Wikilambda the Ultimate

The Wikimedia foundation's search for the perfect language

Michael Falk

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, p. 19)

1 Introduction

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)

Such a "risk of failure" is hardly surprising, because Abstract Wikipedia is a project of quite astonishing ambition. Abstract Wikipedia aims to transform Wikipedia from a *multi*lingual encyclopaedia into a *meta*lingual one. It will provide a way for contributors to write articles in a universal "template language," which will then be automatically translated into all the world's human languages. Abstract Wikipedia thus combines two equally extreme ambitions: (1) the ambition to devise a single language in which all the world's knowledge can be easily expressed; and (2) the ambition to provide 300-6,000 "renderers" that can translate this "abstract content" into readable text. As the leader of the Abstract Wikipedia project admits, the project's "ambitious goal" is not new, and "such ambitions have repeatedly failed" (Vrandečić 2020, p. 180, 2021, p. 41). Abstract Wikipedia is the latest in a long line of attempts to create a "perfect" or "universal" language—to be more precise, it is the latest in a long line of "polygraphies," or universal writing systems (Eco 1995, chap. 9). Since every prior attempt at polygraphy has failed, it is hardly surprising that Abstract Wikipedia risks failing also!

Why, then, was it *news* in 2023 that Abstract Wikipedia was "at substantial risk of failure"? The project was an audacious gamble from the start. What had changed?

Three years in to the Abstract Wikipedia project, it had become apparent that the project aimed to create not *one* perfect language, but *two*. Work on the template language for writing articles had barely begun. Instead, the project had devoted nearly all its effort to the creation of a brand new programming language in which the 300-6,000 "renderers" for Abstract Wikipedia would be written. This programming language would make Abstract Wikipedia possible, by allowing universal access to programming:

> 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. (2023a)

The new language that underlies this universally accessible programming system remains nameless to this day. I will refer to the language as "Wikilambda", because that is the name of the software that implements it, but it has other names, including "composition," "the Function Model," "ZObjects" and "Wikifunctions" (the name of the database where the language is hosted).

As *The Signpost* reported, all was not well with Wikilambda. Ten days prior to the article, a group of Google employees seconded to Abstract Wikipedia had released a damning report on Meta-Wiki, the site where projects of the Wikimedia Foundation are documented:

> 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)

This is a significant crisis for the Wikimedia Foundation. Abstract Wikipedia was the first new project sponsored by the Foundation in 10 years. In 2022-23, the project was awarded US$1 million from the Wikimedia Endowment (2023b; Foundation 2023). A prototype of the project's "ad-hoc" and "questionable" programming system is already available at wikifunctions.org. How did the project get into this situation, and what does its future hold?

In this article, I examine the documentation and source code of Wikilambda to critique both its ideals and its implementation. There is in fact an intimate link between the project's ideals and its blithe ignorance of "decades of programming language research." The project expresses, in a typically extreme way, a certain kind of hacker utopianism. It is relentlessly future-oriented. The present arrangement of things is an historical accident. Through a simple hack, the project will bring into being a *new* arrangement of things, in which information is freer, the world is smaller, the mind is larger. McKenzie Wark (2004, para. 098) poetically summarises this attitude to history: "The past weighs like insomnia upon the consciousness of the present." The past can be brushed aside, because it has already been achieved. What matters is what comes next, and what comes next can *only* come about through the ingenuity and creativity of the hacker. The only *faits* worth caring about are *faits accomplis*.

To make this case, I analyse the project on three levels. In Section 2, I describe the overall architecture of the Abstract Wikipedia project, and explain how the new programming language Wikilambda fits into the structure. In Section 3, I examine Wikilambda's documentation, critiquing the arguments used by its developers to establish the perfection and universality of this new programming language. Finally in Section 4, I closely read the source code for the "function orchestrator," the JavaScript application whose job is to evaluate expressions in the Wikilambda language. Not only does the code *implement* the project's ideals, it *expresses* them. In the code itself, we can perceive the struggle of Vrandečić and his team to realise their ideals in practice.

My aim in this research is partly hermeneutic and partly methodological. On the hermeneutic level, I wish to understand what the project means to its creators, and to set their hacker utopianism in a broader context. On the methodological level, I wish to fashion new tools for the burgeoning community of Critical Code Studies (CCS). Thus in Section 4 I present a theory of *abstraction* and *metaphor* in source code. Programmers create software by developing abstractions to describe the behaviour of their programs. These abstractions begin life as metaphors. The classic theory of abstraction in programming holds that abstraction is a form of "information hiding" (Colburn and Shute 2007). This theory has had noticeable impact on early research in CCS (Hua and Raley 2023, para. 14; Rountree and Condee 2023, paras. 8–10). I argue that the "information hiding" theory is one-sided and ignores the metaphoricity of abstractions. I propose a theory, *abstraction as the virtualisation of metaphor*, and use this theory to elucidate the "extrafunctional significance" of the Wikilambda code (Marino 2020).

2 Architecture: Abstract Wikipedia/Wikifunctions

> The conclusion is that the Wikimedia movement does not believe that language is the right dimension to split knowledge—it is a historical decision, driven by convenience. (Vrandečić 2020, p. 182)

The Wikimedia Foundation already provides Wikipedias in more than 300 languages. Each language edition of Wikipedia has its own editorial culture, though it is also common for editors to contribute to multiple language editions (Ford 2022; Avieson 2022). The language editions vary greatly in number and quality of articles. It has been common for editors of smaller language editions, such as Cebuano, Swedish, Waray, Egyptian Arabic and Romanian, to rely on machine translation or bots to generate articles (Gnatiuk and Glybovets 2021; Ford et al. 2024).

Denny Vrandečić, the leader of the Abstract Wikipedia project, objects to inconsistencies between the language editions. His previous project for the Wikimedia Foundation, Wikidata, automates the synchronisation of "language links" between Wikipedia articles, and provides a source of structured data about the entities those Wikipedia articles describe. For example, Joseph Furphy's novel *Such is Life* has the Wikidata ID Q7632777. At the time of writing, Wikidata records that two Wikipedias contain articles about *Such is Life*, namely the English and Serbo-Croatian Wikipedias. The Wikidata item for the novel contains structured data, such as the author and publication date, which could be used to generate an Infobox or basic article about the novel in other Wikipedias. If only more Wikipedia articles would draw on Wikidata in this way, laments Vrandečić (2020, p. 178), then the more "comprehensive, current, and accessible" the many Wikipedias would become.

Abstract Wikipedia extends Vrandečić's ideal of "comprehensive, current, and accessible" knowledge. In his view, knowledge is a collection of atomistic facts. Knowledge is "comprehensive" if it includes all the facts, "current" if none of them are out of date, and "accessible" if you can read a statement of each fact in a language you understand. Vrandečić illustrates these ideas using the example of San Francisco. Ideally, it should not matter what languages you read, you should be able to access San Francisco's current ranking by population among Californian cities (2024a), the name of its current Mayor (Vrandečić 2020, pp. 176–77), and the fact that it is currently the cultural, commercial and financial centre of Northern California (Vrandečić 2018, 2021). Since these facts are not currently available to readers of all languages on Wikipedia, argues Vrandečić (2020, p. 182), the Wikipedia project has thus far failed to allow "access to the sum of all knowledge to every single reader, no matter what their language."

