Features of a dream programming language: 2nd draft

Magne Matre Gåsland¹

¹Affiliation not available

February 17, 2022

Since I improved [the original article](#) quite a bit, I saved the original for posterity. To respect the contents of the original URL (where it may have been shared), and to respect the ensuing [Hacker News discussion](#) based on the original article. The original article received some [encouraging feedback](#) from [Ruby's](#) creator Matz, and positive feedback and excellent critique from [Roc's](#) creator Richard Feldman.

The article is a long read, so I suggest first skimming it and dipping into the sections where you find interest or disagreement. Then, if your curiosity is piqued by the ideas, I hope you will take time read it carefully over a few evenings, mull over the ideas presented (a few are novel), consider your own dream language features, and contribute those, as well as your own insights and experience, in the comments below.

TL;DR / Summary:

I long for a very constrained language for web-app + systems development, prioritizing readability (left-right and top-down) and reason-ability above all, which is designed for fast onboarding of complete beginners (as opposed to catering to a specific language community who already have the [curse of expertise](#)).

The most important familiar features it should have: - Functional Programming, but based on data-first piping. - Immutability, but w/ opportunistic in-place mutation. - Gradually typed: dynamic for development, static for production (w/ full & sound type inference). - CSP for concurrency (goroutines). - Ecosystem: interoperable with existing languages. - Transpiles to JS or compiles to WASM.

The most important esoteric features it should have: - Crucial evolvability / backward- and forward-compatibility. - Content-addressable code. - Transparent upgrades without any breaking changes. - Data First Functional Programming w/ SVO-syntax. - Interpreted, for development. But compiled, incrementally, for production. - Interactive: facilitates an IDE-plugin (VS Code) that shows the contents of data structures while coding. Enable REPL'ing into a live system. - Aggressively parallelizable. Inspired by Verilog and Golang. - Scales transparently from single CPU to multi-core to distributed services, without needing refactors.

The complete set of desirable features are detailed below.

First, a few overarching guiding principles:

- **All programming languages are, in actuality, built to overcome *human* limitations.** Otherwise, one might as well be typing 0's and 1's, or a lower-level language like Assembly.
- Software architecture in general (and frameworks in specific) is *a way to organize the mind* of the developer(s), categorising the conceptual world into what's closely or merely remotely related (giving

rise to principles like ‘cohesion over coupling’ etc.). (This might explain OOP’s popularity.) The machine would be perfectly content with executing even spaghetti code. Inspired by [Martin Fowler](#).

- A programming language has certain [affordances](#), allowing you to talk specifically about/with some concepts (typically the [first-class citizens](#) of the language), and avoid having to talk about other things (e.g. memory management, language runtime concerns). This does not only apply to [DSL](#)’s.
- “each programming language has a tendency to create a certain mind set in its programmers. ... you tend to have a mental model of how to do things based on that language. ... Such a mind set may make it difficult to conceive of solutions outside of the model defined by the language.” - [Dennis J. Frailey](#)
- Only by accounting for human limitations (like cognitive capacity, and familiarity), could one derive a specification for the ideal programming language.
- **A bug is an error in thinking.** Either by the developer, or the language-designer for not sufficiently accounting for human psychology ([Sapir-Whorf: the language you write/speak determine what you can/do think](#)). Even [Dijkstra](#) himself equates programming with thinking. Programming does not simply *require* thinking, but *structure* thinking.
- To reduce bugs, a language should ensure **simple, safe, and scalable *ways of thinking***. For instance:
 - [Type systems](#) are a way to use the compiler to help us [verify our beliefs](#) about our own code: they help us think consistently.
 - [Closures](#), enables the programmer to specify and share behaviors that are already half-way thought through (i.e. already set up with some external data).
 - [Transducers](#), allows the programmer to define and [compose behaviors/processes](#) without having to think explicitly about the particular thing which is behaving.
 - [Currying](#), allows the programmer to take one grand behavior and break it down into smaller behaviors that can be reused independently, or used in sequence.
 - Composition allows the programmer to think & build piece by piece, instead of all at once, and without the context influencing too much.
 - “[Open for extension](#), closed for modification”: A programmer can recognize something useful, and add more pieces to it, without having to change the original thing (e.g. extension methods in C#), and without tying those new pieces too closely with the original thing (e.g. subclassing), thereby limiting their reuse.
- A language determines **WHAT** you can & have to think about, but also **HOW** you have to think about it.
- **“Things that are different should look different”**. Counter-inspired by [D](#). Inspired by [Lary Wall on Perl’s postmodernism](#) and [my own frustrations with modern component frameworks like React](#), and my impression that Lisp/Clojure is [perceived as hard to learn](#) because it has [so little syntax](#): when everything [looks the same](#) it is [hard to tell things apart](#). **Syntax matters**, so when people balk at noise like too many parentheses, you ought to listen ([like React did](#)), rather than ignore it ([like Clojure did and still does](#); even though the Lisp inventor [McCarthy didn’t even like the S-expressions syntax](#) himself), in order to [remove friction](#) from onboarding and growth. It can be the barrier between mass adoption and remaining niche. A language should optimize for mass adoption, due to network effects, since then *everyone wins*: learning, communication, portability, ecosystem etc. It doesn’t mean pleasing everyone’s ultimate desires, just avoid having most people turn at the door. Counter-inspired by Lisp/Clojure, and [Haskell](#). Although it is prize-worthy to **stay very frugal with syntax**, since more syntax necessitates more learning/documentation (knowledge debt, info overload), more avenues for confusion (the best code is no code), and more complications (language intricacies can lead to software intricacies, which can lead to bugs). My philosophy leads more towards [Golang](#) ([less features](#),

readability is reliability, simplicity scales best) and [Python](#) (“explicit over implicit”, “one way over multiple ways”), than [Ruby](#) (provide sharp knives) and [Perl](#) (postmodern plurality, coolness/[easiness](#) is justification enough in itself, aka. [the smell of a toy language](#)). Even though I come from Ruby and love it, and also cannot help admiring Lisp for its elegance and [crucial evolvability](#).

- [Programming should be fun and painless](#). Inspired by Ruby.

Purpose: What should this dream language of mine primarily be for?

- Webapp / app + systems development. In Rich Hickey’s words: [Information-driven situated programs](#). Ideally, also open to extension into [more areas of programming](#).
- Scripting and prototyping, but also scalable to production use (app/webapp)
- Systems development (compiled)

I believe there is enough cross-over between web app (in-browser + web server) and systems development that a turing-complete language could address both successfully. The language Go / Golang tries to do that (for web servers and systems), for instance. Though the question is up for debate whether or not it is a good idea, or if we should always have specialized languages for those domains.

So, from various sources of inspiration, and the aforementioned principles in mind, here is the list of features that my dream programming language would have.

Features:

Features in **bold** are considered most important.

- **Readability and reasonability** as top priority. Reduce dev mind cycles > reduce CPU cycles. Human-oriented and DX-oriented. Willing to sacrifice some performance, but not much, and not to overly gain comparability with natural language (counter-inspired by [SQL](#); inspired by Cypher). Willing to sacrifice immediate power in the language itself, esp. if that can be achieved through abstracted-away libraries.
- Should always be able to be read top-to-bottom, left-to-right. No `<expression> if <condition>` like Ruby allows. [Certainly](#) no `<code.block> while <condition>`.
- [Syntax matters](#): Readability should not imply a one to one match with natural language (counter-inspired by [SQL](#)), since natural language is inconsistent, duplicitous, ambivalent and multi-faceted. Consistency is key to a programming language. But it should borrow some similarities from natural language (like its popular Subject-Verb-Object, SVO, structure; see also DFFP) to make adoption easier (more at-hand/intuitive). The SVO syntax also aligns elegantly with the [Graph Data model of RDF](#) (subjec-predicate-object triples), and if code-is-data (homoiconicity or pseudo-homoiconicity is preserved) it could be interesting to have the code map well to a graph database, opening up avenues for analysis in the form of [advanced graph algorithms](#), which could be useful for, say, code complexity analysis (e.g. more straightforward [cyclomatic complexity](#) analysis). Homoiconicity (code structure mirroring a data structure) could potentially also help with respect to typing, since we want to be able to execute the same code at compile-time (statics) and run-time (dynamics), to avoid [the biformity and inconsistency of static languages](#): *“the ideal linguistic abstraction is both static and dynamic; however, it is still a single concept and not two logically similar concepts but with different interfaces.”* To avoid duplication. This is also inspired by [NexusJS](#) and [io-ts](#): *“The advantage of using io-ts to define the runtime type is that we can validate the type at runtime, and we can also extract the corresponding static type, so we don’t have to define it twice.”*

