](#lst-scheme-factorial), we write:

(factorial 1)

This becomes

(if (= 1 0)  
 1  
 (\* 1 (factorial (- 1 1))))

Now, if we follow the usual rule of applicative order evaluation, before we can apply the if function, we need to evaluate all its inputs. The function has three inputs, namely the *condition*, the *consequent* and the *alternative*. Let us evaluate them.

(if #f  
 1  
 (\* 1 (factorial 0)))

Line 1

It is false (#f) that n is equal to 1.

Line 2

The value of 1 is one. Evaluating this expression is trivial.

Line 3

This is a complex expression. Before we can apply the \*, we need to evaluate its two inputs. The first input is simply 1. The second input, (factorial (- 1 1)) evaluates to (factorial 0).

We still haven’t fully evaluated the inputs to if, so now we need to evaluate (factorial 0). Then we will be able to multiply it by 1 to finish evaluating that last line of the code. (factorial 0) becomes:

(if (= 0 0) ; evaluates to #t, "true"  
 1 ; evaluates to 1  
 (\* 0 (factorial (- 0 1)))

Ah, we have another complex expression, (factorial (- 0 1)). Before we can apply if to its three inputs, we need to work out the value of (factorial (- 0 1)), which is of course (factorial -1). Before we can evaluate that, we need to work out the value of (factorial (- -1 1)), which is of course (factorial -2), and so on. Our code is broken. It will never stop subtracting from , and will never return us an answer.

The solution in Scheme is to treat if as a “special form.” When we call an ordinary function in Scheme, all the inputs are evaluated straight away. But when we use if, only the *condition* is evaluated. Then, depending on the outcome of that, *either* the consequent *or* the alternative is evaluated. Thus in fact the Scheme code does work. When we reach (factorial 0), the following will happen:

; first step  
(if (= 0 0)  
 1 ; ignore  
 (\* 0 (factorial (- 0 1)))) ; ignore  
  
; second step  
(if #t  
 1  
 (\* 0 (factorial (- 0 1))))  
  
; final answer   
1

Line 2

First evaluate the condition.

Line 7

It is true (#t).

Line 8

Therefore evaluate the *consequent*.

Line 9

We ignore the *alternative*, and don’t try to evaluate (factorial -1).

Line 12

The output is 1.

Sussman and Steele Jr. (1975, 26) argue that any “practical lambda calculus interpreter” must include a mixture of applicative-order functions like \*, whose inputs are evaluated straightaway, and “lazy” or “normal-order” functions like if, whose inputs are only evaluated when needed. Aside from if, Scheme provides several other “special forms” to support essential programming constructs (“Special Forms” 1997).

Instead of providing “special forms” to users, Wikilambda works throughout like Scheme’s if. All Wikilambda functions are “lazily evaluated.” The inputs to a function are only ever evaluated when they are used inside the function. And crucially, each expression is evaluated only once. If, for example, Wikilambda has to calculate the value of , it will remember that the answer is , and simply fetch the answer from memory if it sees again. In fact there are no “special forms” at all in Wikilambda, except for some “built-in” functions that are currently unavailable to end-users, and the “Quote” type ([Z99](https://www.wikifunctions.org/view/en/Z99)). Not only does Wikilambda adopt lazy evaluation, but it provides no convenient way to avoid it, even when it may be convenient to do so.

By adopting lazy evaluation, Wikilambda sharply departs from other dialects of LISP, which are generally applicative order languages like Scheme. There are many known advantages to lazy evaluation. It can increase the modularity of code, and enable the use of “infinite” data structures (Hudak 1989, 385). But very few mainstream languages use it as the default evaluation strategy, because it is difficult for language designers to implement efficiently and correctly. The most famous lazily evaluated language is Haskell, which is in many respects the opposite of Scheme or Wikilambda. Rather than a small, simple language designed to give direct access to the abstract syntax tree, it is a complex language with many syntactic rules and a powerful “optimising” compiler that transforms the code into performant software. By adopting lazy evaluation, Wikilambda seems to be heading in two directions at once: towards the simplicity of Scheme, and the complexity of high-tech functional languages like Haskell.

In fact, Scheme itself is not so simple. Aside from “special forms,” Scheme implements many additional features, such as tail-call optimisation, garbage collection, pattern matching and an object system, which make Scheme’s simple syntax useable in practice. Its surface simplicity is the result of a lot of careful software engineering under the hood. The source code for MIT/GNU Scheme, one of the major Scheme implementations, is nearly 1.4 million lines long. By contrast, the “function orchestrator” is a little over 5400 lines.[[8]](#footnote-74) The function orchestrator is written in a “natural and intuitive” style, to quote the polite phrasing of the Google Fellows. By “natural and intuitive,” they appear to mean *simple and naive*, and they go on to suggest that the system needs to be re-written to cope with serious computations (“Abstract Wikipedia/Semantics of Wikifunctions” 2024). The Wikilambda team rest their hopes on two factors. First, that functions in the Wikifunctions database will mostly be implemented in existing languages that manage performance and memory issues; and second, that results from calling Wikifunctions will be cached, so that the system can rapidly return pre-saved answers as often as possible. In both these cases, the function orchestrator and its new language would have little to do. The unnamed Wikilambda language, the glue between implementations, the shaker of Anglocentrism, the language that is not a language, the pure expression of the abstract syntax tree, is both the keystone and the vanishing point of the system.

