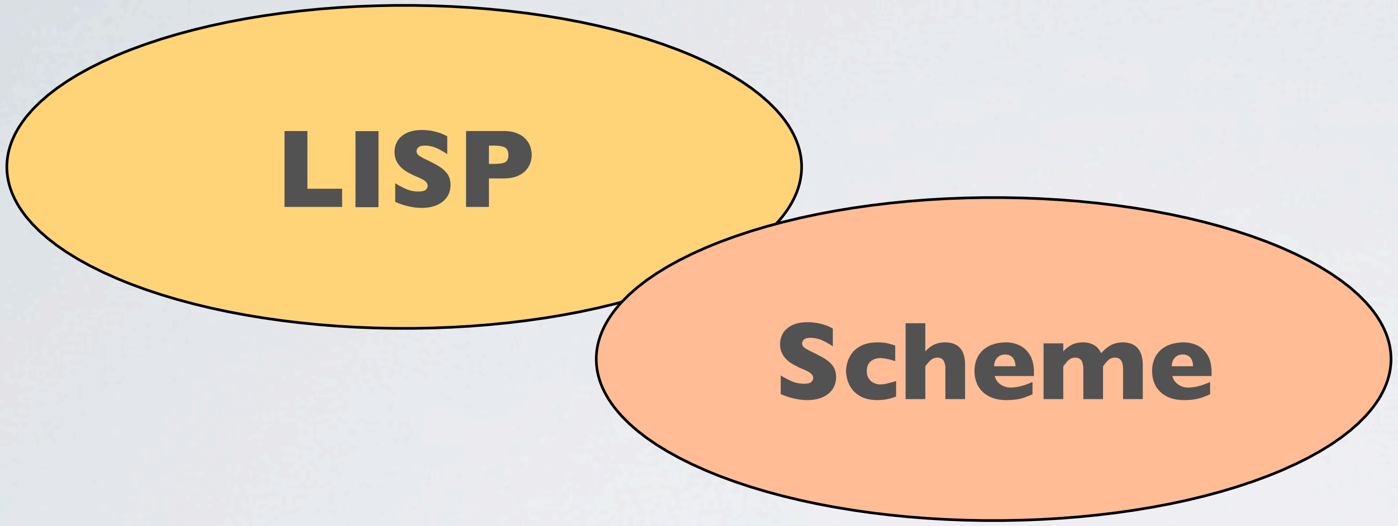
MULTILINGUAL COMPONENT PROGRAMMING IN RACKET

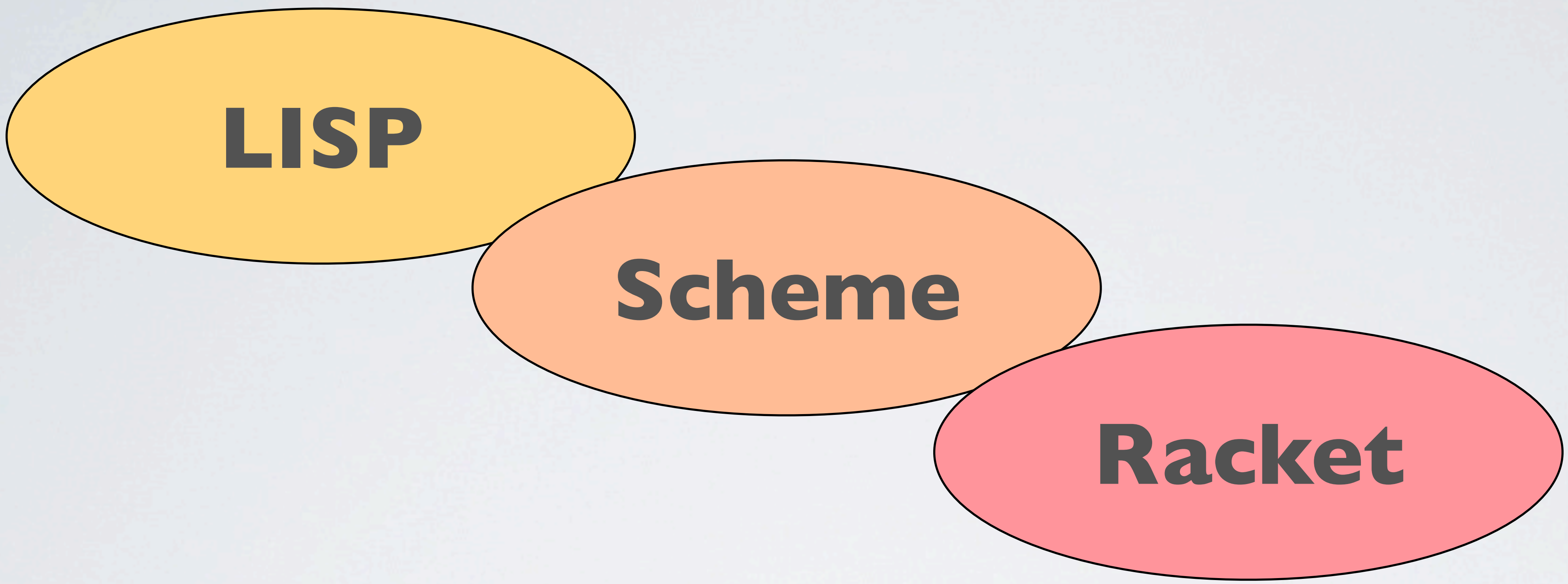
Matthias Felleisen (PLT)

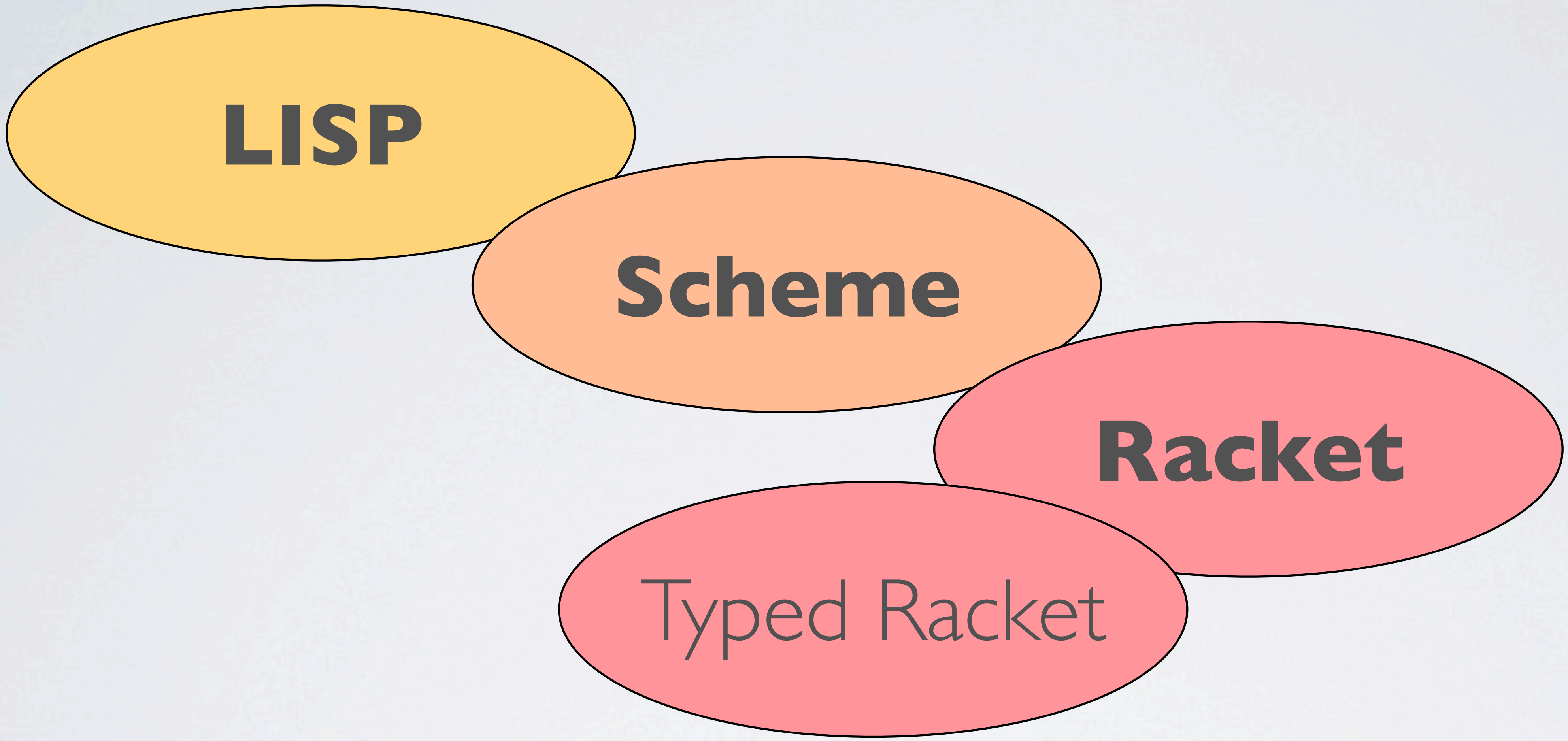
I am a salesman,
and I will sell you Racket.