The central aim of Abstract Wikipedia is to complete the "sum" of human knowledge. Abstract Wikipedia will not be a new Wikipedia language edition, but rather an extension to the Wikidata database, complemented by a brand-new Wikimedia project, Wikifunctions. Wikidata will be extended, so that contributors can write "abstract content" about Wikidata items in a new template language. The Wikifunctions database will store the "constructors" that define this template language, and the "renderers" that will translate the abstract content into readable content in natural languages. Editors of individual language editions will have control over how abstract content is imported into their encyclopaedia: they can choose to sync their language edition with Wikidata, automatically including all abstract pages, or they can import them piecemeal, or import the abstract content as the starting point for a manually-written article (2024a, b).

The intended architecture of the system is shown in Figure 1. The part that implements a new programming language is the "function orchestrator," whose source code I analyse in Section 4. The orchestrator not only implements a new programming language, but serves as the main entry point for the Wikilamdba software. Imagine that an editor wishes to import an "abstract" article into Croatian Wikipedia. In this case, a "parser function request" will be sent from the "MediaWiki App Server" for Croatian Wikipedia. This "parser function request" will be received by the "function orchestrator," which will "orchestrate" the generation of the new Croatian article. First it will send a "data fetch" to Wikidata to get the abstract content for the article; then it will pass this data to the "function evaluators" that do all the work of transforming the abstract content into readable Croatian text. We will see why there are multiple "function evaluators" in Section 3: the programming language implemented by the "function orchestrator" is a meta-language that allows code in many programming languages to be combined. Each supported programming language will require a separate "evaluator."

Abstract Wikipedia is thus a machine for broadcasting facts to Wikipedia editions. Vrandečić takes the metaphor of the "sum" of knowledge very seriously. As far as Wikipedia is concerned, there is one, universal set of facts, and language is merely a device for encoding these facts in sentences.

Vrandečić (2020) expounds this linguistic theory of knowledge in a chapter justifying the project. He argues that "Language Does Not Align With Culture," and denies that linguistic diversity is a structuring principle of the Wikipedia project (2020, p. 181). To justify this argument, he provides a somewhat self-defeating example. He observes that there are three Wikipedia editions for readers of Serbian and Croatian: Serbian Wikipedia, Croatian Wikipedia, and Serbo-Croatian Wikipedia. First he complains that the distinction between these three languages is arbitrary: "Linguistically, the differences among the dialects of Croatian are often larger than the differences between standard Croatian and standard Serbian" (2020, p. 183). Then in the next sentence, he complains that Croatian Wikipedia has fascist leanings. There is a subtle contradiction between these arguments, which defeats Vrandečić's point. He uses the term "language" in two senses. When he argues that there is very little "linguistic" difference between standard Serbian and standard Croatian, he is referring to what linguists call language$_1$, or language defined in terms of its vocabulary and syntax. When he complains that Croatian Wikipedia is fascist, he is referring to what linguists call language$_2$, or language as a social institution, recognised and cherished by its speakers. It is not surprising that editors with fascist leanings would prefer to edit Croatian Wikipedia: for them, Croatian would be a language$_2$ that symbolises the Croatian nation. More liberal or cosmopolitan editors would presumably prefer to edit the Serbo-Croatian Wikipedia: for them, Serbo-Croatian would be a language$_2$ that symbolises a transnational, pan-Slavic community. Such linguistic politics

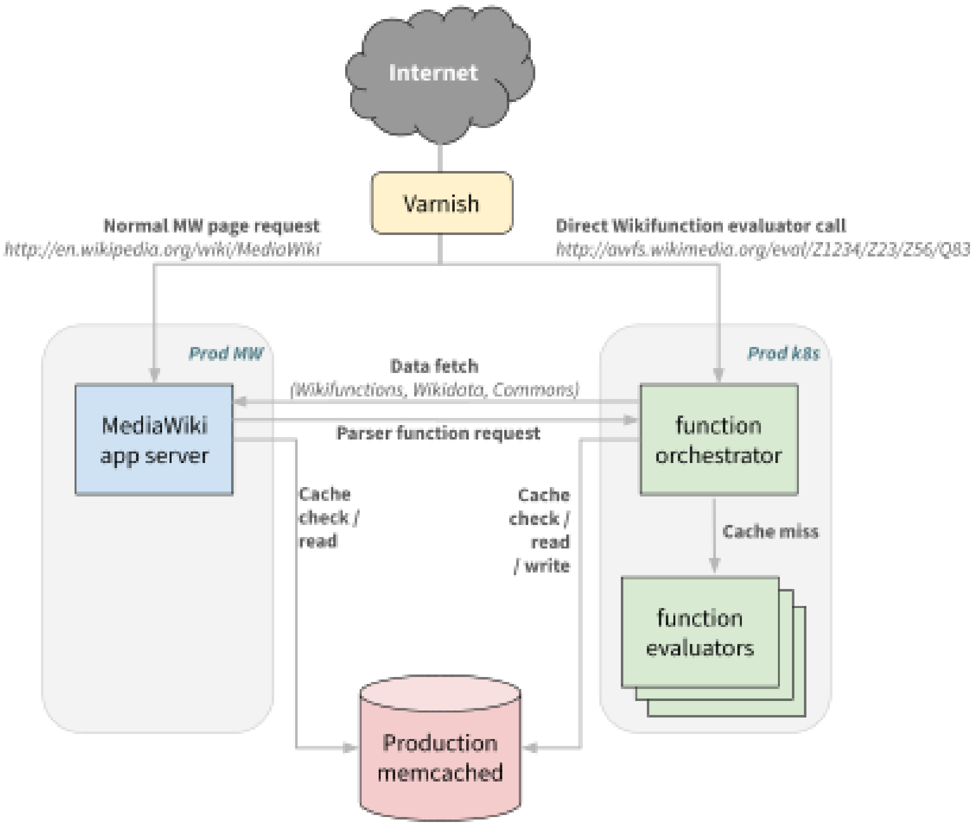


Figure 1: High-level model of the Wikilambda Extension. Source: (2024c). By jdforrester (Wikimedia Foundation) CC BY-SA 4.0.

are common. In India and Pakistan, for example, liberal intellectuals have often promoted the "Hindustani" language as a non-sectarian alternative to Urdu and Hindi (Hakala 2016; Dubrow 2018). Although Hindi and Urdu are "linguistically" very similar, as Vrandečić might say, they are socially, politically, and religiously distinct—sometimes bloodily so. Vrandečić needs to prove that "Language Does Not Align With Culture," so he can establish that it does not matter what language is used to express a fact. But in trying to prove his point, he presents a powerful example that language$_2$ *does* strongly align with culture.

Perhaps Vrandečić could repair his argument, and remove the contradiction. Yes, it is true that people place a *subjective value* on their language$_2$, but Wikipedia is an encyclopaedia, its aim is to communicate knowledge, and in this context all that matters is that *objective facts* can be expressed in some intelligible language$_1$. If you want to state the fact that "San Francisco is the cultural centre of Northern California," it does not matter whether you state this in English, Serbo-Croatian, Hindustani or Jaminjung.

