A-Normal Form and ANF Conversion

CIS400 (Compiler Construction)
Kris Micinski, Fall 2021

Output of assignment 2

```
; e ::= (let ([x e] ...) e)
;     | (lambda (x ...) e)
;     | (lambda x e)
;     | (apply e e)
;     | (e e ...)
;     | (prim op e ...)
;     | (apply-prim op e)
;     | (if e e e)
;     | (set! x e)
;     | (call/cc e)
;     | x
;     | (quote dat)
```

Implicit nesting structure, lots of necessary continuations…

```
; e ::= (let ([x e] ...) e)
;     | (lambda (x ...) e)
;     | (lambda x e)
;     | (apply e e)
;     | (e e ...)
;     | (prim op e ...)
;     | (apply-prim op e)
;     | (if e e e)
;     | (set! x e)
;     | (call/cc e)
;     | x
;     | (quote dat)
```

If we build an interpreter for this *direct-style* language, we end up having to push stack frames at many different forms…

```
(if e e e) ;; to evaluate first e
(e e …)
  ;; step through each e until we have
  ;; (v0 v1 …) where v0 is a closure
(call/cc e) ;; to eval e to a closure
```

Why ANF convert?

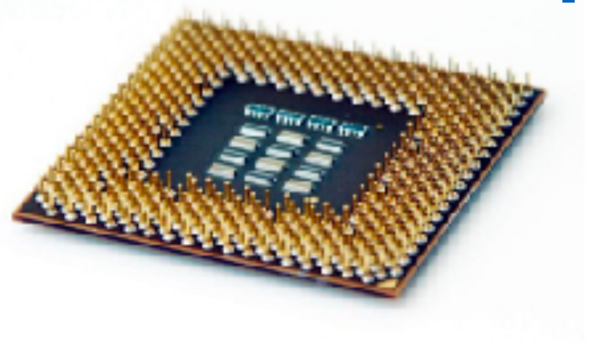
ANF conversion can be thought of as *explicating subcomputations*
If you've ever "single stepped" in a debugger, executing each subcomputation one at a time…

```
x = 2 + 3 * (4 + 5);
```

True assembly languages **require** every operation be atomic

```
movq $r0, 4
movq $r1, 5
addq $r0, $r1
movq $r1, 3
mulq $r0, $r1
movq $r1, 2
addq $r0, $r1
```

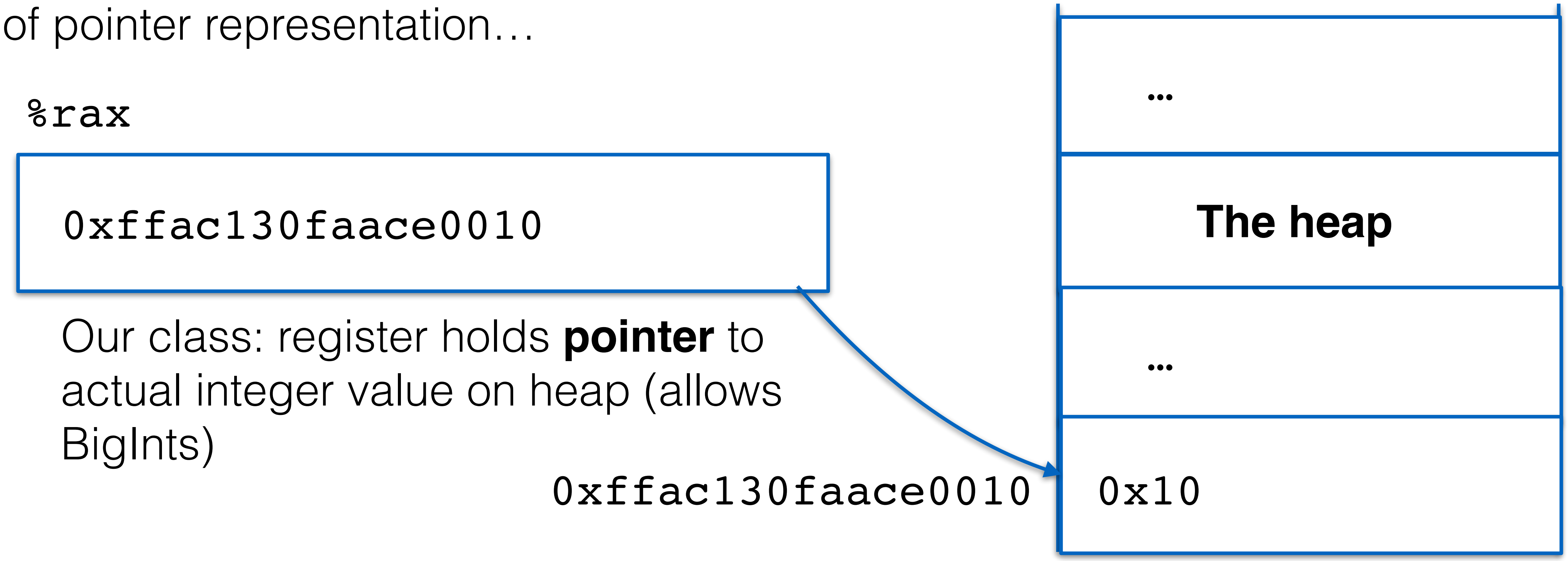
Because atomic values **can fit into registers**



