CMSC 430 - 20 Oct 2026

Compiling function definitions and calls

- Iniquity - Function definitions and calls
  - designing a calling convention for Iniquity
  - arity checking at run-time
- Jig - tail calls

CMSC 430 - 20 Oct 2026

Announcements

- Assignment 5 due tonight
- Assignment 6 due next Thursday
- Survey released after class, due by start of Thursday

Iniquity

(cont.)

Designing our own calling convention

Function calls are like “`let` at a distance”

```
(f 3 4)
```

```
(define (f x y)
  (+ x y))
```

is like

```
(let ((x 3) (y 4))
  (+ x y))
```

Except the code for f is not part of the application expression

Designing our own calling convention

A first attempt (doesn't work)

```
(f 3 4)
```

```
(define (f x y)
  (+ x y))
```

Idea: arguments passed on the stack,

return point after arguments,

caller pushes and pops

(Push 3)

(Push 4)

(Call ‘f)

(Pop)

(Pop)

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))

(Ret)

Designing our own calling convention

Same thing without Call (still doesn't work)

```
(f 3 4)
```

(Push 3)

(Push 4)

(Lea ‘rax ‘r)

(Push ‘rax)

(Jmp ‘f)

(Label ‘r)

(Pop)

(Pop)

```
(define (f x y)
  (+ x y))
```

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))
(Ret)

Idea: arguments passed on the stack,

return point after arguments,

caller pushes and pops

Designing our own calling convention

Return point before arguments (still doesn't work)

```
(f 3 4)
```

(Lea ‘rax ‘r)

(Push ‘rax)

(Push 3)

(Push 4)

(Jmp ‘f)

(Label ‘r)

(Pop)

(Pop)

```
(define (f x y)
  (+ x y))
```

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))
(Ret)

Idea: arguments passed on the stack,

return point *before* arguments,

caller pushes and pops

Designing our own calling convention

Return point before arguments (works!)

```
(f 3 4)
```

(Lea ‘rax ‘r)

(Push ‘rax)

(Push 3)

(Push 4)

(Jmp ‘f)

(Label ‘r)

```
(define (f x y)
  (+ x y))
```

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))

(Pop)

(Pop)

(Ret)

Idea: arguments passed on the stack,

return point *before* arguments,

caller pushes, *callee pops*

Compiling function definitions

```
;; Defn -> Asm
(define (compile-define d)
  (match d
    [(Defn f xs e)
     (seq (Label (symbol->label f))
          (Cmp r8 (length xs)) ; arity check
          (Jne 'err)
          (compile-e e (reverse xs))
          (Add rsp (* 8 (length xs))) ; pop args
          (Ret))]))
```

Compiling function applications

```
;; Id [Listof Expr] CEnv -> Asm
;; The return address is placed above the arguments, so callee pops
;; arguments and return address is next frame
(define (compile-app f es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es es (cons #f c))
         (Mov r8 (length es)) ; pass arity info
         (Jmp (symbol->label f))
         (Label r))))
```

Compiling function arguments

```
;; [Listof Expr] CEnv -> Asm
(define (compile-es es c)
  (match es
    ['() '()]
    [(cons e es)
     (seq (compile-e e c)
          (Push rax)
          (compile-es es (cons #f c)))]))
```

Jig

Consider this program

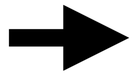
```
(define (f x)
  (if (zero? x)
      0
      (f (sub1 x))))

(f 100)
```

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

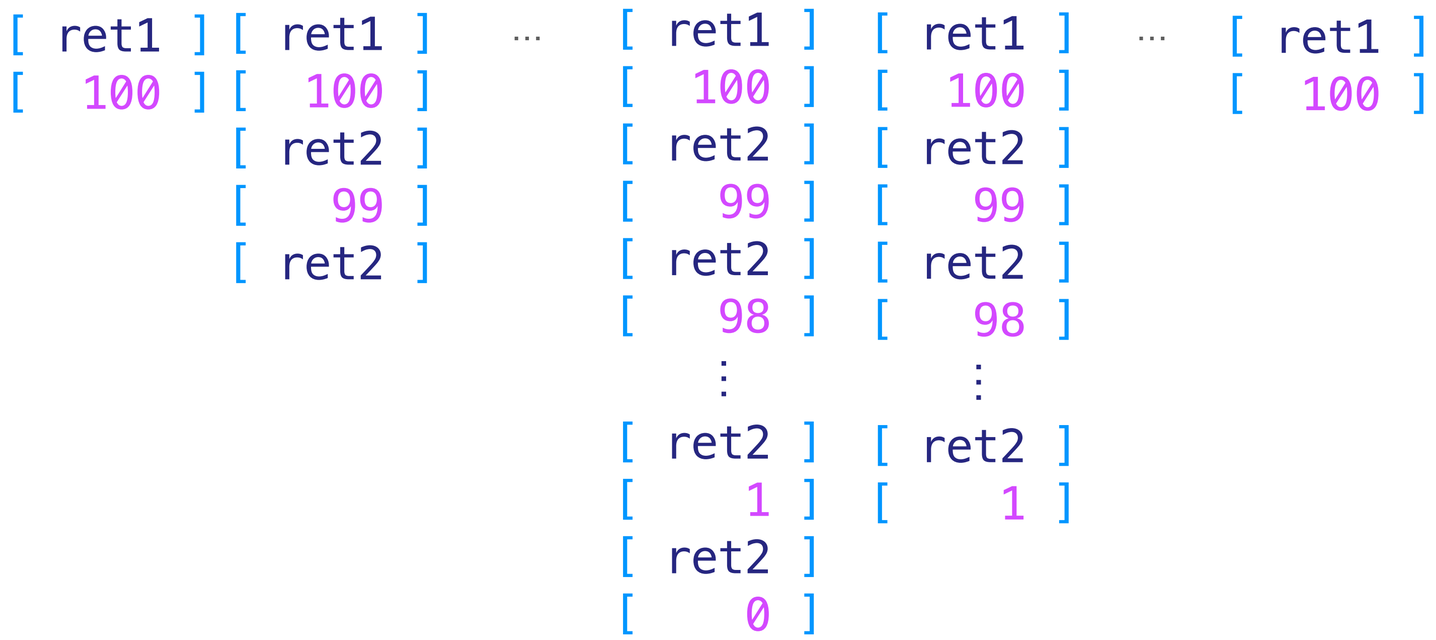
A view of the stack



```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

A view of the stack



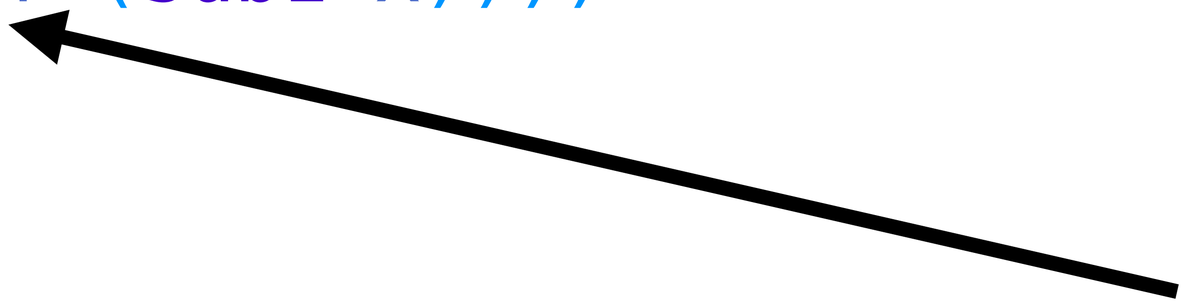

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

A view of the stack