## 5 Code: a most perfect and universal hack

Through the production of new forms of abstraction, the hacker class produces the possibility of the future. (Wark 2004, sec. 077)

Dear reader, if you have solved similar problems–e.g., if you implemented programming languages before–and know the better way, have at it :). (“README,” Massaro et al. 2024b)

What makes the function orchestrator so “natural and intuitive”? What is the “nature” it reflects? What are the “intuitions” it appeals to? Critical Code Studies is still a young field, and scholars are still developing the critical procedures to answer such questions. In this section, I draw inspiration from *The Structure and Interpretation of Computer Programs* (Sussman, Sussman, and Abelson 1996). In SICP, the authors describe how programmers build up “abstractions” that allow them to describe the problem domain of their software. The key question in program design is where to place the “abstraction barriers.” From a literary perspective, an abstraction barrier is a layer of metaphor. The metaphors in an abstraction layer allow the programmer to describe elements of the program as though they are things and processes in the real world. They can ignore the concrete elements below the layer of metaphor (e.g. electrons whizzing about in the CPU) and focus on metaphorical objects above the layer (e.g. with ZObjects in the Wikifunctions database). In this section, I consider some of the metaphors that structure the code of the function orchestrator. The “naturalness” and “intuitiveness” of these metaphors vividly communicate the underlying ideals of the Wikilambda language.

To illustrate the metaphorical nature of code, I begin with a short example from the “function evalutor,” an even simpler piece of software than the “function orchestrator.” The role of the function evaluator is to run implementations in the Wikifunctions database. It does so by maintaining an ExecutorPool. Executors are little pieces of software than can run particular code: if a function in Wikifunctions is implemented in Python, it will be executed by a Python executor; if it is implemented in JavaScript, by a JavaScript executor, and so on.

The Wikilambda developers clearly enjoyed themselves when they wrote the code that empties the ExecutorPool at the end of a session. To empty the ExecutorPool, the programmer needs to drainTheSwamp. When they drain the swamp, the program creates a hitList of executors in the pool, and then uses killExecutors to delete them all. The metaphor becomes gentler in the next phase of the process. To “kill” an individual executor, the program calls on the executor to use its own immaHeadOut method, which offers the suicidal executor a dignified exit ([Listing 3](#lst-immaHeadOut)). The executor destroys the childProcess it has been using to execute code, and then posts an obituary\_ so the rest of the program knows it has done so.

|  |
| --- |
| Listing 3: The method that ‘kills’ an unneeded executor on the ‘hitList’ after a call to ‘drainTheSwamp’. Source: src/Executor.js, in Massaro et al. (2024a)  async immaHeadOut() {  // Kill the executor child process if it has survived.  if ( this.childProcess\_ !== null && !this.childProcess\_.killed ) {  await killProcessFamily( this.childProcess\_.pid );  }   // Announce the death of the child.  this.obituary\_(); } |

Pool, swamp, hitman, netspeak, funeral—the Wikilambda software is written with a mixture of bruality and tenderness that bespeaks the passion of its programmers. There is nothing clinical or professional about code that riffs on Trumpian rhetoric and mourns the euthanasia of children. This is vivid code, full of metaphors that the Wikilambda developers use to communicate with one another. Wikilambda’s “function evaluator” is a simple program, which performs tasks familiar to any web developer. The “function orchestrator” is a much more complex and unusual piece of software, which the Wikilambda developers admit they do not fully understand themselves. Its metaphors are accordingly more vital to the project, and less confident in their application, than those of the gleeful “evaluator.”

### 5.1 First metaphor: the orchestra

The entire function orchestrator is built around a central metaphor: orchestration. When a “parser function request” arrives at Wikifunctions ([Figure 1](#fig-wikilambda-architecture)), a piece of Wikilambda code such as [Listing 2](#lst-wikilambda-factorial) is retrieved from the database, and sent to the function orchestrator. The function orchestrator calls the orchestrate function on the code, and the process begins.

This metaphor is unusual. There is no orchestrate function in major implementations of Python, JavaScript and Scheme, though the word sometimes crops up in documentation.[[9]](#footnote-78) All these languages use the metaphor of “interpretation.” Indeed, the function orchestrator contains an “interpreter” in the usual computer-programming sense of the word. The function orchestrator stands between the raw Wikilambda code (e.g. [Listing 2](#lst-wikilambda-factorial)) and the machine, which understands only the machine language hard-coded into its CPU. The function orchestrator interprets the code so that the machine can understand it.

Why depart from the dominant metaphor, and call this an “orchestrator”? The metaphor probably reflects the Wikilambda developers’ reluctance to view Wikilambda as a programming language. An “interpreter” is a translator, who works with words. An “orchestrator” is a composer, who fleshes out a musical composition by assigning parts to the instruments of an orchestra. It is hard to know how literally to take this metaphor, but it does certainly harmonise with the notion of “function composition.” In this extended metaphor, the contributor to Wikifunctions is the original composer, who creates the spine of the function, and then Wikilambda realises the composition by adding in the orchestration.

