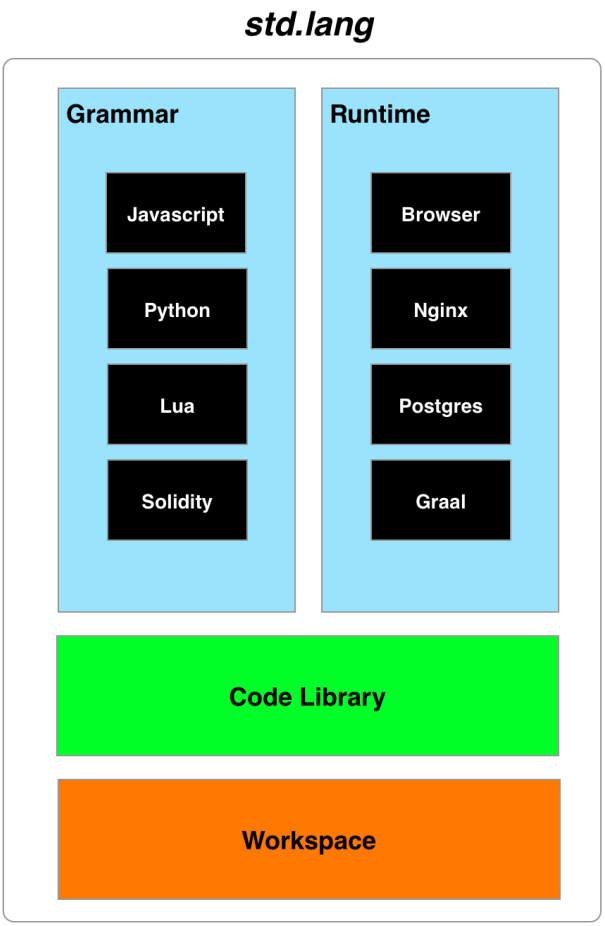
The Universal Transpiler

1 - Introduction

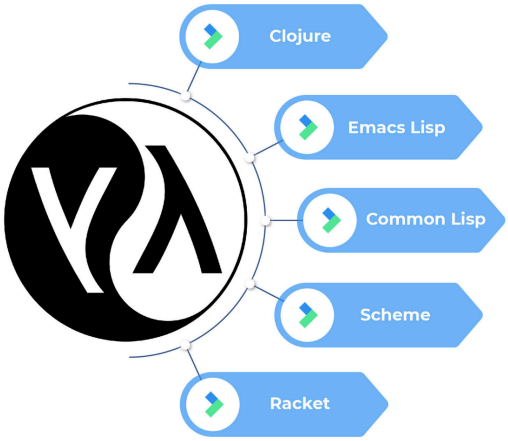
What *std.lang* is solving is a translation problem. There are 4 individuals in a room, each speaking a different language. If each person needs to communicate to another, they can either learn a new language (which takes time) or they can go through an interpreter. *std.lang* is that interpreter. Put simply, it creates a space to allow better interoperability with any runtime, running any language. Note that in this document *std.lang* and universal transpiler refers to the same technology.

- Section 1.5 presents the key points outlined in the introduction.
- Section 2 describes the architecture and design of the transpiler.
- Section 3 provides current case studies of transpiler code used in the wild.
- Section 4 provides additional use cases as well as future directions.



It is hoped that the rest of the introduction will welcome the reader into the world of LISP and the give an overview of the current state of the art in programming language research. We provide an answer for the question: “What is LISP and why is it different to all the other languages?” as well as how the *std.lang* fits into the overall picture.

1.1 - To LISP or Not to LISP



Within the LISP world, there tends to be a Jedi-like zeal as to how one should communicate their love of LISP, whether it may be the grand orders of Scheme, Common Lisp, Racket, Clojure or Emacs. Each LISP order has their own ethos, their way of uplifting members and their way to internalise why their order is the best. Generally though, there is no need to sell. Members have already brought in and are happy content citizens. For the author, it was and still is Planet Clojure, watched over benevolently by Rich Hickey (Clojure's Creator) and the Knights of Nubank, nee Cognitect.

Trying to explain or induct an outsider however is a slightly different experience to the "LISP gives you Superpowers" mantra that gets internally recycled every 1 or 2 years. Outreach feels eerily similar to the experience of trying to convince a work colleague on the merits for microdosing: "Trust me bro, you need to try it out. It's going to make your life just much better. If you need proof, look at me - I am SOO creative right now. The benefits come only if you buy in but it has to your decision and your decision alone. Also, read Paul Graham's essays. He's rich so you can 100% believe him for sure."

There is great pride to being a LISPer and even more so if one can be of the tool builder variety. One might ask - How rare are these tool builders? To put it in Javascript terms, it would be the number of developers who can maintain a library such as Babel or Webpack (build tools that developers use on an everyday basis). In general, the percentage of tool builders is extremely small in any language ecosystem. There are generally a greater percentage of tool builders wondering about the LISP communities. Due to the small size of the developer base and the lack of help these builders create grand, all-encompassing architectures that will remain long after they leave. It reminds one of the exquisite 18th century English buildings still operating in Calcutta, barely maintained, slowly decaying and a sight to behold. It is much easier for some to build than it is to communicate to others what one is building.