```
(define (f x)
  (if (zero? x)
      0
      (f (sub1 x))))

(f 100)
```



Call with a return label so that we can pop off x and return….

But we're done with x. What if we popped first?

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

Pop before call

```
(Lea 'r8 'ret2)
(Push 'r8)   ; push return
(Push 'rax)  ; push argument
(Add 'rsp 8) ; pop x
(Jmp 'f)
(Label 'ret2)
(Label 'done)
(Ret)
```



Call with a return label so that we can pop off x and return….

But we're done with x. What if we popped first?

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

Pop before call

```
(Lea 'r8 'ret2)
(Push 'r8)   ; push return
(Add 'rsp 8) ; pop x
(Push 'rax)  ; push argument
(Jmp 'f)
(Label 'ret2)
(Label 'done)
(Ret)
```



Call with a return label so that we can pop off x and return….

But we're done with x. What if we popped first?

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

Pop before call

```
(Label 'f)
(Mov 'rax (Offset 'rsp 0))
(Add 'rsp 8) ; pop x
(Cmp 'rax 0)
(Je 'done)
(Sub 'rax (value->bits 1))
(Lea 'r8 'ret2)
(Push 'r8)   ; push return
(Push 'rax)  ; push argument
(Jmp 'f)
(Label 'ret2)
(Label 'done)
(Ret)
```



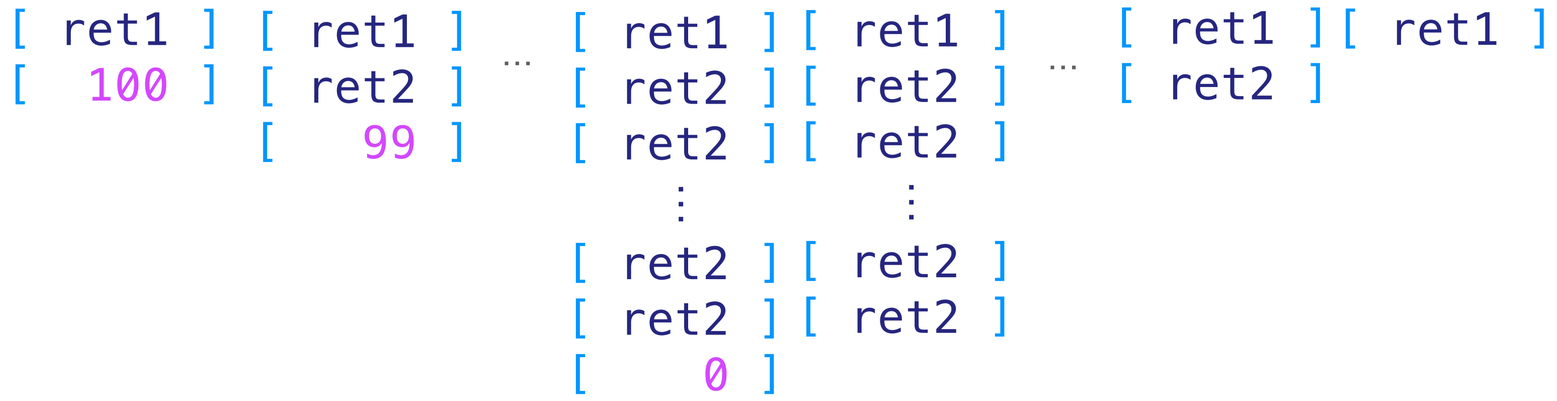
Call with a return label so that we can pop off x and return….

But we're done with x. What if we popped first?

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Add 'rsp 8) ; pop x
     (Ret)
```

