CMSC 430 - 27 August 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 Tues & Thurs (except holidays, midterms)

~6 assignments

2 midterms (take-home, 24 hour); dates on web page

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…

Human readable way to create computer instructions

Combination of syntax and semantics for creating behavior

Has to be Turing Complete (maybe)

Way of adding abstractions to make coding faster

Toolbox that prioritizes certain methods of solving problems

What is a compiler?

No, really, I'm asking…

Takes something into one language and turns it into something a machine can read (or something else)

Can be used to optimize programs from one language to another

Help ensure program correctness

Can create executables

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 August 2024

OCaml to Racket

Announcements

- Getting to Know You survey on ELMS, due 9/3 **by start of class**
- Assignment 1 posted; due Thurs 9/5; may be done collaboratively
- Gradescope for Assignment 1 will open by tonight

Ocaml to Racket

Racket = OCaml - Types - Syntax

You should:

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

CMSC 430 - 3 September 2024

OCaml to Racket, cont.

Announcements

- ELMS survey deadline extended, due tonight!
- Assignment 1 open on Gradescope, extended to Tues 9/10;
  may be done collaboratively
- (Some) office hours on web page (more soon)
- Quiz on Racket Basics due by start of next class (ELMS)
- Slides on ELMS under Files
- Request into Facilities about temperature in this room

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 September 2024

A little assembly

Announcements

- Assignment 2 out, due 9/12; may be done collaboratively
- More Racket Basics quiz due Tues 9/7 by start of class

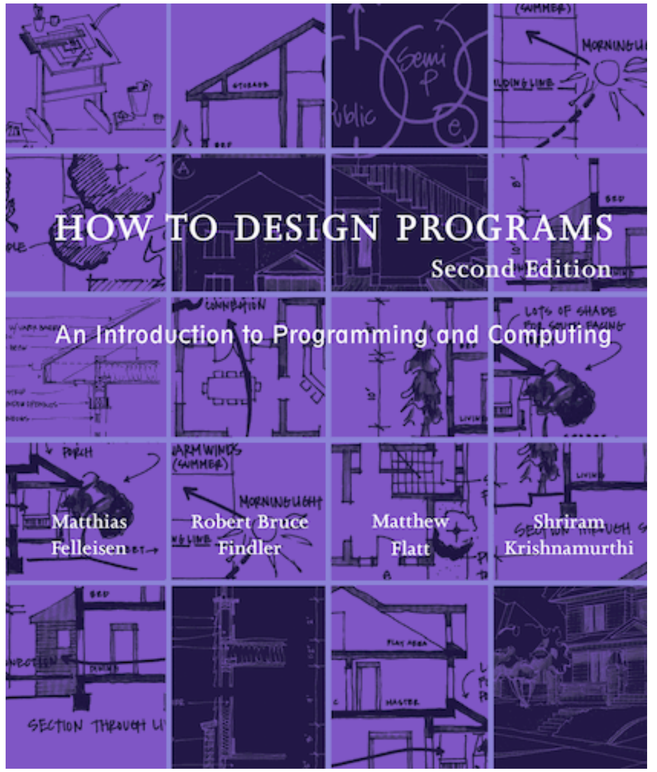
Today

- More 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

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 - 10 September 2024

More a86

Announcements

- a86 Basics quiz out after class: due before class 9/12
- More office hours on web page

Today

- More a86
- Almost all of the instructions we will use
- Reading & writing memory
- Using the stack

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

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 int64 - 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

(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!

Cmp

- Cmp reg int64: 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.

CMSC 430 - 12 September 2024

Our first compiler

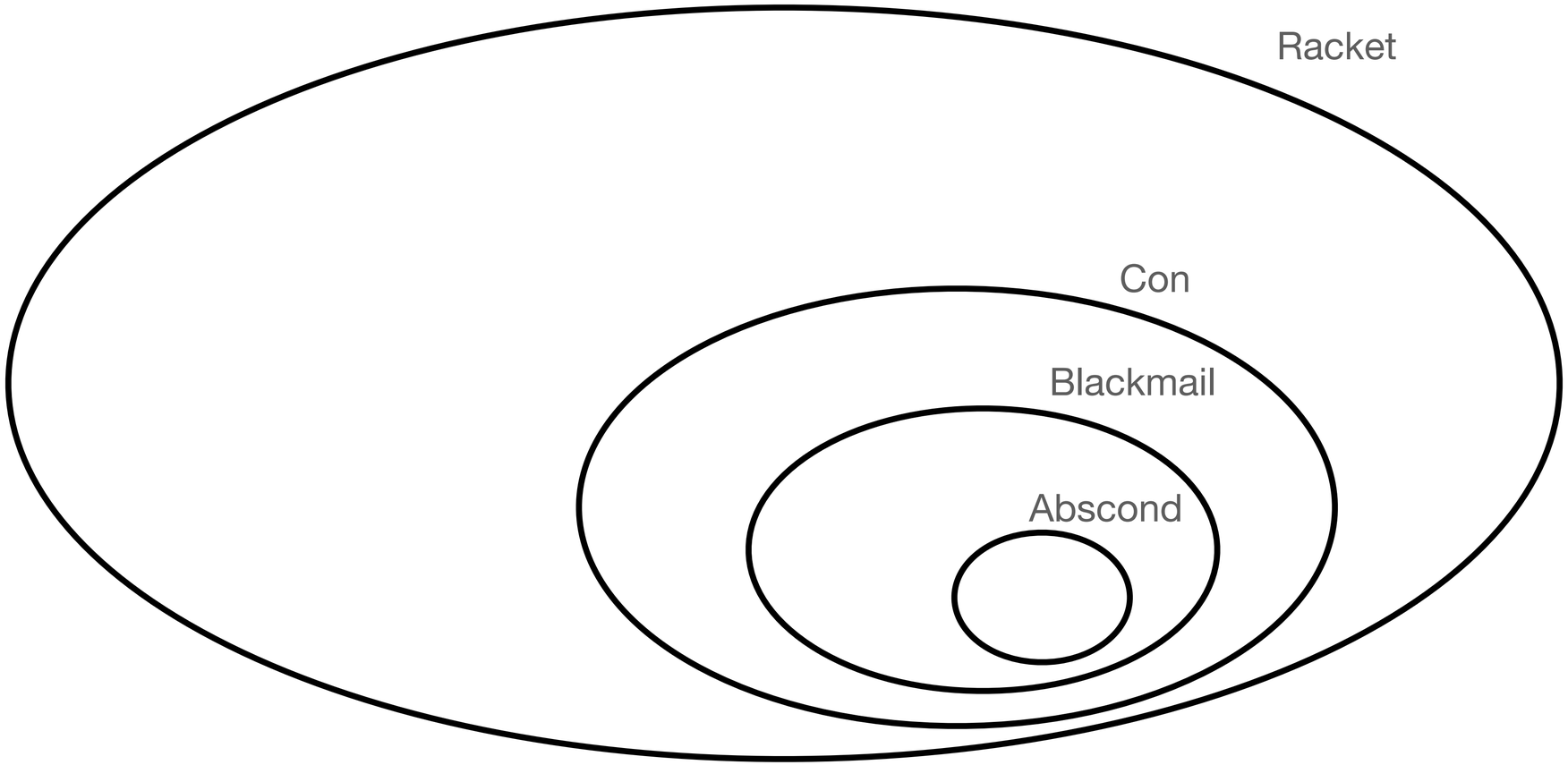
Announcements

- Assignment 2 due tonight
- No quiz today
- Added slides on a86 instructions covered so far

Today

- Abscond: our first language
- Blackmail: a successor
- Con: conditional execution

Language subsets



Parts of our compiler

Reader : Input → S-Expr

Parser: S-Expr → Expr

Compiler: Expr → a86

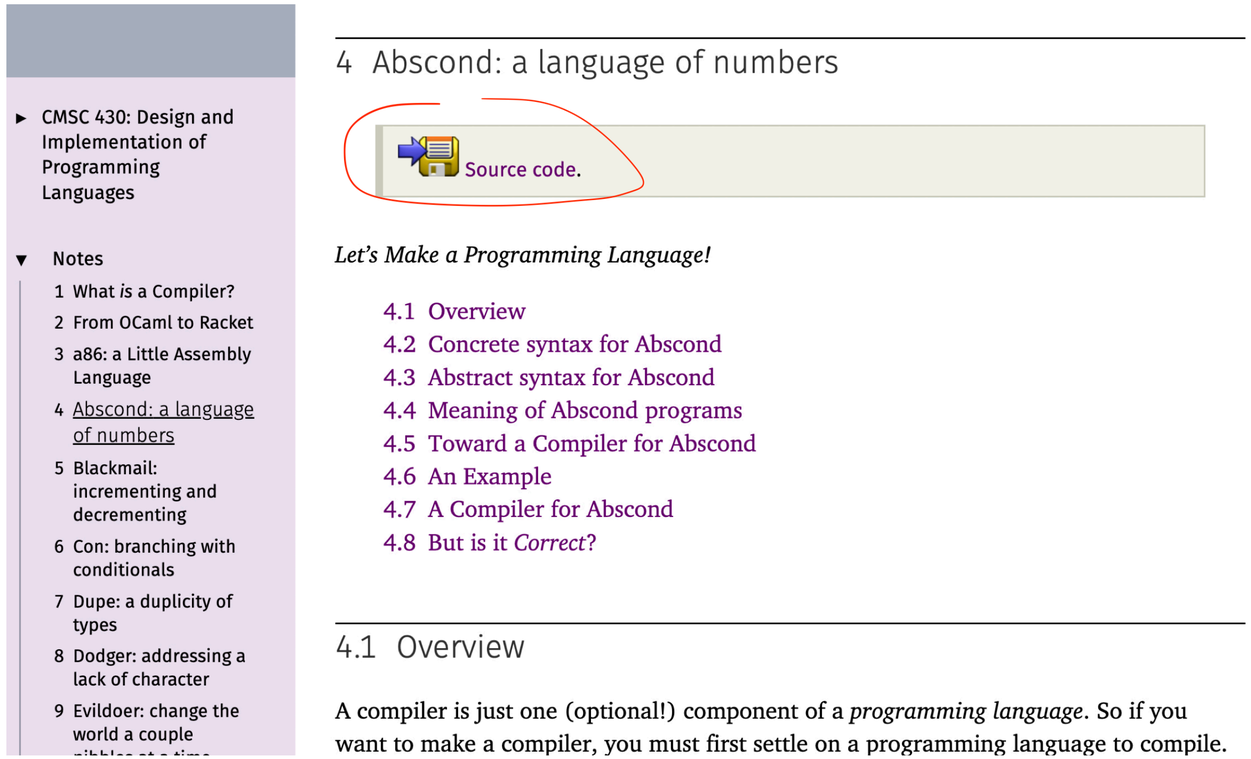
Assembler: a86 → Object

Linker: Object → Executable

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

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

4 Abscond: a language of numbers

Source code.

Let's Make a Programming Language!

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.

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

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

CMSC 430 - 17 September 2024

A few more compilers

Announcements

- Assignment 3 released tonight (due 2 weeks)
- Quiz released after class

Today

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

Abstract Syntax Trees

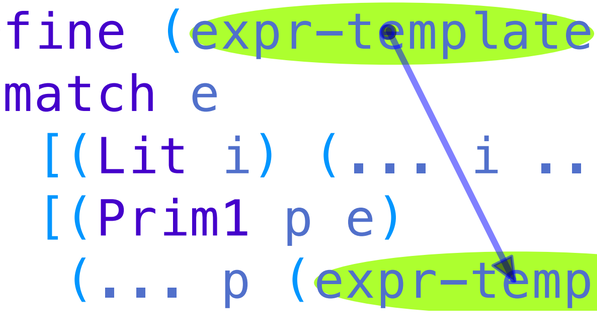
Backbone of all compilers, interpreters, etc.

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) ...)]))
```



Compile follows shape of Expr

Backbone of all compilers, interpreters, etc.

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

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

```
;; Expr -> Asm
(define (compile-e e)
  (match e
    [(Lit i) (seq (Mov rax i))]
    [(Prim1 p e)
     (seq (compile-e e)
          (compile-op1 p))]))
```

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)))
```

Interp also follows shape of Expr

Backbone of all compilers, interpreters, etc.

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

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

```
;; Expr -> Integer
(define (interp e)
  (match e
    [(Lit i) i]
    [(Prim1 p e)
     (interp-prim1 p (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

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)
```

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

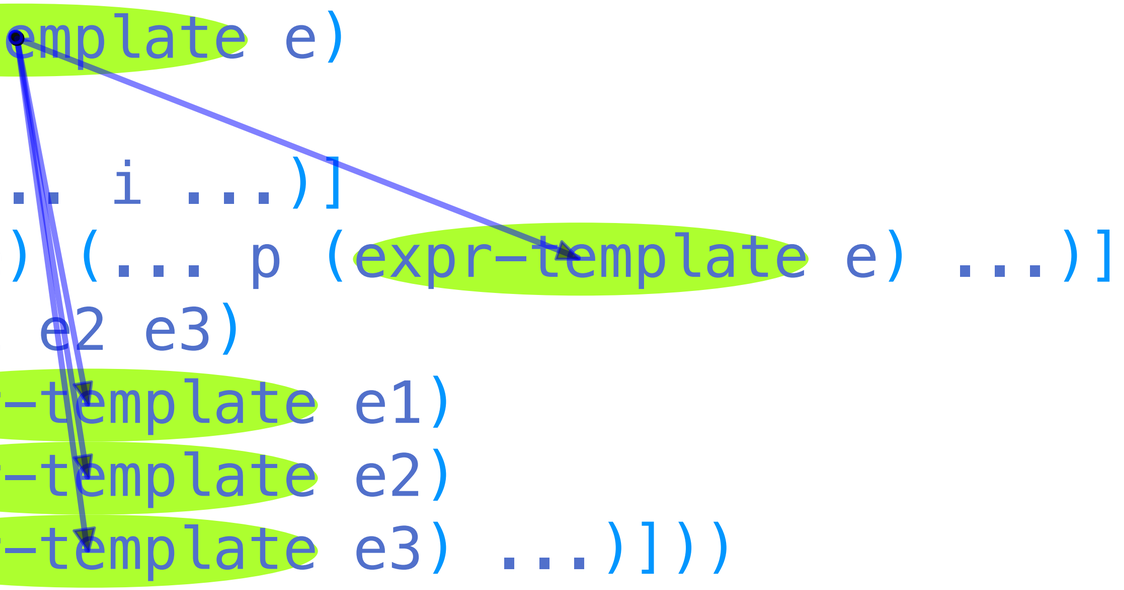
Parser goes from concrete to abstract syntax:

```
;; S-Expr -> Expr
(define (parse s)
  (match s
    [(? exact-integer?) (Lit s)]
    [(list (? op1? o) e) (Prim1 o (parse e))]
    [(list 'if (list 'zero? e1) e2 e3)
     (IfZero (parse e1) (parse e2) (parse e3))]
    [_ (error "Parse error")]))

(define (op1? x)
  (memq x '(add1 sub1)))
```

Con: adding conditional expressions

Revised template for Con

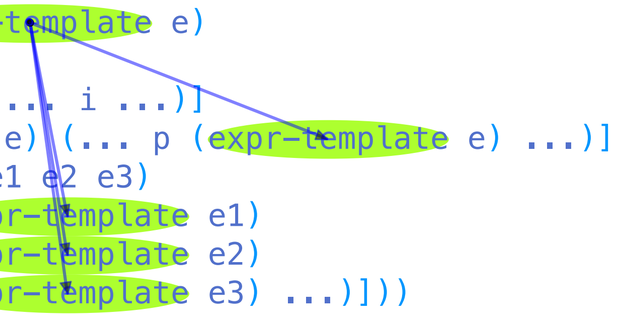
```
;; type Expr = (Lit Integer)
;;           | (Prim1 Op1 Expr)
;;           | (IfZero Expr Expr Expr)

;; 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) ...)]))
```



Con: adding conditional expressions

Use examples and template to write interp

```
;; 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) ...)]))
```



```
(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

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

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

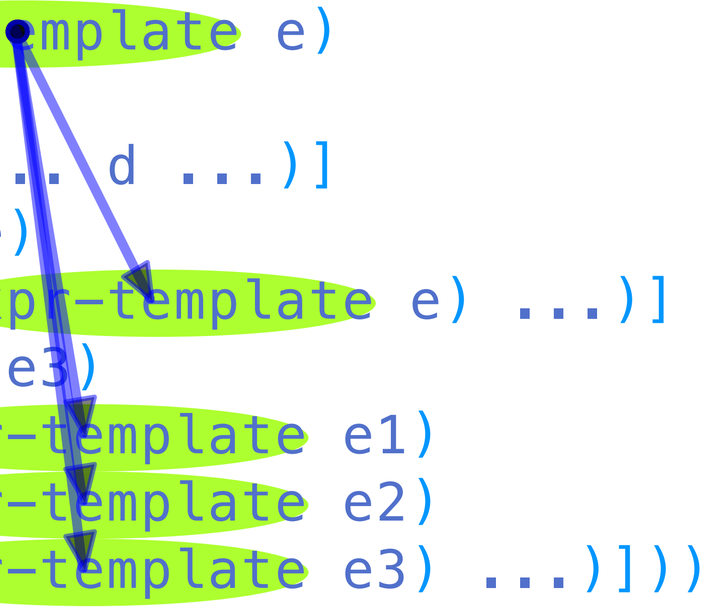
Parser goes from concrete to abstract syntax:

```
;; S-Expr -> Expr
(define (parse s)
  (match s
    [(? datum?)           (Lit s)]
    [(list (? op1? o) e) (Prim1 o (parse e))]
    [(list 'if e1 e2 e3)
     (If (parse e1) (parse e2) (parse e3))]
    [_ (error "Parse error")]))


;; Any -> Boolean
(define (datum? x)
  (or (exact-integer? x)
      (boolean? x)))

(define (op1? x)
  (memq x '(add1 sub1 zero?)))
```

Dupe: adding Boolean values

Revised template for Dupe

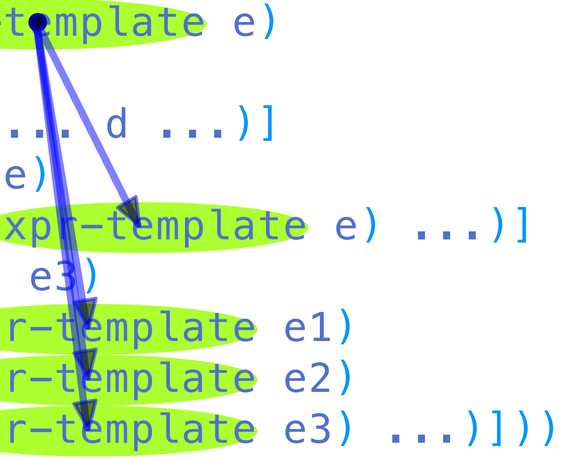
```
;; type Expr = (Lit Datum)
;;           | (Prim1 Op1 Expr)
;;           | (If Expr Expr Expr)

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



Dupe: adding Boolean values

Use examples and template to write interp

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



```
(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

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)))
```

Dupe: adding Boolean values

How do we make Booleans and Integers out of Integers?

