Extensible Pattern Matching

Sam Tobin-Hochstadt
PLT @ Northeastern University

IFL, September 3, 2010

Extensible Pattern Matching in an Extensible Language

Sam Tobin-Hochstadt
PLT @ Northeastern University

IFL, September 3, 2010

```
(: magnitude : Complex -> Real)
(define (magnitude n)
  (cond
    [(eq? (first n) 'cart)
     (sqrt (+ (sqr (second n))
              (sqr (third n))))]
    [(eq? (first n) 'polar)
     (second n)]))
```

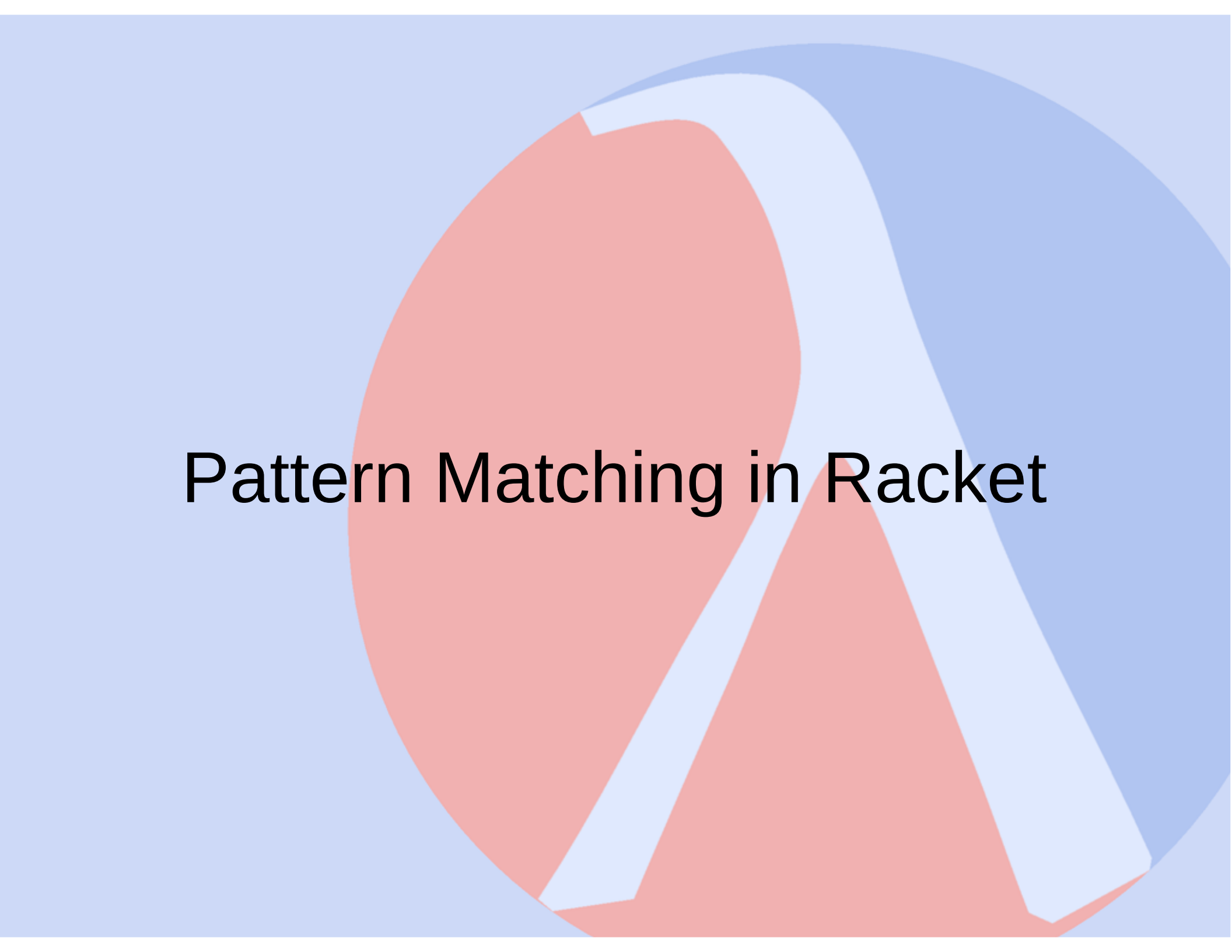
```
(: magnitude : Complex -> Real)
(define (magnitude n)
   (if (not (pair? n))
       (error 'bad-input)
       (let ([t1 (first n)]
             [t1* (rest n)])
         (if (not (pair? t1*))
             (error 'bad-input)
             (let ([t2 (first t1*)]
                   [t2* (rest t1*)])
               (if (not (pair? t3))
                   (error 'bad-input)
                   (let ([t3 (first t2*)]
                         [t3* (rest t2*)])
                     (if (not (null? t3))
                         (error 'bad-input))
                     (cond [(eq? t1 'cart)
                            (sqrt (+ (sqr t2) (sqr t3)))]
                           [(eq? t1 'polar)
                            t2]
                           [else (error 'bad-input)])))))))))
```

```
(: magnitude : Complex -> Real)
(define (magnitude n)
  (match n
    [(list 'cart x y)
     (sqrt (+ (sqr x) (sqr y)))]
    [(list 'polar r theta)
     r]))
```

```
(: magnitude : Complex -> Real)
(define (magnitude n)
  (match n
    [(list 'cart xs ...)
     (sqrt (apply + (map sqr xs)))]
    [(list 'polar r theta ...)
     r]))
```

```
(: magnitude : Complex -> Real)
(define (magnitude n)
  (match n
    [(cart xs ...)
     (sqrt (apply + (map sqr xs)))]
    [(polar r theta ...)
     r]))
```

```
(: magnitude : Complex -> Real)
(define (magnitude n)
  (match n
    [(polar r theta ...) r]))
```