[Listing 4](#lst-orchestrate-callTuples) indicates what the activity of “orchestration” is like. In this extract from the orchestrate function, the function creates a to-do list for itself (called callTuples). This to-do list contains four tasks:

1. normalize the Wikilambda code it has received (this lightly reformats the code, as explained in the “Normal Form” section of “Wikifunctions:Function Model” (2024));
2. check that the code is a Z7, a function call, otherwise raise an Error;
3. make a WrappedResultEnvelope to contain the final output of the computation; then
4. execute the actual function contained in the Wikilambda code.

|  |
| --- |
| Listing 4: The ‘callTuples’ variable contains a list of key tasks for the orchestrator to perform. Source: src/orchestrate.js, in Massaro et al. (2024b).  const callTuples = [  [ normalize, [ /\* generically= \*/true, /\* withVoid= \*/ true ], 'normalize' ],  [ Z7OrError, [], 'Z7OrError' ],  [ makeWrappedResultEnvelope, [], 'wrapAsZObject' ],  [  execute, [  invariants, /\* doValidate= \*/invariants.orchestratorConfig.doValidate,  /\* implementationSelector= \*/implementationSelector,  /\* resolveInternals= \*/true, /\* topLevel= \*/true ],  'execute'  ] ]; |

It is at step 4 that the orchestration metaphor becomes most vivid. The implementationSelector is the key parameter. This parameter tells the execute function how to choose an implementation of any functions that are invoked by the Wikilambda code. For example, function [Z13539](https://www.wikifunctions.org/wiki/Z13539) (“multiply two natural numbers”) has five implementations at the time of writing: three “composition” implementations in the Wikilambda language, one Python implementation and one JavaScript implementation. If the execute function encounters [Z13539](https://www.wikifunctions.org/wiki/Z13539), for instance when calculating a factorial, which implementation should it use? Eventually the Wikilambda team intend to apply a clever implementationSelector algorithm that selects the fastest and most reliable implementation. At the moment, the only two algorithms supported by the system are: 1. use the first implementation, or 2. use a random one. By determining the implementationSelector algorithm, the orchestrate function decides how to marshall the resources of the Wikifunctions database. Like the orchestrator who allocates some parts to the strings and others to the winds, the orchestrate function decides how to allocate parts between the Python executor, JavaScript executor, any future executors, and itself.

This metaphor vividly communicates the role of the Wikilambda language as the glue of a large and complex system. This is not an ordinary language, which programmers use to write discrete applications for particular machines. Wikilambda is not an interpreter or compiler, that boils source code down into a lean sequence of machine- or bytecode instructions. Wikilambda is the first distributed online language. It conducts an orchestra of server-racks, in accordance with the score of the contributor-composer.

### 5.2 Second metaphor: wrappers and envelopes

The most prominent metaphors in the code otherwise are metaphors *for abstraction itself*. To understand this point, it is necessary first to understand JavaScript objects, which are the main tool that the Wikilambda developers use to introduce abstractions into their code.

To create an abstraction in JavaScript, you can define a new “class” of “object.” For example, if you are running an online marketplace, you might define a Customer class to represent customers, and a Merchant class to represent merchants in your marketplace. These are abstractions, because they allow you to treat all Customers and all Merchants the same, even if they are different under the hood. For example, perhaps some Merchants in your system are humans, and others are corporations, and these different kinds of Merchant have different data associated with them, different access protocols, and so on. Inside the Merchant class, you would write code that handles these differences. This means that other parts of your program can safely ignore the differences between human and corporate Merchants. For example, the Merchant class might have a name method that tells you the name of the Merchant. If you ever need the name, for instance, to generate the title on a seller page, you can simply write merchant.name() to retrieve it.[[10]](#footnote-81) For a human merchant, the first name and surname will be correctly combined. For a corporate merchant, the preferred brand name will be returned. Thus we have successfully “abstracted away” the difference between human and corporate merchants. We can write simpler code in which there is just one kind of Merchant we have to worry about.

The most important class in the orchestrator code is the ZWrapper. The ZWrapper class has a complex task. During a function orchestration, a tree of ZWrappers grows. Each time a new function is added to the tree, a ZWrapper is created to store information about that function and its “scope.” Scope is a tricky topic, but basically it prevents name clashes in the code. Imagine that you use a function that peforms five multiplications, for example. In this case, there will be five things called multiplier and five things called multiplicand. The ZWrapper class ensures that the correct multipliers and multiplicands are paired with one another in the right place.