Data Layout and Primitive Operations…

```
(define (celsius F)          ; celsius: mov F, a0
 (let ((t1 (/ 5 9)))         ;          div t1, 5, 9
   (let ((t2 (- F 32)))      ;          sub t2, F, 32
    (* t1 t2))))             ;          mul rv, t1, t2
                             ;          ret rv
;; above code fine if using unboxed values, but
;; needs to use explicit calls if using boxed rep
```

Note: primitive calls may (as above) be compiled to instructions (`div/sub/…`) rather than using an explicit call instruction (e.g., to a function implementing subtraction). However, this depends on data representation: if language is using "boxed" objects, direct uses of, e.g., `sub` may be impossible because of pointer representation…



Our class: register holds **pointer** to actual integer value on heap (allows BigInts)

In this example, we end up pushing lots of stack frames to compute the argument to the `call/cc`!

```
(define (return-in x e)
  (if (= x 0) e (return-in (- x 1) e)))

(call/cc (return-in 1000 (lambda (x) 3)))
```

For lots of reasons, it will be simpler if there is **as little ambiguity as possible** in control structure for our language

ANF makes evaluation order explicit in that all forms evaluate either
(a) atomically or…
(b) by calling (and getting a return from) a function

A-Normal Form (ANF) — Definition

In ANF (Administrative Normal Form), all subexpressions (of every expression) are "administratively" bound

ANF allows only **one** form to extend the stack: `let`
All values are implicitly return points

```
(+ 2 (+ 3 (+ 4 5)))
```

```
(let ([x0 (+ 4 5)]
 (let ([x1 (+ 3 x0)])
  (let ([x2 (+ 2 x1)])
    x2)
```

ANF—why do *we* care (as compiler authors)?

ANF is essentially straight-line code interspersed with function calls, but there is always a definite next step that is either a “call and continue” (let) or “return”

For example, because `if`’s guard may be evaluated immediately, the evaluation of if will be atomic

This is useful because it mirrors **machine instructions**, *assuming* the machine has a `call` instruction

```
(let ([x0 (+ 4 5)]
 (let ([x1 (+ 3 x0)])
  (let ([x2 (+ 2 x1)])
    x2)
```

```
;; imaginary assembly-ish code
r0 = call(+,4,5)
r1 = call(+,3,r0)
r2 = call(+,2,r1)
ret r2
```

```
)]
x0)])
 x1)])
```

```
;; imaginary assembly-ish code
r0 = call(+,4,5)
r1 = call(+,3,r0)
r2 = call(+,2,r1)
ret r2
```

