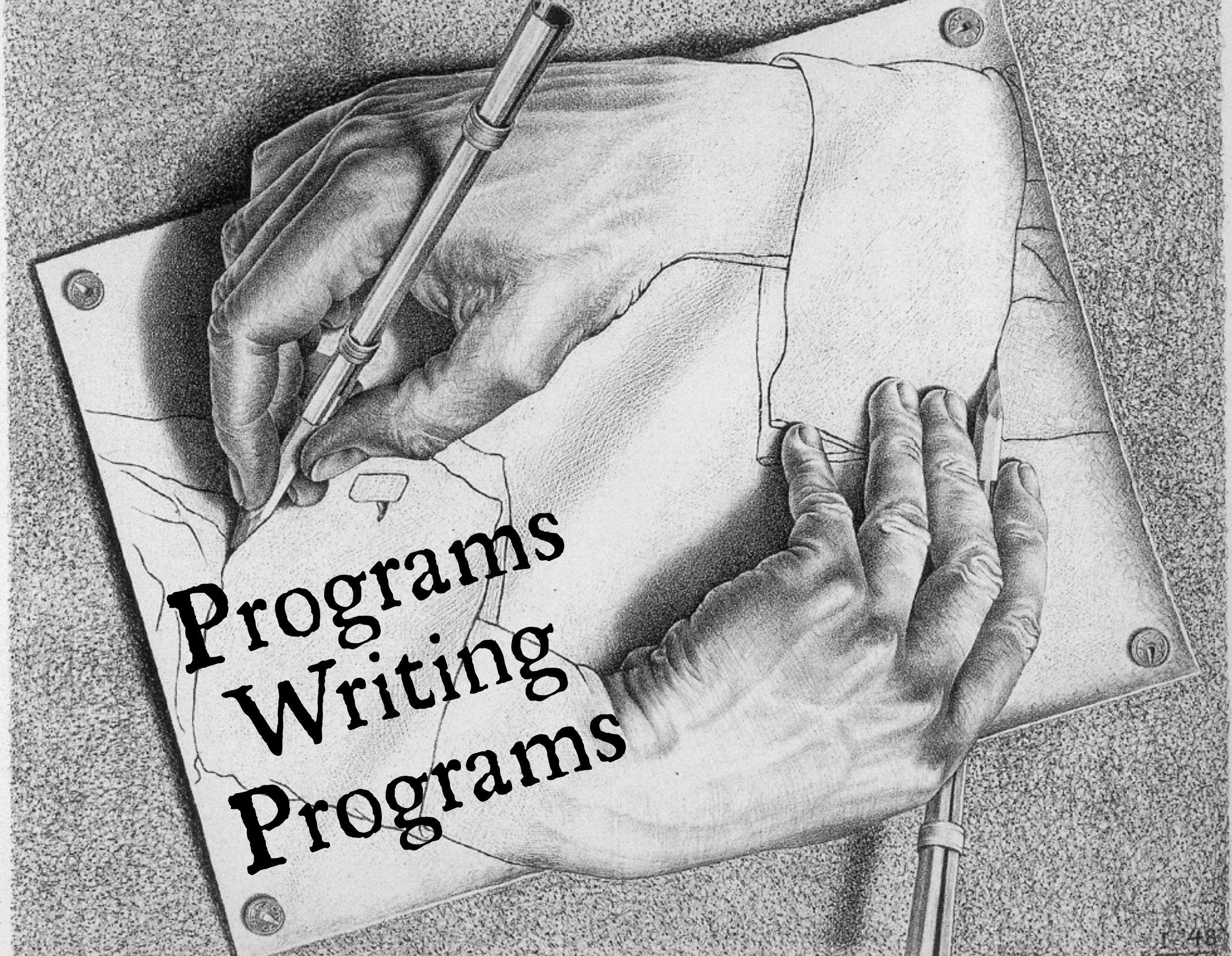
The Macronomicon

:fogus

```
READY.
```

```
READY.
```


Programs
Writing
Programs

Code generation

- **XDoclet**
- **XText**
- **Lombock**
- **Velocity, Erb**
- **MDA**
- **Rails**
- **Hacks**

Many moons ago

```
#define foo "salad

int main(int argc, char* argv[]) {
    printf(foo bar");
    return EXIT_SUCCESS;
}

; salad bar
```

Many moons ago

```
#define foo "salad

int main(int argc, char* argv[]) {
    printf(foo bar");
    return EXIT_SUCCESS;
}

; salad bar
```

(foo bar”);

ZOMG!

Fixed?

```
#define foo "salad”

int main(int argc, char* argv[]) {
    printf(“foo bar");
    return EXIT_SUCCESS;
}

; foo bar
```

Hating expressions

```
#define discr(a,b,c) b*b-4*a*c

int main(int argc, char* argv[]) {
    int x=1, y=2;
    printf(2*discr(x-y, x+y, x-y)*5);
    return EXIT_SUCCESS;
}

; ??
```

Hating expressions

```c
#define discr(a,b,c) b*b-4*a*c

int main(int argc, char* argv[]) {
    int x=1, y=2;
    printf(2*discr(x-y, x+y, x-y)*5);
    return EXIT_SUCCESS;
}

; ??
```

2*x+y*x+y-4*x-y*x-y*5

Hating expressions

```c
#define discr(a,b,c) b*b-4*a*c

int main(int argc, char* argv[]) {
    int x=1, y=2;
    printf(2*discr(x-y, x+y, x-y)*5);
    return EXIT_SUCCESS;
}

; ??
```

(2*x)+(y*x)+y-(4*x)-(y*x)-(y*5)

Hating expressions

```c
#define discr(a,b,c) b*b-4*a*c

int main(int argc, char* argv[]) {
    int x=1, y=2;
    printf("%d", (2*discr(x-y, x+y, x-y)*5);
    return EXIT_SUCCESS;
}

; ??
```

2*(x+y)*(x+y)-4*(x-y)*(x-y)*5

Fixed?

```
#define discr(a,b,c) b*b-4*a*c

int main(int argc, char* argv[]) {
    int x=1, y=2;
    printf(2*discr(x-y, x+y, x-y)*5);
    return EXIT_SUCCESS;
}

; ??
```

```
discr(a,b,c) ((b)*(b)-4*(a)*(c))
```

FX

```
#define discr(a,b,c) b*b-4*a*c

int main(int argc, char* argv[]) {
    int x=1, y=2;
    printf(2*discr(x-y, x--, x-y)*5);
    return EXIT_SUCCESS;
}

; ??
```

2*(x--)*(x--)-4*(x-y)*(x-y)*5


C++

Fact!

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N-1>::value };
};

template <>
struct Factorial<1>
{
    enum { value = 1 };
};

// example use
int main()
{
    const int fact5 = Factorial<15>::value;
    std::cout << fact5 << endl;
    return 0;
}

// 120
```

Böhm and Jacopini

```
template <>
struct AValue<>
{
    enum { value = 42 };
};
```

values

Böhm and Jacopini

```
template <>
struct AValue<>
{
    enum { value = 42 };
};
```

values

```
template <int X, int Y>
struct AFunction
{
    enum { result = X + Y };
};

AFunction<1, 2>::result
//=> 3
```

functions

Böhm and Jacopini

```
template <>
struct AValue<>
{
    enum { value = 42 };
};
```

values

```
template <int X, int Y>
struct AFunction
{
    enum { result = X + Y };
};

AFunction<1, 2>::result
//=> 3
```

functions

```
template <bool cond, class Then, class Else>
struct If
{
    typedef Then RET;
};

template <class Then, class Else>
struct If<false, Then, Else>
{
    typedef Else RET;
};
```

branching

Böhm and Jacopini

```
template <>
struct AValue<>
{
    enum { value = 42 };
};
```

values

```
template <int X, int Y>
struct AFunction
{
    enum { result = X + Y };
};

AFunction<1, 2>::result
//=> 3
```

functions

```
template <bool cond, class Then, class Else>
struct If
{
    typedef Then RET;
};

template <class Then, class Else>
struct If<false, Then, Else>
{
    typedef Else RET;
};
```

branching

```
template <int N>
struct Recur
{
   enum { result = N * Recur<N-1>::result };
};

template <>
struct Recur<0>
{
    enum { result = 1};
};
```

recursion

Böhm and Jacopini

```
template <>
struct AValue<>
{
    enum { valu
};
```

```
template <int X, int Y>
struct AFunction
{
                    ult = X + Y };
```

```
>::result
```

val

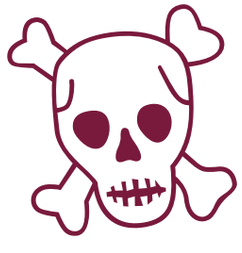
ctions

Turing

Completeness

```
template <bool cond
struct If
{
    typedef Then RE
};

template <class Then
struct If<false, The
{
    typedef Else RET;
};
```

```
 * Recur<N-1>::result };
```



```
esult = 1};
};
```

branching

recursion

Böhm and Jacopini

```
template <>
struct AValue<>
{
    enum { value = 42 };
};
```

values

```
template <int X, int Y>
struct AFunction
{
    enum { result = X + Y };
};

AFunction<1, 2>::result
//=> 3
```

functions

```
template <bool cond, class Then, class Else>
struct If
{
    typedef Then RET;
};

template <class Then, class Else>
struct If<false, Then, Else>
{
    typedef Else RET;
};
```

branching

```
template <int N>
struct Recur
{
   enum { result = N * Recur<N-1>::result };
};

template <>
struct Recur<0>
{
    enum { result = 1};
};
```

recursion

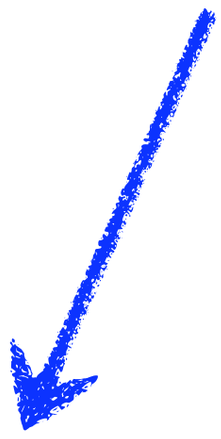

OCaml

From this...

```
List.map (fun x -> x+1) [1; 2; 3]
```

To this...

```
List.map (fun x -> x+1) [1; 2; 3]
```



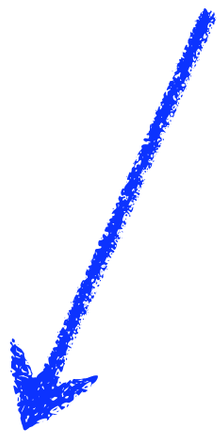
```
ExApp (loc, ExApp (loc, ExAcc (loc,
   ExUid (loc, "List"), ExLid (loc, "map")),
     ExFun (loc, [(PaLid (loc, "x"), None,
       ExApp (loc, ExApp (loc, ExLid (loc, "+"),
         ExLid (loc, "x")), ExInt (loc, "1")))])),
   ExApp (loc, ExApp (loc, ExUid (loc, "::"),
    ExInt (loc, "1")),
     ExApp (loc, ExApp (loc, ExUid (loc, "::"),
       ExInt (loc, "2")), ExApp (loc,
         ExApp (loc, ExUid (loc, "::"), ExInt (loc, "3")),
         ExUid (loc, "[]")))))
```

Just this...