- **Isolation / Encapsulation.** To analyze a program (i.e. break it down and understand it in detail), you need to be able to [understand parts of the program in isolation](#). So everything should be able to be encapsulated (all code, whether on back-end and front-end), since **encapsulation affords reasonability** (and testability), by limiting places bugs (i.e. errors in thinking) can hide. Counter-inspired by Rails views (sharing a global scope) and instance variables. Inspired by the testability of pure functions.
- No need to manipulate data structures in the human mind. Programmer should always be able to **see the data structure** he/she is working on, at any given time, in the code. Inspired by [Bret Victor](#), and Smalltalk. Ideally with example data, not only the data type. Also, it should be possible to visualise/animate an algorithm. Since [“An algorithm has to be seen to be believed”](#), as D.E. Knuth said. It shouldn't be necessary for the programmer to take the effort to visualize it in his mind (with the error-proneness that entails). So the language should make such visualization and code-augmentation easy for tooling to support. But without being a whole isolated universe in its own right like a VM or an isolated [image](#). Counter-inspired by Smalltalk. Some have described this as [REPL-driven-development](#), or interactive-programming. Especially good for debugging: getting the [exact program state from loading up an image of it](#) that someone sent you. Inspired by Clojure. But with the ability to see the content of the data structures within your IDE. Inspired by [QuokkaJS](#). The REPL-driven development approach should ideally afford simply changing code in the code editor, detecting the change, and showing the result then and there, without you having to go to back-and-forth to a separate REPL-shell and copy-pasting / retyping code. Inspired by [ClojureScript](#). In fact, since a program is about binding values to the symbols in your code, when running your code, the IDE, enabled by the content-addressable code feature), could replace variables in the text with their bound values, successively. Effectively animating the flow of data through your code. Without you having to go to an external context like a debug window to see the bindings.
- Params: Function parameters must be named, but no need to repeat yourself, if the argument is named the same as the parameter (i.e. [keyword arguments can be omitted](#)). Inspired by JS object params, and Ruby. Counter-inspired by the [Mysterious Tuple Problem](#) in Lisp, and inspired by the [labeled arguments](#) in ReasonML, and the [interleaving of keywords and arguments](#) in Smalltalk. If currying, then input params should be [explicit](#) at every step (for clarity, refactorability and to aid the compiler). Counter-inspired by Point-free style in FP (since “explicit is better than implicit”, inspired by Python). Should [probably not](#) auto-curry functions, since “it makes default, optional, and keyword parameters difficult to implement”.
- No [Place-oriented programming \(PLOP\)](#), iow. avoid order-dependence at almost any cost, since it isn't adaptable/scalable. Inspired by Clojure. This goes for [reorderability of expressions](#) due to pure functions having no side-effects. Such reordering is desired (see: “4.2 Complexity caused by Control”, [Out of the Tar Pit, 2006](#)) since it allows structuring programs in a finish-to-start/high-to-low-level manner, enabling the reader to incrementally drill down into the code with the underlying implementation (the same [reason that JS has function hoisting](#)). Order-independence also goes for parameter lists to functions. I don't want to have to use a `_` placeholder for places where there *could* be a parameter, just because I didn't supply one. Shouldn't have to [sacrifice piping](#) just to get named arguments, either (piping should use an [explicit pipe operator](#)). Counter-inspired by Elm, and inspired by Hack. Consequence (?): would need a data structure like a [record](#) but which ideally can be accessed in an order-independent manner (similar to a map). Plus, functions should be able to take in such records but also a **rest** parameter that can represent an arbitrary number of extra fields (to make functions more reusable and less coupled to their initial context, e.g. they should be able to be moved up/down in a component hierarchy without major changes to their parameter lists). Counter-inspired by Elm.
- No **unless** or other counter-intuitive-prone operators. Counter-inspired by Ruby.
- No abstract mathematical jargon. Counter-inspired by Haskell. As it impedes onboardig, and induces a [mindset of theorizing](#) and [premature abstraction/generalization](#) that impedes rapid development.

Should be accessible for as wide a community as possible, with as little foreknowledge as possible. Inspired by [Quorum](#).

- Do not presume contextual knowledge. In UX this is known as “No modes!”. Code should be able to be read A to B without having been educated/preloaded with any foreknowledge (like ‘in this context, you have these things implicitly available’). Counter-inspired by class inheritance and Ruby magic, and JavaScript’s [runtime bound this](#) keyword and associated [scoping problems](#). Turns out too much dynamism (runtime contextualisation) can be harmful. Counter-inspired by JavaScript: *“In JS, this is one truly awful part. this is a dynamically scoped variable that takes on values depending on how the current function was invoked.”* – [sinelaw](#).
- Should facilitate and nudge programming in the language towards code with low [Cognitive Complexity](#) score.
- **Dynamic Verboseness:** Should be able to show more/less of syntax terms in the code (think: length of variable names). Beginners will want more self-documenting code. Whereas experts typically desire terser code, so they can focus on their the problem domain without clutter from the language (e.g. mathematics). A programmer will typically gradually go from beginner to expert on a codebase, even his own. See: content-addressable code. Content-addressable code would afford dynamic verboseness, which is important because: *“A language should be designed in terms of an abstract syntax and it should have perhaps, several forms of concrete syntax: one which is easy to write and maybe quite abbreviated; another which is good to look at and maybe quite fancy... and another, which is easy to make computers manipulate... all should be based on the same abstract syntax... the abstract syntax is what the theoreticians will use and one or more of the concrete syntaxes is what the practitioners will use.”* – [John McCarthy, creator of Lisp](#).
 - **Dynamic code-arrangement:** code should be able to be rearranged in order of start-to-finish/low-to-high-level or finish-to-start/high-to-low-level, because each is beneficial at various points when writing/reading code. Today, code is fixed how it is written. Typically in either an imperative way, displaying steps chronologically from start-to-finish, akin to chaining e.g. `data.func1.func2.func3`, or in a functional way: from outside-in, e.g.: `func3(func2(func1(data)))`, but when reading to understand the order of execution you have to read from the innermost to the outermost expression. This is problematic, at various times, and has arguably made functional programming less accessible to newcomers. But both ways of reading are desirable, at different times: At one time you just care about viewing the overall result/conclusion (e.g. `func3` and what it returns), and potentially working your way backwards/inwards a little bit. Maybe you start with an end goal in mind (name of a function), and then drill down to a more and more concrete implementation. But at another time you care about going the other way around: seeing how it is executed from start to the finishing result/conclusion (think: piping). This duality of thinking/reading mirrors how we approach reading in other domains. This feature could be enabled by content-addressable code, since the arrangement of code itself could be made more malleable and dynamic. See: content-addressable code.
- Not indentation based (counter-inspired by Python), since it is brittle: copy/paste bugs when the source and destination are indented differently. But also not require semicolons. Inspired by Ruby, and semi-colon-free JS. Might consider indentation-based if the language has a standard style and formatter.
- Fast feedback to the programmer is second top priority. Inspired by TypeScript hints, [QuokkaJS](#) (!), Webpack Hot Reload, and Expo Live Reload.
- REPL / interactive shell. Can be done even if compiled, by having an interpreter on top of a VM to the compiler.
- **Refactorability / change-ability.**

- Similar-looking and non-interacting code-lines should be able to change place without breaking anything. Counter-inspired by not able to add comma to last line in JSON, not being able to re-order/extract from comma-separated multi-line variable declarations in JS, and also counter-inspired by the contextualised expression terminators in Erlang.
- Consistent syntax, optimized for code refactoring: “The language syntax facilitates code reuse by making it easy to move code from local block - local function - global function”. Inspired by [Jai](#).
- **Backward- and forward-compatible.** Should be able to not worry about (or make poor future tradeoffs due to) backward-compatibility. (Counter-inspired by [ECMAScript](#).) To make the language optimally and freely evolvable, and worrisome to upgrade. [Backward-compatibility](#) and [Forwards-compatibility](#): Code in a one language version should be transformable (in a legible way) to another version (both ways; backward and forward). In the case of lossy changes, the old version should be stored (so it is revertible), and in the case of a “gainy” changes then the compiler should notify the programmer where in the code it is now missing explicit information (based identifying locations in the code where the old language constructs are used). There should also be solutions to either: have simple CLI tools to **automatically refactor old code to new language versions**, to always stay optimally **adaptable**, without having breaking changes. Maybe using some form of **built-in self-to-self transpilation**. Will likely need to be able to treat code-as-data. Might need compile-time macros. Or a solution could be to: with every breaking language revision, include an incremental language adapter, which would allow upgrading whilst ensuring backward compatibility. Could be solved with [Mechanical Source Transformation](#), enabled by [gofmt](#), so developers can use [gofix](#) to automatically rewrite programs that use old APIs to use newer ones. Which is crucial in **managing breaking changes**. A breaking (aka. widely deviating) change, should in effect, not actually break anything (that current languages and systems do, is considered a “pretty costly” design flaw).
- [Content-addressable code](#): names of functions are simply a uniquely identifiable hash of their contents. The name (and the type) is only materialized in a single place, and stored alongside the AST in the codebase. Avoids renaming leading to breaking third-parties, and avoids defensively supporting and deprecating several versions of functions. Avoids codebase-wide text-manipulation, eliminates builds and dependency conflicts, robustly supports dynamic code deployment. Code would also need to be stored immutably and append-only for this to work. All inspired by [Unison](#).
- Configuration, but with sane conventional defaults. (Not rely solely on convention over configuration (CoC), due to potentially too much implicitness/magic. Counter-inspired by Ruby on Rails.) “if you apply [CoC] dogmatically you end up with an awful lot of convention that you have to keep in your head. It’s always a question of balance; Hard Coding vs. Configuring vs. Convention, and it’s not easy to hit the optimum (which depends on the circumstances).” as [Peer Reynders reminded us](#).
- The language should be “open to extension” by any in the community, without permission. So that it can evolve and converge to a consensus, based on real-world experience and feedback. This is mirrored in the important talk [Growing a Language](#), by [Guy Steele](#), and the point on [crucial evolvability](#). But the culture of the language community should *not* encourage bending the language in unintended ways just for the sake of it, as staying close to the overarching general-purpose language (GPL) makes knowledge transfer/usage generally applicable across domains (being able to move between projects within the same language should *in general* be made easy), and afford a more cohesive ecosystem. Counter-inspired by [DSL](#)’s (see: avoid DSL’s).
- Modularity. Module system which is sensible. Counter-inspired by the NodeJS controversy. Code-splittable and [tree-shakeable](#). Inspired by Rollup. [Function-level dead code elimination](#), inspired by Elm. This is possible in Elm because functions there cannot be redefined or removed at runtime. This could potentially conflict with the envisioned Hot Upgrade feature inspired by Clojure (see: [Interactive](#)). This problem could perhaps be removed by disallowing modification/overloading of functions and data types in the standard library (or any 3rd party library). Alternatively: it should be possible

to specify what parts of the application should be tied down and optimized (the client code), and which part should, at the potential expense of larger assets, be Hot Upgradeable (the server code).

