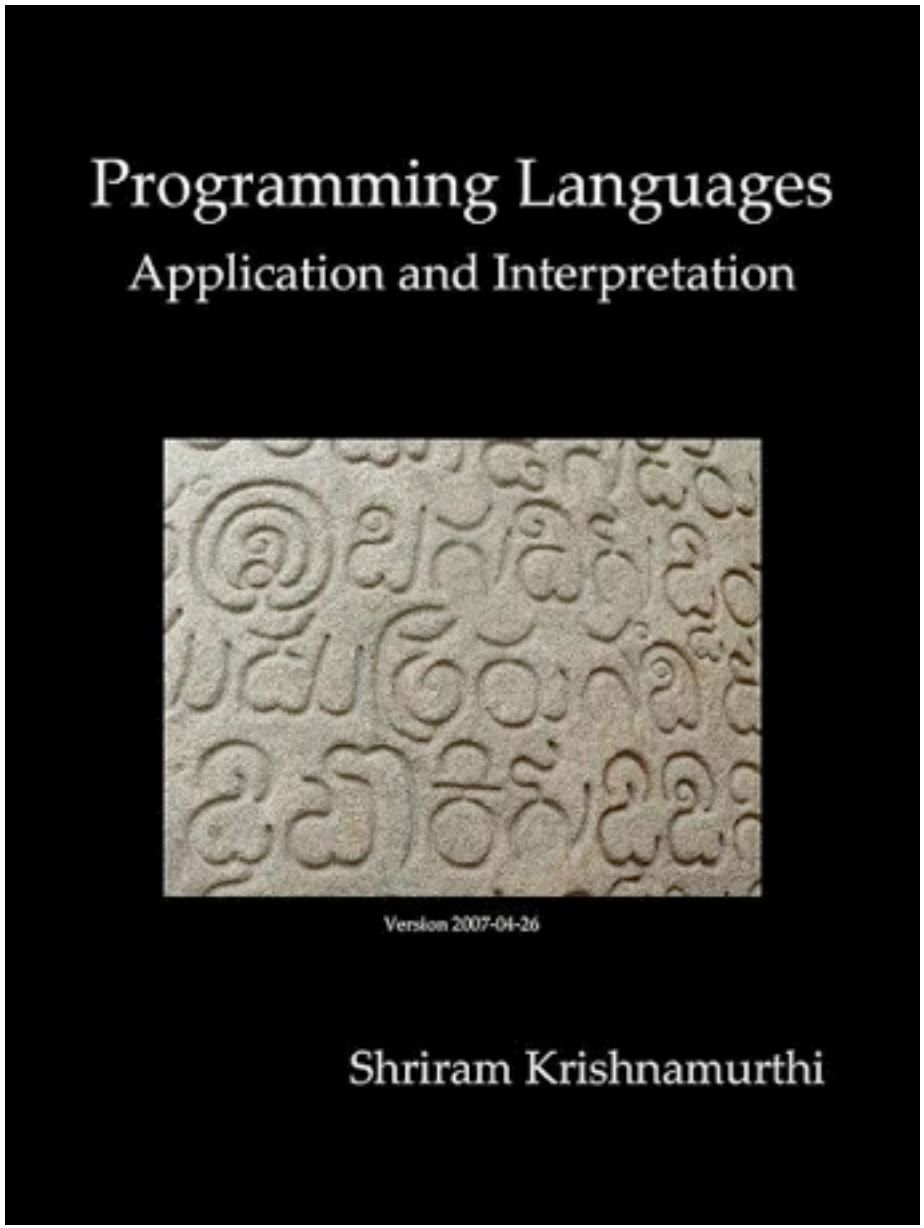
#lang play

Éric Tanter & Ismael Figueroa

Pleiad

University of Chile

undergrad PL course @ U.Chile



2/3

Object-Oriented Programming Languages: Application and Interpretation

last updated: Friday, November 9th, 2012

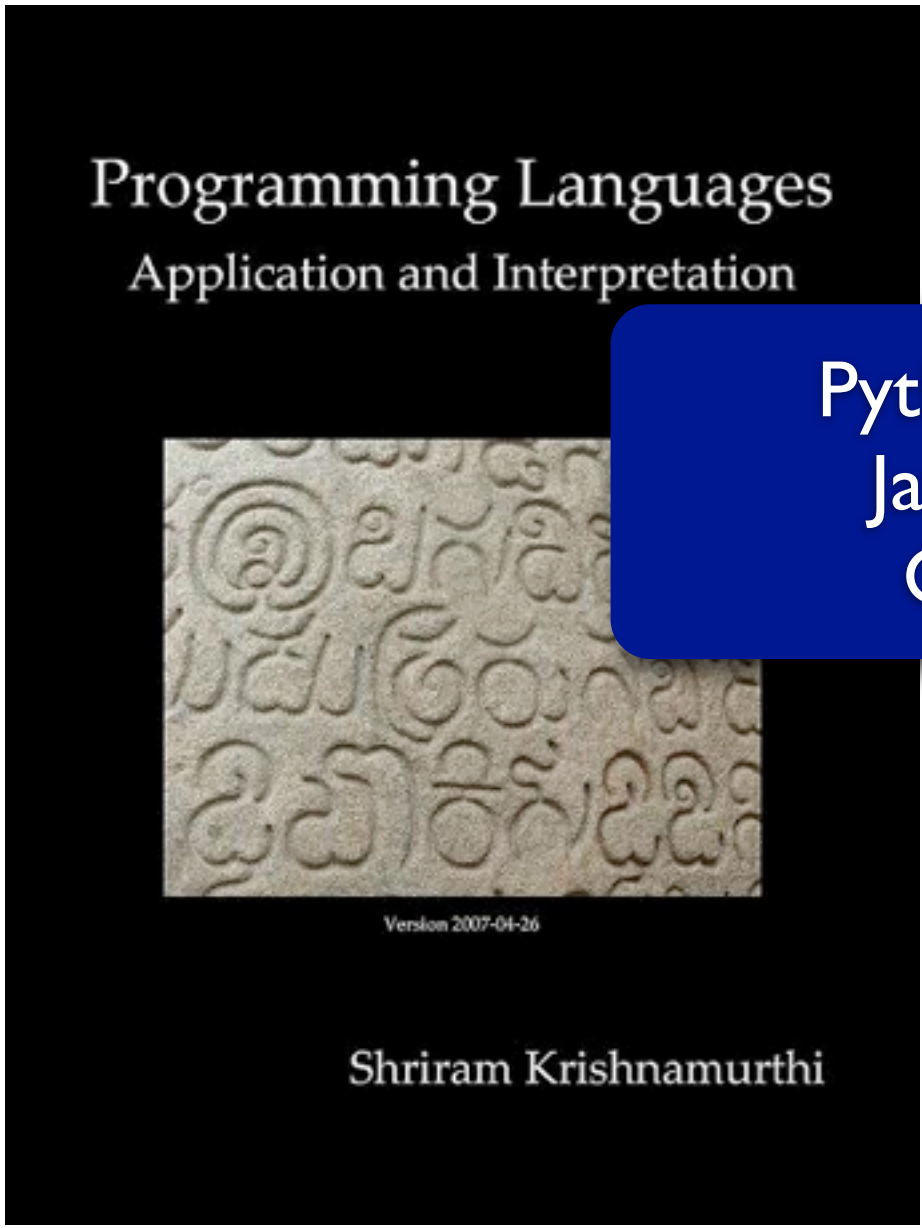
This booklet exposes fundamental concepts of object-oriented programming languages in a constructive and progressive manner. It follows the general appr of the PLAI book by Shriram Krishnamurthi (or at least I'd like to think it does The document assumes familiarity with the following Parts of PLAI: I-V (espe first-class functions, lexical scoping, recursion, and state), as well as XII (macr