```
List.map (fun x -> x+1) [1; 2; 3]
```

Well, this...

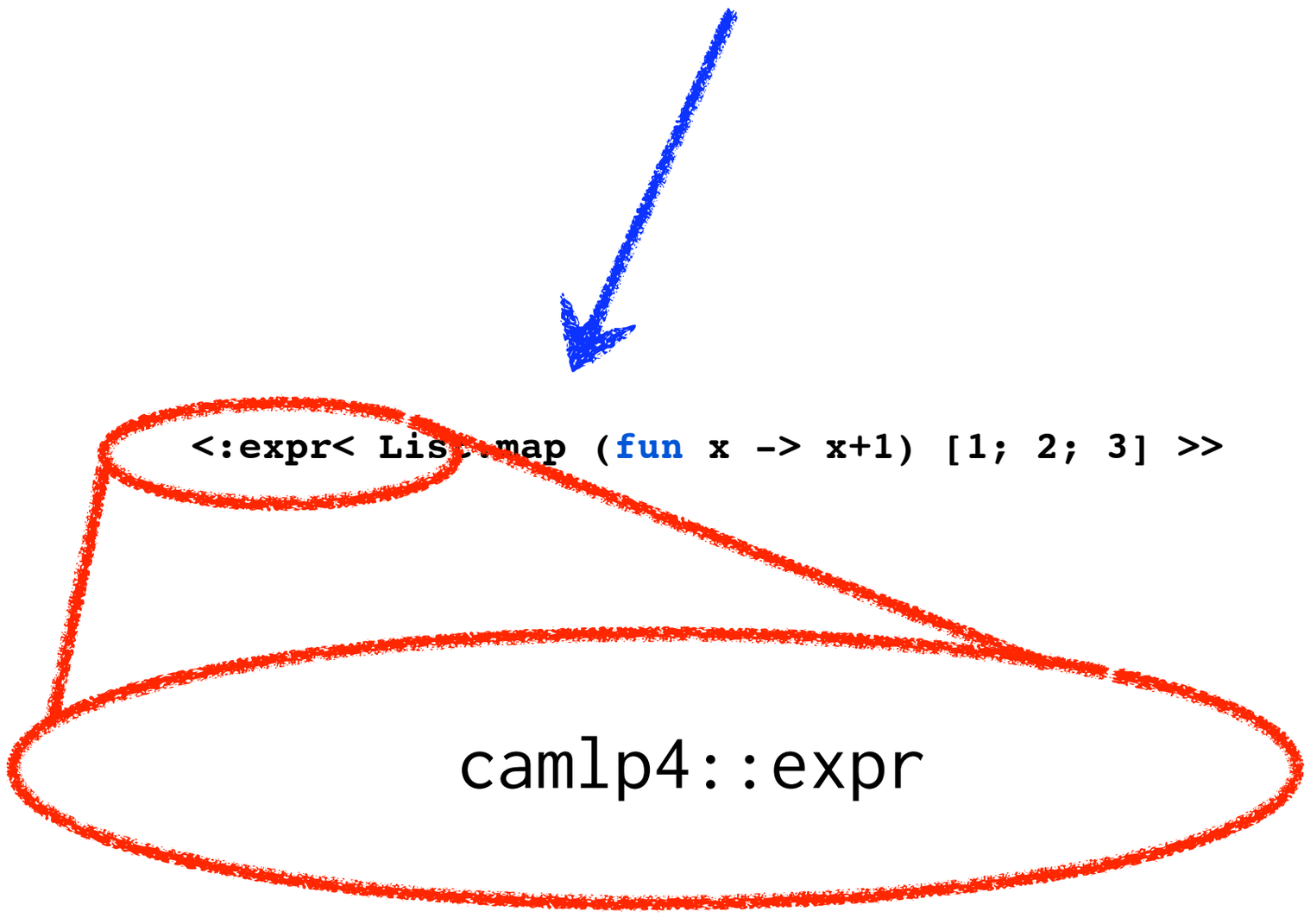
```
List.map (fun x -> x+1) [1; 2; 3]
```

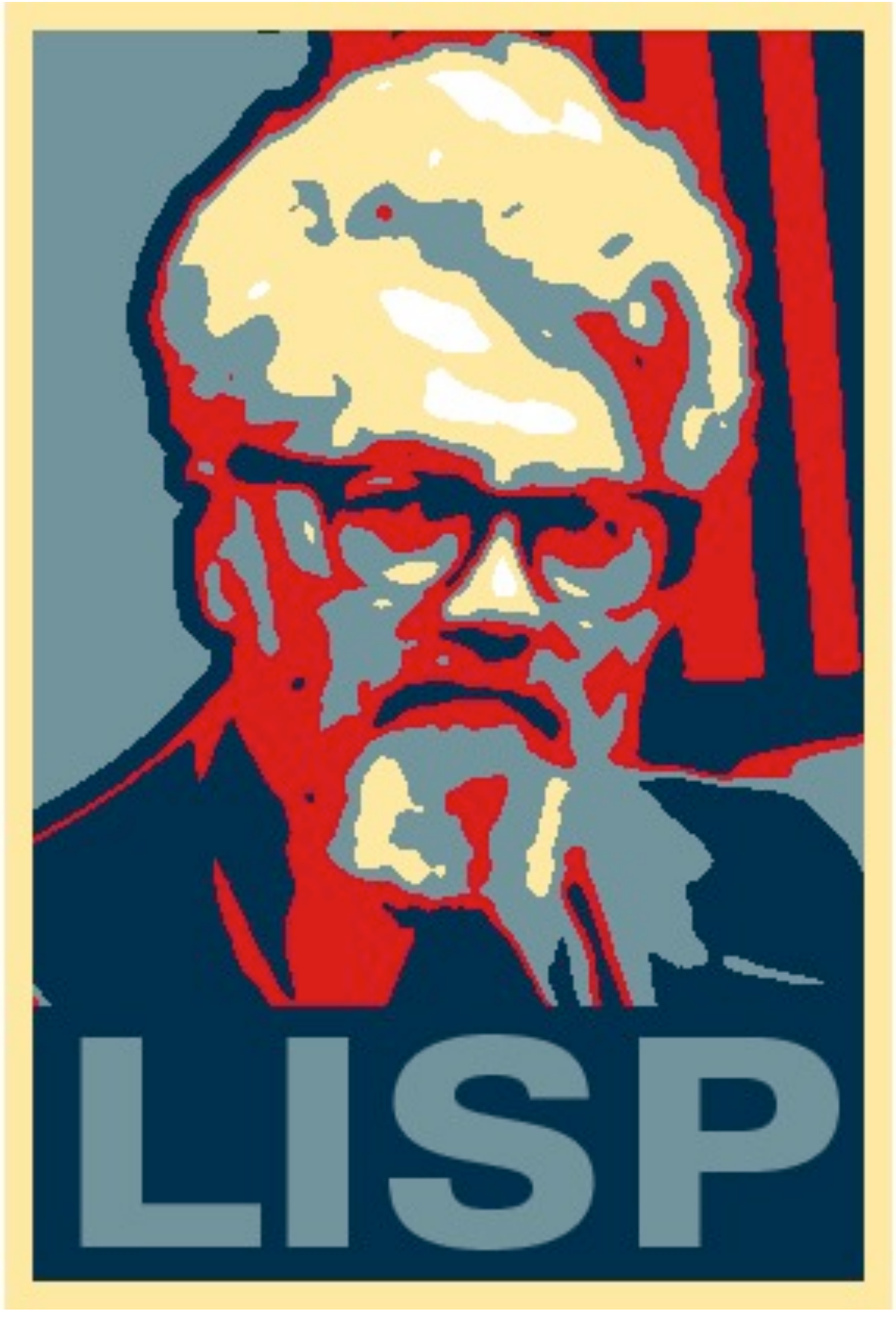


```
<:expr< List.map (fun x -> x+1) [1; 2; 3] >>
```

The expander

```
List.map (fun x -> x+1) [1; 2; 3]
```





1927.09.04 - 2011.10.23

Lisp Macros

Use cases

- **Creating binding forms**
- **Control flow**
- **Icing**

History

LISP

```
(label eval (lambda (expr binds)
              (cond
                ((atom expr) (assoc expr binds))
                ((atom (car expr))
                 (cond
                   ((eq (car expr) (quote quote)) (cadr expr))
                   ((eq (car expr) (quote atom))  (atom   (eval (cadr expr) binds)))
                   ((eq (car expr) (quote eq))    (eq     (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote car))   (car    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cdr))   (cdr    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cons))  (cons   (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote cond))  (eval-cond (cdr expr) binds))
                   (t (eval (cons (assoc (car expr) binds)
                                  (cdr expr))
                            binds))))
                ((eq (caar expr) (quote label))
                 (eval (cons (caddar expr) (cdr expr))
                       (cons (list (cadar expr) (car expr)) binds)))
                ((eq (caar expr) (quote lambda))
                 (eval (caddar expr)
                       (append (pair (cadar expr) (eval-args (cdr expr) binds))
                               binds)))
                (t (assoc expr binds)))))
```

AIM-57

- MACRO Definitions in LISP
- by Timothy Hart
- 1963

LISP with macros

```
(label eval (lambda (expr binds)
              (cond
                ((atom expr) (assoc expr binds))
                ((atom (car expr))
                 (cond
                   ((eq (car expr) (quote quote)) (cadr expr))
                   ((eq (car expr) (quote atom))  (atom   (eval (cadr expr) binds)))
                   ((eq (car expr) (quote eq))    (eq     (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote car))   (car    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cdr))   (cdr    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cons))  (cons   (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote cond))  (eval-cond (cdr expr) binds))
                   (t (eval (cons (assoc (car expr) binds)
                                  (cdr expr))
                            binds))))
                ((eq (caar expr) (quote label))
                 (eval (cons (caddar expr) (cdr expr))
                       (cons (list (cadar expr) (car expr)) binds)))
                ((eq (caar expr) (quote lambda))
                 (eval (caddar expr)
                       (append (pair (cadar expr) (eval-args (cdr expr) binds))
                               binds)))
                ((eq (caar expr) (quote macro))
                 (cond
                   ((eq (cadar expr) (quote lambda))
                    (eval (eval (car (cdddar expr))
                                (cons (list (car (caddar expr))
                                            (cadr expr))
                                      binds))
                          binds)))))))
```

LISP with macros