- Standard library should even be tree-shakeable. Inspired by [Zig](#).
- **Quick to get started and produce something.** Inspired by JS. Counter-inspired by JS tooling.
- Not too unfamiliar (to a large group of programmers, and to what they teach in universities). “Familiarity and a smooth upgrade path is a really big deal.” [source](#)
- Sensible, friendly, and directly helpful error messages. Inspired by Elm.
- **Data First Functional Programming (DFFP).** Mimics the style of object-orientation, but is simply structs and functions under-the-hood. (Also: functional programming patterns over procedural code.) Because it is human to see and visualize the world ([as well as computing](#)) in terms of objects and verbs, and to use verbs to signify the [causal relations between objects](#). Focusing to heavily on only one of the paradigms (OOP or FP) can typically lead to anti-patterns (God classes/objects, Factory objects, and Singletons, in OOP), or program structures far removed from the business domain model which also has linguistically unintuitive syntax, as in FP (c.f. [FP is not popular because it is backwards](#)). This is since only 12% of natural languages start with the verb, either Verb-Subject-Object (VSO) or Verb-Object-Subject (VOS), [ref](#)), as FP tends to do. So [Subject-Verb-Object \(SVO\)](#) should be preferred. 88% of natural languages start with something concrete, the Subject/Object (and the Object of one sentence is typically the Subject of the next sentence; similar to call chains). I think the SVO alignment is one significant but under-appreciated reason for OOP’s success. That, together with enabling a stepwise/imperative construction of programs, makes for a more intuitive approach for beginners, which is vitally important for onboarding & growth. “Objects and methods” could be merely syntax sugar for structs and functions (see: [interchangeability of method-style and procedure-call-style](#), or the [pipe first operator](#) in ReScript, which also illustrates *emulating object-oriented programming*), if one leaves out troublesome inheritance (which might be good, since composition > inheritance). Inspired by Golang. The quote “Data, not behavior, is the more crucial part of programming.” [is attributed to Linus Torvalds and Fred Brooks](#). If data is the focus-point, the language should mirror that. Interestingly, in addition to the more intuitive API, data-first also [affords](#) better IDE integration, simpler compiler errors, and more accurate type inference. Inspired by ReScript.
- Functional programming patterns like `.map`, `.filter`, over procedural code like `for-loops` etc., since the latter would encourage mutating state, and we want immutability.
- [Tree-shakeable code](#) (esp. useful for client-server webapps). So it should need a source code dependency between the calling code and the called function. Which makes the language more FP than OOP, according to one [definition of FP vs. OOP](#). In general, shifting concerns from runtime to compile-time is considered a good thing, as it makes the language more predictable, optimizable, and affords helpful coding tools. Having consequences of code changes appear at runtime is a bad thing (see: [The Fragile Base Class problem of OOP](#)).
- [Referentially transparent expressions](#). Which means variables cannot be reassigned, so a name will always refer to the same value (see principle: “Things that are different should look different”). Inspired by Haskell. This feature should enable [a high degree of modularization](#) but could also lead to easy automatic parallelization and memoization.
- [Formally verifiable](#) / provable. Nice-to-have, not must-have.
- [Automatic TCO \(tail-call optimization\)](#). To keep the processing lean, and avoid potential stack overflow, by avoiding allocating new stack frames for recursive function calls. Counter-inspired by [Clojure / JVM](#). Inspired by [Scheme and ML-languages](#), and [Lua](#). TCO should align well with the desire to have a language where the programmer can have and mutate a mental model to carry forward, without having to rely on remembering and returning to previously remembered values.

- **Aggressively Parallelizable:** Parallelization made natural. [Aided by pure functions](#). The language should nudge programmers and make it easy/natural to use parallelism, through language constructs (like [executing several sequential lines simultaneously](#)). To avoid common [overly sequential thinking](#), which can lead to suboptimal performance (due to not parallelizing work). But humans think sequentially. So we ought to pay heed to [Dijkstra's wisdom](#) that since: our intellectual powers to visualize processes evolving in time are relatively poorly developed, we should shorten the conceptual gap between the static program and the dynamic process, by spreading out the process in text space. In simpler terms: enabling the programmer to *trace the flow of execution* by simply reading the code. Another important reason the language should steer the programmer to aggressively utilize parallelization, is due to [Amdahl's Law](#), which states that, when parallelizing, the limiting factor will be the serialized portion of the program. Notably the queueing delay, due to contention over shared resources, like CPU time. So whatever part of the program which is not parallelized, will eventually, under high enough load, turn into a bottleneck. The language construct nudging developers to parallelization could be inspired by [Verilog's fork/join](#) construct, or the very similar [Nurseries](#), which are an alternative to `go` statements (since `go` statements don't afford local reasoning, automatic error propagation and reliable resource cleanup, though some may be achieved in a [WaitGroup](#)). But as opposed to the [fork/join example](#), the language should enforce a deterministic order upon joining, which should simply be guaranteed by the sequential top-down order of the lines in the fork/join code block (a novel idea, to my knowledge, which would need to be experimented with thoroughly... more thoughts in [this issue](#)). NB: Need to research if on today's hardware, automatic parallelization could in fact be a potential pessimization in practice instead of an optimization, as Richard Feldman pointed out. In any case, we do not envision making every function call parallelized, but to make simple, contained constructs (like nurseries, fork/join), that the programmer can use to signify separable pieces of the problem/algorithm. The runtime should then parallelize those portions aggressively.
- Alternatively: Take inspiration from [Chapel](#), by providing core primitives to control parallelization directly. But preferably through declarative means and not through such imperative control structures as for-loops, which [Chapel uses](#).
- Alternatively: Take [inspiration from Golang's elimination of the sync/async distinction](#) and allow programming everything in a sequential manner, but do a degree of parallelism under the hood (so concurrency, in practise). The sync/async barrier elimination, however, doesn't necessarily nudge programmers towards using parallelization (spinning off new threads) within the context of a program (thread). That style might conflict, or it might be synergistic with the goal of nudging programmers towards making more use of parallelization.
- **Syntax enabled parallelization.** Inspired by [Verilog](#) and [Chapel](#). Ideally, the language runtime should be able to use parallelization to handle multiple independent processes (like client/server requests; goroutines for concurrency), but also *automatically distribute a single program across multiple CPU cores (facilitated implicitly by the language constructs/structure, without special imperative directives like thread/go)* when those cores are idle. To do that, the language should [not attempt to automatically make specifically sequential code parallel](#), since such [automatic parallelization](#) requires complex program analysis based on parameters not available at compile-time. Instead, the language should nudge towards constructs that afford natural use of multi-threading instead of single-threading (cf. principle that a language should afford *scalable modes of thinking*). But without compromising readability/reasonability, which is the top priority. The programmer should be concerned with, and simply **describe independent sets of causal/logical connections**, and the language runtime should automatically take care of as much parallelization as possible/needed. Inspired by [Haskell](#).
- [Safe parallelization](#). Inspired by [Haskell](#).
- Compiled, but also interpreted and/or incrementally compiled (for dev mode). Inspired by [Dart](#)..
- Interpreted / incrementally compiled: So developer can write quick scripts and get fast feedback.

Sacrifices runtime speed for compile-speed. Except it also needs quick startup/load time.

- **Compiled:** For production. Sacrifices compile-speed for runtime speed. Compiles to a binary. Inspired by [Deno](#).
- **Small core language:** Compiled down to a small instructions set, which can be used/targeted as a starting point to generate code for other programming languages (i.e. generate JS).
- **Portability:** Be able to target and run on multiple computer architectures.
- **Easy to build from source.** Inspired by Zig and Golang.
- **Mutable API, and opportunistic in-place mutation,** but data structures are automatically made **immutable when shared**. Inspired by [Roc](#). Immutable/[persistent data structures](#) (like [Lean-HAMT](#)) and structural sharing, to allow incremental update, while also avoiding duplication of data. Inspired by Clojure.
- **In-place mutation,** where data structures only become immutable when they're shared (presumes keeping track of borrowing / reference counting). Inspired by Rust, [Roc](#), and Clojure's [transients](#). Immutability gives the benefit of facilitating concurrency and avoid race-conditions. As a bonus you could get time-series and thus time-travel for data.
- **Mutable API:** The desirability of a mutable API (mutating objects instead of always having to pass in functions) is inspired by the JS libraries [Immer](#) and [Valtio](#). But for algorithms, instead of using the mutable API in an imperative style, it should allow keeping to a functional style, possibly with something akin to Clojure's [transients](#). Alternatively: A mutable context (block scope) could be mandated for mutations (similar to [Immer](#)), which could also afford resource cleanup (if we want to avoid having a GC). Inspired by Rust.
- **Deep immutability:** Cloning/Copying a data structure should not simply copy references below the first level, or if the data structure contains certain data structures. Because it is unintuitive/unexpected: a copy should be a full copy (at least as far as the programmer is concerned; it can use structural sharing under-the-hood). Counter-inspired by JS/TS. Alternatively: Instead of using Immutability to defeat Shared Mutable State, restrict the split-brain duplicity of keeping onto a reference to some data while also sharing a reference to it, like [Rust](#) does: by simply disallowing local references to data after it has been shared (aka. "moving" data).
- **Very constrained.** Since "*constraints liberate, liberties constrain*", as [Bjarnason said](#). Inspired by Golang's minimalism, and Elm's guardrails. For learnability and maintainability. Since [discipline doesn't scale](#) ([obligatory xkcd](#): with too much power, and the wrong nudges, all it takes is a moment of laziness/crunch-time to corrupt a strong foundation), and a complex language affords [nerd-sniping](#) kinds of [puzzles](#), and [bikeshedding and idiomatic analysis-paralysis](#). Counter-inspired by Haskell. The virtue of functional programming is that it [subtracts features that are too-powerful/footguns](#) (compared to OOP), namely: mutation & side-effects. The language designers should take care of and standardize all the idiomacy (natural modes of expression in the language). "*Inside every big ugly language there is a small beautiful language trying to come out.*" – [sinelaw](#). The language should assume the developer is an *unexperienced, lazy, (immediately) forgetful, and habitual* creature. As long as software development is done by mere humans. This assumption sets the bar (the worst case), and is a good principle for DX, as well as UX. The constrained nature of the language should allow for quick learning and proficiency. Complexity should lie in the system and domain, not the language. When the language restricts what *can* be done, it's easier to understand what *was* done (a smaller space of possibilities reduces ambiguity and increases predictability, which gives speed for everyone, at a small initial learning cost). The language should avoid [Pit of Despair programming](#), and leave the programmer in the Pit of Success: where its rules encourage you to write correct code in the first place. Inspired by [Eric Lippert](#), but also by Rust.

