

CMSC 430 - 1 June 2026

Introduction to Introduction to Compilers

Logistics

Overview of Compilers

Intro to Racket

Logistics

Learning objectives

Logistics

You should be able to answer:

- Where is the course web page?
- How do I find the syllabus?
- What is the basic structure of this course?
- How do I answer my own questions about the course?
- What strategies will help me succeed?

About me

David Van Horn (he/him)

You can call me: DVH or Professor Van Horn

Research area: Programming Languages

Background: UVM (B.S.), Brandeis (Ph.D.), Northeastern (Research Prof.)

Professor at UMD since 2014

Designed CMSC 430 and its materials

Contact: dvanhorn@cs.umd.edu



Compilers comes at you fast

Course logistics

Lectures every Weekday (except holidays, exams)

Assignments: 10, due Mon/Thur

Exams: 3 - 6/12, 6/26, 7/10

Several surveys and quizzes (on ELMS)

Course is very cumulative; building essentially one program all semester

Summer Compilers comes at you faster

Course logistics

Unrelenting pace

Plan to spend several hours a day on 430

Every two weekdays is a WEEK of material

First exam is **TWO WEEKS**

Semester at a glance

Course logistics

Week 1-2	Assignment 1-3	Exam 1
Week 3-4	Assignment 4-7	Exam 2
Week 5-6	Assignment 8-10	Exam 3

Grades

Course logistics

Assignments	30%
Quizzes & surveys	20%
Exams	50%

Key resources

Course logistics

- Class web page (syllabus, assignments, course notes, slides)
`https://www.cs.umd.edu/class/summer2026/cmsc430/`
- ELMS (announcements, recordings, grades)
- Piazza (communication, discussion)
- Gradescope (submissions)

Day to day: lectures

How to CMSC 430

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

Day to day: studying

How to CMSC 430

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

Day to day: assignments

How to CMSC 430

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

Day to day: exams

How to CMSC 430

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

* If you have already mastered the material!

Day to day: office hours

How to CMSC 430

- Use online forums (Piazza)
- Prof available 30 minutes before class for public discussion
- Email to set up private session
- TA office hours scheduled will be announced soon

Day to day: feedback

How to CMSC 430

- Feedback is welcome!
- Send email or Piazza post if OK with non-anonymous
- Anonymous feedback:
<https://forms.gle/99yTz7HVfopCaDMz9>

Advice from future you

Quotes from Fall 2025 End of the Road Survey

- Start assignments early and read the spec carefully. Understanding the concepts first saves a lot of time later.
- Start projects as early as you possibly can, and do the practice midterms before the real midterms. Spend more time rereading and understanding the notes outside of class.
- Small misunderstandings can compound quickly, so staying ahead is much more important than trying to catch up later.
- Spending time to design effective tests is very important, and helps you understand how your compiler works, what might need to be changed, and potential edge cases.
- Step through the assembly generated by the compiler if you're stuck.
- Spend less time trying to do everything in your head and more time making memory diagrams, writing out your thought process, and manually tracing your code.
- Just because a portion of the compiler/interpreter is written for you doesn't mean that you shouldn't be reading them to better understand what YOU need to write.
- It's going to be a grind, but just keep sticking with it.

Compilers

Learning objectives

Compilers

You should be able to answer:

- What is a programming language?
- What makes a programming language:
 - general-purpose vs domain-specific?
 - high-level vs low-level?
 - statically-typed vs dynamically typed vs untyped?
- Why does programming language design matter?
- What is a compiler?
- What does it mean for a compiler to be correct?
- Why does compiler correctness matter?

What is a programming language?

No, really, I'm asking...

A set of instructions that the computer can understand to execute

Abstracted language that is eventually translated down to binary, higher level is closer to natural language

Let's you implement algorithms

What is a programming language?

A programming language is a formal language for **expressing computations**, with

- rules for what programs look like (**syntax**), and
- rules for what those programs mean or do when run (**semantics**).

What is a programming language?

A general purpose programming language is intended to express computations across many domains.

Examples: C, Java, Rust, OCaml, Racket, JavaScript, WASM

A domain-specific programming language targets a particular domain.

Examples: PostScript, SQL, Makefiles, Shader languages

What is a programming language?

A *high-level* programming language is designed to express computations without mentioning machine details.

Examples: Java, Racket, Haskell, OCaml, Python

Often have:

- Automatic memory management
- Strong safety guarantees
- Abstractions based in mathematics: functions, relations, types

What is a programming language?

A *low-level* programming language is designed to express computations while mentioning machine details.

Examples: C, Rust, Assembly, WASM

Often have:

- Explicit control over machine state
- Security vulnerabilities
- An imperative view of computations

What is a programming language?

A *statically-typed* programming language has syntactic rules in addition to grammatical rules for forming programs. These rules ensure good properties of programs before they are run.

Examples: Java, OCaml, Rust

Often have:

- Strong safety guarantees
- Classes of errors eliminated by the type system
- Good programs ruled out by the type system

What is a programming language?

A dynamically-typed programming language forgoes syntactic rules and instead imposes checks at run-time. These checks ensure good properties of programs when they are run.

Examples: Racket, Python, JavaScript

Often have:

- Strong safety guarantees
- Run-time type errors
- Overhead associated with run-time checking

What is a programming language?

None of these have precise meanings, they are cultural:

- dynamically-typed, statically-typed
- high-level, low-level
- general-purpose, domain-specific

Real languages occupy design points along a spectrum

What isn't a programming language?

- Musical notation: syntax and semantics, not computational
- Pseudocode: describes computations, but lacks formal syntax and semantics
- Markdown: syntax and semantics for document structure, not computational
- JSON: syntax and semantics for data values, but not computational

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...

Translate from one programming language to another

Traditionally compiling to machine code

What is a compiler?

A compiler is a program that translates programs written in one language into programs written in another. A compiler is one possible implementation strategy for a language. Often compilers translate from higher-level languages to lower-level languages. Often compilers improve the efficiency of a program but doing work at compile time in order to avoid work at run-time. A compiler must preserve the meaning of the original program.

Languages are not compiled. Programs are compiled.

Design & implementation of programming languages

Quick overview

We're going to build a programming language

- with modern features
- specified via interpreter
- implemented via compiler
- paying close attention to correctness

Design & implementation of programming languages

Quick overview

We're going to build a programming language

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

Design & implementation of programming languages

Quick overview

We're going to build a programming language

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

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

Racket

Learning objectives

Racket

You should be able to answer:

- What is the syntax of Racket?
- What is the computational model of Racket?
- What are the values of Racket?
- How is Racket similar to OCaml?
- What is the subset of Racket we will need for this course?
- What will we be using Racket for?

Ocaml to Racket

Racket = OCaml - Types - Syntax

Download and install Racket

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

Symbols

An atomic string-like datatype

Symbols are a useful datatype for representing enumerations

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

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

Lists and pairs in Racket vs OCaml

Constructors in OCaml

OCaml lists:

- `[]` : α list
- `::` : $\alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$
- `[. ; . ; . ; ...]` convient notation for lists

OCaml pairs (and tuples):

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

Pairs and lists: **fundamentally different things**

Lists and pairs in Racket vs OCaml

Constructors in Racket

Racket lists:

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

Racket pairs (and tuples):

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

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)

`hd, tl` functions for lists

Lists and pairs in Racket vs OCaml

Destructors in Racket

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

`car`, `cdr` functions for pairs

`first`, `rest` functions for lists

Literal pairs and lists

A notation for writing down compound literals

Lists of literals can be written using the quote notation:

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

Pairs of literals can be written using the quote notation:

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

Structures

Defining new record types

`(struct coord (x y z))` defines:

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

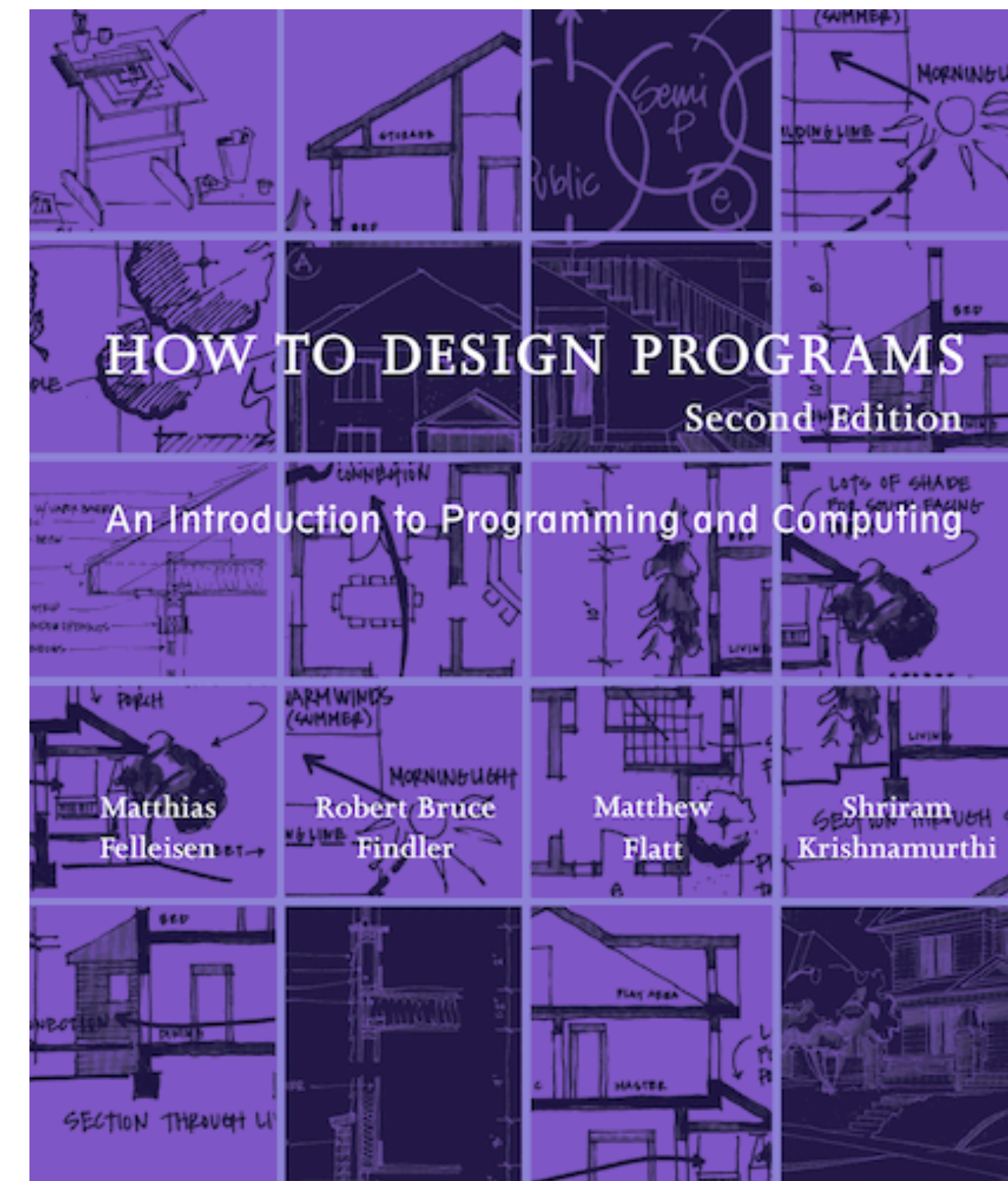
Systematic programming

Steps of systematic program design

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

Keys:

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



For next time

- Read syllabus
- Take ELMS survey & quiz
- Install Racket & langs package
- Read OCaml to Racket notes
- Read a86 notes

CMSC 430 - 2 June 2026

Our first compiler

Quick Racket review

Intro to a86

Our first compiler

- Abscond

CMSC 430 - 2 June 2026

Announcements

- Exam dates fixed on syllabus: 6/12, 6/26, 7/10
- Practice assignments 1 & 2: Racket, a86
- Video solution for practice assignment 1 (ELMS > Panopto)
- Assignment 1 & 2 released today
- New quiz released today, due by start of class tomorrow
- Make sure you're on Piazza & Gradescope

Racket

Learning objectives

Racket

You should be able to answer:

- What is the syntax of Racket?
- What is the computational model of Racket?
- What are the values of Racket?
- How is Racket similar to OCaml?
- What is the subset of Racket we will need for this course?
- What will we be using Racket for?
- What is the difference between a syntax error and run-time error?