```
(label eval (lambda (expr binds)
              (cond
                ((atom expr) (assoc expr binds))
                ((atom (car expr))
                 (cond
                   ((eq (car expr) (quote quote)) (cadr expr))
                   ((eq (car expr) (quote atom))  (atom   (eval (cadr expr) binds)))
                   ((eq (car expr) (quote eq))    (eq     (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote car))   (car    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cdr))   (cdr    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cons))  (cons   (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote cond))  (eval-cond (cdr expr) binds))
                   (t (eval (cons (assoc (car expr) binds)
                                  (cdr expr))
                            binds))))
               ((eq (caar expr) (quote label))
                (eval (cons (caddar expr) (cdr expr))
                      (cons (list (cadar expr) (car expr)) binds)))
               ((eq (caar expr) (quote lambda))
                (eval (caddar expr)
                      (append (pair (cadar expr) (eval-args (cdr expr) binds))
                              binds)))
               ((eq (caar expr) (quote macro))
                (cond
                  ((eq (cadar expr) (quote lambda))
                   (eval (eval (car (cdddar expr))
                               (cons (list (car (caddar expr))
                                           (cadr expr))
                                     binds))
                         binds)))))))
```

LISP with macros

```
(label eval (lambda (expr binds)
              (cond
                ((atom expr) (assoc expr binds))
                ((atom (car expr))
                 (cond
                   ((eq (car expr) (quote quote)) (cadr expr))
                   ((eq (car expr) (quote atom))  (atom   (eval (cadr expr) binds)))
                   ((eq (car expr) (quote eq))    (eq     (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote car))   (car    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cdr))   (cdr    (eval (cadr expr) binds)))
                   ((eq (car expr) (quote cons))  (cons   (eval (cadr expr) binds)
                                                          (eval (caddr expr) binds)))
                   ((eq (car expr) (quote cond))  (eval-cond (cdr expr) binds))
                   (t (eval (cons (assoc (car expr) binds)
                                  (cdr expr))
                            binds))))
                ((eq (caar expr) (quote label))
                 (eval (cons (caddar expr) (cdr expr))
                       (cons (list (cadar expr) (car expr)) binds)))
                ((eq (caar expr) (quote lambda))
                 (eval (caddar expr)
                       (append (pair (cadar expr) (eval-args (cdr expr) binds))
                               binds)))
                ((eq (caar expr) (quote macro))
                 (cond
                   ((eq (caadar expr) (quote lambda))
                    (eval (eval (car (cdddar expr))
                                (cons (list (car (caddar expr))
                                            (cadr expr))
                                      binds))
                          binds)))))))
```

Lightweight fn/blocks

```
(defmacro when-not
  [condition & body]
  `(when (not ~condition)
     ~@body))

(when-not (even? 1)
  (println “not even))
;; not even
```

control flow

```
def when_not(condition, &blk)
  yield unless condition
end

when_not(1.even?) {
  puts "not even"
}
## not even
```

```
(defmacro scope
  [vals names & body]
  `(let [[~@names] ~vals]
     ~@body))

(scope (range) [a b]
 (println (str a " and " b)))
;; 0 and 1
```

bindings

```
module Enumerable
  def scope &blk
    yield *(self.to_a)
  end
end

(0..20).scope { | a, b |
  puts "#{a} and #{b}"
}
## 0 and 1
```

Baysick

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
     20 LET ('dist := 100)
     30 LET ('v := 1)
     40 LET ('fuel := 1000)
     50 LET ('mass := 1000)
     60 PRINT "You are a in control of a lunar lander, drifting towards the moon."
     80 PRINT "Each turn you must decide how much fuel to burn."
     90 PRINT "To accelerate enter a positive number, to decelerate a negative"
    100 PRINT "Dist " % 'dist % "km, " % "Vel " % 'v % "km/s, " % "Fuel " % 'fuel
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"
    140 GOTO 100
    150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
    160 LET ('fuel := 'fuel - ABS('burn))
    170 LET ('dist := 'dist - 'v)
    180 IF 'dist > 0 THEN 100
    190 PRINT "You have hit the surface"
    200 IF 'v < 3 THEN 240
    210 PRINT "Hit surface too fast (" % 'v % ")km/s"
    220 PRINT "You Crashed!"
    230 GOTO 250
    240 PRINT "Well done!"
    250 END
    RUN
  }
}
```

Baysick

Symbol

Symbol

Symbol

Symbol

Symbol

Symbol

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)
    60 PRINT "You are a in control of a lunar lander, drifting towards the moon."
    80 PRINT "Each turn you must decide how much fuel to burn."
    90 PRINT "To accelerate enter a positive number, to decelerate a negative"
   100 PRINT "Dist: " % 'dist % "km, " % "Vel " % 'v % "km/s, " % "Fuel " % 'fuel
   110 INPUT 'burn
   120 IF ABS('burn) <= 'fuel THEN 150
   130 PRINT "You don't have that much fuel"
   140 GOTO 100
   150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
   160 LET ('fuel := 'fuel - ABS('burn))
   170 LET ('dist := 'dist - 'v)
   180 IF 'dist > 0 THEN 100
   190 PRINT "You have hit the surface"
   200 IF 'v < 3 THEN 240
   210 PRINT "Hit surface too fast (" % 'v % ")km/s"
   220 PRINT "You Crashed!"
   230 GOTO 250
   240 PRINT "Well done!"
   250 END
   RUN
  }
}
```

Baysick

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)
    60 PRINT "You are a in control of a lunar lander, drifting towards the moon."
    80 PRINT "Each turn you must decide how much fuel to burn."
    90 PRINT "To accelerate enter a positive number, to decelerate a negative"
   100 PRINT "Dist: " % 'dist % "km, " % "Vel " % 'v % "km/s, " % "Fuel " % 'fuel
   110 INPUT 'burn
   120 IF ABS('burn) <= 'fuel THEN 150
   130 PRINT "You don't have that much fuel"
   140 GOTO 100
   150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
   160 LET ('fuel := 'fuel - ABS('burn))
   170 LET ('dist := 'dist - 'v)
   180 IF 'dist > 0 THEN 100
   190 PRINT "You have hit the surface"
   200 IF 'v < 3 THEN 240
   210 PRINT "Hit surface too fast (" % 'v % ")km/s"
   220 PRINT "You Crashed!"
   230 GOTO 250
   240 PRINT "Well done!"
   250 END
   RUN
  }
}
```

Symbol

Symbol

Symbol

Symbol

Symbol

Implicit

Implicit

Implicit

Implicit

Implicit

Implicit

Implicit

Baysick

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)
    60 PRINT "You are a in control of a lunar lander, drifting towards the moon."
    80 PRINT "Each turn you must decide how much fuel to burn."
    90 PRINT "To accelerate enter a positive number, to decelerate a negative"
   100 PRINT "Dist: " % 'dist % "km, " % "Vel " % 'v % "km/s, " % "Fuel " % 'fuel
   110 INPUT 'burn
   120 IF ABS('burn) <= 'fuel THEN 150
   130 PRINT "You don't have that much fuel"
   140 GOTO 100
   150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
   160 LET ('fuel := 'fuel - ABS('burn))
   170 LET ('dist := 'dist - 'v)
   180 IF 'dist > 0 THEN 100
   190 PRINT "You have hit the surface"
   200 IF 'v < 3 THEN 240
   210 PRINT "Hit surface too fast (" % 'v % ")km/s"
   220 PRINT "You Crashed!"
   230 GOTO 250
   240 PRINT "Well done!"
   250 END
   RUN
  }
}
```

Symbol

Symbol

Symbol

Symbol

Symbol

Closure

Closure

Implicit

Implicit

Implicit

Implicit

Implicit

Implicit

Implicit

Baysick

```
object Lunar extends Baysick {
  def main(args: Array[String]) = {
    10 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)
    60 PRINT "You are a in control of a lunar lander, drifting towards the moon."
    80 PRINT "Each turn you must decide how much fuel to burn."
    90 PRINT "To accelerate enter a positive number, to decelerate a negative"
   100 PRINT "Dist: " % 'dist % "km, " % "Vel: " % 'v % "km/s, " % "Fuel: " % 'fuel
   110 INPUT 'burn
   120 IF ABS('burn) <= 'fuel THEN 150
   130 PRINT "You don't have that much fuel"
   140 GOTO 100
   150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
   160 LET ('fuel := 'fuel - ABS('burn))
   170 LET ('dist := 'dist - 'v)
   180 IF 'dist > 0 THEN 100
   190 PRINT "You have hit the surface"
   200 IF 'v < 3 THEN 240
   210 PRINT "Hit surface too fast (" % 'v % ")km/s"
   220 PRINT "You Crashed!"
   230 GOTO 250
   240 PRINT "Well done!"
   250 END
   RUN
  }
}
```

Map lookup

Map lookup

Map lookup

Symbol

Symbol

Symbol

Symbol

Symbol

Closure

Closure

Implicit

Implicit

Implicit

Implicit

Implicit

Implicit

Implicit

Baysick

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)
    60 PRINT "You are a in control of a lunar lander, drifting towards the moon."
    80 PRINT "Each turn you must decide how much fuel to burn."
    90 PRINT "To accelerate enter a positive number, to decelerate a negative"
   100 PRINT "Dist: " % 'dist % "km, " % "Vel " % 'v % "km/s, " % "Fuel " % 'fuel
   110 INPUT 'burn
   120 IF ABS('burn) <= 'fuel THEN 150
   130 PRINT "You don't have that much fuel"
   140 GOTO 100
   150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
   160 LET ('fuel := 'fuel - ABS('burn))
   170 LET ('dist := 'dist - 'v)
   180 IF 'dist > 0 THEN 100
   190 PRINT "You have hit the surface"
   200 IF 'v < 3 THEN 240
   210 PRINT "Hit surface too fast (" % 'v % ")km/s"
   220 PRINT "You Crashed!"
   230 GOTO 250
   240 PRINT "Well done!"
   250 END
   RUN
  }
}
```