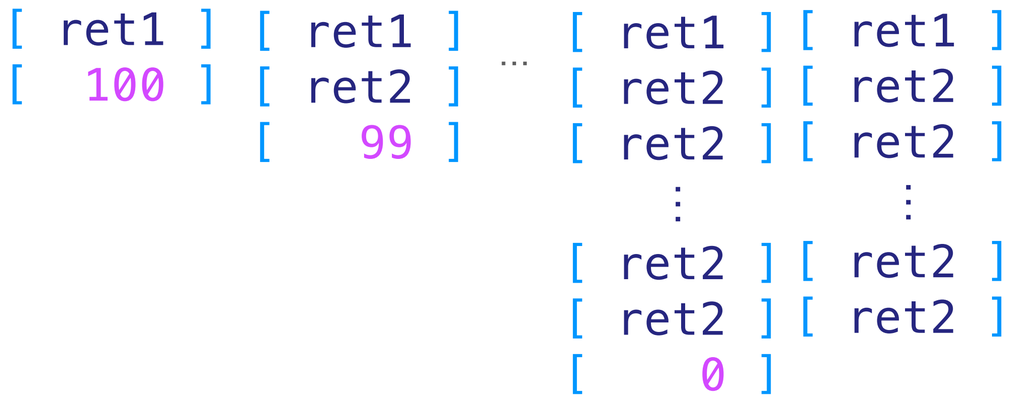
Revised view of the stack




```
;; Revised:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Add 'rsp 8) ; pop x
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Ret)
```

Returning just to return





Pushes a return point so that when control returns, we can return to the caller.

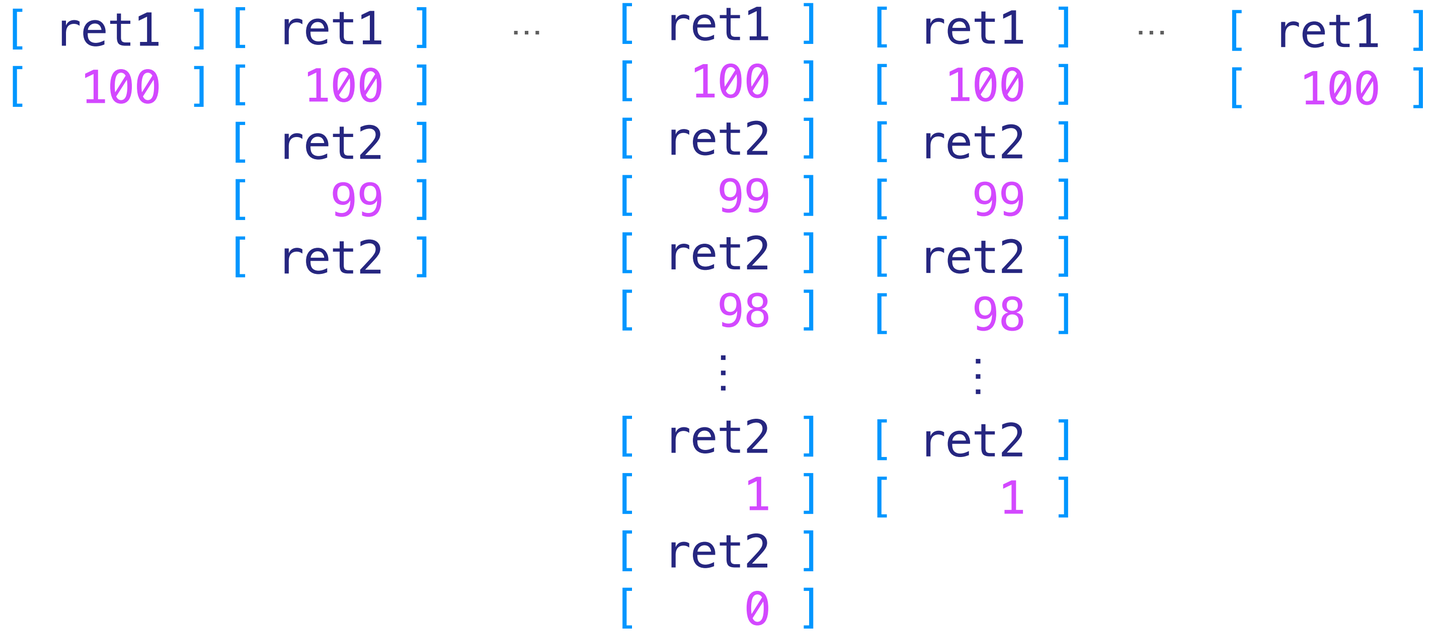
What if we just didn’t?

```
;; Revised:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (Label 'f)
     (Mov 'rax (Offset 'rsp 0))
     (Add 'rsp 8) ; pop x
     (Cmp 'rax 0)
     (Je 'done)
     (Sub 'rax (value->bits 1))
     (Lea 'r8 'ret2)
     (Push 'r8)   ; push return
     (Push 'rax)  ; push argument
     (Jmp 'f)
     (Label 'ret2)
     (Label 'done)
     (Ret)
```

Tail calls (asymptotically) save *SPACE*

Non-tail call:



Tail call:



(saving space may save time too)

Key idea

Some calls don’t need to be returned to

When a function call doesn’t need to be returned to:

- Pop local environment
- Push arguments
- Jump

When can’t you do this?

When there’s work to do after the call

```
(define (f x)
  (if (zero? x)
      0
      (f (sub1 x))))

(f 100)
```



```
(define (g x)
  (if (zero? x)
      0
      (add1 (g (sub1 x)))))

(g 100)
```



Sometimes you can rewrite to enable tail calls

So there's no work to do after the call

```
(define (g x)
  (if (zero? x)
      0
      (add1 (g (sub1 x)))))

(g 100)
```



```
(define (g x)
  (g/a x 0))

(define (g/a x a)
  (if (zero? x)
      a
      (g/a (sub1 x) (add1 a))))

(g 100)
```



Sometimes you can rewrite to enable tail calls

So there's no work to do after the call

```
(define (reverse xs)
  (match xs
    ['() '()]
    [(cons x xs)
     (append (reverse xs)
             (list x))]))
```



```
(define (reverse xs)
  (append-reverse xs '()))
```



```
;; (append-reverse xs ys)
;;    ≡ (append (reverse xs) ys)
(define (append-reverse xs ys)
  (match xs
    ['() ys]
    [(cons x xs)
     (append-reverse xs (cons x ys))]))
```



DrRacket will show you tail positions

Hover over a function definition

```
#lang racket
(define (reverse xs)
  (append-reverse xs '()))

(define (append-reverse xs ys)
  (match xs
    ['() ys]
    [(cons x xs)
     (append-reverse xs (cons x ys))]))
```

Any function call in these positions is a tail call.

