Compiling to λ

λ-calculus and church encodings

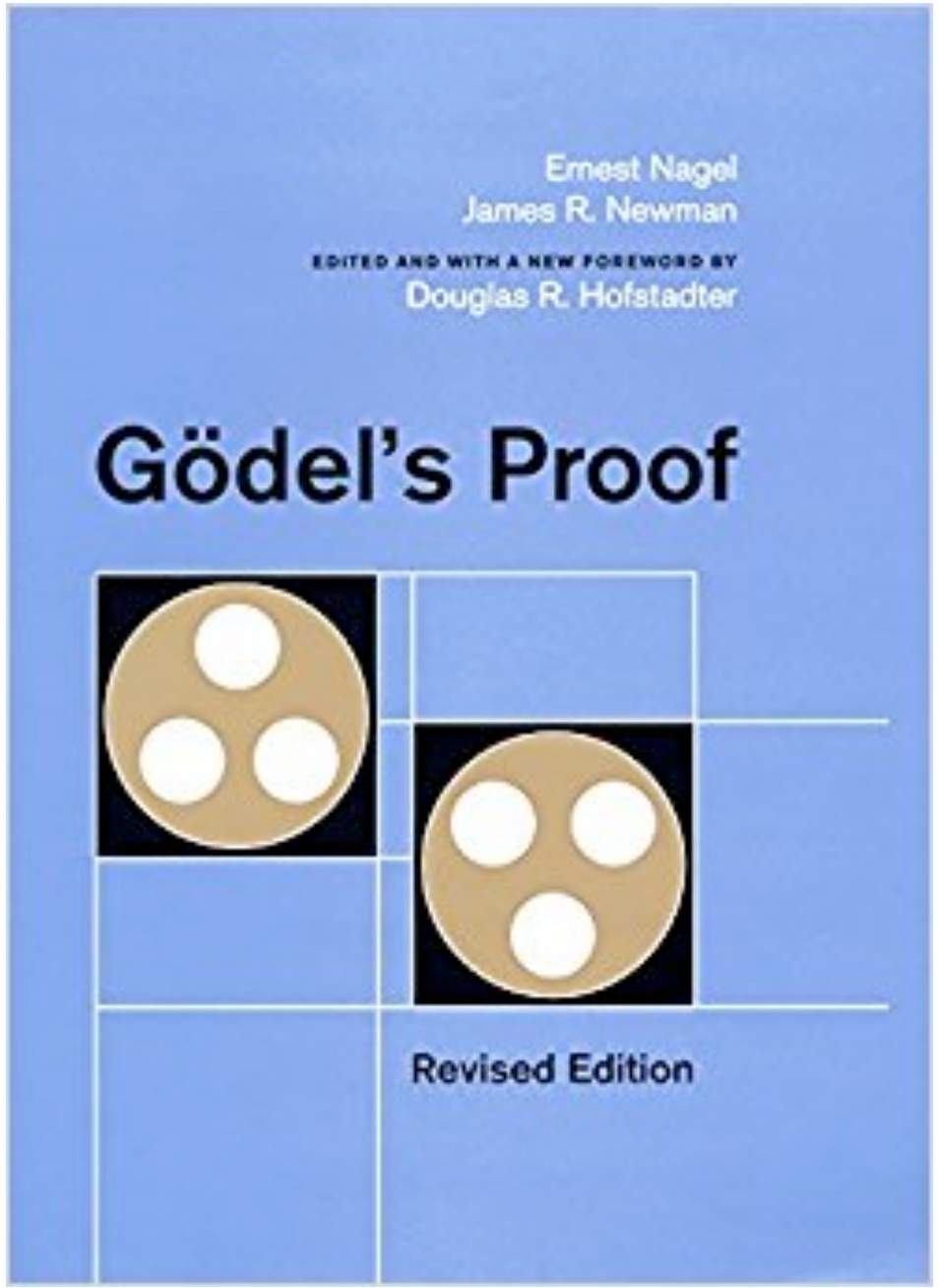
```
e ::= (lambda (x) e)
    | (e e)
    | x
```

