Contents

1 Introduction

- Learn a language by answering the following questions:
  - What is the typing model? Static dynamic, strong weak?
  - What is the programming model? OOP, functionall, procedural, hybrid of which?
  - How will you interact? Compiled, interpreted, VMs?
  - Design constructs/core data structures? Pattern matching, collections, unification?
  - Core features that make it unique?
- The languages:
  - Ruby—OOP representative.
  - Io—concurrency constructs w/ simplicity, uniformity and minimality of syntax.
  - Prolog—Parent to Erlang? Old. Nothing else mentioned.
  - Scala—Functional + OOP to Java.
  - Erlang—Functional w/ concurrency, distribution + fault tolerance *right*. BAse of CouchDB.
  - Clojure—On JVM, same concurrency as versioned dbs. Lisp dialect
  - Haskell—Pure functional, archetypal typing model.
- Glossary (to be all on the same page):

  Interpreted Executed by an interpreter rather than a compiler.

  Strongly Typed Errors when types collide.

  Dynamically Typed Types bound at runtime rather than compile time. Generally means types inside functions are only checked on execution.

Duck Typing If an object has a function then that function is invokable without type checking for the parent.

Object Oriented Encapsulation (data + behavior together), inheiritance and polymorphism.

Prototype Language Every object is a clone of another, a style of OOP.

2 Ruby

- Optimized w/ syntactic sugar, programmer efficiency.
- Interpreted, OOP, dynamically typed, strongly typed, duck typed scripting language.
- Every piece of code returns, even if only `nil`.
  - Functions return the value of the last expression.
- Purely OOP, e.g. `4.class = Fixnum` and has methods viewable by `4.methods`.
- `if`, `unless`, `while`, `until` can be used either inline or in block form.
- `nil`, `false` are only falsey values, `0` is true!
- Each object natively understands equality.
- *Symbols* are prefixed with `:identifier`. Identical symbols point to the same physical object, unlike identical objects, can tell by checking their `:identifier.object_id`.
- Arrays are Ruby's primary ordered collection (Ruby 1.9 has ordered hashes).
  - Out of bounds yields `nil`.
  - Negative counts backwards.
  - `arr[0..1]` returns a slice, since `0..1` is a `Range`.
  - `[]` is a function on `Array`.
  - No need to be homogeneous types.
  - Implement queue, LL, stack, se etc.
- Hashes are labeled collections, key-value pairs.
- *Code blocks*
  - Code blocks are unnamed functions, between braces or `do/end`, former when single line, latter when multiple lines.
  - Can be passed as function argument, prototype says `&block` and can invoke with `block.call`.
  - `yield` calls whatever block is passed to the function.
  - Can be used for delaying execution and conditional execution as well.
- OOP
  - `initialize` constructor
  - Class names are camel cased, instance variables and method names are snake cased, constants all caps.
  - Instance variables are prepended with a single , class variables with two .
  - `modules` to solve multiple inheritance, collection of functions and constants, `include`ed by `class`es.
  - `modules` can call functions it does not define but expect `include`ees to define, duck typing! Implicit "abstract functions" from Java.
- *Metaprogramming* is writing programs that write programs.

- *Open Classes* allow us to modify existing classes in-line, even built-ins like `NilClass`.
    - A fun use case is to override the `self.method_missing` function, which is called whenever an attribute is not found. Then, a class called `Roman` can have attributes like `Roman.XII` and use `method_missing` to compute the value! Wow! ☺.
- `Modules` are extremely adept at metaprogramming, since a modulee's `included` method is called whenever it is included, so it can metaprogram on inclusion.
- Core strengths
    - Duck typed with OOP is out-of-the-box polymorphism.
    - Fast for scripting, well-supported for various extensions.
    - Rails!! Fast time to market.
- Weaknesses
    - Performance: getting much faster, but still slow. Metaprogramming makes any compilation nigh impossible. Also against the core design philosophy of programmer's experience vs performance.
    - Concurrency is hard with OOP.
    - No type safety.

3 Io

- Prototype language like Lua and Javascript, no distinction between objects and classes.
- Everything is a message that returnss another receiver. Program by chaining messages, e.g. `"Hello World" print`. Message passing is a strong concurrency model.
- Objects and classes are the same, create new objects by cloning existing ones e.g. `Vehicle := Object clone`.
    - Inheritance is equivalent to sending the `clone` message to a parent prototype.
- Objects have "slots", and a collection of slots is like a hash. Objects are basically collections of slots. Can `Object slotNames` to get list of slots.
- When a slot is not found on an object, it is forwarded up to parent prototypes or until not found.
- Lowercase clones do not override parent's `type` slot.
- *Methods* are objects with `type Block`. Can be attached to object slots, are invoked when the slot is invoked.
- `Lobby` is an object with a slot for each name in the global namespace.
- Lists `list()` are the prototype for all ordered collections, and Maps `map()` are the prototype for all key value stores.
- `true, false, nil` are *singletons*, i.e. their `clone` returns themselves rather than a clone of them! Lots of cool tricks by overriding core functionality like this.
- Can see list of operators directly with precedence by `OperatorTable` and create new operators. Use case: short JSON → `Map` parser.
- Message reflection is possible with the `call` operator inside method bodies, e.g. `call message arguments`.

  - The reason message reflection works is because the full message context (sender, target, message) are all pushed onto the execution stack.
  - In Io, messages passed as arguments to a method are only pushed onto the stack and *not evaluated*.
  - This means that a receiver can call `call sender *` and hit an arbitrary sender slot.
- Can override `forward` message slot same way as `method_missing` before.
- Concurrency
  - *Coroutines* are functions w/ multiple entry/exits. Firing a message with returns a future, with two returns `nil` and kicks off a new thread.
  - `yield` yields control inside a coroutine.
  - *Actors* place incoming messages on a queue and dequeue with coroutines. An object becomes an actor when sent an asynchronous (`,`) message.
  - *Futures* return immediately, but when accessed block until the asynchronous result is returned.
- Strengths
  - Tiny footprint, heavily used for embedded systems.
  - Compact syntax, fast rampup.
  - Flexibility because all slots and operators are exposed.
- Weaknesses
  - Minimal syntactic sugar.
  - Slow single-threaded execution speed.

Illustrative example of reflection, to print slots of ancestors of any object that clones `Object`:

```
Object ancestors := method(
    prototype := self proto
    if(prototype != Object,
        writeln("Slots of ", prototype type)
        prototype slotNames forEach(name, writeln(slotName))
        writeln
        prototype ancestors
    )
)
```