When does a call not need to be returned to?

When it occurs in tail position

```
(define (f x ...) <tail>)

<tail> ::=
  (if e1 <tail> e3)
  (if e1 e2 <tail>)
  (let ((x e)) <tail>)
  (begin e <tail>)
  (f e1 ...)
```

Compile calls in these positions to pop environment, push arguments, and jump.

Compile calls in other positions to push return label, push arguments, jump.

When does a call not need to be returned to?

When it occurs in tail position

```
;; Expr CEnv Boolean -> Asm
(define (compile-e e c t?) ...)
```

t? - is this expression in tail position

```
;; Defn -> Asm
(define (compile-define d)
  (match d
    [(Defn f xs e)
     (seq (Label (symbol->label f))
          (compile-e e (reverse xs) #t)
          (Add rsp (* 8 (length xs))) ; pop args
          (Ret))]))
```

When does a call not need to be returned to?

When it occurs in tail position

```
;; Expr CEnv Boolean -> Asm
(define (compile-e e c t?) ...)
```

t? - is this expression in tail position

```
;; Expr Expr Expr CEnv Boolean -> Asm
(define (compile-if e1 e2 e3 c t?)
  (let ((l1 (gensym 'if))
        (l2 (gensym 'if)))
    (seq (compile-e e1 c #f)
         (Cmp rax (value->bits #f))
         (Je l1)
         (compile-e e2 c t?)
         (Jmp l2)
         (Label l1)
         (compile-e e3 c t?)
         (Label l2))))
```

When does a call not need to be returned to?

When it occurs in tail position

```
;; Expr CEnv Boolean -> Asm
(define (compile-e e c t?) ...)
```

t? - is this expression in tail position

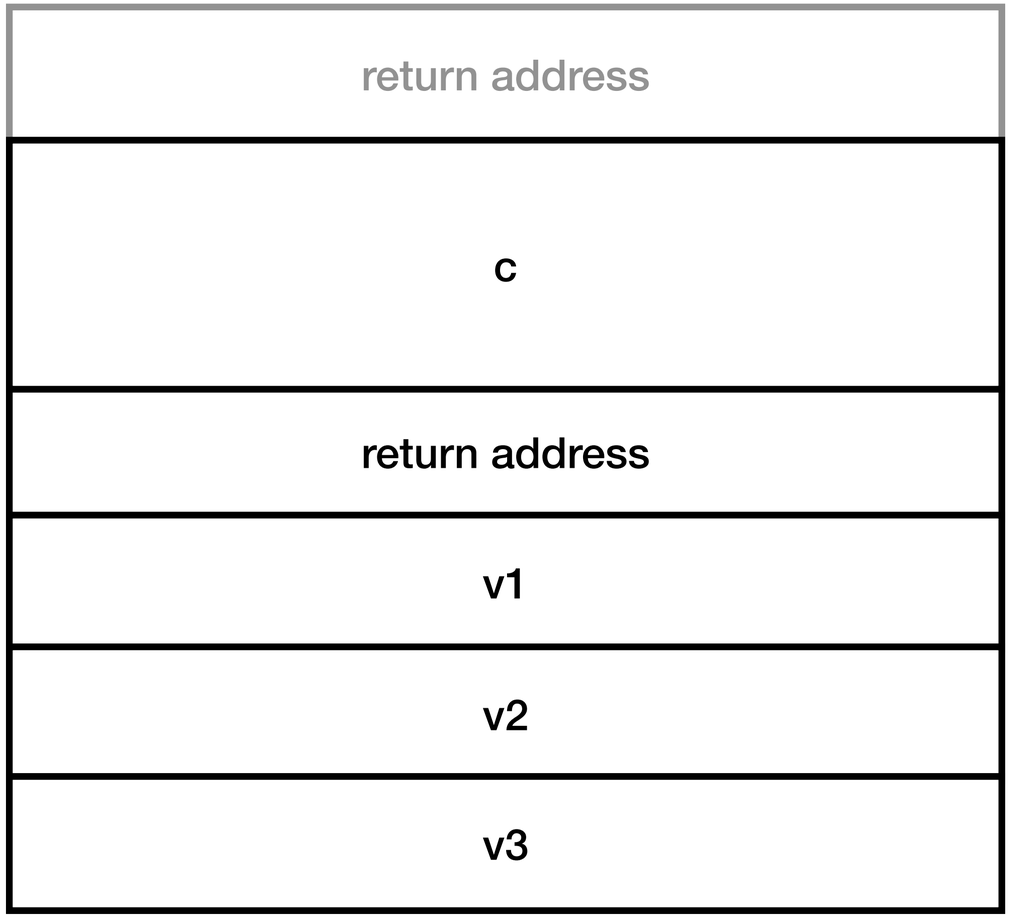
```
;; Id [Listof Expr] CEnv Boolean -> Asm
(define (compile-app f es c t?)
  (if t?
      (compile-app-tail f es c)
      (compile-app-nontail f es c)))
```

Non-tail calls

More work to do after call, so need to return

```
(compile-app-nontail 'f (list e1 e2 e3)) c)


(seq
  (Lea rax (Mem 'return))
  (Push rax)
  (compile-e e1 (cons #f c))
  (Push rax)
  (compile-e e2 (cons #f (cons #f c)))
  (Push rax)
  (compile-e e3 (cons #f (cons #f (cons #f c))))
  (Push rax)
  (Jmp 'f)
  (Label 'return))
```