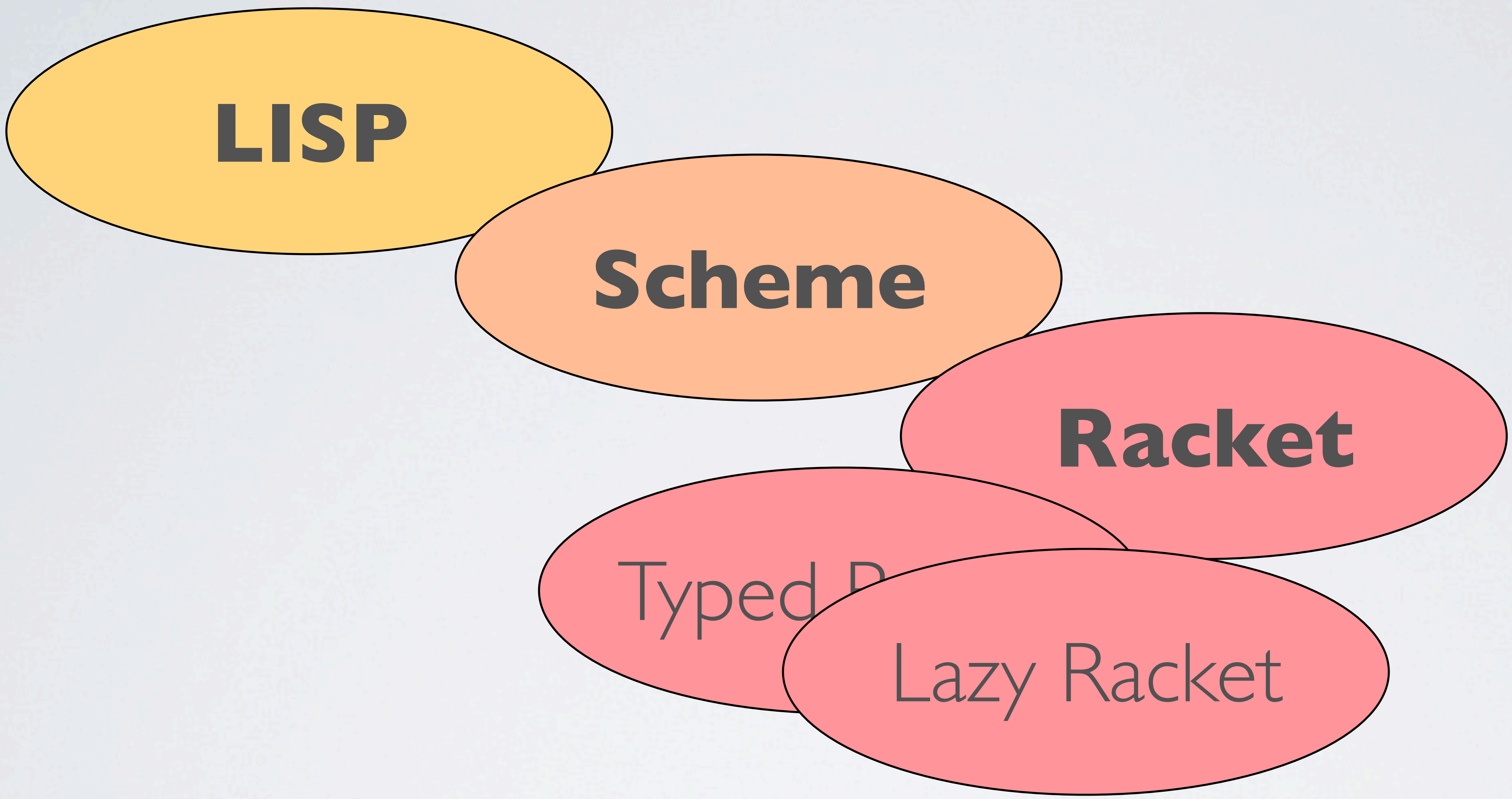
LISP

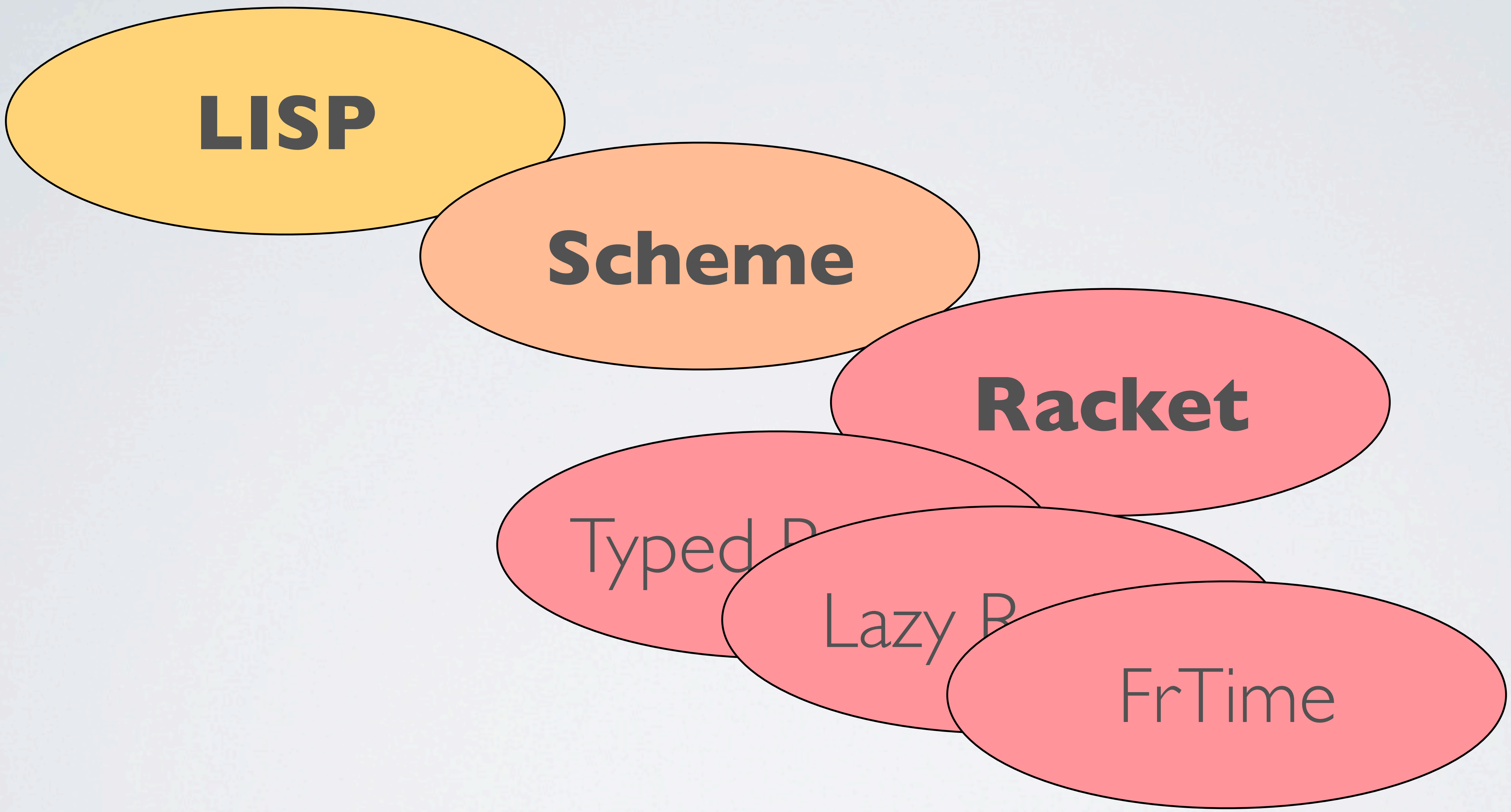


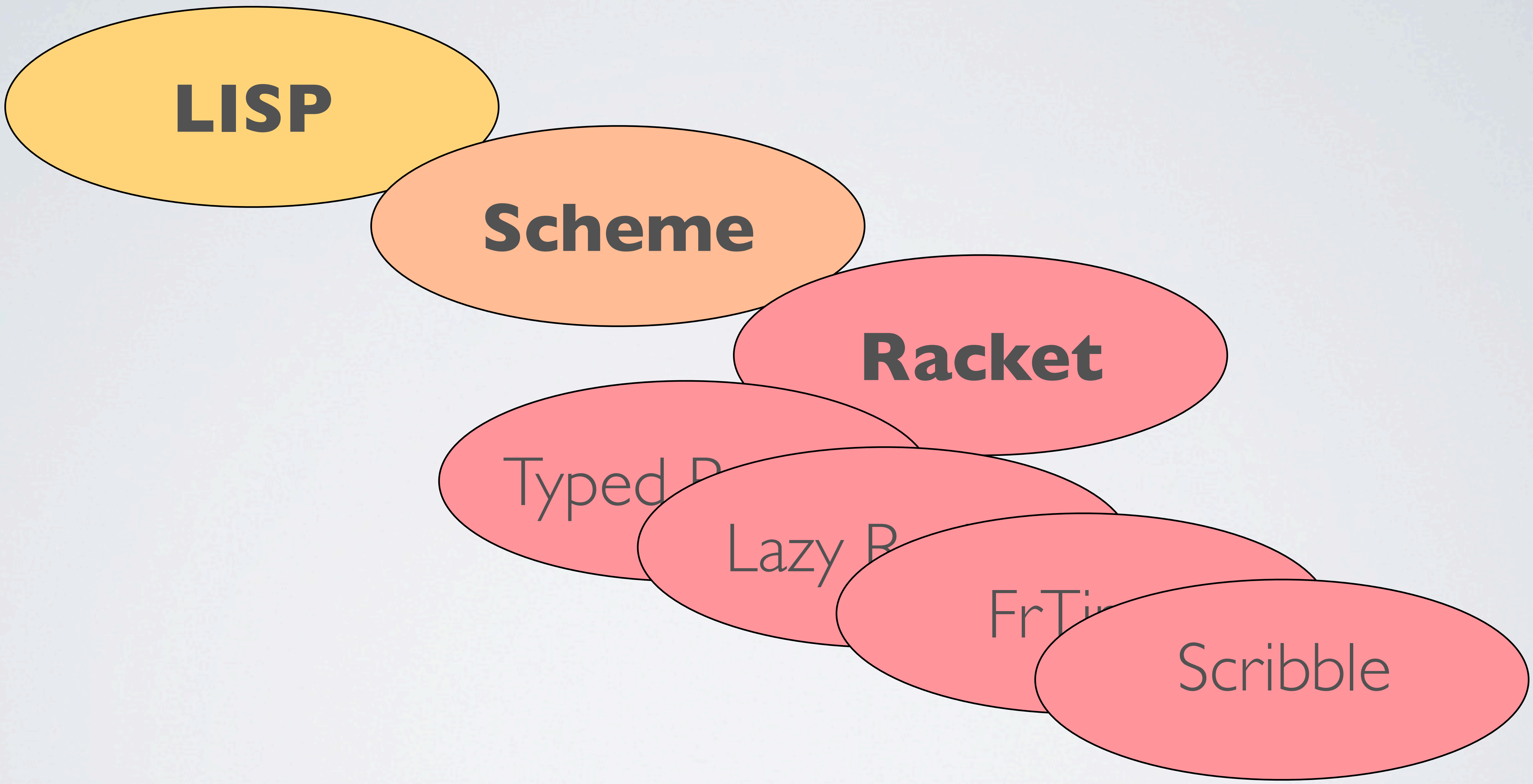
Saturday, October 22, 2011

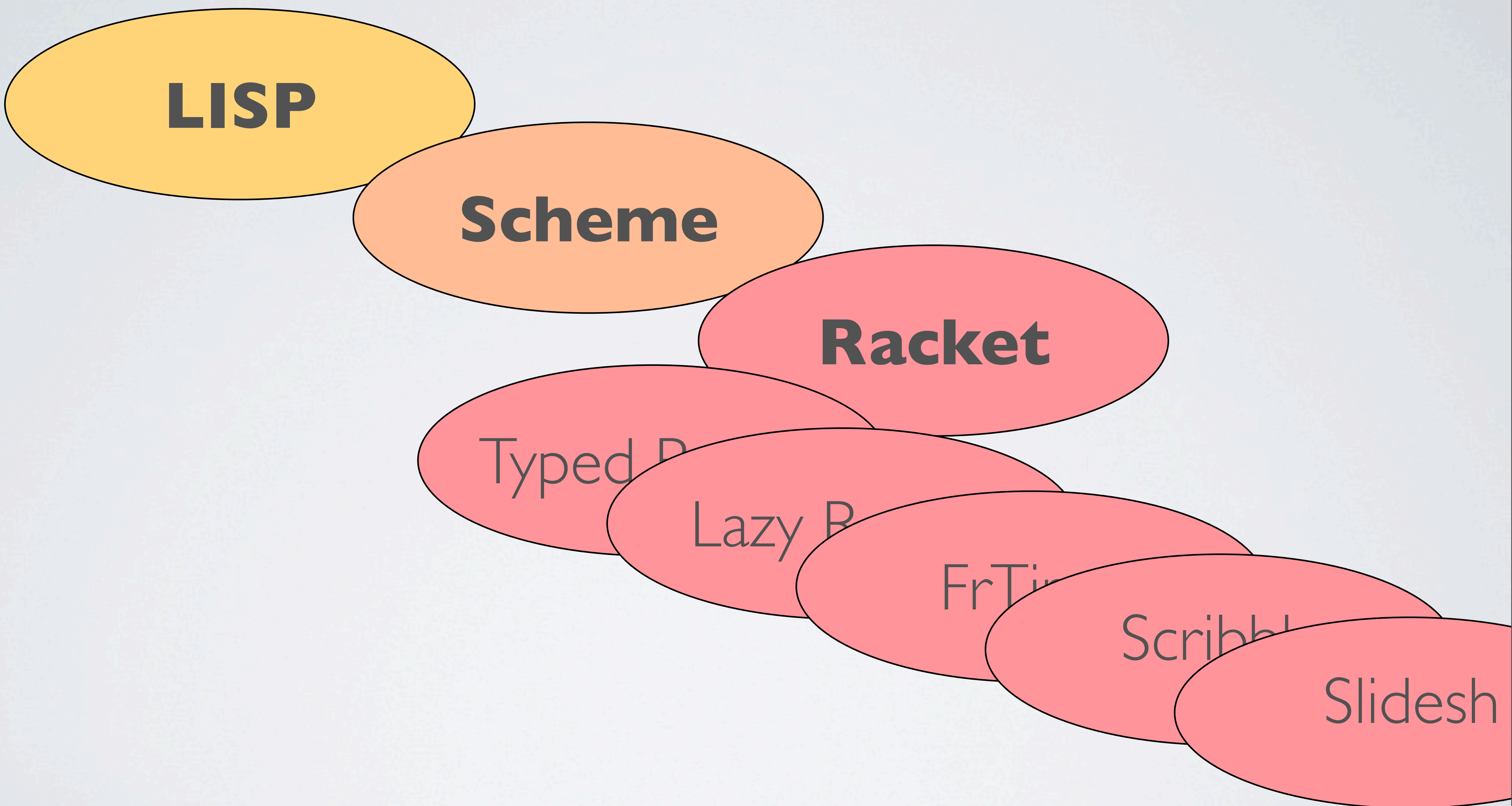

LISP
Scheme
Racket

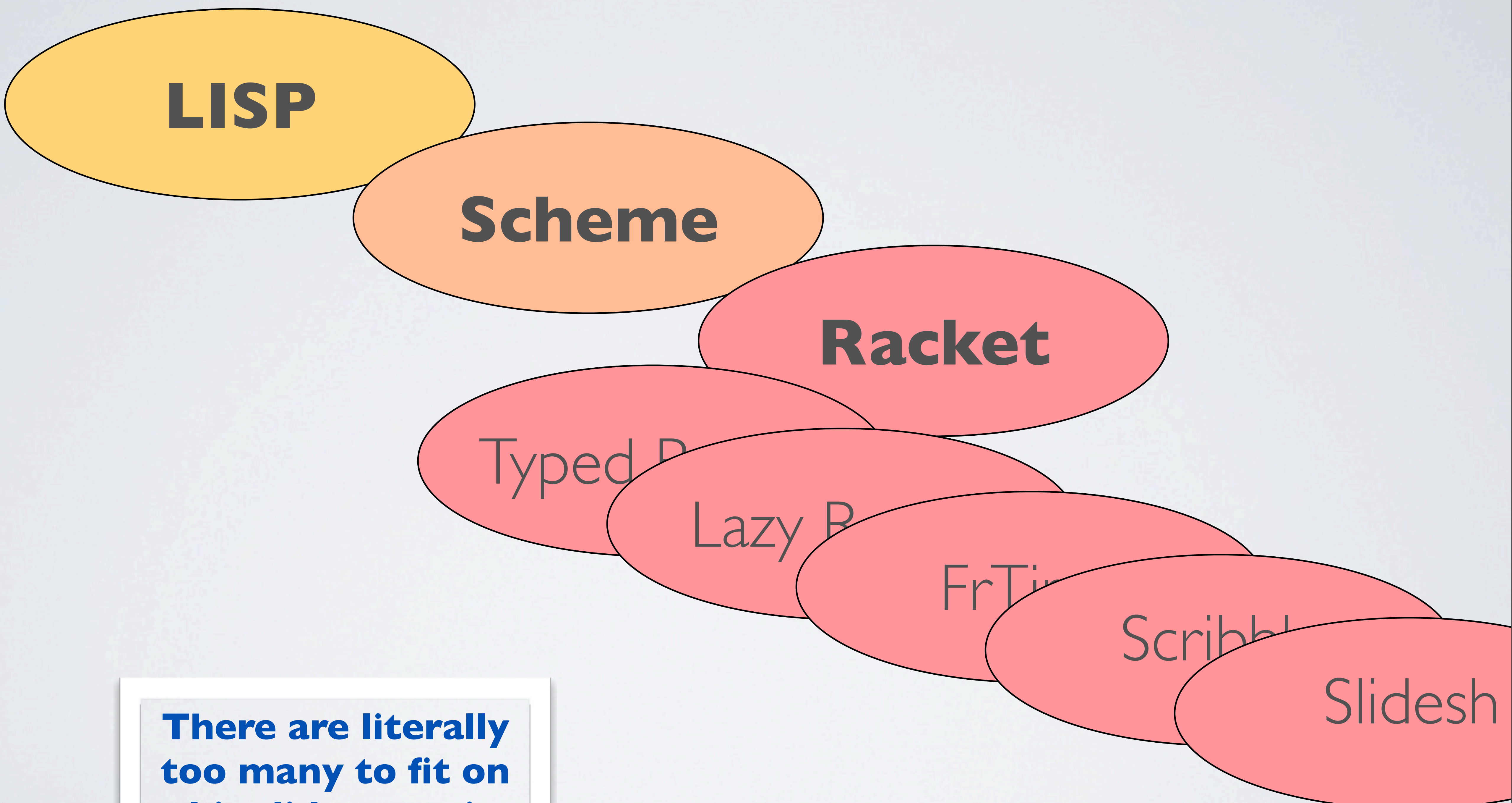

LISP
Scheme
Racket
Typed Racket


LISP
Scheme
Racket
Typed R
Lazy Racket


LISP
Scheme
Racket
Typed R
Lazy R
FrTime


LISP
Scheme
Racket
Scribble


LISP
Scheme
Racket
Typed R
Lazy R
FrT
Scribb
Slidesh



There are literally too many to fit on this slide, margin or body.

The Racket language

- higher-order functions
- classes and objects
- cross-platform GUIs
- extensive libraries
- rich web programming

The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The Racket language
- higher-order functions
- classes and objects
- cross-platform GUIs
- extensive libraries
- rich web programming

The *Typed* Racket language

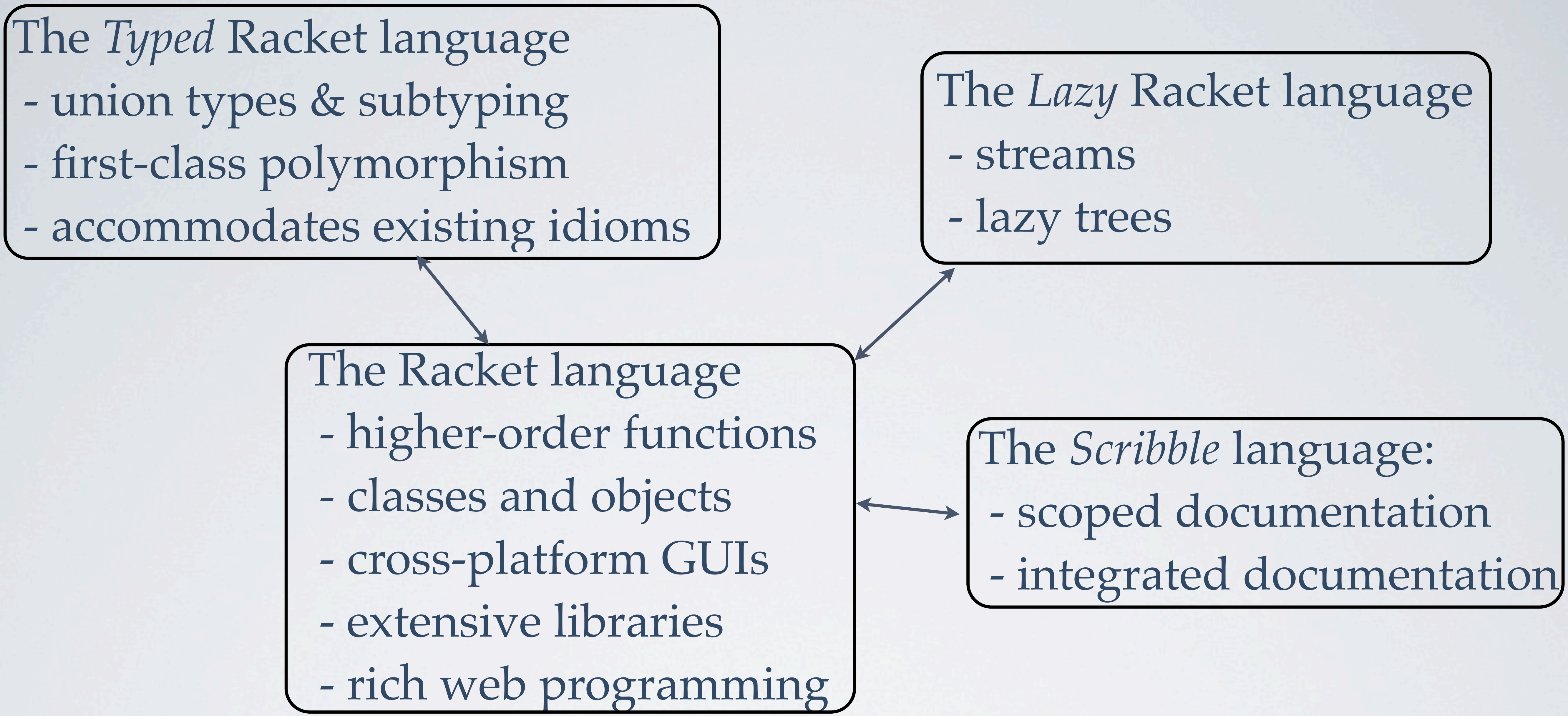
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

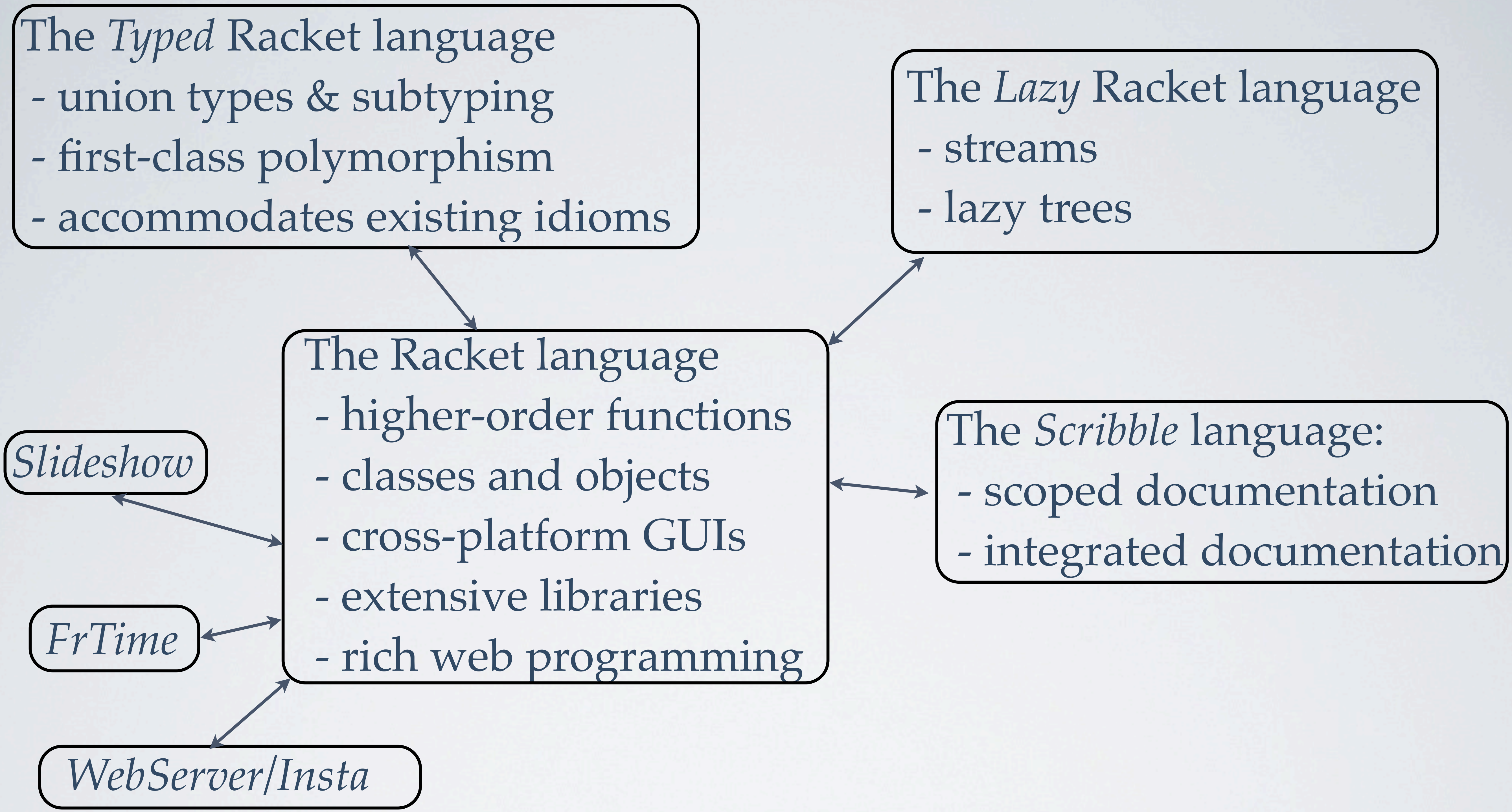
The *Lazy* Racket language

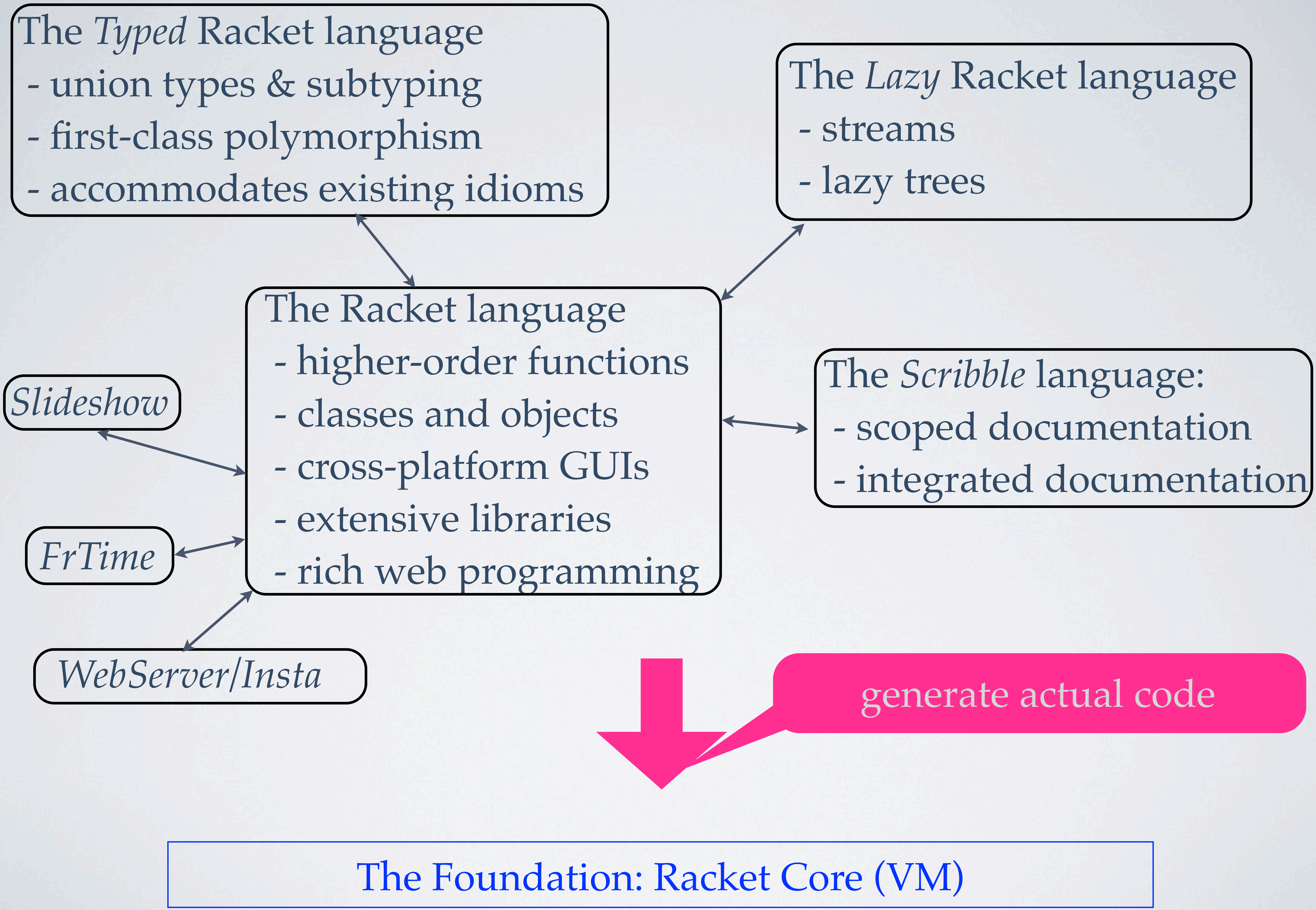
- streams
- lazy trees

The Racket language

- higher-order functions
- classes and objects
- cross-platform GUIs
- extensive libraries
- rich web programming