OOPLAI is also available in PDF version. Note however that OOPLAI is subje change at any time, so accessing it through the web is the best guarantee to be viewing the latest version.

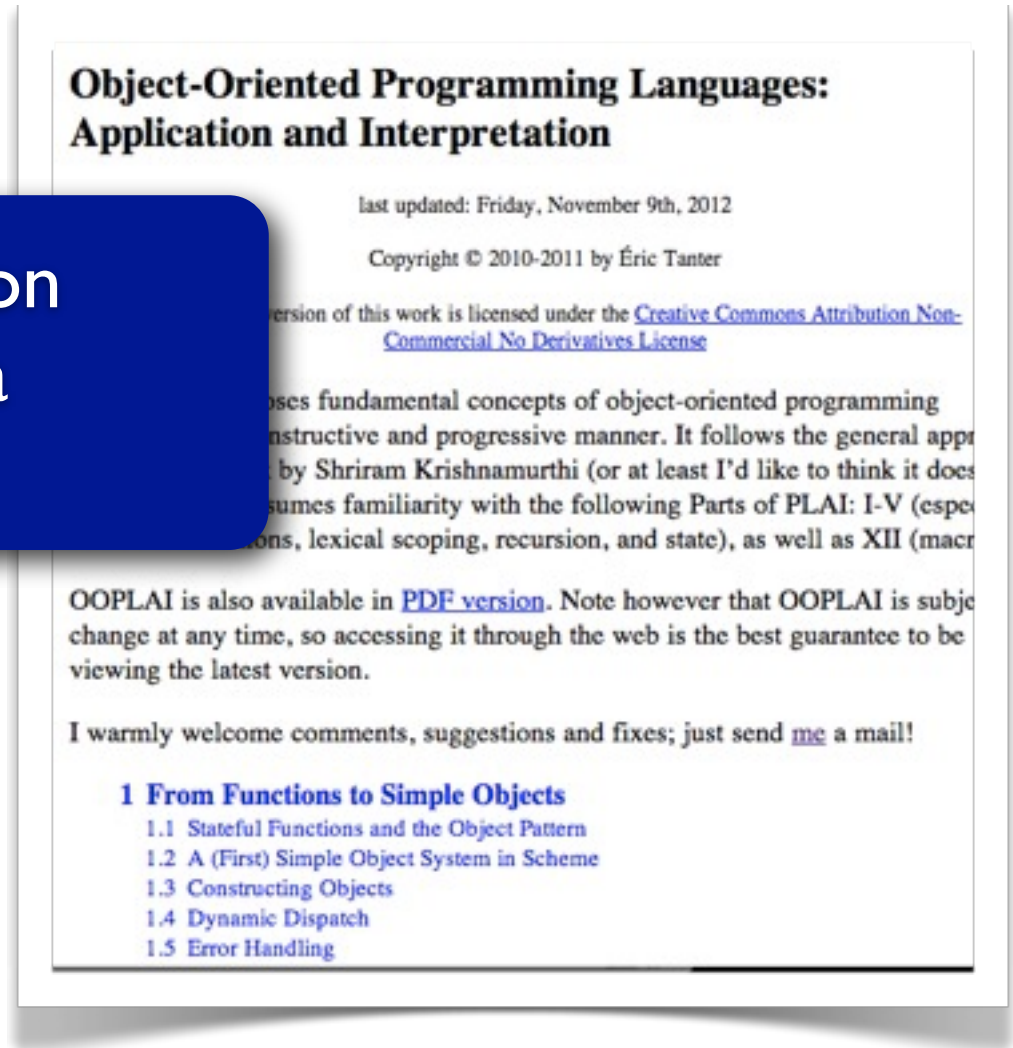
I warmly welcome comments, suggestions and fixes; just send me a mail!