Map lookup

Map lookup

Map lookup

Symbol

Symbol

Symbol

Symbol

Symbol

Closure

Closure

Implicit

Implicit

Implicit

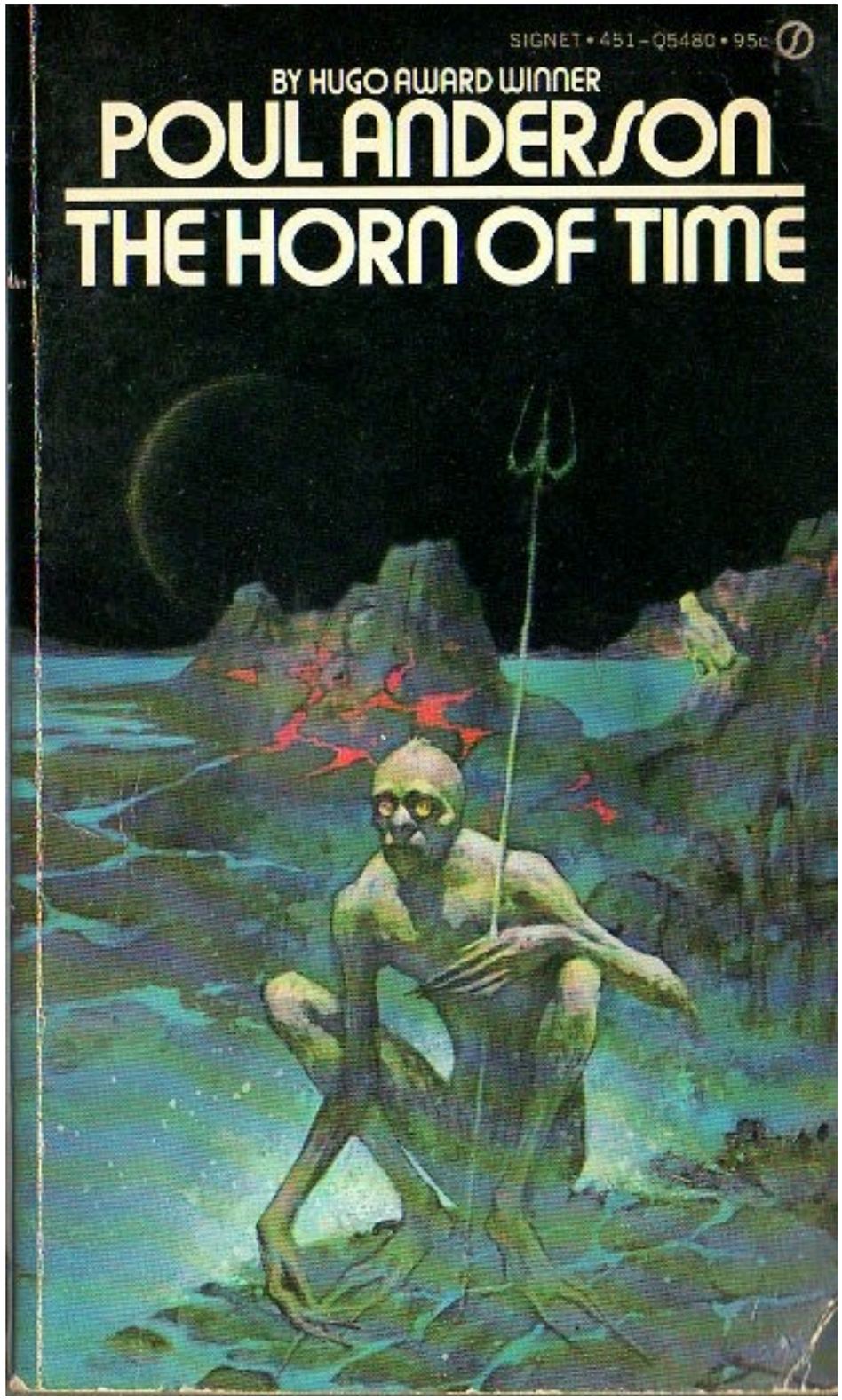
Implicit

Implicit

Implicit

Implicit

TRAMPOLINE!!!



Mapping Dilemma

One of the issues here is actually encapsulation: without macros, the ability to create abstractions is limited by the degree to which underlying language constructs peek through the abstraction barrier.

Use cases

- **Creating binding forms**
- **Control flow**
- **Icing**

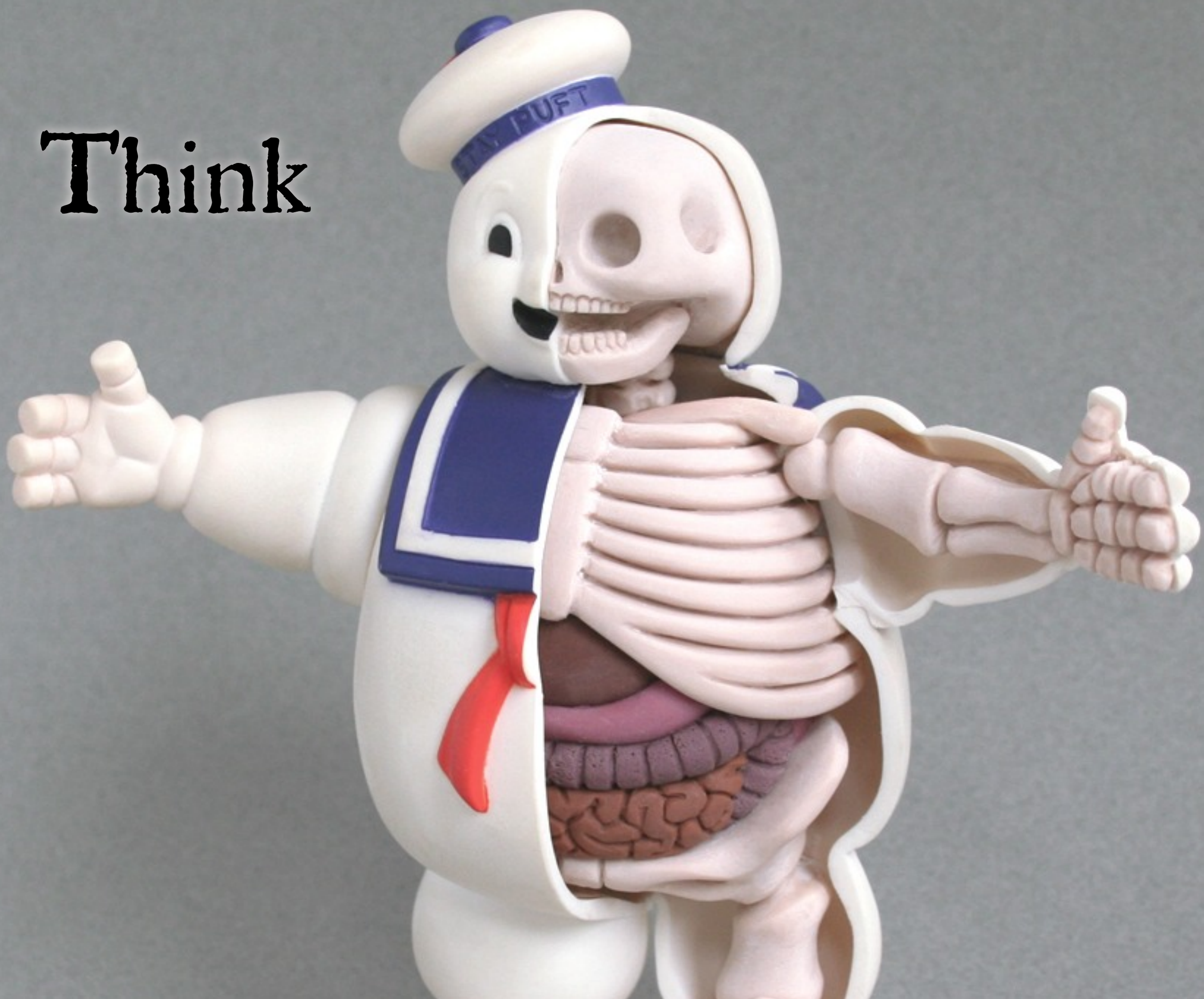
"All legitimate uses of macros are legitimate"

— Yogi Berra

Use cases

- **Creating binding forms**
- **Control flow**
- **Abstraction**
- **Transformation (Houdini, boilerplate)**
- **Optimization**
- **True power awesomeness**
- **Icing**

Think



What do you want?

- **A way to define local bindings**
- **Properly scoped**

Do you need a macro?

```
(defn build-memoizer
  ([cache-factory f & args]
     (let [cache (atom (apply cache-factory f args))]
       (with-meta
        (fn [& args]
          (let [cs (swap! cache through f args)]
            @(fogus.clache/lookup cs args)))
        {:unk cache
         :unk-orig f}))))

;; usage

(defn memo
  ([f] (memo f {}))
  ([f seed]
     (build-memoizer
       basic-cache-factory
       f
       seed)))
```

What does it look like?

```
(LET ([rise 1]
      [run  2])
  (/ rise run))

;=> 1/2
```

What is it really?

```
((fn [rise]
   ((fn [run]
      (/ rise run))
    2))
 1)

;=> 1/2
```

Type

```
(defn foldr [f acc [h & t]]
  (if h
    (f h (foldr f acc t))
    acc))

(defmacro LET [bindings & body]
  (foldr (fn [[n v] subexpr]
           `((fn [~n] ~subexpr) ~v))
         `(do ~@body)
         bindings))
```

Quick Tip: binding idiom

```
(defmacro LET [bindings & body]
  (assert (vector? bindings))
  (reduce (fn [form [n v]] `((fn [~n] ~form) ~v))
          `(do ~@body)
          (partition 2 bindings)))

(LET [rise 1
      run  2]
  (/ rise run))

;=> 1/2
```

Hygiene

Hygiene

An hygienic macro is one where the meanings of symbols that aren't parameters to the macro are bound at the definition site rather than the expansion site.

— James Iry @ Lambda the Ultimate

Degenerative Case I

```clojure
(defmacro tis-true? [a]
  `(when a
     :twas-true))

;; somewhere
(def a false)

;; somewhere else
(tis-true? true)
;=> nil
```

Degenerative Case 1

```
(when my-ns/a
  :twas-true)
```

Degenerative Case I

```clojure
(defmacro tis-true? [a]
  `(when ~a
     :twas-true))

(tis-true? true)
;=> :twas-true
```

VS.

Degenerative Case 2

```clojure
(defmacro awhen
  [condition & body]
  `(if-let [~'it ~condition]
     ~@body))

(awhen (:a {:a 42})
  (println it))
; 42
```

Degenerative Case 2

```clojure
(defmacro awhen
  [condition & body]
  `(if-let [~'it ~condition]
     ~@body))

(awhen (:a {:a 42})
  (println it))
; 42
```

Degenerative Case 2

```clojure
(def it :not-it)

(awhen (:a {:a 42})
  (println it))
; 42
```

Better

```clojure
(def it :not-it)

(when-let [it (:a {:a 42})]
  (println it))
; 42
```

~ ,

VS.

