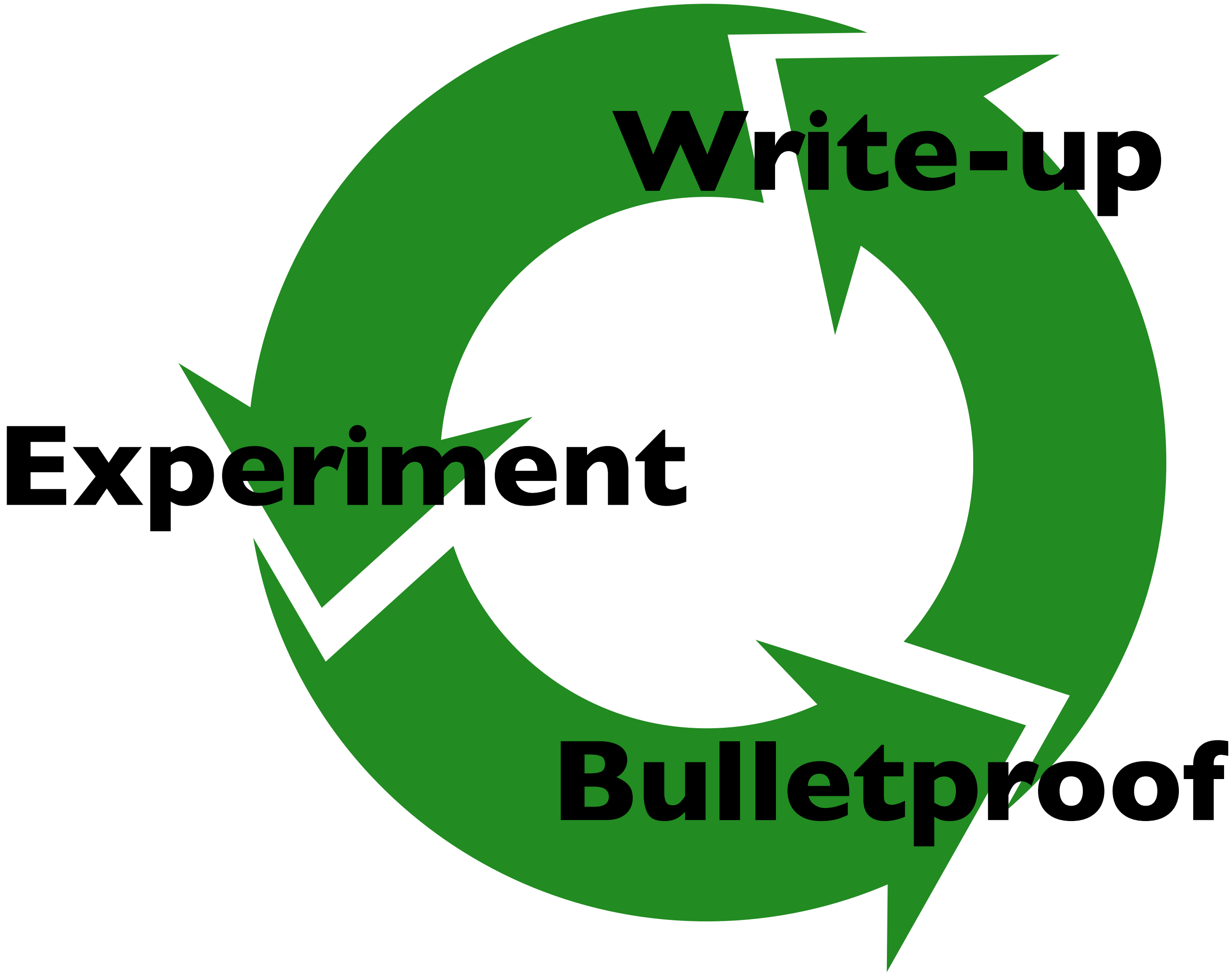
# Redex:

## Lightweight Semantics Engineering

Robby Findler & Burke Fetscher
Northwestern University & PLT

# Semantics Lifecycle



Write-up

Experiment

Bulletproof

# Redex DSL Desiderata:

- Executable
- Notation matches existing literature

⇒ reject extensions to Redex that cannot be typeset

# Outline

- **An overview of Redex**
- **You watch me type**
- **We watch you type**

# Outline

- **An overview of Redex**
  - Operational semantics
  - Redex's DSL & tools
  - First-order class model

- **You watch me type**
  - Develop a higher-order variant of model
  - How good is random testing?

- **We watch you type**
  - Amb: Redex's tutorial
  - Underspecification: Scheme's order of evaluation
  - Types: rewrite expressions to their types

$\mathscr{Eval}$ : program ⇴ answer

$\mathscr{Eval}(\mathrm{p}) = \mathrm{a}$ iff $\quad \cdots \mathrm{p} \cdots \mathrm{a} \cdots$

$\mathscr{Eval} : \text{program} \mathrel{-\!\!\circ\!\!\!\rightarrow} \text{answer}$

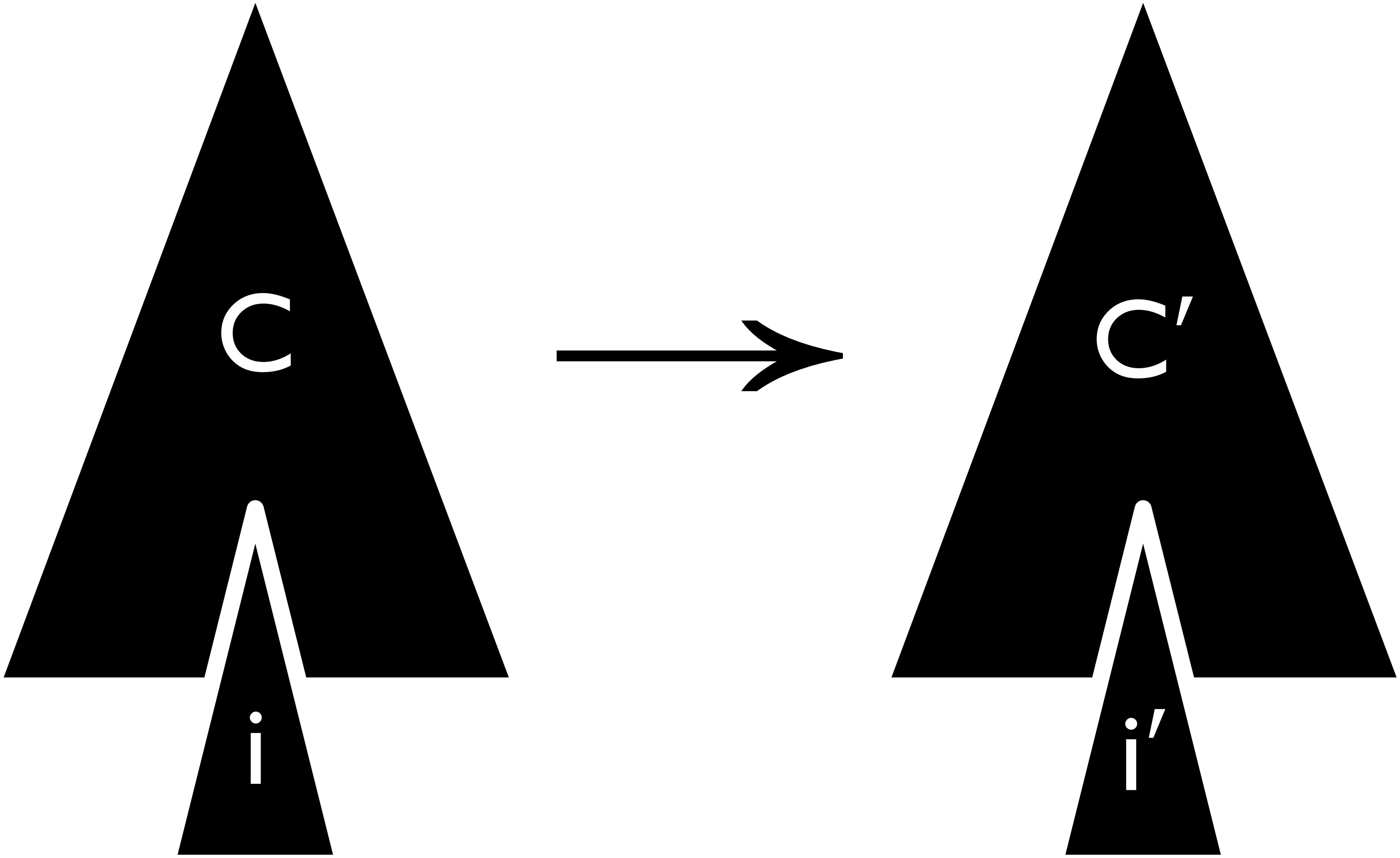
$\mathscr{Eval}(\text{p}) = \text{a iff p} \longrightarrow^{*} \text{a}$

