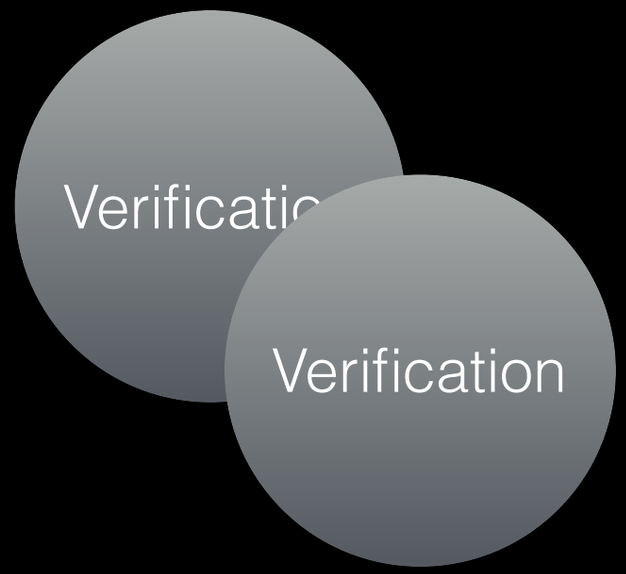
Language-Oriented Programming

matthias, racketeer

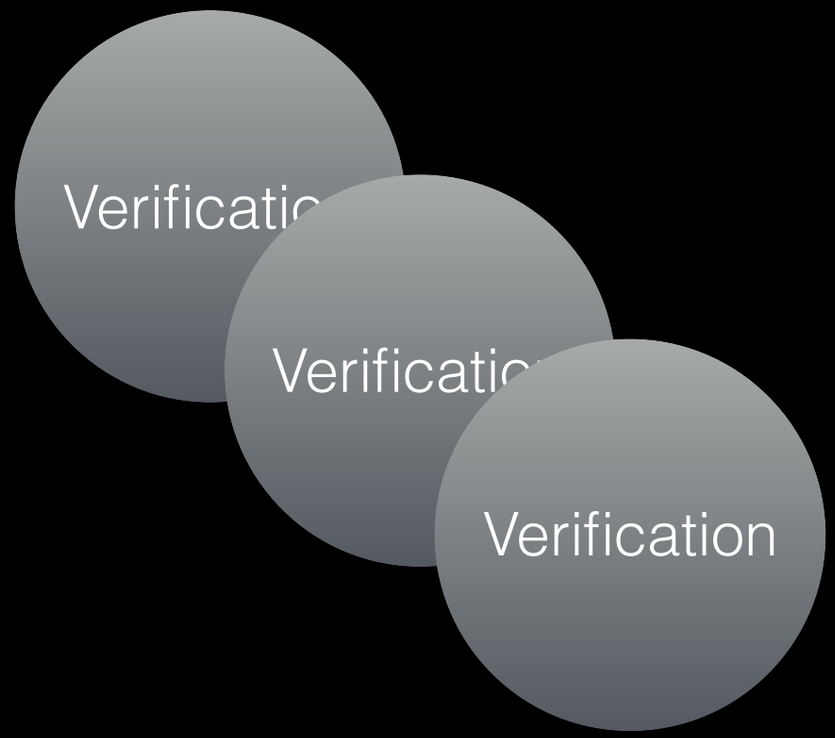
Academic Trends in FP

When You Attend a Programming Language Conference

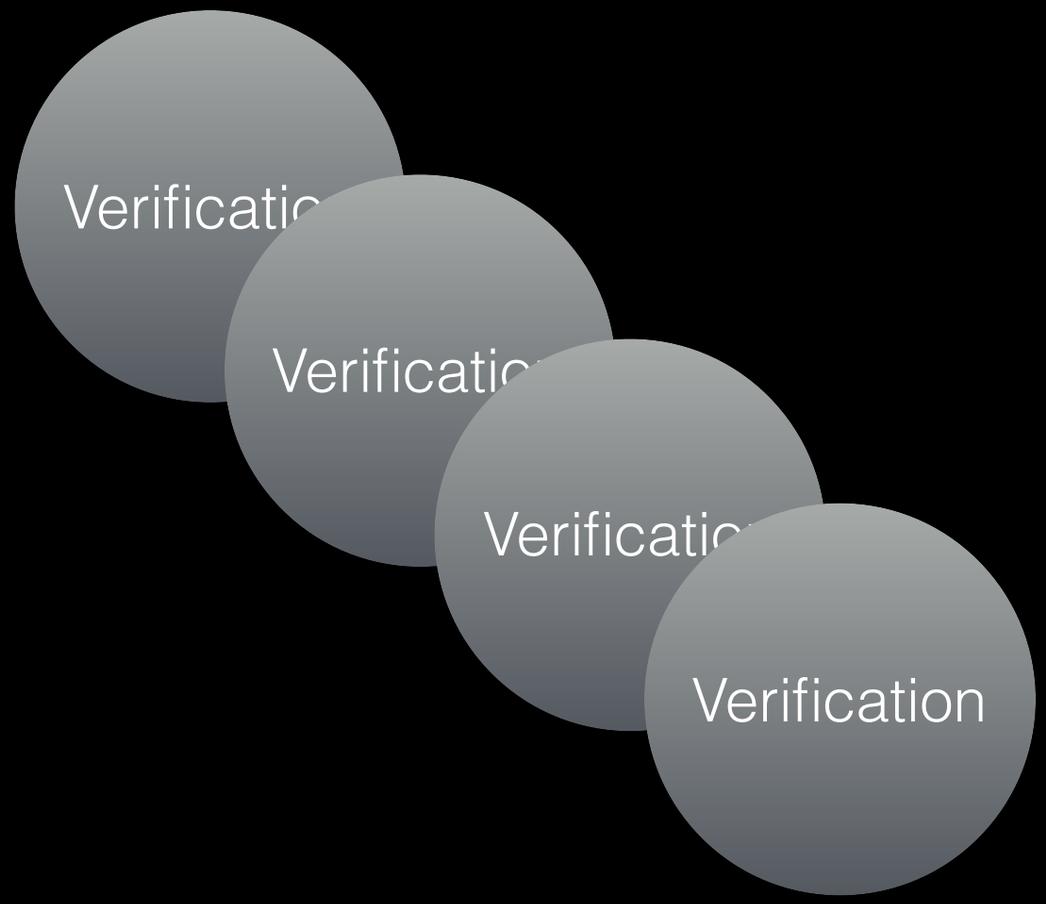
When You Attend a Programming Language Conference



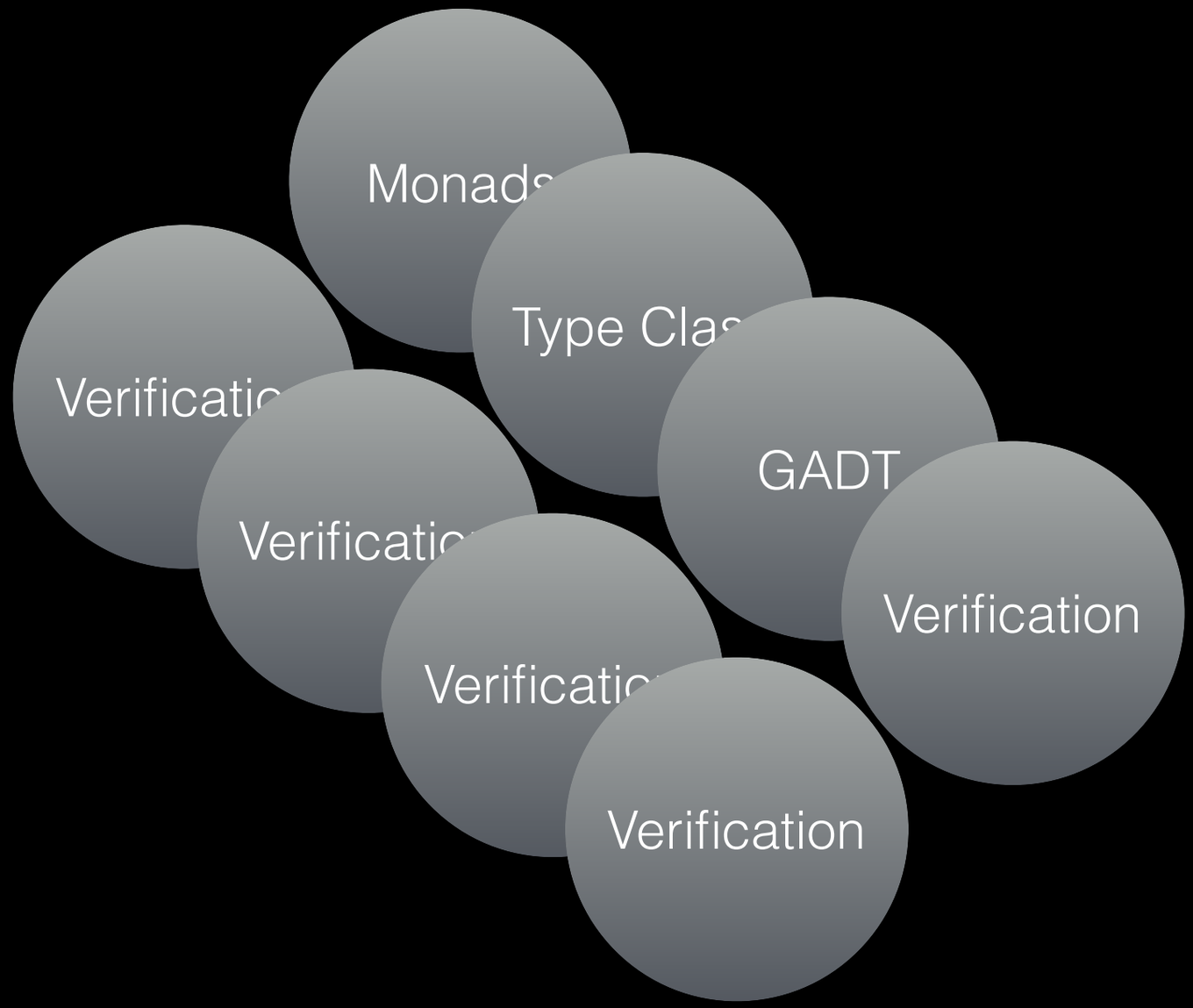
When You Attend a Programming Language Conference



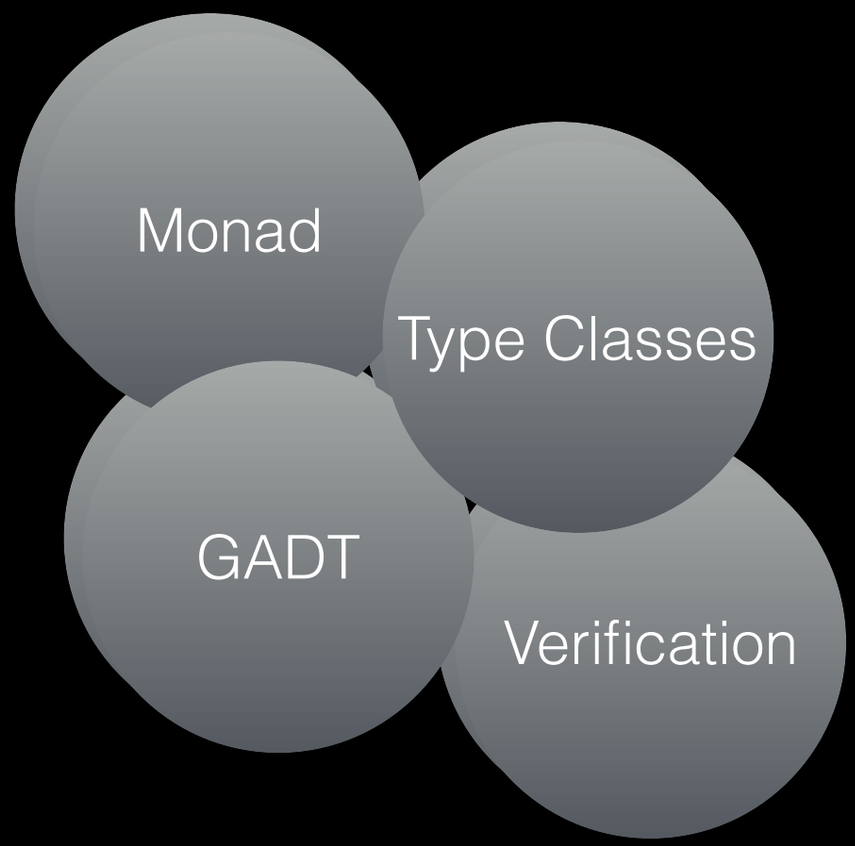
When You Attend a Programming Language Conference



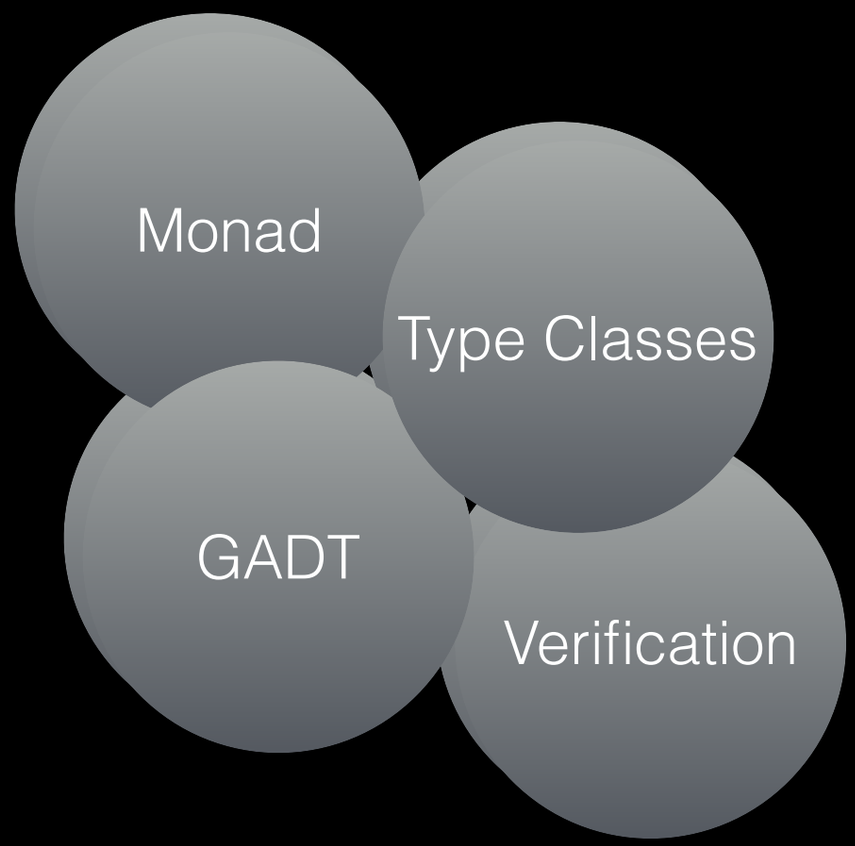
When You Attend a Programming Language Conference



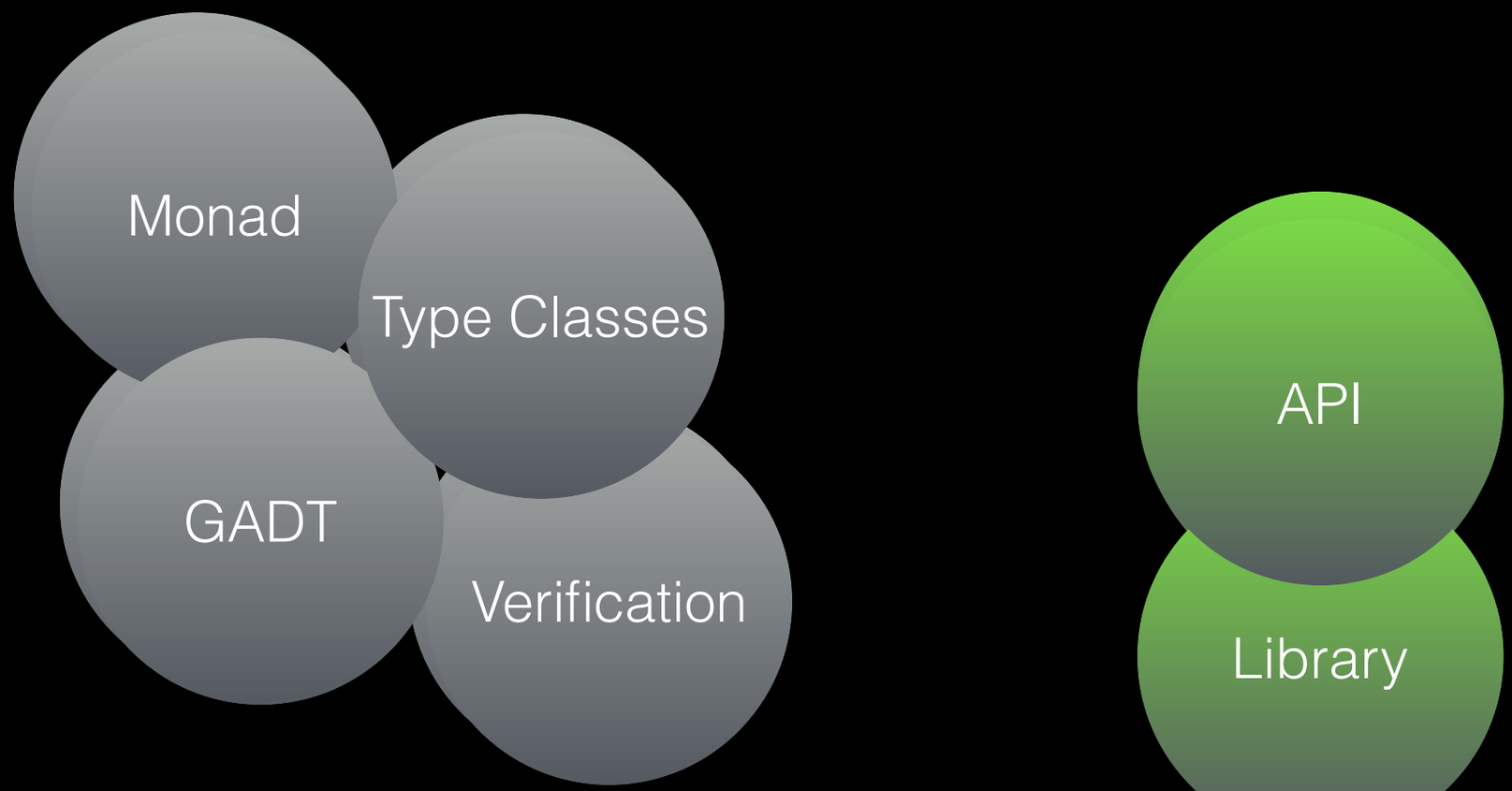
Languages Emerge as Abstractions to Conquer Complexity