The name of the ZWrapper class is interesting, because Wrapper is a synonym for “abstraction.” In their comments to the code, the developers frequently use the term “wrap” in this sense, e.g. the ReferenceResolver class “wraps MediaWiki” and the Evaluator class “[w]raps API calls to the function-evaluator service” (“src/invariants.js,” “src/Evalutor.js,” Massaro et al. 2024b). They also frequently use “wrap” and its morphemes in the names of functions and variables, e.g. wrappedResult, wrapInZ6, wrapInQuote, makeWrappedResultEnvelope (“src/builtins.js,” “src/execute.js,” Massaro et al. 2024b). In all these cases, the “wrapping” operation conceals the original item, giving it a new and hopefully more useable exterior. This is the very process of abstraction, in its rawest form. What does a ZWrapper wrap? A ZObject, which is more-or-less the thing portrayed in [Listing 2](#lst-wikilambda-factorial). It is a JavaScript object extracted from the Wikifunctions database, identified by a Z-code such as [Z13667](https://www.wikifunctions.org/wiki/Z13667). Thus a ZWrapper is a kind of JavaScript object that “wraps” another kind of JavaScript object called ZObject. It may as well be called ZMetaphor or ZAbstraction or ZObjectObject.

This vocabulary dramatises the Wikilambda team’s struggle to create new abstractions. A Wrapper that wraps some Object is barely an abstraction at all. What makes this ZObject special? What does the act of “wrapping” it achieve? If I create Merchant and Customer classes in my online marketplace code, I enable myself to write code that actually describes the customers and merchants in the real world of my marketplace. What exactly do ZWrappers and ZObjects allow me to describe? The Wikilambda team are still evolving the metaphors they need. As they gain a deeper understanding of the software, the ZWrapper will likely dissolve into a cluster of better-labelled and more clearly specified software components.

Another case is presented by the term envelope. Throughout the code, the developers use the term envelope to refer to an “Evaluation Result,” that is, the output of a function that has been executed. Despite this consistent usage of the term, there is no Envelope class, although an envelope can be placed in a ZEnvelopeWrapper when it needs to be interpreted. In this case, there is already an available abstraction, but the developers decline to use it. Why not refer to results as results? Or as [Z22](https://www.wikifunctions.org/wiki/Z22)s? In this case, instead of referring to a part of the code using an available metaphor, they refer concretely to the way the result is “enveloped” in a JavaScript object. This is not a failure to abstract, but rather a decision to concretise.

Other parts of the code are more abstract. These more abstract parts of the code fall into two groups. There is the group of simple abstractions: BaseFrame, Invariants, RateLimitCache and ArgumentState all represent simple pieces of data or functionality that are relatively easy to implement. The other group of strong abstractions concern Wikimedia databases. The Implementation class represents function implementations in the Wikifunctions database. The Evaluator class represents the function evaluator service ([Figure 1](#fig-wikilambda-architecture)). The ReferenceResolver abstracts away both Wikifunctions *and* Wikidata, allowing the orchestrator to straightforwardly retrieve functions, lexemes or entities from these databases. These abstractions are not always perfect. For example, the ReferenceResolver should ideally abstract away the different kinds of reference, allowing the programmer to resolve whatever they like without thinking about which database it comes from. It does not. But there is nonetheless a familiar pattern to these metaphors. The parts of the code that deal with the creation of a new programming language are uncertain and vague. The parts that deal with databases and the Internet are confident and clear.

It is possible to observe the team actively creating the abstractions they need, by reviewing the public history of the code. The process is deeply “hacky.” For example, in August 2021, a developer introduced a new class into the code, ArgumentState, to allow “further laziness of function orchestration” (Massaro et al. 2024b, commit [7dba4d5](https://gitlab.wikimedia.org/repos/abstract-wiki/wikifunctions/function-orchestrator/-/commit/7dba4d5db545a920f7f1ad01d87aef8857e664ab)).[[11]](#footnote-85) Unlike the wrappers and envelopes we have been considering, ArgumentState is a communicative metaphor for what the code is doing, and allows the programmer to ignore exactly how this “state” is managed. The metaphor was added to the code in response to a simple bug report.[[12]](#footnote-86) For three years, the change was meagerly documented, until the Google Fellows finally explained Wikilambda’s lazy evaluation strategy in their informal report on the language’s semantics (“Abstract Wikipedia/Semantics of Wikifunctions” 2024). For Wikilambda, the rule is: hack first, explain later.

### 5.3 Wikilambda as hack

These metaphors—swamps, orchestras, wrappers, envelopes—indicate the “hackiness” of the Wikilambda code. The Wikilambda team are actively developing the abstractions they need in the very process of writing the code to implement the language.

Why are the Wikilambda developers reinventing the wheel? There are two commonsense answers. One is that the team is inexperienced, as the Google Fellows allege, and as the team suggest in their own documentation. The other is that the Wikilambda language is a brand new kind of language, and the right abstractions might not exist yet. Wikilambda is possibly the first language to actually be hosted in a database. Other languages, such as Python, JavaScript, Haskell and Scheme, are standalone languages, whose primitive elements and syntactic rules are hard-coded into the language’s interpreter or compiler. Wikilambda is on online language, whose primitive elements are stored in a database (e.g. [Z7](https://wikifunctions.org/Z7) for function calls or [Z22](https://wikifunctions.org/Z22) for evaluation results), and which has essentially no syntax.

There is a deeper meaning to the “hackiness” of the code, however. The team could have adopted the standard approach to language design: convene a committee, specify desired features, define the syntax in Backus-Naur form, etc. Instead, they first articulated a “Vision,” and then leapt into a prototype. This hacky approach reflects the team’s utopian optimism. If the Vision is sound, the technology will follow. Perhaps this approach is inevitable whenever a language is intended to be perfect or universal. Scheme was an experimental mini-language invented in an afternoon or two by a couple of young MIT hackers. Scheme went through many iterations, and it was years until it had anything like a formal specification. Like Scheme, Wikilambda may one day hack its way to greatness.