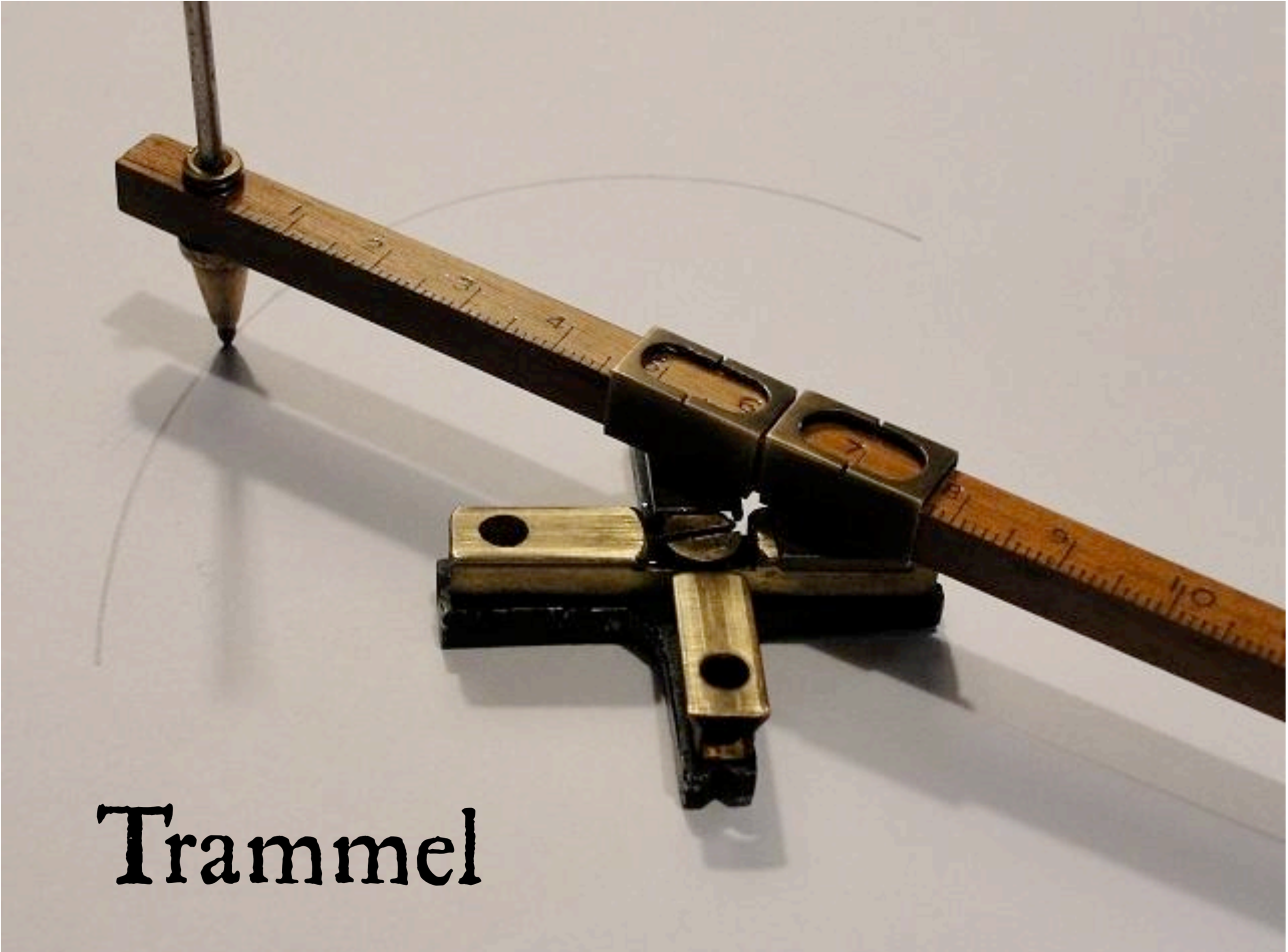
Use

Use

Abuse


DSL
Domain-specific Languages


MSL
Mood-specific Languages
-- Ian Piumarta



Trammel

Step 0:
What did I want?

- **Goals**
  - **New :pre/:post syntax**
  - **Decomplected contracts**
  - **Invariants on records, types, references**
  - **Better error reporting**
  - **Misc.**

Step 1:
Did I need a macro?

Yes.

Step 1:
Did I need a macro?

✓ **Yes.**

Second-class Forms

Macros are first-class at making second-class forms

```clojure
(defrecord Date [year month day])
(defn Date? [d] (= klass (type d)))

(defn new-Date
  [& {:or {year 1970, month 1, day 1},
      :as m
      :keys [year month day]}]
  {:pre [(every? number? [year month day])
         (< month 13)
         (> month 0)
         (< day 32)
         (> day 0)]}
  (->
    (Date. 1970 1 1)
    (merge m)))
```

Macros

Why Wait for Rich?

Trammel is...

New :pre/:post Syntax

```
(defconstrainedfn sqr
[n],[number? (not= 0 n) => pos? number?]
(* n n))
```

Trammel is...

Decomplected Contracts

```
(defn gregorian-last-day-of-month
  ([d] (gregorian-last-day-of-month (:month d) (:year d)))
  ([month year]
    (if (and (= month 2)
             (leap-year? year))
      29
      (nth *canonical-days* (dec month)))))

(provide/contracts
 [gregorian-last-day-of-month
  "Gregorian last day calculation constraints"
  [d]   [Date? :month :year => number? pos?]
  [m y] [all-positive? => number? pos?]])
```

Trammel is...

Record Invariants

```
(defconstrainedrecord Date [year 1970 month 1 day 1]
  [(every? number? [year month day])
   (< month 13) (> month 0)
   (< day 32) (> day 0)])


(new-Date :month 11 :day 12 :year “AD”)

;; Assert failed: (every? number? [year month day])
```

Trammel is...

Better Error Reporting

```
(sqr 10)
;=> 100

(sqr :a)
; Pre-condition Error: (pos? :a)

(sqr 0)
; Post-condition Error: (pos? 0)
```

Piecewise Transformation

Piecewise Transformation

```
(defmacro defconstrainedfn
  [name & body]
  (let [mdata (if (string? (first body))
                {:doc (first body)}
                {})
        body  (if (:doc mdata)
                (next body)
                body)
        body  (if (vector? (first body))
                (list body)
                body)
        body  (for [[args cnstr & bd] body]
                (list* args
                       (if (vector? cnstr)
                         (second (build-cnstr-map args cnstr))
                         cnstr)
                       bd))]
    `(defn ~name
       ~(str (:doc mdata))
       ~@body)))
```

Piecewise Transformation

```
(defmacro defmulti
  [mm-name & options]
  (let [docstring   (if (string? (first options))
                      (first options)
                      nil)
        options     (if (string? (first options))
                      (next options)
                      options)
        m           (if (map? (first options))
                      (first options)
                      {})
        options     (if (map? (first options))
                      (next options)
                      options)
        dispatch-fn (first options)
        options     (next options)
        m           (if docstring
                      (assoc m :doc docstring)
                      m)
...)
```

Piecewise Transformation

as-futures

```
(as-futures [<arg-name> <all-args>]
  <actions-using-arg-name>
  :as <results-name>
 =>
  <actions-using-results>)
```

Piecewise Transformation

as-futures

```
(defn sum-facts [& ns]
  (as-futures [n ns]
   (reduce * (range 1 n))
   :as results
  =>
  (reduce #(+ % (deref %2))
          0
          results)))

(sum-facts 100 13 123 56 33)
```

```
;=>
98750442008336013624115798714482080125644041
37071685821402414202949370696061281065394095
61103888585087569526895318024046688120956989
53738011044836999122093443893501757150547517
55177029244305801263900160O
```

Piecewise Transformation

as-futures

```clojure
(defmacro as-futures [[a args] & body]
  (let [parts             (partition-by #{'=>} body)
        [acts _ [res]]    (partition-by #{:as} (first parts))
        [_ _ task]        parts]
    `(let [~res (for [~a ~args] (future ~@acts))]
       ~@task)))
```

Piecewise Transformation

as-futures

```
(defmacro as-futures [[a args] & body]
  (let [parts                  (partition-by #{'=>} body)
        [acts _ [res]]  (partition-by #{:as} (first parts))
        [_ _ task]      parts]
    `(let [~res (for [~a ~args] (future ~@acts))]
       ~@task)))
```

Piecewise Transformation

as-futures

```clojure
(defn- extract-results-name [[_ [nom]]]
  (if nom
    nom
    (throw (Exception. "Missing :as clause!"))))

(defmacro as-futures [[a args] & body]
  (let [parts          (partition-by #{'=>} body)
        [acts & as]    (partition-by #{:as} (first parts))
        res            (extract-results-name as)
        [_ _ task] parts]
    `(let [~res (for [~a ~args] (future ~@acts))]
       ~@task)))
```

What lies beneath...

Primacy of Semantics

I also regard syntactical problems as essentially irrelevant to programming languages at their present stage ... the urgent task in programming languages is to explore the field of semantic possibilities.

— Christopher Strachey - “Fundamental Concepts in Programming Languages”

A Contract

```clojure
(defn sqr-contract [f n]
  {:pre  [(number? n)]
   :post [(number? %) (pos? %)]}
  (f n))

(defn sqr-actions [n]
  (* n n))

(def sqr (partial sqr-contract sqr-actions))

(sqr :a)
;; AssertionError (number? n)
```

Piecewise Contracts

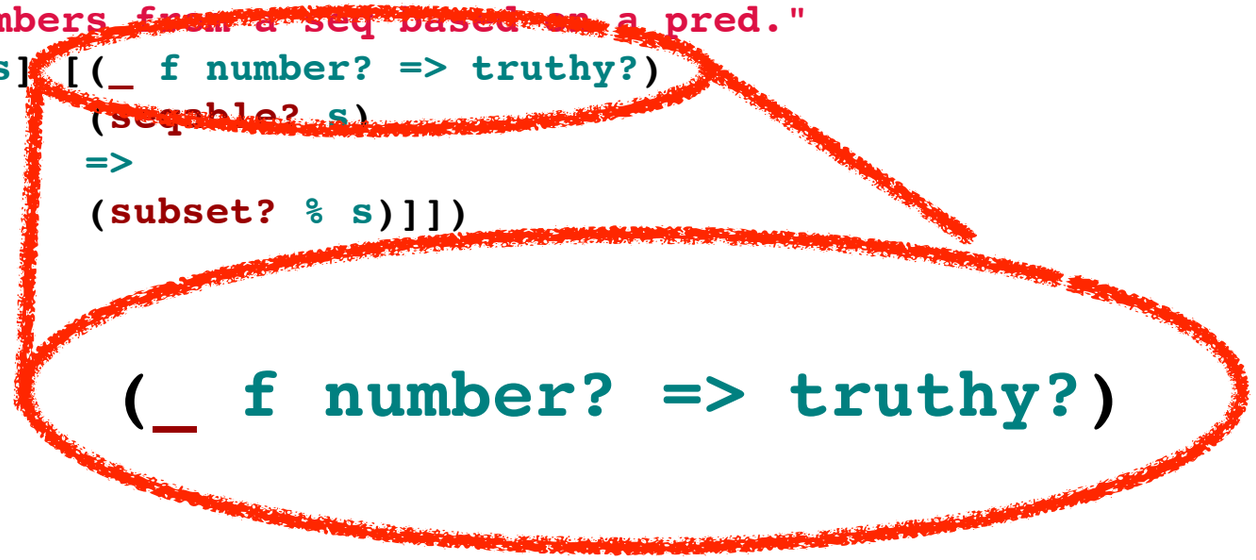
```clojure
(defn sqr-contract-zero [f n]
  {:pre  [(not= 0 n)]}
  (f n))

(def sqr (partial sqr-contract-zero
                  (partial sqr-contract
                           sqr-actions)))

(sqr :a)
;; AssertionError (number? n)

(sqr 0)
;; AssertionError (not= 0 n)

(sqr 10)
;=> 100
```

