MANY MACROS, TONS OF TYPES

Matthias Felleisen (PLT)
Northeastern University

PLT Scheme is a programming language designed for conducting research and running educational programs that is *also* used for commercial purposes.

Don't run out and switch your heavy-duty commercial e-business site to the PLT Scheme web server without conducting a feasibility study.

Instead, this talk is about ideas that you can take away and apply to your Lisp.

PLT Scheme is really many different languages.

There is **PLT Scheme** proper.

There is **R6R Scheme**.

We have **Typed Scheme**.

For real programmers, there's **Lazy Scheme**.

And documentation is a program, too. For that we provide **Scribble.**

PLT Scheme is really many different languages.

There is **PLT Scheme** proper.

There is **R6R Scheme**.

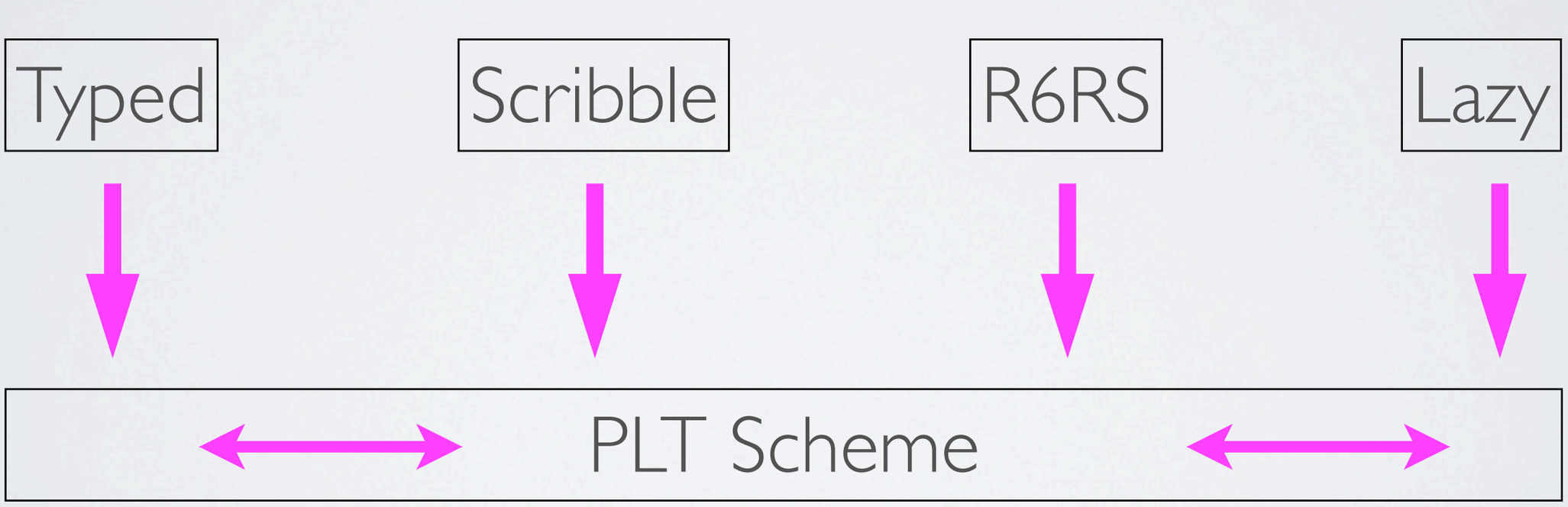
We have **Typed Scheme**.

For real programmers, there's **Lazy Scheme**.

Productivity: Systems consist of many modules each written in one of these "dialects" and importing/exporting from each other.

And documentation is a program, too. For that we provide **Scribble.**

And it all works via macro expansion and a high level of value communication.



doc.scrbl

```
#lang scribble/base

@title{The Fibonacci Sequence}

The Fibbonacci sequence begins with two copies of
the number 1 and continues @emph{forever} by adding
the two most recent numbers together to get the next
number. The first seven numbers of the sequence are
1, 1, 2, 3, 5, 9, 14, ... because 1 + 1 is 2,
2 + 3 is 5, and so on.

@section{Fibs in nature}

It is a well-known rumor that rabbits ...
```

doc.scrbl

```
#lang scribble/base

@title{The Fibonacci Sequence}

The Fibbonacci sequence begins with two copies of
the number 1 and continues @emph{forever} by adding
the two most recent numbers together to get the next
number. The first seven numbers of the sequence are
1, 1, 2, 3, 5, 9, 14, ... because 1 + 1 is 2,
2 + 3 is 5, and so on.

@section{Fibs in nature}

It is a well-known rumor that rabbits ...
```

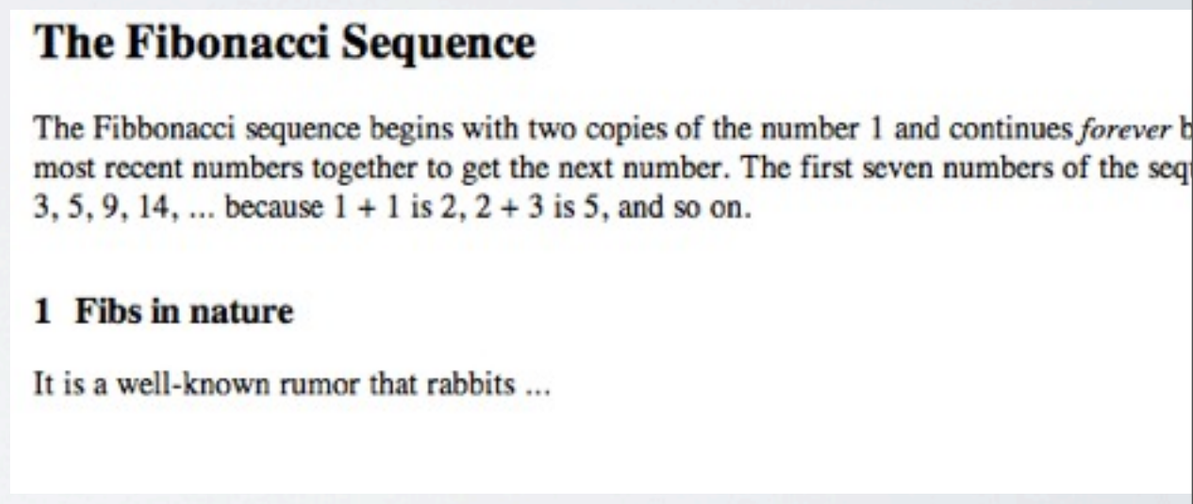
expand

```
(module doc scribble/base/lang
  (#%module-begin
   doc
   values
   ()
   "\n"
   "\n"
   (title "The Fibonacci Sequence")
   "\n"
   "\n"
   "The Fibbonacci sequence begins with two copies of"
   "\n"
   "the number 1 and continues "
   (emph "forever")
   " by adding"
   "\n"
   "the two most recent numbers together to get the next"
   "\n"
   "number. The first seven numbers of the sequence are"
   "\n"
   "1, 1, 2, 3, 5, 9, 14, ... because 1 + 1 is 2,"
   "\n"
   "2 + 3 is 5, and so on."
   "\n"
   "\n"
   (section "Fibs in nature")
   "\n"
   "\n"
   "It is a well-known rumor that rabbits ..."))
```

doc.scrbl

```
#lang scribble/base

@title{The Fibonacci Sequence}

The Fibbonacci sequence begins with two copies of
the number 1 and continues @emph{forever} by adding
the two most recent numbers together to get the next
number. The first seven numbers of the sequence are
1, 1, 2, 3, 5, 9, 14, ... because 1 + 1 is 2,
2 + 3 is 5, and so on.

@section{Fibs in nature}

It is a well-known rumor that rabbits ...
```

html



The Fibonacci Sequence

The Fibbonacci sequence begins with two copies of the number 1 and continues *forever* b
most recent numbers together to get the next number. The first seven numbers of the seq
3, 5, 9, 14, ... because 1 + 1 is 2, 2 + 3 is 5, and so on.