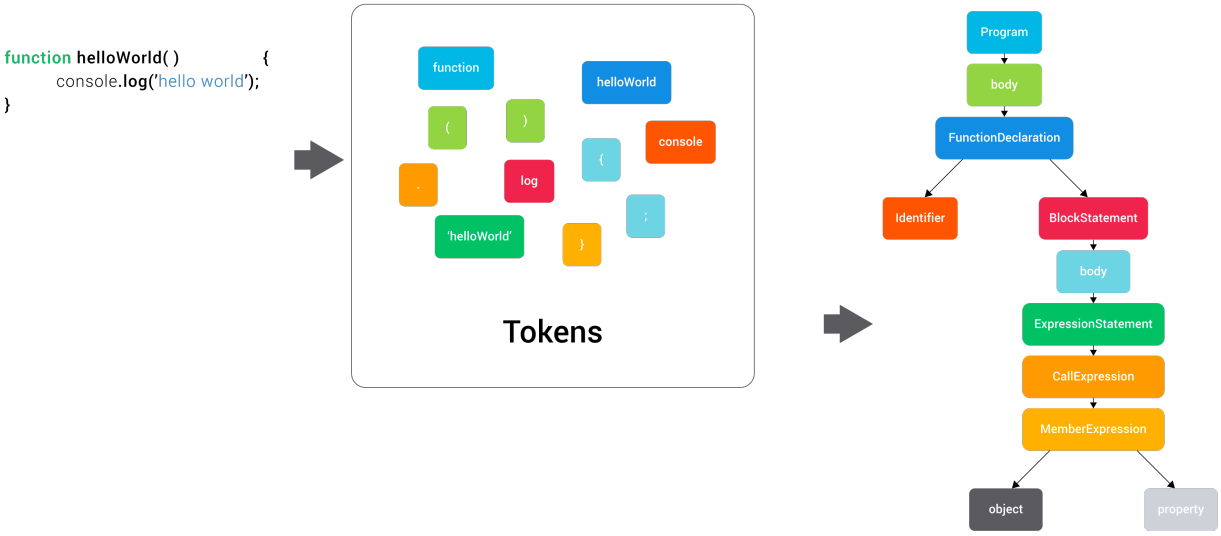
Another reason why LISP has not caught on is because of industry. Industry requires articulate, readable code. LISP, with its infinite amount of possibilities run counter to the process driven, "developers are replaceable" attitude that the HR departments promote. There is huge project and personnel risk if a team goes with an exotic language that everyone has to learn. For a lot of industries, this is an unacceptable level of risk. However, the dismissive attitude by the status quo tends to be worn as mark of pride within the programming counter culture which celebrates individual brilliance and creativity. It pays to place one self at the center the pack, but once in a while, the ones at the front will discover something so special that the center cannot help but be pulled towards them. This is every innovator's dream.

1.2 - LISP Demystified

When talking about categories of high-level programming languages, there are only two - ALGOL and LISP. Every language can be classified as either one or the other. However, it is a misnomer that the two categories are completely unrelated - we need to peak under the hood to get to the full story. In doing so, we invoke Zcaudate's Lemma, which stipulates that:

"Every ALGOL can be also represented as a LISP".

ALGOL languages are transformed by the compiler into an intermediate representation called an AST or Abstract Syntax Tree. This representation has exact equivalence to LISP code. Downstream in the compiler process, the AST then gets turned into either byte code or machine code through the compilation process and additional transforms. The downstream conversion to executable code is the same for LISP and ALGOL and the entire set of code transformations can be grouped as 'Compiler Technology'.



Without understanding the AST and how it transforms, languages cannot be created and therefore every language creator can be considered a LISP programmer of the tool builder variety. Not that it's a secret or anything - James Gosling of Java is a Lisper, Brendan Eich of Javascript is a Lisper. Yukihiro Matsumoto of Ruby is a Lisper, Guido Van Rossum of Python is a Crypto Lisper, Roberto Leumocheschy of Lua is a Scheme Affectionado, Prolog, OCAML and Julia all started off as LISP projects. Might I go on? For the conspiratorially minded, it may seem self evident that there is a Cabal of High Level LISP Initiates around the world controlling language and dictating as to how all programmers will be writing code for the next 1000 years. That was a Haskell joke.

Within any programming language, it is the keywords and syntax that capture a pattern of computation for a programmer to express. As one builds a Language, more and more of these patterns can be introduced for the consuming programmer to use and integrate into their programming toolkit. What we are really talking here about is pure expression. Being closer to the machine, a programmer using LISP can create and manipulate these patterns faster than than they can using ALGOL languages. ALGOL languages aim for readability and user adoption whilst LISPs aims for feature adoption. Therefore LISPs generally leads at the forefront of programming paradigms shifts such as Language and Compiler research, the first generation Artificial Intelligence as well as Immutability and Concurrent Systems research.

