Announcements 04/21

- Assignment 8: Due 04/28.
- Finale project: Will be released after a8. Due:05/18

- Today

  Jig: Tail calls

CMSC 430: Introduction to Compilers

Jig: tail calls

Jig

- Jig generates space-efficient code for function calls when those functions are in tail position
- Tail position:
  - A function call is in tail position when the function that makes the call does not perform any further operations with the result of the call before returning

```
(define (f x)
  (if (zero? x)
      0
      (f (sub1 x))))
(f 100)
```

Tail Call Optimization

- A compiler optimization that can be applied to recursive functions when the recursive call is in tail position

- When a function call is in tail position, the compiler can reuse the current stack frame for the next function call, instead of creating a new one. This can prevent stack overflow errors and improve performance.

Consider this program

Recursive function:

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

Calling the function:

```
(f 100)
```

Use Mem for `Offset`

```
;; 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)
```

Recursion VS Loop: X86 Code

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

```
(Label 'f)
 (Mov 'rax (Mem rsp 0))
 (Cmp 'rax 0)
 (Je 'done)
 (Lea 'r8 'ret2)
 (Push 'r8)
 (Sub 'rax 16)
 (Push 'rax)
 (Jmp 'f)
 (Label 'ret2)
 (labe 'done)
 (Add 'rsp 8)
 (Ret)
```

```
f(x){
  Loop:
  if x=0
     0
  else
    x = x - 1
    goto Loop
}
```

```
(Label 'f)
(Mov 'rax (Mem rsp 0))
(Cmp 'rax 0)
(Je 'done)
(Sub 'rax 16)
(Mov (Mem rsp 0) 'rax)
(Jmp 'f)
(Label 'done)
(Add 'rsp 8)
(Ret)
```

Recursion VS Loop

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

```
f(x){
  Loop:
  if x=0
     0
  else
    x = x - 1
    goto Loop
}
```

```
(Label 'f)
(Mov 'rax (Mem rsp 0))
(Cmp 'rax 0)
(Je 'done)
(Lea 'r8 'ret2)
(Push 'r8)
(Sub 'rax 16)
(Push 'rax)
(Jmp 'f)
(Label 'ret2)
(labe 'done)
(Add 'rsp 8)
(Ret)
```

```
(Label 'f)
(Mov 'rax (Mem rsp 0))
(Cmp 'rax 0)
(Je 'done)
(Sub 'rax 16)
(Mov (Mem rsp 0) 'rax)
(Jmp 'f)
(Label 'done)
(Add 'rsp 8)
(Ret)
```

Recursion VS Loop

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

```
f(x){
  Loop:
  if x=0
     0
  else
    x = x - 1
    goto Loop
}
```

```
(Label 'f)
(Mov 'rax (Mem rsp 0))
(Cmp 'rax 0)
(Je 'done)
(Lea 'r8 'ret2)
(Push 'r8)
(Sub 'rax 16)
(Push 'rax)
(Jmp 'f)
(Label 'ret2)
(labe 'done)
(Add 'rsp 8)
(Ret)
```

```
(Label 'f)
(Mov 'rax (Mem rsp 0))
(Cmp 'rax 0)
(Je 'done)
(Sub 'rax 16)
(Mov (Mem rsp 0) 'rax)
(Jmp 'f)
(Label 'done)
(Add 'rsp 8)
(Ret)
```

A86 Code for Recursive Functions

```
(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)
```

A86 Code for Recursive Functions

```
(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)
```

When x =0 →

How Does Recursive Call Work?

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

How Does Recursive Call Work?

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

rax=100

```
    (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)
```

rax=99

```
(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)
```

How Does Recursive Call Work?

```
;; 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)
```

rax=100

rax=99

```
(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)
```

rax=0

```
(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)
```

We have reviewed the application and reached preliminary decisions. I was planning to se

