

# **CMSC 430 - 4 September 2025**

## **Introduction to Introduction to Compilers**

Course logistics

How to 430

What is a compiler?

Quick overview

Ocaml to Racket

Before next class

# **Compilers comes at you fast**

## **Course logistics**

Lectures every Tues & Thurs (except holidays, midterms)

~10 assignments

2 midterms (take-home, 24 hour); 10/16, 11/13

1 final project (counts as final exam); 12/18

Several surveys and quizzes (on ELMS)

Course is very cumulative; building essentially one program all semester

# Key resources

## Course logistics

- Class web page (syllabus, assignments, course notes)
- ELMS (announcements, recordings, grades, slides)
- Piazza (communication, discussion)
- Gradescope (submissions)

# Day to day: lectures

## How to CMSC 430

- Lectures will use slides, live-coding, lecturing, and discussing
- Be here now
- Recordings of lectures posted after class
- Quizzes help reinforce lecture concepts due **before** next class

# **Day to day: studying**

## **How to CMSC 430**

- Course notes are comprehensive and flesh out lecture material
- Best learned by doing
- Material is alive, interact with it
- Requires significant time outside of class
- Talk to your peers
- Spend time with TAs in off-peak hours
- Participate on Piazza (teaching others is a great way to learn)

# Day to day: assignments

## How to CMSC 430

- Study the material before starting
- Think first
- Start early
- Submit often
- Approach problems systematically (more on this later)
- Writing lots of code is the wrong path
- Going slower will get you there faster

# Day to day: midterms

## How to CMSC 430

- 24-hour take-home exams
- Same advice as assignments
- Designed to be easily done in a couple hours without rush\*
- Will take more than 24 hours if you haven't already mastered material
- See practice midterms for best preview

\* If you have already mastered the material!

# Day to day: final project

## How to CMSC 430

- Like an assignment, slightly larger, but also more time
- Designed to relieve stress at end of semester\*
- Due at end of final exam period for this class

\* If you don't leave it to last minute!

# **What is a programming language?**

**No, really, I'm asking...**

An abstracted way to tell a computer what operations to perform

Human readable way of representing instructions for a computer

All program languages have syntax and rules in order to be valid

# **What is a compiler?**

**No, really, I'm asking...**

It's what reads and deciphers programs

Breaks down abstractions to machine code the machine can understand

Ensures that programs are executable

Optimizes code

Responsible for machine code of different systems

A program

Can understand multiple languages

# **Design & implementation of programming languages**

## **Quick overview**

We're going to build a programming language

- with modern features
- implemented via compilation
- paying close attention to correctness

# Design & implementation of programming languages

## Quick overview

We're going to build a programming language

- with modern features: higher-order functions, data-types & pattern matching, automatic memory management, memory safety, etc.
- implemented via compilation: targeting an old, widely used machine-level language, x86, with a run-time system written in C
- paying close attention to correctness: using interpreters as our notion of specification, validated by informal reasoning and testing

# Design & implementation of programming languages

## Quick overview

We're going to build a programming language

- Source language: Racket (like OCaml w/o types, different syntax)
- Target language: x86
- Host language: Racket

Final result: self-hosting compiler (compiles its own source code)

# Ocaml to Racket

**Racket = OCaml - Types - Syntax**

Download and install Racket

Read and follow course notes chapter on “From OCaml to Racket”

# Things to do

## Before next class

- Getting to know you survey on ELMS (**due by midnight**)
- Racket Basics quiz on ELMS (due by **start** of class Tuesday)
- Read syllabus on course web page
- Download and install Racket
- Study “From OCaml to Racket”
- Get a start on Assignment 1: Racket primer (due 9/11)
- Take a look at a86 notes covered Tuesday

# CMSC 430 - 9 September 2025

## OCaml to Racket (cont)

### Today

- More Racket: symbols, lists, structs, s-expressions, systematic programming
- Introduction to a86

### Announcements

- Assignment 1: due 9/11 by midnight; may be done collaboratively
- Racket Basics quiz on ELMS, due 9/11 **by start of class**
- Slight tweak to office hours schedule, now with evening hours & new TA

# Addendum to last time

## Additions, corrections, and changes

Anonymous feedback form

ADS accommodations

Final project does **not** provide graded feedback before deadline

Grade intervals changed to be more familiar

# Quick look at Survey Results

Favorite languages: Python, Java, C, OCaml, Rust

Something that would help you succeed:

- Clear instructions, examples, and communication
- Office hours and TA support
- Staying on top of assignments

How many hours per day outside of class: 1 – 3

Have you read the syllabus: 50/50 split

# Quick review of errors

## Parse, syntax, & run-time errors

- **Parse** error: not grammatically well formed string
- **Syntax** error: not correctly “shaped” expression, unbound variables
- **Run-time** error: well-formed program that errors when run

# Symbols

## An atomic string-like datatype

Symbols are a useful datatype for representing enumerations

- 'red 'yellow 'green
- 'up 'down 'left 'right

Symbols are literals, written with the quote notation (more later). Two symbols are equal (`eq?`) if they are spelled the same.

# Lists and pairs in Racket vs OCaml

## Constructors in OCaml

OCaml lists:

- `[]` :  $\alpha$  list
- `(::)` :  $\alpha \rightarrow \alpha$  list  $\rightarrow \alpha$  list
- `[ . ; . ; . ; ... ]` convient notation for lists

OCaml pairs (and tuples):

- `( , )` :  $\alpha \rightarrow \beta \rightarrow \alpha * \beta$

Pairs and lists: **fundamentally different things**

# Lists and pairs in Racket vs OCaml

## Constructors in Racket

Racket lists:

- `'()` :  $\alpha$  list
- `cons` :  $\alpha \rightarrow \alpha$  list  $\rightarrow \alpha$  list
- `list` convenient function for lists

Racket pairs (and tuples):

- `cons` :  $\alpha \rightarrow \beta \rightarrow \alpha * \beta$

Every *list* is either the empty list or the cons of an element onto a *list*.

Every *pair* is the cons of two values.

(All non-empty lists are pairs, too)

(Chains of pairs that don't end in the empty list are called "improper lists" and print with a ".")

Pairs and lists: **made out of the same stuff**

# Lists and pairs in Racket vs OCaml

## Destructors in OCaml

Pattern matching using constructors for empty, cons, and tuples:

[ ] , :: , ( \_, \_ )

fst, snd functions for pairs (2-tuples)

hd, tl functions for lists

# Lists and pairs in Racket vs OCaml

## Destructors in Racket

Pattern matching using constructors for empty, cons: '(), cons

car, cdr functions for pairs

first, rest functions for lists

# Literal pairs and lists

## A notation for writing down compound literals

Lists of literals can be written using the quote notation:

- ' ()
- ' (1 2 3)
- ' (x y z)
- ' ("x" "y" "z")
- ' ((1) (2 3) (4))

Pairs of literals can be written using the quote notation:

- ' (#t . #f)
- ' (7 . 8)
- ' (1 2 3 . #f)

# Structures

## Defining new record types

(struct coord (x y z)) defines:

- coord : constructor, pattern
- coord- {x, y, z} : accessor functions
- coord? : predicate

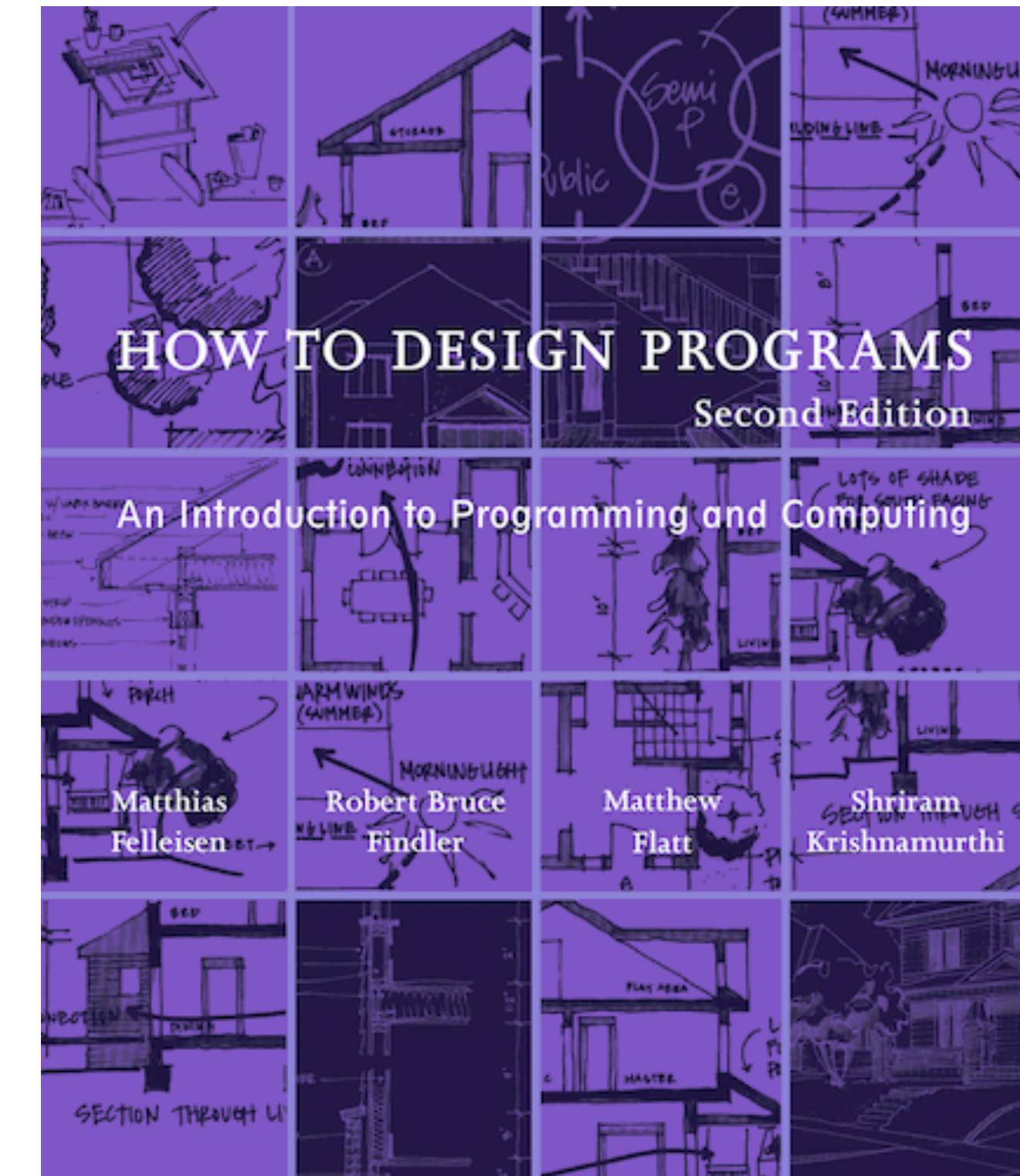
# Systematic programming

## Steps of systematic program design

1. From Problem Analysis to Data Definitions
2. Signature, Purpose Statement, Header
3. Examples
4. Template
5. Definition
6. Testing

Keys:

- write examples before code
- structure of functions follow structure of data
- live and die by the function signature



# CMSC 430 - 11 September 2025

## A little assembly

### Today

- x86: the terrible
- a86: the not-so-bad

### Announcements

- More Racket Basics quiz on ELMS, due 9/16 **before class**
- Assignment 2, due 9/18; may be done collaboratively
- Assignment 3, due 9/25; **not collaborative**

# **Quick reminders**

## **Useful things**

- Anonymous feedback form on webpage
- Lecture recordings on ELMS (under Panopto)
- Slides on ELMS (under Files), updated after class
- Course notes on webpage

# x86: Computational model

## von Neumann style

Machine state:

- registers - hold 64-bit integers
- flags - a few bits
- memory - big array of bits
- program counter - location of current instruction (represented by bits)

Computation:

- fetch instruction at pc
- execute (reads and modifies state)
- rinse and repeat

x86 is a programming language,  
with a *physical* interpreter

# Some a86 instructions

- Mov
- Add, Sub
- And, Or, Xor, Not
- Sal, Sar
- Label, Jmp, Call, Ret
- Push, Pop
- Cmp
- J\*: Je, Jne, Jl, Jle, Jg, Jge
- Cmov\*: Cmove, Cmovne,  
Cmovl, Cmovle, Cmovg,  
Cmovge

# Mov

- Mov reg int64 - copy integer literal into reg
- Mov reg1 reg2 - register move: copy contents of reg2 into reg1
- Mov offset reg - memory write: copy contents of reg into memory
- Mov reg offset - memory read: copy contents of memory into reg
- ~~Mov offset1 offset2~~ - illegal instruction, have to go through a register
- Offset reg int64 - interpret reg as a pointer and dereference at given offset

# Add, Sub

- Add reg int64 - add integer literal into reg
- Add reg1 reg2 - add reg2 into reg1
- Sub reg int64 - subtract integer literal into reg
- Sub reg1 reg2 - subtract reg2 into reg1

# And, Or, Xor, Not

- And reg1 reg2 - bitwise and of reg1 and reg2, result in reg1
- And reg int64 - bitwise and of reg and int literal, result in reg
- Or, Xor: like And but bitwise or and xor respectively.
- Not reg - flip each bit in reg

# Sal, Sar

- Sal reg int - shift arithmetic left: shift bits of reg to the left (padding with zero on the right)
- Sar reg int - shift arithmetic right: shift bits of reg to the right (padding with the sign bit)
- Mathematically, Sal does iterative doubling, Sar does iterative halving

# CMSC 430 - 16 September 2025

## A little assembly; our first compiler

### Today

- a86: review
- a compiler for a subset of Racket

### Announcements

- a86 Basics quiz on ELMS, due 9/18 **before class**
- Assignment 2, due 9/18; may be done collaboratively
- Assignment 3, due 9/25; **not collaborative**

# Label, Jmp, Call, Ret

- Label name: not an instruction, but a name for a place in the code
- Jmp name: jump to the place in the code labelled name
- Call name: jump to the place in the code labelled name (and set up Ret to jump back here)\*
- Ret: jump to the instruction after the most recent Call\*

\* There's more nuance to this that we'll see when we discuss the stack.

# Push, Pop

- Push reg - push contents of reg onto the stack
- Push int32 - push integer literal onto the stack
- Pop reg - pop top element of stack into reg
- Under the hood: these instructions manipulate the rsp register, which holds a pointer to the “top” of the stack
  - Push: decrement rsp, write to memory
  - Pop: read from memory, increment rsp

# The rsp register

Special designated register that points to the stack

- Stack grows “downward”, toward lower addresses
- If you overwrite the rsp register, you lose access to the stack (unless you stashed a pointer somewhere else)
- Need to leave stack in the state you found it in (if you push, you need to eventually pop)
- What if you Pop an empty stack? 
- What if you access elements beyond the end of the stack? 

# Reading the stack (without popping)

- Use memory reads via rsp to read elements of the stack without doing a Pop
- Mov reg (Offset rsp 0) - copies first element of stack into reg
- Mov reg (Offset rsp 8) - copies second element of stack into reg
- Mov reg (Offset rsp  $n*8$ ) - copies  $(n+1)$ th element of stack into reg
- Why multiples of 8? Memory is byte-addressed.  
1 byte = 8 bits, 8 bytes = 64 bits.

# Writing to the stack (without pushing)

- Use memory writes via rsp to overwrite elements of the stack without doing a Push
- Mov (Offset rsp 0) reg - copies reg into first element of stack
- Mov (Offset rsp 8) reg - copies reg into second element of stack
- Mov (Offset rsp  $n^*8$ ) reg - copies reg into  $(n+1)$ th element of stack

# Cmp

- Cmp reg int32: compare contents of reg to integer literal
- Cmp reg1 reg2: compare contents of reg1 to reg2
- Comparison instruction updates the state of the CPU to reflect how the comparison came out. Other instructions depend on this state.

# J\*: conditional jumps

- A family of instructions that may jump, depending on the comparison state of the CPU
- Je name: jump to location labelled name *if* comparison was equal
- Jne name: jump to location labelled name *if* comparison wasn't equal
- Jg name: jump to location labelled name *if* comparison was greater
- ... Jl, Jge, Jle: jump if lesser, greater or equal, lesser or equal, resp.

# Cmov\*: conditional moves

- A family of instructions that may move, depending on the comparison state of the CPU
- Cmove reg1 reg2: move reg2 into reg1 *if* comparison was equal
- Cmovne reg1 reg2: move reg2 into reg1 *if* comparison wasn't equal
- ... Cmovg, Cmovl, Cmovge, Cmovle: move if greater, lesser, greater or equal, lesser or equal, resp.

# (De)Allocating on the stack

- (Increment) decrement on the stack to (de-) allocate on the stack without pushing or popping
- Sub rsp 8 - allocate one (undefined) element on stack
- Sub rsp 16 - allocate two (undefined) elements on the stack
- Sub rsp  $n^*8$  - allocate  $n$  (undefined) elements on the stack
- Add rsp  $n^*8$  - deallocate  $n$  elements

# Push, Pop (for real)

- Push reg =  
Sub rsp 8  
Mov (Offset rsp 0) reg
- Pop reg =  
Mov reg (Offset rsp 0)  
Add rsp 8

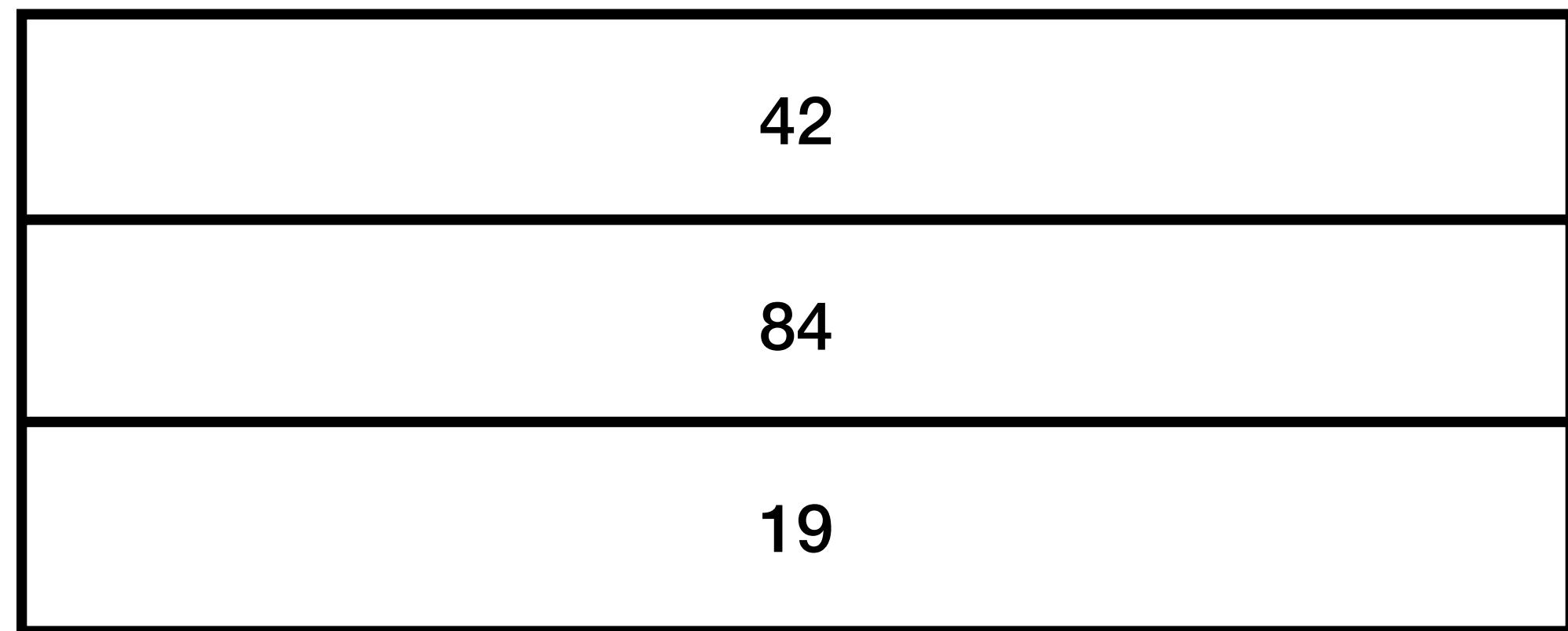
# Call, Ret revisited

- Call and Ret both manipulate the stack to save where to return to
- Call: pushes the location of this instruction on to the stack
- Ret: pops the top element of the stack and jumps to that location
- Call name =  
Push current instruction location  
Jmp name
- Ret =  
Add rsp 8  
Jmp (Offset rsp -8)
- Careful to always restore stack to original state before Ret, otherwise not returning where you intended!