## 6 Coda

It is impossible to judge the practical significance of Wikilambda. It is still in development, and its first major application, Abstract Wikipedia, is just an idea. But the project has already demonstrated its technological and ideological significance.

### 6.1 Technology: Platform Level 4

If Wikilambda succeeds, then Wikifunctions could mark a new stage in the technological history of internet platforms. In an influential blog post, Marc Andreessen (2007) distinguishes three “levels” of internet platform, depending on the platform’s degree of programmability:

* Level 1: the platform allows external applications to access its data
* Level 2: the platform allows external applications to be embedded in its user interface
* Level 3: the platform allows internal applications, that is, it allows developers to write and run code in the platform itself

Wikipedia is already a Level 3 platform. Since Wikilambda merely extends Wikipedia, it meets the same criteria. But it provides another capability. It allows outside developers to extend its own API. Wikifunctions is a Platform Level 4.

To see this, we can compare Wikifunctions with another Platform Level 3, Roblox. Roblox is an online platform on which creators write computer games using the Lua programming language. It provides [an extremely rich API for creators](https://create.roblox.com/docs/reference/engine), allowing even beginning programmers to create realistic 3D characters, branching dialogue, atmospheric cloud effects, and so on. What if you find that you want a feature in your game, but the Roblox API does not support it? Maybe, for instance, you are creating a surreal game that needs 4D objects. Lua is a powerful programming language, so you could perhaps implement 4D objects yourself, or copy the code from another creator. But you could not actually add support for 4D objects to the Roblox API, because it is controlled by the platform.

Wikifunctions is different. As soon as anyone adds a function to the database, it is immediately accessible by all other users of the platform, whether they are writing an external or internal application. The owners of Wikifunctions do not limit or control the API. If you are frustrated by Wikifunctions’s lack of support for 4D graphics, then you can simply add the needed functions to the database. Your only disappointment would be Wikilambda’s abysmal slowness, which could tarnish the surreal brilliance of your game.

### 6.2 Ideology: Moral Codes

Wikifunctions aims to enable “the next **big leap in knowledge representation**” (“Wikifunctions:Vision” 2023). In the age of large language models (LLMs), this is a bold and countercultural aim. Wikifunctions presents a stark alternative to LLMs such as Gemini, Llama or chatGPT. These LLMs rely on vast concealed datasets. Wikifunctions draws its data from public Wikimedia databases. These LLMs generate text using opaque algorithms that even their designers cannot understand.[[13]](#footnote-95) Wikifunctions makes every part of every algorithm available to anyone. In short, Wikifunctions is *contestable*, as will be Abstract Wikipedia. If you ask chatGPT to generate an article on a topic, the only way to contest its algorithm is to click 👍 or 👎 (Crawford and Gillespie 2016). If you are unhappy with an article generated by Abstract Wikipedia, you will be able to: change the “abstract” content in Wikidata; change the algorithms that generate the article in Wikifunctions; or sever the connection between the article and Abstract Wikipedia, and edit it the old-fashioned way in Wikipedia. The role of Wikilambda in all this is to make algorithms “defeasible” (Blanton et al. 2022). Every part of every algorithm is there, and can be contested on the platform itself, even if that contestation may be culturally and politically constrained (Tkacz 2015; Ford 2022).

The universality and perfection of Wikilambda are essential to this project. If the language is not universal, then not everyone will be able to contest it. If the language is not perfect, then it will limit the space of possible algorithms, and thus limit the creativity of the community. Wikilambda is an attempt to design what Alan Blackwell (2024) calls a “moral code”: it combines More Open Representation, Access to Learning, and Creating Opportunities for Digital Expression. If nothing else, Wikilambda is a thundering critique of corporate AI hype.

Even before the rise of LLMs, various Wikipedias were struggling to cope with *incontestable* machine-generated text. The Swedish, Cebuano, Waray and Egyptian Arabic Wikipedias have been largely generated by “bots,” which extract data from public sources to create masses of formulaic text (Ford et al. 2024). Although the generated *articles* can be contested on Wikipedia, the generation *algorithms* cannot. Swedish editors have deleted most of lsjbot’s articles, but they were not able to alter its code or data sources. Wikilambda presents an alternative to these prior efforts at machine-generation of Wikipedia articles.

In some ways, Wikilamba’s utopian vision is quite consistent with the familiar cyberlibertarianism of Silicon Valley. Wikilambda is a utopia for the disembodied (Turner 2006, 261). In the Wikilambda world, there are only symbols. Some symbols are data. Others are algorithms. The world’s 6000 human languages are convenient user interfaces, allowing rapid symbol-exchange between disparate human minds. Wikilambda treats human language as an engineering challenge, hoping to overcome its richness and informality, its role as a “carrier of culture” (Thiong’o 1986), its stickiness, specificity and resistance to translation (Steiner 1998), its embeddedness in “life” (Bakhtin 1986, 63ff.). But on these rocky shoals of human speech have foundered all prior attempts at polygraphy. Words “bite and scratch,” and “You never learn / the chemical process of separating them” (Frame 2008, 117). Will a ZWrapper one day achieve this “chemical process”? The revolutionary cadres of the Abstract Wikipedia project are hacking daily to find out.