Except that it does. There are at least three concepts in the statement that are metaphorical: "cultural," "centre" and "Northern." In Jaminjung there is no concept of "North," for instance. Speakers of Jaminjung do not orient themselves according to compass points, but rather according to the flow of the Victoria River (Hoffmann 2019). Is Northern California *buya* (downstream) or *manamba* (upstream)? The word "Northern" relies on the metaphor: THE WORLD IS A MAP. The word "centre" also relies on THE WORLD IS A MAP—otherwise why should an important place be located in the middle? Would a Jaminjung speaker who sees THE WORLD AS A WATERCOURSE also talk about "cultural *centres*"? The word "culture" itself relies on a different metaphor: HUMAN MANNERS ARE A FARM. Thus the apparently bland statement, "San Francisco is the cultural centre of Northern California," relies on at least two metaphors that are baked into the English language. This is not merely a matter of language$_2$, of a speaker's self-conscious identification with a speech-community; it is also a matter of language$_1$, of persistent structures of vocabulary and syntax that make statements intelligible.

Persistent metaphorical structures such as THE WORLD IS A MAP and HUMAN MANNERS ARE A FARM frame experience in a certain way. "San Francisco is the cultural centre of Northern California" is not a neutral fact. It is only relevant or meaningful if you already imagine the world as a flat surface divided into bounded, contiguous, commensurable zones (i.e. as a MAP), and believe that certain spots in those regions yield a greater crop of human achievement (i.e. they are more productive FARMS). If you speak a European language, you may well find "cultural centre" a meaningful designation. If you speak another language, it could well be meaningless.

I am making a familiar argument, that languages are "carriers of culture" (Thiong'o 1986). They carry culture mainly in the form of metaphors, which "partially" but "coherently" structure our experience (Lakoff and Johnson 1980). Vrandečić himself, and the Abstract Wikipedia project as a whole, embody one of the most powerful metaphors in the English language, the well-known "conduit metaphor" (Reddy 1993). According to this metaphor, LANGUAGE IS A CONDUIT. When we speak or write, we pack "content" into a sentence, which is then delivered to a speaker or reader who unpacks the content at the other end. This metaphor is virtualised[1] in the architecture of the Abstract Wikipedia/Wikifunctions project:

> The main components of the project are the following three:
>
> 1. **Constructors** – definitions of Constructors and their slots, including what they mean and restrictions on the types for the slots and the return type of the Constructor [...]
> 2. **Content** – abstract calls to Constructors including fillers for the slots [...]
> 3. **Renderers** – functions that take Content and a language and return a text, resulting in the natural language representing the meaning of the Content [...] (2024a)

A "constructor" is a container, with "slots" that get "filled" by "content". This "content" is then delivered to a "renderer," which extracts the "meaning." This "meaning," of course, is unchanged by the processes of "construction" and "rendering," because LANGUAGE IS A CONDUIT, and merely transports pre-existing meanings. Within the overall project, Wikilambda's purpose is to provide the constructors that package the content, and the renderers that unpack it at the other end of the conduit. In their attempt to escape language and universalise knowledge, Vrandečić and his team have plunged into English, the very language they have been most concerned to escape.

The Abstract Wikipedia/Wikifunctions project attempts not one, but two escapes from language. Ultimately, the Wikilambda developers want to escape from spoken languages, into the "abstract content" of Abstract Wikipedia's template language. But before attempting this, they have attempted another escape, an escape from programming languages into the "abstract syntax tree" of the Wikilambda language—a language which they deny is a language at all.

3 Documentation: Wikilambda the Perfect

> **Wikifunctions is not a programming language, nor is trying to evangelise a particular language** (2024d)

Umberto Eco (1995, p. 73) distinguishes two kinds of ideal language: the "perfect" and the "universal." A "perfect" language is one that is "capable of mirroring the true nature of objects." Such a language must analyse the world into its constituent parts, and provide means to build it back up again. Each word must correspond to a real component of nature, and each syntactic rule must correspond to a way that nature combines primitive elements into complex entities. Jonathan Swift (2005) parodies the ideal of a "perfect" language in *Gulliver's Travels*. On the floating island of Laputa, the enlightened inhabitants have attempted to do away with the imperfections of words, and communicate directly with *things*. As a result, they need to cart

[1]See Section 4.

around enormous loads of objects to provide their vocabulary. A “universal” language is ideal in a different way: it is a language “which everyone might, or ought to, speak.” Esperanto is an example among the spoken languages. Among programming languages, BASIC, Logo, Python and Scratch are examples of languages that are intended to be universally accessible (Vee 2017, pp. 43–46).

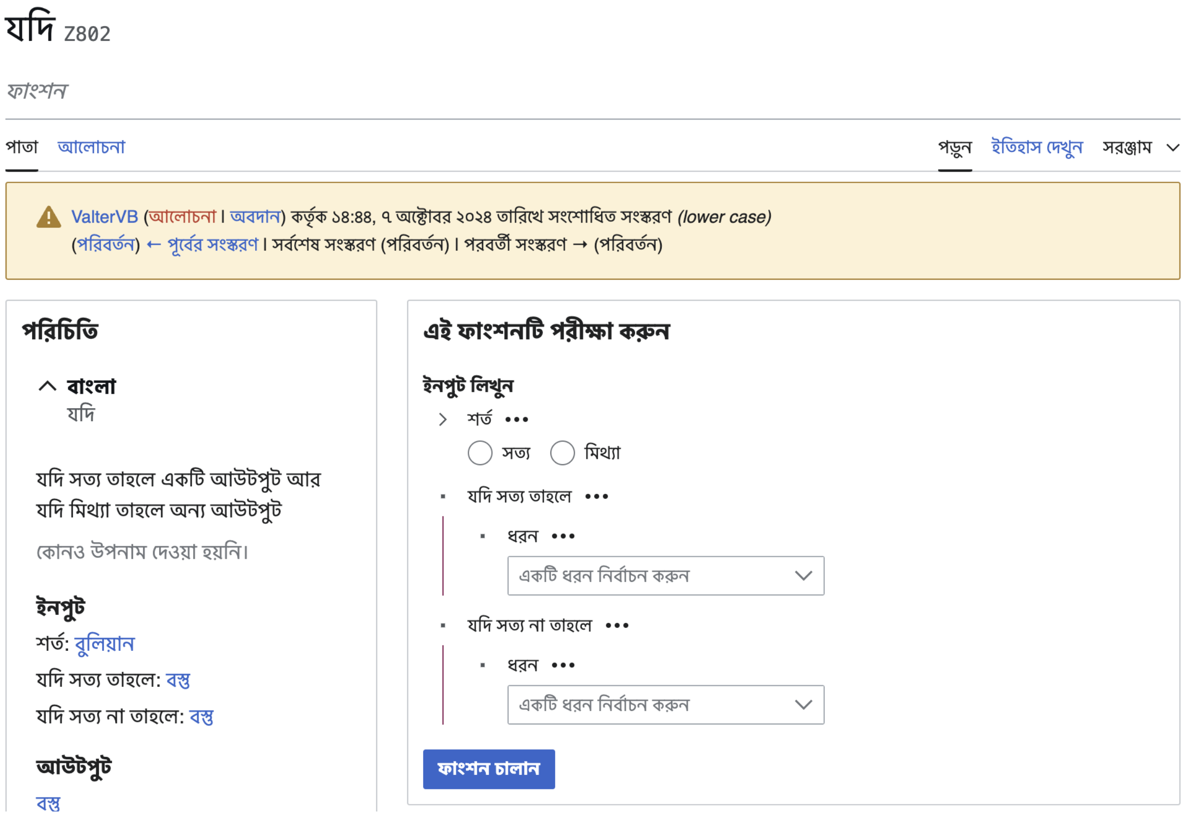
The proposed “template language” for Abstract Wikipedia is intended to be both perfect and universal: it will be perfectly able to express any fact, and universally accessible by writers all over the world. To implement this “template language,” the Abstract Wikipedia team have gone about developing another perfect and universal language: Wikilambda. This programming language will enable the people of the world to collaborate to build the constructors and renderers that will define and express the sum of human knowledge. According to the Wikilambda developers, Wikilambda is *universal* because it breaks the hegemony of English; it is *perfect* because it is not actually a language.