1 Fibs in nature

It is a well-known rumor that rabbits ...

doc.scrbl

```
#lang scribble/base

@title{The Fibonacci Sequence}

The Fibbonacci sequence begins with two copies of
the number 1 and continues @emph{forever} by adding
the two most recent numbers together to get the next
number. The first seven numbers of the sequence are
1, 1, 2, 3, 5, 9, 14, ... because 1 + 1 is 2,
2 + 3 is 5, and so on.

@section{Fibs in nature}

It is a well-known rumor that rabbits ...
```

Ouch!

fib.ss

```
#lang lazy

(require "syn-support.ss")

;; fib: 1, 1, 2, 3, 5, ...
(define fib$
  (cons 1
        (cons 1
              ((rec add-2
                 (lambda (str$)
                   (cons (+ (first str$) (second str$))
                         (add-2 (rest str$)))))
               fib$))))


(provide fib$ take)
```

fib.ss

```
#lang lazy

(require "syn-support.ss")

;; fib: 1, 1, 2, 3, 5, ...
(define fib$
  (cons 1
        (cons 1
              ((rec add-2
                 (lambda (str$)
                   (cons (+ (first str$) (second str$))
                         (add-2 (rest str$)))))
               fib$))))


(provide fib$ take)
```

syn-support.ss

```
#lang scheme

(define-syntax-rule
   (rec f e)
   ;; ==>
   (letrec ((f e)) f))

(provide rec)
```

fib.ss

```
#lang lazy

(require "syn-support.ss")

;; fib: 1, 1, 2, 3, 5, ...
(define fib$
  (cons 1
        (cons 1
              ((rec add-2
                 (lambda (str$)
                   (cons (+ (first str$) (second str$))
                         (add-2 (rest str$)))))
               fib$))))

(provide fib$ take)
```

syn-support.ss

```
#lang scheme

(define-syntax-rule
   (rec f e)
   ;; ==>
   (letrec ((f e)) f))

(provide rec)
```

fib$: the stream of fibonacci numbers
take: a library function of the **lazy** lang

doc-v2.scrbl

```
#lang scribble/base

@(require lazy/force "fib.ss")

@title{The Fibonacci Sequence}

@(define fib7 (!! (take 7 fib$)))

The Fibbonacci sequence begins with two copies of
the number 1 and continues @emph{forever} by adding
the two most recent numbers together to get the next
number. The first seven numbers of the sequence are
@(foldr (lambda (f r)
          (string-append (format "~a, " f) r))
        "..."
        fib7)
because 1 + 1 is 2, 2 + 3 is 5, and so on.


@section{Fibs in nature}

It is a well-known rumor that rabbits ..
```

doc-v3.scrbl

```
#lang scribble/base

@(require lazy/force "fib.ss" "lib-support.rkt")

@title{The Fibonacci Sequence}

@(define fib7 (!! (take 7 fib$)))

The Fibbonacci sequence begins with two copies of
the number 1 and continues @emph{forever} by adding
the two most recent numbers together to get the next
number. The first seven numbers of the sequence are
@(foldr (lambda (f r)
          (string-append (format "~a, " f) r))
        "..."
        fib7)
because 1 + 1 is 2, 2 + 3 is 5, and so on. Another way
to illustrate this idea is with this kind of table:

@(fib-tab fib7)
...
```

html

The Fibonacci Sequence

The Fibbonacci sequence begins with two copies of the number 1 and continues *fore*
most recent numbers together to get the next number. The first seven numbers of th
3, 5, 8, 13, ... because 1 + 1 is 2, 2 + 3 is 5, and so on. Another way to illustrate thi
kind of table:

n	n+1	n+2
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13
8	13	...

...

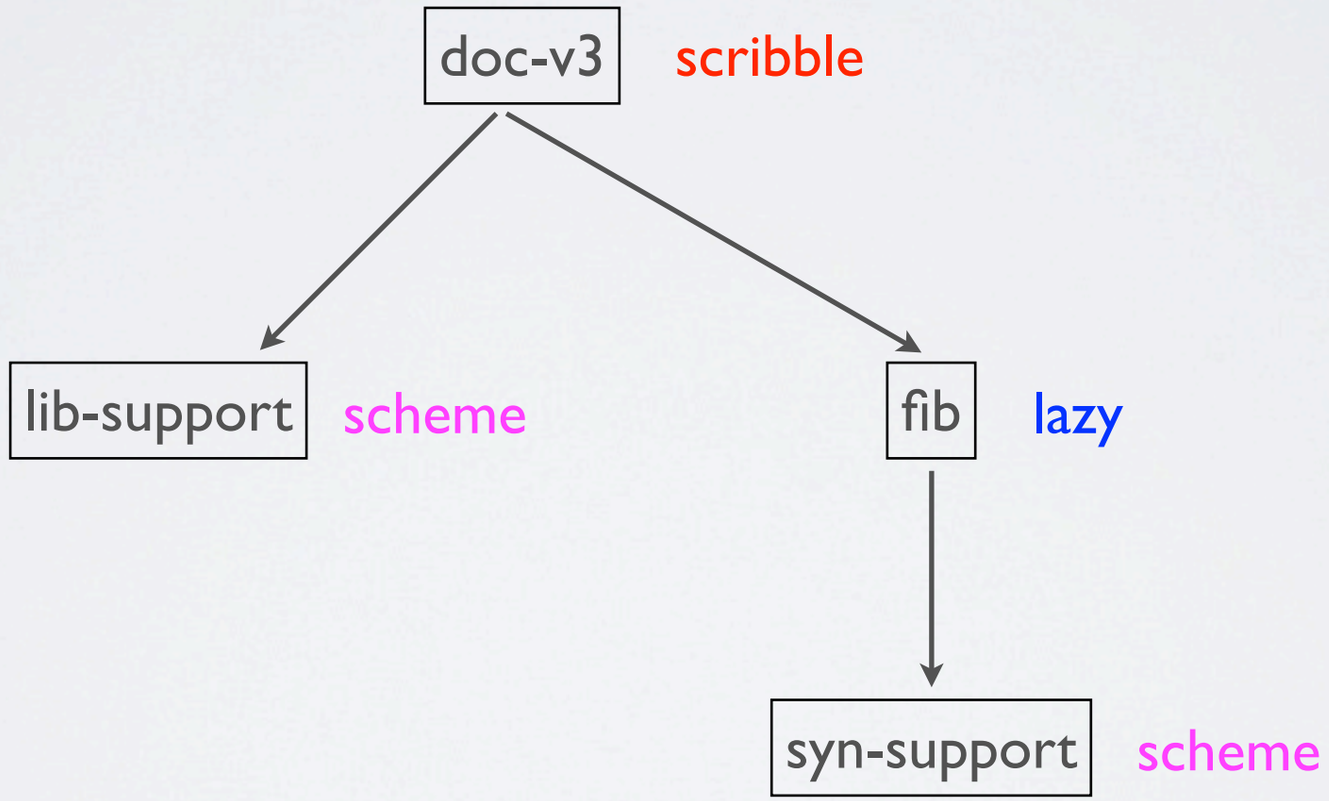
1 Fibs in nature

It is a well-known rumor that rabbits ...

