CMSC 430 - 24 January 2024

Introduction to Introduction to Compilers

Course logistics

What is a compiler?

Quick overview

Ocaml to Racket

Course logistics

Compilers comes at you fast

Lectures every Mondays & Wednesday (except holidays, midterms)

~6 assignments

2 midterms (take-home, 24 hour); dates TBD

1 final project (counts as final exam)

Several surveys and quizzes (on ELMS)

Course is very cumulative; building essentially one program all semester

Course logistics

Key resources

Class web page (syllabus, assignments, course notes)

ELMS (announcements, recordings, grades)

Piazza (communication, discussion)

José’s videos on Youtube

What is a programming language?

No, really, I'm asking…

An interface with the computer

A formal language (a set of programs) that can describe any Turing machine

Has a set of formal rules for computing (semantics)

A way of generating instructions for a computer

Is HTML a programming language?

A mechanism for communicating computational ideas with other people

Legible by humans (?)

What is a compiler?

No, really, I’m asking…

A program that translates between two different PLs

A program to tokenize every part of a program into objects or ideas

From something human understandable to something machine understandable

Quick overview

The Design and Implementation of Programming Languages

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

Quick overview
The Design and Implementation of Programming Languages

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”

CMSC 430 - 29 January 2024

OCaml to Racket

Announcements

- Getting to Know You survey on ELMS, due 1/31
- Piazza and Gradescope are up
- Midterm dates have been set: 3/6, 4/17
- Assignment 1 posted; due Mon 2/5; may be done collaboratively
- Office hours will start on Wednesday (schedule posted by Tues COB)
- Anonymous feedback form

Ocaml to Racket

Racket = OCaml - Types - Syntax

You should have:

- Installed Racket
- Read the notes on “From OCaml to Racket”

CMSC 430 - 31 January 2024

OCaml to Racket, cont.

Announcements

- Reminder: ELMS survey due tonight!
- Reminder: Assignment 1 due Mon 2/5; may be done collaboratively
- Office hours started: schedule on web page
- Quiz on Racket Basics due by start of next class (ELMS)

Today

- More Racket: symbols, lists, structures, s-expressions, and systematic programming

Quick review of errors

Parse, syntax, & run-time errors

- Parsing: not grammatically well formed string (e.g. unbalanced parens)
- Syntax: not correctly “shaped” expression, unbound variables
- Run-time: well-formed program that crashes 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 if they are spelled the same.

Lists and pairs in Racket vs OCaml

Constructors in OCaml

OCaml lists:

- `[] : ‘a list`
- `(::) : ‘a -> ‘a list -> ‘a list`
- `[.;.;.;…]` convient notation for lists

OCaml pairs (and tuples):

- `(,) : ‘a -> ‘b -> ‘a * ‘b`

Pairs and lists: fundamentally different things

Lists and pairs in Racket vs OCaml

Constructors in Racket

Racket lists:

- `'() : 'a list`
- `cons : 'a -> 'a list -> 'a list`
- `list` convenient function for lists

Racket pairs (and tuples):

- `cons : 'a -> 'b -> 'a * 'b`

Pairs and lists: made out of the same stuff

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 ".")

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).

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

CMSC 430 - 5 February 2024

A little assembly

Announcements

- Assignment 2 out tonight, due 2/12; may be done collaboratively
- More Racket Basics quiz due Wed 2/7 at **3:30PM**

Today

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

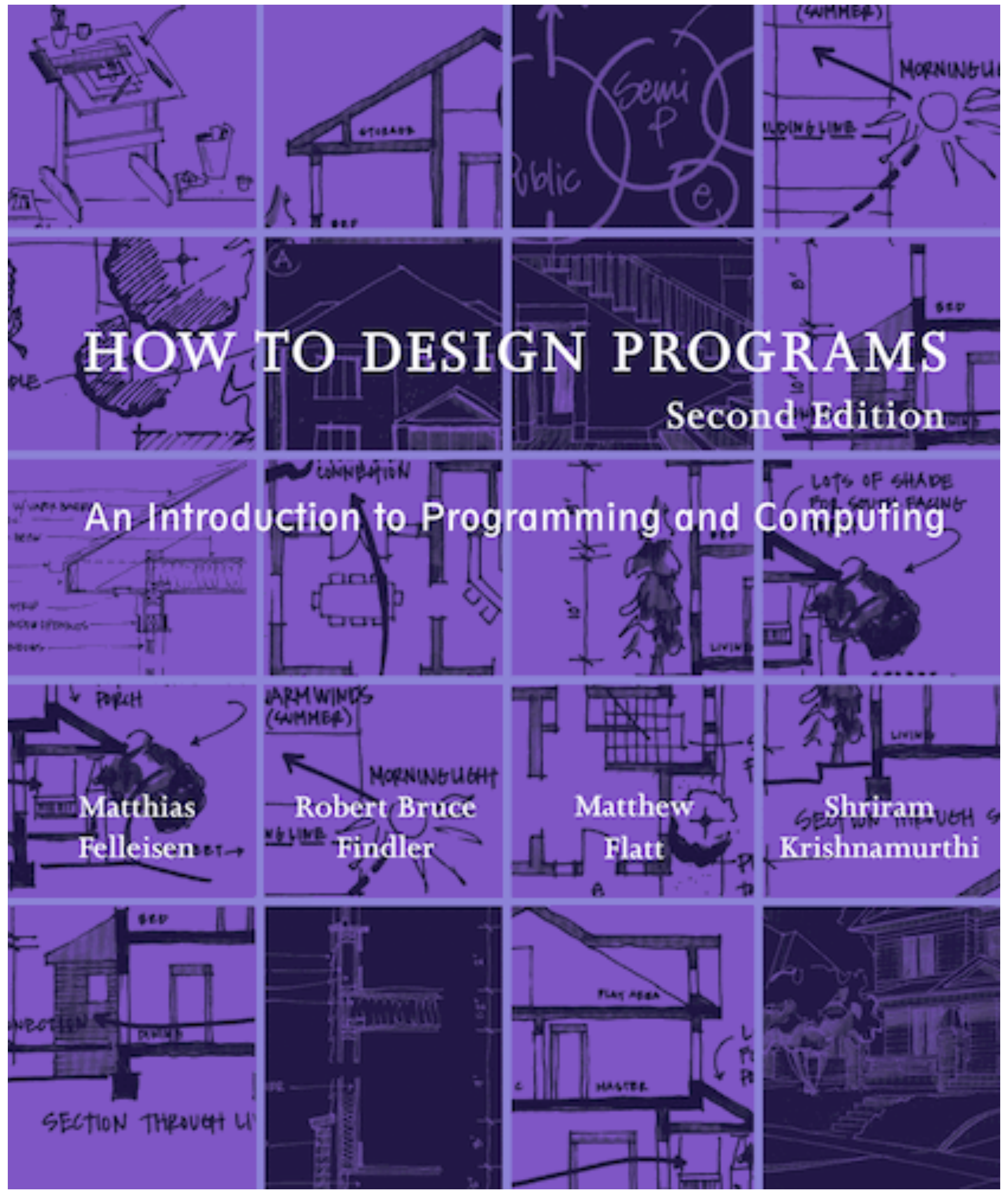
Systematic programming

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 - 7 February 2024

Our first compiler

Announcements

- Assignment 2 out tonight
- a86 Basics quiz out after class: due before class 2/12

Today

- More a86
- Abscond: our first language
- Blackmail: a successor

Some a86 instructions

- Mov
- Add, Sub
- And, Or, Xor, Not
- Sal, Sar
- Label, Jmp, Call, Ret
- Push, Pop
- Cmp
- Je, Jne, Jl, Jle, Jg, Jge
- Cmov*

CMSC 430 - 12 February 2024

Our first compiler

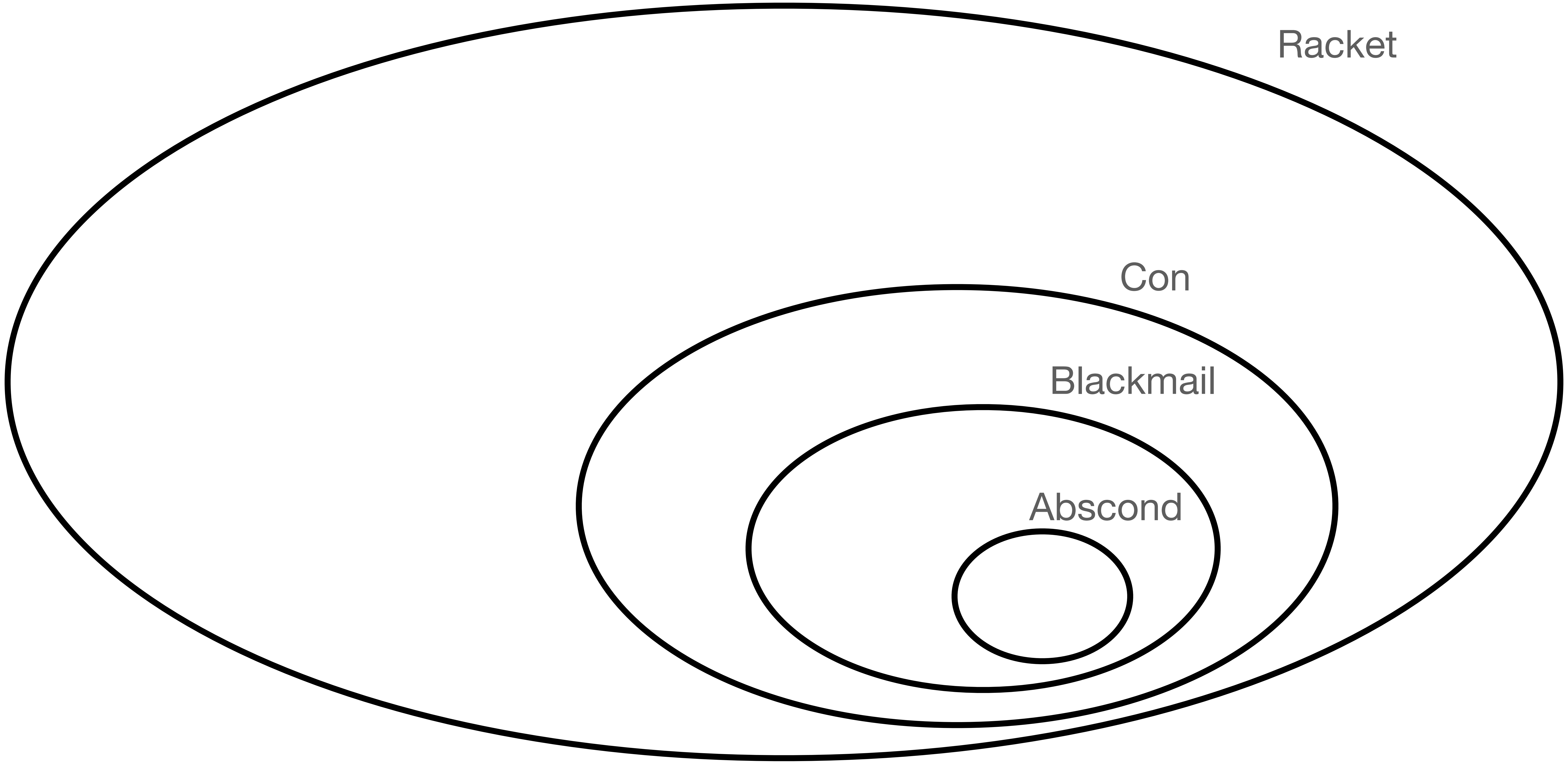
Announcements

- Assignment 2 due Wednesday
- No quiz today

Today

- Abscond: our first language
- Blackmail: a successor

Language subsets



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

CMSC 430 - 14 February 2024

A few more compilers

Announcements

- Assignment 2 due by midnight
- Assignment 3 released tonight (part 1: 1 week; part 2: 2 weeks)
- Quiz out after class

Today

- Interpreters: our approach to specification
- Con: conditional execution
- Dupe: a couple of types, what could go wrong?