Consider this program

```
;; 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)
```

rax=100

```
(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)
```

rax=99

```
(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)
```

rax=0

A view of the stack

[ret addr 1]
[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

[ret addr 1]
[100]
[ret addr 2]
[99]

```
;; 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

[ret addr 1]
[100]
[ret addr 2]
[99]

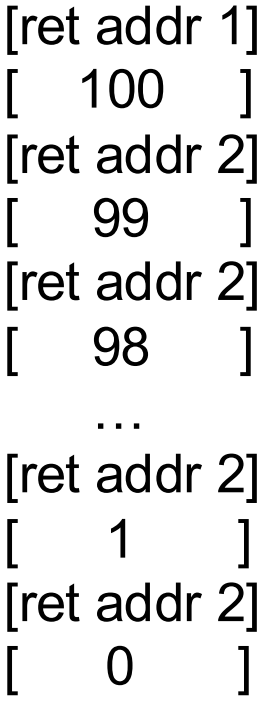
```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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

```
[ret addr 1]
[    100    ]
[ret addr 2]
[     99    ]
[ret addr 2]
[     98    ]
```

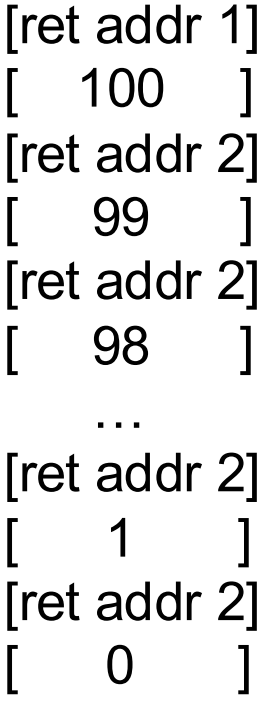