## 7 References

Abbate, Janet. 1999. *Inventing the Internet*. Cambridge (Mass.): MIT press.

“Abstract Wikipedia/Architecture.” 2024. *Wikimedia Meta-Wiki*. <https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Architecture&oldid=26987678>.

“Abstract Wikipedia/Components.” 2024. *Wikipedia Meta-Wiki*. <https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Components&oldid=26987679>.

“Abstract Wikipedia/Semantics of Wikifunctions.” 2024. *Wikimedia Meta-Wiki*. <https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Semantics_of_Wikifunctions&oldid=26035186>.

“Abstract Wikipedia/Template Language for Wikifunctions.” 2024. *Wikimedia Meta-Wiki*. <https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Template_Language_for_Wikifunctions&oldid=26290340>.

Andreessen, Marc. 2007. “The Three Kinds of Platforms You Meet on the Internet.” *Pmarchive*. <https://web.archive.org/web/20240329034257/https://pmarchive.com/three_kinds_of_platforms_you_meet_on_the_internet.html>.

“Annual Report 2022-23.” 2023. San Francisco: Wikimedia Endowment. <https://wikimediaendowment.org/wp-content/uploads/2024/04/Endowment-Annual-Report-22-23.pdf>.

Bakhtin, Mikhail. 1986. *Speech Genres and Other Late Essays*. Edited by Caryl Emerson and Michael Holquist. Translated by Vern McGee. Austin: University of Texas Press.

Barbrook, Richard, and Andy Cameron. 1996. “The Californian Ideology.” *Science as Culture* 6 (1): 44–72. <https://doi.org/10.1080/09505439609526455>.

Bayer, Tilman. 2023. “Wikimedia Foundation’s Abstract Wikipedia Project ’at Substantial Risk of Failure’.” *The Signpost*, January. <https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_Signpost/2023-01-01/Technology_report>.

Blackwell, Alan F. 2024. *Moral Codes: Designing Alternatives to AI*. The MIT Press. <https://doi.org/10.7551/mitpress/14872.001.0001>.

Blanton, Cai, Cory Massaro, David Martin, Denny Vrandečić, Genoveva Galarza Heredero, James Forrester, Julia Kieserman, and Stef Dunlap. 2022. “Abstract Wikipedia/Google.org Fellows Evaluation - Answer.” *Wikimedia Meta-Wiki*. <https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Google.org_Fellows_evaluation_-_Answer&oldid=24262326>.

Chalmers, David J. 1996. “Does a Rock Implement Every Finite-State Automaton?” *Synthese* 108 (3): 309–33. <https://doi.org/10.1007/BF00413692>.

Church, Alonzo. 1932. “A Set of Postulates for the Foundation of Logic.” *Annals of Mathematics* 33 (2): 346–66. <https://doi.org/10.2307/1968337>.

Crawford, Kate, and Tarleton Gillespie. 2016. “What Is a Flag for? Social Media Reporting Tools and the Vocabulary of Complaint.” *New Media & Society* 18 (3): 410–28. <https://doi.org/10.1177/1461444814543163>.

Dillon, Sarah, and Jennifer Schaffer-Goddard. 2023. “What AI Researchers Read: The Role of Literature in Artificial Intelligence Research.” *Interdisciplinary Science Reviews* 48 (1): 15–42. <https://doi.org/10.1080/03080188.2022.2079214>.

Eco, Umberto. 1995. *The Search for the Perfect Language*. Making of Europe. Oxford, UK ; Cambridge, Mass., USA: Blackwell.

“Extension:WikiLambda.” 2024. *MediaWiki*. <https://www.mediawiki.org/w/index.php?title=Extension:WikiLambda&oldid=6467929>.

“Factorial, Composition (Z13863).” 2024. *Wikifunctions*. <https://www.wikifunctions.org/wiki/Z13863?uselang=en&oldid=101571>.

Ford, Heather. 2022. *Writing the Revolution: Wikipedia and the Survival of Facts in the Digital Age*. Cambridge, Massachusetts: The MIT Press.

Ford, Heather, and Andrew Iliadis. 2023. “Wikidata as Semantic Infrastructure: Knowledge Representation, Data Labor, and Truth in a More-Than-Technical Project.” *Social Media + Society* 9 (3): 20563051231195552. <https://doi.org/10.1177/20563051231195552>.

Ford, Heather, Francesca Sidoti, Michael Falk, Tamson Pietsch, and Tom Byers. 2024. “How Australian Places Are Represented on Wikipedia.” 2. University of Technology, Sydney. <https://doi.org/10.5281/zenodo.13910503>.

Foundation, Wikimedia. 2023. “First Grants Announced from the Wikimedia Endowment to Support Technical Innovation Across Wikipedia and Wikimedia Projects.” *Wikimedia Foundation*. <https://wikimediafoundation.org/news/2023/04/13/first-grants-announced-from-the-wikimedia-endowment/>.