Racket: Computational model

functional style

Machine state:

- Expression being reduced
- Substitute equals for equals
- Function application replaces parameter with argument in body

Computation:

- Reduce until reaching a value or error

Racket: Values

- numbers, booleans, characters, symbols, ...
- pairs, lists, vectors, boxes
- structures
- functions

Racket vs OCaml

Similarities

- Functional style
- Memory safety
- Garbage collected
- Shared lineage
- First-class functions, pattern matching, etc.

Racket vs OCaml

Differences

- Uniform prefix syntax
- No static type system
- Variable arity functions
- No automatic currying
- Pairs and (non-empty) lists made out of same constructor
- Quote

a86

Learning objectives

a86

You should be able to answer:

- What is the syntax of a86?
- What is the computational model of a86?
- What are the values of a86?
- What will we be using a86 for?

x86: Computational model

von Neumann style

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

Machine state:

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

Computation:

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

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 mem reg - memory write: copy contents of reg into memory
- Mov reg mem - memory read: copy contents of memory into reg
- ~~Mov mem1 mem2~~ - illegal instruction, have to go through a register
- Mem 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 int32 - push integer literal onto the stack
- Pop reg - pop top element of stack into reg
- Under the hood: these instructions manipulate the rsp register, which holds a pointer to the “top” of the stack
 - Push: decrement rsp, write to memory
 - Pop: read from memory, increment rsp

The rsp register

Special designated register that points to the stack

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

Reading the stack (without popping)

- Use memory reads via `rsp` to read elements of the stack without doing a `Pop`
- `Mov reg (Mem rsp 0)` - copies first element of stack into `reg`
- `Mov reg (Mem rsp 8)` - copies second element of stack into `reg`
- `Mov reg (Mem 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 (Mem rsp 0) reg` - copies `reg` into first element of stack
- `Mov (Mem rsp 8) reg` - copies `reg` into second element of stack
- `Mov (Mem rsp  $n \cdot 8$ ) reg` - copies `reg` into $(n+1)$ th element of stack

Cmp

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

J*: conditional jumps

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

Cmov*: conditional moves

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

(De)Allocating on the stack

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

Push, Pop (for real)

- Push reg =
Sub rsp 8
Mov (Mem rsp 0) reg
- Pop reg =
Mov reg (Mem 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
Jump name
- Ret =
Add rsp 8
Jump (Mem rsp -8)
- Careful to always restore stack to original state before Ret, otherwise not returning where you intended!

Examples with the stack

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

Stack:

42
84
19

rax: ~~84~~ 42

Examples with the stack

- Push 42 ←
- Push 84 ←
- Push 19 ←
- Add 'rsp 8 ←
- Add 'rsp 8 ←
- Add 'rsp 8 ←

Stack:

42
84
19

rax: ?

Examples with the stack

- Push 42 ←
- Push 84 ←
- Push 19 ←
- Add 'rsp 24 ←

Stack:

42
84
19

rax: ?

Examples with the stack (memory view)

- Push 42 ←
- Push 84 ←
- Push 19 ←
- Add 'rsp 8 ←
- Add 'rsp 8 ←
- Add 'rsp 8 ←

Stack:

42
84
19

rax: ?

Examples with the stack (memory view)

- Push 42 ←
- Push 84 ←
- Push 19 ←
- Add 'rsp 24 ←

Stack:

42
84
19

rax: ?

Abseond

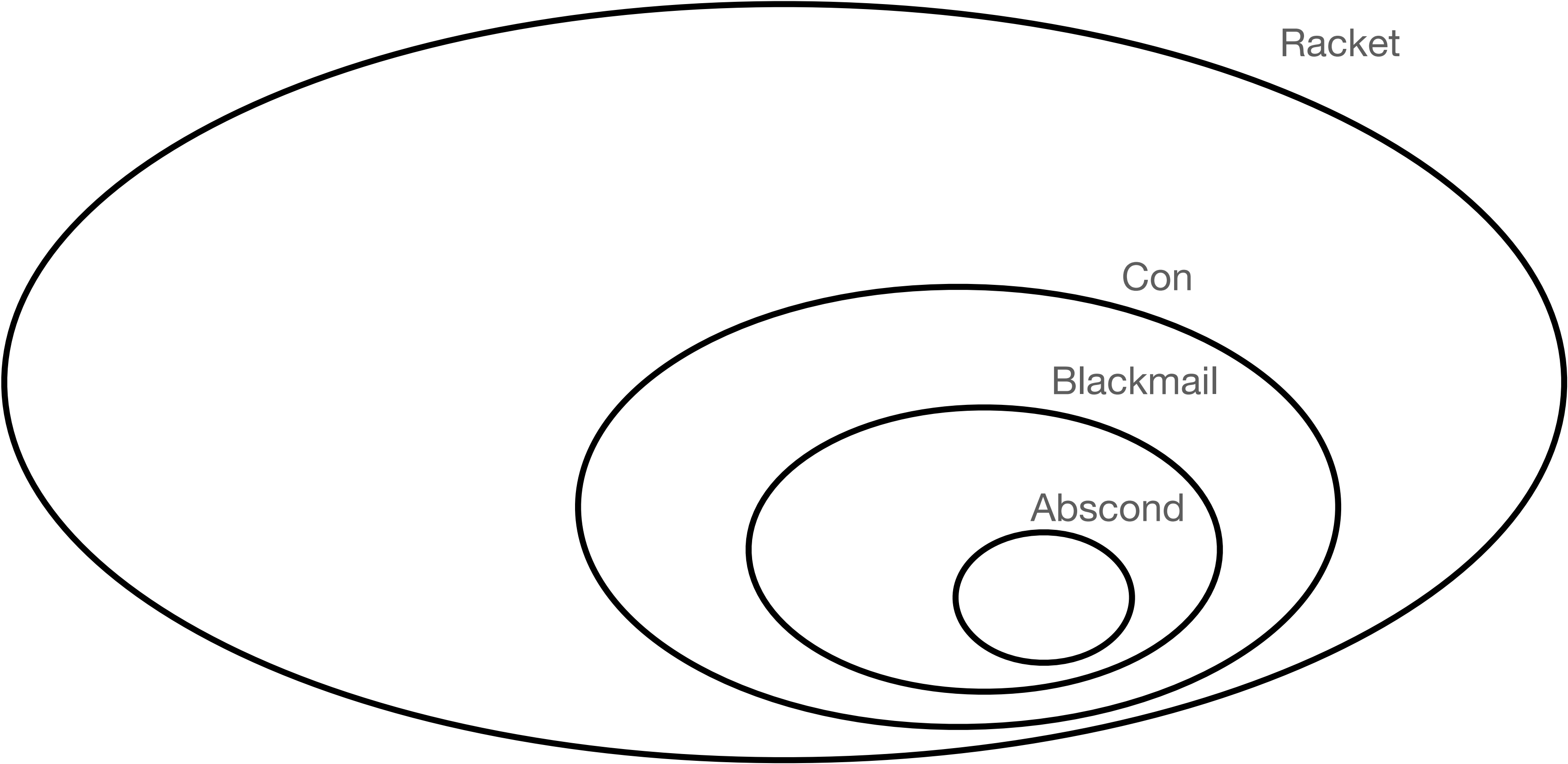
Learning objectives

Abscond

You should be able to answer:

- What is the structure of the Abscond compiler?
- How are Abscond programs represented abstractly?
- What invariant relates source programs and compiled code?

Language subsets



Example Abscond program

Concrete syntax

```
#lang racket
```

```
42
```

Example Abscond program

Abstract syntax

(Lit 42)

```
:: type Expr =  
:: | (Lit Integer)  
(struct Lit (n))
```


Example Abscond program

Parser constructs AST from string input

#lang racket
42

parse → (Lit 42)

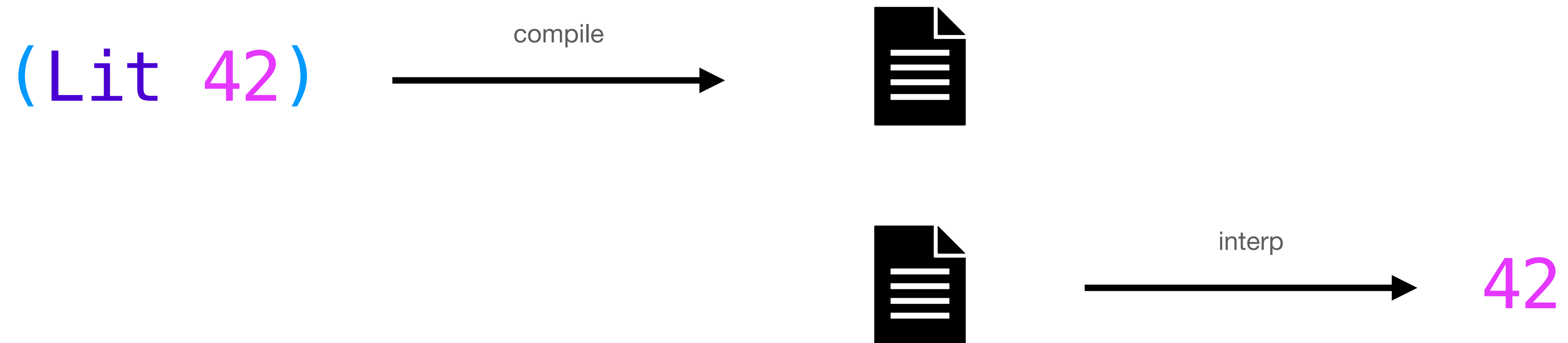
Example Abscond program

Interpreter computes meaning from AST



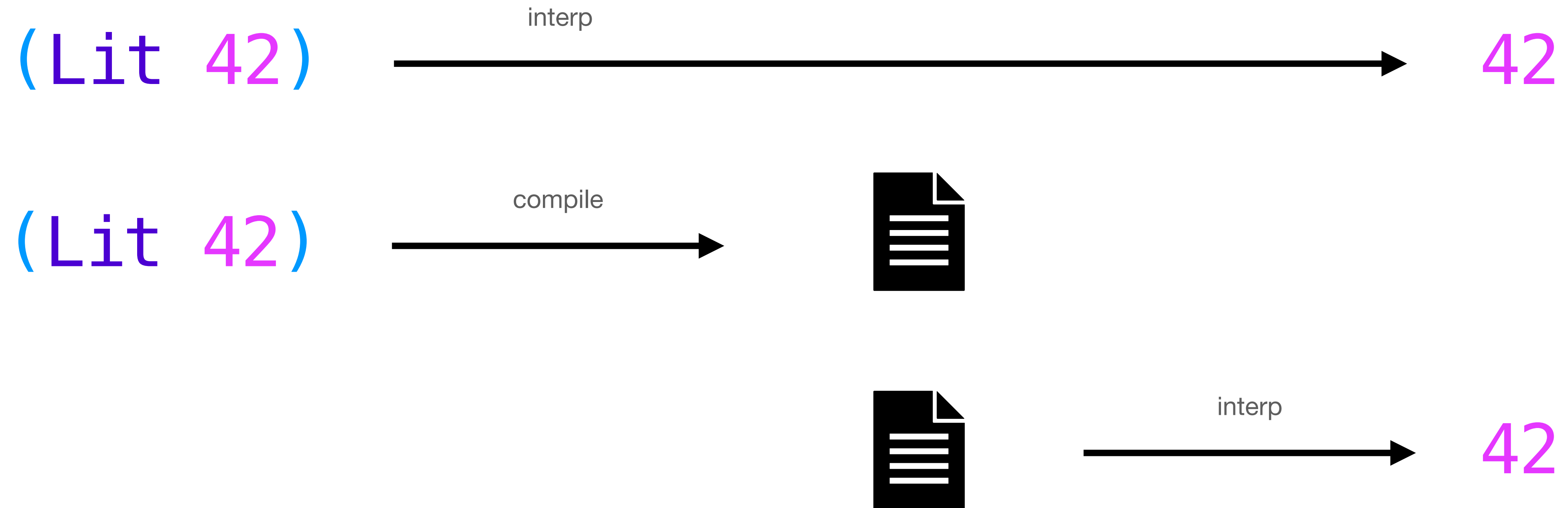
Example Abscond program

Compiler computes a program that computes meaning



Example Abscond program

Compiler specification



For next time

- Study Abscond, Blackmail, Con, and Dupe notes
- Take ELMS quiz

CMSC 430 - 3 June 2026

Our next compilers

- Abscond - integer literals
- Blackmail - unary primitives
- Con - conditional expressions
- Dupe - booleans, revised conditional

