Languages as Libraries

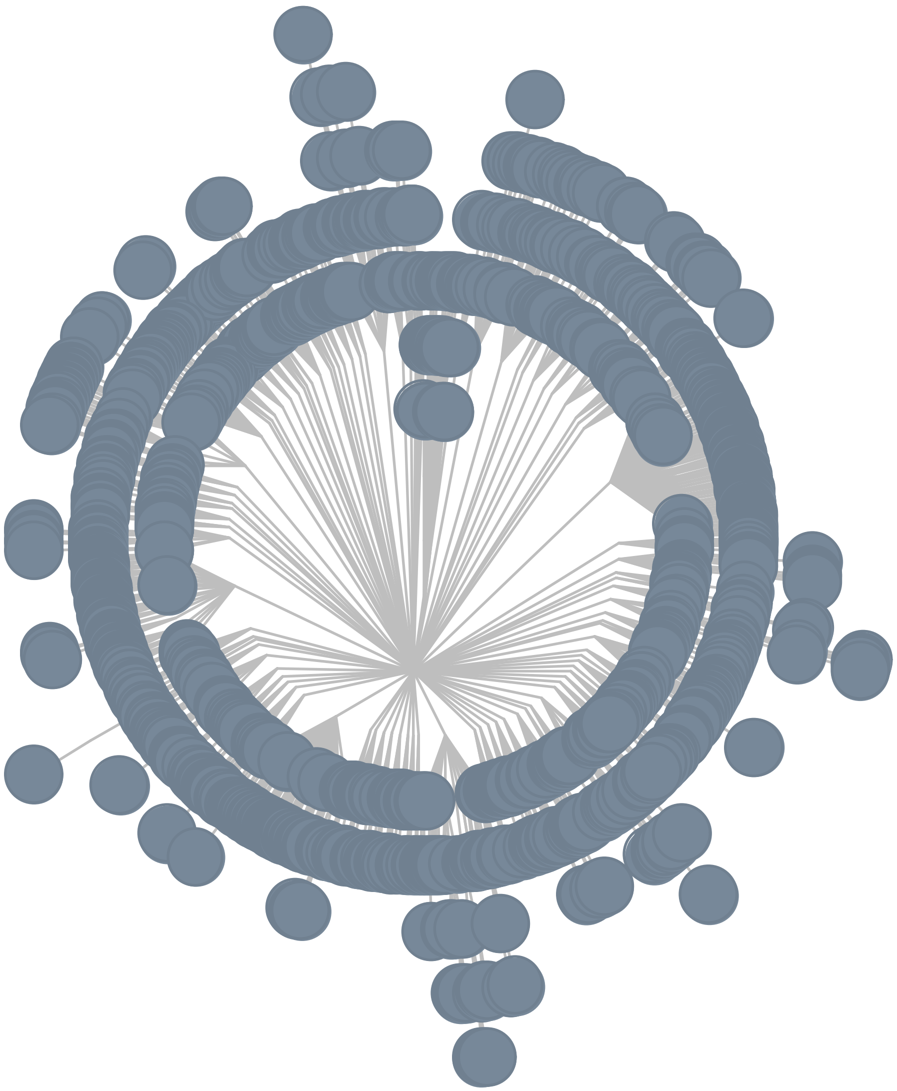
or, implementing the next 700 programming languages

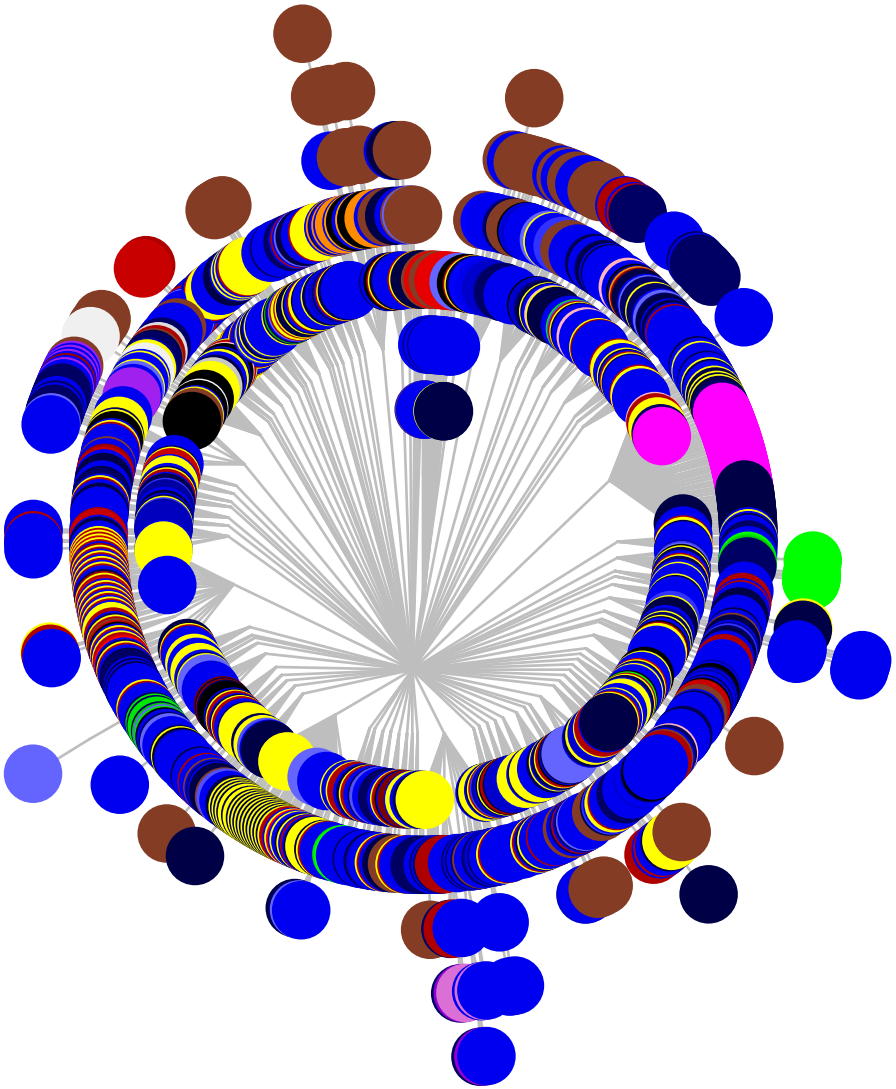
Sam Tobin-Hochstadt
PLT @ Northeastern University