The main argument for Wikilambda’s *universality* is that it will break the hegemony of English. Most programming languages, observe Wikilambda’s creators, use English as a source of vocabulary. JavaScript has *objects*, *functions* and *if*-statements, rather than *Objekte*, *Funktionen* and *wenn*-statements. Since languages like JavaScript use English words, they force budding programmers to “learn English first” before they learn to program, which is unfair (2023a). To solve this problem, Wikilambda does not use words to denote parts of a computation. Instead, each part of the computation is assigned a `Z`-number or `Z`-key in the Wikifunctions database. When a person visits a function in the Wikifunctions interface, they are presented with a translation of these `Z`-numbers and `Z`-keys into their preferred language. If English is the preferred language in your browser, then when you visit function `Z802` you will see a function called `if`, with three parameters: the `condition`, `then` and `else`. If Bengali is your preferred language, you will instead see a function called with the parameters , and (Figure 2).

Accessibility is essential to the aims of the Abstract Wikipedia project, for two reasons. The first reason is practical. To write “renderers” for all the world’s languages, Wikifunctions must pool the expertise of all the world’s programmers and all the world’s native speakers. A reader of Swahili or Motu must be able to scrutinise the “renderers” for their language so they can check that they implement the correct linguistic rules. The second reason is cognitive. Abstract Wikipedia presents an extremely severe engineering challenge, and this challenge will only be soluble if *all* the world’s minds are engaged, not only those minds that are trapped in the English language:

> 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 [...] (Blanton et al. 2022)

Here a contradiction emerges. On the one hand, the project relies on the CONDUIT metaphor to prove that it does not matter what language is used to express a fact. On the other hand, here they argue that the English language is not a mere conduit, but is an “imperialist” language



যদি Z802

ফাংশন

পাতা আলোচনা

পড়ুন ইতিহাস দেখুন সরঞ্জাম

ValterVB (আলোচনা | অবদান) কর্তৃক ১৪:৪৪, ৭ অক্টোবর ২০২৪ তারিখে সংশোধিত সংস্করণ *(lower case)*
(পরিবর্তন) ← পূর্বের সংস্করণ | সর্বশেষ সংস্করণ (পরিবর্তন) | পরবর্তী সংস্করণ → (পরিবর্তন)

পরিচিতি

বাংলা
যদি

যদি সত্য তাহলে একটি আউটপুট আর যদি মিথ্যা তাহলে অন্য আউটপুট

কোনও উপনাম দেওয়া হয়নি।

ইনপুট
শর্ত: বুলিয়ান
যদি সত্য তাহলে: বস্তু
যদি সত্য না তাহলে: বস্তু

আউটপুট
বস্তু

এই ফাংশনটি পরীক্ষা করুন

ইনপুট লিখুন
শর্ত •••
সত্য মিথ্যা
- যদি সত্য তাহলে •••
  - ধরন •••
  একটি ধরন নির্বাচন করুন
- যদি সত্য না তাহলে •••
  - ধরন •••
  একটি ধরন নির্বাচন করুন

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

that imposes "Western thinking" on the world. The programmers who are attempting to break the hegemony of English are moreover English speakers, using English to describe and defend their project to overcome English. How can they cut through this thicket of contradiction?

Their solution is not to build a language at all: "Wikifunctions is not a programming language" (2024d). This is the sense in which Wikilambda is a "perfect" language. If a perfect language mirrors the true nature of objects, then Wikilambda mirrors the true nature of computation. What is the true nature of computation? Wikilambda's answer is implicit in the name of the software (Wikilambda) and the name of the online database where it runs (Wikifunctions). Computation can be boiled down to a single kind of primitive element, the *function* (or a *lambda*, to use the Greek terminology of computer science pioneer Alonzo Church (1932)). Complex computations can be created by *composing* functions together. Wikilambda allows users to define *functions* and *compose* them together "and that's it" (2024d). Wikilambda is not a "language," infected with English vocabulary, which comes between the user and the computation. It is an "an abstract syntax tree," which allows the user to modify the structure of the computation directly (Blanton et al. 2022).

There are two key problems with Wikilambda's claim to perfection. The first is that it *is* a language, as its creators uneasily admit in the "README" file for the "function orchestrator": "the orchestrator kind of implements a functional programming language" (Massaro et al. 2024a). As a result, the team have become involved in tricky choices, about things like evaluation order, recursion depth and error handling. They joke that the Wikilambda language is "something like LISP in JSON" (2024e). Wikilambda is indeed very much like LISP, but in their drive for extreme simplicity, the Wikilambda developers have overlooked key elements that make LISP usable in practice, such as tail recursion, the REPL, partial compilation, garbage collection, specialised data structures, special forms, and quasiquotation (see generally Sussman and Steele Jr. 1975; Steele Jr. 1977; Steele Jr. and Sussman 1978; Abelson et al. 1996). Simply put, even just to provide access to the "abstract syntax tree," Wikilambda will need to become a lot more powerful; to become usable enough to support the complex software for Abstract Wikipedia, it will need to become more powerful still. And through all this, the English-speaking Wikilambda developers will be telling their own "one-voice story" in the code they write for the orchestrator.

There is an alternative future, which however reveals the second problem with Wikilambda's claim to perfection. The Wikilambda language could remain extremely simple (and therefore basically unusable), and all the real programming could be done by providing "implementations" of the various "renderers" in existing programming languages. Instead of programming the Cantonese renderer in Wikilambda, for example, a programmer could simply enter a single function into the database called "render article into Cantonese," and then write an implementation of this "render" function in Python, JavaScript, or some other language supported by the Wikifunctions system. In this case, Wikilambda's claim not to be a programming language will be literally realised—it will not be used to do any programming! And all the supposed problems with "English" programming languages such as Python and JavaScript will remain.

The Wikilambda team envisage a middle course between these extremes: implementation languages such as Python and JavaScript will be used to implement particular functions in the Wikifunctions database, and the Wikilambda language will be used to compose these functions into larger wholes. Perhaps they will be able to steer this middle course, but it seems impossible that Wikilambda will be able to remain so minimal and under-specified.

If the developers do find this middle course, will they achieve their central goal, of escaping from the hegemony of English, and opening programming to the non-English-speaking world? The example of Wikidata would suggest not. Wikidata also has a language-neutral interface, which is automatically translated into the reader's preferred language. But all discussion and decision on the platform is conducted in English, because this is the world's *lingua franca* (Ford and Iliadis 2023). It is possible that Wikifunctions will avoid this fate, because parts of the platform will be language-specific. For example, functions intended to render abstract content into Ndebele might be discussed and debated in Ndebele—but then any programmers who do not already read and write Ndebele would be hampered from contributing code.

More deeply, the notion that budding programmers must "learn English first" is misconceived. The fact is that "words in code simply do not behave the way they do in spoken or written language" (Marino 2020, pp. 145–147). For example, the function `mapcar` in the Scheme programming language might look like English, but it 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*. More subtly, Python's `if` keyword has precisely three distinct meanings, depending on whether it is used to create a `statement`, an `expression` or a `guard`. Needless to say, the English word "if" does not divide into these three distinct meanings. The fact is that programming languages are themselves human languages, with their own culture and idioms (Blackwell 2017, p. 37). If a programming language contains an English word, then it is a loan-word, which becomes part of the programming language, loses some or all of its English meaning, and may well be a false friend to beginning programmers.