# Examples with the stack

- Push 42 ←
- Push 84 ←
- Push 19 ←
- Pop 'rax ←
- Pop 'rax ←
- Pop 'rax ←

Stack:

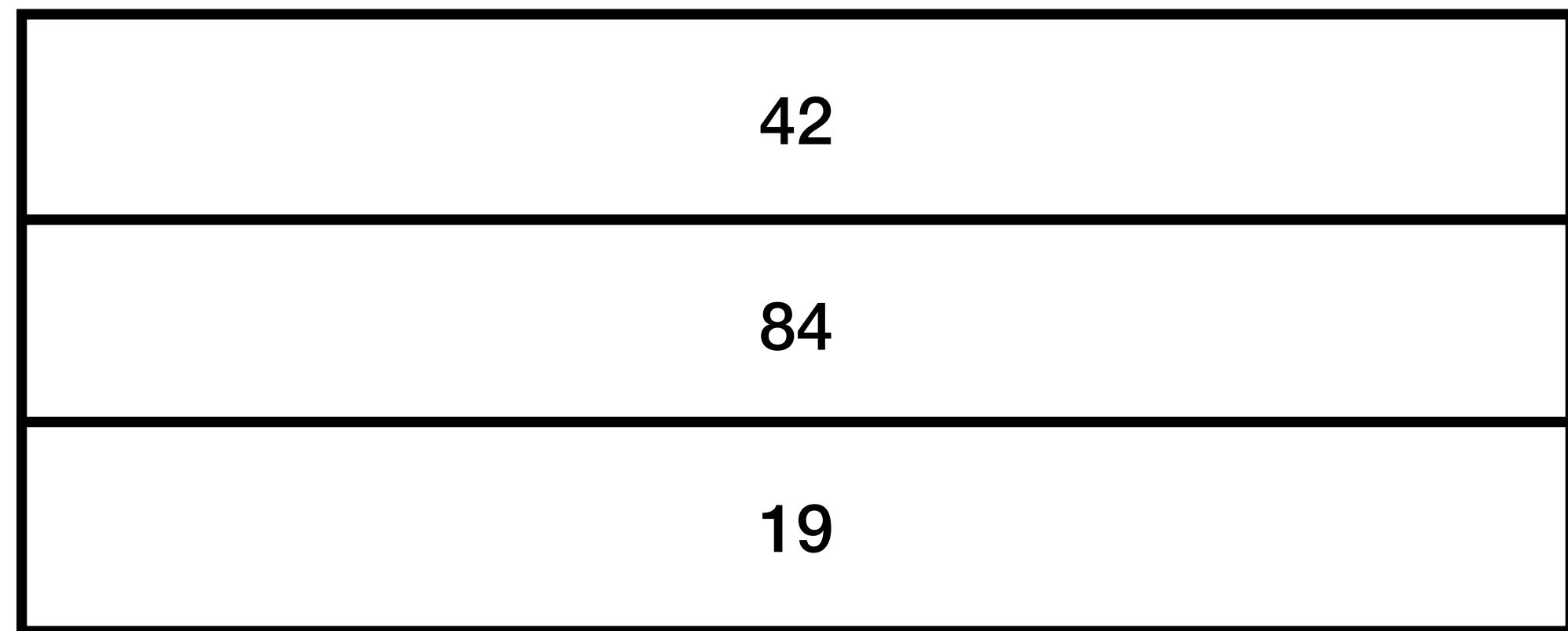


rax: 84

# Examples with the stack

- Push 42 ←
- Push 84 ←
- Push 19 ←
- Add ‘rsp’ 8 ←
- Add ‘rsp’ 8 ←
- Add ‘rsp’ 8 ←

Stack:

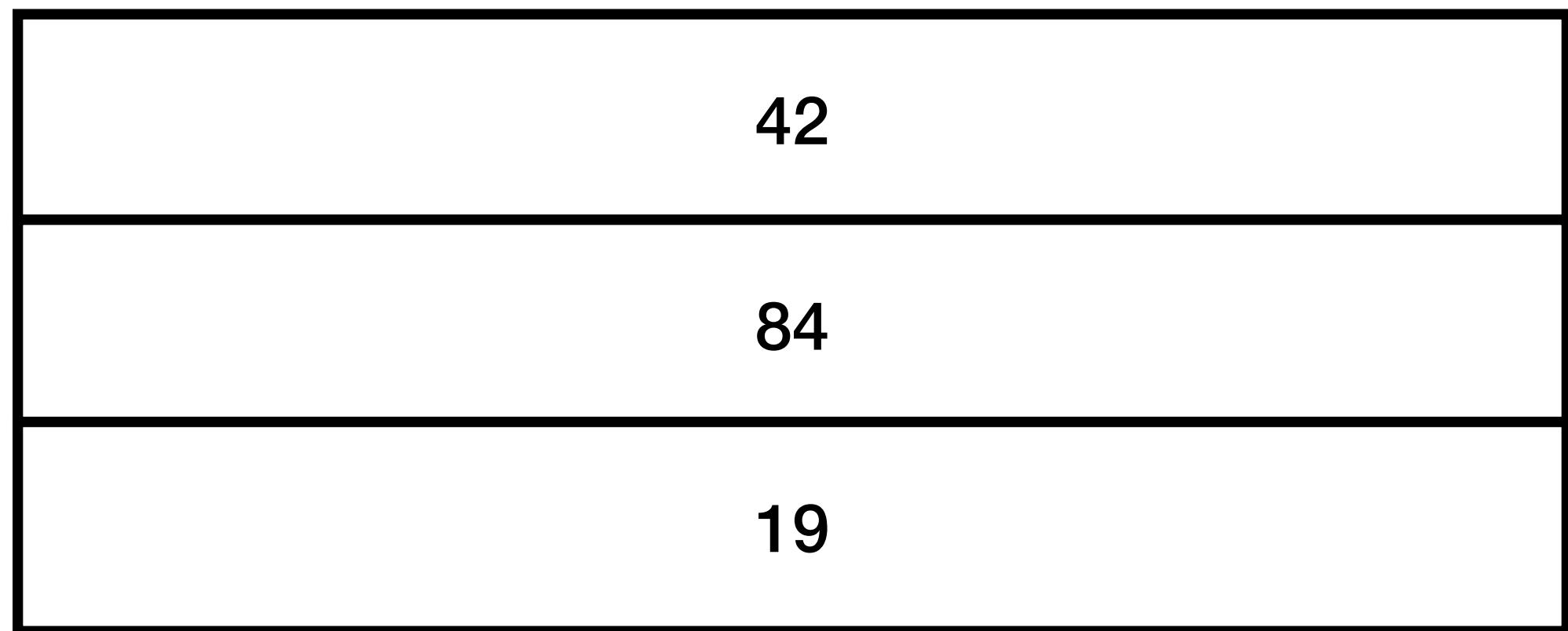


rax: ?

# Examples with the stack

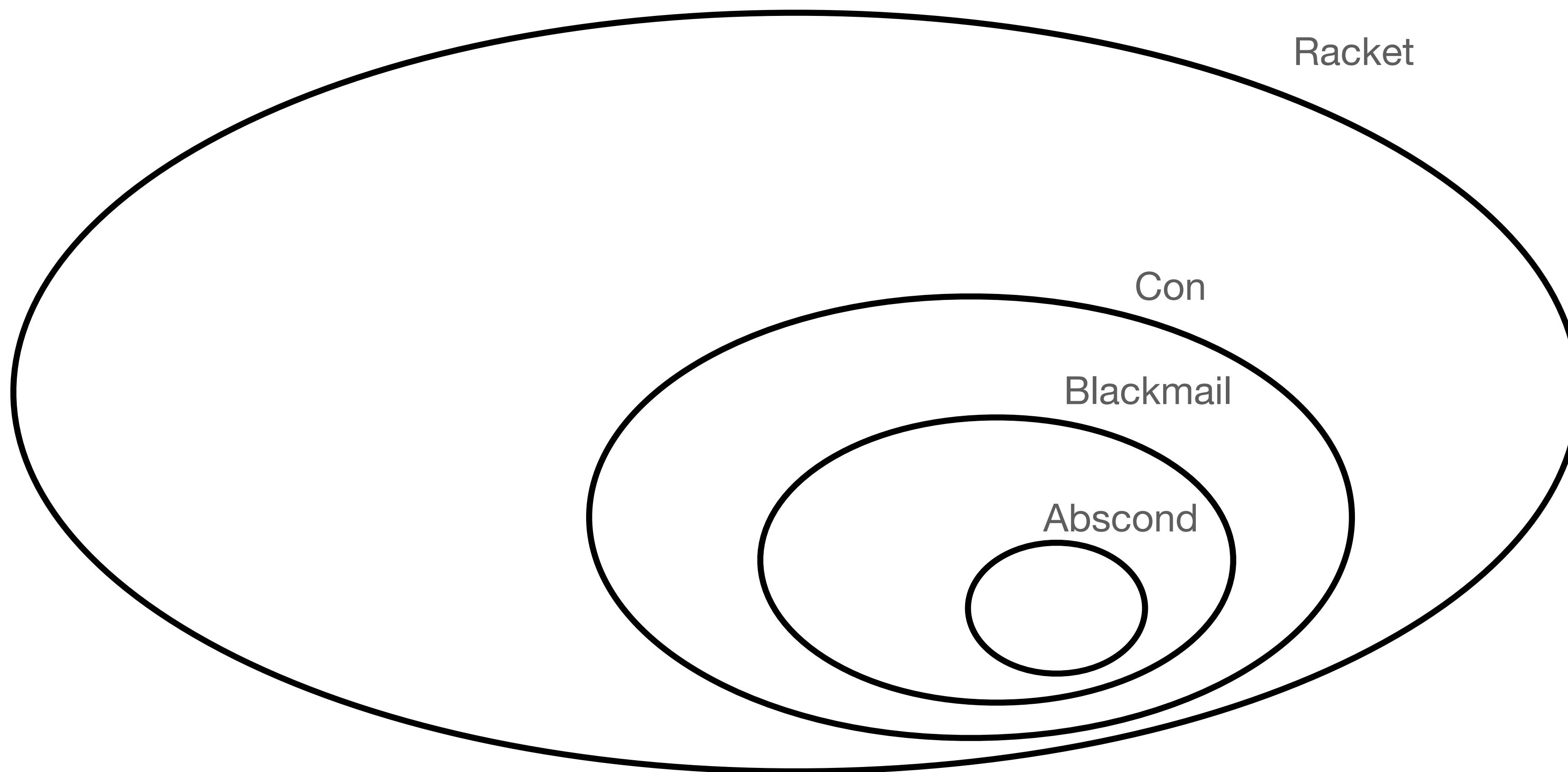
- Push 42 ←
- Push 84 ←
- Push 19 ←
- Add ‘rsp’ 24 ←

Stack:



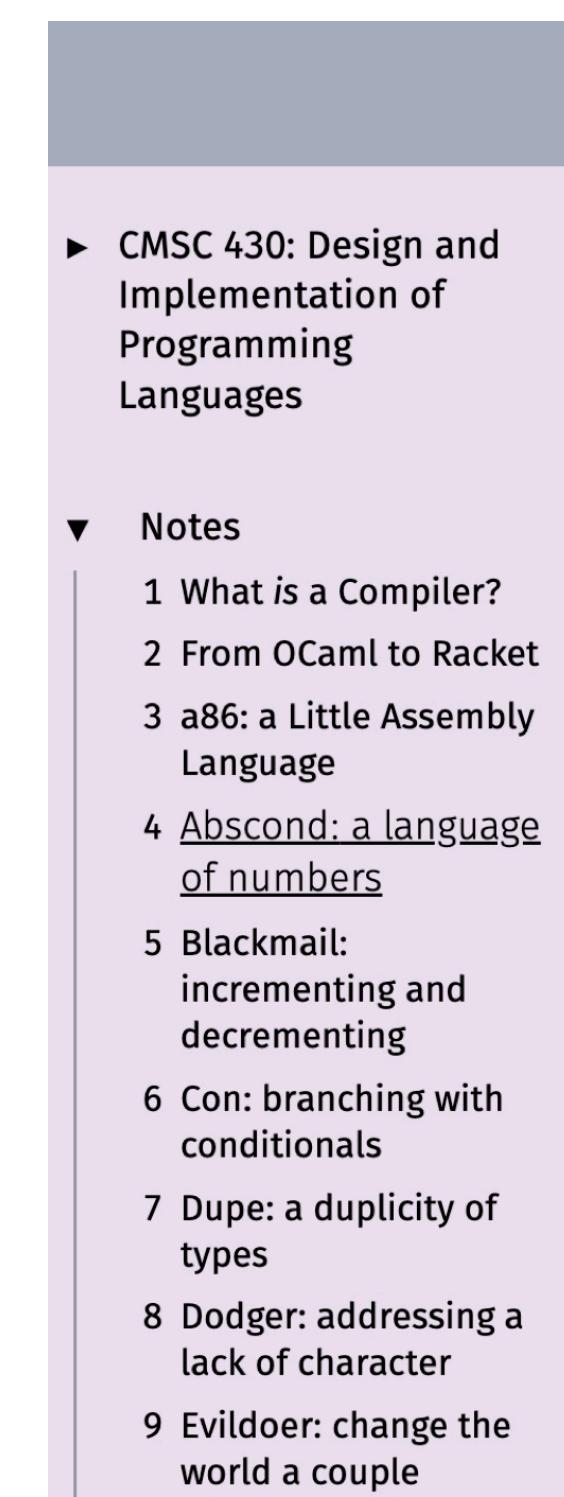
rax: ?

# Language subsets



# Accessing the source code

Complete source code  
for each language linked  
to in notes:

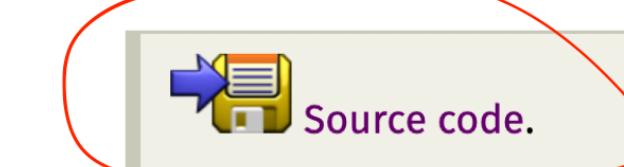


► CMSC 430: Design and Implementation of Programming Languages

▼ Notes

- 1 What is a Compiler?
- 2 From OCaml to Racket
- 3 a86: a Little Assembly Language
- 4 [Abscond: a language of numbers](#)
- 5 Blackmail: incrementing and decrementing
- 6 Con: branching with conditionals
- 7 Dupe: a duplicity of types
- 8 Dodger: addressing a lack of character
- 9 Evildoer: change the world a couple of bits at a time

## 4 Abscond: a language of numbers



*Let's Make a Programming Language!*

- 4.1 Overview
- 4.2 Concrete syntax for Abscond
- 4.3 Abstract syntax for Abscond
- 4.4 Meaning of Abscond programs
- 4.5 Toward a Compiler for Abscond
- 4.6 An Example
- 4.7 A Compiler for Abscond
- 4.8 But is it *Correct*?

### 4.1 Overview

A compiler is just one (optional!) component of a *programming language*. So if you want to make a compiler, you must first settle on a programming language to compile.

# Interpreters

## One approach to language specification

Idea: write a program: interp : Expr → Value

- simpler than writing compiler
- consider it the specification for compiler

```
;; Expr -> Boolean
;; Is the compiler correct on e?
(define (compiler-correct? e)
  (= (asm-interp (compile e))
     (interp e)))
```

# Parts of our interpreter

Reader : Input → S-Expr

Parser: S-Expr → Expr

**Interpreter: Expr → Value**

# Interpreter structure

## Key files and usage for interpreter

`interp-stdin.rkt`: interpret source code on stdin to value on stdout

`ast.rkt`: type definition for AST

`parse.rkt`: s-expression to AST parser

`interp.rkt`: AST interpreter

`racket -t interp-stdin.rkt -m`: runs the interpreter

# Parts of our compiler

Reader : Input → S-Expr

Parser: S-Expr → Expr

**Compiler: Expr → a86**

Assembler: a86 → Object

Linker: Object → Executable

Runtime system: C code linked together w/ program object code

# Compiler structure

## Key files and usage for compiler

compile-stdin.rkt: compile source code on stdin to x86 on stdout

ast.rkt: type definition for AST

parse.rkt: s-expression to AST parser

compile.rkt: AST to a86 compiler

main.c, print.c, print.h: run-time system

racket -t compile-stdin.rkt -m: runs the compiler

# Compiler utilities

## Key files and usage for compiler utilities

`run-stdin.rkt`: compile code on stdin, execute it

`exec.rkt`: AST to result via compilation

`racket -t run-stdin.rkt -m`: runs the compiler and executes

# Recipe for growing a language

- Write examples
- Extend concrete syntax
- Extend abstract syntax
- Extend parser
- Revise interpreter to specify semantics
- Revise compiler & run-time system to implement semantics
- Test against examples

# Con: adding conditional expressions

Concrete examples:

```
(if (zero? 1) 2 3) ;=> 3
(if (zero? (sub1 1)) 2 3) ;=> 2
(add1 (if (zero? (sub1 1)) 2 3)) ;=> 3
(add1 (if (zero? (sub1 1)) (add1 1) 3)) ;=> 3
```

# Con: adding conditional expressions

Non-examples (these are not in the set Con):

( zero? 0 )

( if #t 2 3 )

( if 1 2 3 )

# CMSC 430 - 18 September 2025

## Introducing a duplicity of types

### Today

- Quick review: Abscond, Blackmail, Con
- Dupe: a duplicity of types
- Dodger: more types

### Announcements

- Assignment 2, due by midnight; may be done collaboratively
- Assignment 3, due 9/25; **not collaborative**
- No quiz out today

# Abstract Syntax Trees

Backbone of all compilers, interpreters, etc.

The AST datatype is the backbone of our compiler (and any computation that processes programs): `compile : Expr -> Asm`

```
;; type Expr = (Lit Integer)
;;           | (Prim1 Op1 Expr)
```

As language evolves, so will AST definition and all functions on ASTs:

```
;; Expr -> ?
(define (expr-template e)
  (match e
    [((Lit i) (... i ...))
     ((Prim1 p e)
      (... p (expr-template e) ....))]))
```

# Compiler for Blackmail

```
;; Expr -> Asm
(define (compile e)
  (prog (Global 'entry)
        (Label 'entry)
        (compile-e e)
        (Ret)))

;; Expr -> Asm
(define (compile-e e)
  (match e
    [(Lit i) (seq (Mov rax i))]
    [(Prim1 p e) (compile-prim1 p e)]))

;; Op1 Expr -> Asm
(define (compile-prim1 p e)
  (seq (compile-e e)
       (compile-op1 p)))

;; Op1 -> Asm
(define (compile-op1 p)
  (match p
    ['add1 (Add rax 1)]
    ['sub1 (Sub rax 1)]))
```

# Con: adding conditional expressions

Concrete examples:

```
(if (zero? 1) 2 3)
(if (zero? (sub1 1)) 2 3)
(add1 (if (zero? (sub1 1)) 2 3))
(add1 (if (zero? (sub1 1)) (add1 1) 3))
```

Abstract examples:

```
(IfZero (Lit 1) (Lit 2) (Lit 3))
(IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3))
(Prim1 'add1 (IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3)))
(Prim1 'add1 (IfZero (Prim1 'sub1 (Lit 1)) (Prim1 'add1 (Lit 1))
(Lit 3)))
```

# Con: adding conditional expressions

## Revised template for Con

```
;; type Expr = (Lit Integer)
;;           | (Prim1 Op1 Expr)
;;           | (IfZero Expr Expr Expr)

;; Expr -> ?
(define (expr-template e)
  (match e
    [(Lit i) (... i ...)]
    [(Prim1 p e) (... p (expr-template e) ...)]
    [(IfZero e1 e2 e3)
     (... (expr-template e1)
          (expr-template e2)
          (expr-template e3) ...)])))
```

The diagram illustrates the recursive nature of the `expr-template` function. Three green ovals highlight the occurrences of `(expr-template e)` in the code. A blue arrow points from the first oval (in the `Prim1` case) to the second (in the `IfZero` case), and another blue arrow points from the second to the third (in the `Else` case), demonstrating how the function template is applied to each branch of the conditional expression.

# Con: adding conditional expressions

## Essence of the compiler for Con

```
;; Expr Expr Expr -> Asm
(define (compile-ifzero e1 e2 e3)
  (... (compile-e e1)
        (compile-e e2)
        (compile-e e3) ...))
```

# Con: adding conditional expressions

## Essence of the compiler for Con

```
;; Expr Expr Expr -> Asm
(define (compile-ifzero e1 e2 e3)
  ;; Goal: instrs that leave (if (zero? e1) e2 e3) value in rax
  (... (compile-e e1) ;; instrs that leave e1's value in rax
        (compile-e e2) ;; instrs that leave e2's value in rax
        (compile-e e3) ;; instrs that leave e3's value in rax
        ...))
```

# Con: adding conditional expressions

## Essence of the compiler for Con

```
;; Expr Expr Expr -> Asm
(define (compile-ifzero e1 e2 e3)
  ;; Goal: instrs that leave (if (zero? e1) e2 e3) value in rax
  (seq (compile-e e1) ;; instrs that leave e1's value in rax
        (Cmp rax 0)
        (Jne 'else)
        (compile-e e2) ;; instrs that leave e2's value in rax
        (Jmp 'done)
        (Label 'else)
        (compile-e e3) ;; instrs that leave e3's value in rax
        (Jmp 'done))))
```

Let's try this out.

# Con: adding conditional expressions

Now we have a spec, implement it

```
;; Expr -> Boolean
;; Is the compiler correct on e?
(define (compiler-correct? e)
  (= (asm-interp (compile e))
     (interp e)))
```

# Recipe for growing a language

- Write examples
- Extend concrete syntax
- Extend abstract syntax
- Extend parser
- Revise interpreter to specify semantics
- Revise compiler & run-time system to implement semantics
- Test against examples
- *Rinse and repeat*

# Dupe: adding Boolean values

## Concrete Examples

```
(if (zero? 1) 2 3) ;=> 3
(if #t 1 2) ;=> 1
(if #f 1 2) ;=> 2
(if 0 1 2) ;=> ?
(zero? (if #t 1 2)) ;=> #f
(if (zero? (sub1 1)) (zero? 0) (zero? 1)) ;=> #t
```

# Dupe: adding Boolean values

## Abstract Syntax

```
;; type Expr = (Lit Datum)
;;           | (Prim1 Op1 Expr)
;;           | (If Expr Expr Expr)
```

```
;; type Datum = Integer
;;           | Boolean
```

```
;; type Op1 = 'add1 | 'sub1
;;           | 'zero?
```

```
(struct Lit (d) #:prefab)
(struct Prim1 (p e) #:prefab)
(struct If (e1 e2 e3) #:prefab)
```

# Dupe: adding Boolean values

## Abstract Examples

Concrete examples:

(if (zero? 1) 2 3)	;=> 3
(if #t 1 2)	;=> 1
(if #f 1 2)	;=> 2
(if 0 1 2)	;=> ?
(zero? (if #t 1 2))	;=> #f
(if (zero? (sub1 1)) (zero? 0) (zero? 1))	;=> #t

Abstract examples:

```
(If (Prim1 'zero? (Lit 1)) (Lit 2) (Lit 3))
(If (Lit #t) (Lit 1) (Lit 2))
(If (Lit #f) (Lit 1) (Lit 2))
(If (Lit 0) (Lit 1) (Lit 2))
(Prim1 'zero? (If (Prim1 'sub1 (Lit 1)) (Prim1 'zero? (Lit 0)) (Prim1 'zero? (Lit 1))))
```

# Dupe: adding Boolean values

## Interpreter

```
;; type Value =
;; | Integer
;; | Boolean

;; Expr -> Value
(define (interp e)
  (match e
    [(Lit d) d]
    [(Prim1 p e)
     (interp-prim1 p (interp e))])
    [(If e1 e2 e3)
     (if (interp e1)
         (interp e2)
         (interp e3)))]))

;; Op1 Value -> Value
(define (interp-prim1 op v)
  (match op
    ['add1 (add1 v)]
    ['sub1 (sub1 v)]
    ['zero? (zero? v)]))
```

# Recipe for growing a language

- Write examples
- Extend concrete syntax
- Extend abstract syntax
- Extend parser
- Revise interpreter to specify semantics
- **Revise compiler & run-time system to implement semantics**
- Test against examples
- *Rinse and repeat*

# Start with examples

What to do with new literals?

```
(compile-e (Lit 42)) ;=> (Mov rax 42)
(compile-e (Lit #f)) ;=> (Mov rax ??)
```

# Dupe: adding Boolean values

How do we make Booleans and Integers out of Integers?