May 11, 2011 MIT

"A domain specific language is the ultimate abstraction."

— Paul Hudak





- mzscheme
- racket
- racket/private
- racket/unit
- racket/private/base
- #%kernel
- racket/load
- racket/base
- racket/private/provider
- racket/signature
- slideshow
- racket/gui
- at-exp scheme/base
- at-exp racket/base
- scribble/doc
- scribble/manual
- scribble/base/reader
- scribble/lp
- deinprogramm/DMdA
- htdp/asl
- htdp/isl+
- htdp/bsl
- frtime/lang-utils
- frtime/frtime-lang-only
- frtime
- syntax/module-reader
- web-server/insta
- meta/web
- web-server
- srfi/provider
- typed/racket
- typed-scheme/minimal
- r6rs
- r5rs
- setup/infotab
- everything else

Racket ships more than

40 documented languages

```
#lang racket ; An echo server
(define listener (tcp-listen 12345))
(let echo-server ()
  (define-values (in out) (tcp-accept listener))
  (thread (lambda () (copy-port in out)
                     (close-output-port out)))
  (echo-server))
```

A modern programming language

```
#lang web-server/insta
; A "hello world" web server
(define (start request)
  (response/xexpr
   '(html
     (body "Hello World"))))
```

A language for writing web servers

```
#lang lazy
; An infinite list:
(define fibs
  (list* 1 1 (map + fibs (cdr fibs))))
; Print the 1000th Fibonacci number:
(print (list-ref fibs 1000))
```

Lazy evaluation

```
#lang scribble/base
@; Generate a PDF or HTML document
@title{Bottles --- @italic{Abridged}}
@(apply itemlist
   (for/list ([n (in-range 100 0 -1)])
     @item{@(format "~a" n) bottles.}))
```

A domain-specific language (and syntax) for documentation

```
#lang datalog
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :-
  parent(A, C), D = C, ancestor(D, B).
parent(john, douglas).
parent(bob, john).
ancestor(A, B)?
```

Integrated logic programming

```
#lang typed/racket
(struct: person ([first : String]
                 [last  : String]))
(: greeting (person -> String))
(define (greeting n)
  (format "~a ~a"
          (person-first n) (person-last n)))
(greeting (make-person "Bob" "Smith"))
```

Racket with static types and full interoperation

All built with macros

- Comprehensions
- Recursive Modules
- Logic Programming
- Classes, Mixins, Traits
- Generic Methods a la CLOS
- Documentation
- Contracts
- Lazy Programming
- Web Programming
- Lexing + Parsing
- Teaching

All built with macros

- Comprehensions
- Recursive Modules
- Logic Programming
- Classes, Mixins, Traits
- Generic Methods a la CLOS
- Documentation
- Contracts
- Lazy Programming
- Web Programming
- Lexing + Parsing
- Teaching
- Typechecking

Are Macros Hard?

No

Macros are just programming with trees

Yes

Macros are metaprogramming

Are Macros Hard?

No

Macros are just programming with trees

Yes

Macros are metaprogramming

Maybe

Macros are our best tool for managing metaprogramming

The Plan

Pre-history: Macros to 1985 (Lisp)

The Hygiene Revolution (Scheme)

A Rich Macro API (Racket)

On to Typed Racket

Macros in Lisp

What is a macro?