1/3

undergrad PL course @ U.Chile



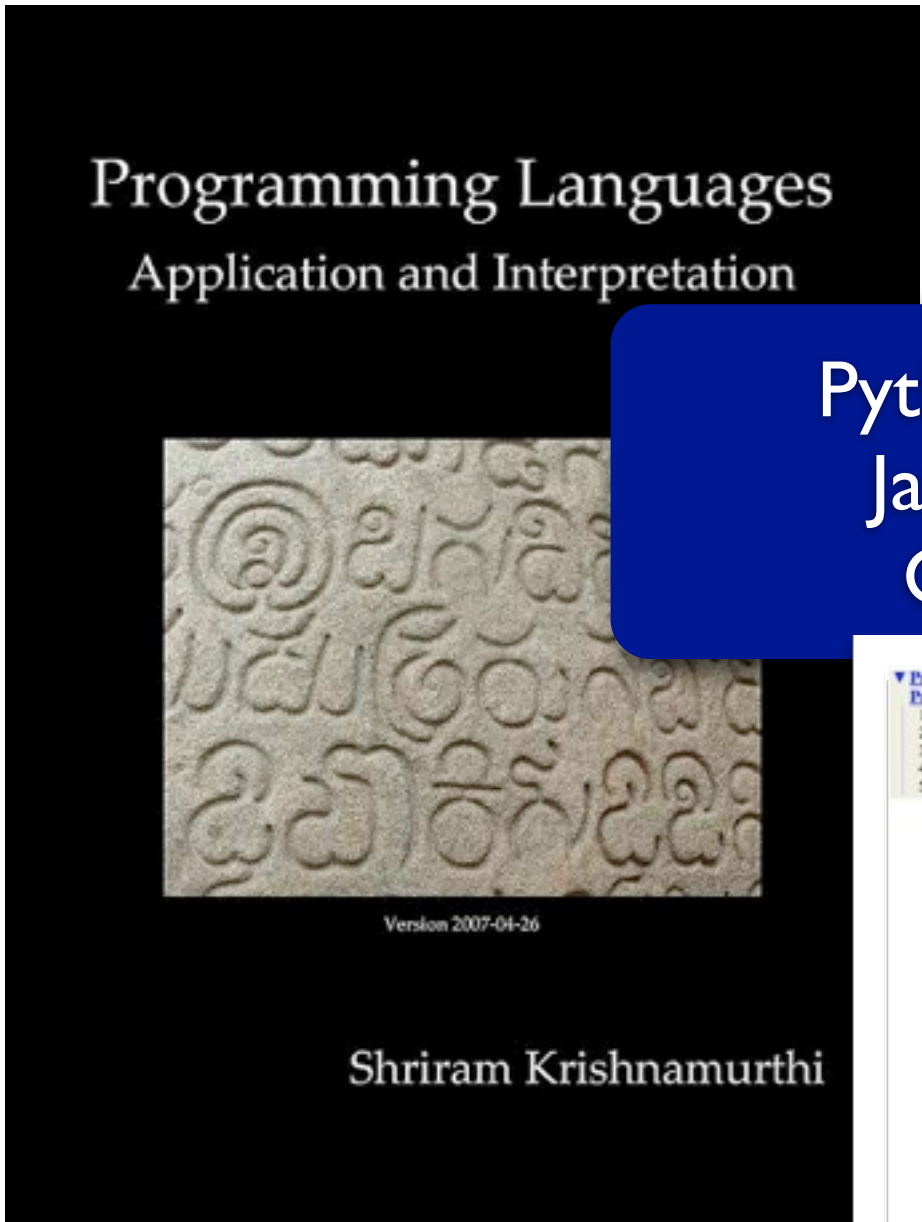
Python
Java
C



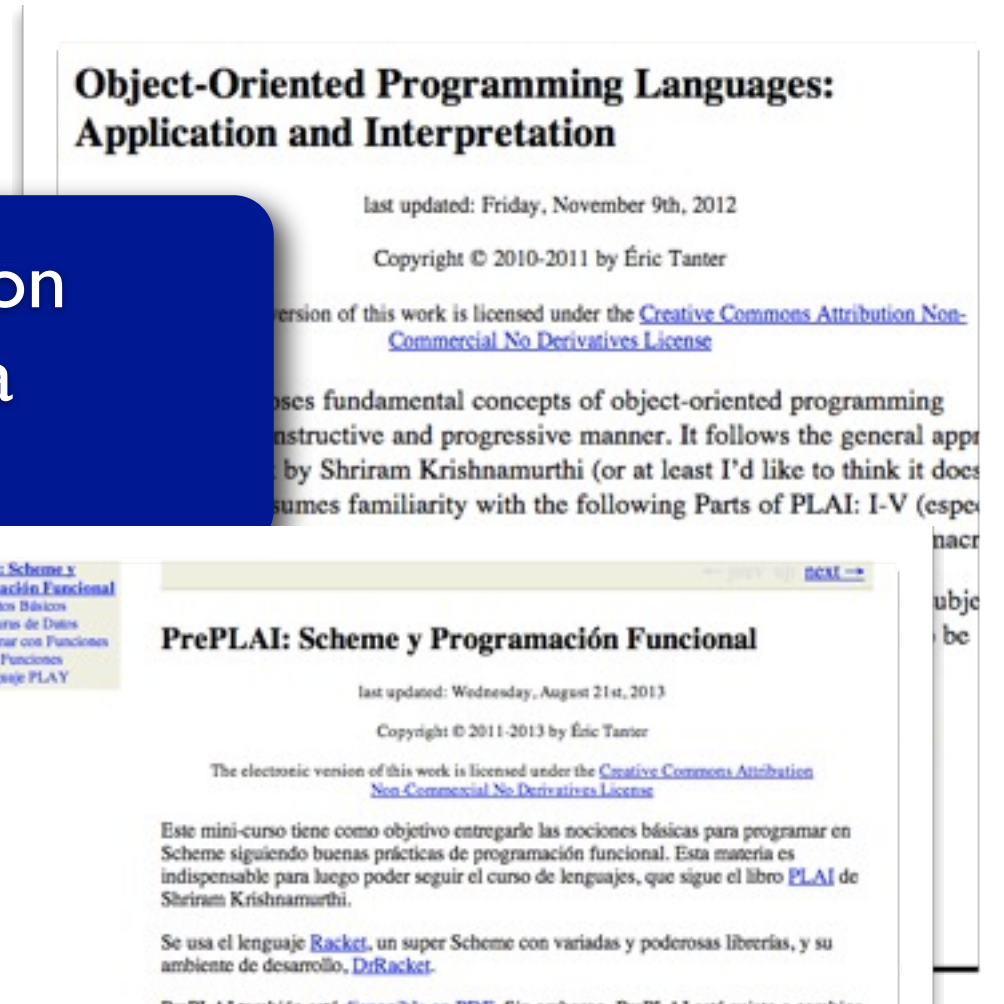
2/3

1/3

undergrad PL course @ U.Chile



Python
Java
C



2/3

1/3

undergrad PL course @ U.Chile

Programming Languages

Application and Interpretation

Version 2007-04-26

Python
Java
C

Object-Oriented Programming Languages: Application and Interpretation

last updated: Friday, November 9th, 2012

Copyright © 2010-2011 by Éric Tanter

ersion of this work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License

oses fundamental concepts of object-oriented programming
nstructive and progressive manner. It follows the general appr
by Shriram Krishnamurthi (or at least I'd like to think it does
sumes familiarity with the following Parts of PLAI: I-V (espe

PrePLAI: Scheme y Programación Funcional

last updated: Wednesday, August 21st, 2013

Copyright © 2011-2013 by Éric Tanter

The electronic version of this work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License

Este mini-curso tiene como objetivo entregarle las nociones básicas para programar en Scheme siguiendo buenas prácticas de programación funcional. Esta materia es indispensable para luego poder seguir el curso de lenguajes, que sigue el libro PLAI de Shriram Krishnamurthi.