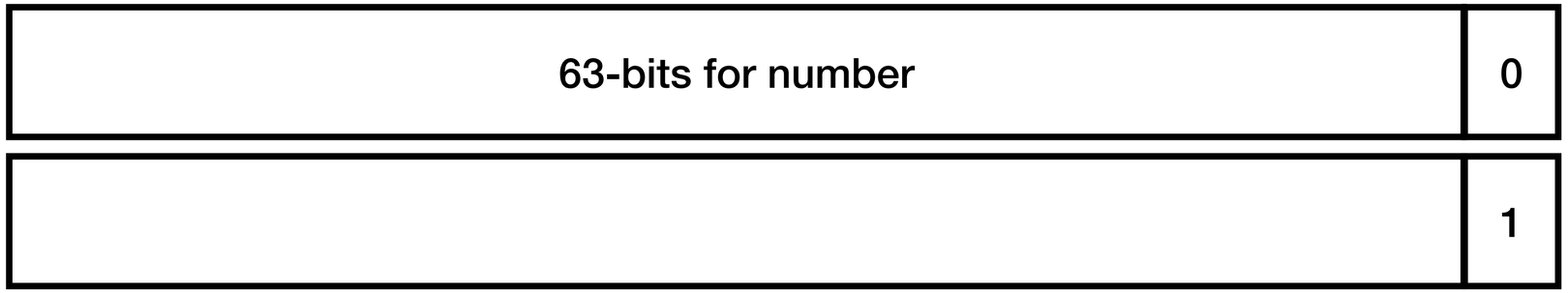
```
#lang racket
#t
```

compiles to

```
(Mov rax ???)
```

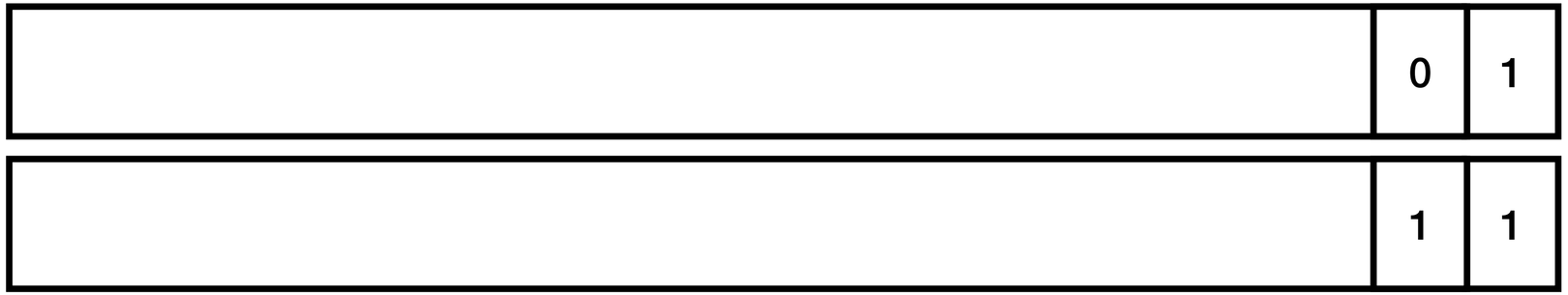
Encoding values in Dupe

Type tag in least significant bits



Integers

Booleans



`#t`

`#f`

Compiler structure

Key files and usage for compiler (updates)

`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, compile-ops.rkt`: AST to a86 compiler
`main.c, print.{c,h}, values.{c,h}, types.h`: run-time system

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

CMSC 430 - 19 September 2024

Representation matters

Announcements

- Assignment 3 out tonight (for real)
- Videos posted on ELMS
- Quiz out today

Today

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

Dupe: adding Boolean values

How do we make Booleans and Integers out of Integers?

```
#lang racket
#t
```

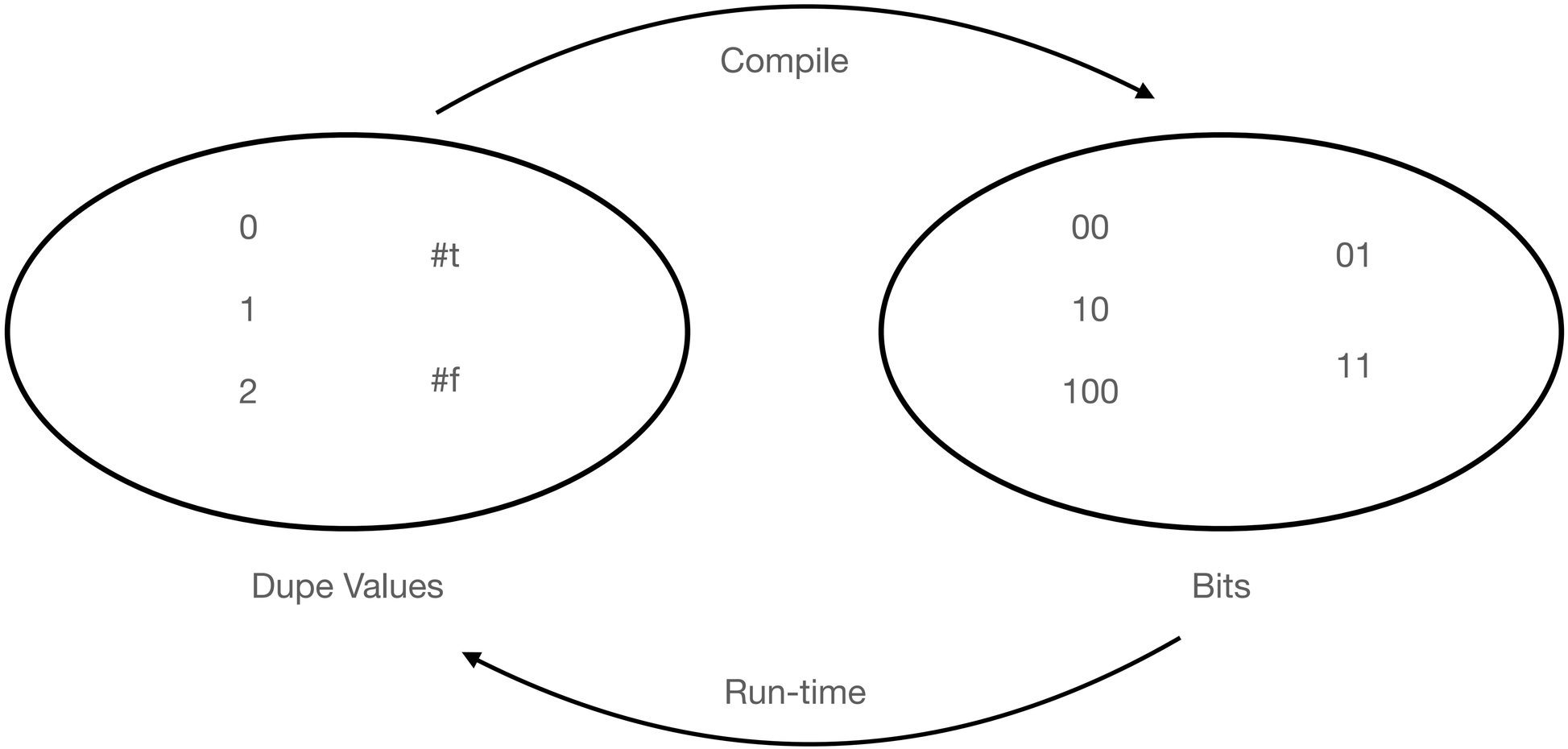
compiles to

```
(Mov rax ???)
```

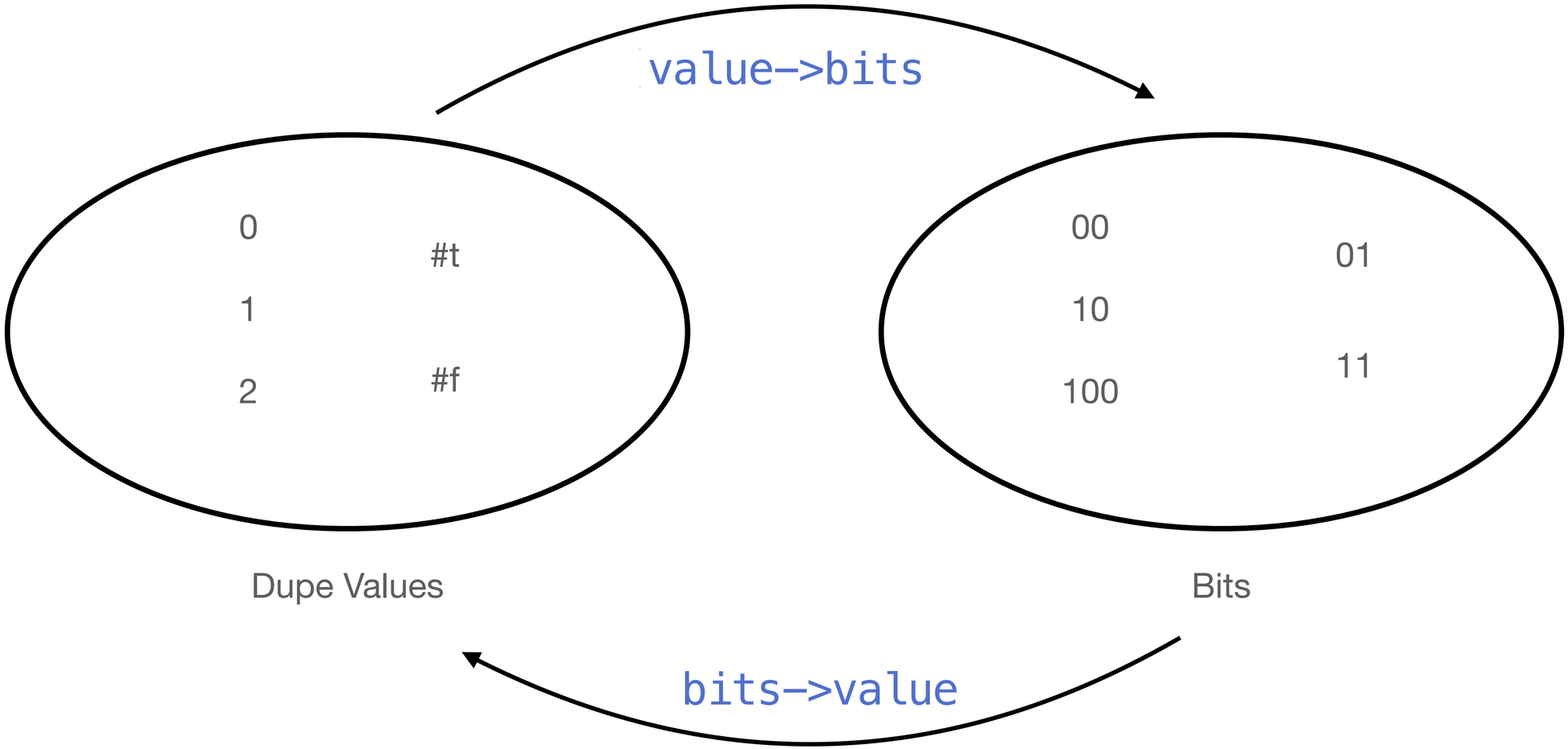
Dupe: adding Boolean values

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

Representing Values with Bits in Dupe



Functions for mapping between worlds

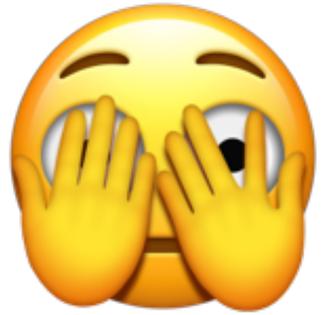


Dupe: adding Boolean values

```
;; 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)))
```



Characters

Like the Booleans, but more of them

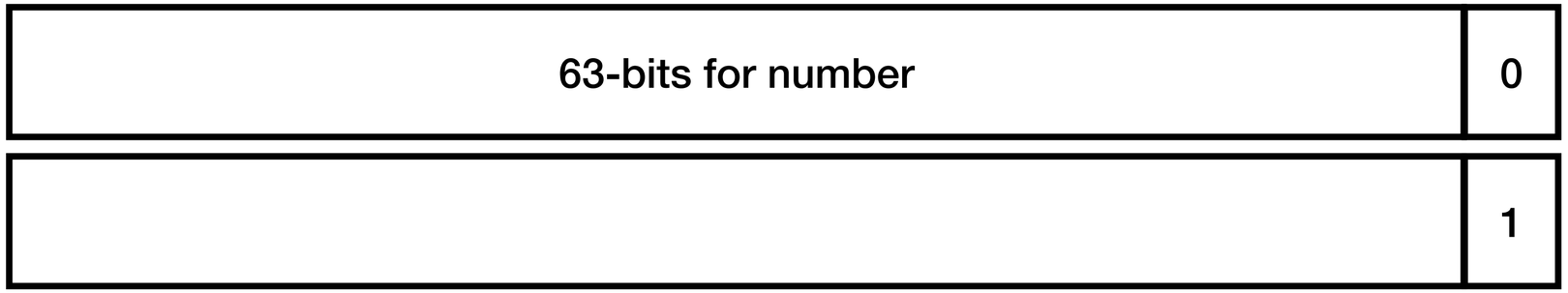
`#\a #\b #\c ... #\λ #\絳 ...`

Unicode: roughly 150K characters.