Frame, Janet. 2008. *Storms Will Tell: Selected Poems*. Tarset, Northumberland: Bloodaxe Books.

Friedman, Daniel P., Matthias Felleisen, and Duane Bibby. 2007. *The Little Schemer*. 4. ed. Cambridge, Mass.: MIT Press.

Hanson, Chris, Guillermo J. Rozas, Taylor R. Campbell, Stephen Adams, Matt Birkholz, Arthur Gleckler, Joe Marshall, et al. 1979. “MIT/GNU Scheme.” Cambridge, Mass: Massachusetts Institute of Technology. <https://git.savannah.gnu.org/cgit/mit-scheme.git/>.

Helmond, Anne. 2015. “The Platformization of the Web: Making Web Data Platform Ready.” *Social Media + Society* 1 (2): 2056305115603080. <https://doi.org/10.1177/2056305115603080>.

Hickey, Rich. 2020. “A History of Clojure.” *Proceedings of the ACM on Programming Languages* 4 (HOPL): 1–46. <https://doi.org/10.1145/3386321>.

Hudak, Paul. 1989. “Conception, Evolution, and Application of Functional Programming Languages.” *ACM Computing Surveys* 21 (3): 359–411. <https://doi.org/10.1145/72551.72554>.

———. 2000. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge, U.K. ; New York: Cambridge University Press.

Livneh, Ori, Ariel Gitman, Ari Assaf, and Mary Yang. 2022. “Abstract Wikipedia/Google.org Fellows Evaluation.” *Wikipedia Meta-Wiki*. <https://meta.wikimedia.org/w/index.php?title=Abstract_Wikipedia/Google.org_Fellows_evaluation&oldid=24268563>.

Marino, Mark C. 2020. *Critical Code Studies*. The MIT Press. <https://doi.org/10.7551/mitpress/12122.001.0001>.

Massaro, Cory, Denny Vrandečić, Daphne Smit, David Martin, Ecarg, Genoveva Galarza, Jeena Huneidi, James Forrester, Stef Dunlap, and Vaughn Walters. 2024a. “Function Evaluator.” <https://gitlab.wikimedia.org/repos/abstract-wiki/wikifunctions/function-evaluator>.

———. 2024b. “Function Orchestrator.” <https://gitlab.wikimedia.org/repos/abstract-wiki/wikifunctions/function-orchestrator>.

Mateas, Michael, and Nick Montfort. 2005. “A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics.” In, 144–53. IT University of Copenhagen.

Montfort, Nick, ed. 2013. *10 PRINT CHR$(205.5+RND(1));:GOTO 10*. Software Studies. Cambridge, Mass: MIT Press.

Nand Sharma, Pankajeshwara, Bastin Tony Roy Savarimuthu, and Nigel Stanger. 2022. “Unearthing Open Source Decision-Making Processes: A Case Study of Python Enhancement Proposals.” *Software: Practice and Experience* 52 (10): 2312–46. <https://doi.org/10.1002/spe.3128>.

Orlikowski, Wanda J., and JoAnne Yates. 2002. “It’s about Time: Temporal Structuring in Organizations.” *Organization Science* 13 (6): 684–700. <https://www.proquest.com/docview/213829873/abstract/57070C934917444CPQ/1>.

“Special Forms.” 1997. *An Introduction to Scheme and Its Implementation*. <https://docs.scheme.org/schintro/schintro_17.html>.

Steele, Guy L., and Richard P. Gabriel. 1996. “The Evolution of Lisp.” In *History of Programming Languages—II*, edited by Thomas J. Bergin and Richard G. Gibson, 233–330. New York, NY, USA: ACM. <https://doi.org/10.1145/234286.1057818>.

Steele Jr., Guy Lewis. 1976. “LAMBDA: The Ultimate Declarative.” 379. Cambridge, Mass: Massachusetts Institute of Technology. <https://dspace.mit.edu/handle/1721.1/6091>.

———. 1977. “Debunking the "Expensive Procedure Call" Myth; or, Procedure Call Implementations Considered Harmful; or, Lambda: The Ultimate Goto.” 443. Cambridge, Mass: Massachusetts Institute of Technology. <https://apps.dtic.mil/sti/pdfs/ADA062381.pdf>.

Steele Jr., Guy Lewis, and Gerald Jay Sussman. 1976. “LAMBDA: The Ultimate Imperative.” 353. Cambridge, Mass: Massachusetts Institute of Technology.

———. 1978. “The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two).” 453. Cambridge, Mass: Massachusetts Institute of Technology. <https://dspace.mit.edu/bitstream/handle/1721.1/6094/AIM-453.pdf>.

Steiner, George. 1998. *After Babel: Aspects of Language and Translation*. 3. ed. Oxford: Oxford University Press.

Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1975. “Scheme: An Interpreter for Extended Lambda Calculus.” 349. Cambridge, Mass: Massachusetts Institute of Technology. <https://dspace.mit.edu/bitstream/handle/1721.1/5794/AIM-349.pdf>.

Sussman, Gerald Jay, Julie Sussman, and Harold Abelson. 1996. *Structure and Interpretation of Computer Programs*. Second. Cambridge: MIT Press.