**where**

answer ⊂ program

$\longrightarrow \subset \text{program} \times \text{program}$

# Reduction



# Semantics Recap

- Specify programs & answers (grammar)
- Specify evaluation contexts (grammar)
- Specify a reduction relation

```
p ::= (begin d ... e)
d ::= (define z c)
c ::= (class object%
        (init-field i)
        m ...
        (super-make-object))
m ::= (define/public (x y ...)
        e)
e ::= (make-object x e)
    | (send e x e ...)
    | x
    | this
    | number
    | (if0 e e e)
    | (+ e ...)
```

```
p ::= (begin d ... e)
d ::= (define z c)
c ::= (class object%
        (init-field i)
        m ...
        (super-make-object))
m ::= (define/public (x y ...)
        e)
e ::= (make-object x e)
    | (send e x e ...)
    | x
    | this
    | number
    | (if0 e e e)
    | (+ e ...)
```

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...)))
```

```
p ::= (begin d ... e)
d ::= (define z c)
c ::= (class object%
        (init-field i)
        m ...
        (super-make-object))
m ::= (define/public (x y ...)
        e)
e ::= (make-object x e)
    | (send e x e ...)
    | x
    | this
    | number
    | (if0 e e e)
    | (+ e ...)
```

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

A grammar associates non-terminals with patterns; parens are significant, indicating tree structure (aka “regular tree grammars”)

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Some patterns, e.g. **number**, are like built-in non-terminals; in this case matching all Racket numbers

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

The ellipsis is a Kleene star; a post-fix operator that allows zero or more repetitions of the pattern it follows

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

These are the non-terminals; the definitions come from the grammar

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Which leaves the literals; this is a catch-all category and they act like keywords for the language

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

This pattern matches any identifier except literals (making it sensitive to the language where it appears)

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Enough of Redex—now for the language; a program consists of definitions & an expression

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Definitions pair variables with classes

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Each class has a single initialization argument `i`, a bunch of methods, and a call to `super-make-object`; much of this is to mimic the syntactic structure of Racket's class system

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Each method has a name **x**, multiple arguments **y**, and a body **e**

```
(define-language Roo
  (p  (begin d ... e))
  (d  (define z c))
  (c  (class object%
        (init-field i)
        m ...
        (super-make-object)))
  (m  (define/public (x y ...)
        e))
  (e  (make-object x e)
      (send e x e ...)
      x
      this
      number
      (if0 e e e)
      (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

Expressions either are object creation, method invocation, variables, the **this** keyword, numbers, conditionals, or addition expressions

```
(define-language Roo
  (p (begin d ... e))
  (d (define z c))
  (c (class object%
       (init-field i)
       m ...
       (super-make-object)))
  (m (define/public (x y ...)
       e))
  (e (make-object x e)
     (send e x e ...)
     x
     this
     number
     (if0 e e e)
     (+ e ...))
  ((i x y z)
   variable-not-otherwise-mentioned))
```

# Java

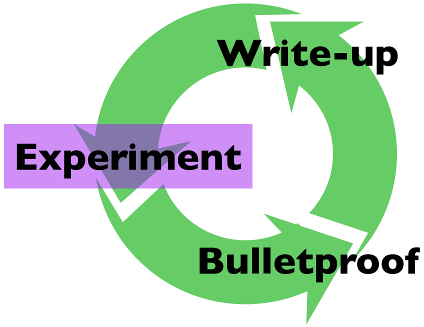
```
interface I {
  int len();
}
class Link
    implements I {
  I n;
  Node(I n) {
    this.n=n;
  }
  int len() {
    return n.len()+1;
}
class Empty
    implements I {
  int len() {
    return 0;
}
```

## Java

```
interface I {
  int len();
}
class Link
    implements I {
  I n;
  Node(I n) {
    this.n=n;
  }
  int len() {
    return n.len()+1;
}
class Empty
    implements I {
  int len() {
    return 0;
}
```

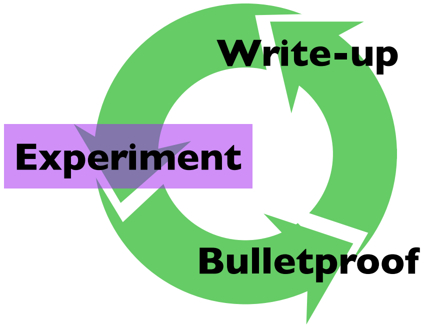
## Racket (model)

```
(define Link
  (class object%
    (init-field n)
    (define/public (len)
      (+ (send n len) 1))
    (super-make-object)))

(define Empty
  (class object%
    (init-field ignore)
    (define/public (len)
      0)
    (super-make-object)))
```



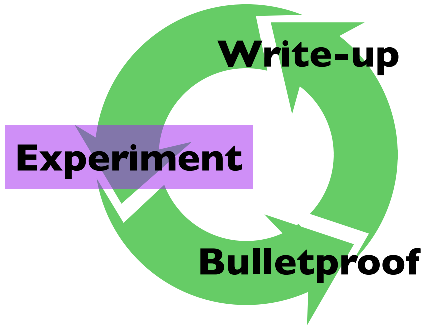
```
> (redex-match Roo
               (+ e_1 e_2)
               (term (+ (if0 1 2 3)
                        4)))

(list
 (match
  (list
   (bind 'e_1 '(if0 1 2 3))
   (bind 'e_2 4))))
```

A **`redex-match`** expression accepts a language, a pattern, and a term; it tests the pattern against the expression and returns bindings for the pattern variables



```
> (redex-match Roo
               (+ e_1 e_2)
               (term (+ (if0 1 2 3)
                        4)))

(list
 (match
  (list
   (bind 'e_1 '(if0 1 2 3))
   (bind 'e_2 4))))
```

A **redex-match** expression accepts a language, a pattern, and a term; it tests the pattern against the expression and returns bindings for the pattern variables



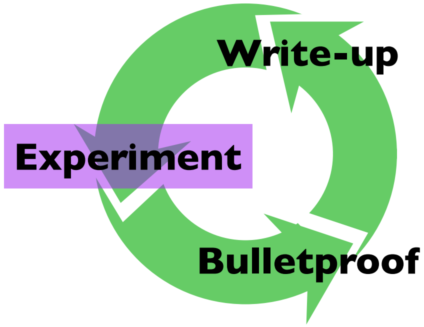
Write-up

Experiment

Bulletproof

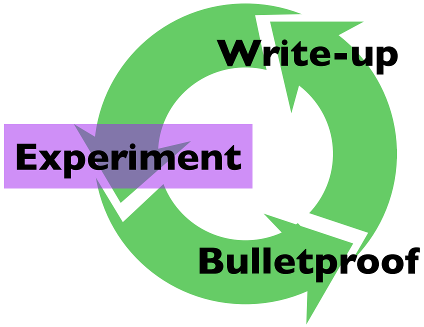
```
> (redex-match Roo
               (+ e_1 e_2)
               (term (+ (if0 1 2 3)
                         4)))

(list
 (match
  (list
   (bind 'e_1 '(if0 1 2 3))
   (bind 'e_2 4))))
```

A **redex-match** expression accepts a language, a pattern, and a term; it tests the pattern against the expression and returns bindings for the pattern variables



```
> (redex-match Roo
               (+ e_1 e_2)
               (term (+ (if0 1 2 3)
                        4)))

(list
 (match
  (list
   (bind 'e_1 '(if0 1 2 3))
   (bind 'e_2 4))))
```

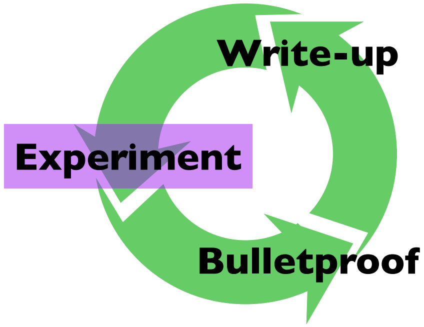
A **`redex-match`** expression accepts a language, a pattern, and a term; it tests the pattern against the expression and returns bindings for the pattern variables



```
> (redex-match Roo
               (+ e_1 e_2)
               (term (+ (if0 1 2 3)
                        4)))
```

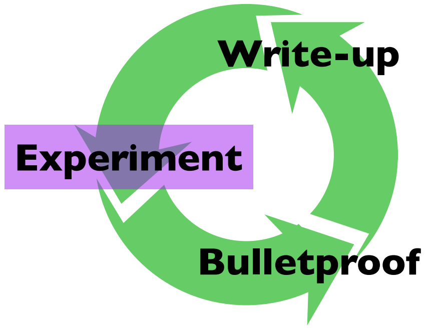
```
(list
 (match
  (list
   (bind 'e_1 '(if0 1 2 3))
   (bind 'e_2 4))))
```

A **`redex-match`** expression accepts a language, a pattern, and a term; it tests the pattern against the expression and returns bindings for the pattern variables



```
> (redex-match Roo
               (+ e ...)
               (term (+ (if0 1 2 3)
                         4)))

(list
 (match
  (list
   (bind 'e '((if0 1 2 3) 4)))))
```

When a pattern variable is behind an ellipsis, it is bound to a list whose elements match pieces of the term



```
> (redex-match
   Roo
   (+ e_1 ... e_2 e_3 ...)
   (term (+ 1 2 3)))
```

Doubled ellipses are ambiguous; so here we get three possible matches, with e_2 taking on either 1, 2, or 3, and e_1 and e_3 abosrbing the remaining numbers

```
(list
 (match
  (list
   (bind 'e_1 '())
   (bind 'e_2 1)
   (bind 'e_3 '(2 3))))
 (match
  (list
   (bind 'e_1 '(1))
   (bind 'e_2 2)
   (bind 'e_3 '(3))))
 (match
  (list
   (bind 'e_1 '(1 2))
   (bind 'e_2 3)
   (bind 'e_3 '()))))
```