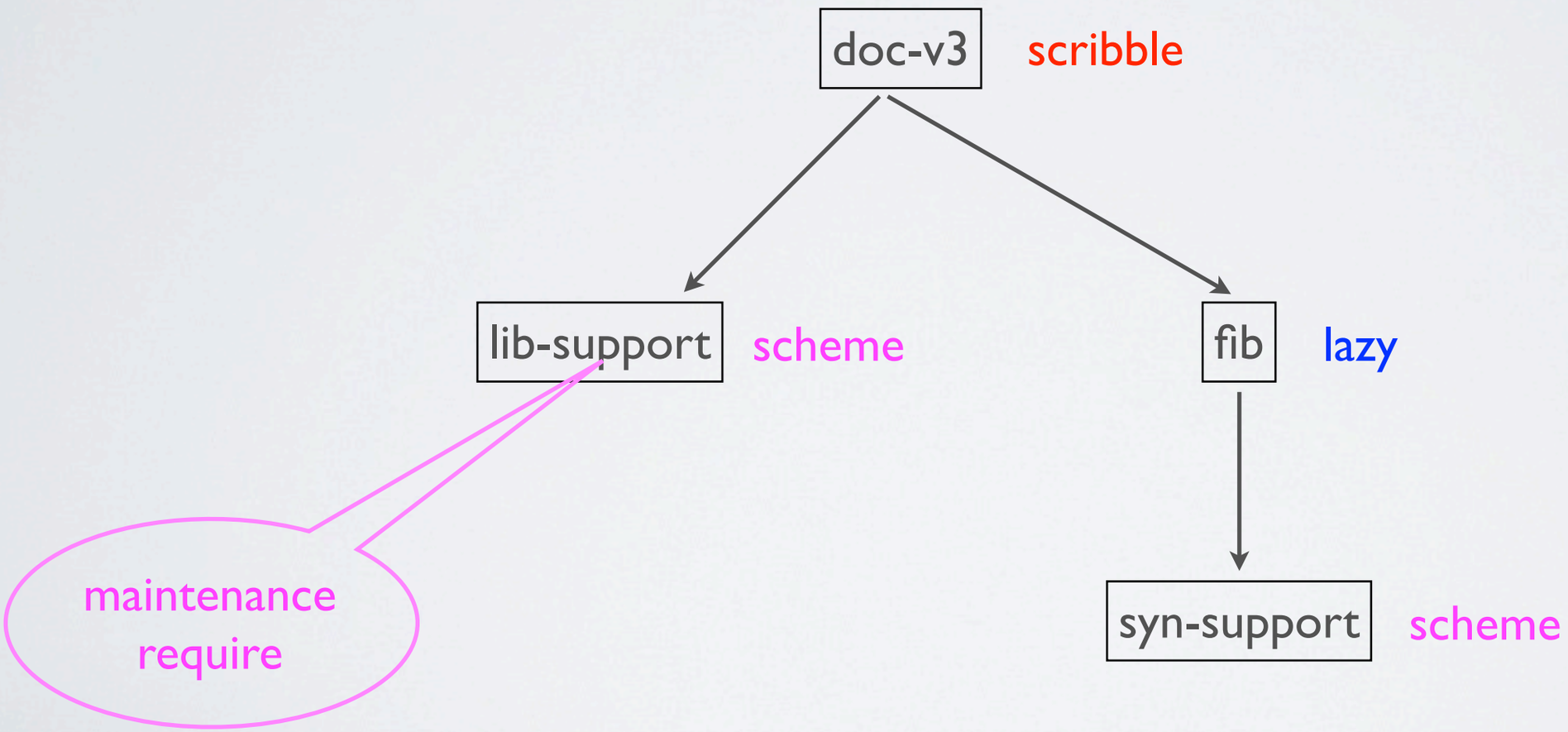
lib-support.ss

```
#lang scheme

;; (cons Natural (cons Natural [Listof Natural])) -> TABLE
;; convert a list of at least two Nats into a scribble table
(define (fib-tab l)
  ;; [Listof Natural] -> Any
  (define (result lst)
    (if (cons? (rest (rest lst))) (third lst) "..."))
  ;;Any -> PARAGRAPH
  (define (b x)
    (make-paragraph (make-style #f '[]) (format "~a" x)))
  ;; -- IN --
  (make-table (make-style 'boxed '())
              (cons (map b (list "n" "n+1" "n+2"))
                    (let loop ([l l])
                         (if (empty? (rest l))
                             '()
                             (cons (map b (list (first l) (second l) (result l)))
                                   (loop (rest l))))))))

(require scribble/core)

(provide fib-tab)
```

How the modules hang together



How the modules hang together



You need to recall the "types" you had in mind originally.

lib-support.ss

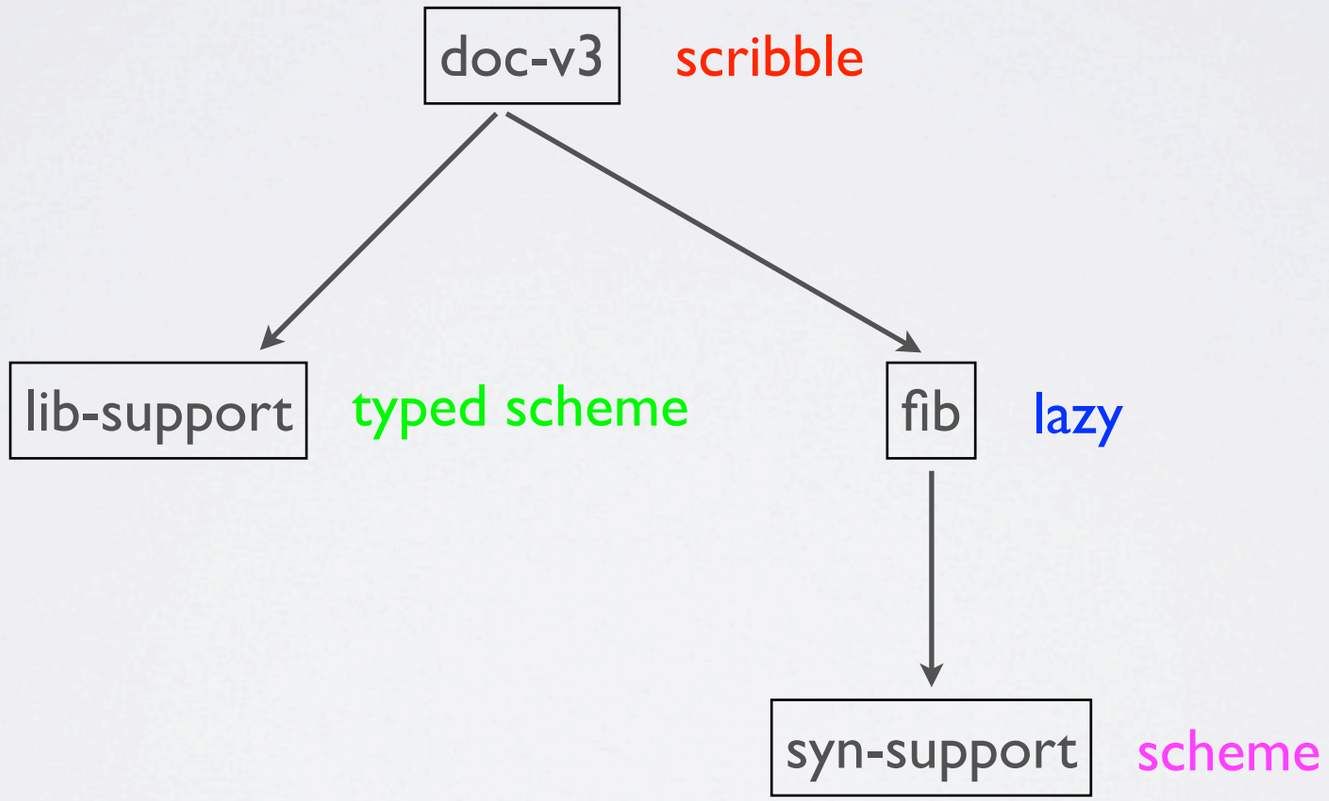
```
#lang scheme

;; (cons Natural (cons Natural [Listof Natural])) -> TABLE
;; convert a list of at least two Nats into a scribble table
(define (fib-tab l)
  ;; [Listof Natural] -> Any
  (define (result lst)
    (if (cons? (rest (rest lst))) (third lst) "..."))
  ;; Any -> PARAGRAPH
  (define (b x)
    (make-paragraph (make-style #f '[]) (format "~a" x)))
  ;; -- IN --
  (make-table
         (make-style 'boxed '())
         (cons (map b (list "n" "n+1" "n+2"))
               (let loop ([l l])
                    (if (empty? (rest l))
                        '()
                        (cons (map b (list (first l) (second l) (result l)))
                              (loop (rest l))))))))

(require scribble/core)

(provide fib-tab)
```