In the evolution of a programming language, LISP is the primordial ooze. It's just the AST. Languages can spring out of that ooze and become fish, or birds, or mammals. They will look back at LISP and say - that's a lot of parens, macros and useless muck they have back there. LISP looks at those fish, birds, mammals and snuggly say to themselves "these languages will never evolve ever again".

1.3 - The Runtime Menace

Now that the relationship between LISP and ALGOL have been qualified, we can talk about runtimes - the engines where the code is run. Code can only be expressed through runtimes. Runtimes therefore are the gateways between the programmer and the end consumer. Runtimes are underrated the way languages are overrated. It is the runtimes that popularises a language, not the other way around - Javascript is what it is today because of the browser whilst Java is what it is because of the JVM.

In the battle for mindshare, more and more exotic runtimes are being released on the market by the consumer-facing tech organisations both large and small. Runtimes attract consumers by promising speed, low cost of purchase and a variety of options. Runtimes attract programmers by promising speed, ease of development, low cost of entry and access to the end consumer.

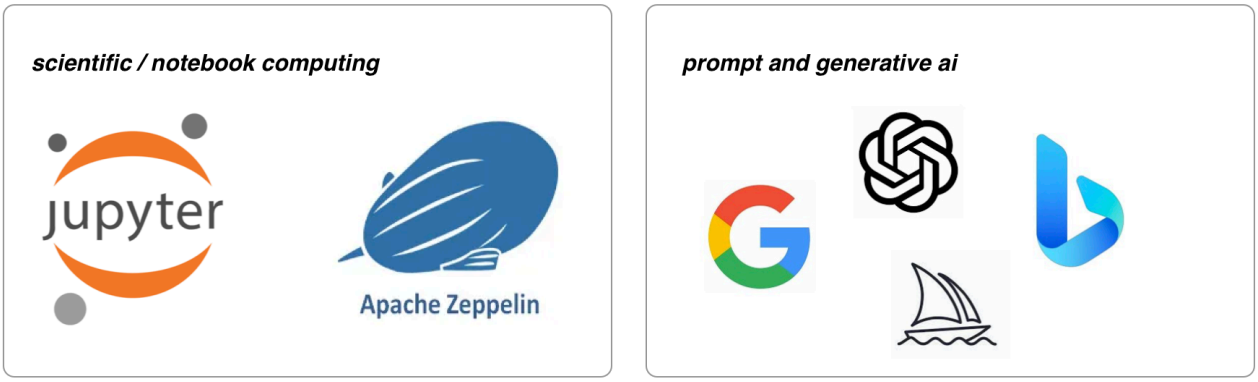
Runtimes come in different shapes and sizes. Most are general purpose while others are quite specialised. Runtimes tend also to expose functionality in the form of libraries as well as a language interface. Each runtime tend to come with their own set of tooling for deploying code as well as testing and documenting code. Some examples of runtimes, listed with their function and language are:

- Nodejs (General Purpose Backend, Javascript)
- Chrome Browser (Graphical Display, Javascript)
- World of Warcraft (MMORPG Game, Lua)
- Blender (Graphical Editor, Python)
- NeoVim (Text Editor, Lua)
- OpenCL (GPU Kernel, C)
- Polygon Chain (Blockchain, Solidity)
- Postgres (Database, Plpgsql)
- Redis (Cache, Lua)

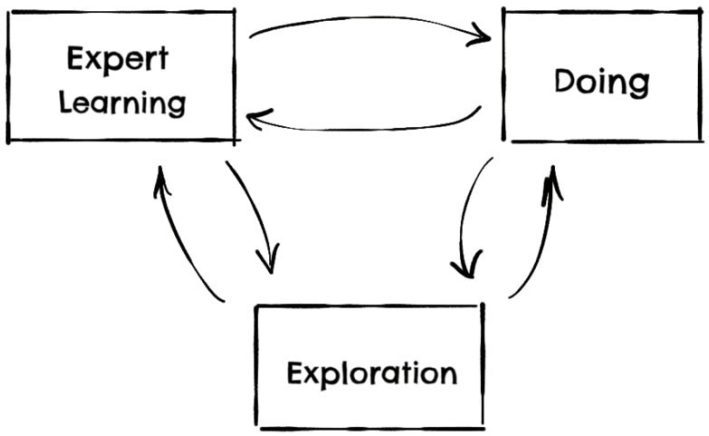
Additionally, apart from the software-based, containerisable runtimes, there are embedded runtimes for both industrial and consumer based applications. With the introduction of Apple Vision and Tesla OS, it seems that programming will become more and more specialised to industry verticals. With such developments, open standards and cross platform tooling will become ever more are important as they form the connectors to an increasing choice of runtime substratum.

1.4 Literate Programming and AI

There are many ideas of what Literate Programming is - from generated doc strings to code walkthroughs to whole encompassing systems like Org Mode on Emacs that makes documentation first and code second. One key area for literate programming is in Scientific computing and notebook style programming such as Jupyter allow a broader user base accessibility to AI and Machine Learning through a more gentle interface. The raise of ChatGPT and generative AI is another. Asking a prompt to 'write a program' was unthinkable just 5 years ago and the possibilities of development tooling 5 years from now just beggars belief.



