Ether's Almanac

Your handbook for anything Ether, Ray & OrbitMines.

OrbitMines Research
Fadi Shawki
fadi.shawki@orbitmines.com

31 December 2026 DRAFT: POSSIBLY IMPRACTICALLY VAGUE

Discussion Channel

Introduction

Greetings! This is the Ether's Almanac! Anything OrbitMines-related you may need like using the Ether (IDE) and the Ray programming language is contained in it. We'll update the almanac regularly, expect larger updates at the end of each year. The Ether is a collective name for projects at OrbitMines. OrbitMines' goal is to eventually do research on the gamification of science, engineering and education: literally turning them into a sandbox videogame! But my judgement is that this in practice turns out to be a particularly difficult problem to think about. So instead (for the moment) we turn ourselves to a more practical problem: Programming infrastructure. If it is such a difficult problem to think about, why not advance the very tools with which we do our thinking! The thinking being that it might stimulate future thought towards this hard problem. Currently, the plan is as follows: - (1) Develop the Ray programming language, as a temporary placeholder to build out infrastructure for the future. - (2) Build a reprogrammable visual interface (an IDE) which will allow us to go beyond what a typical programming language does. - (3) Build tools within that interface for analyzing, comparing and cherry-picking features of ALL existing (programming) languages and their libraries: The Ether Library Project. - (4) Do the same for rendering/physics/visual/game engines. - (5) By the time this is all set in motion, have thought of an idea to get started with gamification. If you're interesting in following along with these projects, or even contributing!, join Our Discord. If you're here to learn more, then let me, without further ado, get you started on the Ray programming language!

A. The Ray Programming Language

If you're a beginner and have never looked at a programming language before, no worries, we got you covered! But especially for those who are already familiar with a programming language, let me right out of the gate throw some code at you to look at, without having explained anything yet about the programming language. Perhaps that might already give you quite some information.namespace Unicode class CodePoint < Hexadecimal{length ═ 1..6} class Scalar < CodePoint   dynamically assert this < 0x110000 && !(0xD800 ⩽ this ⩽ 0xDFFF)
 class UTF-8 < TF, sequence: (prefix: 1[]{length ═ 0..4},   U0: Binary{length ═ 8 - prefix.length}{⊢0},   (10
U1: Binary^6) if prefix ⊢11$_2$ (10$_2$, U2: Binary^6) if prefix ⊢111$_2$ (10$_2$, U3: Binary^6) if prefix ⊢1111$_2$)[]   as (═ CodePoint[]) => sequence.map(.U0, .U1, .U2, .U3)
U+{codepoint: CodePoint.String}: Scalar => codepoint
U+1F525 // NULNote that all the codeblocks in this book are executable and editable! Try for instance changing the hexadecimal codepoint to another unicode character!   The Ray programming language hopefully should feel somewhat familiar, somewhat alien to anyone with a programming language background. While it has many characteristics of a usual programming language, it deviates from the typical in quite a few ways, as you'll see. The hope is that the language is particularly useful for modelling the behavior of other (programming) languages (and replace them where convenient), whether low-level (close to the machine's instructions) or high-level. At least for the UTF-8 example above, the code to express what that encoding means is very minimal. With that comes the additional complexity of being able to express everything that they do, and hopefully better. Aesthetically, the aim is to minimize verbosity while maximizing clarity. Of course there will be a certain know-how required to feel completely comfortable with a programming language. But especially if you're used to existing programming languages, the hope is that working with the Ray programming language feels very freeing. So whether you're interested in developing web applications, compiler engineering, (future) game programming or formalizing mathematics. This should be for you!

§0. For Beginners

If you're starting out learning a programming language for the first time, great! This section is for you. If not

Skip ahead to §1

.

§0.1

§1. How to Install

There are several ways of installing the Ray programming language: - (1) You can download Ether, which includes the Ray programming language by following this link:DownloadΣther for Linux- (2) You can otherwise go to the GitHub Releases and download the relevant installer of the latest version. - (3) Or manually clone and build the application git clone git@github.com:orbitmines/ray.gitcd ray && ./install.sh

§2. Programming Fundamentals

Let's start with a bunch of important things many programming languages cover! And importantly, how the Ray programming language differs from the usual approach. Though plenty should feel familiar regardless of your programming language background.

