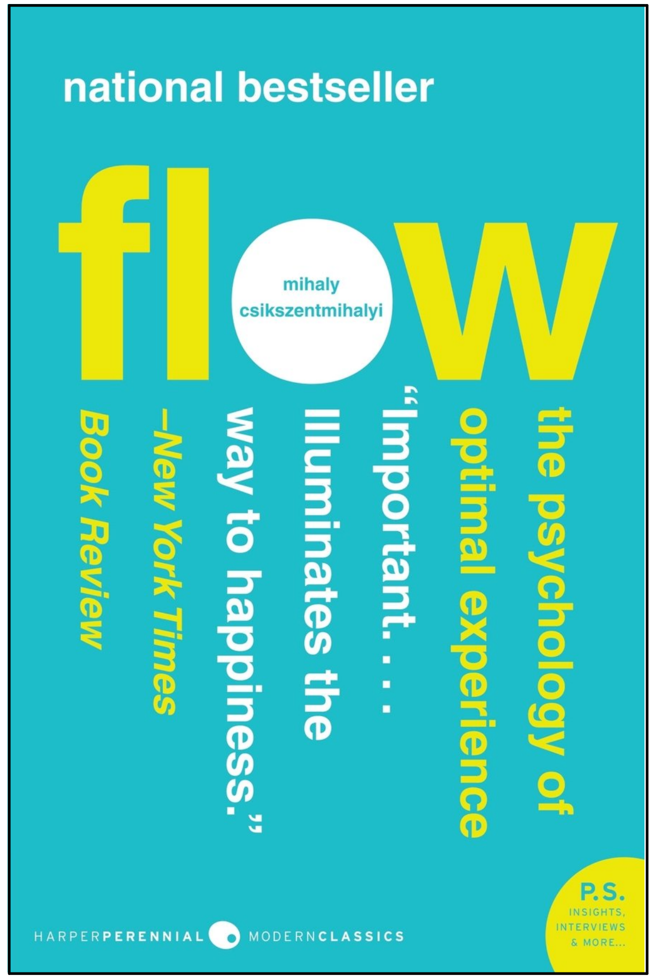
Algebraic Racket in Action

RacketCon 2019

Algebraic Racket is ...

A **dialect** for
optimal programming ***experience***

Programming ***is*** flow control



The Big Idea

1. Types describe interesting structures

2. Structures are useful without types

Programming should be fun. Programs should be beautiful.

Think Structurally

```
                  ⋮

(define app1
  (function
    [(App t1 t2)
     (let ([t1* (step t1)])
       (and t1* (App t1* t2)))]
    [_ #f]))
(define app2
  (function
    [(App v1 t2)
     (let ([t2* (step t2)])
       (and t2* (App v1 t2*)))]
    [_ #f]))

                  ⋮
```

Think Structurally

⋮

```
(define app1 (function [(App t1 t2) (let ([t1* (step t1)]) (and t1* (App t1* t2)))] [_ #f]))
(define app2 (function [(App v1 t2) (let ([t2* (step t2)]) (and t2* (App v1 t2*)))] [_ #f]))
```

⋮

Think Structurally

```
(define-steps
  ⋮
  [app1 (App t1 t2) (sub-step t1 t1* (App t1* t2))]
  [app2 (App v1 t2) (sub-step t2 t2* (App v1 t2*))]
  ⋮ )
```

Prelude: Common Function Aliases

```
curry        >> ; or >>*
curryr       << ; or <<*
values       id
cons         ::
list*        ::*
append       ++
compose      ..
conjoin      &&
disjoin      ||
```

Prelude: Composed Curry Totem

```
((.. (>> $ id)
     (>> map add1)
     (>> map syntax-e)
     list)
 #'1 #'2 #'3)
```

Prelude: Growing a Totem

```
#'(1 2 3)
```

```
#<syntax (1 2 3)>
```

Prelude: Growing a Totem

```
(syntax-e #'(1 2 3))
```

```
'(#<syntax 1> #<syntax 2> #<syntax 3>)
```

Prelude: Growing a Totem

```
(map syntax-e (syntax-e #'(1 2 3)))
```

```
'(1 2 3)
```

Prelude: Growing a Totem

```
(map add1 (map syntax-e (syntax-e #'(1 2 3))))
```

```
'(2 3 4)
```

Prelude: Growing a Totem

```
($ id (map add1 (map syntax-e (syntax-e #'(1 2 3)))))
```

```
2
3
4
```

Prelude: Growing a Totem

```
($ id (map add1 (map syntax-e (syntax-e #'(1 2 3)))))
```

Prelude: Growing a Totem

```
((>> $ id)
 (map add1 (map syntax-e (syntax-e #'(1 2 3)))))
```

