Robust, Intuitive Programming Language (`ripl`)

Motivation, Design, and Implementation

Ian McCall

University of British Columbia
CPEN 499B

Supervisor: Dr. Sathish Gopalakrishnan

August 22, 2018

Abstract

`ripl` is a programming language that is intended to combine the safety and purity of a language like Haskell, with the efficient, low-level capabilities of a language like C++ or Rust, and the type-level and compile-time programming capabilities of a language like Idris or D. The `ripl` compiler is written in Scala with an LLVM backend, and is still in development. This document provides an overview of `ripl`'s motivation, design and implementation, in addition to a quantitative comparison with 19 other languages over 42 language features. The results of this comparison suggest that `ripl`'s feature set is highly unique, and I believe it will offer a robust, intuitive, powerful, and performant middle-ground between high-level purely-functional languages and lower-level imperative languages, with some interesting new features as well.

Symbols in Section Names

+ Feature included in `ripl`
- Feature not included in `ripl`
? Feature may be included in future

Contents

List of Tables

List of Figures

Listings

1 Introduction

Although I have heard it said that programming languages are "just tools", that any language can be "learned in a week", that the choice of language "doesn't matter" and that the differences between programming languages are superficial or primarily syntactic[1], I think that the differences between languages are substantial and important. To quote Edsger Dijkstra, "the tools we are trying to use and the language or notation we are using to express or record our thoughts, are the major factors determining what we can think or express at all"[2].

Certain kinds of bugs, problems, and anti-patterns (such as null pointer exceptions, memory leaks, hidden side-effects, and shared or global mutable state), issues that can slow development, block teams, produce unpredictable programs, negatively impact users, and cost thousands of dollars, effect only some languages. Just as importantly, and as mentioned by Dijkstra, the ability to express certain thoughts and ideas is contingent on the features of the language in use.

One language that prevents many of the problems mentioned above while providing many expressive constructs, is Haskell, a language which (to quote Dijkstra again) "though not perfect, is of a quality that is several orders of magnitude higher than Java, which is a mess"[3]. Before making any criticisms of Haskell, it is worth mentioning that I think highly of it, that it has taught me to think of many programming patterns in a more abstract and general way (e.g. with type-classes and discriminated unions), and that the highest quality open-source libraries that I have read or used have been written in Haskell.

Unfortunately, by confining side-effects and mutation to specific monads (namely IO and ST), Haskell solves the problems associated with side effects and mutation at the cost of prohibiting straightforward and performant imperative programming, and the combined abstractions of lazy evaluation, implicit indirection, and monads make it much harder (in my experience) to optimize or reason about computationally intensive, highly stateful, and highly interactive programs in Haskell than it is in other languages (especially those that are designed for this purpose, like C++ and Rust).

What's more, although Haskell's performance is often promoted as one of its strengths[4], a comparative study of programming languages (namely C, C#, F#, Go, Haskell, Java, and Python) by Sebastian Nanz and Carlo A. Furia of ETH Zurich[5], which analyzed 7087 solutions to 745 programming problems on Rosetta Code, found Haskell to be the second slowest language considered at computationally intensive tasks (faster only than Ruby), and found it to be the second most memory intensive (using less memory only than F#).

This is not to say that Haskell's performance is not suitable for many purposes, because it is, or that monads are not useful, because they are, but rather that there are other ways to impose constraints on the mutability of data (as seen in C++, D, and Rust) and purity of functions (as seen in D), that do not require the additional levels of complexity and abstraction of monads, lazy evaluation, and implicit indirection.

Consequently, `ripl` exists to combine the robust constraints of a language like Haskell, with the straightforward imperative capabilities of a language like C++ or Rust, and the type-level and compile-time programming capabilities of a language like Idris or D. If `ripl` can achieve these goals, it will be something like a more functional D, a more high-level Rust, or a more imperative Idris, and I believe a language of this kind would be well worth using.

[1] The people I've heard say these things are C++, Java, and Python programmers, so the languages they use may be fairly homogeneous.

[2] Edsger Dijkstra, EWD 340: The Humble Programmer, `https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html`

[3] Edsger Dijkstra, On Haskell, `https://www.cs.utexas.edu/users/EWD/transcriptions/OtherDocs/Haskell.html`

[4] There have been some benchmarks that suggest Haskell is *faster* than C, but these are largely suspect or have been debunked: `https://medium.com/@n0mad/when-competing-with-c-fudge-the-benchmark-16d3a91b437c` `https://news.ycombinator.com/item?id=5080210`

[5] Sebastian Nanz, Carlo A. Furia, A Comparative Study of Programming Languages in Rosetta Code, `https://arxiv.org/pdf/1409.0252.pdf`

2 Notes on Language Specification

Although this document describes many of `ripl`'s features, it is not a language specification. Instead the compiler and accompanying unit tests serve as the language specification. This is done for a number of reasons:

1. Programming languages are more precise than human languages.
2. A compiler may be tested to validate it, whereas an English language specification may not.
3. An English language specification would add significant documentation overhead, and this time is better spent implementing the language.

A number of the features described in this document and that appear in code listings have not yet been implemented, and thus have not yet been specified.

3 Project Status

The language and compiler are still in development and are not yet in a usable state. A detailed overview of the implementation status can be found in § 9.7. The source of this document and the compiler can be found at `https://github.com/SongWithoutWords/ripl`.

4 Features at a Glance

- Strong, static typing
- Type inference
- Strict evaluation
- Type-safe discriminated unions
- Null safety
- Pattern matching
- Mutable data structures
- Type-level constraints on the mutability of data and purity of functions
- Name overloading
- Subtyping via built-in and user-defined implicit conversions
- Parametric polymorphism and type-level programming via templates
- Compile time function evaluation
- Expression orientation
- Readable, uniform syntax that is suitable for metaprogramming, inspired by Readable Lisp S-expressions[6]

5 Influence of Other Languages

Below is a list of programming languages ordered by the number of mentions in this document. The order provides a rough proxy for the influence of other languages on `ripl`'s design, and is consistent with my expectations. This influence does not necessarily imply similarity, because `ripl`'s design has been influenced both by example and counter-example. `ripl`'s similarity to other languages is covered in § 10.7.

[6]David A. Wheeler, Alan Manuel K. Gloria, Egil Möller, Readable Lisp S-expressions, `https://sourceforge.net/p/readable/wiki/Home/`

Table 1: Language Mentions as a Proxy for Influence on `ripl`'s Design

Language	Mentions in this Document
Haskell	75
Lisp, Racket, and Scheme	53
D	48
C++	45
Rust	43
Idris	35
Scala	30
C	28
C#	26
ML	25
Java	24
Python	23
Go	16
JavaScript	16
Kotlin	16
Swift	13
Fortran	2
BASIC	1
COBOL	1

6 Keywords and Symbols

Below is a table of `ripl`'s keywords and symbols. The language also provides arithmetic functions overloaded for built-in numeric types (`+`, `-`, `*`, `/`, and `%`), logical functions for boolean values (`and`, `or`, and `not`), comparison functions overloaded for built in types (`<`, `<=`, `==`, `/=`, `>`, `>=`), in addition to other low-level conversions and operations. These built-in functions can be overloaded like other names as described in § 8.3.9.

Table 2: `ripl` Keywords and Symbols

Symbol	Description
`~`	Prefix type modifier used to designate mutable types. Discussed in § 8.1.8.
`^`	Prefix type modifier, used to designate reference types. Discussed in § 8.1.8.
`@`	World-state parameter, used to designate impure functions. Discussed in § 8.1.9.
`.`	Applies two expressions from right to left, e.g. `list.length` is equivalent to `(length list)`
`->`	Creates a function type, e.g. `(-> i32 i32 point)` is a function from two `i32`'s to a point.
`bool`	The type of boolean values.
`block`	Creates a block of expressions evaluated in order; the last expression is the result of the block.
`define`	Defines a constant, variable, or function within a namespace or the top level of a file.
`f32`	The type of 32-bit floating-point values.
`f64`	The type of 64-bit floating-point values.
`i32`	The type of 32-bit integral values.
`i64`	The type of 64-bit integral values.
`if`	Creates an if-expression from a boolean condition and two alternative expressions.
`let`	Defines a name as an expression within a block. Example in § 8.3.1
`match`	Pattern matches an expression (especially applicable to unions). Discussed in § 8.3.6.
`namespace`	Creates a namespace composed of definitions and other namespaces.
`string`	The type of string values.
`struct`	Creates a record type composed of members variables, each consisting of a type and a name.
`template`	Defines a type-level function introducing one or more type-level variables. Example in § 8.1.2.
`union`	Creates a union type from a number of other types. Discussed in § 8.1.2.

7 Minimal Example

This section provides a quick introduction to the language in the form of a small `ripl` program that computes the factorial of 5, followed by a brief discussion:

```
define (main) (factorial 5)

define (factorial (i32 n)) i32
  if (<= n 1)
    1
    * n (factorial (- n 1))
```

Listing 1: Factorial in `ripl`

Although small, this example demonstrates many of the language's defining characteristics:

1. `ripl`'s syntax is expression oriented in that most of its syntactic constructs produce values rather than directing control flow (like Haskell, Lisp, ML, Rust, Scala, etc., and unlike C, C++, C#, Java, JavaScript, Python, etc.).
2. `ripl`'s syntax is Lisp-like, and as such:
   a. Parentheses group expressions (expressions may also be grouped by whitespace, as described below).
   b. Names are separated by whitespace, parentheses, or one of a small number of reserved characters.
   c. Functions are applied by grouping as in Haskell, ML, and Lisp (i.e. `(f x1 ... xn)`)[7], as opposed to the traditional mathematical notation of languages with C-style syntax (i.e. `f(x1, ... xn)`).
   d. The structure of the source code reflects the structure of the abstract syntax tree.
   e. Consequently, `ripl` does not have infix notation, operator precedence, or associativity. The absence of these features (as seen in Lisp) is counter-intuitive for many (myself included), due possibly to the fact that people are accustomed to this syntax for mathematical expressions from a young age. Whether `ripl` will include infix notation in future is unclear. Two potential methods for adding infix notation to a Lisp-like syntax are discussed in § 8.3.7.
3. `ripl`'s syntax includes some extensions over traditional Lisp syntax, inspired by Readable Lisp S-expressions:
   a. Two or more expressions on a line are grouped.
   b. Lines are extended to include all subsequent expressions at the next level of indentation.
4. `ripl` does not distinguish between functions and operators, and thus names can be composed of unicode characters, with the exception of unicode control characters and a small set of reserved characters.
5. `ripl` provides a number of built in forms (e.g. `define`, `if`), functions (e.g. `*`, `-`, `<=`) and types (e.g. `i32`)
6. The entry point of a program is a function called `main`.
7. Type annotations are required for function parameters; most other types can be inferred.
8. Return type annotations are required for recursive functions.
9. Names may be referenced in source files before they are defined.