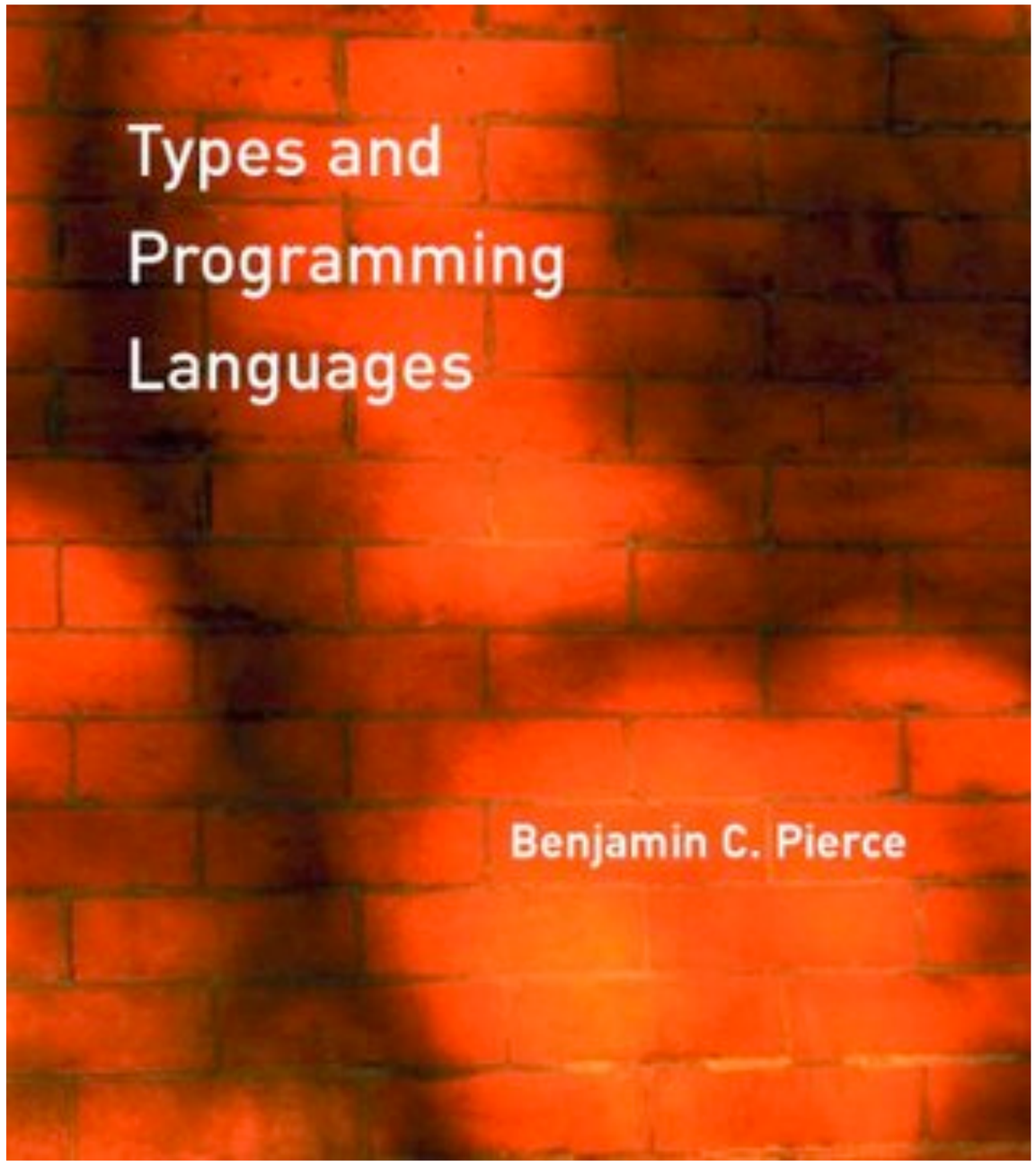
Alonzo Church

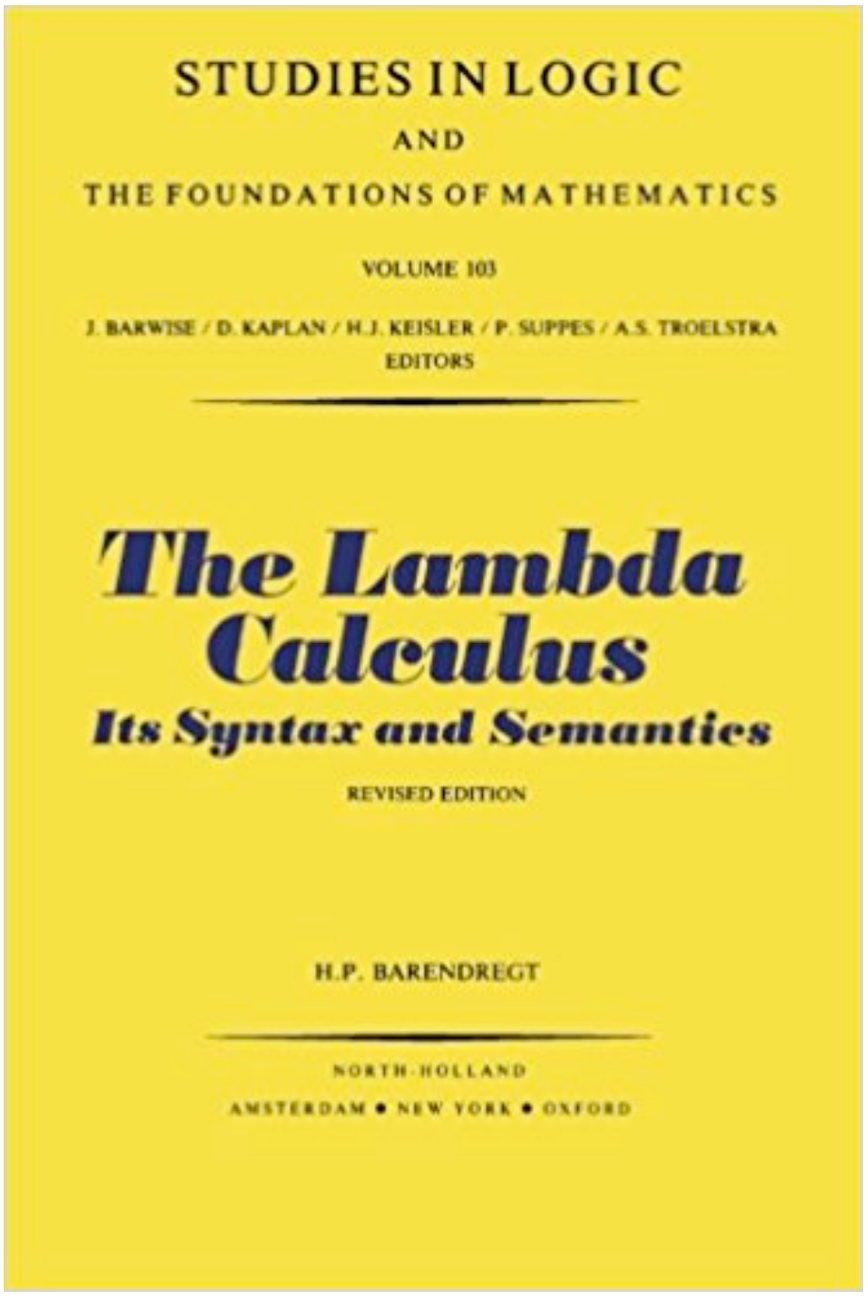
$$\Omega = (U\ U) = ((\lambda\ (u)\ (u\ u))\ (\lambda\ (u)\ (u\ u)))$$

$$\downarrow \beta$$

$$(u\ u)[u \leftarrow U] = ((\lambda\ (u)\ (u\ u))\ (\lambda\ (u)\ (u\ u)))$$


Ernest Nagel
James R. Newman
EDITED AND WITH A NEW FOREWORD BY
Douglas R. Hofstadter
Gödel's Proof
Revised Edition


Types and
Programming
Languages
Benjamin C. Pierce


STUDIES IN LOGIC
AND
THE FOUNDATIONS OF MATHEMATICS
VOLUME 103
J. BARWISE / D. KAPLAN / H.J. KEISLER / P. SUPPES / A.S. TROELSTRA
EDITORS
The Lambda Calculus
Its Syntax and Semanties
REVISED EDITION
H.P. BARENDREGT
NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD

Church encoding

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (+ e e) | (* e e)
    | (cons e e) | (car e) | (cdr e)
    | d
d ::= ℕ | #t | #f | '()
x ::= <vars>
```

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

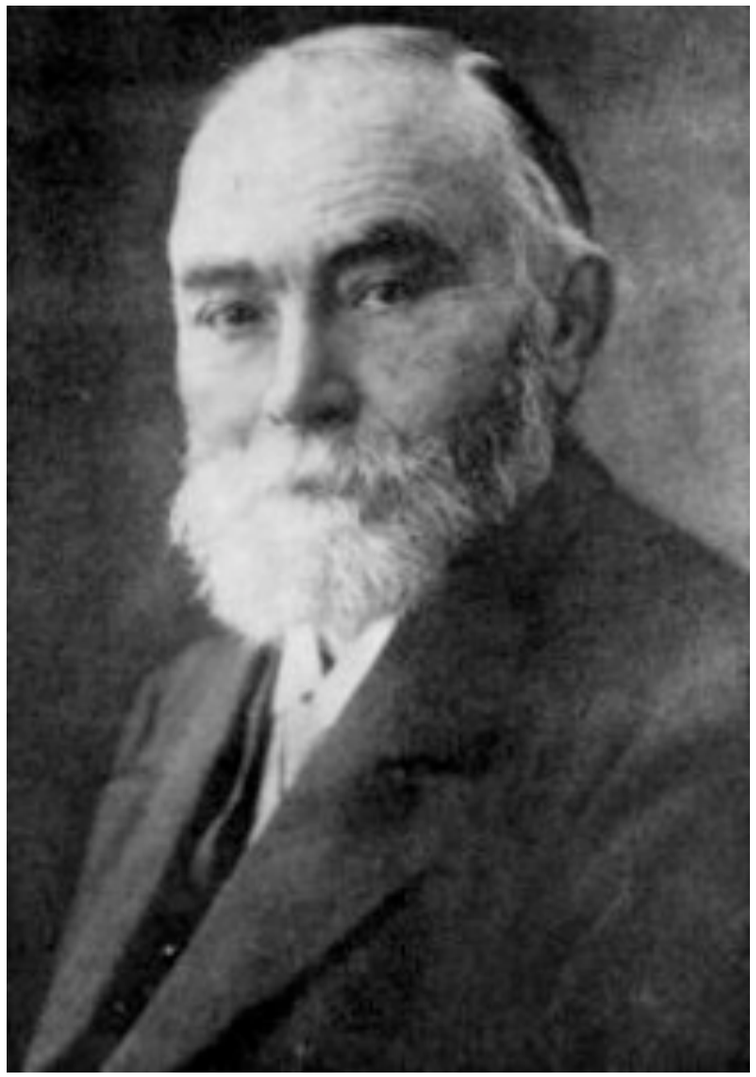
Desugaring Let

```
(let ([x e] …) ebody)
```

```
(let ([x e] …) ebody)
```

```
((λ (x …) ebody) e …)
```

Currying



Gottlob Frege

Moses Schönfinkel

```
(λ (x y z) e)  ⟶  (λ (x)
                    (λ (y)
                      (λ (z) e)))
```

```
(λ (x) e)  ⟶  (λ (x) e)
```

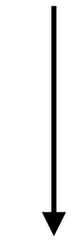
```
(λ () e)  ⟶  (λ (_) e)
```

```
(f a b c d)  ⟶  ((((f a) b) c) d)

(f a)        ⟶  (f a)

(f)          ⟶  (f f)
```

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | (if e e e)
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ℕ | #t | #f | '()
x ::= <vars>
```

Conditionals & Booleans

(if #t e_T e_F)



e_T

(if #f e_T e_F)

e_F

$((\lambda\ (t\ f)\ t)\ e_T\ e_F)$ $((\lambda\ (t\ f)\ f)\ e_T\ e_F)$

$((\lambda\ (t\ f)\ t)\ v_T\ v_F)$ $((\lambda\ (t\ f)\ f)\ v_T\ v_F)$

v_T v_F

$$((\lambda\ (t\ f)\ t)\ e_T\ \Omega)$$

. . . . . .

$((\lambda\ (t\ f)\ (t))\ (\lambda\ ()\ e_T)\ (\lambda\ ()\ \Omega))$

$((\lambda\ ()\ e_T))$

e_T

v_T

Turn all values into verbs!

(Focus on the *behaviors* that are implicit in values.)

