Arrays and strings are values, not references

Function parameters are passed as immutable references, but they can optionally be mutable or copied

Probably gradual typing

Arrays copy on assignment (probably should implement as copy on write).

Rust function regex (in JS): /^(?:fn) \w+\(([^,)]+)?(?:,([^,)]+))\*\)/mg

From Antman to D:

* Change function main() to void main()
* Automatically import std.stdio
* Change let to auto
* Function parameters should default to immutable
* Change out to ref

Return as reference or copy? Actually, probably return by move

Assign objects/structs as reference or copy? Probably copy, maybe requiring a “copy” keyword

All assignments are unique! Use “alias” keyword (without an equals sign) to create an alias

* Still a bit confusing when to use alias vs var

How to swap variables without creating unnecessary copies?

References within objects/structs? Use Pool and Noodle (region-based memory management)

Struct syntax: mix of Rust, Swift, and JavaScript (classes)?

Name: Serene?

For indexes that may be invalid:

* either (let x = pool[noodle]?) or return False;

pool[pool[pool[head].next].next]

head.next(pool).next(pool)

[[[head pool].next pool].next pool]

Declarators declare declarables: things like var, val, function, class (or struct), etc.

Values: any value of any type (as well as types themselves?)

Accessors: New, Look, Copy, Move, Lend

Modifiers: private

Control flow: if, elseif, else, for, while, etc.

Operators: +, -, \*, etc.

Probably use @ for macros (like Julia)?

Mutating methods are written like list.sort!()

Should functions be nominally typed? It would make higher-order functions difficult

Functions defined inside of functions? Maybe? Rust allows it, but doesn’t allow them to use local vars

**Structs**

Structs should be similar to those in Rust, using traits (which are basically the same as interfaces) and impls. Unlike Rust, the impl can optionally be inside the struct definition.

No overriding methods! It just creates a whole bunch of “invisible behavior” that confuses the reader. Need to figure out a good way to have something as convenient as default implementation without relying on overriding

Can new traits be implemented for built-in types? (Rust allows this)

**Generics**

Ugh. I guess I need these…

**Language Design Principles**

From Python: There should be one-- and preferably only one --obvious way to do it.

From Ruby (unofficially): Principle of least surprise

From Go: Orthogonality

In a language with no shared mutability, shadowing for immutable variables doesn’t really make sense. Because replacing a value and mutating the original value effectively do the same thing here, shadowing an immutable variable essentially allows you to mutate it. If you’re going to allow a variable to be replaced, you might as well make it mutable.

Best choice is probably to have variables be mutable by default, but also have “unshadowable” immutable variables/constants