Hopefully the discussion of this example has helped to provide you with an intuition for the language, the features of which are discussed in more detail in § 8.

[7]though in Haskell and ML expressions are often grouped by the parser rather than explicitly by parentheses

8 Design Goals and Related Features

8.1 Robust

8.1.1 + Static Typing

Static typing has a wide range of applications and advantages. It can catch errors earlier in the development process and nearer to the source than the corresponding run-time errors, can improve performance by informing optimizations and reducing the number of run-time checks required, can be used to disambiguate names via overload resolution (as in C++, C#, D, Idris, Java, and Scala), can ensure that only certain functions have side effects (as in D, Idris, and Haskell), can ensure that only certain aspects of certain variables can be modified (as in C++, D, and Rust), and can be used as a basis for compile-time programming and metaprogramming (as in C++, D, Idris, and Haskell).

When combined with type inference, these advantages can be had with little increase in program length or programmer effort. Consequently, one of `ripl`'s primary goals is to embrace static typing and to extend the range of invariants that can be encoded within the type system, so that the language can be used to develop robust programs with predictable behaviour at scale.

8.1.2 + Type-Safe Discriminated Unions

Type-safe discriminated unions, or sum types, (as seen in Haskell, Idris, ML, Rust, Scala, etc.) provide a powerful and intuitive way of modelling polymorphic data and computations that may take one of a number of forms. Some examples in `ripl` are shown below:

```
;; the union keyword can be used to create type-safe discriminated unions
union expression
  struct add (^expression a) (^expression b)
  struct sub (^expression a) (^expression b)
  struct mul (^expression a) (^expression b)
  struct div (^expression a) (^expression b)
  f32

;; it can be combined with the template keyword to create a parameterized union
template (list a)
  union
    struct nil
    struct non-empty
      a head
      ^(list a) tail
```

Listing 2: Discriminated Unions in `ripl`

Unlike untagged unions that do not record the type of the union's value, and non-type-safe discriminated unions in which a type tag is manually maintained and branched on by the programmer, type-safe discriminated unions include a type tag that is automatically maintained and automatically branched on during pattern matching. An example of pattern matching in `ripl` can be seen in § 8.3.6.

Although discriminated unions are analogous in some respects to OOP style inheritance subtyping (which can even be used as a basis for discriminated unions, as in Scala), I would argue that type-safe discriminated unions when used in conjunction with pattern matching, result in code that is more robust, precise, straightforward and less tightly coupled than OOP style inheritance. Consequently, discriminated unions are an important feature of `ripl`'s design, the advantages of which are highlighted in § 8.1.3 on null-safety, for which they provide an excellent solution.

8.1.3 + Type-Level Constraints on Existence (null safety)

The ability to substitute `null`, `nil`, etc. for many or all values is a frequent source of ambiguity and error in many languages, including C, C++, C#, D, Java, JavaScript, Lisp, Python, and Scala.

The null reference was invented in 1965 by Tony Hoare, who later described it as a "billion-dollar mistake" when speaking at a software conference called QCon London in 2009[8].

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Although the unrestricted and potentially unsafe use of `null` is a significant problem, the ability to represent a value that may or may not exist remains highly important. To date I've encountered two viable mechanisms by which a language can express potentially non-existent values while maintaining null-safety:

1. **Dependent Typing:** dependent typing is a language feature in which the type of an expression may depend on its value. Kotlin employs a limited form of dependent typing to differentiate between nullable and non-nullable pointers at compile time, based on type annotations in addition to control flow[9].
2. **Type-Safe Discriminated Unions:** discriminated unions, as discussed in § 8.1.2, provide a very robust and safe way of representing polymorphic types, and is employed by Haskell, ML, and Rust, among others, to represent potentially non-existent values in a type safe way[10].

Between these options I prefer type-safe discriminated unions, because they are simpler than full-blown dependent typing (as seen in languages like Idris, which is roughly speaking a strictly evaluated and dependently typed Haskell), and because they are much more widely applicable than the limited form of dependent typing seen in Kotlin. In support of this idea, Idris, which has both discriminated unions *and* dependent typing, implements its `Maybe` type as a union[11]; `ripl` will do the same.

8.1.4 + Temporary, Local Variables

Although a number of languages have had a shaky history with temporary, local variables (including BASIC, COBOL[12] and Fortran[13]), we are fortunate that temporary, local variables are ubiquitous in modern languages. The locality of these variables reduces the scope in which their state can be accessed, and their transience reduces the state of the program that would otherwise persist between function calls. All variables in `ripl` not declared at the top level are temporary and local.

8.1.5 ? Encapsulation

Considered one of the defining features of object-oriented programming, encapsulation is another feature that helps to limit the scope of program state. Although I do not have concrete plans for encapsulation and access modifiers in `ripl`, encapsulation warrants mentioning because it demonstrates that not only functional languages are concerned with limiting the scope of mutable state but also imperative and object-oriented languages.

---

[8]Tony Hoare, Null References: The Billion Dollar Mistake, `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`

[9]Kotlin Language Reference, Null Safety, `https://kotlinlang.org/docs/reference/null-safety.html`

[10]Although Scala has type safe discriminated unions and an option type, it is not null-safe. The following expression type checks correctly and produces a null pointer exception at runtime: `Some(null) match { case Some(x) => x.toString; case _ => ""}`

[11]Idris Standard Library, Maybe, `https://github.com/idris-lang/Idris-dev/blob/master/libs/prelude/Prelude/Maybe.idr`

[12]`http://www.jeromegarfunkel.com/authored/cobol_apology.htm`

[13]`http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/5-Fortran/scoping.html`

8.1.6 + Expression Orientation

Expression orientation is a language feature that allows programmers to perform computations by composing expressions rather than directing control flow or mutating intermediary values. Expression orientation is a continuum, from assembly languages and compiler intermediary representations that are highly imperative, to imperative languages with both expressions and statements (like C++, C#, Java, etc.), to fully functional languages in which everything or nearly everything is an expression (like Haskell, Lisp, ML, Scala, etc.).

Expression orientation helps to reduce the statefullness of a program by reducing the number of variables in scope and reducing the need to mutate these variables. Everything in `ripl` that is not a top-level definition is an expression. Expression orientation is discussed from a usability perspective in § 8.3.5.

8.1.7 - Monadic Statefullness and IO

One way of constraining mutation, as seen in Haskell and Idris, is to limit mutation to occurring within monads (namely IO and ST, in both languages):

> Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers.[14]

Although I agree with the designers of these languages that it's important to separate pure and impure code, and that the way they have modelled stateful computations within a purely functional language is elegant, in practice I find that this additional monadic abstraction can make stateful code significantly harder to write (especially when combined with laziness, as in Haskell).

For example, during the semantic phase of the `ripl` compiler, every untyped expression is reduced to a value, a type, or a typed expression. Most expressions will depend on other definitions in the program, and these definitions can occur in any order. To deal with this, I reduce the abstract syntax tree lazily, and feed the result back into the `reduce` function (a process called "tying the knot") so that the type or value of each definition can be computed in terms of others. Although this works perfectly in many cases, in order to handle cyclic dependencies the computation must be stateful and track the definitions it has already visited so that it does not loop infinitely. After two weeks of trying to get this to work in Haskell with the ST monad, I tried it in Scala, got it to work in a single afternoon, and subsequently ported the entire compiler to Scala.

Although this anecdote does not demonstrate that the above problem could not be solved with laziness and monads in Haskell, or that this problem could not be solved without resorting to mutation at all, it is an example in which Haskell's approach to statefullness made a problem intractable for a user. While constraints on mutability and function purity are important, straightforward imperative and stateful programming is also valuable, and at times necessary. `ripl`'s approach to encoding these constraints while preserving the ability to perform straightforward stateful programming is discussed in § 8.1.8 and § 8.1.9.

8.1.8 + Type-Level Constraints on Mutability

A middle ground between the unconstrained or under-constrained mutability and impurity of languages like C#, Java, ML, and Scala, and the functional purity of languages like Haskell and Idris, are per-variable type-level constraints on mutability, as seen in C++, D, and Rust. This approach has the advantage of constraining what variables can be modified within what scopes, while still allowing mutation where necessary.

In C++ and D, types can be made immutable using the `const` keyword, with some differences[15]:

1. `const` in C++ can be bypassed using `const_cast` or `mutable`, which undermines its legitimacy.

[14]Haskell Website, Purely Functional, https://www.haskell.org/

[15]D Language, const(FAQ), https://dlang.org/articles/const-faq.html#cpp-const

2. `const` in D applies recursively to all types that a composite type is composed of, a quality they refer to as transitive. This has the disadvantage of reducing the range of types that can be expressed, and may force the use of entirely mutable types when only parts of these types need to be mutable. For example, a function that simulates interactions between entities might operate on an immutable list of references to mutable entities, thereby expressing its intent to modify the entities themselves, and not the container. Unfortunately, this distinction cannot be expressed with D's transitive `const`.

In Rust, types can be made mutable using the `mut` keyword. The advantage of immutability by default, is that the keyword is *required* to mutate a value; whereas in C++ and D data can be mutated or not mutated without the need to specify. Rust then uses this feature to prevents data races at compile time with the following rule: "At any given time, you can have *either* one mutable reference *or* any number of immutable references."[16] Whether `ripl` can achieve the same in future will depend on a choice between garbage collection and a Rust-style ownership system, a decision that is discussed in § 8.1.11.

The equivalent in `ripl` of Rust's `mut` keyword is the ~ symbol, which was chosen because:

1. It is not a commonly used symbol in programming.
2. It is shorter than `mut`.
3. It looks fluid, hence changing, hence mutable.

The purpose of the mutable type modifier in `ripl` is to restrict mutation to a set of variables that are explicitly mutable within the present scope. Assignment between mutable and immutable values and references are handled according to the following table:

Table 3: Assignment Between Mutable and Immutable Values and References in `ripl`

Type	Assign to `T`	Assign to `~T`	Assign to `^T`	Assign to `^~T`
`T`	value copied	value copied	value referenced	type error
`~T`	value copied	value copied	value referenced	value referenced
`^T`	value copied	value copied	reference copied	type error
`^~T`	value copied	value copied	reference copied	reference copied

These rules can be applied recursively to composite types like functions and templates. For the purpose of type-checking, this boils down to the following rule: mutable references cannot be created to immutable data.

8.1.9 + Type-Level Constraints on Purity

8.1.9.1 Discussion of Purity An impure function is one that depends on or modifies global, mutable state like global variables and singletons, or performs system-level IO like interacting with the file-system, performing textual IO, invoking other processes or drawing to the screen. Although this IO is the purpose for which we create programs, there are some disadvantages to impure, or potentially impure[17] functions, including:

1. Their behaviour may depend on global, mutable state.
2. Their inputs and dependencies may not be clear from their signature.
3. Their outputs and effects may not be clear from their signature.