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= ℕ | '()
x ::= <vars>
```

Natural Numbers

$$(\lambda\ (f)\ (\lambda\ (x)\ (f^N\ x)))$$

0: $(\lambda\ (f)\ (\lambda\ (x)\ x))$

1: $(\lambda\ (f)\ (\lambda\ (x)\ (f\ x)))$

2: $(\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ x))))$

3: $(\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ (f\ x)))))$

```
church+ = (λ (n) (λ (m)
(λ (f) (λ (x)
   …))))
```

```
church+ = (λ (n) (λ (m)
          (λ (f) (λ (x)
            ((n f) ((m f) x))))))
```

```
church* = (λ (n) (λ (m)
(λ (f) (λ (x)
    …))))
```

```
church* = (λ (n) (λ (m)
          (λ (f) (λ (x)
            ((n (m f)) x)))))
```

$$f^{N^M} = f^{N*M}$$

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((cons e) e) | (car e) | (cdr e)
    | d
d ::= '()
x ::= <vars>
```

Lists

The fundamental problem:

We need to be able to case-split.

The solution:

We take two callbacks as with `#t`, `#f`!

```
‘() = (λ (when-cons) (λ (when-null)
          (when-null)))
```

```
(cons a b) = (λ (when-cons) (λ (when-null)
               (when-cons a b)))
```

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
x ::= <vars>
```

Y combinator

```
(letrec ([fact (λ (n)
                 (if (= n 0)
                     1
                     (* n (fact (- n 1)))))])
  (fact 5))
```

```
(letrec ([fact (λ (fact) (λ (n)
                           (if (= n 0)
                               1
                               (* n (fact (- n 1))))))])
  (fact 5))
```

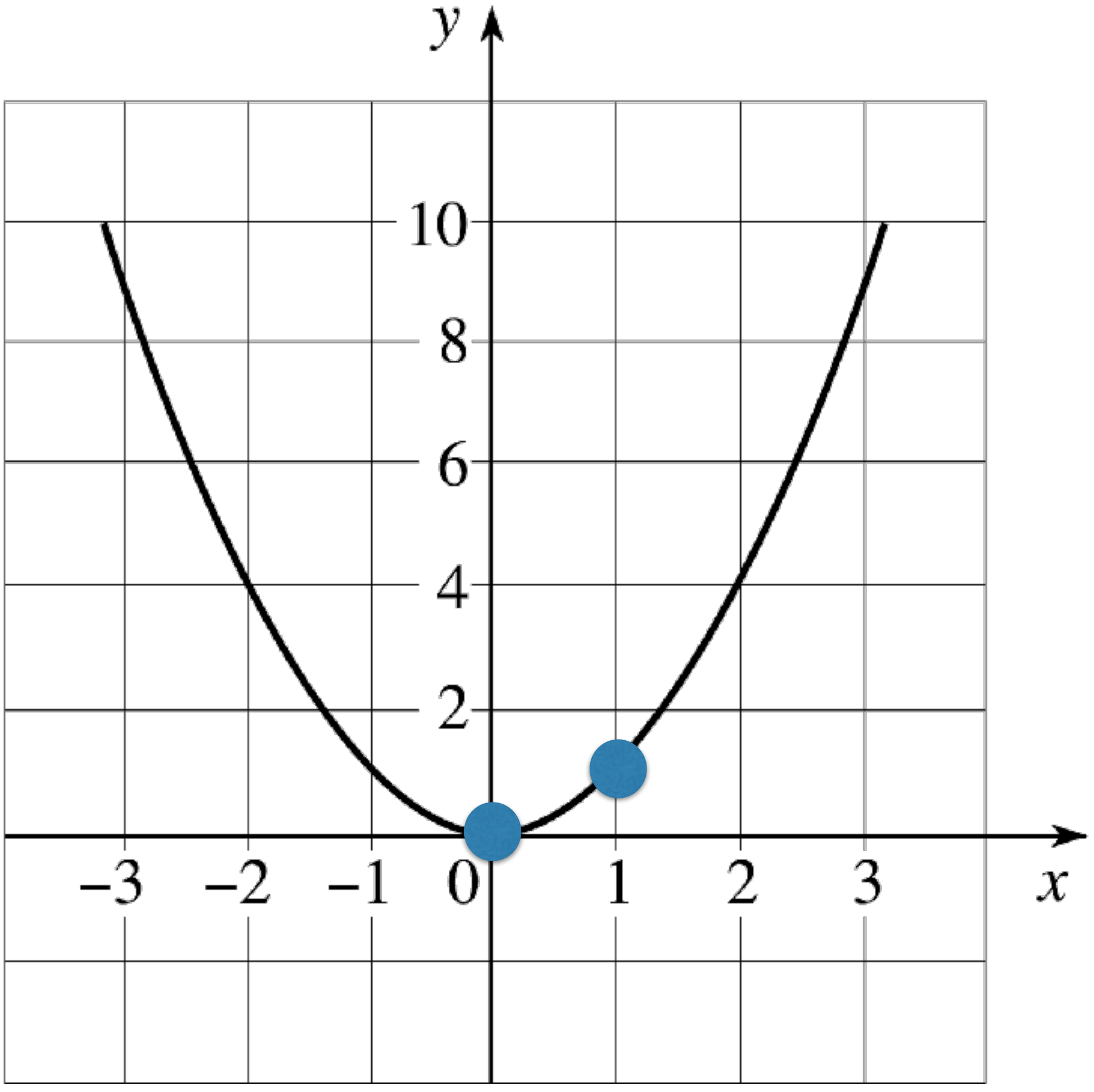
```
(letrec ([mk (λ (fact) (λ (n)
                         (if (= n 0)
                             1
                             (* n (fact (- n 1)))))))])
  ((mk mk) 5))
```