Non-tail calls

More work to do after call, so need to return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```

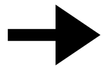


Non-tail calls

More work to do after call, so need to return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```



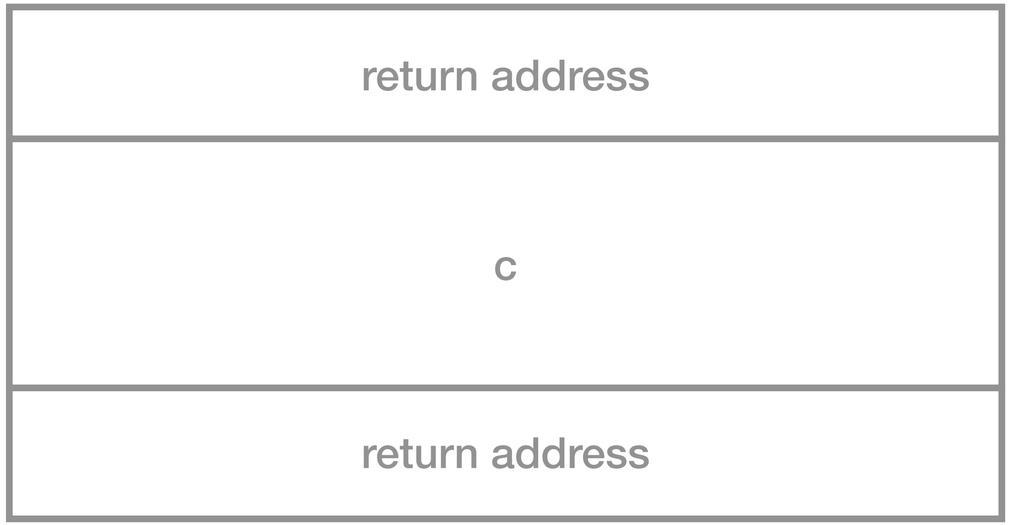


Non-tail calls

More work to do after call, so need to return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))


(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```

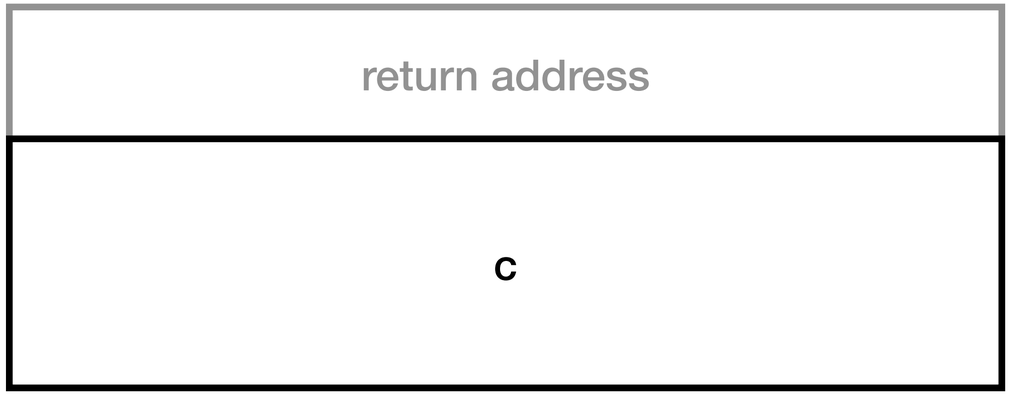


Non-tail calls

More work to do after call, so need to return

```
(compile-app-nontail ‘f (list e1 e2 e3)) c)


(seq
  (Lea rax 'return)
  (Push rax)
  (compile-e e1 (cons #f c))
  (Push rax)
  (compile-e e2 (cons #f (cons #f c)))
  (Push rax)
  (compile-e e3 (cons #f (cons #f (cons #f c))))
  (Push rax)
  (Jmp 'f)
  (Label 'return))
```

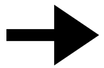


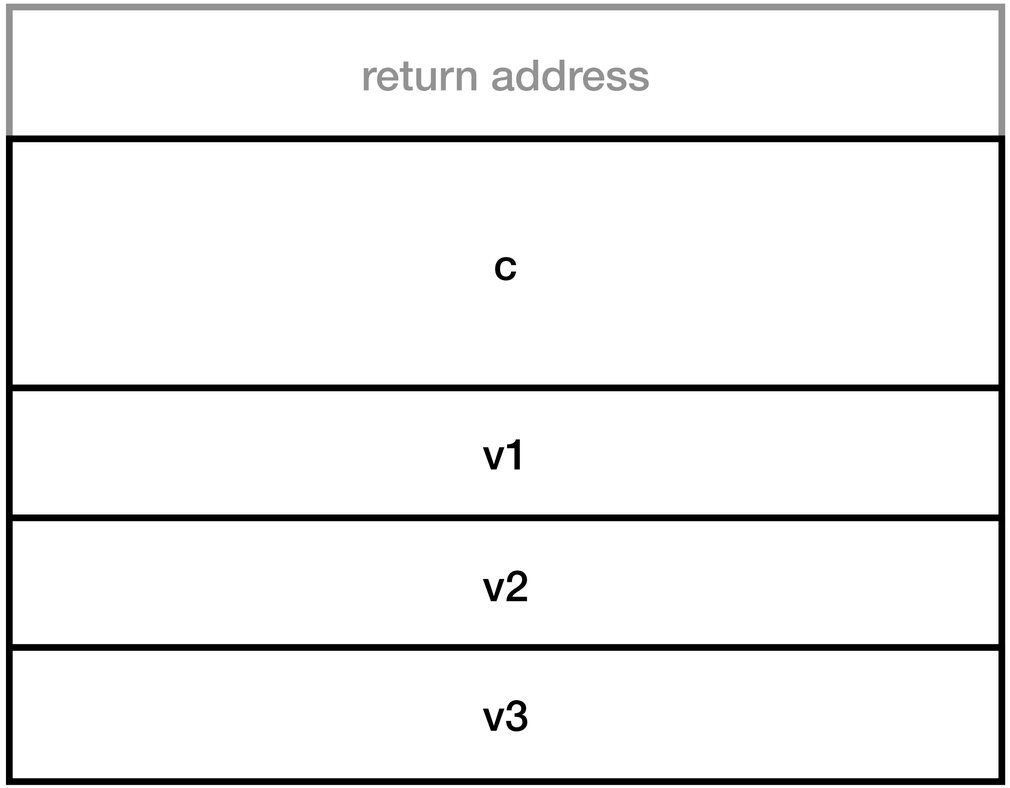
Tail calls (first attempt)