- Names: No *alias names* for keywords in the language or for functions in the standard library (except for as documentation reference to other languages). Inspired by [Python](#) (“explicit over implicit”, “one way over multiple ways”). Counter-inspired by [Perl](#) (postmodern plurality) and *aliasing* in the [Ramda](#) library. *All things tend toward disorder, as programmers it is our job to Fight Entropy*. The language should favor one consistent vocabulary, since it increases predictability and reduces variability. Even at the cost of expressiveness (the language should afford just enough expressiveness for the domain, see: configurable language, community grown). Names should not mimic any other programming language per se, but attempt to cater to complete beginners, because [notation has a large impact on novices](#), a principle inspired by [Quorum](#). There should be a VS Code plugin that allows people coming from various languages to type function names as they know them and the editor will translate on the fly. E.g. typing in `array.filter` gets turned into `array.keep` in the code.
- Guardrails: “_<insert your favorite programming paradigm here> works extremely well if used correctly. _ as [Willy Schott said](#). The ideal programming language should both **work extremely well even when used incorrectly** (which all powerful tools will be), but first and foremost **be extremely hard to use incorrectly**. Inspired by Rust and Elm.
- Not overly terse. Counter-inspired by C. Maybe give compiler warnings if the programmer writes names with less than about 4 characters. **Reading >>> writing**, since time spent reading is well over 10x time spent writing (inspired by [Robert C. Martin](#)), and writing can be alleviated with auto-complete, [text macro expansions](#), and [snippets](#), in the IDE.
- No runtime reflection. Counter-inspired by meta-programming and runtime type inspection in Ruby.
- Not overly verbose. Counter-inspired by XML and Java. Maybe compiler warnings if the programmer writes names with more than about 20 characters.
- [The Rule of Least Power] (<https://www.w3.org/2001/tag/doc/leastPower.html>) (by WC3), suggest a language should be the least powerful language still suited for its purpose. To minimise its complexity and surface-area. For better reuse, but more importantly: to make programs, data, and (I will include) *data flows*, easier to analyse and predict. Inspired by [FSM](#) & [XState](#). It needs, however, to be just powerful enough to be generally useful (and not limited to a DSL). Possibly Turing-complete. Given these considerations, a Lisp-style language comes to mind. But there’s reasons Lisp never became hugely popular. My guess: readability. So while it could be a Lisp-language (or compile to one), it should read better than one.
- *Removing variability in the syntax makes it more targetable for tooling and static analysis*. This further benefits the ecosystem.
- It should be small, but extensible by using simple primitives (see: community grown, configurable language). Pragmatically, it should use LLVM to compile to binary. Inspired by [Roc](#). The language should probably be built using [OCaml](#), [Rust](#), [Racket](#) or *maybe* Haskell (LLVM has bindings to these languages). Should do more with less. Inspired by Lisp. Since predictability is good for humans reading, and for machines interpreting, and if it’s predictable to machines, humans also benefit. **Important:** *“As one adds features to a language, it ramps up the complexity of the interpreter. The complexity of an analyzer rises in tandem.”* - [Matt Might, on static analysis](#)
- **Code-Formatter**, like [gofmt](#), inspired by Golang. A tool to auto-format code into a standard. Since standardisation creates readability and faster onboarding of new developers. It also enables mechanical source transformation, which is crucial for language evolvability.
- Self-hosting: In the future, the language should *maybe* be made [self-hosting](#), meaning it’s compiler would be built in its own language. For portability and independence. But it’s more important that the language is built initially (using LLVM) to facilitate the targeted usecases (webapp + systems dev.), rather than being [implicitly optimized for writing a compiler](#). Also: building a compiler in the language could potentially mean dealing with so many low-level concerns that the restricted and

high-level nature of the language will be compromised. But then again, the language should ideally be suitable for systems development...

- **Containability and explicitness.** Inspired by **pure functions**. Perhaps the language should even restrict a function's scope only to what's sent in through its parameters. So no one can reference **hidden inputs** (i.e. side-causes). Thus enforcing more predictable functions, where it is always apparent where it is used: what the function takes in and what it returns. So to achieve partial application of functions (i.e. useful closures), without addressing the outer scope implicitly, could be to supply constants from the outer scope as default/preset/front-loaded parameters. Since "explicit is better than implicit" (inspired by **Python's principles**). That way, they would be declared in the function signature, so you don't have to dive into the function to discover/investigate them. With the added benefit that the function could be customized by the caller through overriding the defaults.
- **Memoization**, automatically, but measured and only applied dynamically when runtime finds it beneficial. Aided by pure functions. The programmer shouldn't have to think about **memoization** when programming, but should be able to tune the degree of memoization (since it is a space/time tradeoff) through general configuration, for advanced cases not optimal from the default. Run time optimisations such as these are not critical features, but certainly nice to have, and should be considered in the language design in those cases where it can affect the implementation of the language runtime. Memoization of math might not always be worth it (**06:43 @ Andrew Kelley on Data-oriented design**) so the adaptive runtime should measure math calculations and decide whether or not to memoize them in main memory (RAM), or just recompute the calculations because the CPU and it's cache are so fast and accessing RAM would be slower. These are hardware concerns that are subject to change as hardware progresses, and such concerns should thus not be encoded in the language syntax, but transparently taken care of by the language's runtime environment. See: adaptive runtime.
- **Explicit imports**, so tree-shaking (to remove unused code) can be done. Inspired by JS. Also, so that it is clear where imported functions come from. Counter-inspired by **Golang**.
- **Pattern matching**. Inspired by **Elixir**, **Rust** and **ReScript**. The expression-oriented nature of the language should make this natural, without extra/fancy syntax. Pattern matching could preferably **replace if/else** conditional logic, perform totality checking to ensure you've covered every possible condition, and even enable conditional branching based on type.
- **Not file boundary dependent**. Can be split into files, but execution shouldn't be dependent on file boundaries. So the programmer is free to keep code tightly together. Inspired by SolidJS.
- **Niceties**. Inspired by **Bagel**.
- "Only single-quotes can be used for strings, and all strings are template strings and can be multiline ('Hello, \${firstName}!')"
- No triple-equals; double-equals will work the way triple-equals does in JS.
- Range operator for creating iterators of numbers (5..10, 0..arr.length, etc).
- No magic / hidden control. Control-flow should be easy to trace, because it makes it easy to understand and debug. Less magic. Counter-inspired by Ruby on Rails. Inspired by Elixir **Phoenix routing / endpoint plugs**. Testing isolated parts is made possible by explicitness. Explicit is better than implicit. Inspired by **Python's principle**. Explicitness makes testing isolated parts of the system possible. So Explicit > Implicit, almost always. (Although implicitness is preferred when one may intuitively and robustly determine the convention through the context. E.g. Needing **self**. references to access class variables inside the class methods would just be **noise**. This is counter-inspired by Python, and inspired by Ruby. However, using **self** and **this** are considered **an anti-pattern** in general.)
- **Make Inversion of Control (IoC) hard/impossible (?)**. Should ideally always return control to the programmer, not take it away. To enable the programmer to always follow the control-flow by simply

reading and following references. Thus, `yield` should also be avoided (counter-inspired by Ruby). Problem: Could make domain code dependent on integrations which goes against [the dependency inversion rule](#). So other patterns, like containing integration coupling in an intermediary abstraction ('port/adaptor' function or library, or 'channels'), would need to be developed.

- Libraries over frameworks, as a [strongly recommended](#) community convention (because frameworks cannot be prevented by a language, afaik). Frameworks utilise inversion of control. That creates Stack Traces which are really hard to debug, because they reference the framework and not your own code, [esp. problematic with concurrency](#). And when yielding control to various (micro-) frameworks, compatibility becomes a specific issue. The programmer shouldn't ever have to ask: "Is this library/framework compatible with this other one?". Counter-inspired by [JS Fatigue](#). Or have to ask "Where is the execution path of this program?". Counter-inspired by the magic of Ruby on Rails. When the control is always returned to the programmer (no IoC), he/she may likely mix and match more as pleased, without up-front worrying about compatibility (leading to analysis paralysis).
- Meta-programming: No first-class macros (runtime), since it is a too powerful footgun. But should have compile-time macros. Inspired by Clojure. So that the language can be extended by the community, and so that legacy code could be updated to latest language version by processing the code with macros to transform the syntax.
- **Expressions over statements.** The calling code should always get something back ([Is. 55:11](#)). Because the returned object can be further chained. Inspired by Clojure [and Haskell](#). Counter-inspired by JavaScript. [Statements suck](#), as even the inventor of JavaScript, [Brendan Eich admits](#). A goal should be to eliminate the subjective/anthropocentric bias that afflicts programming (especially the Imperative kind), because: *It is not you, the programmer, which is, or should be, calling code, but code should be calling code (and not terminating in the void, like as if it's you the programmer who is acting on the machine).*
- Abstractions which are powerful, made from simple primitives. *Maybe* homoiconicity... since it would make writing the compiler easier, and making the language more readily available to evolve in the community on its own (permissionless). Inspired by Lisp and Clojure's Rich Hickey.
- But this would allow meta-programming, and the associated complexity..?
- The language should maybe also not be so powerful that programs become entirely composed by very high-level domain-specific abstractions, since it encourages esotericity and sociolects, but most importantly: code indirection when reading/browsing. Coding should not feel like designing an AST, so should try to encourage keeping the code flattened (by piping perhaps?) and as down-to-earth as possible. Could maybe be alleviated by an IDE plugin which would allow temporary automatic code inlining (editable previews).
- **Reversible debugging / time-travel debugging (TTD).** "Reverse debugging is the ability of a debugger to stop after a failure in a program has been observed and go back into the history of the execution to uncover the reason for the failure." [Jakob Engblom](#). Inspired by [Elm](#). Re: Accounting for human limitations and affording the most natural way of thinking: *"The problem you are trying to fix is at the end of a trail of breadcrumbs in the program's execution history. You know the endpoint but you need to find where the beginning is, so working backward is the logical approach."* [source](#). Should at least have this. Could be enabled by, but not necessarily need:
- Reversible / invertible control flow: "A reversible programming language produces code that can be stopped at any point, reversed to any point and executed again. Every state change can be undone." [source](#). Maybe. Might not be feasible, or desirable, when it comes down to it. Might be aided by immutability, and persistent data structures (if they are extended with history-traversal / operation logging features, in addition to structural sharing).