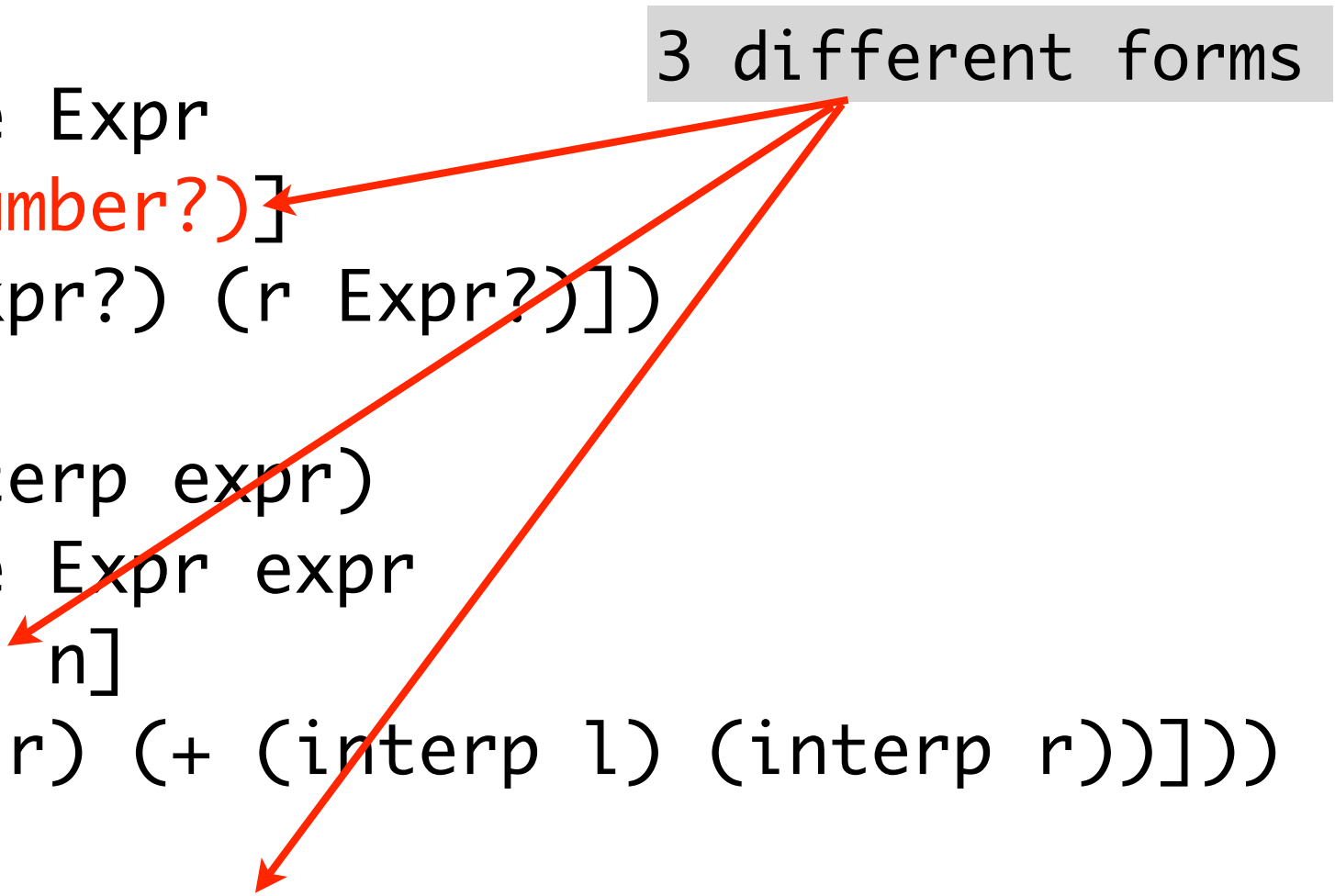
live programming on DrRacket
⇒ screen estate is important

```
#lang plai

(define-type Expr
  [num (n number?)]
  [add (l Expr?) (r Expr?)])

(define (interp expr)
  (type-case Expr expr
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]))

(interp (add (num 1) (num 3)))
```

issue #1: data

```
#lang plai

(define-type Expr
  [num (n number?)]
  [add (l Expr?) (r Expr?)])

(define (interp expr)
  (type-case Expr expr
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]))

(interp (add (num 1) (num 3)))
```

```
#lang plai

(define-type Expr
  [num (n number?)]
  [add (l Expr?) (r Expr?)])

(define (interp expr)
  (type-case Expr expr
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]))

(interp (add (num 1) (num 3)))
```



issue #2: type-case for pattern matching

```
#lang plai

(define-type Value*Store
  (v*s (value VBCFAE-Value?)
       (store Store?)))


(define (interp expr env store)
  (type-case Expr expr
    [ ...
    [(setbox (box-expr val-expr)
       (type-case Value*Store (interp box-expr env store)
         [v*s (box-val box-sto)
           (type-case Value*Store (interp val-expr env box-sto)
             [v*s (val-val val-sto)
               (v*s val-val
                    (aSto (boxV-location box-val)
                          val-val
                          val-sto))])]))]
```

```
#lang plai

(define-type Value*Store
  (v*s (value VBCFAE-Value?)
       (store Store?)))


(define (interp expr env store)
  (type-case Expr expr
    [ ...
    [(setbox (box-expr val-expr)
       (type-case Value*Store (interp box-expr env store)
         [v*s (box-val box-sto)
           (type-case Value*Store (interp val-expr env box-sto)
             [v*s (val-val val-sto)
               (v*s val-val
                    (aSto (boxV-location box-val)
                          val-val
                          val-sto))])])))]
```

```
#lang plai

(define-type Value*Store
  (v*s (value VBCFAE-Value?)
       (store Store?)))


(define (interp expr env store)
  (type-case Expr expr
    [ ...
    [(setbox (box-expr val-expr)
       (type-case Value*Store (interp box-expr env store)
         [v*s (box-val box-sto)
           (type-case Value*Store (interp val-expr env box-sto)
             [v*s (val-val val-sto)
               (v*s val-val
                    (aSto (boxV-location box-val)
                          val-val
                          val-sto))])])))]
```

"right drifting"

http://www.ccs.neu.edu/home/matthias/Style/style/

How to Program Racket

4.2 Definitions

Racket comes with quite a few definitional constructs, including `let`, `let*`, `letrec`, and `define`. Except for the last one, definitional constructs increase the indentation level. Therefore, favor `define` when feasible.

