(not= DSL macros)

Writing DSLs is hard

- At least for me
- Enlive, Moustache, Parsley:
  - Several iterations
  - And counting!
  - Main sin: using macros
- Learnt some lessons along the way
  - Applied here: github.com/cgrand/regex

So, what's a DSL?

DSL in Clojure

- "mini-languages" in core:
  - destructuring, seq comprehensions, anonymous fns, syntax quote, pre and post-conditions, -> and ->> etc.
- ClojureQL, Clout, Hiccup, Matchure, Enlive, Moustache etc.

What's a DSL?

- Domain Specific Language
  - Scope-limited
  - not a General Purpose Language
  - not always Turing-complete
- Internal DSL:
  - Written in the host language (Clojure)
  - Mostly intended for developers

What's a DSL?

- Succinct notation for DS things
  - Data
  - Logic
- DSL benefits
  - Usually more declarative
  - Reduce incidental complexity

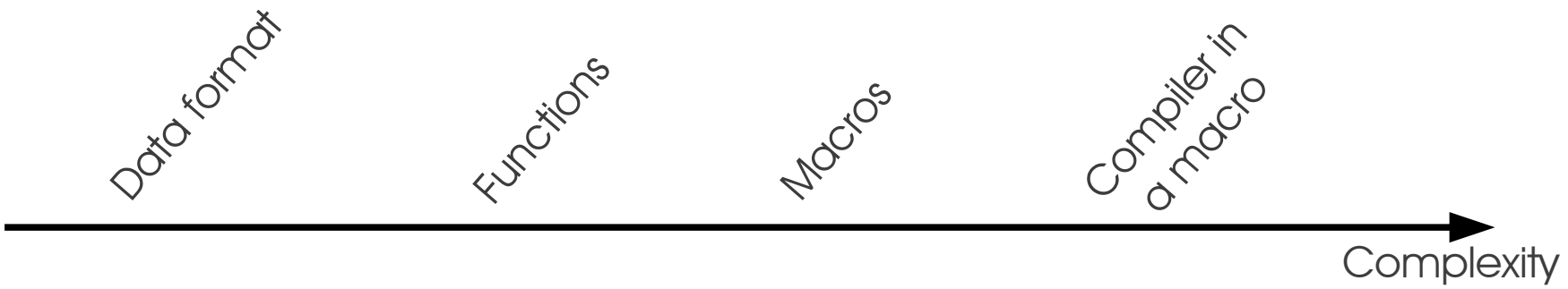
What's a DSL?

- Succinct notation for DS things
  - Data
  - Logic
- DSL benefits
  - Usually more declarative
  - Reduce incidental complexity

Succinct declarative notation for DS data & logic

A blurry line

- A continuum
  - from data schemas
  - to compiler-in-a-macro
- Every API is a DSL



Complexity

Enlive users complained about:

- selectors not being first class
- and being hard to compose

Complexity

Enlive users complained about:

- selectors not being first class
- and being hard to compose

I rewrote it and removed macros

Complexity

ClojureQL users complained about:

- queries being too suprising
- and hard to compose

Complexity

ClojureQL users complained about:

- queries being too suprising
- and hard to compose

Kotarak and Lau are rewriting it and removing macros

Complexity

Do you spot a pattern?

Spoilt users!

- Users don't want a DSL
- Users want to use all the power of Clojure
- Conclusion: **Users want values**

Spoilt users!

- Users don't want a DSL
- Users want to use all the power of Clojure
- Conclusion: **Users want values**

Values + functional core

(+ macros as icing-on-the-cake)

= DSL success!

Limiting complexity

- Limit scope
  - Don't reinvent lexical scoping or control flow
  - Rely on Clojure for these
- Limit syntax
  - Use closures
  - Use higher order functions
  - Ugly? Syntax is icing, add it later!
- No macros

Start easy, simple, humble

- Don't rush for macros!
  - Premature optimization
  - Too much rope
- Start humble:
  - Data
  - a handful of functions

Data

- Focus on DS values rather than DS Language
- Give DS semantics to datatypes
  - Clojure's ones
  - Your own ones
  - Except lists and symbols
    - Avoid confusion
    - Not pretty (need to be quoted)
    - Visual escape to regular Clojure

But I really need macros!