CMSC 430 - 3 June 2026

Announcements

- Assignment 1 & 2 released yesterday
- New quiz released today, due by start of class tomorrow

Abseond

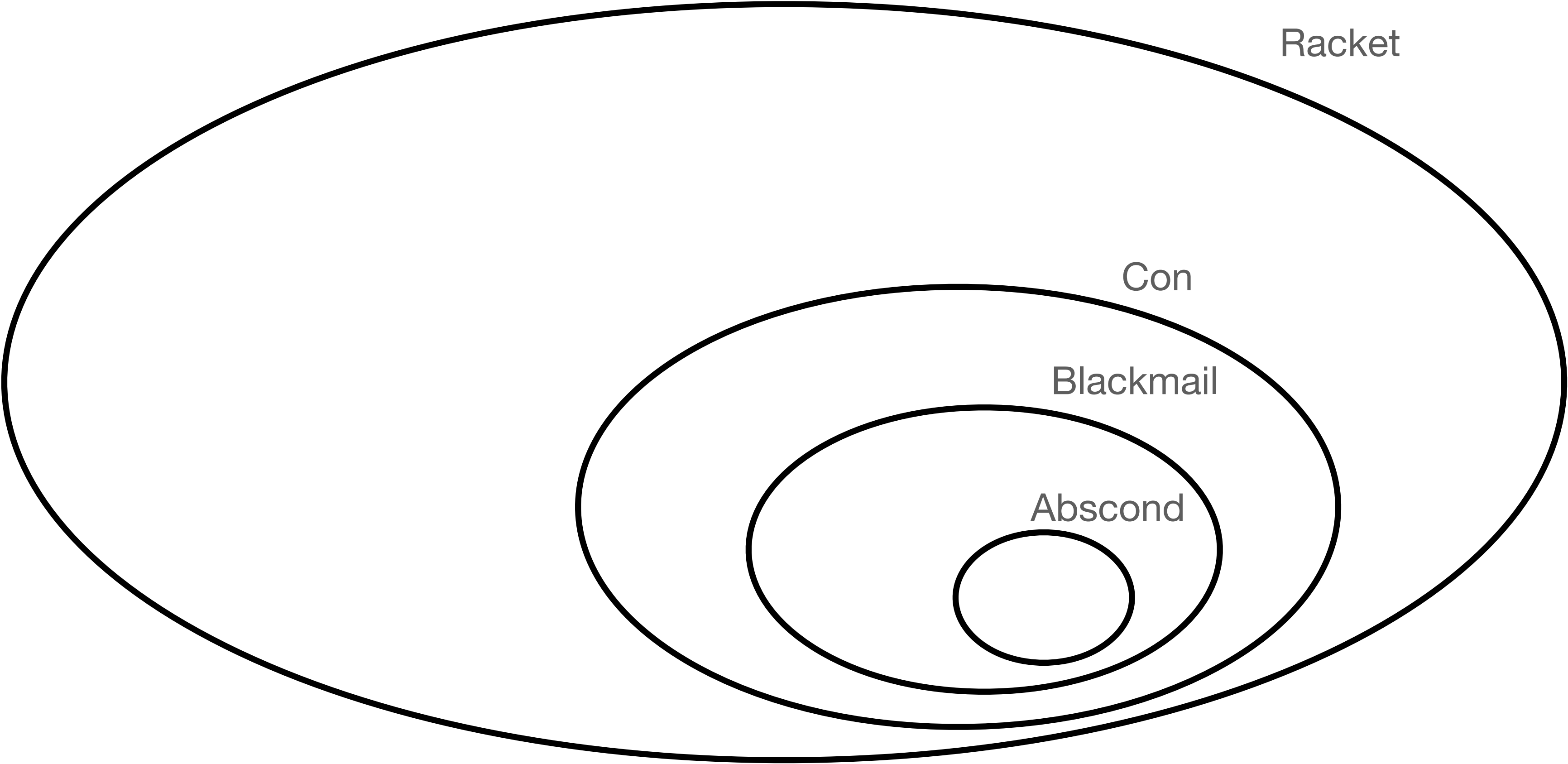
Learning objectives

Abscond

You should be able to answer:

- What is the structure of the Abscond compiler?
- How are Abscond programs represented abstractly?
- What invariant relates source programs and compiled code?

Language subsets



Example Abscond program

Concrete syntax

```
#lang racket
```

```
42
```

Example Abscond program

Abstract syntax

(Lit 42)

```
:: type Expr =  
:: | (Lit Integer)  
(struct Lit (n))
```

Example Abscond program

Parser constructs AST from string input

#lang racket
42

parse

→ (Lit 42)

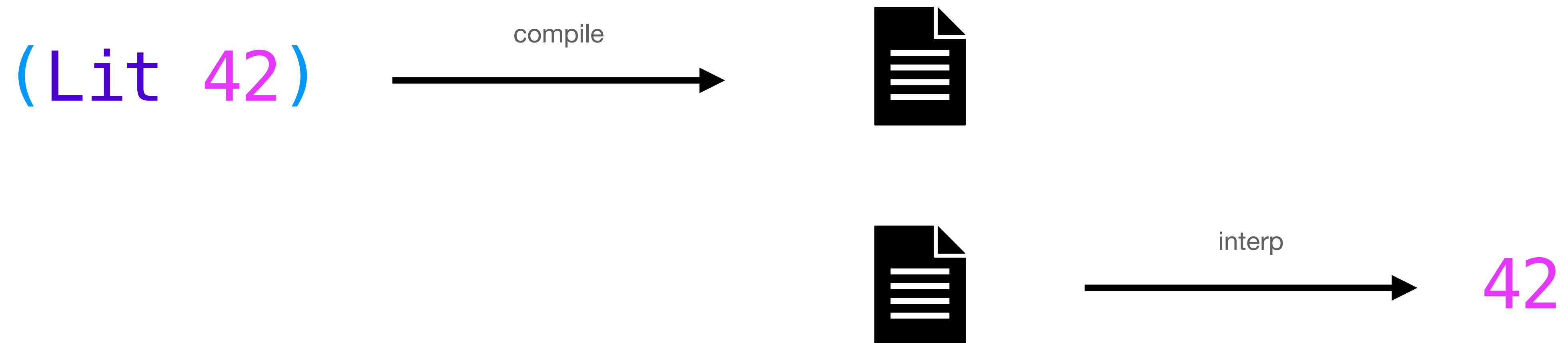
Example Abscond program

Interpreter computes meaning from AST



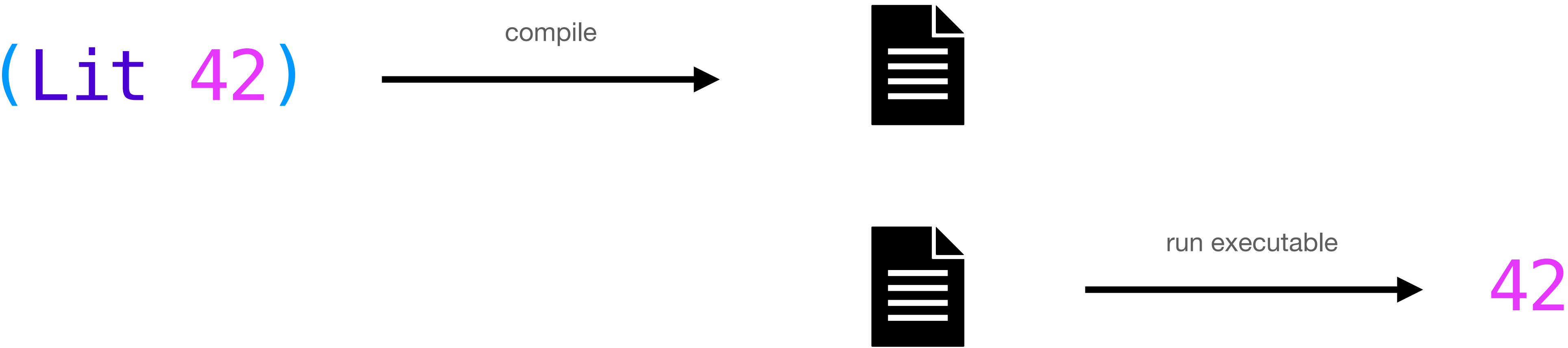
Example Abscond program

Compiler computes a program that computes meaning



Example Abscond program

Compiler specification



Compile time

Run time

Example Abscond program

Our approach to parsing: reader + parser

#lang racket
42



42



(Lit 42)

String

S-Expr

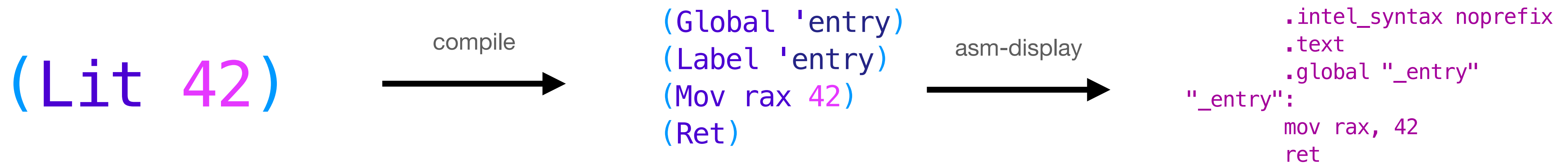
Expr

Semi-structured data;
ensures parens balanced, etc.