HOF Contracts

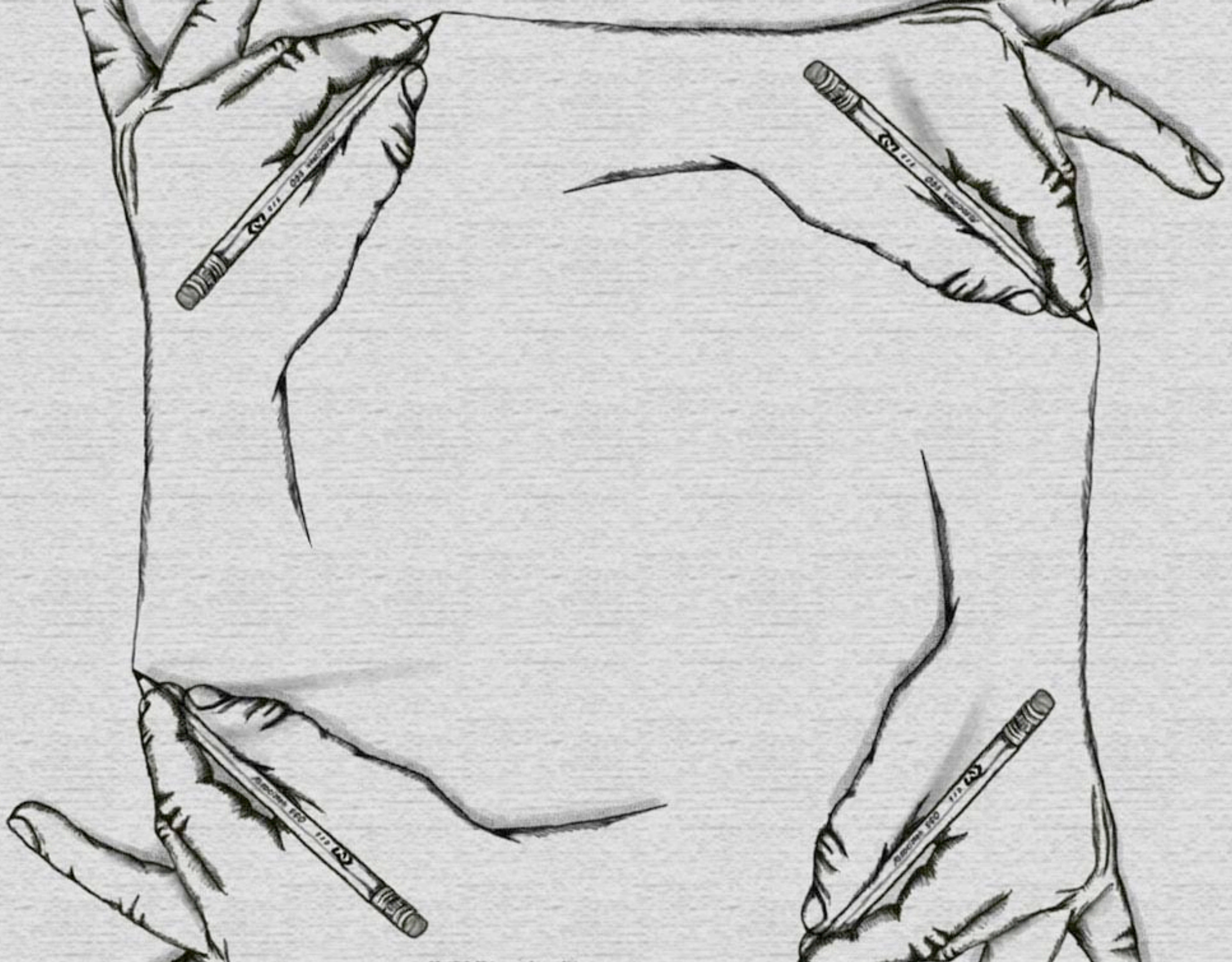
```
(provide/contracts
 [choose-numbers
  "Contract for a function that chooses
   numbers from a seq based on a pred."
  [f s] [(_ f number? => truthy?)
         (seqable? s)
         =>
         (subset? % s)]])
```

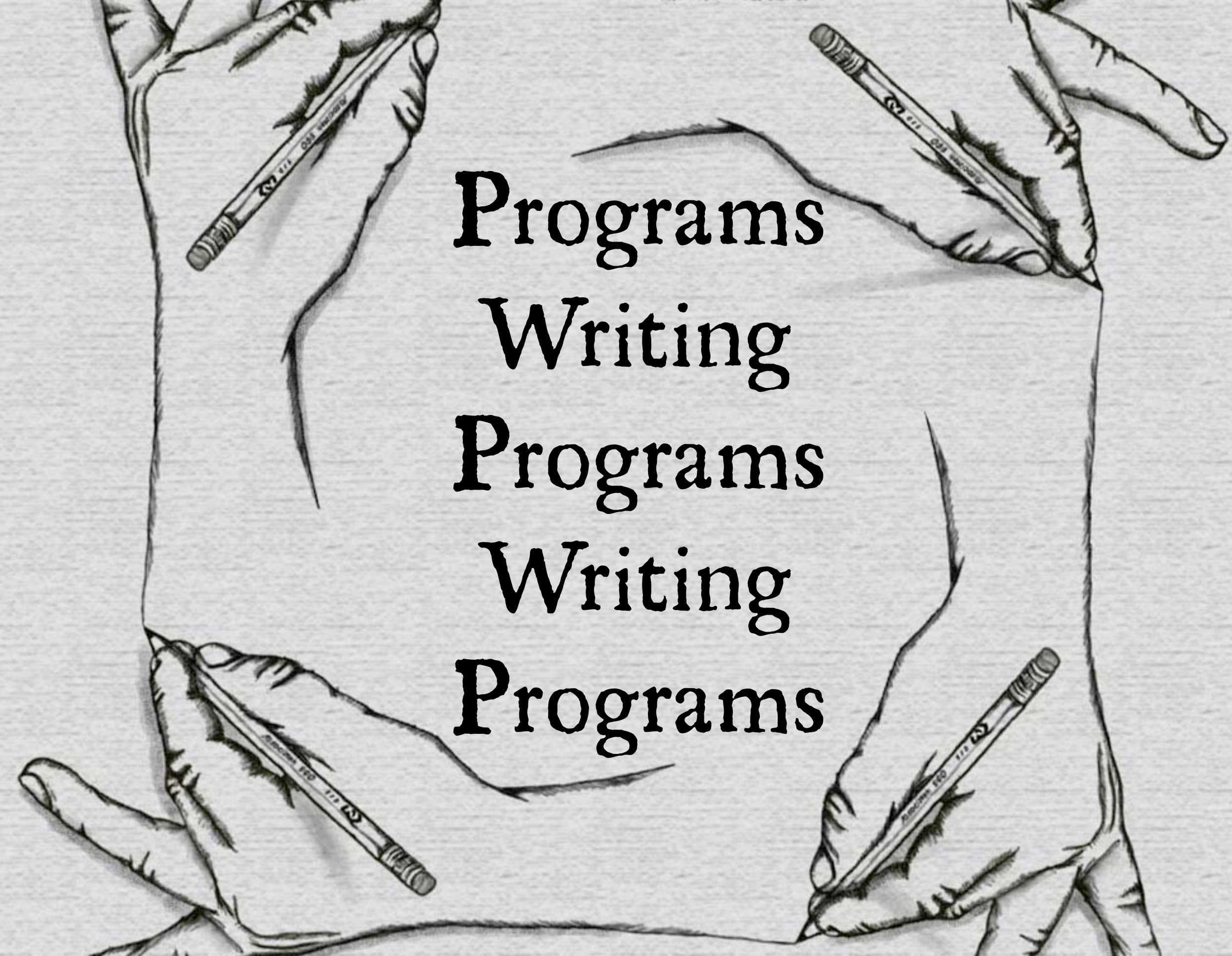
HOF Contracts

```
(provide/contracts
 [choose-numbers
  "Contract for a function that chooses
   numbers from a seq based on a pred."
  [f s] [(_ f number? => truthy?)
         (seqable? s)
         =>
         (subset? % s)]])
```