good

```
#lang racket

(define (swap x y)
  (define t (unbox x))
  (set-box! x (unbox y))
  (set-box! y t))
```

bad

```
#lang racket

(define (swap x y)
  (let ([t (unbox x)])
    (set-box! x (unbox y))
    (set-box! y t)))
```

```
(app (fun-expr arg-expr)
     (type-case Value*Store (interp fun-expr env store)
       (v*s (fun-val fun-sto)
            (type-case VBCFAE-Value fun-val
              (closureV (cl-param cl-body cl-env)
                        (type-case Value*Store (interp arg-expr env fun-sto);; then
                          (v*s (arg-val arg-sto)
                               (local ((define new-loc (next-location arg-sto)))
                                 (interp (closureV-body fun-val)
                                         (aEnv (closureV-param fun-val)
                                               new-loc
                                               (closureV-env fun-val))
                                         (aSto new-loc
                                               arg-val
                                               arg-sto))))))
(refclosV (cl-param cl-body cl-env)
          (local ((define arg-loc (env-lookup (id-name arg-expr) env)))
            (interp cl-body
                    (aEnv cl-param
                          arg-loc
                          cl-env)
                    fun-sto)))
(else (error 'interp "not a function"))))))

(if0 (test then else)
     (type-case Value*Store (interp test env store)
       (v*s (test-val test-sto)
            (if (num0? test-val)
                (interp then env test-sto)
                (interp else env test-sto)))))

(seqn (e1 e2)
      (type-case Value*Store (interp e1 env store)
        (v*s (e1-val e1-sto)
             (interp e2 env e1-sto))))

(newbox (val-expr)
        (type-case Value*Store (interp val-expr env store)
          (v*s (val-val val-sto)
               (local ((define new-loc (next-location val-sto)))
                 (v*s (boxV new-loc)
                      (aSto new-loc val-val val-sto)))))))

(setbox (box-expr val-expr)
        (type-case Value*Store (interp box-expr env store)
          (v*s (box-val box-sto)
               (type-case Value*Store (interp val-expr env box-sto)
                 (v*s (val-val val-sto)
                      (v*s val-val
                           (aSto (boxV-location box-val)
                                 val-val
                                 val-sto)))))))

(openbox (box-expr)
         (type-case Value*Store (interp box-expr env store)
           (v*s (box-val box-sto)
                (v*s (store-lookup (boxV-location box-val) store)
                     box-sto))))
(mset (id val)
      (type-case Value*Store (interp val env store)
        (v*s (val-val val-sto)
             (local ((define the-loc (env-lookup id env)))
               (v*s val-val
                    (aSto the-loc val-val val-sto)))))))
```

issue #3: macros

beyond define-syntax-rule...

```
#lang racket
(define-syntax (for x)
  (syntax-case x (from to in)
    [(for from low to high in bodies ...)
     (with-syntax ([it (datum->syntax-object
                        (syntax for) 'it)])
       (syntax
         ; here goes the stuff ))]))
```

beyond define-syntax-rule...

```
#lang racket
(define-syntax (for x)
  (syntax-case x (from to in)
    [(for from low to high in bodies ...)
     (with-syntax ([it (datum->syntax-object
                        (syntax for) 'it)])
       (syntax
         ; here goes the stuff ))]))
```

beyond define-syntax-rule...

```
#lang racket
(define-syntax (for x)
  (syntax-case x (from to in)
    [(for from low to high in bodies ...)
     (with-syntax ([it (datum->syntax-object
                        (syntax for) 'it)])
       (syntax
         ; here goes the stuff ))]))
```

(main goal: teaching macros as a means to study OO concepts)

The good thing about Racket...

The good thing about Racket...

The good thing about Racket...

#lang play

1. deftype
2. def
3. defmac

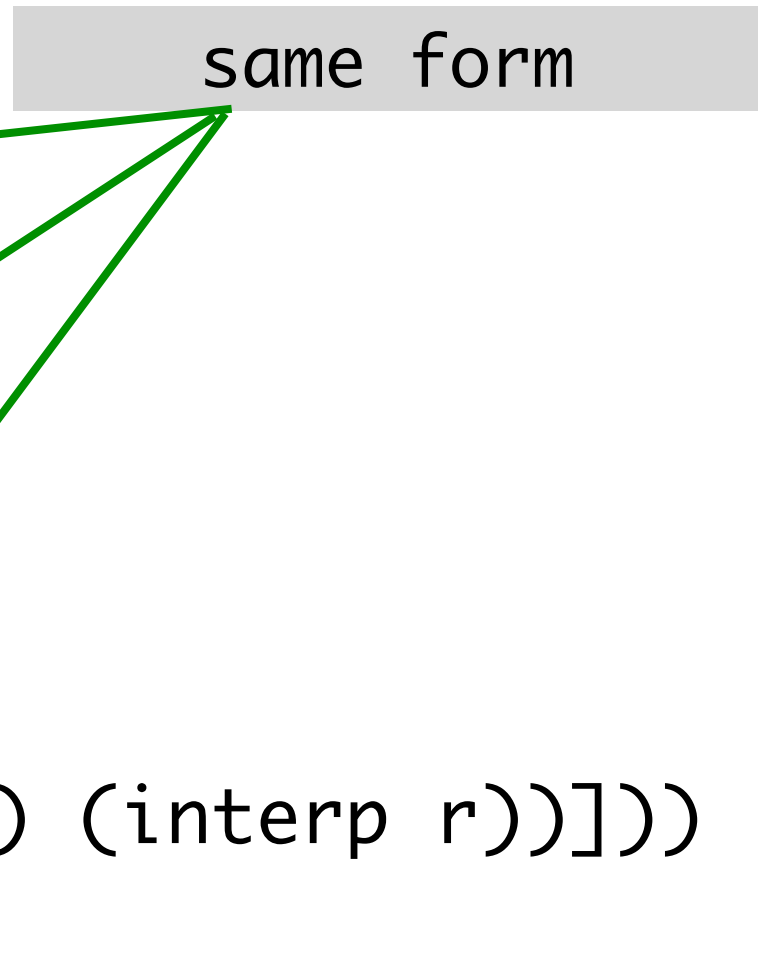
(4. defun)