Pattern Matching in Racket

`match` works for arbitrary data

```
(match e
  [(list a b) (+ a b)]
  [(? string? a) (string-length a)]
  [(? number? a) a])
```

`match` provides expressive patterns

```
(match e
  [(app add1 n) n])
```

`match` is an optimizer

```
(match e
  [(list (? B?)) do-something-else])
```

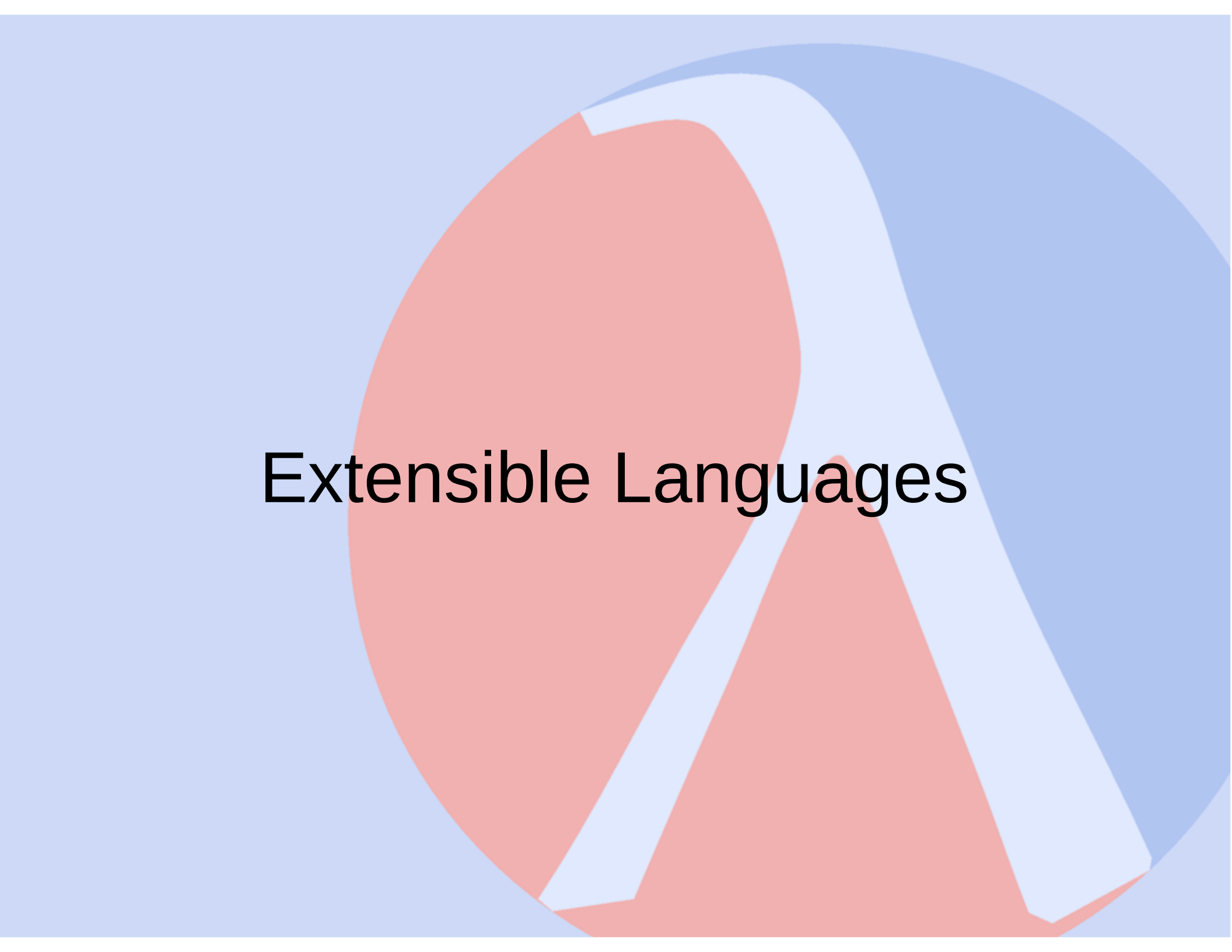
[Le Fessant & Maranget]