Structured data; ensures
grammatically well-formed

Example Abscond program

Our approach to compiling: compiler + stand-alone runtime



Expr

Construct AST of assembly
program

Asm

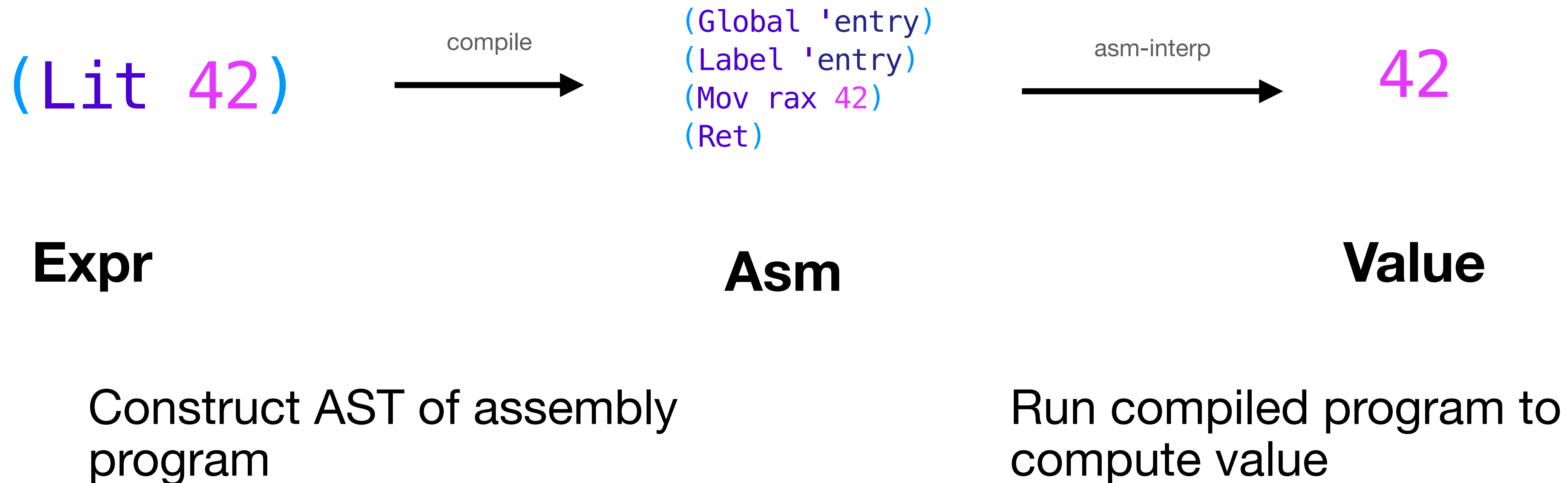
.S

Print to standard assembly
format; call assembler

Link w/ stand-alone runtime
system

Example Abscond program

Our approach to testing: compiler + asm-interp



Abscond

At the heart of it

```
;; Expr -> Asm  
(define (compile e) ...)  
;; Asm -> Value  
(define (run a) ...)
```

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

```
;; compile ◦ run ≡ interp
```

Abscond interpreter

At the heart of it

```
:: Expr -> Integer  
(define (interp e)  
  (match e  
    [(Lit i) i]))
```


Abscond compiler

At the heart of it

```
;; Expr -> Asm  
(define (compile e)  
  (list (Global 'entry)  
        (Label 'entry)  
        (match e  
          [(Lit i) (Mov rax i)]  
          (Ret)))
```

Abscond correctness

At the heart of it

```
;; Expr -> Void  
(define (check-compiler e)  
  (check-equal? (interp e)  
                (asm-interp (compile e))))
```

Hosted vs Standalone Runtime System

The runtime system is code that is needed at runtime to assist execution of compiled code. We will use two different runtime systems:

- Hosted: written in Racket w/ asm-interp; requires Racket at runtime
- Stand-alone: written in C; doesn't require Racket at runtime

Mostly we used hosted for development and testing, but the standalone is useful for when you want to make standalone executables.

Abscond directories

- `syntax/` - AST definition, parser
- `interpreter/` - interpreter, command line interpreter
- `compiler/` - compiler, command line compiler
- `executor/` - hosted runtime, command line executor
- `runtime/` - standalone runtime
- `test/` - unit tests

What do Abscond programs mean?

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)))
```


Blackmail

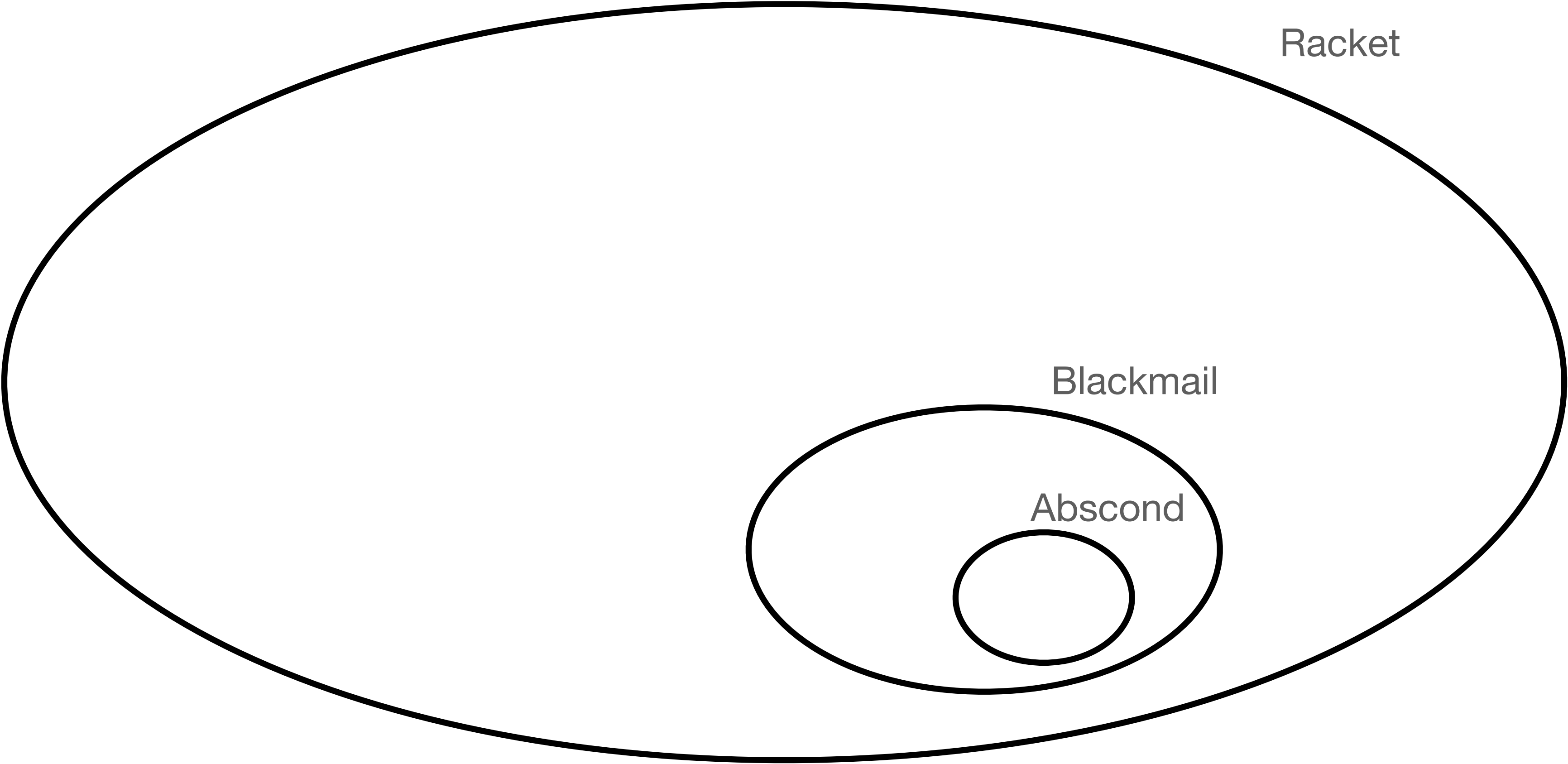
Learning objectives

Blackmail

You should be able to answer:

- How are Blackmail programs represented abstractly?
- What's the process for systematically growing our language?
- How do we structure an interpreter and compiler for an inductively defined AST data type?
- How do we both use and ensure the compiler invariants?

Language subsets



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*

Example Blackmail program

Concrete syntax

```
#lang racket  
(add1 (add1 40))
```

Example Blackmail program

Abstract syntax

```
(Prim1 'add1 (Prim1 'add1 (Lit 40)))
```

```
:: type Expr = (Lit Integer)
::           | (Prim1 Op1 Expr)
:: type Op1 = 'add1 | 'sub1
(struct Lit (i))
(struct Prim1 (p e))
```


Example Abscond program

Interpreter computes meaning from AST

`(Prim1 'add1 (Prim1 'add1 (Lit 40)))` $\xrightarrow{\text{interp}}$ 42

Example Abscond program

Interpreter computes meaning from AST

```
(Prim1 'add1 (Prim1 'add1 (Lit 40)))
```

compile



```
(Global 'entry)
(Label 'entry)
(Mov rax 40)
(Add rax 1)
(Add rax 1)
(Ret)
```

asm-interp



42

Expr is an inductive data type

Abstract syntax

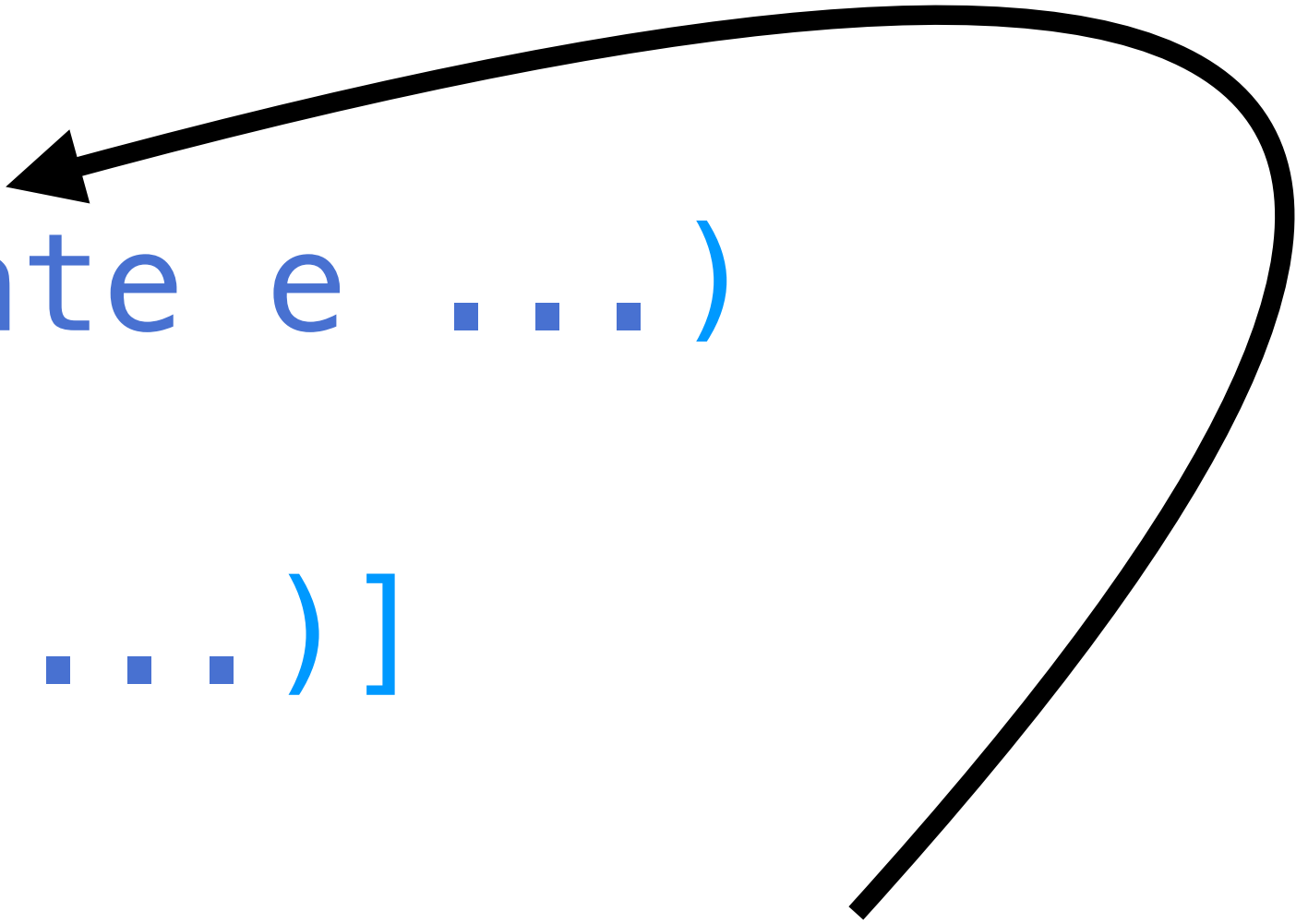
```
;; type Expr = (Lit Integer)
;;           | (Prim1 Op1 Expr)
;; type Op1 = 'add1 | 'sub1
(struct Lit (i))
(struct Prim1 (p e))
```



Expr is an inductive data type

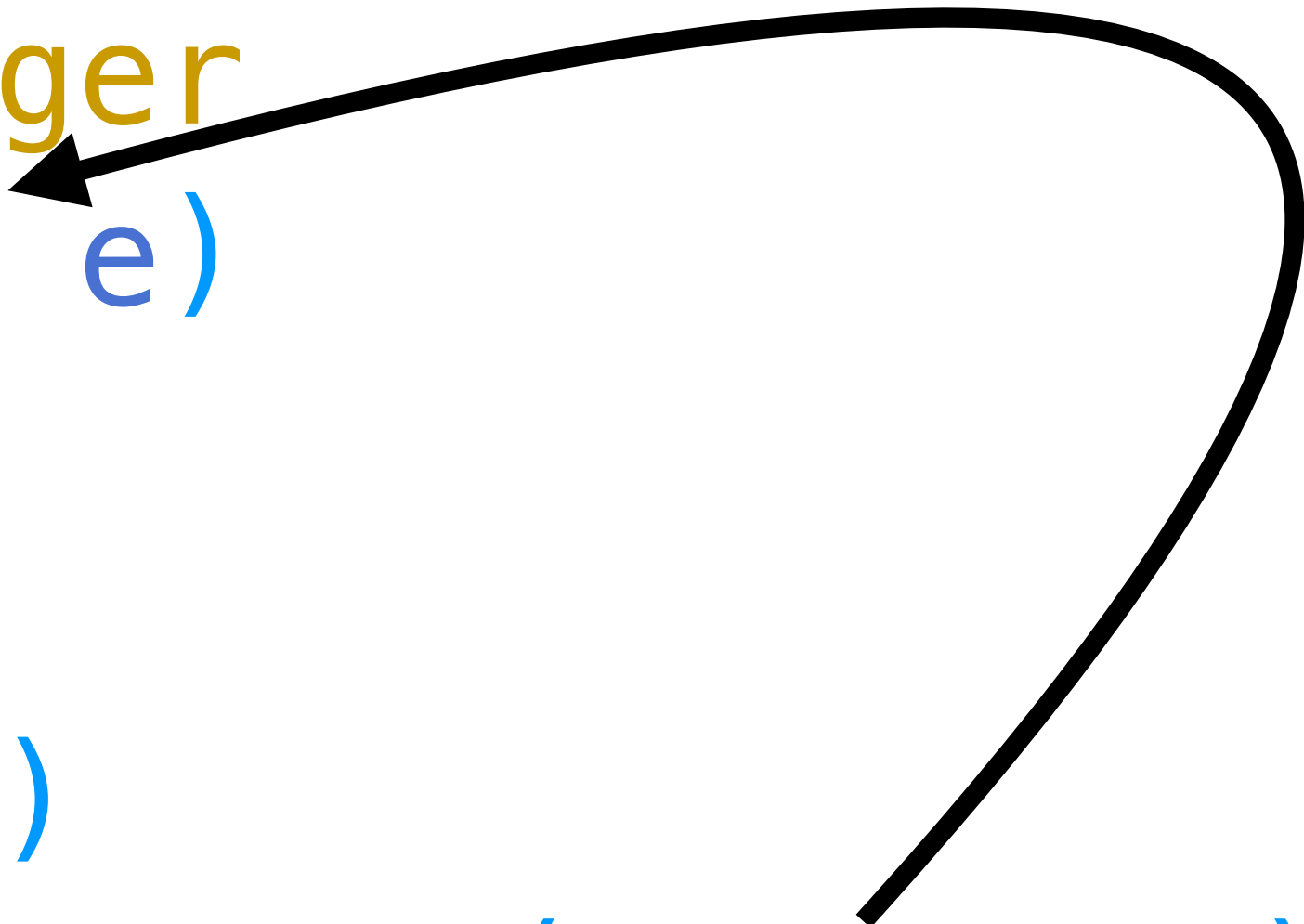
Abstract syntax