Interpreters

One approach to language specification

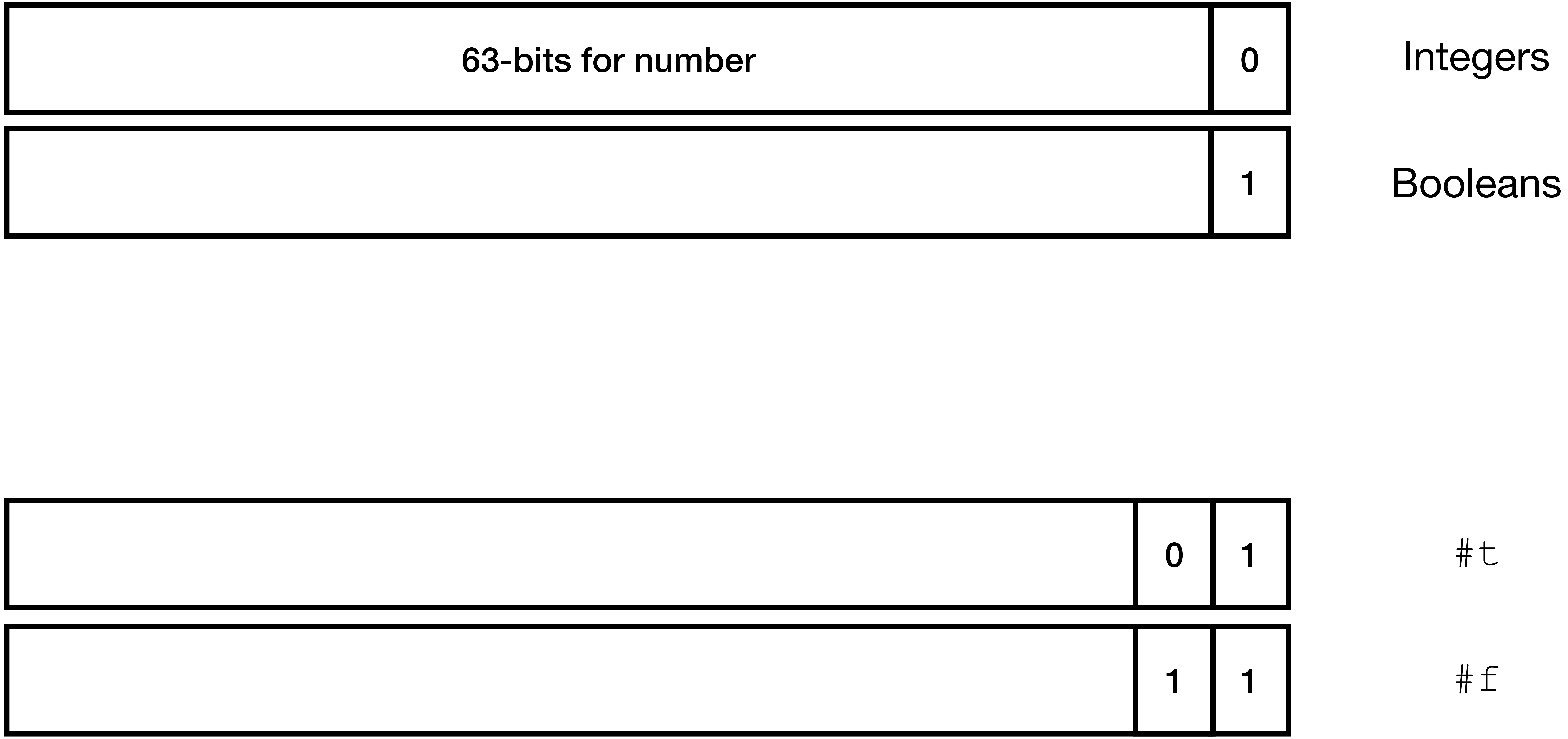
Idea: write a program: `interp : Expr -> Value`

- 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)))
```

Encoding values in Dupe

Type tag in least significant bits



CMSC 430 - 19 February 2024

Representation matters

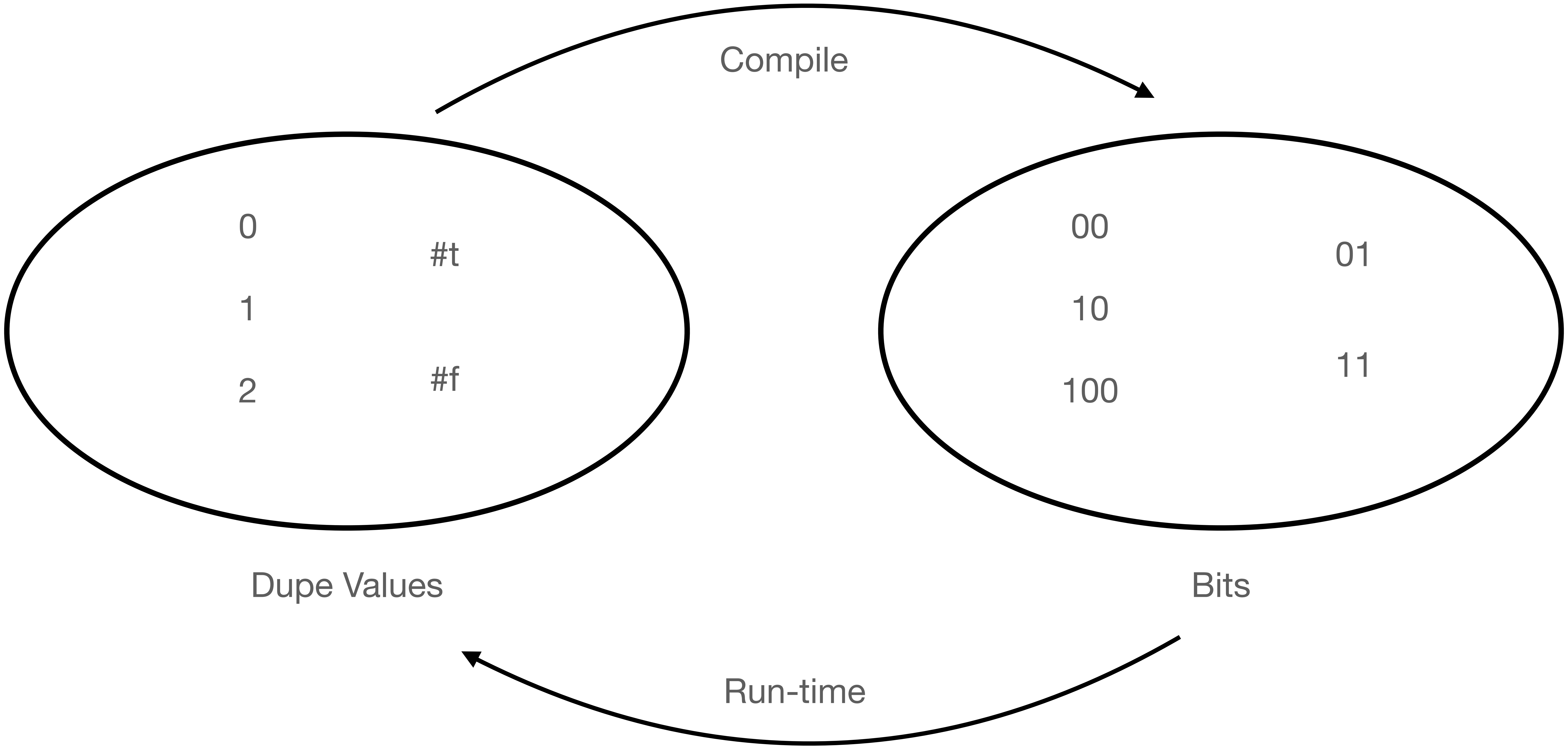
Announcements

- Assignment 3 out (part 1: due Wed; part 2: next Wed)
- Slides on ELMS
- Practice M1 on ELMS and Gradescope
- YAQO today

Today

- Demo Assignment 3
- Dupe: a couple of types, what could go wrong?
- Dodger: more types

Representing Values with Bits in Dupe



Characters

Like the Booleans, but more of them

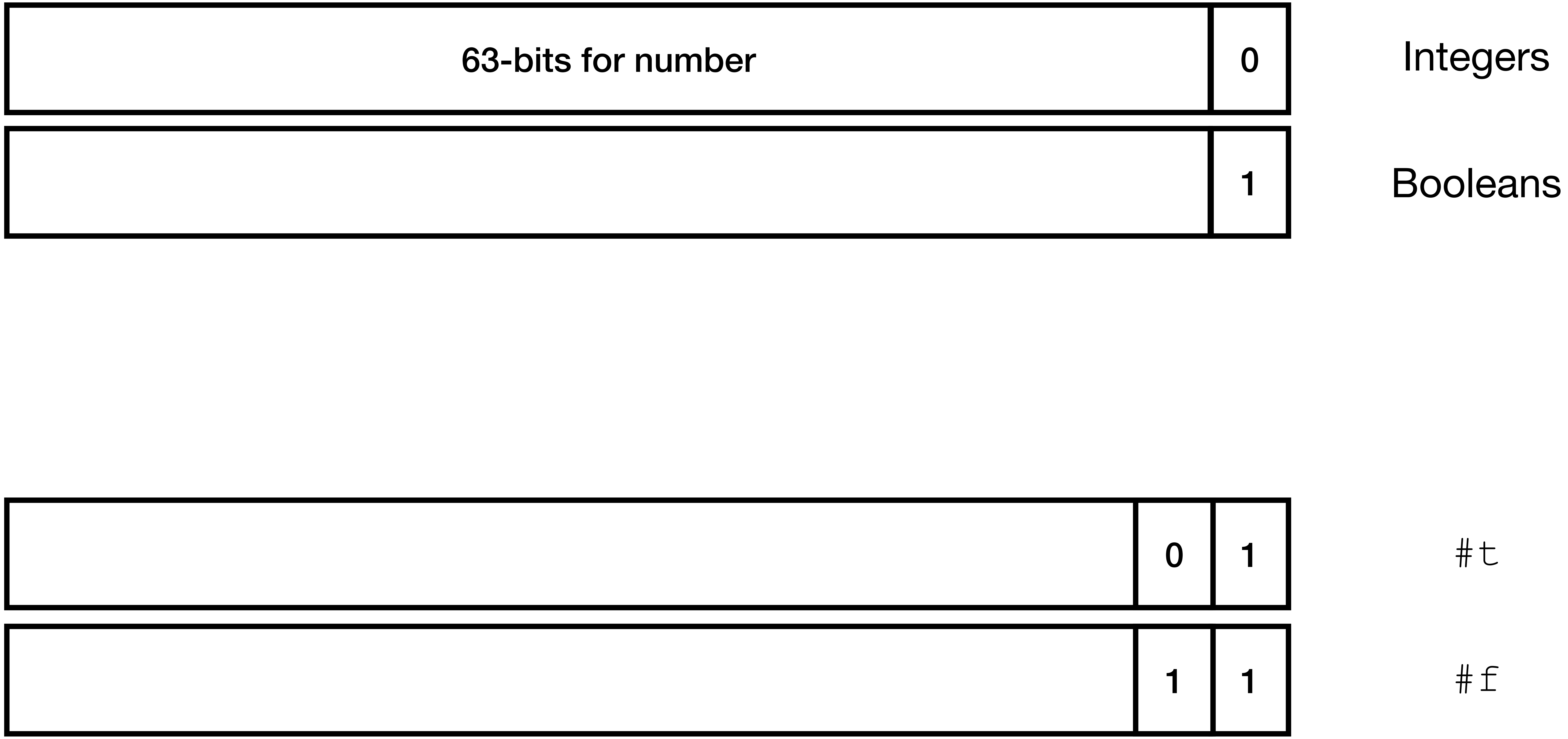
`#\a #\b #\c` … `#\λ #\絳` …

Unicode: roughly 150K characters.

Operations: `char? integer->char char->integer`

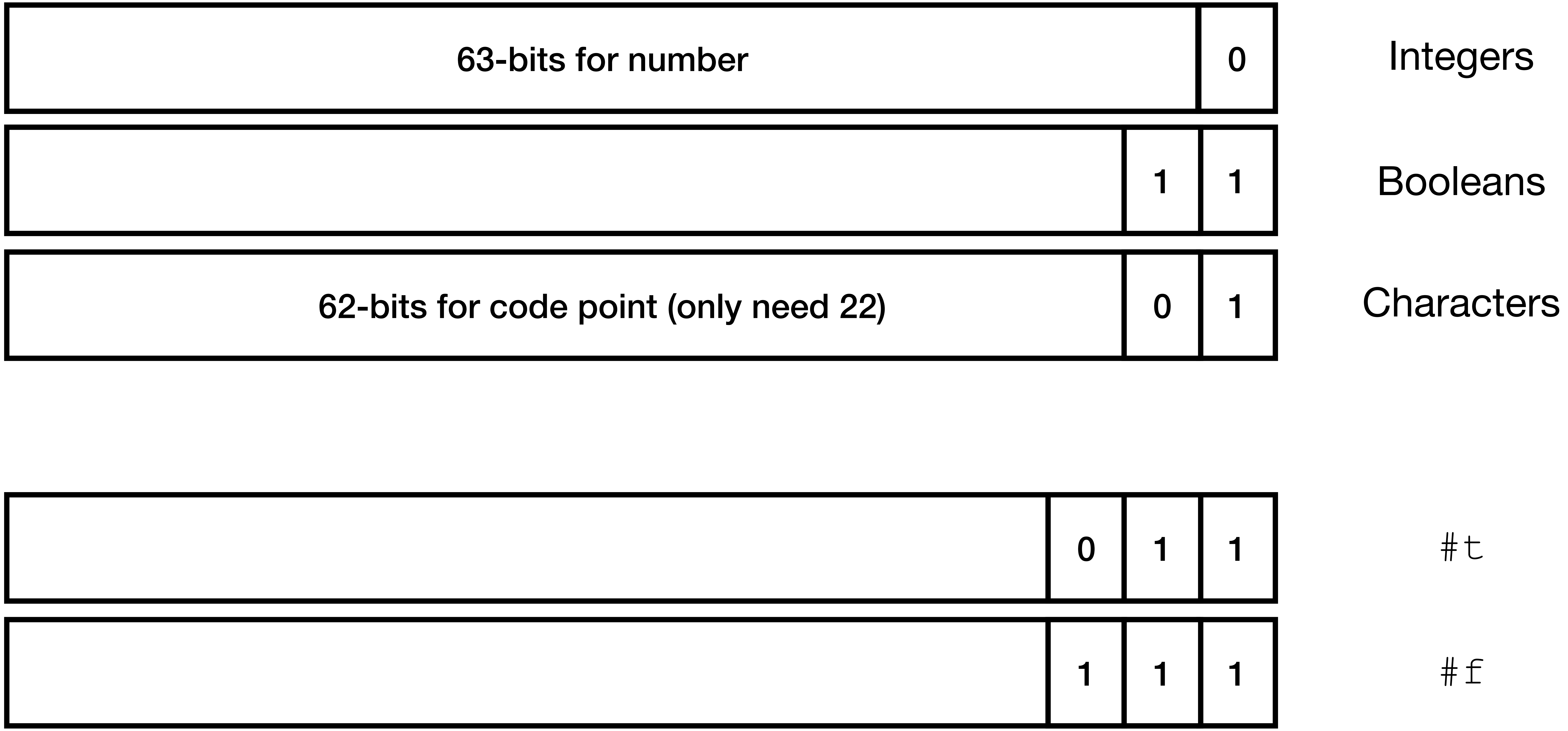
Encoding values in Dupe

Type tag in least significant bits

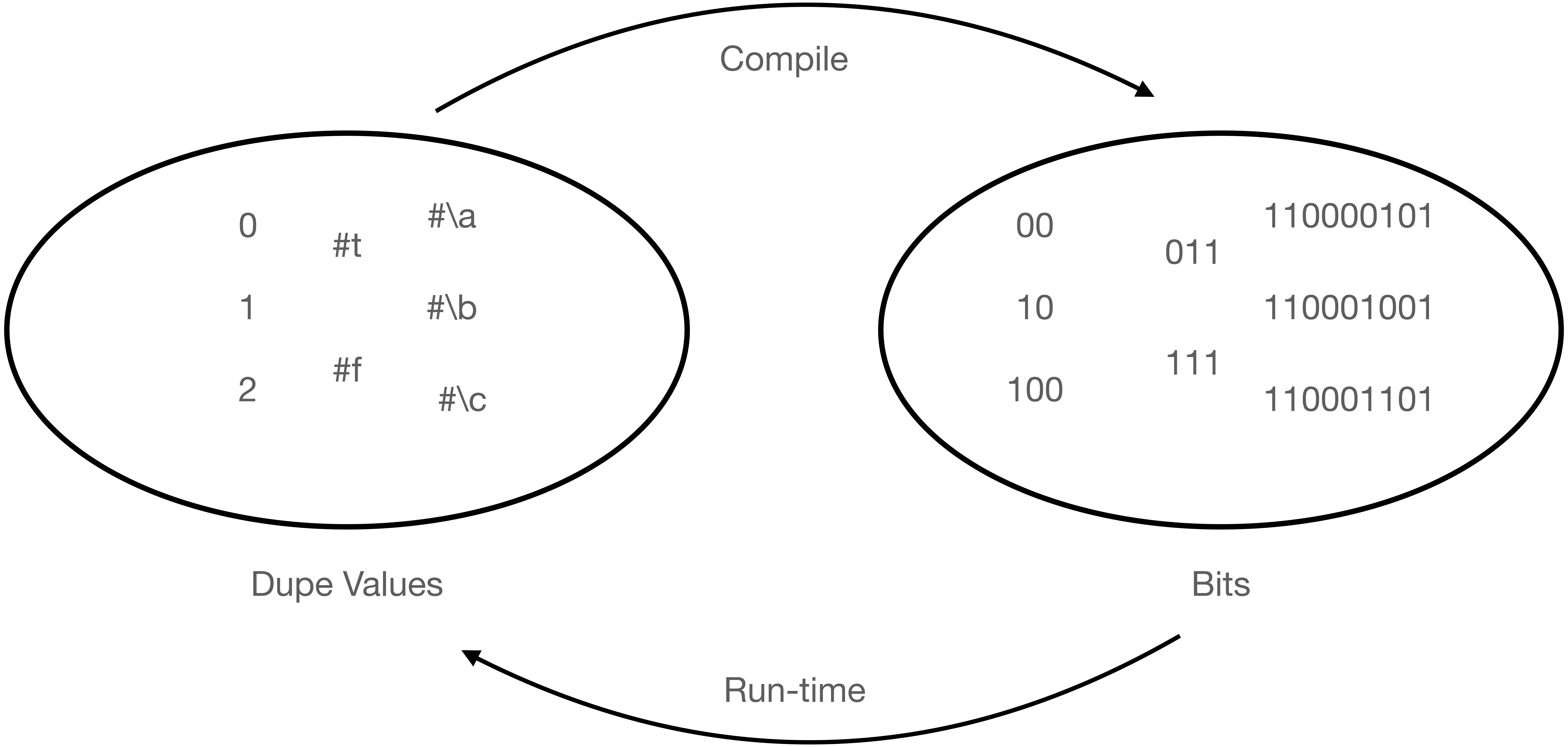


Encoding values in Dodger

Type tag in least significant bits



Representing Values with Bits in Dodger



CMSC 430 - 21 February 2024

Changing the world

Announcements

- Assignment 3 out (part 1: due tonight!; part 2: next Wed)
- Survey on ELMS
- Social tomorrow 4PM in Iribe lobby

Today

- Finish `char?` and Quick look at Dupe and Dodger run-time
- I/O in Evildoer
  - Evildoer run-time system
  - The System V ABI for making calls

Syntactic additions in Evildoer

Concrete syntax	Abstract syntax
`(begin e1 e2)`	`(Begin e1 e2)`
`(read-byte)`	`(Prim0 'read-byte)`
`(peek-byte)`	`(Prim0 'peek-byte)`
`(void)`	`(Prim0 'void)`
`(write-byte e)`	`(Prim1 'write-byte e)`
`(eof-object? 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();
}
```