```
(letrec ([mk (λ (mk) (λ (n)
                       (if (= n 0)
                           1
                           (* n ((mk mk) (- n 1)))))))])
  ((mk mk) 5))
```

Y

$$(Y\ f) = f\ (Y\ f)$$

(It’s a fixed-point combinator!)


y
10
8
6
4
2
−3
−2
−1
0
1
2
3
x

```
Y  =  (U (λ (y) (λ (f)
        (f (λ (x) (((y y) f) x)))))
```

```
(letrec ([fact (Y (λ (fact) (λ (n)
                              (if (= n 0)
                                  1
                                  (* n (fact (- n 1))))))))])
  (fact 5))
```

```
e ::= (lambda (x) e)
    | (e e)
    | x
```

De-churching

```
(define (church->nat cv)
                      )


(define (church->list cv)

                           )

(define (church->bool cv)

                               )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
                                  )

(define (church->bool cv)
                                    )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) '())))


(define (church->bool cv)

                        )
```

De-churching

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) '())))


(define (church->bool cv)
        ((cv (λ () #t))
         (λ () #f)))
```

```
(letrec ([map (λ (f lst)
                (if (null? lst)
                    '()
                    (cons (f (car lst))
                          (map f (cdr lst))))))])
  (map (λ (x) (+ 1 x))
       '(0 5 3)))
```

```
(define lst
  (((((((((λ (Y-comb)
             (λ (church:null?)
               (λ (church:cons)
                 (λ (church:car)
                   (λ (church:cdr)
                     (λ (church:+)
                       (λ (church:*)
                         (λ (church:not)
                           ((λ (map)
                              ((map
                                (λ (x)
                                  ((church:+ (λ (f) (λ (x) (f x))))
                                   x)))
                               ((church:cons (λ (f) (λ (x) x)))
                                ((church:cons
                                  (λ (f)
                                    (λ (x) (f (f (f (f (f (f (f x))))))))
                                  ((church:cons
                                    (λ (f) (λ (x) (f (f (f (f x))))))
                                    (λ (when-cons)
                                      (λ (when-null)
                                        (when-null (λ (x) x))))))))))))
                            (Y-comb
                              (λ (map)
                                (λ (l)
                                  ((((church:null? l)
                                     (λ (x) (λ (y) y)))
                                    (λ (x)
                                      ((church:cons (church:* (church:car l) (church:car l)))
                                       (map (church:cdr l)))))
                                   0)))))))))))))
            (λ (f)
              ((λ (g) (f (λ (x) ((g g) x))))
               (λ (g) (f (λ (x) ((g g) x))))))))
   (λ (f) (λ (x) x)))
  (λ (f) (λ (x) (f x)))))

> (map church->nat (church->list lst))
'(1 6 4)
```

Now we can write a church encoder!