deftype

lightweight & uniform

```
#lang plai

(define-type Expr
  [num (n number?)]
  [add (l Expr?) (r Expr?)])

(define (interp expr)
  (type-case Expr expr
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]))

(interp (add (num 1) (num 3)))
```

```
#lang play

(deftype Expr
  (num n)
  (add l r))

(define (interp expr)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l) (interp r))]))

(interp (add (num 1) (num 3)))
```



```
<expr> ::=
  <num>
| {+ <expr> <expr>}
| {- <expr> <expr>}
| <id>
| {fun {<id>} <expr>}
| {refun {<id>} <expr>}
| {<expr> <expr>}
| {if0 <expr> <expr> <expr>}
| {seqn <expr> <expr>}
| {newbox <expr>}
| {setbox! <expr> <expr>}
| {openbox <expr>}
| {set! <id> <expr>}
```

```
                                  #lang racket

<expr> ::=                        (struct expr ())
  <num>                           (struct num expr (n) #:transparent)
| {+ <expr> <expr>}                (struct add expr (left right) #:tra
| {- <expr> <expr>}                (struct sub expr (left right) #:tra
| <id>                            (struct id  expr (name) #:transpare
| {fun {<id>} <expr>}             (struct fun expr (param body) #:tra
| {refun {<id>} <expr>}           (struct refun expr (param body) #:t
| {<expr> <expr>}                 (struct app expr (fun-expr arg-expr
| {if0 <expr> <expr> <expr>}      (struct if0  expr (c t f) #:transpa
| {seqn <expr> <expr>}            (struct seqn expr (e1 e2) #:transpa
| {newbox <expr>}                 (struct openbox expr (box-expr) #:t
| {setbox! <expr> <expr>}         (struct setbox! expr (box-expr val-
| {openbox <expr>}                (struct newbox expr (val-expr) #:tr
| {set! <id> <expr>}              (struct mset   expr (name val-expr)
```

```
<expr> ::=
  <num>
| {+ <expr> <expr>}
| {- <expr> <expr>}
| <id>
| {fun {<id>} <expr>}
| {refun {<id>} <expr>}
| {<expr> <expr>}
| {if0 <expr> <expr> <expr>}
| {seqn <expr> <expr>}
| {newbox <expr>}
| {setbox! <expr> <expr>}
| {openbox <expr>}
| {set! <id> <expr>}
```

```
<expr> ::=
  <num>
| {+ <expr> <expr>}
| {- <expr> <expr>}
| <id>
| {fun {<id>} <expr>}
| {refun {<id>} <expr>}
| {<expr> <expr>}
| {if0 <expr> <expr> <expr>}
| {seqn <expr> <expr>}
| {newbox <expr>}
| {setbox! <expr> <expr>}
| {openbox <expr>}
| {set! <id> <expr>}
```

```
#lang play

(deftype expr
 (num n)
 (add left right)
 (sub left right)
 (id name)
 (fun param body)
 (refun param body)
 (app fun-expr arg-expr)
 (if0 c t f)
 (seqn e1 e2)
 (openbox box-expr)
 (setbox! box-expr val-expr)
 (newbox val-expr)
 (mset name val-expr))
```

def

match-define in 3 letters

```
(define-type Value*Store
  (v*s (value VBCFAE-Value?)
       (store Store?)))
```

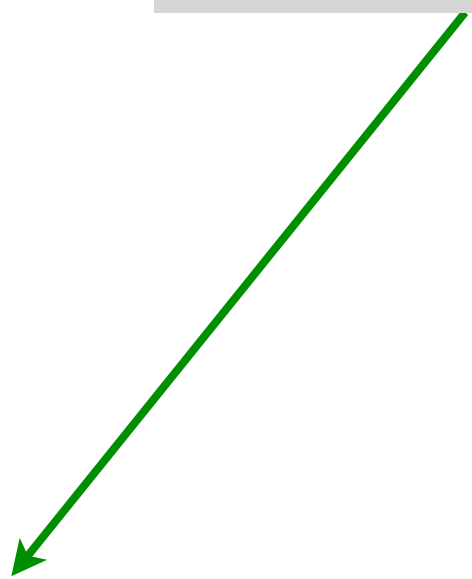
```
(define (interp expr env store)
  (type-case Expr expr
    [ ...
    [(setbox (box-expr val-expr)
       (type-case Value*Store (interp box-expr env store)
         [v*s (box-val box-sto)
           (type-case Value*Store (interp val-expr env box-sto)
             [v*s (val-val val-sto)
               (v*s val-val
                    (aSto (boxV-location box-val)
                          val-val
                          val-sto))])])))]
```

```
(deftype Value*Store
  (v*s value store))
```

```
(define (interp expr env store)
  (match expr
    [ ...
    [(setbox box-expr val-expr)
     (def (v*s box-val box-sto) (interp box-expr env sto))
     (def (v*s val-val val-sto) (interp val-expr env box-sto))
     (v*s val-val
          (aSto (boxV-loc box-val)
                val-val
                val-sto))]
```

```
(deftype Value*Store
  (v*s value store))
```