CMSC 430 - 26 February 2024

Taking errors seriously

Announcements

- Assignment 3, part 2: Wed
- Fixed Dodger and Evildoer source zip files on web page

Today

- Explaining the stack invariant in Evildoer
- Testing programs that do I/O & linking the runtime to `asm-interp`
- Specifying error behaviors in Extort

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?

Specifying Error Behavior

So far we have not specified error behavior

- our specification says:

```
∀ e i . (equal? (run/io (compile e) i)
                (interp/io e i))
```

Whenever interp crashes, the specification holds vacuously.

Solution: update the specification to compute errors as results.

CMSC 430 - 28 February 2024

Taking errors seriously (in compilation)

Announcements

- Midterm 1 — next week (24 hours of Wednesday; no lecture)

Today

- Make examples — a mantra
- Last time: specifying error behaviors in Extort
- This time: implementing error behaviors in the compiler
- Fraud: binary operations and variable bindings

Mantra: make examples

Lots of struggling with `case`.

The problem: translate expression into assembly instructions that carry out the expressions evaluation: `compile-e : Expr -> Asm`

You can't write the function without knowing what it should compute first.

What should it compute? Let's make examples.

Mantra: make examples

A first cut

```
(case 0
  [(1) 4]
  [else 9])
```

```
(Mov ‘rax (value->bits 0))
(Cmp ‘rax (value->bits 1))
(Jne ‘else)
(Mov ‘rax (value->bits 4))
(Jmp ‘end)
(Label ‘else)
(Mov ‘rax (value->bits 9))
(Label ‘end)
```

Mantra: make examples

A little more complicated

```
(case 0
 [(1 2 3) 4]
 [else 9])
```

```
(Mov ‘rax (value->bits 0))
(Cmp ‘rax (value->bits 1))
???
(Jne ‘else)
(Mov ‘rax (value->bits 4))
(Jmp ‘end)
(Label ‘else)
(Mov ‘rax (value->bits 9))
(Label ‘end)
```

Mantra: make examples

A little more complicated

```
(case 0
 [(1 2 3) 4]
 [else 9])
```

```
(Mov 'rax (value->bits 0))
(Cmp 'rax (value->bits 1))
(Je 'rhs)
(Cmp 'rax (value->bits 2))
(Je 'rhs)
(Cmp 'rax (value->bits 3))
(Je 'rhs)
(Jmp 'else)
(Label 'rhs)
(Mov 'rax (value->bits 4))
(Jmp 'end)
(Label 'else)
(Mov 'rax (value->bits 9))
(Label 'end)
```

Mantra: make examples

Simplest example, revisited

```
(case 0
  [(1) 4]
  [else 9])
```

```
(Mov 'rax (value->bits 0))
(Cmp 'rax (value->bits 1))
(Je 'rhs)
(Jmp 'else)
(Label 'rhs)
(Mov 'rax (value->bits 4))
(Jmp 'end)
(Label 'else)
(Mov 'rax (value->bits 9))
(Label 'end)
```

Mantra: make examples

More complicated

```
(case 0
 [(1 2 3) 4]
 [(4 5 6 7) 8]
 [else 9])
```

```
(Mov 'rax (value->bits 0))
(Cmp 'rax (value->bits 1))
(Je 'rhs1)
(Cmp 'rax (value->bits 2))
(Je 'rhs1)
(Cmp 'rax (value->bits 3))
(Je 'rhs1)
(Cmp 'rax (value->bits 4))
(Je 'rhs2)
(Cmp 'rax (value->bits 5))
(Je 'rhs2)
(Cmp 'rax (value->bits 6))
(Je 'rhs2)
(Cmp 'rax (value->bits 7))
(Je 'rhs2)
(Jmp 'else)
```

```
(Label 'rhs1)
(Mov 'rax (value->bits 4))
(Jmp 'end)
(Label 'rhs2)
(Mov 'rax (value->bits 8))
(Jmp 'end)
(Label 'else)
(Mov 'rax (value->bits 9))
(Label 'end)
```

Mantra: make examples

More complicated

```
(case 0
  [(1 2 3) 4]
  [(4 5 6 7) 8]
  [else 9])
```

```
(Mov 'rax (value->bits 0))
(Cmp 'rax (value->bits 1))
(Je 'rhs1)
(Cmp 'rax (value->bits 2))
(Je 'rhs1)
(Cmp 'rax (value->bits 3))
(Je 'rhs1)
(Cmp 'rax (value->bits 4))
(Je 'rhs2)
(Cmp 'rax (value->bits 5))
(Je 'rhs2)
(Cmp 'rax (value->bits 6))
(Je 'rhs2)
(Cmp 'rax (value->bits 7))
(Je 'rhs2)
(Jmp 'else)
```

```
(Label 'rhs1)
(Mov 'rax (value->bits 4))
(Jmp 'end)
(Label 'rhs2)
(Mov 'rax (value->bits 8))
(Jmp 'end)
(Label 'else)
(Mov 'rax (value->bits 9))
(Label 'end)
```

This is just a sketch, there may be further issues to consider…

CMSC 430 - 4 March 2024

Variable bindings and lexical addresses

Announcements

- Midterm 1 — On Wednesday; no lecture
- Slides updated on ELMS

Today

- More on interpreting variable bindings and variable references
- Observing an invariant about environments
- Compiling Fraud

Interpreting variables and bindings

Environments tell us the meaning of variables

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

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

;; Expr -> Answer
(define (interp e)
  (interp-env e '()))

;; Expr Env -> Answer
(define (interp-env e r)
  (match e
    [(Var x) (lookup r x)]
    [(Let x e1 e2)
     (match (interp-env e1 r)
       ['err 'err]
       [v (interp-env 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.