The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms
The *Lazy* Racket language
- streams
- lazy trees
The Racket language
- higher-order functions
- classes and objects
- cross-platform GUIs
- extensive libraries
- rich web programming
The *Scribble* language:
- scoped documentation
- integrated documentation


The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms
The *Lazy* Racket language
- streams
- lazy trees
The Racket language
- higher-order functions
- classes and objects
- cross-platform GUIs
- extensive libraries
- rich web programming
The *Scribble* language:
- scoped documentation
- integrated documentation
Slideshow
FrTime
WebServer/Insta


The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms
The *Lazy* Racket language
- streams
- lazy trees
The Racket language
- higher-order functions
- classes and objects
- cross-platform GUIs
- extensive libraries
- rich web programming
Slideshow
The *Scribble* language:
- scoped documentation
- integrated documentation
FrTime
WebServer/Insta
generate actual code
The Foundation: Racket Core (VM)

Racket

```
#lang racket

;; (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)
```

Lazy

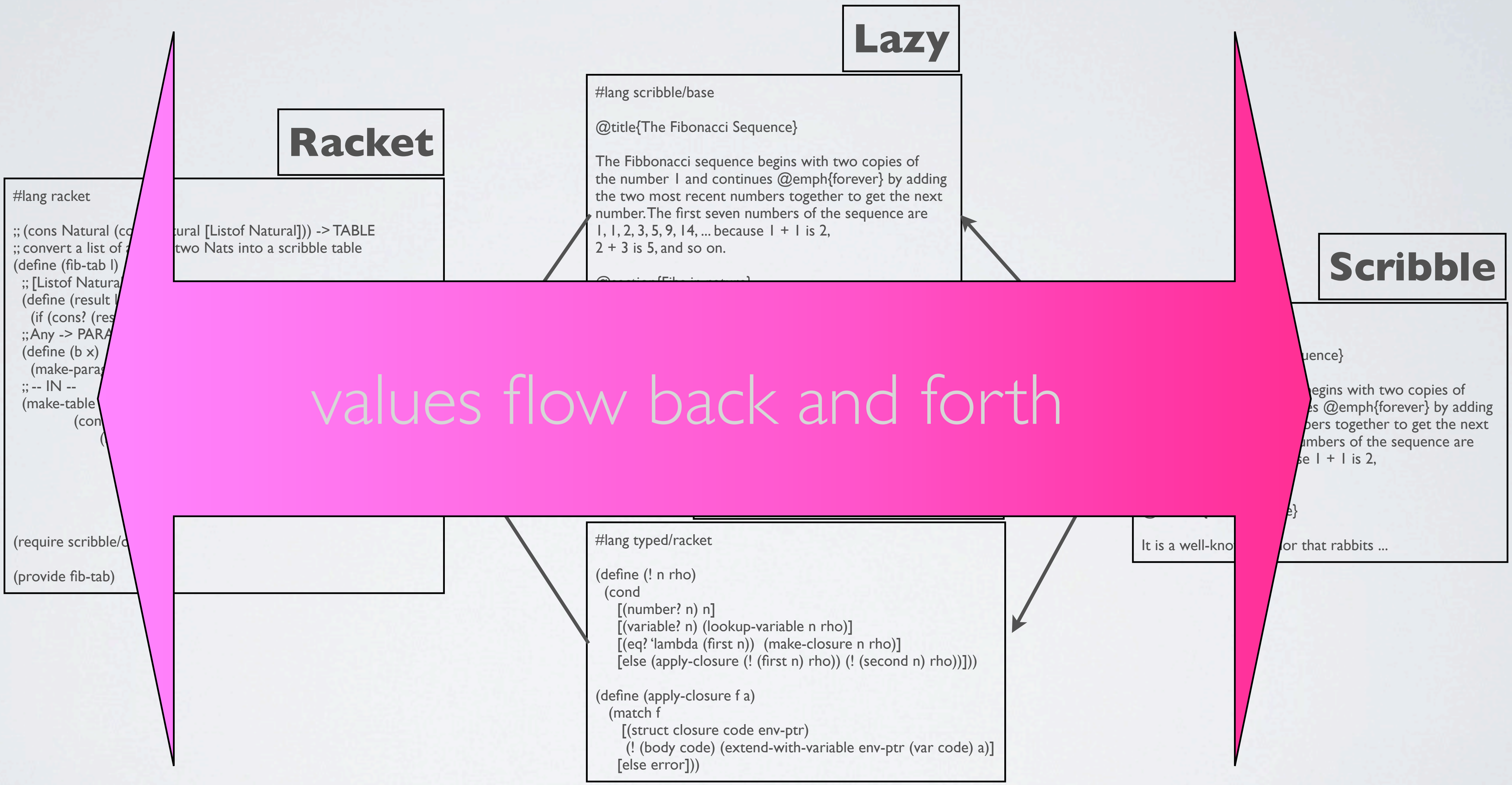
```
#lang scribble/base

@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 ...
```

Scribble

```
#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 ...
```

Typed Racket

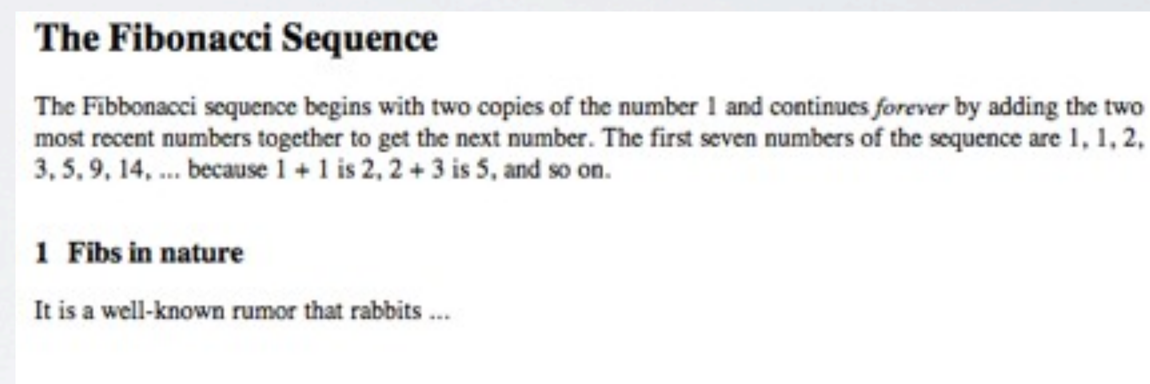
```
#lang typed/racket

(define (! n rho)
  (cond
    [(number? n) n]
    [(variable? n) (lookup-variable n rho)]
    [(eq? 'lambda (first n))  (make-closure n rho)]
    [else (apply-closure (! (first n) rho)) (! (second n) rho))]))

(define (apply-closure f a)
  (match f
    [(struct closure code env-ptr)
     (! (body code) (extend-with-variable env-ptr (var code) a)]
    [else error]))
```


Lazy
Racket
Scribble
values flow back and forth
#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.
#lang racket
;; (cons Natural (co
tural [Listof Natural])) -> TABLE
;; convert a list of
two Nats into a scribble table
(define (fib-tab l)
(require scribble/d
(provide fib-tab)
#lang typed/racket
(define (! n rho)
(cond
[(number? n) n]
[(variable? n) (lookup-variable n rho)]
[(eq? 'lambda (first n)) (make-closure n rho)]
[else (apply-closure (! (first n) rho)) (! (second n) rho))]))
(define (apply-closure f a)
(match f
[(struct closure code env-ptr)
(! (body code) (extend-with-variable env-ptr (var code) a)]
[else error]))
It is a well-kno
or 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* 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.

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 ...
```

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 ...
```

9

Ouch!

fib.rkt

```
#lang lazy

(require "syn-support.rkt")

;; 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.rkt

```
#lang lazy

(require "syn-support.rkt")

;; 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)
```

etc.rkt

```
#lang racket

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

(provide rec)
```

fib.rkt

```
#lang lazy

(require "syn-support.rkt")

;; 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)
```

etc.rkt

```
#lang racket

(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 (map number->string (!! (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
@(string-join 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-v2.scrbl

```
#lang scribble/base

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

@title{The Fibonacci Sequence}

@(define fib7 (map number->string (!! (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
@(string-join 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 ..
```

We can do better still -- add a table.

We can do better still -- add a table.

doc-v3.scrbl

```
#lang scribble/base

@(require lazy/force "fib.rkt" "tabulate.rkt")

@title{The Fibonacci Sequence}

@(define fib7 (map number->string  (!! (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
@(string-join 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:

@(tabulate 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

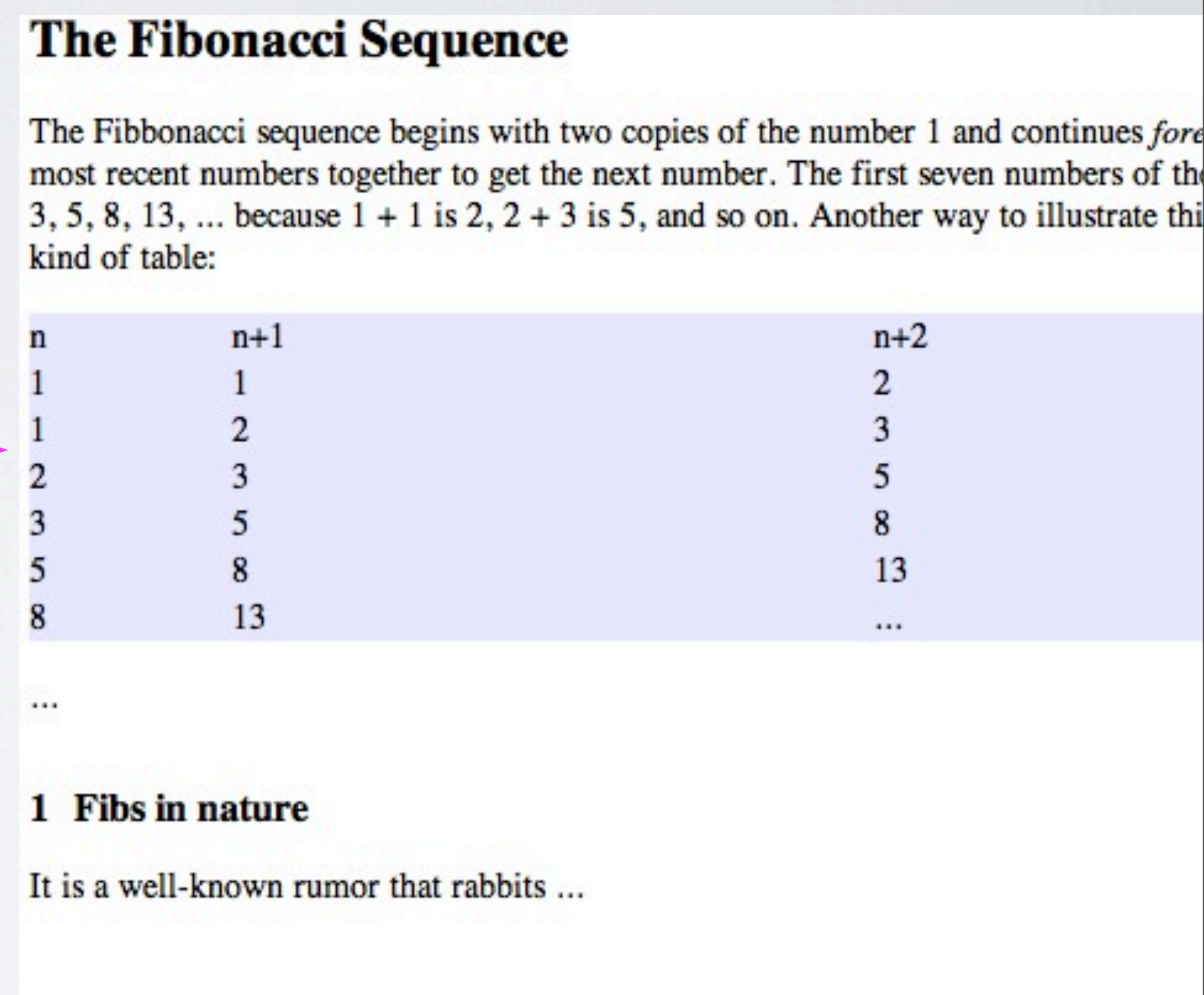
It is a well-known rumor that rabbits ...

We can do better still -- add a table.

doc-v3.scrbl

```
#lang scribble/base

@(require lazy/force "fib.rkt" "tabulate.rkt")

@title{The Fibonacci Sequence}

@(define fib7 (map number->string  (!! (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
@(string-join 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:

@(tabulate 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 the
3, 5, 8, 13, ... because 1 + 1 is 2, 2 + 3 is 5, and so on. Another way to illustrate thi
kind of table:

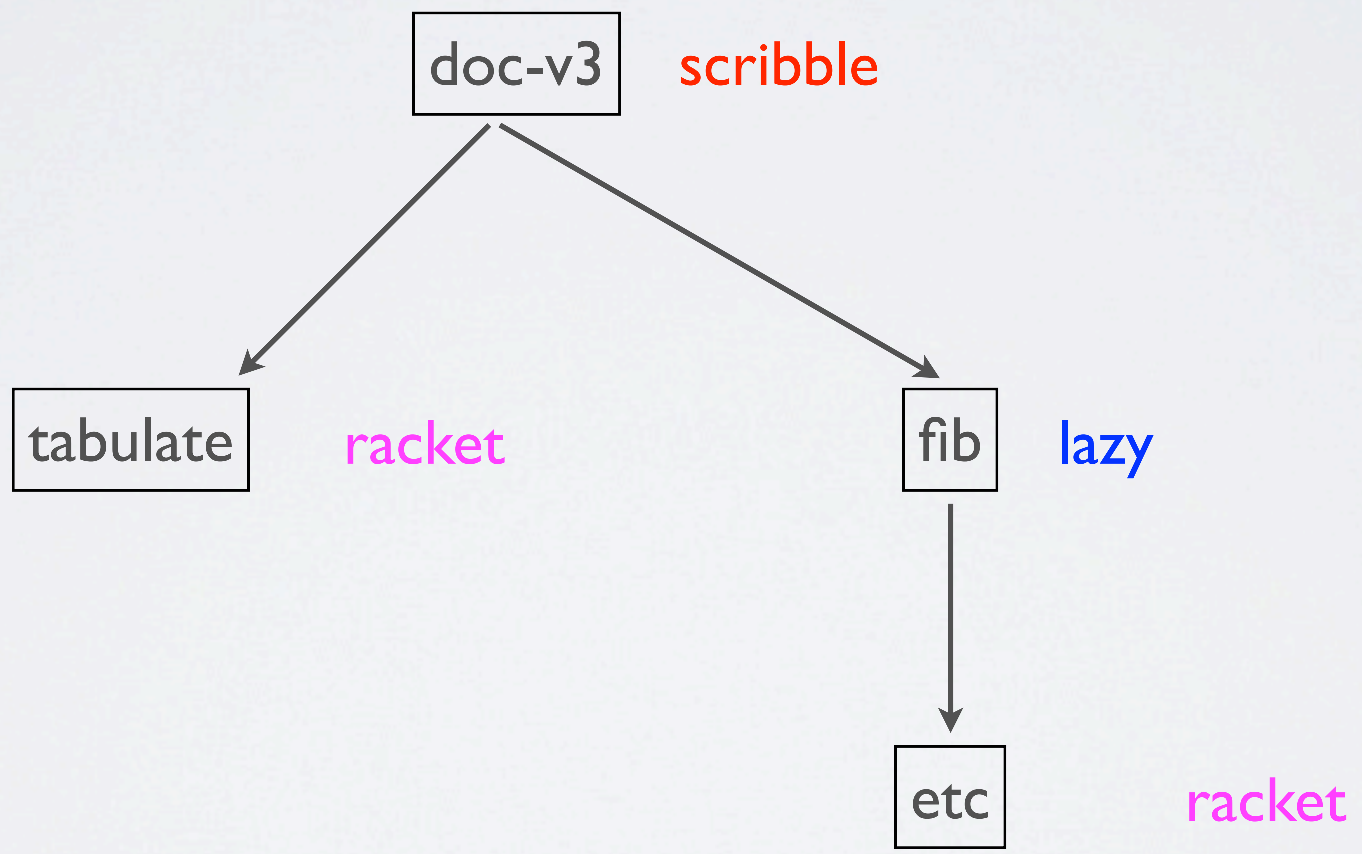
n	n+1	n+2
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13
8	13	...

...