No more work to do after call, so don’t return

```
(compile-app-tail 'f (list e1 e2 e3)) c)

(seq
  (compile-e e1 c)
  (Push 'rax)
  (compile-e e2 (cons #f c))
  (Push 'rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push 'rax)
  (Jmp 'f))
```



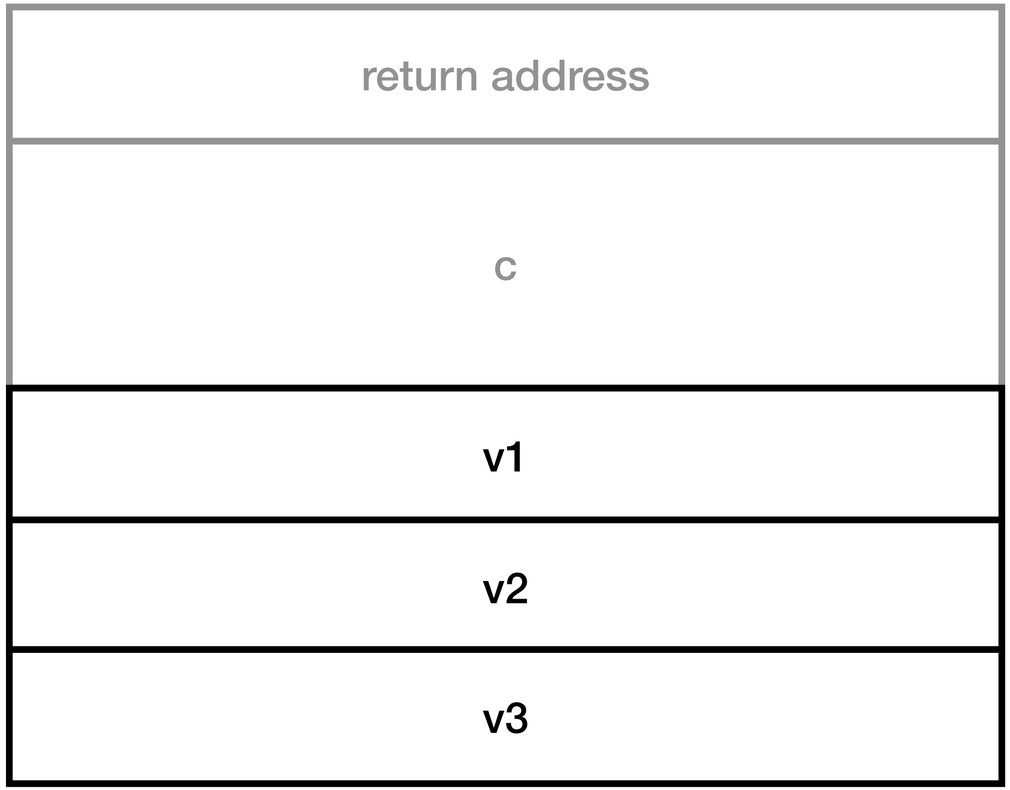


Tail calls (first attempt)

No more work to do after call, so don’t return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```

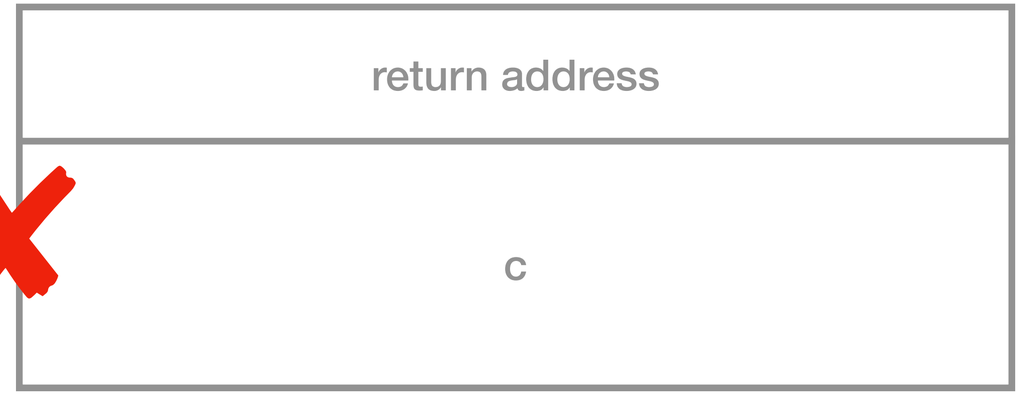


Tail calls (first attempt)

No more work to do after call, so don’t return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))


(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```

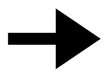


Tail calls (second attempt)