From a programmer's point of view, literate programming is about feedback. It is about being able to understand the context about a piece of code - how it is used, where it is used, what arguments it takes and what it outputs for a set of sample input. There should be clear workflow loops, enabling devs to shape a project the way that a potter might shape clay through incremental steps. In most software projects, the developer/team will need to go through the stages of not knowing to knowing and pioneering teams generally rely on trial and error to get to the optimal solution. When there are ways to fail fast, better programs are created.



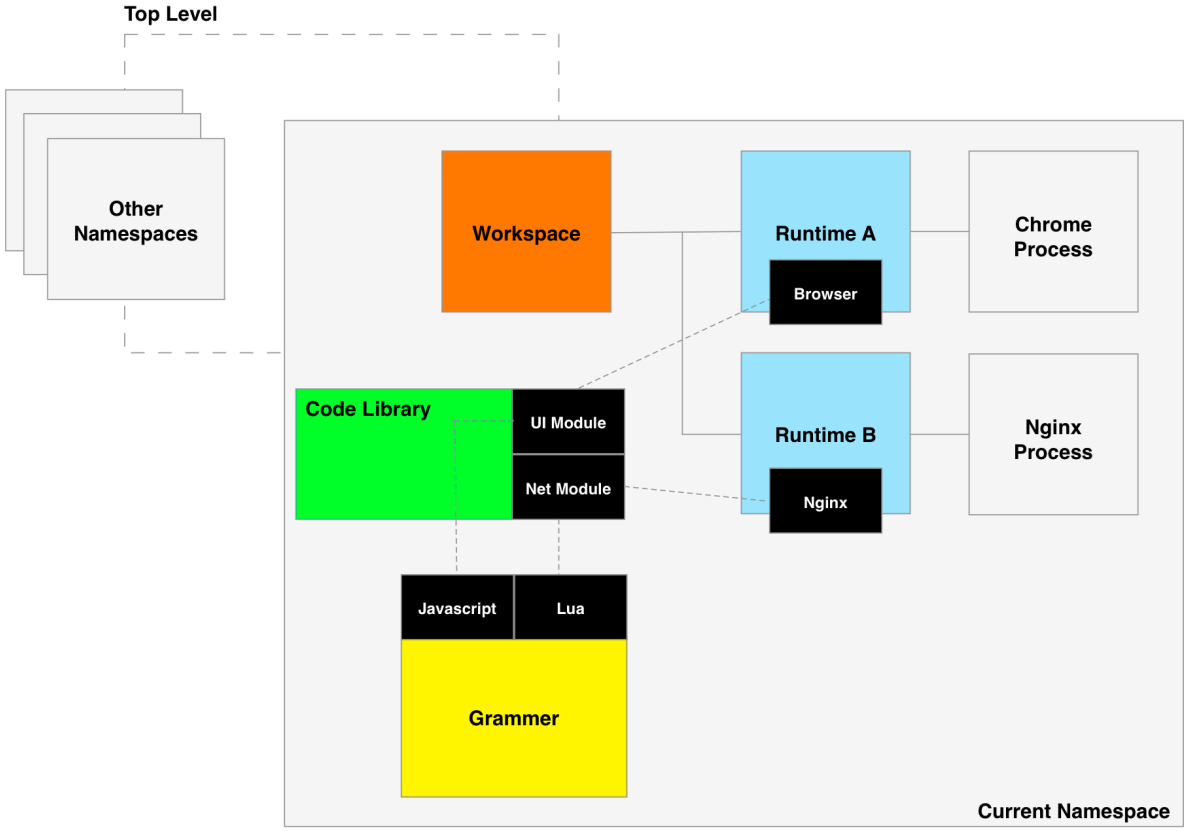
LISP environments provide immediate feedback within a live executable environment. Code blocks can be run directly within the editor and many things such as documentation are accessible with a couple of shortcut keys. A LISP the programmer lives in the runtime that they are coding for and this is a fundamentally different experience from developing a program using an IDE.

1.5 Summary of Ideas

Having introduced the themes governing the context around language and programming research, Let's revisit the big ideas that have been previously presented and condense them as points in context for discussion:

- All languages can be represented as LISP (Zcaudate's Lemma)
- Languages have very similar feature sets, especially the general purpose ones.
- Even if two runtimes share the same language, they might still be functionally very different
- Each runtime may have their own way of testing/documenting/executing code and generally does not provide methods to communicate with other runtimes
- Literate programming is very important in the management of a project's lifecycle.
- LISP environments are live and provide fast feedback, greatly enhancing knowledge gathering and implementation

std.lang solves the problem of interacting with one or more runtimes using a common framework to define and store code as standard LISP representation. Custom connectors to runtimes created as plugins as well as custom language grammars enable code of multiple languages and multiple runtimes to run and interact seamlessly over a common LISP base environment.



All code have access to a common test/documentation infrastructure on the native (Clojure) runtime for a faster and more integrated development experience and the runtime connectors developed for embedded systems allow for unprecedented ease for rapid application development on platforms that would be otherwise painfully difficult to target.