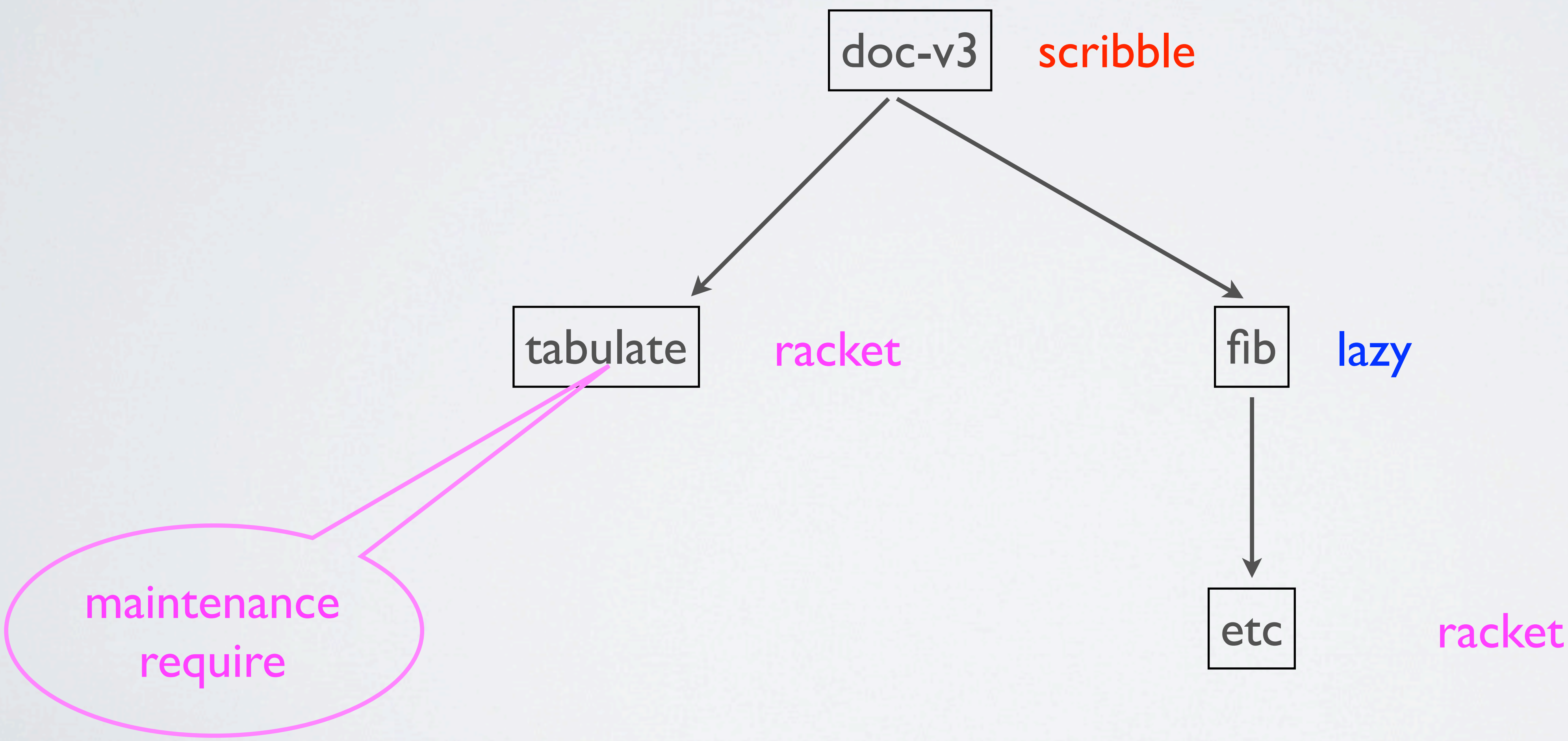
1 Fibs in nature

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

How the modules hang together



How the modules hang together



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

tabulate.rkt

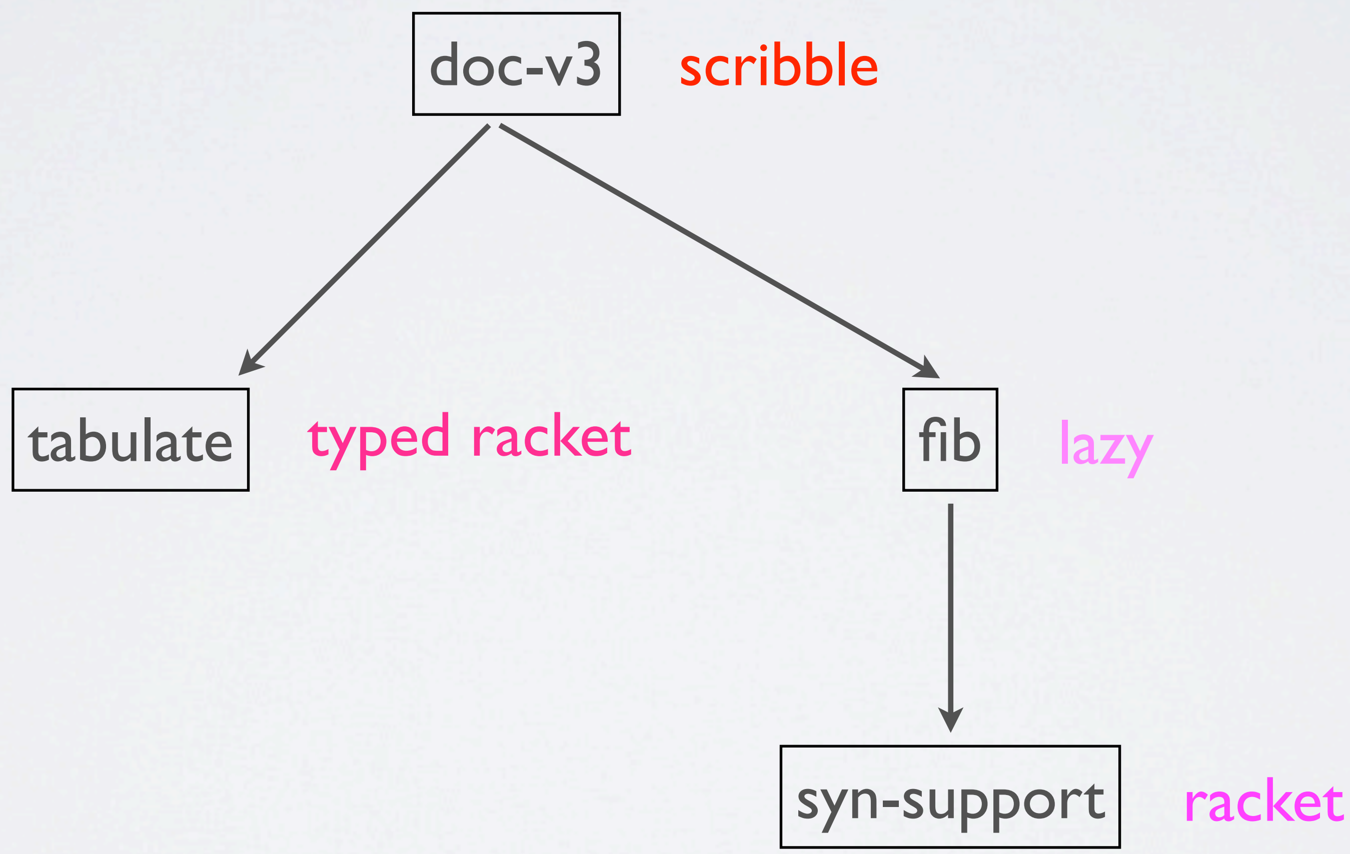
```
#lang racket

;; (cons Natural (cons Natural [Listof Natural])) -> TABLE
;; convert a list of at least two Nats into a scribble table
(define (tabulate 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 tabulate)
```

You might as well make them explicit and checkable.

tabulate.rkt

```
#lang typed/racket

(: fib-tab ((cons Natural (cons Natural [Listof Natural])) -> table))
;; convert a list of at least two Nats into a scribble table
(define (tabulate 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 ([l l])
                      (if (empty? (rest l))
                          '()
                          (cons (map b (list (first l) (second l) (result l)))
                                (loop (rest l))))))))

(require/typed scribble/core (struct style ...) ...)

(provide tabulate)
```

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



Two ideas worth studying

Two ideas worth studying

- generative programming to implement languages
- safe component interaction in a multi-lingual world

SAFE INTERACTIONS

Racket

```
#lang racket

;; (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)
```

FrTime

```
#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 ...
```

How does a *reactive* program safely access Racket's GUI library?

Racket

```
#lang racket

;; (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)
```

Lazy

```
#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 ...
```

Lazy values are promises of plain values.
How do we ensure safe access?

Racket

```
#lang racket

;; (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)
```

Typed Racket

```
#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 ...
```

Typed values are plain values.

But how do you guarantee *type soundness*?

What is type safety in a world of *typed* and *untyped* components?

inc.rkt

```
#lang typed/racket

(: inc5 (Integer -> Integer))
;; increment argument by 5
(define (inc5 i)
   (+ i 5))

(provide inc5)
```

main.rkt

```
#lang racket

(require “inc.rkt”)

(printf  “~a\n” (inc5 6))
```

inc.rkt

```
#lang typed/racket

(: inc5 (Integer -> Integer))
;; increment argument by 5
(define (inc5 i)
   (+ i 5))

(provide inc5)
```

main.rkt

```
#lang racket

(require "inc.rkt")

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

This works because *typed* and *untyped* Racket use the *same* set of values.

inc.rkt

```
#lang typed/racket

(: inc5 (Integer -> Integer))
;; increment argument by 5
(define (inc5 i)
   (+ i 5))

(provide inc5)
```

main.rkt

```
#lang racket

(require "inc.rkt")

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

inc.rkt

```
#lang typed/racket

(: inc5 (Integer -> Integer))
;; increment argument by 5
(define (inc5 i)
   (+ i 5))

(provide inc5)
```

main.rkt

```
#lang racket

(require “inc.rkt”)

(printf  “~a\n” (inc5 true))
```

bang!

W/o care, the **typed** component will be blamed for a type error in the **untyped** module.

inc.rkt

```
#lang typed/racket

(: inc5 (Integer -> Integer))
;; increment argument by 5
(define (inc5 i)
   (+ i 5))

(provide inc5)
```

main.rkt

```
#lang racket

(require “inc.rkt”)

(printf “~a\n” (inc5 true))
```

bang!

Solution: check `Integer` on call

W/o care, the **typed** component will be blamed for a type error in the **untyped** module.

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.rkt

```
#lang racket

(require "encode.rkt")

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

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

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.rkt

```
#lang racket

(require "encode.rkt")

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

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

bang!

The **typed** component will be blamed for a type error in the **untyped** module.

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

bang!

main.rkt

```
#lang racket

(require "encode.rkt")

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

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

Solution: check

`(Integer -> Integer)`

on call

The **typed** component will be blamed for a type error in the **untyped** module.

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.rkt

```
#lang racket

(require "encode.rkt")

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

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

bang!

Solution: check

(Integer -> Integer)

on call

The **typed** component will be blamed for a type error in the **untyped** module.

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.rkt

```
#lang racket

(require "encode.rkt")

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

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

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.rkt

```
#lang racket

(require "encode.rkt")

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

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

Solution 1: wrap contract

`(integer? -> integer?)`

around `code`

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.rkt

```
#lang racket

(require “encode.rkt”)

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

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

Solution 1: wrap contract

`(integer? -> integer?)`

around `code`

Solution 2: contract

`(integer? -> integer?)`

checks each call to `code`

encode.rkt

```
#lang racket


;; encode output of f
(define (encode f)
    (+ (f 21) 42))

(provide encode)
```

main.rkt

```
#lang typed/racket

(require "encode.rkt")

(define (code i)
   (format "~a: hello world" (encode (λ (x) x))))

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

encode.rkt

```
#lang racket


;; encode output of f
(define (encode f)
   (+ (f 21) 42))

(provide encode)
```

main.rkt

```
#lang typed/racket

(require “encode.rkt”)

(define (code i)
  (format "~a: hello world" (encode (λ (x) x))))

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

stop!

The **typed** component needs a type for the **untyped** import for type checking.

encode.rkt

```
#lang racket


;; encode output of f
(define (encode f)
   (+ (f 21) 42))

(provide encode)
```

main.rkt

```
#lang typed/racket

(require/typed "encode.rkt"
  (encode ((Integer -> Integer) -> Integer)))

(define (main i)
   (format "~a: hello world" (encode (λ (x) x))))

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

encode.rkt

```
#lang racket


;; encode output of f
(define (encode f)
    (+ (f 21) 42))

(provide encode)
```

←

main.rkt

```
#lang typed/racket

(require/typed "encode.rkt"
  (encode ((Integer -> Integer) -> Integer)))

(define (main i)
   (format "~a: hello world" (encode (λ (x) x))))

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

Solution 1: state type

`((Integer -> Integer) -> Integer)`

for import `main`

encode.rkt

```
#lang racket

;; encode output of f
(define (encode f)
   (+ (f 21) 42))

(provide encode)
```

main.rkt

```
#lang typed/racket

(require/typed "encode.rkt"
  (encode ((Integer -> Integer) -> Integer)))

(define (main i)
   (format "~a: hello world" (encode (λ (x) x))))

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

Solution 1: state type
`((Integer -> Integer) -> Integer)`
for import `main`

Solution 2: interpret types as contracts

encode.rkt

```
#lang typed/racket

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

(provide encode)
```

main.

```
#lang typed/racket

(require"encode.rkt")

(define (main i)
   (format "~a: hello world"
               (encode (λ (x) x))))

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

encode.rkt

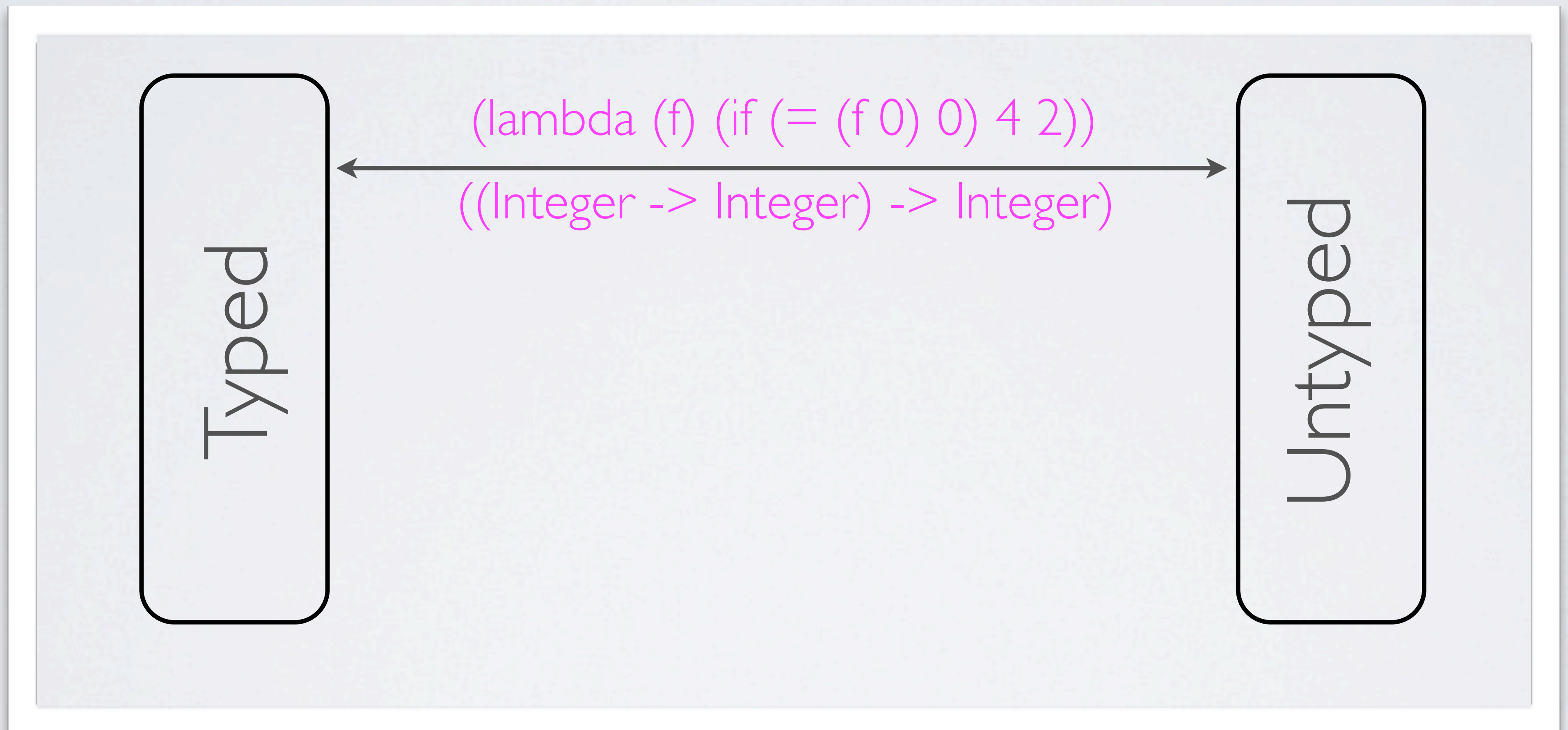
```
#lang typed/racket

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

(provide encode)
```

main.

```
#lang typed/racket

(require"encode.rkt")

(define (main i)
  (format "~a: hello world"
          (encode (λ (x) x))))

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

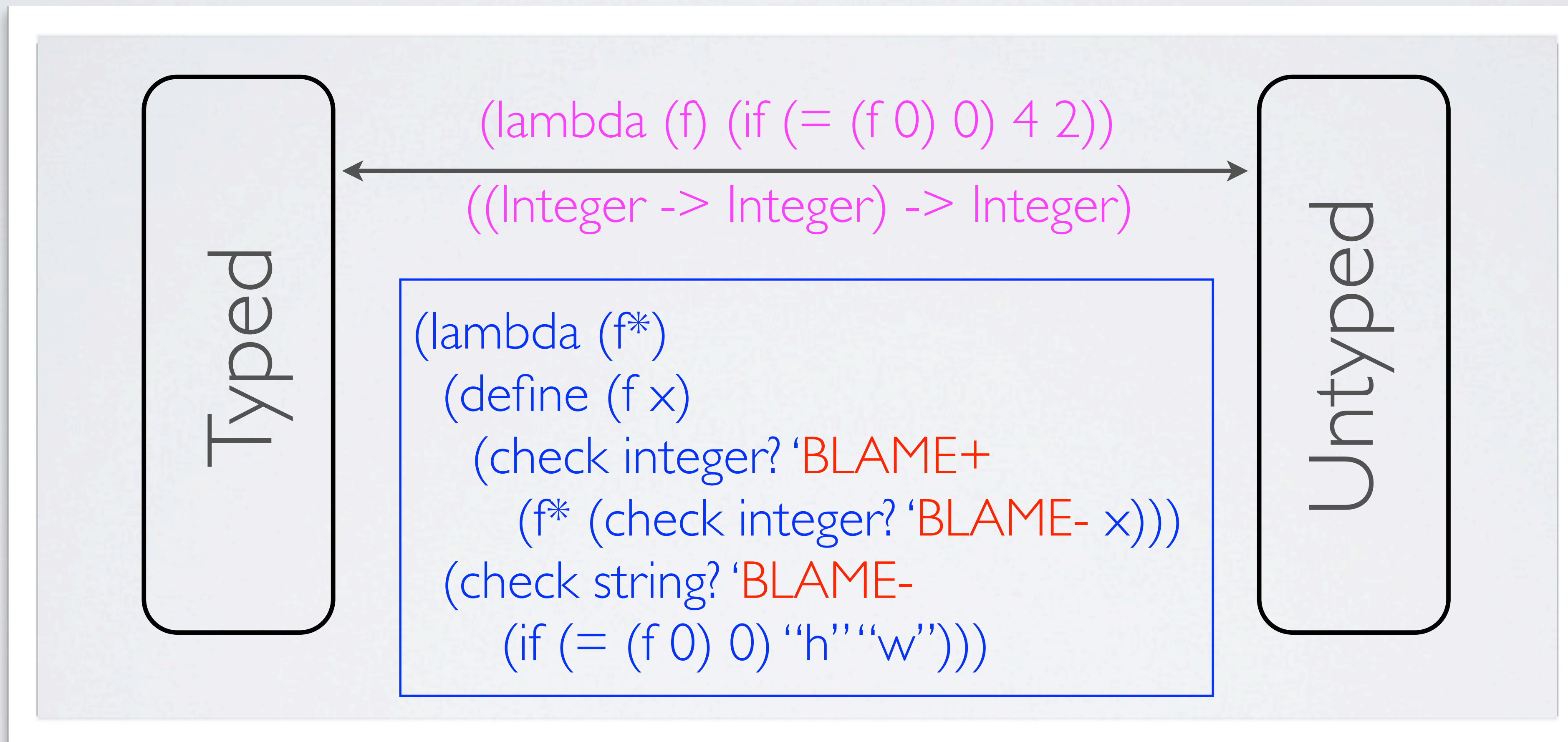
Interactions between components of the **same** kind do not need controls.