Prelude: Growing a Totem

```
((.. (>> $ id)
     (>> map add1))
 (map syntax-e (syntax-e #'(1 2 3))))
```

Prelude: Growing a Totem

```
((.. (>> $ id)
     (>> map add1)
     (>> map syntax-e))
 (syntax-e #'(1 2 3)))
```

Prelude: Growing a Totem

```
((.. (>> $ id)
     (>> map add1)
     (>> map syntax-e)
     list)
 #'1 #'2 #'3)
```

Prelude: Growing a Totem

```
(define args+1->values
  (.. (>> $ id)
      (>> map add1)
      (>> map syntax-e)
      list))
(args+1->values #'1 #'2 #'3)
```

Prelude: Conjunctions and Disjunctions

```
(define (syntax-singleton? x)
  ((|| (&& syntax?
           (.. syntax-singleton? syntax-e))
       (&& pair?
           (.. syntax? car)
           (.. null? cdr)))
   x))
(syntax-singleton? #'(1)) ; ↝ #t
(syntax-singleton? #'1) ; ↝ #f
```

A New Beginning

Racket is a *programming language*

Code is code, and data is data.

The Call to Adventure

What Is Language-Oriented Programming?

MAGIC

The Call to Adventure

How Do I Make Racket Work for Me?

Algebraic Data Types

The Call to Adventure

How Do I Make Racket Work for Me?

Algebraic Data Structures

Algebraic Data

A ***sum*** is an enumeration of ***product constructors***

```
(data Void ())
(data Unit (Unit))
(data Maybe (Nothing Just))
```

- complete ⇒ **Eq**
- ordered ⇒ **Ord**
- parsable ⇒ **Read**
- printable ⇒ **Show**

Data Leads to Functions

```
(data VizCommand (; run-time state
                  Dump Load Reset Pause Unpause Zoom Pan
                  ; nodes
                  Node Set Add Drop
                  ; edges
                  Edge Update Connect Disconnect)
      VizResult (FAIL OK))
```

Data Leads to Functions

```
(define/public command
  (function
    ; run-time environment
    [Dump (get-state)]
    [(Load state) (set-state state)]
    [Reset (reset!)]
    [Pause (set! paused? #t) (OK)]
    [Unpause (set! paused? #f) (OK)]
    [(Zoom 'in) (zoom Δzoom) (OK)]
    [(Zoom 'out) (zoom (- Δzoom)) (OK)]
    [(Pan Δx Δy) (pan Δx Δy)]
    ; nodes
    [(Node name) (get-node name)]
    [(Set name proc pos) (set-node name proc pos)]
    [(Add name proc pos) (add-node name proc pos)]
    [(Drop name) (drop-node name)]
    ; edges
    [(Edge from-node to-node) (get-edge from-node to-node)]
    [(Update from-node to-node proc) (set-edge from-node to-node proc)]
    [(Connect from-node to-node proc) (add-edge from-node to-node proc)]
    [(Disconnect from-node to-node) (drop-edge from-node to-node)]
    ; else
    [_ FAIL]))
```

```
(send surface Pause)
(send surface (Zoom 'in))
(send surface
      (Add 'node1
           (λ (canvas) ...)
           #f))
```

Functions Lead to Macros

```
(define-syntax event-let
  (μ* (([var:id evt] ...+) body-evt ...+)
    (bind evt ... (λ (var ...) (seq body-evt ...)))))
```

Functions Lead to Macros

```
(define-syntax event-let*
  (macro*
    [(() body-evt ...+) (seq body-evt ...)]
    [(([var:id evt] . bindings) body-evt ...+)
     (bind evt (φ x (event-let* bindings body-evt ...)))]))
```

Macros Lead to Suffering

```
(define-syntax with-instance
  (macro*
    [(instance-id:id expr ...+)
     (with-instance [|| instance-id] expr ...)]
    [([prefix:id instance-id:id] expr ...+)
     #:if (instance-id? #'instance-id)
     #:do [(define members (instance-members #'instance-id))
           (define ids (map car members))]
     #:with (id ...) ids
     #:with (id/prefix ...) (map (prepend this-syntax #'prefix) ids)
     #:with (def ...) (map cadr members)
     (letrec-values
         ([(id/prefix ...)
           (letrec-syntax ([id (make-variable-like-transformer #'def)] ...)
             (values id ...))])
       expr ...)]))
```

Macros Lead to Suffering

```
(define-syntax with-instance
  (macro*
    [(instance-id:id expr ...+)
     #,(replace-context this-syntax #'(with-instance [|| instance-id] expr ...))]
    [([prefix:id instance-id:id] expr ...+)
     #:if (instance-id? #'instance-id)
     #:do [(define members (instance-members #'instance-id))
           (define ids (map car members))
           (define re-context (>> replace-context this-syntax))]
     #:with (id ...) (map re-context ids)
     #:with (id/prefix ...) (map (prepend this-syntax #'prefix) ids)
     #:with (def ...) (map (.. re-context cadr) members)
     (letrec-values
         ([(id/prefix ...)
           (letrec-syntax ([id (make-variable-like-transformer #'def)] ...)
             (values id ...))])
       expr ...)]))
```

The Inmost Cave

Racket is a *meta-programming language*

Code is not code, and data is not data.

