P4: Church Encoding

CIS352 — Fall 2022

Kris Micinski

Last lecture: Church numerals and operations over arithmetic.

After last lecture, you should be able to use Church encoding to express things like this:

$$2 + 3 * (4 + 1)$$

Example: assume the terms **plus** and **mult** are defined (i.e., you may use them as free variables). Translate the following expression:

2 + 3 * (4 + 1)

Example: assume the terms **plus** and **mult** are defined (i.e., you may use them as free variables). Translate the following expression:

2 + 3 * (4 + 1)

```
((plus (λ (f) (λ (x) (f (f x)))))
 ((mult (λ (f) (λ (x) (f (f (f x))))))
  ((plus
    (λ (f) (λ (x) (f (f (f (f x)))))))
   (λ (f) (λ (x) (f x))))))
```

Exercise (part of p4):

Write a Racket function (nat->church n), which consumes a natural number n, and produces a quoted, church-encoded number n

```
(nat->church 0) ;; (lambda (f) (lambda (x) x))
(nat->church 1) ;; (lambda (f) (lambda (x) (f x)))

;; hint
(define (nat->church n)
  (define (gen n)
    (if (= n 0)
        'x
        'todo))
  `(λ (f) (λ (x) ,(gen n))))
```

In this project, we'll translate (most) Scheme programs to the lambda calculus.

This project: how do we translate the **rest** of Scheme?

```
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 | …
```

(Language used in project p4)

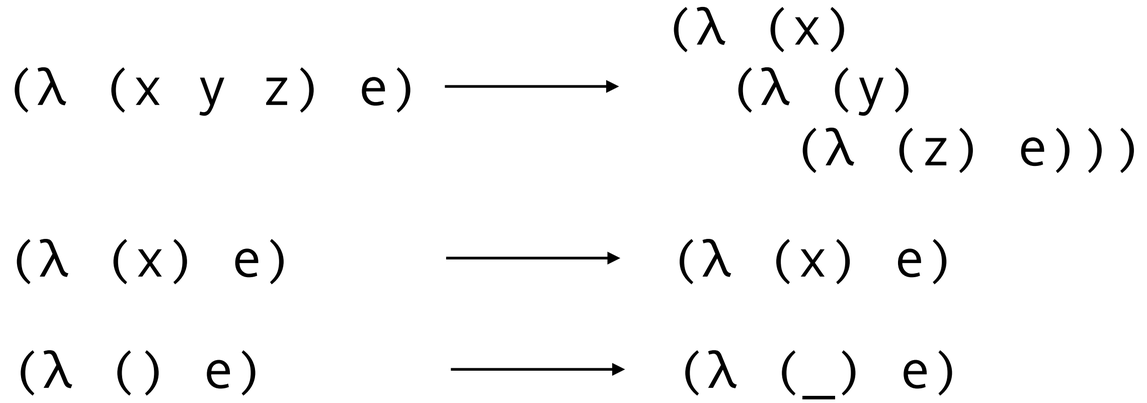
Output language

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

x ::= <vars>
```

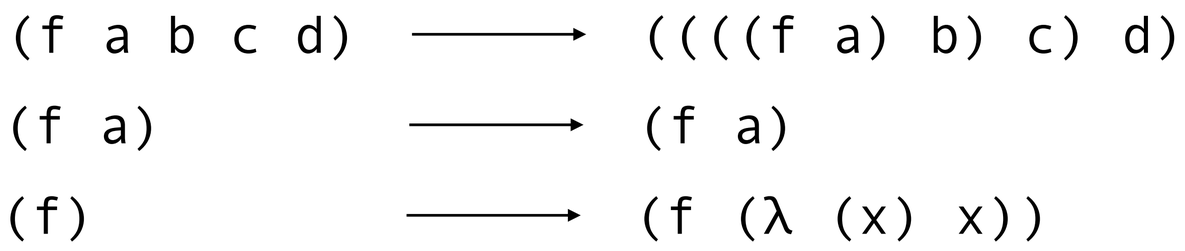
We will eliminate forms one-by-one—starting with the most complex, and translating each term down into the λ-calculus

Currying is a trick where you translate multi-arg lambdas into **sequences** of lambdas



```
;; '(λ (x y z) x) → '(λ (x) (λ (y) (λ (z) x)))
(define (curry-lambda expr)
  (match expr
    [`(lambda ,xs ,e-body) 'todo]))
```

Of course, you **also** need to fix up **callsites**



```
;; '(e0 e1 e2) -> ((e0 e1) e2)
(define (curry-callsite expr)
  'todo)
```

So far: started with **this**

