Church encoding,
the CC machine,
and the CK machine

CS245 — Fall 2019

Authors: Kris Micinski + Thomas Gilray

Writing a functional compiler

Functional compilers translate input expressions in one intermediate representation (intermediate language) into *equivalent* expressions in another, simpler IR.

```
translate : IR₀ -> IR₁
```

$$\begin{tikzcd}
\mathrm{IR}_0 \arrow[r, "\texttt{interp}"] \arrow[d, "\texttt{compile or desugar}"'] & 2 \arrow[d, phantom, "\rotatebox{90}{=}"] \\
\mathrm{IR}_1 \arrow[r, "\texttt{interp}"] & 2
\end{tikzcd}$$

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

"If I only let you use the lambda calculus, can you still write normal programs (e.g., ones that use recursion/+/if/etc...)?"

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

"If I only let you use the lambda calculus, can you still write normal programs (e.g., ones that use recursion/+/if/etc…)?"

YES!

Church encoding

Church encoding is the process of encoding all values as lambda abstractions. E.g., Church numerals are an encoding of numbers, 0, 1, 2, …, as first-class functions. Church booleans are an encoding of #t and #f as functions. Church lists are an encoding of lists (pairs and null) as functions.

Project 4: **Church compiler**

Your goal is to compile a significant subset of Scheme (with a few simplifications) down to the lambda calculus. This requires you to desugar (simplify) most forms, to curry all functions, and to church-encode all non-function values.

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

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

$$\begin{tikzcd}
\text{Scheme IR} \arrow[r, "\texttt{interp}"] \arrow[d, "\texttt{church-encode}"'] & 2 \\
\lambda\text{-calculus} \arrow[r, "\texttt{interp}"] & (\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ x)))) \arrow[u, dashed, "\texttt{church->nat}"']
\end{tikzcd}$$

Today, we'll start with:

```
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>
```

Desugaring Let

```
(let ([x e] …) ebody)
```

```
(let ([x e] …) ebody)
```

↓

```
((λ (x …) ebody) e …)
```

Currying

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

```
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

$$\begin{tikzcd}
((\lambda\ (t\ f)\ t)\ e_T\ e_F) \arrow[d] & ((\lambda\ (t\ f)\ f)\ e_T\ e_F) \arrow[d] \\
((\lambda\ (t\ f)\ t)\ v_T\ v_F) \arrow[d] & ((\lambda\ (t\ f)\ f)\ v_T\ v_F) \arrow[d] \\
v_T & v_F
\end{tikzcd}$$

What issues arise with
this encoding?

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

$$\downarrow$$

....

$$\begin{tikzcd}
\texttt{((}\lambda\texttt{ (t f) (t)) (}\lambda\texttt{ () }e_T\texttt{) (}\lambda\texttt{ () }\Omega\texttt{))} \arrow[d] \\
\texttt{((}\lambda\texttt{ () }e_T\texttt{))} \arrow[d] \\
e_T \arrow[d] \\
v_T
\end{tikzcd}$$

```
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

Hint: turn all nouns into verbs!

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

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

```
0:  (λ (f) (λ (x) x))

1:  (λ (f) (λ (x) (f x)))

2:  (λ (f) (λ (x) (f (f x))))

3:  (λ (f) (λ (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))))))
```

$$fN^M = fN*M$$

```
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)))
```

Try an Example. How can we define null?

Try an Example. How can we define null?

```
church:null? = (λ (p)
                 (p (λ (a b) #f)
                    (λ () #t)))
```