Thiong’o, Ngugi wa. 1986. *Decolonising the Mind: The Politics of Language in African Literature*. Woodbridge: Boydell & Brewer, Limited.

Tkacz, Nathaniel. 2015. *Wikipedia and the Politics of Openness*. Chicago ; London: University of Chicago Press.

Turing, Alan. 1936. “On Computable Numbers, with an Application to the Entscheidungsproblem.” *Proceedings of the London Mathematical Society* 42: 230–65.

———. 1937. “Computability and λ-Definability.” *The Journal of Symbolic Logic* 2 (4): 153–63. <https://doi.org/10.2307/2268280>.

Turner, Fred. 2006. *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism*. Chicago; London: Chicago University Press.

Vee, Annette. 2017. *Coding Literacy: How Computer Programming Is Changing Writing*. The MIT Press. <https://doi.org/10.7551/mitpress/10655.001.0001>.

Vrandečić, Denny. 2020. “Collaborating on the Sum of All Knowledge Across Languages,” October. <https://doi.org/10.7551/mitpress/12366.003.0016>.

———. 2021. “Building a Multilingual Wikipedia.” *Communications of the ACM* 64 (4): 38–41. <https://doi.org/10.1145/3425778>.

Wark, McKenzie. 2004. *A Hacker Manifesto*. Cambridge, Mass: Harvard University Press. <https://doi.org/10.4159/9780674044845>.

“Wikifunctions:About.” 2024. *Wikifunctions*. <https://www.wikifunctions.org/w/index.php?title=Wikifunctions:About&oldid=119089>.

“Wikifunctions:Determinism.” 2024. *Wikifunctions*. <https://www.wikifunctions.org/w/index.php?title=Wikifunctions:Determinism&oldid=118687>.

“Wikifunctions:Function Model.” 2024. *Wikifunctions*. <https://www.wikifunctions.org/w/index.php?title=Wikifunctions:Function_model&oldid=107530>.

“Wikifunctions:Vision.” 2023. *Wikifunctions*. <https://www.wikifunctions.org/w/index.php?title=Wikifunctions:Vision&oldid=50281>.

“Wikifunctions:What Wikifunctions Is Not.” 2024. *Wikifunctions*. <https://www.wikifunctions.org/w/index.php?title=Wikifunctions:What_Wikifunctions_is_not&oldid=71136>.

1. In what follows, I use “Wikifunctions” and “Wikilambda” interchangeably, as my sources dictate. There is no risk of confusion, because Wikifunctions is currently the only database running the Wikilambda software. [↑](#footnote-ref-20)
2. A proof-of concept of the “template language” does exist, but it was not implemented using Wikilamba (“Abstract Wikipedia/Template Language for Wikifunctions” 2024). [↑](#footnote-ref-34)
3. It seems that the plan is eventually to allow contributors to “compose” Wikifunctions functions using ordinary programming languages such as Python, but this feature has not yet been implemented: “Implementations can be written in a number of different programming languages, *but they will be able to call functions implemented in other programming languages*.” (“Wikifunctions:About” 2024, emphasis added) [↑](#footnote-ref-37)
4. In the proof-of-concept for Abstract Wikipedia’s template language, the process of rendering an article goes through five distinct phases, in each of which the entire abstract article is scanned, and various transformations applied to it (“Abstract Wikipedia/Template Language for Wikifunctions” 2024). Each phase may require hundreds of distinct functions to be composed together. [↑](#footnote-ref-38)
5. See [Section 4.2](#sec-wikilambda-and-scheme-compared) for a worked example. [↑](#footnote-ref-41)
6. For a brilliant and readable introduction to recursive functions, I would recommend Friedman, Felleisen, and Bibby (2007). [↑](#footnote-ref-57)
7. Also known as “Cambridge Polish” syntax: Steele Jr. and Sussman (1978, 2). [↑](#footnote-ref-65)
8. To calculate these figures, I navigated to the src directory of MIT/GNU Scheme (Hanson et al. 1979) and the function orchestrator (Massaro et al. 2024b), and ran git ls-files | xargs cat | wc -l from the command line. [↑](#footnote-ref-74)
9. The source code for CPython and the V8 JavaScript Engine can be searched easily on Github. I searched the MIT/GNU Scheme code on my own machine. [↑](#footnote-ref-78)
10. By convention, in JavaScript you use PascalCase to name *classes* or *types* of objects, and camelCase to name particular *instances* of or *individual* objects. There may be a thousand individual merchants in your system, but there is only one kind of Merchant. [↑](#footnote-ref-81)
11. In fact, prior to this commit, the function orchestrator was not actually lazy, because it allowed “arguments” to be evaluated more than once (see Hudak 1989). [↑](#footnote-ref-85)
12. Bug [T288635](https://phabricator.wikimedia.org/T288635) on the Wikimedia foundation’s central bug-tracking service. [↑](#footnote-ref-86)
13. Of course, they understand the *learning* algorithms which they use to train the systems. These learning algorithms are extremely simple. What they cannot understand are the *learned* algorithms—the algorithms LLMs “learn” from the training data. [↑](#footnote-ref-95)