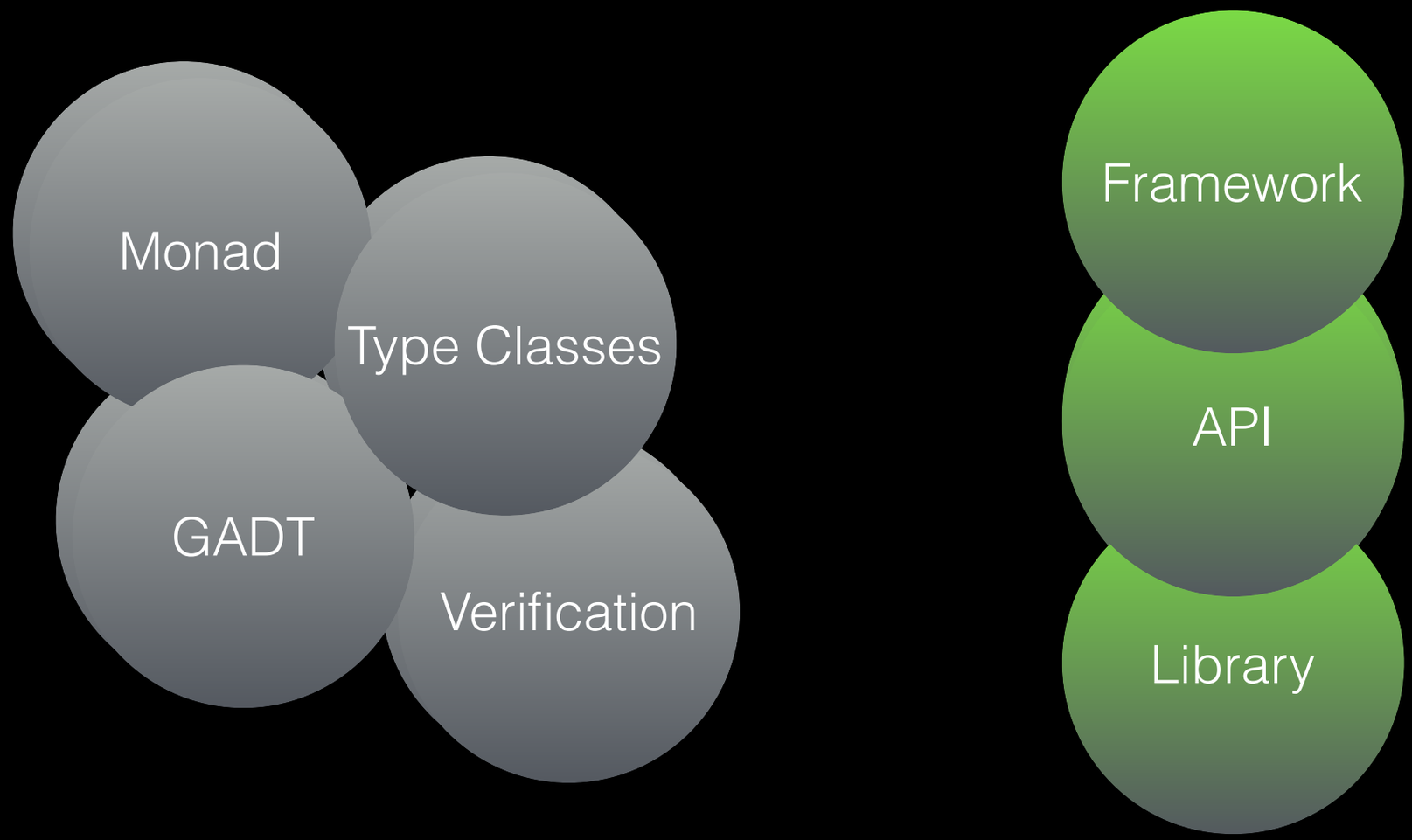
Languages Emerge as Abstractions to Conquer Complexity



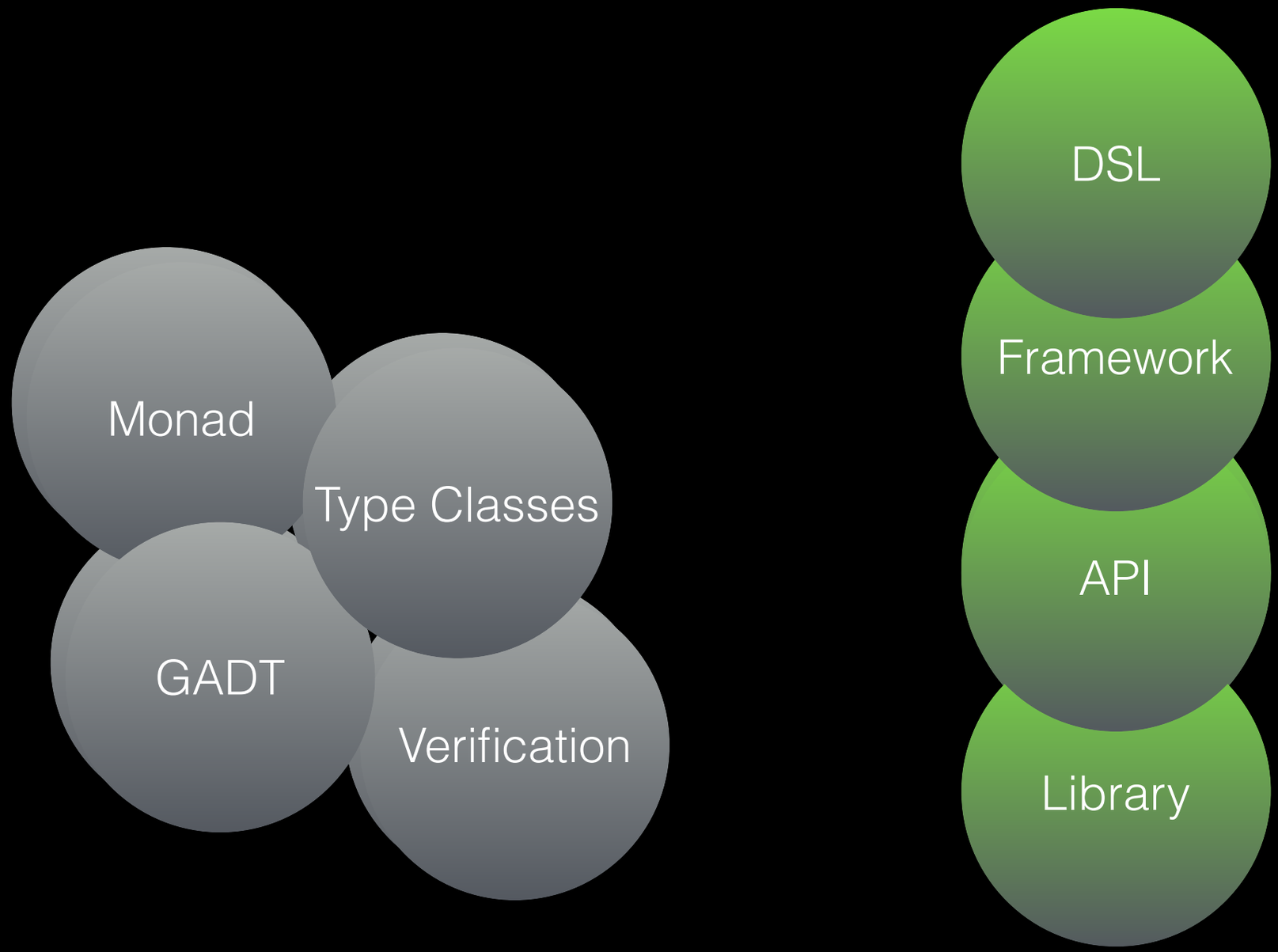
Languages Emerge as Abstractions to Conquer Complexity



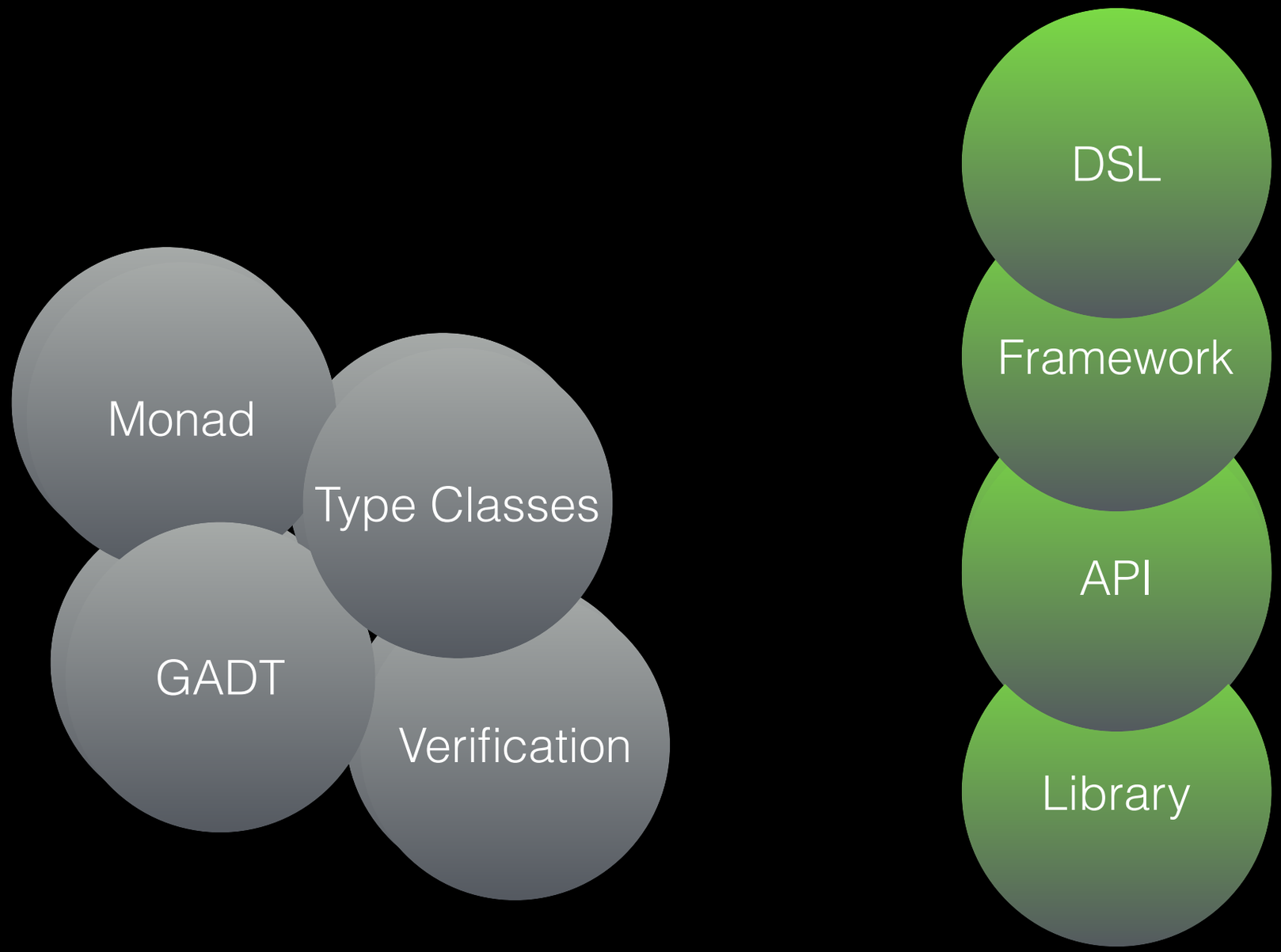
Languages Emerge as Abstractions to Conquer Complexity



Languages Emerge as Abstractions to Conquer Complexity



Languages Emerge as Abstractions to Conquer Complexity

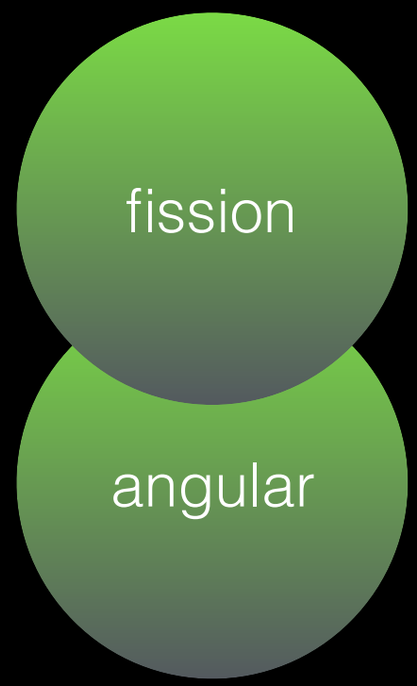


In Every Interface, There is a Language Trying to Get Out

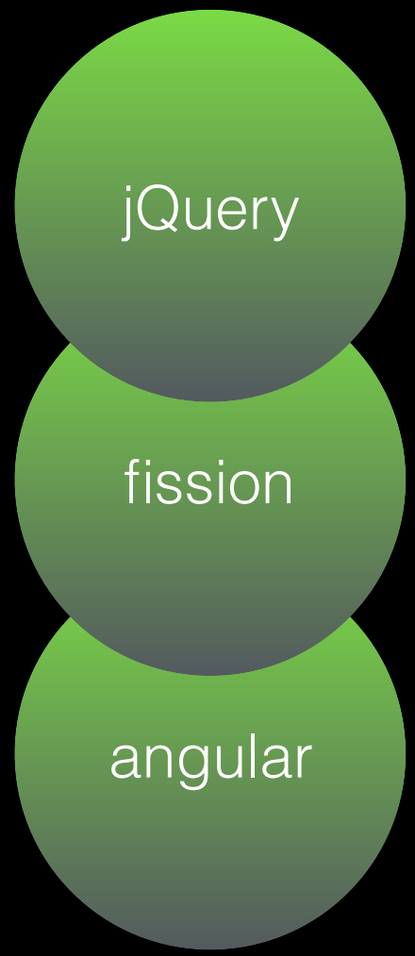
When You Are Working on the Web Stack, Especially the Front

When You Are Working on the Web Stack, Especially the Front

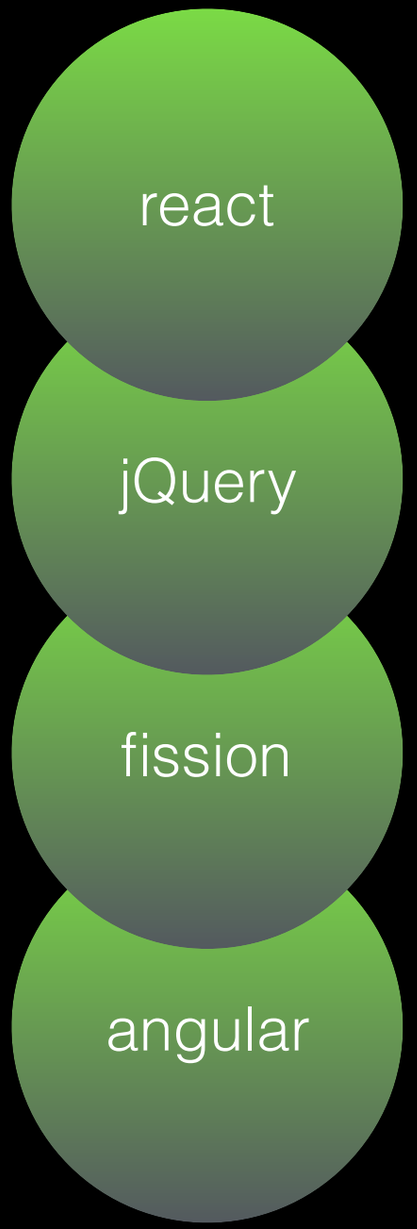
When You Are Working on the Web Stack, Especially the Front



When You Are Working on the Web Stack, Especially the Front

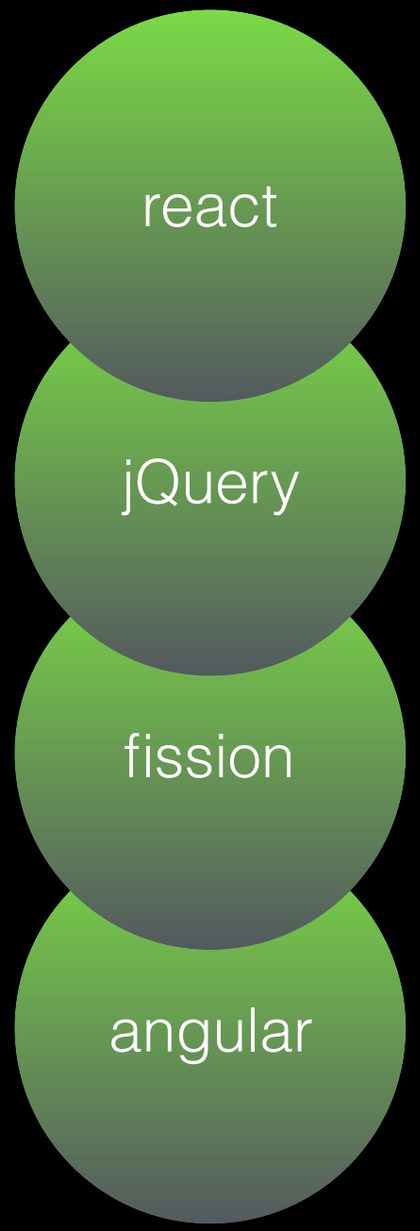


When You Are Working on the Web Stack, Especially the Front



When You Are Working on the Web Stack, Especially the Front

Developers program in a multi-lingual way, even though it's all in one PL.



1960s Structured Programming

1960s Structured Programming

1990s Object-Oriented Programming

1960s Structured Programming

1990s Object-Oriented Programming

2020s Language-Oriented Programming

1960s Structured Programming

1990s Object-Oriented Programming

2020s Language-Oriented Programming

formulate *all* solutions in problem-specific DSLS

1960s Structured Programming

1990s Object-Oriented Programming

2020s Language-Oriented Programming

formulate *all* solutions in problem-specific DSLS

make the DSLs if you have to

1960s Structured Programming

1990s Object-Oriented Programming

2020s Language-Oriented Programming

formulate *all* solutions in problem-specific DSLS

make the DSLs if you have to

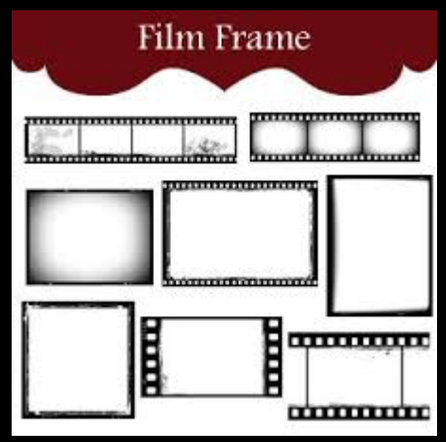
link these solutions into one multi-lingual system