- **Transpiler, configurable**, so it could translate between all language dialects and variations. So that the language could evolve in multiple directions, and consolidate later, without harm.
- Homoiconicity could perhaps enable this.
- Eager evaluation, by default. Since it is more straightforward to reason about in most cases, simpler to analyze/monitor, and spreads memory consumption out more in time, than [lazy evaluation](#) which would pile up work and in worst case could [overflow memory at an unexpected time](#) (in any case, the programmer shouldn't have to worry about evaluation strategies, including [space usage performance and evaluation stack usage](#)). But it should use the more efficient lazy evaluation when currying functions, or chaining methods, unless intermediate error-handling or similar requires value realization (and even here, transducers could potentially alleviate unnecessary value realization). Inspired by Lazy.js. But this is an optimisation that could wait. Concurrent operations across threads/processes should [not be lazy](#). You'd want to start exercising that machine as soon as possible. Counter-inspired by Haskell. Although it must be said: I am eager to be convinced that [lazy is better](#) and that [space leakage](#) and the [bookkeeping overhead](#) can be minimized). But in general, **the programmer shouldn't need to worry about *when the machine executes some piece of code***. Why wouldn't it be possible for a compiler to figure out at compile-time how and where functions are referenced, and choose eager or lazy evaluation depending on which is more suitable? For sequentially [chaining of operations on data structures](#), it could be lazy, and for [other operations](#) (potentially further apart in the program, with potentially memory intensive operations in between..) it could choose to be eager (get the work done, so the memory can be free'd asap). Or?
- **Async: blocking/sync interface, but non-blocking I/O**. Inspired by Golang, and to lesser extent JS / Node.js too. Should not have to litter code with async/await repeatedly (see: [what color is your function?](#) and [the problem with function annotations](#), and [async everything](#)). Could be solved with [Async Transparency](#), inspired by Hyperscript. But hiding the async nature with synchronous seeming abstractions could create a dangerous model-code gap with a potential impedance-mismatch and cause for design errors and bugs ([inspired by Simon Brown](#))... So the language should make some abstractions around async simple (like goroutines in Golang). But also inspired by declarative and easily statically analysable async contexts, made with JSX, like Suspense (async if-statement), in React and SolidJS.
- Alternatively: Async everything? The feature [referential transparency](#), obtained if the language enforces Pure Functions (i.e. no side-effects), could potentially [open up an avenue](#) for making everything async by default (and letting the compiler insert `await` instructions where it figures out functions are not I/O bound and thus can be optimized direct/synchronous CPU execution instead, without the overhead of asynchronicity).
- **Ease of reasonability is first priority**, and I believe it is best afforded by **simple and clear abstractions** (without model/code impedance mismatch, as made important by failures of ORM's and distributed contexts). The choice of sync interface here as opposed to async, is similar to how the wish for [lazy evaluation](#) by default was discarded for eager evaluation by default. One [argument by Ryan Dahl of Node.js](#) is that sync by default with explicit async (he mentions goroutines in Go) is a nicer programming model than async everything (like in Node). Because it's easier to think through what the program is doing in one sequential control flow, than jumping into other function calls like in Node.js (if you are using async callbacks). See the "fragments your logic" point below. Reasonability is a top priority, so we cannot make a compromise here.
- **Async Channels** do not block until the sender and receiver is ready, but simply put messages onto the queue stack of the receiver (see "Machines" concept under the "Scalable" feature). Counter-inspired by Golang. So that the sender can continue working without waiting for the receiver (freeing CPU time at the expense of memory).
 - [Rich Hickey also has some good arguments against async by default](#) (when implemented with

- callbacks as in JS), namely that it:
 - fragments your logic (spread out into handlers), instead of keeping it together. Programmer has to deal with multiple contexts at once (complicated), instead of one overarching context (simple).
 - callback handlers perform some action once in the future, but the state they are operating on may have mutated in the meanwhile. So it may give a false confidence in being able to get back to the state as it were when the callback was made. Want to avoid the dreaded [Shared Mutable State](#). May be solved with only allowing immutable constructs.
 - On the other hand, having sync by default, and *async through Channels*:
 - gives the control back immediately (in line with functional composition) instead of functions that effectively evoke side-effects on the real world on the other end (as callback handlers do). In line with our principle: Always give control back to the programmer.
 - channels are generalized pieces of code that can handle many connections (pub/sub).
 - channels afford safe concurrency (thread handling), whilst with callback handlers (unless used in an event-loop system such as JS) the programmer has to ensure safe concurrency (which we [don't want](#)).
 - channels afford choice on when to handle an event, whereas with a callback it gets called whenever it gets called (event-loop). Channels work in line with our principle: Always give control back to the programmer.
- All of the above have implications for reasonability. Needs to be investigated further... Golang's way of handling async seems to be [the current gold standard](#), touted by many bright people, since *"Golang has eliminated the distinction between synchronous and asynchronous code"* (by letting the programmer code everything in a sync fashion, but doing async I/O under the hood). Golang's principle of *"Don't communicate by sharing memory; share memory by communicating."* avoids the dreaded [Shared Mutable State](#) and affords itself better to ensure **simple, safe, and scalable modes of thinking** (our core principle): It's hard to think of something, if it has changed the next time you think about it (thus: immutability). Or if thinking about it changes it (manifesting in code the cognitive equivalent of [Heisenbug's](#)): Programmers need to be able to reason about a program's state without simultaneously modifying that state ([inspired by CQRS](#)).
- Another, but more radical idea: **The programmer shouldn't have to think about *when* or *where* the code will run**. It should be managed by the language runtime, based on the specified platform. If the program is a local program for one machine then it could be specified to run the work *synchronously*. If run over multiple machines, it could be specified to use async by default, to delegate work. But then if results don't arrive within time (from a remote machine/CPU-core), it could chose to perform the work itself, but lazily when the result is needed. So there should be some built-in semi-lazy evaluation measure based on CPU monitoring. Also, for the work it decides to do itself, the runtime should decide when to perform it: if the CPU-cores are idle, then it should eagerly execute the work, but if not then it should postpone just enough work so that the CPUs are adequately exercised. Currently, in languages without this nuanced model, the programmer has to make an either-or distinction based on a generalized heuristic of whether or not async or lazy makes sense, and apply it in a fixed fashion. But these assumptions do not necessarily hold for operational scenarios. Ideally, the programmer shouldn't have to think about such operational, low-level matters.
- **Concurrency. For Multi-Core and Distributed.** Probably a CSP model, or a similar or novel model, due to easier to [debug concurrency](#). Inspired by Golang.
- Async: Concurrency should integrate well with the async feature of the language. Default should be to easily ship tasks off to be completed elsewhere (other thread/process/worker/server). Inspired by Golang and JS.
- Probably not implemented as an Actor Model. Since [Actors statefulness is complex](#). Also, since events going all over the place in a non-stateful app, is harder to reason about than stricter promise-based operations (using callbacks under-the-hood). Counter-inspired by [StimulusJS](#). Inspired by ReactJS.

- **Concurrency vs. Parallelism** should be up to the runtime, not something the programmer should have to worry about. If the runtime has multiple cores, then parallelize the tasks onto those cores. If the runtime only has one core to work with, then interleave the execution of the tasks concurrently on that single core.
- **Scalable:** From single core to multiple core CPUs, and from one to a distributed set of machines. Without needing refactors. Inspired by Alan Kay’s vision of computing, and the purpose of the [Actor Model](#), utilized in Erlang/Elixir and [Pony’s actors \(w/ async functions\)](#). But rather than state-driven Actors, I’d rather want it implemented with “Machines”, which simply gives stateless functions a call queue each. Inspired by [Smalltalk](#), but stateless. They call each other by sending Messages (containing the parameters) to the other function’s call queue, take into consideration error-handling, and the [unreliability of distributed computing](#). (The caller chooses if their call should be sync/blocking or async/non-blocking, since sync/async aka. blocking/non-blocking, it is not a feature of the called function, but the call itself. [Counter-inspired by JS](#)) We name such functions “Machines”. Each of them are in fact a mini-computer, or a [computer-within-the-computer](#), if you will, but without inherent state (which ought to be stored and managed by a DB or in-memory DB). Such Machines should be able to be moved to distributed systems without rewriting the code. Inspired by Alan Kay and Actor Model systems (Akka), and languages where actors are first-class citizens, like in [Pony](#). A single Machine could work as a minimal microservice, or better yet, a Cloud Function, but likely you’d want multiple endpoints which expose a Machine each.
- **No global variables:** Because global variables don’t match to a distributed setting, so scaling up couldn’t be done without a rewrite. Instead, a function calls another function by sending a message. See aforementioned “Machine” concept. Messages can be passed through chains of function calls by piping and/or [...rest parameters](#).
- No [variable shadowing](#) *within* functions, but insides of a function may shadow the outside. Inspired by [C#](#). Since variable shadowing is [a reason](#) to have keywords such as `let`, then not allowing variable shadowing could afford the opportunity to not have such keywords, for minimalism. But functions [should shadow external variables](#) to the function, since it ensures the writer/reader doesn’t need to know about potential name collisions with external/global variables.
- Ideally, for performance, when code is compiled to be run on a single machine, the compiler should be able to be optimise away the Mailboxes, so that Machines can be turned into (simpler and faster) synchronously executed functions.
- Facilitate and nudge developers’s towards creating “Functional Core, Imperative Shell” architectures (inspired by [Bernhardt at 31:56 in his Boundaries talk](#)), to preserve the purity of functions as far as possible, while also **containing side-effects**:
 - **Configurable language:** [Platform config](#) that encapsulate all I/O primitives, which introduces a separation between trusting a particular platform and trusting the language. Inspired by [Roc](#). Could even make certain features of the language only available on certain platforms, i.e. Browser platform doesn’t have access to low level memory management. So that the language can be as restrictive as possible for the environment, ensuring that code is written idiomatically for the target platform / environment, since a restricted language has value because for a given platform the programmer would encounter less diversity in the language and thus have less to learn. This strikes a balance between on one hand providing sharp knives as global tools programmers can apply anywhere (i.e. potential footguns, leading to [The Pit of Despair](#) like in C++), and on the other hand avoiding being so restrictive that programmers can’t talk/write/think about what they want/need to for their given environment. The language itself should be [massively configurable](#): **It is not reasonable to assume that the language designers will have accounted for all possible usecases** (various memory management strategies etc.). So the language primitives/keywords should be able to be given different underlying effects (e.g. [stack](#)