```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



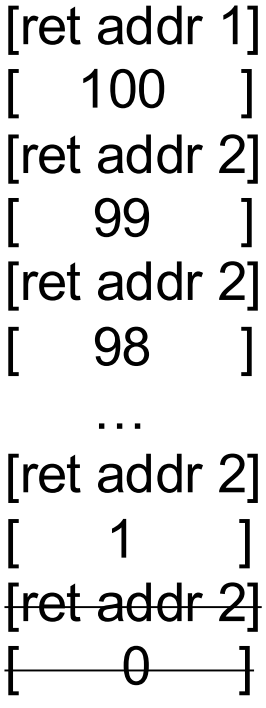

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



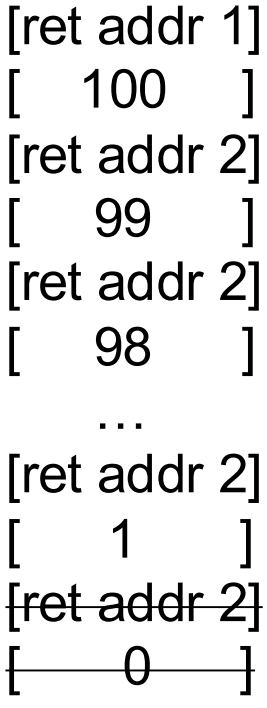

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



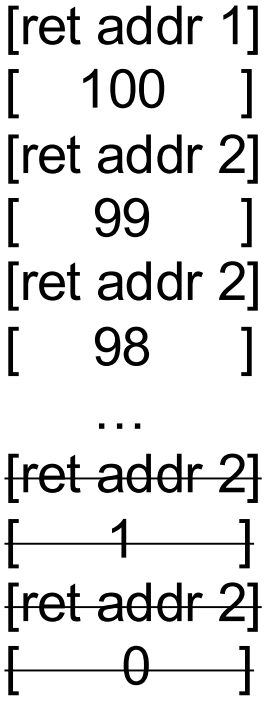

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



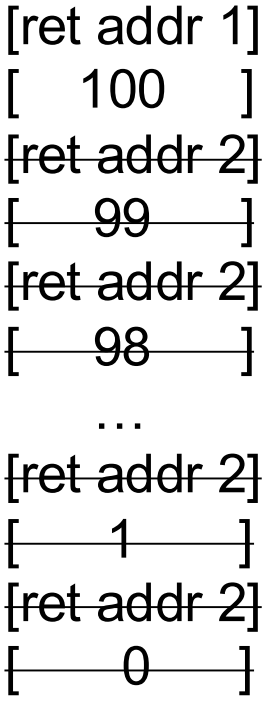

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



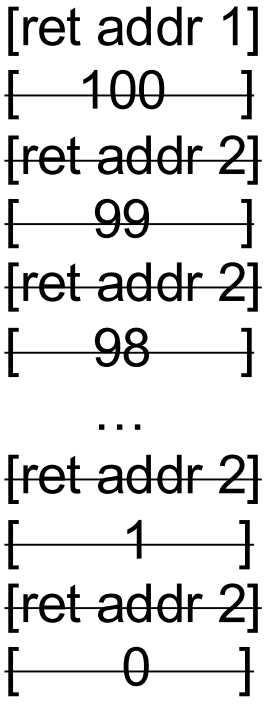

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



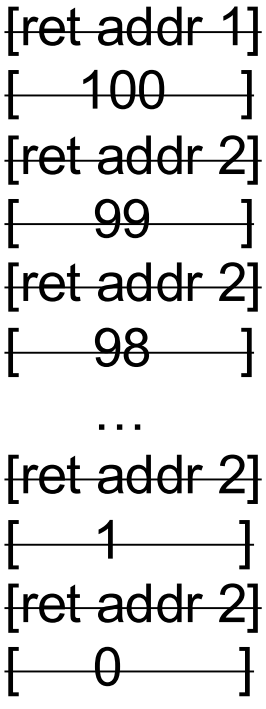

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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



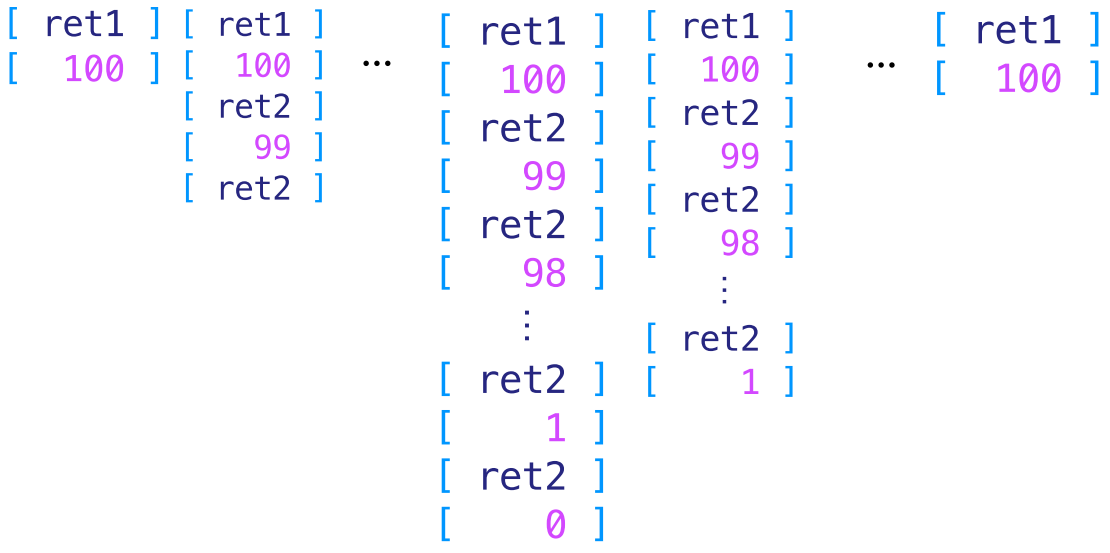