Video, a Case Study

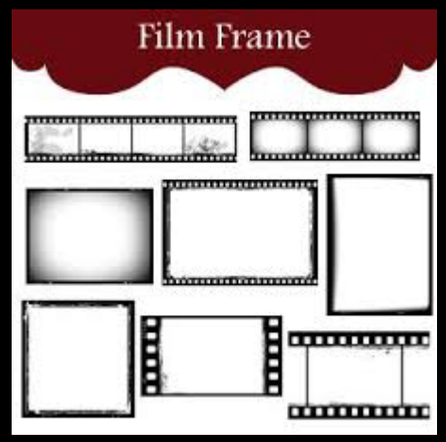
Andersen, Chang, Felleisen @ ICFP 2017

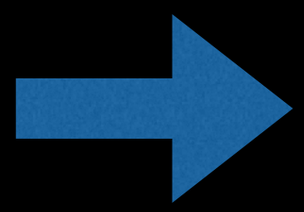
Benjy Montoya

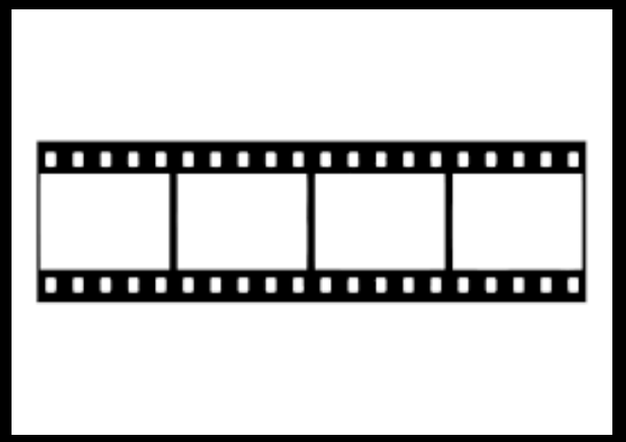
Benjy Montoya

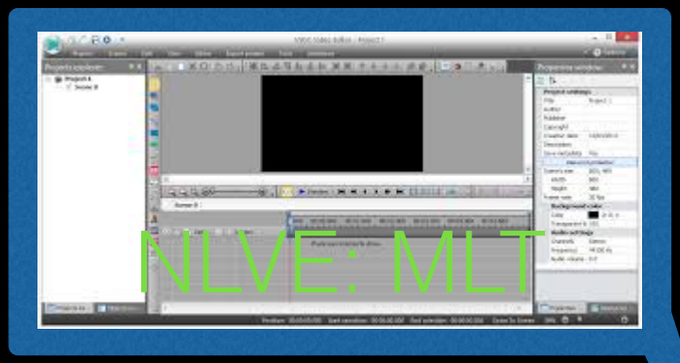


Benjy Montoya

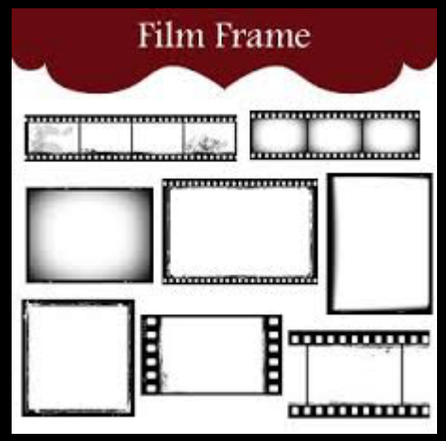


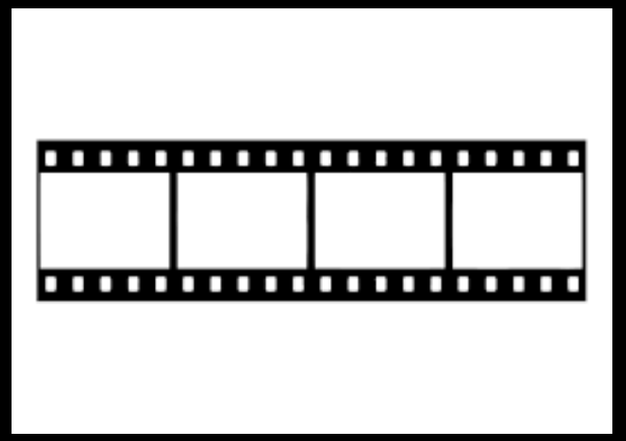




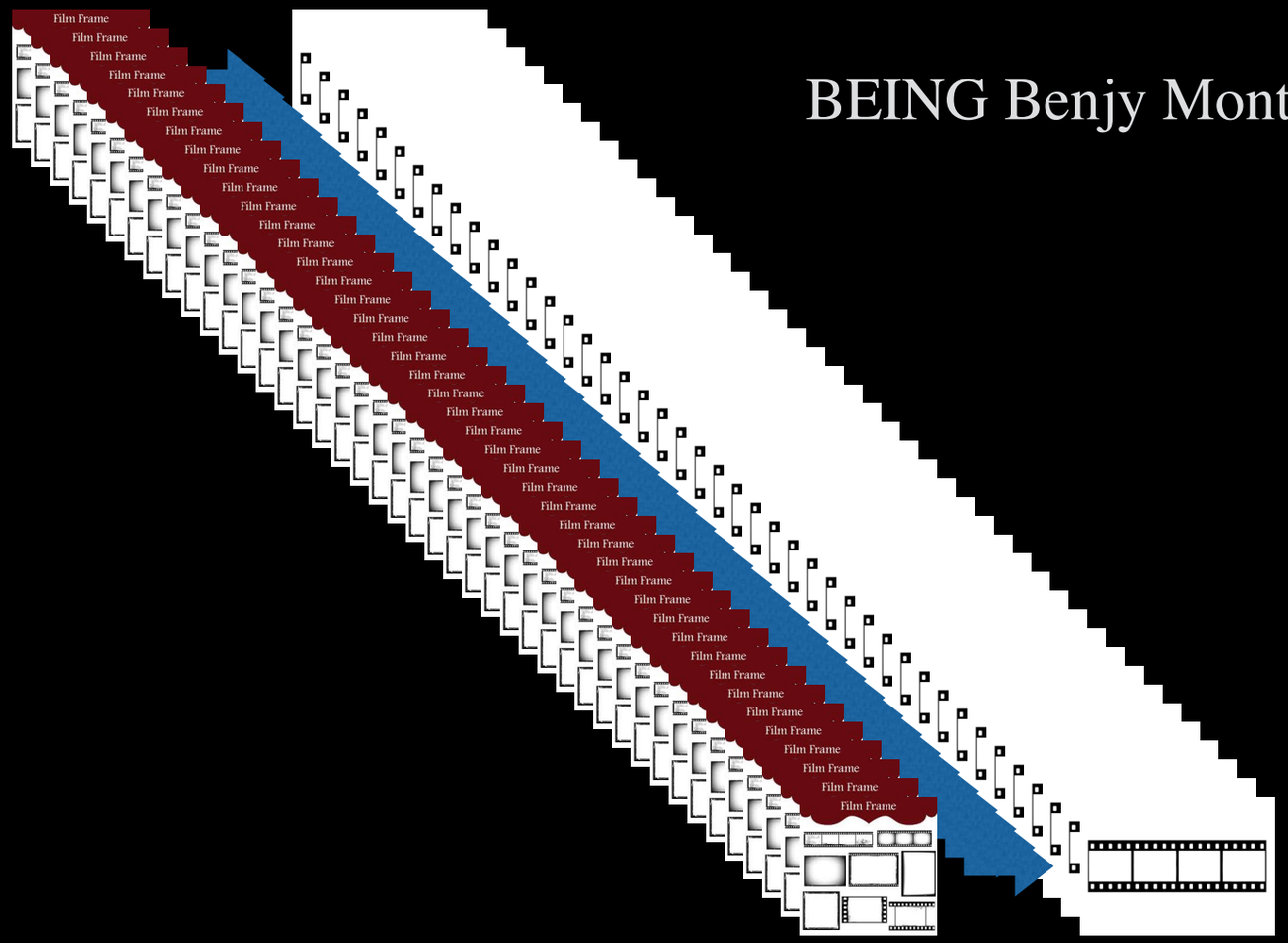


Benjy Montoya



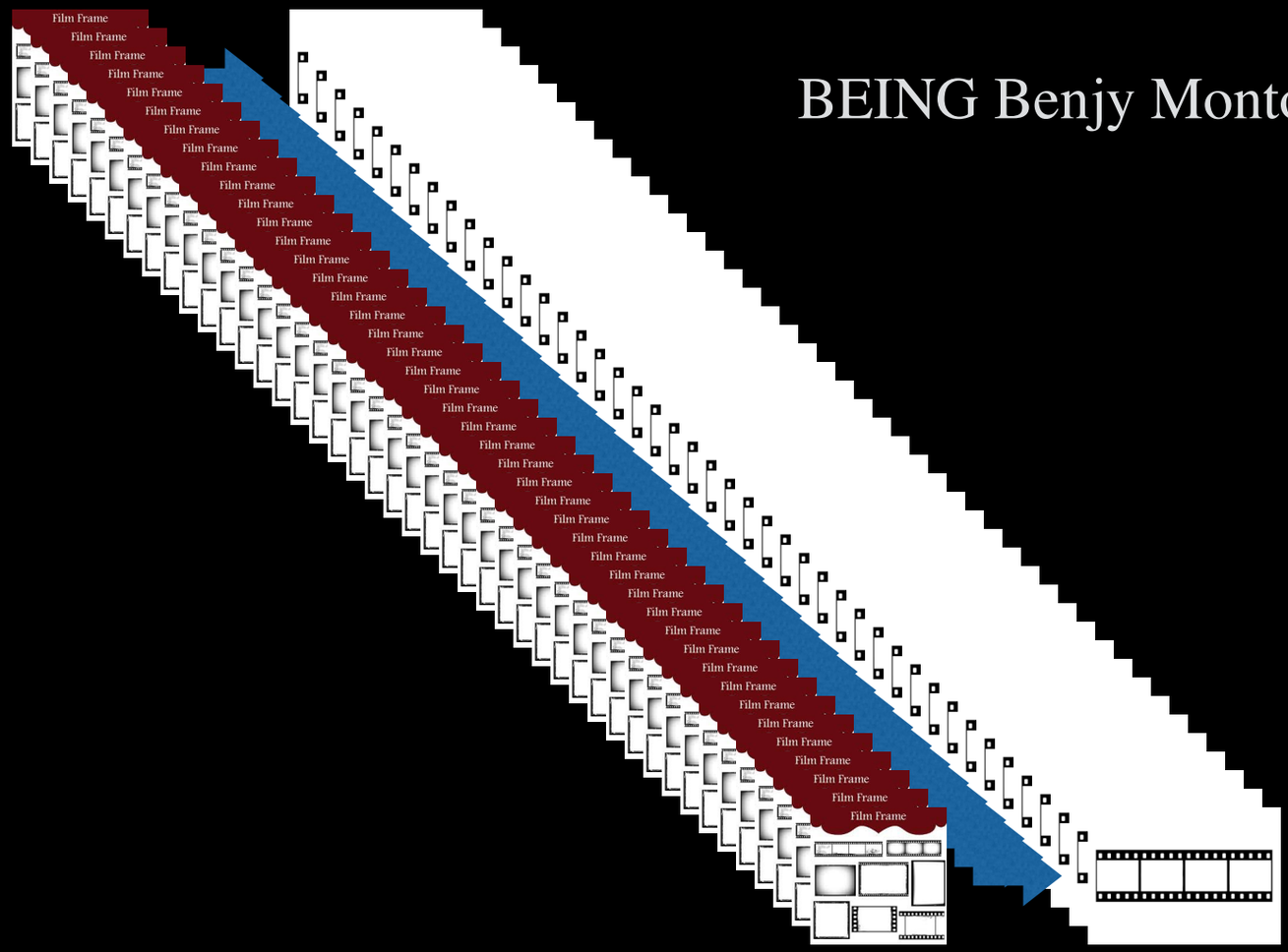


BEING Benjy Montoya



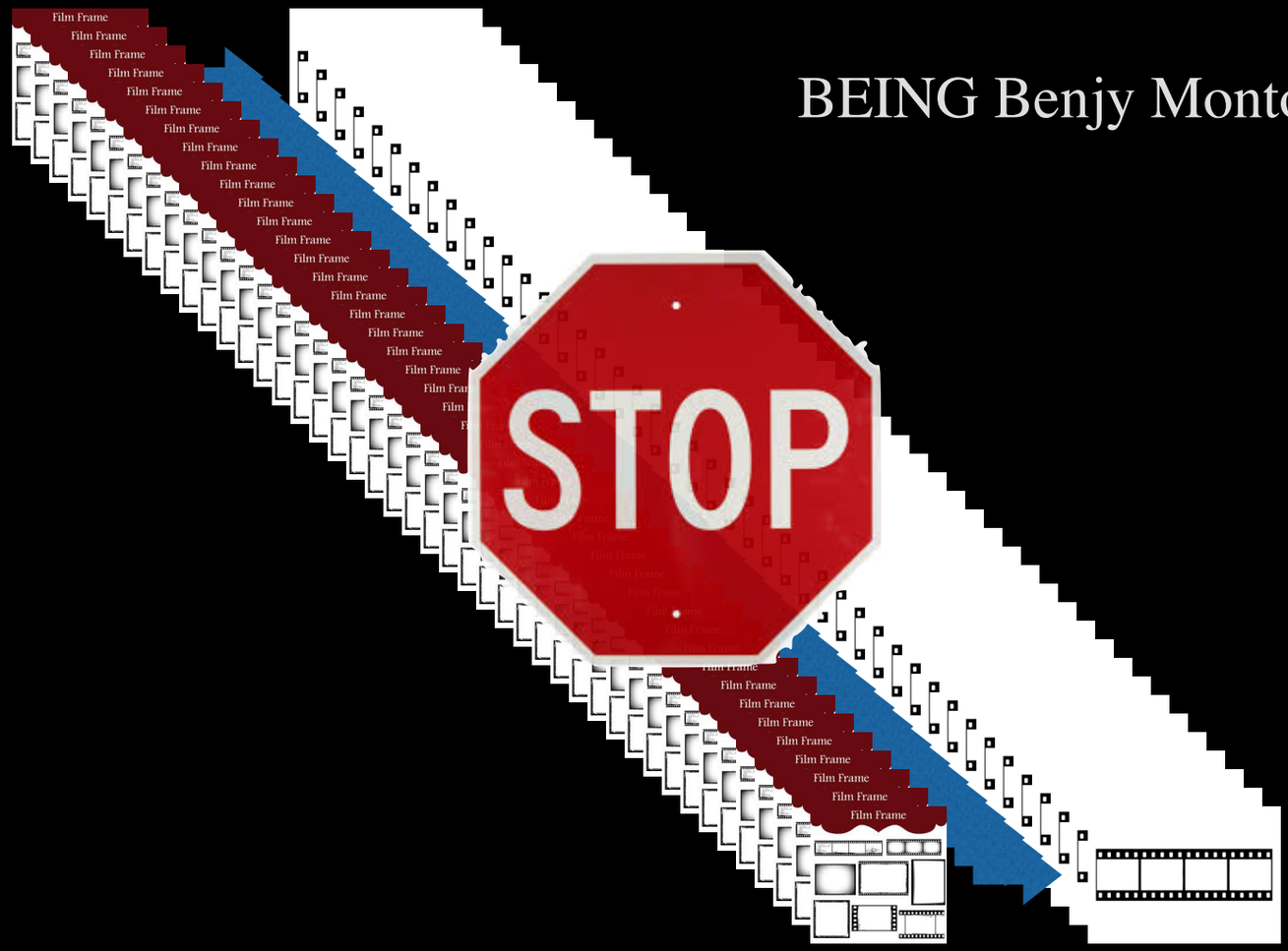
BEING Benjy Montoya

What's a programming language guy going to do?



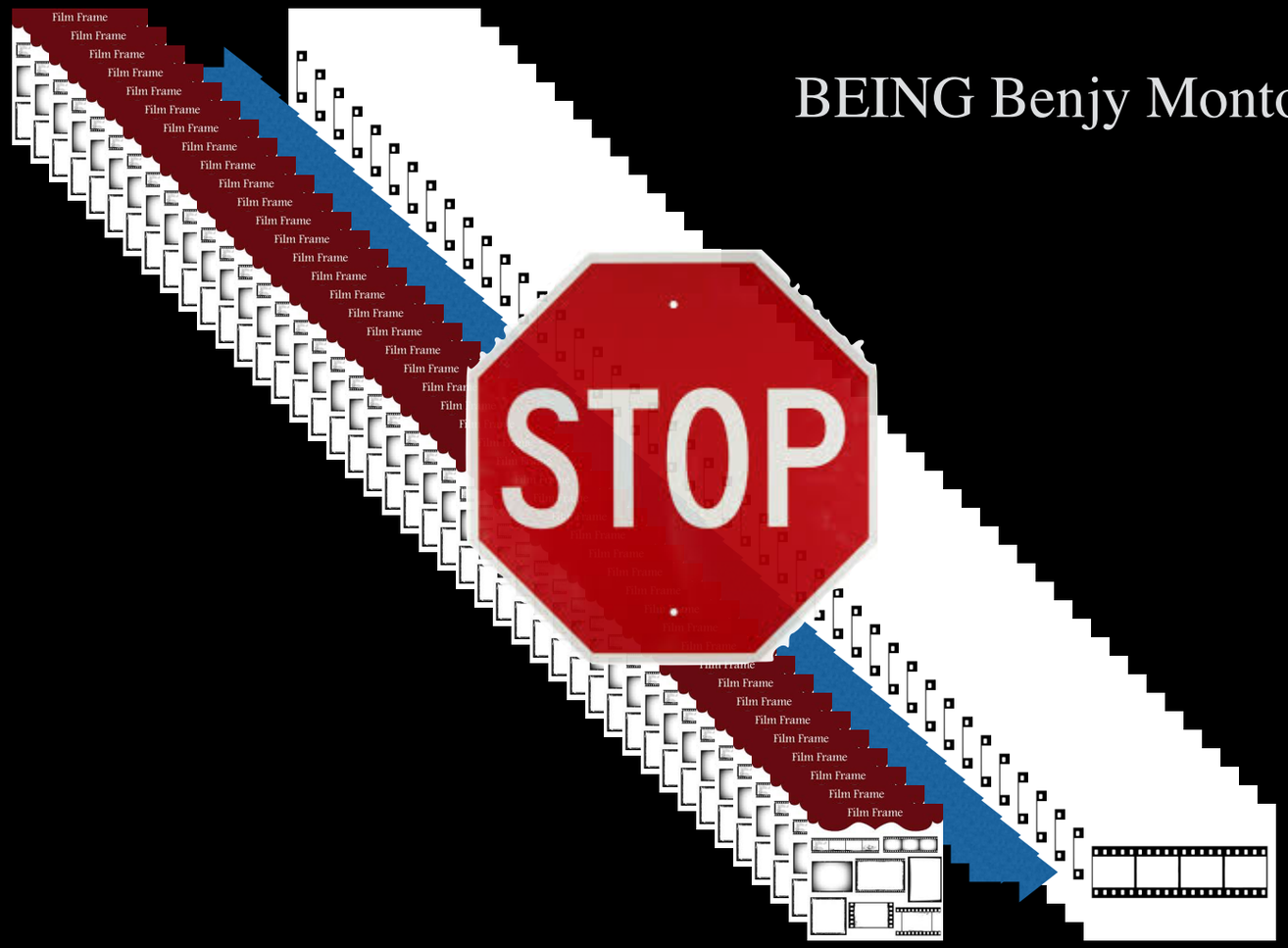
BEING Benjy Montoya

What's a programming language guy going to do?



BEING Benjy Montoya

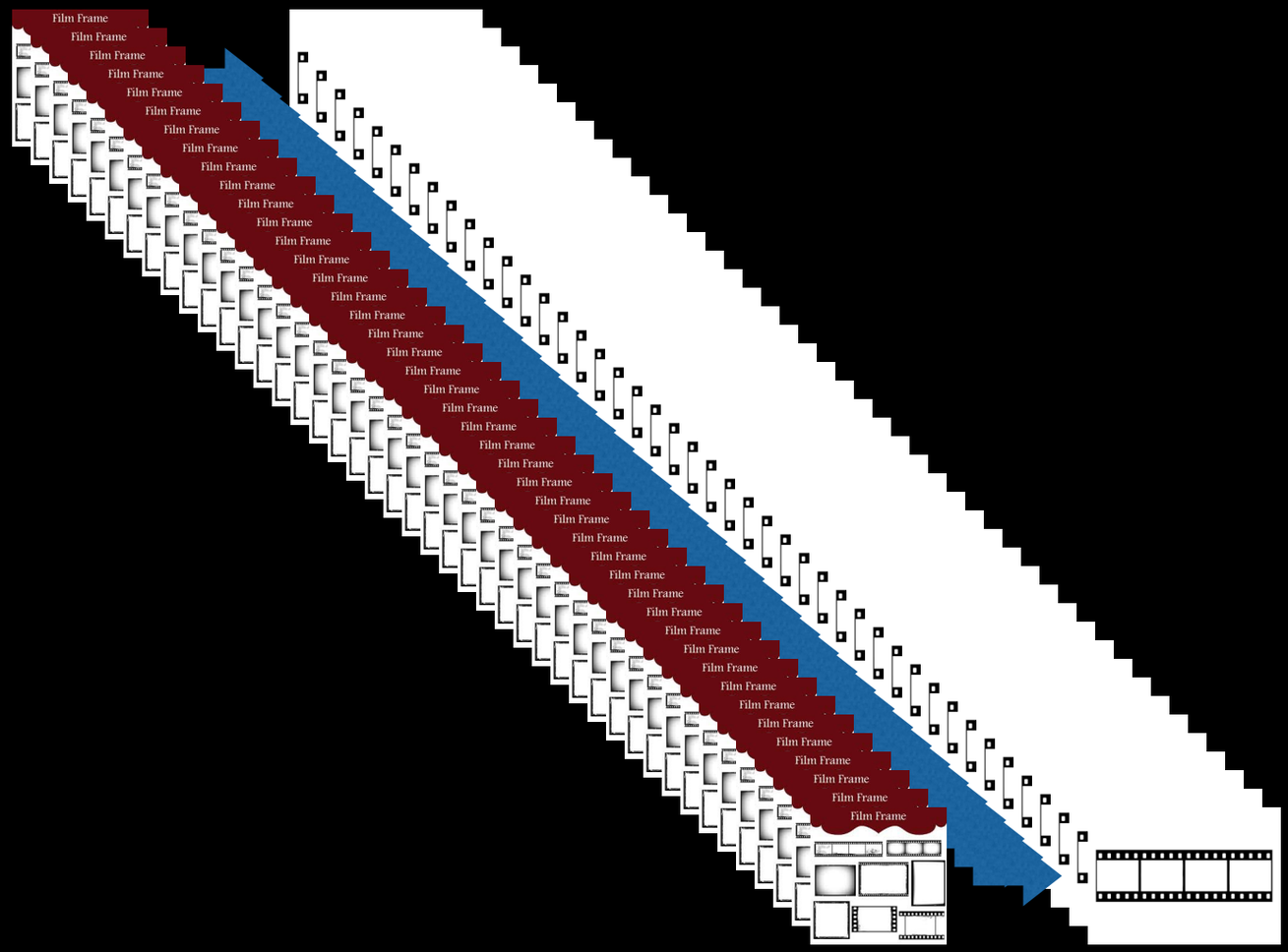
What's a programming language guy going to do?