Indeed, in order to *know* how such potentially impure functions may interact with the program, it is necessary to recursively read all of the functions they call, and understand how all of these functions effect and are

[16] https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html#the-rules-of-references

[17] The purity of a function in a language that does not distinguish between pure and impure functions can only be determined by recursively reading it and all of the functions it calls, which may not be feasible.

effected by the global state of the program, in addition to the feedback between them. In a suitably large and impure program, this complexity is not possible to comprehend. In a suitably large and impure program, the programmer may arrange functions to produce the desired effect in one place and break something somewhere else in the process. For these reasons, impure or potentially impure functions are harder to test, harder to debug and harder to reason about.

At its most extreme, systemic impurity entirely subverts the purpose of function signatures in documenting what functions do, and thereby undermines the structure of the program. When a language fails to distinguish (as most do) between the signature of the entry point of the program (something like `int main()`), a function that can do *anything*, and the signature of a pure function like addition (something like `int +(int, int)`), how can any function in this language be trusted?

In a purely functional program you can tell how the pieces fit together from their types, whereas in a more imperative program there may be a way to arrange and order the pieces such that they fit, but it may not be immediately obvious how. In a pure language like Haskell, we know a lot about a function with a type like `A -> B`. We know that it will use an `A` to compute a `B` without depending on or modifying the state of the program in any way[18], and consequently that:

1. It always produces the same output given the same input.
2. It does not effect the program and so can be called any number of times without consequence.
3. It behaves the same way within the context of the program as it does when tested in isolation.
4. It can be evaluated at compile time if its arguments are known at compile time.

Even if a function performs computations with mutable state internally, as long as these internal mutations do not escape to the outside world, all of the above properties still hold. In pure functional languages like Haskell and Idris, this encapsulation of effects and separation of pure and impure code is done using monads (such as IO and ST), as discussed in § 8.1.7. As mentioned in that section, I think this encapsulation of effects is positive, but I have concerns about the complexity of this approach, both for the programmer and for the machine.

8.1.9.2 Purity in D A solution to this problem in an impure language can be found in D, and is described quite well by David Nadlinger[19]. D allows functions to be annotated using a `pure` keyword, which prevents them from performing impure computations or calling other impure functions. Combined with compile-time evaluation of pure functions and templates that can take values of any type arguments, this feature provide a basis for powerful type-level programming and type-level constraints on purity in D.

8.1.9.3 Purity in `ripl` `ripl`'s method of constraining purity is similar to that of D, but differs in some respects. Rather than using a modifier keyword like D, `ripl` has a global state parameter `@`, that may be taken as a parameter to `main` and distributed to the rest of the program as an argument to other functions. In order to read global state (such as reading global variables, reading files, checking the current time, or using memory addresses in computations) functions must take `@` as a parameter. In order to modify global state (such as writing global variables, writing files, or writing to the console) functions must take `~@` as a parameter. `~@` may be substituted for `@` just as `^~T` may be substituted for `^T` as described in § 8.1.8. The advantages of this approach include:

1. It leverages the same syntax and scoping rules as function parameters, so should be intuitive.
2. It is easily and intuitively encoded in function types, e.g. `main` may have type `(-> ~@ i32)`.
3. Function purity is visible at the call site in addition to the signature (e.g. `println ~@ "Hello world!"`).

[18]Unless it circumvents the type system by some mechanism like Haskell's `unsafePerformIO`, but this is uncommon.

[19]David Nadlinger, Purity in D, http://klickverbot.at/blog/2012/05/purity-in-d/

4. It's possible to express the difference between read-only impurity `@` and read-write impurity `~@`.

Below is a table comparing pure and impure function signatures in various languages. Of the languages considered, D, Haskell and `ripl` are able to express the difference between pure and impure functions and C++ and Rust are not. The ability to express this difference is actually quite rare among languages, and the only others that I know of in which this is possible to express are purely functional languages like Idris, Clean and Frege. `ripl` is the only language I know of that uses a global state parameter, and can express the difference between read-only and read-write impurity.

Table 4: Comparison of Pure and Impure Function Signatures in Various Languages

Language	Potentially Impure	Pure with Mutable Arguments	Pure
C++	`int main()`	`void normalize(Vector& v)`	`Point operator+(Point a, Point b)`
D	`int main()`	`pure void normalize(ref Vector v)`	`pure Point add(Point a, Point b)`
Haskell	`main :: IO ()`	`normalize :: Vector -> Vector`	`(+) :: Point -> Point -> Point`
`ripl`	`(main ~@)`	`(normalize (^~Vector v))`	`(+ (Point a) (Point b)) Point`
Rust	`fn main()`	`fn normalize(v: &mut Vector)`	`fn add(a: Point, b: Point) -> Point`

A `ripl` function that does not take the global state parameter, but takes one or more mutable references is weakly pure; a `ripl` function that takes neither the global state parameter, nor any mutable reference is strongly pure[20]. In addition to aiding the creation of robust programs as described throughout this section, this type-level information on function purity will help the `ripl` compiler determine what functions can be evaluated at compile time (as described in § 8.4.2), and may useful in directing optimizations in future.

8.1.10 + Namespaces

Although they are referred to by many names (packages, modules, namespaces, and possibly others), and there are a lot of variations in their behaviour between languages, namespaces are essentially a system to restrict the visibility of names and avoid name collisions. This is important, and prevents the need to prefix every symbol name with the library that it comes from, as may be necessary in languages without this feature like C and some Lisps. `ripl`'s namespace feature is inspired by and very similar to that of C#. In addition to this system, which is pretty simple, I would like to add a feature that will allow the compiler to infer namespaces from the directory structure, to reduce the potential for inconsistency between the namespace structure and directory structure of a project.

8.1.11 ? Garbage Collection

The choice of memory management technique is a nuanced decision, as evidenced by the fact that new languages are emerging both with garbage collection (e.g. Dart and Go), without (e.g. Rust and Swift), and with optional garbage collection (e.g. D). A comparison of these approaches is shown below:

Table 5: Comparison of Memory Management Techniques

Garbage Collection	Deterministic Destructor Calls
+ Robust	− Potentially error prone
+ Handles cycles well	− Does not handle cycles well
+ Performed automatically	− May require use of smart pointers and writing of destructors
+ Better amortized efficiency	− Worse amortized efficiency
+ Better memory locality via heap compaction	− Allocations likely fragmented without manual slab allocation
− Unpredictable latency	+ Predictable latency
− Limited to memory management	+ Can be used to manage other resources via RAII

[20]This terminology is used by, and possibly introduced by, the D programming language:
https://tour.dlang.org/tour/en/gems/functional-programming

With the exception of real-time and soft real-time applications, I think that garbage collection is undoubtedly better, because it frees the programmer from needing to manage memory using destructors and smart pointers (or god forbid by using malloc and free). Furthermore, the inability of reference counting to adequately manage potentially cyclic data structures is a major disadvantage because recursive and mutually-recursive data types provide such a powerful and natural way of describing common data structures like lists, trees, and graphs, and manifestations thereof, such as abstract syntax trees, file systems, JSON, S-expressions, and XML.

Because `ripl` is intended to be suitable for game development, and because I do not know whether the improved memory locality and amortized efficiency of garbage collection can compensate for its potentially unpredictable latency in soft real time applications like games, I have not yet decided whether `ripl` should be garbage collected or not.

8.2 Performant

As a statically-typed and compiled language with mutable data-structures, and without virtual functions, lazy evaluation, or implicit indirection (e.g. boxing), `ripl` is susceptible to a similar range of optimizations as languages like C++ and Rust. By using LLVM-IR as a compile target, as does the Rust compiler rustc, and C++ compiler Clang, `ripl` can leverage many of the same optimizations. If `ripl` adopts an ownership system inspired by Rust instead of garbage collection (a decision that is discussed in § 8.1.11), then `ripl` may have similar performance characteristics to C++ and Rust (though actual performance will depend on the implementation).

8.3 Ergonomic, Intuitive, and Concise

In order for a language to be enjoyable to use (or at least unobtrusive), it needs to be ergonomic, intuitive and concise. Although some people don't seem to take syntax very seriously (by dismissing it as superficial bike-shedding, describing it as a "solved problem", or wondering why discontent users of verbose languages are "afraid of typing"), I'm inclined to agree with Simon Peyton Jones, when he wrote in a presentation about Haskell's design[21]:

> ~~Syntax is not important~~
>
> Syntax is the user interface of a language
>
> ~~Parsing is the easy bit of a compiler~~
>
> The parser is often the trickiest bit of a compiler

Although some syntax elements may be a matter of preference, there is at least one measurable aspect of syntax with a high degree of variation: verbosity. A study by Sebastian Nanz and Carlo A. Furia of ETH Zurich, of 7087 programs in Rosetta Code (referenced also in the introduction), found that[§ 5]:

> Languages are clearly divided into two groups: functional and scripting languages tend to provide the most concise code, whereas procedural and object-oriented languages are significantly more verbose. The absolute difference between the two groups is major; for instance, Java programs are on average 2.2–2.9 times longer than programs in functional and scripting languages.

While these findings are consistent with my own experiences, their magnitude exceeds my expectations (and validates my frustration with certain verbose languages). Among the largest effects in their study, they found that, of the programs in their data set, programs in C# were on average 2.7 times as long as programs in Haskell and 3.6 times as long as programs in Python.

[21]Simon Peyton Jones, Wearing the Hair Shirt: A Retrospective on Haskell, slide 9, http://www.cs.nott.ac.uk/~pszgmh/appsem-slides/peytonjones.ppt?ref=driverlayer.com/web

Although I've heard apologists of verbose languages defend their verbosity by insisting that code is read more often than it is written, code that is 2-3 times longer is longer both to read and to write. This is not to say that more concise is always better (adequately descriptive names are good), but I do not think that C++, C#, D, Java, etc. have gained any clarity by their verbosity: instead I think that this excessive verbosity and boilerplate obscures the logic of the program and any errors it may contain.

8.3.1 Brief History of `ripl`'s Syntax

As the user interface of a language (per Simon Peyton Jones§ 21), syntax warrants serious care and consideration. Nearly all aspects of `ripl`'s syntax have changed dramatically over the course of its history, as part of an ongoing process of development and improvement.

Following its inception in February 2017, `ripl`'s syntax was a Python-like BNF grammar with C-style function application and whitespace delimited blocks. It had both statements and expressions, and both if-expressions and if-statements (also like Python). As time went on, I started to think that this distinction between statements and expressions was redundant and inelegant. In a commit in January 2018, statements were removed from the grammar, and the language started to become expression oriented. By early June 2018, nearly all constructs had become expressions in the grammar, including composite types like structs, unions, and function types.

In mid-June 2018, I came across Readable Lisp S-expressions§ 6, and was very impressed by this notation, which combines the simplicity, elegance, generality, and homoiconicity (self-representing nature) of Lisp's S-expressions with the brevity and legibility of whitespace delimitation. I immediately set about changing `ripl` to use this new syntax (examples of which can be seen in this section, and in § 7 and § 8.1.2).