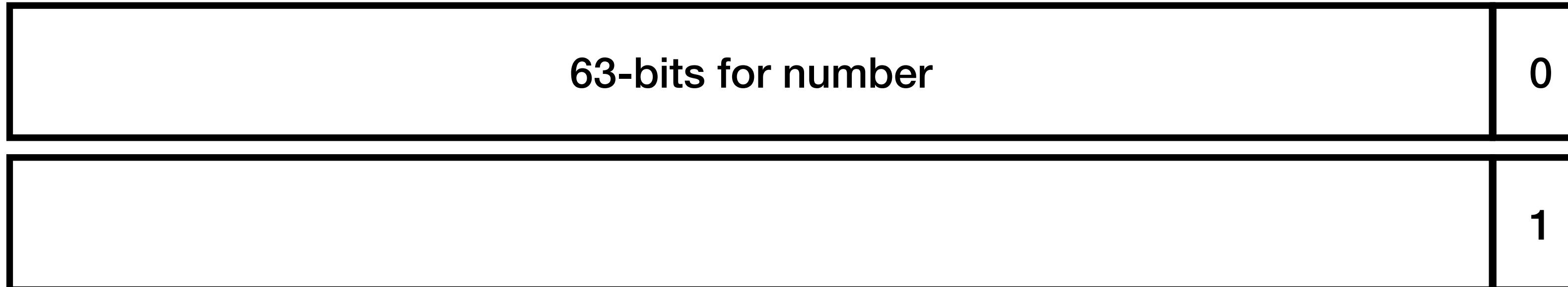
```
#lang racket  
#t
```

compiles to

(Mov rax ???)

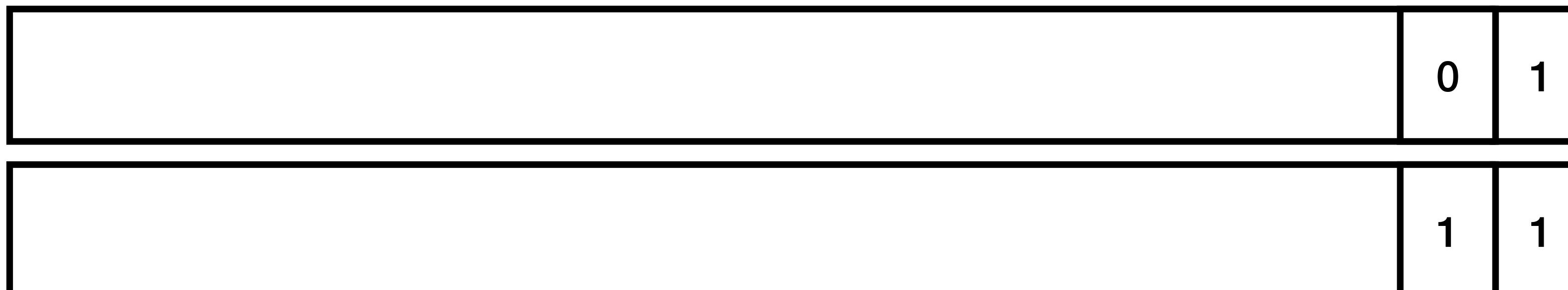
# Encoding values in Dupe

Type tag in least significant bits



Integers

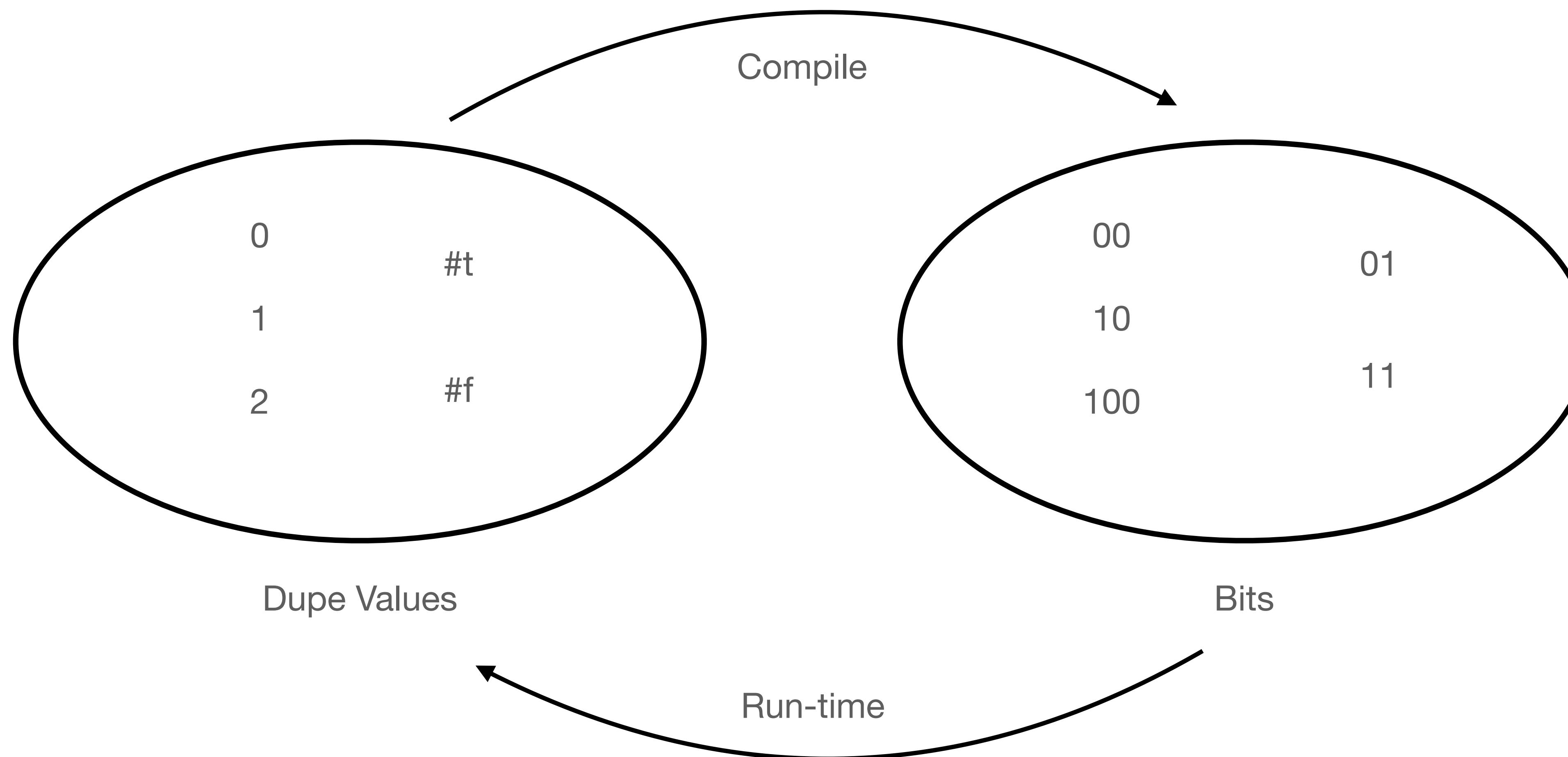
Booleans



#t

#f

# Representing Values with Bits in Dupe

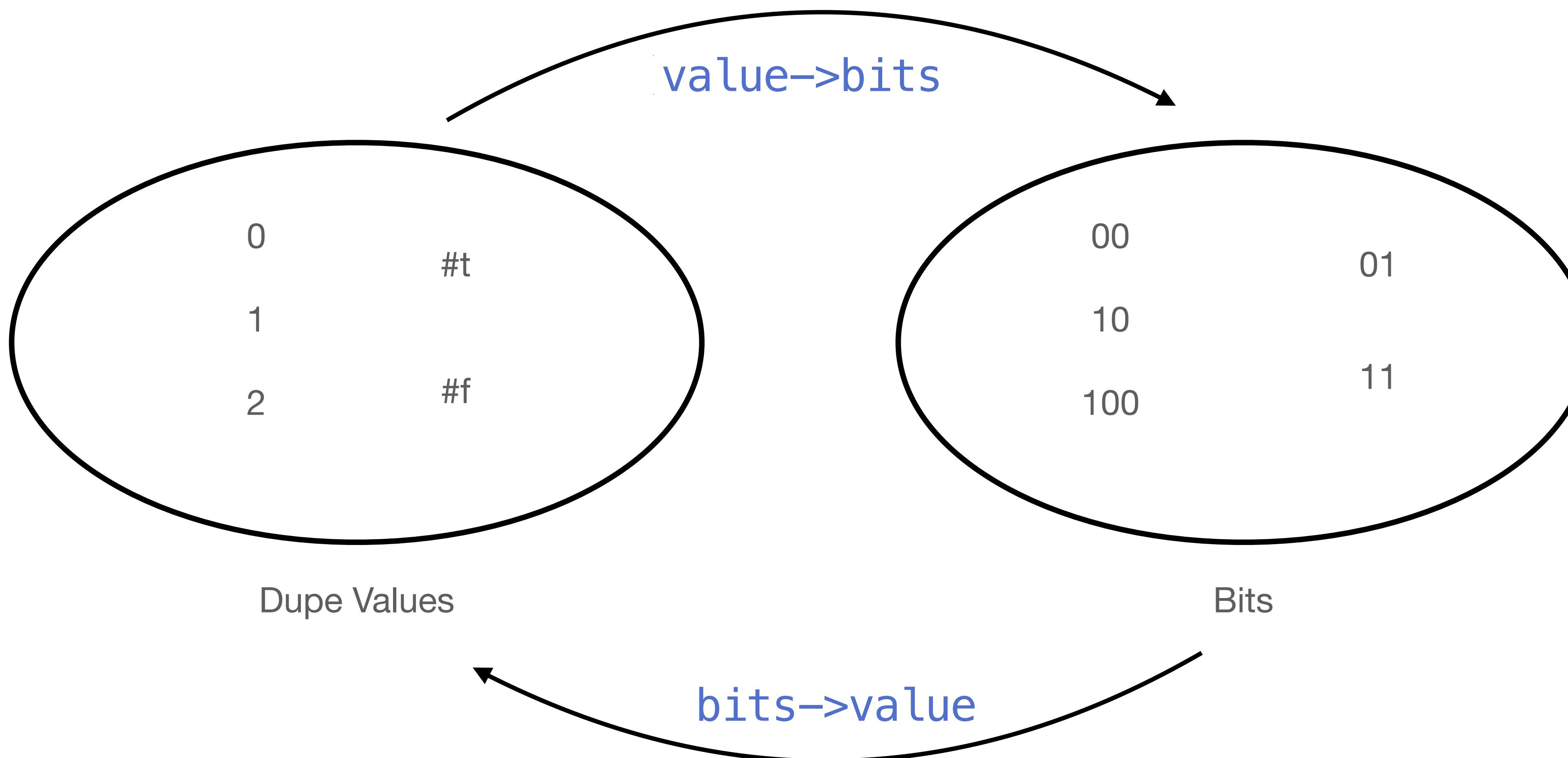


# Dupe: adding Boolean values

```
;; Expr -> Boolean
;; Is the compiler correct on e?
(define (compiler-correct? e)
  (= (asm-interp (compile e))
     (interp e)))
```



# Functions for mapping between worlds



# Dupe: adding Boolean values

```
;; Expr -> Boolean
;; Is the compiler correct on e?
(define (compiler-correct? e)
  (equal? (bits->value (asm-interp (compile e)))
          (interp e)))
```



# Dupe: adding Boolean values

```
;; Expr -> Boolean
;; Is the compiler correct on e?
(define (compiler-correct? e)
  (equal? (bits->value (asm-interp (compile e)))
          (interp e)))

(compiler-correct? (parse `(add1 #f)))
```



# CMSC 430 - 23 September 2025

## Characters and I/O

### Today

- Dodger: adding a bit of character
- Evildoer: I/O, run-time calls

### Announcements

- Assignment 3, due 9/25; **not collaborative**
- Assignment 4, due 10/2; **not collaborative**

# Hustle: adding pairs and boxes

Concrete examples:

' ()	;=> ' ()
(cons 1 2)	;=> ' (1 . 2)
(box 8)	;=> #&8
(car (cons 1 2))	;=> 1
(cdr (cons 1 2))	;=> 2
(unbox (box 8))	;=> 8
(cons? (cons 1 2))	;=> #t
(box? (box 8))	;=> #t

# Syntactic additions in Dodger

## Character literals and operations

Concrete syntax

#\c

(char->integer e)

(integer->char e)

(char? e)

Abstract syntax

(Lit #\c)

(Prim1 'char->integer e)

(Prim1 'integer->char e)

(Prim1 'char? e)

# Semantic additions in Dodger

```
;; type Value =
;; | Integer
;; | Boolean
;; | Character
```

```
Welcome to DrRacket, version 8.14 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> #\a
#\a
> #\b
#\b
> (char? #\a)
#t
> (char? #t)
#f
> (char->integer #\a)
97
> (integer->char 97)
#\a
```

# Encoding values in Dodger

## Type tag in least significant bits



Integers



Booleans



Characters



#t



#f

# Compiling Dodger

```
> (compile-e (Lit #\a))
```

# Compiling Dodger

```
; Value -> Integer
(define (value->bits v)
  (cond [(char? v)
          (bitwise-ior #b01 (arithmetic-shift (char->integer v) 2))]
        [#;...]))
```

```
> (value->bits #\a)
389
> (+ (arithmetic-shift 97 2) #b01)
389
```

# Dodger run-time

**Every change to Values requires a change in run-time**

```
void print_result(val_t x)
{
    switch (val_typeof(x)) {
        case T_INT:
            printf("%" PRId64, val_unwrap_int(x));
            break;
        case T_BOOL:
            printf(val_unwrap_bool(x) ? "#t" : "#f");
            break;
        case T_CHAR:
            print_char(val_unwrap_char(x));
            break;
        case T_INVALID:
            printf("internal error");
    }
}
```

print\_char: uses UTF-8 encoding of Unicode  
Details are not important, code is given to you

# Syntactic additions in Evildoer

## Concrete syntax

(begin e1 e2)

(read-byte)

(peek-byte)

(void)

(write-byte e)

(eof-object? e)

## Abstract syntax

(Begin e1 e2)

(Prim0 'read-byte)

(Prim0 'peek-byte)

(Prim0 'void)

(Prim1 'write-byte e)

(Prim1 'eof-object? e)

# Semantic additions in Evildoer

```
;; type Value =
;; | Integer
;; | Boolean
;; | Character
;; | Eof
;; | Void
```

```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (read-byte)
a
97
> (read-byte)
10
> (read-byte)
#<eof>
> (void)
> (cons (void) (void))
'(#<void> . #<void>)
> (begin 1 2)
2
> (write-byte 97)
a
> (begin (write-byte 97)
          (write-byte 98))
ab
>
```

# Encoding values so far in Evildoer

## Type tag in least significant bits

63-bits for number	0	Integers
62-bits for code point (only need 21)	0 1	Characters
	0 1 1	#t
	1 1 1	#f
	1 0 1 1	eof
	1 1 1 1	void

# I/O support in Run-time

## io.c

```
#include "types.h"
#include "values.h"
#include "runtime.h"

val_t read_byte(void)
{
    char c = getc(in);
    return (c == EOF) ? val_wrap_eof() : val_wrap_byte(c);
}

val_t peek_byte(void)
{
    char c = getc(in);
    ungetc(c, in);
    return (c == EOF) ? val_wrap_eof() : val_wrap_byte(c);
}

val_t write_byte(val_t c)
{
    putc((char) val_unwrap_int(c), out);
    return val_wrap_void();
}
```

# Calling the Run-time

Compiler generates code to call RT:

- has to obey ABI (Application Binary Interface) that C compiler uses
- ABI specifies:
  - return value in rax
  - first parameter in rdi

# Evildoer run-time

```
void print_result(val_t x)
{
    switch (val_typeof(x)) {
        case T_INT:
            printf("%" PRId64, val_unwrap_int(x));
            break;
        case T_BOOL:
            printf(val_unwrap_bool(x) ? "#t" : "#f");
            break;
        case T_CHAR:
            print_char(val_unwrap_char(x));
            break;
        case T_EOF:
            printf("#<eof>");
            break;
        case T_VOID:
            break;
        case T_INVALID:
            printf("internal error");
    }
}
```

# Stack alignment

According to System V ABI:

The stack pointer must be aligned to 16-bytes when calling a function.

- what does “aligned” mean?
- why is this relevant now?
- how did we solve the problem?

# Nobody noticed

```
;; Expr -> Asm
(define (compile e)
  (prog (Global 'entry)
        (Extern 'peek_byte)
        (Extern 'read_byte)
        (Extern 'write_byte)
        (Label 'entry)
        (Sub rsp 8)
        (compile-e e)
        (Add rsp 8)
        (Ret)))
```

- By ABI, stack aligned when RT called entry
- Call pushes return pointer on stack (64 bits)
- Stack is therefore unaligned at entry

# CMSC 430 - 25 September 2025

## Testing I/O, Defining Errors

### Today

- Quick Assign 4 demo
- Evildoer: how to test programs that do I/O
- Extort: specifying errors
- Addressing a long-standing bug in our compilers

### Announcements

- Video from Tuesday now posted
- Assignment 3, due tonight!
- Assignment 4, due 10/2, includes Assignment 3

# How to test programs that do I/O?

```
(check-expect (run ' (read-byte) ) ???)
```

Observe:

- What to expect depends on the state of the input port
- Value is no longer a function of the expression alone

# Interpreting I/O operations just does I/O

```
(interp (parse '(read-byte))      ;; waits for input  
(interp (parse '(write-byte 97)) ;; writes 'a'
```

We can use Racket facilities to redirect I/O to make functional tests:

```
(interp/io (parse '(read-byte)) "a") => '(97 . "")  
(interp/io (parse '(write-byte 97)) "")  
=> '#<void> . "a")
```

# Interpreting I/O operations just does I/O

We can use Racket facilities to redirect I/O to make functional tests:

```
(interp/io (parse '(read-byte)) "a") => '(97 . "")  
(interp/io (parse '(write-byte 97)) "")  
=> '#<void> . "a")
```

Observe:

- State of input port given explicitly
- State of output port produced explicitly
- Behavior of program is a (mathematical) function once again

# Executing compiled I/O operations

We can use run/io (compiles and sets up run-time appropriately):

```
(run/io (compile (parse '(read-byte)) "a")) => '(97 . "")  
(run/io (compile (parse '(write-byte 97)) ""))  
=> '#<void> . "a")
```

# Refining correctness

Need to take I/O into account

```
;; Expr String -> Boolean
(define (correct? e in)
  (let ((r (with-handlers ([exn:fail? identity])
              (interp/io e in))))
    (and (not (exn? r))
         (equal? r (exec/io e in)))))
```

Observe:

- If interpreter errors, behavior is unspecified
- Program is interpreted and executed with same input
- Checks that both value and output match

# Refining correctness

Need to take I/O into account

```
;; Expr String -> Boolean
(define (correct? e in)
  (let ((r (with-handlers ([exn:fail? identity])
             (interp/io e in))))
    (and (not (exn? r))
         (equal? r (exec/io e in)))))
```

A counter-example is an expression *and* input that produce different results or output

# run.rkt facility for setting up run-time in asm-interp

```
Welcome to DrRacket, version 8.14 [cs].  
Language: racket, with debugging; memory limit: 128 MB.  
> (require "run.rkt")  
> (run (compile (Lit 97)))  
97  
> (run/io (compile (Prim1 'write-byte (Lit 97))) "")  
'(#<void> . "a")  
> (run/io (compile (Prim0 'read-byte)) "a")  
'(97 . "")
```

# The Phase Distinction comes into Focus

## Compile-time vs run-time illuminated with I/O

With the introduction of effects, *when* things happen matter:

```
(compile (parse '(write-byte 97))) ; doesn't write
(compile (parse '(loop)))           ; doesn't loop
(compile (parse '(/ 1 0)))          ; doesn't crash
(compile (parse '(launch-nukes)))   ; doesn't kill us
```

# Extort: Specifying Errors

Undefined behavior considered harmful

Let's revise interp to specify results for programs that error.