This intermediate representation is useful to think about as humans, because we often like to think about ***blocks of variables*** and we are used to think about direct-style call/return

ANF forces us to linearize subcomputations into straight-line sequences, mirroring the block-based nature of assembly

```
(+ 1 (if (f 2) 3 4))
```

```
(let ((g391612 (f 2)))
   (let ((g391613 (if g391612 3 4)))
  (+ 1 g391613)))
```

We let-bind `(f 2)` (so `if`'s guard is atomic), 3 and 4 are return points, and the top-level if is `let`-bound to `r0`

In essence, ANF corresponds to an assembly-like language where we can perform function calls and bind names, but control structure (“where to go next”) is always very simple and explicit

Also… Converting our code to ANF will make it significantly easier to perform CPS conversion (next), as continuation structure becomes much simpler in ANF!

Values

- Recall: values are **atomic objects** of the language.
- Syntactically, values are forms which need no evaluation:
    - Variables, constants, quoted datums, etc…
    - ***Not*** applications (e.g., `(list 1 2 3)`), if, etc…

ANF Conversion—Examples

```
(+ 1 2)
```

$$\Big\downarrow$$

ANF Conversion—Examples

$$\begin{tikzcd}
\texttt{(+ 1 2)} \arrow[d] \arrow[dr] & \\
\texttt{(+ 1 2)} & \cancel{\begin{array}{l}\texttt{(let ([r0 1])} \\ \quad\texttt{(let ([r1 2])} \\ \qquad\texttt{(+ r0 r1)))}\end{array}}
\end{tikzcd}$$

Both 1 and 2 are *already* values,

WRONG

(Don’t need to let-bind 1, it’s already a value)

ANF Conversion—Examples

```
((f g) (h x) 3)
```

$\downarrow$

Note: 3 already atomic, no binder necessary.

ANF Conversion—Examples

```
((f g) (h x) 3)
```

$\downarrow$

```
(let ([t0 (f g)])
 (let ([t1 (h x)])
  (t0 t1 3)))
```

Note: 3 already atomic, no binder necessary.

```
(define (f n)
     (if (= n 0)
         1
         (* n (f (- n 1)))))

(f 20)
```

```
(define (f n)
     (if (= n 0)
         1
         (* n (f (- n 1)))))

(f 20)
```

```
(define f
  (λ (n)
    (let ((g1478 (= n 0)))
      (if g1478
        1
        (let ((g1479 (- n 1)))
          (let ((g1480 (f g1479)))
            (* n g1480)))))))
(f 20)
```

Notice how guard is let-bound

ANF Conversion—Examples

```
(+ (begin (displayln “hello”) 1)
   (begin (displayln “world”) 2))
```

Note that we needed to be careful to ensure ordering

ANF Conversion—Examples

```
(+ (begin (displayln “hello”) 1)
   (begin (displayln “world”) 2))
```

↓

```
(let ([r0 (begin (let ([r1 (displayln “hello”)]) 1))])
  (let ([r2 (begin (let ([r3 (displayln "world")]) 2))])))
```

Note that we needed to be careful to ensure ordering

ANF Conversion—Examples

```
(+ (begin (displayln “hello”) (displayln “there”) 1)
   (begin (displayln “world”) 2))
```

$$\Big\downarrow$$

```
;; Exercise…
```

Note that we needed to be careful to ensure ordering

ANF Conversion—Examples

```
(if (z (+ 2 (call/cc (f y)))) 3 (g z))
```

↓

```
(let ([r0
       (let ([r1
              (let ([r2 (let ([r3 (f y)]) (call/cc r3))])
                (+ 2 r2))])
         (z r1))])
  (if r0
      3
      (let ([r4 (g z)])
        r4)))
```