At the time of writing I have not yet ported all constructs from the old Python-like grammar to the new Lisp-inspired grammar. For example, I do not yet know what the syntax will be for multi-expression blocks, though it may be something like this:

```
define (power-of-8 (f32 x1))
  block
    let x2 (* x1 x1)
    let x4 (* x2 x2)
    * x4 x4
```

Listing 3: Multi-Expression Blocks in `ripl`

8.3.2 + Whitespace Delimitation

Although indentation delimited languages are somewhat uncommon, people speak highly of them (e.g. Haskell and Python). I am a proponent of this style for a number of reasons:

1. It leverages the visual structure that people *rely on*[22] to read code effectively.
2. It reduces the number of tokens and visual clutter.
3. It ends any discussion or inconsistency over whether opening braces should occur on a new line.
4. It prevents inconsistency between the visual and the semantic structure of the code, thereby reducing the potential for error.

For these reasons, expressions in `ripl` *may* be grouped by indentation. It is, however, possible to write `ripl` code that is explicitly delimited[23], because indent and dedent tokens are not emitted within S-expressions.

[22] Richard J. Miara et al, Program Indentation and Comprehensibility, https://www.cs.umd.edu/~ben/papers/Miara1983Program.pdf

[23] The only reasons I can think of for doing so would be to embed `ripl` code within some other data format, to serialize it more compactly, or to operate on it with tools that are designed to work with S-expressions.

8.3.3 + Type Inference

Type inference makes it possible to omit some or all type annotations while maintaining the benefits of static typing. Broadly speaking, there are two styles of type inference: Hindley-Milner or full type inference and bidirectional or partial type inference. Hindley-Milner style type inference has the advantage that it can infer the types of *all* expressions within the program, including function parameters.

However, the syntactical unification algorithm often used for Hindley-Milner type inference[24] is complicated by the presence of overloading. Furthermore, it breaks down in the presence of subtyping[25] because the type constraints generated from the program no longer constitute a system of type equations that can be solved via substitution, but rather a system of subtyping relationships that are non-strict type inequalities (`T1 <: T2` being analogous to `a <= b`).

Although alternative algorithms have been developed to support Hindley-Milner style type inference with subtyping[26, 27], `ripl` is proceeding with bidirectional type inference for the following reasons:

1. Because supplying type annotations for function parameters is not so burdensome, and is even considered good practice in languages like Haskell in which these types can be inferred.
2. Because bidirectional subtyping operates on the level of expressions rather than type constraints, it's relatively easy to combine with compile time evaluation (a feature of `ripl` discussed in § 8.4.2).

In summary, `ripl` trades full type inference (that could infer parameters types) for overloading, subtyping, type classes, and compile time evaluation.

8.3.4 + Subtyping via Implicit Conversion

Subtyping is a common feature among object-oriented programming languages. It is much less common among functional languages, possibly because of the complexity it adds to type inference, as discussed in § 8.3.3. Although it is less essential in a language without inheritance (like `ripl`), subtyping helps to reduce the need for explicit type conversions.

Although subtyping and implicit conversion have somewhat of a bad reputation[28], I think that implicit conversions can add value if chosen judiciously. Subtyping in `ripl` is achieved via implicit conversions. The `ripl` compiler provides a built-in conversions from integral numbers to floating point numbers, and may in future provide implicit conversions from the variants of a union to the union itself. Additionally, the compiler is structured to allow for user-defined implicit conversions. Although the syntax for declaring user-defined implicit conversions has not yet been chosen, it will probably consist of defining a pure function with a single input using a distinct keyword like `implicit` instead of the usual `define` keyword. An example of subtyping in `ripl` is given below:

---

[24] Cornell University, CS3110, Type Inference and Unification,
http://www.cs.cornell.edu/courses/cs3110/2011fa/supplemental/lec26-type-inference/type-inference.htm#3

[25] `ripl` began without subtyping or overloading and used constraint generation and the unification algorithm for type inference. When I added overloading to `ripl`, I continued to use the unification algorithm by deferring the unification of constraints with overloads until the types of the overloads and the types of the arguments were known. When I then added subtyping to the language it became clear that unification would no longer work, and I reverted to bidirectional type checking.

[26] Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow, Extending Hindley-Milner Type Inference with Coercive Structural Subtyping,
https://www21.in.tum.de/~nipkow/pubs/aplas11.pdf

[27] Stephen Dolan, Algebraic Subtyping, https://www.cl.cam.ac.uk/~sd601/thesis.pdf

[28] This is especially true in languages like C, in which widespread and questionable implicit conversions allows some types to be used almost interchangeably that should not be, like booleans, integers, and pointers.

```
define (add-int-and-float (i32 x) (f32 y))
  ;; The only viable overload is (+ f32 f32), and so x is implicitly converted
  + x y
```

Listing 4: Subtyping via Implicit Conversion in `ripl`

The rules used for the selection of overloads in the presence of implicit conversions are described in § 8.3.9.

8.3.5 + Expression Oriented Syntax

Expression orientation is discussed within the context of the Robust design goal in § 8.1.6, because it enables programming with fewer local variables thereby reducing statefullness within functions. Within the context of usability, I think that expression orientation lends itself to a more composable, ergonomic, elegant, and concise programming style. I would not be surprised if the tendency against expression orientation among imperative languages accounts for some of the verbosity of these languages that was found by Nanz and Furia[§ 5]. Everything in `ripl` that is not a top-level definition is an expression.

8.3.6 + Pattern Matching

Discriminated unions are a feature of `ripl` that is discussed in § 8.1.2. Although pattern matching may be extended in a number of ways, at minimum it provides a type-safe and ergonomic method of extracting information from discriminated unions, and works behind the scenes by branching on type tags. In addition to type safety, pattern matching has the advantage of producing code that tends to reflect the structure of the data. Below is an example of pattern matching in `ripl` used to write a simple evaluator for some floating point expressions:

```
;; the union keyword can be used to create type-safe discriminated unions
union expression
  struct add (^expression a) (^expression b)
  struct sub (^expression a) (^expression b)
  struct mul (^expression a) (^expression b)
  struct div (^expression a) (^expression b)
  f32

define (evaluate (^expression e))
  ;; the match keyword can be used to operate on unions
  match e
    (add a b) (+ (evaluate a) (evaluate b))
    (sub a b) (- (evaluate a) (evaluate b))
    (mul a b) (* (evaluate a) (evaluate b))
    (div a b) (/ (evaluate a) (evaluate b))
    (f32 val) val
```

Listing 5: Pattern Matching in `ripl`

8.3.7 ? Infix Notation and Word Order

Word order (including subject-object-verb or SOV, subject-verb-object or SVO, and verb-subject-object or VSO) is a characteristic of the grammar of both human and programming languages, some aspects of which are summarized in the table below.

Table 6: Word Order in Human and Programming Languages

	Percent of Human Population[29]	Example Languages[30]	Programming Constructs	Example Code
SOV	45%	Farsi, Hindi	Reverse Polish notation	`map key contains`
SVO	42%	English, Mandarin	Methods in OOP	`map.contains(key)`
VSO	9%	Arabic, Hebrew	Traditional functions	`contains map key`
VOS	3%	Baure, Malagasy	Traditional functions	`contains key map`

Some studies have found that people tend naturally to use subject-verb-object order when communicating with an established lexicon (even when they are accustomed to another order)[31], and that people tend naturally to use subject-object-verb order when communicating with an improvised lexicon (even when they are accustomed to another order)[32]. Combined with the relative unpopularity of verb-first word orders in human languages, the findings of these studies may suggest that verb-first word orders are less suited to human comprehension, and may even explain some of the popularity of object oriented languages (which allow subject-object-verb order by means of method syntax) and unpopularity of Lisps (which typically do not even allow infix notation).

In addition to word order, many programming languages include infix notation with precedence and associativity (as used in conventional mathematical notation), which, like the method syntax of object-oriented languages, allows some or all verbs to appear in infix position, as in SVO ordering. Although I find it quite convenient to add terms by writing `(+ a b c d)`, as opposed to `(a + b + c + d)`, or to determine if terms are ordered a certain way by writing `(< a b c d)`, as opposed to `(a < b && b < c && c < d)`, programming languages with only function application and without infix notation or method syntax (notably Lisps) are often said to be unintuitive or hard to read (although I've been unable to find any rigorous evidence of this).

Together these observations pose a number of questions:

1. Does word order have a significant influence on people's comprehension of programming languages?
   a. If so, to what degree does it depend on the word order people are accustomed to?
   b. If so, can it be overcome by continued use, or are some word orders inherently advantageous?
2. Does infix notation have a significant influence on people's comprehension of programming languages?
   a. If so, can its absence be overcome by continued use, or is infix notation inherently advantageous?

If allowing infix and subject-verb-object ordering is desirable (of which I'm still not entirely convinced) there are a number of ways in which this could be achieved in `ripl`:

1. Surround infix expressions in braces as proposed by Readable Lisp S-expressions§6.
   For example `{a + b + c}` would be equivalent to `(+ a b c)`.
2. Typecheck S-expressions with the first two expressions reversed, and use the reversed order if it results in fewer errors and implicit conversions than the non-reversed order (analogous to an implicit conversion).

Because this is a rather complicated topic, and because both of the potential solutions above could be easily added, I am inclined to leave them out for the time being, and add them in future if their absence is felt.

8.3.8 + Selection Syntax

Somewhat related to word-order (discussed in § 8.3.7) is selection syntax. The idea is to allow `a.b` as a left-associative shorthand for `(b a)`, so that it's possible to write things like `(math.bits.xor a b)` instead of `((xor (bits math)) a b)`, or `(- character.transform.position camera.transform.position)` instead of `(- (position (transform character)) (position (transform camera)))`. In the examples above, I find the expressions with selection syntax clearer and more readable.