# Evaluation contexts, answers, and values

*a* ::= **(begin** *d* ... *v***)**
*v* ::= **(make-object** *x v***)**
  | *number*
*P* ::= **(begin** *d* ... *E***)**
*E* ::= **(make-object** *x E***)**
  | **(send** *E x e* ...**)**
  | **(send** *v x v* ... *E e* ...**)**
  | **(if0** *E e e***)**
  | **(+** *v* ... *E e* ...**)**
  | **[]**

```
(a (begin d ... v))
(v (make-object x v)
   number)
(P (begin d ... E))
(E (make-object x E)
   (send E x e ...)
   (send v x v ... E e ...)
   (if0 E e e)
   (+ v ... E e ...)
   hole)
```

# Evaluation contexts, answers, and values

The only new pattern here, **`hole`**, collaborates with **`in-hole`** to decompose terms into contexts and expressions at the hole

```
(a (begin d ... v))
(v (make-object x v)
   number)
(P (begin d ... E))
(E (make-object x E)
   (send E x e ...)
   (send v x v ... E e ...)
   (if0 E e e)
   (+ v ... E e ...)
   hole)
```

# Evaluation contexts, answers, and values

An answer is the final result from a program, definitions plus a value

```
(a (begin d ... v))
(v (make-object x v)
   number)
(P (begin d ... E))
(E (make-object x E)
   (send E x e ...)
   (send v x v ... E e ...)
   (if0 E e e)
   (+ v ... E e ...)
   hole)
```

# Evaluation contexts, answers, and values

Values are objects and numbers

```
(a (begin d ... v))
(v (make-object x v)
   number)
(P (begin d ... E))
(E (make-object x E)
   (send E x e ...)
   (send v x v ... E e ...)
   (if0 E e e)
   (+ v ... E e ...)
   hole)
```

# Evaluation contexts, answers, and values

P stands for a program evaluation context; evaluation only happens in the main expression

```
(a (begin d ... v))
(v (make-object x v)
   number)
(P (begin d ... E))
(E (make-object x E)
   (send E x e ...)
   (send v x v ... E e ...)
   (if0 E e e)
   (+ v ... E e ...)
   hole)
```

# Evaluation contexts, answers, and values

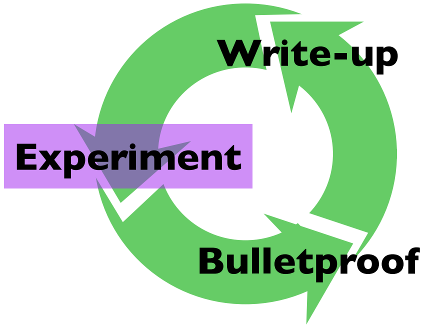
Expression evaluation can happen in the argument to `make-object`, in `send` expressions, `if0` expressions, and `+` expressions

```
(a (begin d ... v))
(v (make-object x v)
   number)
(P (begin d ... E))
(E (make-object x E)
   (send E x e ...)
   (send v x v ... E e ...)
   (if0 E e e)
   (+ v ... E e ...)
   hole)
```

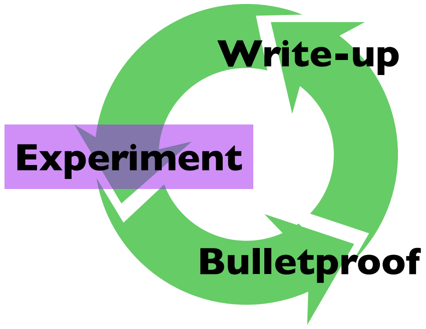
# Evaluation contexts, answers, and values

Note that use of doubled ellipses forcing left-to-right order of evaluation

```
(a  (begin d ... v))
(v  (make-object x v)
    number)
(P  (begin d ... E))
(E  (make-object x E)
    (send E x e ...)
    (send v x v ... E e ...)
    (if0 E e e)
    (+ v ... E e ...)
    hole)
```



```
> (redex-match
   Roo
   (in-hole E (+ number_1 number_2))
   (term (if0 (+ 1 2) 3 4)))
(list
 (match
  (list
   (bind 'E (list 'if0 (hole 'the-hole) 3 4))
   (bind 'number_1 1)
   (bind 'number_2 2))))
```



Write-up

Experiment

Bulletproof

```
> (redex-match
   Roo
   (in-hole E e)
   (term (if0 (+ 1 2) 3 4)))
(list
 (match
  (list
   (bind 'E (hole 'the-hole))
   (bind 'e '(if0 (+ 1 2) 3 4))))
 (match
  (list
   (bind 'E (list 'if0 (hole 'the-hole) 3 4))
   (bind 'e '(+ 1 2))))
 (match
  (list
   (bind 'E (list 'if0 (list '+ 1 (hole 'the-hole)) 3 4))
   (bind 'e 2)))
 (match
  (list
   (bind 'E (list 'if0 (list '+ (hole 'the-hole) 2) 3 4))
   (bind 'e 1))))
```

$$P[\,\texttt{(+}\ \mathit{number}\ \ldots\texttt{)}\,] \longrightarrow P[\Sigma[\![\mathit{number}\ \ldots]\!]] \qquad [+]$$

$$P[\,\texttt{(if0}\ 0\ e_1\ e_2\texttt{)}\,] \longrightarrow P[e_1] \qquad [\mathrm{if0t}]$$

$$P[\,\texttt{(if0}\ \mathit{number}\ e_1\ e_2\texttt{)}\,] \longrightarrow P[e_2] \qquad [\mathrm{if0f}]$$

$$\text{where } \mathit{number} \neq 0$$

```
(define num-rules
  (reduction-relation
   Roo
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 number e_1 e_2))
        (in-hole P e_2)
        (side-condition (term (nonzero number)))
        "if0f")))
(define-metafunction Roo
  Σ : number ... -> number
  [(Σ number ...) ,(apply + (term (number ...)))])
```

```
(define num-rules
  (reduction-relation
   Roo
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 number e_1 e_2))
        (in-hole P e_2)
        (side-condition (term (nonzero number)))
        "if0f")))
(define-metafunction Roo
  Σ : number ... -> number
  [(Σ number ...) ,(apply + (term (number ...)))])
```

Three reduction rules; the prefix operator --> introduces each one

```
(define num-rules
  (reduction-relation
   Roo
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 number e_1 e_2))
        (in-hole P e_2)
        (side-condition (term (nonzero number)))
        "if0f")))
(define-metafunction Roo
  Σ : number ... -> number
  [(Σ number ...) ,(apply + (term (number ...)))])
```

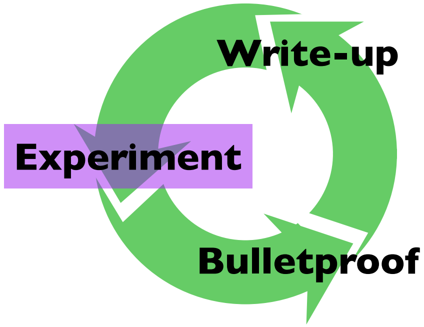
+ reduces via the metafunction Σ (redex supports unicode so Σ is just a regular identifier)

```
(define num-rules
  (reduction-relation
   Roo
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 number e_1 e_2))
        (in-hole P e_2)
        (side-condition (term (nonzero number)))
        "if0f")))
(define-metafunction Roo
  Σ : number ... -> number
  [(Σ number ...) ,(apply + (term (number ...)))])
```

The comma means unquote, so we are just exploiting Racket's + to implement addition in the model

```
(define num-rules
  (reduction-relation
   Roo
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 number e_1 e_2))
        (in-hole P e_2)
        (side-condition (term (nonzero number)))
        "if0f")))
(define-metafunction Roo
  Σ : number ... -> number
  [(Σ number ...) ,(apply + (term (number ...)))])
```

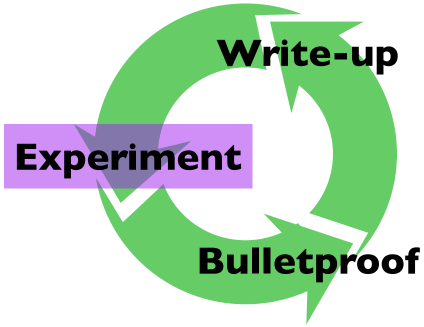
The first if0 rule uses the literal 0 but the second has to use a trivial metafunction (not shown) to test non-zeroness


Write-up

Experiment

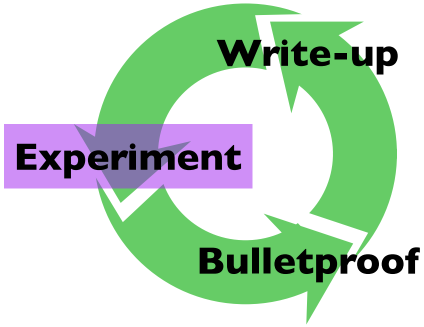
Bulletproof

```
> (apply-reduction-relation
   num-rules
   (term (begin (if0 (+ 1 2) 3 4))))
'((begin (if0 3 3 4)))
```



```
> (traces
   num-rules
   (term (begin (+ (if0 0 1 2)
                   (if0 3 4 5)
                   (if0 6 7 8)))))
```