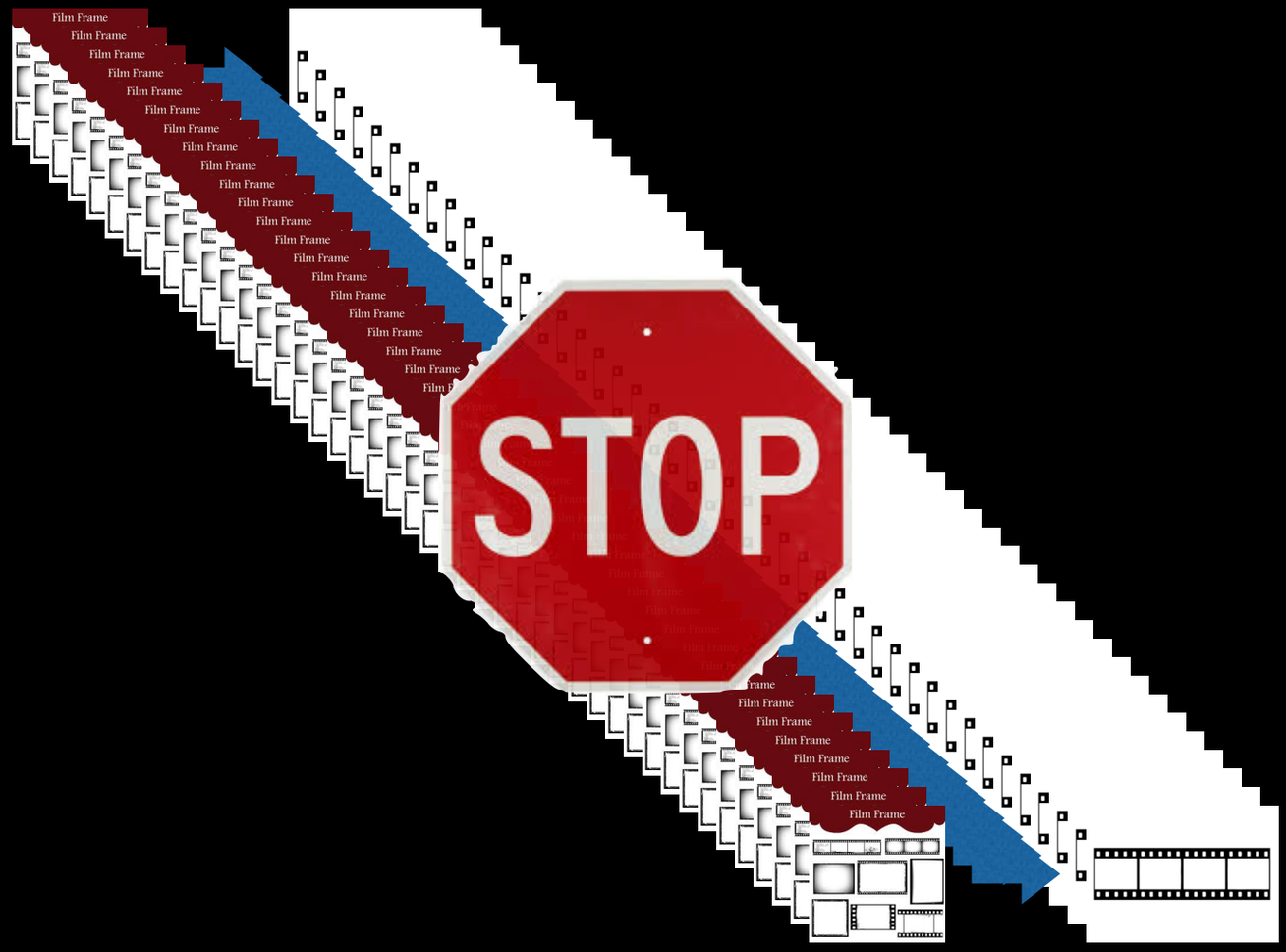
BEING Benjy Montoya

Tell me "script the MLT framework" was your first answer.

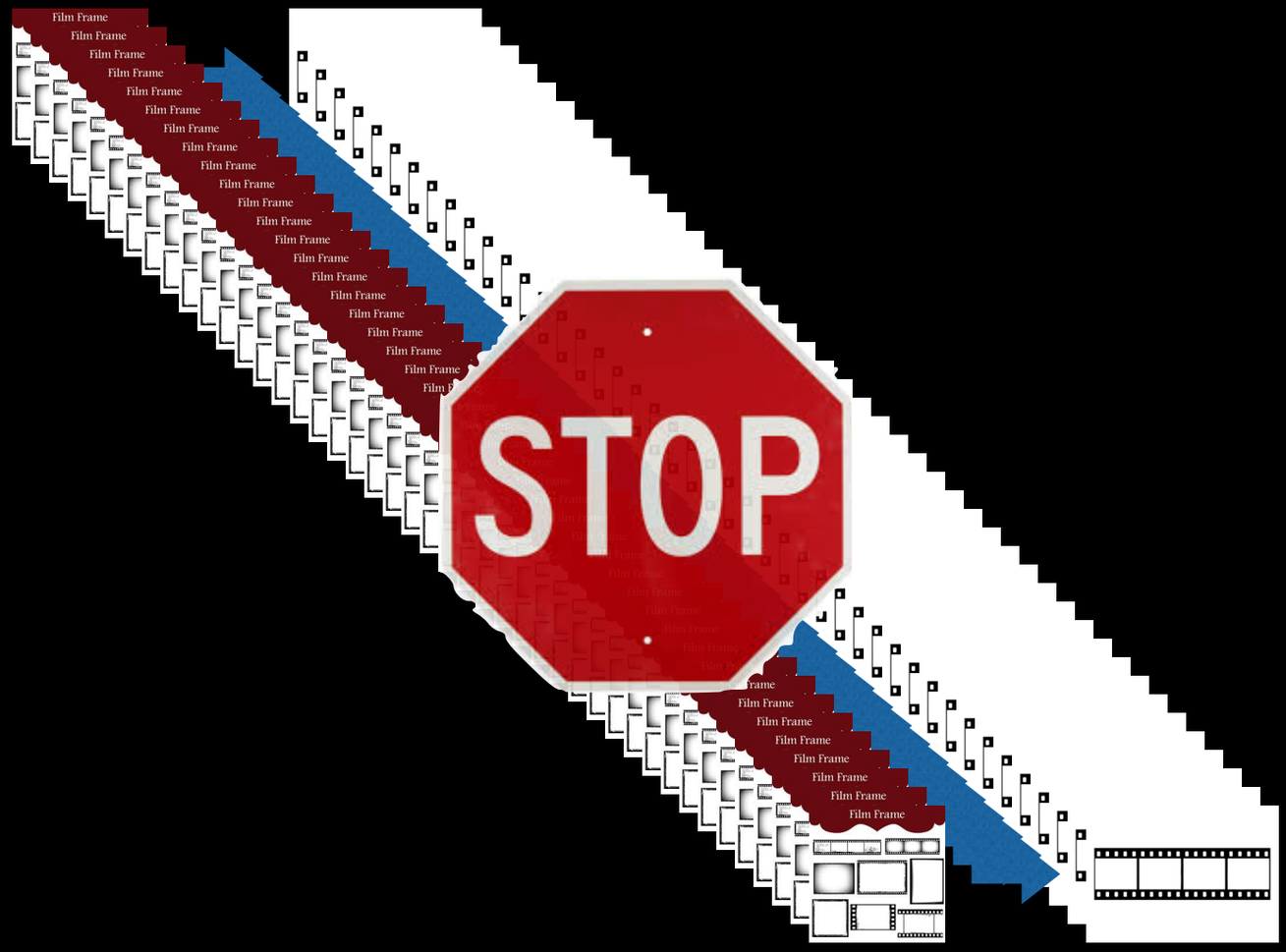
Scripting MLT is not something end users do. What now?



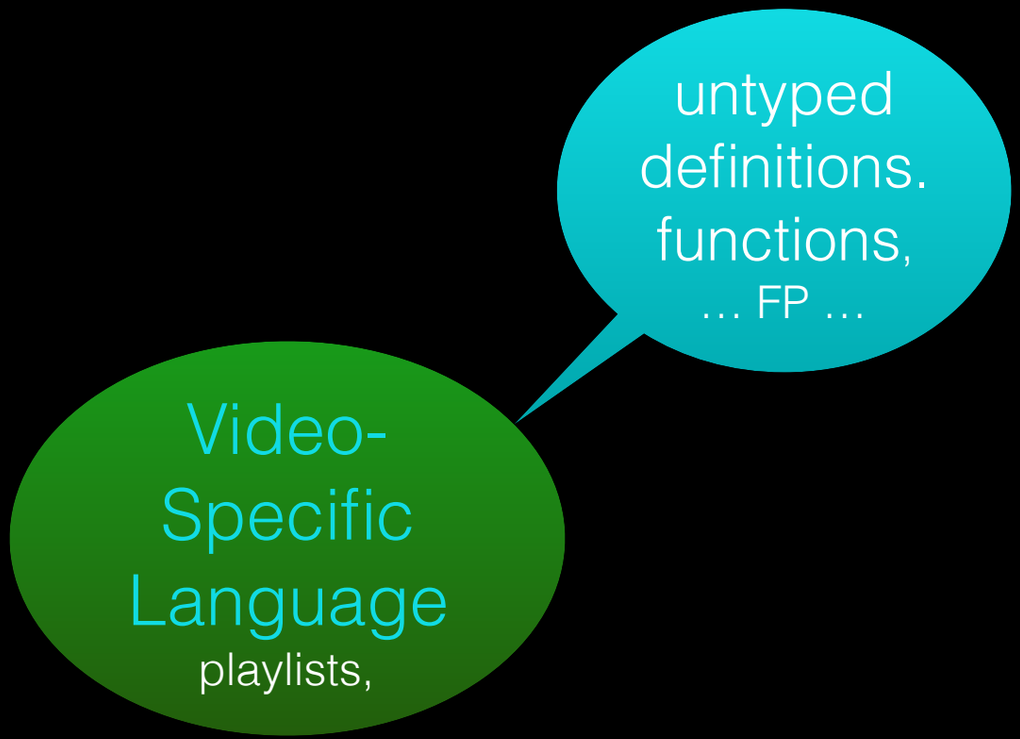
Scripting MLT is not something end users do. What now?



Scripting MLT is not something end users do. What now?



“Build a DSL for scripting MLT” was your answer. Right?


untyped
definitions.
functions,
... FP ...
Video-
Specific
Language
playlists,

Video-Specific Language

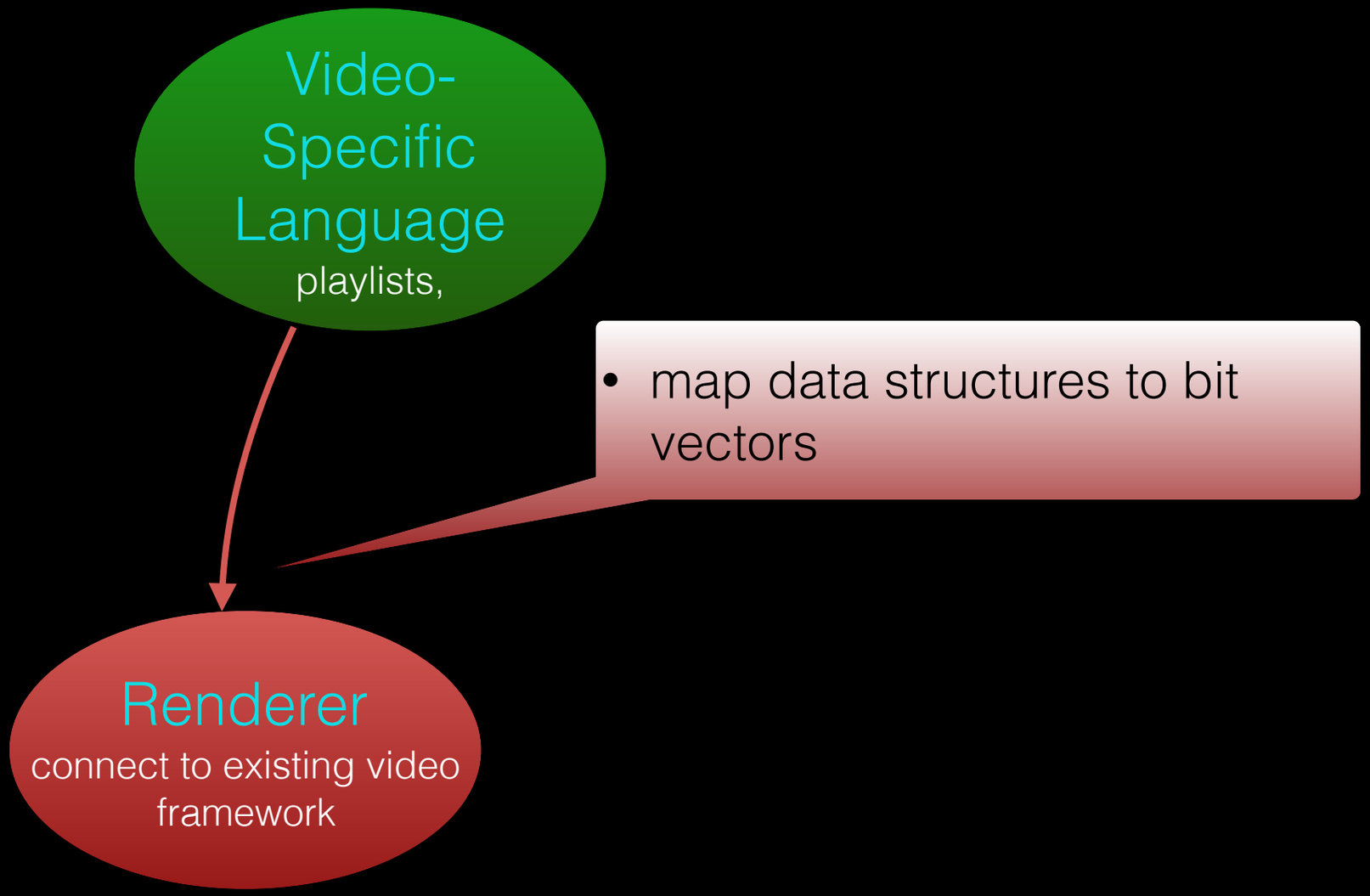
playlists,

- map data structures to bit vectors

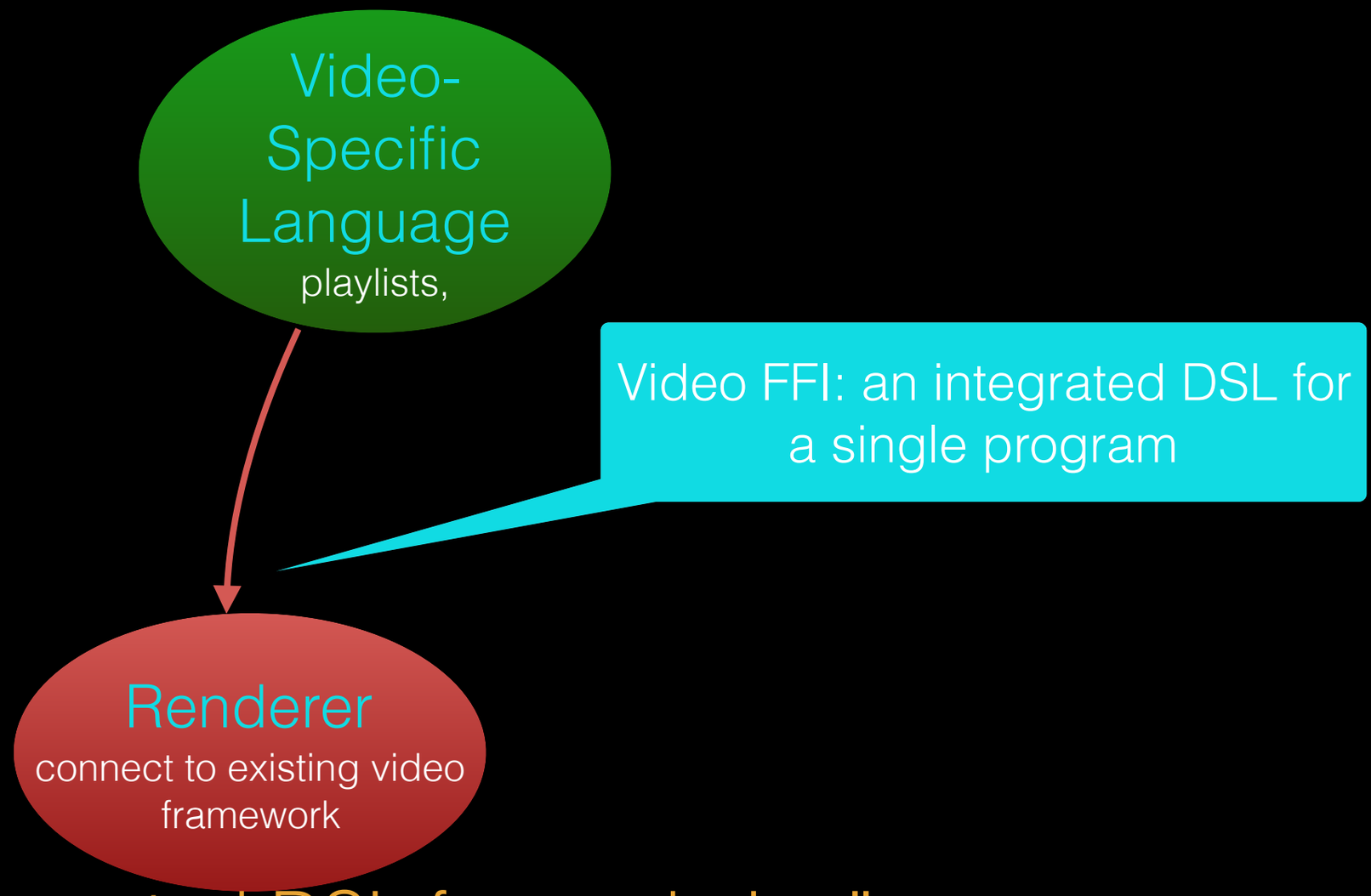
Renderer

connect to existing video framework

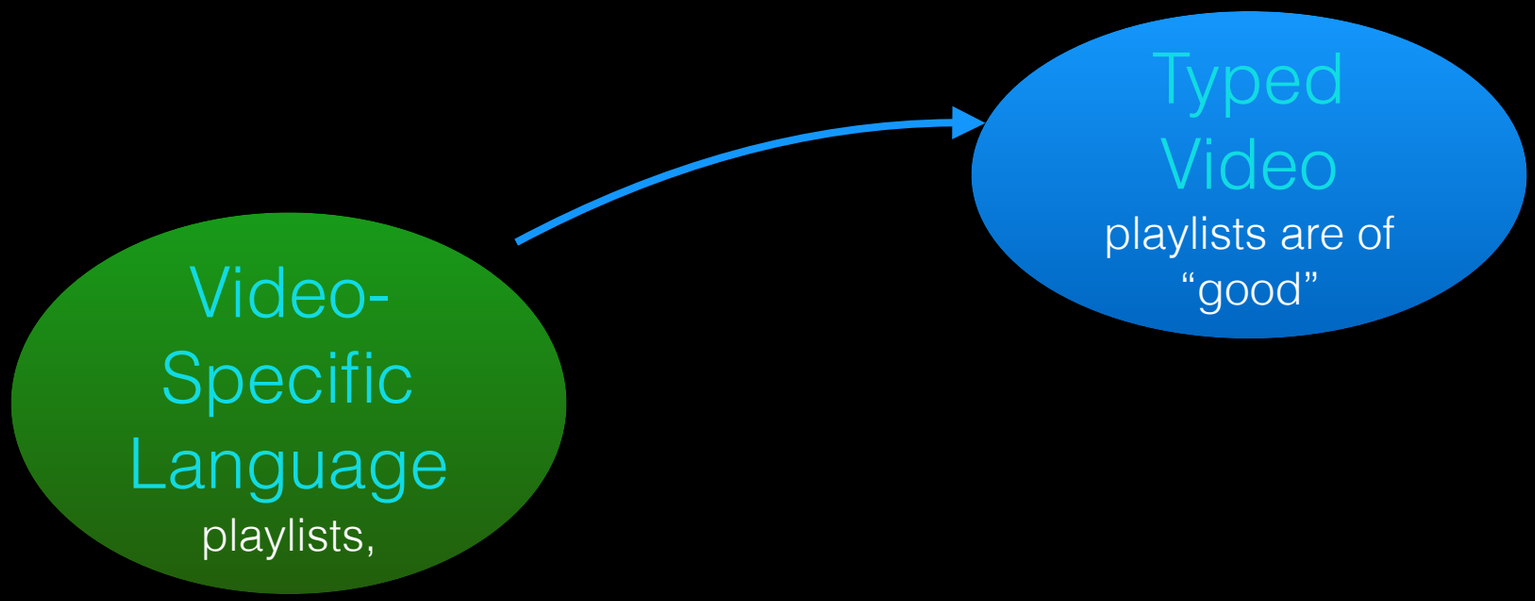
What is a programming language guy going to do?