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

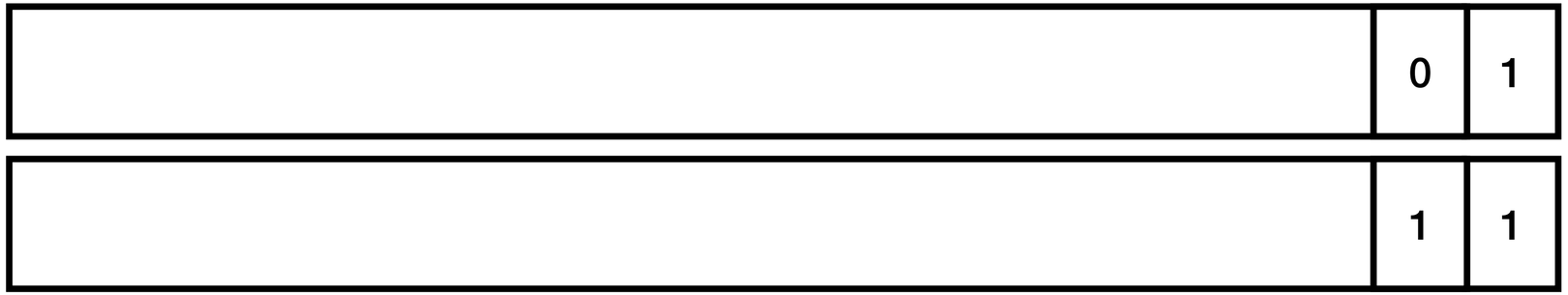
Encoding values in Dupe

Type tag in least significant bits



Integers

Booleans

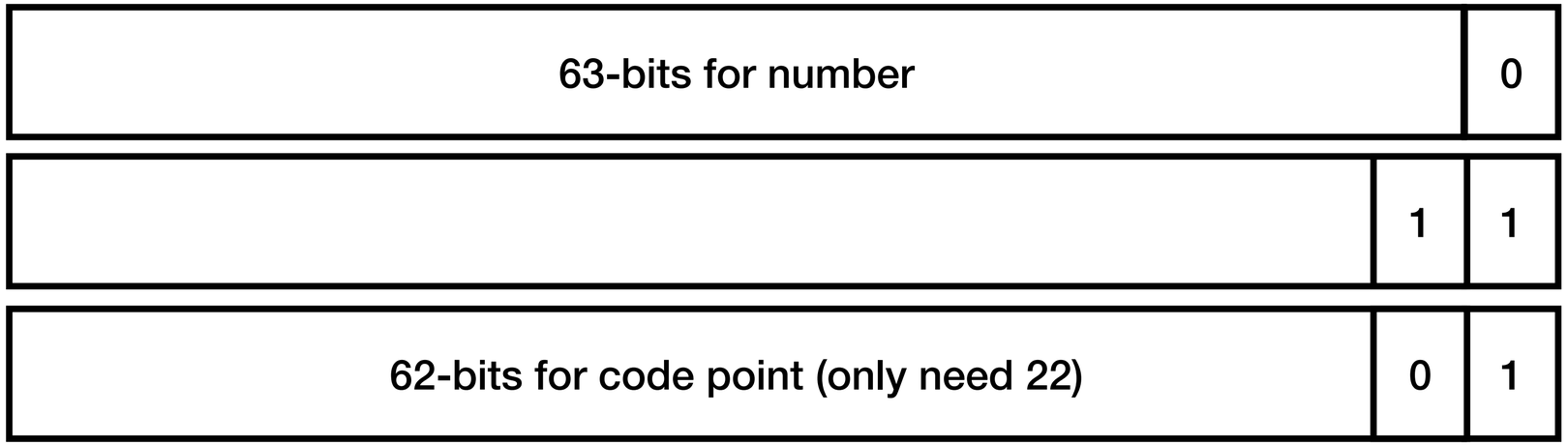


`#t`

`#f`

Encoding values in Dodger

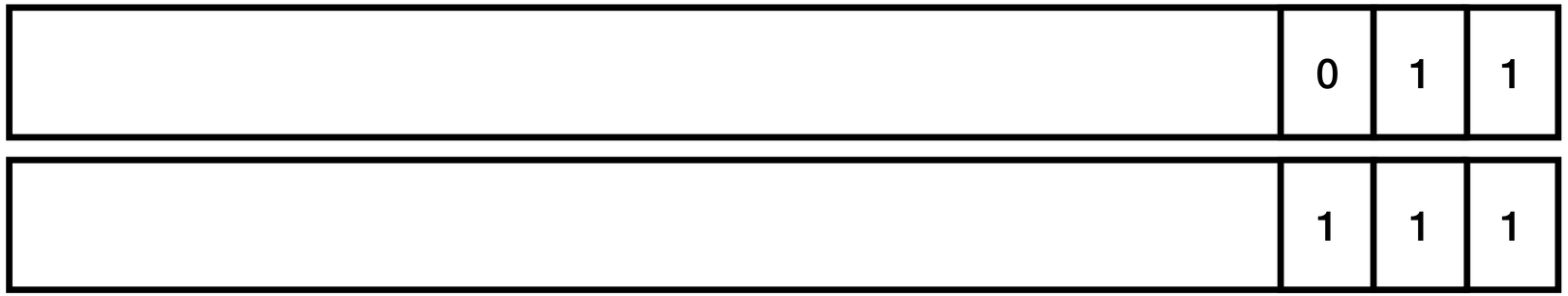
Type tag in least significant bits



Integers

Booleans

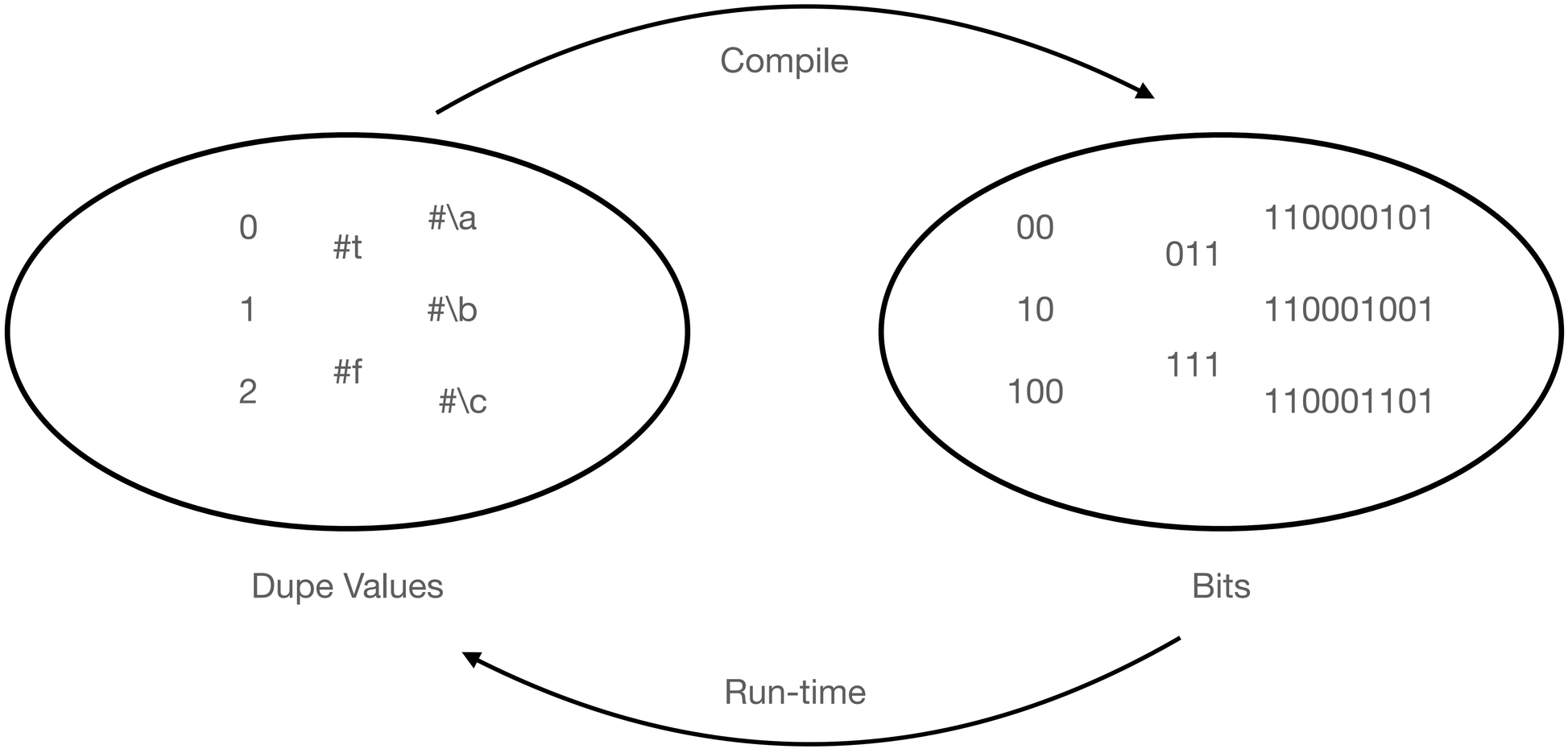
Characters



`#t`

`#f`

Representing Values with Bits in Dodger



CMSC 430 - 24 September 2024

Cancelled

Announcements

- Home sick, no lecture :(

CMSC 430 - 26 September 2024

Changing the world

Announcements

- Survey on ELMS
- Practice exam published soon

Today

- Dodger, quickly
- I/O in Evildoer
  - Evildoer run-time system
  - The System V ABI for making calls

Syntactic additions in Dodger

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)
```

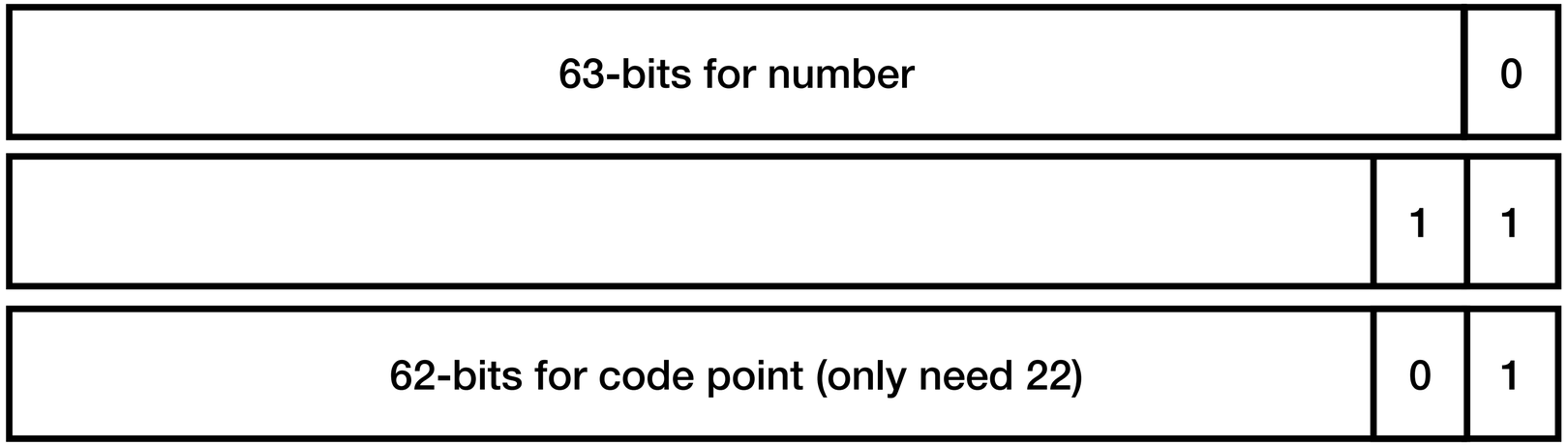
Semantic additions in Dodger

```
;; 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

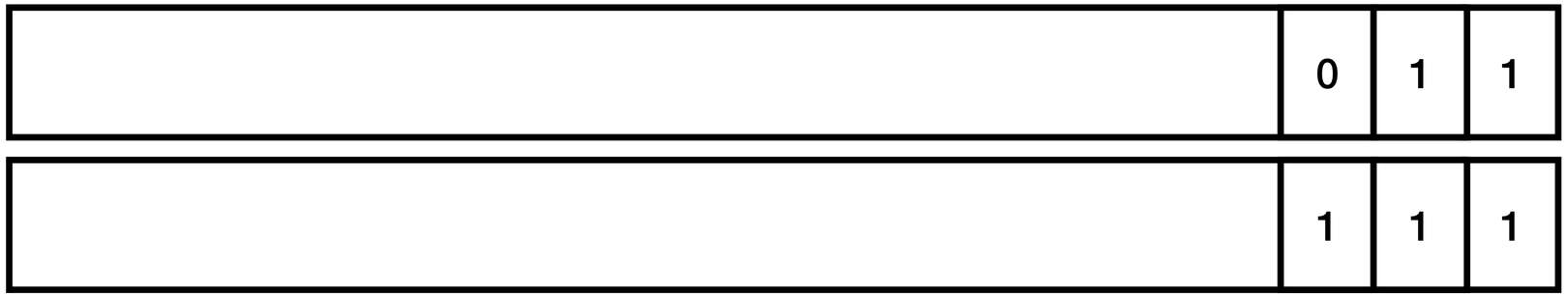
Type tag in least significant bits



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)
```

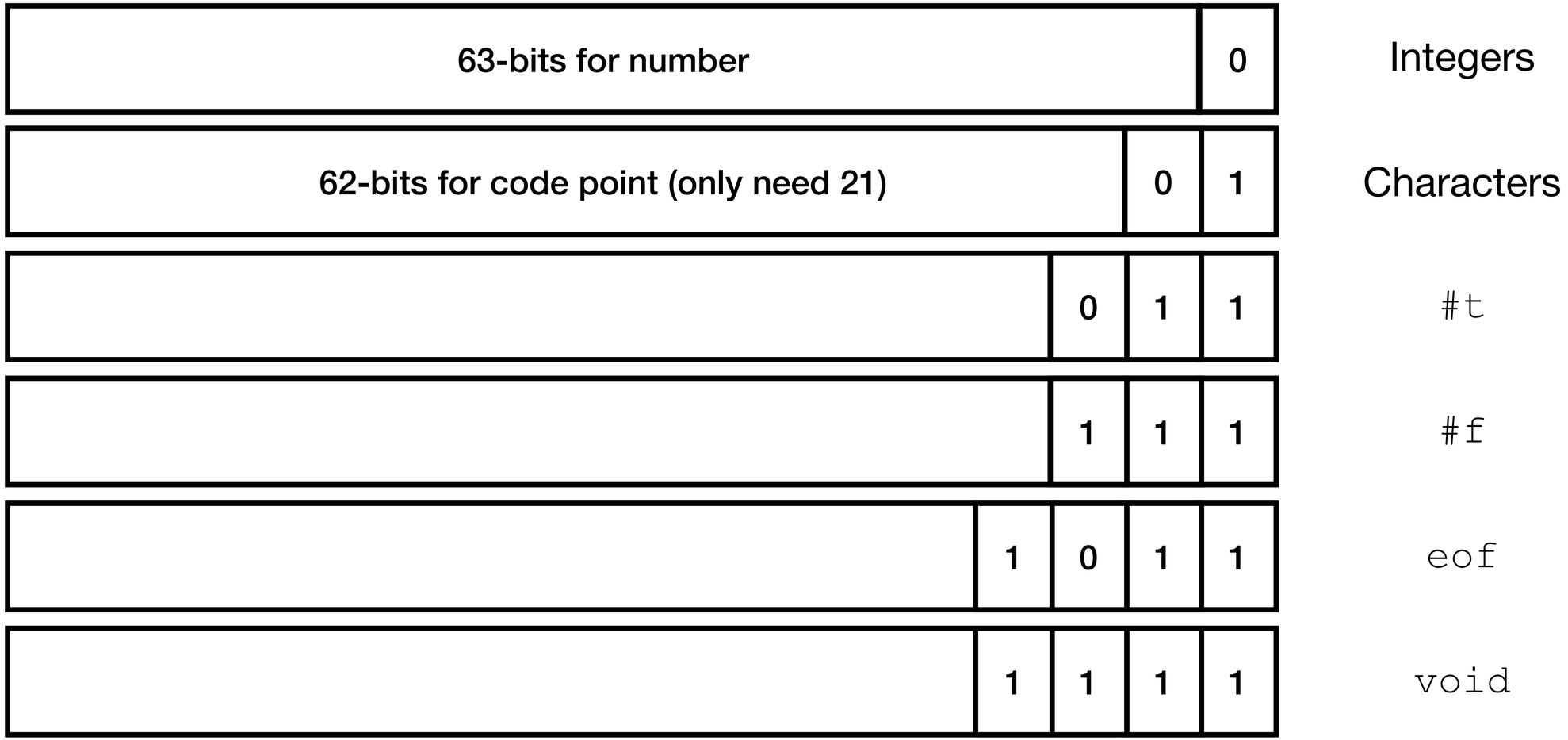
Semantic additions in Evildoer

```
;; 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



I/O support in Run-time

io.c

```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");
  }
}
```

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 . "")
```

Compile-time vs Run-time

When something happens now really matters

```
;; The perfect optimizing compiler
;; Expr -> Asm
(define (compile-e-opt e)
  (Mov rax (value->bits (interp e))))
```

CMSC 430 - 1 October 2024

Taking errors seriously

Announcements

- Practice Midterm 1 on ELMS and Gradescope

Today

- Explaining the stack invariant in Evildoer
- 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?

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

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 - 3 October 2024

Committing Fraud

Announcements

- Understanding Extort quiz out (due by Tues start of class)
- Assignment 3 due tonight
- Midterm 1 — next week (24 hours of Thursday; no lecture)

Today

- Help on Assignment 3: Make examples — a mantra
- 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…

Fraud (1): adding binary operations

Concrete examples

```
(+ 3 4) ;=> 7
(- 4 3) ;=> 1
(< 3 4) ;=> #t
(= 3 4) ;=> #f
```

Fraud (1): adding binary operations

Abstract examples

```
(Prim2 '+ (Lit 3) (Lit 4))
(Prim2 '- (Lit 4) (Lit 3))
(Prim2 '< (Lit 3) (Lit 4))
(Prim2 '= (Lit 3) (Lit 4))
```

Fraud (1): adding binary operations

Updated parser

```
;; S-Expr -> Expr
(define (parse s)
  (match s
    ;; ...
    [(list (? op2? o) e1 e2)
     (Prim2 o (parse e1) (parse e2))]
    [_ (error "Parse error")]))

(define (op2? x)
  (memq x '(+ - < =)))
```

Fraud (1): adding binary operations

Updated semantics

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

Unchanged!

Fraud (1): adding binary operations

Updated semantics

```
;; Expr -> Answer
(define (interp e)
  (match e
    ;; ...
    [(Prim2 p e1 e2)
     (match (interp-env e1)
       ['err 'err]
       [v1
        (match (interp-env e2)
          ['err 'err]
          [v2 (interp-prim2 p v1 v2)])])]))
```

Fraud (1): 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 (1): adding binary operations

Updated semantics

```
> (interp (parse '(+ 3 4)))
7
> (interp (parse '(- 4 3)))
1
> (interp (parse '(< 3 4)))
#t
> (interp (parse '(= 3 4)))
#f
```

Fraud (1): adding binary operations

Updated compiler

```
;; Expr -> Asm
(define (compile-e e)
  (match e
    ;; ...
    [(Prim2 p e1 e2) (compile-prim2 p e1 e2)]))
```

Fraud (1): adding binary operations

Updated compiler

```
;; Op2 Expr Expr -> Asm
(define (compile-prim2 p e1 e2)
  (seq (compile-e e1)
       (compile-e e2)
       (match p
         ;; ...
         ['+ (Add rax rax)])))
```

Fraud (1): adding binary operations

Updated compiler

```
> (run (compile (parse '(+ 1 1))))
2
```

Fraud (1): adding binary operations

Updated compiler

```
;; Op2 Expr Expr -> Asm
(define (compile-prim2 p e1 e2)
  (seq (compile-e e1)
       (Mov r8 rax)
       (compile-e e2)
       (match p
         ;; ...
         ['+ (Add rax r8)])))
```

Fraud (1): adding binary operations

Updated compiler

```
> (run (compile (parse '(+ 1 1))))
2
> (run (compile (parse '(+ 1 2))))
3
```

Fraud (1): adding binary operations

Updated compiler

```
;; Op2 Expr Expr -> Asm
(define (compile-prim2 p e1 e2)
  (seq (compile-e e1)
       (Push rax)
       (compile-e e2)
       (Pop r8)
       (match p
         ;; ...
         ['+ (Add rax r8)])))
```

Fraud (1): adding binary operations

Updated compiler

```
> (run (compile (parse '(+ 1 1))))
2
> (run (compile (parse '(+ 1 2))))
3
```

Fraud (1): adding binary operations

Updated compiler

```
#lang racket
(+ 1 (begin (write-byte 97) 2))
```

Breaks the ABI by calling a function
with an unaligned stack

(We'll come back to the fix later.)

Fraud (2): 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 (2): 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 (2): 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
  y means 12
  z means #f }
```

```
'((x 42)
  (y 12)
  (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:
;; ∀ r x v,
(lookup (ext r x v) x) ≡ v
```

CMSC 430 - 8 October 2024

Variable bindings and lexical addresses

Announcements

- Midterm 1 — On Thursday; no lecture
- Slides updated on ELMS after class

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.

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 -> 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))])]
    #;...))
```

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.*

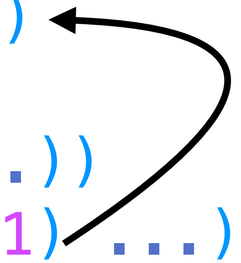
Lookup doesn’t need to be a linear search, we can compute the index of the value in the list.

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 - 15 October 2024

Compiling Fraud + Inductive Datatypes

Announcements

- Midterm 1 graded this week
- Assignment 4 out tonight, due in 2 weeks
- Fraud quiz due before Thursday's class

Today

- Compiling variable bindings and variable references
- Dynamically aligning the stack
- Hustle: inductive datatypes

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))
```

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).

How to represent pointers?

