Term Rewriting with Meander

As a programmer, my job is to write the function to transform inputs to outputs. Squares go in, circles come out. Defining functions is my fundamental form of expression.

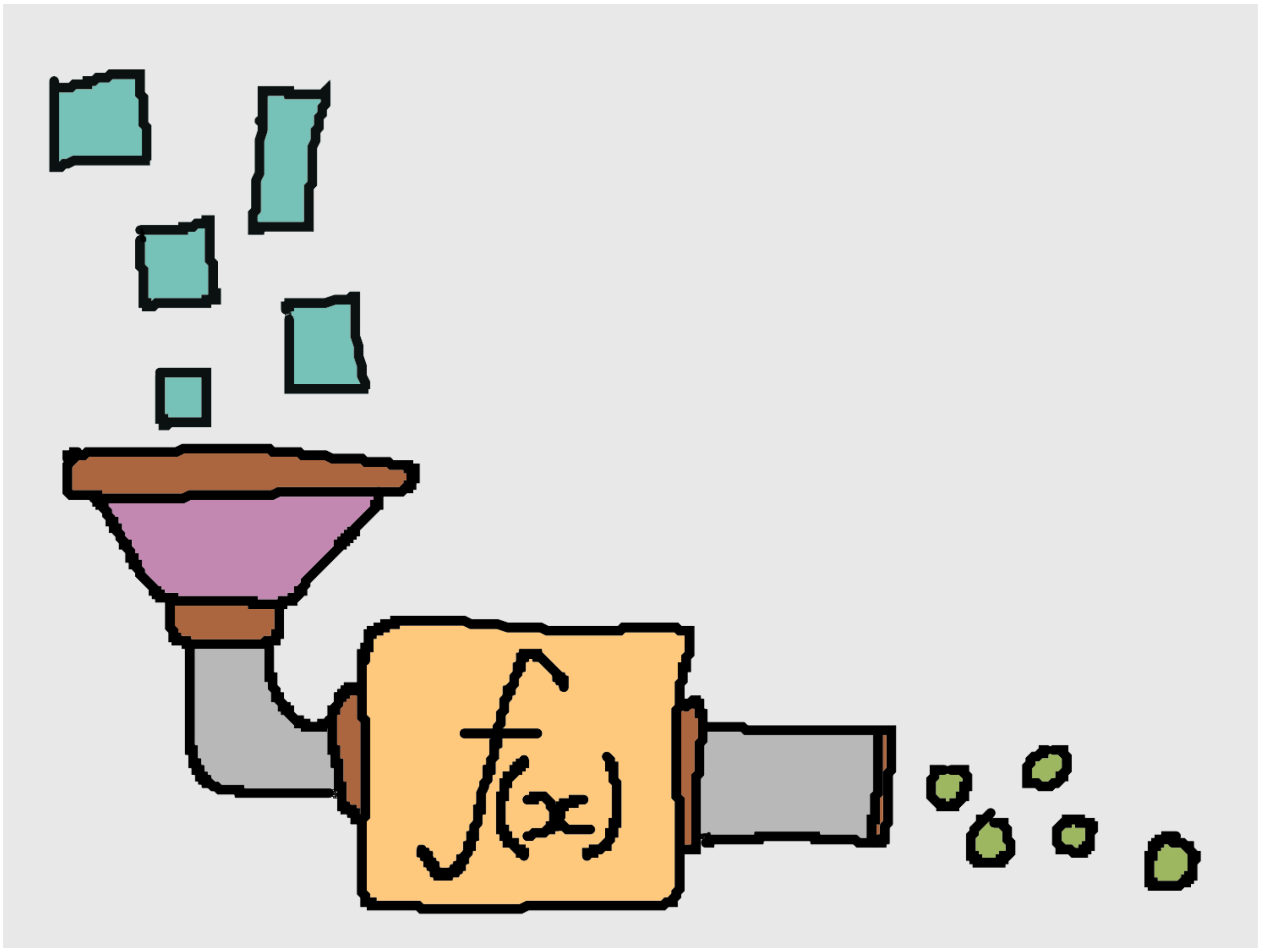


Figure 1. The square to circle machine ™

As a mathematician, functions are important, but *equations* are my fundamental form of expression. Equations define valid rules for transforming one expression into another. My job is to solve problems by symbolic manipulation.

```
\begin{align*}
(a + b)^2 &= (a + b)(a + b)\\
          &= aa + ab + ba + bb\\
          &= a^2 + 2ab + b^2
\end{align*}
```

Functions and equations are both ways to specify transformations. So what does it look like to program data transformation like equations instead of functions? Today we will look at some examples of this approach. Let's establish some definitions before diving into the examples.