`match` supports recursive patterns

```
(match (list 2 4 6 8 10)
  [(list (? even? y) ...)
   (foldr + 0 y)])
```

`match` supports recursive patterns

```
(match '(3 2 1 3)
  [(list-no-order 1 2 3 ...) 'yes]
  [_ 'no])
```


Extensible Languages

Simple Language Extension

```
(define-syntax
  (let ([x e] ...) body)
  ((lambda (x ...) body) e ...))
```

```
(let ([x 1] [y 2]) (+ x y))
```

Simple Language Extension

```
(define-syntax
  (let ([x e] ...) body)
  ((lambda (x ...) body) e ...))
```

```
(let ([x 1] [y 2]) (+ x y))
```

```
((lambda (x y) (+ x y)) 1 2)
```

Simple Language Extension

```
(define-syntax
  (let ([x e] ...) body)
  ((lambda (x ...) body) e ...))
```

```
(let ([x 1] [y 2]) (+ x y))
```

```
((lambda (x y) (+ x y)) 1 2)
```

[Kohlbecker et al, 1980s]

Adding Computation

```
(define-syntax (numbers start end)
  (list (in-range start end)))
```

```
(numbers 1 10)
```

Adding Computation

```
(define-syntax (numbers start end)
  (list (in-range start end)))
```

```
(numbers 1 10)
```

```
(list 1 2 3 4 5 6 7 8 9 10)
```

Adding Computation

```
(define-syntax (numbers start end)
  (list (in-range start end)))
```

```
(numbers 1 10)
```

```
(list 1 2 3 4 5 6 7 8 9 10)
```

[Dybvig et al, 1990s]

Racket

Modular Language Extension

Compiler API

Arbitrary Language Rewriting

. . .