The Wikilambda team try to skirt this issue by creating an extremely minimal, abstract language, which exposes the raw structure of computation to the programmer, and will therefore be empty of linguistic borrowings. This strategy seems to me impossible. If users do flock to the platform, then Wikilambda will develop its own culture: idioms, metaphors, styles, ideals. It does not much matter if `Z802` is transliterated into `if` or   : this transliteration does not change the fact that `Z802` is a single word in the Wikilambda language, and acquires meaning in that language. The process of acculturation has already begun. Wikilambda is already idiomatic, metaphorical, stylish and idealistic. Its culture is expressed in the (English) documentation I have been analysing in this section. Its culture is expressed even more vividly in the source code for the "function orchestrator," which implements the Wikilambda language, and which is replete with powerful (English) metaphors.

4 Code: Abstraction and Metaphor in the Orchestrator

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

A programming language, argue Abelson et al. (1996, p. 607), is "a coherent family of abstractions erected on the machine language." The language itself provides a family of abstractions to the programmer: using JavaScript, I can create an `object` or interact with a `document` without worrying about the particular sequence of CPU instructions that makes an `object` or a `document` behave as it does. The language does not merely *provide* abstractions, however—it is *made* of abstractions. Someone has to write the code that defines what a JavaScript *object* or *document* is. The code that implements a programming language will contain many abstractions, such as *expression*, *symbol* or *token*, which are used to describe the language being implemented. Perplexingly, many programming languages are implemented *in themselves*. The C compiler, for example, is itself written in C. C provides a coherent family of abstractions, which is then used to describe and implement C, which provides a coherent family of abstractions, which is then used to describe and implement C, which …

Wikilambda is not as dizzying as C. It is a new language, which is not yet powerful enough to implement itself. It is implemented in JavaScript. In this section, I conduct a close analysis of the "function orchestrator" (Massaro et al. 2024a), the piece of JavaScript code that implements the Wikilambda language. What abstractions have the Wikilambda developers invented to describe their new language? What can these abstractions tell us about the nature and intent of their project?[2]

One answer to these questions is: not much. According to a popular view, programming "abstractions" exist merely to "hide information" (Colburn and Shute 2007). Abstractions exist to conceal the complexity of the underlying machine code from the programmer. As a JavaScript programmer, I know that eventually all the code I write will be translated into machine code that the CPU can understand. But this is all too much to think about, so I invent fictitious entities such as *objects* and *documents* to work with, allowing me to focus on the bigger picture. This idea has had a powerful effect in humanistic critiques of technology. Media theorists such as Friedrich Kittler (2014, p. 223), Alexander Galloway (2012, p. 64) and Denis Tenen (2017, p. 32) have argued that code is illusory. In reality, software is made up of "varying electric potentials" or "electromagnetic charges" in the computer hardware. The code supervenes on these electrical realities: at best, code "signifies" or "substitutes for" the electrons in the CPU; at worst, it "obfuscates" them. If this is so, then there is little reason to interpret the abstractions in source code. Why should we concern ourselves with the code, rather than with the material reality it signifies, substitutes or obfuscates?

[2]The function orchestrator (and its companion, the evaluator), are both in active development. I have analysed them at the time I cloned the code: that is commit `86514fdaa663a75c9ebbda232c35e3248adbec5d` of the orchestrator, and commit `dbfededdc4d83ff8175b2bab42b00327e1a4a6aa` of the evaluator.

I do not have space here to refute this view. Instead, I simply wish to establish the plausibility of an alternative. Such an alternative is a requirement for CCS. Though some CCS scholars have attempted to ground their practice on the "information hiding" view (Hua and Raley 2023; Rountree and Condee 2023), such attempts are in my view doomed to fail, because the "information hiding" view denies the ultimate reality of code.

As an alternative, I suggest that source code abstractions are *virtualisations* of *metaphors.* The nub of truth in the "information hiding" view is that programming abstractions are metaphorical. Abstractions provide a concrete framework for thinking about the behaviour of the machine. Some of these abstractions are so familiar that we have forgotten they are metaphors.[3] It may seem that computers literally have a "memory," but this is only because we rely on an underlying metaphor: THE COMPUTER IS A BRAIN. The Victorian computer pioneer Charles Babbage used a different metaphor: THE COMPUTER IS A FACTORY, and therefore his proposed Analytical Engine had a "storehouse" rather than a "memory" (Lovelace and Menabrea 2021). This example shows that even the *hardware* of the computer is abstract and metaphorical. Even the "varying electrical charges" of Kittler and his followers are abstractions over underlying quantum effects, which are themselves abstractions developed by human mathematicians. At no depth does the metaphorical tower of abstraction descend to a literal foundation.

Thus programming abstractions are *metaphors.* In what sense are they *virtualisations*? Programming abstractions *do things.* You can write some functions in JavaScript, deploy them to a server, and then those functions will make the Wikifunctions database appear in a visitor's web browser. The "information hiding" view sees this as a process of actualisation. The vaporous abstractions in the source code are made real only when converted into machine instructions actually executed by a computer. But this is only part of the story. The abstractions in source code certainly have the virtue of executability, but they also have other virtues, and therefore, other effects. In the future, for example, Wikilambda may be ported into another language. In this case, developers might read the Wikilambda code (written in JavaScript), and translate its abstractions into another language (such as Clojure, Rust or C++). In this case, the abstractions *frame an implementation.* In the future, Wikilambda may be established as an IEEE Standard. In this case, whichever abstractions are deemed essential to the Wikilambda language will be codified in a document. If someone wants to write their own implementation of Wikilambda, it may be compulsory for them to include certain abstractions, such as `ZWrapper` or `ArgumentState`. If they fail to implement the right abstractions, then their purported implementation of Wikilambda will simply fail to be an implementation of Wikilambda. In this case, the abstractions *provide criteria of correctness.* In the future, most crucially, Wikilambda will continue to be debugged and extended by its core developers. They will need to understand the structure and intent of the software, and their understanding will affect the future course of the project. They will understand Wikilambda's structure and intent by using abstractions. They will encounter these abstractions in the source code, in the documentation, in the conversation of their workmates, in textbooks, in blogs, at conferences, or abed on sleepless nights puzzling over fiendish bugs. In this case, the abstractions *impose a*

[3]They have become "metaphors we live by" (Lakoff and Johnson 1980).

"path of abstraction" on the software, partially determining its future development (Carrillo and Martínez 2023).

Wikilambda's source code is a privileged medium for symbolising the software's structure and intent, because the source code is executable and can be deployed to actualise the software on the Web. But, like the Pope, the code is merely *primus inter pares*, and the abstractions it expresses lead an itinerant lifestyle, travelling between documents, diagrams and discussions as the software evolves. This is why I say programming abstractions are *virtualisations.* They exist "by virtue of" their ability to shape computations.

The main virtue of programming abstractions is the virtue of functional specification. Programming abstractions say *what the machine is supposed to do.* If the machine does what it is supposed to, we say the abstraction is "correct." If the machine misbehaves, we say the abstraction has a "bug." If we do not understand the machine's behaviour, or we decide it should behave differently altogether, we say that we need a "new abstraction." The electrical charges in the hardware are not the same thing as the software, because the electrical charges can be wrong.[4] This wrongness may be the software's fault, if there is a bug. But the wrongness could also have a physical cause: defects in the hardware, electronic interference in the atmosphere, or quantum fluctuations in the circuitry.