```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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




```
;; 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)
```

Argument is not used after the call

```
; 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)
```

rax=100

```
(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)
```

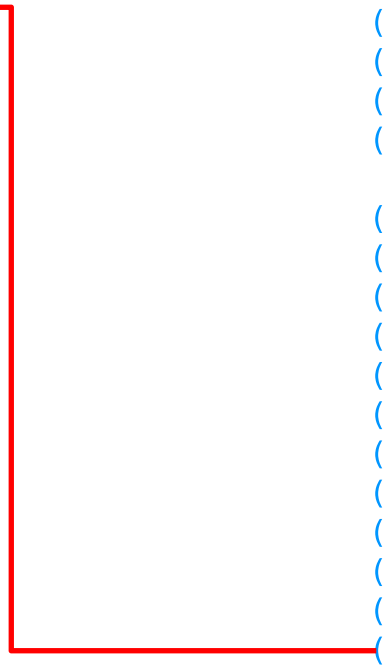
rax=99

```
(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)
```

rax=0

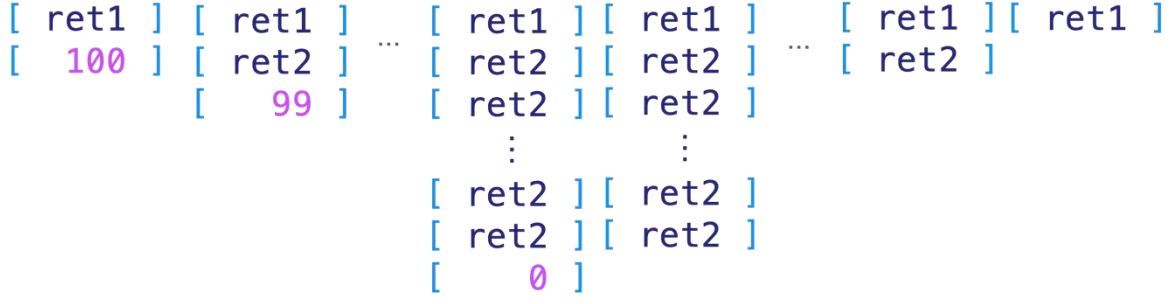
Pop before Call

```
(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)
```



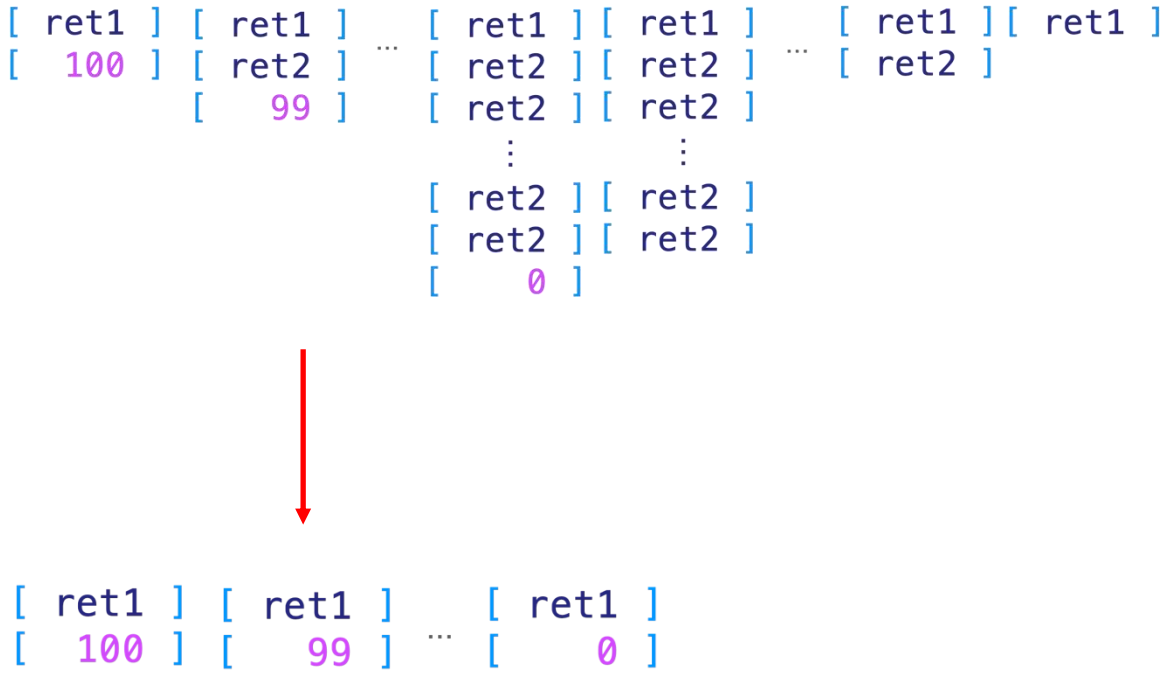
```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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




```
;; 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)
```

Pop before Call