```
(define (expr-template e ...)  
  (match e  
    [(Lit i) (... i ...)]  
    [(Prim1 o e)  
     (... o ... (expr-template e ...) ...)]))
```



Blackmail interpreter

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



Blackmail interpreter

```
:: Op1 Integer -> Integer  
(define (interp-prim1 op i)  
  (match op  
    [ 'add1 (add1 i)]  
    [ 'sub1 (sub1 i)]))
```

Blackmail compiler

Compiling programs

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

Blackmail compiler

Key invariant

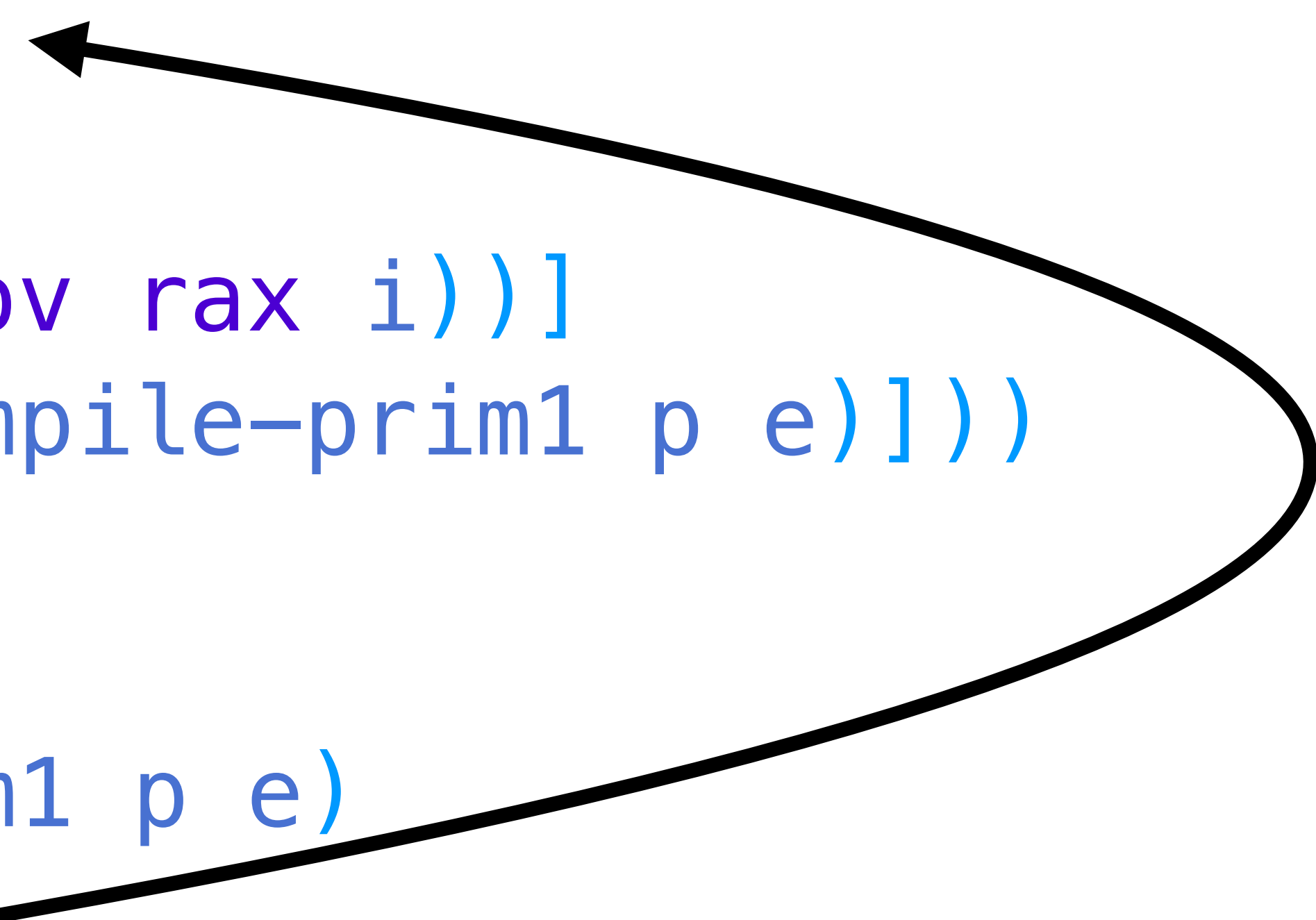
```
;; Invariant:  
;; Produces instructions that when run,  
;; leave e's value in rax  
(compile-e e)
```


Blackmail compiler

Compiling expressions

:: Expr -> Asm

```
(define (compile-e e)  
  (match e  
    [(Lit i) (seq (Mov rax i))]  
    [(Prim1 p e) (compile-prim1 p e)]))
```



:: Op1 Expr -> Asm

```
(define (compile-prim1 p e)  
  (seq (compile-e e)  
        (compile-op1 p)))
```

Blackmail compiler

Compiling primitives

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


Blackmail compiler

Example

```
> (asm-interp (compile (parse '(add1 (add1 40)))))  
42
```

Con

Learning objectives

Con

You should be able to answer:

- How are Con programs represented abstractly?
- How do we support conditional execution?
- How can code generate unique labels for conditional execution?

Example Con program

Concrete syntax

```
#lang racket  
(if (zero? 5)  
    (add1 7)  
    (sub1 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:

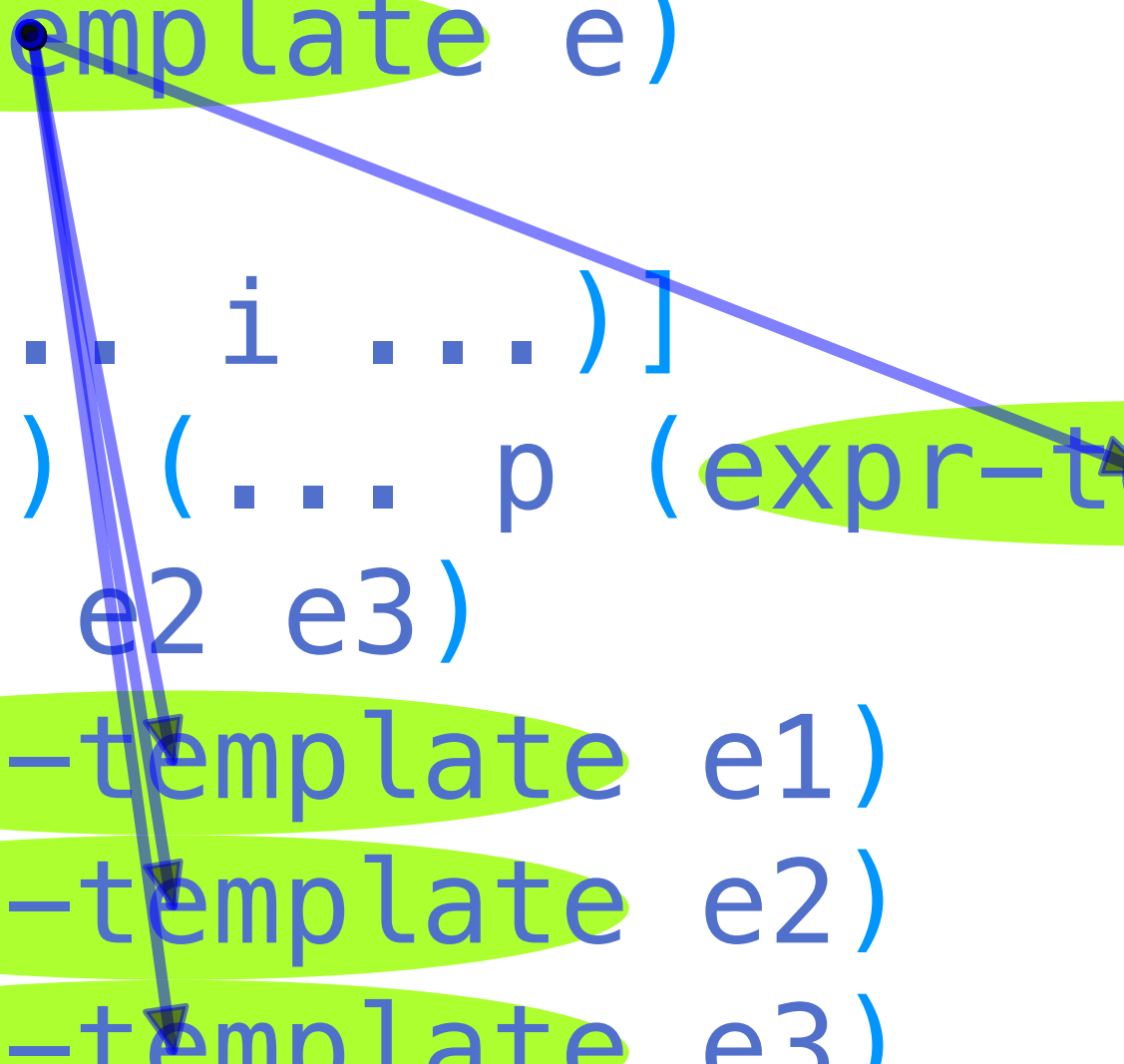
```
(IfZero (Lit 1) (Lit 2) (Lit 3))
(IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3))
(Prim1 'add1 (IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3)))
(Prim1 'add1 (IfZero (Prim1 'sub1 (Lit 1)) (Prim1 'add1 (Lit 1))
(Lit 3)))
```