§2.1 Superposing Variables

"A" | "B"boolean = false | true"A" & "B""A" & "B" // Multiple possible objects"A" &+ "B" // Single object, multiple components"A" + "B" // Same as &+, but overwrites.The difference (and usefulness) of &, &+ is best stated with an example. (&) You can have multiple programs, and (&+) each program can be executing in many places. In the following section (§2.2 Rays: Arrays, Trees, Graphs) you'll see another use for it.Program   // SingleProgram & Program   // ManyProgram & (Program &+ Program) // Many, and one has many cursorsx =

"AB" + truex.next // ═ falsex = true + "AB"x.next // ═ "A"x = "AB" &+ truex.next // ═ "A" & false+1 <- x: "A" -> -1+1 <- numberline -> -1"A" + numberline"ABC".next // "A"("A" + numberline).next // "A" + 1 ═ "B"x = "A" &+ numberlineYou can extract the components in two ways. (1) By using types. Which we'll talk about later §2.4 Types: Patterns. Or (2) by using the '##' operator.string: String = xequipped_structure: Ray = x
x## // ═ [string, equipped_structure]Let us turn to the next section to unpack what that means. Starting with what this 'equipped structure' (our numberline) called a Ray is.

§2.2 Rays: Arrays, Trees, Graphs

The Ray programming language is a rather high-level programming language: though it allows you to define pretty low-level stuff! In its own abstractions it ignores how datastructures are usually encoded in computers and it ignores what is supposedly the 'more efficient' approach when dealing with our current hardware. Instead it relies heavily on its compiler to sort out what is appropriate and efficient. It uses the most fundamental data structure to the Ray programming language - and with that the only: - the Ray (; hence the name). All datastructures are made from it: Objects (; called Nodes in Ray), Numbers, Types, Arrays, Trees, (Hyper)Graphs, Functions: everything. We'll cover each of them separately, so strap in! x: Hypergraph = 1, 2, 3x: Graph = 1, 2, 3x: Tree = 1, 2, 3x: Array = 1, 2, 3x: Graph = 1, "2a" & "2b", 31, +2, +3, +4// 1, 3, 6, 101 | +2 | +3 | +4// 1 | 3 | 6 | 10Superposed values, map all the possible values according to any function called on them, like:(1 & 2 & 3) * 2 // 2 & 4 & 6That functionality is also available to the iterable structures:[1, 2, 3].map(*2) // [2, 4, 6]It's worth noting that mapping, retains structure. So if we for instance have the following graph.And we map it:x: Graph = false, true & false, truex.map(!) // true, false & true, falseStructure is retained:Usually in a programming language, the structure which we're mapping over isn't available to the mapping function, but it is for the Ray programming language. Whenever you map over a structure, each entry also has the equipped Ray alongside it (it's a component which overrides the original entry (+). This is necessary as certain things, like Numbers, already have structure equipped; a number line for example. As we'll discuss in the following section):x: Number = [1, 2, 3]x.map(entry: Number + Ray => entry + entry.index)// [1, 3, 5]The mapping function can also include a filter which decides which entries should be mapped. (In category theory this is referred to as a lens.) This filter is applied just like any type filter, but instead on the mapping function.[1, 2, 3].map{.index ═ 2}(*10) // [1, 2, 30]Since structure is accessible to mapping function, one of the things you might want to do is to rewrite that structure in place with a different structure.
Now the limitation here is that the mapping function only maps over all the entries, perhaps you'd want to do something slightly more complicated. Like matching to, and then rewriting substructures. (As in typical graph rewriting)

§2.3 Numbers

Like other languages, there are for loops in the language to iterate over a number of things in some iterable. Although the syntax is slightly different, in Ray it is just treated like any other definition. In the following example, (~) is just a filter over the iterable. And anything in the following block get's executed for each entry.(0 -> +1) ~{< 10} for i => /* */Or you can use the following equivalent code, using (->), which means an infinitely generating iterable, without any structure/values defined on it. (But you do get access to the structure that is the iterable):(->) ~{index < 10} for => /* Use .index here */Another equivalence would be:10.times => /* .index is also available here */