```
;; 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)
```

Before and After

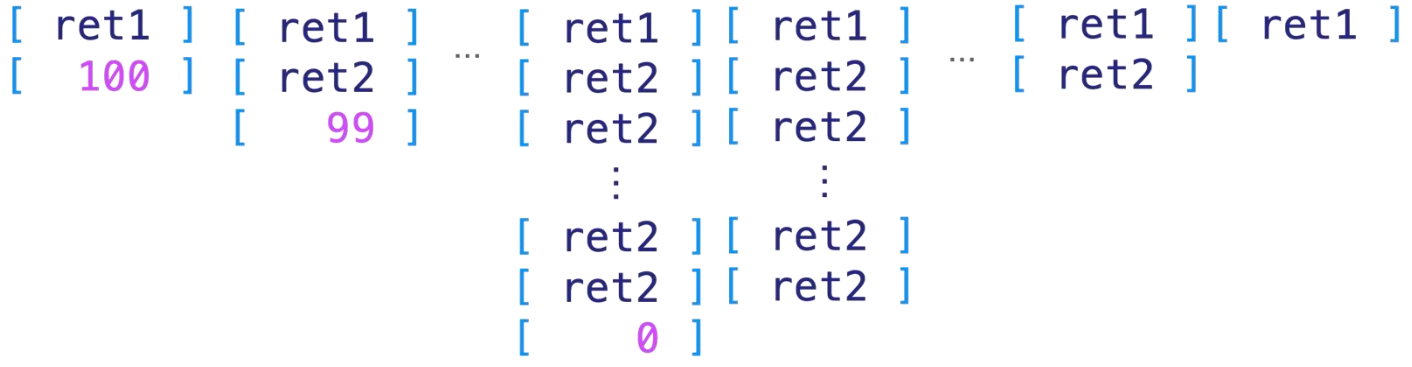
```
;; Revised:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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)
```

```
;; 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)
```

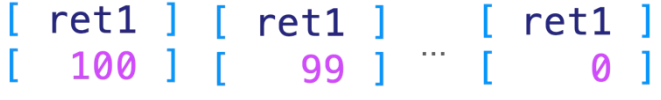
Return to caller

Tail calls save Space

Non-tail call:



Tail call:



Tail Call: 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 to function entry

Can’t optimize if it is not a tail call

Tail Call

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

(f 100)
```

Non-Tail Call

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

(g 100)
```

Rewrite to enable tail calls

```
(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)
```

How to know if a call is 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.

Detecting the tail call

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

t? - is this expression in tail position

Function body is in a 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))]))
```

Detecting the tail call

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

t? - is this expression in tail position

If expression:

```
;; 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))))
```

Detecting the tail call

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

t? - is this expression in tail position

Function Application

```
;; 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)))
```

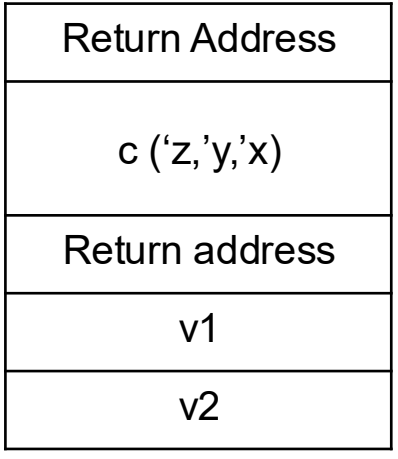
Example Program

```
(define (f a b )
  (let ((x 1))
   (let ((y 2))
    (let ((z 3))
       (+ a b)))))

(f (+ 9 1) 20)))
```

Non-tail calls

- More work to do after call, so need to return

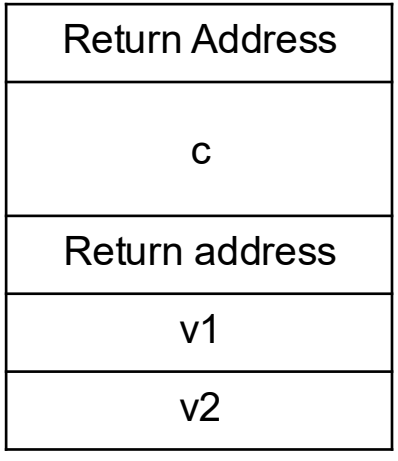
```
(compile-app-nontail 'f (list e1 e2)) c)

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



Non-tail calls

- More work to do after call, so need to return

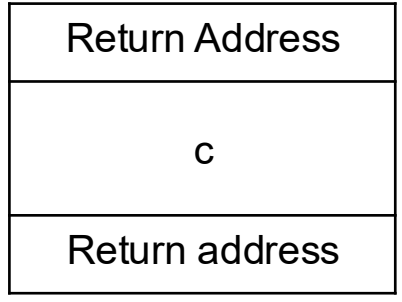
```
(compile-define (Defn 'f '(x y z) e))