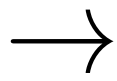
```
(defmacro (let-pair . args)
  `(let ((,(car args) (car ,(caddr args)))
         (,(cadr args) (cdr ,(caddr args))))
     ,(cadddr args)))
```

What is a macro?

```
(defmacro (let-pair . args)
  `(let ((,(car args) (car ,(caddr args)))
         (,(cadr args) (cdr ,(caddr args))))
     ,(cadddr args)))

(let-pair a b (cons 1 2) (+ a b))
```

What is a macro?

```
(defmacro (let-pair . args)
  `(let ((,(car args) (car ,(caddr args)))
         (,(cadr args) (cdr ,(caddr args))))
     ,(cadddr args)))

(let-pair a b (cons 1 2) (+ a b))
```



```
(let ((a (car (cons 1 2)))
      (b (cdr (cons 1 2))))
  (+ a b))
```

How Lisp macros work

To evaluate `(let-pair`  `)`:

1. Call the definition of `let-pair` with 
2. Then *evaluate* the result

It's great!

Abstract over everything

Build DSLs

Simple reasoning about macros

It's great!

Abstract over everything

Build DSLs

Simple reasoning about macros

It's terrible!

Hard to program: witness caddrs

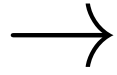
Imagine the error messages

Inherently buggy

Hygenic Macros in Scheme

Pattern-matching macros

```
(define-syntax-rule (let-pair x y p body)
  (let ([x (car p)] [y (cdr p)]) body))

(let-pair a b (cons 1 2) (+ a b))
```



```
(let ((a (car (cons 1 2)))
      (b (cdr (cons 1 2))))
  (+ a b))
```

Macro by Example [Kohlbecker & Wand 87]

The hygiene bug

```
(define-syntax-rule (let-pair x y p body)
  (let ([x (car p)] [y (cdr p)]) body))

(let-pair a b (cons 1 2) (+ a b))
```

The hygiene bug

```
(define-syntax-rule (let-pair x y p body)
  (let* ([a p] [x (car a)] [y (cdr a)]) body))

(let-pair a b (cons 1 2) (+ a b))
```

The hygiene bug

```
(define-syntax-rule (let-pair x y p body)
  (let* ([a p] [x (car a)] [y (cdr a)]) body))

(let-pair a b (cons 1 2) (+ a b))
```

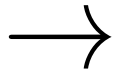
$\longrightarrow$

```
(let* ([a (cons 1 2)]
       [a (car a)]
       [b (cdr a)])
  (+ a b))
```

Uh-oh

The solution

```
(define-syntax-rule (let-pair x y p body)
  (let* ([a p] [x (car a)] [y (cdr a)]) body))

(let-pair a b (cons 1 2) (+ a b))
```



```
(let* ([a1 (cons 1 2)]
       [a (car a1)]
       [b (cdr a1)])
  (+ a b))
```

[Kohlbecker et al 88, Clinger & Rees 91]

It's great!

Abstract over everything

Build DSLs

Simple reasoning about macros

Pattern matching

Automatic Hygiene

It's great!

Abstract over everything

Build DSLs

Simple reasoning about macros

Pattern matching

Automatic Hygiene

It's terrible!

Limited language

Error messages still not good

syntax-case and syntax objects

```
(define-syntax (let-pair stx)
  (syntax-case stx ()
    [(_ x y p body)

     #'(let* ([a p] [x (car a)] [y (cdr a)])
         body)]))

(let-pair a b (cons 1 2) (+ a b))
```

$\longrightarrow$

```
(let* ([a p] [a (car p)] [b (cdr p)])
  (+ a b))
```

[Dybvig et al, 91]

syntax-case and syntax objects

```
(define-syntax (let-pair stx)
  (syntax-case stx ()
    [(_ x y p body)

     #'(let* ([a p] [x (car a)] [y (cdr a)])
         body)]))

(let-pair 0 1 (cons 1 2) (+ a b))
```

$\longrightarrow$

```
let*: bad syntax (not an identifier) in: 0
```

[Dybvig et al, 91]

syntax-case and syntax objects

```
(define-syntax (let-pair stx)
  (syntax-case stx ()
    [(_ x y p body)
     (and (identifier? #'x) (identifier? #'y))

     #'(let* ([a p] [x (car a)] [y (cdr a)])
         body)]))

(let-pair 0 1 (cons 1 2) (+ a b))
```

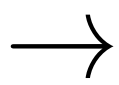
$\longrightarrow$

```
let-pair: bad syntax
```

[Dybvig et al, 91]

syntax-case and syntax objects

```
(define-syntax (let-pair stx)
  (syntax-case stx ()
    [(_ x y p body)
     (unless (identifier? #'x)
       (raise-syntax-error "expected ..."))
     #'(let* ([a p] [x (car a)] [y (cdr a)])
         body)]))

(let-pair 0 1 (cons 1 2) (+ a b))
```



```
let-pair: expected an identifier first
```

[Dybvig et al, 91]

It's great!

Abstract over everything

Build DSLs

Use the full language

Pattern matching

Automatic Hygiene

Good error reporting

It's great!