What is a programming language guy going to do?



“Build a integrated DSL for rendering” was your answer. Right?


Video-
Specific
Language
playlists,
Typed
Video
playlists are of
“good”

Video-Specific Language

playlists,

Typed Video

playlists are of "good"

COLOR-N

$$\frac{\Gamma \vdash \texttt{e} : \texttt{String}}{\Gamma \vdash \texttt{(color e \#:length n)} : \texttt{(Producer n)}}$$

CLIP

$$\frac{\Gamma \vdash \texttt{f} : \texttt{File} \quad |\texttt{f}| = \texttt{n}}{\Gamma \vdash \texttt{(clip f)} : \texttt{(Producer n)}}$$

COLOR

$$\frac{\Gamma \vdash \texttt{e} : \texttt{String}}{\Gamma \vdash \texttt{(color e)} : \texttt{Producer}}$$

PLAYLIST

$$\frac{\Gamma \vdash \texttt{p/t} : \tau \quad \tau <: \texttt{(Producer n)} \text{ or } \tau <: \texttt{(Transition m)} \quad \ldots}{\Gamma \vdash \texttt{(playlist p/t ...)} : \texttt{(Producer (- (+ n ...) (+ m ...)))}}$$

What is a programming language guy going to do?

Video-Specific Language

playlists,

Typed Video

playlists are of "good"

COLOR-N

$$\frac{\Gamma \vdash \texttt{e} : \texttt{String}}{\Gamma \vdash \texttt{(color e \#:length n)} : \texttt{(Producer n)}}$$

CLIP

$$\frac{\Gamma \vdash \texttt{f} : \texttt{File} \quad \texttt{|f|} = \texttt{n}}{\Gamma \vdash \texttt{(clip f)} : \texttt{(Producer n)}}$$

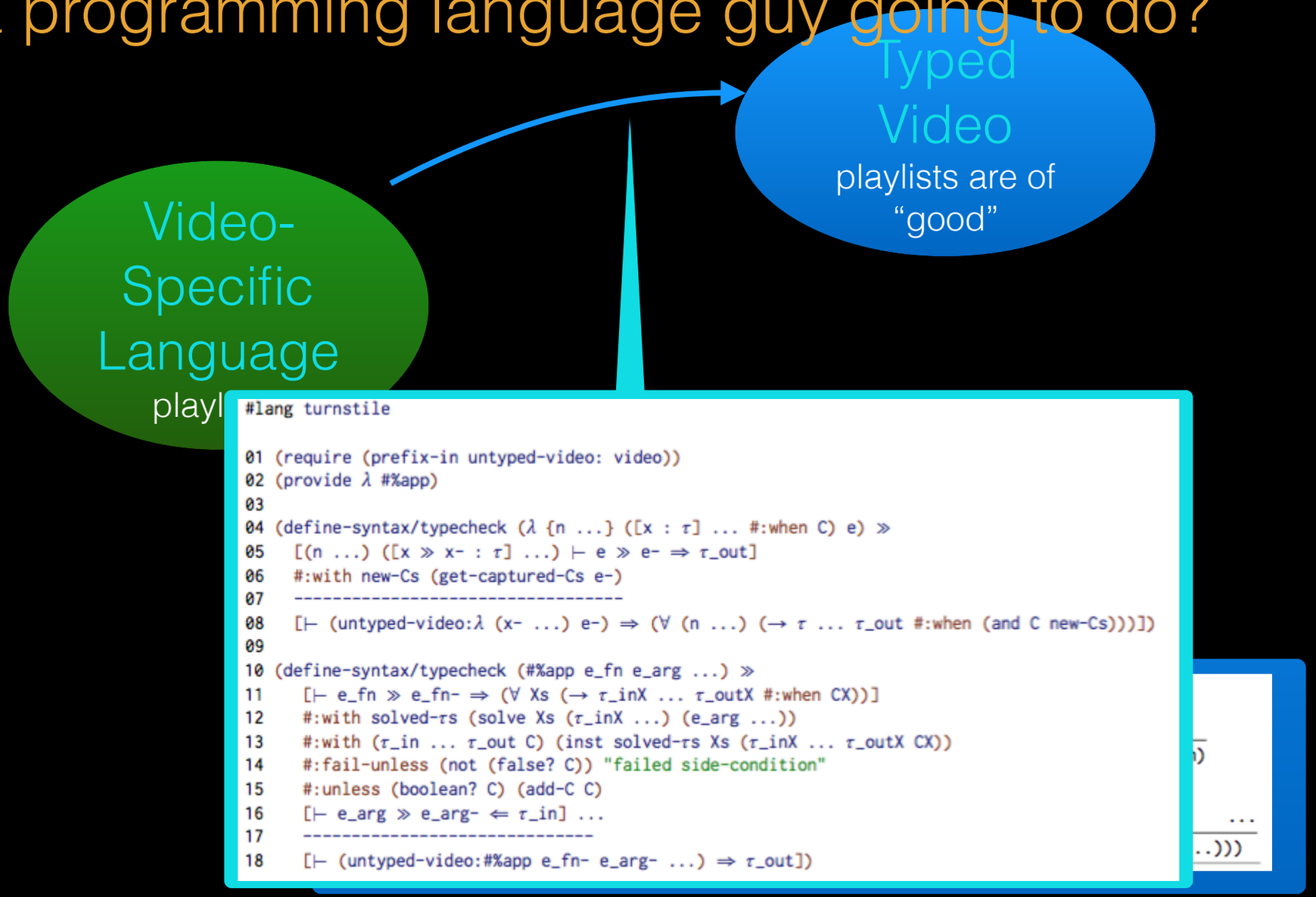
COLOR

$$\frac{\Gamma \vdash \texttt{e} : \texttt{String}}{\Gamma \vdash \texttt{(color e)} : \texttt{Producer}}$$

PLAYLIST

$$\frac{\Gamma \vdash \texttt{p/t} : \tau \quad \tau \texttt{ <: (Producer n) or } \tau \texttt{ <: (Transition m)} \quad \ldots}{\Gamma \vdash \texttt{(playlist p/t ...)} : \texttt{(Producer (- (+ n ...) (+ m ...)))}}$$

What is a programming language guy going to do?



“Build a integrated DSL for types” was your answer. Right?

the core DSL

Video-Specific Language

playlists,

Typed Video

playlists are of "good"

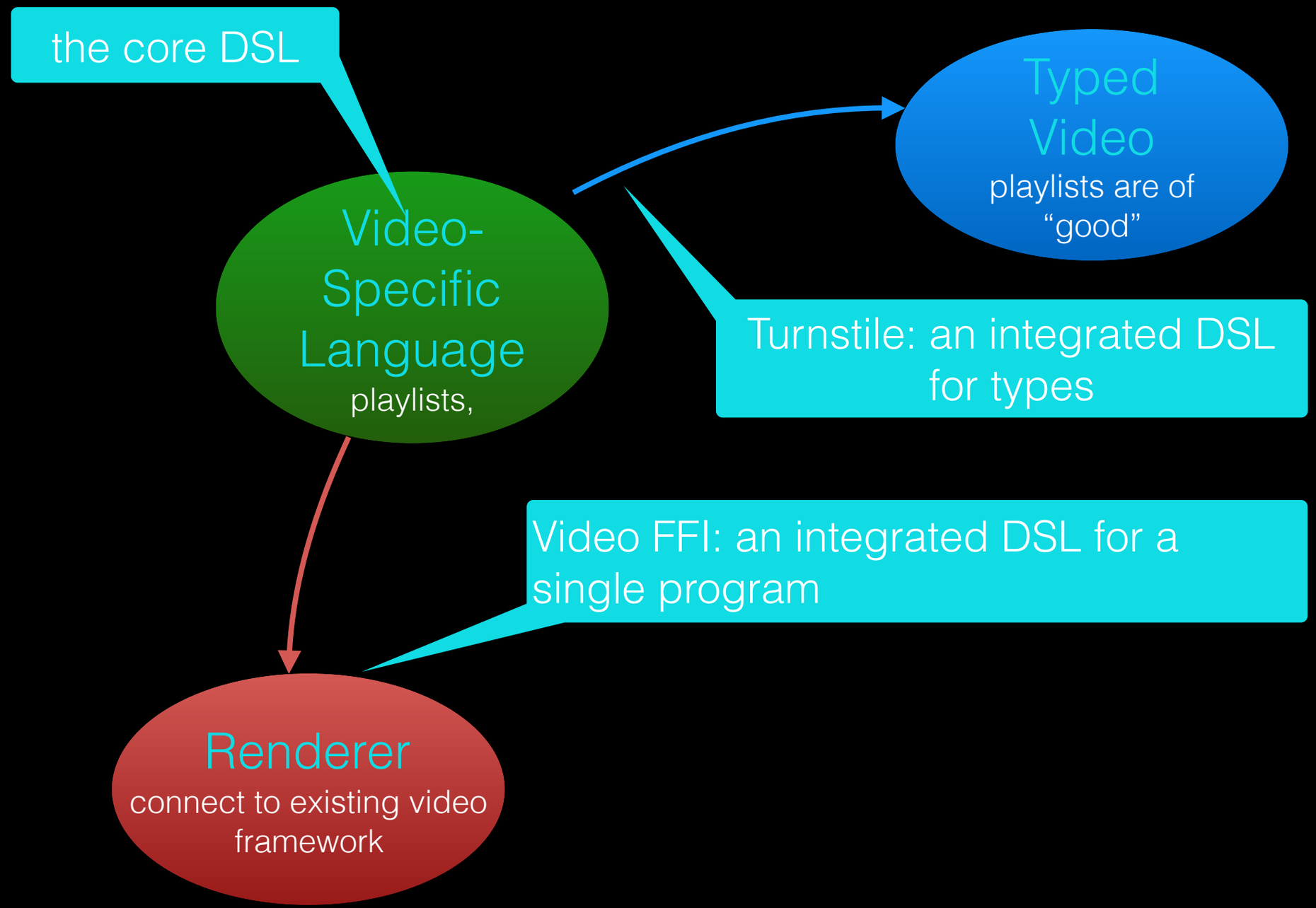
Turnstile: an integrated DSL for types

Video FFI: an integrated DSL for a single program

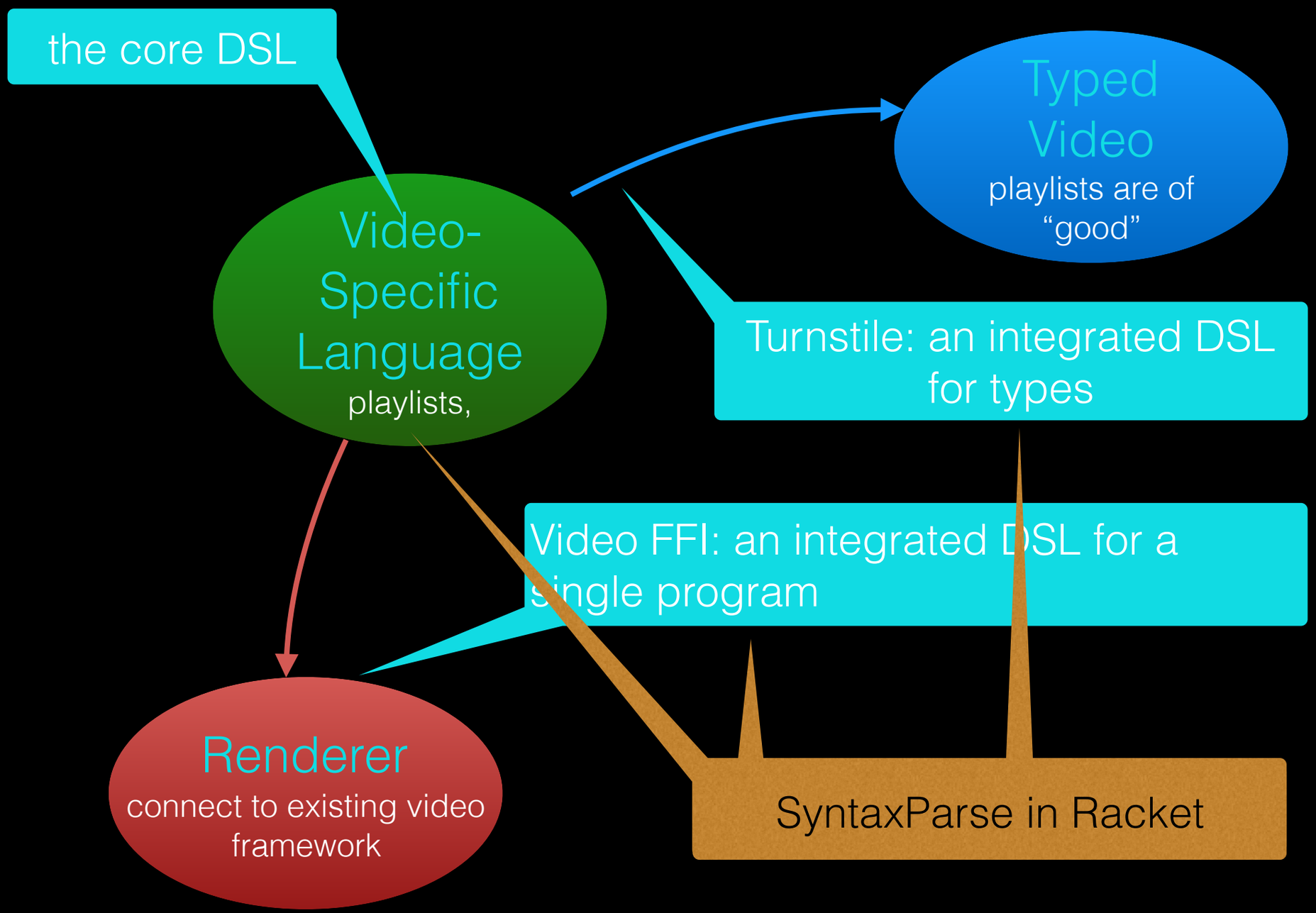
Renderer

connect to existing video framework

What is a programming language guy going to do?



What is a programming language guy going to do?



"Build an integrated DSL for building integrated DSLs".

the core DSL

Typed Video

playlists are of "good"