You might as well make them explicit and checkable.

lib-support.ss

```
#lang typed/scheme

(: fib-tab ((cons Natural (cons Natural [Listof Natural])) -> table))
;; convert a list of at least two Nats into a scribble table
(define (fib-tab l)
  (: result ([Listof Natural] -> Any))
  (define (result lst)
    (if (cons? (rest (rest lst))) (third lst) "..."))
  (: b (Any -> paragraph))
  (define (b x)
    (make-paragraph (make-style #f '[]) (format "~a" x)))
  ;; -- IN --
  (make-table (make-style 'boxed '())
              (cons (map b (list "n" "n+1" "n+2"))
                (let: loop : [Listof [Listof paragraph]] ([l : [Listof Natural] l])
                  (if (empty? (rest l))
                      '()
                      (cons (map b (list (first l) (second l) (result l)))
                            (loop (rest l))))))))

(require/typed
 scribble/core
 [struct style ([name : (U False Symbol)] [properties : [Listof Any]])]
 [struct paragraph ([style : style] [content : String])]
 [struct table ([style : style] [blockss : [Listof [Listof paragraph]]])])
(provide fib-tab)
```

How the modules hang together, still, even with types added.



Two ideas worth studying

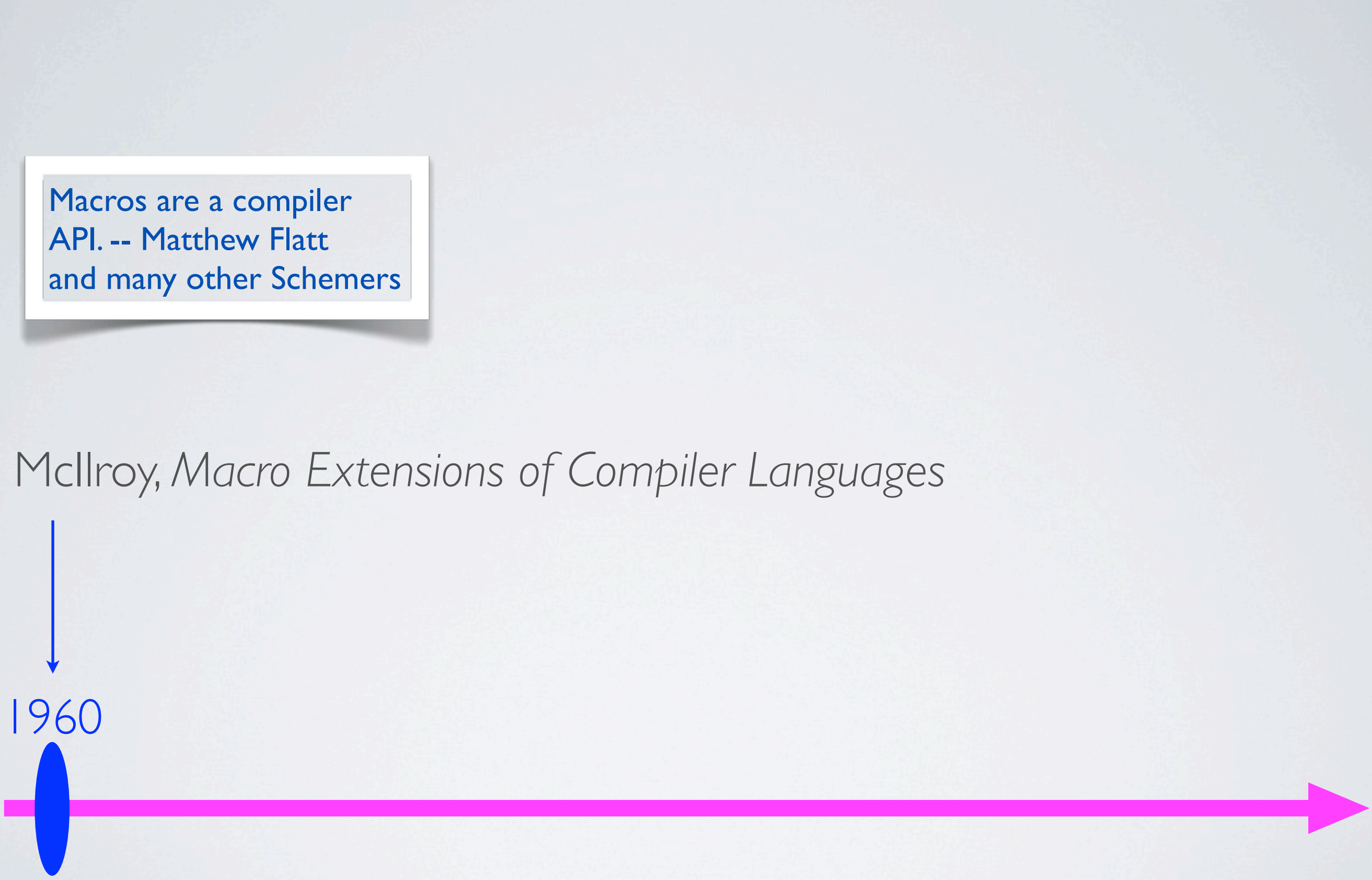
- how to implement languages like this with macros (Lisp macros vs PLT Scheme macros)

- how to add types and protect the integrity of multi-lingual modules (typed vs untyped)

MACROS MATTER

Macros are a compiler API. -- Matthew Flatt and many other Schemers

McIlroy, *Macro Extensions of Compiler Languages*



Macros are a compiler API. -- Matthew Flatt and many other Schemers

McIlroy, *Macro Extensions of Compiler Languages*

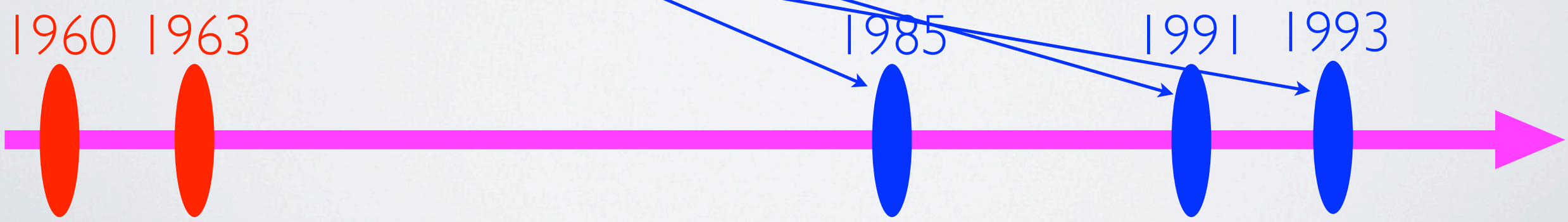
Hart, *Macro Definitions for LISP*

> Macros ought to be abstractions, that is definition and use should be independent.