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

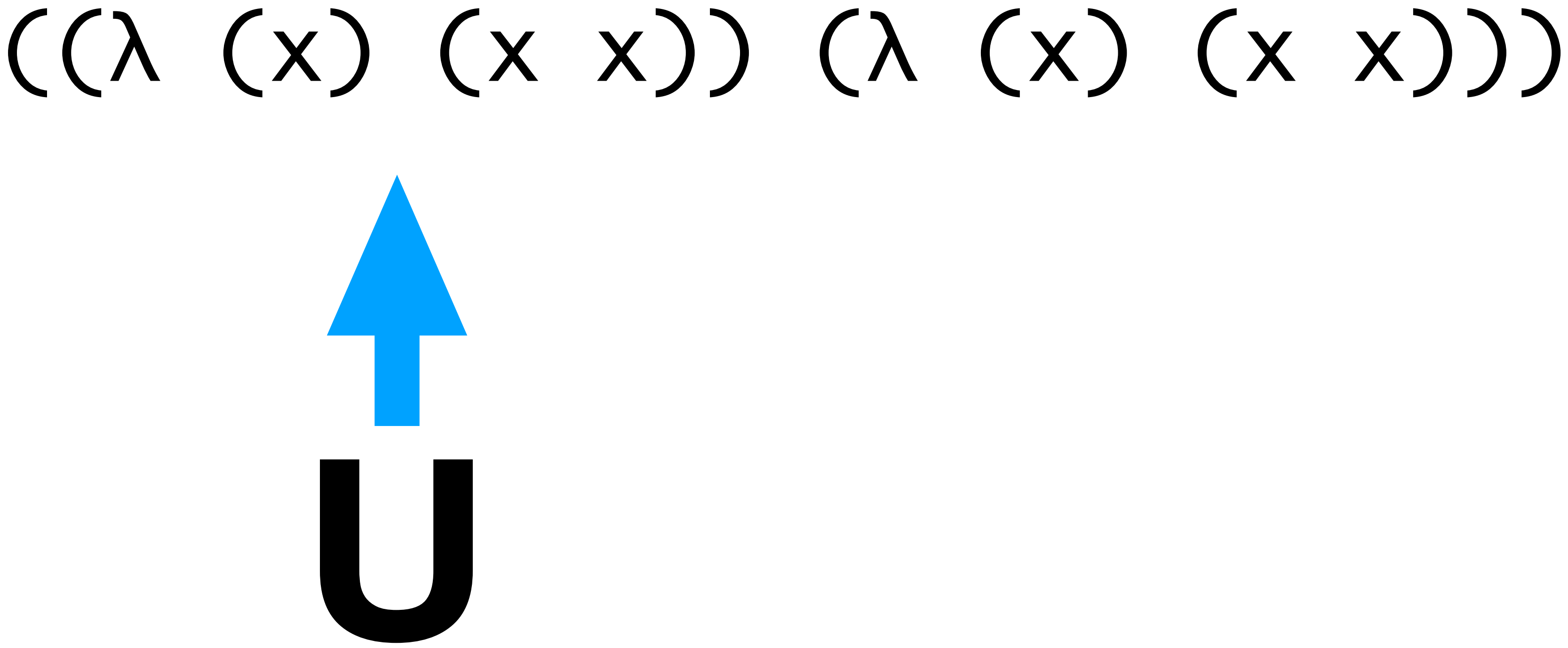
Infinity

Ω

```
((λ (x) (x x)) (λ (x) (x x)))
```

Key: U takes a function and calls it on itself

Ω



```
((λ (x) (x x)) (λ (x) (x x)))
```

U

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
                (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))])
  (fib 3))
```

What can I type right here to make fib work?

(Hint: the answer can be written in 5 characters)

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
                (lambda (x) (if (= x 0) 1 (* x (... (- x 1))))))))])
  (fib 3))
```

What can I type right here to make fib work?

(f f)

Y combinator

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

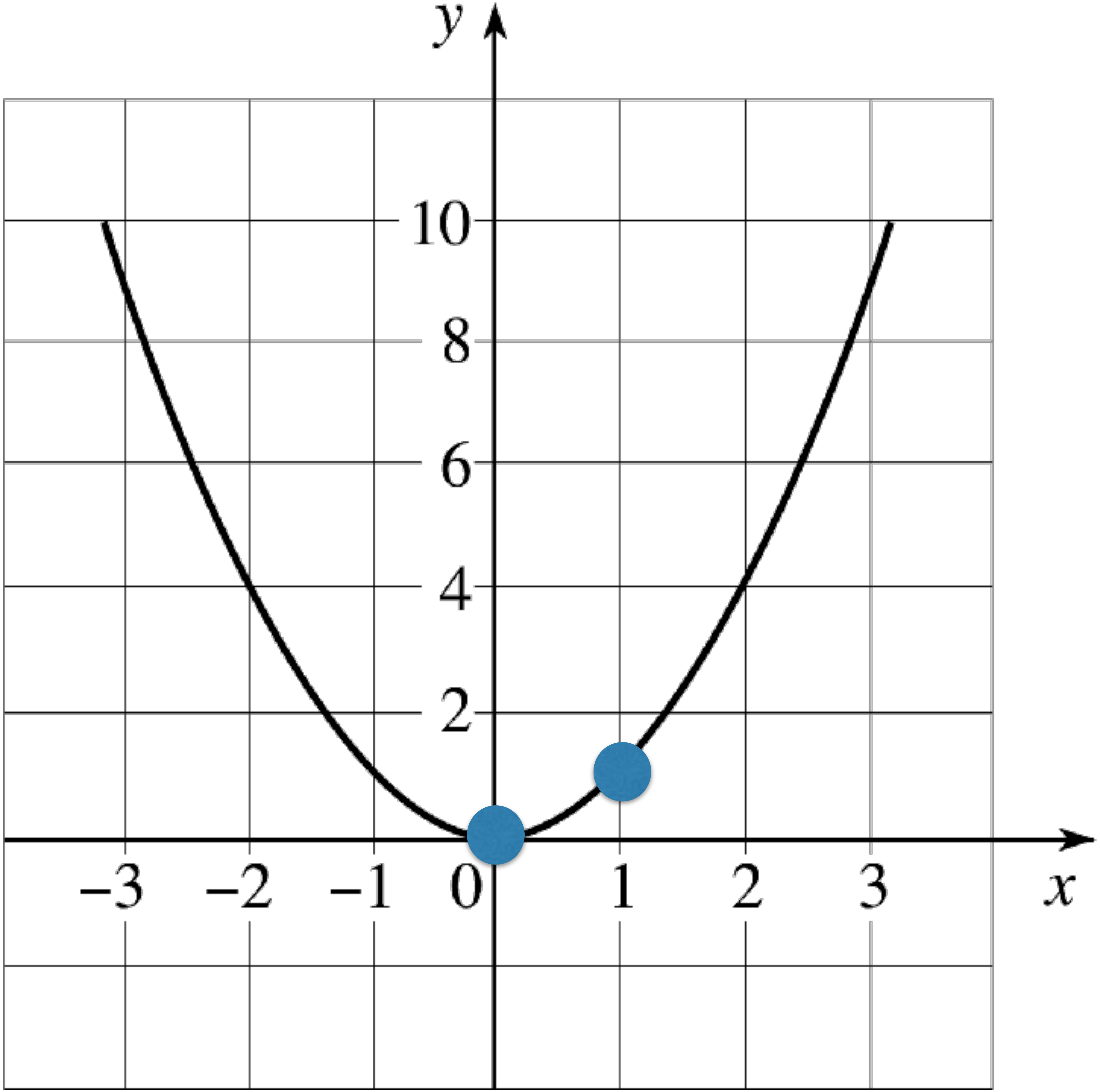
Key idea: instead of

```
(let ([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

Three step process for deriving Y