- Syntax doesn't change
- Semantic result extended to include errors (distinct from values)
- If interp is a *total function*, then no undefined behavior

```
;; type Error = ...
;; type Answer = Value | Error
```

```
;; Expr -> Answer
(define (interp e) ...)
```

# Extort: Specifying Errors

Undefined behavior considered harmful

We'll use 'err to represent errors and use exceptions to signal them.

```
;; Expr -> Answer
(define (interp e)
  (with-handlers ([(λ (x) (eq? x 'err)) identity])
    (interp-e e)))
```

interp-e is the main helper function that does recursive evaluation

# Extort: Specifying Errors

Undefined behavior considered harmful

We'll use 'err to represent errors and use exceptions to signal them.

```
;; Expr -> Value { raises 'err }
(define (interp-e e)
  (match e
    [((Lit d) d)
     ((Eof) eof)
     [((Prim0 p)
       (interp-prim0 p))]
     [((Prim1 p e)
       (interp-prim1 p (interp-e e)))]
     [((If e1 e2 e3)
       (if (interp-e e1)
           (interp-e e2)
           (interp-e e3)))]
     [((Begin e1 e2)
       (begin (interp-e e1)
              (interp-e e2))))])
```

# Extort: Specifying Errors

Undefined behavior considered harmful

Errors arise from primitives that are applied to arguments not in their domain:

```
;; Op1 Value -> Value { raises 'err }
(define (interp-prim1 op v)
  (match (list op v)
    [((list 'add1 (? integer?)) )           (add1 v)]
    [((list 'sub1 (? integer?)) )           (sub1 v)]
    [((list 'zero? (? integer?)) )          (zero? v)]
    [((list 'char? v) )                   (char? v)]
    [((list 'integer->char (? codepoint?)) ) (integer->char v)]
    [((list 'char->integer (? char?)) )     (char->integer v)]
    [((list 'write-byte (? byte?)) )        (write-byte v)]
    [((list 'eof-object? v) )               (eof-object? v)]
    [_ (raise 'err)])
```

# Extort: Compiling with Errors

Undefined behavior considered harmful

Specifying errors ties the compiler's hands:

```
;; Expr String -> Boolean
(define (correct? e in)
  (equal? (interp/io e in)
          (exec/io e in)))
```

# Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

```
> (compile-op1 'add1)
(list
  (Push 'rax)
  (And 'rax 1)
  (Cmp 'rax 0)
  (Pop 'rax)
  (Jne 'err)           Jumps to place that signals an error if the thing in rax is not an integer
  (Add 'rax 2)))
```

# Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

```
> (exec (parse `(add1 #f)))  
'err
```

# Extort: Compiling with Errors

Undefined behavior considered harmful

Don't need update encodings of values (errors are not values)

Do need to add a function to be called when an error happens:

```
;; Expr -> Asm
(define (compile e)
  (prog (Global 'entry)
        (Extern 'peek_byte)
        (Extern 'read_byte)
        (Extern 'write_byte)
        (Extern 'raise_error)
        (Label 'entry)
        (Sub rsp 8)
        (compile-e e)
        (Add rsp 8)
        (Ret))
  ;; Error handler
  (Label 'err)
  (Call 'raise_error)))
```

```
void raise_error()
{
    printf("'err\n");
    exit(1);
}
```

# **CMSC 430 - 30 September 2025**

## **Variables, Binding, Binary Operations**

### **Today**

- Variables, environments
- Lexical addressing
- Compile-time vs run-time environments

### **Announcements**

- Assignment 3 grades posted
- Assignment 4, due 10/2, includes Assignment 3

# Fraud: adding variable bindings

## Concrete examples

```
(let ((x 0)) 1) ;=> 1
(let ((x 0)) x) ;=> 0
(let ((x 1)) (+ x x)) ;=> 2
(let ((x 1))
  (let ((y 2))
    (+ x y))) ;=> 3
(let ((x 1))
  (let ((x 2))
    (+ x x))) ;=> 4
```

# Fraud: adding variable bindings

## Abstract examples

```
(Let 'x (Lit 0) (Lit 1))  
(Let 'x (Lit 0) (Var 'x))  
(Let 'x (Lit 1) (Prim2 '+ (Var 'x) (Var 'x)))  
(Let 'x (Lit 1)  
      (Let 'y (Lit 2)  
            (Prim2 '+ (Var 'x) (Var 'y)))))  
(Let 'x (Lit 1)  
      (Let 'x (Lit 2)  
            (Prim2 '+ (Var 'x) (Var 'x)))))
```

# Fraud: adding variable bindings

## Updated semantics

```
;; Expr -> Answer
(define (interp e)
  (match e
    ;; ...
    [(Var x) ...]
    [(Let x e1 e2)
     (... (interp e1)
          (interp e2) ...))]))
```



# What do variables mean?

It varies

```
(let ((x 42))  
  (... x ...))
```

```
(let ((x 10))  
  (... x ...))
```

```
(interp (Var 'x)) ;=> 42  
(interp (Var 'x)) ;=> 10
```

Not a function!



# Need for an environment

The meaning of a variable depends on the environment in which it occurs

- It's not enough to specify the meaning of an expression in isolation
- Need to know the meaning of variables that occur in that expression
- Meaning of an expression is a function of:
  - the expression
  - the meaning of its (free) variables

# What do variables mean?

It varies

```
(let ((x 42))  
  (... x ...))
```

```
(let ((x 10))  
  (... x ...))
```

A function!

```
(interp (Var 'x) { x means 42 }) ;=> 42  
(interp (Var 'x) { x means 10 }) ;=> 10
```



# Representing the environment

Using an association list

```
{ x means 42           ' ((x 42)
  y means 12           (y 12)
  z means #f }         (z #f))
```

```
;; type Env = (Listof (List Id Value))
```

# Operations on the environment

## Using an association list

```
;; Lookup up x's value in r
;; Env Id -> Value
(define (lookup r x) ...)

;; Extend r so that x means v
;; Env Id Value -> Env
(define (ext r x v) ...)
```

;; Fundamental law:  
;;  $\forall r \ x \ v,$   
 $(\text{lookup} (\text{ext} \ r \ x \ v) \ x) \equiv v$

# Interpreting variables and bindings

## Environments tell us the meaning of variables

The interpreter uses an environment to manage associations between variables and their values.

```
;; ClosedExpr -> Answer
(define (interp e)
  (with-handlers ([err? identity])
    (interp-e e '())))
```

```
;; type Env = (Listof (List Id Value))
;; Expr Env -> Value { raises 'err }
(define (interp-e e r) ;;; where r closes e
  (match e
    [(Var x) (lookup r x)]
    [(Let x e1 e2)
     (let ((v (interp-e e1 r)))
       (interp-e e2 (ext r x v)))]
    #;...))
```

# Compiling variables and bindings

Using the interpreter the guide us

The interpreter uses an environment to manage associations between variables and their values.

Appears we need run-time representation of environments, identifier names, and implementation of lookup and ext in assembly. Seems... hard.

```
;; type Env = (Listof (List Id Value))

;; Expr Env -> Value { raises 'err }
(define (interp-e e r) ;; where r closes e
  (match e
    [(Var x) (lookup r x)]
    [(Let x e1 e2)
     (let ((v (interp-e e1 r)))
       (interp-e e2 (ext r x v)))]
    #;....))
```

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (let ((z ...))
      ; ; what do you know about the
      ; ; environment used to evaluate e?
      e)))
```

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (+ (let ((z ...))
         ;; what do you know about the
         ;; environment used to evaluate e1?
         e1)
        ;; what about e2?
        e2)))
```

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))  
  (let ((y ...))  
    (let ((z ...))  
      ;; where will y's binding  
      ;; be in the environment?  
      y)))
```

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))  
  (let ((y ...))  
    (let ((z ...))  
      ;; where will z's binding  
      ;; be in the environment?  
      z)))
```

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))  
  (let ((y ...))  
    (let ((z ...))  
      ;; where will z's binding  
      ;; be in the environment?  
      x)))
```

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
( let ((x ...))  
  (let ((y ...))  
    (+ (let ((z ...))  
        ...)  
    ;; where will y's binding  
    ;; be in the environment  
    y)))
```

# Observing an invariant about bindings

## Using the interpreter the guide us

Suppose we get to this point in interpreting a program:

```
(interp-e (Var 'x) '(((y 1) (x 99) (p 7)))
```

What can you say about the program surrounding this occurrence of `x`?

```
(let ((p ...))  
  ...  
  (let ((x ...))  
    ...  
    (let ((y ...))  
      ... x ...)))
```

It has to have looked like this!

# Observing an invariant about bindings

Using the interpreter the guide us

Suppose we know the program looks like this:

```
(let ((p ...))  
    ...  
    (let ((x ...))  
        ...  
        (let ((y ...))  
            ... x ...)))
```

What can you say about the environment that will be used when `x` is interpreted?

' ((y ??) (x ??) (p ??))

It must look like this!

But now we can see that lookup will retrieve the second element;  
The location of a variables binding in the environment is a static property

# Generalizing the observation

The text of the program tells you a lot about the structure of the environment

For each variable occurrence, we can precisely calculate the location in the environment *before interpreting the program.*

Lookup doesn't need to be a linear search, we can compute the index of the value in the list.

# Variable names are irrelevant

## Writing an interpreter that uses lexical addresses

```
(let ((p ...))  
  ...  
  (let ((x ...))  
    ...  
    (let ((y ...))  
      ... x ...))))
```

```
(let ((_ ...))  
  ...  
  (let ((_ ...))  
    ...  
    (let ((_ ...))  
      ... (Var 1) ...))))
```

(Var 1) means:  
“there’s one let between this occurrence and its binder”

Environment can change from [Listof (List Id Value)] to [Listof Value]  
lookup for (Var i) becomes (list-ref r i).  
ext becomes cons.

# **CMSC 430 - 7 October 2025**

## **Binary operations; stack alignment revisited**

### **Today**

- Quick review of let bindings, compile-time and run-time environments
- Binary operations
- Fixing the stack-alignment problem

### **Announcements**

- Understanding Fraud quiz; due by start of class Thursday
- Assignment 4 grades posted
- Assignment 5 out today; due Oct 23
- Midterm 1, Oct 16

# Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
  (let ((y ...))
    (+ (let ((z ...))
         ;; what do you know about the
         ;; environment used to evaluate e1?
         e1)
        ;; what about e2?
        e2)))
```

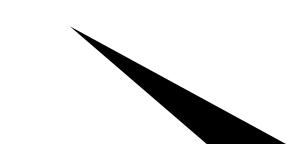
# Compile-time environments

## compile-e now takes an environment

```
;; type CEnv = (Listof Id)
```

```
;; Expr CEnv -> Asm
```

```
(define (compile-e e c) ...)
```



Describes the binding context of e

```
(compile-e (Var 'x) '(x y z)) ;=> (Mov rax (Offset rsp 0))  
(compile-e (Var 'y) '(x y z)) ;=> (Mov rax (Offset rsp 8))  
(compile-e (Var 'z) '(x y z)) ;=> (Mov rax (Offset rsp 16))
```

# Fraud (2): adding binary operations

## Updated semantics

```
;; Op2 Value Value -> Answer
(define (interp-prim2 op v1 v2)
  (match (list op v1 v2)
    [([list '+ (? integer?) (? integer?)) (+ v1 v2)]
     [([list '- (? integer?) (? integer?)) (- v1 v2)]
      [([list '< (? integer?) (? integer?)) (< v1 v2)]
       [([list '> (? integer?) (? integer?)) (> v1 v2)]
        [_ 'err])])
```



# Fraud (2): adding binary operations

## Updated compiler

```
; ; Op2 Expr Expr CEnv -> Asm
(define (compile-prim2 p e1 e2 c)
  (seq (compile-e e1 c) ; e1's value in rax
        ...
        (compile-e e2 c) ; e2's value in rax
        ...
        (compile-op2 p)))
```



# Fraud (2): adding binary operations

## Updated compiler

```
;; Op2 Expr Expr CEnv -> Asm
(define (compile-prim2 p e1 e2 c)
  (seq (compile-e e1 c) ; e1's value in rax
        (Push rax)         ; e1's value on stack
        (compile-e e2 c) ; e2's value in rax
        (compile-op2 p))) ; have op pop
```



# Fraud (2): adding binary operations

## Updated compiler

```
;; Op2 -> Asm
(define (compile-op2 p)
  (match p
    ['+
     (seq (Pop r8)
          (assert-integer r8)
          (assert-integer rax)
          (Add rax r8))]

    ['-
     (seq (Pop r8)
          (assert-integer r8)
          (assert-integer rax)
          (Sub r8 rax)
          (Mov rax r8))]

    #; ...))
```



# Fraud (2): adding binary operations

Updated compiler

```
> (exec  
  (parse `(let ((x 1)  
             (+ 5 x)))) )
```

10



# Invariants (Fraud)

## Various facts about the Fraud compiler

Registers:

`rax` - return value

`rsp` - stack pointer

`rdi` - first param when calling run-time system

Stack is 8-byte (64-bit) aligned,  
i.e. divisible by 8,  
i.e. ends in `#b000`  
(Must align to 16-bytes to call)

`(compile-e e c)` - leaves stack initial state

Length of compile time environment =  
Number of elements on stack at RT

# Stack-alignment in Fraud

## Always 8-byte, sometimes 16-byte aligned

Stack is 8-byte aligned,  
i.e. divisible by 8,  
i.e. ends in #b000

Must align to 16-bytes to call,  
i.e. divisible by 16,  
i.e. ends in #b0000

```
Mov r15 rsp  
And r15 #b1000  
Sub rsp r15  
Call f  
Add rsp r15
```

r15 is a “callee-saved” or  
“non-volatile” register

The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile (caller-saved).  
The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile (callee-saved).

r15 is 0 when rsp ends in #b0000  
r15 is 8 when rsp ends in #b1000

# CMSC 430 - 9 October 2025

## Inductive data

### Today

- Quick overview of Midterm 1 process
- Adding inductively defined data

### Announcements

- Assignment 5 parser fixed
- No class Tuesday, Oct 14
- Midterm 1, Thursday, Oct 16

# Recipe for growing a language

- Write examples
- Extend concrete syntax
- Extend abstract syntax
- Extend parser
- Revise interpreter to specify semantics
- Revise compiler & run-time system to implement semantics
- Test against examples

# How to represent pointers?

## Addressing memory

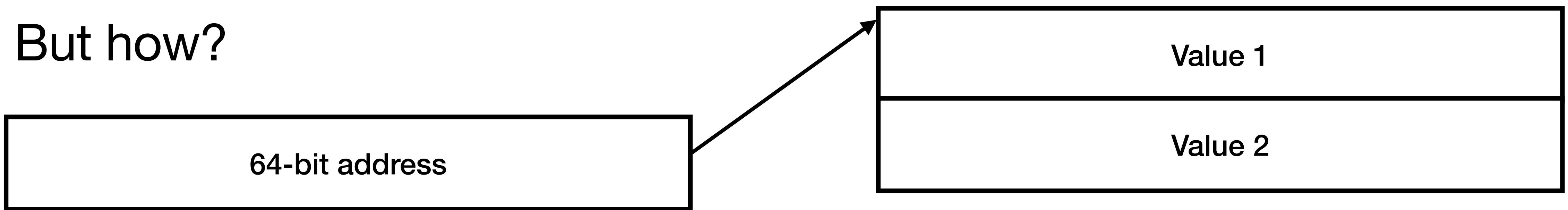
Basic idea:

A pair is allocated as two words in memory

The pair *value* will be represented by the address

+ something indicating the value is a pair

But how?



Hint: we'll always allocate memory in multiples of 8-bytes (64-bits)

# Invariants (Hustle)

## Various facts about the Hustle compiler

Registers:

rax - return value

rsp - stack pointer

rdi - first param when calling run-time system

rbx - heap pointer

(compile-e e c)

- leaves stack in initial state

Length of compile time environment =  
Number of elements on stack at RT

Stack is 8-byte (64-bit) aligned,  
i.e. divisible by 8,  
i.e. ends in #b000  
(Must align to 16-bytes to call)

Heap is 8-byte (64-bit) aligned,  
i.e. divisible by 8,  
i.e. ends in #b000

↑  
Key to our tagging scheme for pointer types

# Encoding pointer values (Hustle)

Type tag in least significant bits



0	0	1
---	---	---

Box



0	1	0
---	---	---

Cons

# **CMSC 430 - 21 October 2025**

## **More heap-allocated data**

### **Today**

- Review of pairs and boxes
- Array data: vectors, strings

### **Announcements**

- Midterm 1 will be graded by Thursday
- Assignment 6 out tonight
- Notes revised for Hustle and Hoax, posted soon

# Encoding immediate values (Hustle)

Type tag in least significant bits

60-bits for number	0	0 0 0	Integers
59-bits for code point (only need 21)	0 1	0 0 0	Characters
	0 1 1	0 0 0	#t
	1 1 1	0 0 0	#f
	1 0 1 1	0 0 0	eof
	1 1 1 1	0 0 0	void
Immediate tag			

# Encoding pointer values (Hustle)

Type tag in least significant bits



0	0	1
---	---	---

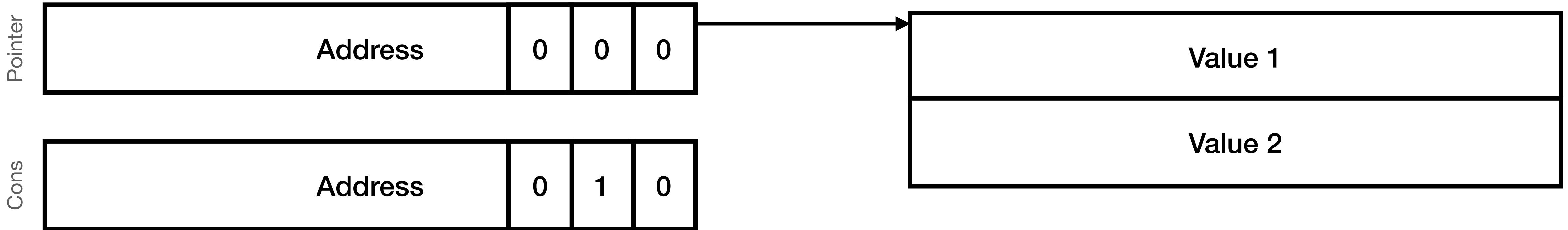
Box



0	1	0
---	---	---

Cons

# Values that point to memory



Idea:

A pair is allocated as two words in memory

The pair *value* will be represented by the address + type tag in 3 least significant bits

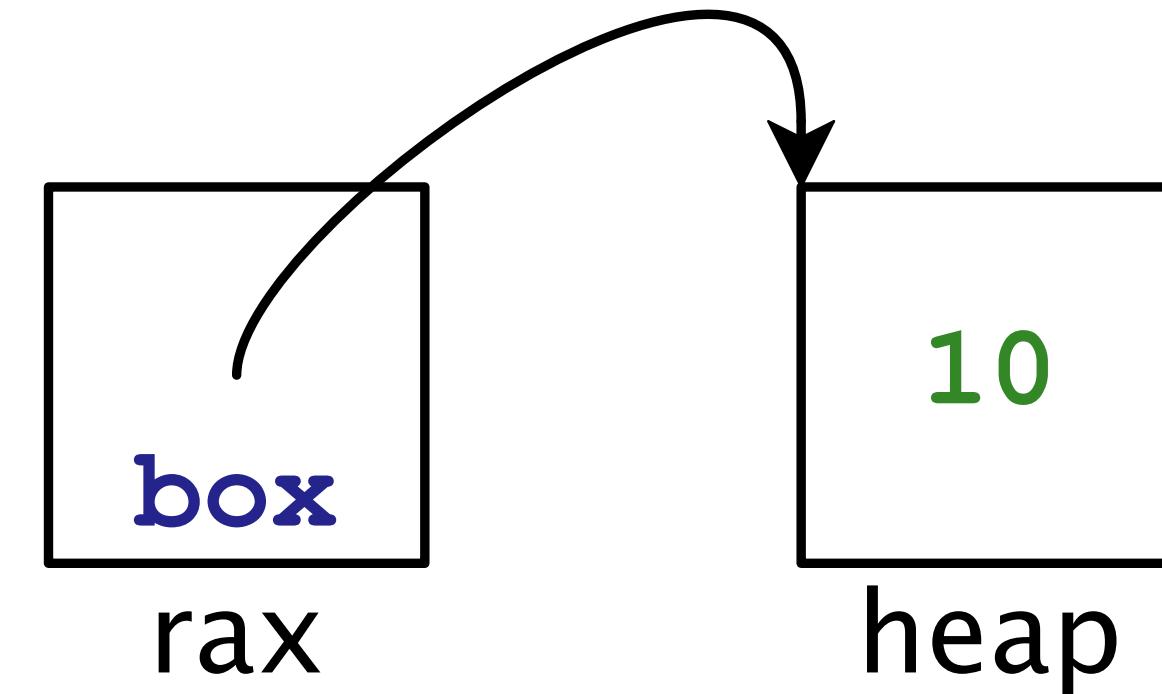
Pointers end in #b000 because we allocate in multiples of 8 bytes: advantage of this and stash type tag there (no shifting).

**Observe: pointers don't have a type, values do!**

# Creating a box

A pair is a tagged pointer to one value in memory

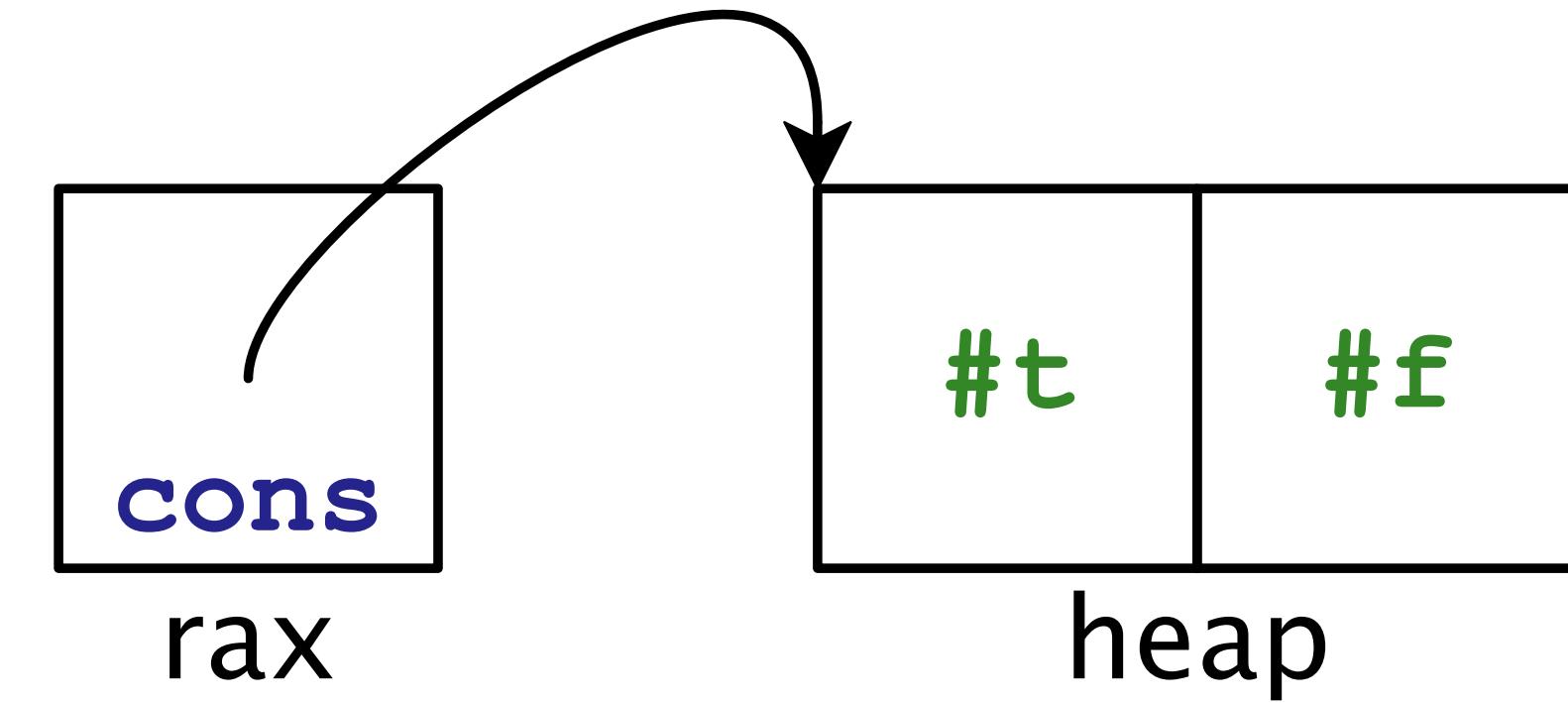
```
> (compile-op1 'box)
(list
  (Mov (Mem 'rbx) 'rax)
  (Mov 'rax 'rbx)
  (Xor 'rax 1)
  (Add 'rbx 8))
> (exec (parse '(box 10)))
'#&10
```



# Creating a pair

A box is a tagged pointer to two values in memory