In sum, the abstractions in source code are virtually the real thing, and scholars of CCS should treat them as such.[5]

With the help of this account, I reframe the central questions for this section: What are the key metaphors in the Wikilambda source code? How are they virtualised in the abstractions of the program?

Although the main focus of this section is the "function orchestrator," I begin with a brief quotation from the "function evaluator," another piece of software whose task is to locate and execute "implementations" of functions in the Wikifunctions database (Massaro et al. 2024b). Listing 1 vividly demonstrates the vivid, hacky, intermediate state of the Wikilambda source code. The code is so metaphorical that its symbols are only barely "abstractions." This code snippet shuts down "executors" on the server when they are no longer needed. An "executor" is a server process that executes code in a particular "native language." For instance, if Wikifunctions has recently executed some Python code, then it may have some active Python executors on the server. These need to be periodically cleaned up to keep the server available for other work. To keep track of executors it has opened, the program maintains

[4]The pattern of electrical charges will also differ for every computer chip. Indeed, due to the abstraction of chip architecture, two chips may share the same machine code, but implement that machine code using different circuitry. Taken at is extreme, the "information hiding" view implies that a piece of software becomes a new piece of software every time it is run on a different chip or with different data.

[5]I hope I have established the plausibility of this account. It raises several live problems: Is there a difference between signs and symbols (Langer 1957)? What is the relation between metaphors and abstractions (Lakoff 1993; Ragg 2006; Jamrozik et al. 2016)? What is the meaning of terms such as reality, possibility, actuality and virtuality (Bergson 1908; Deleuze 1998, 2014)? CCS must address these issues for its own sake, but also promises to enrich these venerable discussions with new data in a fascinating new kind of symbolism.

an ExecutorPool. At the end of a session, the program runs the drainTheSwamp method to empty the ExecutorPool and shut down the unneeded executors. When it drains 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—this is the method shown in Listing 1. 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. It knows whether an executor is still active on the server, because if the executor has been shut down already, it will will be either null or killed.

Listing 1 The method that 'kills' an unneeded executor on the 'hitList' after a call to 'drainTheSwamp'. Source: src/Executor.js, in Massaro et al. (2024b)

```
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 brutality 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."

The first key abstraction in the "function orchestrator" is in the name. The "orchestrator" and its central orchestrate function virtualise the metaphor: A PROGRAM IS A SYMPHONY. In this metaphor, a program is made up of many musicians, who each play their own role in the computation. The orchestrator's role is to select the right mixture of instruments to achieve the composition. The orchestrate function takes as its input a piece of Wikilambda code (a ZObject), some configuration settings (invariants) and an ImplementationSelector. Its task is to run the given Wikilambda code, using the ImplementationSelector to choose between available "implementations" in the Wikifunctions database.

It is this `ImplementationSelector` that most clearly virtualises the "orchestration" metaphor. Normally, a programming language will have *just one* way of doing each action: one function for addition, one for integer division, one for instantiating an array, and so on. If there are two ways of doing something, it would normally be up to the *programmer* to decide: perhaps there are two division routines, one that is fast and approximate and one that is slow but exact, and the programmer can select which one is appropriate for their task. The Wikilambda language is different, because there may be many ways of performing each operation, and it is the orchestrator's job rather than the programmer's to choose between them. At the time of writing, for example, there are three implementations of the "add Integers" function (`Z16693`), and six implementation of the function that checks if a word is a palindrome (`Z10096`). The programmer has no control over which "implementation" is used—the orchestrator decides. When a human programmer performs "function composition" with the help of the "function orchestrator," they really are like a composer working with a separate orchestrator. The composer-programmer writes a piano score describing the structure of the symphony, while the orchestrator decides which instruments should fill out which parts.

In other programming languages, the equivalent of the the Wikilambda "orchestrator" would be known as the "evaluator" or the "interpreter."[6] Neither Python nor JavaScript have an `orchestrate` function; instead they have an `eval` function that performs the analogous operation. The metaphor in this case is different. "Interpreting" or "evaluating" a program implies that THE PROGRAM IS A TEXT, whose meaning must be sought. The TEXT metaphor throws the emphasis on the vocabulary, syntax and intelligibility of the language. It situates the programmer as an author, the program as a poem, and the computer as a reader. The central problem for the "interpreter" is *understanding what the program says.* The SYMPHONY metaphor throws the emphasis on the machinery of computation. It situates the programmer as a composer, the program as a piano score, and the computer as an orchestrator. The central problem for the orchestrator is *getting the right instruments to play the right notes.* In the symphony metaphor, therefore, we can once again detect Wikilambda's central claim about language: that it can be done away with. If Wikilambda is not a language, it cannot be used to write TEXTS. There is no need to "evaluate" or "interpret" what the code says, because it literally says what it means. It is a perfect language, which exposes the "abstract syntax tree" of the computation directly. Accordingly, the system has only the humble task of filling in the blanks of the "abstract syntax tree," by selecting concrete "implementations" from its philharmonic database of casually-employed musicians.

As we have already seen, there is no escape from language. The orchestrator *does* interpret the code it orchestrates, but without the aid of the TEXT metaphor, the Wikilambda developers have struggled to express this process of interpretation clearly. This has led them to introduce many vague abstractions into the code, using the guiding metaphor AN ABSTRACTION IS A CONTAINER. This metaphor is virtualised most clearly the second key abstraction of the function orchestrator: the `ZWrapper` class.

[6]The "function evaluator" of the Abstract Wikipedia project is really more like the "primitive operations" than the "evaluator" of a normal programming language.

The role of the `ZWrapper` is to maintain information about the 'scope' of symbols in a Wikilambda program. This is required, because the same symbol may appear multiple times in a program with different meanings. For example, the "add Integers" function (`Z16693`) has two inputs, "left integer" (`Z16693K1`) and "right integer" (`Z16693K2`). Imagine that you implemented the following function in Wikilambda:

$$f(x) = \frac{2 + x}{3} - \frac{x + (-3)}{4}$$

Here there are at least *two* additions: $2 + x$ and $x + (-3)$. In the first case, the "left integer" (`Z16693K1`) is 2, while the "right integer" (`Z16693K2`) is x. In the second case, it is the "left integer" (`Z16693K1`) that is x, while the "right integer" (`Z16693K2`) is -3. The role of the `ZWrapper` is to keep track of which `Z16693K1` or `Z16693K2` is which. It also needs to keep track of x. If this function is used more than once in the program, then x may also have a different value each time. Keeping track of symbols is the main purpose of the `ZWrapper` class, but this role is not at all apparent from its name. What does "wrapping" have to do with *remembering the referents of symbols*? What exactly is being "wrapped"?

Listing 2 shows the code that "wraps" a `ZObject` in a `ZWrapper`. A `ZObject` is a piece of Wikilambda code from the Wikifunctions database. It can be in one of two states, as shown in line 2 of the listing. It can be a piece of unprocessed text (`isString( zobjectJSON )`), or it can be a `ZObject` that has already been "wrapped" (`instanceof ZWrapper`). If the raw code has not yet been "wrapped," the orchestrator works out if the `ZObject` is a `ZEnvelope` or not. If it *is* a `ZEnvelope`, then it wraps it up in a `ZEnvlopeWrapper`. If not, it wraps it in an ordinary `ZWrapper`. The last few lines of the function are the ones that do the work of remembering which symbol corresponds to which value. They do this by creating a `scope` in the `ZWrapper`, which is `populated` with the correct values for each symbol (e.g. remembering what the "left" and "right" integers are in an integer addition). As this explication reveals, the code does not explain itself at all. Nothing in the name `ZObject` indicates that it is an expression in the Wikilambda language. And what exactly is the difference between an `Envelope` and a `Wrapper`? These names are empty. They use the metaphor AN ABSTRACTION IS A CONTAINER to indicate that the code contains abstractions, but they do not indicate what those abstractions are or why they are needed. An envelope or a wrapper is a container for some kind of object. But for what kind of object, and why does it need to be contained?

There are many reasons why the code should be this way. The Wikilambda developers charmingly admit that they are not very experienced in language design: "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.md," Massaro et al. 2024a). But I would suggest the code reveals a deeper pattern than mere inexperience. Repeatedly in the code, the Wikilambda developers eschew well-known abstractions in programming language design. In Listing 2, for example, `createInternal` is basically a synonym for "evaluate" or "eval-dispatch," and `zobjectJSON` is a synonym for "expression." The two `if` statements at the start determine the "expression type" (there are three types), and choose the correct evaluation

Listing 2 The method that wraps a `ZObject` in a `ZWrapper` as required. Source: src/ZWrapper.js, in Massaro et al. (2024a)