Kohlbecker, Friedman, Felleisen, Duba *Hygienic Macros*

Clinger & Rees *Macros that Work*

Dybvig, Bruggeman, Hieb *Syntactic Abstractions*



Languages are modules and have two parts: syntax and run-time.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))
```

syntax

run-time library

```
(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

Languages are modules and have two parts: syntax and run-time.

lib-support.ss

syntax

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))
```

run-time library

```
(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

Languages are modules and have two parts: syntax and run-time.

lib-support.ss

syntax

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     #:do ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))
```

run-time library

```
(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

Languages are modules and have two parts: syntax and run-time.

lib-support.ss

syntax

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))
```

run-time library

```
(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(close-by (posn 0 0) (posn 3 4))
```

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(close-by (posn 0 0) (posn 3 4))
```

expand

```
(let ([delta (posn- (posn 0 0) (posn 3 4))])
  (if (<= (posn-distance0 delta) EPSILON)
      (posn 0 0)
      (posn 3 4)))
```

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(close-by (posn 0 0) (posn 3 4))
```

expand

```
(let ([delta (posn- (posn 0 0) (posn 3 4))])
  (if (<= (posn-distance0 delta) EPSILON)
      (posn 0 0)
      (posn 3 4)))
```

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(define (f x)
  (define (posn- p)
    (posn (- (posn-x p)) (- x (posn-y p))))
  ;; -- IN --
  (close-by (posn 0 0) (posn 12 5)))
```

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(define (f x)
  (define (posn- p)
    (posn (- (posn-x p)) (- x (posn-y p))))
  ;; -- IN --
  (close-by (posn 0 0) (posn 12 5)))
```

Macro bodies are substituted for macro calls. Which *posn-* is meant?

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(define (f x)
  (define (posn- p)
    (posn (- (posn-x p)) (- x (posn-y p))))
  ;; -- IN --
  (close-by (posn 0 0) (posn 12 5)))
```

Macro bodies are substituted for macro calls. Which *posn-* is meant?

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
   [(_ p q)
    ;; ==>
    #'(let ([delta (posn- p q)])
        (if (<= (posn-distance0 delta) EPSILON)
            p
            q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(define delta
   (posn- (posn 12 6) (posn 0 1)))


(close-by delta origin)
```

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

```
(define delta
  (posn- (posn 12 6) (posn 0 1)))


(close-by delta origin)
```

Macros arguments are substituted into the macro body. Which *delta* is meant?

Hygiene ensures that two different substitutions work.

lib-support.ss

```
#lang scheme

(define-syntax (close-by stx)
  (syntax-case stx ()
    [(_ p q)
     ;; ==>
     #'(let ([delta (posn- p q)])
         (if (<= (posn-distance0 delta) EPSILON)
             p
             q))]))

(define EPSILON 1)

(struct posn (x y) #:prefab)

(define (posn- p q)
  (posn (- (posn-x p) (posn-x q)) (- (posn-y p) (posn-y q))))

(define (posn-distance0 p)
  (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
```

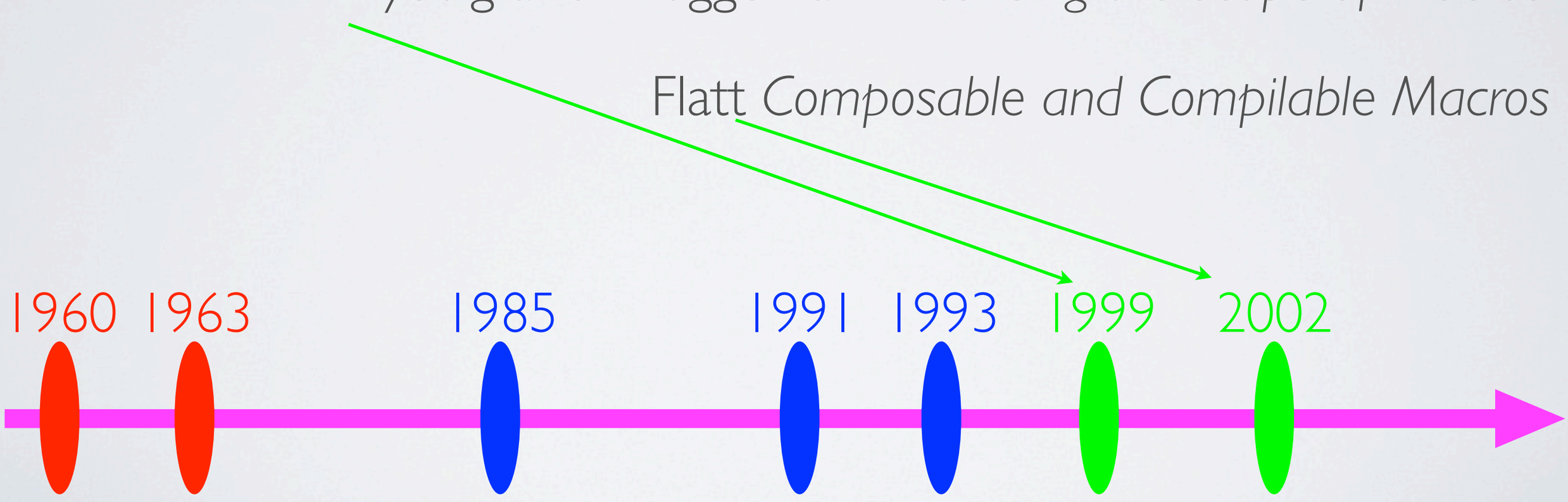
```
(define delta
  (posn- (posn 12 6) (posn 0 1)))

(close-by delta origin)
```

Macros arguments are substituted into the macro body. Which *delta* is meant?

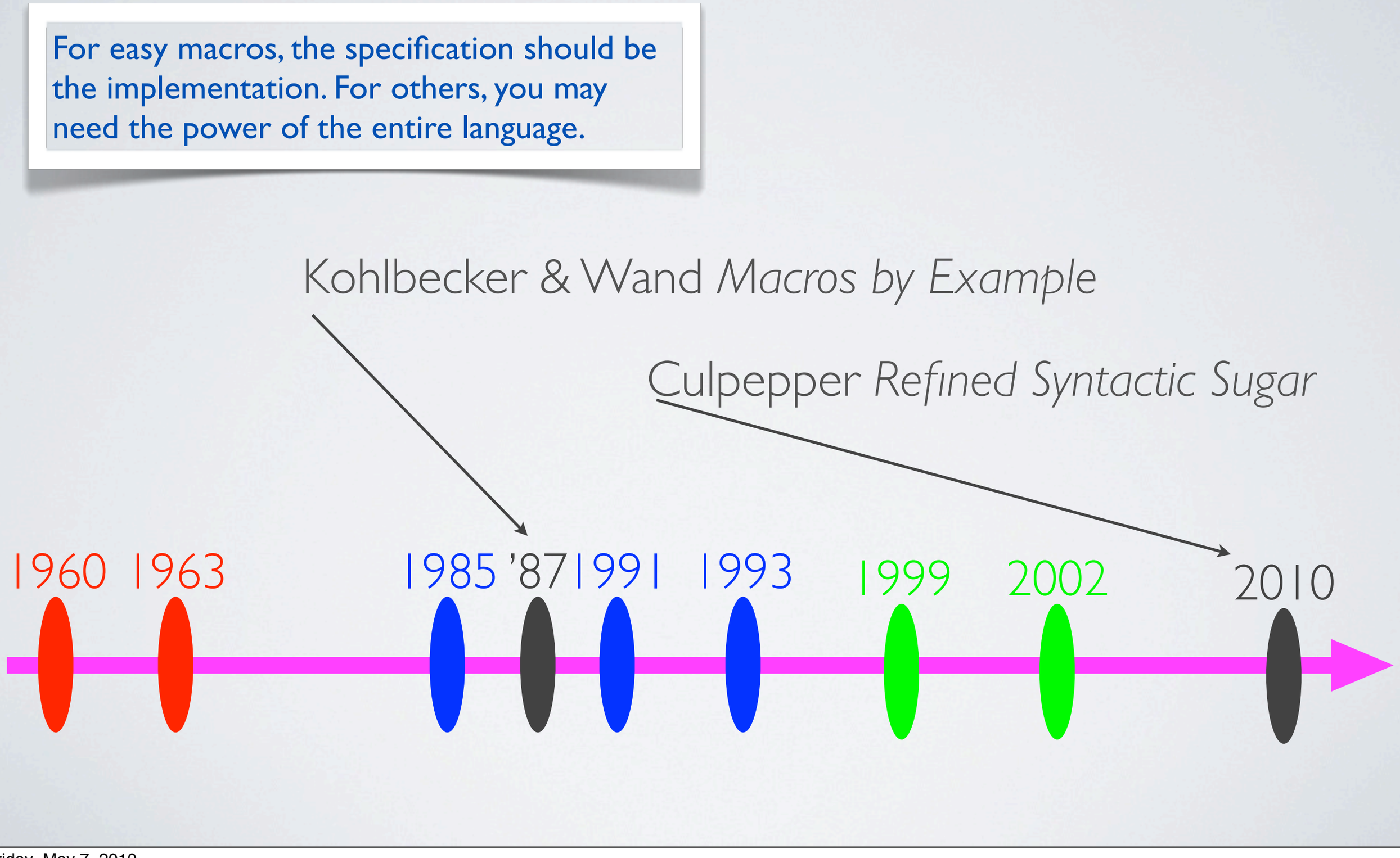
Modules and macros must interact properly. Macro expansion must live in its own separate phase and enable separate compilation.

Dybvig and Bruggeman *Extending the Scope of Macros*

Flatt *Composable and Compilable Macros*



For easy macros, the specification should be the implementation. For others, you may need the power of the entire language.



65, Lisp