```
> (compile-op2 'cons)
(list
  (Mov (Mem 'rbx 8) 'rax)
  (Pop 'rax)
  (Mov (Mem 'rbx 0) 'rax)
  (Mov 'rax 'rbx)
  (Xor 'rax 2)
  (Add 'rbx 16))
> (exec (parse `(cons #t #f)))
'(#t . #f)
```



# Hoax: Adding vectors

## Concrete examples

```
(make-vector 3 #t) ;=> '#(#t #t #t)
(let ((v (make-vector 3 #t)))
  (vector-ref v 0)) ;=> #t
(let ((v (make-vector 3 #t)))
  (vector-ref v 3)) ;=> err
(let ((v (make-vector 3 #t)))
  (vector-length v)) ;=> 3
(let ((v (make-vector 3 #t)))
  (begin (vector-set! v 0 #f)
         v)) ;=> '#(#f #t #t)
```

# Hoax: Adding vectors

## AST additions

```
;; type Op1 = ...
;;           | 'vector? | 'vector-length
;; type Op2 = ...
;;           | 'make-vector | 'vector-ref
;; type Op3 = 'vector-set!
```

# Hoax: Adding vectors

## Semantic additions

```
;; type Value = ...
;; | (vector Value ...)
```

# Hoax: Adding vectors

## Semantic additions

```
;; Op2 Value Value -> Value { raises 'err }
(define (interp-prim2 op v1 v2)
  (match (list op v1 v2)
    #; ...
    [(list 'make-vector (? integer?) _)
     (if (<= 0 v1)
         (make-vector v1 v2)
         (raise 'err))]
    [(list 'vector-ref (? vector?) (? integer?))
     (if (<= 0 v2 (sub1 (vector-length v1)))
         (vector-ref v1 v2)
         (raise 'err)))]))
```

# Hoax: Adding vectors

## Semantic additions

```
;; Op3 Value Value Value -> Value { raises 'err }
(define (interp-prim3 p v1 v2 v3)
  (match (list p v1 v2 v3)
    [(list 'vector-set! (? vector?) (? integer?) _)
     (if (<= 0 v2 (sub1 (vector-length v1)))
         (vector-set! v1 v2 v3)
         (raise 'err))]
    [_ (raise 'err)]))
```

# CMSC 430 - 23 October 2025

## Vectors and strings

### Today

- Array data:
  - **vectors:** make-vector, vector-length, vector-ref, vector-set!
  - **strings:** make-string, string-length, string-ref, **string literals**
- Static data

### Announcements

- Assignment 6 out tonight (for real)
- Notes revised for Hustle and Hoax, posted soon (for real)
- Midterm 1 grades tomorrow

# Encoding pointer values (Hoax)

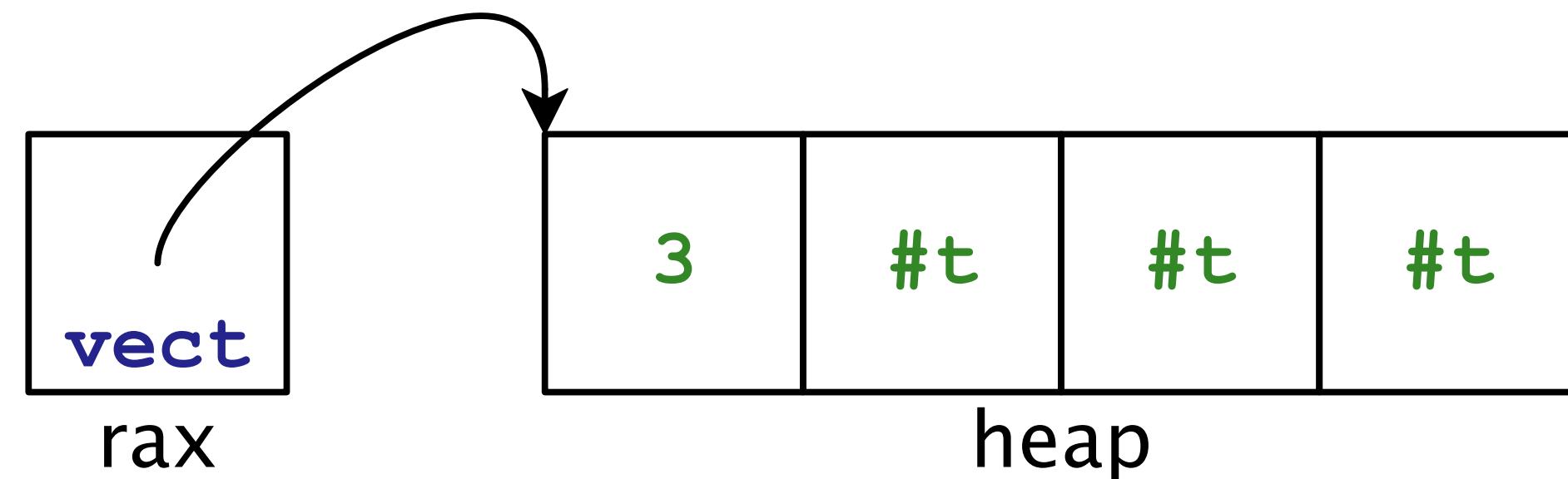
Type tag in least significant bits

61-bits for address	<table border="1"><tr><td>0</td><td>0</td><td>1</td></tr></table>	0	0	1	Box
0	0	1			
61-bits for address	<table border="1"><tr><td>0</td><td>1</td><td>0</td></tr></table>	0	1	0	Cons
0	1	0			
61-bits for address	<table border="1"><tr><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	Vector
0	1	1			
61-bits for address	<table border="1"><tr><td>1</td><td>0</td><td>0</td></tr></table>	1	0	0	String
1	0	0			

# Creating a vector

A vector is a tagged pointer to a sized array of values in memory

```
> (exec (parse '(make-vector 3 #t)))  
'#(#t #t #t)
```



# CMSC 430 - 28 October 2025

## Strings and functions

### Today

- Hints on assignment 6
- Finish: make-vector
- Strings, quickly: make-string, string-length, string-ref, string literals
- Iniquity: function definitions and calls

### Announcements

- Midterm 1 grades out
- Quiz due by start of class Thursday

# Hints for Assignment 6

## Some thoughts on the current assignment

- Determining if something is a list requires traversing it
- Computing the length of a list requires traversing it
- Temporal and spatial ordering of memory reads/writes can be arbitrary
  - You can write the elements starting from the end
  - You can write the length last, etc.
- Working examples out on paper will get you there faster
  - Draw the memory you expect
- Using static data to make examples that you interactively develop can help
- All of the primitives can be implemented in a single pass over input and without allocating temporary memory (i.e. only allocating the output value)
- Reference solution < 90 LOC
- Programming in assembly sucks, someone should develop a high-level... oh

# Revised make-vector

Assume length argument is non-zero, for now

```
(seq (Pop r8)
     (assert-natural r8)
     (Mov (Mem rbx 0) r8) ; write length
     (Sar r8 1)           ; convert to bytes
     (Mov r9 r8)          ; save for heap adjustment

     ; start initialization
     (Label loop)
     (Mov (Mem rbx r8) rax)
     (Sub r8 8)
     (Cmp r8 0)
     (Jne loop)
     ; end initialization

     (Mov rax rbx)
     (Xor rax type-vect) ; create tagged pointer
     (Add rbx r9)        ; acct for elements
     (Add rbx 8))         ; and stored length
```

# Revised make-vector

## Remove assumption, handle length 0 specially

```
(seq (Pop r8)
     (assert-natural r8)

     ; special case for length = 0
     (Cmp r8 0)
     (Jne nonzero) ; need to allocate and initialize
     (Lea rax (Mem 'empty type-vect)) ; return canonical empty vector
     (Jmp theend)

     (Label nonzero)
     (Mov (Mem rbx 0) r8) ; write length
     (Sar r8 1)           ; convert to bytes
     (Mov r9 r8)          ; save for heap adjustment

     ; start initialization
     (Label loop)
     (Mov (Mem rbx r8) rax)
     (Sub r8 8)
     (Cmp r8 0)
     (Jne loop)
     ; end initialization

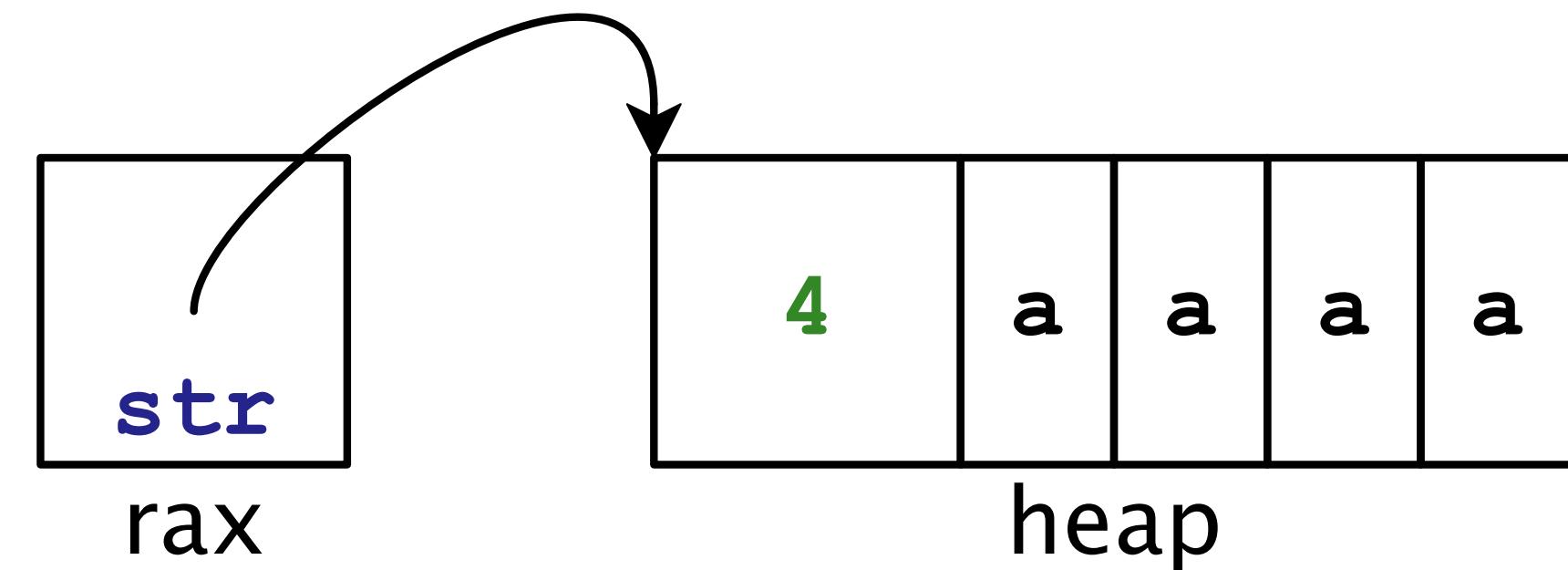
     (Mov rax rbx)
     (Xor rax type-vect) ; create tagged pointer
     (Add rbx r9)        ; acct for elements
     (Add rbx 8)          ; and stored length
     (Label theend))))]
```

;; ClosedExpr → Asm  
(define (compile e)  
 (prog ...  
 (Data)  
 (Label 'empty)  
 (Dq 0)))

# Creating a string

A string is a tagged pointer to a sized array of *codepoints* in memory

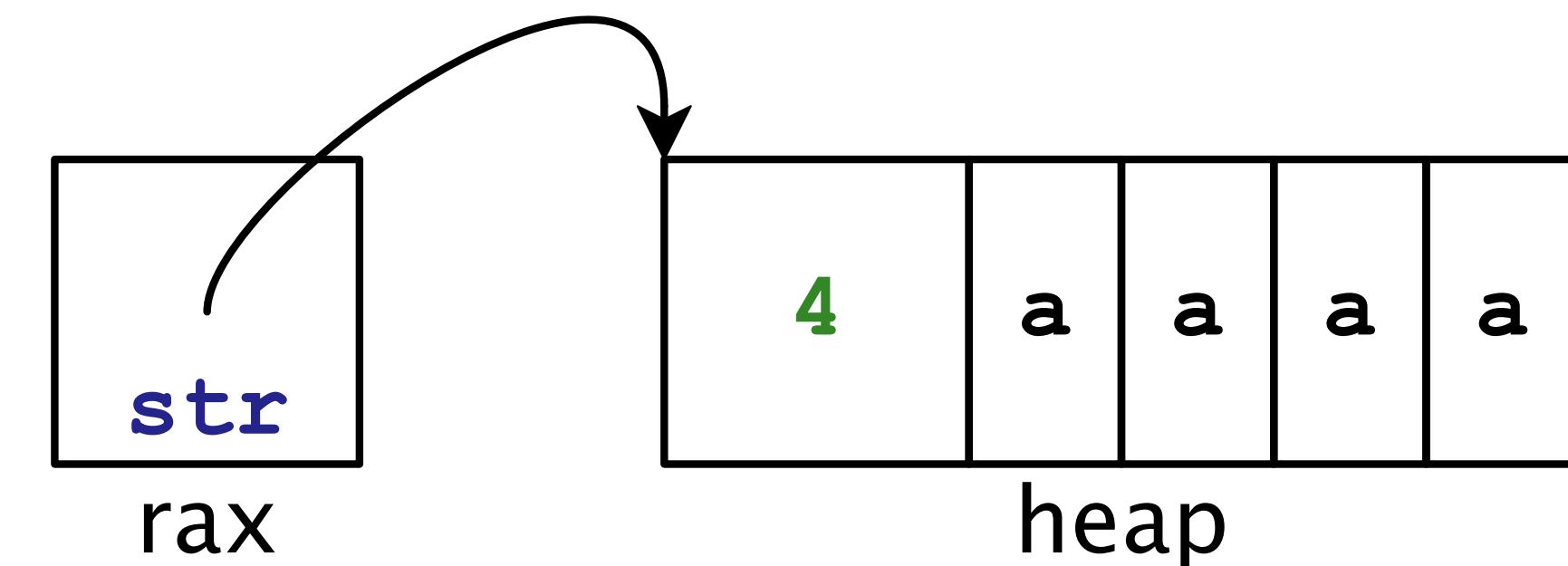
```
> (exec (parse '(make-string 4 #\a)))  
"aaaa"
```



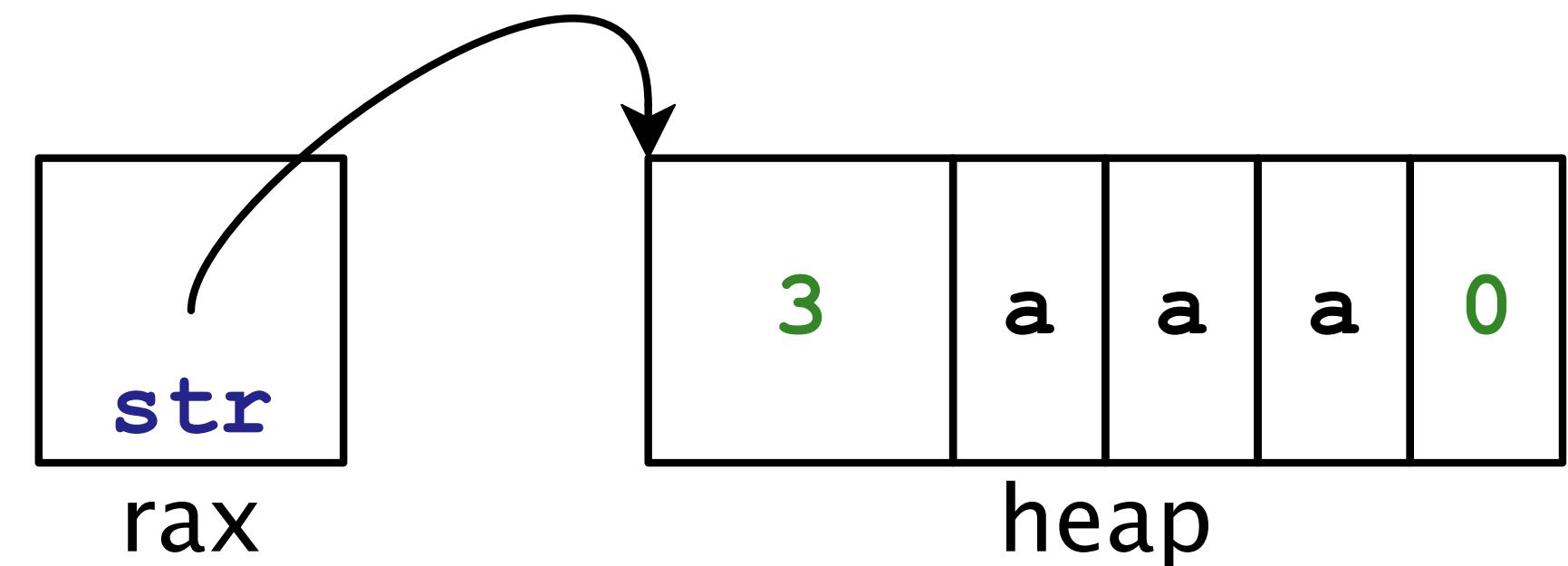
# Creating a string

A string is a tagged pointer to a sized array of *codepoints* in memory

```
> (exec (parse '(make-string 4 #\a)))  
"aaaa"
```



```
> (exec (parse '(make-string 3 #\a)))  
"aaa"
```



# **string-length**

**Just like vector-length**

```
(seq (assert-string rax)
      (Mov rax (Mem rax (- type-str))))
```

# string-ref

Similar to vector-ref, but different

```
(seq (Pop r8)
      (assert-natural rax)
      (assert-string r8)
      (Mov r9 (Mem r8 (- type-str)))
      (Cmp rax r9)
      (Jge 'err)
      (Sar rax 2)
      (Mov eax (Mem r8 rax (- 8 type-str)))
      (Sal rax char-shift)
      (Xor rax type-char))
```

string-ref

```
(seq (Pop r8)
      (assert-natural rax)
      (assert-vector r8)
      (Mov r9 (Mem r8 (- type-vect)))
      (Cmp rax r9)
      (Jge 'err)
      (Sar rax 1)
      (Mov rax (Mem r8 rax (- 8 type-vect))))
```

vector-ref

# **string-set!**

**We don't include, thus strings are immutable**

[This space intentionally blank.]

# make-string

Similar to make-vector, but different

```
(seq (Pop r8)
     (assert-natural r8)

     ; special case for length = 0
     (Cmp r8 0)
     (Jne nonzero)
     ; return canonical representation
     (Lea rax (Mem 'empty type-str))
     (Jmp theend)

     ; Code for nonzero case
     (Label nonzero)
     (Mov (Mem rbx 0) r8) ; write length
     (Sar r8 2)           ; convert to bytes
     (Mov r9 r8)          ; save for heap adjustment

     (Sar rax char-shift) ; convert to codepoint

     ; start initialization
     (Label loop)
     (Mov (Mem rbx r8 4) eax)
     (Sub r8 4)
     (Cmp r8 0)
     (Jne loop)
     ; end initialization

     (Mov rax rbx)
     (Xor rax type-str) ; create tagged pointer
     (Add rbx r9)       ; acct for elements and stored length
     (Add rbx 8)
     ; Pad to 8-byte alignment
     (Add rbx 4)
     (Sar rbx 3)
     (Sal rbx 3)
     (Label theend)))
```

Let's take it in pieces

# **make-string (1/7)**

**Similar to make-vector, but different**

(Pop r8)  
(assert-natural r8)

Just like make-vector

# make-string (2/7)

Similar to make-vector, but different