[29]Russell S. Tomlin, Basic word order. Functional principles. London: Croom Helm, 1986. Page 308, https://www.cambridge.org/core/journals/journal-of-linguistics/article/russell-s-tomlin-basic-word-order 7542AFB4A8B28D651F6E109B810F4C04

[30]Wikipedia, Subject-verb-object, https://en.wikipedia.org/wiki/Subject%E2%80%93verb%E2%80%93object

[31]Alan Langus, Marina Nespor, Cognitive systems struggling for word order, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4534792/

[32]Hannah Maro, Alan Langus, et al, A new perspective on word order preferences: the availability of a lexicon triggers the use of SVO word order, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4534792/#B23

8.3.9 + Name Overloading

Overloading is a feature that is more noticeable when absent than present. In Haskell, for example, (which lacks traditional overloading) you cannot have two functions with the same name (including simple accessors like `name` or `size`) without creating a type class that declares the function, and then implementing the type class for all required types. Thus, name collisions in Haskell can be avoided either by using type classes (which is cumbersome), or by embedding parameter types in function names (which is also cumbersome). This absence of traditional overloading is clumsy, and has made Haskell's standard prelude inconvenient in a number of ways:

1. Haskell's standard prelude defines `id` as the identity function, so the name `id` cannot be used for variables that represent numeric identifiers (a common convention).
2. Haskell's standard prelude defines `map` on lists, so the more general operation of mapping over functors (including lists) needed to be called `fmap` to avoid a name collision.
3. Because many collections, such as `Map` and `Set`, have operations with the same name, such as `size`, `null`, and `empty`, it's often necessary to import these modules qualified to avoid name collisions, in which case their contents must be referred to by a qualified name, such as `Set.size mySet` or `Map.size myMap`.
4. Because the `Num` type classes in Haskell's standard prelude declares `+`, `-`, `*`, and the `Fractional` type class extends the `Num` type class with `/` (among other operations)[33], any type for which these operations are defined must implement all of the operations declared by these type classes. The way that these operations are defined, in addition to some of the other operations declared by these type classes, do not make sense for vector math, and so use of these symbols for vector math is incompatible with Haskell's standard prelude. (Rust avoids this problem by using a different type-class or trait for each operator.)

Although the problems mentioned above can be circumvented using an alternate prelude[34], these problems are all symptoms of the absence of overloading combined with a lack of foresight. The lack of overloading is inconvenient enough that there are a number of proposals within the Haskell community to fix it[35]. In order to avoid these same problems, `ripl` supports overloading, according to the following rules:

1. When an overloaded name is type checked, each potential definition is type checked within its context.
2. The overloads are then sorted in ascending order, first by the number of errors, and then by the number of implicit conversions they produce within this context.
3. The first (best) overload is chosen.
4. If there is a tie for first place, then the overload is ambiguous and an error is raised.

8.3.10 + Simple and General Design

It's not uncommon in my experience that people (myself included) conceive of and solve problems at a lower level of generality and abstraction than possible. When this occurs in programming language design, it may introduce unnecessary complexity and limitations. Some examples of this include:

1. Needless grammatical distinction between statements and expressions.
2. Needless grammatical distinction between types and expressions.
3. Needless distinction between invocations of non-virtual member methods and regular functions (solved either by universal function call syntax, as in D, or by not having methods, as in C, Haskell, ML, `ripl`, Rust, and Scheme).
4. Needless distinction between array access and function application.

[33] `http://hackage.haskell.org/package/base-4.11.1.0/docs/Prelude.html#g:7`

[34] `http://hackage.haskell.org/package/classy-prelude`, `https://github.com/sdiehl/protolude`

[35] `https://wiki.haskell.org/TypeDirectedNameResolution`, `https://ghc.haskell.org/trac/ghc/wiki/SyntaxFreeTypeDirectedNameResolution`

5. Needless distinction between functions and operators (solved either by allowing all functions to be invoked normally or in infix, as in Haskell and Scala, or by not having infix notation as in Lisp).

Below is a table that compares various classes of features that can be used to accomplish the same goal in C++ and `ripl`, which demonstrates that languages can vary pretty widely in terms of complexity:

Table 7: Comparison of Various Constructs in `ripl` and C++

Feature	`ripl`	C++
Code organization	Namespaces	Headers, namespaces, and modules (expected in C++20[36])
Conditions	`if`	`if` statements and ternary expressions
Enumerations	`union`	`enum` and `enum class`
Functions	Functions	Functions and methods
Indirection	References	Pointers and references
Initialization	`define a (A b)`	`A a(b)`, `A a{b}`, `A a = {b}`, and `A a = A(b)`
Iteration	Recursion	`for`, range-based `for`, `while`, `do...while`, and recursion
Metaprogramming	Templates	Templates and textual macros
Printing	`println`	C-style `printf` and C++ style IO streams
Record types	`struct`	`class` and `struct`
Strings	`string`	`char[MAX_PATH]`, `char*`, `wchar*`, `std::string`, `std::wstring`
Type conversions	Functions	C-style, dynamic, reinterpret, static, and const casts[37]
Type-aliases	`define`	`typedef` and `using`
Unions	`union`	`union` and `std::variant`

By adopting more general solutions I think it's possible to reduce the complexity of a language. This is one of `ripl`'s goals, and its design should be continually revised as new way to generalize it are discovered. If the language gets a user base in future this could be done with non-backwards compatible major versions whenever a suitable number of potential improvements have been identified.

8.4 Powerful

8.4.1 + Templates

In statically typed languages, especially those without a top type (e.g. `Object` in Java or `Any` in Scala), parametric polymorphism is necessary to write generic functions and generic types (notably collections). Templates in C++ and D are like generics in other languages, with the addition of features that enable them to be used as a basis for type-level programming and compile-time metaprogramming. These capabilities enable the solution of problems that are be hard or impossible to solve otherwise, including:

1. Generic definitions of operations over vectors and arrays of arbitrary static length
2. Compile time dimensional analysis (for type-safe programming with physical quantities)
3. Emulation of type-safe discriminated unions without language-level support (as per C++17's std::variant)

Although templates in `ripl` have not yet been implemented (and thus have not yet been specified), example syntax can be seen in § 8.1.2.

8.4.2 + Compile-Time Evaluation

Compile-time evaluation (called compile-time function evaluation in D and constexpr in C++), is the ability to evaluate expressions without side-effects at compile time. In addition to being useful for performing compu-

[36] Dmitry Guzeev, A Few Words on C++ Modules, https://medium.com/@wrongway4you/brief-article-on-c-modules-f58287a6c64

[37] These casts are not functions but language level features, https://en.cppreference.com/w/cpp/keyword

tations that would otherwise need to be performed at run time, when combined with templates compile time evaluation is useful for type level programming.

The intent for `ripl` is that the compiler will be able to evaluate all pure expressions and functions, as long as their parameters can be computed at compile time, and that these functions can operate on types in addition to values.

8.4.3 + Type Classes

Whereas template parameters in C++ and D are essentially duck-typed at compile time (supplying a type for which an operation is not defined to a template that requires this operation will fail during template expansion), generic parameters in some other languages can be constrained up front using a construct called type-classes in Haskell, called interfaces in Idris, and called traits in Rust. The type-class approach has a number of advantages over the duck-typing approach in that it can be used to inform clearer error messages (C++ template expansion error messages are notoriously poor) and to establish clear type-level interfaces[38].

When combined with existential types, as in Haskell, type classes can be used to emulate OOP-style dynamic dispatch[39]. OOP-style dynamic dispatch is possibly the only feature of object-oriented programming that I thought `ripl` might ultimately lack, so it's good to see that there is a solution in use that does not require inheritance.

8.4.4 ? Lisp-style Macros

Because `ripl` has a number of things in common with Lisp, including partial evaluation of expressions before they are executed, and a uniform syntax, it may be possible to add some Lisp-style macros in future. There seems to be a fair bit of diversity between the macro systems of Common Lisp, Scheme, and Racket, so some research would need to be done if one of these were to be emulated in `ripl`.

[38] A similar construct called constraints and concepts is planned for C++20:
`https://en.cppreference.com/w/cpp/language/constraints`

[39] Haskell Wiki, Existential Type, Dynamic dispatch mechanism of OOP,
`https://wiki.haskell.org/Existential_type#Dynamic_dispatch_mechanism_of_OOP`

9 Implementation

9.1 Lexing

The lexer of the `ripl` compiler is quite straightforward. It takes the string input of the source code to be compiled and breaks it down into tokens, such as symbols, strings, and numbers, in addition to tracking changes in indentation and emitting indent and dedent tokens. These tokens are then handed off to the parser.

9.2 Parsing

Although earlier versions of `ripl` with a more complicated grammar used a parser generator (Happy when the compiler was written in Haskell, and ANTLR after it was ported to Scala), since the adoption of a Lisp-like syntax inspired by Readable Lisp S-expressions§6 (as described in § 8.3.1), the parser is now written directly in Scala, and the implementation is about half the length it was with ANTLR.

9.3 Post-Parsing

Post parsing is a step that converts free-form s-expressions into `ripl`'s untyped abstract syntax tree, and so performs conversions like `SExp(Name("if"), Name("a"), Name("b"), Name("c"))` to `If(Name("a"), Name("b"), Name("c"))`. It does such little work, that it might be wise in a future to fold it into the parsing step to cut down on unnecessary traversals. The only advantage of performing this as a separate step is that it is more modular and can be tested separately.

9.4 Reduction

Reduction is by far the most complicated and important aspect of the `ripl` compiler. It is the semantic stage that is responsible for overload resolution, namespace resolution, compile-time evaluation, type inference, and type checking. Presently it works by reducing all definitions in the AST lazily, so that evaluation of their results can be deferred until required by another definition.

The only problem with this lazy approach occurs with recursive definitions, which would loop endlessly as each attempt to compute their result would trigger another computation of their result. This is resolved by maintaining a history of visited definitions to detect cycles, and requiring recursive functions to have an explicit return type.

9.5 Code Generation

Code generation is the process of translating the post-reduction `ripl` AST into the LLVM-IR AST. It is pretty straightforward and relies on a monad ported from llvm-hs called `IRBuilder` to keep track of a name supply, local bindings, and blocks.

9.6 LLVM-Bindings

The LLVM-bindings are a port of the llvm-hs bindings and llvm-hs-pretty printer from Haskell to Scala, that enable the LLVM-IR AST to be represented and serialized as text.

9.7 Detailed Implementation Status

- ☒ Lexing
  - ☒ Comments
  - ☒ Indentation
  - ☒ Numbers
  - ☒ Names
  - ☒ Special symbols (e.g. `~`, `@`, `^`)
  - ☒ Strings
- ⊟ Parsing
  - ☒ S-expressions
  - ☒ S-expressions delimited by whitespace
  - ☐ Infix notation (e.g. `{x + y}`)
  - ☐ Prefix modifiers (e.g. `~`, `^`)
  - ☐ Selection syntax (e.g. `math.pi`)
- ⊟ Post-Parsing
  - ☒ Application
  - ☒ Function definition
  - ☒ Global variables/constants
  - ☒ If-expressions
  - ☒ Structs
  - ☐ Implicit conversions
  - ☐ Local Variables
  - ☐ Pattern matching
  - ☐ Prefix modifiers (e.g. `~`, `^`)
  - ☐ Templates
  - ☐ Type Classes
  - ☐ Unions
- ⊟ Reduction
  - ☒ Built-in arithmetic and logic
  - ☒ Compile-time expression evaluation
  - ☒ If-expressions
  - ☒ Implicit conversions
  - ☒ Lambdas
  - ☒ Overloading
  - ☒ Namespaces
  - ☒ Recursive functions
  - ☒ Selection
  - ☒ Structs
  - ☒ Type-checking
  - ☒ Type-inference
  - ☐ Arrays
  - ☐ Compile-time function evaluation
  - ☐ Constraints on purity
  - ☐ Constraints on mutability
  - ☐ Local variables
  - ☐ Pattern Matching
  - ☐ Templates
  - ☐ Type Classes
  - ☐ Unions
- ⊟ Code Generation
  - ☒ Application
  - ☒ Built-in arithmetic and logic
  - ☒ Functions
  - ☒ If-expressions
  - ☒ Structs
  - ☐ Arrays
  - ☐ Closure and lambdas
  - ☐ Local Variables
  - ☐ Pattern matching
  - ☐ Strings
  - ☐ Templates
  - ☐ Unions
- ☒ LLVM-Bindings
  - ☒ Representation of LLVM AST
  - ☒ Serialization of LLVM AST

10 Quantitative Comparison with Other Languages

In order to compare `ripl` with other languages in an objective way, so that it can be understood in relation to them, I identified 41 quantifiable language features to be used as a basis for comparison. I then evaluated each language by assigning ternary values of `+`, `?`, and `-` for features that were present, uncertain or not applicable, and absent respectively, resulting in the language feature table in § 10.2.