Addressing memory

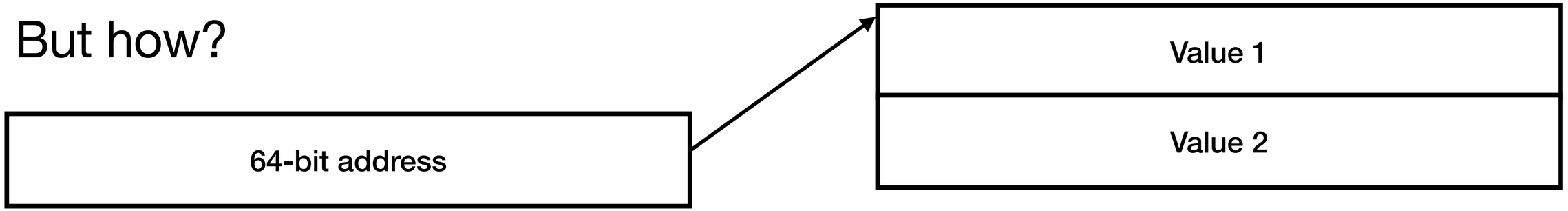
Basic idea:

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

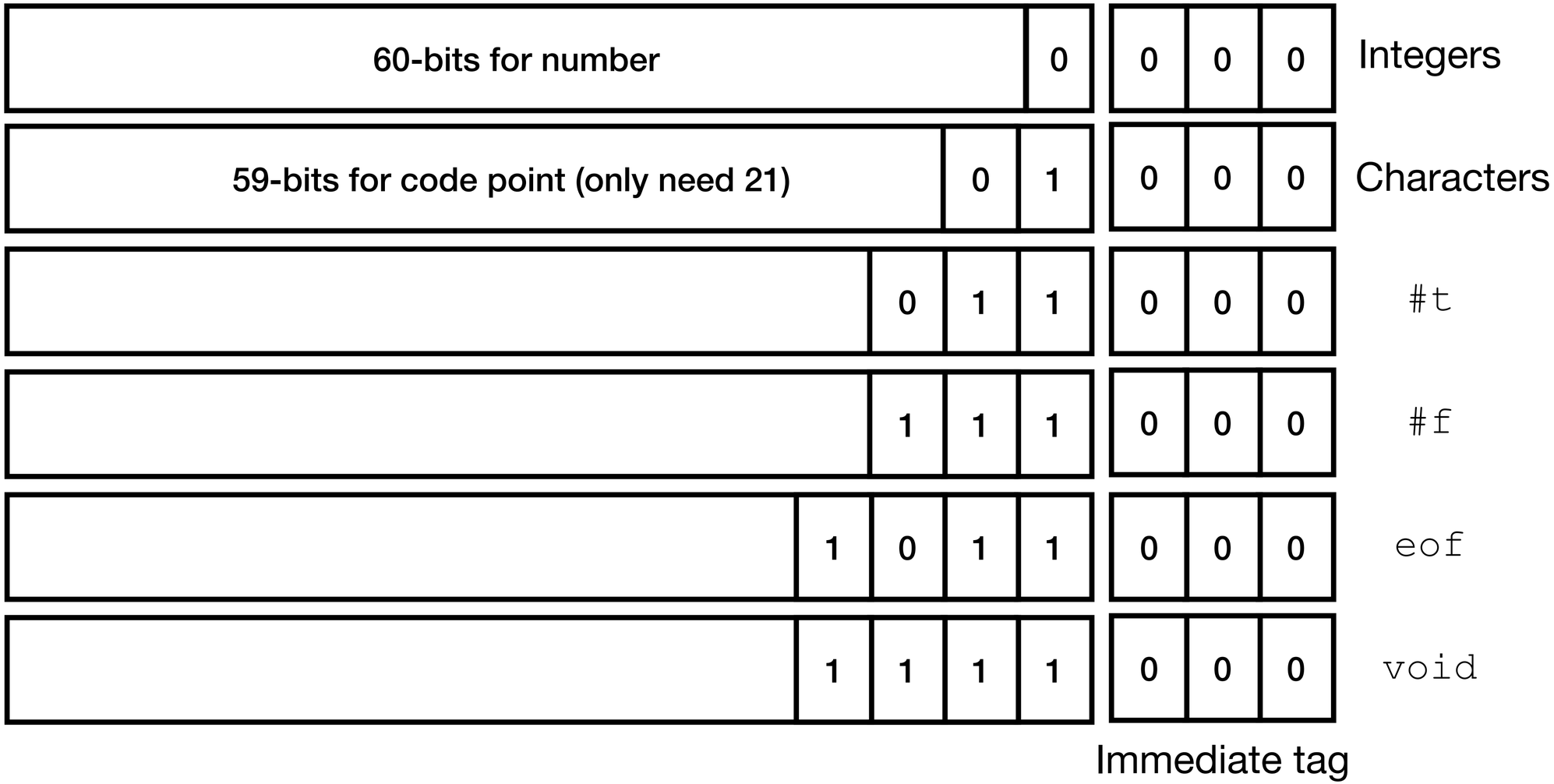
But how?



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

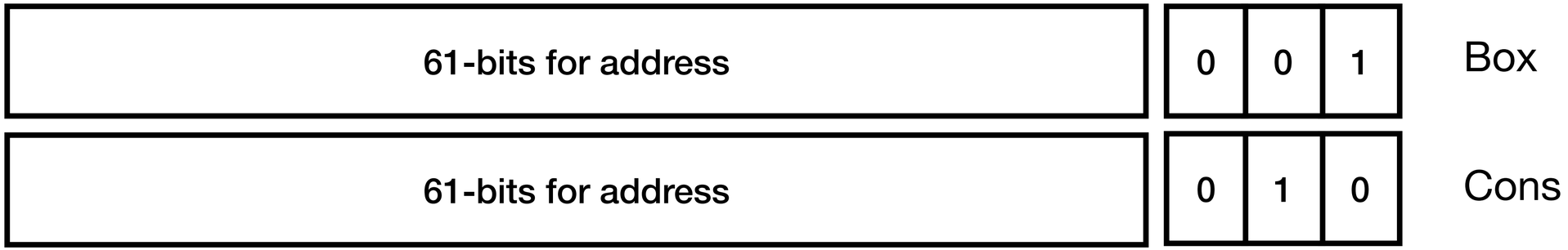
Encoding immediate values (Hustle)

Type tag in least significant bits



Encoding pointer values (Hustle)

Type tag in least significant bits



CMSC 430 - 17 October 2024

Inductive data and memory allocation

Announcements

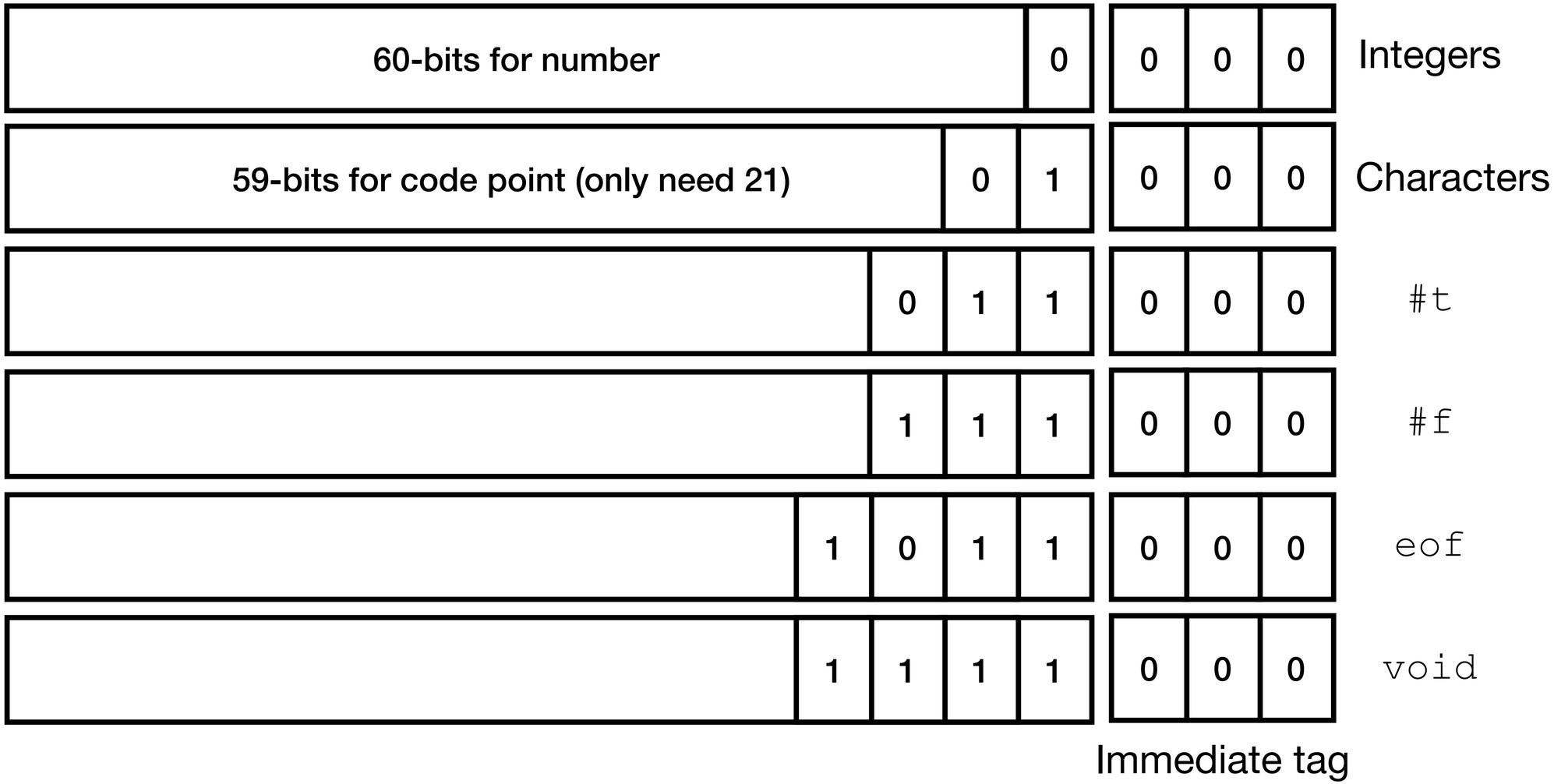
- Assignment 4 released tonight (for real)
- Midterm 1 still being graded

Today

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

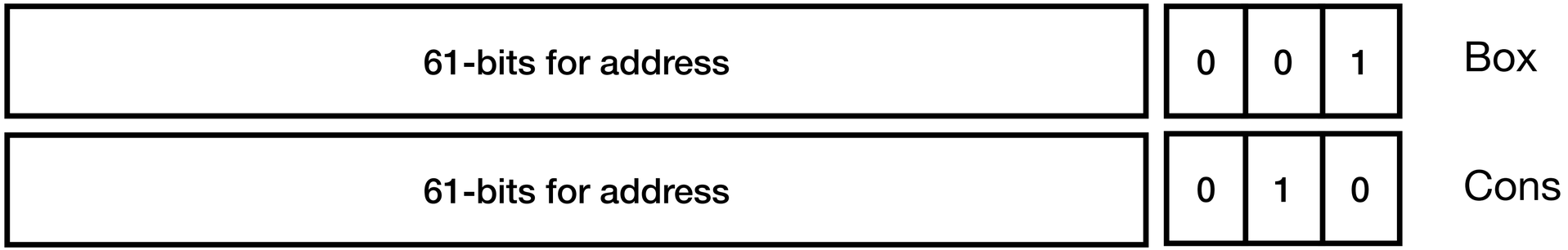
Encoding immediate values (Hustle)

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

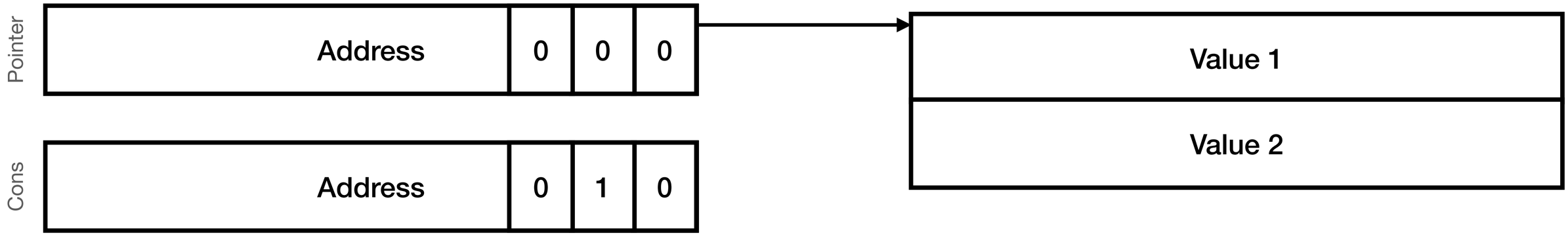
Length of compile time environment =
Number of elements on stack at RT

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

Values that point to memory



Idea:

A pair is allocated as two words in memory

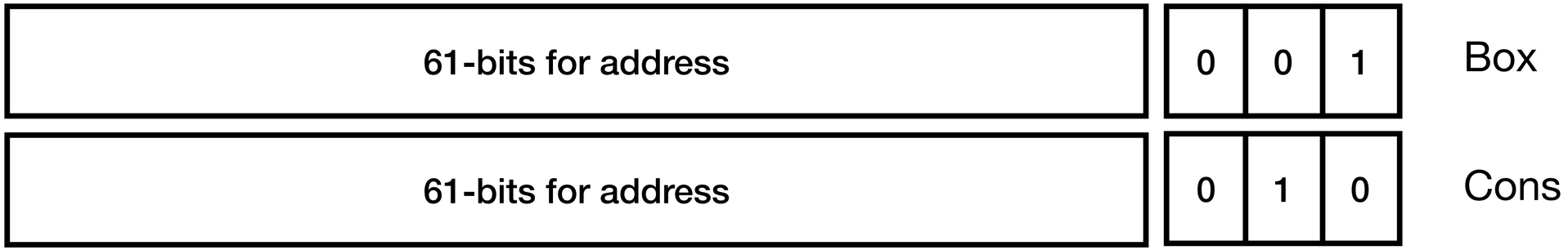
The pair *value* will be represented by the address + type tag in 3 least significant bits

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!

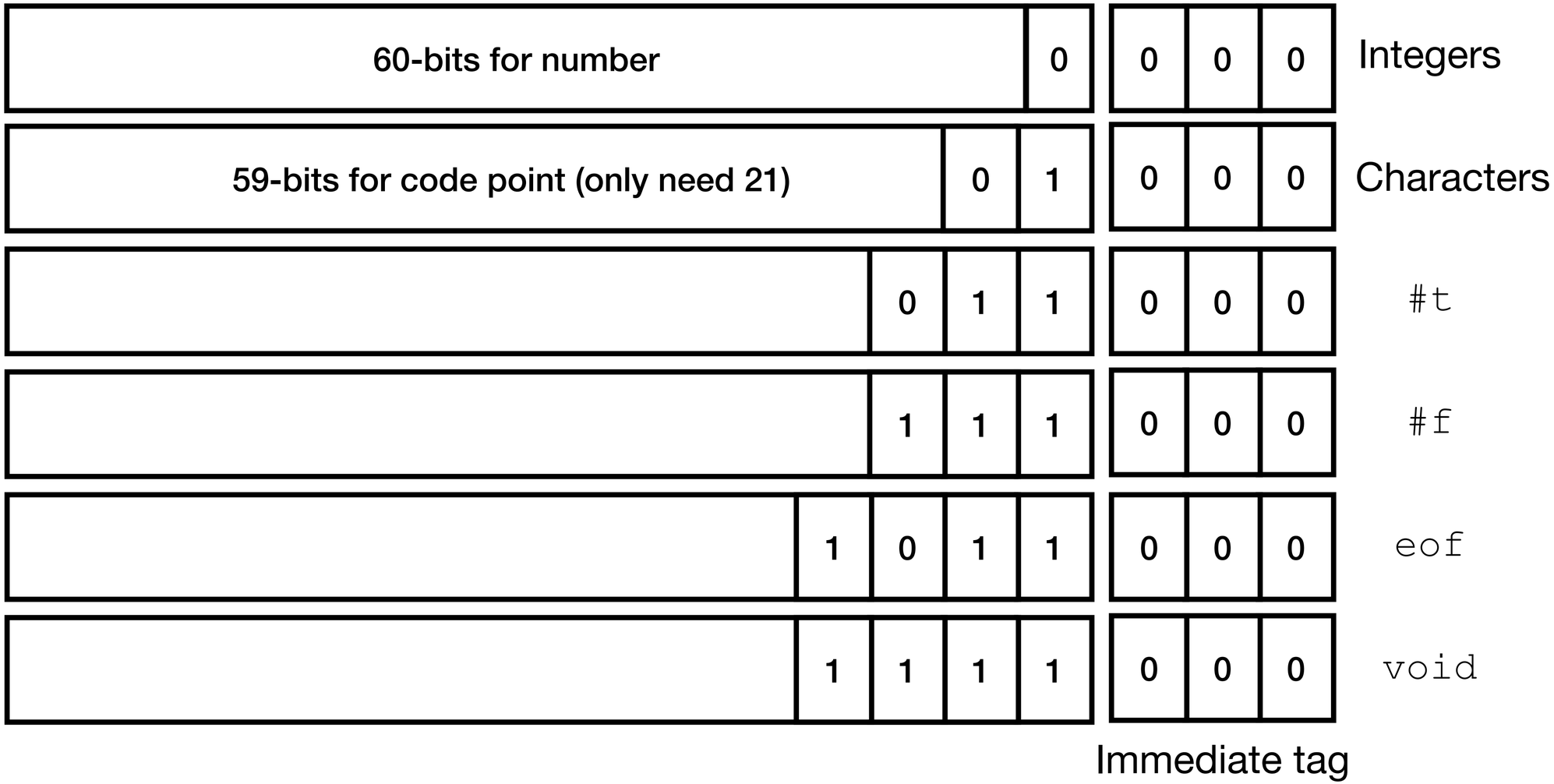
Encoding pointer values (Hustle)

Type tag in least significant bits



Encoding immediate values (Hustle)

Type tag in least significant bits



CMSC 430 - 22 October 2024

Array types and memory allocation

Announcements

- Midterm 1 still being graded, but released today or tomorrow

Today

- Hoax: array data:
  - vectors: heterogenous arrays
  - strings: homogenous arrays
  - Run-time support for heap-allocated values
- Static data (time permitting)

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

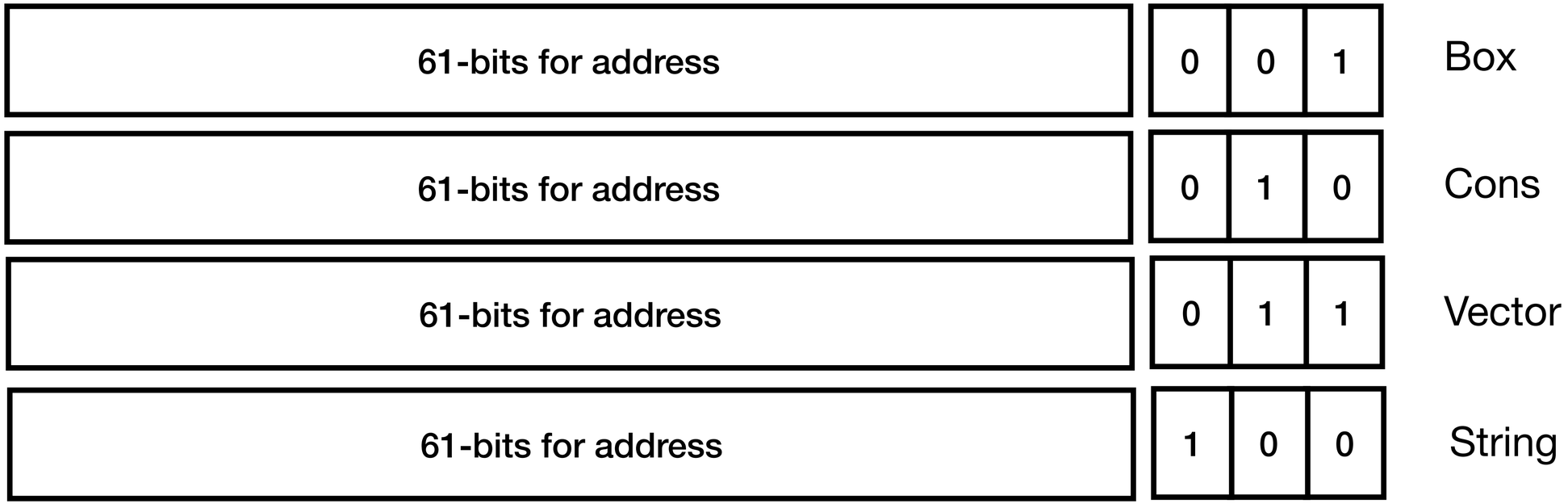
Stack is 8-byte (64-bit) aligned,
i.e. divisible by 8,
i.e. ends in `#b000`
(Must align to 16-bytes to call)

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

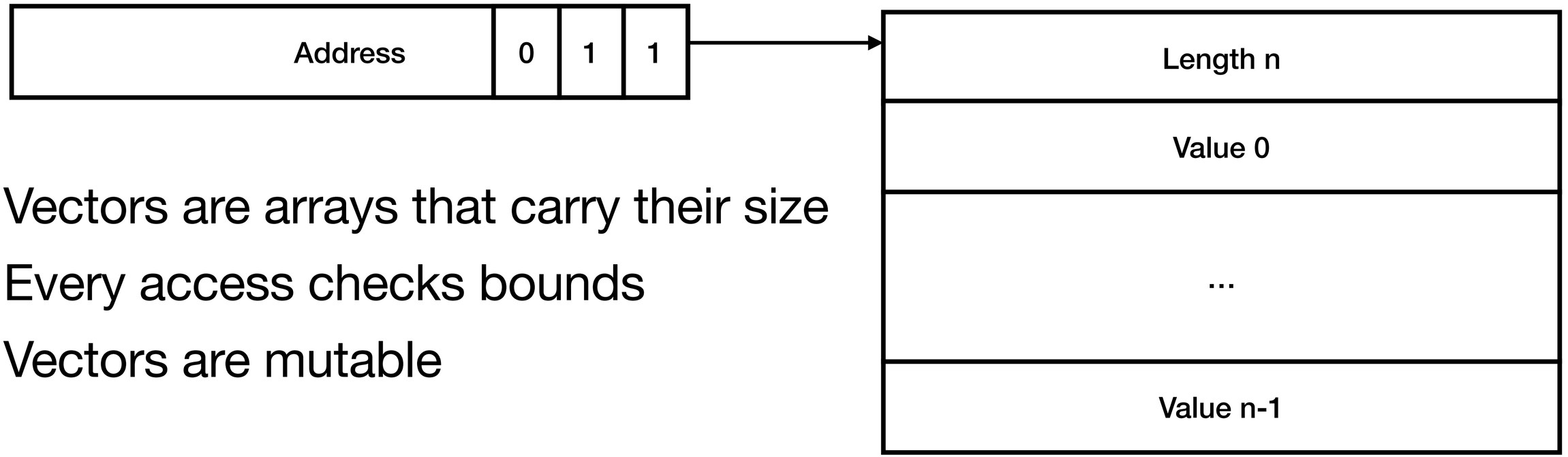
Encoding pointer values (Hoax)

Type tag in least significant bits



How to represent vectors?