vs. heap allocation) based on which platform or use-case is specified (without having to be explicit about every such effect in every environment). But the effects should be inconsequential for the reasonability of the code. Meaning that they should be at the bare-metal performance level, not at the level where operators are overloaded to do something different, cf. our principle that *things that are different (i.e. have different effects at the language level the programmer is operating at) should look different*. The platform config will depend on which implementation makes most sense for that platform (or use-case?) (i.e. browser webapp vs. systems development, vs. game development, potentially). **The language should be configurable by libraries**, that will define how it works, and can extend the core to platforms where the programmer needs to think about specific matters to that platform. Inspired by Clojure. **The same program specification should be able to have different runtime characteristics on different platforms**, depending on the platform configuration. This could be enabled by the programming language concerning itself with modeling causal relationships, instead of **place-oriented-programming**.

- **Encapsulated I/O**, so functions can avoid having side-effects. Inspired by Haskell. **Alternative #1: Algebraic Effects** for I/O, so that side-effects can be contained in a given context. Algebraic Effects are also a powerful general concept that **could** help with concurrency, async/await, generators, backtracking, etc. Inspired by OCaml. **Alternative #2: use an IO action of an IO type** (inaccurately named “IO Monad” at 30:44 in the **Boundaries talk**), transparently (without actually having to **deal with the concept of a Monad**). Where you effectively **construct a sequence of I/O operations to be executed later**. Inspired by **Haskells separation between pure functional code and code that produces external effects** (cf. “functional core, imperative shell” concept (at 31:56 in the **Boundaries talk**) which was inspired by Haskell). Something like this is needed because the Mailbox is stateful (it is constructive/destructive, like a queue), and I/O messaging would be a side-effect. The Machine/Mailbox is inspired by the Actor Model from Erlang. Ideally, since all I/O is wrapped, it should be able to turn on/off the execution of IO actions, based on setting some initial config. This could be useful for testing. You could even do a sample run to collect data, which you could snapshot to use as mock data for test runs. **Alternative #3: Potentially by syntactic rules: A function should either return a value, or don't return anything** (i.e. be simply a void procedure). And a procedure can never be placed within a function. NB: Could lead to **the colored functions problem**. **Alternative #4: Use Uniqueness Type**, which allows mutability and pass-by-value while also preserving the crucial **referential transparency** (since side-effects are ok in a pure language as long as variables are never used more than once). Inspired by **Clean** and **Idris**. Possibly use **Simplified Uniqueness Typing**, inspired by **Morrow**. **Alternative #5: Simply be able to turn on/off side-effects like external output operations**. If all output operations are done through an IO module in the standard library, it could afford a simple “off” switch, to be able to do testing. That would prevent side-effects from acting on the outside world during testing. The challenge is **side-causes** (aka. hidden inputs), however. The language could have the IO module require a default/fallback parameter to be set for external input operations. `IO.readFile(fileName, "Default file content fallback.")`. Which would be used during testing (the benefit being that the mocks would already be present). Another problem with side-effects have to do with ecosystem (especially interop with other languages): If you use a 3rd party package, how do you know it won't leak data to a 3rd party server during runtime? This ought to be solved by a sandboxed runtime environment (inspired by Deno), where it should automatically log any attempts at IO access not explicitly made through your own application code (using the language's IO module). Inspired and counter-inspired by **Elm**.
- **Reactive**. Inspired by **Functional Reactive Programming**, and **Elm**, and **The Reactive Manifesto**. Though the latter is geared at distributed systems, it could also be a model for local computation (rf. Actor model, and **Akka**). The programming language should make default and implicit the features of reactivity and streaming, as opposed to preloading and batch processing. (Reactive Streaming Data: Asynchronous non-blocking stream processing with backpressure.)

- No single-threaded event loop that can block the main thread. Counter-inspired by JS.
- [Transducers](#), under-the-hood, to compose and collate/reduce transformation functions (chains of map, filter etc. turn into a single function, [visualised here](#)). Chaining function calls should use language-supported transducers implicitly. Language should not require special `compose` syntax.
- Gradually typed, as types can add boilerplate, create [unnecessary friction](#), obstruct a programmer's [tinkering flow-state](#), and create noise in the code. Counter-inspired by TypeScript, and inspired by Elm and Jai. As many types as possible should be inferred. Inspired by TypeScript but even more inspired by OCaml and ReScript.
- No runtime type errors. Inspired by Elm (and Haskell). See 'Error Handling & Nullability'.
- Union Types: Types should be associative/commutative/composable/symmetric (i.e. $A|B$ should equal $B|A$), inspired by Dotty/Scala3, and the '[Maybe Not](#)' talk by Rich Hickey.
- Types should be enforced statically at program exit boundaries (so external libraries or outgoing I/O are ensured existing typings).
- [Structural subtyping](#) (inspired by TypeScript, OCaml), instead of nominally typed (counter-inspired by [Java](#) and [Haskell](#)). Since it is the closest you'll get to duck-typing within a statically typed language. But it should also have support for nominal types at the few cases where that might be beneficial (i.e. opaque types, not possible with only structural subtyping). Inspired by [ReScript](#), and counter-inspired by TypeScript.
- [Strongly typed](#) (checked at compile time), not weakly typed, since implicit type coercion (at runtime) can be unpredictable, and variables that can potentially change their type at runtime is [madness](#). Inspired by TypeScript and ReScript. Counter-inspired by JavaScript. [No runtime/ad-hoc polymorphism](#) (aka. dynamic dispatch), so function/operator overloading would not be possible (e.g. `+` can't be used both for summing ints and joining strings, so you'd have to use `++` or similar), but we'd gain the more important ability to fully infer static types for programs, without having to write type annotations, and compiling could get really really fast. Inspired by OCaml. Counter-inspired by Clojure, Java, Ruby.
- [Generics / Type parameters / Parametric polymorphism](#). Inspired by ReScript and OCaml. Counter-inspired by how C++ and Java [handles generics](#).
- Type inference, sound and fully decidable and with 100% coverage. Inspired by OCaml and [Roc](#). To not have to declare types everywhere. For increased readability and convenience (though not essential, cf. popularity of Rust). But local type inference inside the body of functions are what's most important (inspired by Scala), since functions input/output types should always be declared, for documentation purposes. But they could probably be generated after the prototyping phase / exploratory coding is done and you want to ossify the code. In that case, they should not be inline (like in TS), but next to the function definition (like in Elm).
- Pragmatic type bindings for external libraries: should allow you to write type bindings that mirror how you will use the library in your own project, instead of getting stuck at generalizing potentially complex types. Inspired by [ReScript](#).
- [Typed Holes / Meta Variables](#). Inspired by Idris. Since it "allows computing with incomplete programs", and "allow us to inspect the local typing context from where the hole was placed. This allows us to see what information is directly available when looking to fill the hole". I.e. the compiler provides hints about its attempt to infer the type of the missing value (aka. hole). As opposed to either requiring that a program is fully typed, it can afford a [Live programming environment](#) that give feedback to the programmer while editing about how it would be executed.
- **Composable**. Favour composition over inheritance. Inspired by Robert C. Martin, Martin Fowler,

and [JSX in React](#). Composability entails it should be easy to write code that is *declarative*, *isolated* and *order-independent*. See “strongly typed”.

