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.

Statically Typed Infer polymorphism by relation to other types rather than just "is this function defined", checked at compile time.

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.

Declarative Language "Throw facts and inferences at the language and it will reason for you."

Purely Functional A function invoked with the same inputs always produces the same output. *Alias: stateless.*

Higher-order Functions Functions that map functions to functions.

Threads vs. Processes Both have their own execution path but threads share resources in exchange for being more lightweight.

Atoms Symbols that represent singular values, like entries in an enumerated type.

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, developed in 2002.
- 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
    )
)
```

4 Prolog

- Declarative, from 1972.
- First letter capitalization says whether an identifier is an *atom*, fixed value like a Ruby symbol and lower cased, or *variable* whose values can change and are upper cased or start with underscore.
- Three building blocks, facts, rules and queries. Facts and rules go into a *knowledge base*, the Prolog compiler serializes facts and rules to be query efficient.

  Facts `likes(person, object)` declares a fact.

  Rules `friend(X,Y):-\+(X=Y), likes(X,Z),likes(Y,Z)` is an example of a rule.

  - `:-` is a *subgoal*. Rules can declare multiple subgoals, only one of which needs be fulfilled.
  - `\+` logical negation, $X \neq Y$.
  - `friend(X,Y)` means $X \neq Y$ but both X, Y like Z!

  Queries Run queries after loading a knowledge base, e.g. `likes(person, java).` Same syntax!
- Can specify *variables* in query to query for *all possible values* rather than just `yes/no` above, e.g. `likes(Who, java).`
  - Seems like responses are pagined, with `;` to advance and `a` to get all of them.
- Core ideas of logic engine:
  - Unification assigns two structures to be identical, `X = animal`.
  - Prolog takes a query and unifies any variables in the query with any variables in the corresponding rule definition. The right hand side of the rule then becomes a list of *goals* which are DFS'd.
- Recursive rules are powerful but should tail recurse, recursive subgoal should be last.
- Prolog also has lists `[]` and tuples `()` which unify element by element. Lists can be deconstructed with `[Head|Tail]`.
- Can unify with `_` wildcard, discards.
- Use `Variable is Expression` to unify `Variable` against `Expression`, e.g. `Count is TailCount + 1`.
- Amazing to code up solutions to classic puzzles, many minds were boggled reading through the book exmaples.

5 Scala

- Unifies OOP and functional. Statically typed, not purely functional. Compiled, the interpreter is actually a compiler + executor all in one. XML is a first-class programming construct!
- Key differences from Java:

  Type Inference Statically typed but can infer types.

  Immutable Variables Taking `final` to the next level.

Functional Programming lol.

Higher-level Abstractions e.g. actors in concurrency.

- Type inference e.g. `4 + "abc" = "4abc"`, though `val a: Int = 1` can declare explicitly.
- `val` is immutable, `var` is not to declare.
- Supports first-class `Ranges` (`0 until 10`) and `Tuples` (fixed length, can do multivalue assignments).
- OOP features look mostly like Java with some small syntactic sugar (seems the `varName: Type` annotation is the more popular one now, as Scala uses it too).
- `object` defines singleton classes, vs `class` for instance templates. An `object`'s functions are like `static` functions in vanilla Java, and since you can name an `object` and `class` the same name, the *companion object* approach lets you put `static` Java functions in the `object` and the instance functions in the `class`.
- `traits` are like Java `Interfaces` plus an implementation. `implements` becomes `with` in Scala.
- Moving onto more functional features, can define naked functions too.
- Also have `Lists`, can mix types (to JVM, list of `Anys`). List access is done with `()`.since it is a function. Also, native `Maps`.
- `Sets` can be unioned, differenced and intersected with `++,--,\item`.
- Equality works by value for lists and sets!
- All three of these collections can be iterated in various more functional ways e.g. `foreach, fold`.
- Can assign XML to variables, query, iterate, pattern match! Supports XPath syntax for querying. Apparently useful since Java is XML heavy.
- Pattern matching both looks a lot like switch case but also hearkens to various pattern matching we've seen in Prolog and will see in other functional programming languages!
- Concurrency done with actors and message passing, send message using `!` e.g. `actor ! message`. `Actor` is a parent class to `extends`.
  - `loop` to loop while waiting for message
  - `react { case ... }` for pattern-matched message-passing handling, or `receive` equivalently.

Pattern matching + XML example

```
val movies = <movies>
    <movie>a</movie>
    <film>b</film>
</movies>
(movies \ "_").foreach { movie =>
    movie match {
        case <movie>{movieName></movie> => println(movieName)
        case <film>{filmName></film> => println(filmName + " (film)")
```