Definitions

Term rewriting	*Replacing terms with other terms*
Term	*A variable or function application:* `a`, `(a + b)`, `(a + b)^2`
Variable	`a,b`
Function	`add`, `multiply`, `square`
Rule	*A pair of matching terms and substitution terms:* `(a + b)^2 = a^2 + 2ab + b^2`
Matching	`(1 + 3)^2` *can be made equal to* `(a + b)^2` *when* `a=1,b=3`
Substitution	*Creation of new terms from old ones* `(1 + 3)^2 => 1^2 + 2\cdot1\cdot3 + 2^2`

Math equations must preserve equivalence. Where we are heading, we don’t need that restriction.

Meander

A term rewriting system.

A Clojure library for data transformation.

```
[meander/epsilon "0.0.512"]
```

```
(ns meander.examples
  (:require [meander.epsilon :as m]))
```

A *structural pattern syntax* for data transformation.

Rewrite

Let’s do our first transformation from a vector to a map:

`[a b c d] ⇒ {a d}`

```
; Meander!
(m/rewrite [:a 1 :c 2] ; input
  [?a ?b ?c ?d]        ; a match pattern
  {?a ?d})             ; a substitution pattern

;=> {:a 2}             ; output
```

- The *match pattern* is a data literal with the shape of the input
- The *substitution pattern* is a data literals with the shape of the output
- `?a` is called a *logic variable*

`m/rewrite` is a macro; it defines how to interpret the patterns. I think of Meander as a syntax rather than a library. I specify structure with data literals to pattern match and substitute. Hence why I call it a "structural pattern syntax".

Logic fail

Regular variables are assigned values.
Logic variables are either matched or not.

```
(m/rewrite [1 2] ; input
  [?a ?a]        ; a match pattern
  {?a ?a})       ; a substitute pattern

;=> nil
```

No match.
`?a` cannot match both `1` and `2`.

```
(m/rewrite [1 1] ; input
  [?a ?a]        ; a match
  {?a ?a})       ; a substitution pattern

;=> {1 1}
```

Matched `?a` with `1`

Unification means resolving logical matches.

Repeating terms

```
(m/rewrite [:a 1 :c 2 :d 3]
  [!a ...]
  [!a ...])

;=> [:a 1 :c 2 :d 3]
```

`!a`	memory variable: *an array that can collect many values*
`...`	0 or more repeated occurrences

The match pattern is symmetrical with substitution pattern, so we got back out exactly what we put in.

Repeat scope

```
(m/rewrite [1 2 3 4 5 6]
  [!a !b ...]      ; match pairs
  [!b !a ...])     ; substitute flipped pairs

;=> [2 1 4 3 6 5]
```

All terms (`!a !b`) are repeated, not just the last thing.

Separator

```
(m/rewrite [:hello 1 2 3]
  [:hello . !v ...]
  [:world . !v ...])

;=> [:world 1 2 3]
```

Only terms between `.` and `⋯` repeat.

Structurally explicit

Most Clojure functions will coerce their inputs to seqs. Meander does not do this.

`[!x ⋯]`	will only match **vectors**
`(!x ⋯)`	will only match **lists** and **seqs**

`{:k ?v}`	will only match **maps**
`#{1}`	will only match **sets**
`(m/seqable !x …)`	will match **maps**, **vectors**, **strings**, etc…

I like that it will only match the same types unless you choose to be more permissive with `m/seqable`.

Nesting

As you might expect, match and substitute patterns may be nested.

```
(m/rewrite {:xs [1 2 3 4 5]
            :ys [6 7 8 9 10]}
  {:xs [!xs ...]
   :ys [!ys ...]}
  [!xs ... . !ys ...])

;=> [1 2 3 4 5 6 7 8 9 10]
```

And Meander does the rearranging for you.

Pop Quiz!