Sized heterogeneous arrays



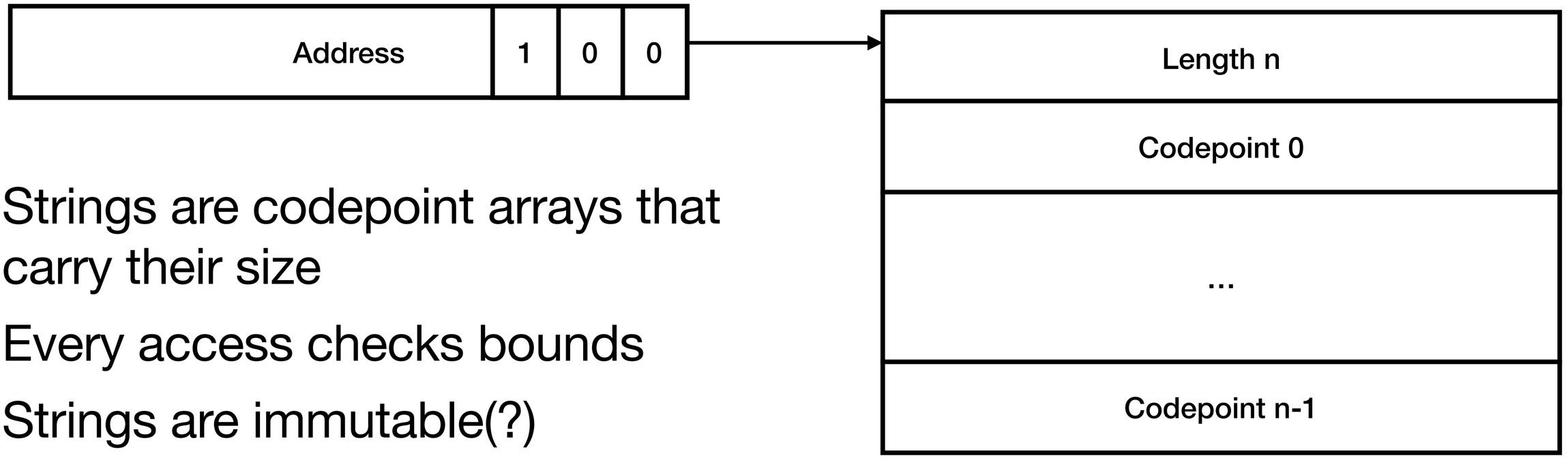
Vectors are arrays that carry their size

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.

CMSC 430 - 24 October 2024

Strings, static memory, interning

Announcements

- Midterm 1 graded, released shortly after class
- Regrade instructions will be posted on Piazza on Monday
- Hustle quiz out, due Tuesday before class

Today

- *The* empty vector
- Strings (like vectors, but different)
- Run-time support for heap-allocated values
- Static data
- Interning literals

The problem with the empty vector

Current design

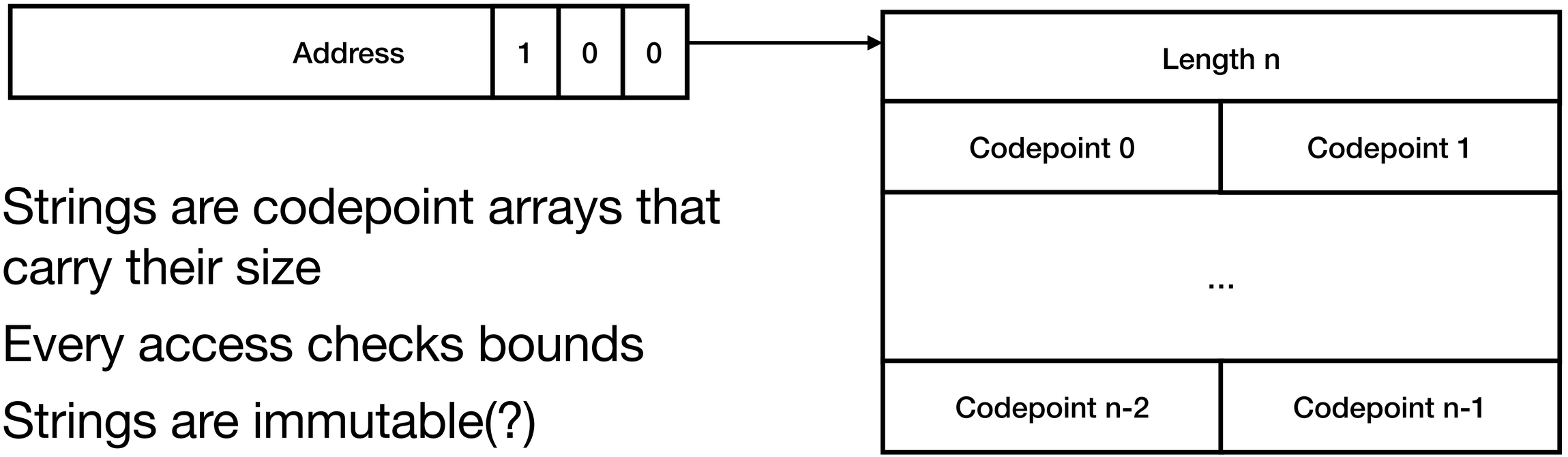
- Wasteful: allocates a word to hold 0 for every empty vector
- Every empty vector is distinct (doesn’t match semantics)

Alternative designs

- Special case the null pointer to represent the empty vector
- Statically allocate a single empty vector and use it

How to represent strings?

Sized homogeneous arrays



Strings are codepoint arrays that carry their size

Every access checks bounds

Strings are immutable(?)

CMSC 430 - 29 October 2024

Interning string literals

Announcements

- Issue with Midterm 1 grades, will be released tonight for real
- Regrade instructions will be posted on Piazza on Thursday
- Looks like Thursday's lecture recording was not upload; will fix soon

Today

- Statically allocating string literals
- Interning string literals
- Run-time support for heap-allocated values
- A look at Iniquity

CMSC 430 - 31 October 2024

Function definitions and calls

Announcements

- Practice Midterm 2 out on ELMS and Gradescope
- Midterm 2: Nov 7 (one week)
- Did you launch DrRacket today?

Today

- Iniquity: function definitions and calls
  - Syntax and semantics
  - Calling convention
  - Compiling calls and functions

Syntactic changes

Programs are now sequences of definitions and an expression

Concrete syntax:

```
(define (f0 x00 ...) e0)
(define (f1 x10 ...) e1)
...
e
```

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))
```

Previously:

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

Now:

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

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

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

Semantic changes

Expressions interpreted in context of definitions

Previously:

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

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

Now:

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

;; Expr Env [Listof Defn] -> Answer
(define (interp-env e r ds) ...)
```

Semantic changes

Expressions interpreted in context of definitions

```
;; Prog -> Answer
(define (interp p)
  (match p
    [(Prog ds e)
     (interp-env e '() ds)]))
```

Meaning of a program is the meaning of the top-level expression, in the context of the program's definitions

Semantic changes

Expressions interpreted in context of definitions

```
;; Expr Env [Listof Defn] -> Answer
(define (interp-env e r ds)
  (match e
    [(Lit d) d]
    [(Eof)   eof]
    [(Var x) (lookup r x)]
    [(Prim0 p) (interp-prim0 p)]
    [(Prim1 p e)
     (match (interp-env e r ds)
       ['err 'err]
       [v (interp-prim1 p v)])]
    [(Prim2 p e1 e2)
     (match (interp-env e1 r ds)
       ['err 'err]
       [v1 (match (interp-env e2 r ds)
             ['err 'err]
             [v2 (interp-prim2 p v1 v2)])])]
    #;...))
```

The definition context doesn’t change, but threaded through the interpreter

Otherwise, just like before

Semantic changes

Expressions interpreted in context of definitions

```
;; Expr Env [Listof Defn] -> Answer
(define (interp-env e r ds)
  (match e
    #;...
    [(App f es) ...]))
```

The definition context doesn’t change, but threaded through the interpreter

Otherwise, just like before… except now there are function call expressions

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)))))
```

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)
```

What if an argument raises an error?

Then the whole thing is an error

```
(interp-env (App 'f (list e1 e2 ...)) r ds) ;=> 'err
;;   if (interp-env ei r ds) => 'err for any i
```

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 - 5 November 2024

Pattern Matching

Announcements

- Assignment 5 out tonight, due 11/21
- Midterm 2 - Thursday
- Slides updated on ELMS right after class

Today

- Fixing one last issue with calls
- 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 - 12 November 2024

Compiling Pattern Matching

Announcements

- Assignment 5 out, due 11/26 (<— pushed back)
- Final project out soon
- Midterm 2 survey due by next class

Today

- Compiling pattern matching

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)
```

Recall: pattern matching doing several things at once:
- Destructuring
- Conditional evaluation
- Binding

Compiling a match expression

```
;; Expr [Listof Pat] [Listof Expr] CEnv Bool -> Asm
(define (compile-match e ps es c t?)
  (let ((done (gensym)))
    (seq (compile-e e c #f)
         (Push rax)     ;; save away to be restored by each clause
         (compile-match-clauses ps es (cons #f c) done t?)
         (Jmp 'err)
         (Label done)   ;; where to jump if you've finished a RHS
         (Add rsp 8)))) ;; pop the saved value being matched
```

Compiling a list of match clauses

```
;; [Listof Pat] [Listof Expr] CEnv Symbol Bool -> Asm
(define (compile-match-clauses ps es c done t?)
  (match* (ps es)
    [('() '()) (seq)]
    [((cons p ps) (cons e es))
     (seq (compile-match-clause p e c done t?)
          (compile-match-clauses ps es c done t?))]))
```

Just the sequence of compiling each clause

Compiling a match clause

```
;; Pat Expr CEnv Symbol Bool -> Asm
(define (compile-match-clause p e c done t?)
  (let ((next (gensym)))
    ;; Compiling a pattern needs to know two things:
    ;; - a compile-time environment that describes bindings made so far
    ;; - where to jump in case the pattern doesn't match
    (match (compile-pattern p '() next)
      ;; Compiling a pattern produces two things:
      ;; - instructions that will either:
      ;;   * match value currently in rax,
      ;;     binding things by pushing on the stack
      ;;   * not match, clear out the bindings, and jump to next
      ;; - a compile-time environment that describe the bindings
      ;;   of the pattern, should it succeed
      [(list i cm)
       (seq (Mov rax (Offset rsp 0))       ;; restore value being matched into rax
            i                              ;; do matching, binding, maybe jumping to next
            ;; If you made it here, the pattern matched and the bindings described
            ;; by cm have been pushed onto the stack
            (compile-e e (append cm c) t?) ;; Do the RHS
            (Add rsp (* 8 (length cm)))    ;; Clear out the bindings
            (Jmp done)                     ;; Jump to bottom of clauses code
            (Label next))])))              ;; Where to jump if p doesn't match
```

Compiling a wildcard or variable pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - Always matches
    ;; - No new bindings
    [(Var '_)
     (list (seq) cm)]
    ;; - Always matches
    ;; - Binds x to value in rax
    [(Var x)
     (list (seq (Push rax)) (cons x cm))]
    #;...))
```

Compiling a literal pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - Matches if l is equal to value in rax
    ;; - No new bindings
    [(Lit l)
     (let ((ok (gensym)))
       (list (seq (Mov r8 rax)
                  (compile-value l) ;; puts l in rax
                  (Cmp rax r8)
                  (Je ok)
                  (Add rsp (* 8 (length cm)))
                  (Jmp next)
                  (Label ok))
             cm))]
    #;...))
```

Compiling a box pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - matches if rax is a box and value in box matches p
    ;; - Adds any bindings that p adds
    [(Box p)
     (match (compile-pattern p cm next)
       [(list i1 cm1)
        (let ((ok (gensym)))
          (list
           (seq (Mov r8 rax)
                (And r8 ptr-mask)
                (Cmp r8 type-box)
                (Je ok)
                ; haven't pushed anything yet
                (Add rsp (* 8 (length cm)))
                (Jmp next)
                (Label ok)
                (Xor rax type-box)
                (Mov rax (Offset rax 0))
                i1)
           cm1))])]
    #;...))
```

Compiling an and pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Conj p1 p2)
     (match (compile-pattern p1 (cons #f cm) next)
       [(list i1 cm1)
        (match (compile-pattern p2 cm1 next)
          [(list i2 cm2)
           (list
            (seq (Push rax) ;; save away value so it can be restored for i2
                 i1
                 ;; Reach up into the stack to grab the stowed away value
                 (Mov rax (Offset rsp (* 8 (- (sub1 (length cm1)) (length cm)))))
                 i2)
            cm2)])])]
    #;...))
```

Compiling a cons pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Cons p1 p2)
     (match (compile-pattern p1 (cons #f cm) next)
       [(list i1 cm1)
        (match (compile-pattern p2 cm1 next)
          [(list i2 cm2)
           (let ((ok (gensym)))
             (list
              (seq (Mov r8 rax)
                   (And r8 ptr-mask)
                   (Cmp r8 type-cons)
                   (Je ok)
                   (Add rsp (* 8 (length cm))) ; haven't pushed anything yet
                   (Jmp next)
                   (Label ok)
                   (Xor rax type-cons)
                   (Mov r8 (Offset rax 0))
                   (Push r8)                   ; push cdr
                   (Mov rax (Offset rax 8)) ; mov rax car
                   i1
                   (Mov rax (Offset rsp (* 8 (- (sub1 (length cm1)) (length cm)))))
                   i2)
              cm2))])])]
    #;...))
```

CMSC 430 - 19 November 2024

Tail calls

Announcements

- Final project out (autograder open soon)
- Please don’t talk during the lecture

Today

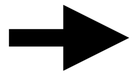
- Being smarter about some function calls
- What is a tail position?
- Compiling calls in tail position

Consider this program

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

(f 100)
```

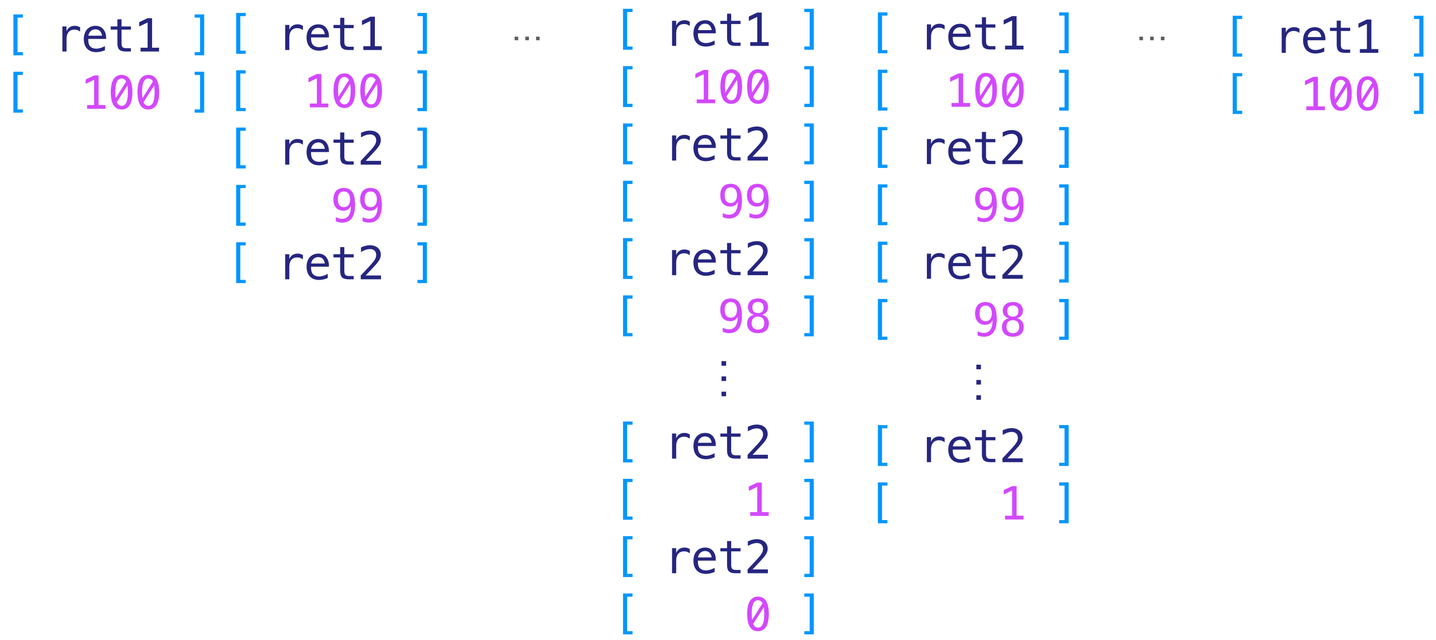
```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