```
        case _ => println(" error")
    }
}
```

6 Erlang

Oh boy *Few languages have the mystique of Erlang, the concurrency language that makes hard things easy and easy things hard.*

- Near real-time, fault-tolerant distributed applications, hot swappable. Functional, language-level message passing. Compiled but has interpreter. Dynamically typed.
  - Much syntax comes from Prolog, was implemented over Prolog at first!
- Erlang goes for processes instead of threads, tries to keep them lightweight. Eliminates need for locks.
- Actors are processes, pattern match off a queue of messages to decide how to process.
- Erlang handles reliability by "let it crash" mantra, then spinning processes back up. Lightweight processes important!
- Periods terminate lines wow ☺.
- `String`s are just `List`s, i.e. an array of ASCII codes prints as a string in `erl`.
- Basic type coercion but no string $\leftrightarrow$ ints.
- Lowercase symbols are atoms, not even assignable and only used in pattern matching, uppercase are variables, assignable but immutable..
- Tuples `{}` and lists `[]` exist, former is fixed length is all.
- Maps are generally implemented as list/tuple of tuples, and to access we pattern match!
  - Convention is to use the first entry in list as atom corresponding to "type" of object, so it's easy to guarantee pattern matching matches the right object type/attributes.
- *Bit matching* lets you pack data into effectively structs w/ predefined bit lengths for each attribute, e.g. `<<A:3, B:5, C:5, D:5>>` is a struct w/ 16 total bits, and can unpack via pattern matching too.
- `module(name)`s `export([function/numArgs])` in a file, where we use shorthand `f/1` to say function `f` takes one arg. Namespaced with `module:function`.
  - `mirror(Obj) -> Obj.` is valid syntax in Erlang. Note differs from before since we don't specify type of `Obj` but the return type of `mirror` is obvious by its argument type. Dynamic typing!
- Erlang also tail recurses, but does not overflow on 2000! and is instantaneous apparently according to book example!
- Tend to use `case` statements to parse message passing, common `_ ->` wildcard match.
  - Semicolon after each case except for last, so no `comma-dangle` eslint rule.

  - Commas separate lines within each case

- `if` statements use *guards*, so kinda pattern match on condition. Can have more than two guards in an if statement.
  - Multiple guards can match, ordering is important.
- `fun(args)` defines an anonymous function of `args`, can assign to a Variable.
  - Usual applications, `lists:foldl`, `lists:map`, `lists:filter` and more under the `lists` module
- Build lists using `[|]` head/tail notation on the RHS of an assignment.
- List comprehensions with `[op(X) || <clauses>]` where `clauses` can be generators for multiple variables and/or constraints on these variables.
- Three basic operations, `!` sending message, `spawn`ing a process and `receive`ing a message.
  - Similar to `loop` syntax of Ruby, the canonical way to "infinite loop" in Erlang is to have a function `loop()` call itself via tail recursion.
  - We then `spawn(loop/0)` a process that runs a function, in this case the infinite loop `loop()` above. Returns a `Pid`.
  - Then we `! message` to the `Pid`. When no explicit return is specified by an actor (or if the actor is dead), returns the message itself.
  - To achieve synchronous message passing, the receiver can take a `Pid` and message pass the return back, and the sender can send `self()` and then `receive` the returned message and `->` return it.
- Erlang does provide checked exceptions, but instead *let it fail* and handle its failure by linking processes together and handling error cases.
  - `process_flag(trap_exit, true)` registers the current process as *the* process one traps exits.
  - Receives `{'EXIT', From, Reason }` provided by registering under `process_flag`.
  - Can also receive another message that takes a `Process` and `link(Process)`es it to receive exit messages. Can use `spawn_link` to spawn and link immediately.
  - Then, if in the `EXIT` message, the exit trapper `spawn_link`s another process, then we have built in error handling!
- Strengths
  - Dynamically typed yet the most reliable core libraries.
  - Lightweight processes, isolation but easy to let die and spin up.
  - Open Telecom Platform (OTP) enterprise libraries are vast, owing to Erlang's original roots in a telecom company.
  - Let it crash very easy to reason about concurrency.
- Weaknesses
  - Syntax: everybody hates Erlang syntax ☹.

7 Clojure

- Lisp on the JVM. Dynamically, weakly typed. Lisp is:
  - Language of lists: prefix notation is just a list w/ operator first.
  - *Data as code* data structures express code.
- Concurrency done via agents. Rather than being fully immutable, transactional memory.
- Atoms/symbols in clojure are preceeded with `:`.