(seq
  (Label 'f)
  (compile-e e '(z y x) #t)
  (Add 'rsp 16)
  (Ret))
```



Non-tail calls

- More work to do after call, so need to return

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

(seq
  (Label 'f)
  (compile-e e '(z y x) #t)
  (Add 'rsp 16)
  (Ret))
```



Non-tail call Example

Assume, we compile the following code withoout tail-call optimization

```
(define (f a b)
  (let ((x 10)) (let ((y 20))
    (f (+ x y) (+ x y))))))
```

Non-tail call Example

```
(define (f a b)
  (let ((x 10)) (let ((y 20))
   (f (+ x y) (+ x y))))))
```

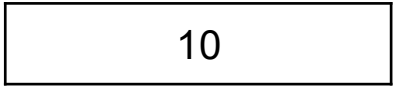
```
(Mov 'rax 160)
(Push 'rax)
(Mov 'rax 320)
(Push 'rax)
(Lea 'rax `ret1)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
```

Argument 1 of f

Argument 2 of f

Non-tail call Example

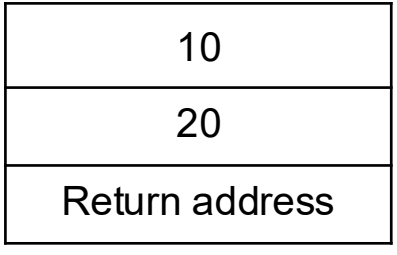
```
(let ((x 10)) (let ((y 20))
  (f (+ x y) (+ x y))))))
```




```
(Mov 'rax 160)
(Push 'rax)
(Mov 'rax 320)
(Push 'rax)
(Lea 'rax 'ret1)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
```

Non-tail call Example

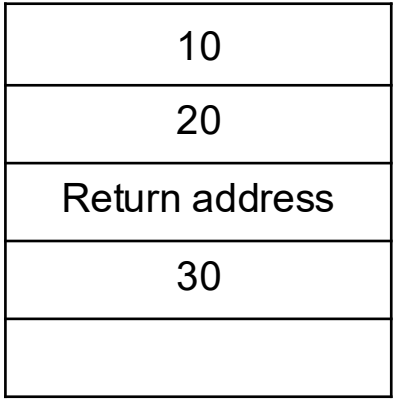
```
(let ((x 10)) (let ((y 20))
  (f (+ x y) (+ x y))))))
```




```
(Mov 'rax 160)
(Push 'rax)
(Mov 'rax 320)
(Push 'rax)
(Lea 'rax `ret1)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
```

Non-tail call Example

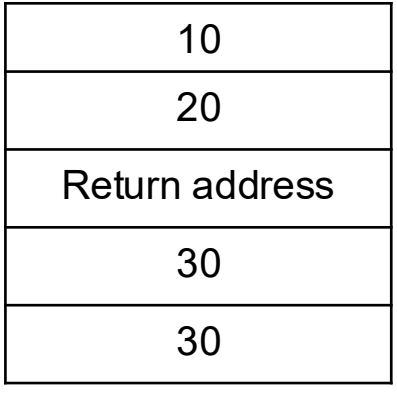
```
(let ((x 10)) (let ((y 20))
  (f (+ x y) (+ x y))))))
```




```
(Mov 'rax 160)
(Push 'rax)
(Mov 'rax 320)
(Push 'rax)
(Lea 'rax `ret1)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
```

Non-tail call Example

```
(let ((x 10)) (let ((y 20))
  (f (+ x y) (+ x y))))))
```