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

;; Expr -> Answer
(define (interp e)
  (interp-env e '()))

;; Expr Env -> Answer
(define (interp-env e r)
  (match e
    [(Var x) (lookup r x)]
    [(Let x e1 e2)
     (match (interp-env e1 r)
       ['err 'err]
       [v (interp-env e2 (ext r x v))])]
    #;...))
```

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

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-env (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 - 11 March 2024
Compiling Fraud

Announcements

- Post-midterm 1 survey due by start of class Wed
- Quiz on compiling Fraud due by start of class Wed
- Assignment 4 out tonight

Today

- Compiling Fraud
- Dynamically aligning the stack

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

`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).

CMSC 430 - 25 March 2024

Inductive data and memory allocation

Announcements

- Assignment 4: due dates pushed back for both parts
  - Part 1: Wednesday
  - Part 2: April 8
- Midterm grades still in progress (soon!)

Today

- Hustle: heaps and lists
  - inductive data
  - allocating memory

How to represent pointers?

Addressing memory

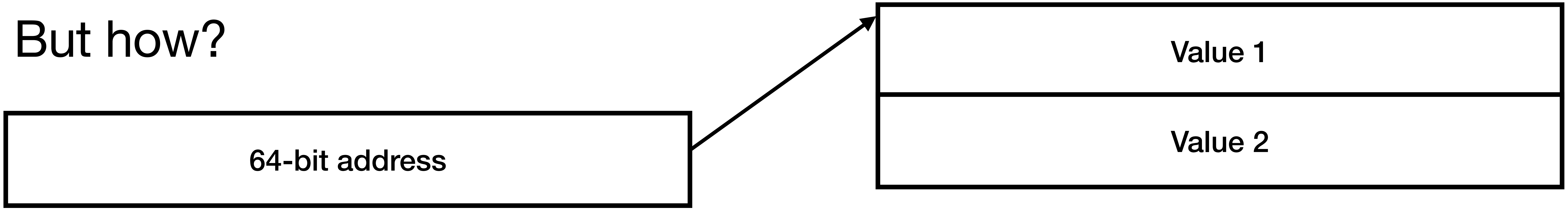
Basic idea:

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

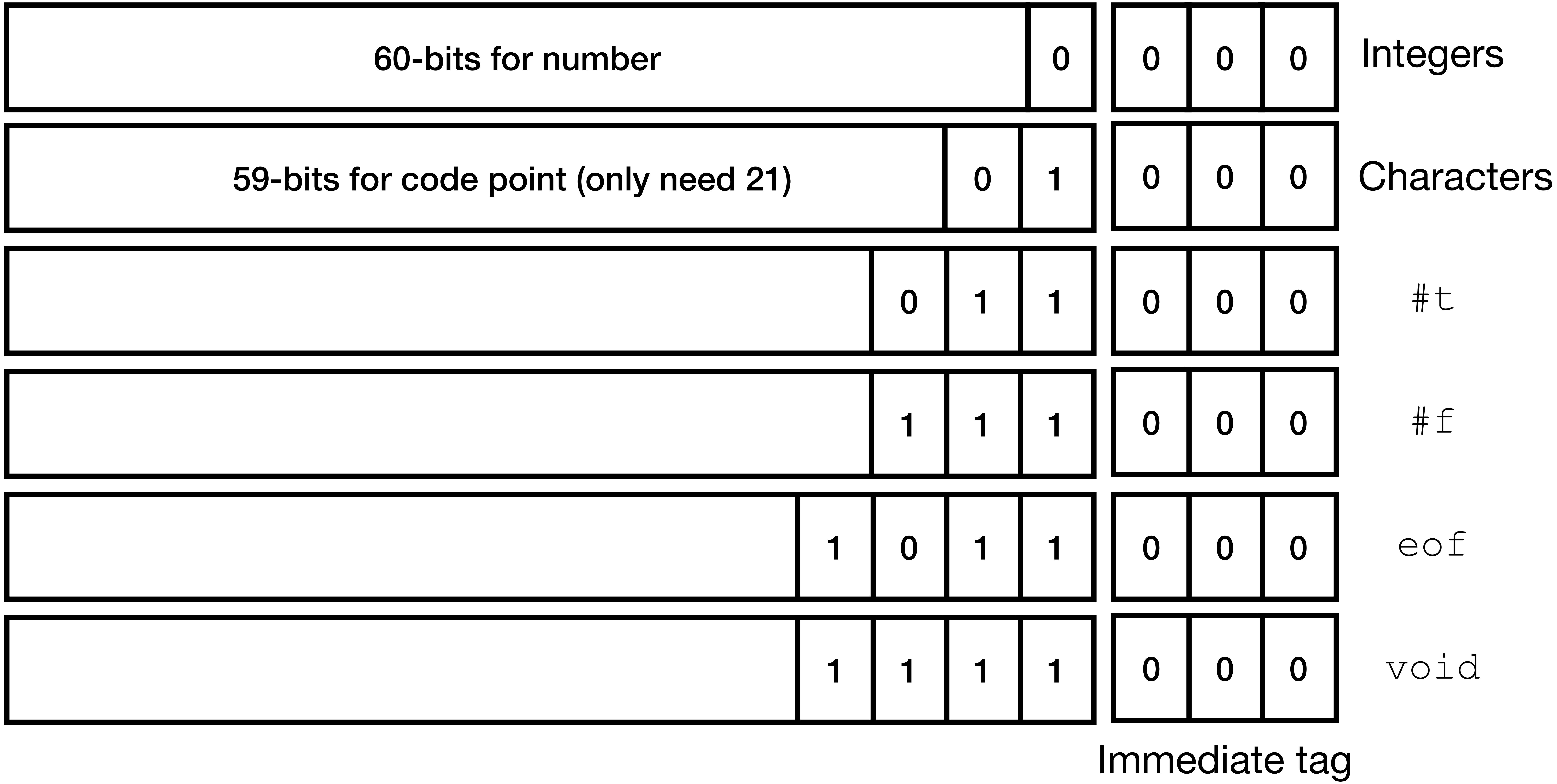
But how?



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

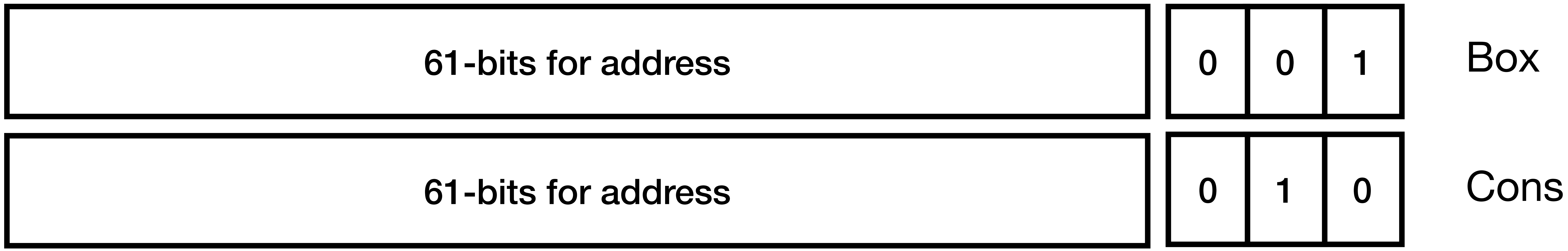
Encoding immediate values (Hustle)

Type tag in least significant bits



Encoding pointer values (Hustle)

Type tag in least significant 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

CMSC 430 - 27 March 2024

Inductive data and memory allocation

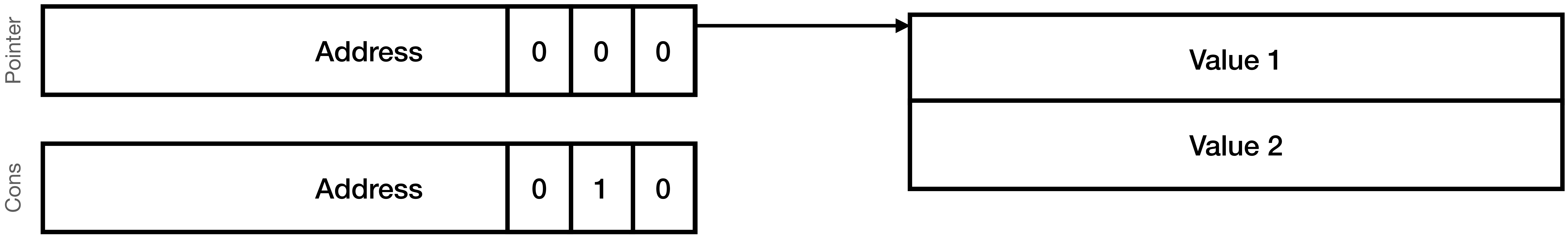
Announcements

- Assignment 4: Part 1: tonight! Part 2: April 8
- Part 2 autograder released
- Hustle quiz due by next class

Today

- Hustle: pairs and boxes
- Hoax: array data: vectors and strings

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

Our pointers always end in #b000 because we allocate in multiples of 8 bytes, so take advantage of this and stash type tag there (no shifting).

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

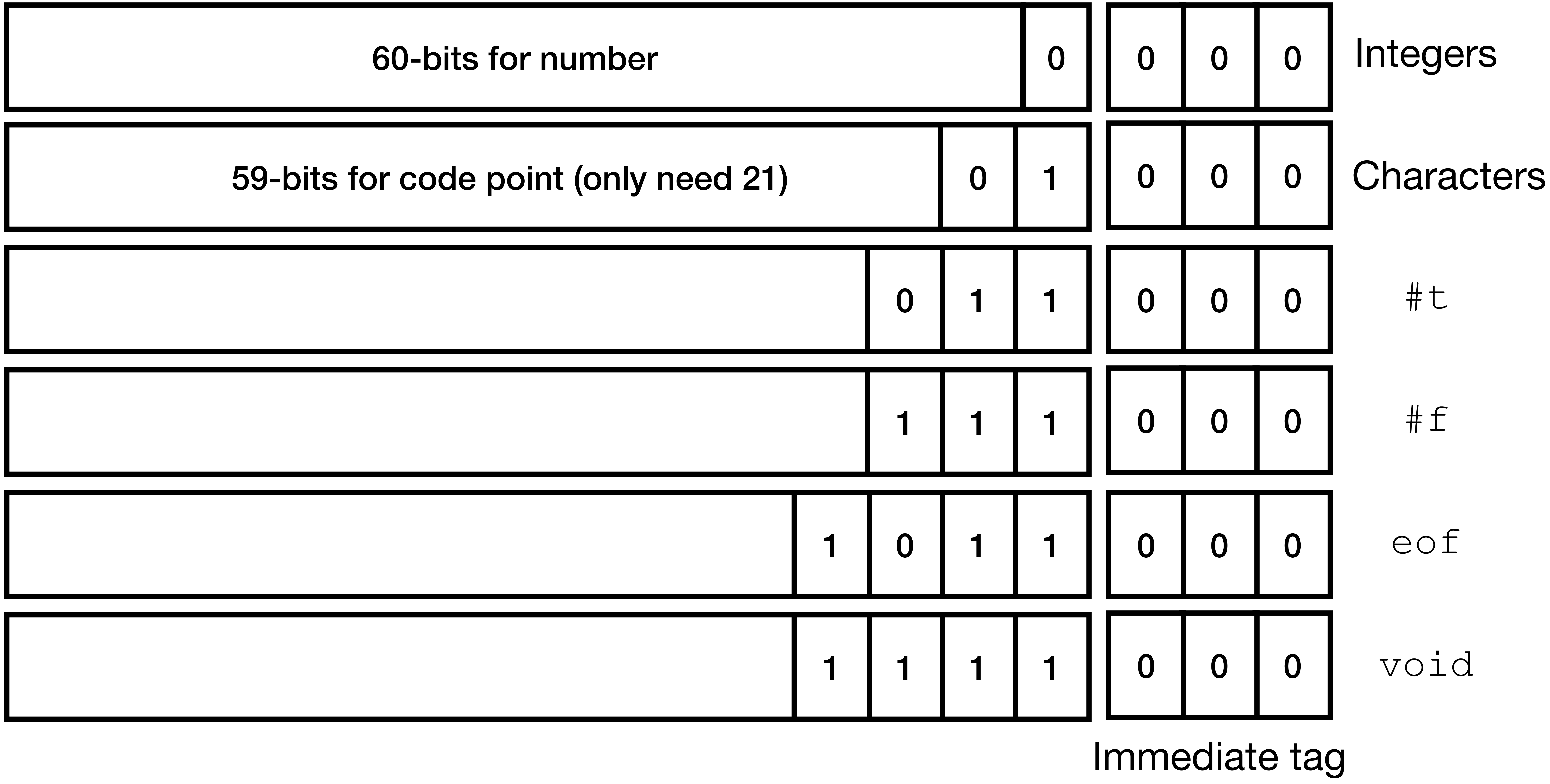
Encoding pointer values (Hustle)

Type tag in least significant bits

61-bits for address	0	0	1	Box
61-bits for address	0	1	0	Cons

Encoding immediate values (Hustle)

Type tag in least significant bits



CMSC 430 - 1 April 2024

Array types and memory allocation

Today

- Hoax: array data:
  - vectors: heterogenous arrays
  - strings: homogenous arrays

Invariants (Hoax)

Various facts about the Hoax compiler

Registers:

- `rax` - return value
- `rsp` - stack pointer
- `rdi` - first param when calling run-time system
- `rbx` - heap pointer

`(compile-e e)`

- 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

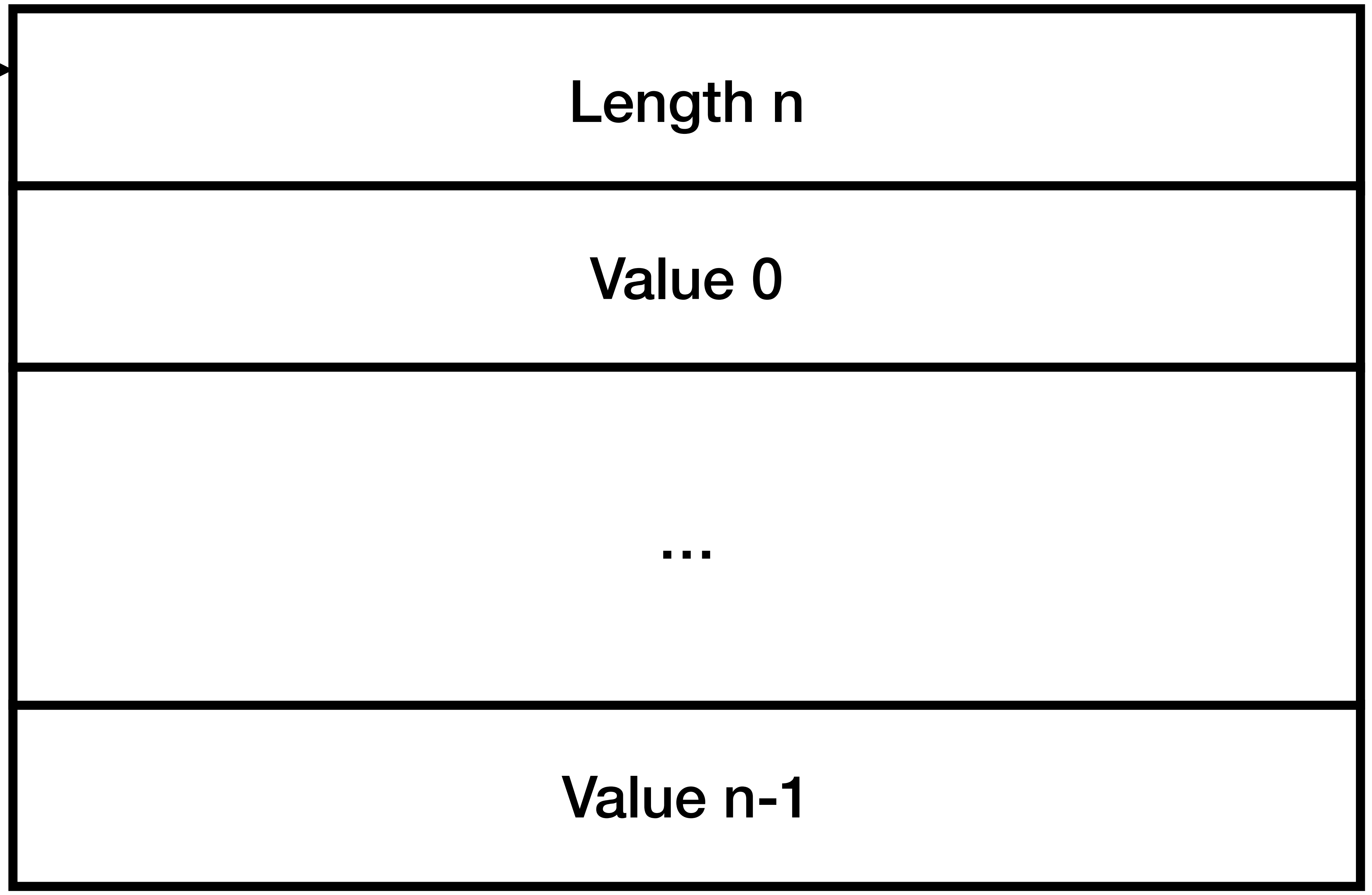
Encoding pointer values (Hoax)

Type tag in least significant bits

61-bits for address	0	0	1	Box
61-bits for address	0	1	0	Cons
61-bits for address	0	1	1	Vector
61-bits for address	1	0	0	String

How to represent vectors?

Sized heterogeneous arrays



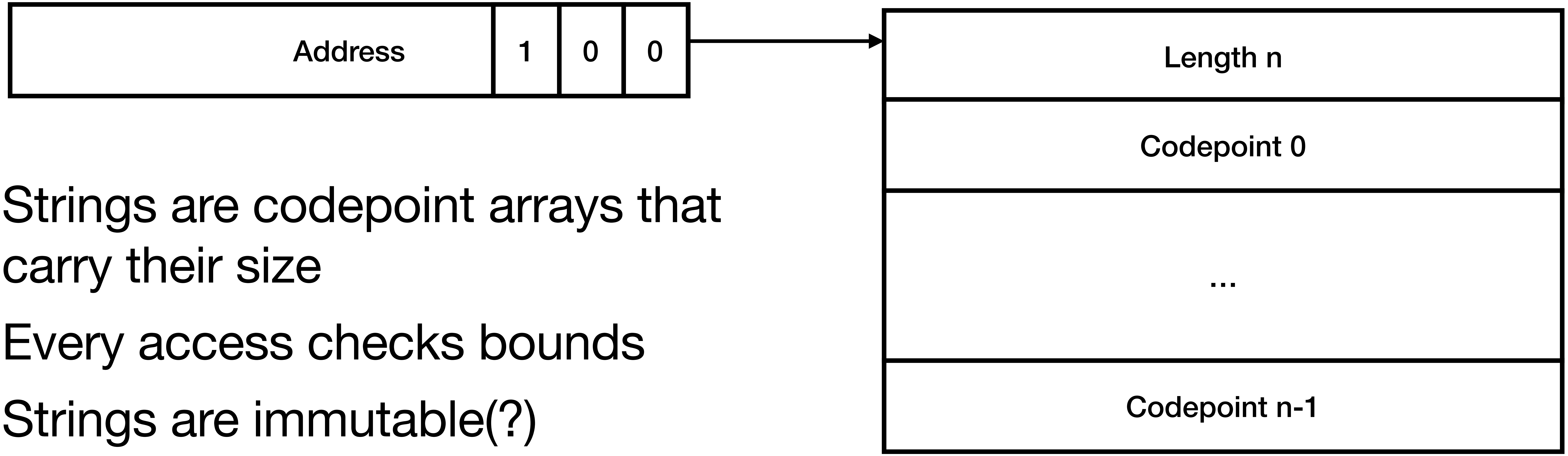
Vectors are arrays that carry their size

Every access checks bounds

Vectors are mutable

How to represent strings?

Sized homogeneous arrays



Strings are codepoint arrays that carry their size

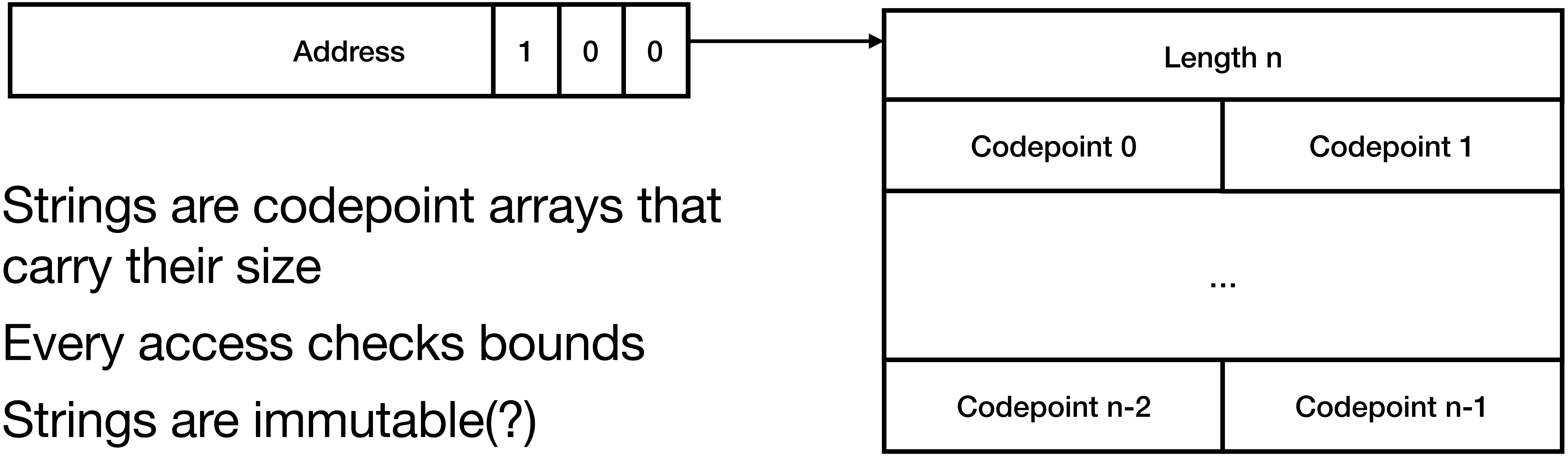
Every access checks bounds

Strings are immutable(?)

Codepoints are only 21-bits wide so this is wasteful.

How to represent strings?

Sized homogeneous arrays



Strings are codepoint arrays that carry their size

Every access checks bounds

Strings are immutable(?)

CMSC 430 - 3 April 2024
Array types and memory allocation

Announcements

- Assignment 4: Part 2: April 8, Monday

Today

- Finishing off compilation of vectors and strings
- Addressing issues with “the” empty vector / string
- Statically allocated data
  - Data sections
  - Data pseudo instructions
  - Lea instruction

CMSC 430 - 8 April 2024

Interpreting function definitions and calls

Announcements

- Assignment 5 out Wednesday

Today

- Iniquity: function definitions and calls
  - Syntax and semantics

CMSC 430 - 10 April 2024

Compiling function calls and definitions

Announcements

- Midterm 1 grades on Gradescope (read Piazza post)
- Assignment 5 out tonight
- Midterm 2 - next Wednesday
- Practice Midterm 2 out tonight

Today

- Inductive data + recursive functions = real computing power
- Quick review of syntax
- Compiling function definitions
- Devising our own calling convention
- Time permitting: Tail calls

Designing our own calling convention

Function calls are like "`let` at a distance"