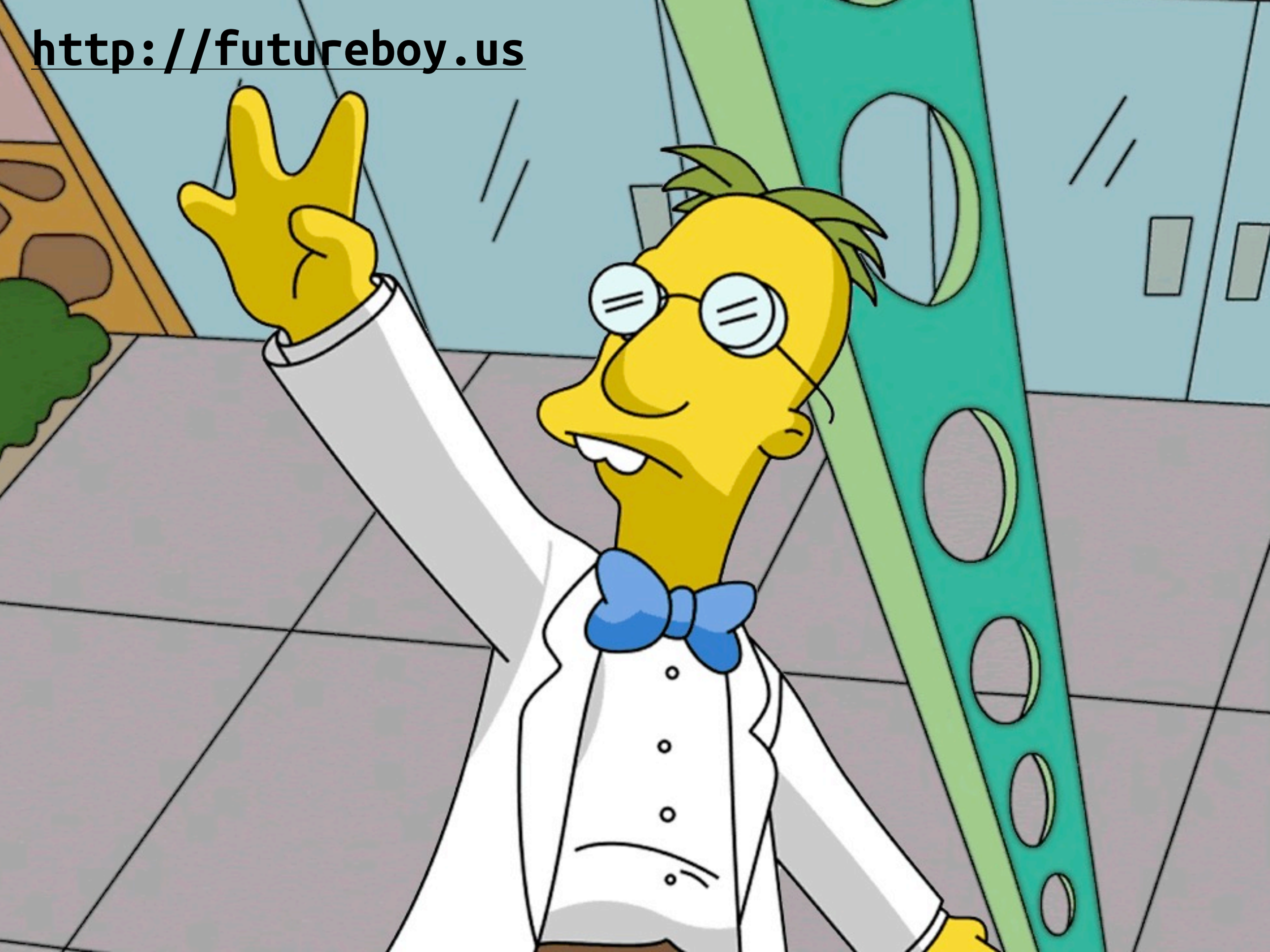
```
(set! *assert* false)
```

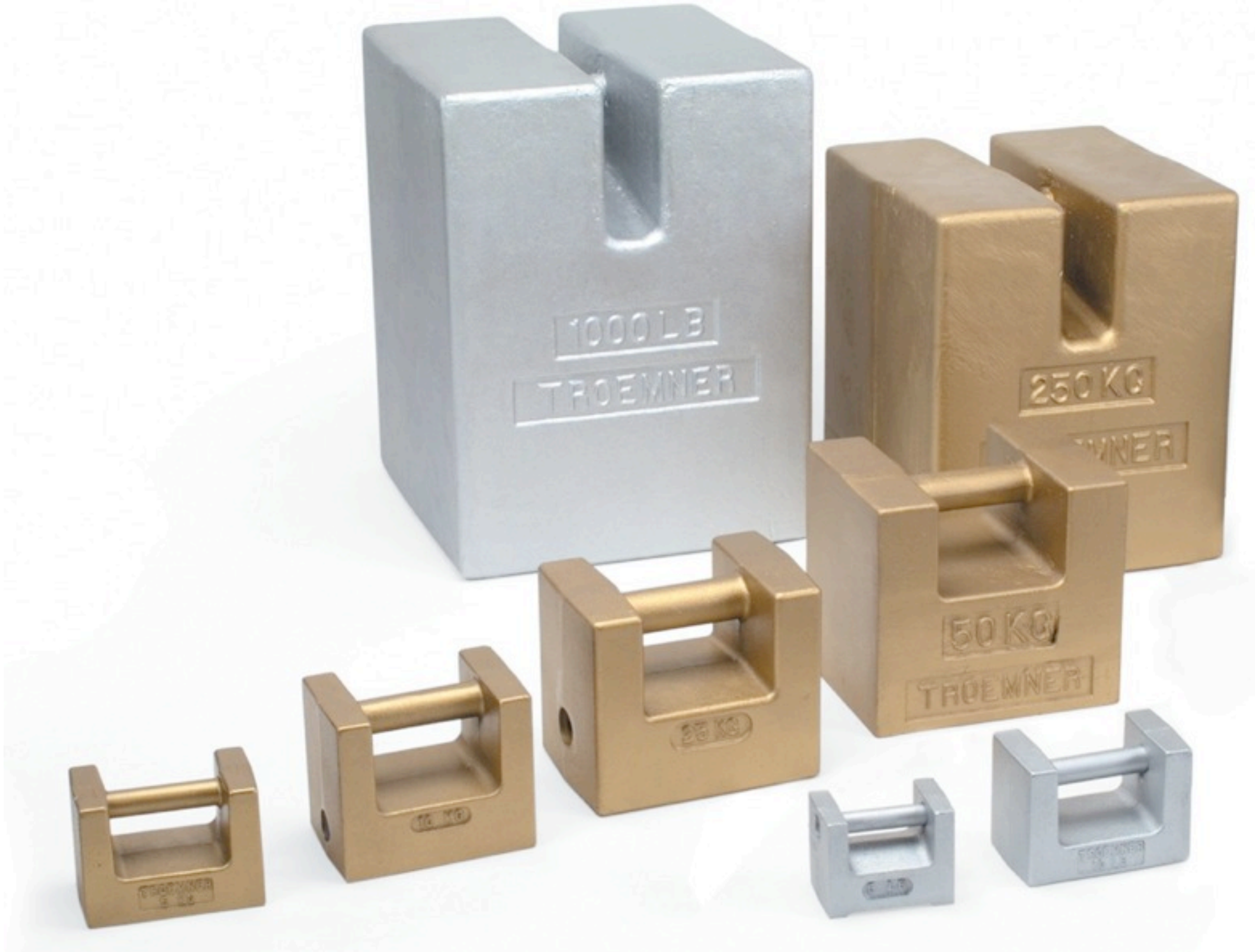



Programs
Writing
Programs

Minderbinder

http://futureboy.us




1000 LB
TROEMNER
250 KG
50 KG
TROEMNER

Unit Conversion

```clojure
(defn meters->feet [m]
  (* 1250/381 m))

(defn meters->yards [m]
  (/ (meters->feet m) 3))

(defn meters->inches [m]
  (* 12 (meters->feet m)))
```

```clojure
(meters->feet 10)
;=> ~ 32.81

(meters->yards 10)
;=> ~ 10.9

(meters->inches 10)
;=> ~ 393.7
```

Unit Conversion

feet → meters

inches → meters

yards → meters

Unit Conversion

centimeters → shackles

ramsden-chains → fathoms

feet → meters

inches → meters

yards → meters

feet → kilometers

yards → centimeters

old-british-fathom → meters

Unit Conversion

centimeters → shackles

ramsden-chains → fathoms

feet → meters

inches → meters

yards → meters

feet → kilometers

yards → centimeters

old-british-fathom → meters

Unit Conversion

CM->MI

MM->M

F->IN

IN->Y

F->M

Y->M

CM->Y

F->MM

M->MM

CM->M

F->Y

centimeters→shackles

ramsden-chains→fathoms

feet→meters

inches→meters

yards→meters

feet→kilometers

yards→centimeters

old-british-fathom →meters

Unit Conversion

bit->nibble

megabyte

megabyte

byte->megabyte

octet->megabyte

exabyte

bit->byte

CM->MI

MM->M

F->IN

IN->Y

F->M

Y->M

CM->Y

F->MM

M->MM

CM->M

F->Y

centimeters→shackles

ramsden-chains→fathoms

feet→meters

inches→meters

yards→meters

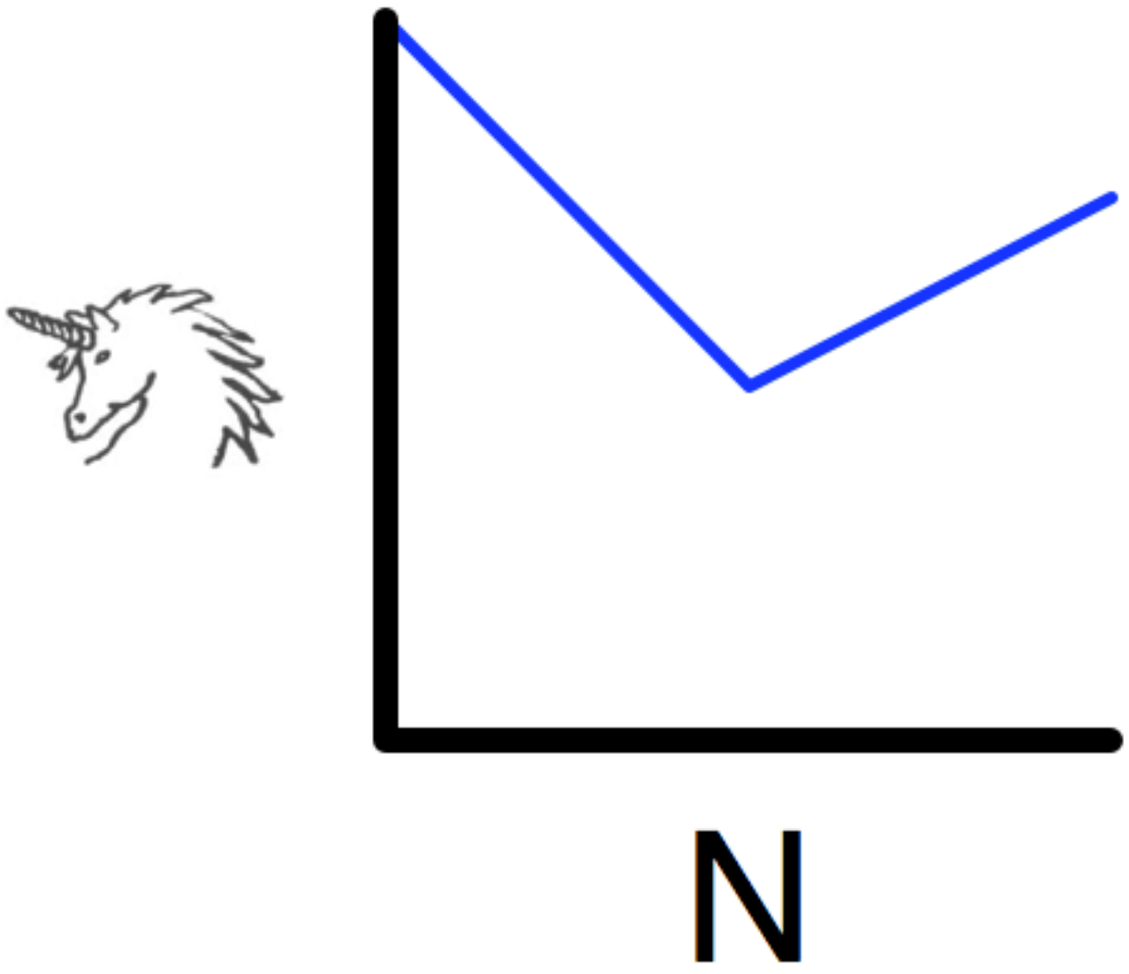
feet→kilometers

yards→centimeters

old-british-fathom→meters


DonKeyHotey

Boilerplate



Length Specification

- NIST Special Publication 330, 2008 Edition
- Checking the Net Contents of Packaged Goods - NIST 133