```
(define-macro (let . stx)
   (define decls (second stx))
   (define binds (map first decls))
   (define vals   (map second decls))
   (define body (rest (rest stx))
   ;; -- IN --
   `((lambda ,@binds . body) ,@vals)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

65, Lisp

```
(define-macro (let . stx)
   (define decls (second stx))
   (define binds (map first decls))
   (define vals   (map second decls))
   (define body (rest (rest stx))
   ;; -- IN --
   `((lambda ,@binds . body) ,@vals)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

87, Scheme

```
(define-syntax-rule
   (let ((lhs rhs) ... ) body ...)
   ;; ==>
   ((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

65, Lisp

```
(define-macro (let . stx)
   (define decls (second stx))
   (define binds (map first decls))
   (define vals   (map second decls))
   (define body (rest (rest stx))
   ;; -- IN --
   `((lambda ,@binds . body) ,@vals)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

87, Scheme

```
(define-syntax-rule
   (let ((lhs rhs) ... ) body ...)
   ;; ==>
   ((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

94, R5RS

```
(define-syntax let
  (syntax-rules ()
   (___ ((lhs rhs) ... ) body ...)
   ;; ==>
   ((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

95, Chez

```
(define-syntax (let stx)
  (syntax-case stx ()
   (___ ((lhs rhs) ... ) body ...)
   ;; ==>
   #'((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

95, Chez

```
(define-syntax (let stx)
  (syntax-case stx ()
   (___ ((lhs rhs) ... ) body ...)
   ;; ==>
   #'((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

10, PLT

```
(define-syntax (let stx)
  (syntax-parse stx
   (___ ((lhs:identifier rhs:expr) ... )
      body:expr ...)
   ;; ==>
   #'((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

95, Chez

```
(define-syntax (let stx)
  (syntax-case stx ()
   (___ ((lhs rhs) ... ) body ...)
   ;; ==>
   #'((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

10, PLT

```
(define-syntax (let stx)
  (syntax-parse stx
   (___ ((lhs:identifier rhs:expr) ... )
      body:expr ...)
   ;; ==>
   #'((lambda (lhs ...) body ...) rhs ....)
```

```
(let ((x 8) (y 5))
   (+ x y))
```

```
(let ((x 8 'BAD) (y 5))
   (+ x y))
```

```
(let ((17 x) (y 5))
   (+ x y))
```

automatic synthesis of
syntactic checks
and error messages

And a true compiler API comes with a lot more power ...

- automatic source tracking
- lifting definitions to module top-level
- local expansion
- module-body expansion

Macro-implemented features are indistinguishable from *native* features now.

- classes (two systems: Java-like, CLOS-like)
- keyword args, C ffi, first-class components
- scribble (plus reader)
- and of course Typed Scheme.

MEANINGFUL TYPES FOR LISPERS

Goal: adding types to systems written in Scheme on a module-by-module basis

Ungoal: Types are optional compiler hints.

Goal: types are **explicit** and **checked**; the system is sound

Goal: adding types should demand as few changes to the code as possible.

Goal: these types are solid maintenance information for all programmers, including versions of our future selves.

But, adding these kinds of types is non-trivial.

problem 1: what is *type soundness* in a system of *typed* and *untyped* Scheme modules? what do *types* mean?

But, adding these kinds of types is non-trivial.

problem 1: what is *type soundness* in a system of *typed* and *untyped* Scheme modules? what do *types* mean?

problem 2: programmers should not need to modify code to accommodate the type checker but vice versa. how can a *type system accommodate* Scheme?

MEANINGFUL TYPES

...

T

```
#lang scheme
;; inc5: Integer -> Integer
(define (inc5 i) (+ i 5))
(printf "~a\n" (inc -3))

(provide inc5)
```

U

```
#lang scheme

(require T)

(printf "~a\n" (inc5 6))
```

T

```
#lang scheme
;; inc5: Integer -> Integer
(define (inc5 i) (+ i 5))
(printf "~a\n" (inc -3))

(provide inc5)
```

U

```
#lang scheme

(require T)

(printf "~a\n" (inc5 6))
```

T

```
#lang typed/scheme
(: inc5 (Integer -> Integer))
(define (inc5 i) (+ i 5))
(printf "~a\n" (inc -3))

(provide inc5)
```

U

#**lang** scheme

(require **T**)

... (inc5 true) ...

T

#**lang** typed/scheme

(: inc5 (Integer -> Integer))
(define (inc5 i) (+ i 5))

(provide inc5)

U

#**lang** scheme

(require **T**)

... (inc5 true) ...

T

#**lang** typed/scheme

(: inc5 (Integer -> Integer))
(define (inc5 i) (+ i 5))

(provide inc5)

U

#**lang** scheme

(require **T**)

... (inc5 true) ...

T

#**lang** typed/scheme

(: inc5 (Integer -> Integer))
(define (inc5 i) (+ i 5))

(provide inc5)

bang!

T

```
#lang typed/scheme



(define (encode f)
   (+ (f 21) 42))

(provide encode)
```

U

```
#lang scheme

(require T)

(define (hello i)
   (format "~a: hello world" i))

(printf "~a\n" (encode hello))
```

T

```
#lang typed/scheme

(: encode ((Integer -> Integer) ->  Integer))
(define (encode f)
   (+ (f 21) 42))

(provide encode)
```

U

```
#lang scheme

(require T)

(define (hello i)
   (format "~a: hello world" i))

(printf "~a\n" (encode hello))
```

T

```
#lang typed/scheme

(: encode ((Integer -> Integer) ->  Integer))
(define (encode f)
   (+ (f 21) 42))

(provide encode)
```

U

```
#lang scheme

(require T)

(define (hello i)
   (format "~a: hello world" i))

(printf "~a\n" (encode hello))
```

T

```
#lang typed/scheme

(: encode ((Integer -> Integer) -> Integer))

(define (encode f)
  (+ (f 21) 42))

(provide encode)
```

bang!

U

```
#lang scheme

(require T)

(define (hello i)
  (format "~a: hello world" i))

(printf "~a\n" (encode hello))
```

U

```
#lang scheme

(define (inc5 x) (+ x 5))

(provide inc)
```

T

```
#lang typed/scheme

(require U)

(define (f x)
   ... (inc5 ???) ??? 10 ... )
```

U

```
#lang scheme

(define (inc5 x) (+ x 5))

(provide inc)
```

T

```
#lang typed/scheme

(require U)

(define (f x)
   ... (inc5 ???) ??? 10 ... )
```

T

```
#lang typed/scheme

(require/typed U
    (f (Integer -> Integer)))

(define (f x)
   ... (inc5 ???) ... 10 ...)
```

U

```
#lang scheme

(define (inc5 x)
  (if (= 42 x)
      "hello world"
      (+ x 5)))

(provide inc)
```

T

```
#lang typed/scheme

(require/typed U
   (f (Integer -> Integer)))