```
(f 3 4)
```

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

is like

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

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

Designing our own calling convention

A first attempt (doesn't work)

```
(f 3 4)
```

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

Idea: arguments passed on the stack,

return point after arguments,

caller pushes and pops

(Push 3)

(Push 4)

(Call ‘f)

(Pop)

(Pop)

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))

(Ret)

Designing our own calling convention

Same thing without Call (still doesn't work)

```
(f 3 4)
```

(Push 3)

(Push 4)

(Lea ‘rax ‘r)

(Push ‘rax)

(Jmp ‘f)

(Label ‘r)

(Pop)

(Pop)

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

(Label ‘f)

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

Idea: arguments passed on the stack,

return point after arguments,

caller pushes and pops

Designing our own calling convention

Return point before arguments (still doesn't work)

```
(f 3 4)
```

(Lea ‘rax ‘r)

(Push ‘rax)

(Push 3)

(Push 4)

(Jmp ‘f)

(Label ‘r)

(Pop)

(Pop)

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

(Label ‘f)

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

Idea: arguments passed on the stack,

return point *before* arguments,

caller pushes and pops

Designing our own calling convention

Return point before arguments (works!)

```
(f 3 4)
```

(Lea ‘rax ‘r)

(Push ‘rax)

(Push 3)

(Push 4)

(Jmp ‘f)

(Label ‘r)

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

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))

(Pop)

(Pop)

(Ret)

Idea: arguments passed on the stack,

return point *before* arguments,

caller pushes, *callee pops*

CMSC 430 - 15 April 2024
Pattern Matching

Announcements

- Assignment 5 didn't go out, but will soon (with 2 weeks to complete)
- Midterm 2 - Wednesday
- Practice Midterm 2 autograder problem: resolved very soon
- Slides updated on ELMS right after class
- April 29: lecture will be recorded video on ELMS

Today

- On hold: Tail calls (Jig)
- Instead: pattern matching (Knock): syntax, semantics, compilation

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))
```

CMSC 430 - 22 April 2024

Compiling Pattern Matching

Announcements

- Assignment 5 out tonight, 2 weeks
- Final project out soon, due by end of semester
- Post Midterm 2 survey due by Wednesday

Today

- Compiling pattern matching
- Tail calls (time permitting)

CMSC 430 - 24 April 2024
Tail calls

Announcements

- Assignment 5 autograder out
- Final project out soon, due by end of semester
- Quiz on tail calls due by start of class Monday
- Reminder: Monday’s lecture will be a video recording on ELMS

Review: function calls

- how are function definitions compiled?
- how are calls compiled?