Unit of length

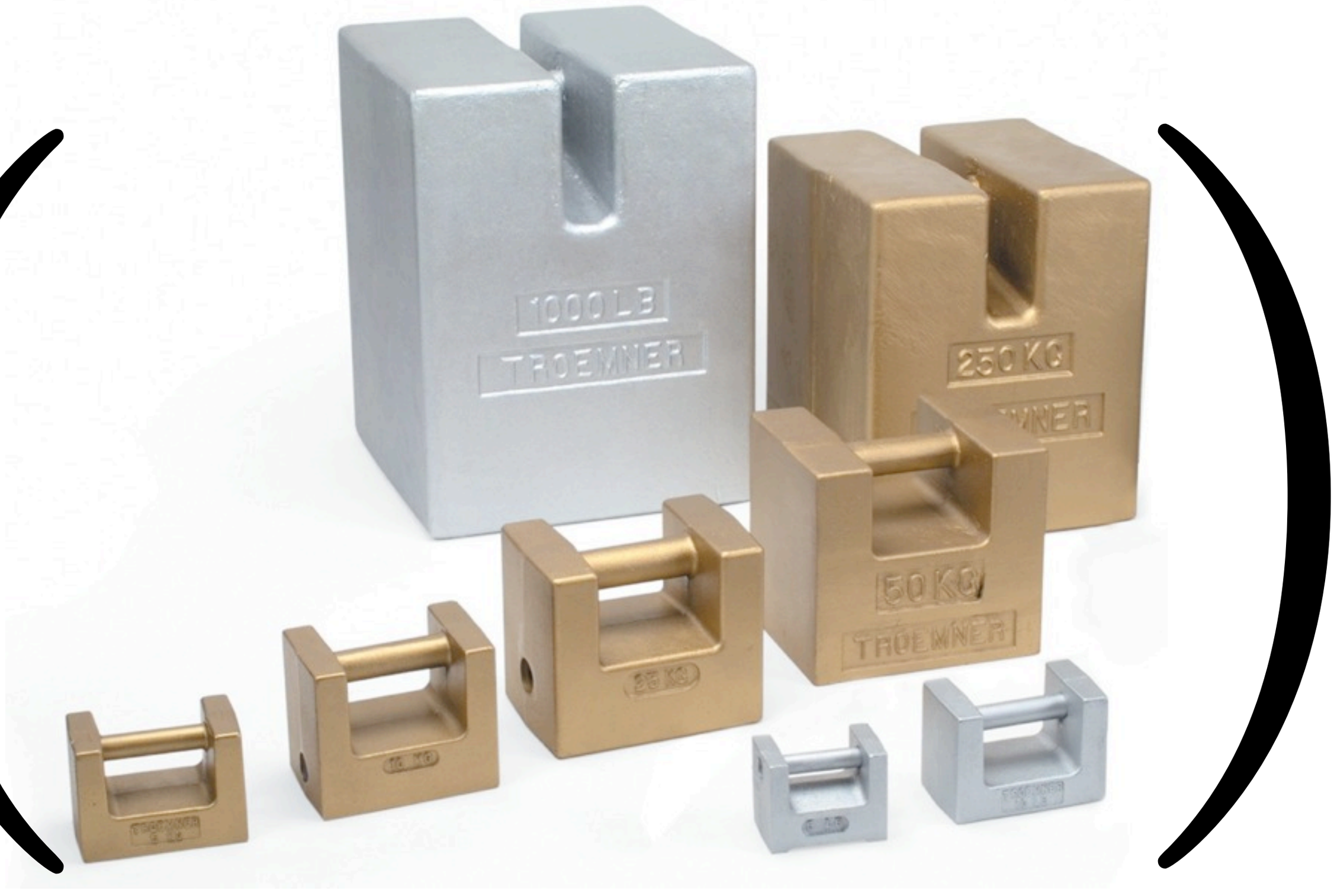
Base unit: meter

The meter is the length of the path travelled by light in vacuum during a time interval of 1/299,792,458 of a second.

1 inch == 0.0254 meters

1 foot == 12 inches

1 yard == 3 feet


1000 LB
TROEMNER
250 KG
50 KG
TROEMNER

Length Specification

- NIST Special Publication 330, 2008 Edition
- Checking the Net Contents of Packaged Goods - NIST 133

(Unit of length

[Base unit: meter]

The meter is the length of the path travelled by light in vacuum during a time interval of 1/299,792,458 of a second.

[1 inch == 0.0254 meters]

[1 foot == 12 inches]

[1 yard == 3 feet])

Primacy of Syntax

one could say that all semantics is being represented as syntax
... semantics has vanished entirely to be replaced with pure syntax.
— John Shutt - “Primacy of Syntax”

Length Specification

- NIST Special Publication 330, 2008 Edition
- Checking the Net Contents of Packaged Goods - NIST 133

(unit-of length meter

The meter is the length of the path travelled by light in vacuum during a time interval of 1/299,792,458 of a second.

inch == 0.0254 meter

foot == 12 inch

yard == 3 foot)

Primacy of Data

Acceptable or not, sir, it is the truth.
— Data - ST:TNG “Coming of Age”

Length Specification

- NIST Special Publication 330, 2008 Edition
- Checking the Net Contents of Packaged Goods - NIST 133

(unit-of length meter

The meter is the length of the path travelled by light in vacuum during a time interval of 1/299,792,458 of a second.

inch == 0.0254 meter

foot == 12 inch

yard == 3 foot)

Length Specification

- NIST Special Publication 330, 2008 Edition
- Checking the Net Contents of Packaged Goods - NIST 133

(unit-of ‘length ::meter

“The meter is the length of the path travelled by light in vacuum during a time interval of 1/299,792,458 of a second.”

::inch ‘== 0.0254 ::meter

::foot ‘== 12 ::inch

::yard ‘== 3 ::foot)

7 x 6

42

From the DSL

```
(defunits-of  length   ::meter
 "The meter is the length of
 the path travelled by light in
 vacuum during a time interval
 of 1/299,792,458 of a second."

   ::inch  0.0254
   ::foot  [12 ::inch]
   ::yard [3 ::foot])
```

To a map

```
{::meter 1,
 ::inch 0.0254,
 ::foot 0.3048,
 ::yard 0.9144}
```

To another macro

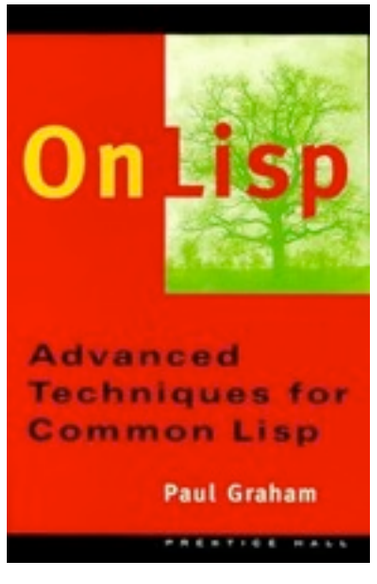
```
(defmacro unit-of-length
[quantity unit]
  `~(*
      (case unit
        ::meter 1
        ::inch 0.0254
        ::foot 0.3048
        ::yard 0.9144)))
```

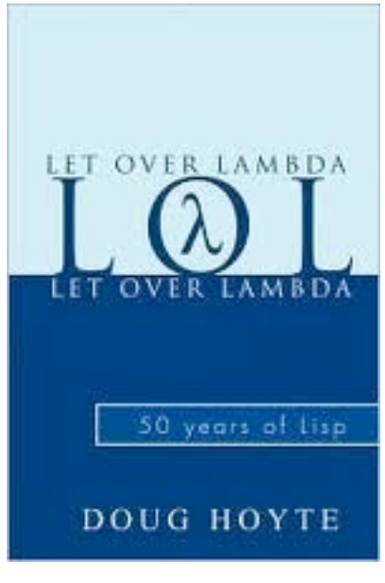
To a use

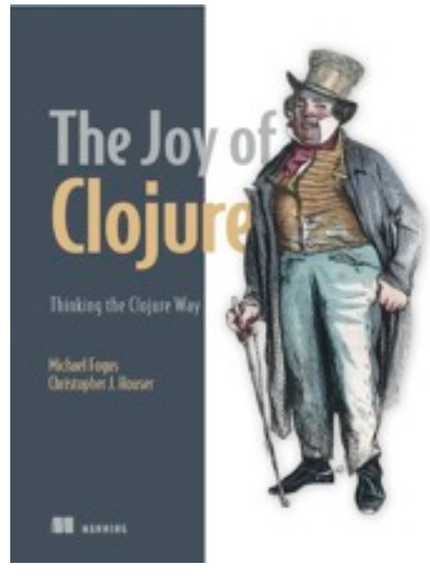
```
(unit-of-length 1 ::yard)
```

0.9144

Learn More

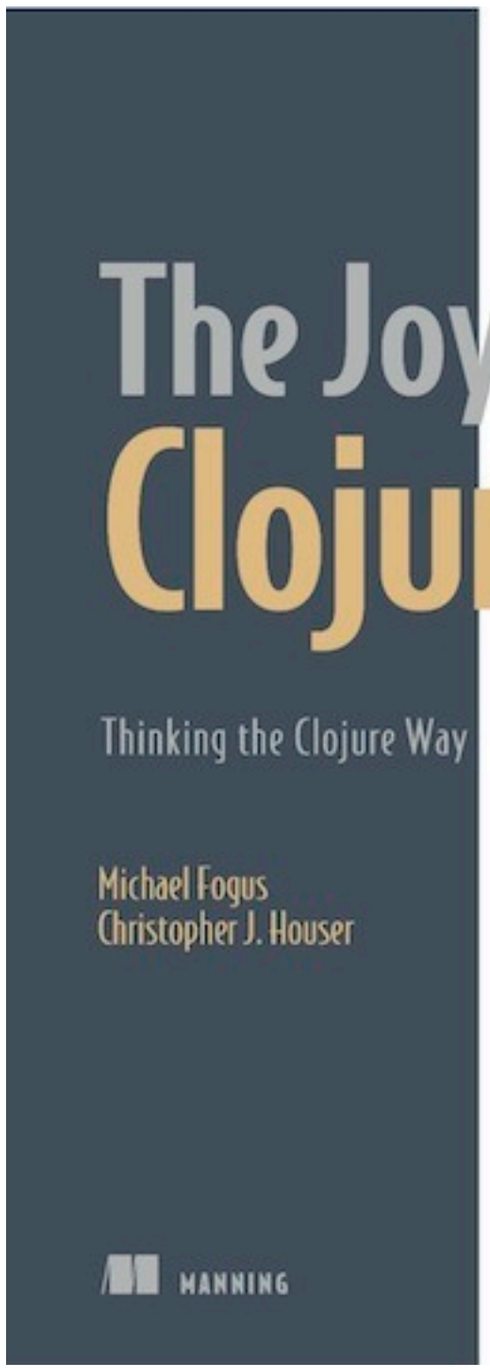






- http://macronomicon.org
- “One Day Compilers” by Graydon Hoare
- http://github.com/fogus/trammel
- http://github.com/fogus/minderbinder
- "Hygienic macros through explicit renaming" by William Clinger

Questions?



http://joyofclojure.com
@fogus

- **Thanks to**
  - **Rich Hickey & Clojure/core**
  - **Jamie Kite**
  - **Clojure/dev**
  - **Relevance**
  - **CAPCLUG**
  - **Manning Publishing**
  - **Wife and kids**
  - **You**