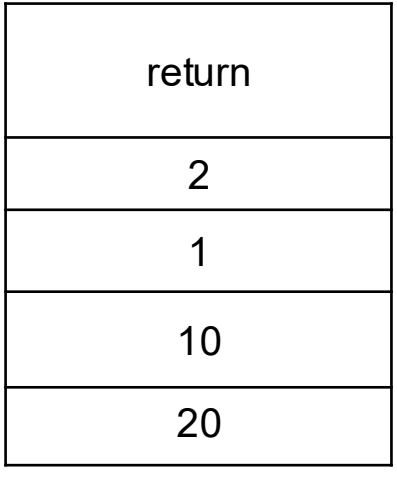



```
(Mov 'rax 160)
(Push 'rax)
(Mov 'rax 320)
(Push 'rax)
(Lea 'rax `ret1)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 16)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Push 'rax)
(Mov 'rax (Mem '(+ rsp 24)))
(Pop 'r8)
(Add 'rax 'r8)
(Push 'rax)
```

Non-tail call Example

```
(let ((x 10)) (let ((y 20))
  (f (+ x y) (+ x y))))))
```

```
(f 1 2)
```




```
(Jmp 'label_f_5e96933745)  ;;recursivel call
(Label 'ret76975)          ;;return from recursion
(Add 'rsp 8)               ;;pop (y,20)
(Add 'rsp 8)               ;;pop (x,10)
(Add 'rsp 16);;pop the arguments to f
(Ret)
```

Non-tail call Example

```
(let ((x 10)) (let ((y 20))
  (f (+ x y) (+ x y))))))
```

Return address (to main)

```
(Label 'label_f_5e96933745)
 (Mov 'rax (Mem '(+ rsp 8)))
 (Push 'rax)
 (Mov 'rax (Mem '(+ rsp 8)))
 (Pop 'r8)
 (Add 'rax 'r8)
 (Add 'rsp 16)
 (Ret)
```

Tail call Example

```
(define (f a b)
  (let ((x 1))
   (let ((y 2))
    (let ((z 3))
      (if (= a 0)
          1
          (f (sub1 a) b))))))

'(f 1 2)))
```

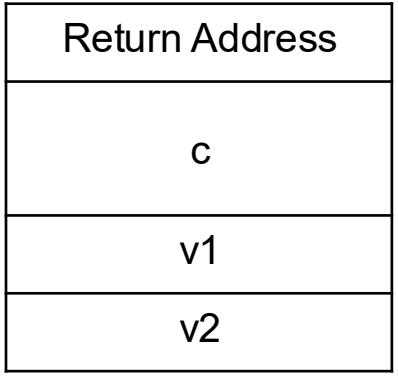
Move Args: (move-args 2 5)

```
(define (f a b)
  (let ((x 1))
  (let ((y 2))
  (let ((z 3))
  (if (= a 0)
     1
      (f (sub1 a) b))))))

'(f 10 20)))
```

Tail calls (first attempt)

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

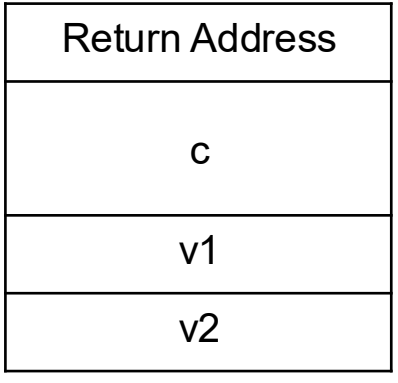
```
(compile-app-tail 'f (list e1 e2)) c)

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



Tail calls (first attempt)

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

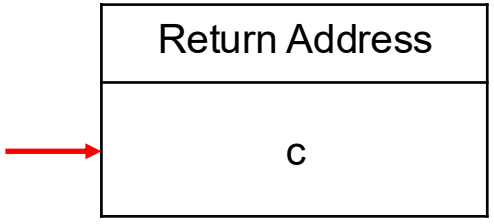
```
(compile-define `f `(b a) e))
(seq
  (Label `f)
  (compile-e e `(b a) #t)
  (Add `rsp 16)
  (Ret))
```



Tail calls (first attempt)

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

```
(compile-define 'f '(b a) e))