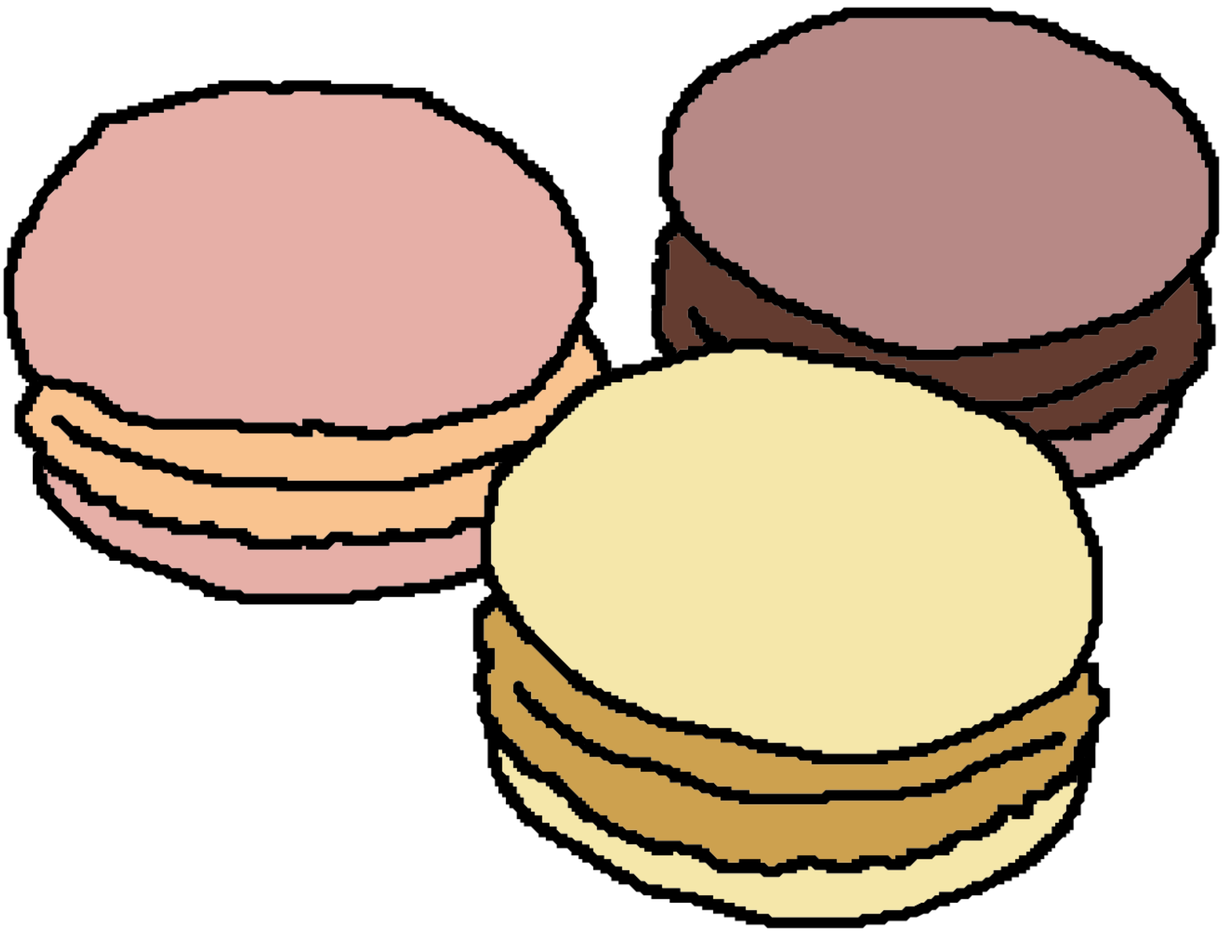
```
(m/rewrite [:hello 1 2 3]
  [?k . !v ...]
  [?k !v ...])

;=> [:hello 1 :hello 2 :hello 3]
```

Why?

When matching, we expect a single value, then repeated values. The substituting pattern has no separator, so we are substituting pairs.

Virtual prize



Much more in Meander

Meander has a rich feature set, and I have only shown you the core syntax. Rather than explain all of Meanders features in detail, we are now going to shift gears and look at some examples in the wild. I'll introduce a few bits of unexplained syntax as we go. Don't worry, most of them are obvious.

For a comprehensive treatment of Meander features, see Meander on GitHub which links to documentation, a cookbook, blog posts and a Strange Loop talk.