Video-Specific Language

playlists,

Turnstile: an integrated DSL for types

Video FFI: an integrated DSL for a single program

Renderer

connect to existing video framework

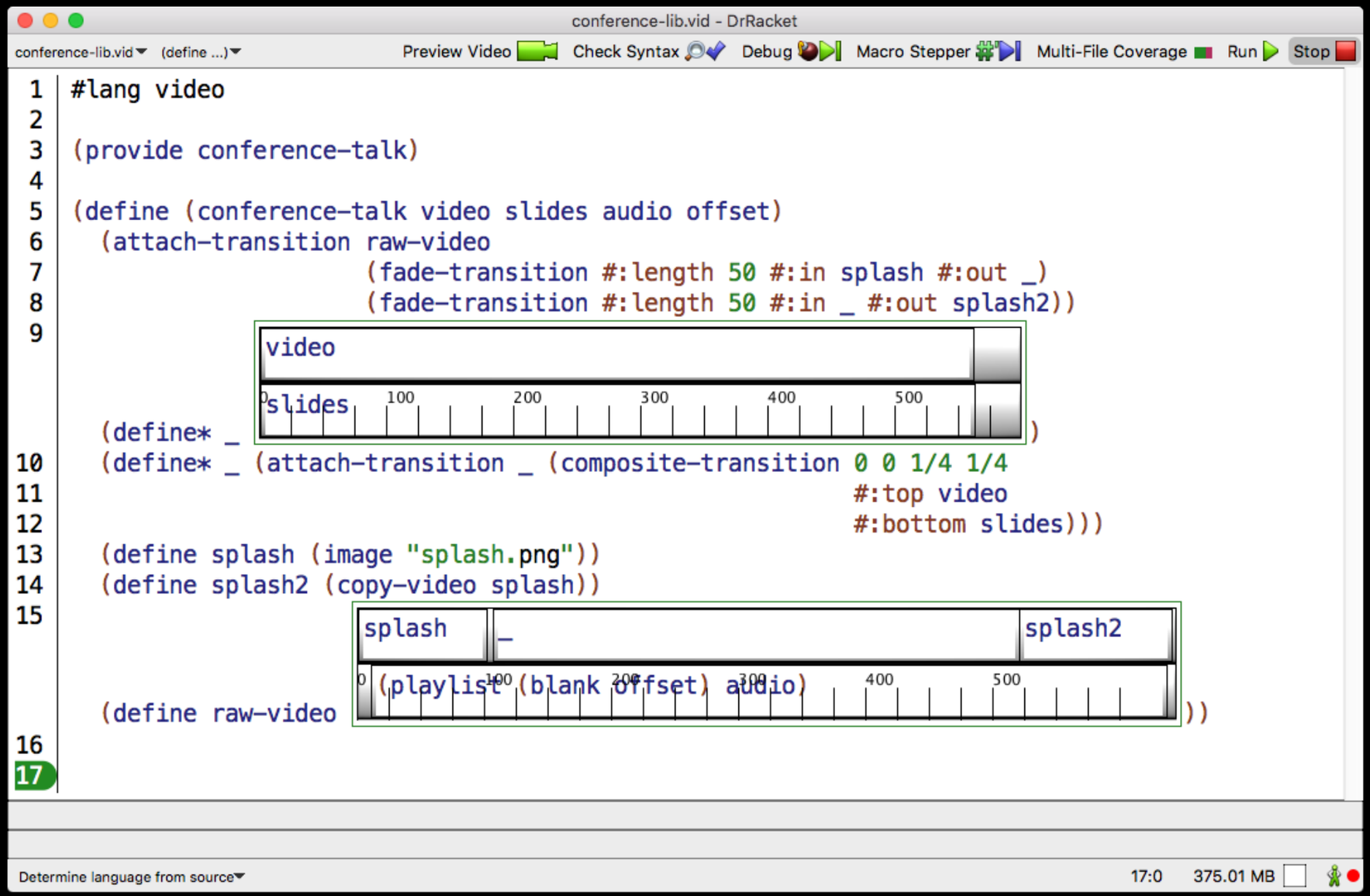
SyntaxParse in Racket

Racket

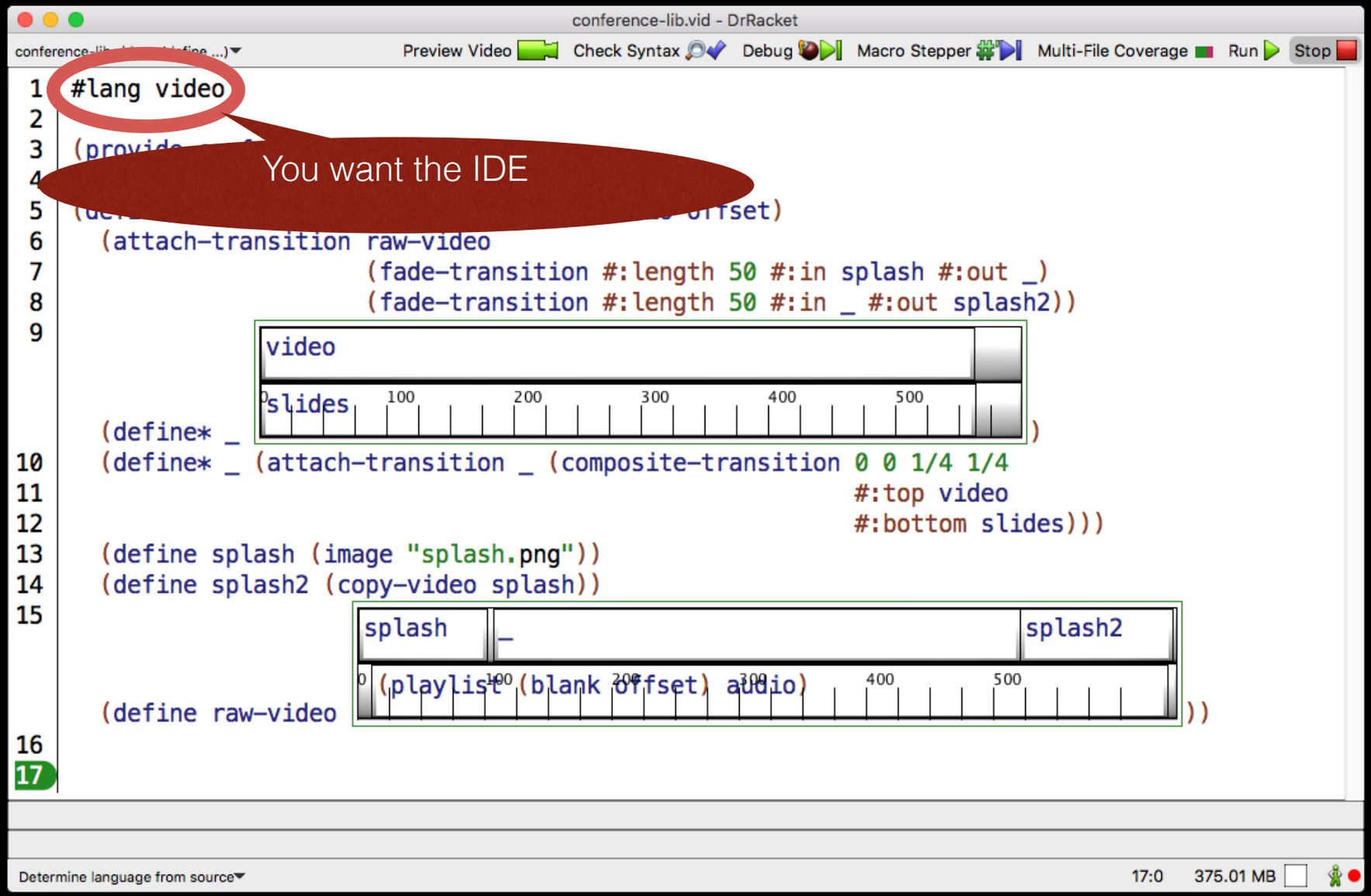
an LoP programming language

Doesn’t a Programming Language Need an IDE in this Day and Age?

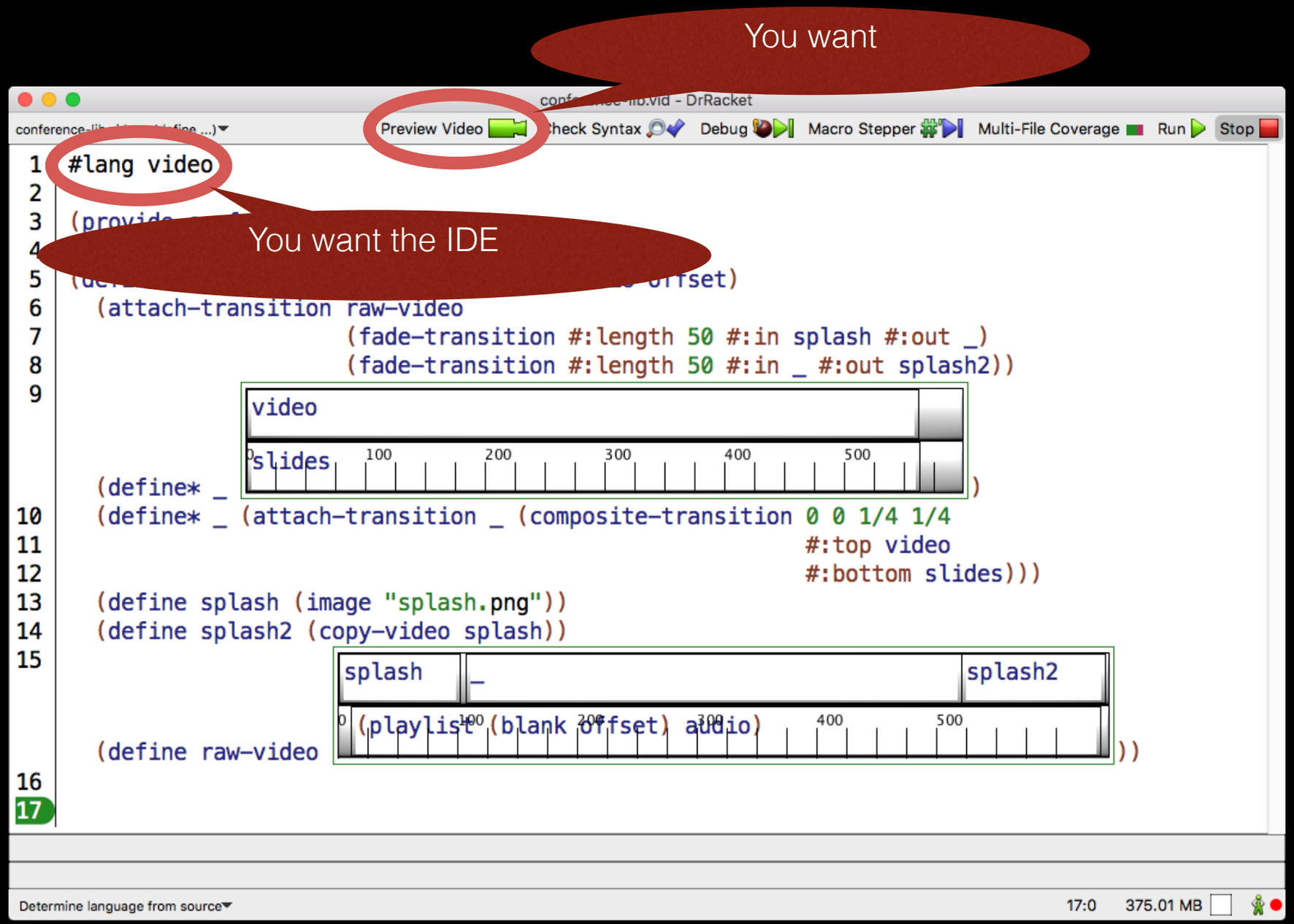
Doesn’t a Programming Language Need an IDE in this Day and Age?



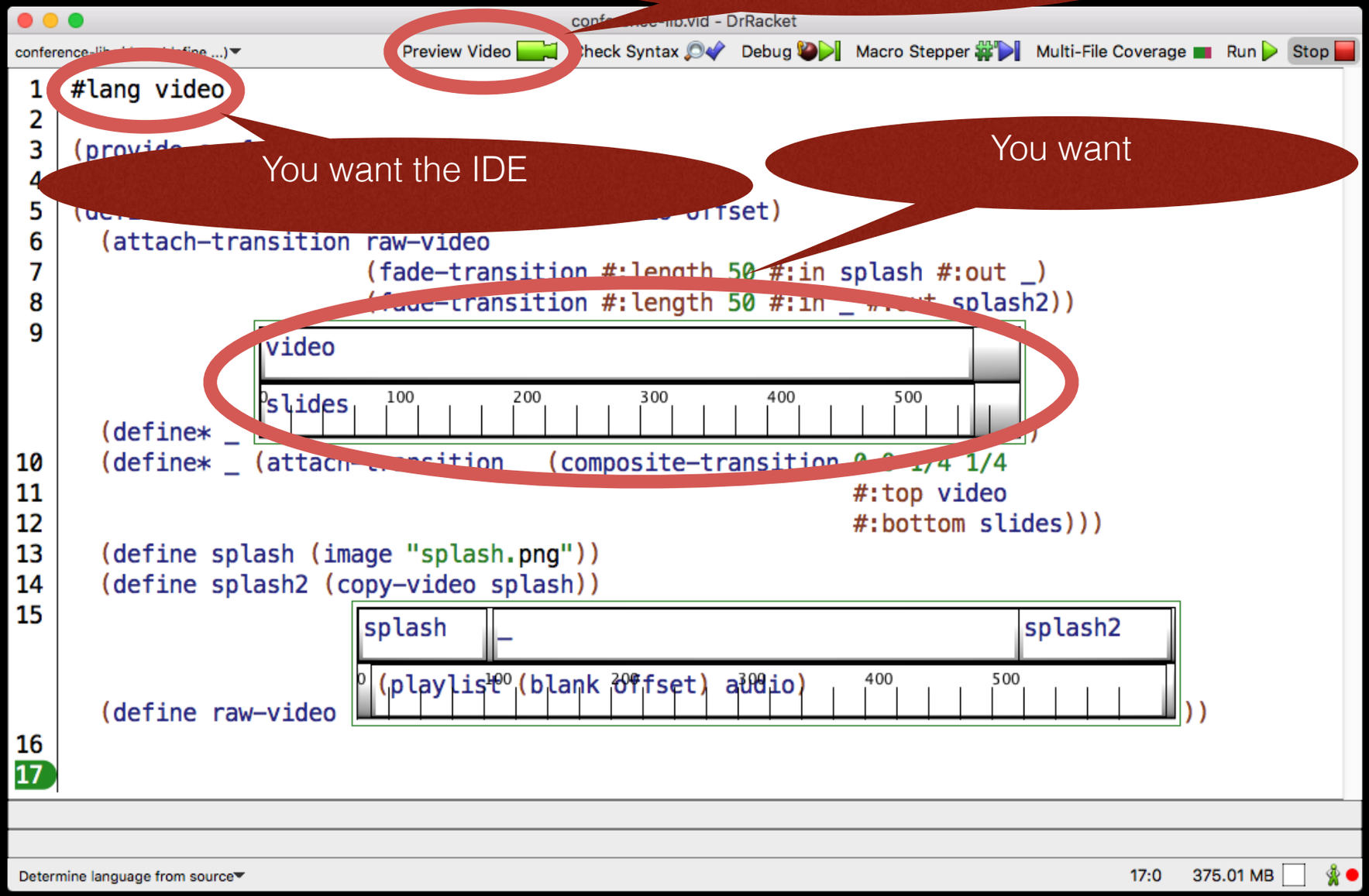
Doesn’t a Programming Language Need an IDE in this Day and Age?



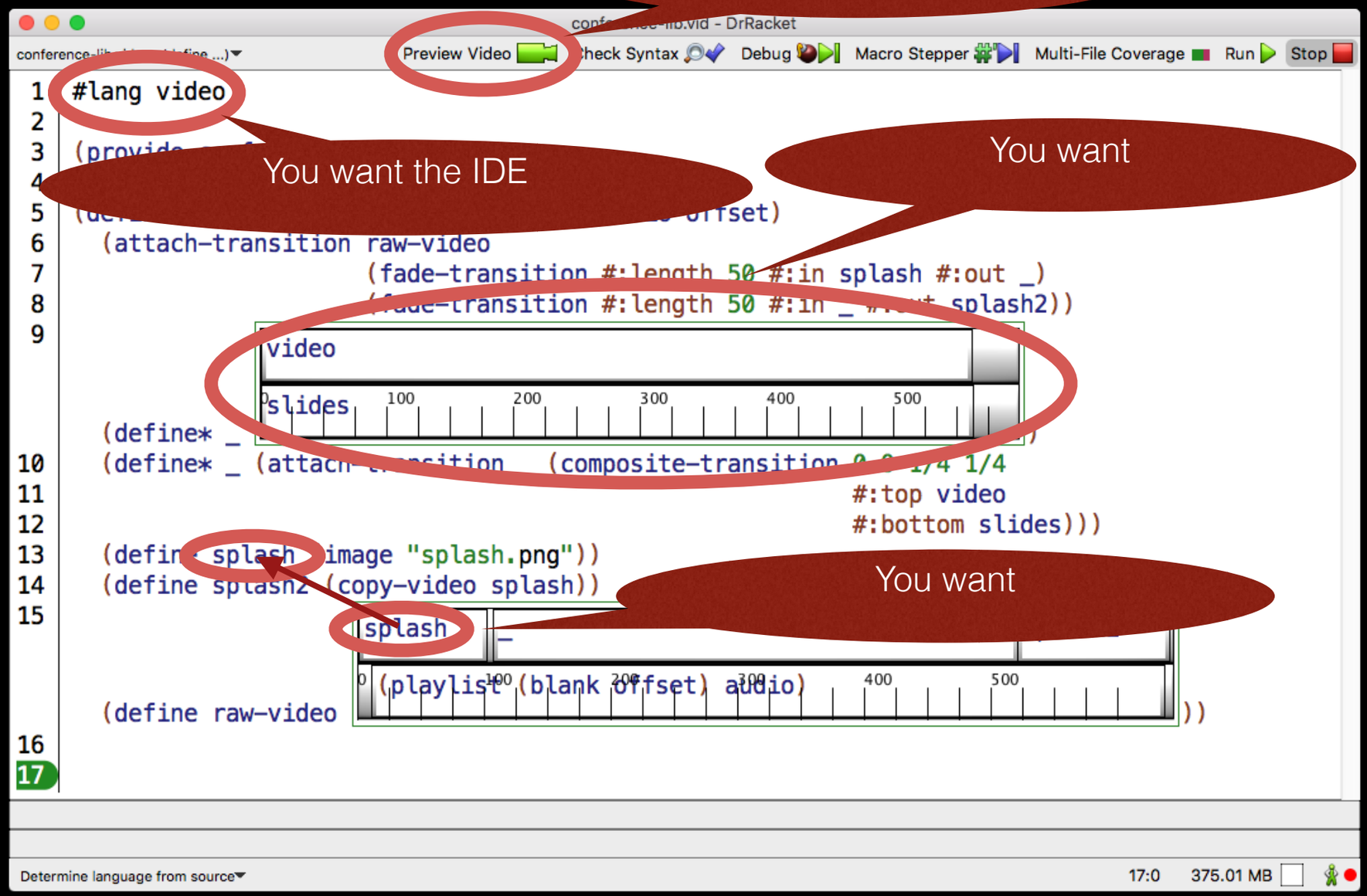
Doesn’t a Programming Language Need an IDE in this Day and Age?



Doesn't a Programming Language Need an IDE in this Day and Age?



Doesn't a Programming Language Need an IDE in this Day and Age?

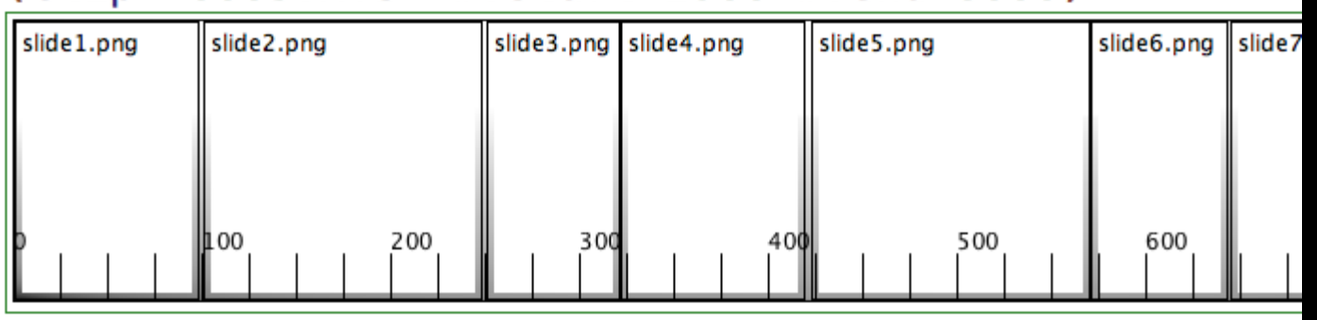


What it would take Benjy Montoya now to produce one conference video

```
#lang video

(require "conference-lib.rkt")

(make-conference-talk
 (clip "0005.MTS" #:start 2900 #:end 8000)
```



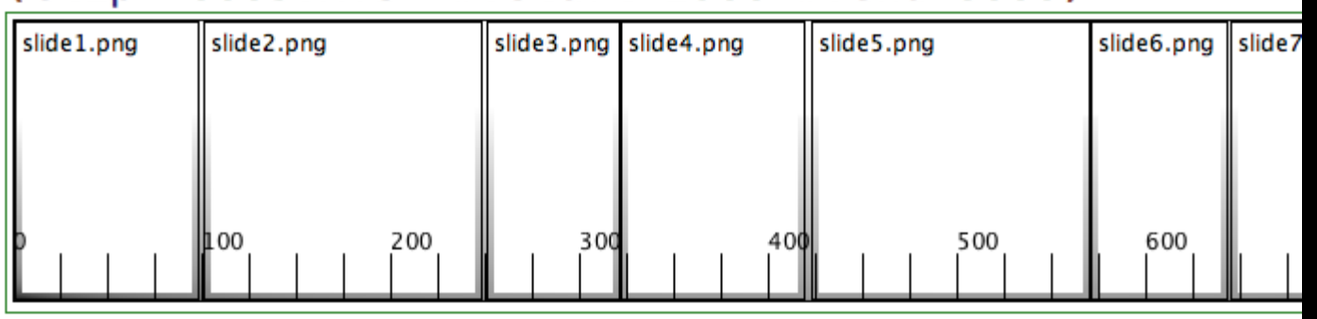

```
(playlist (clip "0001.wav") (clip "0002.wav")))
```

What it would take Benjy Montoya now to produce one conference video

```
#lang video

(require "conference-lib.rkt")

(make-conference-talk
 (clip "0005.MTS" #:start 2900 #:end 8000)
```




```
(playlist (clip "0001.wav") (clip "0002.wav")))
```

And with `map`, he can do a complete conference video channel.