```
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 | …
```

Now we have…

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

Now let's encode if

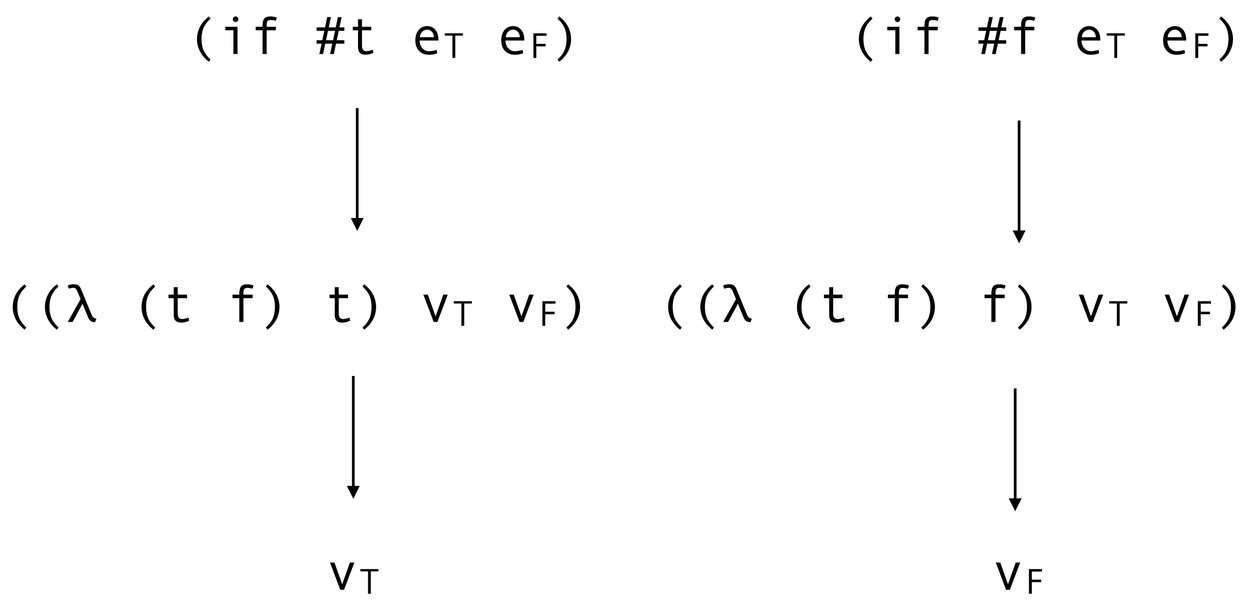
`(if #t` e_T e_F`)` `(if #f` e_T e_F`)`

e_T

e_F

We need an encoding that does this…

Let's say we encode true as `(λ (t f) t)`



This is critically broken!

Because if we did that, then the encoding of

```
(if #t 0 Ω) ;; Ω = ((lambda (x) (x x)) (lambda (x) (x x)))
```

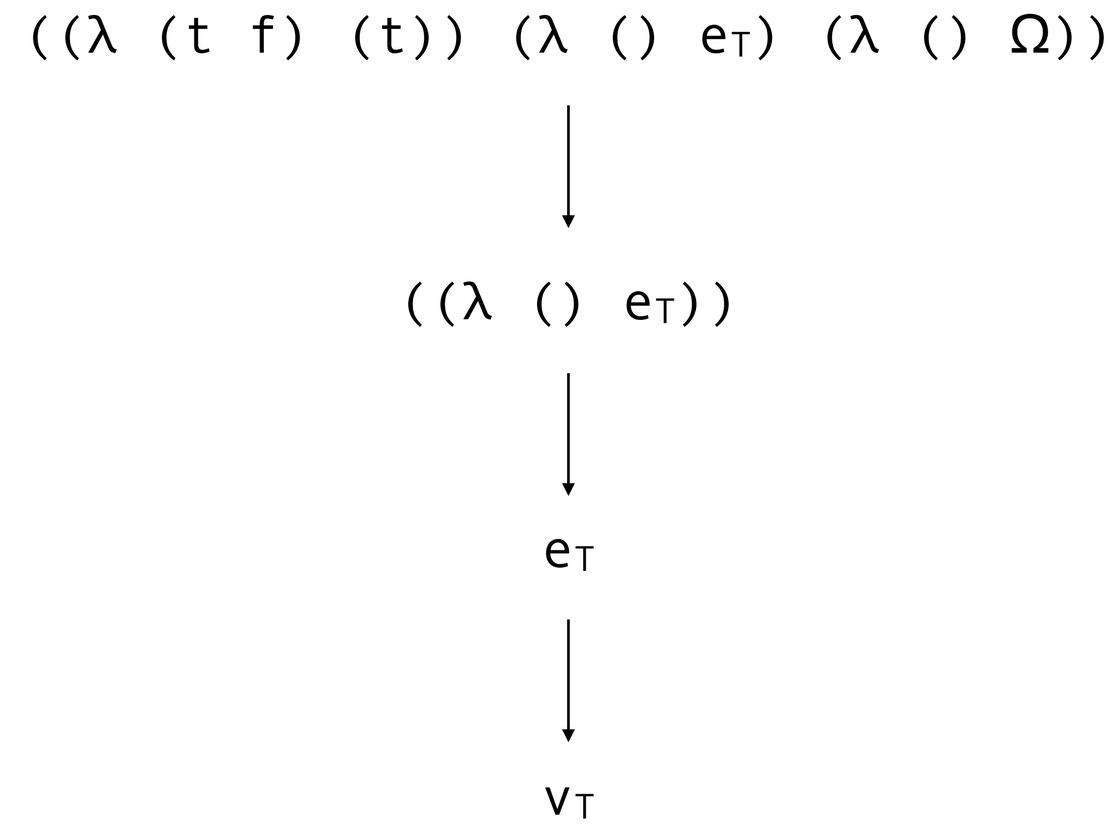
```
((λ (t f) t) 0 Ω)
```