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



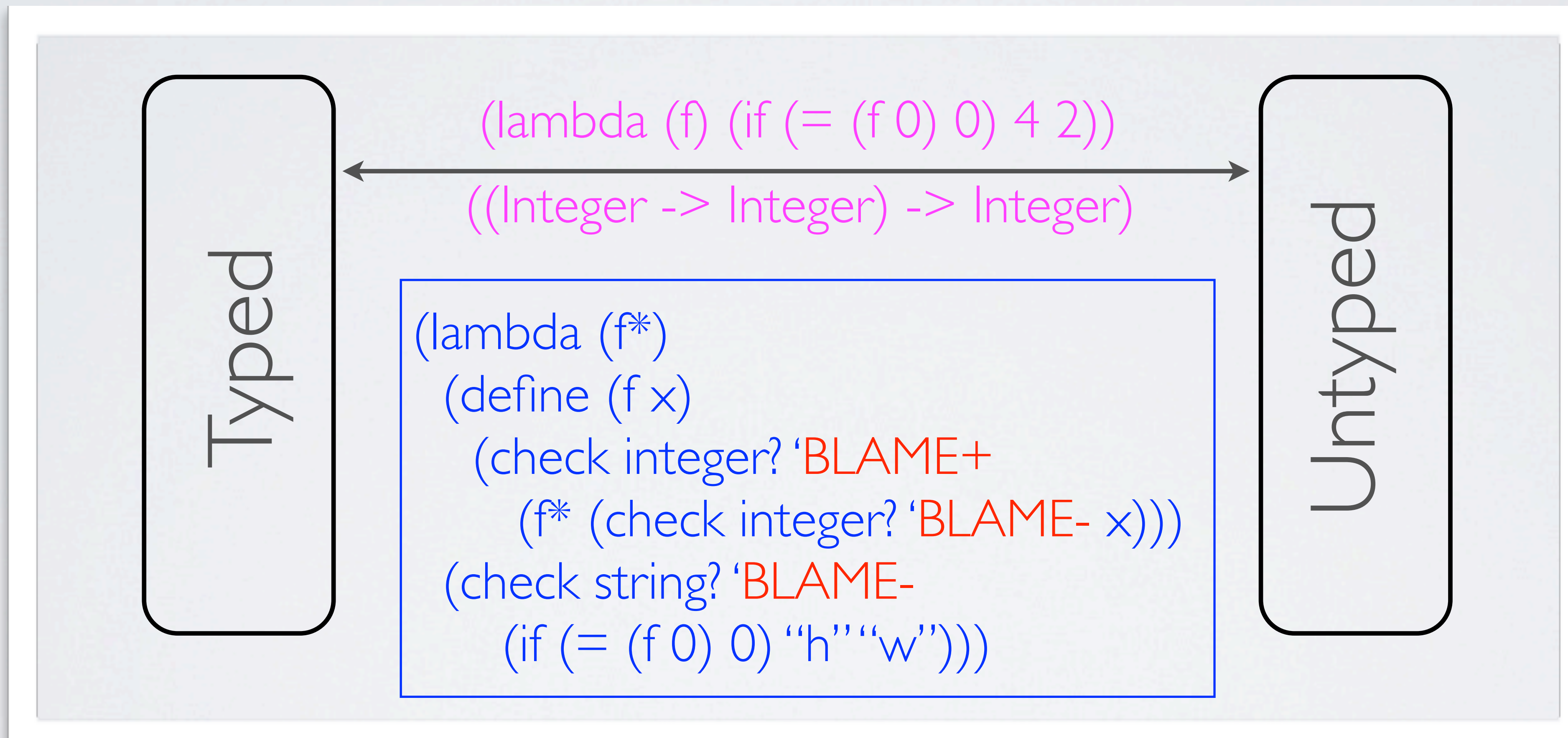
step 1: typed 'modules' must specify types for all imported variables and specify types for all exports

step 2: when values cross component boundaries, types are interpreted as contracts and wrapped around values to protect the typed components



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

step 2: when values cross component boundaries, types are interpreted as contracts and wrapped around values to protect the typed components



step 3: value flow between typed modules is free

Blame 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*.

Sam Tobin-Hochstadt
Dynamic Language Symposium
Portland. OR. 2006

LANGUAGES FROM MACROS

```
#lang typed/racket

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

(provide encode)

(: decode (Integer ->  ((Integer -> Integer) -> Integer)))
;; decodes input for f
(define (decode f)
    (if (f 0) (lambda (g) (g 42)) (lambda (h) (h 0))))
```

```
#lang typed/racket

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

(provide encode)

(: decode (Integer -> ((Integer -> Integer) -> Integer)))
;; decodes input for f
(define (decode f)
    (if (f 0) (lambda (g) (g 42)) (lambda (h) (h 0))))
```

What does this mean?

Racket languages are
components that implement
a **compiler** and
a **run-time library.**

Syntax Rewriting
+ Run-time Functions
= New Languages

Pattern-based Syntax Rewriting

```
(define-syntax-rule
  (pop x)

  ;; ==>>

  (begin0 (first x) (set! x (rest x))))
```

Pattern-based Syntax Rewriting

```
(define-syntax-rule
  (pop x)

  ;; ==>>

  (begin0 (first x) (set! x (rest x))))
```

Pattern-based Syntax Rewriting

```
(define-syntax-rule
  (pop x)

  ;; ==>>

  (begin0 (first x) (set! x (rest x))))
```

Procedural Syntax Rewriting

```
(define-syntax (define-un-serialize stx)
  (syntax-parse stx
    [(_ name:id (argument:id ...) unparser:expr parser:expr)

     (define serialize   (postfix stx "serialize"   (syntax-e #'name)))
     (define deserialize (postfix stx "deserialize" (syntax-e #'name)))

     #`(define-values (#,serialize #,deserialize)
          (values (lambda (argument ...) unparser)
                  (lambda (msg) parser)))]))
```

Procedural Syntax Rewriting

```
(define-syntax (define-un-serialize stx)
  (syntax-parse stx
    [(_ name:id (argument:id ...) unparser:expr parser:expr)

     (define serialize   (postfix stx "serialize"   (syntax-e #'name)))
     (define deserialize (postfix stx "deserialize" (syntax-e #'name)))

     #`(define-values (#,serialize #,deserialize)
         (values (lambda (argument ...) unparser)
                 (lambda (msg) parser)))]))
```

Procedural Syntax Rewriting

```
(define-syntax (define-un-serialize stx)
  (syntax-parse stx
    [(_ name:id (argument:id ...) unparser:expr parser:expr)

     (define serialize   (postfix stx "serialize"   (syntax-e #'name)))
     (define deserialize (postfix stx "deserialize" (syntax-e #'name)))

     #`(define-values (#,serialize #,deserialize)
         (values (lambda (argument ...) unparser)
                 (lambda (msg) parser)))]))
```

Procedural Syntax Rewriting

```
(define-syntax (define-un-serialize stx)
  (syntax-parse stx
    [(_ name:id (argument:id ...) unparser:expr parser:expr)

     (define serialize   (postfix stx "serialize"   (syntax-e #'name)))
     (define deserialize (postfix stx "deserialize" (syntax-e #'name)))

     #`(define-values (#,serialize #,deserialize)
         (values (lambda (argument ...) unparser)
                 (lambda (msg) parser)))]))
```

Syntax Rewriting

\+ Run-time Functions

= New Languages

Syntax Rewriting
+ Run-time Functions
= New Languages

```
(define-syntax-rule
    (: id a-type)
    ;; ==>>
    (let ([identifier (expand 'id (this-module))]
           [its-type    (normalize 'a-type))])
      (insert identifier its-type)))
```

Syntax Rewriting

\+ Run-time Functions

= New Languages

```
(define-syntax-rule
   (: id a-type)
   ;; ==>>
   (let ([identifier (expand ‘id (this-module))]
         [its-type   (normalize ‘a-type))])
     (insert ‘identifier its-type)))
```

Syntax Rewriting
+ Run-time Functions
= New Languages

```
(define-syntax-rule
    (: id a-type)
    ;; ==>>
    (let ([identifier (expand 'id (this-module))]
          [its-type   (normalize 'a-type)])
      (insert 'identifier its-type)))
```

```
(define (expand identifier module-path)
   (form-full-path identifier module-path '()))

(define (normalize type)
   (sort-unions (get-type-names type)))

(define (insert identifier its-type))
   (send *type-environment* add-set
            identifier its-type)))
```

Syntax Rewriting
+ Run-time Functions
= New Languages

typed/racket.rkt

```
(define-syntax-rule
   (: id a-type)
   ;; ==>>
   (let ([identifier (expand 'id (this-module))]
         [its-type   (normalize 'a-type))])
     (insert identifier its-type)))
```

```
(define (expand identifier module-path)
  (form-full-path identifier module-path '()))

(define (normalize type)
  (sort-unions (get-type-names type)))

(define (insert identifier its-type))
  (send *type-environment* add-set
        identifier its-type)))
```

Substitution 1: Macro bodies are substituted for macro calls.

```
#lang typed/racket

(: f (Integer -> Integer))
...
```

typed/racket.rkt

```
(define-syntax-rule
   (: id a-type)
   ;; ==>>
   (let ([identifier (expand 'id (this-module))]
         [its-type   (normalize 'a-type)]])
     (insert 'identifier its-type)))
```

```
(define (expand identifier module-path)
  (form-full-path identifier module-path '()))

(define (normalize type)
  (sort-unions (get-type-names type)))

(define (insert identifier its-type))
  (send *type-environment* add-set
        identifier its-type)))
```

Substitution 1: Macro bodies are substituted for macro calls.

```
#lang typed/racket

(: f (Integer -> Integer))
...
```

expand ↓

```
#lang typed/racket

(let ([identifier
        (expand 'f (this-module))]
      [its-type
        (normalize
          '(Integer -> Integer))])
  (insert identifier its-type))
...
```

typed/racket.rkt

```
(define-syntax-rule
   (: id a-type)
   ;; ==>>
   (let ([identifier (expand 'id (this-module))]
         [its-type   (normalize 'a-type)])
     (insert identifier its-type)))
```

```
(define (expand identifier module-path)
  (form-full-path identifier module-path '()))

(define (normalize type)
  (sort-unions (get-type-names type)))

(define (insert identifier its-type))
  (send *type-environment* add-set
        identifier its-type)))
```

Substitution 1: Macro bodies are substituted for macro calls.

```
#lang typed/racket

(: f (Integer -> Integer))
...
```

expand ↓

```
#lang typed/racket

(let ([identifier
        (expand 'f (this-module))]
      [its-type
        (normalize
          '(Integer -> Integer))])
  (insert identifier its-type))
...
```

typed/racket.rkt

```
(define-syntax-rule
   (: id a-type)
   ;; ==>>
   (let ([identifier (expand 'id (this-module))]
         [its-type   (normalize 'a-type)])
     (insert identifier its-type)))
```

```
(define (expand identifier module-path)
  (form-full-path identifier module-path '()))

(define (normalize type)
  (sort-unions (get-type-names type)))

(define (insert identifier its-type))
  (send *type-environment* add-set
        identifier its-type)))
```

Substitution 1: Macro bodies are substituted for macro calls.

```
#lang typed/racket

(: f (Integer -> Integer))
...
```

expand

```
#lang typed/racket

(let ([identifier
        (expand 'f (this-module))]
      [its-type
        (normalize
          '(Integer -> Integer))])
  (insert identifier its-type))
...
```

typed/racket.rkt

```
(define-syntax-rule
   (: id a-type)
   ;; ==>>
   (let ([identifier (expand 'id (this-module))]
         [its-type   (normalize 'a-type)])
     (insert identifier its-type)))
```

```
(define (expand identifier module-path)
   (form-full-path identifier module-path '()))

(define (normalize type)
   (sort-unions (get-type-names type)))

(define (insert identifier its-type))
   (send *type-environment* add-set
         identifier its-type)))
```

Substitution 1: Macro bodies are substituted for macro calls.

```
#lang typed/racket

(define (expand x) 42)
(: f (Integer -> Integer))
...
```

expand

```
#lang typed/racket

(define (expand x) 42)
(let ([identifier
        (expand 'f (this-module))]
      [its-type
        (normalize
          '(Integer -> Integer))])
  (insert identifier its-type))
...
```

typed/racket.rkt

```
(define-syntax-rule
  (: id a-type)
  ;; ==>>
  (let ([identifier (expand 'id (this-module))]
        [its-type   (normalize 'a-type)])
    (insert identifier its-type)))
```

```
(define (expand identifier module-path)
  (form-full-path identifier module-path '()))

(define (normalize type)
  (sort-unions (get-type-names type)))

(define (insert identifier its-type))
  (send *type-environment* add-set
        identifier its-type)))
```

Substitution 2: Macro arguments are substituted into macro bodies.

typed/racket.rkt

```
#lang typed/racket

(define-type Shapes (U Square Circle))
...
```

```
(define-syntax-rule
   (define-type T Type)
   ;; ==>>
   (begin
     (define NormalType (normalize 'Type))

     (define T NormalType)))
```

```
...

(define (normalize type)
  (sort-unions (get-type-names type)))

...
```

Substitution 2: Macro arguments are substituted into macro bodies.

```
#lang typed/racket

(define-type Shapes (U Square Circle))
...
```

typed/racket.rkt

```
(define-syntax-rule
  (define-type T Type)
  ;; ==>>
  (begin
    (define NormalType (normalize 'Type))

    (define T NormalType)))
```

```
...

(define (normalize type)
  (sort-unions (get-type-names type)))

...
```

Substitution 2: Macro arguments are substituted into macro bodies.

typed/racket.rkt

```
#lang typed/racket

(define-type Shapes (U Square Circle))
...
```

expand

```
#lang typed/racket

(define NormalType (normalize 'Type))


(define T NormalType))
```

```
(define-syntax-rule
   (define-type T Type)
   ;; ==>>
   (begin
     (define NormalType (normalize 'Type))

     (define T NormalType))
```

```
...

(define (normalize type)
  (sort-unions (get-type-names type)))

...
```

Substitution 2: Macro arguments are substituted into macro bodies.

```
#lang typed/racket

(define-type Shapes (U Square Circle))
...
```

expand

```
#lang typed/racket

(define NormalType (normalize 'Type))

(define T NormalType))
```

typed/racket.rkt

```
(define-syntax-rule
   (define-type T Type)
   ;; ==>>
   (begin
     (define NormalType (normalize 'Type))

     (define T NormalType)))
```

```
...

(define (normalize type)
  (sort-unions (get-type-names type)))

...
```

Substitution 2: Macro arguments are substituted into macro bodies.

```
#lang typed/racket

(define (NormalType x) x)
(define-type Shapes (U Square Circle))
...
```

expand

```
#lang typed/racket

(define (NormalType x) x)
(define NormalType (normalize 'Type))

(define T NormalType))
```

typed/racket.rkt

```
(define-syntax-rule
   (define-type T Type)
   ;; ==>>
   (begin
     (define NormalType (normalize 'Type))

     (define T NormalType)))
```

```
...

(define (normalize type)
   (sort-unions (get-type-names type)))

...
```

Macro hygiene ensures that two different substitutions work as intended **by default**.

Macro hygiene ensures that two different substitutions work as intended **by default**.

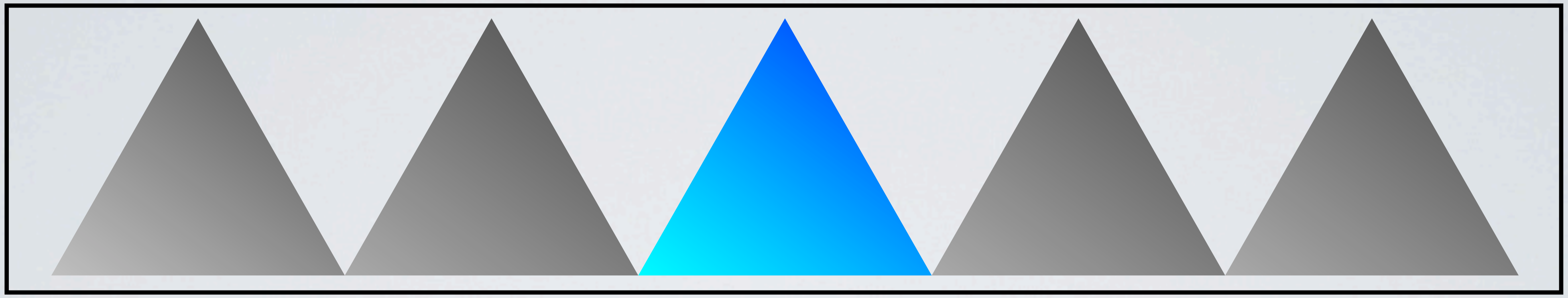
Programmers can override the defaults.

Contrary to rumors in the CL world:
Hygienic macros **increase** the **expressive power** of the macros system.

But macros are only half the story.

But macros are only half the story.