An Ordeal

What Is Language-Oriented Programming?

EXHAUSTING

An Ordeal

How Do I Stop Working for Racket?

Type Classes

An Ordeal

How Do I Stop Working for Racket?

Structure Classes

Algebraic Classes

A ***class*** is a collection of names

```
(class Monad
  [>>=]
  [>>M (λ (m k) (>>= m (λ _ k)))]
  [return pure]
  [fail error]
  minimal ([>>=]))
```

```
(define-syntax MaybeMonad
  (instance Monad
    [return Just]
    [>>= (function*
           [((Just x) k) (k x)]
           [( Nothing _) Nothing])]
    [fail (φ _ Nothing)]))
```

```
(with-instance MaybeMonad (>>= (Just 2) (.. return add1))) ; ↝ (Just 3)
(with-instance MaybeMonad (>>=  Nothing (.. return add1))) ; ↝ Nothing
```

Algebraic Classes: Do-Notation

```
(with-instance ListMonad
  (do (x) <- '(1 2)
      (y) <- '(A B)
      (return x y))) ; ↝ '(1 A 1 B 2 A 2 B)
```

```
(with-instance ValuesMonad
  (do (x '! . y) <- (λ () (id 1 '! 2))
      zs <- (λ () (id 'A 'B))
      (return x y zs))) ; ↝ 1 '(2) '(A B)
```

The Event Monad

```
(define ◊ always-evt)

(define-syntax EventFunctor
  (instance Functor
    [fmap (flip handle-evt)]))

(define-syntax EventMonad
  (instance Monad
    extends (EventFunctor)
    [>>= replace-evt]
    [return (λ xs (fmap (λ _ ($ id xs)) ◊))]))
```

The Event Monad

```
(define-syntax EventApplicative
  (instance Applicative
    extends (EventMonad)
    [pure return]
    [liftA2 (λ (f a b)
              (do xs <- a
                  ys <- b
                  (return ($ f (++ xs ys)))))]))


(with-instance EventApplicative
  (sync (async-values
         (pure 1) (pure 2) (pure 3) (pure 4))))
```

Asynchronous Values

```
(with-instance EventApplicative
  (sync (async-values (pure 1) (pure 2) (pure 3) (pure 4))))



(define (async-values . as)
  (if (null? as)
      (handle-evt always-evt (λ _ (values)))
      (replace-evt
       (apply choice-evt
              (map (λ (a) (handle-evt a (λ (x) (cons a x))))
                   as))
       (λ (a+x)
         (handle-evt
          (apply async-values (remq (car a+x) as))
          (λ xs (apply values (cons (cdr a+x) xs))))))))
```

Asynchronous Values

```
(with-instance EventApplicative
  (sync (async-values (pure 1) (pure 2) (pure 3) (pure 4))))
```

```
(define (async-values . as)
  (with-instance EventMonad
    (if (null? as)
        (return)
        (do let identified (φ a (fmap (>> :: a) a))
            ((a . x)) <- ($ choice-evt (map identified as))
            xs <- ($ async-values (remq a as))
            ($ return (:: x xs))))))
```

Asynchronous Let

```
(define-syntax async-let
  (μ* (([var:id evt] ...) body ...+)
    (let-values ([(var ...) (sync (async-values evt ...))])
      body ...)))
```

```
(define ch (make-channel))
(for/list ([x 4])
  (thread (λ () (sleep (random)) (channel-put ch x))))
```

```
(async-let ([a ch] [b ch]                ; '(1 2 0 3)
            [c ch] [d ch])               ; '(2 0 1 3)
  (list a b c d))                        ; '(3 0 2 1)
```

The Elixir

Racket is a *language-oriented programming language*

Code is code, and data is data.

An Epiphany

What Is Language-Oriented Programming?

TRUTH

Begin the Journey

What Can I Do for Racket?

Seek Truth

Share Stories

Document Modules

Eric Griffis
dedbox@gmail.com

@dedbox
https://dedbox.github.io