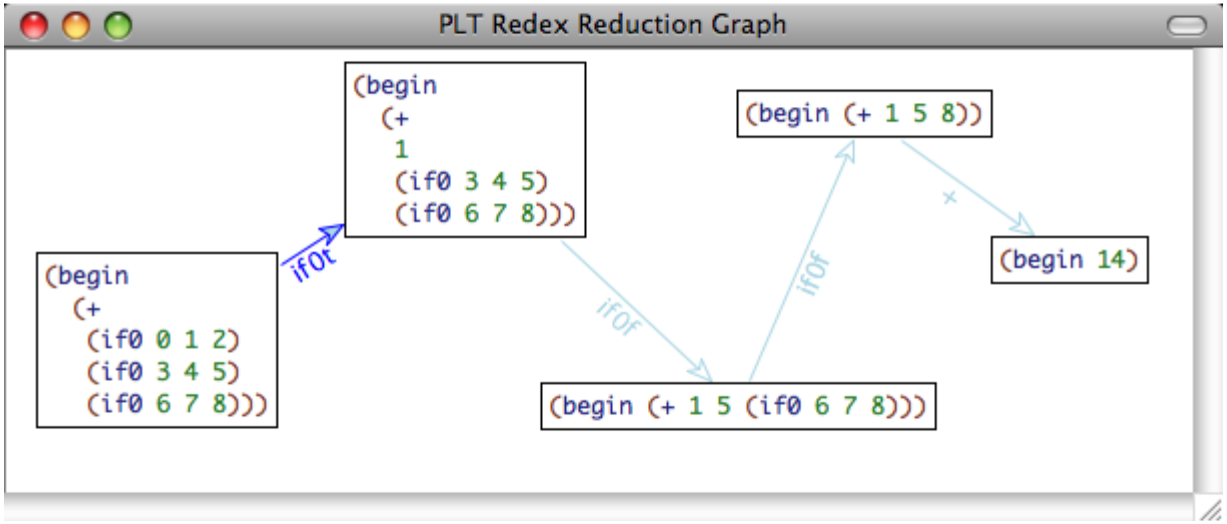
Run



Write-up

Experiment

Bulletproof



```
(begin
  (+
   (if0 0 1 2)
   (if0 3 4 5)
   (if0 6 7 8)))
```

```
(begin
  (+
   1
   (if0 3 4 5)
   (if0 6 7 8)))
```

```
(begin (+ 1 5 (if0 6 7 8)))
```

```
(begin (+ 1 5 8))
```

```
(begin 14)
```

```
(begin                                      ⟶ (begin                                    [send]
  d1 ...                                        d1 ...
  (define z                                     (define z
    (class object%                                (class object%
      (init-field i)                                (init-field i)
      m1 ...                                        m1 ...
      (define/public (x y ...)                      (define/public (x y ...)
        e)                                            e)
      m2 ...                                        m2 ...
      (super-make-object)))                         (super-make-object)))
  d2 ...                                        d2 ...
  E[(send (make-object z vi)                    E[e{y := vy  ...,
          x                                         i := vi,
          vy ...)])                                 this := (make-object z vi)}])
```

```
(begin
  d_1 ...
  (define z
    (class object%
      (init-field i)
      m_1 ...
      (define/public (x y ...) e)
      m_2 ...
      (super-make-object)))
  d_2 ...
  (in-hole E (send (make-object z v_i)
                   x
                   v_y ...)))
```

Begin reading at the bottom, in the main expression; this is a method invocation, with arguments `v_y ...` to an object of class `z` with init field `v_i`

```
(begin
  d_1 ...
  (define z
    (class object%
      (init-field i)
      m_1 ...
      (define/public (x y ...) e)
      m_2 ...
      (super-make-object)))
  d_2 ...
  (in-hole E (send (make-object z v_i)
                   x
                   v_y ...)))
```

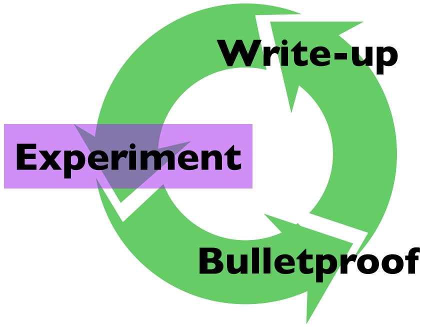
Since the z appears twice, it must be the same identifier; this forces the `d_1` and `d_2` to absorb all of the irrelevant class definitions

```
(begin
  d_1 ...
  (define z
    (class object%
      (init-field i)
      m_1 ...
      (define/public (x y ...) e)
      m_2 ...
      (super-make-object)))
  d_2 ...
  (in-hole E (send (make-object z v_i)
                   x
                   v_y ...)))
```

Ditto for `m_1` and `m_2` absorbing all of the irrelevant methods

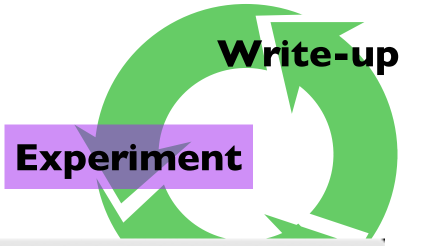
```
(begin
  d_1 ...
  (define z
    (class object%
      (init-field i)
      m_1 ...
      (define/public (x y ...) e)
      m_2 ...
      (super-make-object)))
  d_2 ...
  (in-hole E (substs-e e
                               (y v_y) ...
                               (i v_i)
                               (this
                                (make-object z v_i)))))
```

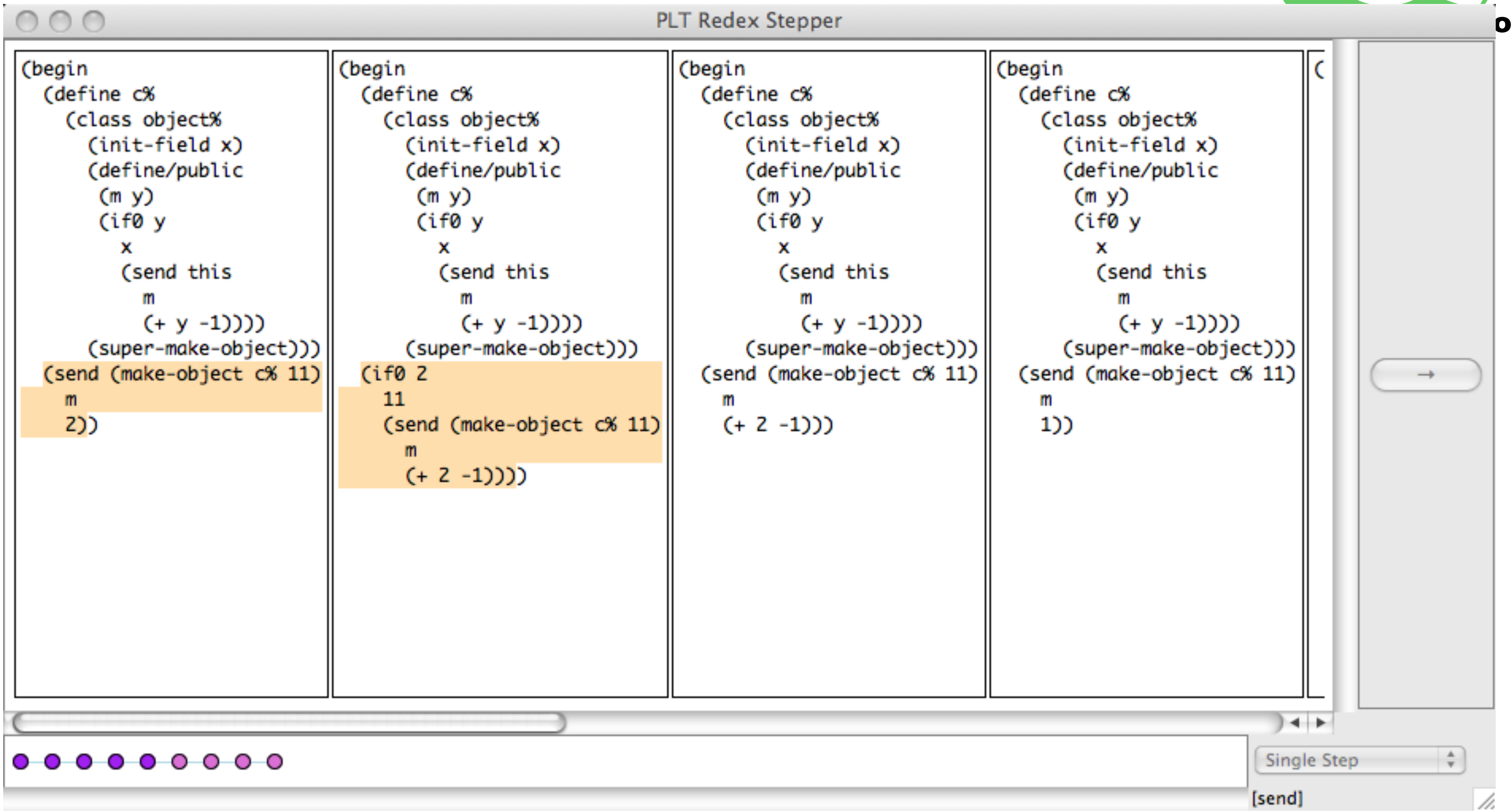
This, the right-hand side of the rule, is just like the left, except we replace method invoation with the body of the method, modulo appropriate substitutions



```
> (stepper
   red
   (term (begin
           (define c%
             (class object%
               (init-field x)
               (define/public (m y)
                 (if0 y
                      x
                      (send this m (+ y -1))))
               (super-make-object)))
           (send (make-object c% 11) m 2))))
```