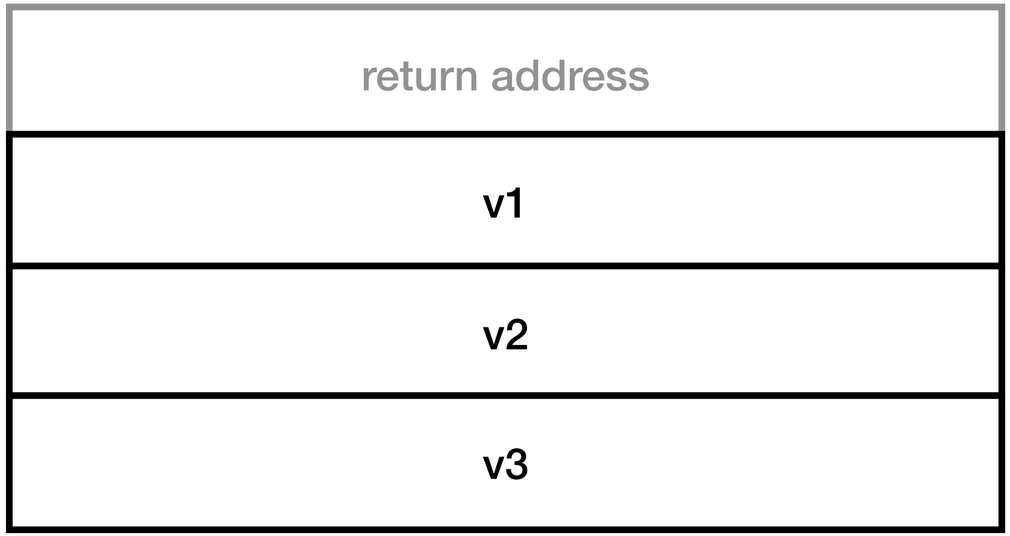
No more work to do after call, so don't return

```
(compile-app-tail 'f (list e1 e2 e3)) c)

(seq
  (Add rsp (* 8 (length c)))
  (compile-e e1 c)
  (Push rax)
  (compile-e e2 (cons #f c))
  (Push rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push rax)
  (Jmp 'f))
```





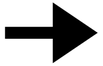


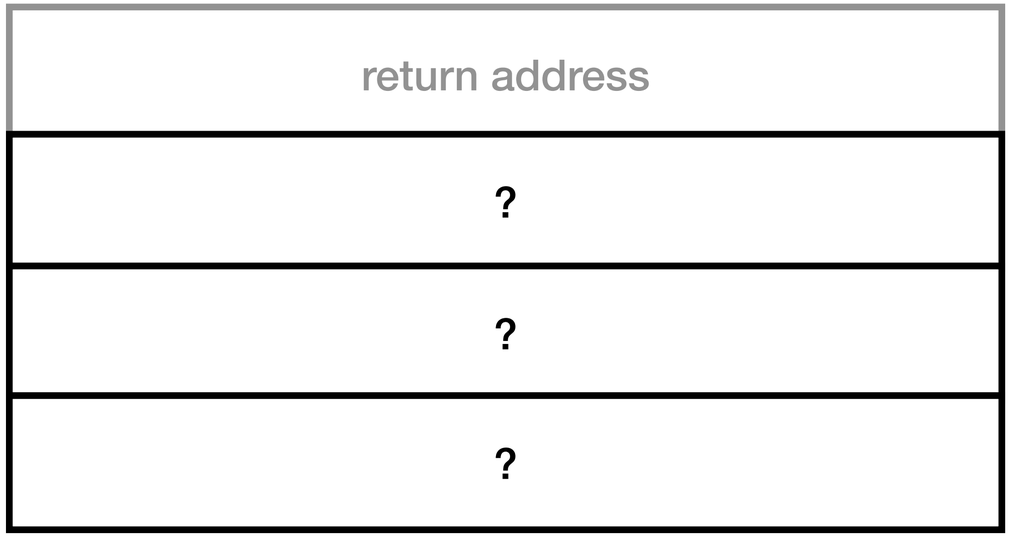
Tail calls (third attempt)

No more work to do after call, so don't return

```
(compile-app-tail 'f (list e1 e2 e3)) c)

(seq
  (compile-e e1 c)
  (Push rax)
  (compile-e e2 (cons #f c))
  (Push rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push rax)
  (Add rsp (* 8 (length c)))
  (Jmp 'f))
```



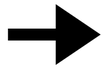


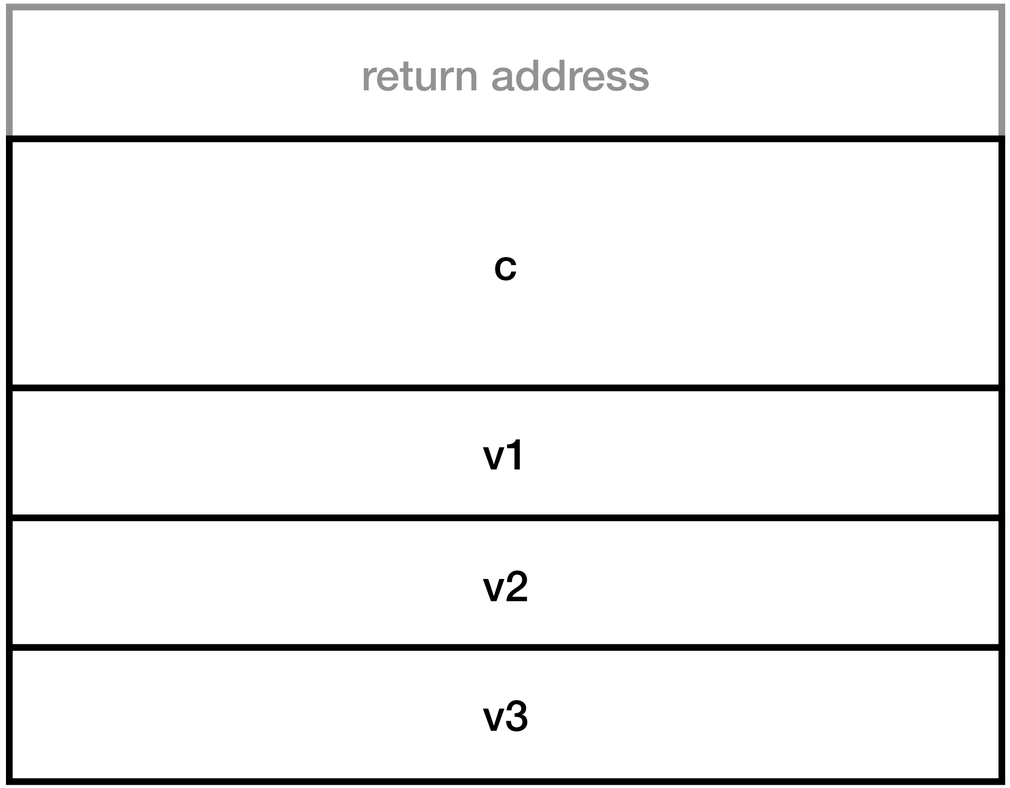
Tail calls (final attempt)