Racket

Modular Language Extension

Compiler API

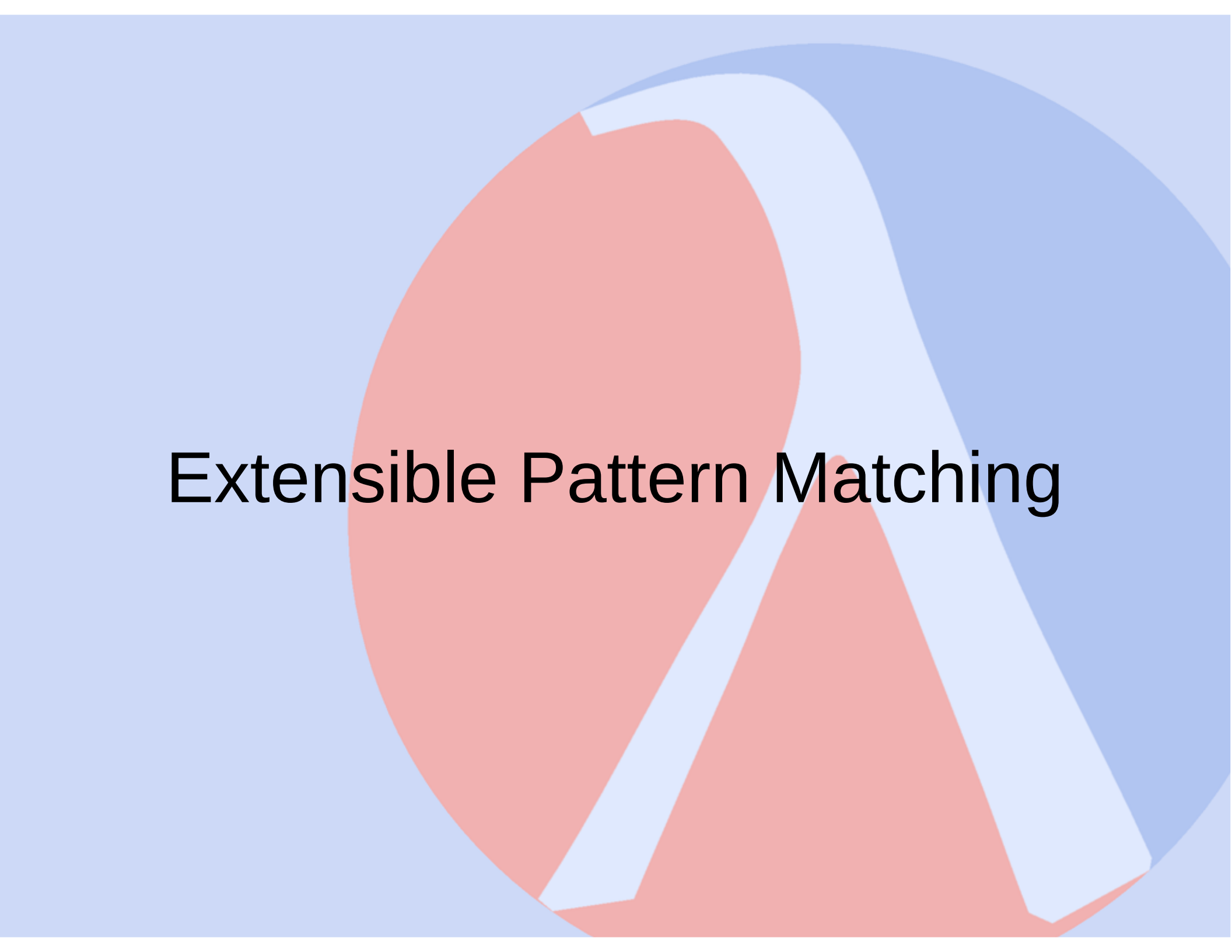
Arbitrary Language Rewriting

. . .

[Flatt et al, 2000s]

```
(define-syntax x 1)

(define-syntax (get-x)
  (syntax-value x))


(get-x)
```

```
(define-syntax x 1)

(define-syntax (get-x)
  (syntax-value x))


(get-x)

1
```


Extensible Pattern Matching

```
(define-syntax (let ([x e] ...) b)
  ((lambda (x ...) b) e ...))
```

```
(define-syntax (let ([x e] ...) b)
  ((lambda (x ...) b) e ...))

(define-matcher (not-false p)
  (? (compose not false?) p))
```

The core of `match`

```
(define (parse-pattern pat)
  (syntax-case pat

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)

                                                ]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)
     #:when (bound-to-match-expander? id)



                                          ]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)
     #:when (bound-to-match-expander? id)

                                    (syntax-value id)

                                          ]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)
     #:when (bound-to-match-expander? id)

            (match-expander-fn (syntax-value id))

                                        ]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)
     #:when (bound-to-match-expander? id)
     (let ([transformer
            (match-expander-fn (syntax-value id))])

                                        )]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)
     #:when (bound-to-match-expander? id)
     (let ([transformer
            (match-expander-fn (syntax-value id))])

       (transformer (id pats ...)) )]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

The extended core

```
(define (parse-pattern pat)
  (syntax-case pat

    [(id pats ...)
     #:when (bound-to-match-expander? id)
     (let ([transformer
            (match-expander-fn (syntax-value id))])
       (parse-pattern
        (transformer (id pats ...))))]

    [(cons pat1 pat2) ...]
    [(? pred pat) ...]
    ...))
```

An Example

```
(define-matcher (not-false p) ...)

(match (list 7 #f)
  [(list (not-false x) ... y) x])
```

An Example

```
(define-syntax not-false
  (match-expander ...))
(match (list 7 #f)
  [(list (not-false x) ... y) x])
```

An Example

```
(define-syntax not-false
  (match-expander ...))
(match (list 7 #f)
  [(list (not-false z) ... y) z])
```

```
(let ([transformer
       (match-expander-fn (syntax-value not-false))])
  (parse-pattern (transformer (not-false z))))
```

An Example

```
(define-syntax not-false
  (match-expander ...))
(match (list 7 #f)
  [(list (not-false z) ... y) z])
```

```
(? (compose not false?) z)
```

An Example

```
(define-syntax not-false
  (match-expander ...))
(match (list 7 #f)
  [(list (? (compose not false?) z) ... y) z])
```

Applications

Views [Wadler 87] as a library

```
(require (planet cobbe/views/views))
(define-view Zero zero? ())
(define-view Succ
  exact-positive-integer? (sub1))
(define (even? n)
  (match n
    [(Zero) true]
    [(Succ (Zero)) false]
    [(Succ (Succ n)) (even? n)]))
```

Web Server Dispatching

```
(dispatch-rules
 [("") list-posts]
 [("posts" (string-arg)) review-post]
 [("archive" (integer-arg) (integer-arg))
  review-archive]
 [else list-posts])
```

Other Extensible Systems

View Patterns [Peyton-Jones et al]: **`app`** patterns

Views [Wadler]: **`define-matcher`** and **`app`**

Active Patterns [Syme et al]: Multiple uses of **`define-matcher`**, **`app`**, and **`?`**

Pattern matching is great

Extensible pattern matching is even better

An expressive and extensible language can give us both


Thanks!
Available at racket-lang.org