Run





```
(begin
  (define c%
    (class object%
      (init-field x)
      (define/public
       (m y)
       (if0 y
         x
         (send this
           m
           (+ y -1))))
      (super-make-object)))
  (send (make-object c% 11)
    m
    2))
```

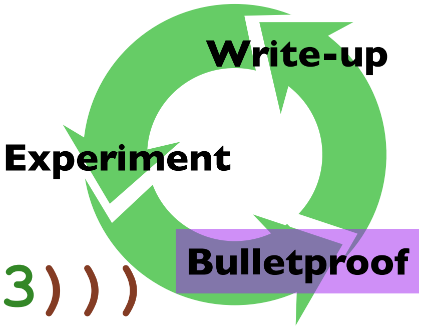
```
(begin
  (define c%
    (class object%
      (init-field x)
      (define/public
       (m y)
       (if0 y
         x
         (send this
           m
           (+ y -1))))
      (super-make-object)))
  (if0 2
    11
    (send (make-object c% 11)
      m
      (+ 2 -1))))
```

```
(begin
  (define c%
    (class object%
      (init-field x)
      (define/public
       (m y)
       (if0 y
         x
         (send this
           m
           (+ y -1))))
      (super-make-object)))
  (send (make-object c% 11)
    m
    (+ 2 -1)))
```

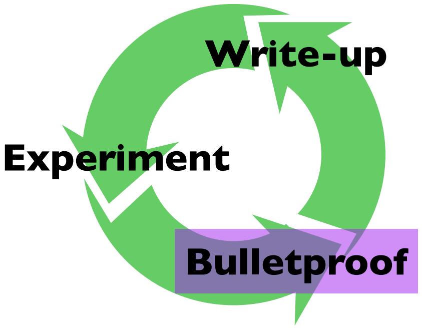
```
(begin
  (define c%
    (class object%
      (init-field x)
      (define/public
       (m y)
       (if0 y
         x
         (send this
           m
           (+ y -1))))
      (super-make-object)))
  (send (make-object c% 11)
    m
    1))
```

```
(
```

[send]

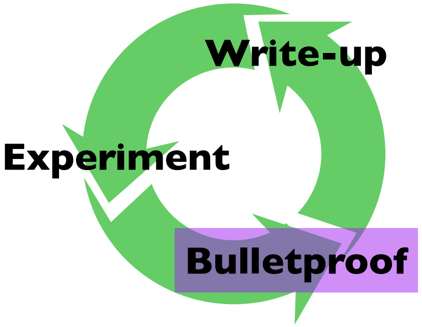


```
> (test-->> red
            (term (begin (if0 1 2 3)))
            (term (begin 3)))
> (test-->> red
            (term (begin (if0 0 1 2)))
            (term (begin 2)))
FAILED slides.rkt:132.11
expected: '(begin 2)
  actual: '(begin 1)
> (test-->> red
            (term (begin (+ 1 2 3 4)))
            (term (begin 10)))
> (test-results)
1 test failed (out of 3 total).
```



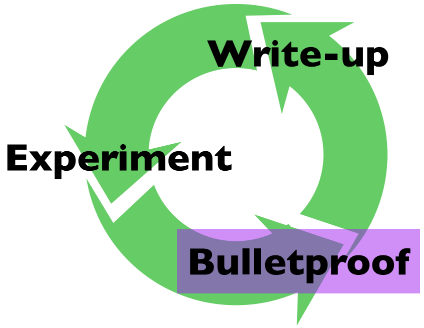
```
> (redex-check
   Roo
   p
   (equal? (eval-expr (term p))
           (reduce-expr (term p))))
```

**`redex-check`** accepts a language, a non-terminal, and a Racket expression, and makes up random examples of the pattern to test the Racket expression, looking for **`#f`**



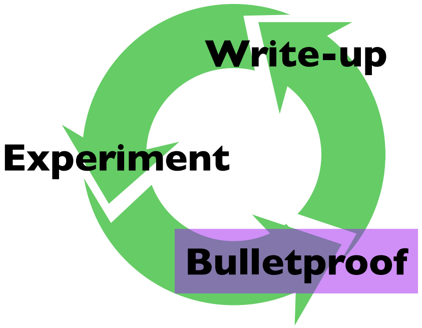
```
> (redex-check
    Roo
    p
    (equal? (eval-expr (term p))
            (reduce-expr (term p))))
```

**`redex-check`** accepts a language, a non-terminal, and a Racket expression, and makes up random examples of the pattern to test the Racket expression, looking for **`#f`**


Write-up
Experiment
Bulletproof

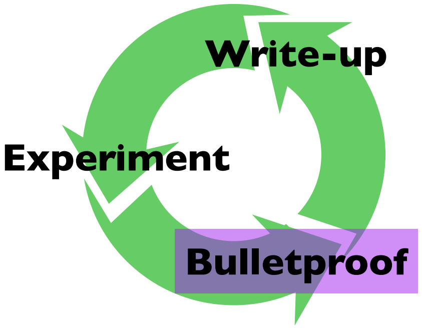
```
> (redex-check
    Roo
    p
     (equal? (eval-expr (term p))
             (reduce-expr (term p))))
```

**`redex-check`** accepts a language, a non-terminal, and a Racket expression, and makes up random examples of the pattern to test the Racket expression, looking for **`#f`**



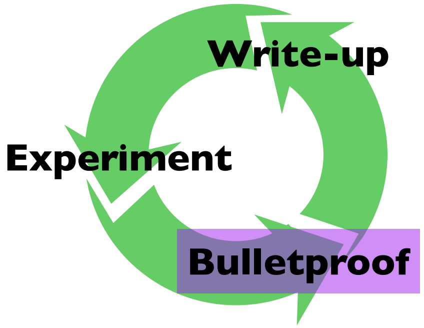
```
> (redex-check
   Roo
   p
   (equal? (eval-expr (term p))
           (reduce-expr (term p))))
```

**`redex-check`** accepts a language, a non-terminal, and a Racket expression, and makes up random examples of the pattern to test the Racket expression, looking for **`#f`**

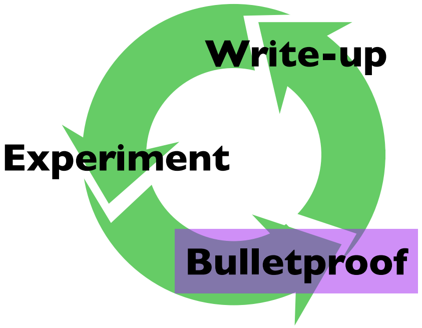


```
> (redex-check
   Roo
   p
   (equal? (eval-expr (term p))
           (reduce-expr (term p))))
```

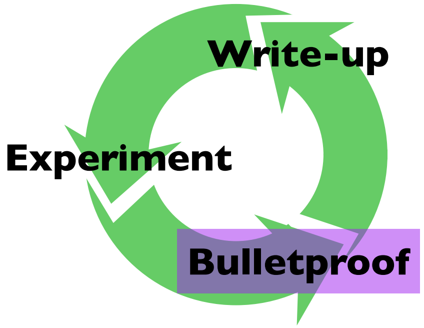
**`redex-check`** accepts a language, a non-terminal, and a Racket expression, and makes up random examples of the pattern to test the Racket expression, looking for **`#f`**



```
; reduce-expr : any -> (or/c 'error
;                            'object
;                            number?)
(define (reduce-expr e)
  (define results (apply-reduction-relation* red e))
  (define result (car results))
  (define value (last result))
  (if (redex-match Roo a result)
      (if (number? value)
          value
          'object)
      'error))

; eval-expr : any -> (or/c 'error 'object number?)
(define (eval-expr e)
  ; evaluate the expression using Racket
   )
```

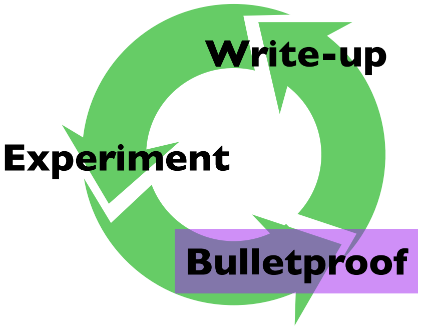


```
> (redex-check
   Roo
   p
   (equal? (eval-expr (term p))
           (reduce-expr (term p))))
```