Abstract over everything

Build DSLs

Use the full language

Pattern matching

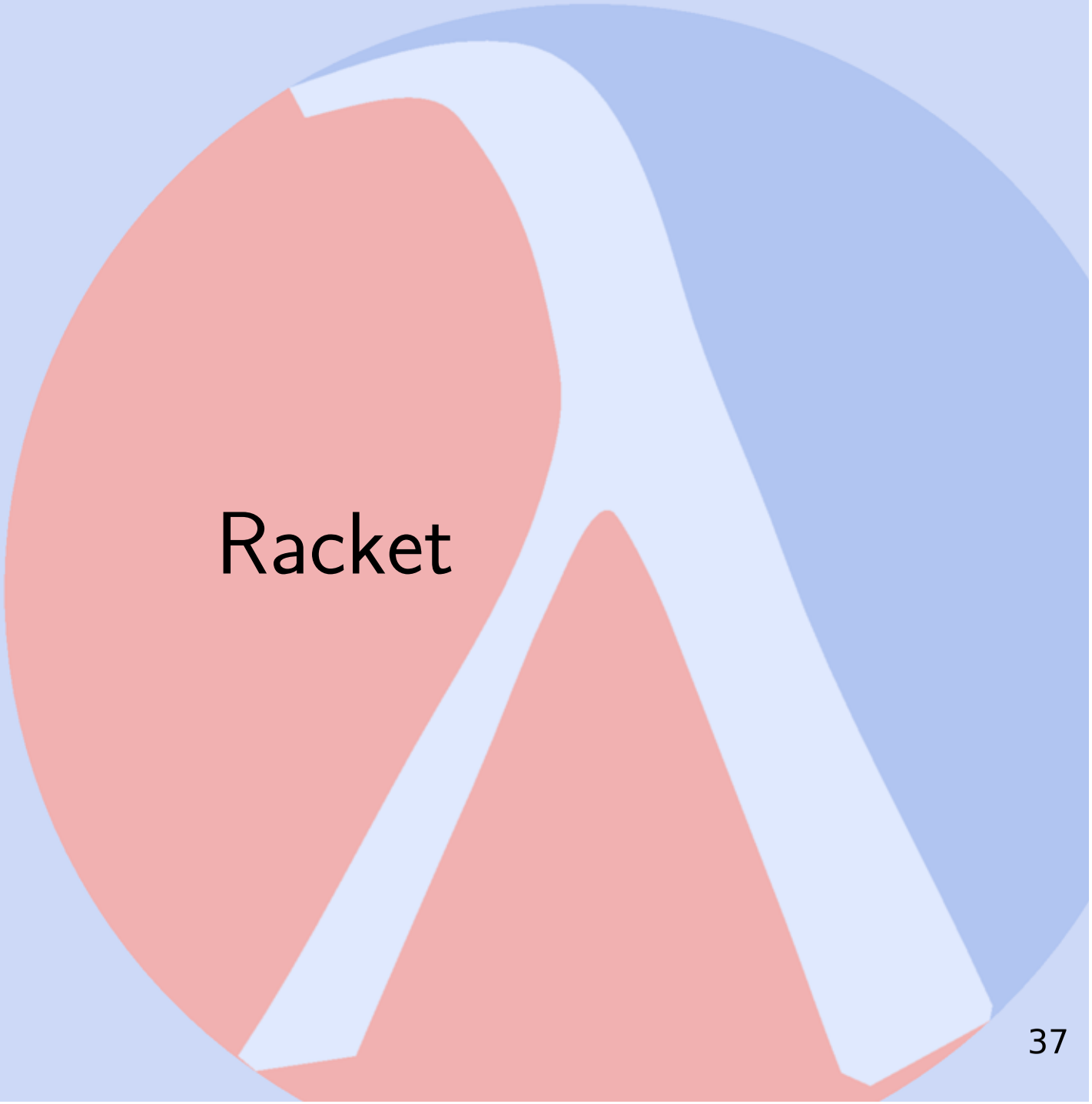
Automatic Hygiene

Good error reporting

What's left?

Hard to understand

Scale up from macros to languages


Racket

Macros and modules

util

```
#lang       racket

(define (f x) ...)
(provide f)
```

mac

```
#lang       racket

(require (for-syntax util))
(define-syntax (mac stx)
  (f stx))
```

A powerful API

Extensible macros

Static binding

Just-once evaluation

Certificates for integrity

...