```
; special case for length = 0
(Cmp r8 0)
(Jne nonzero)
; return canonical representation
(Lea rax (Mem 'empty type-str))
(Jmp theend)
```

Just like make-vector, but  
makes an empty *string*

# make-string (3/7)

Similar to make-vector, but different

```
; Code for nonzero case
(Label nonzero)
(Mov (Mem rbx 0) r8) ; write length
(Sar r8 2)             ; convert to bytes
(Mov r9 r8)            ; save for heap adjustment
```

Just like make-vector, but half  
as many bytes

# **make-string (4/7)**

**Similar to make-vector, but different**

(**Sar rax char-shift**) ; convert to codepoint

Unlike make-vector, don't  
write value, write codepoint

# make-string (5/7)

Similar to make-vector, but different

```
; start initialization
(Label loop)
(Mov (Mem rbx r8 4) eax)
(Sub r8 4)
(Cmp r8 0)
(Jne loop)
; end initialization
```

Just like make-vector, but  
writing 4 bytes at a time

# make-string (6/7)

**Similar to make-vector, but different**

```
(Mov rax rbx)
(Xor rax type-str)      ; create tagged pointer
(Add rbx r9)            ; acct for elements and stored length
(Add rbx 8)
```

Just like make-vector

# make-string (7/7)

Similar to make-vector, but different

```
; Pad to 8-byte alignment  
(Add rbx 4)  
(Sar rbx 3)  
(Sal rbx 3)
```

Unlike make-vector, need to pad to preserve heap invariant when string-length is odd

# String literals

String literals are compiled to static memory

```
> (compile-datum "wilma")
(list
 (Data)
 (Label 'string655906)
 (Dq 80)
 (Dd 119)
 (Dd 105)
 (Dd 108)
 (Dd 109)
 (Dd 97)
 (Dd 0)
 (Text)
 (Lea 'rax (Mem ($ 'string655906) 4)))
```

# Syntactic changes

Programs are now sequences of definitions and an expression

Concrete syntax:

Programs:

```
(define (f1 x11 ...) e1)
(define (f2 x12 ...) e2)
...
e
```

Expressions:

```
(f e1 ...)
```

Abstract syntax:

```
;; type Prog = (Prog (Listof Defn) Expr)
(struct Prog (ds e) #:prefab)

;; type Defn = (Defn Id (Listof Id) Expr)
(struct Defn (f xs e) #:prefab)

;; type Expr = ...
;;           | (App Id (Listof Expr))
```

# Parsing changes

Programs are now sequences of definitions and an expression

```
;; S-Expr ... -> Prog
(define (parse . s) ...)
```

```
;; S-Expr -> Expr
(define (parse-e s) ...)
```

```
;; S-Expr -> Defn
(define (parse-define s) ...)
```

# Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; Prog -> Answer
(define (interp p) ...)
```

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ...)
```

# **CMSC 430 - 30 October 2025**

## **Calling conventions**

### **Today**

- Compiling function definitions and calls
- Review quiz

### **Announcements**

- (Another) quiz due by start of class Tuesday
- Assignment 7 will go out tonight

# Function call example

What should this evaluate to?

Concretely: `(define (f x y) (+ x y))  
(f 1 2)`

```
(interp-env (App 'f (list (Lit 1) (Lit 2)))  
           '())  
           (list (Defn 'f ' (x y)  
                      (Prim2 '+ (Var 'x) (Var 'y)))) )
```

Same thing as:

```
(interp-env (Prim2 '+ (Var 'x) (Var 'y))  
           '((x 1) (y 2))  
           (list (Defn 'f ' (x y)  
                      (Prim2 '+ (Var 'x) (Var 'y)))) )
```

# Function calls in general

What should this evaluate to?

The meaning of a function application is the meaning of the function definition's body expression, where each parameter is bound the value of the argument

```
(interp-env (App 'f (list e1 e2 ...)) r ds)
;;  where (Defn 'f (list x1 x2 ...) e) in ds
```

Same thing as:

```
(interp-env e (list (list x1 (interp-env e1 r ds))
                      (list x2 (interp-env e2 r ds)))
                      ...)
ds)
```

# Designing our own calling convention

Function calls are like “let at a distance”

```
( f 3 4)    (define (f x y)
                  (+ x y))
```

is like

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

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

# 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*

# CMSC 430 - 4 November 2025

## Tail calls

### Today

- Quick fix to function-names-as-labels
- Tour of Assignment 7 features
- JavaScript approach to arity
- Proper tail calls

### Announcements

- (Another) quiz due by start of class Thursday

# Assignment 7

## Variable-arity and arity-overloaded functions

Three kinds of function definition:

```
(define (f1 x y) x)
```

```
(define (f2 x . xs) xs)
```

```
(define f3
  (case-lambda
    [(x y) x]
    [(x . xs) xs]))
```

```
v (f1 3 4)
3
v (f2 1)
'()
v (f2 1 2 3 4)
'(2 3 4)
v (f3 3 4)
3
v (f3 1)
'()
v (f3 1 2 3 4)
'(2 3 4)
```

# Assignment 7

## Kind 1: Plain old function with fixed arity

Three kinds of function definition:

```
(define (f1 x y) x)
```

```
> (f1 3 4)  
3
```

# Assignment 7

## Kind 2: Function with variable-arity using "rest" argument

Three kinds of function definition:

```
(define (f2 x . xs) xs)
          v ( f2 1 )
          ' ( )
          v ( f2 1 2 3 4 )
          ' (2 3 4)
```

# Assignment 7

## Kind 3: Arity-overloading with case-lambda

Three kinds of function definition:

```
(define f3
  (case-lambda
    [(x y) x]
    [(x . xs) xs]))
```

v (f3 3 4)  
3  
v (f3 1)  
'( )  
v (f3 1 2 3 4)  
'(2 3 4)

# 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

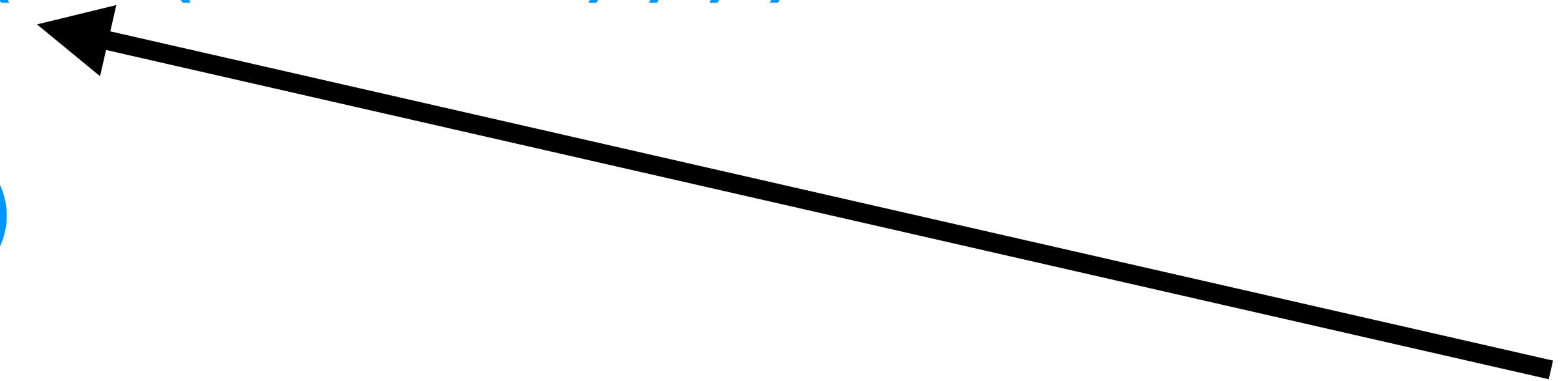
[ ret1 ]	[ ret1 ]	...	[ ret1 ]	[ ret1 ]	...	[ ret1 ]
[ 100 ]	[ 100 ]		[ 100 ]	[ 100 ]		[ 100 ]
[ ret2 ]		[ ret2 ]	[ ret2 ]	[ ret2 ]		
[ 99 ]		[ 99 ]	[ 99 ]	[ 99 ]		
[ ret2 ]		[ ret2 ]	[ ret2 ]	[ ret2 ]		
		[ 98 ]	[ 98 ]	[ 98 ]		
		:		:		
		[ ret2 ]	[ ret2 ]			
		[ 1 ]	[ 1 ]			
		[ ret2 ]				
		[ 0 ]				

```
;; 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)
```



; Morally:

```
(seq (Mov 'rax (value->bits 100))
      (Lea 'r8 'ret1)
      (Push 'r8)
      (Push 'rax)
      (Jmp 'f)
      (Label 'ret1)
      (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?

```
(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

```
[ ret1 ] [ ret1 ] ... [ ret1 ] [ ret1 ] ... [ ret1 ] [ ret1 ]  
[ 100 ] [ ret2 ] ... [ ret2 ] [ ret2 ] ... [ ret2 ]  
[    99 ] [ ret2 ] [ ret2 ]  
          :       :  
          [ ret2 ] [ ret2 ]  
          [ ret2 ] [ ret2 ]  
          [    0 ]
```

;; 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

```
[ ret1 ] [ ret1 ] ... [ ret1 ] [ ret1 ] ... [ ret1 ] [ ret1 ]  
[ 100 ] [ ret2 ] ... [ ret2 ] [ ret2 ] ... [ ret2 ]  
[    99 ] [ ret2 ] [ ret2 ]  
      :       :  
      [ ret2 ] [ ret2 ]  
      [ ret2 ] [ ret2 ]  
      [    0 ]
```

```
[ ret1 ] [ ret1 ] ... [ ret1 ]  
[ 100 ] [    99 ] ... [    0 ]
```

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:

[ ret1 ]	[ ret1 ]	...	[ ret1 ]	[ ret1 ]	...	[ ret1 ]
[ 100 ]	[ 100 ]		[ 100 ]	[ 100 ]		[ 100 ]
	[ ret2 ]		[ ret2 ]	[ ret2 ]		
	[ 99 ]		[ 99 ]	[ 99 ]		
	[ ret2 ]		[ ret2 ]	[ ret2 ]		
			[ 98 ]	[ 98 ]		
				:		:
			[ ret2 ]	[ ret2 ]		
				[ 1 ]	[ 1 ]	
				[ ret2 ]		
					[ 0 ]	

Tail call:

[ ret1 ]	[ ret1 ]	...	[ ret1 ]
[ 100 ]	[ 99 ]	...	[ 0 ]

(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) ; No tail call
     (append (reverse xs)
             (list x))]))
```

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

# 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      t? - is this expression in tail
(define (compile-e e c t?) ...)    position
```

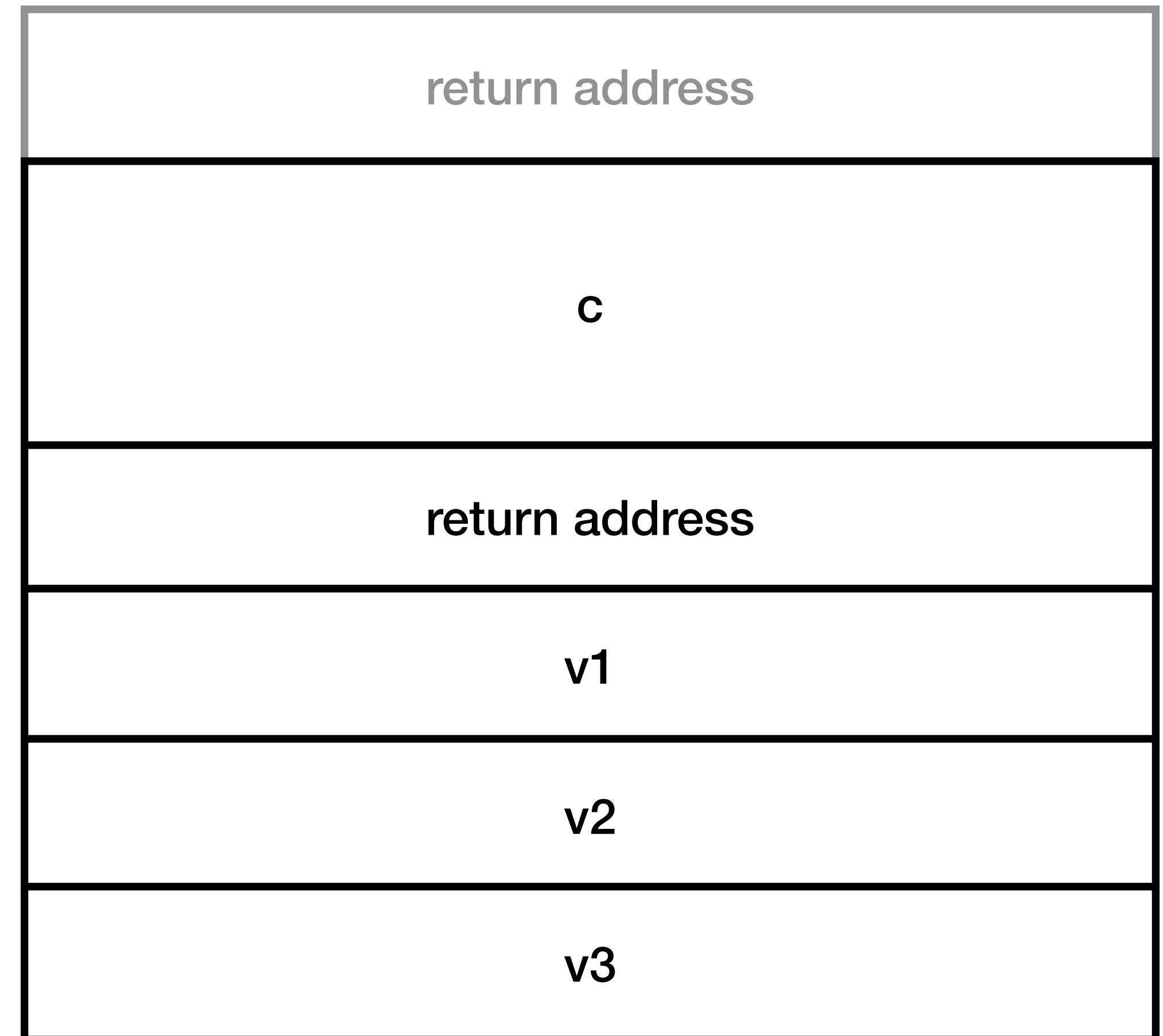
```
;; Id [Listof Expr] CEnv Boolean -> Asm
(define (compile-app f es c t?)
  (if t?
      (compile-app-tail f es c)
      (compile-app-nontail f es c)))
```

# 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))
```

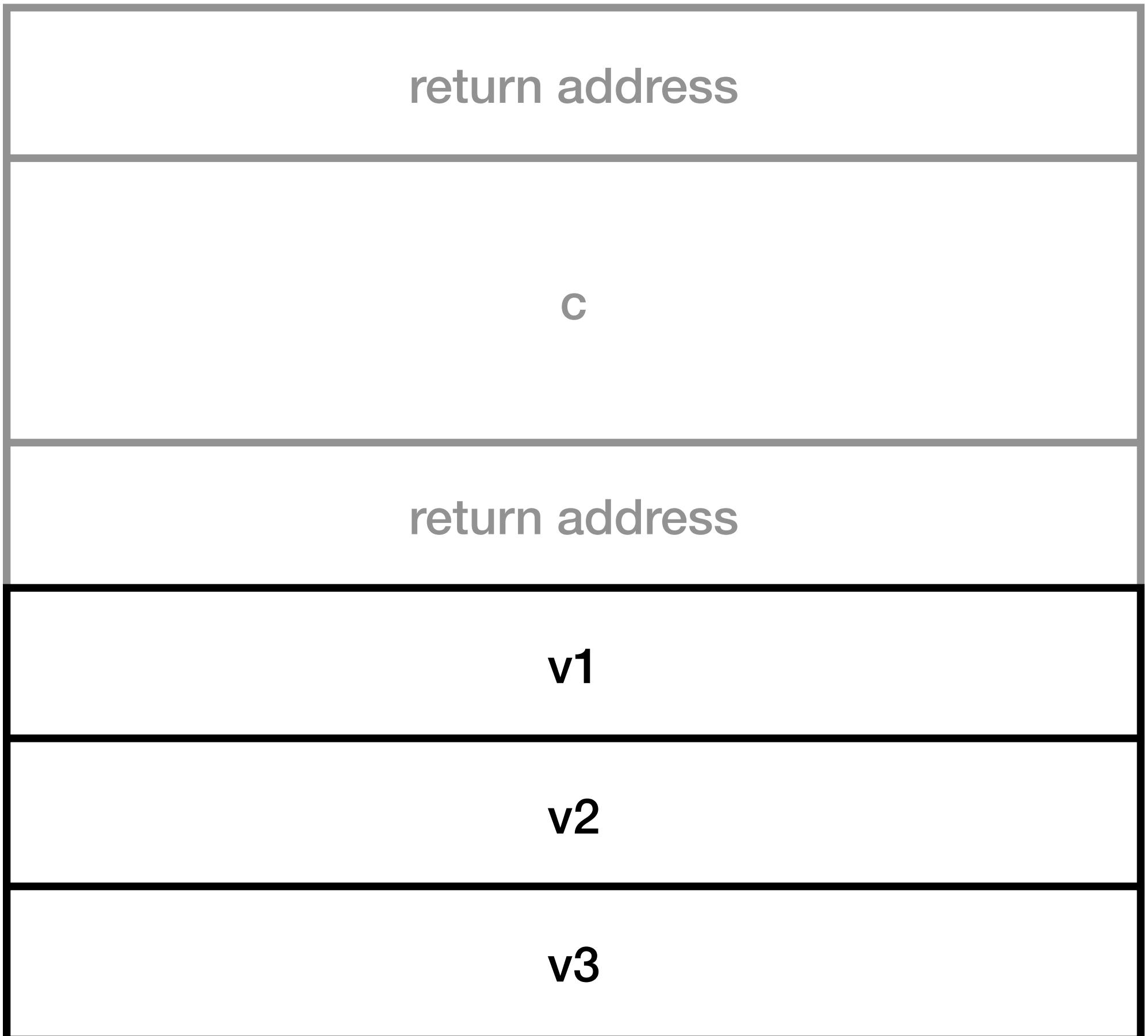


# Non-tail calls

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))
```

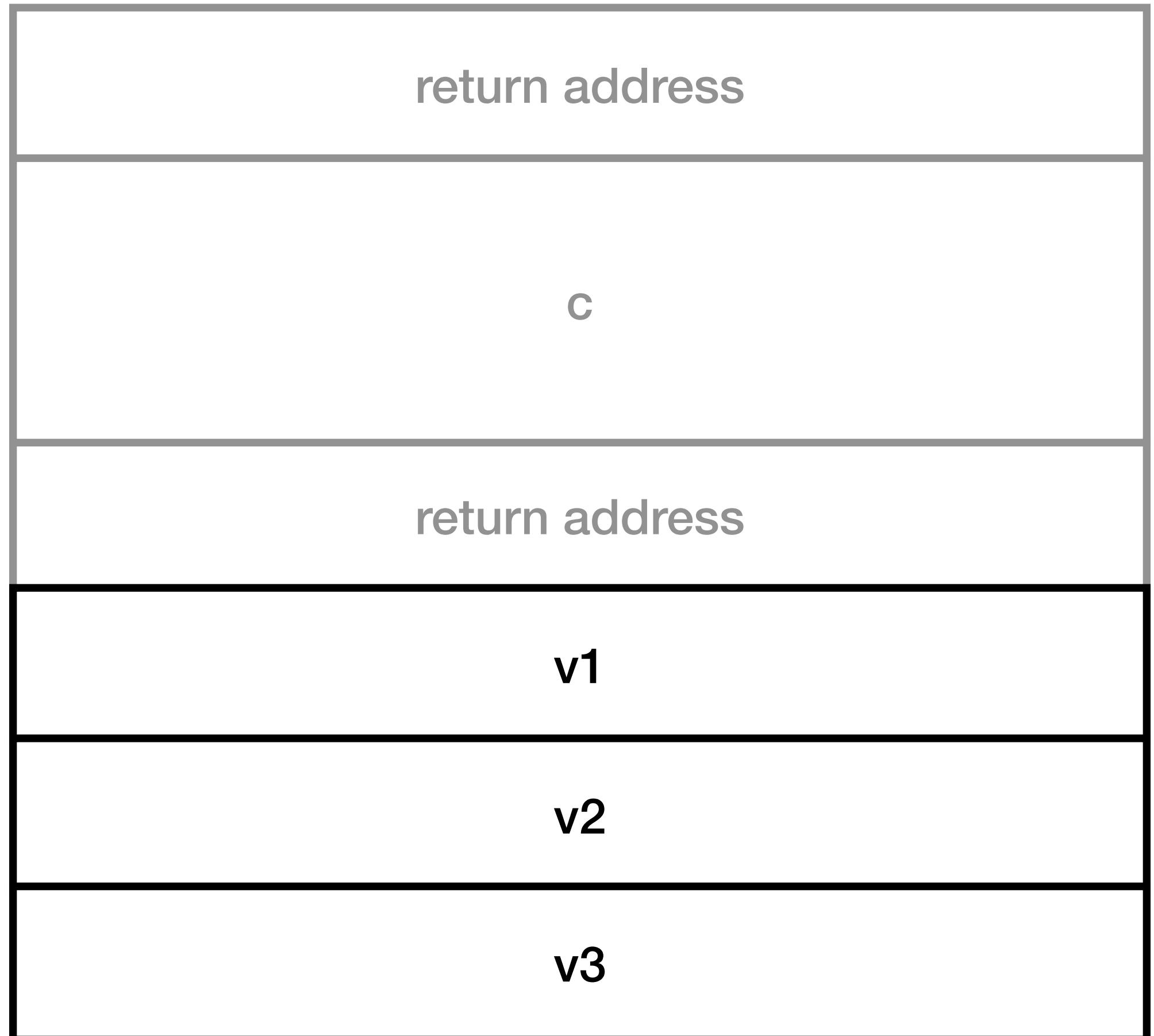
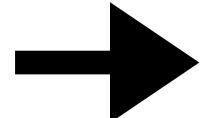


# Non-tail calls

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))
```