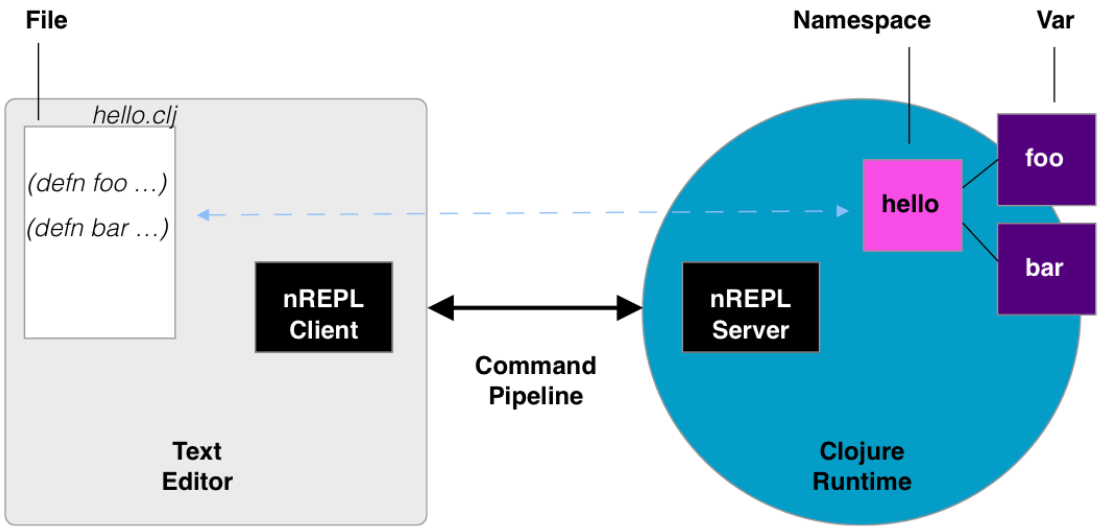
2 - The Universal Transpiler

2.1 - Introduction to Transpilers

What is a Transpiler? It is a mechanism to convert one type of code to another. There are many transpilers in the wild - both open source and commercial. With the development of the Web, transpiling languages to C and WASM have become extremely popular. There are great benefits for being able to transpile code - it means that once transpiled, tooling of transpiled language can be used - such as optimising compilers or third party libraries.

2.2 - The Clojure Native Runtime

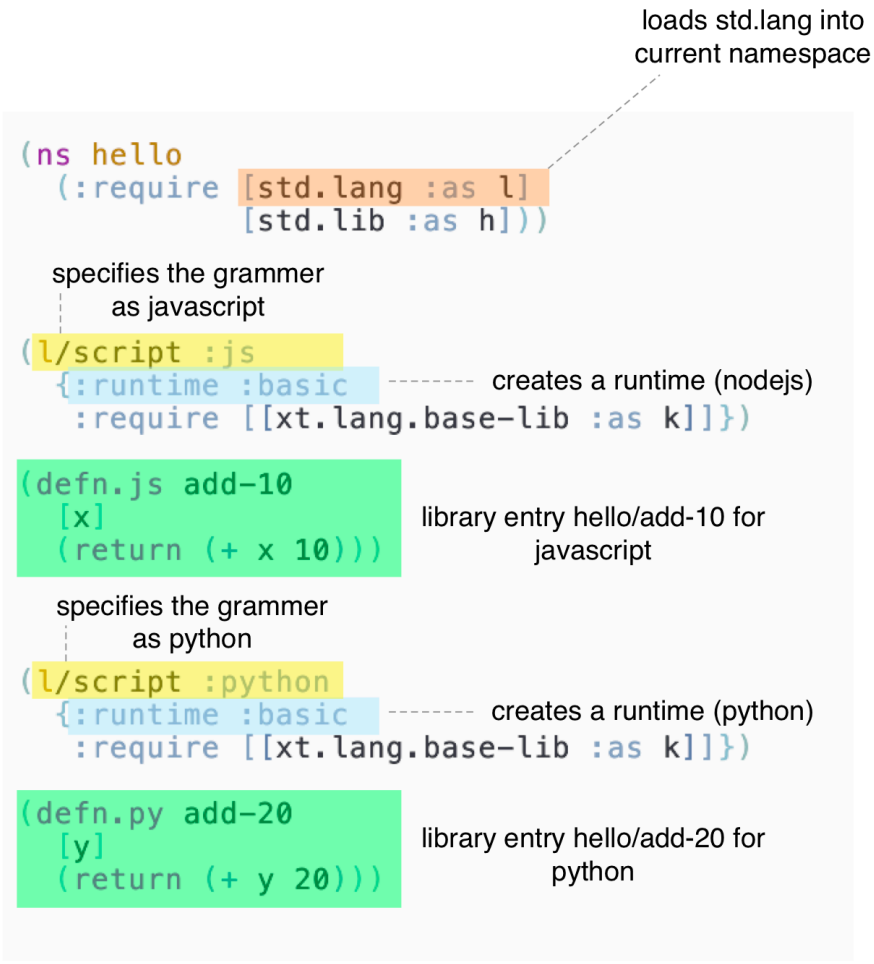
The clojure runtime is a mutable environment, with code arranged in a two-axis grouping by *namespace* and *var* name. A namespace corresponding to a file on the local disk and vars correspond to globally referable entries where code reside.