- [Immutability enables composability](#), because it enables order-independence through managed effects.
- Memory safe, ergonomic, and fast. should be safely and implicitly handled by the language, without a runtime GC.
- [No Garbage-Collector](#) (GC), but also no garbage. Deterministic Object lifetimes, and Ownership tracking (affinity type system). Inspired by Rust and [Carp](#). Alternatively, the language could take inspiration from [concatenative programming languages](#) which doesn’t generate garbage by design, and uses the stack heavily. Garbage is a symptom of memorizing, which is tedious for the programmer, as well as the compiler, as well as the runtime. Garbage comes when you have to clean up something you memorised (allocated memory for, but somehow stopped using further on). Concatenative programming is closely related to FP through [continuation-passing style](#) (CPS) and [tail-call optimization](#) (TCO). The language/compiler should [utilize CPS](#) where possible, so as to reduce/optimize usage of the stack. It should be able to [store a continuation](#) (equivalent to persisting the stack to RAM/Disk) so that stateless programs (like a web server) could be restarted near a point of interruption/error (when the client makes the request again), to simulate statefulness. Inspired by Scheme.
- Memory-management & safety. Automatic Reference Counting. Inspired by [Roc](#). Maybe a Borrow Checker, for memory-safety. Inspired by Rust. But ideally, [Ownership and Borrowing](#) should be implicit by the programming language, so the programmer wouldn’t have to think about low-level concerns such as memory management (e.g. what goes on [the stack vs. the heap](#)) or various kinds of references. To avoid conceptual overhead of manual memory management (as with explicit borrowing semantics), the language should perhaps use or take inspiration from [Koka’s Perceus Optimized Reference Counting](#). Koka apparently allows even more [precise reference counting](#) (see sect: 2.2) than Rust.
- Platform user can config memory management strategy. Inspired by [Roc](#). It should for example enable choosing [Arena-allocation](#) strategy for HTTP request-response cycles, which would be optimal.
- Arena Allocation. Way XXX
- Secure from the start. Secure runtime. Inspired by Deno. Safety has to be a built-in design-goal from the start, it cannot be added on later. As evidenced by the justification of existence of Deno (Node was unsafe), and Rust (C++ was unsafe). Also, see: memory safe.
- **Powerful primitives over batteries-included:** Few, but powerful and composable, core primitives. Based on very few fundamental concepts to learn. Inspired by Lisp. Prefer uniformity and consistency. Counter-inspired by the only half-way interchangeable expressions and statements in JS.
- But avoid [DSL](#)’s, since Domain-Specific Languages typically become mini-languages in their own right. Such languages are akin to dialects/sociolects that hinder generalised understanding and learnability (adds knowledge debt). Counter-inspired by Lisp ([Lisp being too powerful](#)) and Ruby. Even though it might be true that “*Domain-Specific Languages are the ultimate abstractions.*”, as Paul Hudak put it in 1998, some cross-domain terms are usually helpful for onboarding programmers, since they afford familiar knobs on which to hang the other unfamiliar code. Even if you don’t understand the domain (or its plethora of abstractions), you would at least understand *something*. From where you could build your further understanding.
- Small focused core, with powerful composable primitives, that lends itself well to abstraction. Language extensible by library authors. Strong convention and encouragement for abstractions based on generalizable [JTBD](#) naming, instead of business/domain-specific DSL’s (reasoning above).
- Modular composition, configurable. Inspired by [Rollup plugins](#).

- **A language for library authors.** Inspired by the success of C++. The language should be able to **evolve by community convention, not by centralised specification**: the language itself should be extensible with libraries (would probably need to have some limited metaprogramming facility in the form of compile-time macros... good idea?). See: community grown, configurable language.
- **Fast branching and merging:** What would be important is to facilitate fast and simple language *merging*, due to all the branching that would appear (for the aforementioned reasons; library-driven). Inspired by Git ([fast branching and merging was the big idea behind much of Git's success](#)). So the community can easily find back together after a split/branch (if their ideas and goals come back in alignment, and they have converged to an agreement on the features again). See also: "forward- and backward-compatibility".
- Be [general purpose enough](#) to at least write scripts and CLIs, but also web servers/clients.
- Ergonomic to type. Prefer text over special characters like curly brackets (they are hard to tell apart from parentheses in JS). No littering of parentheses. Inspired by Ruby. Counter-inspired by JavaScript, Lisp, and JSON.
- No super-powerful tools which may hurt you or others in the long run. Counter-inspired by meta-programming in Ruby.
- Crash-safe. Can crash at any time and resume computation at exact same spot when restarted. Inspired by Erlang.
- Piping, or some form of it. But always top-to-bottom or left-to-right. Inspired by Bash, and functional programming with pipes (Elixir, BuckleScript, and [ts-belt](#)). [Data-first instead of data-last](#).
- No Exceptions. Inspired by Golang. But Recoverable and Unrecoverable errors. Inspired by Rust. (Definitely [no checked exceptions](#), as it breaks encapsulation by imposing behavior on the callee. Counter-inspired by Java).
- Result data type, for error handling and validation. Inspired by Result from Rust and F#, and [Either from Haskell] (<https://blog.thomasheartman.com/posts/haskells-maybe-and-either-types>) and Elm, although it should not be called Either, as [Either is a confusing misnomer](#).
- Error handling & Nullability: Goal is to eliminate [timid coding patterns](#) like null checks everywhere. Counter-inspired by Golang. No implicit `null` or `nil` value. Meaning no runtime null errors (typically occurring far removed from their point of inception). Inspired by Elm and Rust. Ideally without having to explicitly declare `Maybe` aka. `Option` types (inspired by Hickey's [Maybe Not](#)). Could either automatically represent nilable variables as a union between the type and `nil`, so that the compiler can do null reference checks at compile-time. Inspired by [Crystal](#). Or, *automatically but statically* infer and create/augment a function's return type to a [nullable reference type](#) indicated by a `?` after the typename, whenever there is an unhandled condition that could result in a null value. Or *automatically* create a `NullObject` (see: [NullObject pattern](#)) of the function's declared return type. Maybe even better: [let every type declare and handle their own empty state](#). If all types are [defined in terms of Monoids](#), then *null could be replaced by the identity value (of each Monoid)*, so that combinations within that type never fail, and never alter the result. NB: would make it hard to express something which was supposed to be there but which is missing, like a missing point on a graph curve, instead of plotting a definite 0. So would need careful consideration to choose this approach.
- When a function [becomes more capable](#) (by widening its allowed input, e.g. `string` to `Option<string>`, or tightening its returned result, e.g. `Option<string>` to `string`) it shouldn't [break callers](#) (which then could result in cascading refactors, cf.: [what color is your function?](#)). A way to solve this would be if the language could automatically perform casting of such arguments/results, and have a type system that could account for that. Inspired by Flow, and counter-inspired by TS.

Furthermore, the return type from functions using I/O (like `IO Monad` in Haskell), should always be augmented/inferred from static analysis.

- *Variant Types* for error-handling using return values (like `Result<Type, Error>`, inspired by Rust), instead of special syntax. Counter-inspired by Golang.
- So that [you have less avenues to explore when debugging](#) and [fewer branches to check when programming](#), so you can write [Confident Code](#) focused on the happy-path.
- No possibility of failing silently during runtime (due to syntax errors). Counter-inspired by JS.
- Compilation should be able to target some popular language & ecosystem, like **transpile to JavaScript** or **compile to WASM**, or potentially even the JVM, to get cross-platform interoperability. But not any target for any cost, if it would put unwieldy constraints on the language design. WASM seems like the best candidate.
- [Escape hatches](#): The language should have escape hatches that facilitate integration with other ecosystems, to aid in rapid adoption. This could compromise the strictness and guarantees of the language, but it should be possible for those who want to take on that risk/burden.
- Library compatibility tool. So you can input a list of your stack of libraries, and it will tell if and where they are incompatible. Counter-inspired by [JS Fatigue](#).
- Small standard library. To have some common ground of consolidation, and to provide the basic and most common utils. So usage will be fairly standard, and coming into a new codebase not feel too foreign. But [not too big standard library](#), since it would be connected to language updates, which are slower, and community competition is better for adaptability over the long run.
- The minimal standard library should be designed and decided by one leader with good insight into what users need, and a strong appreciation for consistency. To avoid endless bikeshedding. This is the only place where the language should have a benevolent leader for a limited time.
- **Community grown** / off-hands-leadership: No [BDFL](#), since it impedes evolution & diversity. *“When a language accepts bottom-up adaptations (from the users) it will handle new topics and new problems more efficiently than when it need to wait for top-down approval of such adaptations.”* (from [Will ugly languages always bury pretty ones?](#)). The language designer should more be an arbitrator in discussions the community can lean on regarding what should be the default convention. The designer(s) and stewards of the language should also be nice (so the community will be welcoming and thus flourish). Inspired by Ruby’s [“Matz is nice, so we are nice”](#). Even though none of this is strictly a language “feature”, it is nonetheless of *major* impact to a language and its development, so it deserves a mention. Furthermore, that **all people should have actual ownership of the development of the language**, is vital for contributions and growth. The language might not grow exactly where the designer intends, but a centralizing authority (like a BDFL) may [just as well](#) be [stifling growth](#) (and [causing pain](#)), as it purports to lead it. Counter-inspired by Elm and Clojure. Yes, wild growth might lead to some weeds (bad dialects/libraries), but leadership through conventions and good defaults could alleviate the potential analysis paralysis and decision fatigue. Counter-inspired by [JS fatigue](#). Ultimately, the power and impetus should reside with [The Man in The Arena](#), and that man/woman should always be able to be you. **The language standard / mainstream should upstream/incorporate changes** found to be popular in the wild (amongst all the various dialects/customisations), and at the same time ensure they are incorporated well (coherently and consistently).
- Free experimentation on branches that can be upstreamed: It is hard to predict the effect of novel language features, especially before they’ve been tried in the wild for some time. Languages evolving by centralized committee tend to evolve slowly, as for each new feature they have to come to a consensus, and predict and test its use. Whereas in extensible languages anyone in the community may modify it without permission, and test it on their own. This is much faster, and

dialects can be tried in parallel. Then they could be upstreamed back into to the mainstream language dialect. This could be a sweet-spot.

- How could the language be **very constrained** while at the same time be **community grown**? The language core should be very constrained around composition of a few core primitives (self-hosting), but it could be modified or built upon by others. So that it could evolve in multiple avenues of exploration, and gain from the competition. Where it would be up to the community to decide whether they want to use the constrained version(s) (suitable for large scale complex environments), which I prefer, or the bring-your-own syntax version(s) (suitable for small scale playful experimentation and research) which would inevitably appear. Inspirations here would be Lisp, Clojure and Racket.

What would be important is to facilitate simple language *merging*, due to all the divergence that would appear. Inspired by Git. So the community could easily find back together after a split (if their ideas and goals come back in alignment, and they have converged to an agreement on the features again).