same indent level



```
(define (interp expr env store)
  (match expr
    [ ...
    [(setbox box-expr val-expr)
     (def (v*s box-val box-sto) (interp box-expr env sto))
     (def (v*s val-val val-sto) (interp val-expr env box-sto))
     (v*s val-val
          (aSto (boxV-loc box-val)
                val-val
                val-sto))]
```

```
  (app (fun-expr arg-expr)
       (type-case Value*Store (interp fun-expr env store)
         (v*s (fun-val fun-sto)
              (type-case VBCFAE-Value fun-val
                (closureV (cl-param cl-body cl-env)
                          (type-case Value*Store (interp arg-expr env fun-sto);; then
                            (v*s (arg-val arg-sto)
                                 (local ((define new-loc (next-location arg-sto)))
                                   (interp (closureV-body fun-val)
                                           (aEnv (closureV-param fun-val)
                                                 new-loc
                                                 (closureV-env fun-val))
                                           (aSto new-loc
                                                 arg-val
                                                 arg-sto))))))
  (refclosV (cl-param cl-body cl-env)
          (local ((define arg-loc (env-lookup (id-name arg-expr) env)))
            (interp cl-body
                    (aEnv cl-param
                          arg-loc
                          cl-env)
                    fun-sto)))
   (else (error 'interp "not a function"))))))

  (if0 (test then else)
       (type-case Value*Store (interp test env store)
         (v*s (test-val test-sto)
              (if (num0? test-val)
                  (interp then env test-sto)
                  (interp else env test-sto)))))

  (seqn (e1 e2)
        (type-case Value*Store (interp e1 env store)
          (v*s (e1-val e1-sto)
               (interp e2 env e1-sto))))

  (newbox (val-expr)
          (type-case Value*Store (interp val-expr env store)
            (v*s (val-val val-sto)
                 (local ((define new-loc (next-location val-sto)))
                   (v*s (boxV new-loc)
                        (aSto new-loc val-val val-sto))))))

  (setbox (box-expr val-expr)
          (type-case Value*Store (interp box-expr env store)
            (v*s (box-val box-sto)
                 (type-case Value*Store (interp val-expr env box-sto)
                   (v*s (val-val val-sto)
                        (v*s val-val
                             (aSto (boxV-location box-val)
                                   val-val
                                   val-sto)))))))

  (openbox (box-expr)
           (type-case Value*Store (interp box-expr env store)
             (v*s (box-val box-sto)
                  (v*s (store-lookup (boxV-location box-val) store)
                       box-sto))))
  (mset (id val)
       (type-case Value*Store (interp val env store)
         (v*s (val-val val-sto)
              (local ((define the-loc (env-lookup id env)))
                (v*s val-val
                     (aSto the-loc val-val val-sto))))))
```

```
(app (fun-expr arg-expr)
     (type-case Value*Store (interp fun-expr env store)
       (v*s (fun-val fun-sto)
            (type-case VBCFAE-Value fun-val
              (closureV (cl-param cl-body cl-env)
                        (type-case Value*Store (interp arg-expr env fun-sto);; then
                          (v*s (arg-val arg-sto)
                               (local ((define new-loc (next-location arg-sto)))
                                 (interp (closureV-body fun-val)
                                         (aEnv (closureV-param fun-val)
                                               new-loc
                                               (closureV-env fun-val))
                                         (aSto new-loc
                                               arg-val
                                               arg-sto))))))
(refclosV (cl-param cl-body cl-env)
          (local ((define arg-loc (env-lookup (id-name arg-expr) env)))
            (interp cl-body
                    (aEnv cl-param
                          arg-loc
                          cl-env)
                    fun-sto)))
 (else (error 'interp "not a function"))))))

 (if0 (test then else)
      (type-case Value*Store (interp test env store)
        (v*s (test-val test-sto)
             (if (num0? test-val)
                 (interp then env test-sto)
                 (interp else env test-sto)))))

 (seqn (e1 e2)
       (type-case Value*Store (interp e1 env store)
         (v*s (e1-val e1-sto)
              (interp e2 env e1-sto))))

 (newbox (val-expr)
         (type-case Value*Store (interp val-expr env store)
           (v*s (val-val val-sto)
                (local ((define new-loc (next-location val-sto)))
                  (v*s (boxV new-loc)
                       (aSto new-loc val-val val-sto))))))

 (setbox (box-expr val-expr)
         (type-case Value*Store (interp box-expr env store)
           (v*s (box-val box-sto)
                (type-case Value*Store (interp val-expr env box-sto)
                  (v*s (val-val val-sto)
                       (v*s val-val
                            (aSto (boxV-location box-val)
                                  val-val
                                  val-sto)))))))

 (openbox (box-expr)
          (type-case Value*Store (interp box-expr env store)
            (v*s (box-val box-sto)
                 (v*s (store-lookup (boxV-location box-val) store)
                      box-sto))))
 (mset (id val)
      (type-case Value*Store (interp val env store)
        (v*s (val-val val-sto)
             (local ((define the-loc (env-lookup id env)))
               (v*s val-val
                    (aSto the-loc val-val val-sto))))))
```

```
[(app fun-expr arg-expr)
 (def (v*s fun-val fun-sto) (interp fun-expr env sto))
 (match fun-val
   [(closureV param body fun-env)
    (def (v*s arg-val arg-sto) (interp arg-expr env fun-sto))
    (def new-loc (next-location arg-sto))
    (interp body
            (aEnv param new-loc fun-env)
            (aSto new-loc arg-val arg-sto))]
   [(refclosV param body fun-env)
    (def arg-loc (env-lookup (id-name arg-expr) env))
    (interp body
            (aEnv param arg-loc fun-env)
            fun-sto)])]

[(seqn e1 e2)
 (def (v*s _ sto1) (interp e1 env sto))
 (interp e2 env sto1)]

[(newbox val-expr)
 (def (v*s val val-sto) (interp val-expr env sto))
 (def new-loc (next-location val-sto))
 (v*s (boxV new-loc)
      (aSto new-loc val val-sto))]

[(openbox box-expr)
 (def (v*s box-val box-sto) (interp box-expr env sto))
 (v*s (sto-lookup (boxV-loc box-val) box-sto)
      box-sto)]

[(setbox! box-expr val-expr)
 (def (v*s box-val box-sto) (interp box-expr env sto))
 (def (v*s val-val val-sto) (interp val-expr env box-sto))
 (v*s val-val
      (aSto (boxV-loc box-val)
            val-val
            val-sto))]

[(mset name val-expr)
 (def (v*s val-val val-sto) (interp val-expr env sto))
 (v*s val-val
      (aSto (env-lookup name env)
            val-val
            val-sto))]
```

```
(app (fun-expr arg-expr)
     (type-case Value*Store (interp fun-expr env store)
       (v*s (fun-val fun-sto)
            (type-case VBCFAE-Value fun-val
              (closureV (cl-param cl-body cl-env)
                        (type-case Value*Store (interp arg-expr env fun-sto);; then
                          (v*s (arg-val arg-sto)
                               (local ((define new-loc (next-location arg-sto)))
                                 (interp (closureV-body fun-val)
                                         (aEnv (closureV-param fun-val)
                                               new-loc
                                               (closureV-env fun-val))
                                         (aSto new-loc
                                               arg-val
                                               arg-sto)))))))
(refclosV (cl-param cl-body cl-env)
          (local ((define arg-loc (env-lookup (id-name arg-expr) env)))
            (interp cl-body
                    (aEnv cl-param
                          arg-loc
                          cl-env)
                    fun-sto)))
(else (error 'interp "not a function")))))

(if0 (test then else)
     (type-case Value*Store (interp test env store)
       (v*s (test-val test-sto)
            (if (num0? test-val)
                (interp then env test-sto)
                (interp else env test-sto)))))

(seqn (e1 e2)
      (type-case Value*Store (interp e1 env store)
        (v*s (e1-val e1-sto)
             (interp e2 env e1-sto))))

(newbox (val-expr)
        (type-case Value*Store (interp val-expr env store)
          (v*s (val-val val-sto)
               (local ((define new-loc (next-location val-sto)))
                 (v*s (boxV new-loc)
                      (aSto new-loc val-val val-sto)))))))

(setbox (box-expr val-expr)
        (type-case Value*Store (interp box-expr env store)
          (v*s (box-val box-sto)
               (type-case Value*Store (interp val-expr env box-sto)
                 (v*s (val-val val-sto)
                      (v*s val-val
                           (aSto (boxV-location box-val)
                                 val-val
                                 val-sto)))))))

(openbox (box-expr)
         (type-case Value*Store (interp box-expr env store)
           (v*s (box-val box-sto)
                (v*s (store-lookup (boxV-location box-val) store)
                     box-sto))))
(mset (id val)
     (type-case Value*Store (interp val env store)
       (v*s (val-val val-sto)
            (local ((define the-loc (env-lookup id env)))
              (v*s val-val
                   (aSto the-loc val-val val-sto)))))))
```