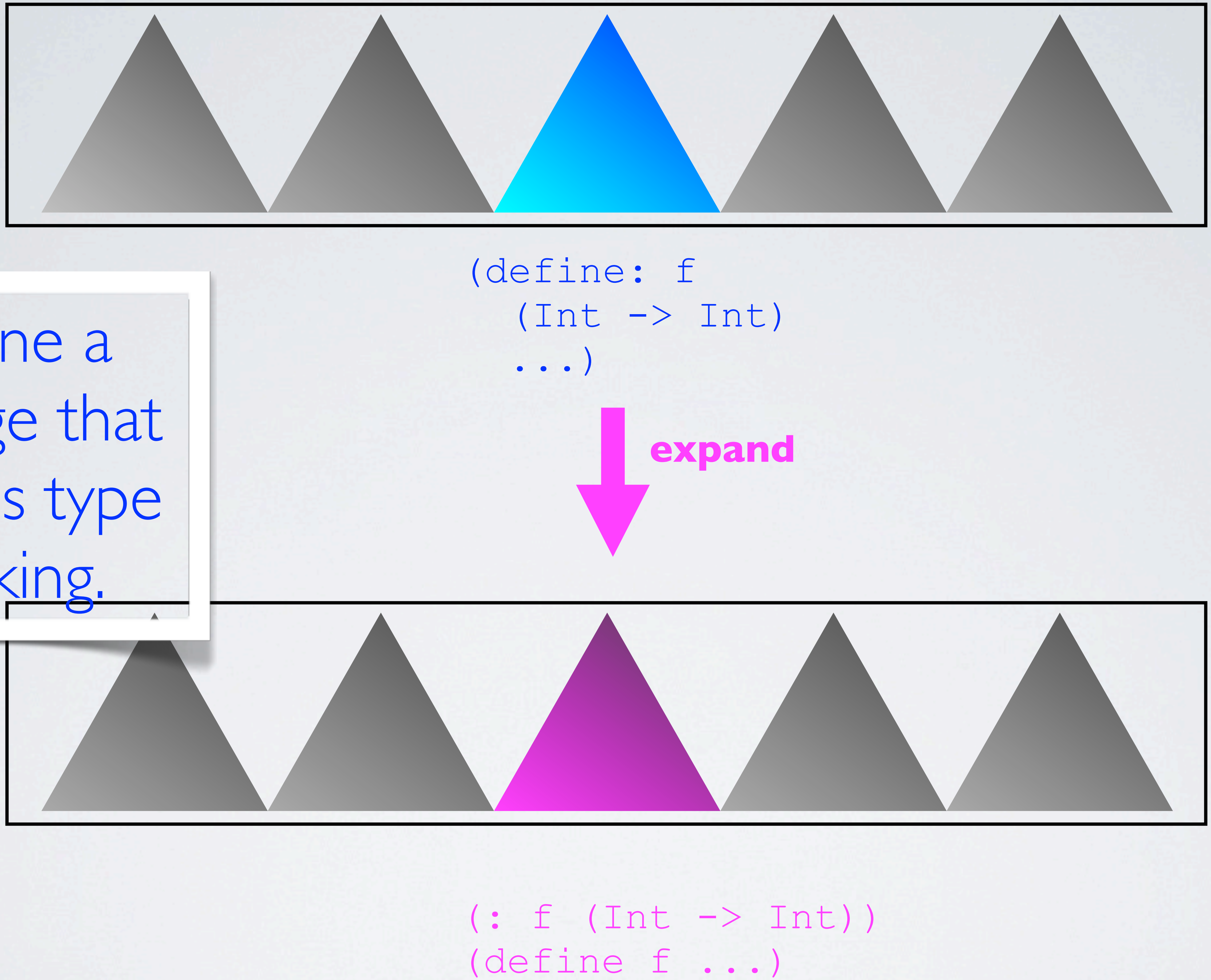
Macros are (mostly) *context-free* rewriting rules.
Implementing languages requires *context-sensitivity*.

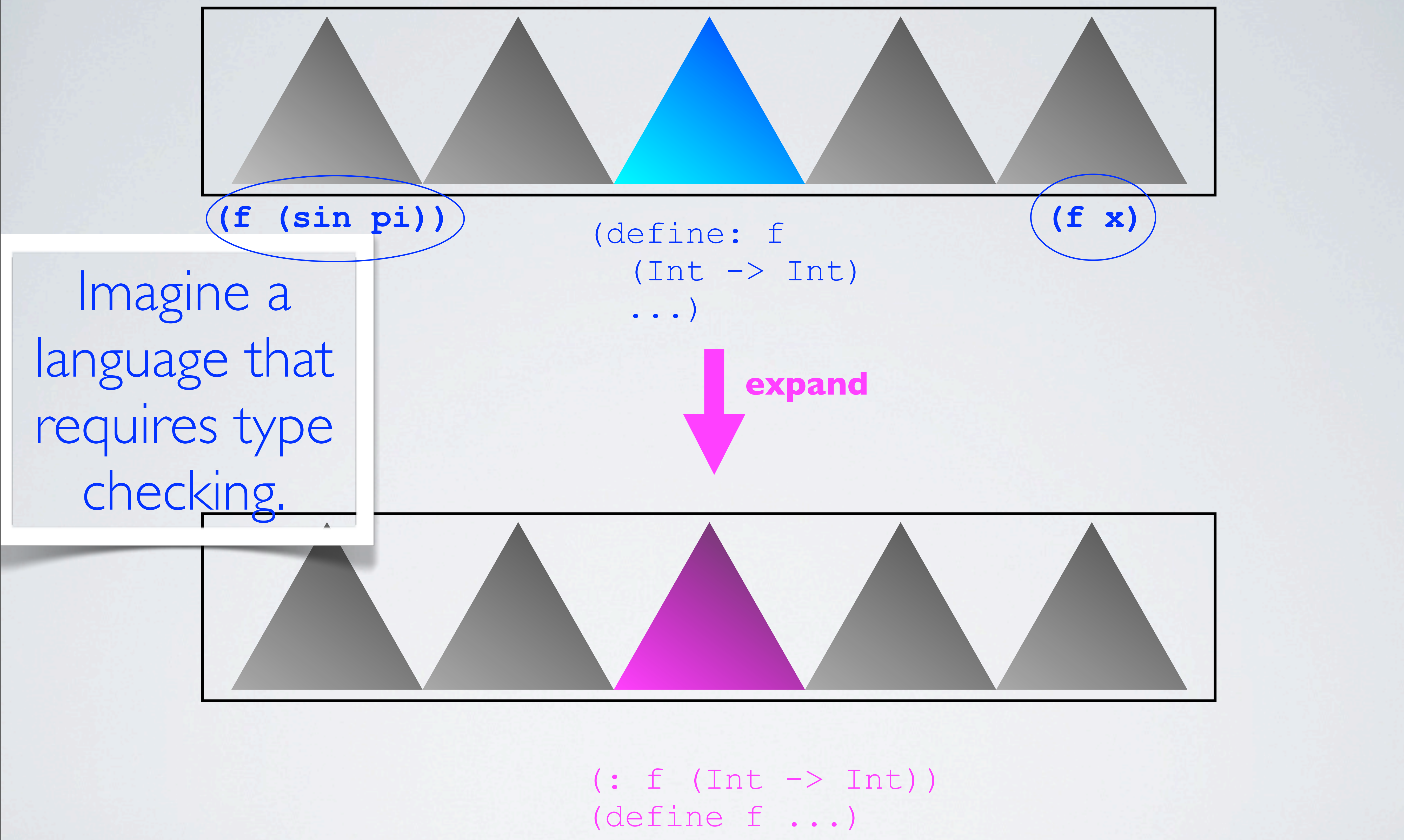


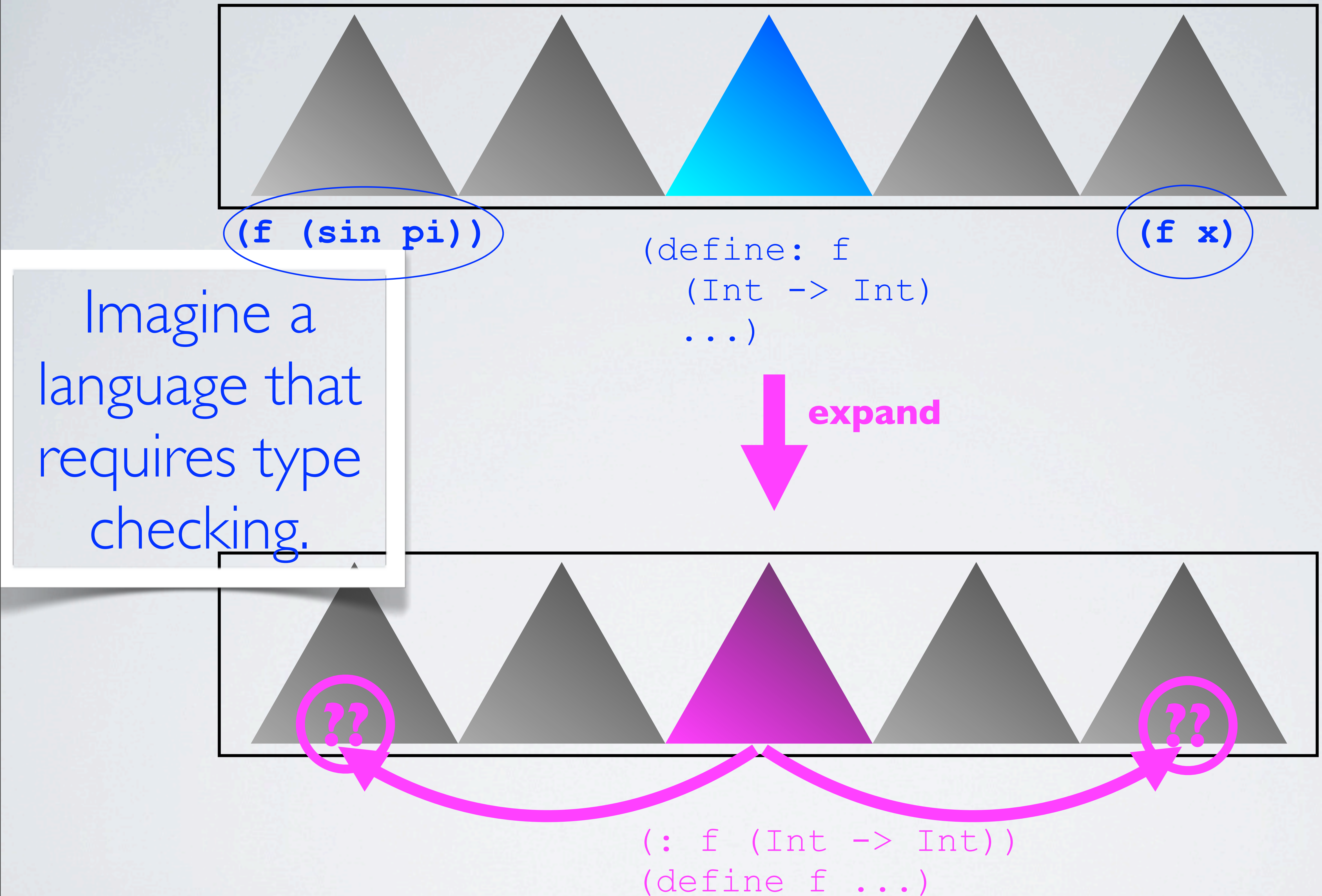
```
(define: f
  (Int -> Int)
  ...)
```

Imagine a language that requires type checking.

Imagine a language that requires type checking.

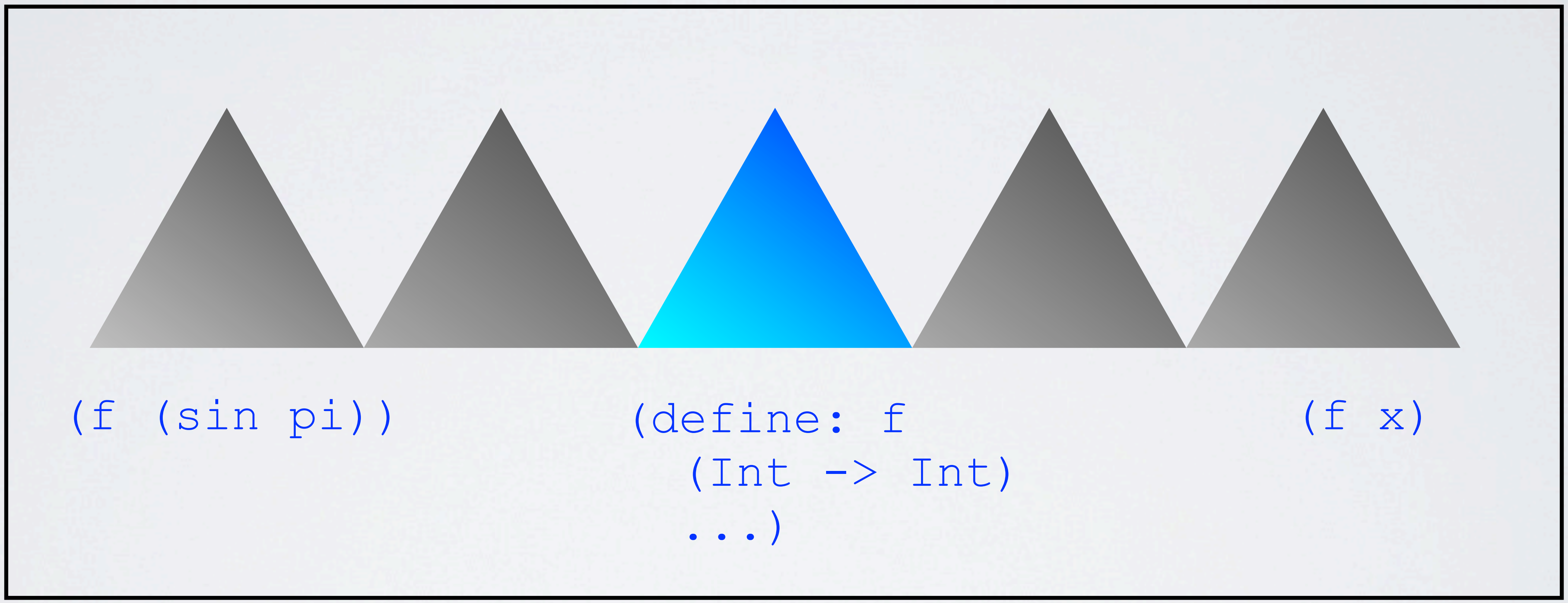



(f (sin pi))
(define: f
(Int -> Int)
...)
(f x)
Imagine a language that requires type checking.
expand
(: f (Int -> Int))
(define f ...)

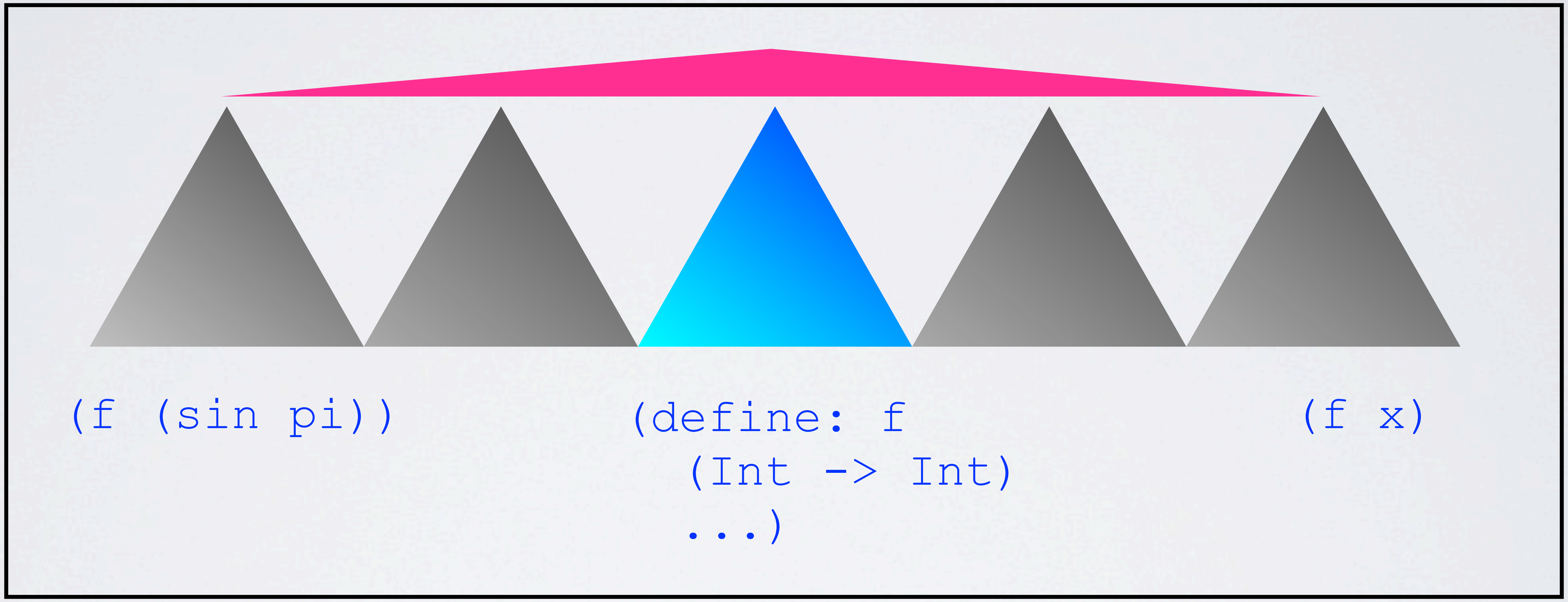

(f (sin pi))
(define: f
(Int -> Int)
...)
(f x)
Imagine a language that requires type checking.
expand
??
??
(: f (Int -> Int))
(define f ...)

Macros rewrite trees.
They cannot communicate to contexts.

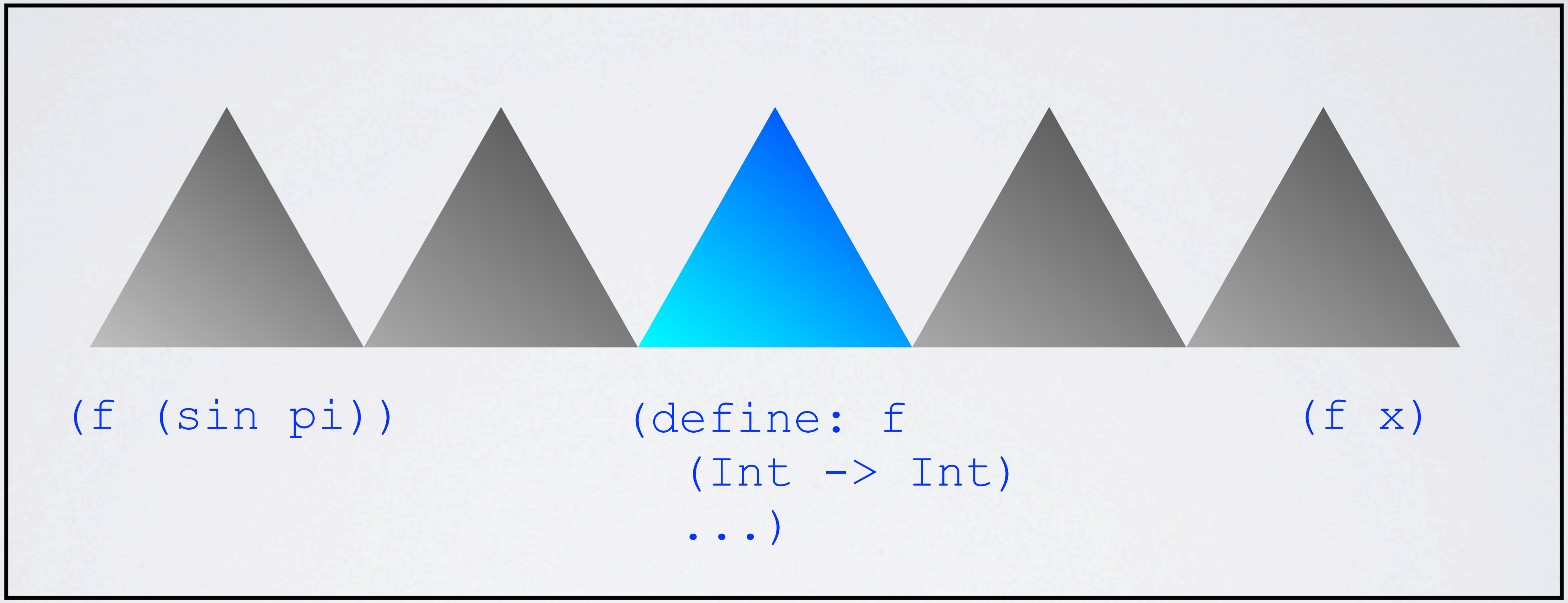
Languages require whole-module processing.