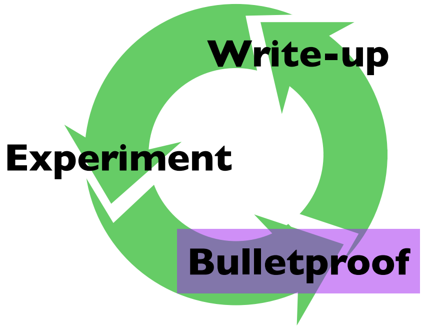


```
> (redex-check
    Roo
    p
     (equal? (eval-expr (term p))
             (reduce-expr (term p))))
```

```
redex-check: checking (begin (define U (class object%
(init-field JJp) (super-make-object))) (define FR (class object%
(init-field x) (define/public (o) this) (define/public (k) this)
(define/public (x) 0) (define/public (f) 6) (define/public (Q) 2)
(super-make-object))) (make-object c (+ (make-object X 3)
(send this hn) 2 (send 2 U) 0 this 0 -2 (make-object C -1) (if0
this 5 6) (+)))) raises an exception:
class: duplicate declared identifier in: x
```

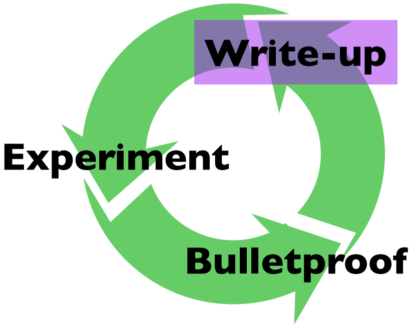


```
> (redex-check
   Roo
   p
   (equal? (eval-expr (term p))
           (reduce-expr (term p))))

redex-check: checking
(begin (define FR (class object%
                    (init-field x)
                    (define/public (x) 0)
                    (super-make-object)))
       0)
raises an exception:
class: duplicate declared identifier in: x
```



```
> (redex-check
   Roo
   p
    (with-handlers ((exn:fail:syntax?
                     (λ (x) #t)))
      (equal? (eval-expr (term p))
              (reduce-expr (term p)))))
```

redex-check: no counterexamples in 1000 attempts



```
> (render-reduction-relation num-rules)
```

$$P[\,\texttt{(+}\ number\ ...\texttt{)}\,] \qquad [+]$$

$$\longrightarrow P[\Sigma[\![number\ ...]\!]]$$

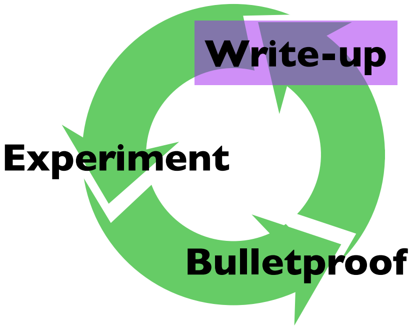
$$P[\,\texttt{(if0}\ 0\ e_1\ e_2\texttt{)}\,] \qquad [\text{if0t}]$$

$$\longrightarrow P[e_1]$$

$$P[\,\texttt{(if0}\ number\ e_1\ e_2\texttt{)}\,] \quad [\text{if0f}]$$

$$\longrightarrow P[e_2]$$

$$\text{where } \mathsf{nonzero}[\![number]\!]$$



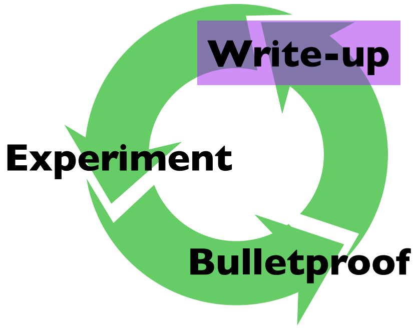
```
> (parameterize ([rule-pict-style 'horizontal])
    (render-reduction-relation num-rules))
```

$$P[\,\texttt{(+}\ \mathit{number}\ \ldots\texttt{)}\,] \longrightarrow P[\Sigma[\![\mathit{number}\ \ldots]\!]] \quad \text{[+]}$$

$$P[\,\texttt{(if0}\ 0\ e_1\ e_2\texttt{)}\,] \longrightarrow P[e_1] \quad \text{[if0t]}$$

$$P[\,\texttt{(if0}\ \mathit{number}\ e_1\ e_2\texttt{)}\,] \longrightarrow P[e_2] \quad \text{[if0f]}$$

$$\text{where nonzero}[\![\mathit{number}]\!]$$



```
> (parameterize ([rule-pict-style 'horizontal])
    (with-rewriters
     (render-reduction-relation num-rules)))
```

$$P[\,\texttt{(+}\ \mathit{number}\ \ldots\texttt{)}\,] \longrightarrow P[\Sigma[\![\mathit{number}\ \ldots]\!]] \quad \text{[+]}$$

$$P[\,\texttt{(if0}\ 0\ e_1\ e_2\texttt{)}\,] \longrightarrow P[e_1] \quad \text{[if0t]}$$

$$P[\,\texttt{(if0}\ \mathit{number}\ e_1\ e_2\texttt{)}\,] \longrightarrow P[e_2] \quad \text{[if0f]}$$

$$\text{where } \mathit{number} \neq 0$$

**Assignment:** Generalize the model to first-class classes (i.e., classes as values)

The next few slides show a model that we went through together in the live talk. It starts with the complete version of the first-order calculus and then adjusts definitions so that any expression can appear on the right-hand side, and then makes classes be expressions. It then exploits Redex's testing facilities to find bugs in the model.

We start with the complete code from the model but with two changes:

- `e` now contains `c` and
- definitions are of the form `(define x e)`.

# 0.rkt

```
      p ::= (begin d ... e)
      d ::= (define z c)
      c ::= (class object%
              (init-field i)
              m ...
              (super-make-object))
      m ::= (define/public (x y ...)
              e)
      e ::= (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin d ... v)
      v ::= (make-object x v)
          | number
      P ::= (begin d ... E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                   ⟶ (begin                                   [send]
  d1 ...                                      d1 ...
   (define z                                   (define z
     (class object%                              (class object%
       (init-field i)                              (init-field i)
       m1 ...                                      m1 ...
       (define/public (x y ...)                    (define/public (x y ...)
         e)                                          e)
       m2 ...                                      m2 ...
       (super-make-object)))                       (super-make-object)))
  d2 ...                                      d2 ...
  E[(send (make-object z vi)                  E[e{y := vy  ...,
          x                                       i := vi,
          vy ...)])                               this := (make-object z vi)}])
```

```
   P[(+ number ...)]              [+]
⟶ P[Σ[[number ...]]]

   P[(if0 0 e1 e2)]               [if0t]
⟶ P[e1]

   P[(if0 number e1 e2)]          [if0f]
⟶ P[e2]
    where number ≠ 0
```

# 1.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
              (init-field i)
              m ...
              (super-make-object))
      m ::= (define/public (x y ...)
              e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin d ... v)
      v ::= (make-object x v)
          | number
      P ::= (begin d ... E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin
  d_1 ...
   (define z
     (class object%
       (init-field i)
       m_1 ...
       (define/public (x y ...)
         e)
       m_2 ...
       (super-make-object)))
  d_2 ...
  E[(send (make-object z v_i)
          x
          v_y ...)])
```

```
--> (begin                                          [send]
      d_1 ...
       (define z
         (class object%
           (init-field i)
           m_1 ...
           (define/public (x y ...)
             e)
           m_2 ...
           (super-make-object)))
      d_2 ...
      E[e{y := v_y  ...,
          i := v_i,
          this := (make-object z v_i)}])
```

```
    P[(+ number ...)]                [+]
--> P[Σ[[number ...]]]

    P[(if0 0 e_1 e_2)]               [if0t]
--> P[e_1]

    P[(if0 number e_1 e_2)]          [if0f]
--> P[e_2]
     where number ≠ 0
```

# 1.rkt Error

```
        p ::= (begin d ... e)
        d ::= (define z e)
        c ::= (class object%
                (init-field i)
                m ...
                (super-make-object))
        m ::= (define/public (x y ...)
                e)
        e ::= c
            | (make-object x e)
            | (send e x e ...)
            | x
            | this
            | number
            | (if0 e e e)
            | (+ e ...)
        a ::= (begin d ... v)
        v ::= (make-object x v)
            | number
        P ::= (begin d ... E)
        E ::= (make-object x E)
            | (send E x e ...)
            | (send v x v ... E e ...)
            | (if0 E e e)
            | (+ v ... E e ...)
            | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                    ⟶ (begin                                        [send]
  d1 ...                                     d1 ...
  (define z                                  (define z
    (class object%                             (class object%
      (init-field i)                             (init-field i)
      m1 ...                                     m1 ...
      (define/public (x y ...)                   (define/public (x y ...)
        e)                                         e)
      m2 ...                                     m2 ...
      (super-make-object)))                      (super-make-object)))
  d2 ...                                     d2 ...
  E[(send (make-object z vi)                 E[e{y := vy ...,
          x                                      i := vi,
          vy ...)])                              this := (make-object z vi)}])

P[(+ number ...)]         [+]
⟶ P[Σ[[number ...]]]