Rearranging logic expressions

```
(def rearrange-logic
  (bottom-up
   ;; nested commutative logic is raised
   ((m/pred #{'and 'or} ?op) . !before ... (?op . !clauses ...) . !after ...)
   (?op . !before ... !clauses ... !after ...)

   ;; moves `or` to the outside, and `and` to the inside to match Datalog rule
convention
   (and . !before ... (or . !clauses ...) . !after ...)
   (or . (and . ~@!before . !clauses . ~@!after) ...)

   ;; identity logic expressions are flattened
   ((m/pred #{'and 'or} ?op) ?body)
   ?body

   ;; double negatives are removed
   (not (not ?body))
   ?body

   ;; moves `not` inside to match Datalog rule convention
   (not (or . !clauses ...))
   (and . (not !clauses) ...)

   (not (and . !clauses ...))
   (or . (not !clauses) ...)))
```

https://github.com/timothypratley/justice/blob/master/src/justice/translation.cljc

Trees to entities

```
(m/rewrite skynet-widgets
  [{:basic-info {:producer-code !producer-code}
    :widgets [{:widget-code !widget-code
               :widget-type-code !widget-type-code} ...]
    :widget-types [{:widget-type-code !widget-type-code
                    :description !description} ...]} ...]
  [[!producer-code !widget-code !description] ...])
;=> [["Cyberdyne" "Model-101" "Resistance Infiltrator"]
;    ["ACME" "Model-102" "Mimetic polyalloy"]]
```

Blog post: "The answer to map fatigue"

Web scraping

```
(def extract-employees
  (s/search
   (m/$
    [:div {:class "directory-tables"} &
     (m/scan
      ;; heading/table pairs
      [:h3 {} ?department & _]
      _
      [:table &
       (m/scan
        [:tbody &
         (m/scan
          [:tr &
           (m/separated
            [:td & (m/scan [:a {} ?name & _])]
            [:td {} ?title & _]
            [:td & (m/scan [:a {:href ?mailto} & _])])])])])])])
   ;;=>
   {:department (str/trim ?department)
    :name ?name
    :title ?title
    :email (subs ?mailto 7)}))
```

https://github.com/timothypratley/chartit/blob/master/src/chartit/justworks.clj

Parsing defn like forms

```
(defn wrap-defn
  "Returns a function that will parse a form according to `defn` semantics.
  Takes a function which will convert fn-spec forms."
  [rewrite-fn-spec]
  (m/rewrite
   (m/and ((pred simple-symbol? ?name) .
           (pred string? !?docstring) ...
           (pred map? !?attr-map) ...
           !tail ...)
        (m/guard (<= (count !?docstring) 1))
        (m/guard (<= (count !?attr-map) 1))
        (let
         (or (([(m/pred simple-symbol? !params) ... :as !param-list] . !forms ... :as
!fn-specs) ..1)
             ([(m/pred simple-symbol? !params) ... :as !param-list] . !forms ... :as
!fn-specs))
          (list* !tail))
        (m/guard (apply distinct? (map count !param-list))))
   ;;>
   (defn ?name . !?docstring ... !?attr-map ...
     ~@(map rewrite-fn-spec !fn-specs))))
```

https://github.com/timothypratley/justice/blob/master/src/justice/defn.cljc

Compare with https://blog.klipse.tech/clojure/2016/10/10/defn-args.html

HappyGAPI

```clojure
(defn summarize-schema [schema request depth]
  "Given a json-schema of type definitions,
  and a request that is a $ref to one of those types,
  resolves $ref(s) to a depth of 3,
  discards the distracting information,
  and returns a pattern for constructing the required input."
  (m/rewrite request
    {:type                 "object"
     :id                   ?id
     :additionalProperties ?ap
     :properties           (m/seqable [!property !item] ...)}
    ;;>
    {& ([!property (m/app #(summarize-schema schema % depth) !item)] ...)}

    {:type  "array"
     :items ?item}
    ;;>
    [~(summarize-schema schema ?item depth)]

    {:type (m/pred string? ?type)}
    ;;>
    (m/app symbol ?type)

    {:$ref (m/pred string? ?ref)}
    ;;>
    ~(if (> depth 2)
       (symbol ?ref)
       (summarize-schema schema (get schema (keyword ?ref)) (inc depth)))))
```

https://github.com/timothypratley/happygapi/blob/master/dev/happy/beaver.clj

AST manipulation

```
(def propagate-types-from-bindings-to-locals
  "We propagate type information which is stored in metadata
  from the the place where they are declared on a symbol
  to all future usages of that symbol in scope."
  (s/rewrite
    {:op   :local
     :form ?symbol
     :env  {:locals {?symbol {:form ?symbol-with-meta
                              :init ?init}}
            :as     ?env}
     &     ?more
     :as   ?ast}
    ;;>
    {:op   :local
     :form ~(propagate-ast-type ?init ?symbol-with-meta ?ast)
     :env  ?env
     &     ?more}

    ;; otherwise leave the ast as is
    ?else
    ?else))
```

https://github.com/echeran/kalai/blob/nanopass/src/kalai/pass/kalai/b_kalai_constructs.clj

Let's reflect

- Those were all pretty complicated transformations
- Could you follow them?
- Is that normal?