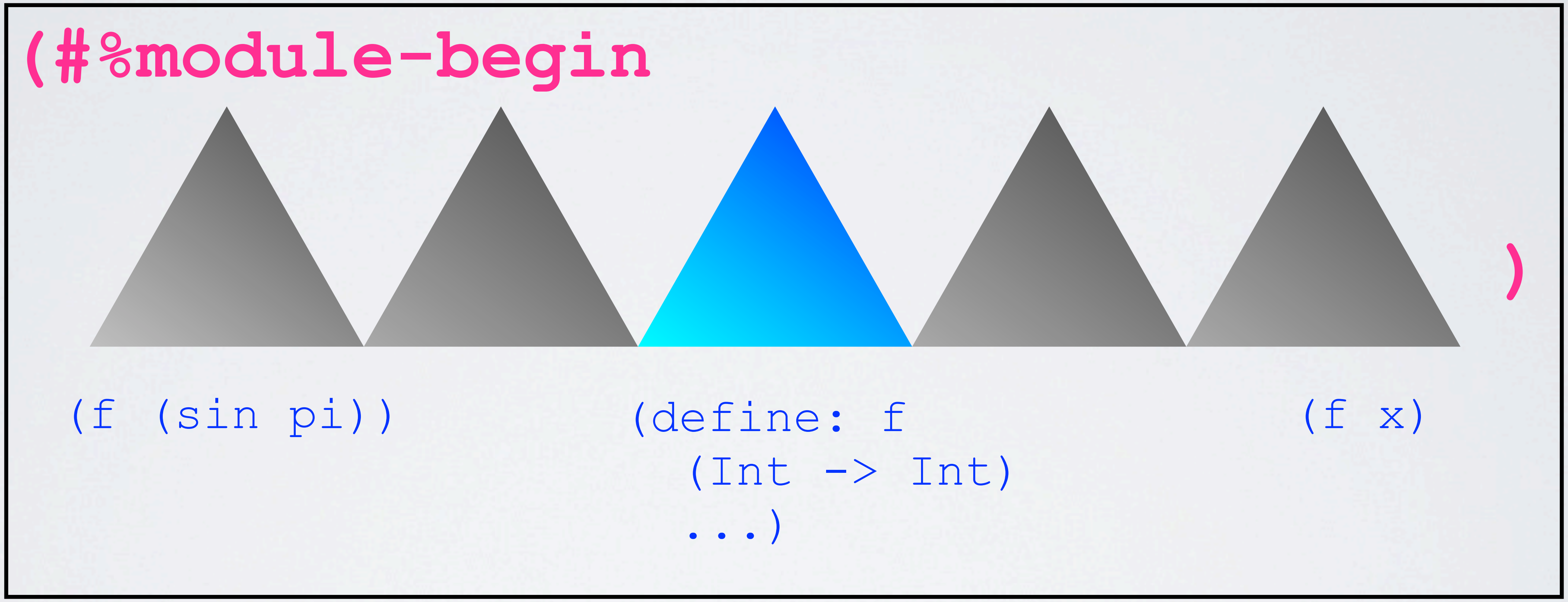
Languages require whole-module processing.



Languages require whole-module processing.

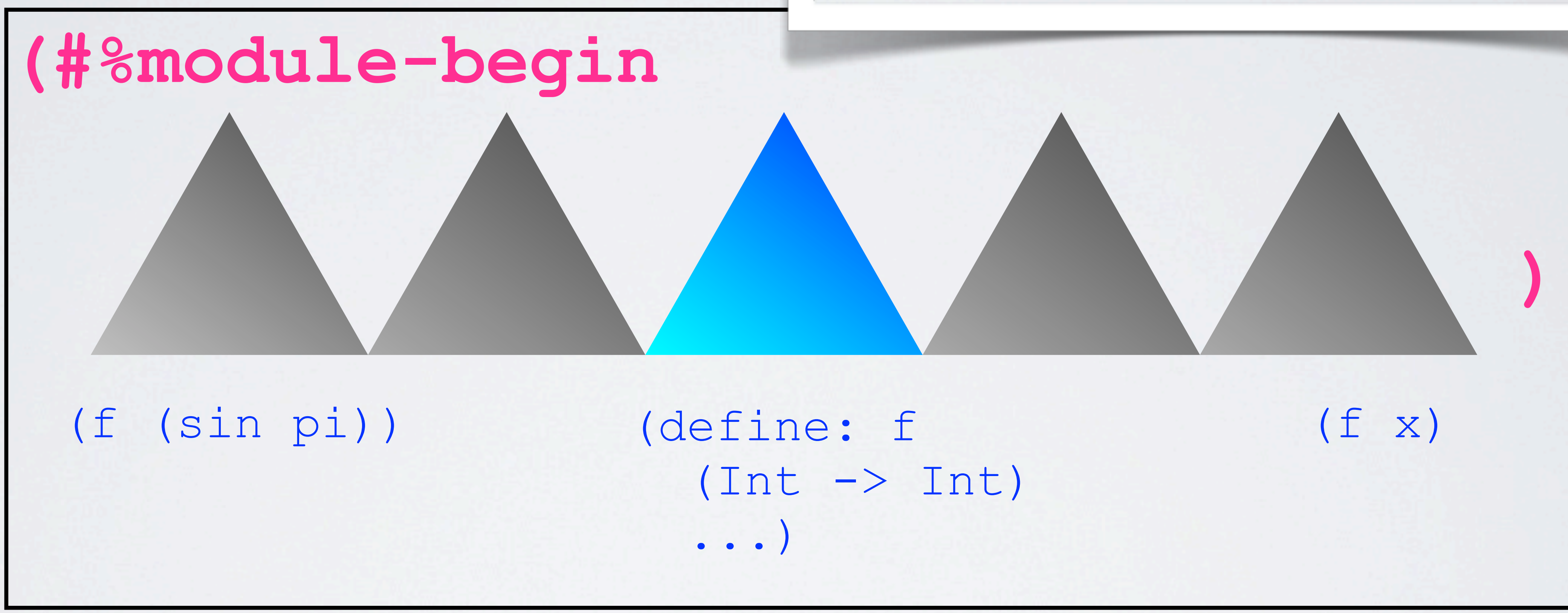


Languages require whole-module processing.



Languages require whole-module processing.

And languages may redefine **#%module-begin**.



Let's make context-sensitive processing concrete.

Let's make context-sensitive processing concrete.

silly.rkt

```
#lang racket

(provide
  ... ;; additional exports
  (rename-out (new-module-begin #%module-begin)))

(define-syntax-rule
  (new-module-begin mexpr ...)
  ;; ==>>
  (#%module-begin
    (begin
      (count++)
      (printf "evaluating the ~a~a part\n" (count) (st-or-th))
      mexpr)
    ...))
```

Let’s make context-sensitive processing concrete.

silly.rkt

```
#lang racket

(provide
  ... ;; additional exports
  (rename-out (new-module-begin #%module-begin)))

(define-syntax-rule
  (new-module-begin mexpr ...)
  ;; ==>>
  (#%module-begin
    (begin
      (count++)
      (printf "evaluating the ~a~a part\n" (count) (st-or-th))
      mexpr)
    ...))
```

Let's make context-sensitive processing concrete.

client.rkt

```
#lang s-exp "silly.rkt"

(define (f x)
  (+ (g (* 10 x)) 1))

(define (g y)
  (/ y 2))
```

silly.rkt

```
#lang racket

(provide
  ... ;; additional exports
  (rename-out (new-module-begin #%module-begin)))

(define-syntax-rule
  (new-module-begin mexpr ...)
  ;; ==>>
  (#%module-begin
    (begin
      (count++)
      (printf "evaluating the ~a~a part\n" (count) (st-or-th))
      mexpr)
    ...))
```

Let's make context-sensitive processing concrete.

client.rkt

```
#lang s-exp "silly.rkt"

(define (f x)
  (+ (g (* 10 x)) 1))

(define (g y)
  (/ y 2))
```

silly.rkt

```
#lang racket

(provide
  ... ;; additional exports
  (rename-out (new-module-begin #%module-begin)))

(define-syntax-rule
  (new-module-begin mexpr ...)
  ;; ==>>
  (#%module-begin
    (begin
      (count++)
      (printf "evaluating the ~a~a part\n" (count) (st-or-th))
      mexpr)
    ...))
```

client.rkt

```
#lang s-exp "silly.rkt"

(define (f x) (+ (g (* 10 x)) 1))

(define (g y) (/ y 2))
```

client.rkt

```
#lang s-exp "silly.rkt"

(define (f x) (+ (g (* 10 x)) 1))

(define (g y) (/ y 2))
```

expand

client.rkt : **expanded**

```
(module simple-in-silly "silly.rkt"
  (#%module-begin
    (count++)
    (printf "evaluating the ~a~a part\n" (count) (st-or-th))
    (define (f x) (+ (g (* 10 x)) 1)))
    (count++)
    (printf "evaluating the ~a~a part\n" (count) (st-or-th))
    (define (g y) (/ y 2)))))
```

client.rkt : **expanded**

```
(module simple-in-silly "silly.rkt"
  (#%module-begin
     (count++)
     (printf "evaluating the ~a~a part\n" (count) (st-or-th))
     (define (f x) (+ (g (* 10 x)) 1)))
     (count++)
     (printf "evaluating the ~a~a part\n" (count) (st-or-th))
     (define (g y) (/ y 2)))))
```

client.rkt : **expanded**

```
(module simple-in-silly "silly.rkt"
  (#%module-begin
    (count++)
    (printf "evaluating the ~a~a part\n" (count) (st-or-th))
    (define (f x) (+ (g (* 10 x)) 1))
    (count++)
    (printf "evaluating the ~a~a part\n" (count) (st-or-th))
    (define (g y) (/ y 2))))
```

run

client.rkt : **run**

```
Welcome to DrRacket, version 5.2.0.1--2011-10-16
(2a43c68/g) [3m].
Language: s-exp "silly.rkt".
evaluating the 1st part
evaluating the 2nd part
> (f 1)
6
>
```

client.rkt : **expanded**

```
(module simple-in-silly "silly.rkt"
  (#%module-begin
    (count++)
    (printf "evaluating the ~a~a part\n" (count) (st-or-th))
    (define (f x) (+ (g (* 10 x)) 1)))
    (count++)
    (printf "evaluating the ~a~a part\n" (count) (st-or-th))
    (define (g y) (/ y 2)))))
```

run

client.rkt : **run**

```
Welcome to DrRacket, version 5.2.0.1--2011-10-16
(2a43c68/g) [3m].
Language: s-exp "silly.rkt".
evaluating the 1st part
evaluating the 2nd part
> (f 1)
6
>
```

Typed Racket's module-begin, mostly.

```
(define-syntax (typed-module-begin stx)
  (syntax-parse stx
    [(_ s ...)
     (with-syntax ([(_ core-s ...)  (local-expand #'(#%module-begin s ...))])

        (for-each typecheck (syntax->list #'(core-s ...)))

        #'(#%module-begin core-s ...))]))
```

Typed Racket's module-begin, mostly.

```
(define-syntax (typed-module-begin stx)
  (syntax-parse stx
    [(_ s ...)
     (with-syntax ([(_ core-s ...)  (local-expand #'(#%module-begin s ...))])

       (for-each typecheck (syntax->list #'(core-s ...)))

       #'(#%module-begin core-s ...))]))
```

Typed Racket's module-begin, mostly.

```
(define-syntax (typed-module-begin stx)
  (syntax-parse stx
    [(_ s ...)
     (with-syntax ([(_ core-s ...)  (local-expand #'(#%module-begin s ...))])

       (for-each typecheck (syntax->list #'(core-s ...)))

       #'(#%module-begin core-s ...))]))
```

Typed Racket's module-begin, mostly.

```
(define-syntax (typed-module-begin stx)
  (syntax-parse stx
    [(_ s ...)
     (with-syntax ([(_ core-s ...)  (local-expand #'(#%module-begin s ...))])

       (for-each typecheck (syntax->list #'(core-s ...)))

       #'(#%module-begin core-s ...))]))
```

Typed Racket's module-begin, one more bit.

server.rkt

```
#lang typed/racket

(: f (Byte -> Index))
(define (f x)
  (+ x 22))

(provide f)
```

Typed Racket's module-begin, one more bit.

untyped.rkt

```
#lang racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

server.rkt

```
#lang typed/racket

(: f (Byte -> Index))
(define (f x)
  (+ x 22))

(provide f)
```

Typed Racket's module-begin, one more bit.

server.rkt

```
#lang typed/racket

(: f (Byte -> Index))
(define (f x)
   (+ x 22))

(provide f)
```

untyped.rkt

```
#lang racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

typed.rkt

```
#lang typed/racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

Typed Racket's module-begin, one more bit.

server.rkt

```
#lang typed/racket

(: f (Byte -> Index))
(define (f x)
   (+ x 22))

(provide f)
```

insert contracts

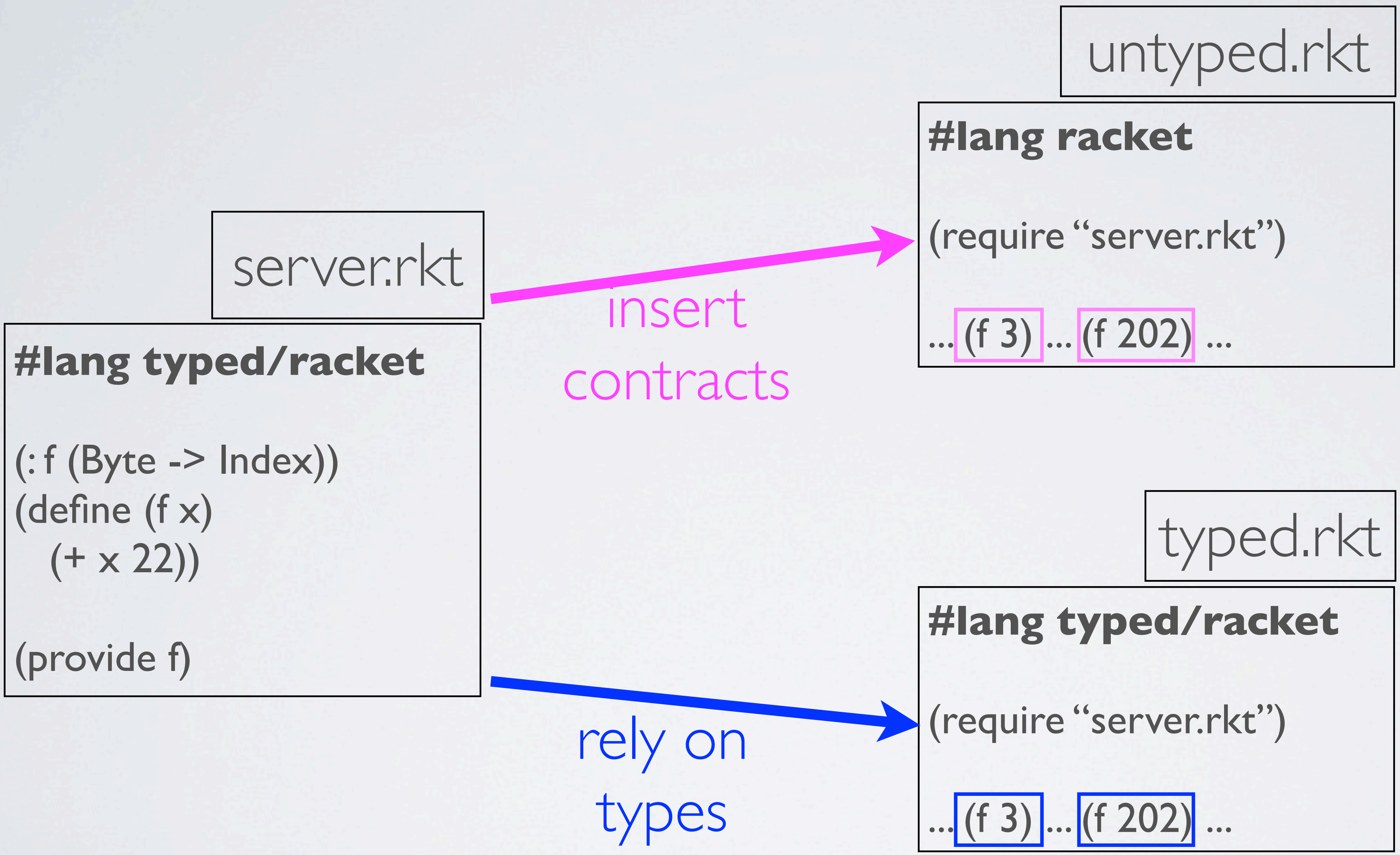
untyped.rkt