```
.........
```

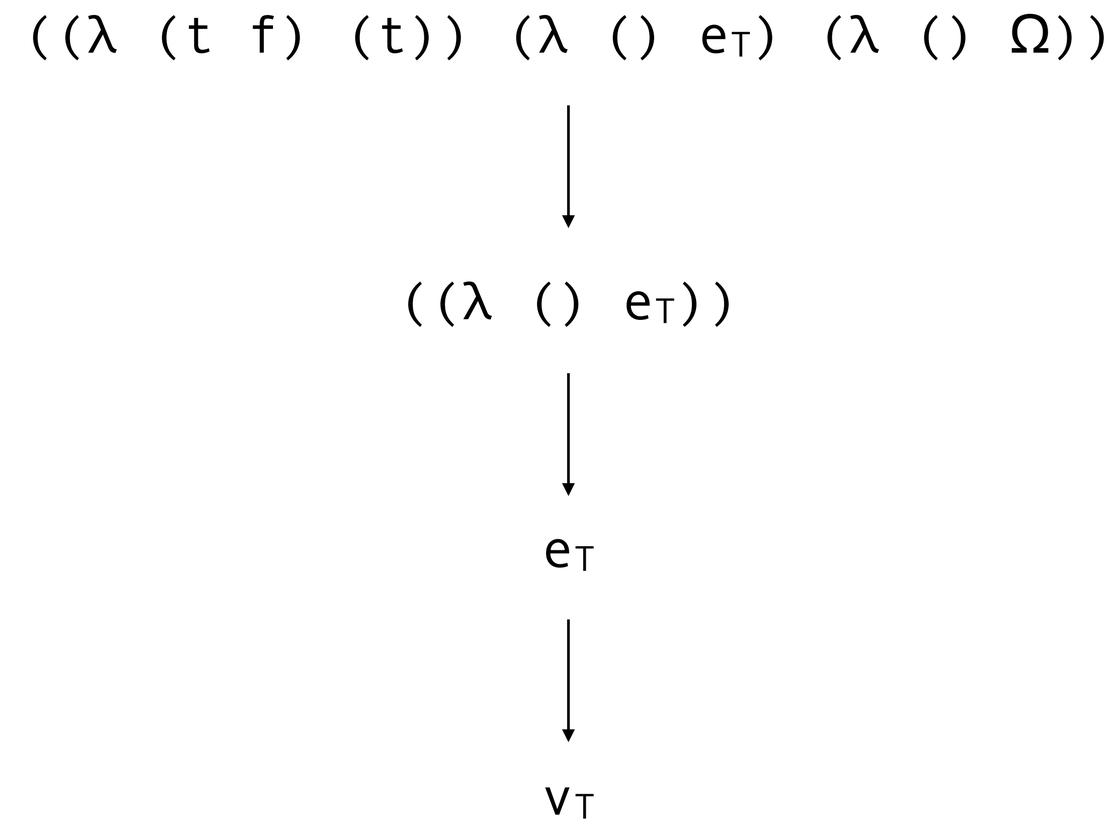
Not right! We want it to be just 0!

Note: already explained how to encode 0-arg lambda…



So our true encoding for if/true/false is…

Note: already explained how to encode 0-arg lambda…



Now we're just down to…

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

We covered **numerals** last lecture

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

So now all we need to do is this…

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

Church-Encoding for *lists*

Recall: a list can **only** be one of *two* things:

- The empty list “null”
- A cons cell (pair) of a left / right element

So we really just need an encoding of each of these

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

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

Using this definition, can you define car, cdr, and null?

```
church:null? = (λ (lst)
                 (lst (λ (a b) #f) ;; when cons
                    (λ () #t)))    ;; when null
```

Now all we have is…

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

To implement letrec, we use a **fixed-point combinator** (such as the Y combinator…). This is a bit tricky, so we'll explain it next week in class.