§2.4 Types: Patterns

"A"[]"A", "A", "A""A", "B", "B" ═.instance_of "A", "B"[]"A", ["B"], ["B"] ═.instance_of "A", ["B"][]"A", ["B", "B"] ═.instance_of "A", ["B"[]]In many languages you have a spread operator if you want to pattern match to an array. So typically that would mean:first, middle: String[], last = "A", "B", "C", "D"The middle here, matching to both "B" and "C".Then for convenience, this spread operator is defined:first, ..middle, last = "A", "B", "C", "D"Which is just alternative syntax for defining an array ([]). So any place you have ([]), you can also use the prefix (..) This for instance would also work:first, middle: ..String, last = "A", "B", "C", "D"x: Binary{length ═ 32} = Binary{length ═ 8}[]{length ═ 4}Because the length check is quite common, there's a cleaner looking equivalence using superscripts:x: Binary^{32} = Binary^{8}[]^{4}Under the hood it uses the (^) operator. So equivalent code is:x: Binary^32 = Binary^8[]^4Notice that there's a slight ambiguity here, the (^) operator is also in use by a Number (Meaning exponentiation), which would be usually overwritten by the type Binary for instance. But as long as you access the base type like Binary, they prefer to use this interpretation of length of the Number. The moment you alter the base type, they default to exponentiation, for instance:Binary{═ 1..4}^{2} // ═ 2 | 4 | 8 | 16, ≠ 1..4[]{length ═ 2} One of the things you might want to do, since every variable is potentially a large number of superposed variables. Is say: I only want a single instance of that object.x: 1 NumberWhich under the hood, simply is a type filter:Number{#.count ═ 1}Or if you have an object with a finite number of permutations; when there's a finite number of possibilities that can fill that type. (Like a Number of finite length) You can then say: I want a percentage of the possible objects.50% ("A" | "B" | "C" | "D")1/2 Binary^{32}0.5 "A" | "B" // Is the same as 1 ("A" | "B")

§2.5 Programs/Functions

Those familiar with other programming languages, might think: what about Variadic functions? (A function with a variable number of arguments) There's a simple interpretation of what that means. If you remember that in types the comma (,) operator just concatenates structures. The same is true for functions. So given the following function:varargs (a: String, b: Number[], c: String[])We actually have, like all functions, only a single argument, it's just that it is described structurally by the variables a, b & c, in sequence. In a future version of the language which isn't just text-based, you can imagine that this 'single' argument which is described structurally, doesn't just need to be an Array. It could be some Graph for instance. We can call it with a variable number of arguments, whether they originate from other arrays or not.part: String[] = "c1", "c2"varargs("a", 1, 2, 3, part, "c3")Remember how ambiguities were handled in types, it's the exact same here!

§2.6 Equality & Equivalence

One of the equivalences which exists in the language, is for instance that a String is equivalent to String[] when concatenated together, such that:"ABC" == "A", "B", "C"
x: String = "A", "B", "C"

§2.7 Transactions and Reversibility

§2.8 Undecidability

§2.9 Classes & Namespaces

Classes and Namespaces are a typical way of grouping a bunch of stuff together in a single entity. (They are not actually primitives in the Ray language like most other languages). Like the if/else functionality and other coroutines, they are defined within the standard library! They are created by binding a function's context and defining some variables on it. Unlike other languages, the entire body of the class is the default constructor, so you don't specify one explicitly. Anything called on static which doesn't depend on an instance's parameters or which wouldn't be effected by getting called multiple times, gets taken out of the constructor if possible. So the following things aren't actually in the constructor:class Example static Var = 5
 static class InnerClassThings like this (+1) would be part of the constructor:class Example static Var = 5
 Var += 1If you want you can accept any positional argument like a function definition:class Example (x: String)Any variable defined on the type is automatically also part of the constructor, by passing it named to the constructor:class Example (x: String) y: String
Example("X", y: "Y")You'll find that you won't need different constructors as often, as you have access to a very expressive syntax for defining different possible types which match the constructor. If for sake of code clarity you still want to separate the constructors, you can by overriding the static constructor (you can omit (super) in it, in which case it will run before your defined constructor):class Example static ()   this // is available in this context.
 static (a: String)   super(property: a)class Enum < .A | .B | .C(: String)class Enum < A | B | C class A class B class C (var: String)x: Enum = Enum.Ax.match A => 1 var: B => var * 2 C("A") => 3 C<var: "B"> => 4 C(var) => var * 5 C => 6