When writing Clojure code, the typical workflow involves booting up a runtime and connecting an editor. The runtime sets up the programming environment and creates an *nrepl* server, allowing any connected client to send commands to the runtime for processing. After the nrepl connection is established and connection made between the runtime and the editor, the entire setup is deemed to be “live”. That code in the editor can be interactively evaluated via the runtime, with messages being sent using the *nrepl* connection.

2.3 - Design and Scope

std.lang piggybacks onto features of the underlying clojure runtime in order to facilitate the transpile and evaluation pipeline for it's code and runtime plugins



The library speeds up development in the following ways:

- there is a base template for describing common operations as well as blocks and loops
- runtimes can be started and stopped, providing a mechanism for complete runtime isolation
- runtimes can be changed on the fly, so that code for a given language can be tested across various runtime configurations
- runtimes can be combined, such as in the event that multiple runtimes are needed to work together

2.4 - Live Execution Architecture

```
(ns hello
  (:require [std.lang :as l]
            [std.lib :as h]))


(l/script :js
  {:runtime :basic
   :require [[xt.lang.base-lib :as k]]})

(defn.js add-10
  [x]
  (return (+ x 10)))
```

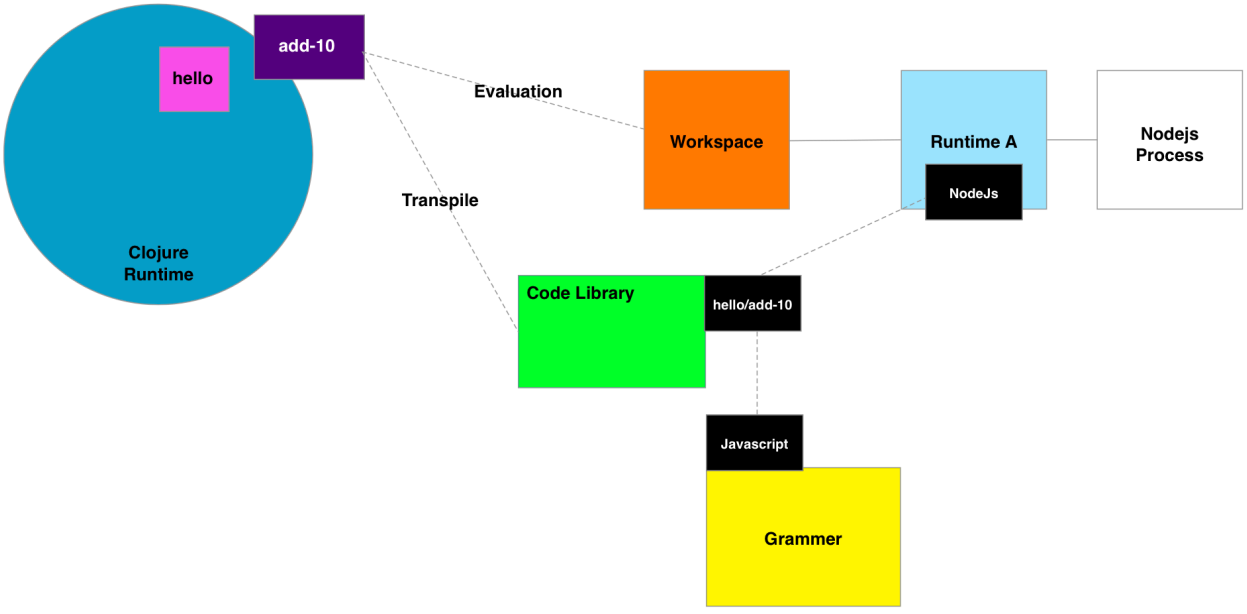
pluggable runtime
(can start and stop)

1. Creates a code library entry for javascript at *hello/add-10*

2. Creates a native var at *hello/add-10* as a *std.lang* pointer to the entry created in Step 1.

The secret to live execution is determined by how *std.lang* interprets the top level forms - *defn.js, defn.py* other such macros to define code. When these forms are evaluated by the compiler, it stores the LISP forms into a *std.lang* library entry and at the same time, creates a *std.lang* pointer to the entry and assigns it to a Clojure *Var*.

The pointer behaves differently depending on which namespace it is being invoked in. For example, if invoked in a namespace without a runtime, the pointer would transpile to javascript and display the string representation. However, if the pointer is invoked in a :*basic* runtime, the pointer would transpile the code entry along all dependent forms, send it to the external process for evaluation and then display the end result. Different runtimes can be added to provide connections to different external processes and to customise behaviour.



2.4 - Extending and Writing Plugins

In general, the transpile process is completely transparent and very easy to modify. A new language grammar can be added in as little as 100 lines of code, whilst a new runtime can be added in as little as 1000 lines of code. Instead of having to learn a new ecosystem every time one changes a language, it is possible to bring that ecosystem into a LISP environment where it can play nicely with all other target languages that plugin to *std.lang*.