Con: adding conditional expressions

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) ...)]))
```



A diagram illustrating the recursive nature of the `expr-template` function. A blue dot is placed on the `expr-template` argument of the `(Prim1 p e)` pattern in the `match` expression. Two blue arrows originate from this dot: one points to the `expr-template` argument of the `(IfZero e1 e2 e3)` pattern, and the other points to the `expr-template` argument of the `(expr-template e1)` call within the `(IfZero e1 e2 e3)` pattern's body. This visualizes how the function calls itself to process sub-expressions.

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

Essence of the compiler for Con

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


Con: adding conditional expressions

Essence of the compiler for Con

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

Dupe

Learning objectives

Dupe

You should be able to answer:

- How are Dupe programs represented abstractly?
- How can we represent disjoint datatypes?
- What's the relationship between bits and values?
- How do disjoint datatypes complicate the statement of compiler correctness?

Dupe: adding Boolean values

Concrete Examples

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

Dupe: adding Boolean values

Abstract Syntax

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

```
:: type Datum = Integer
::           | Boolean
```

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

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

Dupe: adding Boolean values

Abstract Examples

Concrete examples:

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

Abstract examples:

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


Dupe: adding Boolean values

Interpreter

```
:: type Value =  
:: | Integer  
:: | Boolean
```

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

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

Recipe for growing a language

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

Start with examples

What to do with new literals?

`(compile-e (Lit 42))` $\Rightarrow$ `(Mov rax 42)`

`(compile-e (Lit #f))` $\Rightarrow$ `(Mov rax ??)`

For next time

- Read Dodger notes — we won't spend (much) lecture time on it
- Study Evildoer notes
- Take ELMS quiz

CMSC 430 - 4 June 2026

Representing disjoint data; I/O effects

- Dupe - booleans, revised conditional
- Dodger (briefly) - characters
- Evildoer - I/O, system calls, ABI

CMSC 430 - 4 June 2026

Announcements

- TA OH: M, W, 1-2PM on Discord (see Piazza post)
- Reminder: I'm available 30 mins before lecture on Zoom
- Video of Practice Assignment 2 on ELMS now
- New quiz released today, due by start of class tomorrow

Dupe

Learning objectives

Dupe

You should be able to answer:

- How are Dupe programs represented abstractly?
- How can we represent disjoint datatypes?
- What's the relationship between bits and values?
- How do disjoint datatypes complicate the statement of compiler correctness?
- What happens to the runtime system when a new kind of value is introduced?

Dupe: adding Boolean values

Concrete Examples

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

Dupe: adding Boolean values

Abstract Syntax

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

```
:: type Datum = Integer
::           | Boolean
```

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

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


Dupe: adding Boolean values

Abstract Examples

Concrete examples:

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

Abstract examples:

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

Dupe: adding Boolean values

Interpreter

```
:: type Value =  
:: | Integer  
:: | Boolean
```

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

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

Recipe for growing a language

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

Start with examples

What to do with new literals?

`(compile-e (Lit 42))` $\Rightarrow$ `(Mov rax 42)`

`(compile-e (Lit #f))` $\Rightarrow$ `(Mov rax ??)`

Start with examples

What to do with new literals?

A tempting proposal:

```
(compile-e (Lit #t)) ;=> (Mov rax 1)
(compile-e (Lit #f)) ;=> (Mov rax 0)
```



Start with examples

What to do with new literals?

A tempting proposal:

```
(compile-e (Lit #t))  ;=> (Mov rax 1)
(compile-e (Lit #f))  ;=> (Mov rax 0)
(compile-e (Lit 0))    ;=> (Mov rax 0)
(compile-e (Lit 1))    ;=> (Mov rax 1)
```



Start with examples

What to do with new literals?

A tempting proposal:

```
(compile-e (Lit #t))  => (Mov rax 1)
(compile-e (Lit #f))  => (Mov rax 0)
(compile-e (Lit 0))   => (Mov rax 0)
(compile-e (Lit 1))   => (Mov rax 1)
(compile-e (If e1 e2 e3))
  => (seq (compile-e e1) (Cmp rax 0) ...)
```



Start with examples

What to do with new literals?

A tempting proposal:

```
(compile-e (Lit #t))  => (Mov rax 1)
(compile-e (Lit #f))  => (Mov rax 0)
(compile-e (Lit 0))   => (Mov rax 0)
(compile-e (Lit 1))   => (Mov rax 1)
(compile-e (If e1 e2 e3))
  => (seq (compile-e e1) (Cmp rax 0) ...)
(compile-e (If (Lit #f) e2 e3))
  => (seq (Mov rax 0) (Cmp rax 0) ...) ;; OK
```



Start with examples

What to do with new literals?

A tempting proposal:

```
(compile-e (Lit #t))  => (Mov rax 1)
(compile-e (Lit #f))  => (Mov rax 0)
(compile-e (Lit 0))    => (Mov rax 0)
(compile-e (Lit 1))    => (Mov rax 1)
(compile-e (If e1 e2 e3))
  => (seq (compile-e e1) (Cmp rax 0) ...)
(compile-e (If (Lit #f) e2 e3))
  => (seq (Mov rax 0) (Cmp rax 0) ...) ;; OK
(compile-e (If (Lit 0) e2 e3))
  => (seq (Mov rax 0) (Cmp rax 0) ...) ;; !!!
```



Fundamental problem

What to do with new literals?

Integers and booleans are disjoint sets of values.

But they have overlapping *representations* in compiled code.

No matter what integer we pick to represent #f, it will be confused with the integer value.

Possible solution: partition the bits

- some will represent integer values
- some will represent boolean values

Fundamental problem

What to do with new literals?

Possible solution: partition the bits

- some will represent integer values
- some will represent boolean values

Criteria: the contents of rax must uniquely determine the value

Idea: use one bit to tell you the type, integer vs boolean value

Encoding values in Dupe

Type tag in least significant bits

63-bits for number	0

Integers

	1

Booleans

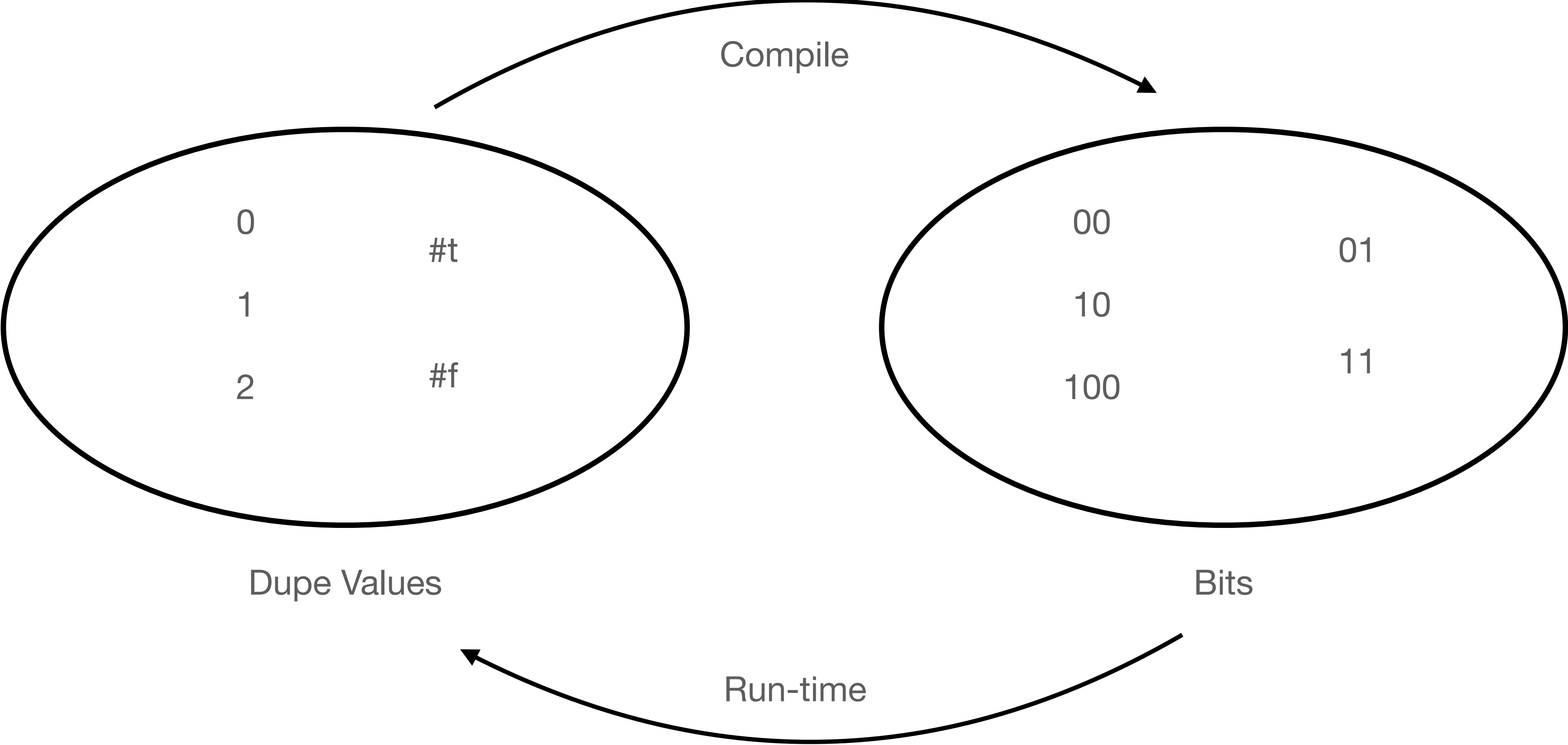
	0	1

#t

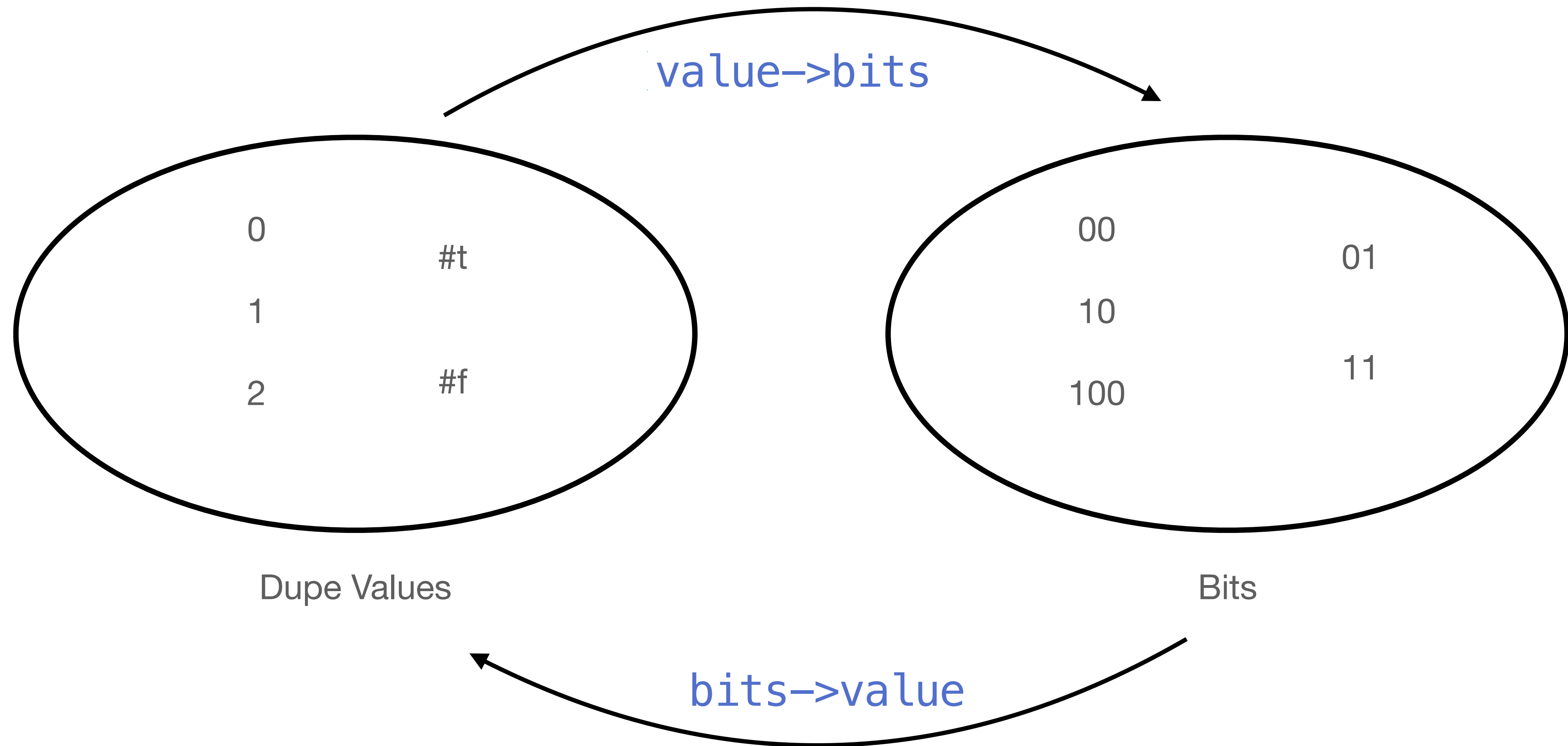
	1	1

#f

Representing Values with Bits in Dupe



Functions for mapping between worlds



Try again with type tagging

Disjoint types are represented with disjoint bits

`(compile-e (Lit #t))` $\Rightarrow$ 1 `#b01`

`(compile-e (Lit #f))` $\Rightarrow$ 3 `#b11`

`(compile-e (Lit 0))` $\Rightarrow$ 0 `#b00`

`(compile-e (Lit 1))` $\Rightarrow$ 2 `#b10`

`(compile-e (If e1 e2 e3))`
 $\Rightarrow$ `(seq ... (Cmp rax 3) ...)`

No integer value has the representation 3!

What about correctness?