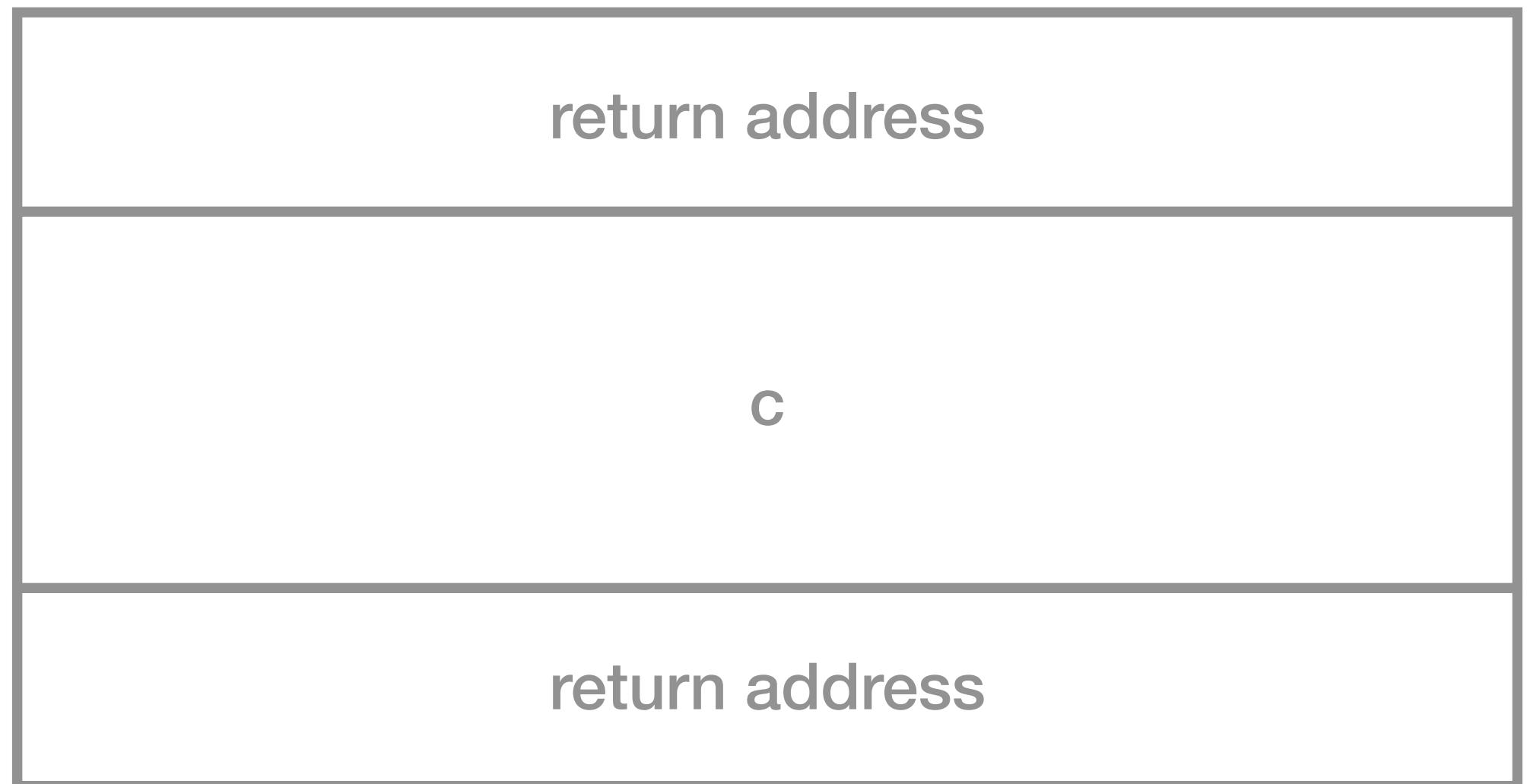


# Non-tail calls

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))
```

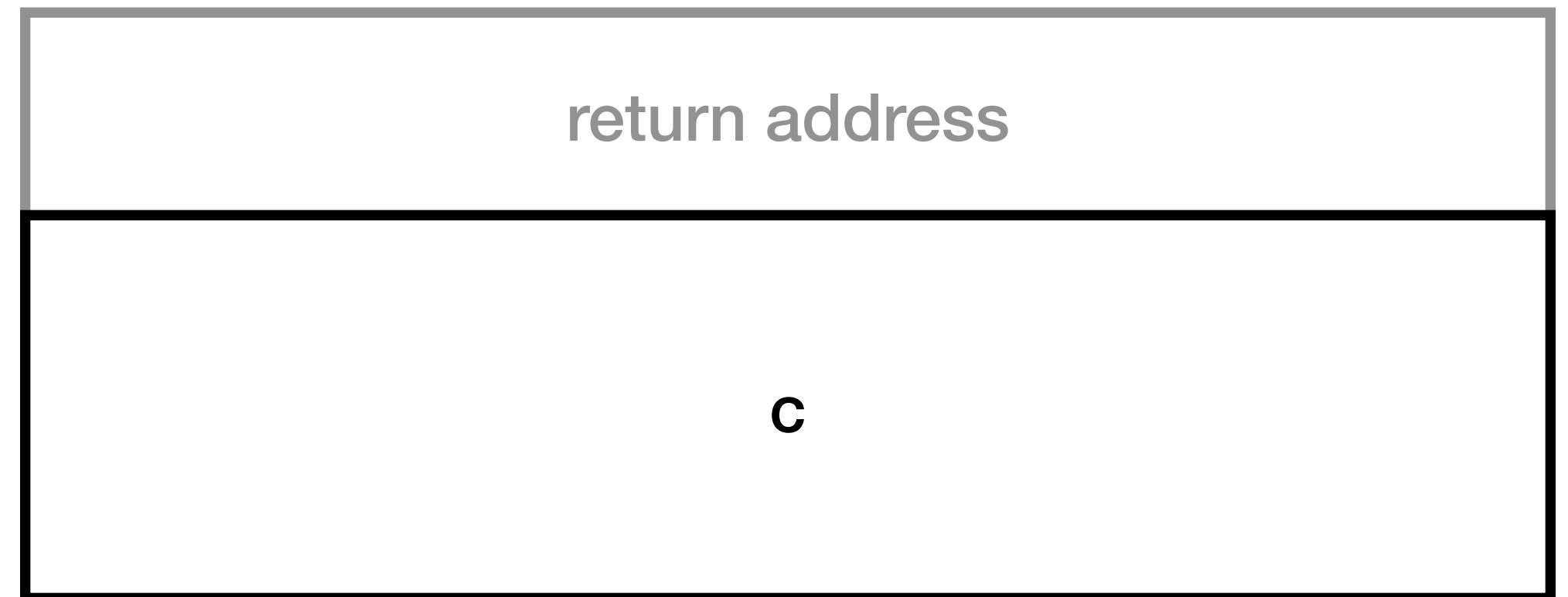


# Non-tail calls

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))
```

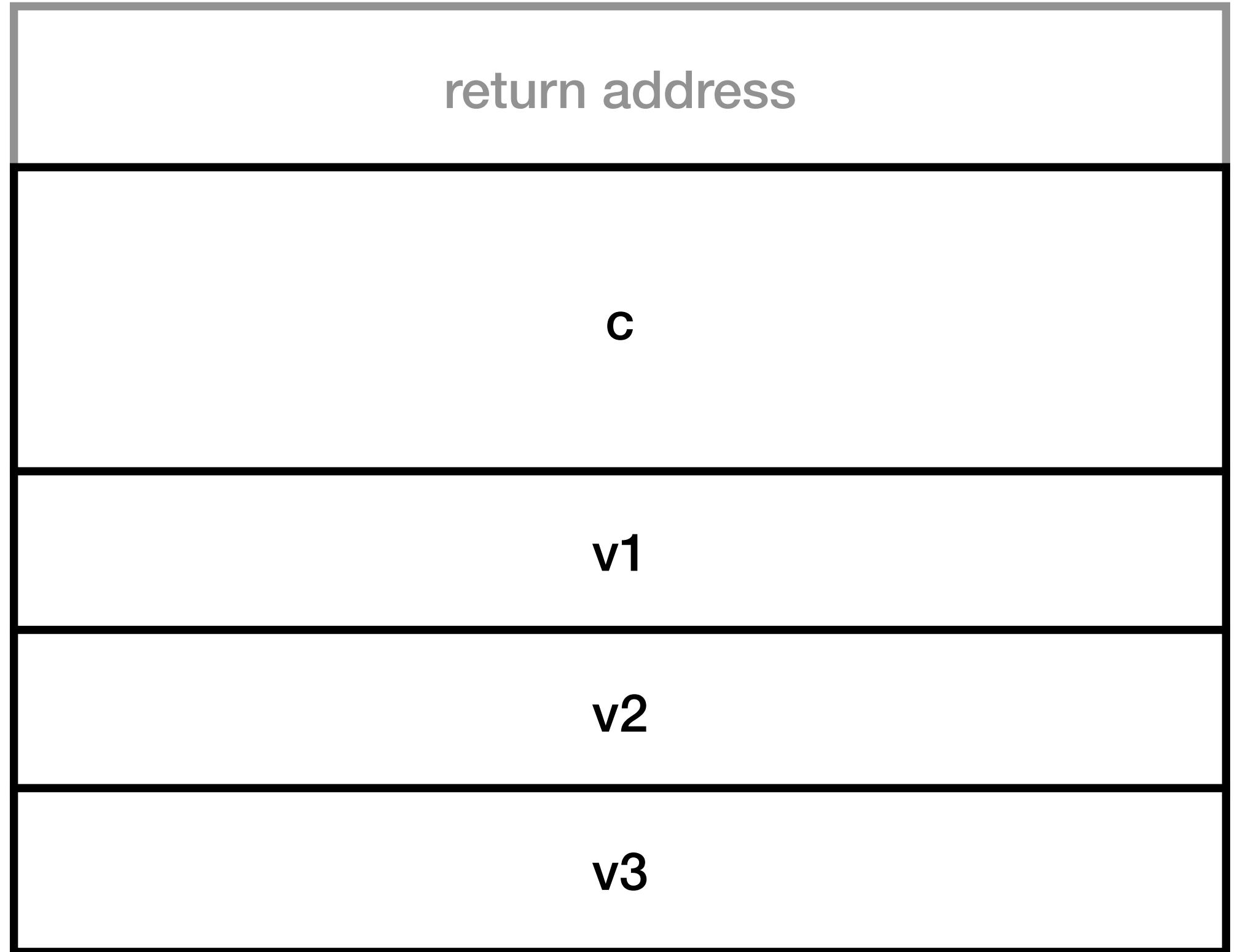


# Tail calls (first attempt)

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) )
```

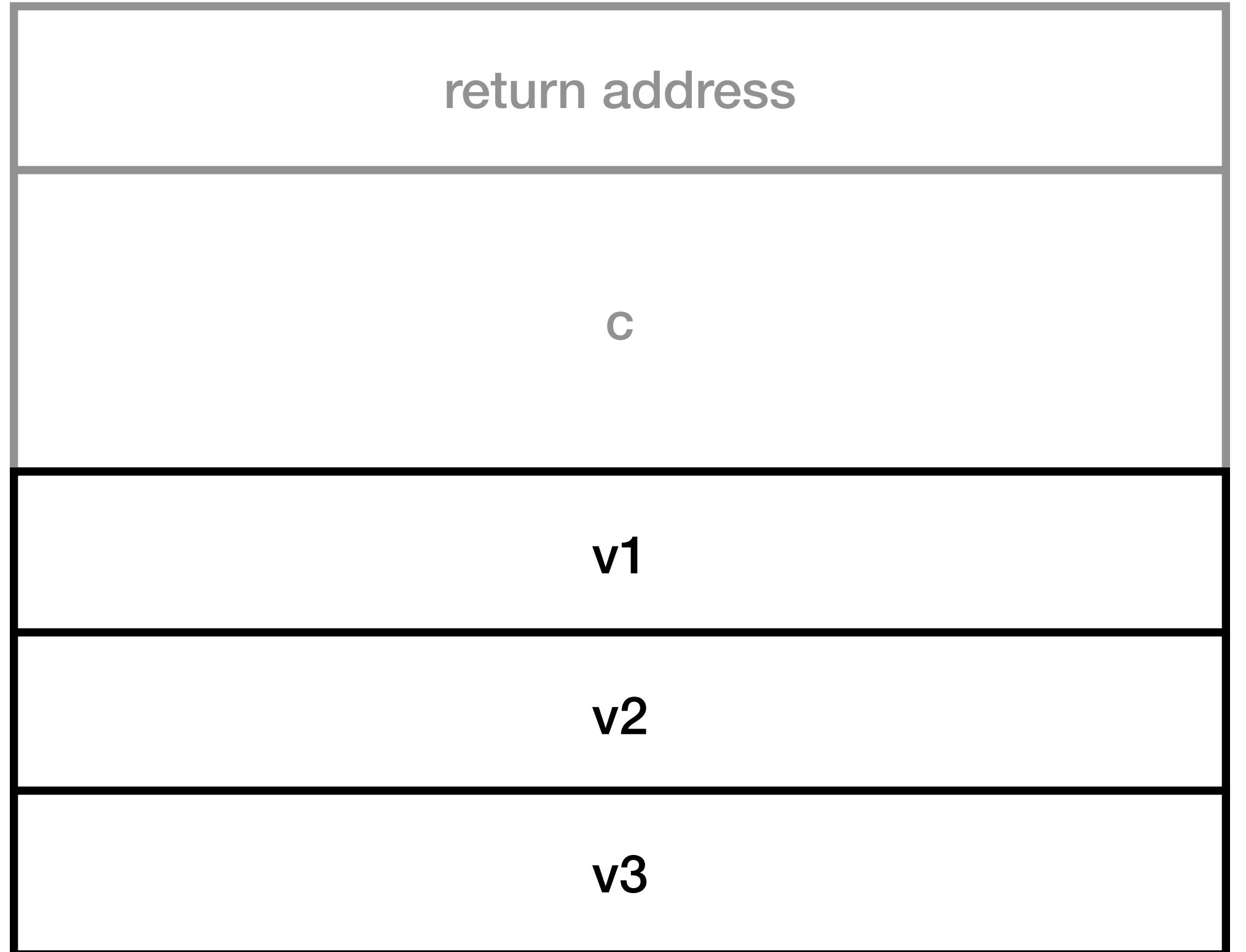


# Tail calls (first attempt)

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))
```

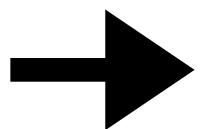


# Tail calls (first attempt)

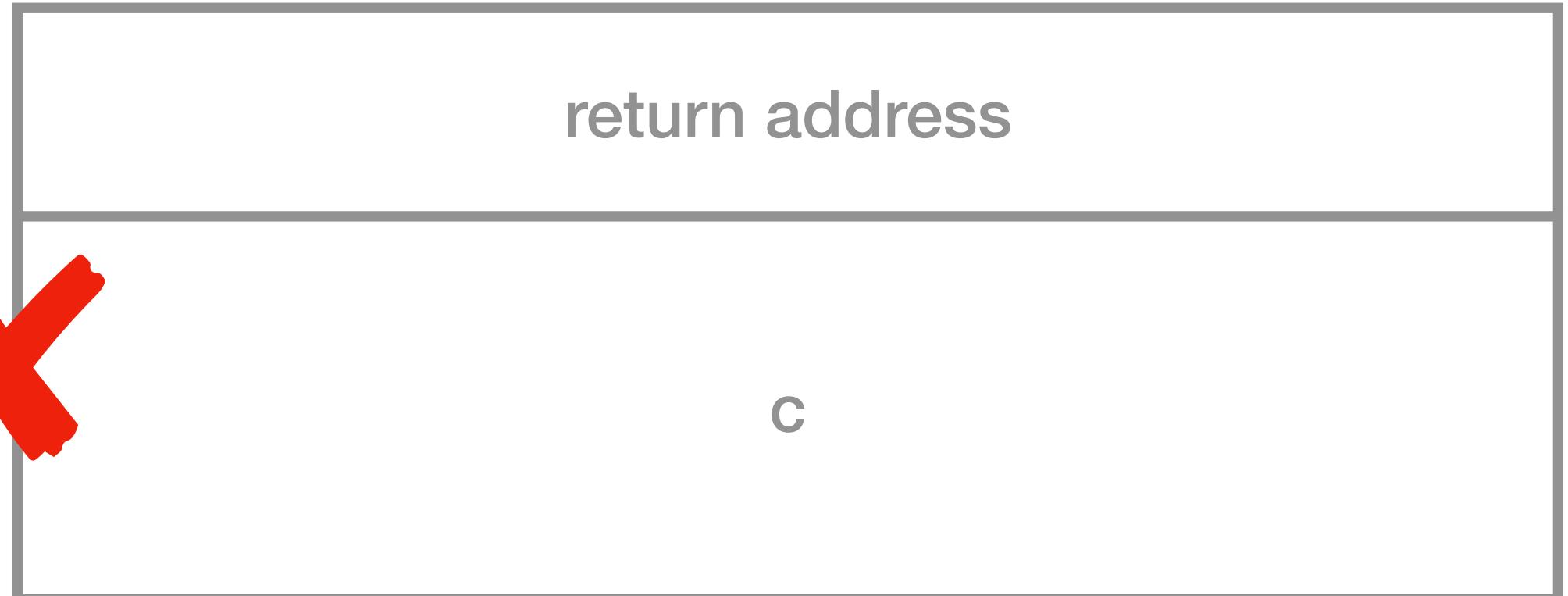
No more work to do after call, so don't return