— anonymous macro addict

You sure? Dynamicity is good...

- Macros as premature optimization
- Example: Enlive's commit 944312b1621
  - Make selectors first class
  - Delete 12 defmacros out of 24
  - Allowed for simpler optimizations (caching)

Net result: faster and more dynamic

Ok, Macros

Legitimate usecases	Tentative workarounds
Control flow	Closures, delays
Binding	Decouple binding

Decouple binding, step by step:

- Design your DSL (values and core fns) **without binding** (but provision it) but **with capturing**
- ???
- Profit!!!

??? explained

- Create a “binding specs → capturing specs” fn
- Create a “binding specs → binding forms” fn

```
(defmacro when-match [pattern value & body]
  (let [matcher (capturing pattern)
        bindings (extract-bindings pattern)]
    `(when-let [~bindings
                (capture ~matcher value)]
       ~@body)))
```

Decouple binding and capturing

Example: a DSL for regexes

simplified version of github.com/cgrand/regex

(e.g. no named groups)

Regex DSL

- A DSL for regexes in Clojure
  - Compile to host regexes
- Not that useful: regexes are already a DSL
  - External and composable though
- Basic building blocks of regexes:
  - Literals, sequences, alternatives, char ranges, repetitions and a wildcard

Giving Regex semantics to types

- Literals
- Sequences
- Alternatives
- Char ranges
- Repetitions
- Wildcard

Notations?

Giving Regex semantics to types

- Literals `"abc", \d`
- Sequences `[this then-that and-also]`
- Alternatives `#{this or-that or-also}`
- Char ranges `{\a \z, \A \Z, \0 \9}`
- Repetitions `(re/repeat this min? max?)`
- Wildcard `re/any`

Evaluating to Java Patterns

regex is the user fn:

```
(defn regex [spec]
  (-> spec pattern Pattern/compile))
```

Then, define `pattern` as a protocol fn

```
(defprotocol RegexNotation
  (pattern [this] "Returns a corresponding pattern (as String)."))
```

Evaluating to Java Patterns

```
;; literals, sequences, alternatives and char ranges
(extend-protocol RegexNotation
  String
    (pattern [s]
      (Pattern/quote s))
  Character
    (pattern [c]
      (pattern (str c)))
  clojure.lang.APersistentVector
    (pattern [v]
      (str/join (map pattern v)))
  clojure.lang.APersistentSet
    (pattern [s]
      (str "(?:" (str/join "|" (map pattern s)) ")"))
  clojure.lang.APersistentMap
    (pattern [m]
      (str "[" (str/join (for [[from to] m]
                           (str from "-" to)))
        "]")))
```

Evaluating to Java Patterns

```
(regex (let [d {\0 \9}
             d2 [d d]
             d4 [d2 d2]]
         ["date: " d4 \- d2 \- d2]))
;; #"\Qdate: \E[0-9][0-9][0-9][0-9]\Q-\E[0-9][0-9]\Q-\E[0-9][0-9]"

(let [d {\0 \9}
      d2 [d d]
      d4 [d2 d2]]
  (regex ["date: " d4 \- d2 \- d2]))
;; or
(let [d {\0 \9}
      d2 [d d]
      date (into ["date: " d2] (interpose \- [[d2 d2] d2 d2]))]
  (regex date))
;; are equivalents

;; DSL fragments are first class,
;; you can use all clojure to build them!
```

We need new types!

Repetition and `any` need their own types:

```
(defrecord Repetition [spec min max]
  RegexNotation
    (pattern [r]
      (str "(?:" (pattern spec) ")"
        "{" (or min 0) "," (or max "") "}")))

(defn repeat
  ([spec] (repeat spec nil nil))
  ([spec min] (repeat spec min nil))
  ([spec min max] (Repetition. spec min max)))

;; any is the only one of its kind -> reify
(def any (reify RegexNotation
           (pattern [_] ".")))
```