```
;; Expr -> Void  
(define (check-compiler e)  
  (check-equal? (interp e)  
                (asm-interp (compile e))))
```



What about correctness?

```
> (check-compiler (parse '#t))
```



FAILURE

```
name:      check-equal?  
location:  correct.rkt:20:2  
actual:    #t  
expected:  1
```

```
> (check-compiler (parse '#f))
```



FAILURE

```
name:      check-equal?  
location:  correct.rkt:20:2  
actual:    #f  
expected:  3
```

```
> (check-compiler (parse '7))
```



FAILURE

```
name:      check-equal?  
location:  correct.rkt:20:2  
actual:    7  
expected:  14
```



What about correctness?

Revised to interpret bits as values

```
;; Expr -> Void  
(define (check-compiler e)  
  (check-equal? (interp e)  
                (bits->value  
                  (asm-interp (compile e)))))
```

```
> (check-compiler (parse '#t))  
> (check-compiler (parse '#f))  
> (check-compiler (parse '7))
```



What about correctness?

Revised to interpret bits as values

```
> (check-compiler (parse '(add1 #f)))
```


What about correctness?

Revised to interpret bits as values

```
> (check-compiler (parse '(add1 #f)))
```

 *interpreter/interp-prim.rkt:8:11: ERROR*

name: check-equal?

location: correct.rkt:21:2

add1: contract violation

expected: number?

given: #f

Dodger

(briefly)

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 Dodger

Type tag in least significant bits

63-bits for number	0

Integers

	1	1

Booleans

62-bits for code point (only need 22)	0	1

Characters

	0	1	1

#t

	1	1	1

#f

Syntactic additions in Dodger

Character literals and operations

Concrete syntax

`#\c`

`(char->integer e)`

`(integer->char e)`

`(char? e)`

Abstract syntax

`(Lit #\c)`

`(Prim1 'char->integer e)`

`(Prim1 'integer->char e)`

`(Prim1 'char? e)`

Semantic additions in Dodger

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

Welcome to [DrRacket](#), version 8.14 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
> #\a  
#\a  
> #\b  
#\b  
> (char? #\a)  
#t  
> (char? #t)  
#f  
> (char->integer #\a)  
97  
> (integer->char 97)  
#\a
```

Evildoer

Learning objectives

Evildoer

You should be able to answer:

- How are Evildoer programs represented abstractly?
- How does I/O change the meaning of programs?
- How can our host and stand-alone runtime systems provide support for I/O operations?
- What is an ABI? What is the System V ABI?
- How can we make calls to external code according to the ABI?

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)`

Syntactic additions in Evildoer

```
:: type Expr = (Lit Datum)
::           | (Eof)
::           | (Prim0 Op0)
::           | (Prim1 Op1 Expr)
::           | (If Expr Expr Expr)
::           | (Begin Expr Expr)
:: type Datum = Integer
::           | Boolean
::           | Character
:: type Op0 = 'read-byte | 'peek-byte | 'void
:: type Op1 = 'add1 | 'sub1
::           | 'zero?
::           | 'char? | 'integer->char | 'char->integer
::           | 'write-byte | 'eof-object?
```

```
(struct Eof () #:prefab)
(struct Lit (d) #:prefab)
(struct Prim0 (p) #:prefab)
(struct Prim1 (p e) #:prefab)
(struct If (e1 e2 e3) #:prefab)
(struct Begin (e1 e2) #:prefab)
```

Semantic additions in Evildoer

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

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


Encoding values so far in Evildoer

Type tag in least significant bits

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

How can we implement I/O operations?

Let's have the run-time system provide functionality

Runtime system provides I/O operations.

Compiler emits calls to run-time system.

This means:

- calling Racket functions in the host run-time system
- Calling C functions in the standalone run-time system

But how?

Application Binary Interface

An ABI standardizes:

- How function arguments are passed
- How return values are returned
- Which registers a function must preserve
- Stack layout and alignment

System V ABI

The UNIX ABI

System V ABI standardizes:

- How function arguments are passed: first arg in rdi
- How return values are returned: in rax
- Which registers a function must preserve: rsp, rbx, rbp, r12, ..., r15
- Stack layout and alignment: aligned to 16-bytes

System V ABI

The UNIX ABI

The System V ABI is what allows us to write asm-interp!

The System V ABI is what allows us to write the stand-alone run-time

The System V ABI is what allows us to call...

...Racket and C from compiled code.

I/O support in standalone run-time system

io.c

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

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

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

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

For next time

- Read Dodger notes
- Study Evildoer notes
- Take ELMS quiz
- Submit Assignment 1

CMSC 430 - 5 June 2026

I/O effects; Errors

- Evildoer - I/O, system calls, ABI
- Extort - Specifying error semantics

CMSC 430 - 5 June 2026

Announcements

- Assignment 1 grades released
- Assignment 2 due Monday by midnight
- Assignment 3 released later today
- **Monday's lecture will be asynchronous:** video recording will be posted to ELMS
- New quiz released today, due by 10AM Monday

Evildoer

Learning objectives

Evildoer

You should be able to answer:

- How are Evildoer programs represented abstractly?
- How does I/O change the meaning of programs?
- How can our host and stand-alone runtime systems provide support for I/O operations?
- What is an ABI? What is the System V ABI?
- How can we make calls to external code according to the ABI?

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)`

Syntactic additions in Evildoer

```
:: type Expr = (Lit Datum)
::           | (Eof)
::           | (Prim0 Op0)
::           | (Prim1 Op1 Expr)
::           | (If Expr Expr Expr)
::           | (Begin Expr Expr)
:: type Datum = Integer
::           | Boolean
::           | Character
:: type Op0 = 'read-byte | 'peek-byte | 'void
:: type Op1 = 'add1 | 'sub1
::           | 'zero?
::           | 'char? | 'integer->char | 'char->integer
::           | 'write-byte | 'eof-object?

(struct Eof () #:prefab)
(struct Lit (d) #:prefab)
(struct Prim0 (p) #:prefab)
(struct Prim1 (p e) #:prefab)
(struct If (e1 e2 e3) #:prefab)
(struct Begin (e1 e2) #:prefab)
```


Semantic additions in Evildoer

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

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

Encoding values so far in Evildoer

Type tag in least significant bits

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

How can we implement I/O operations?

Let's have the run-time system provide functionality

Runtime system provides I/O operations.

Compiler emits calls to run-time system.

This means:

- calling Racket functions in the host run-time system
- Calling C functions in the standalone run-time system

But how?

ABI

Application Binary Interface

An ABI standardizes:

- How function arguments are passed
- How return values are returned
- Which registers a function must preserve
- Stack layout and alignment

System V ABI

The UNIX ABI

System V ABI standardizes:

- How function arguments are passed: in rdi, rsi, rdx, rcx, r8, r9, stack
- How return values are returned: in rax, rdx
- Which registers a function must preserve: rsp, rbx, rbp, r12— r15
- Stack layout and alignment: aligned to 16-bytes

Used by all major flavors of UNIX: Linux, Mac OS, BSD, etc.

System V ABI

The UNIX ABI

The System V ABI is what allows us to write asm-interp!

The System V ABI is what allows us to write the stand-alone run-time

The System V ABI is what allows us to call...

...Racket and C from compiled code.

System V ABI

The UNIX ABI

The ABI is what allows C to call assembly code as though it were a C function.