P[(if0 0 e1 e2)]          [if0t]
⟶ P[e1]

P[(if0 number e1 e2)]     [if0f]
⟶ P[e2]
   where number ≠ 0
```

```
(begin
  (class object%
    (init-field bB)
    (super-make-object)))
```

# 2.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
              (init-field i)
              m ...
              (super-make-object))
      m ::= (define/public (x y ...)
              e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin d ... v)
      v ::= (make-object x v)
          | number
          | c
      P ::= (begin d ... E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin
  $d_1$ ...
  (define z
    (class object%
      (init-field i)
      $m_1$ ...
      (define/public (x y ...)
        e)
      $m_2$ ...
      (super-make-object)))
  $d_2$ ...
  E[(send (make-object z $v_i$)
          x
          $v_y$ ...)])
```

```
--> (begin                                        [send]
      $d_1$ ...
      (define z
        (class object%
          (init-field i)
          $m_1$ ...
          (define/public (x y ...)
            e)
          $m_2$ ...
          (super-make-object)))
      $d_2$ ...
      E[e{y := $v_y$  ...,
          i := $v_i$,
          this := (make-object z $v_i$)}])
```

```
    P[(+ number ...)]              [+]
--> P[Σ[[number ...]]]

    P[(if0 0 $e_1$ $e_2$)]         [if0t]
--> P[$e_1$]

    P[(if0 number $e_1$ $e_2$)]    [if0f]
--> P[$e_2$]
     where number ≠ 0
```

# 2.rkt Error

```
p ::= (begin d ... e)
d ::= (define z e)
c ::= (class object%
        (init-field i)
        m ...
        (super-make-object))
m ::= (define/public (x y ...)
        e)
e ::= c
    | (make-object x e)
    | (send e x e ...)
    | x
    | this
    | number
    | (if0 e e e)
    | (+ e ...)
a ::= (begin d ... v)
v ::= (make-object x v)
    | number
    | c
P ::= (begin d ... E)
E ::= (make-object x E)
    | (send E x e ...)
    | (send v x v ... E e ...)
    | (if0 E e e)
    | (+ v ... E e ...)
    | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                  → (begin                                    [send]
 d1 ...                                     d1 ...
  (define z                                  (define z
   (class object%                             (class object%
    (init-field i)                             (init-field i)
    m1 ...                                     m1 ...
    (define/public (x y ...)                   (define/public (x y ...)
      e)                                         e)
    m2 ...                                     m2 ...
    (super-make-object)))                      (super-make-object)))
 d2 ...                                     d2 ...
 E[(send (make-object z vi)                 E[e{y := vy  ...,
         x                                      i := vi,
         vy ...)])                              this := (make-object z vi)}])
```

```
    P[(+ number ...)]            [+]
 →  P[Σ[[number ...]]]

    P[(if0 0 e1 e2)]             [if0t]
 →  P[e1]

    P[(if0 number e1 e2)]   [if0f]
 →  P[e2]
     where number ≠ 0
```

```
(begin
  (define c (send 0 G))
  3)
```

# 3.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
              (init-field i)
              m ...
              (super-make-object))
      m ::= (define/public (x y ...)
              e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object x v)
          | number
          | c
      P ::= (begin d ... E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                  ⟶ (begin                                  [send]
  d_1 ...                                   d_1 ...
   (define z                                 (define z
     (class object%                            (class object%
       (init-field i)                            (init-field i)
       m_1 ...                                   m_1 ...
       (define/public (x y ...)                  (define/public (x y ...)
         e)                                        e)
       m_2 ...                                   m_2 ...
       (super-make-object)))                     (super-make-object)))
  d_2 ...                                   d_2 ...
  E[(send (make-object z v_i)               E[e{y := v_y  ...,
          x                                     i := v_i,
          v_y ...)])                            this := (make-object z v_i)}])
```

```
   P[(+ number ...)]            [+]
⟶ P[Σ[[number ...]]]

   P[(if0 0 e_1 e_2)]           [if0t]
⟶ P[e_1]

   P[(if0 number e_1 e_2)]      [if0f]
⟶ P[e_2]
    where number ≠ 0
```

# 3.rkt Error

```
        p ::= (begin d ... e)
        d ::= (define z e)
        c ::= (class object%
                (init-field i)
                m ...
                (super-make-object))
        m ::= (define/public (x y ...)
                e)
        e ::= c
            | (make-object x e)
            | (send e x e ...)
            | x
            | this
            | number
            | (if0 e e e)
            | (+ e ...)
        a ::= (begin (define x v) ... v)
        v ::= (make-object x v)
            | number
            | c
        P ::= (begin d ... E)
        E ::= (make-object x E)
            | (send E x e ...)
            | (send v x v ... E e ...)
            | (if0 E e e)
            | (+ v ... E e ...)
            | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                  → (begin                              [send]
  d₁ ...                                    d₁ ...
  (define z                                 (define z
    (class object%                            (class object%
      (init-field i)                            (init-field i)
      m₁ ...                                    m₁ ...
      (define/public (x y ...)                  (define/public (x y ...)
        e)                                        e)
      m₂ ...                                    m₂ ...
      (super-make-object)))                     (super-make-object)))
  d₂ ...                                    d₂ ...
  E[(send (make-object z vᵢ)                E[e{y := v_y  ...,
          x                                       i := vᵢ,
          v_y ...)])                              this := (make-object z vᵢ)}])
```

```
  P[(+ number ...)]           [+]
→ P[Σ[[number ...]]]

  P[(if0 0 e₁ e₂)]            [if0t]
→ P[e₁]

  P[(if0 number e₁ e₂)]       [if0f]
→ P[e₂]
   where number ≠ 0
```

```
(begin (define g (+)) (+))
```

# 4.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
               (init-field i)
               m ...
               (super-make-object))
      m ::= (define/public (x y ...)
               e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object x v)
          | number
          | c
      P ::= (begin (define x v) ...
                   (define x E)
                   (define x e) ...
                  e)
          | (begin (define x v) ...
                  E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                ⟶ (begin                                    [send]
  d_1 ...                                 d_1 ...
   (define z                               (define z
     (class object%                          (class object%
       (init-field i)                          (init-field i)
       m_1 ...                                 m_1 ...
       (define/public (x y ...)                (define/public (x y ...)
         e)                                      e)
       m_2 ...                                 m_2 ...
       (super-make-object)))                   (super-make-object)))
  d_2 ...                                 d_2 ...
  E[(send (make-object z v_i)             E[e{y := v_y  ...,
              x                                i := v_i,
              v_y ...)])                       this := (make-object z v_i)}])
```

```
    P[(+ number ...)]              [+]
⟶  P[Σ[[number ...]]]

    P[(if0 0 e_1 e_2)]             [if0t]
⟶  P[e_1]

    P[(if0 number e_1 e_2)]        [if0f]
⟶  P[e_2]
     where number ≠ 0
```

# 4.rkt Error

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
               (init-field i)
               m ...
               (super-make-object))
      m ::= (define/public (x y ...)
               e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object x v)
          | number
          | c
      P ::= (begin (define x v) ...
                   (define x E)
                   (define x e) ...
                   e)
          | (begin (define x v) ...
                   E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                      → (begin                                      [send]
 d1 ...                                        d1 ...
  (define z                                     (define z
   (class object%                                (class object%
    (init-field i)                                (init-field i)
     m1 ...                                        m1 ...
     (define/public (x y ...)                      (define/public (x y ...)
      e)                                            e)
     m2 ...                                        m2 ...
     (super-make-object)))                         (super-make-object)))
                                               d2 ...
 E[(send (make-object z vi)                    E[e{y := vy ....,
  x                                               i := vi,
  ...)]                                           this := (make-object z vi)}])

P[(+ ...)]                      [+]
→ P[Σ[[number ...]]]
P[(if0 0 e1 e2)]                [if0t]
→ P[e1]
P[(if0 number e1 e2)]           [if0f]
→ P[e2]
 where number ≠ 0
```

```
(begin
  (define R
    (make-object R
      (class
       object%
       (init-field h)
       (super-make-object))))
 7)
```

# 5.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
               (init-field i)
               m ...
               (super-make-object))
      m ::= (define/public (x y ...)
               e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object c v)
          | number
          | c
      P ::= (begin (define x v) ...
                   (define x E)
                   (define x e) ...
                   e)
          | (begin (define x v) ...
                   E)
      E ::= (make-object x E)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
  (begin                                  --> (begin                             [send]
    d_1 ...                                     d_1 ...
    (define z                                   (define z
      (class object%                              (class object%
        (init-field i)                              (init-field i)
        m_1 ...                                     m_1 ...
        (define/public (x y ...)                    (define/public (x y ...)
          e)                                          e)
        m_2 ...                                     m_2 ...
        (super-make-object)))                       (super-make-object)))
    d_2 ...                                     d_2 ...
    E[(send (make-object z v_i)                 E[e{y := v_y  ...,
              x                                     i := v_i,
              v_y ...)])                            this := (make-object z v_i)}])
```

```
    P[(+ number ...)]            [+]
--> P[Σ[[number ...]]]

    P[(if0 0 e_1 e_2)]           [if0t]
--> P[e_1]

    P[(if0 number e_1 e_2)]      [if0f]
--> P[e_2]
     where number ≠ 0
```