Source tracking

Report errors using the original program

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

Source tracking

Report errors using the original program

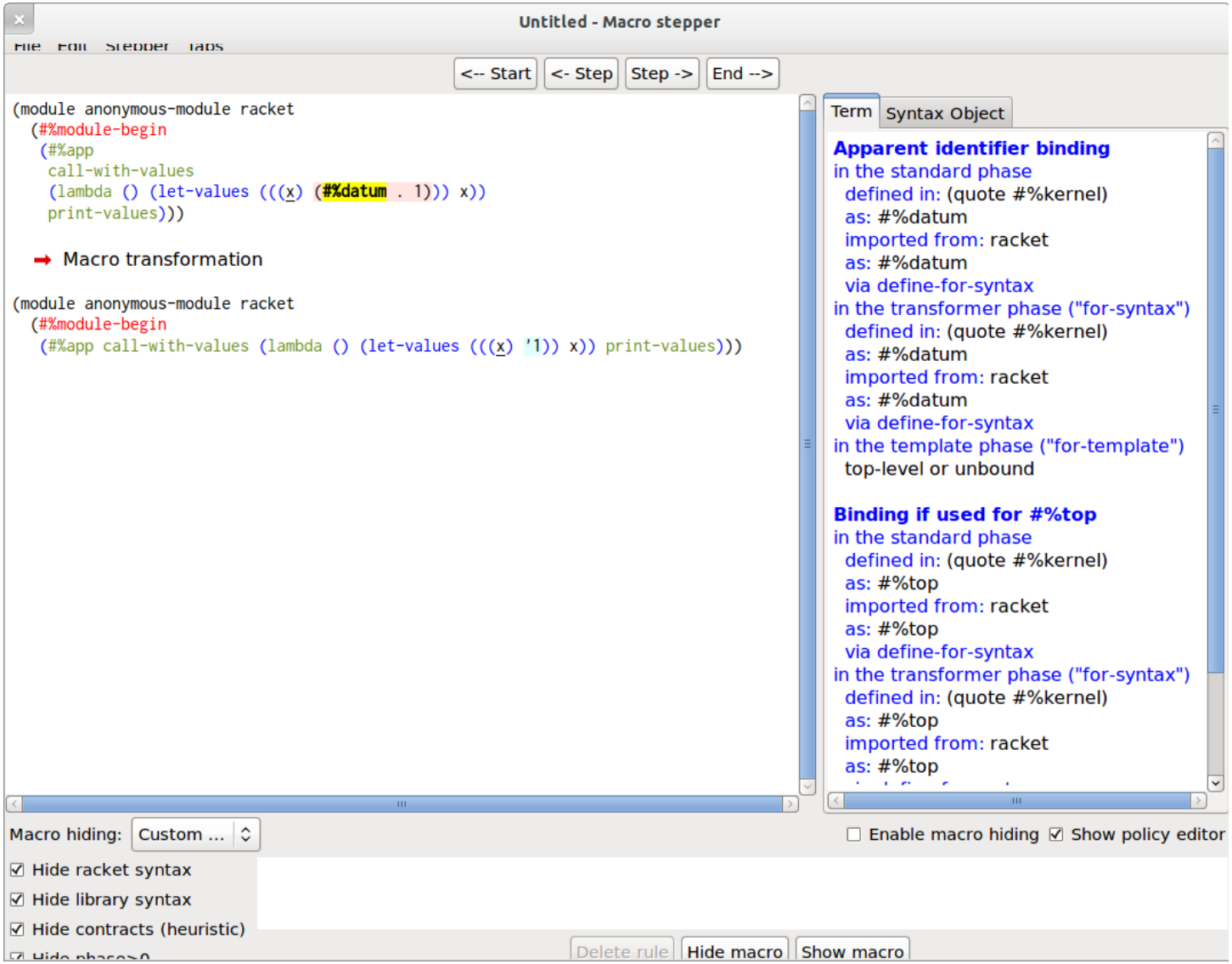
```
(let ([x 1]
      [2 y])
  (+ x y))
```

Tools and experience

```
(define-syntax let-pair
  (syntax-parser
   [(_ x:id y:id p:expr body:expr)
    #'(let* ([a p] [x (car a)] [y (cdr a)])
        body)]))
```