No more work to do after call, so don't return

```
(compile-app-tail 'f (list e1 e2 e3)) c)

(seq
  (compile-e e1 c)
  (Push rax)
  (compile-e e2 (cons #f c))
  (Push rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push rax)
  (move-args (length es) (length c))
  (Add rsp (* 8 (length c)))
  (Jmp 'f))
```



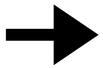


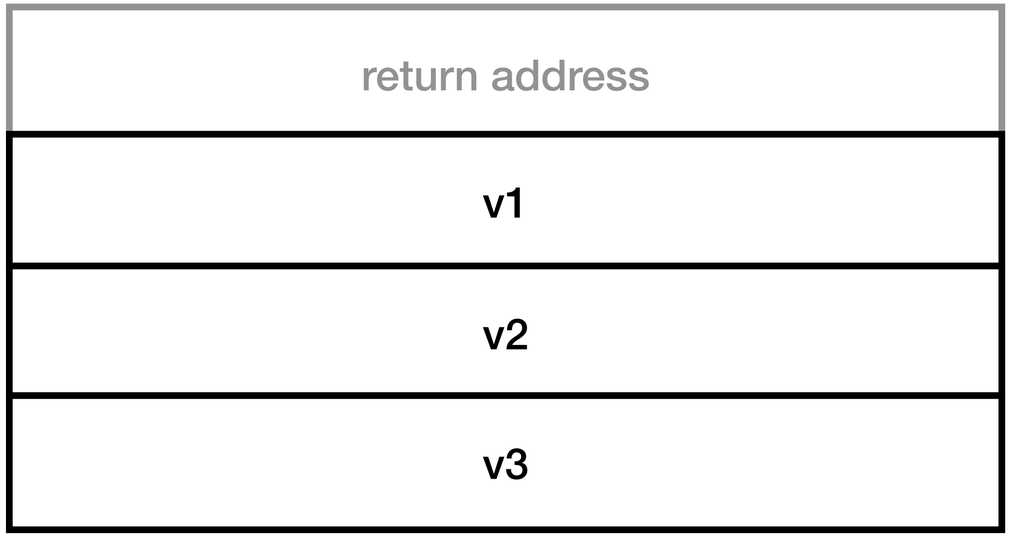
Tail calls (final attempt)

No more work to do after call, so don't return

```
(compile-app-tail 'f (list e1 e2 e3)) c)

(seq
  (compile-e e1 c)
  (Push rax)
  (compile-e e2 (cons #f c))
  (Push rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push rax)
  (move-args 3 (length c))
  (Add rsp (* 8 (length c)))
  (Jmp 'f))
```



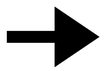


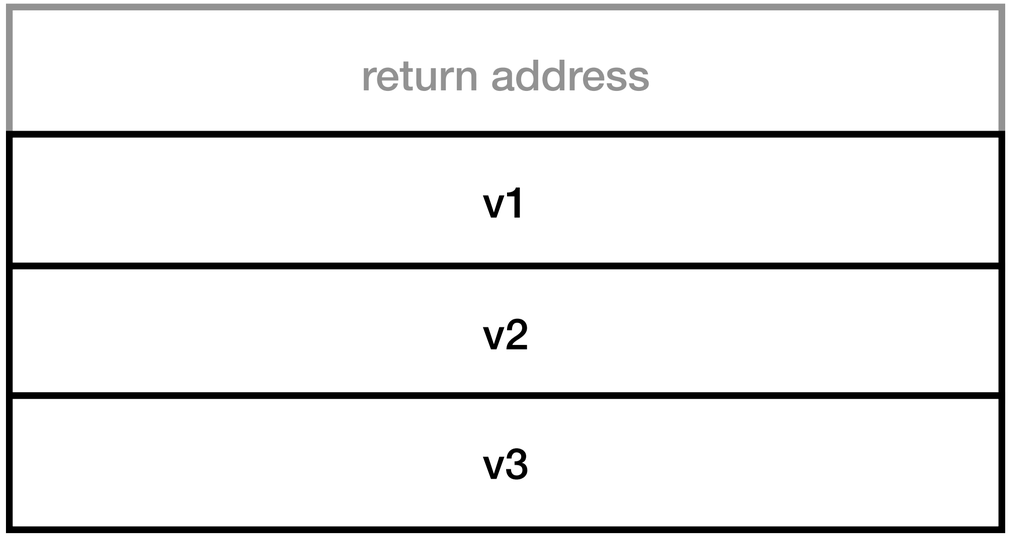
Tail calls (final attempt)

No more work to do after call, so don’t return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```





Tail calls (final attempt)

No more work to do after call, so don’t return

```
(compile-define (Defn 'f (list 'x 'y 'z)) e))
```

```
(seq
  (Label 'f)
  (compile-e e (list 'z 'y 'x) #t)
  (Add rsp 24)
  (Ret))
```