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

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



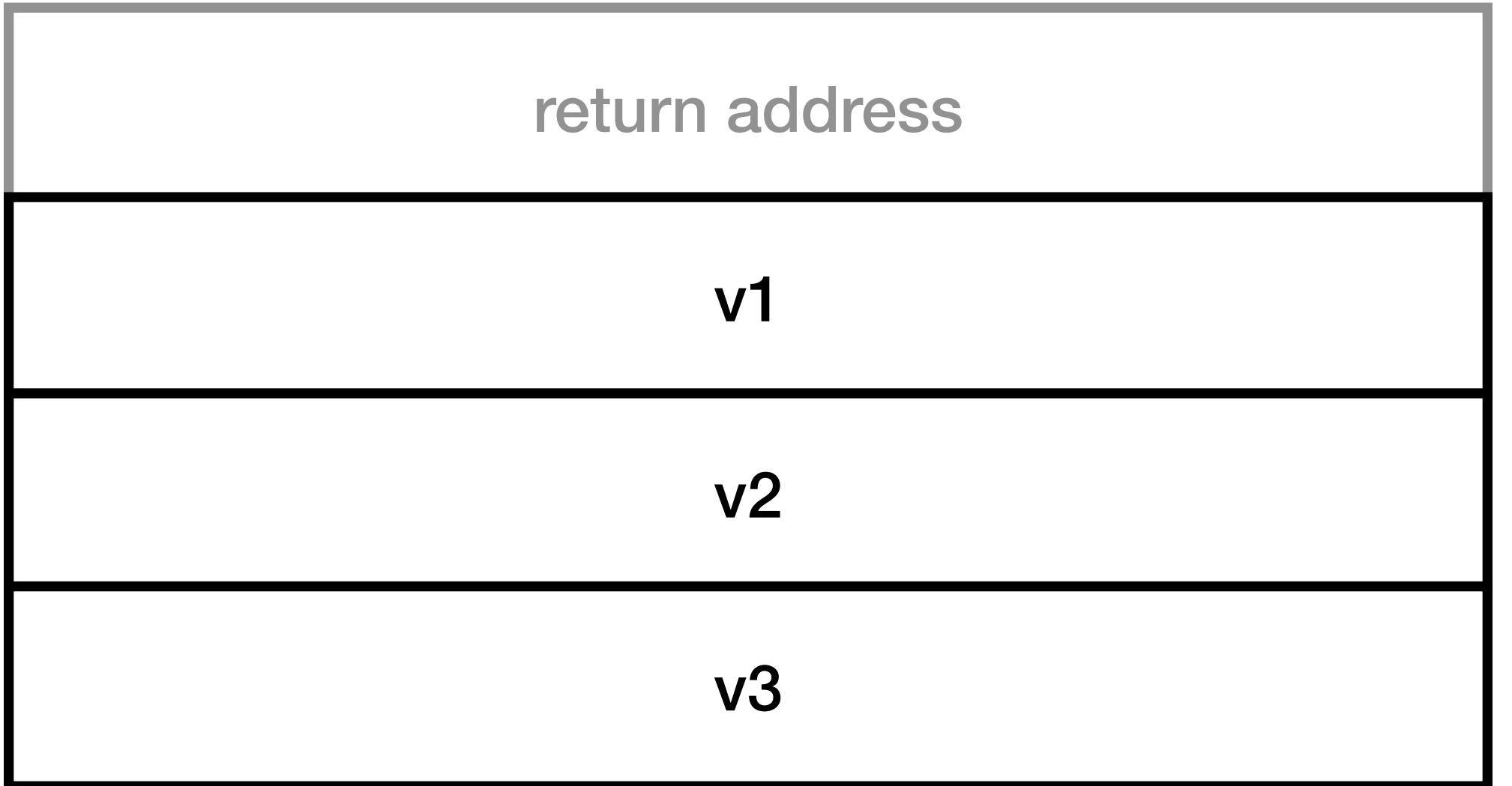
```
(Ret))
```



# Tail calls (second attempt)

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

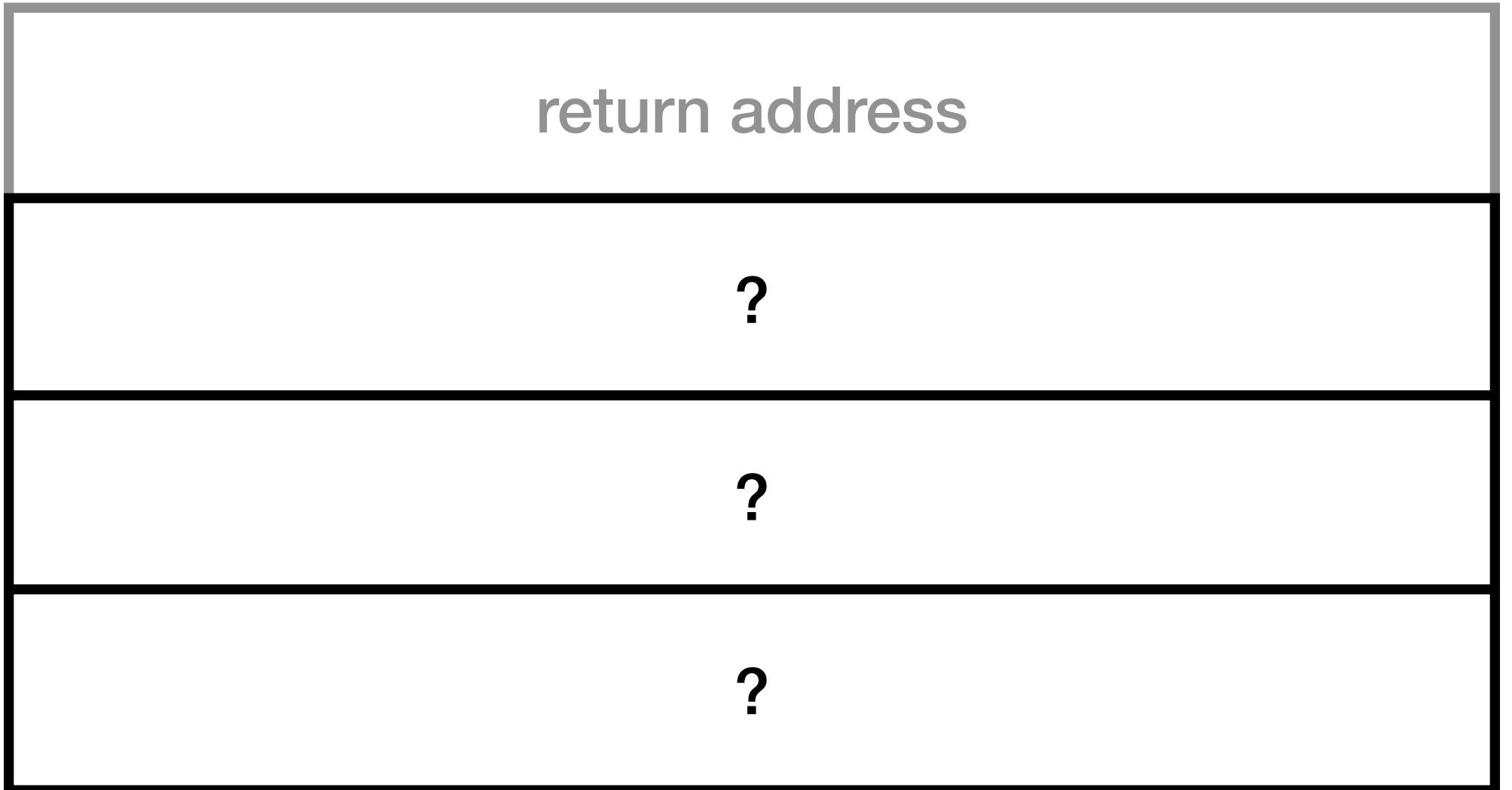
```
(compile-app-tail 'f (list e1 e2 e3) ) c)
(seq
  (Add rsp (* 8 (length c)))
  (compile-e e1 c)
  (Push rax)
  (compile-e e2 (cons #f c))
  (Push rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push rax)
  → (Jmp 'f))
```



# 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))
```

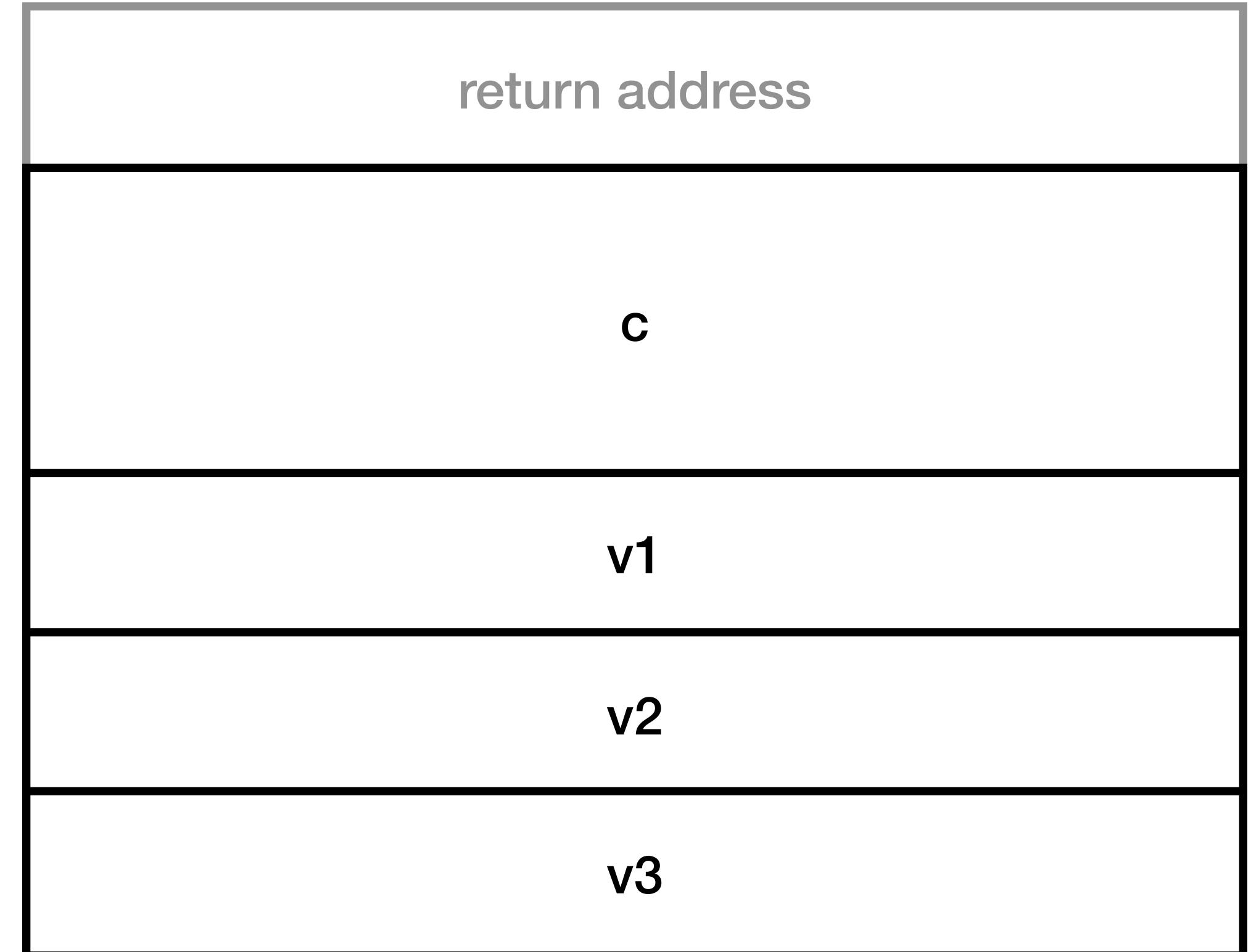


# Tail calls (final attempt)

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))
```

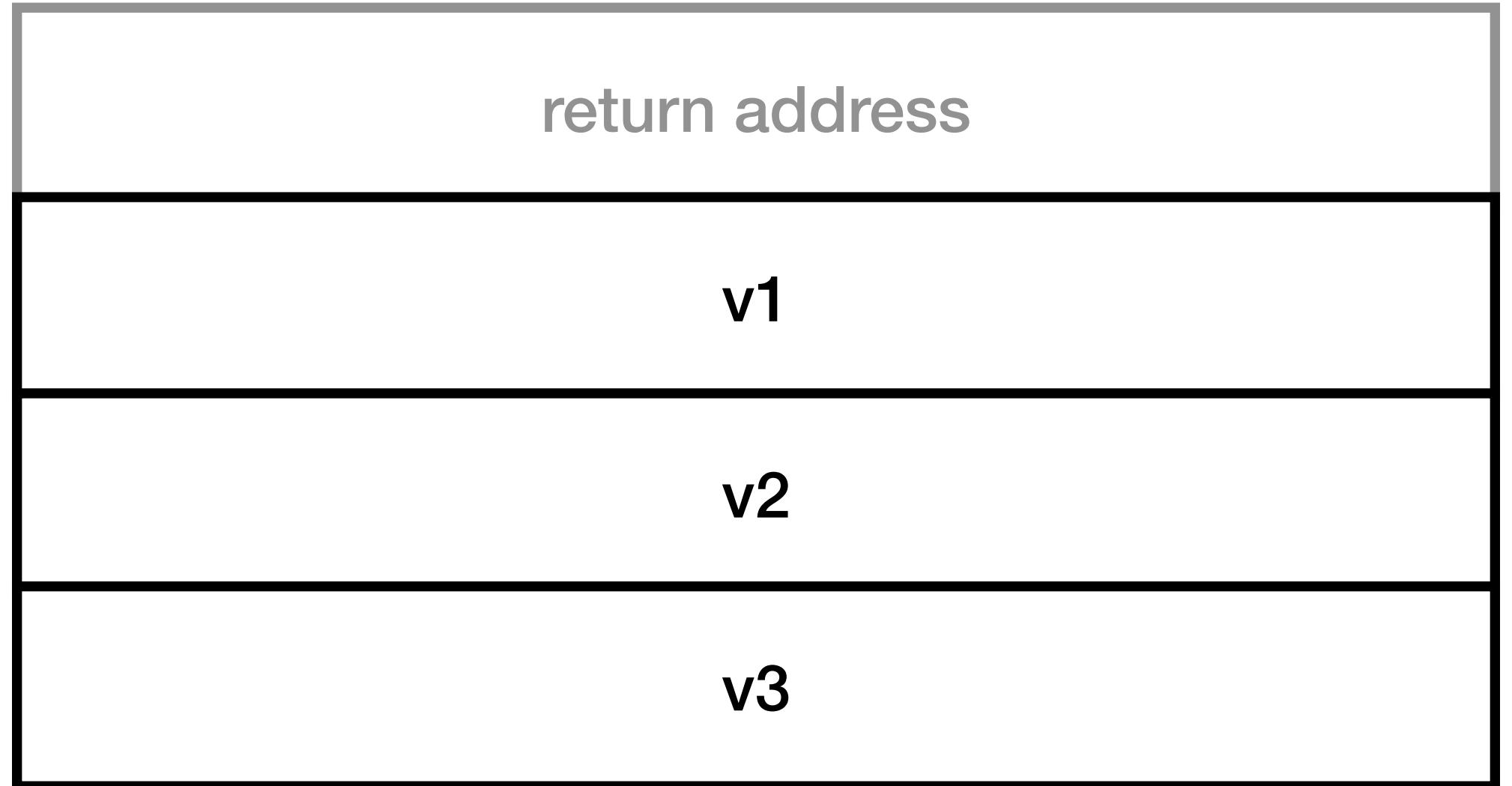


# Tail calls (final attempt)

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

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

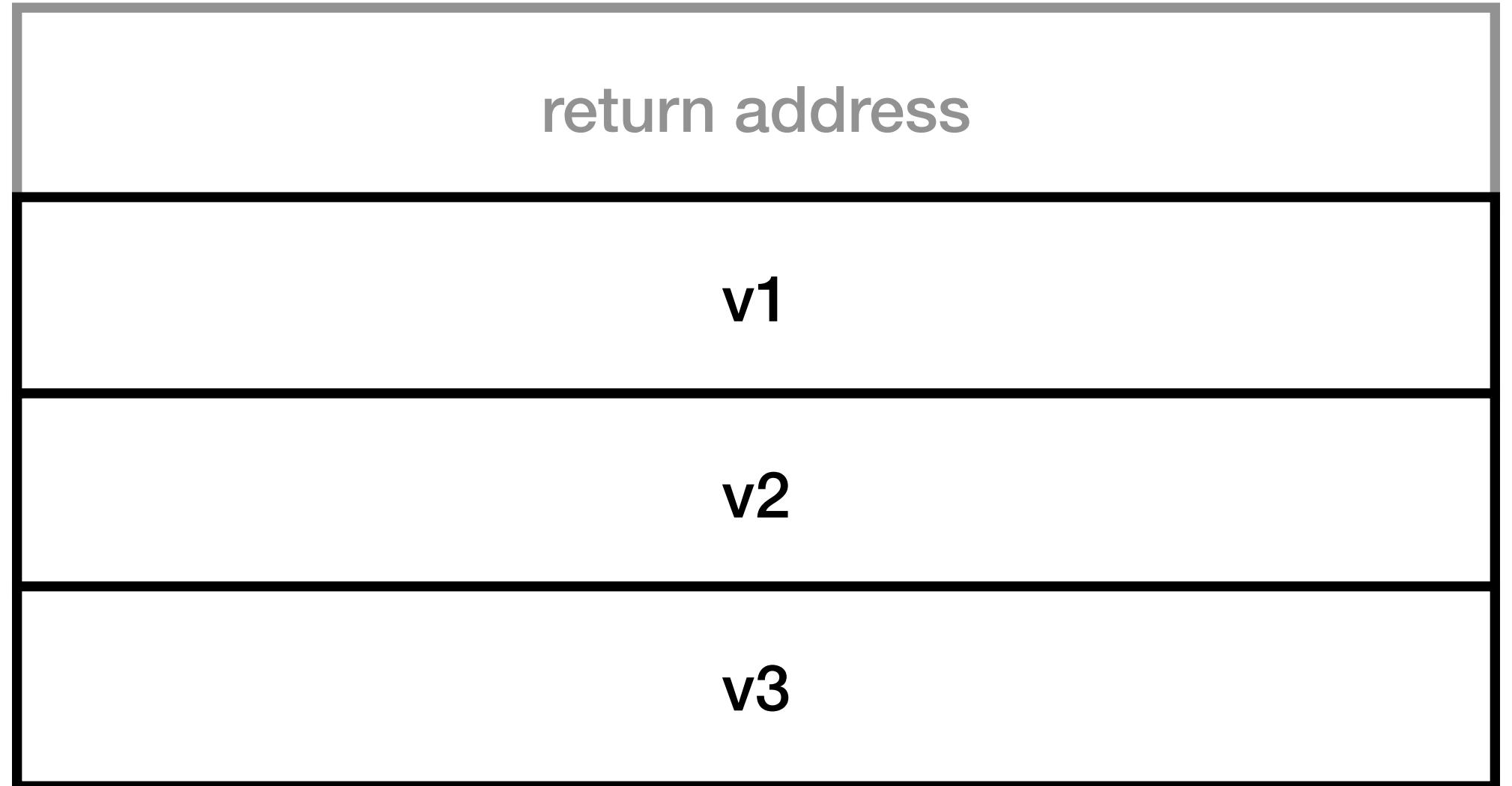
(seq
  (compile-e e1 c)
  (Push rax)
  (compile-e e2 (cons #f c))
  (Push rax)
  (compile-e e3 (cons #f (cons #f c)))
  (Push rax)
  (move-args 3 (length c))
  (Add rsp (* 8 (length c)))
  → (Jmp 'f))
```



# Tail calls (final attempt)

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

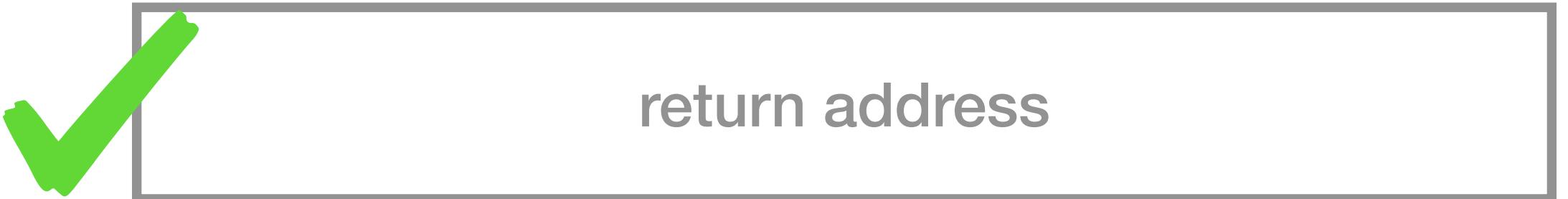
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))  
  (seq  
    → (Label 'f)  
    (compile-e e (list 'z 'y 'x) #t)  
    (Add rsp 24)  
    (Ret) )
```



# 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))
```

# **CMSC 430 - 6 November 2025**

## **Compiling tail calls, maybe patterns**

### **Today**

- Proper tail calls
- Pattern matching, if time allows

### **Announcements**

- Another quiz out shortly after class
- Midterm 2 in one week

# Some calls happen in "tail contexts"

## Grammar of tail calls

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

<tail> ::=

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

# Some calls happen in "tail contexts"

## Grammar of tail calls

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)) )
      e6) )
```

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

<tail> ::=

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

# A tale of a call

**Someone calls g**

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```



return to caller of g

x's value

y's value

z's value

...

# A tale of a call

**Someone calls g**

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)) )
      e6))
```

...  
return to caller of g

x's value

y's value

z's value

p's value

# A tale of a call

Someone calls g

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```

...  
return to caller of g

x's value

y's value

z's value

p's value

return to caller of f

e3's value

e4's value

e5's value



# A tale of a call

Someone calls g

```
(define (f p q r) ...)
```



return to caller of f

e3's value

e4's value

e5's value

# A tale of a call

Someone calls g

```
(define (f p q r) ...)
```



return to caller of f

e3's value

e4's value

e5's value

rax = (f e3 e4 e5)'s value

# A tale of a call

Someone calls g

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```

rax = (f e3 e4 e5)'s value

...

return to caller of g

x's value

y's value

z's value

p's value

# A tale of a call

Someone calls g

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```



return to caller of g

x's value

y's value

z's value

rax = (f e3 e4 e5)'s value

# A tale of a call

**Someone calls g**

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```



rax = (f e3 e4 e5)'s value

...

return to caller of g

x's value

y's value

z's value

# A tale of a call

Someone calls g

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```



return to caller of g

x's value

y's value

z's value

We end up with the result of the call  
to f in rax and return to g's caller

rax = (f e3 e4 e5)'s value

# What it means to be in a tail context

The instructions that will happen *after* returning from the call will be equivalent to:

(Pop)<sup>\*</sup>  
(Ret)

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

# A tale of a tail call

**Someone calls g**

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```

...  
return to caller of g

x's value

y's value

z's value

# A tale of a tail call

**Someone calls g**

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)) )
      e6))
```

...  
return to caller of g

x's value

y's value

z's value

p's value

# A tale of a tail call

Someone calls g

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```

...  
return to caller of g

x's value

y's value

z's value

p's value

return to caller of f

e3's value

e4's value

e5's value



# A tale of a tail call

**Someone calls g**

```
(define (g x y z)
  (if e0
      (begin e1
              (let ((p e2))
                (f e3 e4 e5)))
      e6))
```

...  
return to caller of g

e3's value

e4's value

e5's value



# A tale of a tail call

Someone calls g

```
(define (f p q r) ...)
```



return to caller of g

e3's value

e4's value

e5's value

# A tale of a tail call

Someone calls g

```
(define (f p q r) ...)
```



return to caller of g

e3's value

e4's value

e5's value

We end up with the result of the call  
to f in rax and return to g's caller  
**EXACTLY LIKE BEFORE**

rax = (f e3 e4 e5)'s value

# Pattern matching: syntax

```
;; type Expr = ...
;;           | (Match Expr (Listof Pat) (Listof Expr))
```

```
;; type Pat  = (Var Id)
;;           | (Lit Datum)
;;           | (Box Pat)
;;           | (Cons Pat Pat)
;;           | (Conj Pat Pat)
```

# Pattern matching: semantics

## The key parts

```
;; Expr Env -> Answer
(define (interp-env e r ds)
  (match e
    ;; ...
    [(Match e ps es)
     (match (interp-env e r ds)
       ['err 'err]
       [v
        (interp-match v ps es r ds)])]
```

```
;; Value [Listof Pat] [Listof Expr] Env Defns -> Answer
(define (interp-match v ps es r ds) '....)
```

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r) '....)
```



The heart of pattern matching

# Pattern matching: semantics

## The key part: examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r) '....)
```

```
> (interp-match-pat (Var '_) 99 '())
'()
> (interp-match-pat (Var 'x) 99 '())
'((x 99))
> (interp-match-pat (Lit 99) 99 '())
'()
> (interp-match-pat (Lit 100) 99 '())
#f
> (interp-match-pat (Conj (Lit 99) (Var 'x)) 99 '())
'((x 99))
```

# Pattern matching: semantics

## The key part: examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r) '....)
```

```
> (interp-match-pat (Conj (Lit 99) (Var 'x)) 99 '())
'((x 99))
> (interp-match-pat (Conj (Lit 100) (Var 'x)) 99 '())
#f
> (interp-match-pat (Cons (Var 'x) (Var 'y)) 99 '())
#f
> (interp-match-pat (Cons (Var 'x) (Var 'y)) (cons 99 100) '())
'((y 100) (x 99))
> (interp-match-pat (Cons (Cons (Var 'x) (Var 'y))
                           (Cons (Var 'p) (Var 'q))))
  (cons (cons 99 100)
        (cons #t #f)))
'()
'((q #f) (p #t) (y 100) (x 99))
```