ANF conversion algorithm — Intuition

- ANF algorithm uses a few key insights:
    - Converts direct-style terms to ANF terms
- Key idea: **walk** over the term, whenever you encounter a complex expression, if its arguments are not atoms, generate an administrative let-binding for them and then use them as if they were atoms
- Remember, we **don’t** create let bindings for values—those are implicitly return points

Notice how in this term, we get to an **if** term, and the atom isn’t atomic, so we *make* it atomic by assigning it an intermediate value…

```
                  (if (z                  ) 3 (g z))



 (let ([r0



                  ])
   (if r0
       3

              ))
```

We need to do the same thing with the call to g in the else branch of the let…

```
                (if (z                              ) 3 (g z))



  (let ([r0




                ])
    (if r0
        3
        (let ([r4 (g z)])
          r4)))
```

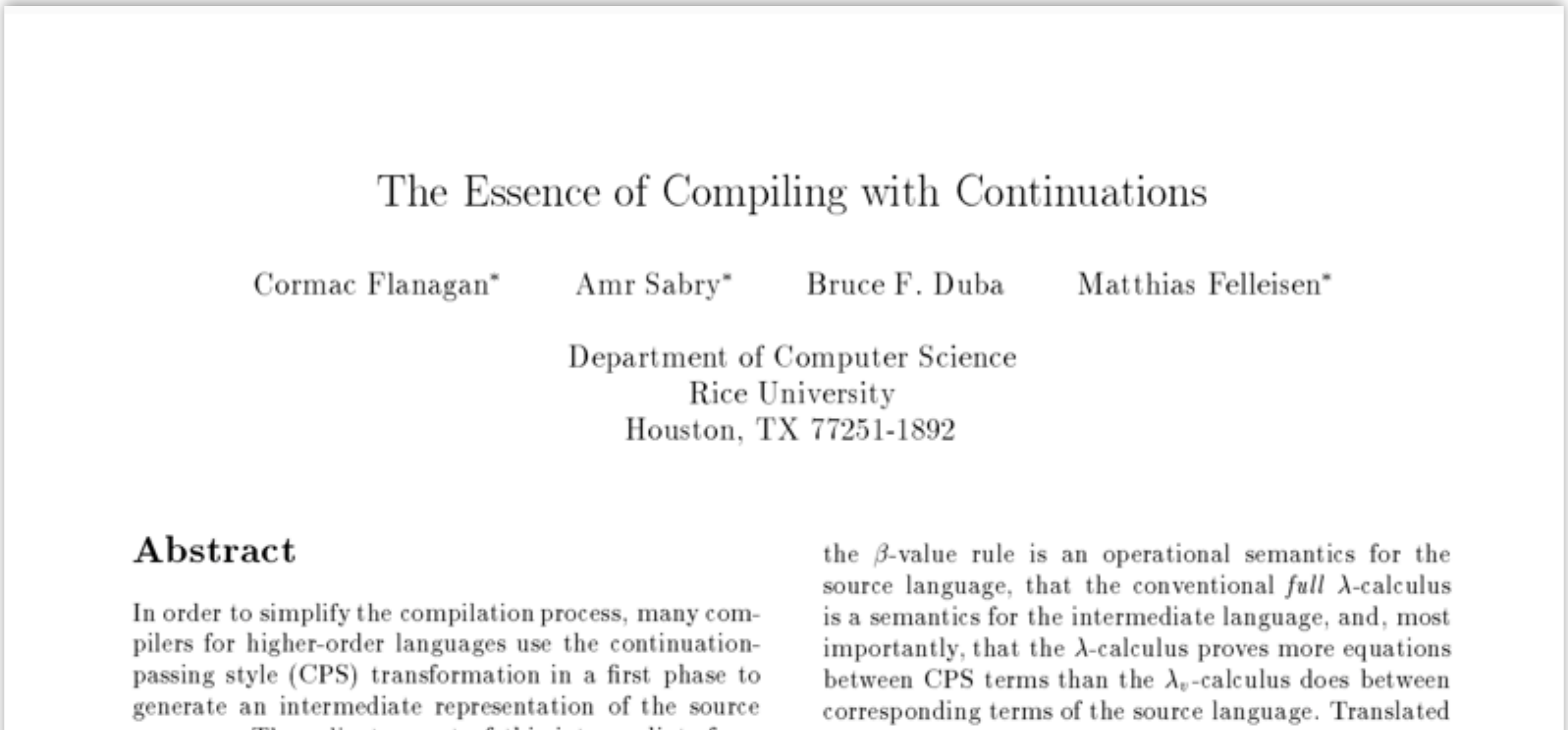
Now, we can look at z, we see it’s a call to + on 2 and call/cc…

```
(if (z (+ 2 (call/cc (f y)))) 3 (g z))
```

```
(let ([r0
       (let ([r1
              (let ([r2 (let ([r3 (f y)]) (call/cc r3))])
                (+ 2 r2))])
         (z r1))])
  (if r0
      3
      (let ([r4 (g z)])
        r4)))
```