```
(Y f) = f (Y f)
```

```
Y = (λ (f) (f (Y f)))
```

1. Treat as definition

```
mY = (λ (mY)
       (λ (f)
         (f ((mY mY) f))))
```

2. Lift to mk-Y,
use self-application

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

3. Eta-expand

U-combinator: `(U U)` **is Omega**

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

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

Try an example!!!

```
(define Y ((λ (x) (x x)) (λ (y) (λ (f)
                                    (f (λ (x) (((y y) f) x)))))))
```

```
(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2)))))
```

Rewrite this to use the Y combinator instead

```
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 x))))))))
                               ((church:cons
                                 (λ (f) (λ (x) (f (f (f x))))))
                                (λ (when-cons)
                                  (λ (when-null)
                                    (when-null (λ (x) x)))))))))
                          (Y-comb
                           (λ (map)
```

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

Abstract Machine Zoo

C	Term-rewriting Machine

Evaluation contexts

Restrict the order in which we may simplify a program's redexes

```
ℰ ::= (ℰ e)
    | (v ℰ)
    | □
```

(left-to-right) CBV evaluation

```
ℰ ::= (ℰ e)
    | □
```

(left-to-right) CBN evaluation

```
v ::= (λ (x) e)
```

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

Context and redex

For CBV a redex must be `(v v)`
For CVN, a redex must be `(v e)`

$$\mathcal{E}[\overbrace{\texttt{(v v)}}^{\texttt{r}}] \;=$$

`(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))`

$\mathcal{E}$ = `(□ (λ (w) w))`

`r = ((λ (x) ((λ (y) y) x)) (λ (z) z))`

Context and redex

$\mathscr{E}$[r] =

(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))

$\mathscr{E}$ = (□ (λ (w) w))

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))

$\rightarrow_{\beta}$ ((λ (y) y) (λ (z) z))

Put the reduced redex back in its evaluation context…

$\mathscr{E}$ = `(□ (λ (w) w))`

r = `((λ (x) ((λ (y) y) x)) (λ (z) z))`

$\rightarrow_{\beta}$ `((λ (y) y) (λ (z) z))`

$\big\downarrow$ $\mathscr{E}$`[r]`

`(((λ (y) y) (λ (z) z)) (λ (w) w))`

Exercises—can you evaluate…

```
1)  (((λ (y) y) (λ (z) z)) (λ (w) w))
```

```
2)  ((λ (u) (u u)) (λ (x) (λ (x) x)))
```

```
3)  (((λ (x) x) (λ (y) y))
     ((λ (u) (u u)) (λ (z) (z z))))
```

Abstract Machine Zoo

C	Term-rewriting Machine
CC	Context and Redex Machine
CK	Control / Continuation Machine

Next time…