Declarative, Robust, Programmed

Tools and experience



Tools and experience

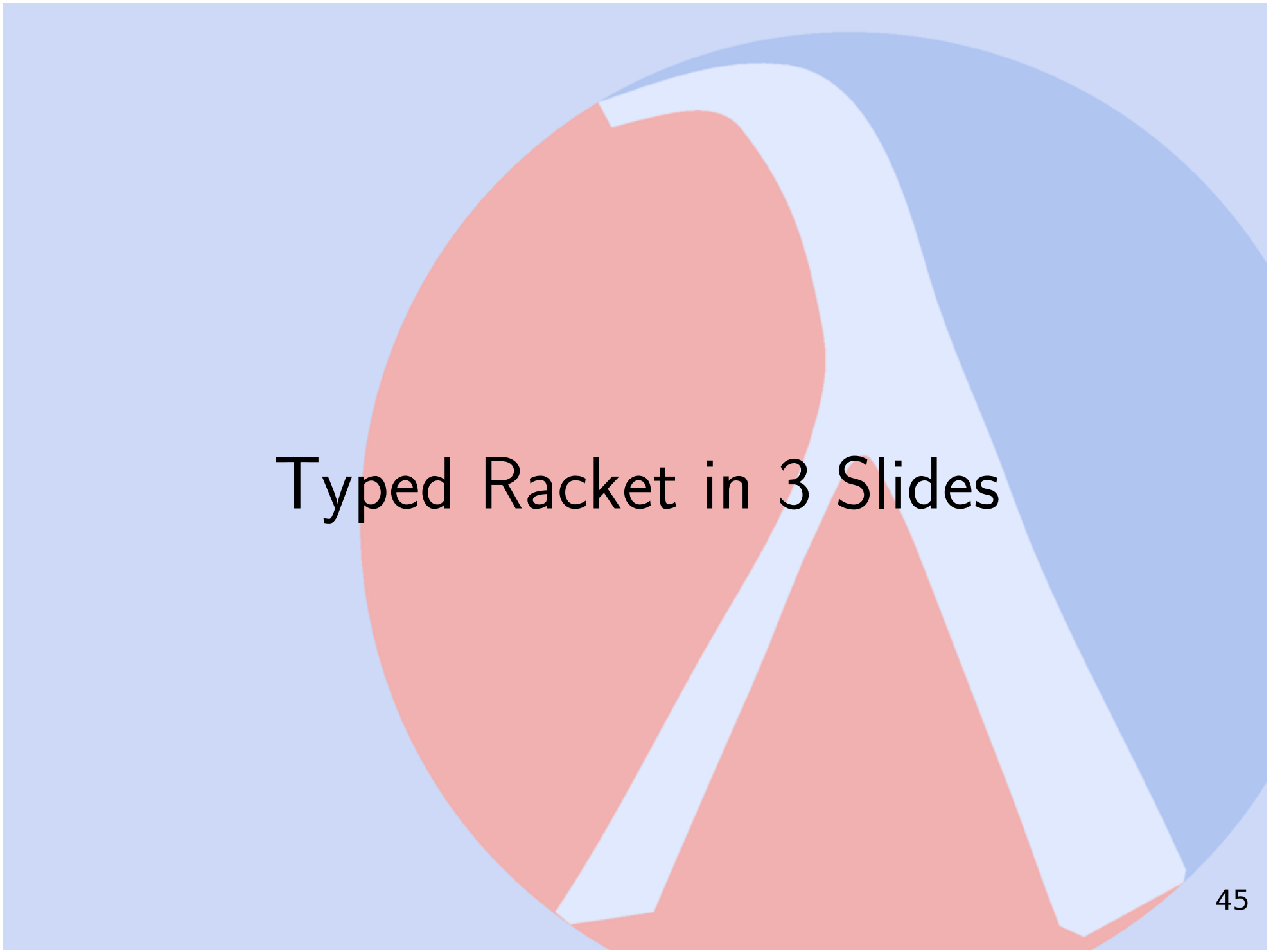
15 years and 500,000 lines of macrology

[Flatt et al 02, 06, 09]

[Culpepper et al 04, 05, 07, 10]

[Eastlund et al 10]

[McCarthy et al 08]


Typed Racket in 3 Slides

Hello World

```
#lang racket                                    hello

(printf "Hello World\n")
```

Hello World

```
#lang typed/racket                                    hello

(printf "Hello World\n")
```

Functions

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Functions

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Modules

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang        racket

(require ack)

(ack 2 3)
```

Modules

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang       racket

(require ack)

(ack 2 3)
```