Tail positions

Compiling tail calls

Calls

```
(compile-app ‘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))
```

Calls

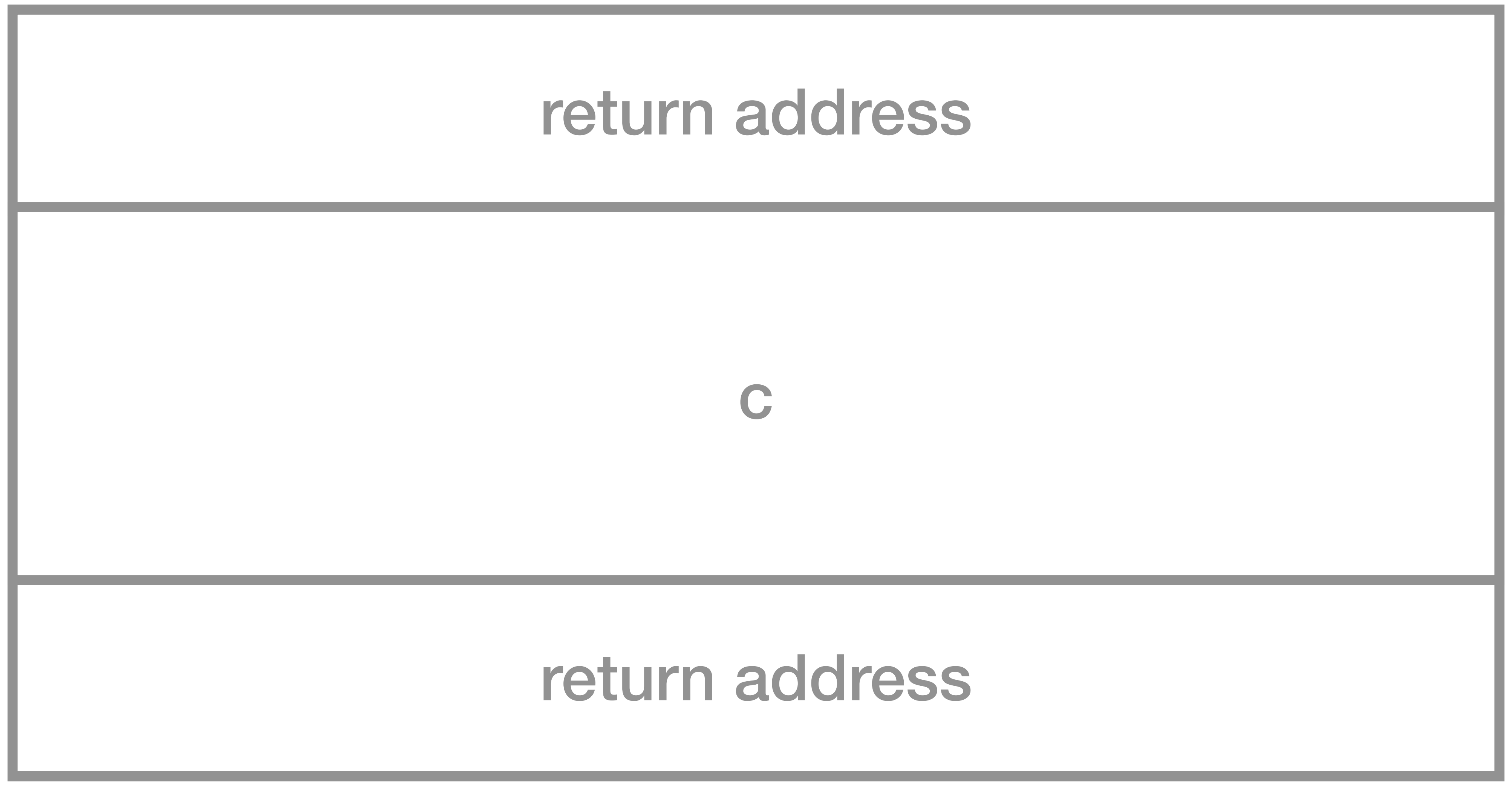
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

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

Calls

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

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

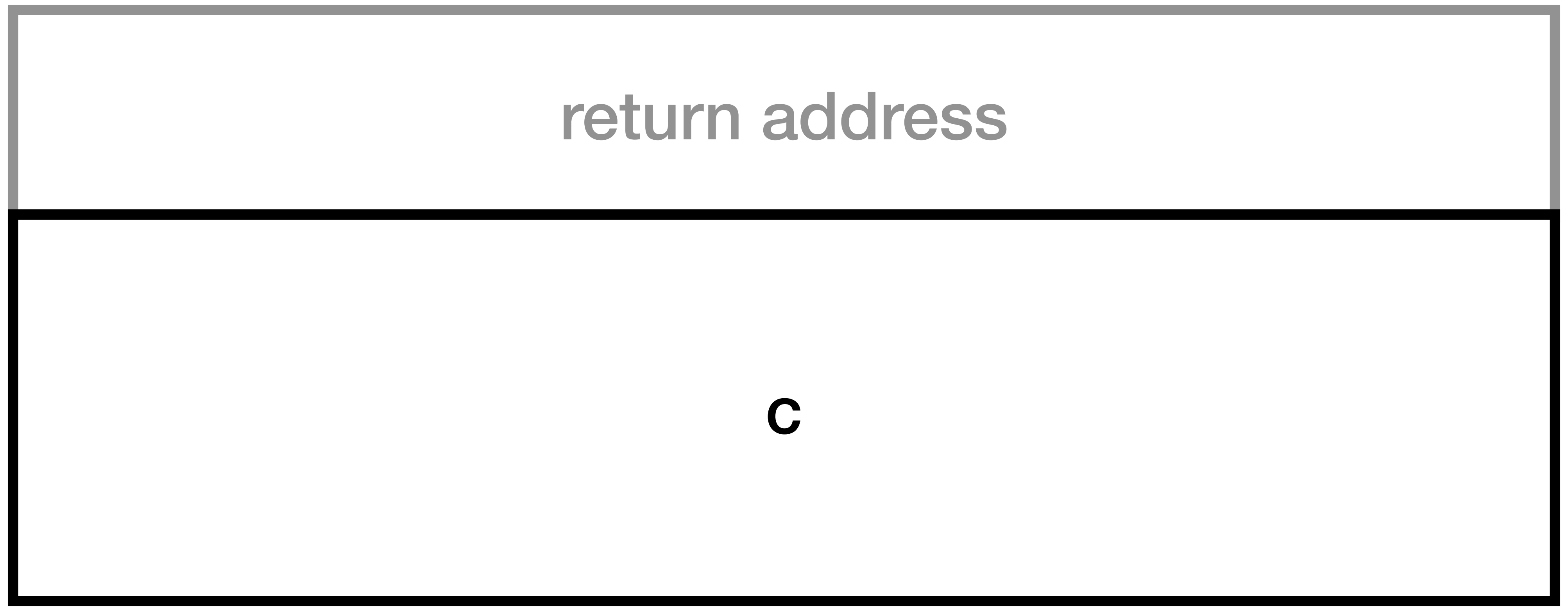
Calls

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

(seq
  (Label ‘f)
  (compile-e e (list ‘z ‘y ‘x) #t)
  (Add ‘rsp 24)
  (Ret))
```



Calls

```
(compile-app '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

Not all calls need to return!

Observation:

- sometimes the result of a call is the result of the surrounding expression

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

(f 100)
```

Tail calls

Not all calls need to return!

return address a
100

```
(define (f n)
  (if (zero? n)
      0
         b
      (f (sub1 n))))

   a
(f 100)
```

Tail calls

Not all calls need to return!

return address a
100
return address b
99

```
(define (f n)
  (if (zero? n)
      0
        b
      (f (sub1 n))))

    a
(f 100)
```

Tail calls

Not all calls need to return!

return address a
100
return address b
99
return address b
98

```
(define (f n)
  (if (zero? n)
      0
         b
      (f (sub1 n))))

    a
(f 100)
```

Tail calls

Not all calls need to return!

return address a
100
return address b
99
return address b
98
⋮
return address b
0

```
(define (f n)
  (if (zero? n)
      0
        b
      (f (sub1 n))))

   a
(f 100)
```

Tail calls don't grow the stack

Don't return to caller, return to the caller's caller

return address a
100

```
(define (f n)
  (if (zero? n)
      0
        b
      (f (sub1 n))))

   a
(f 100)
```

Tail calls don't grow the stack

Don't return to caller, return to the caller's caller

return address a
99

```
(define (f n)
  (if (zero? n)
      0
         b
      (f (sub1 n))))

   a
(f 100)
```

Tail calls don't grow the stack

Don't return to caller, return to the caller's caller

return address a
98

```
(define (f n)
  (if (zero? n)
      0
         b
      (f (sub1 n))))

    a
(f 100)
```

Tail calls don’t grow the stack

Don’t return to caller, return to the caller’s caller

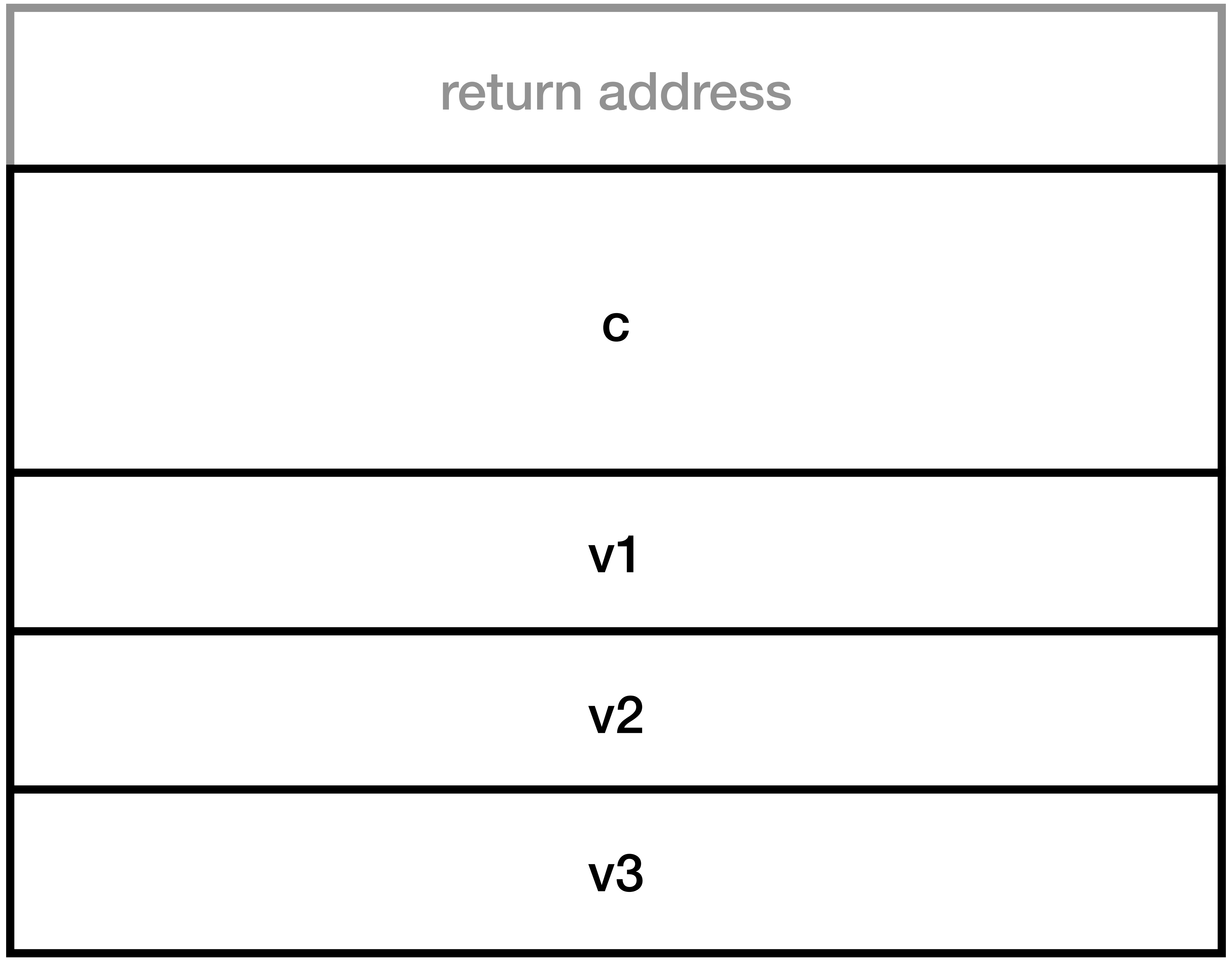
return address a
0

```
(define (f n)
  (if (zero? n)
      0
        b
      (f (sub1 n))))

   a
(f 100)
```

Tail calls (first attempt)

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

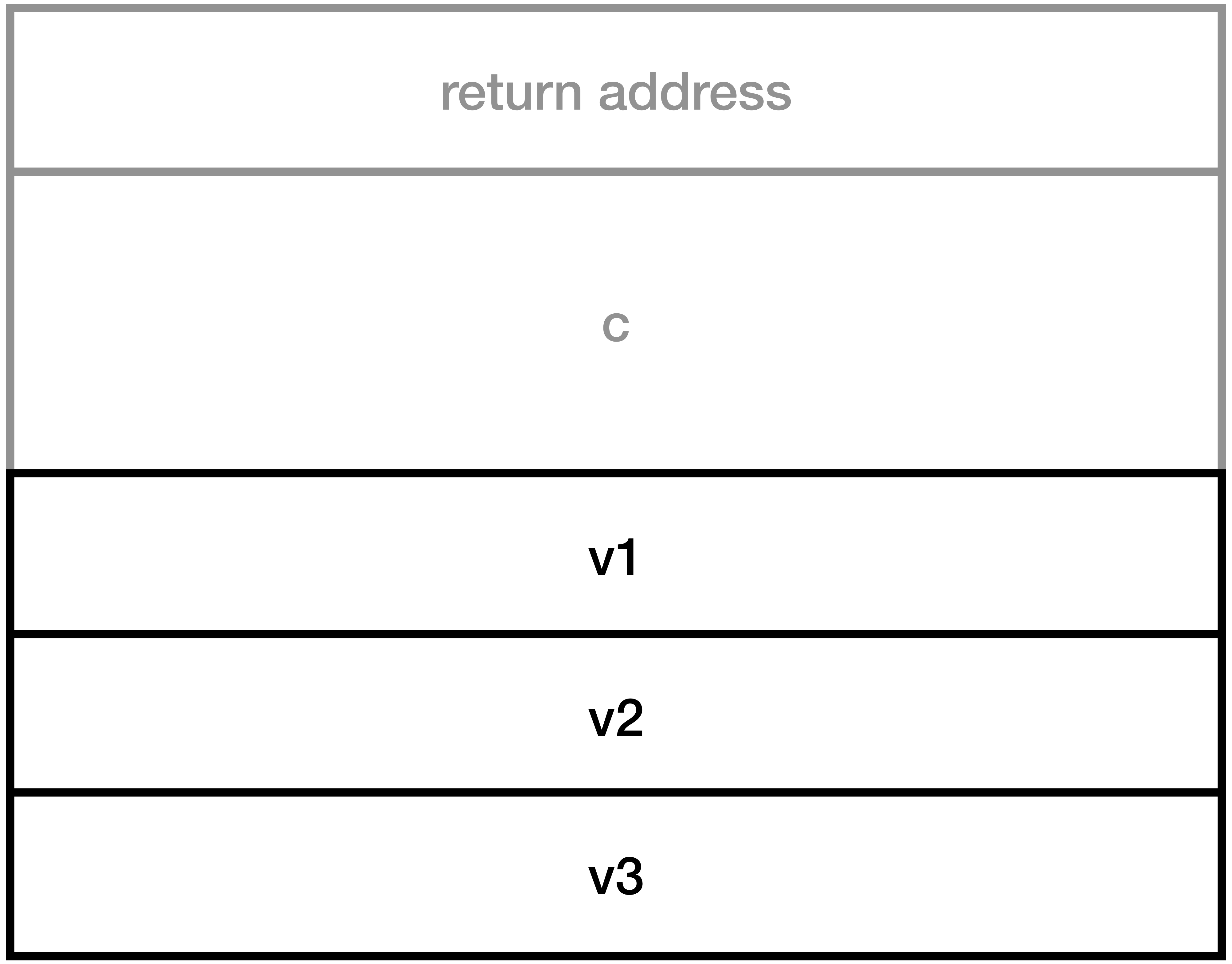
```
(compile-app-tail 'f (list e1 e2 e3)) c)