- In 2015, Leif ran the video production for RacketCon.
- In 2016, Leif produced Video lang and then ran the video production for RacketCon again with DSL programs
- … which took less time than the year before.

Contact Leif Andersen @leifandersen.net for Video DSL.

Programming Languages

Language-Oriented Programming is:

- Program DSLs quickly
- Program in them at the same time/IDE
- Connecting these programs smoothly
- Make these connections safe and secure

Language-Oriented Programming is:

in Racket

- Program DSLs quickly
- Program in them at the same time/IDE

- Connecting these programs smoothly
- Make these connections safe and secure

Racket Programs Consist of Modules

```
#lang language

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

Racket Programs Consist of Modules

every module is coded in its own language

```
#lang language

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

Racket Programs Consist of Modules

```
#lang language

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

Racket Programs Consist of Modules

```
#lang language

(provide
   function … structs … classes … obj
   const
   [rena
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

module may define new syntactic constructs

Racket Programs Consist of Modules

```
#lang language

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

Racket Programs Consist of Modules

```
#lang lang

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

a module may export constructs

Racket Programs Consist of Modules

```
#lang language

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

Racket Programs Consist of Modules

```
#lang language

(provide
   function …
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

on export, a construct can take on the name of an existing construct

Racket Programs Consist of Modules

```
#lang language

(provide
   function … structs … classes … obj
   construct …
   [rename-out
     [new-construct old-name]…])

(define-syntax (new-construct stx)
   .. ..)

(define (function argument ..) ..)
```

demo: basic functional module

A Racket Language ***is*** a Module With At Least One Specific Export

```
#lang language


(provide #%module-begin)
```

A Racket Language ***is*** a Module With At Least One Specific Export

```
#lang language


(provide #%module-begin)
```

Creating a New Language

```
  Base Language
- Existing Constructs
+ New Constructs
+ Reinterpreted Constructs
-------------------------
= New Language
```

Creating a New Language

```
  Base Language                (the #lang one)
- Existing Constructs
+ New Constructs
+ Reinterpreted Constructs
-----------------------------------------------
= New Language
```

Creating a New Language

```
  Base Language                (the #lang one)
- Existing Constructs          (just don't provide)
+ New Constructs
+ Reinterpreted Constructs
-------------------------------------------------
= New Language
```

Creating a New Language

```
  Base Language                (the #lang one)
- Existing Constructs          (just don't provide)
+ New Constructs               (define and provide)
+ Reinterpreted Constructs
-------------------------------------------------
= New Language
```

Creating a New Language

```
  Base Language                (the #lang one)
- Existing Constructs          (just don't provide)
+ New Constructs               (define and provide)
+ Reinterpreted Constructs     (old-name for new)
-------------------------------------------------
= New Language
```

Creating a New Language

```
  Base Language                (the #lang one)
- Existing Constructs          (just don't provide)
+ New Constructs               (define and provide)
+ Reinterpreted Constructs     (old-name for new)
-------------------------------------------------
= New Language                 (New, like Old, But)
```

Constructs you want to re-interpret:

- everything visible in the base language:
  - functions, constants, constructs
- everything invisible aka interposition points:
  - `#%app`
  - `#%module-begin`

Function application as an interposition point:

```
(function argument … argument)
== parses into ==>
(#%app function argument … argument)
```

Module “bodies” as an interposition point:

```
#lang Language Thing … Thing
== parses into ==>
#lang Language (#%module-begin Thing … Thing)
```

Module “bodies” as an interposition point:

```
#lang Language Thing … Thing
== parses into ==>
#lang Language (#%module-begin Thing … Thing)
```

… and this is how we can start rewriting
an entire module

Module “bodies” as an interposition point:

```
#lang Language Thing … Thing
== parses into ==>
#lang Language (#%module-begin Thing … Thing)
```

Ready?

Ready?

In the next 15 seconds,
we will use this idea
to create a very lazy variant of Racket.

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

register a function with the compiler

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

generate code: ignore argument

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

generate code: expand into old `#%module-begin`

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

```
#lang racket

(provide
 [rename-out
  [very-lazy-language #%module-begin]])

(define-syntax (very-lazy-language stx)
  #'(#%module-begin
     (displayln
      `(This is the laziest language of all))))
```

export `very-lazy` as *new* `#%module-begin`

demo: basic in very-lazy

Ready?

Ready?

In the next 15 minutes,
we will use this idea
to create a lazy variant of Racket.

```
#lang racket

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (+ 1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 9))
==>
5
```

```
#lang racket

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (+ 1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
ERROR!!
```

```
#lang racket

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (+ 1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
ERROR!!
```

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (+ 1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
5
```

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (+ 1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
5
```

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (+ 1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0) )
  )
5
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

make this module a DSL

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

import a DSL for making DSLs

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

a compile-time translation

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

match for code

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

```
#lang racket

(provide #%module-begin )

(require (for-syntax syntax/pa ))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

pattern (#%app f x1 .. xn)

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

```
#lang racket                                                    lazy

(provide #%module-begin …)

(require (for-syntax syntax/par

(define-syntax (lazy-function-app
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

generate code: suspend all arguments

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

rename on export

```
#lang racket

(provide #%module-begin)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

client

```
#lang s-exp “lazy.rkt”

(+ 42 (/ 1 0))
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

client

```
#lang s-exp “lazy.rkt”

(+ 42 (/ 1 0))
```

== elaborates to ==>

client

```
#lang s-exp “lazy.rkt”

(#%app + 42 (/ 1 0))
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

client

```
#lang s-exp “lazy.rkt”

(+ 42 (/ 1 0))
```

== elaborates to ==>

client

```
#lang s-exp “lazy.rkt”

(#%app + 42 (/ 1 0))
```

lazy

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

client

```
#lang s-exp “lazy.rkt”

(+ 42 (/ 1 0))
```

== elaborates to ==>

client

```
#lang s-exp “lazy.rkt”

(#%app + 42 (/ 1 0))
```

```
#lang racket                                                    lazy

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function (lambda () argument) …)]))
(provide [rename-out [lazy-function-application #%app]])
```

```
#lang s-exp “lazy.rkt”                                          client

(#%app + 42 (/ 1 0))
```

== elaborates to ==>

```
#lang s-exp “lazy.rkt”                                          client

(#%app + [lambda () 42] [lambda () (/ 1 0)])
```

```
#lang s-exp “lazy.rkt”                                    client

(#%app + [lambda () 42] [lambda () (/ 1 0)])
```

== compile, run, raise exception ==>

```
+: contract violation
 expected: number?
 given: #<procedure: lazy.rkt:28:54>
```

- We must “strictify” the + function in the lazy variant of Racket.
- And we may need to “strictify” other functions, too.

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (add1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
5
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

```
#lang racket

(provide #%module-b

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

simple applicable, unique
because local wrapper around “thunks”

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function
  (syntax-parse stx
    [(_ function:expr argument:
     #'(#%app function [thunked (lam da () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

recursively run wrappers to get underlying value

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-applicati
(struct thunked [th] #:property prop

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

a *curried* function to make functions strict in *all* arguments

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(provide [rename-out [add1-s add1]])
(define add1-s (strictify add1))

(provide [rename-out [+-s +]])
(define +-s (strictify +))
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))
```

```
(provide [rename-out [add1-s add1]])
(define add1-s (strictify add1))

(provide [rename-out [+-s +]])
(define +-s (strictify +))
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(provide [rename-out [add1-s add1]])
(define add1-s (strictify add1))

(provide [rename-out [+-s +]])
(define +-s (strictify +))
```

noticed the repeated syntactic pattern

```
#lang racket


(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(provide-strictified add1)

(provide-strictified +)
```

another function for the compiler to rewrite a name into a `strictly` definition plus a `provide`

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(provide-strictified add1)

(provide-strictified +)
```

```
#lang racket


(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(provide-strictified add1)

(provide-strictified +)
```

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(provide-strictified add1)

(provide-strictified +)
```

worried?

```
#lang racket


(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(provide-strictified add1)

(provide-strictified +)
```

hygiene takes care, but that's 35 years old, so even Scala should have it now

orried?

```
#lang racket

(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(provide-strictified add1)

(provide-strictified +)
```

worried?

```
#lang racket


(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(define-syntax (provide-strictified* stx)
  (syntax-parse stx
    [(_ x:id ...) #'(begin (provide-strictified x) ...)]))

(provide-strictified* +  add1 - / >= first rest empty?)
```

```
#lang racket


(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))

(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name]))))]))

(define-syntax (provide-strictified* stx)
  (syntax-parse stx
    [(_ x:id ...) #'(begin (provide-strictified x) ...)]))

(provide-strictified* + add1 - / >= first rest empty?)
```

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (add1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
5
```

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                                     [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (add1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
5
```

```
#lang s-exp “lazy-racket.rkt”

#;(Real [Listof Real] {Natural Real} -> (U False Natural))
(define (how-many-elements-to-sum-to threshold l
                [count 1] [running-sum 0])
  (cond
    [(empty? l) #false]
    [else
     (define one (first l))
     (define sum (+ one running-sum))
     (if (>= sum threshold)
         count
         (how-many-elements-to-sum-to
           threshold (rest l) (add1 count) sum))]))
```

```
(how-many-elements-to-sum-to
  10
  (list 0 1 2 3 4 5 6 7 8 (/ 1 0)))
==>
5
```

#lang racket

```
(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))


(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name])))]))

(define-syntax (provide-strictified* stx)
  (syntax-parse stx
    [(_ x:id ...) #'(begin (provide-strictified x) ...)]))

(provide-strictified* +  add1 - / >= first rest empty?)
```

```
(provide [rename-out [lazy-list list]])
(define (lazy-list . r) r)
```

#lang racket

```
(provide #%module-begin …)

(require (for-syntax syntax/parse))

(define-syntax (lazy-function-application stx)
  (syntax-parse stx
    [(_ function:expr argument:expr ...)
     #'(#%app function [thunked (lambda () argument)] …)]))
(provide [rename-out [lazy-function-application #%app]])

(struct thunked [th] #:property prop:procedure 0)

(define (force* thunked-or-not)
  (if (thunked? thunked-or-not)
      (force* (thunked-or-not))
      thunked-or-not))

(define ((strictify function) . arguments)
  (apply function (map force* arguments)))


(define-syntax (provide-strictified stx)
  (syntax-parse stx
    [(_ name:id)
     #'(begin
         (define name-strict (strictify name))
         (provide (rename-out [name-strict name])))]))

(define-syntax (provide-strictified* stx)
  (syntax-parse stx
    [(_ x:id ...) #'(begin (provide-strictified x) ...)]))