ANF Conversion Algorithm

From “The Essence of Compiling with Continuations”



The Essence of Compiling with Continuations

Cormac Flanagan* Amr Sabry* Bruce F. Duba Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

In order to simplify the compilation process, many compilers for higher-order languages use the continuation-passing style (CPS) transformation in a first phase to generate an intermediate representation of the source

the β-value rule is an operational semantics for the source language, that the conventional *full* λ-calculus is a semantics for the intermediate language, and, most importantly, that the λ-calculus proves more equations between CPS terms than the λ_v-calculus does between corresponding terms of the source language. Translated

- Four functions:
    - (normalize-term e) — normalizes a term e to ANF
    - (normalize e k) — normalizes e to ANF and applies **`k`**
    - (normalize-name e k) — converts e to ANF and—if the result is complex (i.e., not atomic) calls k on a temporary variable which is `let`-bound to the result (making it atomic rather than complex)
    - (normalize-name* es k) — ^^ same with multiple es

*Sample code from Flanagan et. al and Matthew Might's "A Normalization: Why and How"

`normalize-term` will ANF-normalize the term and return its result…

```
(define (normalize-term e)
  (normalize e (λ (e+) e+)))
```

Note how **`normalize-term`** passes a continuation callback to **`normalize`** to continue the work…

`normalize-name` will ANF-normalize e to a value, take its result e+, and then check to see if e+ is a value:

- If it **is**, then directly **apply k**
    - This allows us to *avoid unnecessary let-binding*
- If `e+` is **not** a value, it is a complex expression that must be `let`-bound because k is assuming it is an atomic expression, so let-bind it as a new var `x` and pass that var to `k`

Note how `normalize-name` passes a continuation callback to `normalize` to continue the work…

```
(define (normalize-name e k)
  (normalize e (λ (e+)
    (if (value? e+) (k e+)
        (let ([x (gensym)])
         `(let ([,x ,N]) ,(k x)))))))
```

*Sample code from Flanagan et. al and Matthew Might's "A Normalization: Why and How"

For example…

```
(normalize-name (+ 1 2) (lambda (x) x))
;; (let ([x (+ 1 2)]) x)
```