(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

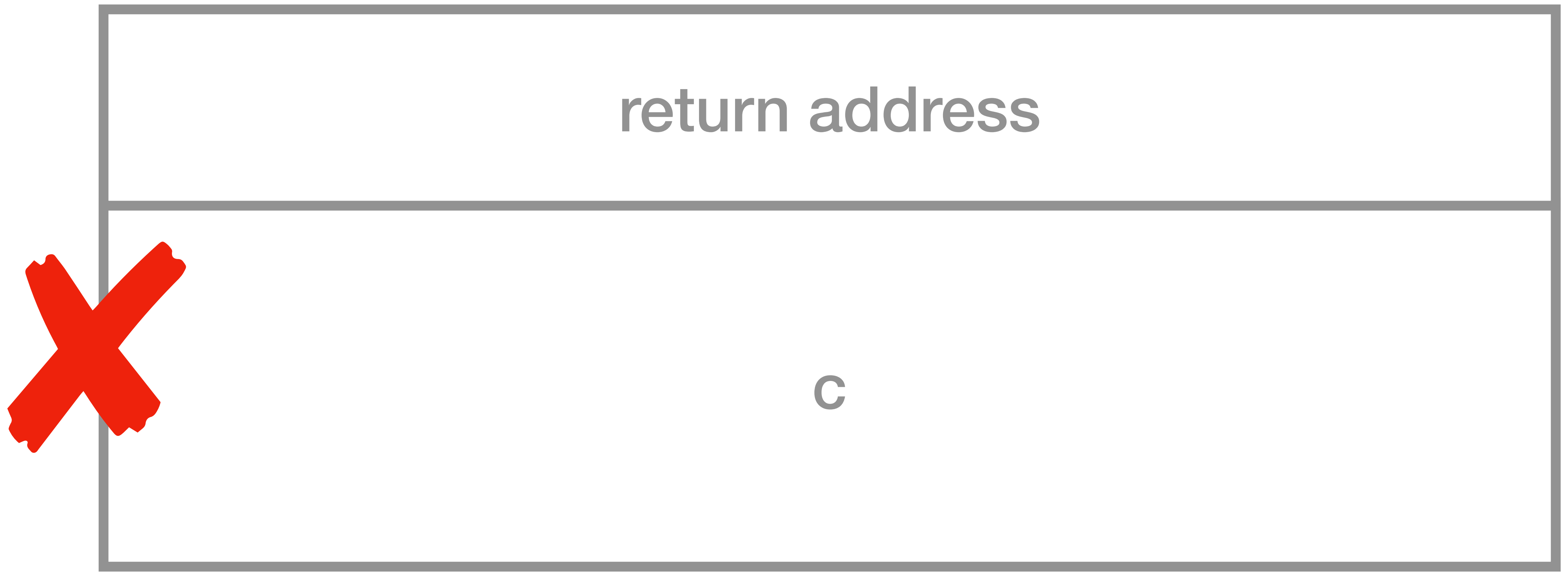
```
(compile-define (Defn ‘f (list ‘x ‘y ‘z)) e))

(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

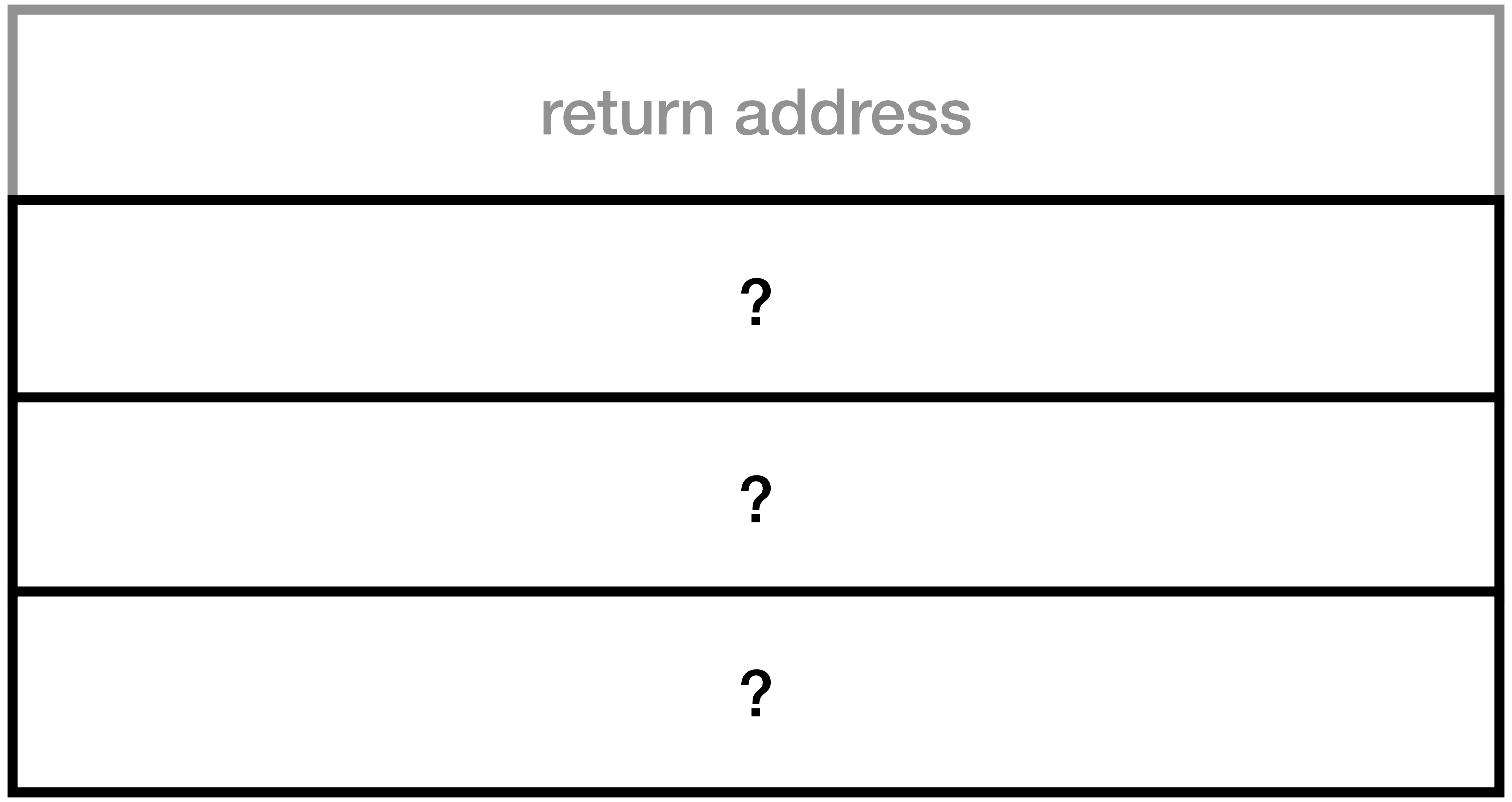
```
(compile-app-tail 'f (list e1 e2 e3)) c)

(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))
```

return address
v1
v2
v3

Tail calls (third attempt)

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

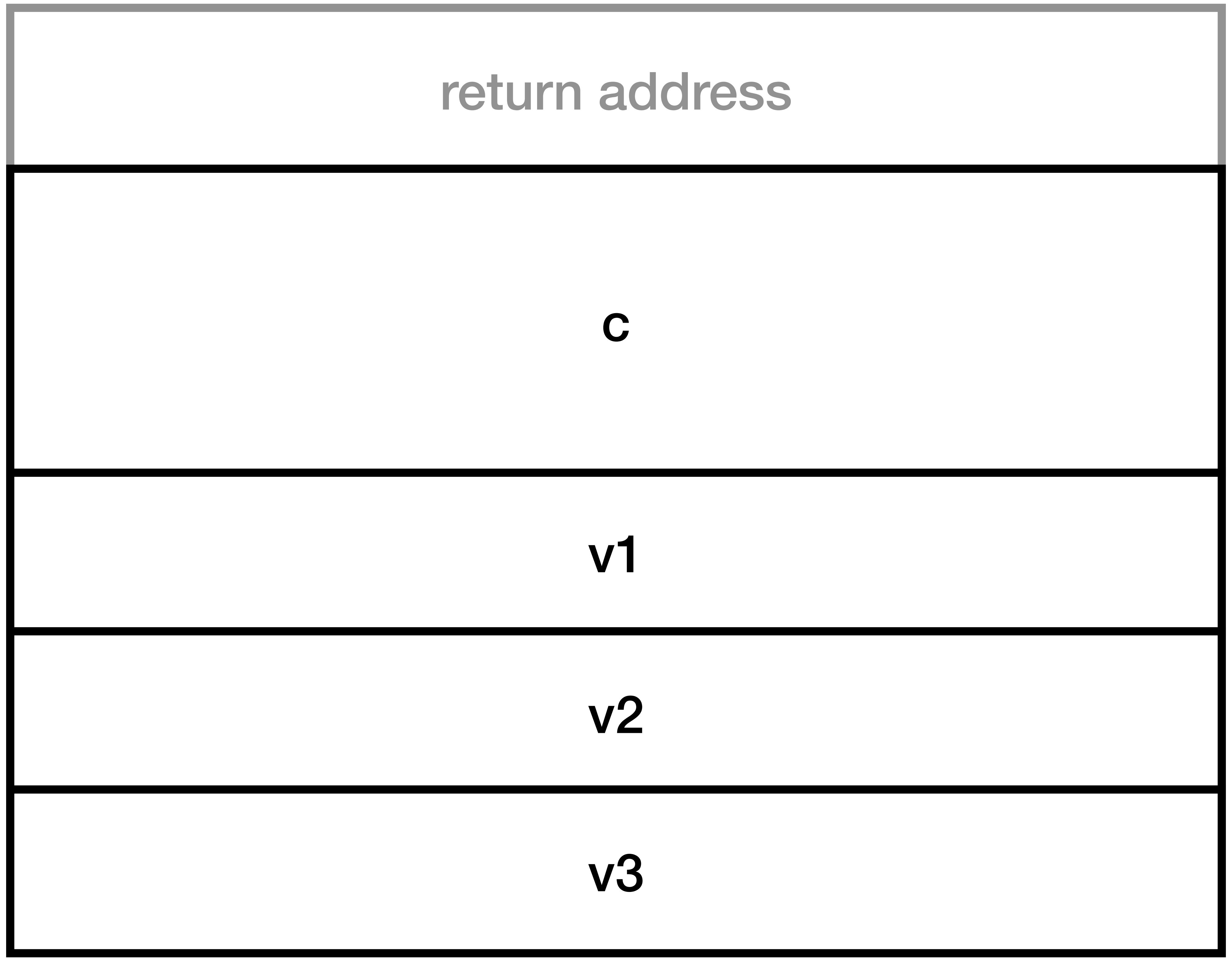
```
(compile-app-tail ‘f (list e1 e2 e3)) c)