I chose ternary values and more quantitative features over continuous values and more qualitative features in an effort to reduce the subjectivity of the results. Language features were not weighted differently, because although some features are more important in differentiating languages than others, choosing weights to account for this would introduce more subjectivity. Languages were chosen on the basis of my familiarity with them.

Although the language feature table is useful in detailing the features of each language, because it consists of 20 data-points in 42 dimensions it is hard to visualize the high-level structure of the data and relationships between the languages by looking at the table itself. Fortunately, there are a number of data visualization and statistical methods that can help to understand this higher dimensional data. The results of these data visualization methods are presented throughout the rest of this section.

10.1 Notes on Statistical Methods

The language feature table symbols `+`, `?`, and `-` are converted to balanced ternary values of +1, 0, and -1 for statistical use. All distances calculated are Euclidean distances, and hierarchical clustering is done with the Ward-2 linkage method. The R code used to produce the figures in this document can be found embedded in this document's source at `https://github.com/SongWithoutWords/ripl/blob/master/README.org`.

10.2 Language Feature Table

(See the next page.)

Table 8: Language Feature Table

	C	C++	C#	D	Dart	Go	Haskell	Idris	Java	JavaScript	Kotlin	LLVM-IR	Lua	ML	Python	`ripl`	Rust	Scala	Scheme	Swift
Garbage Collection	−	−	+	+	+	+	+	+	+	+	+	−	+	+	+	?	−	+	+	−
Explicit Indirection	+	+	−	+	−	+	−	−	−	−	−	+	−	−	−	+	+	−	−	−
Ownership System	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	?	+	−	−	−
Static Typing	+	+	+	+	+	+	+	+	+	−	+	+	−	+	−	+	+	+	−	+
Dynamic Typing	−	−	+	−	+	−	−	−	−	+	−	−	+	−	+	−	−	−	+	−
Type Inference	−	+	+	+	+	+	+	+	−	?	+	−	?	+	?	+	+	+	?	+
Subtyping	+	+	+	+	+	+	−	−	+	+	+	−	+	−	+	+	−	+	−	−
Parametric Polymorphism (Generics)	−	+	+	+	+	−	+	+	+	?	+	−	?	+	?	+	+	+	?	+
Type Classes	−	−	−	−	−	−	+	+	−	?	−	−	?	+	?	+	+	−	−	−
Type Level Programming	−	+	−	+	−	−	+	+	−	?	−	−	?	+	?	+	−	+	?	−
Ad-hoc Polymorphism (Overloading)	−	+	+	+	+	−	−	+	+	?	+	−	?	−	?	+	−	+	?	+
Classical Inheritance	−	+	+	+	+	−	−	−	+	−	+	−	−	−	+	−	−	+	−	+
Prototypal Inheritance	−	−	−	−	−	−	−	−	−	+	−	−	+	−	−	−	−	−	−	−
Strict Evaluation	+	+	+	+	+	+	−	+	+	+	+	+	+	+	+	+	+	+	+	+
Type-safe Discriminated Unions	−	−	−	+	−	−	+	+	−	−	−	−	−	+	−	+	+	+	−	+
Null Safety	−	−	−	−	−	−	+	+	−	−	+	−	−	+	−	+	+	−	+	+
Pattern Matching	−	−	−	−	−	−	+	+	−	−	−	−	−	+	−	+	+	+	−	+
Mutable Data	+	+	+	+	+	+	−	−	+	+	+	+	+	+	+	+	+	+	+	+
Immutable Data	−	+	−	+	−	−	+	+	−	−	−	−	−	+	−	+	+	+	−	+
Constraints on Mutability	−	+	−	+	−	−	−	−	−	−	−	−	−	−	−	+	+	−	−	+
Constraints on Function Purity	−	−	−	+	−	−	+	+	−	−	−	−	−	−	−	+	−	−	−	−
C-style Syntax	+	+	+	+	+	+	−	−	+	+	+	−	?	−	+	−	+	+	−	+
Lisp-style Syntax	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	+	−	−	+	−
ML-style Syntax	−	−	−	−	−	−	+	+	−	−	−	−	−	+	−	−	−	−	−	−
Header Files	+	+	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
Whitespace Sensitive	−	−	−	−	−	−	+	+	−	−	−	−	−	−	+	+	−	−	−	−
Expression Oriented	−	−	−	−	−	−	+	+	−	−	−	−	−	+	−	+	+	+	+	−
Top Level Functions	+	+	−	+	+	+	+	+	−	+	+	+	+	+	+	+	+	−	+	+
Methods	−	+	+	+	+	+	−	−	+	+	+	−	−	?	+	−	−	+	−	+
Uniform Function Call Syntax	?	−	?	+	−	−	?	?	?	−	−	−	−	?	−	+	+	?	?	−
Compile Time Function Evaluation	−	+	−	+	−	−	−	+	−	?	−	−	−	−	?	+	−	−	?	−
Closures	−	+	+	+	+	+	+	+	+	+	+	−	+	+	+	+	+	+	+	+
Member Access Modifiers	−	+	+	+	+	−	−	−	+	+	+	−	−	−	−	−	−	+	−	+
Monadic IO	−	−	−	−	−	−	+	+	−	−	−	−	−	−	−	−	−	−	−	−
Dependent Typing	−	−	−	−	−	−	−	+	−	−	−	−	−	−	−	−	−	−	−	−
Operator Precedence	+	+	+	+	+	+	+	+	+	+	+	−	+	+	+	−	+	+	−	+
Operator Overloading by User	−	+	+	+	+	−	+	+	−	−	+	?	+	−	+	+	+	+	−	+
Names Must Be Declared Before Use	+	+	−	−	−	−	−	+	−	+	−	−	+	−	+	−	−	−	+	−
Deterministic Destructor Calls	−	+	−	−	−	−	−	−	−	−	−	−	−	−	−	?	+	−	−	+
Textual Macros	+	+	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
Lisp-style Macros	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	+	−
C#-style Properties	−	−	+	+	+	−	−	−	−	+	+	−	+	−	+	−	−	+	−	+

10.3 Hierarchical Clustering of Languages

One method that is useful for visualizing higher-dimensional data is agglomerative hierarchical clustering, which works by assigning each data point to its own group, and then repeatedly combining the two nearest groups, until all data points have been organized into a binary tree, often displayed as a dendrogram. The distances between languages are measured in n-dimensional space, where each dimension is a variable corresponding to one of the language features considered. When applied to this data set, this process yields a taxonomy of programming languages based on the features considered in the language features table:

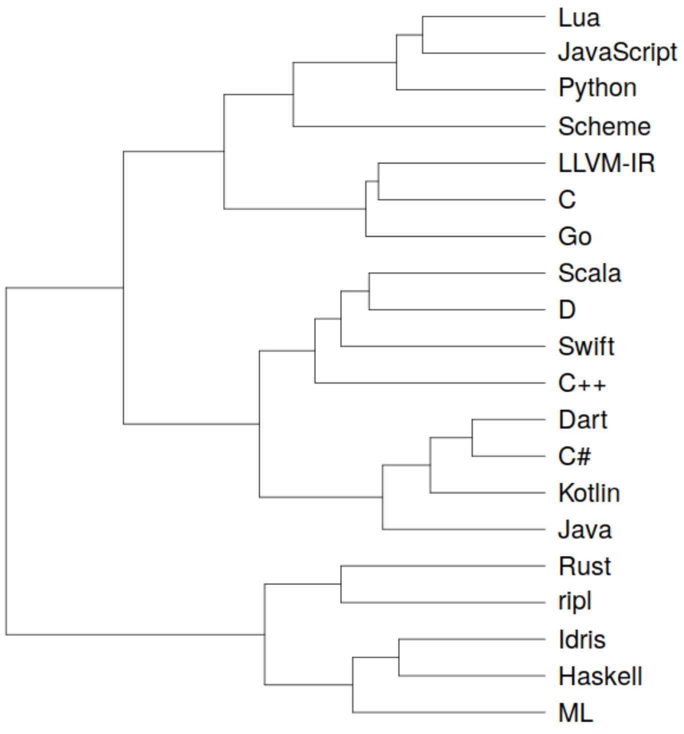

Figure 1: Hierarchical Clustering of Languages

The results suggest that the languages considered fall into one of four clusters, or categories:

Table 9: Language Categories Observed in Hierarchical Clustering

Language Cluster	Languages
Dynamically-typed	JavaScript, Lua, Python, Scheme
Statically-typed and imperative	C, Go, LLVM-IR
Statically-typed and object-oriented	C++, C#, D, Dart, Java, Kotlin, Scala, Swift
Statically-typed and functional	Haskell, Idris, ML, `ripl`, Rust

A more nuanced picture that accounts for the pairwise distance between all languages considered can be seen in the heatmap in § 10.4.

10.4 Heatmap of Distances Between Languages

Below is a heatmap of the distances between the languages considered. The languages are ordered along the axes using the hierarchical clustering seen in section § 10.3. It is worth noting that languages can be close due both to having similar features (e.g. Haskell and Idris) and lacking similar features (e.g. LLVM-IR and Scheme).