```
static createInternal_( zobjectJSON, scope, parentPointer ) {
        if ( isString( zobjectJSON ) || zobjectJSON instanceof ZWrapper ) {
            return zobjectJSON;
        }
        let result;
        if ( isZEnvelope( zobjectJSON ) ) {
            result = new ZEnvelopeWrapper(); // eslint-disable-line no-use-before-define
        } else {
            result = new ZWrapper();
        }
        result.setScope( scope );
        result.parent_ = parentPointer; // will use parent in case scope does not exist
        result.populateKeys_( zobjectJSON );
        return result;
    }
```

method. The `scope` would normally be called the "environment" of the evaluation, and the `populateKeys` method is the function that "extends the environment" to include the "local bindings" of the current "expression." These are not unusual or obscure terms in the world of programming: "expression," "evaluate," "dispatch," "environment," "extend," "local" and "bindings." But they all partake of the dominant metaphor THE PROGRAM IS A TEXT, a metaphor that the Wikilambda developers have rejected in their attempt to create a perfect language that interposes no symbolism between the programmer and their program.

The code for the function orchestrator is in a medial state. The Wikilambda developers are in the process of carving out new abstractions, to describe a utopian project for a new programming language. Their central metaphor, A PROGRAM IS A SYMPHONY, allows them to describe the most remarkable feature of their language: that it lives in a data centre, where many computers, and many little software daemons, are dynamically combined to bring the language to life. When they write code that summons and dismisses resources on the Wikimedia Foundation's servers, they write vivid, clear, metaphorical code (e.g. Listing 1). But when the SYMPHONY metaphor fails—as in the description of the vocabulary and syntax of their new programming language—the Wikilambda developers fall back on the empty metaphor, AN ABSTRACTION IS A CONTAINER (e.g. Listing 2). The SYMPHONY metaphor has imposed a "path of abstraction" on the developers, which has made it difficult for them to explore and express certain aspects of their project (Carrillo and Martínez 2023). Perhaps in years to come they will carve out genuinely new abstractions for programming language design; in the mean time, they probably will continue to resist the best available metaphor, A PROGRAM IS A TEXT, because this metaphor contradicts their aim to create a perfect language.

5 Conclusion

But it is perhaps nothing more than our 'democratic' illusion to imagine that perfection must imply universality. (Eco 1995, p. 100)

The Abstract Wikipedia/Wikifunctions project has a profoundly moral aim: to give human beings control over information in the Age of GenAI. If the problem were simply to populate minority-language Wikipedias with articles, it would be simpler just to get a Large Language Model (LLM) to translate English Wikipedia into those languages. But Wikilambda presents a stark alternative to LLMs such as Gemini, Llama or ChatGPT. LLMs rely on vast concealed datasets. Wikifunctions draws its data openly from public Wikimedia databases. LLMs generate text using opaque algorithms that even their designers struggle to control. Wikilambda makes every part of every algorithm available to anyone. In short, Wikilambda is *contestable*. If you ask an LLM 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 construct or render the article in Wikifunctions; or sever the connection between the article and Abstract Wikipedia, and edit the article 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). 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.

To achieve their aims, the Wikilambda developers are attempting to escape from language. They want to escape from spoken languages into the template language of Abstract Wikipedia. They want to escape from programming languages into the abstract syntax tree of Wikilambda. So far, however, they have only escaped *from* language *into* language. The whole Abstract Wikipedia/Wikifunctions project is conceived in terms of the CONDUIT metaphor, a questionable metalinguistic abstraction baked into the English language (Section 2). The Wikilambda language tries to be no language at all, but is irresistibly becoming another human tongue, with its own culture and idioms (Section 3). In the code of the function orchestrator, the developers tell a "one-voice story" using English metaphors, and they struggle to express themselves in the code, because they have set out on a path of abstraction that forecloses the best available metaphor for what they are doing (Section 4).

In response to criticism, the Wikilambda developers adopt one of the most popular programming metaphors: THE WORLD IS TOLKIEN'S MIDDLE-EARTH (Dillon and Schaffer-Goddard 2023). What they wish to avoid is a " 'One Ring' solution" to the problem of universalising access to knowledge (Blanton et al. 2022). In this metaphor, the Google fellows who criticised the project are agents of Sauron, the Dark Lord. When the fellows suggest that the project abandon its quixotic quest to design the perfect programming language, when they suggest that perhaps an existing technology such as Scribunto or Grammatical Framework might be appropriate, these agents of Sauron (Google) are really suggesting that Abstract Wikipedia forge a magic ring in the fires of Mt Doom (Silicon Valley), and use it to dominate the world.

The tragic contradiction of the Abstract Wikipedia project is that it is itself a " 'One Ring' solution." A group of Anglophone engineers in the heart of Mordor (California) are forging one language that all the world's writers should write, and another language that all the world's programmers should program. They attempt to resolve this contradiction by escaping from language. The template language will boil facts down to their primitive elements. The Wikilambda language will boil computations down to their primitive elements. From these primitive elements the whole world of facts, and the whole universe of computations, will be recombined without the need for words. But words "bite and scratch," writes the poet, and "You never learn / the chemical process of separating them" (Frame 2008, p. 117). It is the grand ambition of Denny Vrandečić and his hackers to learn this chemical process, which has evaded so many linguistic alchemists before them.

References

(2024e) Wikifunctions:Function model. Wikifunctions

(2023a) Wikifunctions:Vision. Wikifunctions

(2024d) Wikifunctions:What Wikifunctions is not. Wikifunctions

(2023b) Annual Report 2022-23. Wikimedia Endowment, San Francisco

(2024a) Abstract Wikipedia/Architecture. Wikimedia Meta-Wiki

(2024b) Abstract Wikipedia/Components. Wikipedia Meta-Wiki

(2024c) Extension:WikiLambda. MediaWiki