(define (f x)
   (+ (inc5 42) 0))
```

U

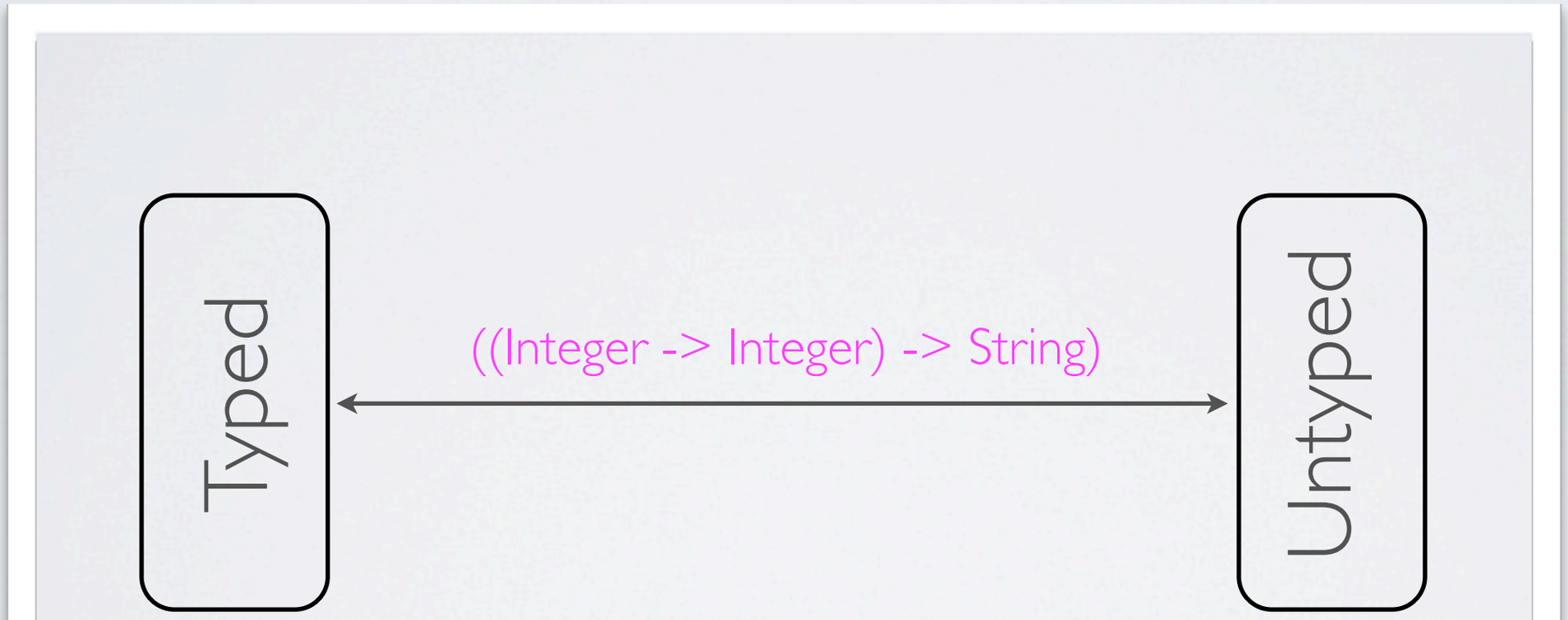
```
#lang scheme

(define (inc5 x)
  (if (= 42 x)
      "hello world"
      (+ x 5)))

(provide inc)
```

T

```
#lang typed/scheme

(require/typed U
   (f (Integer -> Integer)))

(define (f x)
   (+ (inc5 42) 0))
```

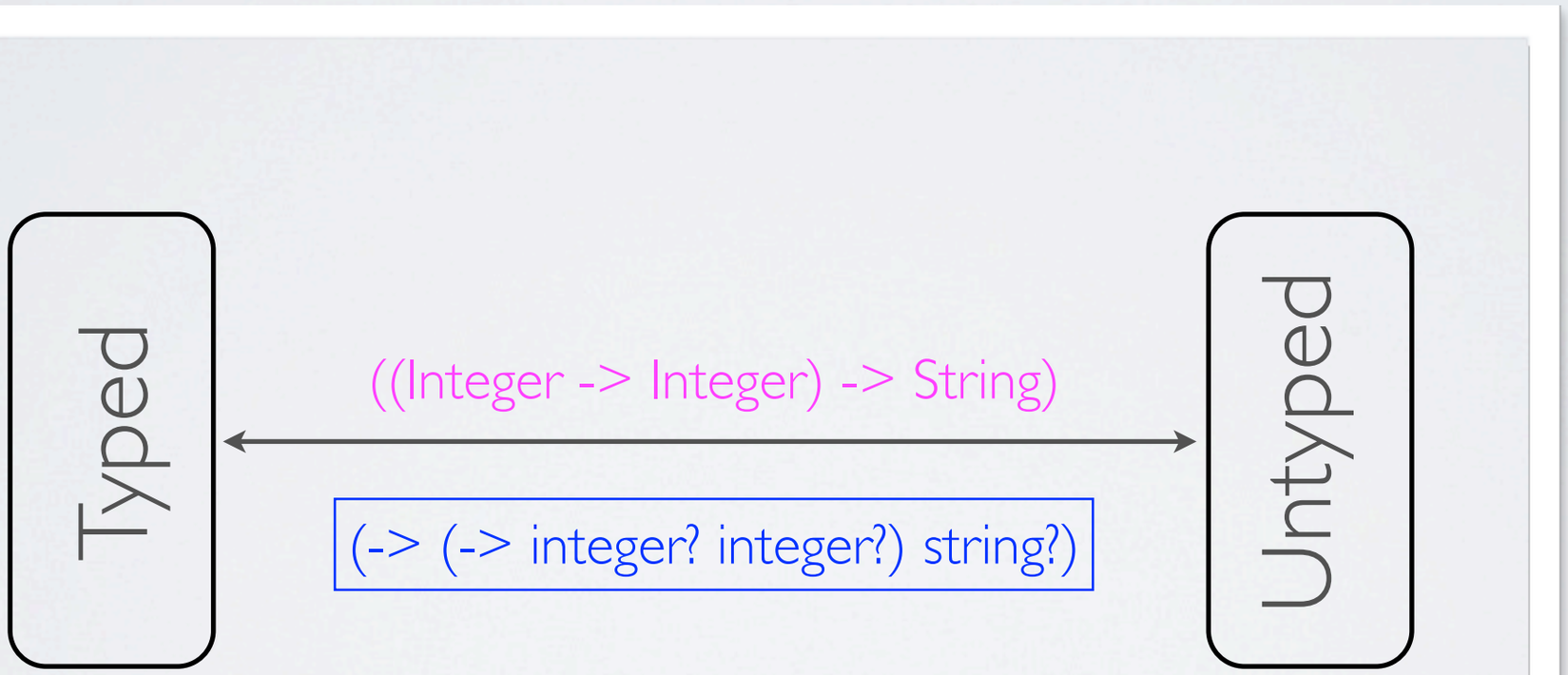
bang!

step 1: typed ‘modules’ must specify types for all imported variables and specify types for all exports



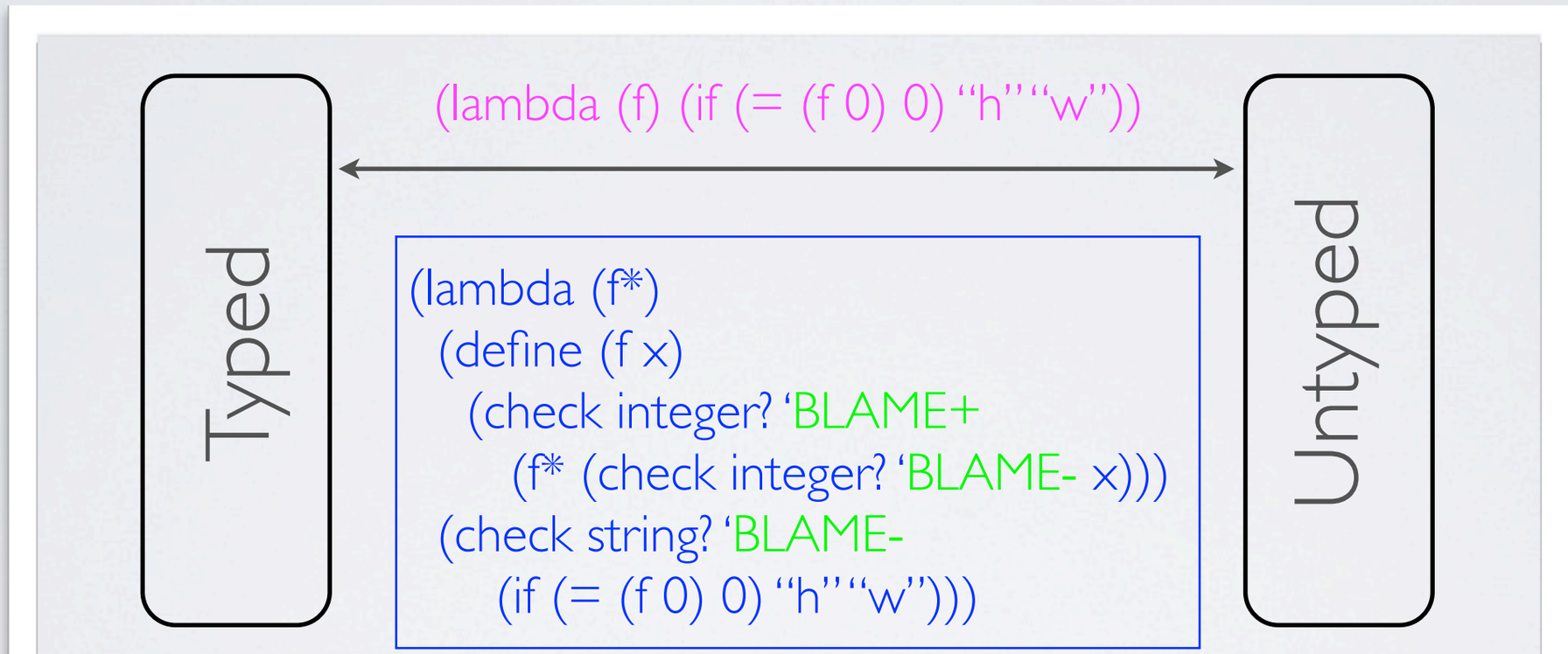
step 1: typed ‘modules’ must specify types for all imported variables and specify types for all exports

step 2: type checking converts ‘interface types’ into contracts that ‘blame’ the violating module --- when needed only!



at “contract boundaries” contracts are attached to a value

when functions are applied, contracts are checked



Theorem: Let P be a mixed program with checked types in interfaces interpreted as contracts. Then

- P yields to a value,
- P diverges, or
- P signals an error *that blames a specific untyped module*.

Theorem: Let P be a mixed program with checked types in interfaces interpreted as contracts. Then

- P yields to a value,
- P diverges, or
- P signals an error *that blames a specific untyped module*.

the "Blame Theorem"
Tobin-Hochstadt & Felleisen (2006)

...TYPES FOR LISPERS

remember: adding types is good;
changing code is bad

remember: adding types is good;
changing code is bad

adding types to structure fields and type declarations for functions is acceptable

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)

;; Circle → Number


(check-within
  (circle-area (make-circle 1)) pi .1)

(define (circle-area c)
  (* pi (circle-radius c) (circle-radius c)))
```