(seq
  (Label 'f)
  (compile-e e '(b a) #t)
  (Add 'rsp 16)
  (Ret))
```



Tail calls (second attempt)

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

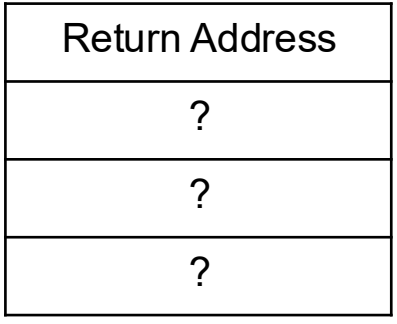
```
(compile-app-tail 'f (list e1 e2) c)

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

Return Address
v1
v2

Tail calls (third attempt)

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

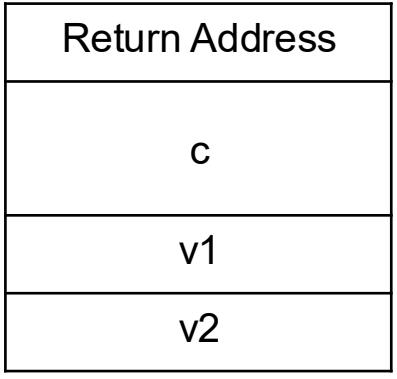
```
(compile-app-tail 'f (list e1 e2) c)

(seq
  (compile-e e1 c)
  (Push 'rax)
  (compile-e e2 (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) c)

(seq
  (compile-e e1 c)
  (Push 'rax)
  (compile-e e2 (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

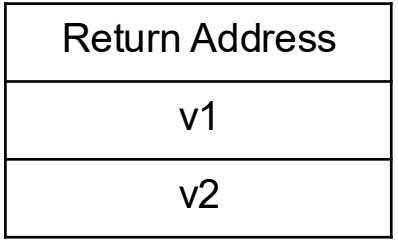
```
(compile-app-tail 'f (list e1 e2) c)

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

Return Address
v1
v2

Tail calls (final attempt)

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

```
(compile-define (Defn 'f '(b a) e))

(seq
  (Label 'f)
  (compile-e e '(b a) #t)
  (Add 'rsp 16)
  (Ret))
```



Tail calls (final attempt)

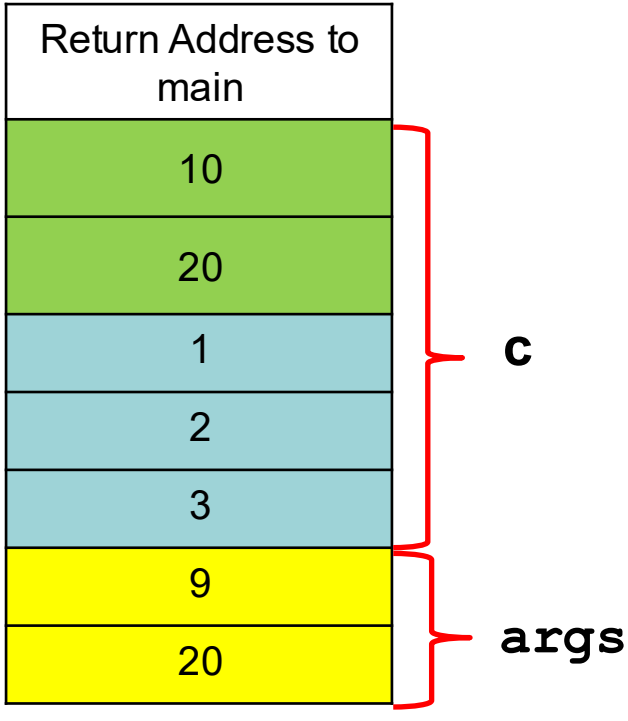
- No more work to do after call, so don't return

```
(compile-define (Defn 'f '(b a) e))

(seq
  (Label 'f)
  (compile-e e '(list '(b a) #t)
  (Add 'rsp 16)
  (Ret))
```

Move Args: (move-args 2 5)

```
(define (f a b)
  (let ((x 1))
  (let ((y 2))
  (let ((z 3))
  (if (= a 0)
      1
      (f (sub1 a) b))))))

'(f 10 20)))
```

Before move args



Move Args: (move-args 2 5)