(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

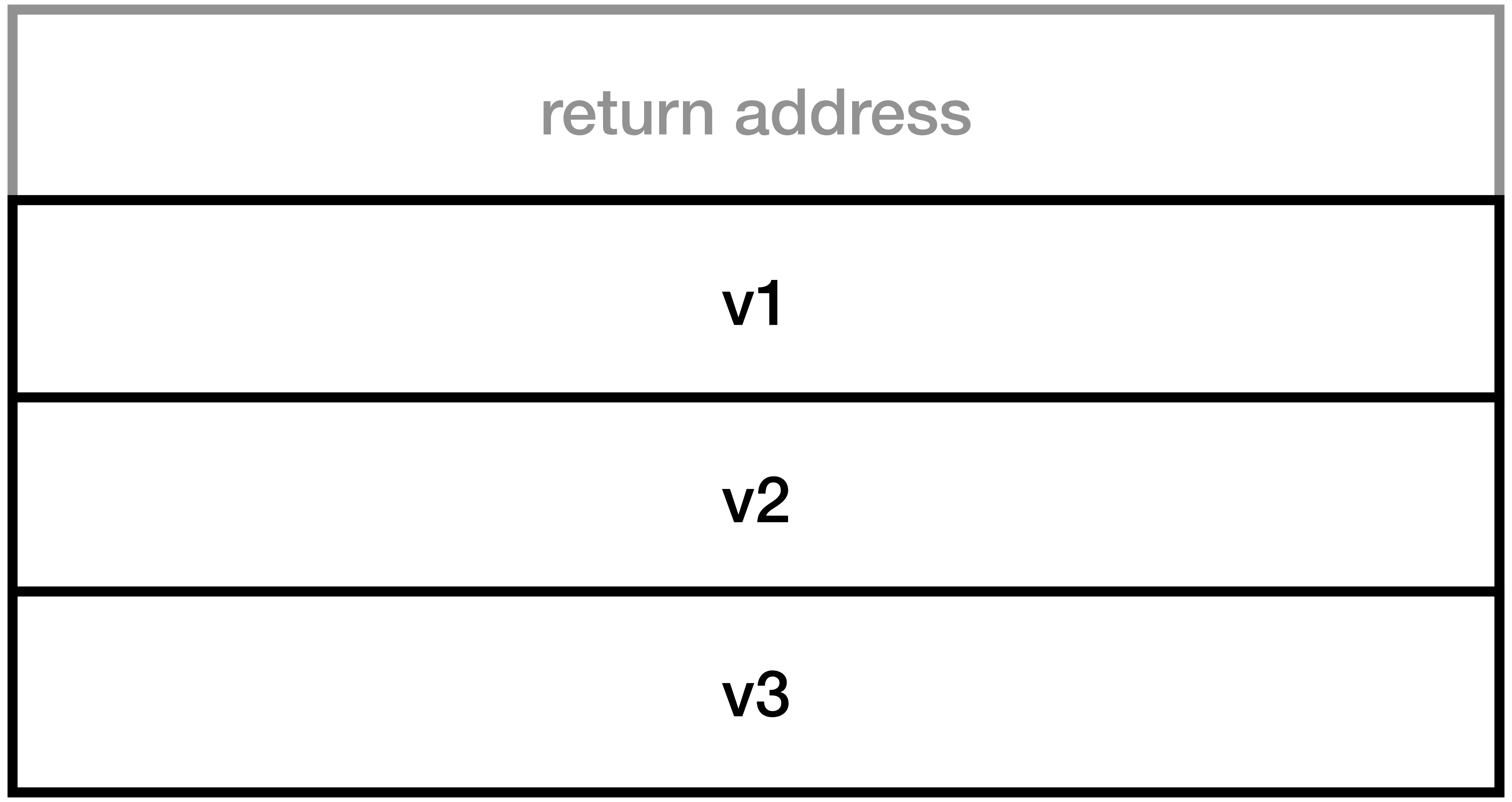
```
(compile-app-tail ‘f (list e1 e2 e3)) c)

(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

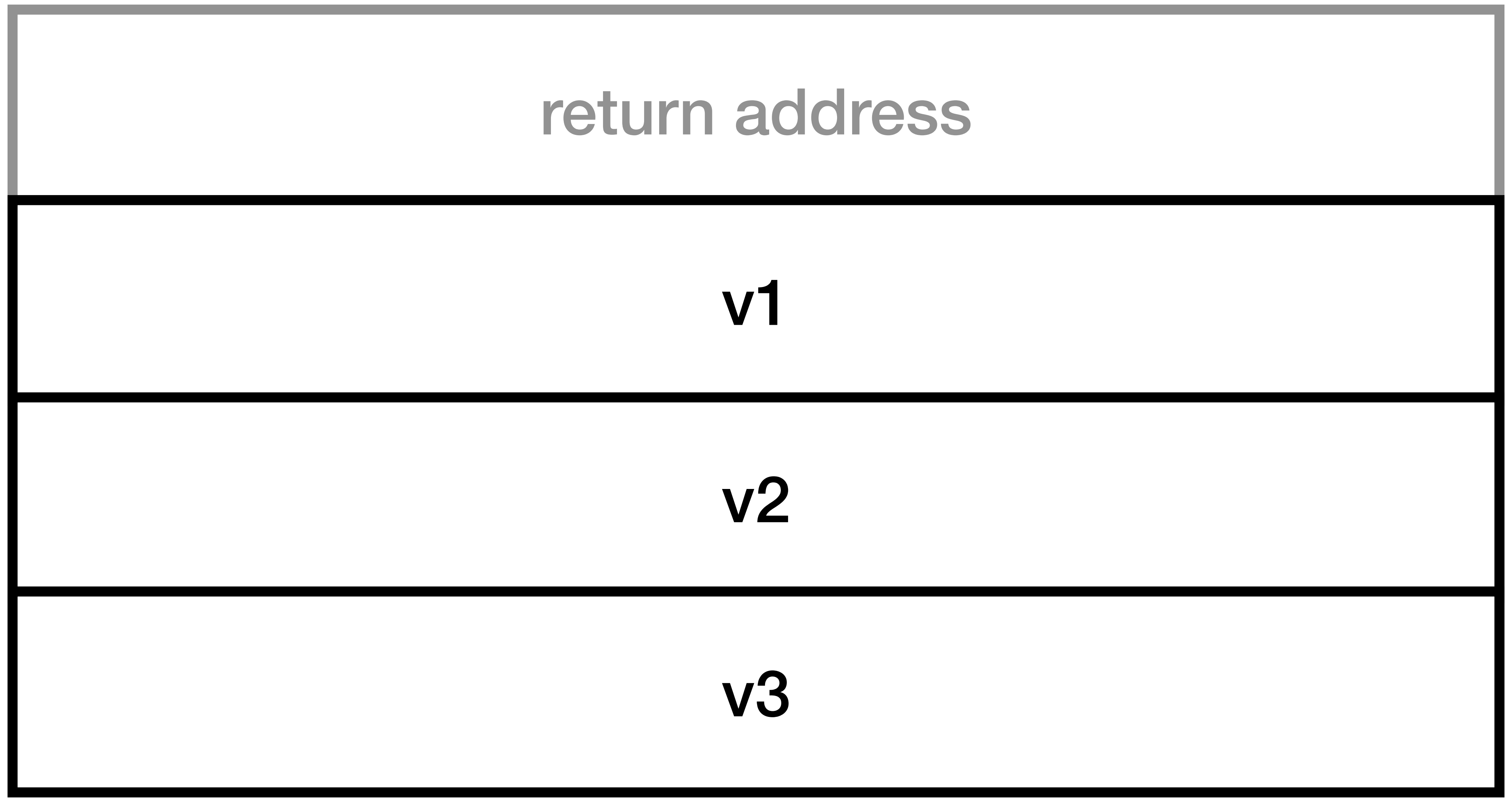
```
(compile-app-tail ‘f (list e1 e2 e3)) c)

(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 - 1 May 2024

Lambda the Ultimate

Announcements: Final project starter code out tonight

Hint on Assignment 5: list patterns

Lambda and first-class functions:

- interpreting with and without functions
- compiling (probably next time)

CMSC 430 - 6 May 2024

Lambda the Ultimate

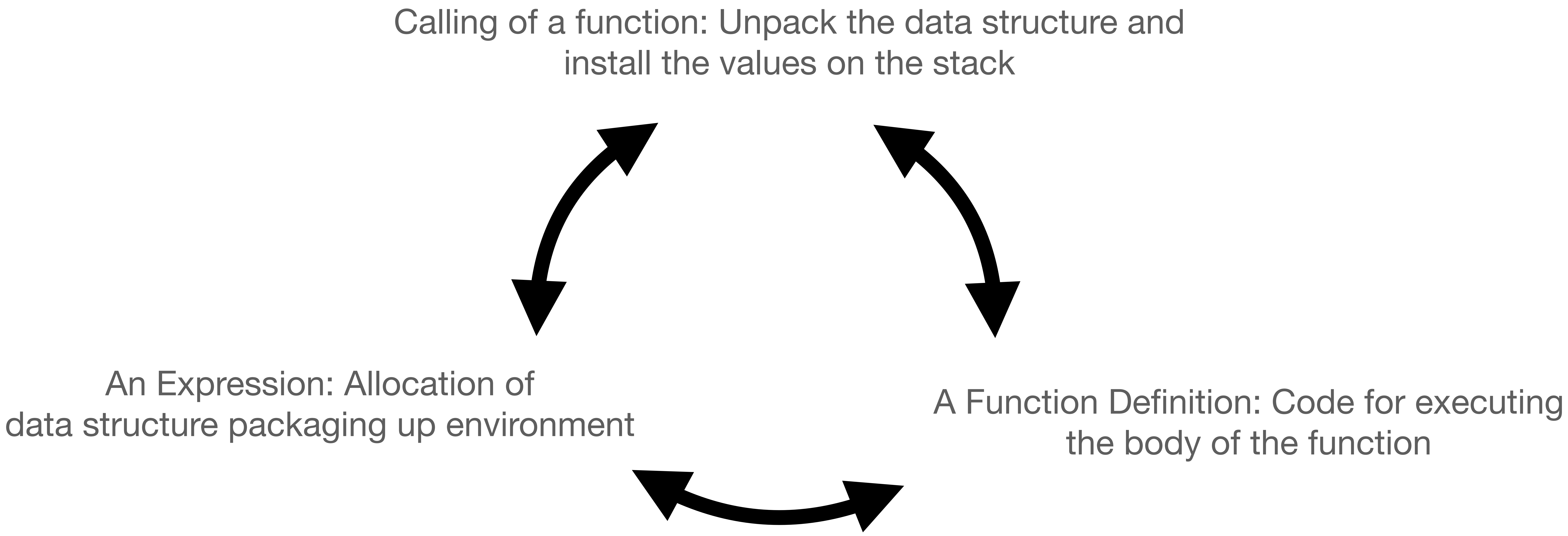
Announcements

- Should we extend the deadline for Assignment 5?
- End of the Road survey out now: due by end of semester
- Course Experience Survey open until May 10: Currently 4% response rate
  - 85% rate: free quiz
  - 95% rate: 2x free quizzes

Today

- Quick look at the final project
- Review the lambda-free interpreter
- Compiling lambda-expressions

The three facets of first-class functions



Encoding pointer values (Loot)

Type tag in least significant bits

61-bits for address	0	0	1	Box
61-bits for address	0	1	0	Cons
61-bits for address	0	1	1	Vector
61-bits for address	1	0	0	String
61-bits for address	1	0	1	Procedure

CMSC 430 - 8 May 2024

The End

Announcements

- Assignment 5 and Final Project accepted through end of semester (Midnight, 5/17)
- Final project autograder issue resolved
- Course Experience Survey due by May 10: Currently 44% response rate
  - 85% rate: free quiz
  - 95% rate: 2x free quizzes

Today

- A look back at everything we learned (hint: it's a lot!)
- Where to go from here?

What we learned

A brief inventory

Writing interpreters & compilers

Code as data

Programs that generate programs

How to specify a programming language

What compiler correctness means

Programming with sophisticated invariants

x86, System V ABI

Testing correctness properties

Programming in a functional language

Representing datatypes in binary

Type-driven programming, recursive programming

How to implement: integers, booleans, characters, strings, lists, pairs, box, vectors, conditional expressions, variable binding, run-time type tag checking, memory allocation, function calls, tail calls, I/O, pattern matching, overloading, rest parameters, apply.

How to build a memory safe programming language out of an unsafe one.

How to build a high-level programming language out of a low-level one.

….

Where to go from here?

From Loot to Outlaw

What are some language features we *use* but didn't *implement*?

- Symbols
- Structures
- Lots of library functions
- Primitives as function values
- Modules
- …

Symbols: Mug

From Loot to Outlaw

Symbols are like strings, but interned.

- Pick a new pointer tag, point to string data
- Use run-time to intern any created symbols

Structures: Neerdowell

From Loot to Outlaw

Structures are like vectors, but each type of structure is different

- Pick a new pointer tag, pointer to an array of elements
- First element is a generated symbol denoting the type
- Remaining elements hold the value of the fields
- Structure definitions generate code for
  - Constructor
  - Predicate
  - Accessors

Library functions

From Loot to Outlaw

Lots of functions we'd like to use can be expressed in our language:

```
(define (length xs)
  (match xs
    ['() 0]
    [(cons x xs)
     (add1 (length xs))]))
```

Rather than write primitives in assembly, compile definition into an object file and link into run-time.

Write a standard library.

Primitives as function values

From Loot to Outlaw

Give primitives their own syntax and move wrapper definitions into standard library:

```
(define (add1 x)
  (%add1 x))
```

Primitive syntax is only available in standard library compiler.

Modules: Punt

From Loot to Outlaw

On Modules, we cheat; we write a utility function that combines a bunch of modules into a single monolithic program.

Where to go from here?

Life after 430

If you found this class interesting, consider

- CMSC 631: Software Foundations
- CMSC 838E: Advanced Compilers
- Undergraduate Research in the PLUM Lab (see Piazza)