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

A view of the stack



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

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

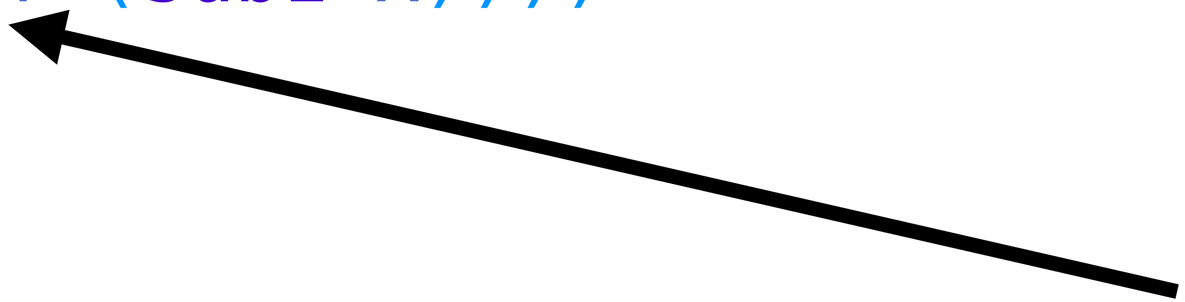
A view of the stack




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

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

A view of the stack

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

(f 100)
```



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

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

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

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

Pop before call

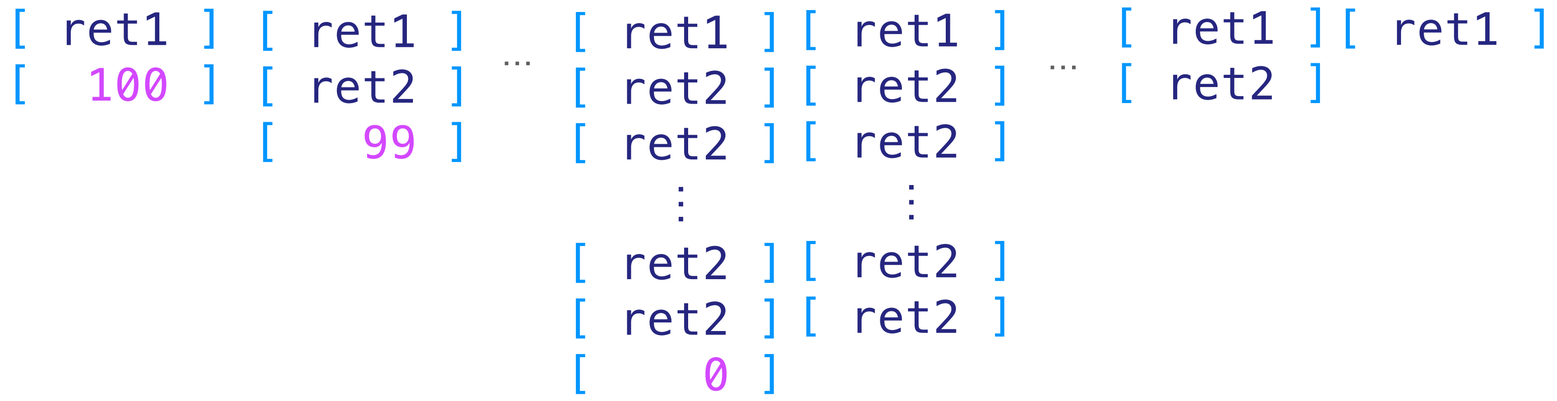
```
(Label 'f)
(Mov 'rax (Offset 'rsp 0))
(Add 'rsp 8) ; pop x
(Cmp 'rax 0)
(Je 'done)
(Sub 'rax (value->bits 1))
(Lea 'r8 'ret2)
(Push 'r8)   ; push return
(Push 'rax)  ; push argument
(Jmp 'f)
(Label 'ret2)
(Label 'done)
(Ret)
```

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

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

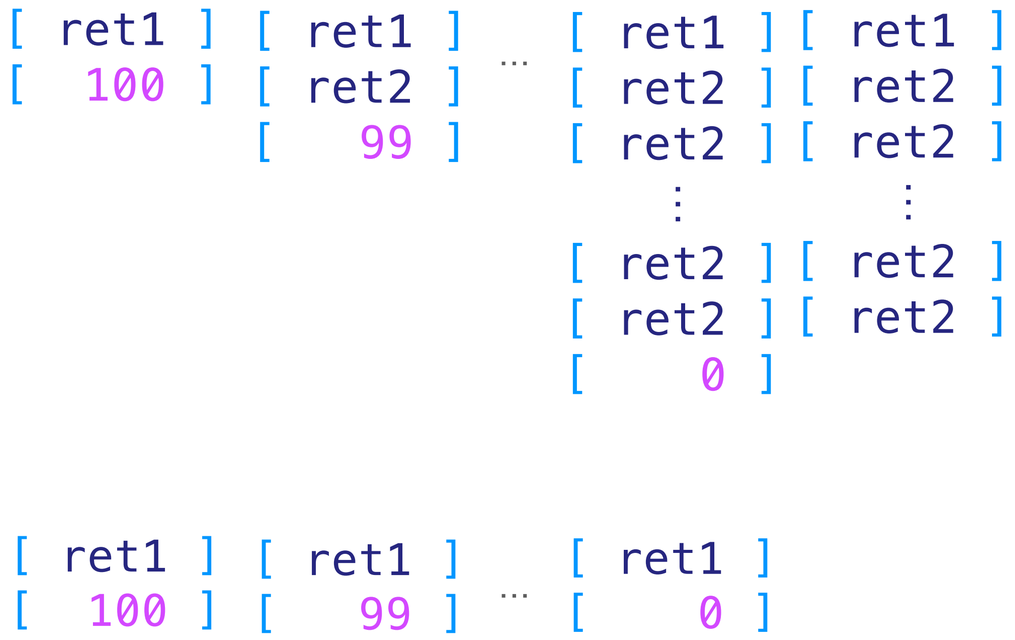
```
;; Morally:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

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

Revised view of the stack




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

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

Returning just to return



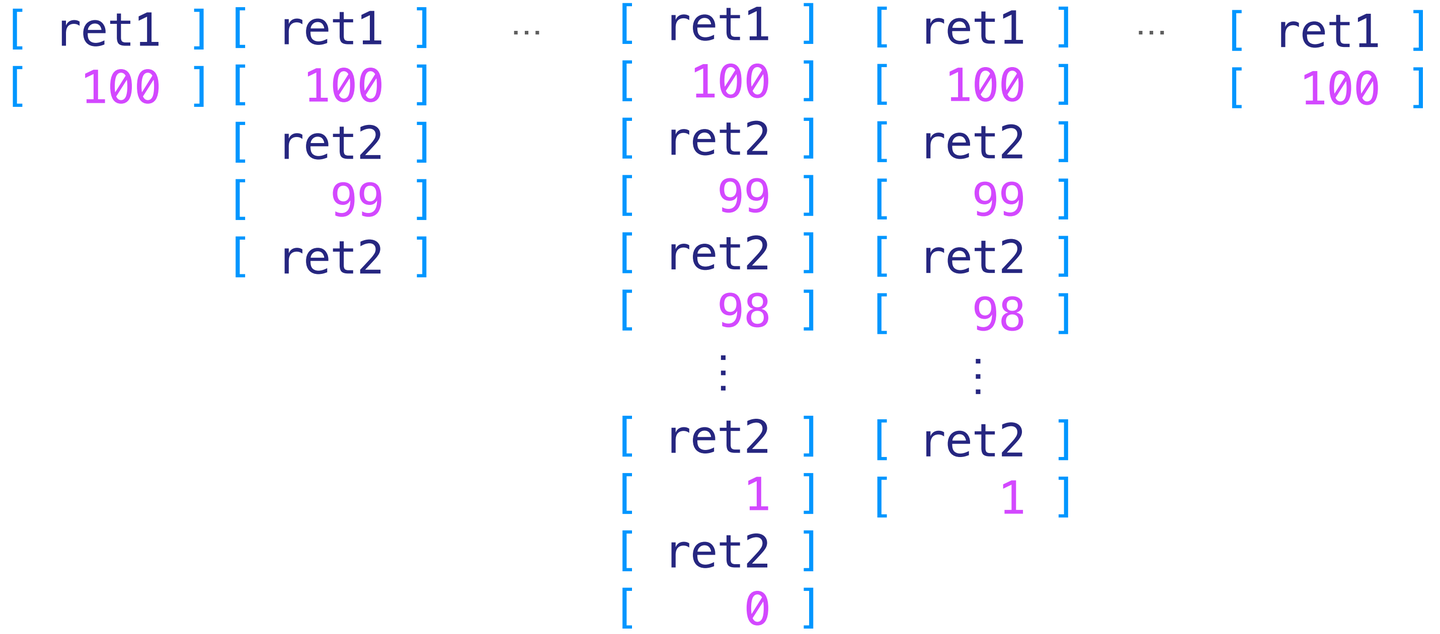

```
;; Revised:
(seq (Mov 'rax (value->bits 100))
     (Lea 'r8 'ret1)
     (Push 'r8)
     (Push 'rax)
     (Jmp 'f)
     (Label 'ret1)
     (Ret)

     (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)
```

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

What if we just didn’t?

Tail calls (asymptotically) save *SPACE*

Non-tail call:



Tail call:

(saving space may save time too)

Key idea

Some calls don’t need to be returned to

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

- Pop local environment
- Push arguments
- Jump

When can’t you do this?

When there’s work to do after the call

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

(f 100)
```

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

(g 100)
```

Sometimes you can rewrite to enable tail calls

So there's no work to do after the call

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

(g 100)
```

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

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

(g 100)
```

Sometimes you can rewrite to enable tail calls

So there's no work to do after the call

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

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

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

When does a call not need to be returned to?

When it occurs in tail position

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

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

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

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

When does a call not need to be returned to?

When it occurs in tail position

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

t? - is this expression in tail position

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

When does a call not need to be returned to?

When it occurs in tail position

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

t? - is this expression in tail position

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

When does a call not need to be returned to?

When it occurs in tail position

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

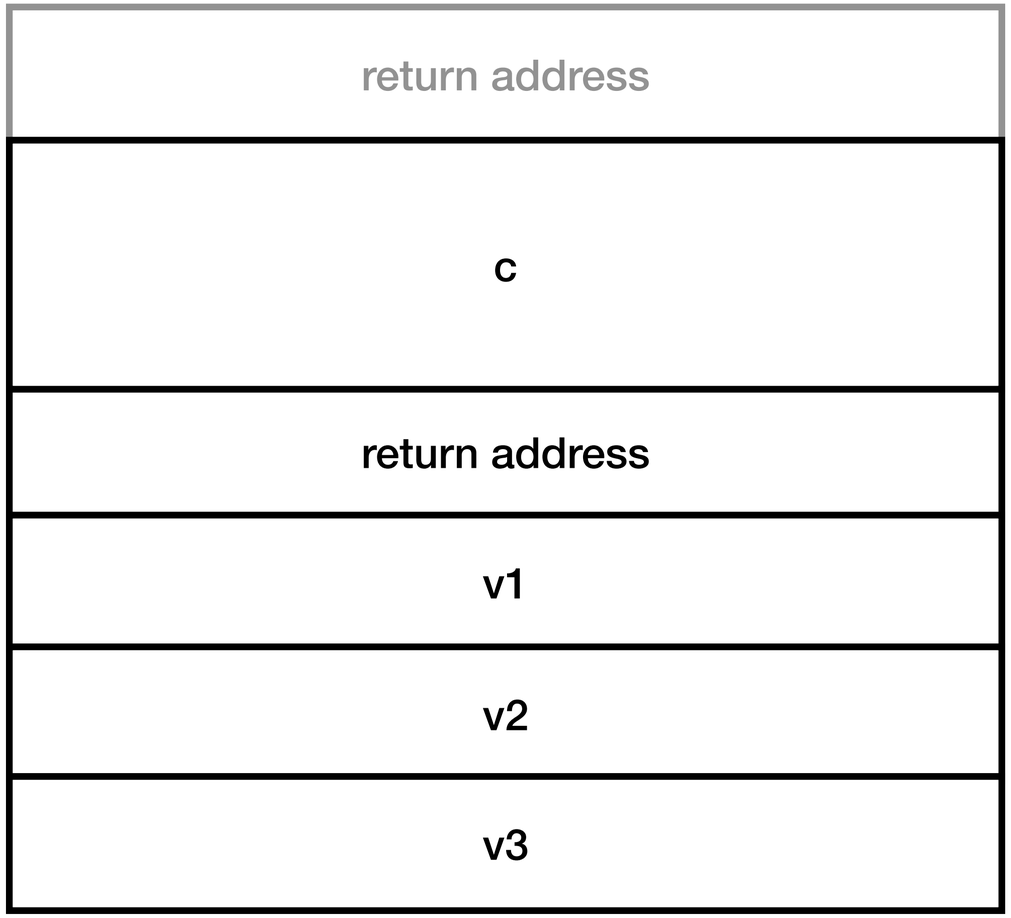
t? - is this expression in tail position

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

Non-tail calls

More work to do after call, so need to return

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

(seq
  (Lea rax '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

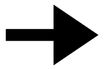
More work to do after call, so need to return

```
(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

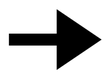
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

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

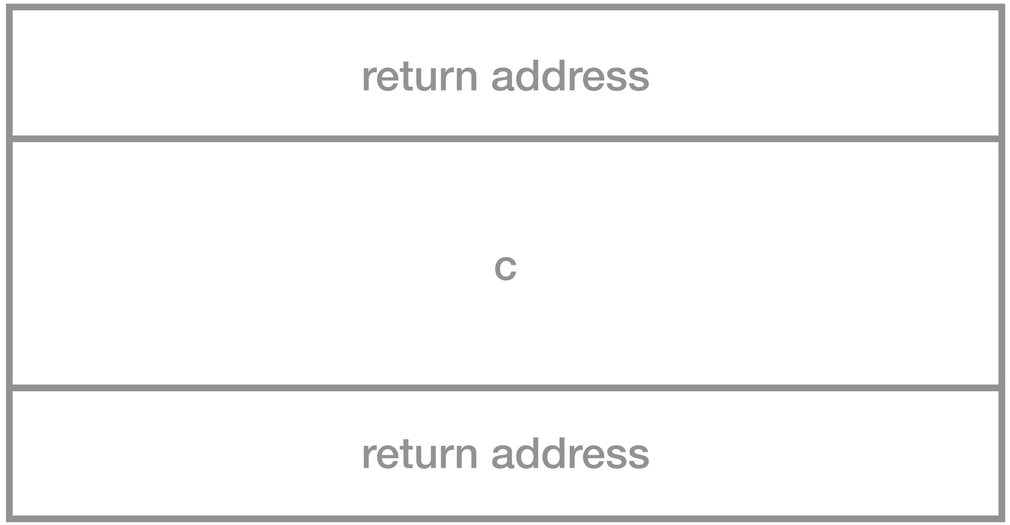


Non-tail calls

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))
```

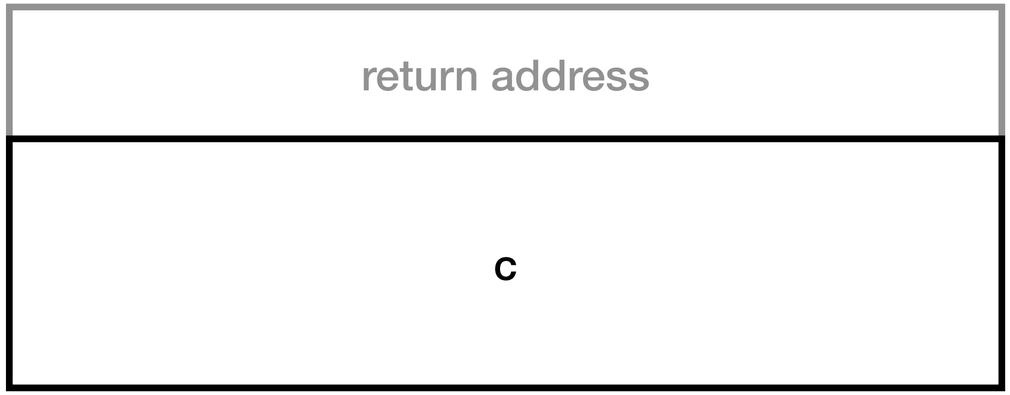




Non-tail calls

More work to do after call, so need to return

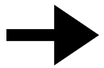
```
(compile-app-nontail 'f (list e1 e2 e3)) c)


(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))
```

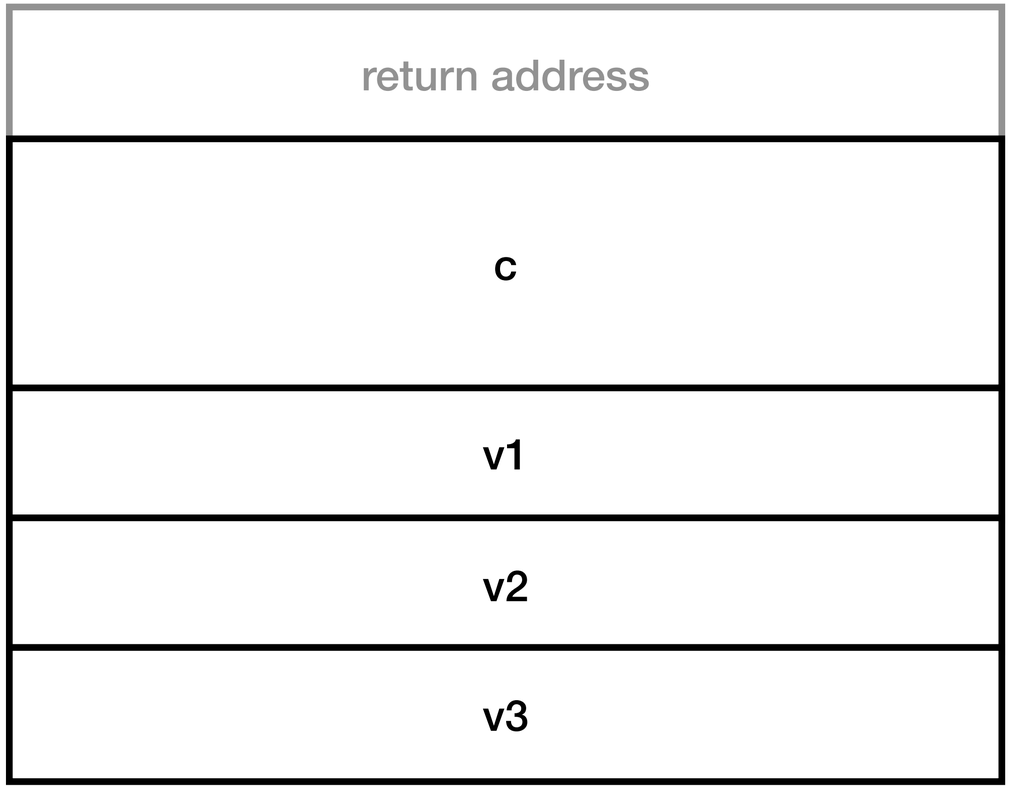


Tail calls (first attempt)