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)

;; Circle → Number

(check-within
  (circle-area (make-circle 1)) pi .1)

(define (circle-area c)
  (* pi (circle-radius c) (circle-radius c)))
```

```
#lang typed/scheme

(define-struct: circle ({radius : Number}))


(: circle-area (circle → Number))

(check-within
   (circle-area (make-circle 1)) pi .1)

(define (circle-area c)
  (* pi (circle-radius c) (circle-radius c)))
```

Scheme demands different types for different occurrences of parameters

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)
...

(define-struct square (length))
;; Square = (make-square Number)
...

;; Shape is one of:
;; -- Circle
;; -- Square
;; ...

;; Shape → Number
(define (shape-area s)
  (cond
    [(circle? s) (circle-area s)]
    [(square? s) (square-area s)]))
```

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)
...

(define-struct square (length))
;; Square = (make-square Number)
...

;; Shape is one of:
;; -- Circle
;; -- Square
;; ...

;; Shape → Number
(define (shape-area s)
  (cond
    [(circle? s) (circle-area s)] ;; programmers knows s : Circle
    [(square? s) (square-area s)]))
```

```
#lang typed/scheme

(define-struct: circle ({radius : Number}))
...
...

(define-struct: square ({length : Number}))
...
...

(define-type-alias shape
  (U circle
       square
       ...))

(: shape-area (shape → Number))
(define (shape-area s)
  (cond
    [(circle? s) (circle-area s)] ;; and so does our type system !
    [(square? s) (square-area s)]))
```

Scheme demands different types for different occurrences of parameters

...

...

and that must hold for **higher-order fragments** too

```
#lang scheme
...
;; (Listof shape) → Number
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0
         (map square-area
              (filter square? l))))
```

```
#lang scheme
...
;; (Listof shape) → Number
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0
         (map square-area  ;; programmer knows: (listof square)
              (filter square? l))))
```

```
#lang scheme
...
;; (Listof shape) → Number
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0 ;; programmer also knows: (listof number)
         (map square-area  ;; programmer knows: (listof square)
              (filter square? l))))
```

```
#lang typed-scheme
...
(: sum-squares ((Listof shape) → Number))
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0
         (map square-area  ;; and so does our type system
              (filter square? l))))
```

"occurrence typing" is also necessary for paths into data structures

```
#lang scheme

;; Atom is either Number or false

;; [Listof Atom] → Number
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))]
    [else (+ (first l) (sum (rest l)))]))
```

```
#lang scheme

;; Atom is either Number or false

;; [Listof Atom] → Number
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))] ;; programmer knows: (first l) = #f
    [else (+ (first l) (sum (rest l)))]))
```

```
#lang scheme

;; Atom is either Number or false

;; [Listof Atom] → Number
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))]
    [else (+ (first l) (sum (rest l)))])) ; programmer: (first l) : Number
```

```
#lang typed-scheme

(define-type-alias Atom (U Number #f))

(: sum ([Listof Atom] → Number))
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))]
    [else (+ (first l) (sum (rest l)))])) ; and so does our type system
```

the type system needs some simple propositional reasoning

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else ;; programmers knows (first l) : Number, (first k) : Number
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
(define-type-alias Atom (U Number #f))

(: mrg ([Listof Atom] [Listof Atom] → [Listof Number]))
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else ;; as does our type system
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

And a true "Lisp" type system comes with a lot more.

- abstraction over predicates
- explicit, first-class polymorphism
- *local* type inference to reduce annotation cost
- PLT Scheme-specific constructs (CL keywords, arity polymorhism, etc)

CONCLUSION

From PLT Scheme to LISP:

- from macros to languages
- types for Lispers

From PLT Scheme to LISP:

Macros are More than Functions on S-expressions.

- scope and macros
- modules and macros
- phases and macros
- specification and macros

From PLT Scheme to LISP:

Types are Promising.

- types for Lisp idioms
- sound types from contracts

THE END

PLT Scheme *is* Racket.
http://racket-lang.org/

Matthew Flatt (Utah)	language, compiler, macros
Shriram Krishnamurthi	macros and modules
Robby Findler (Northwestern)	contracts, IDE
Jay McCarthy (BYU)	macro-languages & web services
Ryan Culpepper (Utah)	macros
Sam Tobin-Hochstadt (Northeastern)	types
Stevie Strickland (Northeastern)	contracts & types

```
(define-type-alias Atom (U Number #f))

(: naughty ((Pair Atom Any) → Boolean : #f @ car))
(define (naughty l) (not (first l)))

(: mrg ([Listof Atom] [Listof Atom] → [Listof Number]))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(naughty? l)
     (cons (first k) (mrg (rest l) (rest k)))]
    [(naughty? k)
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```