The complete spec for the c grammar is shown below:

```
(ns std.lang.model.spec-c
  (:require [std.lang.base.emit :as emit]
            [std.lang.base.grammar :as grammar]
            [std.lang.base.util :as ut]
            [std.lang.base.book :as book]
            [std.lang.base.script :as script]
            [std.string :as str]
            [std.lib :as h]))

(defn tf-define
  "not sure if this is needed (due to defmacro) but may be good for source to source"
  {:added "4.0"}
  [[_ sym & more :as form]]
  (let [[args body] (if (= 1 (count more))
                      [nil (first more)]
                      more)]
    (if args
      (list :- "#define" (list :% sym (list 'quote
                                             (apply list args)))
            body)
      (list :- "#define" sym body))))

(def +features+
  (-> (grammar/build :exclude [:data-shortcuts
                               :control-try-catch
                               :class])
      (grammar/build:extend
       {:define  {:op :define  :symbol '#{define}  :macro #'tf-define
                  :type :def :emit :macro
                  :section :code :priority 1}})))

(def +template+
  (->> {:banned #{:set :map :regex}
        :highlight '#{return break}
        :default {:function  {:raw ""}}
        :data    {:vector    {:start "{" :end "}" :space ""}
                  :tuple     {:start "(" :end ")" :space ""}}
        :block  {:for       {:parameter {:sep ","}}}
        :define {:def       {:raw ""}}}
       (h/merge-nested (emit/default-grammar))))

(def +grammar+
  (grammar/grammar :c
    (grammar/to-reserved +features+)
    +template+))

(def +meta+
  (book/book-meta
   {:module-current   (fn [])
    :module-import    (fn [name _ opts]
                        (h/$ (:- "#include" ~name)))
    :module-export    (fn [{:keys [as refer]} opts])}))

(def +book+
  (book/book {:lang :c
              :meta +meta+
              :grammar +grammar+}))

(def +init+
  (script/install +book+))
```

2.5 - DCC Pipelines and Ad Hoc Networks

DCC or Direct Client to Client systems have become quite popular when combined with debugging or REPL type tooling for many systems. Most DCC clients make use of websockets which to send and receive network data. With *std.lang*, Not only can websockets be used but raw sockets, http protocol, jdbc protocol, redis protocol and many other RPC connectors.

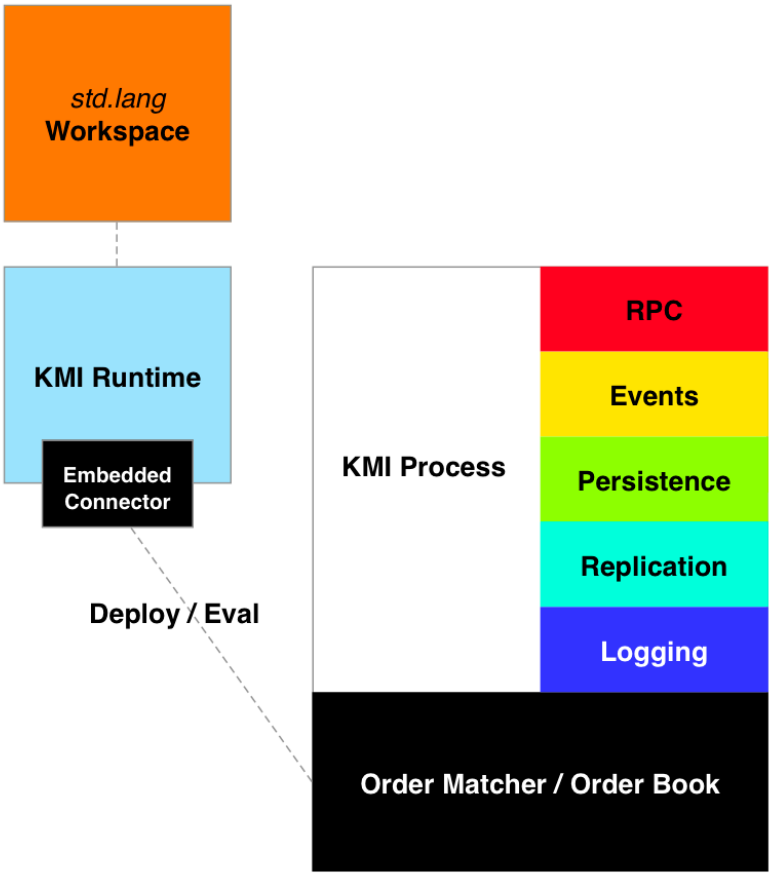
It is envisioned that as more plugin languages and plugin runtimes are developed for *std.lang*, the library can become a hub for experimental cross development between runtimes that usually would be extremely difficult to integrate. Other benefits include code sharing between similar languages through *xtalk*, a language developed for interop between dynamic languages.

3 - Case Studies

3.1 - kmi (embedded order books)

kmi is the fast order book solution at the heart of Statstrade. It is a software based solution, deployable on commercially available hardware which is written for an embedded runtime, with built-in functionality for take care of connections, eventing, storage, node replication and metrics.