Abelson H, Sussman GJ, Sussman J (1996) Structure and interpretation of computer programs, Second. MIT Press, Cambridge

Avieson B (2022) Two Wikipedias in Bhutan: Problems and solutions for knowledge equity in the digital age. Asian Journal of Communication 32:399–416. https://doi.org/10.1080/01292986.2021.1937248

Bayer T (2023) Wikimedia Foundation's Abstract Wikipedia project 'at substantial risk of failure'. The Signpost

Bergson H (1908) Essai sur les données immédiates de la conscience, 6th edn. Alcan, Paris

Blackwell AF (2017) 6,000 Years of Programming Language Design: A Meditation on Eco's Perfect Language. In: Diniz Junqueira Barbosa S, Breitman K (eds) Conversations Around Semiotic Engineering. Springer International Publishing, Cham, pp 31–39

Blackwell AF (2024) Moral Codes: Designing Alternatives to AI. The MIT Press

Blanton C, Massaro C, Martin D, et al (2022) Abstract Wikipedia/Google.org Fellows evaluation - Answer. Wikimedia Meta-Wiki

Carrillo N, Martínez S (2023) Scientific Inquiry: From Metaphors to Abstraction. Perspectives on Science 31:233–261. https://doi.org/10.1162/posc_a_00571

Church A (1932) A Set of Postulates for the Foundation of Logic. Annals of Mathematics 33:346–366. https://doi.org/10.2307/1968337

Colburn T, Shute G (2007) Abstraction in Computer Science. Minds and Machines: Journal for Artificial Intelligence, Philosophy and Cognitive Science 17:169–184. https://doi.org/10.1007/s11023-007-9061-7

Crawford K, Gillespie T (2016) What is a flag for? Social media reporting tools and the vocabulary of complaint. New Media & Society 18:410–428. https://doi.org/10.1177/1461444814543163

Deleuze G (2014) Difference and Repetition, 2nd edition. Bloomsbury Academic, London; New York

Deleuze G (1998) Le bergsonisme, 2. ed. Presses Univ. de France, Paris

Dillon S, Schaffer-Goddard J (2023) What AI researchers read: The role of literature in artificial intelligence research. Interdisciplinary Science Reviews 48:15–42. https://doi.org/10.1080/03080188.2022.2079214

Dubrow J (2018) Cosmopolitan Dreams: The Making of Modern Urdu Literary Culture in Colonial South Asia. University of Hawaii Press, Honolulu

Eco U (1995) The search for the perfect language. Blackwell, Oxford, UK ; Cambridge, Mass., USA

Ford H (2022) Writing the revolution: Wikipedia and the survival of facts in the digital age. The MIT Press, Cambridge, Massachusetts

Ford H, Iliadis A (2023) Wikidata as Semantic Infrastructure: Knowledge Representation, Data Labor, and Truth in a More-Than-Technical Project. Social Media + Society 9:20563051231195552. https://doi.org/10.1177/20563051231195552

Ford H, Sidoti F, Falk M, et al (2024) How Australian places are represented on Wikipedia. University of Technology, Sydney

Foundation W (2023) First grants announced from the Wikimedia Endowment to support technical innovation across Wikipedia and Wikimedia projects. Wikimedia Foundation

Frame J (2008) Storms will tell: Selected poems. Bloodaxe Books, Tarset, Northumberland

Galloway AR (2012) The interface effect. Polity, Cambridge, UK ; Malden, MA

Gnatiuk O, Glybovets V (2021) Uneven geographies in the various language editions of Wikipedia: The case of Ukrainian cities. Hungarian Geographical Bulletin 70:249–266. https://doi.org/10.15201/hungeobull.70.3.4

Hakala W (2016) Negotiating Languages: Urdu, Hindi, and the Definition of Modern South Asia. Columbia University Press, New York, UNITED STATES

Hoffmann D (2019) Restrictions on the Usage of Spatial Frames of Reference in Location and Orientation Descriptions: Evidence from Three Australian Languages. Australian Journal of Linguistics 39:1–31. https://doi.org/10.1080/07268602.2019.1542927

Hua M, Raley R (2023) How to Do Things with Deep Learning Code. Digital Humanities Quarterly 17:

Jamrozik A, McQuire M, Cardillo ER, Chatterjee A (2016) Metaphor: Bridging embodiment to abstraction. Psychonomic Bulletin & Review 23:1080–1089. https://doi.org/10.3758/s13423-015-0861-0

Kittler FA (2014) There is no software. In: The Truth of the Technological World: Essays on the Genealogy of Presence. Stanford University Press, New York, UNITED STATES, pp 219–229

Lakoff G (1993) The contemporary theory of metaphor. In: Ortony A (ed) Metaphor and Thought, 2nd edn. Cambridge University Press, pp 202–251

Lakoff G, Johnson M (1980) Metaphors we Live By. The University of Chicago Press, Chicago; London

Langer SK (1957) Philosophy in a new key; a study in the symbolism of reason, rite, and art., [3d ed.]. Harvard University Press, Cambridge

Livneh O, Gitman A, Assaf A, Yang M (2022) Abstract Wikipedia/Google.org Fellows evaluation. Wikipedia Meta-Wiki

Lovelace A, Menabrea LF (2021) Sketch of the Analytical Engine (1843). In: Lewis H (ed) Ideas That Created the Future. The MIT Press, pp 9–26

Marino MC (2020) Critical Code Studies. The MIT Press

Massaro C, Vrandečić D, Smit D, et al (2024a) Function-orchestrator

Massaro C, Vrandečić D, Smit D, et al (2024b) Function-evaluator

Ragg E (2006) Pragmatic Abstraction vs. Metaphor: Stevens' "The Pure Good of Theory" and "Macbeth". The Wallace Stevens Journal 30:5–29

Reddy MJ (1993) The conduit metaphor: A case of frame conflict in our language about language. In: Ortony A (ed) Metaphor and Thought, 2nd edn. Cambridge University Press, pp 164–201

Rountree B, Condee W (2023) Nonsense Code: A Nonmaterial Performance. Digital Humanities Quarterly 17:

Steele Jr. GL (1977) Debunking the "expensive procedure call" myth; or, procedure call implementations considered harmful; or, lambda: The ultimate goto. Massachusetts Institute of Technology, Cambridge, Mass

Steele Jr. GL, Sussman GJ (1978) The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two). Massachusetts Institute of Technology, Cambridge, Mass

Sussman GJ, Steele Jr. GL (1975) Scheme: An interpreter for extended lambda calculus. Massachusetts Institute of Technology, Cambridge, Mass

Swift J (2005) Gulliver's Travels. OUP, Oxford

Tenen D (2017) Plain text: The poetics of computation. Stanford University Press, Stanford, California

Thiong'o N wa (1986) Decolonising the Mind: The Politics of Language in African Literature. Boydell & Brewer, Limited, Woodbridge

Tkacz N (2015) Wikipedia and the politics of openness. University of Chicago Press, Chicago ;

London

Vee A (2017) Coding Literacy: How Computer Programming Is Changing Writing. The MIT Press

Vrandečić D (2021) Building a multilingual Wikipedia. Communications of the ACM 64:38–41. https://doi.org/10.1145/3425778

Vrandečić D (2018) Capturing meaning: Toward an abstract Wikipedia. In: International Semantic Web Conference 2018 - Outrageous Ideas Track. Monterey, CA

Vrandečić D (2020) Collaborating on the Sum of All Knowledge Across Languages. In: Reagle J, Koerner J (eds) Wikipedia@20: Stories of an Incomplete Revolution. MIT Press, Cambridge, Mass, pp 175–188

Wark M (2004) A Hacker Manifesto. Harvard University Press, Cambridge, Mass