- Single package directory: Some sort of singular reference to a library package information service. So the community can organise around one common point, instead of scattering. Inspired by NPM. But doesn't necessarily need to be centralised package download/storage, the storage/download could be decentralised. But would need to be safe. Cert signing?
- Runtime environment: Be able to run on some existing popular cross-platform runtime (like WASM, or maybe the JVM?). Inspired by Clojure. And/Or have a very minimal programming language runtime (without a GC). Inspired by Rust. But the runtime should in any case handle the scheduling of goroutines, inspired by Golang.
- **Ecosystem: Interoperable** with one or more existing programming language ecosystems. To import or reuse libraries. Without too much ceremony. So the ecosystem doesn't have to start from scratch. Counter-inspired by Elm. *Smooth interoperability with existing ecosystems and other systems, minimising glue code, is one of the largest underestimated features of a language, in terms of enabling its success.* Inspired by C. While a fully integrated system *can be very nice*, it inevitably risks being disrupted by a thousand small cuts (i.e. are made irrelevant to a project because other tools/services outperform them on one or two critical features, or needs interop). Counter-inspired by Ruby on Rails, **Lambdara**, **Elm** and **Dart**. The world is heterogenous, and no single system yet has been able to solve all relevant problems for all people. Many have tried to *own the world*, like Smalltalk, Imba, Darklang, etc., but this can be an impediment to mass adoption. A language as a small focused tool which lives well in a heterogenous environment is the way to go.
- C ABI: Compatible with the **C language Application Binary Interface (ABI)**. So code in the language is usable from other languages. Inspired by Zig. Since compiling to WASM is desirable, **WASM's C ABI** could probably be used, instead of a separate implementation towards the C ABI.
- “**WebAssembly** [WASM] describes a memory-safe, sandboxed execution environment” where **WASM's security guarantees** eliminates “dangerous features from its execution semantics, while maintaining compatibility with programs written for C/C++.” NB: But WASM's restrictions *might* conflict with runtime dynamism and the desired live REPL feature (inspired by Lisp/Clojure)...? “Since compiled code is immutable and not observable at runtime, WebAssembly programs are protected from control flow hijacking [code injection] attacks.”
- Editor integration: Should afford simple integration into editors/IDEs like VS Code. Typically via the Language Server Protocol (**LSP**). Inspired by **Rust**. Syntax highlighting, a **language server** (for autocomplete, error-checking (diagnostics), jump-to-definition etc.), via the **Language Server Protocol (LSP)**.
- Interactive: facilitates an IDE-plugin (VS Code) that shows the contents of data structures while coding. Enable REPL'ing into a live system. Inspired by Clojure. But with some security, so that a

rouge/unwitting programmer can't destroy the system / state. Counter-inspired by [Smalltalk](#). Some form of Hot Reload / Hot Upgrades at runtime, even though the language is statically typed. Perhaps by requiring that the swapped in functions must take in and return the same types as the previous version (i.e. enforced interface). Inspired by [Facebook's usage of Haskell](#). NB: Might conflict with compiling to WASM, since WASM gives a restricted environment. See section on WASM environment.

- “Comments should be separate from code, joined at IDE or joined via manual tooling. This would allow comments to span multiple lines/function and files. IDE could also alert when breaking changes are made. Pairs well with the Content-addressable code wish.” Inspired by [supermancho @ HN](#). You could also show/hide comments, and click on a particular piece of code or variable to see the comments for that. Without having to visually map references on a comment line to the actual variable, which is also prone to documentation drifting out of sync with the code it is documenting. With comments tied to content-addressable code, then when deleting/updating the code you also delete/update the comment. Renamings would be transparent and automatic, but when the code changes structurally the IDE could warn that the corresponding comment/doc needs to be updated.
- The expansion of function definitions inline, on demand. “Take the definition, splice it into the call site, rename local variables to match the caller”, as [JonChesterfield @ HN](#) said. So you don't have to jump around different files, which may get you to lose your state of programming flow. Content/code should even be editable then and there, and simply stored back to the files where they reside. Inspired by [supermancho @ HN](#) Lisp IDE's, and TailwindCSS. Content/code (and navigating it) should be free'd from file boundaries (see also: content-addressable code). Inspired by Git.
- Well-documented. Documentation on language syntax should be accessible from the editor/IDE, via. the LSP.
- [Docs should be versioned](#), so that docs for old versions never disappear, either from the web or from the IDE integration. Inspired by ReScript, and counter-inspired by Emotion (CSS-in-JS).
- No [@decorators](#). Counter-inspired by Angular and NestJS. Decorators feel like “magic” that make the runtime control-flow unobvious. I don't like macro-expansions, either. I don't want to talk/instruct the compiler. I want the compiler to understand how I write my program. Even if that limits the ways in which I can write my program. Rather than allowing a wide range of styles, and then having to decorate certain styles ad-hoc, to disambiguate them. I don't like the aesthetics of decorators either.
- Well-tested.
- Open-source, team: Developed as open-source from the start, of course. By more than 1 hero programmer (see: [bus factor](#)). Preferably 4-5 people working in tandem.
- [Big decisions via public RFC's](#) (Request For Comments).
- All the while, the language should avoid the [fate of the Vasa](#).? Which means a feature creep resistant *core language*. (I am aware the irony of this feature list, but read on...) Which should be designed and decided upon as early as possible (when the degrees of freedom in the design space is as wide as possible), with a holistic view. Boring > clever. Designed to reach a 80% [sweet spot of most important features, foregoing the most exotic and esoteric features](#), and foregoing the ability to solve edge-cases (such should be relegated to interoperability with other more specialized programming languages). Since 80% of the work and complexity would come from the last 20% ([The Pareto Principle](#)).

Goal: Reduce complexity for app developers, by abstraction and wise platform defaults.

This ties back to the principle that:

- A language determines **WHAT** you can & have to think about, but also **HOW** you have to think about it.

And the desired features that the language should be:

- Designed for fast onboarding of complete beginners (as opposed to catering to a specific language community who already have the [curse of expertise](#)).
- Very high level. Abstract as much of the details as possible. Abstraction means to “draw away” the concrete nature of things, so that their commonalities remain.
- Have a small [Meta-Language](#). For onboarding with low overhead. Counter-inspired by [BuckleScript](#).
- Simple, with a well thought out vocabulary. Inspired by Clojure (except `cons` and `conj`, which are too similar-looking).

Most languages presume the app developer will deal with a lot of relatively low level concerns, to get pretty obvious benefits we *should* be able to take for granted (e.g. concurrency and parallelism). The sentiment “I don’t know, I don’t wanna know”, as Rich Hickey put it, applies to this. However, it does not mean hostility towards learning, but a certain amount of healthy scepticism: if you have to document something to a large degree, have you really simplified it enough? “*Everything should be made as simple as possible, but not simpler.*”, accordig to [Occam, Sessions, and Einstein](#). [Too much documentation](#) (i.e. meta-language) is a code smell, since code should be self-documenting. A language’s complexity consist of it’s syntax and semantics, but also its [meta-language](#) (which should be minimised). a language should not burden the speaker/thinker with unnecessary complexity, as that cognitive effort is better spent on the task at hand: the domain is often complex enough in itself! We [shouldn’t invent problems to solve](#), even if the solutions could be beautiful.

Here is the start of a non-exhaustive list of what the application developer should have to be concerned with, i.e. what the language should afford as syntax and semantics when I’m coding (which does not exclude how the language libraries/runtime implements it under-the-hood):

What I want to think about:	I don't want to know or think about:
Splitting up the prob- lem/data into separate pieces .	Concurrency vs. Paral- lelism, gorou- tines, threads, Fibers, Actors, Chan- nels, pro- cesses, CPU cores, Microser- vices, Dis- tributed system topology, Mutexes, Locking, ...
Function composition.	Monads, Monoids, Class struc- ture, Contex- tual precodi- tions, Inheri- tance rules, Type / Category Theory, ...

What I want to think about:	I don't want to know or think about:
Choosing appropri- ate algo- rithms and data structures.	Immutability, Mutable vs. im- mutable refer- ences, Pointers, State manage- ment, Data- flow architec- ture, Complex types, type infer- ence/conversion, ...
Expressing what should go together, co- location of code.	Pointers, Memory manage- ment, Stack vs. Heap alloca- tion, Funda- mental security mea- sures, Sadbox- ing, Scopes, Closures, ...

What I want to think about:	I don't want to know or think about:
How to organize code to commu- nicate the ideas better to the reader, how to conform to conventions.	Performance, ...
What should be done, and to a limited extent also How it should be done	When it should be done (sync vs. async, eager vs. lazy). Where it should be done (runtime environment- concerns, but also if the work should be done on another CPU- core or another machine) ...

What I want to think about:	I don't want to know or think about:
The User Experi- ence (UX) and the actual problem domain!	Anything that detracts my thoughts from the UX and the actual problem domain. (every- thing above in this column)
...	...

You should simply be able to describe to the machine what the problem looks like, how it could be divided up, and how (the algorithm) to solve it. Then the machine (i.e. language runtime) should decide out *when* and *where* it wants to solve it (based on it's hardware/environmental constraints): whether it means to single- or multi-thread parts of the work, or, in case the local resources are/become strained, if it should distribute the work over multiple machines (depending on the measured latency of their inter-network connection). So the language should have an **adaptive runtime**, but in lieu of that, it should at least have a platform-configurable compiler, that could make some generally and universally applied decisions based on configuration of specific platform constraints.

I want a *capable* language runtime/platform, so I can write *lean* programs. Lean, as in: not loaded with what *ought to be* low-level concerns.

The End

One or more of these requirements might be conflicting / mutually exclusive. Maybe. But maybe not?

One can always dream.

This is a list of my preferences. Some would probably be quite controversial. Like my aversion to certain features which a lot of other people like (e.g. meta-programming). I might just not be familiar enough with them to have developed an appreciation for them.

I will try to keep this list updated if and when I change my mind on any point, which I am open to doing. I have already changed my mind from negative to positive on generics and pattern-matching.

What features (or lacking features) would your dream programming language have?