Programming Language are Skins

Languages are just *skins* for paradigms.

Each language should have (IMO) two syntaxes – one for writing and one for reading. The differences can be subtle and are influenced by the biases of the language creator.

One extreme is Lisp – *readable* – after successful compilation.
The other extreme is Pascal – *writable.*

Important paradigms:

- search with backtracking – examples: PROLOG, miniKanren, Nils Holm's PROLOG in 6 Slides
- pipelines – isolation
- Functional – Haskell, Clojure, early Lisp, CPS (supported by Lisp and JS)
- assembler – Lisp, JS – possible to program in any paradigm without being forced in one direction or another
- OO (Object Oriented) – Smalltalk is the classic example, a form of code editing that emphasizes DRY – copying super-classes into final code results in no semantic differences, just DRY ; could be replaced by a smarter editor or edit script (see Paul Bassett's Frames technology)
- ownership – FBP, (flow-based programming), Rust
- early typing – strongly typed languages like Haskell
- late typing – early Lisp (before Common Lisp), interpreted Common Lisp, BASIC, etc.
- coordination – FBP, Linda, Bash, closures, cl-event-passing, etc.
- concurrent – concurrency is a paradigm, concurrency is a prerequisite for parallelism
- looping, recursion – not suitable for distributed apps, suitable for component apps
- sequential – very common. CALL / RETURN, not suitable for distributed apps
- asynchronous - operating systems (e.g. Linux, MacOSX, Windows), closures with a dispatcher
- time-sharing – needed by only a few apps (e.g. operating systems), but supplied in almost every threading library (=accidental complexity, =overkill)
- multiple-stage execution (compiler --> linker --> executable)
- single-stage execution (interpreter)
- waterfall development – assumption that "it" will work, rigor = waterfall, strong typing = waterfall (assumption that the given type system will be what is needed)
- iterative development – assumption that "it" will fail, assume that you don't know the answer and need to develop the answer. This leads to the conclusion that everything must be automated – it won't work the first time and will need to be tweaked ; automation costs "up front", but, saves time when tweaking. This leads to the concept of D.I. (Design Intent) – succinctly say what was intended, then automate the solution into a DSL – it won't work the first time, so you need to be reminded of what you *thought* you were doing and you need to fix that thinking as rapidly as possible. When is it "good enough"? If you don't know what you were doing, then using rigor to implement the wrong solution leads to too many details and long rework times. If the Intention is wrong, it doesn't matter how rigorously (provably) it is implemented. Rigor is useful only after the Design has been proven to solve the problem that it was supposed to solve.
- Abstraction – DRY – a good edit script can accomplish all DRY
- Abstraction – subroutines

- Abstraction – modules for data-hiding – C already had this in 1970's ; see closures
- Abstraction – memory management – Garbage Collection – already existed in early Lisps (late 1950's)
- Abstraction – assign-once variables – why do we need variables and variable names, if they are assigned only once? Typenames should be enough esp. when type sysnonyms exist.
- Abstraction – types – why are objects of different types put on the same stack?
- Abstraction – syntax – why only 1 syntax per language? Why is syntax restricted to rigid grids of NxM bitmaps (aka "characters")? Why not overlap? Why not diagrammatic figures?
- Abstraction – VLSI chips with limited number of I/O pins.
- Encapsulation – not enough – encapsulates data, but smears control-flow
- Isolation – UNIX processes (encapsulate data AND control flow)
- Error Reporting – good == Elm, hostile == JS