Cooking up a function

1. Prepare meat and potatoes	*(example inputs)*
2. Mix ingredients in a pot - A little destructuring - a dash of `let` binding - some sequence seasoning `for` flavor - a teaspoon of `conj`, `update`, and `assoc` - a splash of threading	*(function definition)*
3. Boil	*(iterate, make it work)*
4. Garnish with documentation	*(docstring or types)*
5. Simmer	*(tests)*

^ Easy to forget why it tastes good

^ Easy to skip documentation and testing

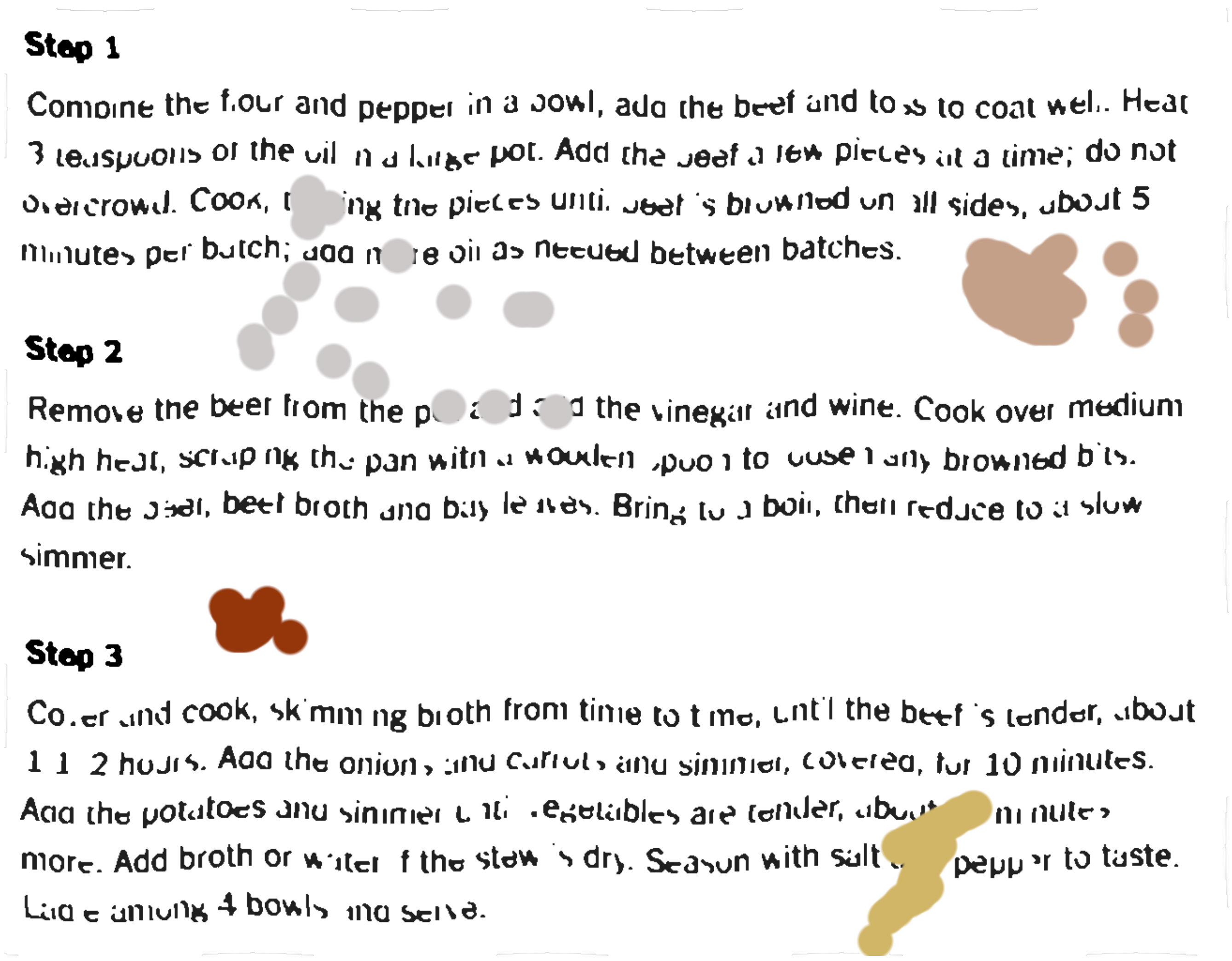
Cooking up a rewrite rule

1. Example input	`[{:k "Meat"} {:k "Potatoes"}]`
2. Parameterize	`[{:k !ingredient} …]`
3. Recombine	`[!ingredient … "Stew"]`
4. Create a test	`(is (= [] (stew example)))`
5. Verify the output	`["Meat" "Potatoes" "Celery" "Stew"]`