Modules

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang typed/racket

(require [ack
          (Integer Integer -> Integer)])
(ack 2 3)
```

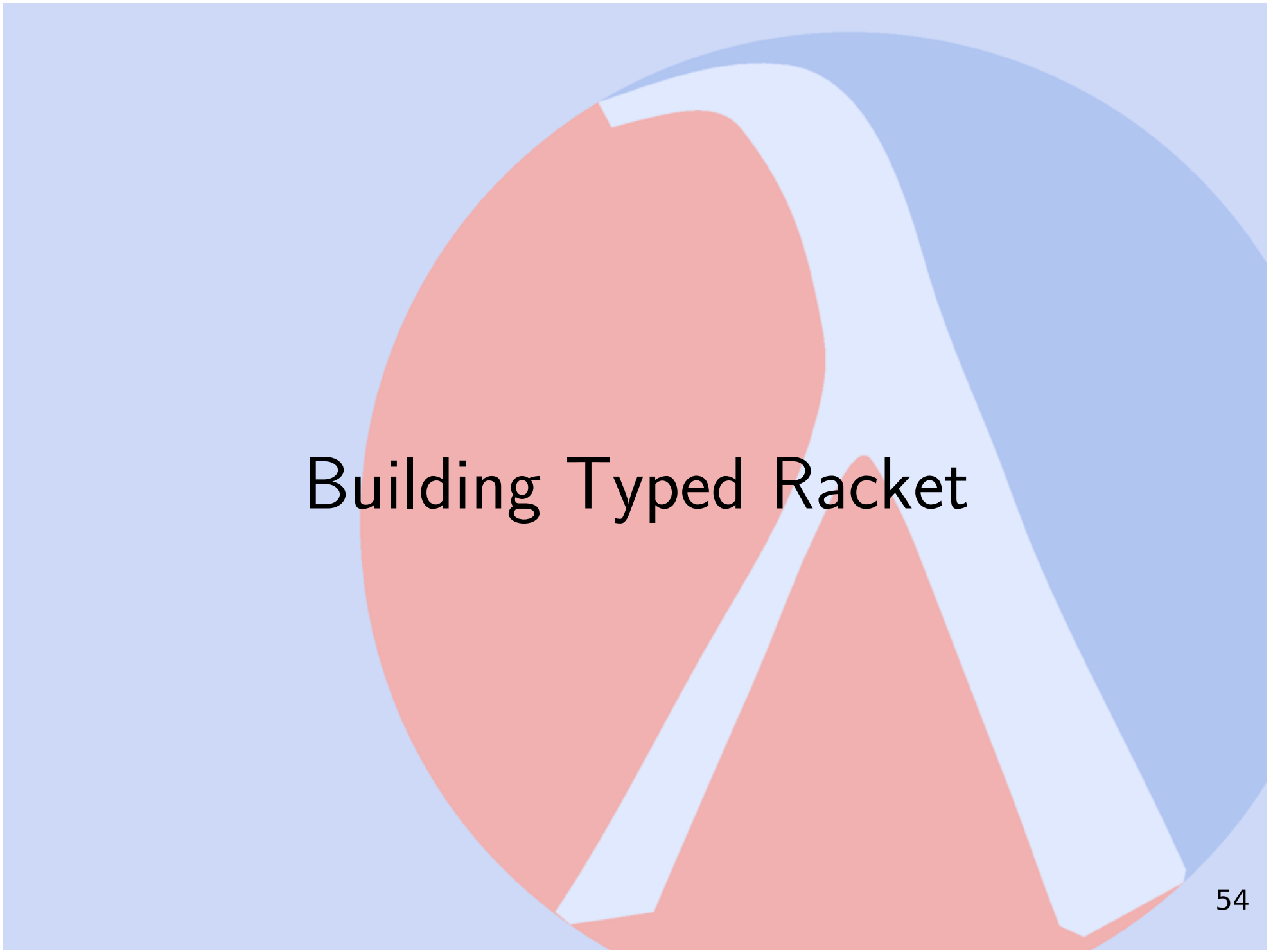
Modules

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang typed/racket

(require ack)

(ack 2 3)
```


Building Typed Racket

Whole modules

Static semantics requires control

ack

```
#lang typed/racket


(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Whole modules

Static semantics requires control

ack

```
#lang typed/racket

(#%module-begin
 (: ack : Integer Integer -> Integer)
 (define (ack m n)
   (cond [(<= m 0) (+ n 1)]
         [(<= n 0) (ack (- m 1) 1)]
         [else (ack (- m 1) (ack m (- n 1)))]))

 (ack 2 3))
```

Whole modules

Static semantics requires control

ack

```
#lang typed/racket

(#%module-begin
 (: ack : Integer Integer -> Integer)
 (define (ack m n)
   (cond [(<= m 0) (+ n 1)]
         [(<= n 0) (ack (- m 1) 1)]
         [else (ack (- m 1) (ack m (- n 1)))]))

 (ack 2 3))
```

Defining A Language

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(_ forms ...)
     (let ()
       (for ([f (syntax->list #'(forms ...))])
         (typecheck f))

       #'(#%plain-module-begin forms ...))]))
```

Typechecking

```
(define (typecheck f)
  (syntax-parse f
    ; variables
    [v:identifier
     (lookup-type #'v)]
    ; abstractions
    [(lambda (x) e)
     (define t (syntax-property #'x 'type-label))
     (set-type! #'x t)
     (typecheck #'e)]
    ; about 10 more cases
    ....))
```

Typechecking

```
(define (typecheck f)
  (syntax-parse f
    ; variables
    [v:identifier
     (lookup-type #'v)]
    ; abstractions
    [(lambda (x) e)
     (define t (syntax-property #'x 'type-label))
     (set-type! #'x t)
     (typecheck #'e)]
    ; about 10 more cases
    ....))
```

Defining A Language

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(_ forms ...)
     (let ()
       (for ([f (syntax->list #'(forms ...))])
         (typecheck f))

       #'(#%plain-module-begin forms ...))]))
```

local-expand

Core forms support arbitrary macros

```
(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

Discover static semantics by expansion

Defining A Language

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(_ forms ...)
     (let ([forms* (local-expand #'(forms ...))])
       (for ([f (rest (syntax->list forms*))])
         (typecheck f))

       #'(#%plain-module-begin #,forms*))]))
```

Typechecking

```
(define (typecheck f)
  (syntax-parse f
    ; variables
    [v:identifier
     (lookup-type #'v)]
    ; abstractions
    [(lambda (x) e)
     (define t (syntax-property #'x 'type-label))
     (set-type! #'x t)
     (typecheck #'e)]
    ; about 10 more cases
    ....))
```

Syntax properties provide side-channels

Defining A Language

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(_ forms ...)
     (let ([forms* (local-expand #'(forms ...))])
       (for ([f (rest (syntax->list forms*))])
         (typecheck f))

       #'(#%plain-module-begin #,forms*))]))
```