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))
```

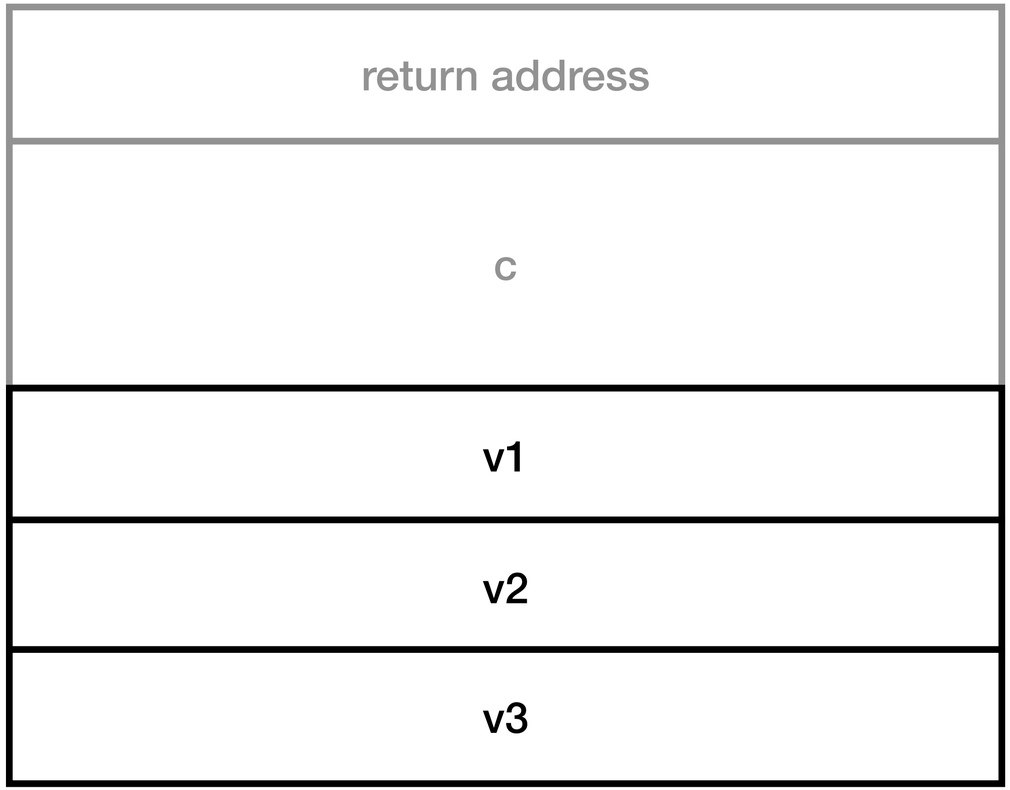




Tail calls (first attempt)

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

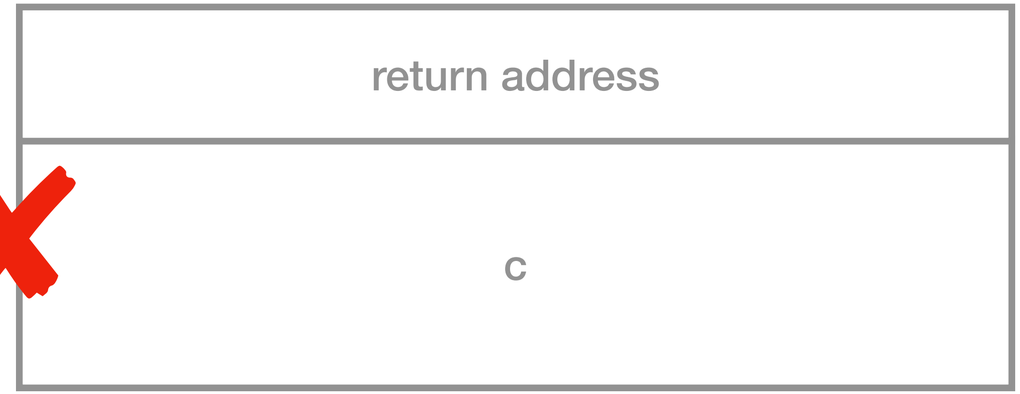
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

(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

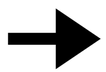
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))

(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))
```

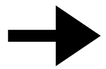


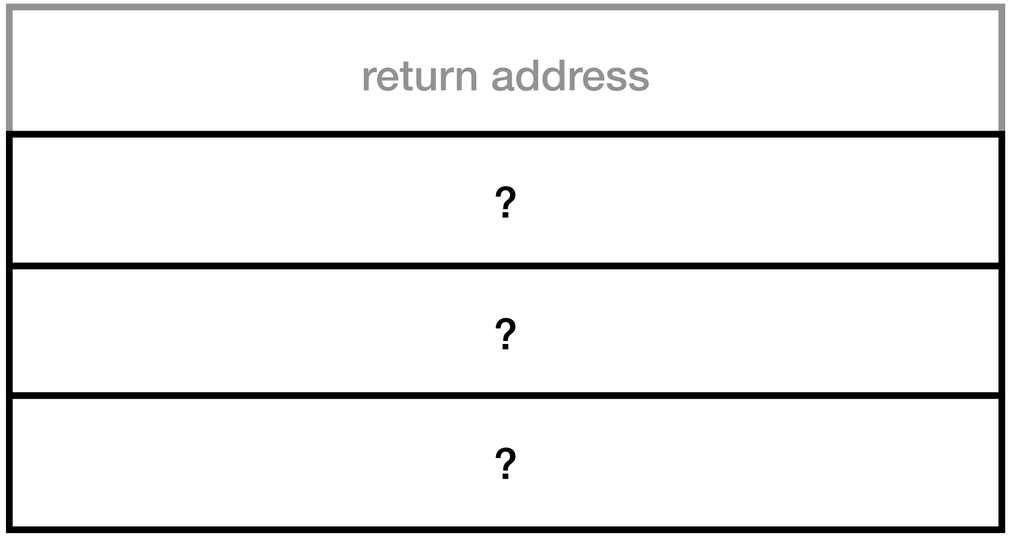
return address
v1
v2
v3

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))
```

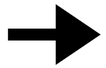


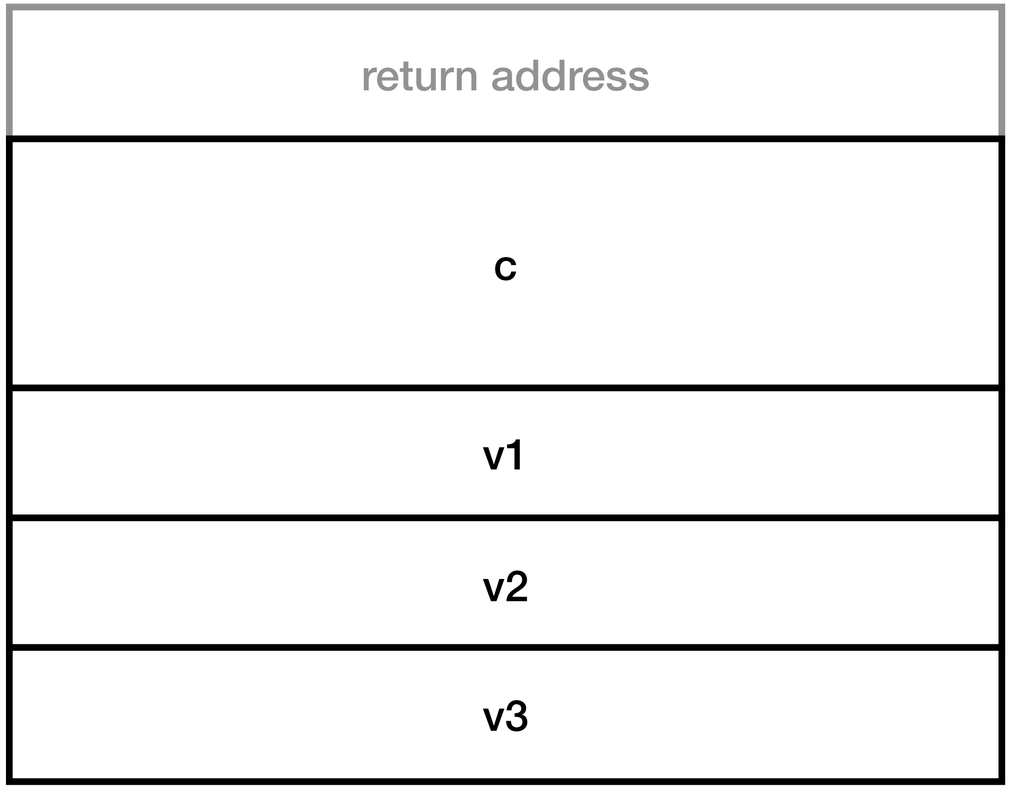


Tail calls (final attempt)

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))
```

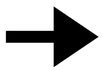


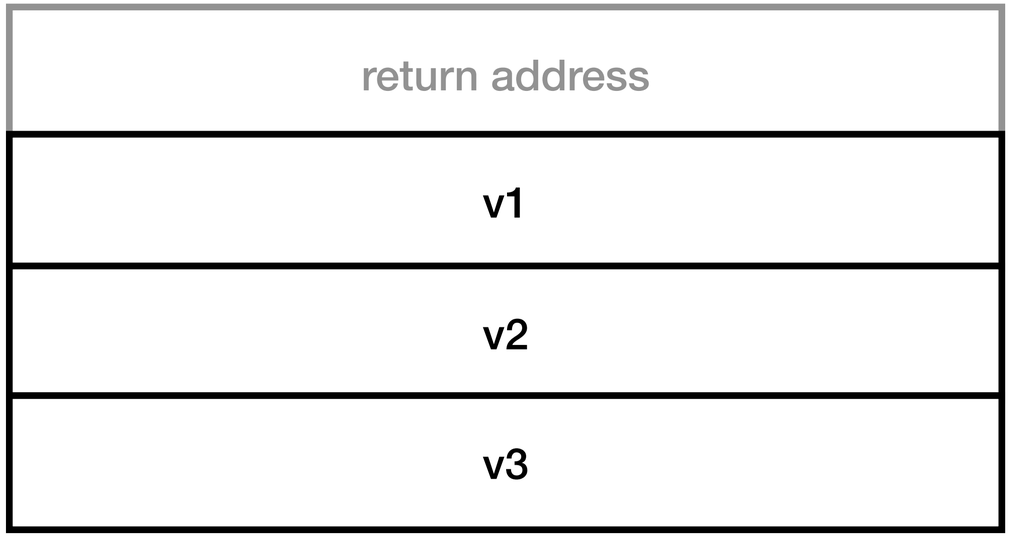


Tail calls (final attempt)

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))
```

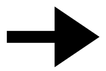


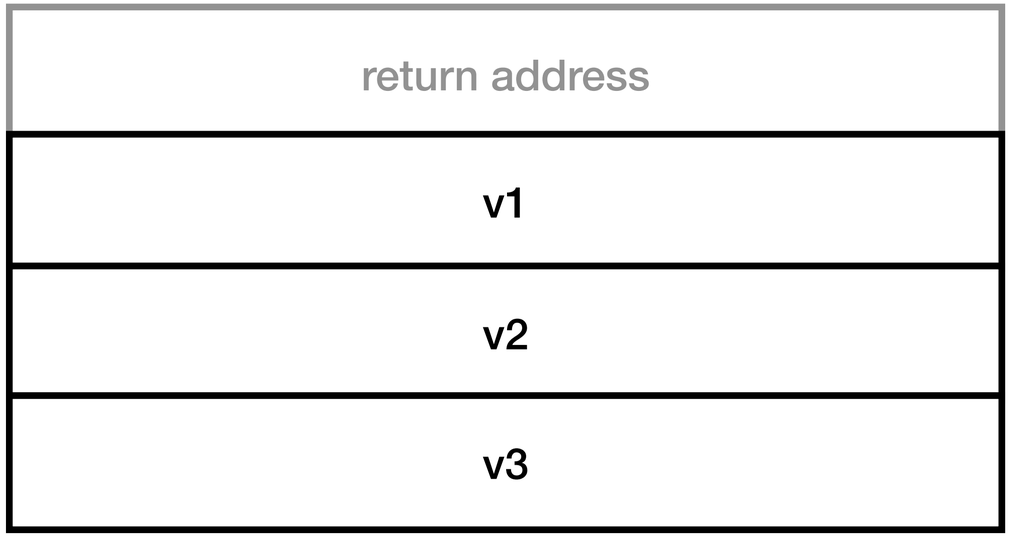


Tail calls (final attempt)

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 - 21 Nov 2024

Lambda the Ultimate

Announcements:

- Midterm 2 grades released (may need to be revised)

Today:

- One last observation on tail calls: the interpreter (what happened?)
- Lambda and first-class functions:
- interpreting with and without functions
- compiling (probably next time)

Syntactic matters

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

- Apps now have an arbitrary expression in function position
- Lambdas have an id (inserted by the parser) to “name” it

Syntactic matters

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

> (parse-e '(λ (x) x))
'#s(Lam lambda6795915 (x) #s(Var x))
> (parse-e '(λ (x) x))
'#s(Lam lambda6795947 (x) #s(Var x))
> (parse-e '(λ (x y z) x))
'#s(Lam lambda6795991 (x y z) #s(Var x))
>
```

Syntactic matters

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

```
> (parse-e '(f x))
'#s(App #s(Var f) (#s(Var x)))
> (parse-e '((if y f g) x))
'#s(App #s(If #s(Var y) #s(Var f) #s(Var g)) (#s(Var x)))
```

Semantic matters

New kind of value: function

```
;; type Value =
;; | Integer
;; | Boolean
;; | Character
;; | Eof
;; | Void
;; | '()
;; | (cons Value Value)
;; | (box Value)
;; | (string Character ...)
;; | (vector Value ...)
;; | ??? Functions
```

Semantic matters

New kind of value: function

```
;; Expr Env Defns -> Answer
(define (interp-env e r ds)
  (match e
    ;; ...
    [(App e es)
     (match (interp-env e r ds)
       ['err 'err]
       [f
        (match (interp-env* es r ds)
          ['err 'err]
          [vs
           '?????])])]
    [(Lam f xs e)
     '????]))
```

Lambda with Lambda

Interpreting lambda expressions using lambda expressions

```
;; type Value =
;; ...
;; | ([Listof Value] -> Value)
```

```
[(Lam _ xs e)
 (λ (vs) (interp-env e (zip xs vs) ds))]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (f vs)])])]
```

What we came up with last time

Any issues?

Lambda with Lambda

Interpreting lambda expressions using lambda expressions

```
;; type Value =
;; ...
;; | ([Listof Value] -> Value)
```

```
[(Lam _ xs e)
 (λ (vs)
   (interp-env e (append (zip xs vs) r) ds))]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (f vs)])])]
```

Save the environment

Any issues?

Lambda with Lambda

Interpreting lambda expressions using lambda expressions

```
;; type Value =
;; ...
;; | ([Listof Value] -> Value)
```

```
[(Lam _ xs e)
 (λ (vs)
   (if (= (length xs) (length vs))
       (interp-env e (append (zip xs vs) r) ds)
       'err))]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (if (procedure? f)
           (f vs)
           'err)])])]
```

With arity and type checking

Lambda with Closure

Interpreting lambda expressions without lambda expressions

```
;; type Value =
;; ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

```
[(Lam _ xs e)
 (λ (vs)
   (if (= (length xs) (length vs))
       (interp-env e (append (zip xs vs) r) ds)
       'err))]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (if (procedure? f)
           (f vs)
           'err)])])]
```

```
[(Lam _ xs e)
 (Closure xs e r)]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (match f
         [(Closure xs e r)
          ; check arity matches
          (if (= (length xs) (length vs))
              (interp-env e (append (zip xs vs) r) ds)
              'err)]
         [_ 'err])])])]
```

Lambda-free interpreter

CMSC 430 - 26 Nov 2024

Lambda the Ultimate

Announcements: ?

Lambda and first-class functions:

- review: interpreting functions without functions
- compiling
  - closed lambda expressions
  - open lambda expressions

Lambda with Closure

Interpreting lambda expressions without lambda expressions

```
;; type Value =
;; ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

```
[(Lam _ xs e)
 (λ (vs)
   (if (= (length xs) (length vs))
       (interp-env e (append (zip xs vs) r) ds)
       'err))]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (if (procedure? f)
           (f vs)
           'err)])])]
```

```
[(Lam _ xs e)
 (Closure xs e r)]
[(App e es)
 (match (interp-env e r ds)
   ['err 'err]
   [f
    (match (interp-env* es r ds)
      ['err 'err]
      [vs
       (match f
         [(Closure xs e r)
          ; check arity matches
          (if (= (length xs) (length vs))
              (interp-env e (append (zip xs vs) r) ds)
              'err)]
         [_ 'err])])])]
```