Recipe

Functions are all about what to do.

Clearly the big difference between functions and rewrite rules is what to do with the inputs. Writing a function is specifying what to do; writing a recipe.



Step 1

Combine the flour and pepper in a bowl, add the beef and toss to coat well. Heat 3 teaspoons of the oil in a large pot. Add the beef a few pieces at a time; do not overcrowd. Cook, turning the pieces until beef is browned on all sides, about 5 minutes per batch; add more oil as needed between batches.

Step 2

Remove the beef from the pot and add the vinegar and wine. Cook over medium high heat, scraping the pan with a wooden spoon to loosen any browned bits. Add the beef, beef broth and bay leaves. Bring to a boil, then reduce to a slow simmer.

Step 3

Cover and cook, skimming broth from time to time, until the beef is tender, about 1 1 2 hours. Add the onions and carrots and simmer, covered, for 10 minutes. Add the potatoes and simmer until vegetables are tender, about [illegible] minutes more. Add broth or water if the stew is dry. Season with salt and pepper to taste. Ladle among 4 bowls and serve.

Figure 2. A recipe for making beef stew

Picture

Rewrite rules are the inputs and outputs; the "what to do part" is completely missing!

Figure 3. A hearty pot of beef stew

Examples are the best kind of documentation. Writing Meander feels like writing examples. It gives me confidence I solved the right problem, and it’s easy to see what it does coming back to it later.

Data transformation

All pure functions are data transformations… so everything?

Largish data reshaping where you would otherwise do destructuring, updates, restructuring, and pipelines.

What’s the catch?

Why isn’t this the default way of writing code?

- Hard to implement
- Limitations

Limitations

Extension is limited
The syntax available in data literals is strictly limited by the host language. Meander syntax is partially extensible through `defsyntax` which allows new s-expression operator behaviors. Extension is necessary and achieved by allowing function calls in Meander.

Failure to match is opaque
Similar to regular expressions, it can be difficult to figure out why a match isn’t made for a given input. Simplifying and decomposing can help.

Use inline function invocation (`~` or `m/app`) to spy on terms.

Unfamiliar

Common concerns

Reduction
Meander has recursion, so you can reduce. You’re better off using `reduce` because recursion has no syntactic advantage. The next brnach of Meander (zeta) contains a neater way to express reduction.

Memory variable correspondence
Nested memory variables in a single expression can get confusing. There is potential for a syntactic simplification. https://github.com/noprompt/meander/issues/129 The next branch of Meander (zeta) has flexibility in how variables behave, which may solve this.

Decomposition
The current options are `m/with` and `m/app`. This is equivalent to functional decomposition.

Performance
In practice on par with hand rolled functions. In general hard to guarantee.

Compared with

How does Meander compare to other existing data transformation approaches?

Clojure.core
Clojure is concise and powerful. Many functions defined on few primary data structures. We operate on data but do not specify what the data is. So it is common to be looking at a function that operates on `x`, and have no context about what `x` aught to be.

What is `x`? Destructuring:: Does one job well. No logic expressions. Core.match, Core.logic:: Work great. No convenient syntax for substitution. Clojure.spec:: Defines the shape of `x` as s-expression regexes. Specs do not look like the data they describe. Specter:: Navigator/action model https://github.com/redplanetlabs/specter improves data manipulation. does not address shape.

Key Strengths of Meander

- Inputs and outputs are instantly recognizable
- Construct solutions from examples
- Declarative style is pleasing
- High quality library: fast and reliable
- Visual > verbal

Most noticeable when returning to code

Contact

https://timothypratley.blogspot.com

@timothypratley

timothypratley@gmail.com

Thank you!

Term rewriting has merits. Meander provides term rewriting as a convenient library for Clojure. I hope you’ll give it a try and find it as useful as I have.

Thanks for reading.