This is a completely different approach to software based order book solutions and is the reason why the kmi solution can be so fast - easily reaching 5k transactions/second on extremely basic hardware specifications.



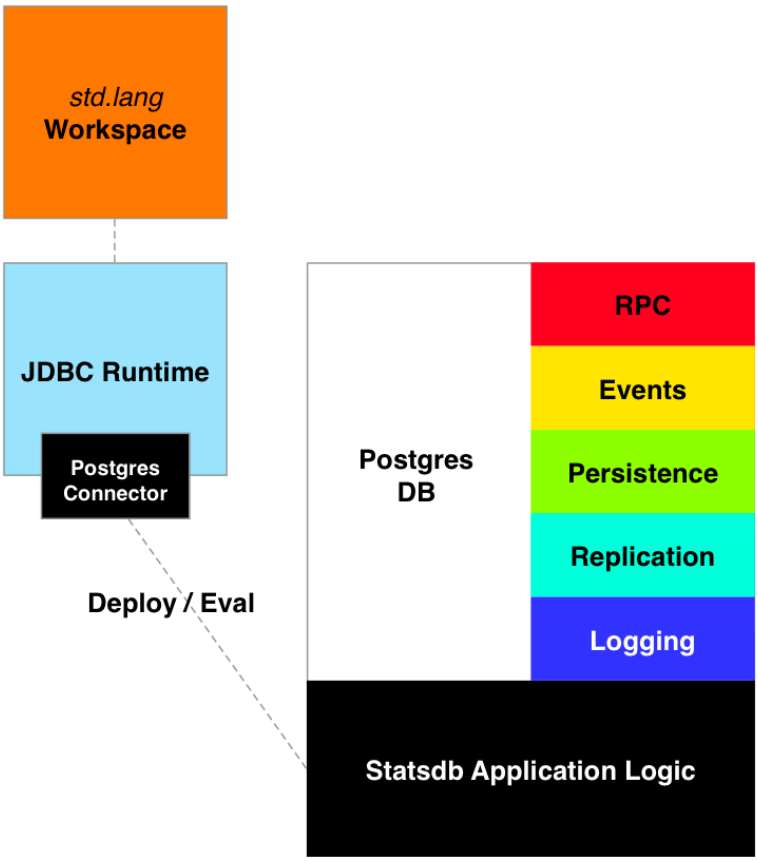
Use of *std.lang* to implement, test and deploy the order book logic was a huge factor in ensuring reliability in the system as well as to integrate the code with other exchange related runtimes.

3.2 - statsdb (embedded trading api)

statsdb is the application api powering the Statstrade API and trading system. The logic has been written through a *plpgsql* grammar plugin for *std.lang* that deploys application code straight into the postgres database.

As with the *kmi* module, use of *std.lang* to implement, test and deploy core was a huge factor in ensuring reliability in the system. A boon to writing *plpgsql* code is that functions are all or nothing. That means they either do a change or all changes fail. This is really important for updates on multiple tables.

Such tasks done with traditional approaches (using the JVM, Nodejs or Python as the Main Runtime) can potentially lead to table locking and synchronisation issues. This can lead to database functions being up to 100x faster than the standard setup. Furthermore, the code is simpler and monitoring only needs to be on the database so overall system code is greatly reduced.



The resulting in benchmarks for order creation reaching 1k transactions/second on extremely basic hardware specifications.

3.3 - xt.db (multi-language sql library)

xt.db is a database access library with code sharing between javascript and lua runtimes, targeting both postgresql and sqlite. It is written in *xtalk* (the cross language templating language) and so can be transpiled to *lua, javascript, python* and additional language grammars that are compatible with *xtalk.*

This saves greatly on coding separately for each language and hopefully with the expansion of *xtalk* compatibility, will see even more bang for buck at the code level.

3.4 - solidity development (testing of blockchain code)

rt.solidity is an mixed language runtime developed exclusively to write, test, and deploy solidity code on the blockchain. Generally, testing solidity code is an extremely tedious setup but the runtime allows the standard approach for testing used on the native runtime, as well as being able to test blockchain methods that are not exposed as public API through code rewrite techniques make possible through use of *std.lang*.

4 - Case Proposals

4.1 - lisp style environment for game editors/game engines

Runtime connectors can be created for game engines/editors like Godot, Blender, Max, Nuke so that developers can more easily run scripts and export data directly from the game engine into a code editor.

Additional visualisation runtimes such as the browser can be plugged in so that assets can be viewable directly on the web.

4.2 - embedded fast deployment for c, verilog and vhdl code

Grammars can be created for hardware base languages such as verilog and vhdl and runtime connectors written for fast deployment using embedded systems so that code can be tested in the same matter as lisp code

4.3 - mixed environments for industry

It should be worth exploring more how industry can use *std.lang* runtime connectors. With the advent of seperate runtimes for AI processing, Visualisation as well as Data Collection, *std.lang* and it's model for code/data sharing can provide a extremely effective and flexible solution for the headache of interoperability.