# 5.rkt Error

```
p ::= (begin d ... e)
d ::= (define z e)
c ::= (class object%
        (init-field i)
        m ...
        (super-make-object))
m ::= (define/public (x y ...)
        e)
e ::= c
    | (make-object x e)
    | (send e x e ...)
    | x
    | this
    | number
    | (if0 e e e)
    | (+ e ...)
a ::= (begin (define x v) ... v)
v ::= (make-object c v)
    | number
    | c
P ::= (begin (define x v) ...
             (define x E)
             (define x e) ...
             e)
    | (begin (define x v) ...
             E)
E ::= (make-object x E)
    | (send E x e ...)
    | (send v x v ... E e ...)
    | (if0 E e e)
    | (+ v ... E e ...)
    | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
(begin                                  --> (begin                                 [send]
  d_1 ...                                     d_1 ...
  (define z                                   (define z
    (class object%                              (class object%
      (init-field i)                              (init-field i)
      m_1 ...                                     m_1 ...
      (define/public (x y ...)                    (define/public (x y ...)
        e)                                          e)
      m_2 ...                                     m_2 ...
      (super-make-object)))                       (super-make-object)))
  d_2 ...                                     d_2 ...
  E[(send (make-object z v_i)                 E[e{y := v_y  ...,
          x                                       i := v_i,
          v_y ...)])                              this := (make-object z v_i)}])
```

```
    P[(+ number ...)]          [+]
--> P[Σ[[number ...]]]

    P[(if0 0 e_1 e_2)]         [if0t]
--> P[e_1]

    P[(if0 number e_1 e_2)] [if0f]
--> P[e_2]
     where number ≠ 0
```

Test case (on line 196)

# 6.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
              (init-field i)
              m ...
              (super-make-object))
      m ::= (define/public (x y ...)
              e)
      e ::= c
          | (make-object x e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object c v)
          | number
          | c
      P ::= (begin (define x v) ...
                   (define x E)
                   (define x e) ...
                   e)
          | (begin (define x v) ...
                   E)
      E ::= (make-object v E)
          | (make-object E e)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
P[(send (make-object                      ⟶ P[e{y := v_y ...,                                   [send]
          (class object%                        i := v_i,
            (init-field i)                      this := (make-object
            m_1 ...                                        (class object%
            (define/public (x y ...)                         (init-field i)
              e)                                             m_1 ...
            m_2 ...                                          (define/public (x y ...)
            (super-make-object))                               e)
        v_i)                                                 m_2 ...
      x                                                      (super-make-object))
      v_y ...)]                                            v_i)}]
```

```
  P[(+ number ...)]         [+]      (begin d_1 ...          ⟶ (begin d_1 ...
⟶ P[Σ[[number ...]]]                        (define x v)              (define x v)
                                            d_2 ...                   d_2 ...
  P[(if0 0 e_1 e_2)]        [if0t]          E[x])                     E[v])
⟶ P[e_1]

  P[(if0 number e_1 e_2)]   [if0f]
⟶ P[e_2]
   where number ≠ 0
```

# 6.rkt Error

```
        p ::= (begin d ... e)
        d ::= (define z e)
        c ::= (class object%
                (init-field i)
                m ...
                (super-make-object))
        m ::= (define/public (x y ...)
                e)
        e ::= c
            | (make-object x e)
            | (send e x e ...)
            | x
            | this
            | number
            | (if0 e e e)
            | (+ e ...)
        a ::= (begin (define x v) ... v)
        v ::= (make-object c v)
            | number
            | c
        P ::= (begin (define x v) ...
                     (define x E)
                     (define x e) ...
                     e)
            | (begin (define x v) ...
                     E)
        E ::= (make-object v E)
            | (make-object E e)
            | (send E x e ...)
            | (send v x v ... E e ...)
            | (if0 E e e)
            | (+ v ... E e ...)
            | []
  x, y, z ::= variable-not-otherwise-mentioned
```

```
P[(send (make-object                     ⟶ P[e{y := v_y ...,                          [send]
          (class object%                          i := v_i,
            (init-field i)                        this := (make-object
            m_1 ...                                          (class object%
            (define/public (x y ...)                           (init-field i)
              e)                                               m_1 ...
            m_2 ...                                            (define/public (x y ...)
            (super-make-object))                                 e)
        v_i)                                                   m_2 ...
      x                                                        (super-make-object))
      v_y ...)]                                             v_i) }]

P[(+ number ...)]               [+]     (begin d_1 ...                ⟶ (begin d_1 ...
⟶ P[Σ[[number ...]]]                      (define x v)                    (define x v)
                                          d_2 ...                         d_2 ...
P[(if0 0 e_1 e_2)]              [if0t]    E[x])                           E[v])
⟶ P[e_1]

P[(if0 number e_1 e_2)]         [if0f]
⟶ P[e_2]
  where number ≠ 0
```

substs-e gets a non-“e” as argument in test cases

# 7.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
               (init-field i)
               m ...
               (super-make-object))
      m ::= (define/public (x y ...)
               e)
      e ::= c
          | (make-object e e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object c v)
          | number
          | c
      P ::= (begin (define x v) ...
                   (define x E)
                   (define x e) ...
                   e)
          | (begin (define x v) ...
                   E)
      E ::= (make-object v E)
          | (make-object E e)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
P[(send (make-object                          ⟶ P[e{y := v_y  ...,                          [send]
              (class object%                        i := v_i,
                (init-field i)                      this := (make-object
                m_1 ...                                       (class object%
                (define/public (x y ...)                        (init-field i)
                  e)                                            m_1 ...
                m_2 ...                                         (define/public (x y ...)
                (super-make-object))                              e)
            v_i)                                                m_2 ...
          x                                                     (super-make-object))
          v_y ...)]                                           v_i)}]
```

```
   P[(+ number ...)]          [+]
⟶ P[Σ[[number ...]]]

   P[(if0 0 e_1 e_2)]         [if0t]
⟶ P[e_1]

   P[(if0 number e_1 e_2)]    [if0f]
⟶ P[e_2]
    where number ≠ 0
```

```
(begin d_1 ...            ⟶ (begin d_1 ...
       (define x v)                (define x v)
       d_2 ...                     d_2 ...
       E[x])                       E[v])
```

# 7.rkt Error

```
    p ::= (begin d ... e)
    d ::= (define z e)
    c ::= (class object%
            (init-field i)
            m ...
            (super-make-object))
    m ::= (define/public (x y ...)
            e)
    e ::= c
        | (make-object e e)
        | (send e x e ...)
        | x
        | this
        | number
        | (if0 e e e)
        | (+ e ...)
    a ::= (begin (define x v) ... v)
    v ::= (make-object c v)
        | number
        | c
    P ::= (begin (define x v) ...
                 (define x E)
                 (define x e) ...
                 e)
        | (begin (define x v) ...
                 E)
    E ::= (make-object v E)
        | (make-object E e)
        | (send E x e ...)
        | (send v x v ... E e ...)
        | (if0 E e e)
        | (+ v ... E e ...)
        | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
P[(send (make-object               → P[e{y := vy ...,                    [send]
           (class object%                i := vi,
             (init-field i)              this := (make-object
             m1 ...                                (class object%
             (define/public (x y ...)                (init-field i)
               e)                                    m1 ...
             m2 ...                                  (define/public (x y ...)
             (super-make-object))                      e)
           vi)                                       m2 ...
```

```
(begin
  (define z 0)
  (define HV z)
  (class object%
    (init-field ÷)
    (super-make-object)))
```

```
                                            (define x v)
                                            d2 ...
                                            E[v])
```

# 8.rkt

```
      p ::= (begin d ... e)
      d ::= (define z e)
      c ::= (class object%
               (init-field i)
               m ...
               (super-make-object))
      m ::= (define/public (x y ...)
               e)
      e ::= c
          | (make-object e e)
          | (send e x e ...)
          | x
          | this
          | number
          | (if0 e e e)
          | (+ e ...)
      a ::= (begin (define x v) ... v)
      v ::= (make-object c v)
          | number
          | c
      P ::= (begin (define x v) ...
                   (define x E)
                   (define x e) ...
                   e)
          | (begin (define x v) ...
                   E)
      E ::= (make-object v E)
          | (make-object E e)
          | (send E x e ...)
          | (send v x v ... E e ...)
          | (if0 E e e)
          | (+ v ... E e ...)
          | []
x, y, z ::= variable-not-otherwise-mentioned
```

```
P[(send (make-object                     ⟶ P[e{y := v_y  ...,                        [send]
           (class object%                       i := v_i,
             (init-field i)                     this := (make-object
             m_1 ...                                        (class object%
             (define/public (x y ...)                         (init-field i)
               e)                                             m_1 ...
             m_2 ...                                          (define/public (x y ...)
             (super-make-object))                               e)
         v_i)                                                 m_2 ...
       x                                                      (super-make-object))
       v_y ...)]                                            v_i)}]
```

```
  P[(+ number ...)]           [+]
⟶ P[Σ[[number ...]]]

  P[(if0 0 e_1 e_2)]          [if0t]
⟶ P[e_1]

  P[(if0 number e_1 e_2)]     [if0f]
⟶ P[e_2]
   where number ≠ 0
```

```
(begin d_1 ...                ⟶ (begin d_1 ...
          (define x v)                  (define x v)
          d_2 ...                       d_2 ...
          E[x])                         E[v])

(begin d_1 ...                ⟶ (begin d_1 ...
          (define x v)                  (define x v)
          d_2 ...                       d_2 ...
          (define y E[x])               (define y E[v])
          d_3 ...                       d_3 ...
          e)                            e)
```

# Racket’s class system

**There's more to the Racket class system, e.g.:**

- The superclass position is an expression ⇒ mixins
- `define-local-member-name`

  (exploiting scope for abstraction)
- inner + super

See Flatt et al [APLAS 2006] or the docs for more

Your turn

# Pair Programming

**`http://bit.ly/sinaia`**