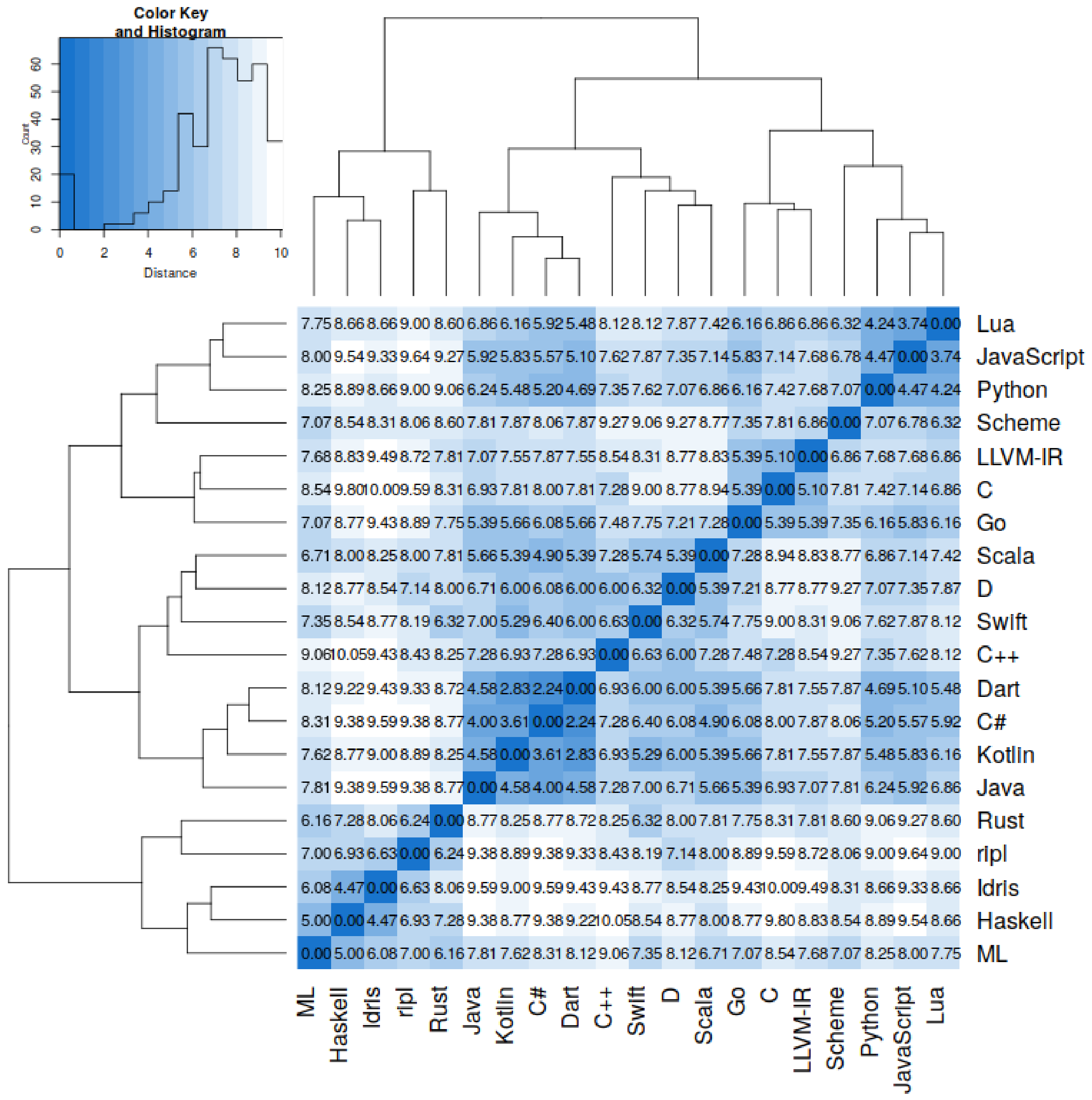

Figure 2: Heatmap of Distances Between Languages

While the blocks of colour along the diagonal reflect the same clusters as in the hierarchical clustering, the full set of distances between all languages in the heatmap reveal additional relationships across clusters, such as:

1. There is a block of similarity between statically-typed, garbage-collected, imperative languages, and dynamically-typed, garbage-collected, imperative languages.
2. Scheme is closer to the functional languages (especially ML) than its dynamically-typed neighbours.
3. ML is closer to other languages that are not both statically typed and functional than its neighbours.
4. `ripl` does not appear to be very close to any particular language (discussed more in section § 10.5).

10.5 Distances of Languages to Their Nearest Neighbours

In the heatmap in § 10.4, `ripl` does not appear to be very close to any other languages. To see if this is true, I determined each language's nearest neighbour, and sorted these nearest-neighbour relationships by distance:

Table 10: Languages Sorted by Distance to Their Nearest Neighbours

Language	Nearest Neighbour	Distance
C#	Dart	2.24
Dart	C#	2.24
Kotlin	Dart	2.83
JavaScript	Lua	3.74
Lua	JavaScript	3.74
Java	C#	4.00
Python	Lua	4.24
Haskell	Idris	4.47
Idris	Haskell	4.47
Scala	C#	4.90
ML	Haskell	5.00
C	LLVM-IR	5.10
LLVM-IR	C	5.10
Swift	Kotlin	5.29
D	Scala	5.39
Go	C	5.39
C++	D	6.00
Rust	ML	6.16
ripl	Rust	6.24
Scheme	Lua	6.32

The results indicate that `ripl` is the second most distant from its nearest neighbour of the languages considered. This is a positive result, because it suggests that `ripl` occupies a unique place in the feature-space considered and is not a duplicate or near duplicate of any of the other languages considered.

10.6 Median Distances of Languages to Others

In order to get a sense of what languages are more or less similar to others, I sorted them by their median distances to other languages. Although this evaluation is somewhat unfair to languages in smaller clusters, I think it is a useful and interesting measure.

Table 11: Languages Sorted by Median Distance to Other Languages

Language	Median Distance
Dart	5.66
Kotlin	5.83
C#	6.08
Go	6.16
Java	6.71
JavaScript	6.78
Lua	6.86
Python	6.86
Scala	6.86
Swift	7.00
D	7.07
C++	7.28
ML	7.35
LLVM-IR	7.68
C	7.81
Scheme	7.81
Rust	8.00
ripl	8.19
Haskell	8.66
Idris	8.66

The results indicate that Dart and Go (both developed by Google) are the two most normal languages (i.e. most similar to the other languages considered), and that Haskell, Idris, and `ripl` are the three most unusual languages (i.e. most dissimilar to the other languages considered). Although these result are somewhat biased by the fact that this sample includes more imperative languages than functional languages, this biased sample is in keeping with the fact that imperative languages are *vastly* more popular than functional languages[40].

Like the results in the section § 10.5 which suggest that `ripl` is unique, I think these results suggesting that `ripl` is unusual are also positive, because if `ripl` was highly similar to existing languages I might doubt it could provide something new.

[40] https://www.tiobe.com/tiobe-index/

10.7 Distance of `ripl` from Other Languages

In order to determine what languages `ripl` is most similar to, I ordered the other languages by their distance from `ripl`. The results suggest that `ripl` is most similar to Rust, Idris, D, and Haskell (in that order), and most dissimilar to C, JavaScript, and Java.

Table 12: Languages Sorted by Distance from `ripl`

Language	Distance from `ripl`
Rust	6.24
Idris	6.63
Haskell	6.93
ML	7.00
D	7.14
Scala	8.00
Scheme	8.06
Swift	8.19
C++	8.43
LLVM-IR	8.72
Go	8.89
Kotlin	8.89
Lua	9.00
Python	9.00
Dart	9.33
C#	9.38
Java	9.38
C	9.59
JavaScript	9.64

10.8 Multidimensional Scaling of Languages

Another data visualization method (that is similar to principal component analysis, but produces better results with this data set) is multidimensional scaling. The purpose of multidimensional scaling is to reduce a higher dimensional data set to a lower dimensionality while aiming to preserve the distances between the points. The results of this procedure on the data from the language feature table can be seen below:

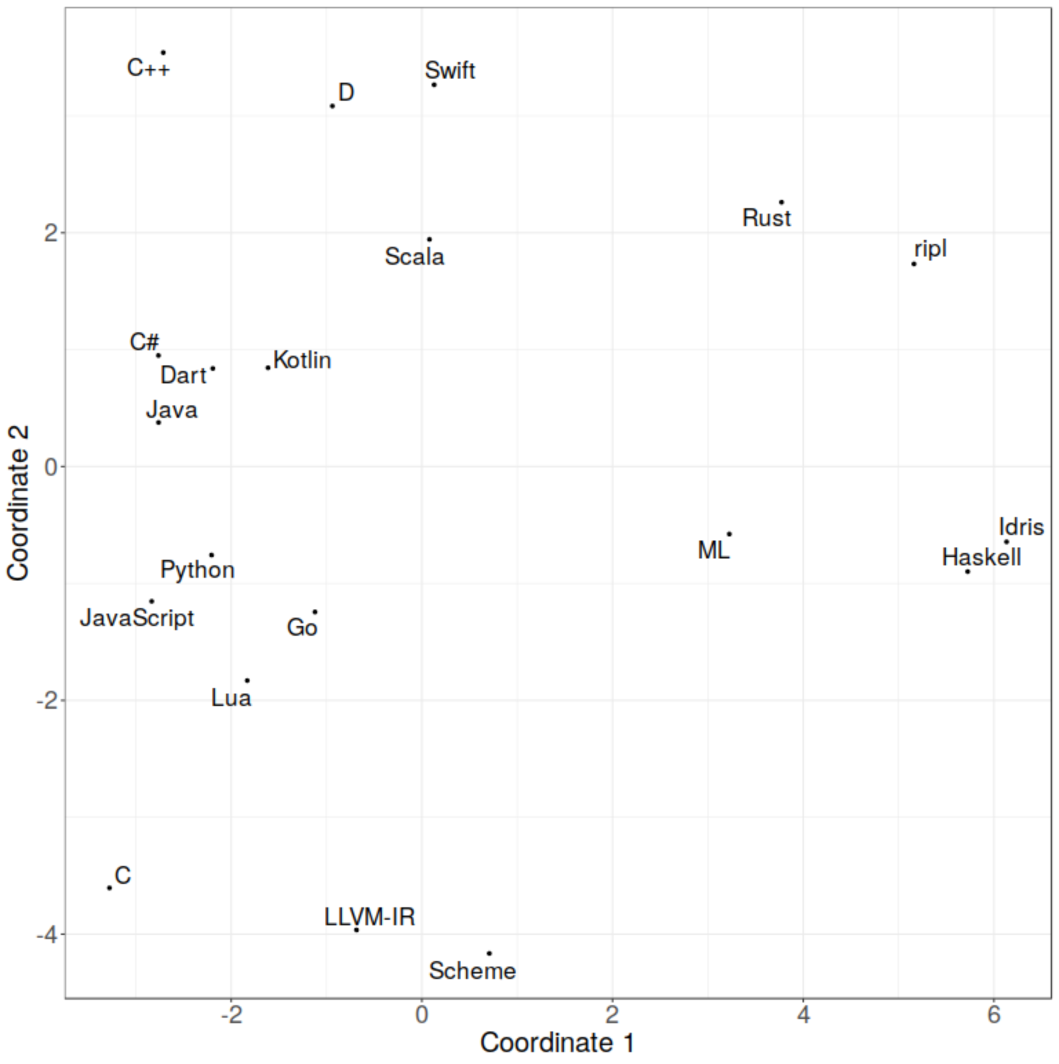

Figure 3: Multidimensional Scaling of Languages

These results are more nuanced than the hierarchical clustering in section § 10.3, and more easily interpreted than the heatmap in section § 10.4 (though not as precise), and place `ripl` roughly where it's intended to be: in the vicinity of Haskell, Rust, and D.