(provide-strictified* +  add1 - / >= first rest empty?)
```

```
(provide [rename-out [lazy-list list]])
(define (lazy-list . r) r)
```

So, we just programmed for 15 minutes.
We now have a pretty good start on a lazy Racket.
That’s what “programming languages” means.

demo: basic with lazy

Techniques for Programming Languages I did ***Not*** Show

Techniques for Programming Languages I did ***Not*** Show

- macro-defining macros

Techniques for Programming Languages I did ***Not*** Show

- macro-defining macros
- macros that define macro-defining macros

Techniques for Programming Languages I did ***Not*** Show

- macro-defining macros
- macros that define macro-defining macros
- macro-defining macro-defining macros

Techniques for Programming Languages I did ***Not*** Show

- macro-defining macros
- macros that define macro-defining macros
- macro-defining macro-defining macros
- no, there really is no limit

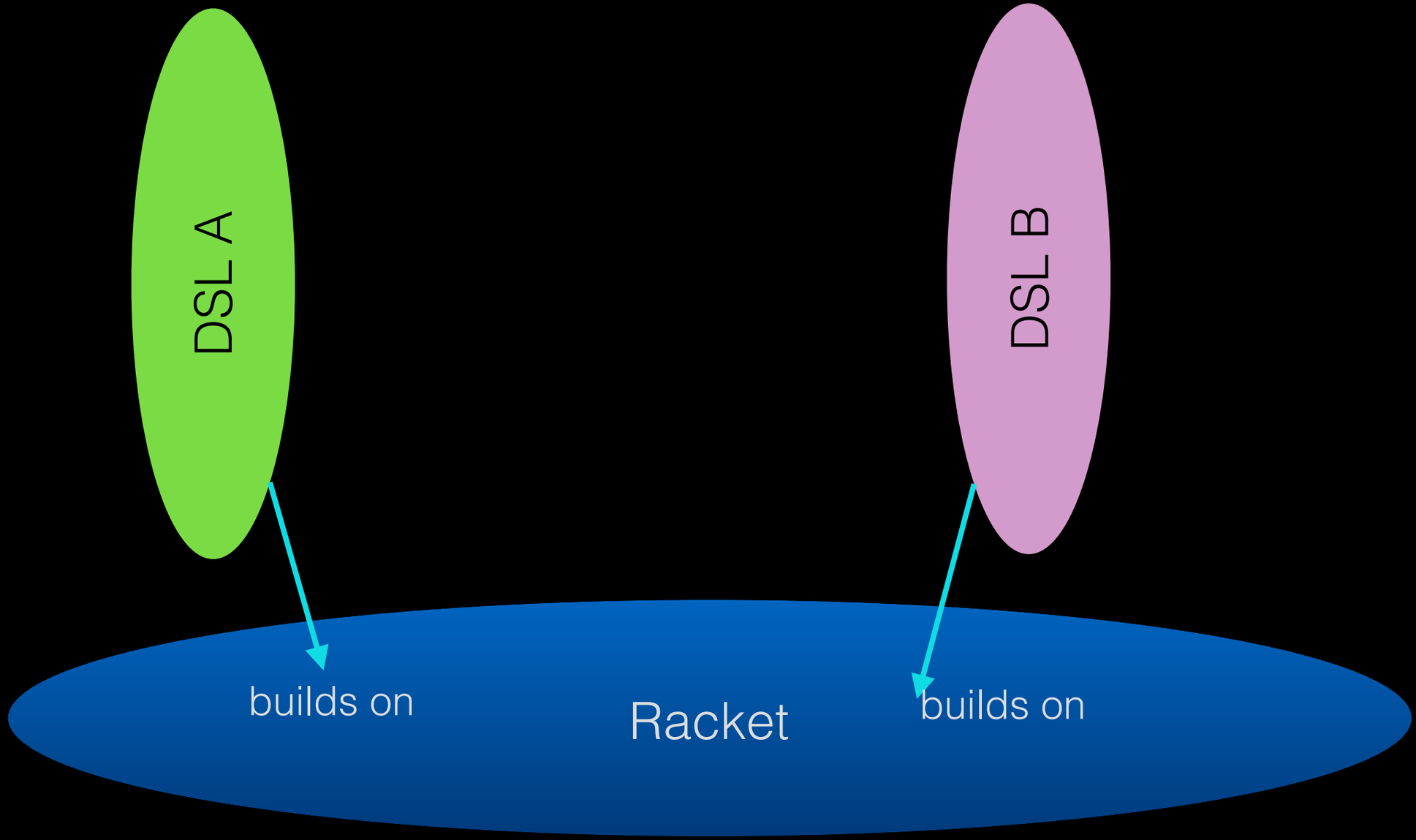
Techniques for Programming Languages I did ***Not*** Show

- macro-defining macros
- module-crossing syntax information
- expander-defining macros
- multi-pass compilation with macros
- parsing “ugly syntax” into macros

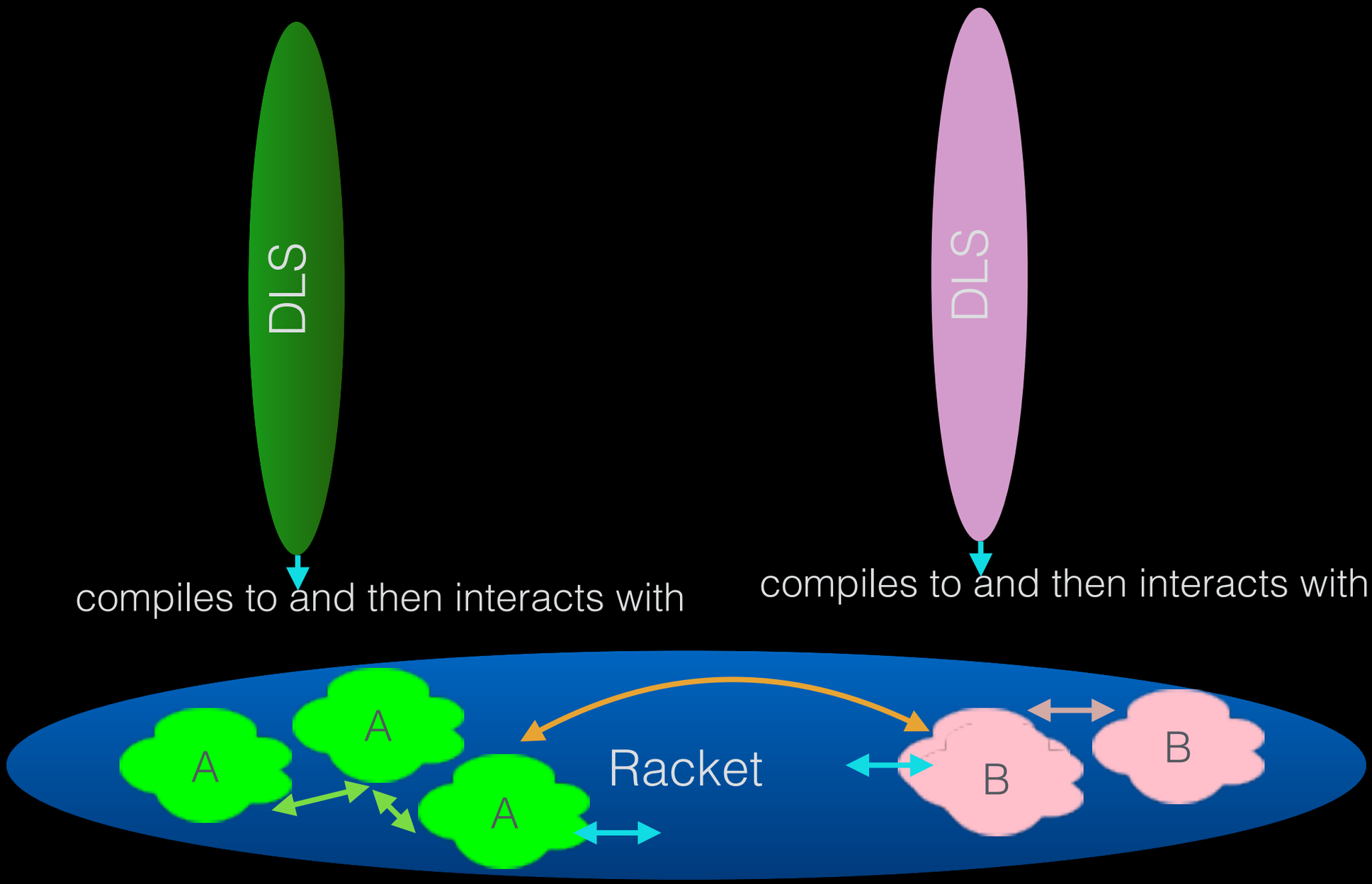
demo: some more languages

Onwards to LOP

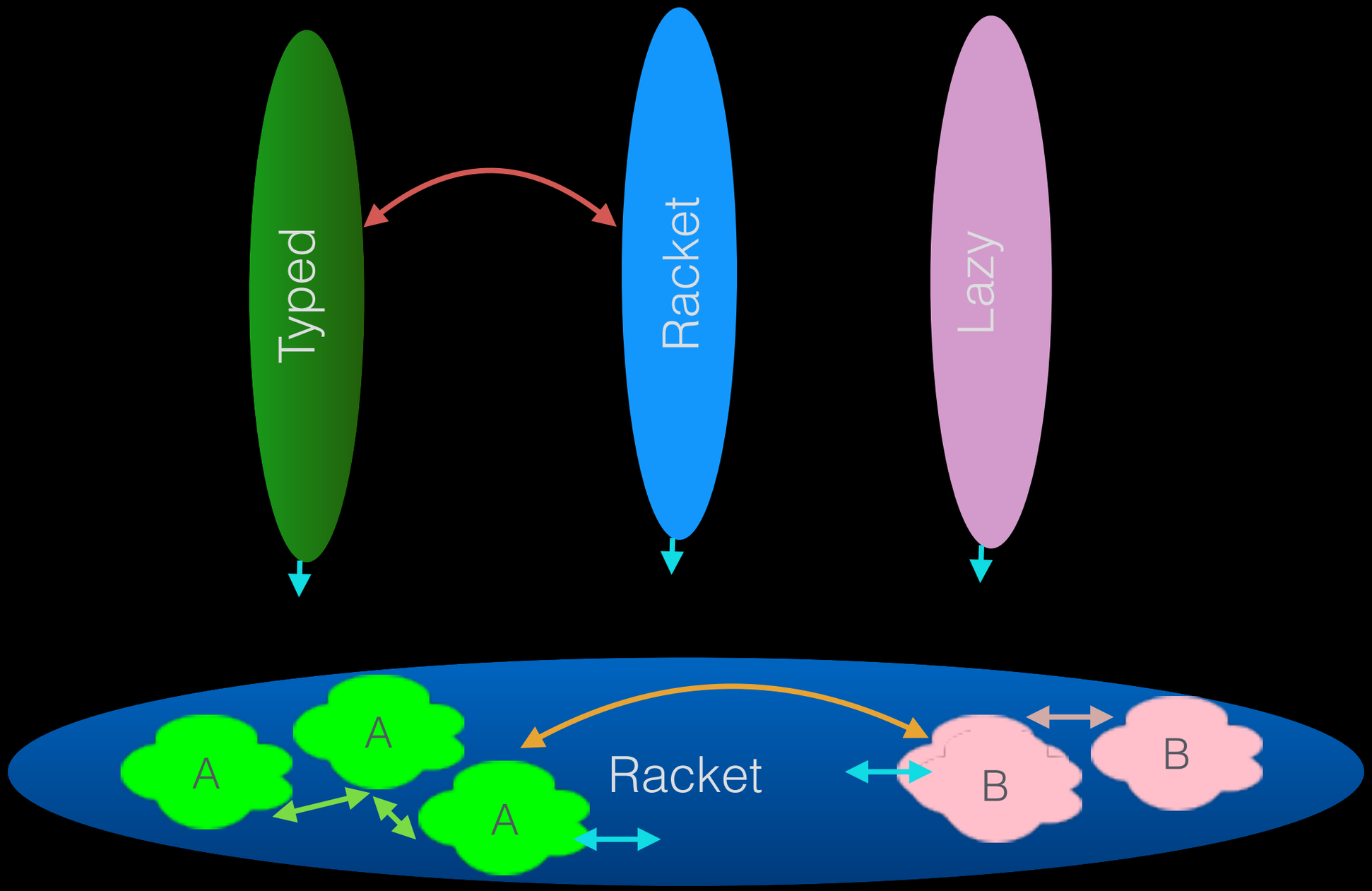
And then you compose those DSLs at will.



And then you compose those DSLs at will.



What does interaction mean here?



What does interaction mean here?

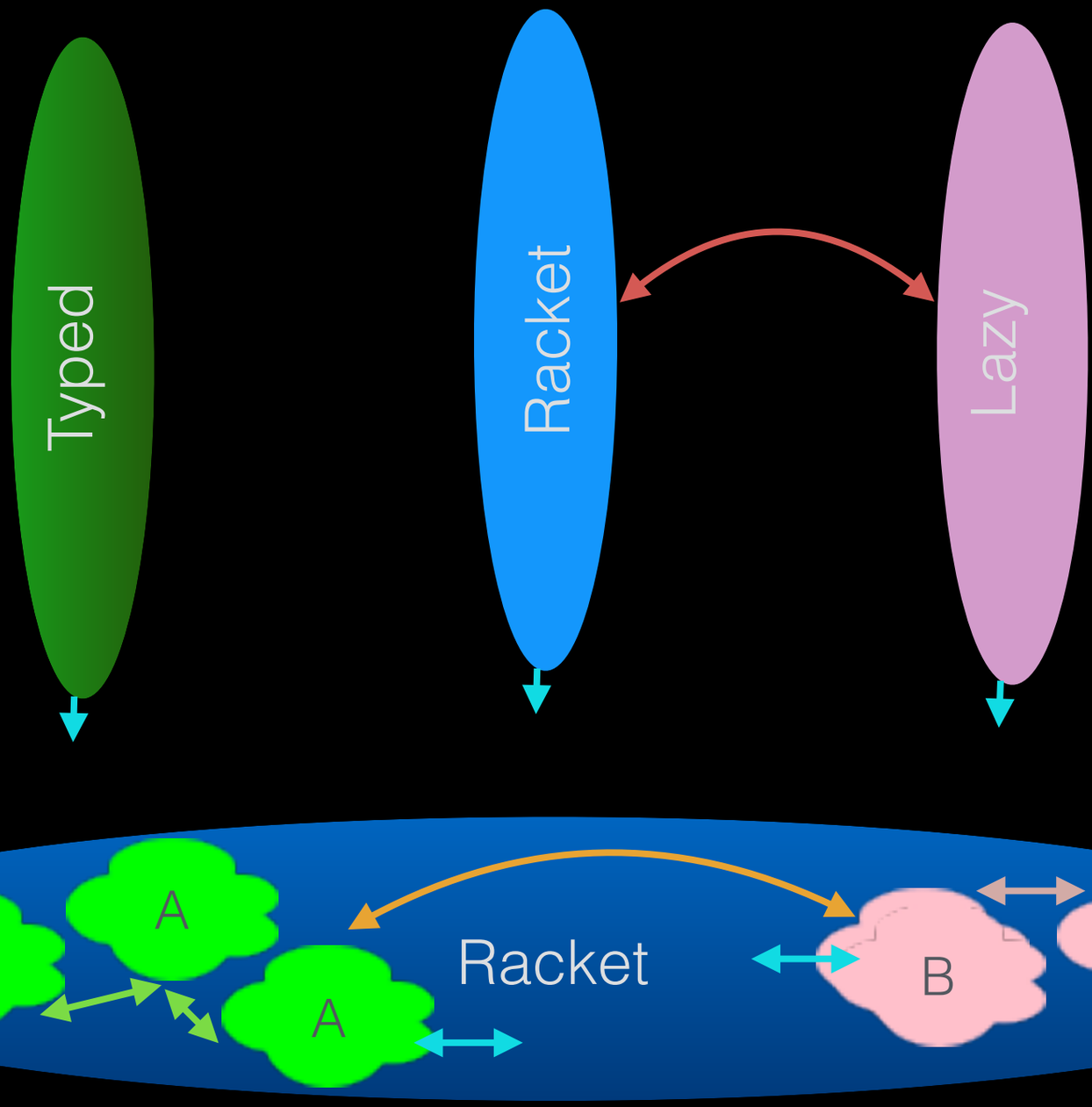
Dimoulas, Findler, Felleisen
Contracts. 2002-2012.

Tobin-Hochstadt, Greenman, Felleisen.
Migratory Types. 2006-2019.

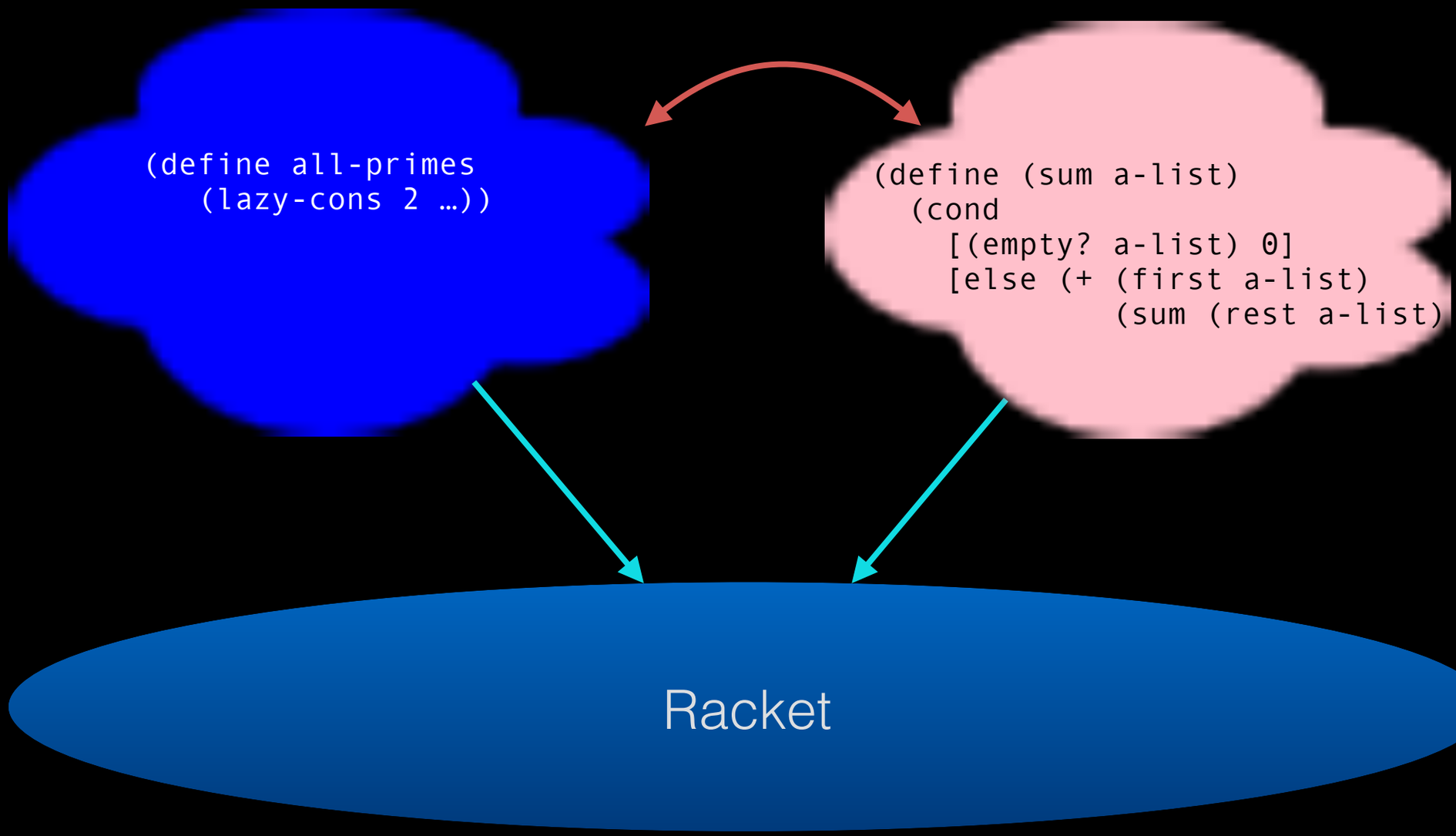
```
(: d/dx ((Real -> Real) -> (Real
(define (d/x f)
  (define (fprime x) …)
  fprime)
```

```
(d/dx
  (lambda (x)
    (if (and (number? x) (< x 4
        (polynomial x)
        "hello world, good bye")
```

What does interaction mean here?

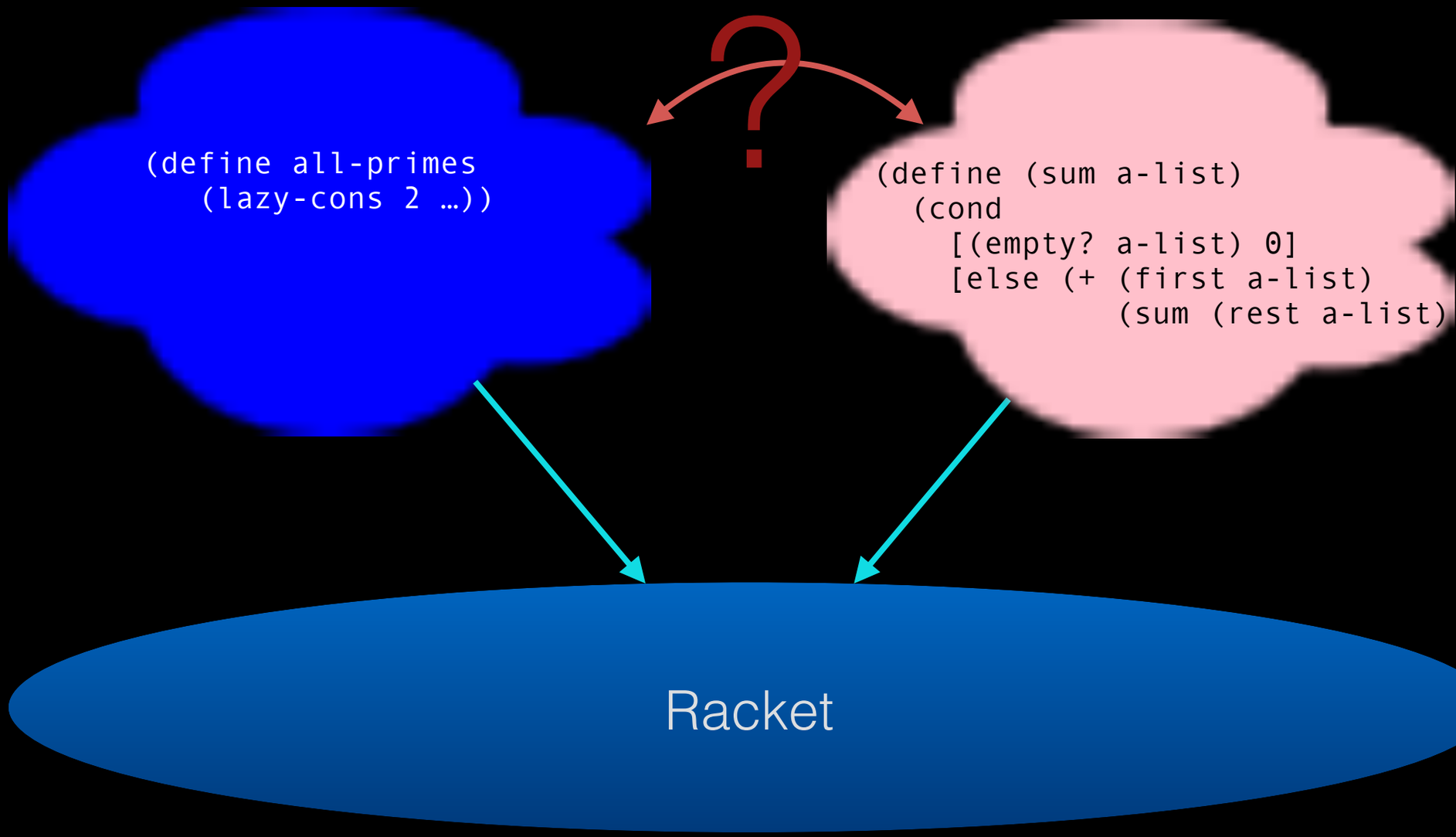


What does interaction mean here?

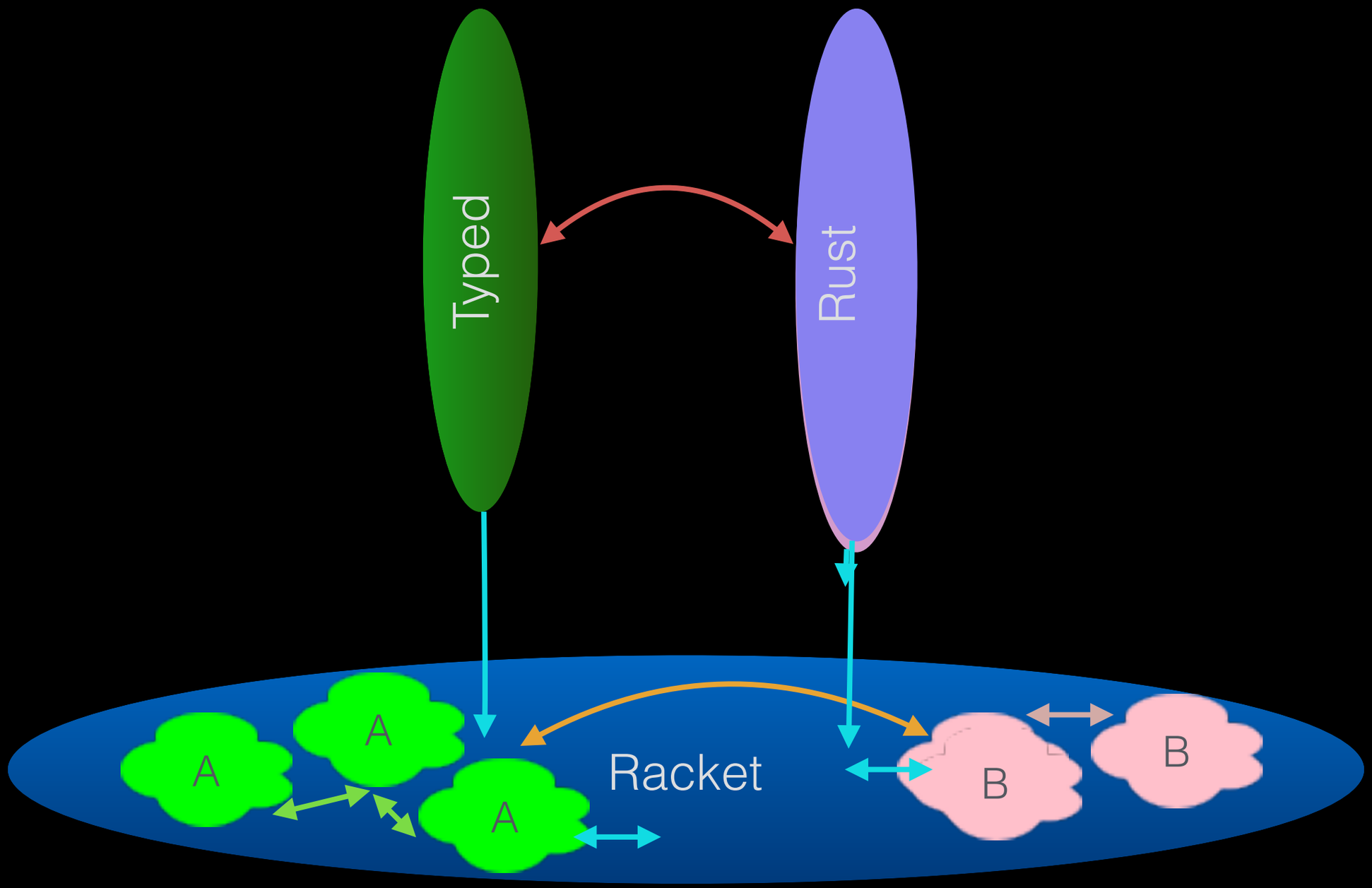


What does interaction mean here?

Chang & Felleisen. 2011-2015.



What does interaction mean here?



What Language-Oriented Programming Still Needs

- types for controlling DSL interactions
- run-time monitors for controlling DSL interactions
- resource controls

Take Away

- Programming a Language in Racket is easy, smooth, and productive
- Programming Languages has become feasible
- LOP with Simple DSL is becoming a reality
- LOP with Complex DSL is still open to research

The End.

Special thanks to Matthew Flatt, Robby Findler, Shriram Krishnamurthi, Sam Tobin-Hochstadt, Eli Barzilay, Jay McCarthy, Christos Dimoulas, Amal Ahmed,
and many others for implementing a sketchy vision and destroying my easy solutions.