- Lists, maps, sets and vectors.
  - Lists: `()`. `first`, `last`, `rest`. `cons` appends.
  - Vectors/arrays: `[]`. `nth` so can random access.
  - Sets: `#{}`. Can `sorted-set`. Are also functions (!) that test for membership!
  - Maps: `{}`. Every other entry is a key, others are values. Traditionally commas every two elements since they're just whitespace. Also functions, do lookup. Can do both `(:key map)`, `(map :key)`.
- Use `def` to bind a value to a variable.
- `defn` to bind a function, `(defn <docstr> func [params] body)`, optional docstring accessible via `(doc func)`.
- Can destructure e.g. the following are equivalent: `(defn f [l] (last l))` and `(defn f [[_ l]] l)`
- Can declare anonymous function with just `(fn [params] body)`. If e.g. mapping, can even syntactic sugar to `(map #(* 2 \%) nums)` where `\%` refers to each item mapped over.
- Clojure does not tail recurse automagically. Instead, use `recur` to re-enter a `loop` to effect tail recursion.
- *Sequences* are an abstraction library over lists, vectors, sets, even I/O.
  - Syntactic sugar for iterating through a list: `(for [x colors] ...)` is like "for x in colors."
  - Lazy sequences! `take` to take a finite number of an infinite sequence.
- OO constructions
  - `defprotocol` defines a Java interface, functions that records implementing the `protocol` must implement.
  - `defrecord` lets you specify what `protocol` you're implementing.
- `macro`s
  - The problem is that w/ many functional languages, we evaluate args then push them onto the call stack. Possible case we do not want to do this is the `unless` keyword from before; do not evaluate function body.
  - Solution is to `defmacro`, treat even executable code as a list and do not execute, push directly onto call stack instead. Powerful b/c data is all code!
- Concurrency is solved via *software transactional memory*.

  - Most operations so far have been immutable, get *references* via `ref` to a variable, then can `alter` it inside a `dosync` transaction, dereference with `@ref` or full form `deref`.
  - Can also `ref-set` instead of operating on the existing value.
  - Can use `atom`s to allow change changing w/o transaction.
  - `agent`s are also wrapped pieces of data that get mutated asynchronously via `send`. To get the value of a reference after an asynchronous message has been processed, `await`.
  - Can also use `future` to block until a value is available.
- Can implement multimethods depending on object metadata (which can be attached to symbols and collections).
- Clojure is a great lisp b/c fewer parens, ecosystem b/c Java, better concurrency, lazy evaluation. It is pretty inaccessible though, being a Lisp.

8 Haskell

I already know a bit of Haskell so I might be a bit terse here, plus I need to finish this book now.

- Absolutely pure functional, born out of functional programming language conference in 1990 and revised in 1998. Strong, static typing, widely considered the most effective type system.
- Few primitive types, numbers, strings, bools. No type coercion, very strong typing!
- Functions can be dynamically typed, allow type variables.
- Same pattern matching as before, takes first of matches. Can use guards `| expr = func` to guard expr to only execute when func.
- Same destructuring syntax as Clojure for the most part, except destructure list using `h:t` rather than `h|t`.
- Lists are homogeneous here!
- Compose functions using `f1.f2`, concat lists using `:` operator.
- List comprehension like Erlang, e.g. `[x*2|x<-[1,2,3]]`.
- Define anonymous functions with `\x -> x`.
- Higher order functions map, filter, fold etc. many many!
- Cool concepts:
  - Currying
  - Lazy evaluation, infinite lists, `take`
- Haskell type system:
  - Define types `data Boolean = True | False`.

- To know how to display, you can `deriving (Show)`.
- Polymorphic functions by typing a function w/ type variables.
- `data` definitions can also be polymorphic.
- `class` defines function typings for people that derive it and can have implementations for these functions for derivees.

- Monads
  - Monads are meant to simulate program state.
  - Consist of a type constructor to hold a value, a `return` function that wraps a function into a monad and a `>>=` to unwrap a function in a monad.
  - `monad >>= return = monad`.
  - Example monad `Position`:

```
1 module Main where
2     data Position t = Position t deriving (Show)
3
4     move (Position d) = Position (d + 2)
5     rtn x = x
6     x >>== f = f x
```

    then the advantage of using a monad for this is that instead of doing `(move (move (move position)))` where the innermost one actually moves first, we can write them in executed order instead!
  - The binding reads "for a given monad `x`, bind the application of function `f` to apply `f x`, so an identity monad.
  - `do` + `return` makes function declarations look imperative but actually uses monads.