§3. Ecosystem

One of the difficult design decisions, was what to do with the whole 'Call by value vs reference' ordeal. By which I mean, should in the following example, (x) get updated: class Example (field = "A")
x = Example()
update(var: Example) var = Example("B")
update(x)x.field // Is it "A" or "B" here?Saying it is always "Call by reference" would mean it is "B" here, saying "Call by value" would mean it's always "A" and that it's impossible to modify an object other than returning a new version. In most modern programming languages it is: Assignment, so (=), is actually reassignment of that variable in scope: And shouldn't effect the variable out of scope. But then suddenly other mutations of that object don't qualify under this distinction. In the Ray programming language, any mutation is an assignment (=) somewhere in the structure. To me it seems kind of arbitrary that one is a more dangerous ground for bugs than the other. Whatever the reason, there currently exists a certain expectation of what kind of mutation should apply out of scope, and which shouldn't. Going for the option of making everything "Call by reference" would surely lead to many bugs in the language; Another approach is required.   Instead we default to "Always call by value", and in this chapter we introduce the idea of variable versions, and variable locations. As one of its uses you can determine to which version and location the mutation should apply. Which in the above example would be done with the location (@) operator combined with (<-) as to indicate that it should be updated in the whole callstack which led to this function and its own context:update(var: Example) var @ <- = Example("B")You can also put this on the parameters, with the same effect:update(var @ <-: Example) var = Example("B")Note that in a concurrent setting, you can also turn that arrow around (->) to view & edit the variable in any thread you gave the variable to.Or if we want to only update the caller's context.var @ &caller = Example("B")Or, as we'll introduce later in this chapter, we might even want to update it in all locations, whether that is some remote location like a database or mirror of that data. We can use (*) to select all locations:var @ * = Example("B")Let's start by exploring how these locations work.

§3.1 Location & Assignment

§3.2 Networking

§3.3 Version Control

§3.4 Access Permissions

§3.5 Hosted Variables & Packages

§4. Extended Fundamentals

§4.1 Probability

§4.2 Choice

While randomization is a useful abstraction, sometimes you might want a slightly different concept. Which is where choice comes in. To flag that a required value can be chosen arbitrarily (by the runtime or even the Player). Unlike a random variable which can't be picked uniformly for infinitely generating structures, choice works just fine: There can be a preference or tendency for a certain kind of object. Choice is simply saying: we don't care about this information. We can use it in filters:[1, 2, 3].map{choose 1}(*10) // [10, 2, 3] | [1, 20, 3] | [1, 2, 30]
Number{choose 5}Call it directly:choose 1 Numberchoose 50% ("A" | "B" | "C" | "D")Or pass it to any function which will fill the type automatically (the choice having to disambiguate where necessary).func (a: String, b: Number[], c: String)func(choose, choose) // Choose two variables, the second can be a Number[] or a StringNote that choose, uses the (===) operator, so you might expect that if a variable is used more than once, it can only get chosen once. But that is not the default behavior, it does use (===), but each location the variable finds itself in, is separately equipped with a Ray (forming a new composed variable). Meaning where it is in the structure. And that structure, is not the same in both locations, and thus the two variables are differentiated as separate, and can be chosen separately.var = 2[1, var, var].map{choose 1}(*10) // [10, 2, 2] | [1, 20, 20]var = 2[1, var, var].map{choose 2}(*10) // [10, 20, 20] | [1, 200, 200]An example of where (choose) is used, is in a function defined on Iterable, the (unordered) function. Which says: I don't care about the order, or even what kind of structure yields the values, I just want it to yield them.class Iterable unordered =>   choose Iterable{.every(this.contains(.)) && .count == count}Which can be useful because certain compiler optimizations might work when order doesn't matter. We can also force a player to make that arbitrary choice:@me.choose String