Lambda-free interpreter

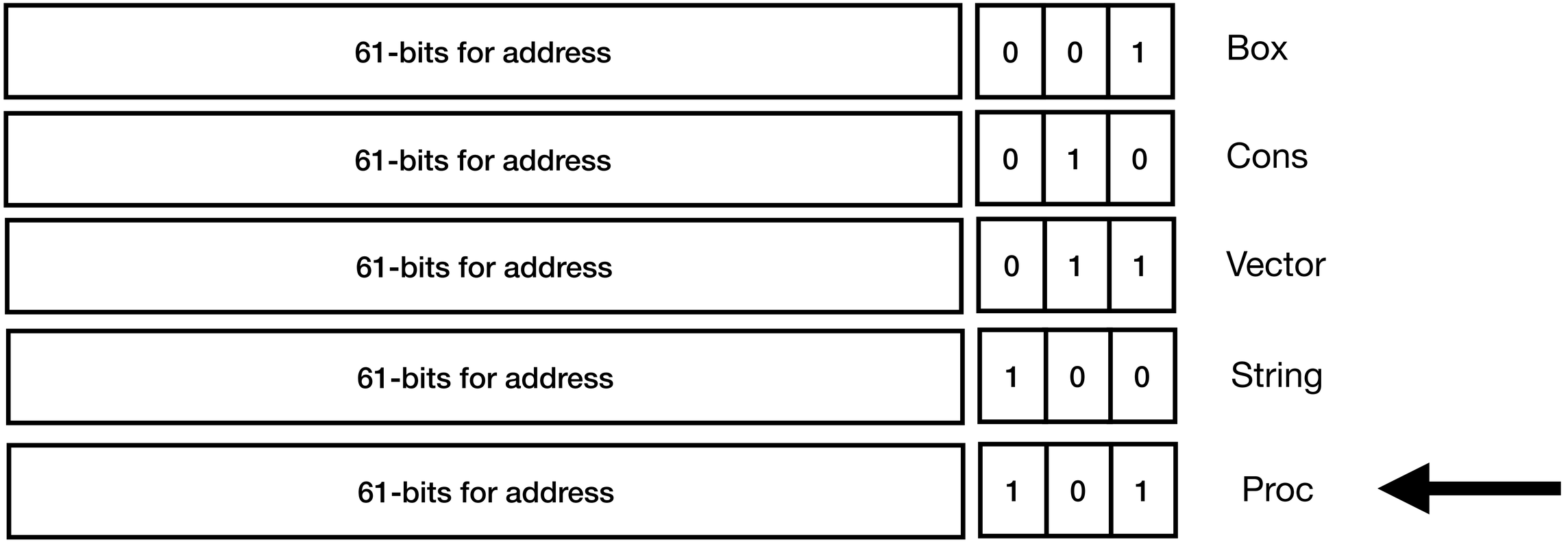
Semantic matters

Functions are a new kind of value

```
;; type Value =
;; | Integer
;; | Boolean
;; | Character
;; | Eof
;; | Void
;; | '()
;; | (cons Value Value)
;; | (box Value)
;; | (string Character ...)
;; | (vector Value ...)
;; | ??? Functions
```

Encoding pointer values (Loot)

Type tag in least significant bits



Compiling (closed) lambda expressions

Lambda expressions define functions

Compile (closed) lambda as definition

Idea: every lambda expression in the program defines a function:

```
;; Id [Listof Id] Expr -> Asm
(define (compile-lambda-define f xs e)
  (seq (Label (symbol->label f))
       (compile-e e (reverse xs) #t)
       (Add rsp (* 8 (length xs))) ; pop env
       (Ret)))
```

This is essentially compile-define

Lambda expressions define functions

Grab all the lambda expressions and compile function definitions

Lambda expressions evaluate to function values

Compile (closed) lambda as an expression

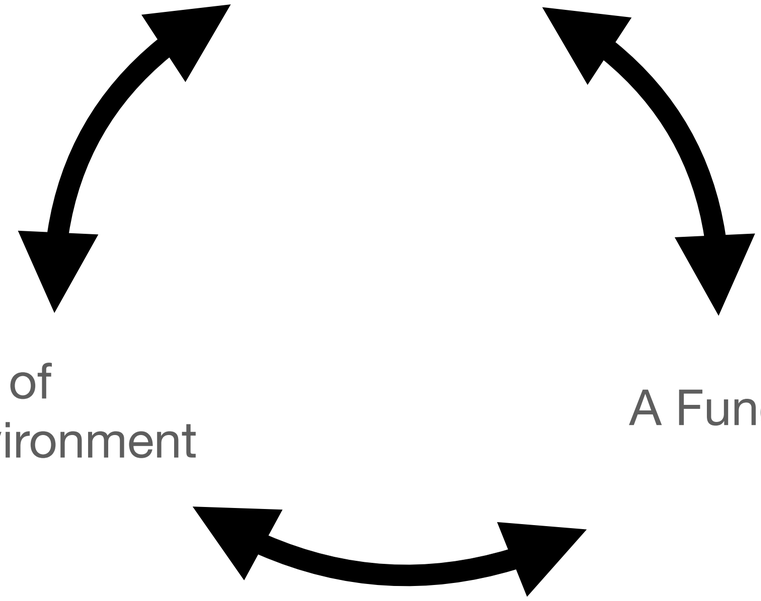
```
;; Id [Listof Id] Expr CEnv -> Asm
(define (compile-lam f xs e c)
  (seq (Lea rax (symbol->label f))
       (Mov (Offset rbx 0) rax)
       (Mov rax rbx)
       (Xor rax type-proc)
       (Add rbx 8)))
```

Write single word of memory that contains address of function's code

Evaluates to tagged pointer to that memory

The three facets of first-class functions

Calling of a function: Unpack the data structure and
install the values on the stack



An Expression: Allocation of
data structure packaging up environment

A Function Definition: Code for executing
the body of the function

CMSC 430 - 3 Dec 2024

Lambda the Ultimate

Announcements:

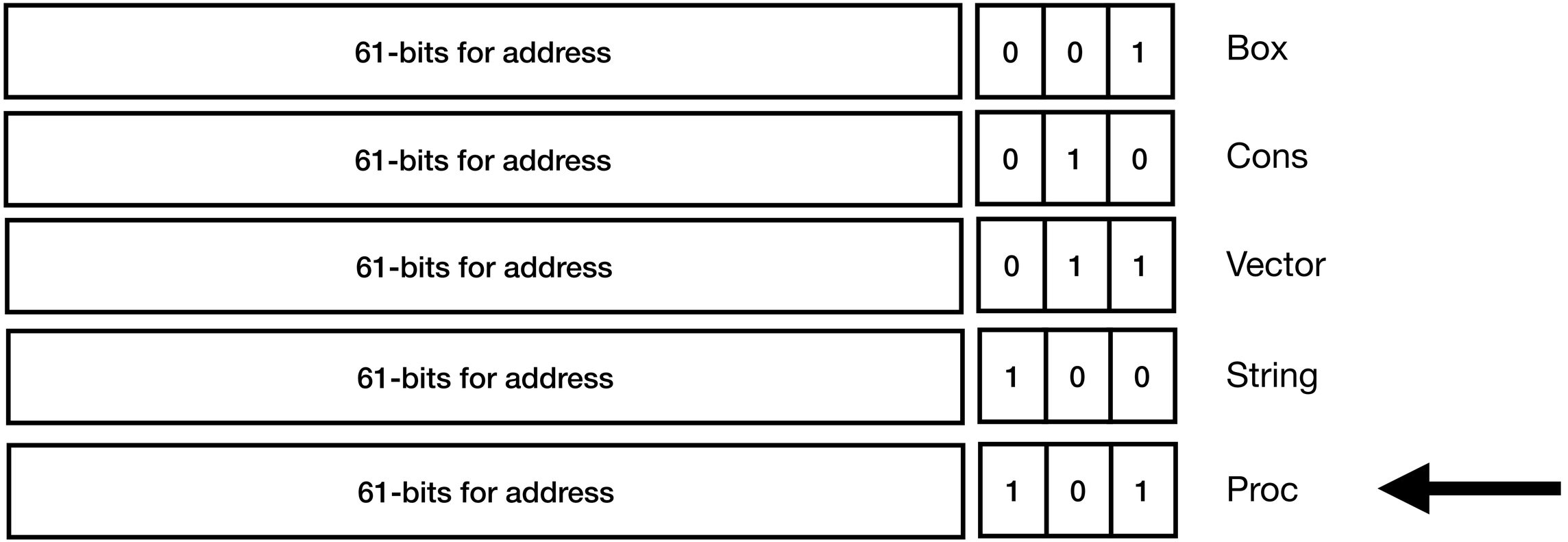
- Assignment 5: possible to submit late for 10% penalty
- Final project: Saturday, December 14, 12:30PM
- Course Experiences open until 12/10:
  - 85% participation: free quiz
  - 95% participation: two free quizzes

Lambda and first-class functions:

- compiling
  - review: closed lambda expressions
  - open lambda expressions

Encoding pointer values (Loot)

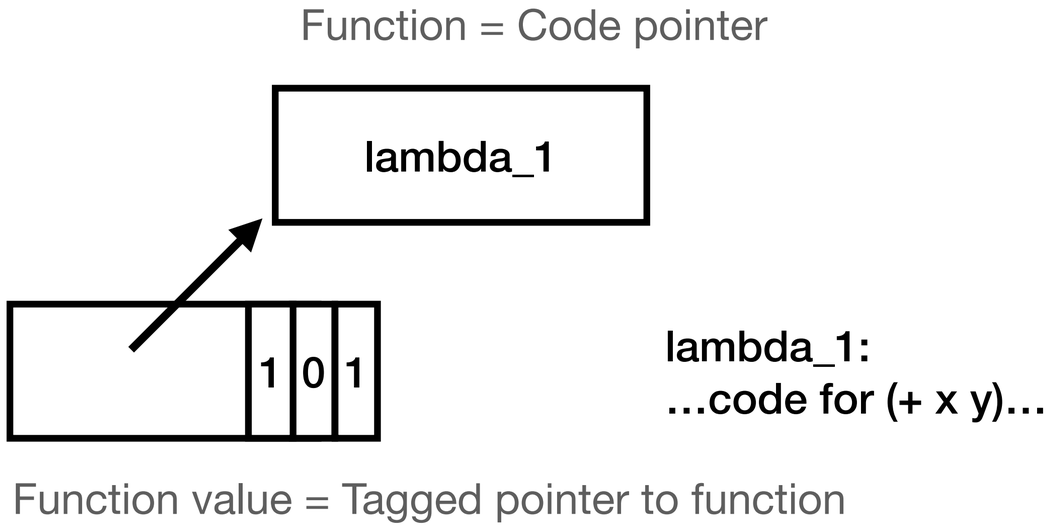
Type tag in least significant bits



Representing (closed) function values

Tagged pointer to memory that holds a code pointer

```
(λ (x y)
   (+ x y))
```



Lambda expressions define functions

First take: compile a function definition for each lambda

```
;; Lam -> Asm
(define (compile-lambda-define l)
  (match l
    [(Lam f xs e)
     (let ((env (reverse xs)))
       (seq (Label (symbol->label f))
            (compile-e e env #t)
            (Add rsp (* 8 (length env))) ; pop env
            (Ret)))]))
```

Assumes closed!!

Lambda expressions evaluate to function values

First take: lambda expressions evaluate to tagged pointers to code label

Assumes closed!!

```
;; Id [Listof Id] Expr CEnv -> Asm
(define (compile-lam f xs e c)
  (seq (Lea rax (symbol->label f))
       (Mov (Offset rbx 0) rax)
       (Mov rax rbx) ; return value
       (Xor rax type-proc)
       (Add rbx 8)))
```

Function calls made indirectly

First take: eval function, untag, dereference, and jump

```
;; Id [Listof Expr] CEnv -> Asm
(define (compile-app-nontail f es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es es (cons #f c))
         (Jmp f)
         (Label r))))
```

Compiler for Jig

```
;; Expr [Listof Expr] CEnv -> Asm
(define (compile-app-nontail e es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es (cons e es) (cons #f c))
         (Mov rax (Offset rsp (length es)))
         (assert-proc rax)
         (Xor rax type-proc)
         ; fetch the code label
         (Mov rax (Offset rax 0))
         (Jmp rax)
         (Label r))))
```

Should be 8 * (length es)

Whoops — broke function call protocol with an extra argument

Aside: What if we did the function last?

Could solve our problem and be more efficient!

```
;; Expr [Listof Expr] CEnv -> Asm
(define (compile-app-nontail e es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es es (cons #f c))
         (compile e (append (make-list (add1 (length es)) #f) c))
         (assert-proc rax)
         (Xor rax type-proc)
         ; fetch the code label
         (Mov rax (Offset rax 0))
         (Jmp rax)
         (Label r))))
```

Pros:
- saves one word fewer word on the stack
- No need to fetch the function from the stack — it's in rax already!
- No longer sends one too many arguments to the function

Cons:
- Broken (w.r.t. our spec)

```
#lang racket
((begin (write-byte 97) (λ (x y) x))
 (begin (write-byte 98) #t)
 (begin (write-byte 99) #f))
```

Aside: What if we did function last?

Could solve our problem and be more efficient!

We could change the spec to go right to left….

```
OCaml version 4.14.0
Enter #help;; for help.

# ((Printf.printf "a"; (fun x y -> x))
   (Printf.printf "b"; true)
   (Printf.printf "c"; false));;
cba- : bool = true
```

OCaml: designed by compiler hackers, for compiler hackers!

OCaml's philosophy: don't rely on order of evaluation in function application; for a particular order, be explicit, e.g.:

```
OCaml version 4.14.0
Enter #help;; for help.

# let f1 = Printf.printf "a"; (fun x y -> x) in
  let v1 = Printf.printf "b"; true in
  let v2 = Printf.printf "c"; false in
  f1 v1 v2;;
abc- : bool = true
```

Lambda expressions define functions

Second take: every function takes implicit extra argument

Assumes closed!!

```
;; Lam -> Asm
(define (compile-lambda-define l)
  (match l
    [(Lam f xs e)
     (let ((env (reverse (cons #f xs))))
       (seq (Label (symbol->label f))
            (compile-e e env #t)
            (Add rsp (* 8 (length env))) ; pop env
            (Ret)))]))
```

Representing functions

Functions are a new kind of value

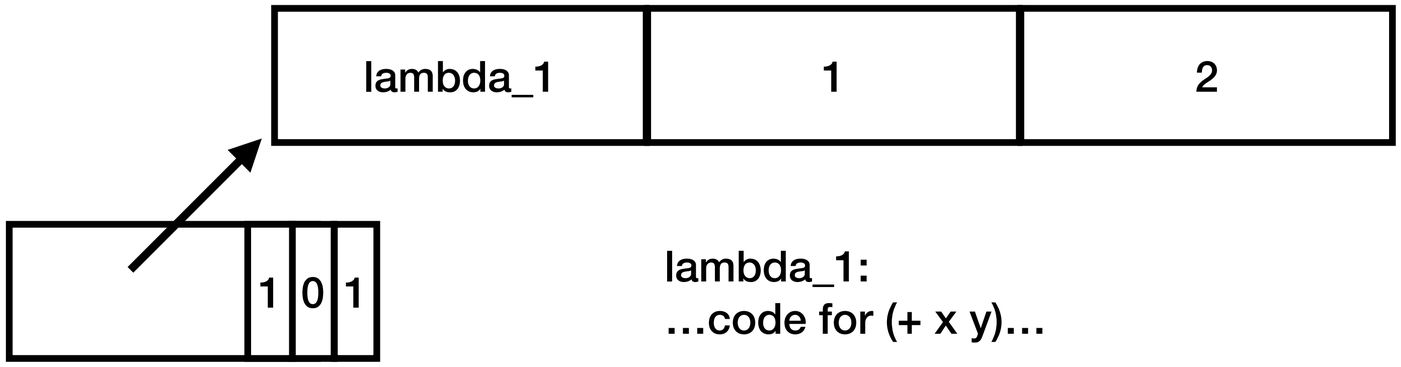
```
;; type Value =
;; ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

Can be used to define a function in assembly

Can be represented as an array of values

```
(let ((x 1))
  (let ((y 2))
    (λ (z) (+ x y))))
```

Closure = Code pointer + run-time environment



Function value = Tagged pointer to closure

CMSC 430 - 5 Dec 2024

430 Wrapped

Quickly:

- Symbols
- Structures

Outlaw:

- Putting it all together
- Standard Library
- Added primitives
- Primitives as Functions
- More I/O
- Self compilation

CMSC 430 - 22 Nov 2022

Static data, symbols, and interned strings

Announcements - m2 autograder results 4/5 done and available,

final project submission opens today (no autograde yet)

Static data

Statically interned string literals

Symbols (dynamically interned strings)

CMSC 430 - 29 Nov 2022

Structures, closing in on self-hosting

Announcements -

Structure definitions

Structure instances

A generic approach to implementing structures

Program syntax

A Program is a sequence of definitions, followed by an expression.

New kind of definition: `(struct foo (x y z))`

Structure Definitions

`(struct foo (x y z))` generates:

`foo?` : predicate

`foo` : constructor

`foo-x` : accessors

`foo-y`

`foo-z`

Introduces a new type of value, disjoint from existing types.

Compiling foos (a sketch)

Suppose we had `(struct foo (x y z))`.

How could you implement?

Designate a new pointer type for `foo`s.

Add new primitives: `foo?`, `foo`, `foo-x`, `foo-y`, `foo-z`

`foo`: allocate 3 words, write args to memory, return foo-tagged pointer

`foo?`: is the arg a foo-tagged pointer?

`foo-{x,y,z}`: assert arg is foo-tagged pointer, deref {1st,2nd,3rd} word

Compiling foos: Problems

Run-time code won’t know about foo type

We’ll run out of type tags pretty quickly

How can compiler know about foo primitives before seeing program?

A different tack: a single, generic structure

```
make-struct : Symbol Value … -> StructVal
struct? : Symbol Value -> Boolean
struct-ref : Symbol Nat StructVal -> Value
```

Example:

```
(let ((x (make-struct 'foo 1 2 3))
  (begin (struct? 'foo x) ; => #t
         (struct-ref 'foo x 0))) ; => 1
```

A different tack: a single, generic structure

```
make-struct : Symbol Value … -> StructVal
struct? : Symbol Value -> Boolean
struct-ref : Symbol Nat StructVal -> Value
```

Another example:

```
(let ((x (make-struct 'foo 1 2 3))
  (begin (struct? 'foo x) ; => #t
         (struct-ref 'bar x 0))) ; => error
```

Implementing generic structs

One struct type to rule them all

Designate a new pointer type for `structs`.

`make-struct : Symbol Value … -> StructVal`

Given n values, allocate n+1 words, write symbol and values, return struct-tagged pointer

`struct? : Symbol Value -> Boolean`

Is the arg a struct-tagged pointer? If yes, is the first word equal to given symbol?

`struct-ref : Symbol Nat StructVal -> Value`

Assert arg is a struct-tagged pointer and first word is equal to given symbol.

Given i, dereference the i+1 word.

But how does this solve the original problem?

`(struct foo (x y z))` generates:

`foo?` : predicate

`foo` : constructor

`foo-x` : accessors

`foo-y`

`foo-z`

```
(define (foo? x) (struct? 'foo x))
(define (foo x y z)
   (make-struct 'foo x y z))
(define (foo-x x) (struct-ref 'foo 0 x))
(define (foo-y x) (struct-ref 'foo 1 x))
(define (foo-z x) (struct-ref 'foo 2 x))
```

But how does this solve the original problem?

Parser translates `(struct foo (x y z))` into:

```
(define (foo? x) (struct? 'foo x))
(define (foo x y z)
   (make-struct 'foo x y z))
(define (foo-x x) (struct-ref 'foo 0 x))
(define (foo-y x) (struct-ref 'foo 1 x))
(define (foo-z x) (struct-ref 'foo 2 x))
```

CMSC 430 - 1 Dec 2022

Bootstrapping a compiler

Announcements -

Putting it all together

Standard Library

Added primitives

Primitives as Functions

I/O