```
#lang racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

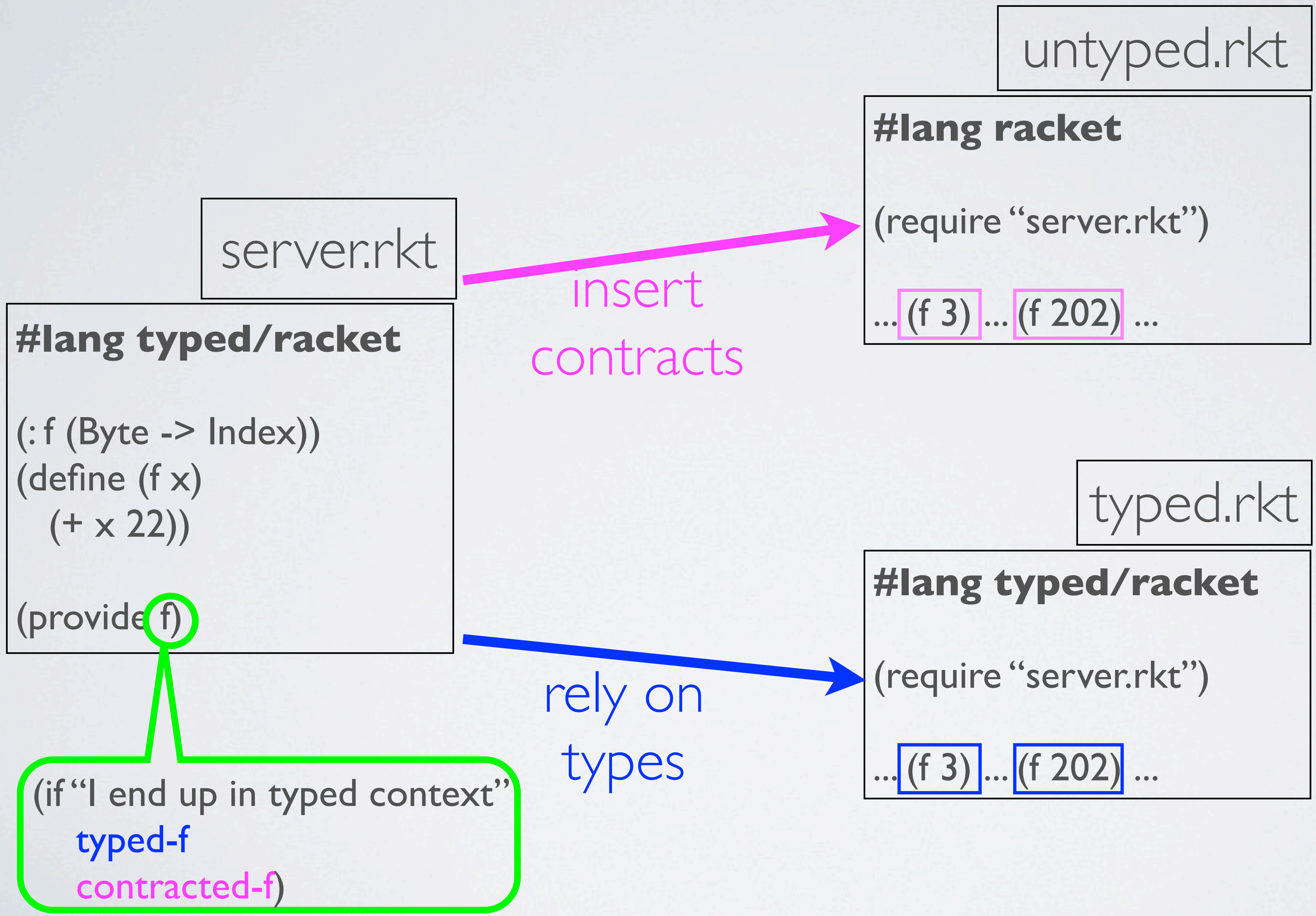
typed.rkt

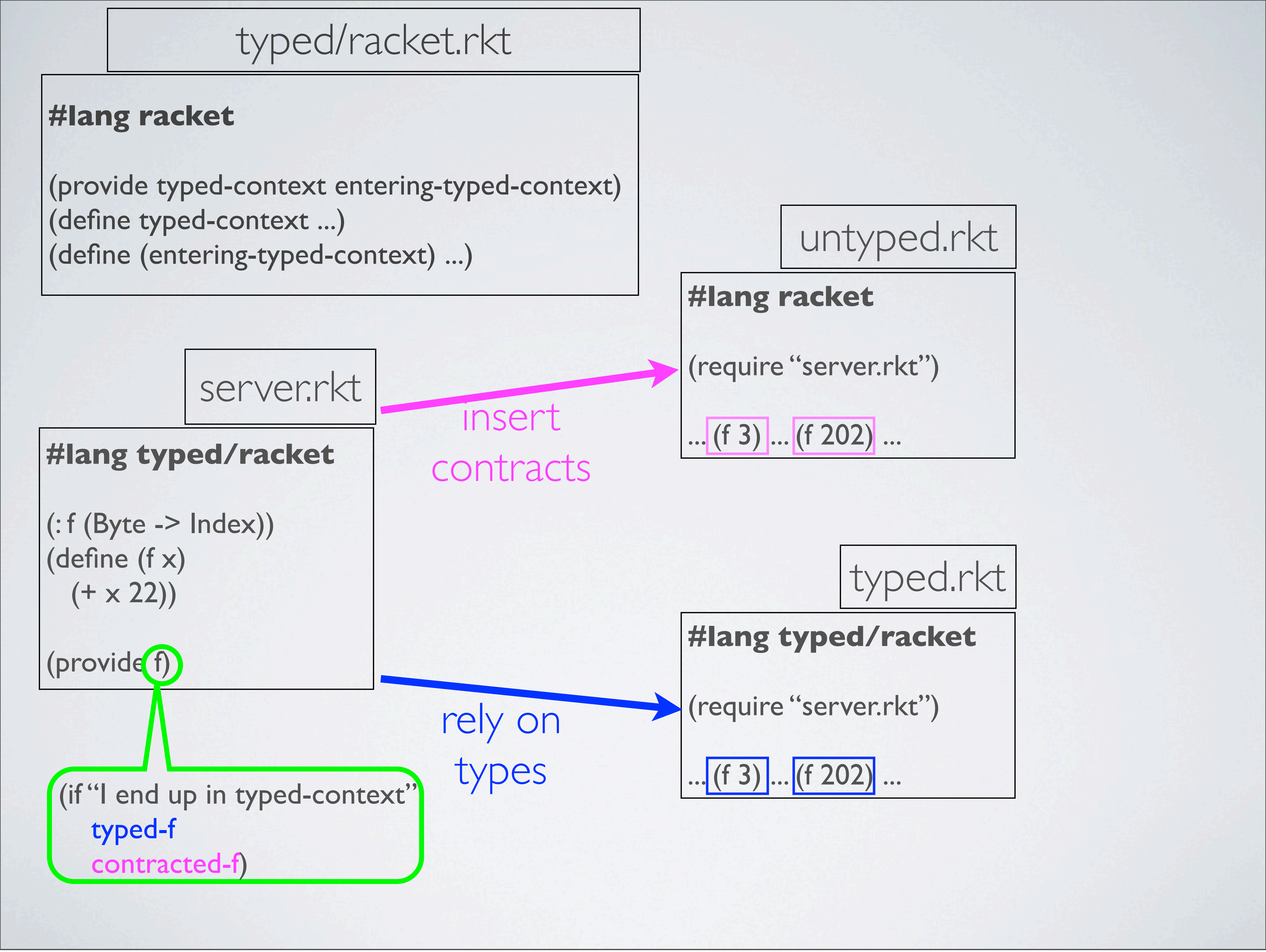
```
#lang typed/racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

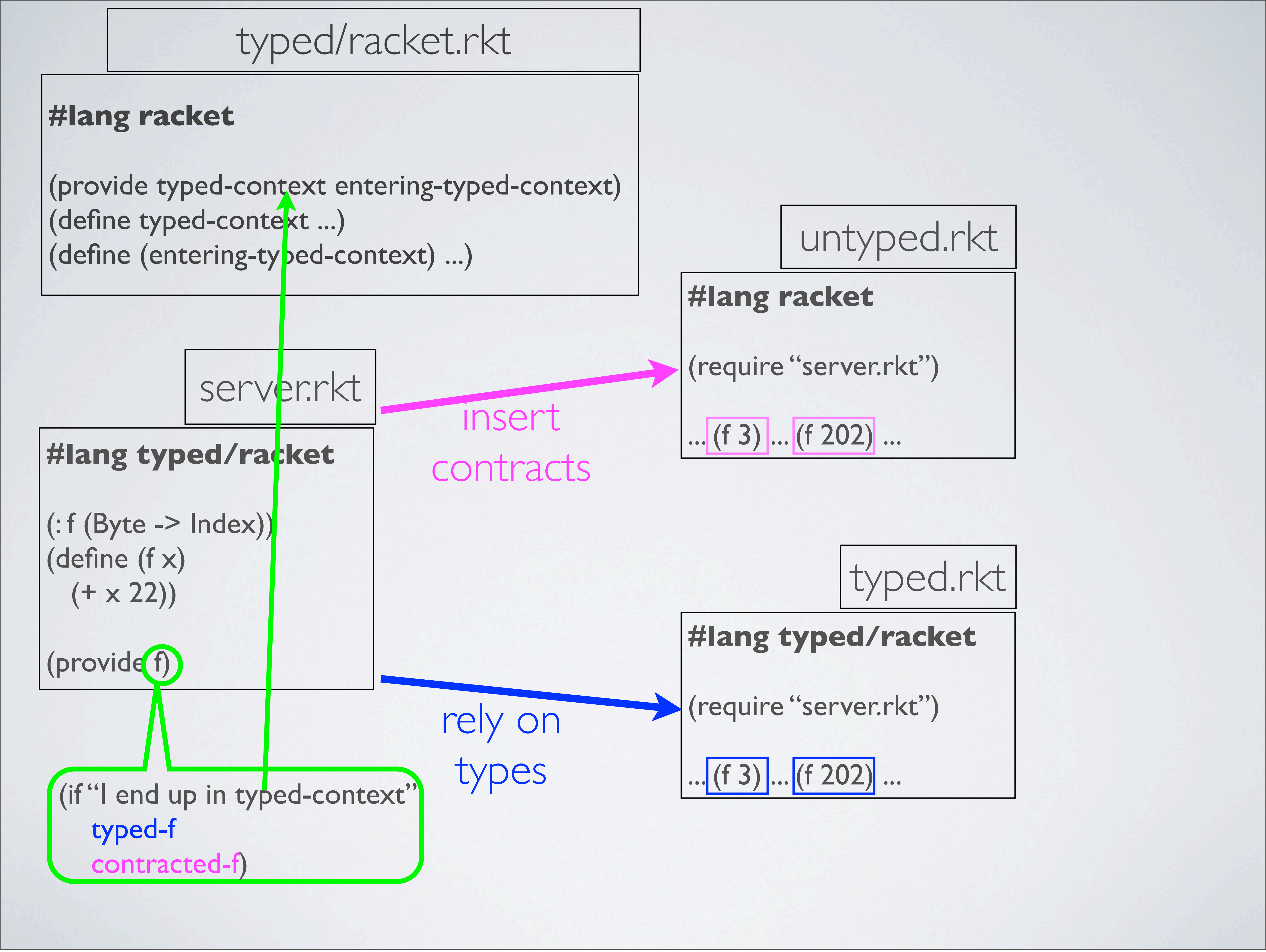
Typed Racket's module-begin, one more bit.

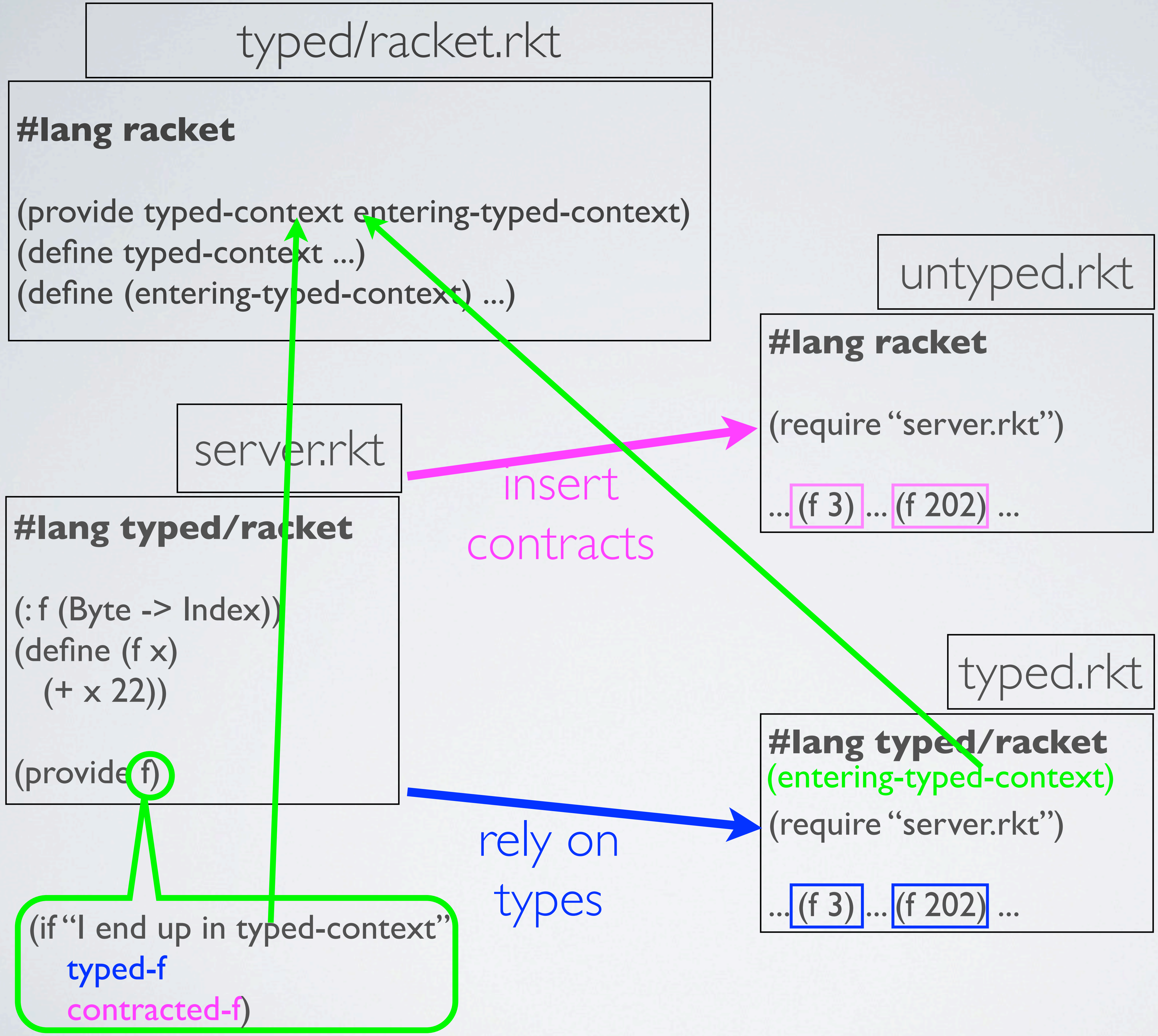


Typed Racket's module-begin, one more bit.




typed/racket.rkt
#lang racket
(provide typed-context entering-typed-context)
(define typed-context ...)
(define (entering-typed-context) ...)
untyped.rkt
#lang racket
(require "server.rkt")
... (f 3) ... (f 202) ...
server.rkt
insert contracts
#lang typed/racket
(: f (Byte -> Index))
(define (f x)
(+ x 22))
(provide f)
typed.rkt
#lang typed/racket
(require "server.rkt")
... (f 3) ... (f 202) ...
rely on types
(if "I end up in typed-context"
typed-f
contracted-f)


typed/racket.rkt
#lang racket
(provide typed-context entering-typed-context)
(define typed-context ...)
(define (entering-typed-context) ...)
untyped.rkt
#lang racket
(require “server.rkt”)
... (f 3) ... (f 202) ...
server.rkt
#lang typed/racket
(: f (Byte -> Index))
(define (f x)
(+ x 22))
(provide f)
insert
contracts
typed.rkt
#lang typed/racket
(require “server.rkt”)
... (f 3) ... (f 202) ...
rely on
types
(if “I end up in typed-context”
typed-f
contracted-f)


typed/racket.rkt
#lang racket
(provide typed-context entering-typed-context)
(define typed-context ...)
(define (entering-typed-context) ...)
untyped.rkt
#lang racket
(require "server.rkt")
... (f 3) ... (f 202) ...
server.rkt
insert
contracts
#lang typed/racket
(: f (Byte -> Index))
(define (f x)
(+ x 22))
(provide f)
typed.rkt
#lang typed/racket
(entering-typed-context)
(require "server.rkt")
... (f 3) ... (f 202) ...
rely on
types
(if "I end up in typed-context"
typed-f
contracted-f)

typed.rkt

```
#lang typed/racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

typed.rkt

```
#lang typed/racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

expand

typed.rkt: **expanded**

```
#lang racket

(entering-typed-context)
(define-syntax f
 (if "I end up in typed-context"
      typed-f
      contracted-f))

... (f 3) ... (f 202) ...
```

typed.rkt

#lang typed/racket

(require "server.rkt")

... (f 3) ... (f 202) ...

expand

typed.rkt: **expanded**

#lang racket

(entering-typed-context)
(define-syntax f
 (if "I end up in typed-context"
  typed-f
  contracted-f))

... (f 3) ... (f 202) ...

fully expanded

typed.rkt: **expanded**

(module typed.rkt racket/base

... (typed-f 3) ... (typed-f 202) ...)

untyped.rkt

```
#lang racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

untyped.rkt

```
#lang racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

expand

untyped: **expanded**

```
#lang racket

(define-syntax f
 (if "I end up in typed-context"
      typed-f
      contracted-f))

... (f 3) ... (f 202) ...
```

untyped.rkt

```
#lang racket

(require "server.rkt")

... (f 3) ... (f 202) ...
```

expand

untyped: **expanded**

```
#lang racket

(define-syntax f
 (if "I end up in typed-context"
     typed-f
     contracted-f))

... (f 3) ... (f 202) ...
```

fully expanded

untyped: **expanded**

```
(module typed.rkt racket/base

   ... (contracted-f 3) ...
       (contracted-f 202) ... )
```

The World of Macros

- Racket, the language
- the macro tools
- experience

Culpepper & **Flatt** et al:
Languages as Libraries, PLDI 2011
Fortifying Macros, ICFP 2010
Debugging Macros, GPCE 2008
Composable, Compilable Macros,
ICFP 2002

CONCLUSION

Two ideas from Racket for everyone at GPCE.

- a macro system to implement *entire* languages
- *safe* component interaction in a multi-lingual world

Macros for *entire* languages require:

- *hygienic* and *fortified* macros
- macros as *module exports*
- *module-level* macros

Macros for *entire* languages require:

- *hygienic* and *fortified* macros
- macros as *module exports*
- *module-level* macros

We have built dozens of large and little languages. How can you import the ideas?

A multi-lingual world isn't free.
Safe interaction among multi-lingual components.

- languages have *invariants*
- interactions must *respect* these *invariants*
- **example**: sound *typed-untyped* interactions

A multi-lingual world isn't free.
Safe interaction among multi-lingual components.

- languages have *invariants*
- interactions must *respect* these *invariants*
- **example**: sound *typed-untyped* interactions

Many more problems exist in this area, and you are in a position to tackle them.

THE END

http://racket-lang.org/

Ryan Culpepper (Utah)	macros, macros, macros
Matthew Flatt (Utah)	language, compiler, macros
Shriram Krishnamurthi (Brown)	macros and modules
Robby Findler (Northwestern)	contracts, IDE
Sam Tobin-Hochstadt (Northeastern)	types