§4.3 Coroutines

§4.4 Concurrency

§4.5 Punctuation

The familiar parenthesis () are used to group certain kinds of operations, to prefer a particular interpretation over another. The Ray programming language extends this notion a little further than most programming languages. Where you're allowed to introduce parenthesis pretty much anywhere, and there's a valid interpretation of what that means. So instead of doing the following:condition ? var == 5 : var ⩽ 5You might do this:var.(condition ? == 5 : ⩽ 5)Or even:var.(condition ? == : ⩽) 5Note that you have to include a preceding (.), as the syntax for [()] is reserved for for function definitions. The same can be done with property getters, so you can have things like:Symbol: Char = Unicode.GeneralCategory.(Punctuation | Symbol)Which superposes both properties with the (|) operator. Note that anything within parenthesis is always a closure. In it, the entire variable you're accessing is loaded in the context. So if it defines a (.next) method, and your scope also has a (.next) method, the .next from the object is used! This could be unexpected behavior.   The Ray programming language is also pretty lenient in it's omittance of parenthesis and allowance for spaces as punctuation. After a superposed variable, you can omit parenthesis by using a space to go straight to its type or call any other method on it:A | B : StringA | B .methodA | B {length == 2}You can omit the use of (.) in favor for a space, as is usually already the case for special character operators, but in Ray you can also do this with any property:"A".lowercase"A" lowercaseWith those two things we can choose between these sorts of syntax, whichever one seems clearer to you:(0 -> +2) ~{< 10}.for(i => /* */)(0 -> +2) ~{< 10}.for i =>(0 -> +2) ~{< 10} for i =>There is also the (--) operator which wraps the whole line before it.1, 2 -- .map(+1) // 2, 3Additionally it can also be used after newlines and with if statements which optionally wrap the line.class IPv6 as (== String)   this   -- .embed_ipv4 if ==.instance_of "::ffff:0.0.0.0/96"   -- .embed_ipv4 if ==.instance_of "64:ff9b::/96"   .compress_zeros   .lowercaseThen there is the (~~) operator, which does the exact same thing, but returns the original thing you call the successive functions on. Which is useful for creating one-liners like:mac_address: Binary^{48} = secure Binary^{47}.random ~~ .[6].push_after(1)

§5. Playerfacing

§5.1 Theorem Proving

§5.2 Language Templating

§5.3 Geometry

§5.4 UI

§6. The v0 Runtime & Compiler

§6.1 Self-modifying Types

In order to understand the runtime, we must first extend the fundamentals with one more concept; a further generalization of dependent types: self-modifying types. While dependent types are incredibly useful, in both looking ahead or behind in a pattern, there is one thing that they typically can't do. Which is to express a pattern which arbitrarily modifies itself. The idea is simple enough to understand: Something is found in the pattern, which changes the pattern arbitrarily: How should one interpret what happens next?; Should you reinterpret the whole thing? Should you only reinterpret what comes next? That's the question we explore in this section. This concept is pretty general and can be applied to everything (like for instance our self-modifying functions which alter its own control-flow). In the context of languages, this self-modifying behavior is known as an Adaptive grammar. And I'm here equivalencing the meaning of a pattern/type/grammar. The Ray programming language uses such an adaptive grammar (or self-modifying type), and usual type matching also supports it! Throughout the standard library you might find definitions for additional syntax like:

```
`An example` // Can also be used in front of definitions
`{string: String}` => string
`An example` // Valid syntax
```

Let's get started on the definition of expressions in the language to understand how that works:

§7. Other Features

§7.1 (Unicode) Strings

§7.2 Units

§7.3 Time

§7.4 UUID

Wrapping up