81

```
[(app fun-expr arg-expr)
 (def (v*s fun-val fun-sto) (interp fun-expr env sto))
 (match fun-val
   [(closureV param body fun-env)
    (def (v*s arg-val arg-sto) (interp arg-expr env fun-sto))
    (def new-loc (next-location arg-sto))
    (interp body
            (aEnv param new-loc fun-env)
            (aSto new-loc arg-val arg-sto))]
   [(refclosV param body fun-env)
    (def arg-loc (env-lookup (id-name arg-expr) env))
    (interp body
            (aEnv param arg-loc fun-env)
            fun-sto)])]

[(seqn e1 e2)
 (def (v*s _ sto1) (interp e1 env sto))
 (interp e2 env sto1)]

[(newbox val-expr)
 (def (v*s val val-sto) (interp val-expr env sto))
 (def new-loc (next-location val-sto))
 (v*s (boxV new-loc)
      (aSto new-loc val val-sto))]

[(openbox box-expr)
 (def (v*s box-val box-sto) (interp box-expr env sto))
 (v*s (sto-lookup (boxV-loc box-val) box-sto)
      box-sto)]

[(setbox! box-expr val-expr)
 (def (v*s box-val box-sto) (interp box-expr env sto))
 (def (v*s val-val val-sto) (interp val-expr env box-sto))
 (v*s val-val
      (aSto (boxV-loc box-val)
            val-val
            val-sto))]

[(mset name val-expr)
 (def (v*s val-val val-sto) (interp val-expr env sto))
 (v*s val-val
      (aSto (env-lookup name env)
            val-val
            val-sto))]
```

64

```
(app (fun-expr arg-expr)
     (type-case Value*Store (interp fun-expr env store)
       (v*s (fun-val fun-sto)
            (type-case VBCFAE-Value fun-val
              (closureV (cl-param cl-body cl-env)
                        (type-case Value*Store (interp arg-expr env fun-sto);; then
                          (v*s (arg-val arg-sto)
                               (local ((define new-loc (next-location arg-sto)))
                                 (interp (closureV-body fun-val)
                                         (aEnv (closureV-param fun-val)
                                               new-loc
                                               (closureV-env fun-val))
                                         (aSto new-loc
                                               arg-val
                                               arg-sto)))))))
(refclosV (cl-param cl-body cl-env)
          (local ((define arg-loc (env-lookup (id-name arg-expr) env)))
            (interp cl-body
                    (aEnv cl-param
                          arg-loc
                          cl-env)
                    fun-sto)))
(else (error 'interp "not a function")))))

(if0 (test then else)
     (type-case Value*Store (interp test env store)
       (v*s (test-val test-sto)
            (if (num0? test-val)
                (interp then env test-sto)
                (interp else env test-sto)))))

(seqn (e1 e2)
      (type-case Value*Store (interp e1 env store)
        (v*s (e1-val e1-sto)
             (interp e2 env e1-sto))))

(newbox (val-expr)
        (type-case Value*Store (interp val-expr env store)
          (v*s (val-val val-sto)
               (local ((define new-loc (next-location val-sto)))
                 (v*s (boxV new-loc)
                      (aSto new-loc val-val val-sto))))))

(setbox (box-expr val-expr)
        (type-case Value*Store (interp box-expr env store)
          (v*s (box-val box-sto)
               (type-case Value*Store (interp val-expr env box-sto)
                 (v*s (val-val val-sto)
                      (v*s val-val
                           (aSto (boxV-location box-val)
                                 val-val
                                 val-sto)))))))

(openbox (box-expr)
         (type-case Value*Store (interp box-expr env store)
           (v*s (box-val box-sto)
                (v*s (store-lookup (boxV-location box-val) store)
                     box-sto))))
(mset (id val)
     (type-case Value*Store (interp val env store)
       (v*s (val-val val-sto)
            (local ((define the-loc (env-lookup id env)))
              (v*s val-val
                   (aSto the-loc val-val val-sto))))))
```

53

81

```
[(app fun-expr arg-expr)
 (def (v*s fun-val fun-sto) (interp fun-expr env sto))
 (match fun-val
   [(closureV param body fun-env)
    (def (v*s arg-val arg-sto) (interp arg-expr env fun-sto))
    (def new-loc (next-location arg-sto))
    (interp body
            (aEnv param new-loc fun-env)
            (aSto new-loc arg-val arg-sto))]
   [(refclosV param body fun-env)
    (def arg-loc (env-lookup (id-name arg-expr) env))
    (interp body
            (aEnv param arg-loc fun-env)
            fun-sto)])]

[(seqn e1 e2)
 (def (v*s _ sto1) (interp e1 env sto))
 (interp e2 env sto1)]

[(newbox val-expr)
 (def (v*s val val-sto) (interp val-expr env sto))
 (def new-loc (next-location val-sto))
 (v*s (boxV new-loc)
      (aSto new-loc val val-sto))]

[(openbox box-expr)
 (def (v*s box-val box-sto) (interp box-expr env sto))
 (v*s (sto-lookup (boxV-loc box-val) box-sto)
      box-sto)]

[(setbox! box-expr val-expr)
 (def (v*s box-val box-sto) (interp box-expr env sto))
 (def (v*s val-val val-sto) (interp val-expr env box-sto))
 (v*s val-val
      (aSto (boxV-loc box-val)
            val-val
            val-sto))]

[(mset name val-expr)
 (def (v*s val-val val-sto) (interp val-expr env sto))
 (v*s val-val
      (aSto (env-lookup name env)
            val-val
            val-sto))]
```

16

64

defmac

define-syntax-rule on steroids

```
#lang racket
(define-syntax (for x)
  (syntax-case x (from to in)
    [(for from low to high in bodies ...)
     (with-syntax ([it (datum->syntax-object
                        (syntax for) 'it)])
       (syntax
         ; here goes the stuff ))]))
```

```
#lang racket
(define-syntax (for x)
  (syntax-case x (from to in)
    [(for from low to high in bodies ...)
     (with-syntax ([it (datum->syntax-object
                        (syntax for) 'it)])
       (syntax
         ; here goes the stuff ))]))
```

```
#lang play
(defmac (for from low to high in bodies ...)
  #:keywords from to in
  #:captures it
  ; here goes the stuff )
```

#lang play

`deftype`	convenience for variants
`def`	`= match-define`
`defmac`	`= Eli's defmac` (rejuvenated)

`/ define-syntax-rule` with optional args

#lang play

`deftype` convenience for variants

`def` = `match-define`

`defmac` = `Eli's defmac` (rejuvenated)

/ `define-syntax-rule` with optional args

`defun` = `define` (unify with `define`/`match`?)

(Haskell-like equations?)

`#lang play`

- simplify definitions
- foster the use of pattern matching
- unify non-PM and PM definition forms
- cost: disunify `define` for function definitions

#lang play

#lang play

in the catalog!

```
$ raco pkg install play
```