Defining A Language

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(_ forms ...)
     (let ([forms* (local-expand #'(forms ...))])
       (for ([f (rest (syntax->list forms*))])
         (typecheck f))
       (let ([forms** (optimize forms*)])
         #'(#%plain-module-begin #,forms**)))]))
```

Optimization

Express guarantees as rewritings

```
(: norm : Float Float -> Float)
(define (norm x y)
  (sqrt (+ (sqr x) (sqr y))))
```

Optimization

Express guarantees as rewritings

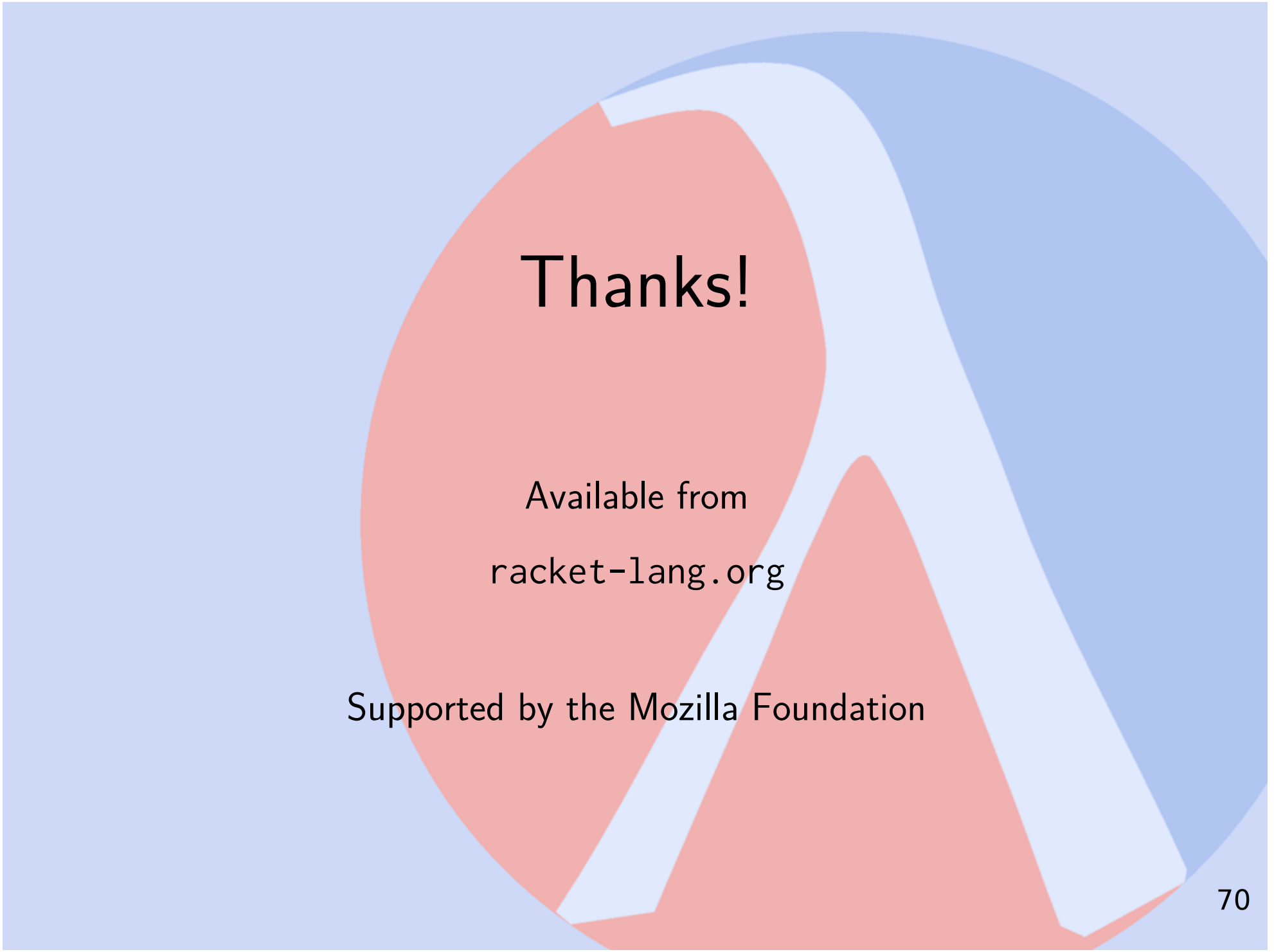
```
(: norm : Float Float -> Float)
(define (norm x y)
  (unsafe-flsqrt
   (unsafe-fl+ (unsafe-fl* x x) (unsafe-fl* y y))))
```

The take-away

Language are powerful abstractions

Racket helps us build new, integrated languages

- Whole-module control
- Local expansion
- Optimization via rewriting


Thanks!
Available from
racket-lang.org
Supported by the Mozilla Foundation