Helper fns

You can easily define helper fns:

```
(defn join
  ([spec separator] (join spec separator nil nil))
  ([spec separator min] (join spec separator min nil))
  ([spec separator min max]
    (if (zero? (or min 0))
      (repeat (join spec separator 1 max) 0 1)
      [spec (repeat [separator spec]
              (dec min) (when max (dec max)))])))

=> (regex (join [{\0 \9} {\0 \9}] \- 1 3))
#"[0-9][0-9](?:\Q-\E[0-9][0-9]){0,2}"
```

Helper fns

The whole truth is:

- `regex` is `eval` for your DSL
- Helper fns act like macros for your DSL

Closing the loop

Making `regex` idempotent

```
(extend-protocol RegexNotation
  Pattern
    (pattern [p] (.pattern p)))

=> (regex #{"hello" "world"})
#"(?:\Qhello\E|\Qworld\E)"
=> (regex (regex #{"hello" "world"}))
#"(?:\Qhello\E|\Qworld\E)"
```

Why does it matter?

- You can canonicalize your inputs
- You can "pre-compile" or cache fragments.

Static optimization

Detecting constant fragments (macros ok...)

- Simplifying them away
- Recognizing your own functions:

Don't check syms, check vars!

```
(when-not (contains? &env sym)
  (resolve sym)) ; 1.2

(resolve &env sym) ; 1.3
```

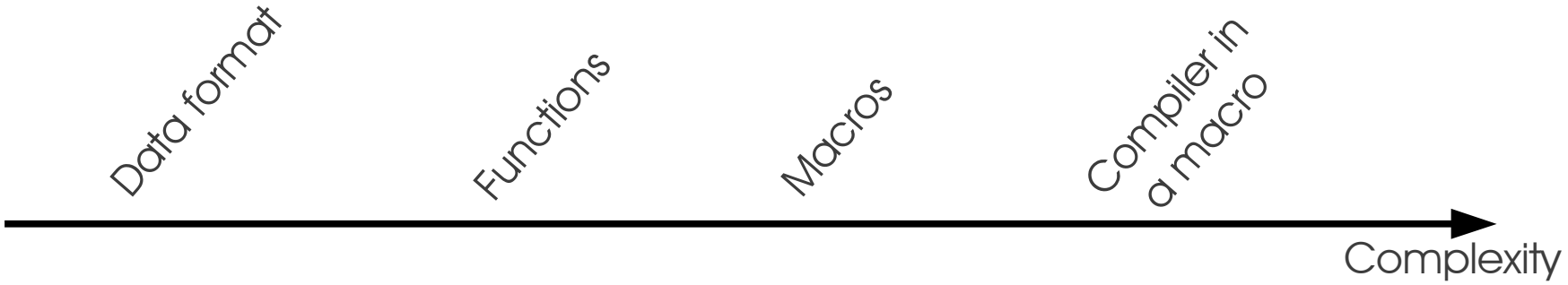
Static optimizations

- Compiling constant fragments
  - "eval" them (e.g. call `re/regex`)
  - Expand to the value when readable or to a factory form

That's all folks!
Questions?


STOP

Complexity



- Complexity for the implementor
  - The joy of tree-walking
- Complexity for the user
  - Different from regular Clojure code
  - What is first class? What's not?

DSL Design

- Code is data
  - Or the other way round
  - Rich range of literals
    - Vectors, maps, sets, keywords, numbers, regexes
  - No need to quote, walk, extract/unquote
    - arguments evaluation without call semantics

```
[:operator arg1 arg2]
(operator arg1 arg2)
```

    - Unless you use lists

DSL Design

- List forms are still cumbersome when doing static analysis

  - Always use a conservative default
    "When in doubt, don't" – Benjamin Franklin
- `&env` relieves the pain
  - ```
    (when-not (contains? &env sym)
      (resolve sym))
    ```
  - test against vars, not syms
  - no more shadowing problems
    ```

Macros

Legitimate macros	Tentative workarounds
Control flow	Closures, delays
Binding	Decouple binding

Decouple binding example, Moustache routes:

```
(let [bindings (derive-bindings route-spec)
      simple-route (derive-simple-route route-spec)]
  `(when-let [~bindings (match-route ~url ~simple-route)]
     ...))
```

- `match-route` is functional
- I could have done without `derive-simple-route`