In sum, **normalize-name** will take an expression e, call normalize (which we will see soon) to convert it to ANF, and —if it's not a value—let-bind it to make it a value and then return that value to the callback `k`.

```
(define (normalize-name e k)
  (normalize e (λ (e+)
    (if (Value? e+) (k e+)
        (let ([x (gensym)])
         `(let ([,x ,N]) ,(k x)))))))
```

```
;; convert e to ANF and apply k to result
(define (normalize e k)
  (match e
    ;; lambdas just return themselves to k
    [`(λ ,params ,body)
      (k `(λ ,params ,(normalize-term body)))]
    ;; return any other value to k
    [(? value?)               (k k)]))
    ;; to normalize let, first normalize its binder, then
    ;; (by passing a callback) let-bind its result to be used
    ;; in the normalized body
    [`(let ([,x ,eb]) ,e-body)
      (normalize eb (λ (e-r)
       `(let ([,x ,e-r])
         ,(normalize e-body k))))]
    ;; to normalize an if, normalize e0 to an atomic value t
    ;; and branch on its result
    [`(if ,e0 ,e1 ,e2)
      (normalize-name e0 (λ (e-r)
       (k `(if ,e-r ,(normalize-term e1)
                    ,(normalize-term e2)))))]
    ;; to normalize a call…
    [`(,Fn . ,M*)
      (normalize-name Fn (λ (t)
       (normalize-name* M* (λ (t*)
        (k `(,t . ,t*))))))]
```

```
(define (normalize-name* M* k)
  (if (null? M*)
      (k '())
      (normalize-name (car M*) (λ (t)
       (normalize-name* (cdr M*) (λ (t*)
        (k `(,t . ,t*)))))))))
```

```
#lang racket

;; anormalize: A simple A-Normalizer for a subset of Scheme

;; Original Author Matt Might (rewritten by Kris Micinski)
;; http://matt.might.net/


(define (atomic? exp)
  (match exp
    [`(quote ,_)         #t]
    [(? number?)         #t]
    [(? boolean?)        #t]
    [(? string?)         #t]
    [(? char?)           #t]
    [(? symbol?)         #t]
    [(or '+ '- '* '/ '=) #t]
    [else                #f]))


(define (normalize-name exp k)
  (normalize exp (λ (aexp)
    (if (atomic? aexp) (k aexp)
        (let ([t (gensym)])
         `(let ([,t ,aexp]) ,(k t)))))))

(define (normalize-name* exp* k)
  (if (null? exp*)
      (k '())
      (normalize-name (car exp*) (λ (t)
       (normalize-name* (cdr exp*) (λ (t*)
        (k `(,t . ,t*))))))))

;; Tests:
(pretty-print
 (normalize-term
  '((f g) (h 3) #f)))
```

```
;; Expression normalization:
(define (normalize-term exp) (normalize exp (λ (x) x)))

;; normalize
(define (normalize exp k)
  (match exp
    [`(λ ,params ,body)
      (k `(λ ,params ,(normalize-term body)))]

    [`(let () ,exp)
      (normalize exp k)]

    [`(let ([,x ,exp1] . ,clause) ,exp2)
      (normalize exp1 (λ (aexp1)
       `(let ([,x ,aexp1])
         ,(normalize `(let (,@clause) ,exp2) k))))]

    [`(if ,exp1 ,exp2 ,exp3)
      (normalize-name exp1 (λ (t)
       (k `(if ,t ,(normalize-term exp2)
                  ,(normalize-term exp3)))))]

    [`(set! ,v ,exp)
      (normalize-name exp (λ (t)
       `(let ([,(gensym 'x) (set! ,v ,t)])
          ,(k '(void)))))]

    [`(,f . ,e*)
      (normalize-name f (λ (t)
       (normalize-name* e* (λ (t*)
        (k `(,t . ,t*))))))]

    [(? atomic?)
     (k exp)]))
```

Examples…

```
(pretty-print
 (normalize-term
  '(λ (z)
     (λ (x)
       (set! z (f x))))))

(pretty-print
 (normalize-term
  '(+ (if (f x) 0 1) 2)))

(pretty-print
 (normalize-term
  '(let ((id (λ (x) x)))
     (let ((apply (λ (f x) (f x))))
       ((id apply) (id 3))))))

(pretty-print
 (normalize-term
  '(let ([id (λ (x) x)]
         [apply (λ (f x) (f x))])
     ((id apply) (id 3)))))
```