10.9 Qualitative Language Attributes as Linear Combinations of Features

In the previous sections we looked at visualizations and analyses of the distances between languages. Although it is possible to infer some more qualitative information from these distance based methods (such as the language clusters identified in § 10.3), these methods in general do not provide much insight into the more qualitative attributes of the languages considered.

In an effort to measure these more qualitative attributes, I expressed them as linear combinations of specific language features, by which they can be computed using matrix multiplication according to the following equation:

```
[Languages x Features] [Features x Attributes] = [Languages x Attributes]
```

The attributes considered are as follows:

- **Ergonomic**: features that make programming easier and more productive.
- **Functional**: features commonly associated with functional programming.
- **Imperative**: features commonly associated with imperative programming.
- **Powerful**: features that enable the expression of abstract ideas, such as meta and type-level programming.
- **Robust**: features that make programming less error prone, such as garbage collection and null safety.
- **Simple**: features that reduce complexity and the absence of features that introduce complexity.

The table on the following page expresses these attributes as linear combinations of the language features considered in § 10.2.

Table 13: Language Feature Attribute Coefficients

	Ergonomic	Functional	Imperative	Powerful	Robust	Simple
Garbage Collection	1.0	0.5	-0.5		1.0	1.0
Explicit Indirection	-0.5	-0.5	0.5			-0.5
Ownership System					0.5	-0.5
Static Typing	-0.5		1.0		1.0	
Dynamic Typing	0.5				-0.25	
Type Inference	0.5	0.5				
Subtyping	0.5	-0.5				
Parametric Polymorphism (Generics)				0.5		-0.25
Type Classes		0.25		0.5		-0.25
Type Level Programming				0.5	0.5	
Ad-hoc Polymorphism (Overloading)	0.5					
Classical Inheritance	0.25	-0.25		0.5		-0.25
Prototypal Inheritance	0.25			0.5		-0.25
Strict Evaluation			0.5			
Type-safe Discriminated Unions		0.5		0.5	0.5	
Null Safety					0.5	
Pattern Matching	0.5	0.5			0.5	
Mutable Data			0.5		-0.5	
Immutable Data		0.25			0.5	
Constraints on Mutability					0.5	-0.25
Constraints on Function Purity					0.5	-0.25
C-style Syntax						
Lisp-style Syntax						
ML-style Syntax						
Header Files	-0.5					-0.25
Whitespace Sensitive	0.5					-0.125
Expression Oriented	1.0	0.5	-0.5			0.25
Top Level Functions	0.5	0.5				0.125
Methods	0.5					
Uniform Function Call Syntax	0.5					
Compile Time Function Evaluation				0.5		
Closures	0.5	0.5	-0.5	0.5		
Member Access Modifiers					0.5	
Monadic IO	-0.5		-0.5			-0.25
Dependent Typing				0.5	0.5	-0.25
Operator Precedence	0.5					
Operator Overloading by User	0.5					
Names Must Be Declared Before Use	-0.5					
Deterministic Destructor Calls					0.5	
Textual Macros				0.25	-0.25	-0.125
Lisp-style Macros				0.5		-0.25
C#-style Properties	0.5					

The values of the coefficients chosen are largely subjective, but do provide a uniform rubric that can be used to evaluate the languages considered across the attributes chosen. The table is presented in a sparse format (without zeroes) to make it easier to read, and the missing values are replaced with zeroes before use.

Table 14: Normalized Language Attribute Scores

	Ergonomic	Functional	Imperative	Powerful	Robust	Simple
C	0.00	0.00	1.00	0.07	0.00	0.30
C++	0.37	0.22	0.86	0.79	0.48	0.00
C#	0.85	0.28	0.57	0.43	0.29	0.80
D	0.81	0.44	0.71	0.86	0.81	0.45
Dart	0.89	0.39	0.57	0.43	0.29	0.85
Go	0.48	0.33	0.71	0.14	0.24	0.85
Haskell	0.74	1.00	0.00	0.71	0.90	0.70
Idris	0.74	1.00	0.14	1.00	1.00	0.60
Java	0.56	0.17	0.57	0.43	0.33	0.80
JavaScript	0.74	0.42	0.29	0.57	0.14	0.85
Kotlin	0.81	0.39	0.57	0.43	0.43	0.85
LLVM-IR	0.00	0.11	1.00	0.00	0.05	0.45
Lua	0.74	0.42	0.29	0.50	0.05	0.85
ML	0.70	1.00	0.43	0.71	0.71	0.95
Python	0.89	0.36	0.29	0.57	0.05	0.80
ripl	0.78	0.72	0.64	0.86	0.90	0.20
Rust	0.56	0.78	0.71	0.57	0.71	0.05
Scala	1.00	0.67	0.43	0.71	0.71	0.90
Scheme	0.59	0.61	0.14	0.50	0.14	1.00
Swift	0.67	0.67	0.71	0.57	0.71	0.35

Table 15: Language Attribute Rankings

	Ergonomic	Functional	Imperative	Powerful	Robust	Simple
1	Scala, 1.00	Haskell, 1.00	C, 1.00	Idris, 1.00	Idris, 1.00	Scheme, 1.00
2	Dart, 0.89	Idris, 1.00	LLVM-IR, 1.00	D, 0.86	Haskell, 0.90	ML, 0.95
3	Python, 0.89	ML, 1.00	C++, 0.86	ripl, 0.86	ripl, 0.90	Scala, 0.90
4	C#, 0.85	Rust, 0.78	D, 0.71	C++, 0.79	D, 0.81	Dart, 0.85
5	D, 0.81	ripl, 0.72	Go, 0.71	Haskell, 0.71	ML, 0.71	Go, 0.85
6	Kotlin, 0.81	Scala, 0.67	Rust, 0.71	ML, 0.71	Rust, 0.71	JavaScript, 0.85
7	ripl, 0.78	Swift, 0.67	Swift, 0.71	Scala, 0.71	Scala, 0.71	Kotlin, 0.85
8	Haskell, 0.74	Scheme, 0.61	ripl, 0.64	JavaScript, 0.57	Swift, 0.71	Lua, 0.85
9	Idris, 0.74	D, 0.44	C#, 0.57	Python, 0.57	C++, 0.48	C#, 0.80
10	JavaScript, 0.74	JavaScript, 0.42	Dart, 0.57	Rust, 0.57	Kotlin, 0.43	Java, 0.80
11	Lua, 0.74	Lua, 0.42	Java, 0.57	Swift, 0.57	Java, 0.33	Python, 0.80
12	ML, 0.70	Dart, 0.39	Kotlin, 0.57	Lua, 0.50	C#, 0.29	Haskell, 0.70
13	Swift, 0.67	Kotlin, 0.39	ML, 0.43	Scheme, 0.50	Dart, 0.29	Idris, 0.60
14	Scheme, 0.59	Python, 0.36	Scala, 0.43	C#, 0.43	Go, 0.24	D, 0.45
15	Java, 0.56	Go, 0.33	JavaScript, 0.29	Dart, 0.43	JavaScript, 0.14	LLVM-IR, 0.45
16	Rust, 0.56	C#, 0.28	Lua, 0.29	Java, 0.43	Scheme, 0.14	Swift, 0.35
17	Go, 0.48	C++, 0.22	Python, 0.29	Kotlin, 0.43	LLVM-IR, 0.05	C, 0.30
18	C++, 0.37	Java, 0.17	Idris, 0.14	Go, 0.14	Lua, 0.05	ripl, 0.20
19	C, 0.00	LLVM-IR, 0.11	Scheme, 0.14	C, 0.07	Python, 0.05	Rust, 0.05
20	LLVM-IR, 0.00	C, 0.00	Haskell, 0.00	LLVM-IR, 0.00	C, 0.00	C++, 0.00

The results suggest that `ripl` can be thought of as something like a more functional D, a more powerful Rust, or a more imperative Idris. Additionally, they suggest that `ripl` is not a master of any one domain, that it sacrifices conceptual simplicity in favour of being robust, powerful, and ergonomic (in that order), and that it strikes a fairly even balance between functional and imperative programming.

10.10 Summary of Findings

The statistical comparison of `ripl` with the 19 other languages considered suggests the following:

1. `ripl` is part of a cluster of statically-typed, functional languages (§ 10.3).
2. `ripl` is most similar to Rust, Idris, D, and Haskell in that order (§ 10.7).
3. `ripl` is the second most distant from its nearest neighbour, suggesting it it unique (§ 10.5).
4. `ripl` has the third highest median distance from other languages, suggesting that it is unusual (§ 10.6).
5. `ripl` is not a master of any one domain (§ 10.9).
6. `ripl` sacrifices conceptual simplicity in favour of being robust, powerful and ergonomic (§ 10.9).
7. `ripl` strikes a balance between functional and imperative programming (§ 10.9).

11 Future Work

So far 32 of the 58 compiler features identified in § 9.7 have been completed, and it is likely that additional work will be discovered during implementation of the remaining features. Some aspects of `ripl`'s design that will be interesting to explore in future include type-level functions (mentioned briefly in § 8.4.1), Lisp style macros (discussed in § 8.4.4), a potential Rust-style ownership system, and decisions around garbage collection (discussed in § 8.1.11). If and when the `ripl` compiler is fully implemented and the language has some standard library functionality, dependent typing would be an interesting and promising avenue for future development.

12 Conclusion

Writing this paper and performing the analysis in § 10 has led me to a number of conclusions:

1. Programming languages are complex. Although the features that were identified in § 10 were useful for conducting a high-level comparison of the languages considered, a feature set of this kind cannot account for the full complexity of a programming language, because similar features may differ between languages in important ways.
2. Programming languages are diverse. Hierarchical clustering of the languages considered yielded four highly distinct language clusters, namely statically-typed and functional, statically-typed and object-oriented, statically-typed and imperative, and dynamically-typed. This diversity among languages is further evidenced by the diversity within these clusters, which contained such heterogeneous elements as Idris and Rust in the functional cluster, C++ and Scala in the object-oriented cluster, Go and LLVM-IR in the imperative cluster, and Python and Scheme in the dynamically-typed cluster.
3. Programming languages continue to improve. Older languages like C++ and Haskell continue to develop, and new and innovative languages like D, Idris, Racket, Rust, and Scala continue to emerge.

This complexity, diversity, and ongoing improvement suggests to me that there is still room for new languages, both for practical use, and as experiments that can inspire and influence other languages. The results of § 10.5 and § 10.6 indicate that `ripl`'s feature set is both unique and unusual among the languages considered in § 10, which suggests that `ripl` may have a niche among programming languages. My hope for `ripl` is that it will encompass the features that I value (detailed in § 8), while maintaining a simple and general design (discussed in § 8.3.10), and striking a balance between low-level and high-level features so that it can be used comfortably and effectively across a broad range of applications.