This is how our standalone runtime can call the compiled code.

```
#include <stdio.h>
#include <inttypes.h>

int64_t entry();

int main(int argc, char** argv)
{
    int64_t result = entry();
    printf("%" PRIu64, result);
    putchar('\n');
    return 0;
}
```


Let's make an example

Calling a C function from assembly

```
(define (c-abs x)
  (asm-interp
    (prog (Global 'entry)
          (Label 'entry)
          (Sub rsp 8)
          (Mov rdi x)
          (Extern 'abs) ; defined in libc
          (Call 'abs)
          (Add rsp 8)
          (Ret))))
```

Let's make an example

Calling a Racket function from assembly

```
(require ffi/unsafe)
(current-externs ; set extern functions available to asm-interp
(list
  ;; each extern has
  ;; - a label name
  ;; - a Racket function
  ;; - a C type signature
  (extern 'gcd gcd (_fun _int64 _int64 -> _int64))))

(asm-interp
  (prog (Global 'entry)
        (Label 'entry)
        (Sub rsp 8)
        (Mov rdi 945)
        (Mov rsi 18)
        (Extern 'gcd)
        (Call 'gcd)
        (Add rsp 8)
        (Ret)))
```

Evildoer

I/O support in hosted run-time system

executor/host.rkt

Our own asm-interp that supports these primitives:

```
(define (asm-interp/host asm)
  (parameterize
    ([current-externs
      (list (extern 'read_byte prim-read-byte (_fun -> _int64))
            (extern 'peek_byte prim-peek-byte (_fun -> _int64))
            (extern 'write_byte prim-write-byte (_fun _int64 -> _int64)))]])
    (asm-interp asm)))
```

I/O support in hosted run-time system

executor/host.rkt

Definition of the primitives:

```
(define (prim-read-byte)
  (value->bits (read-byte)))
(define (prim-peek-byte)
  (value->bits (peek-byte)))
(define (prim-write-byte bs)
  (value->bits (write-byte (bits->value bs)))))
```


I/O support in hosted run-time system

executor/host.rkt

An example:

```
> (with-output-to-string
    (λ ()
      (asm-interp/host
        (prog (Global 'entry)
              (Label 'entry)
              (Sub rsp 8)
              (Mov rdi (value->bits 97))
              (Extern 'write_byte)
              (Call 'write_byte)
              (Add rsp 8)
              (Ret))))))

"a"
```

I/O support in standalone run-time system

runtime/io.c

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

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

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

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

Compiling I/O primitive operations

compiler/compile-ops.rkt

```
;; Op0 -> Asm
(define (compile-op0 p)
  (match p
    ;; ...
    ['read-byte (seq (Extern 'read_byte) (Call 'read_byte))]
    ['peek-byte (seq (Extern 'peek_byte) (Call 'peek_byte))]))
```

```
;; Op1 -> Asm
(define (compile-op1 p)
  (match p
    ;; ...
    ['write-byte
     (seq (Extern 'write_byte)
          (Mov rdi rax)
          (Call 'write_byte))]))
```


What about correctness?

Meaning comes from environment, too

The meaning of a program depends on more than just the program now... It implicitly depends on the input and output port. We can make that explicit:

```
;; String Expr -> (Cons Value String)
;; Interpret e with given string as input,
;; return value and collected output as string
(define (interp/io e input)
  (parameterize ((current-input-port (open-input-string input))
                 (current-output-port (open-output-string)))
    (cons (interp e)
          (get-output-string (current-output-port)))))
```

What about correctness?

Meaning comes from environment, too

We can define a similar function for compilation + execution:

```
:: Expr -> Value
(define (exec e)
  (bits->value (asm-interp/host (compile e))))

:: Asm String -> (cons Value String)
(define (exec/io asm in)
  (parameterize ((current-output-port (open-output-string))
                 (current-input-port  (open-input-string in)))
    (cons (exec asm)
          (get-output-string (current-output-port)))))
```

What about correctness?

Meaning comes from environment, too

Correct means producing the same value and output given same input:

```
;; Expr String -> Void
(define (check-compiler e i)
  (let ((r (with-handlers ([exn:fail? identity])
                    (interp/io e i))))
    (unless (exn? r)
      (check-equal? r (exec/io e i)))))
```


Extort

Learning objectives

Extort

You should be able to answer:

- Why is undefined behavior convenient for compiler writers?
- Why is undefined behavior dangerous?
- How can we semantically specify error behaviors?
- What run-time techniques will be required to ensure safety?

Extort: Specifying Errors

Undefined behavior considered harmful

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

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

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

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

Extort: Specifying Errors

Undefined behavior considered harmful

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

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

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

Extort: Specifying Errors

Undefined behavior considered harmful

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

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


Extort: Specifying Errors

Undefined behavior considered harmful

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

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

Extort: Compiling with Errors

Undefined behavior considered harmful

Specifying errors ties the compiler's hands:

```
;; Expr String -> Void  
(define (check-compiler e i)  
  (check-equal? (interp/io e i)  
                (exec/io e i)))
```

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

```
> (compile-op1 'add1)
(list
  (Push 'rax)
  (And 'rax 1)
  (Cmp 'rax 0)
  (Pop 'rax)
  (Jne 'err)
  (Add 'rax 2))
```

Jumps to place that signals an error if the thing in rax is not an integer

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

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

For next time

- Study Extort notes
- Read Fraud notes
- Take ELMS quiz

CMSC 430 - 8 June 2026

Variables, binding, lexical addresses, and the stack

- Extort - Specifying error semantics; runtime support for errors
- Fraud - Variables and let expressions

CMSC 430 - 8 June 2026

Announcements

- New quiz released, due by 10AM Tues

Extort

(runtime support for signaling errors)

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

```
> (compile-op1 'add1)
(list
  (Push 'rax)
  (And 'rax 1)
  (Cmp 'rax 0)
  (Pop 'rax)
  (Jne 'err)
  (Add 'rax 2))
```

Jumps to place that signals an error if the thing in rax is not an integer

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

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

Extort: Compiling with Errors

Undefined behavior considered harmful

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


Run-time system support

raise_error in standalone run-time system

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

Run-time system support

raise_error in hosted run-time system

```
(extern 'raise_error' (λ () (raise 'err))) (_fun -> _void))
```

```
;; Expr -> Answer
```

```
(define (exec e)
```

```
  (with-handlers ([ (λ (x) (eq? x 'err)) identity ])
```

```
    (bits->value (asm-interp/host (compile e)))))
```


Fraud

Learning objectives

Fraud

You should be able to answer:

- How are Fraud programs represented abstractly?
- How is the meaning of a variable determined?
- What changes in the interpreter with the addition of variables?
- What invariant can be observed about variable occurrences and their position in the environment?
- How can this invariant be used to compile variable occurrences?

Learning objectives

Fraud

You should be able to answer (cont'd):

- What role does the stack play in computing the meaning of variables at run-time?
- What role does the compile-time environment play in the compilation of variable occurrences?
- What is the relationship between the compile-time environment and the run-time stack?
- What's the difference between a static and a dynamic property?

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*

Example Fraud program

Concrete syntax

```
#lang racket  
(let ((x 40))  
  (add1 (add1 x)))
```

Example Fraud program

Abstract syntax

```
(Let 'x (Lit 40)
  (Prim1 'add1
    (Prim1 'add1
      (Var 'x))))
```


Fraud AST

```
:: type Expr = ...  
::           | (Let Id Expr Expr)  
::           | (Var Id)  
  
:: type Id   = Symbol
```

Fraud expression parser

```
> (parse 'x)
'#s(Var x)
> (parse '(let ((x 42)) x))
'#s(Let x #s(Lit 42) #s(Var x))
```


Fraud program parser

Only closed expressions are well-formed programs

```
> (parse 'x)
```

```
'#s(Var x)
```

```
> (parse '(let ((x 42)) x))
```

```
'#s(Let x #s(Lit 42) #s(Var x))
```

```
> (parse-closed 'x)
```



```
../syntax/parse.rkt:16:18: unbound identifiers '(x)
```

```
> (parse-closed '(let ((x 42)) x))
```

```
'#s(Let x #s(Lit 42) #s(Var x))
```

Fraud: adding variable bindings

Updated semantics

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



What do variables mean?

It varies

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

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

```
(interp-e (Var 'x)) ;=> 42  
(interp-e (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 ...))
```

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

A function!



Representing the environment

Using an association list

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

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


Operations on the environment

Using an association list

;; Lookup up x's value in r

;; Env Id $\rightarrow$ Value

(define (lookup r x) ...)

;; Extend r so that x means v

;; Env Id Value $\rightarrow$ Env

(define (ext r x v) ...)

;; Fundamental law:

;; $\forall r x v,$

(lookup (ext r x v) x) $\equiv$ v

Interpreting variables and bindings

Environments tell us the meaning of variables

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

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

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

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

;; type ClosedExpr =
;;   { e ∈ Expr | e contains no free variables }
```


Compiling variables and bindings

Using the interpreter the guide us

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

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

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

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

Observe something about environments

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

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

Observe something about environments

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

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

Observing an invariant about bindings

Using the interpreter the guide us

Suppose we get to this point in interpreting a program:

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

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

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

It has to have looked like this!

Observing an invariant about bindings

Using the interpreter the guide us

Suppose we know the program looks like this:

```
(let ((p ...))
```

```
...
```

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

```
...
```

```
(let ((y ...))
```

```
... x ...)))
```

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

'((y ??) (x ??) (p ??)) It must look like this!

But now we can see that `lookup` will retrieve the second element;

The location of a variables binding in the environment is a static property

Generalizing the observation

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

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

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

Static vs dynamic properties

Variable bindings are partly static and partly dynamic

- A static property of a program can be determined without running
- A dynamic property of a program cannot be determined without running

A variables location in the environment is static

A variables value in the environment is dynamic

Static vs dynamic properties

Variable bindings are partly static and partly dynamic

```
(let ((x ...))  
  (let ((y (read-byte)))  
    (begin  
      (let ((z ...))  
        ...)  
      ;; Where will y's binding  
      ;; be in the environment?  
      y)))
```

y's binding will be the second element in the environment

y's value is unknowable without running the program

Key idea

Compile-time environment and run-time stack

- Compute lexical addresses at compile-time
- Use stack to hold value of bindings at run-time
- Use compile-time environment to describe binding structure
- Maintain agreement w/ compile-time environment and run-time stack

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 (Mem rsp 0))
```

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

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


Compile-time environments

compile-e now takes an environment

```
;; Id CEnv -> Asm
(define (compile-variable x c)
  (let ((i (lookup x c)))
    (seq (Mov rax (Mem rsp i))))))
```

```
;; Id Expr Expr CEnv -> Asm
(define (compile-let x e1 e2 c)
  (seq (compile-e e1 c)
        (Push rax)
        (compile-e e2 (cons x c))
        (Add rsp 8)))
```

Compile-time environments

compile-e now takes an environment

```
;; Id CEnv -> Integer
(define (lookup x cenv)
  (match cenv
    [(cons y rest)
     (match (eq? x y)
       [#t 0]
       [#f (+ 8 (lookup x rest))])])])
```


Success

Our compiler works for programs with variables

```
> (exec (parse '(let ((x 40)) (add1 (add1 x)))))  
42
```



What about this?

Are we still minding the ABI requirements...

```
> (compile (parse '(let ((x 40)) (write-byte x))))  
(list  
  (Global 'entry)  
  (Label 'entry)  
  (Mov 'rax 80)  
  (Push 'rax)  
  (Mov 'rax (Mem 'rsp 0))  
  (Extern 'write_byte)  
  (Push 'rax)  
  (And 'rax 1)  
  (Cmp 'rax 0)  
  (Pop 'rax)  
  (Jne 'err)  
  (Cmp 'rax 0)  
  (Jl 'err)  
  (Cmp 'rax 510)  
  (Jg 'err)  
  (Mov 'rdi 'rax)  
  (Call 'write_byte)  
  (Add 'rsp 8)  
  (Ret)  
  (Label 'err)  
  (Extern 'raise_error)  
  (Call 'raise_error))
```



For next time

- Study Fraud notes
- Take ELMS quiz

CMSC 430 - 9 June 2026

Binary operations and the stack

- Fraud - Variables and let expressions, quick review
- Fraud - Binary primitive operations

CMSC 430 - 9 June 2026

Announcements

- Assignment 2 grades released
- Assignment 3 due Thursday by midnight
- Practice midterm released later today
- New quiz released, due by start of class tomorrow

Fraud

Learning objectives

Fraud

You should be able to answer (cont'd):

- What role does the stack play in computing the meaning of binary operations?
- What role does the compile-time environment play in the compilation of binary operations?
- How the ABI requirements for stack-alignment be achieved dynamically?

Fraud: adding binary operations

Concrete examples

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

Fraud: 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: adding binary operations

Updated semantics

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

Unchanged!



Fraud: adding binary operations

Updated semantics

```
;; Expr Env -> Value { raises 'err }  
(define (interp-e e r) ;; where r closes e  
  (match e  
    ;...  
    [(Prim2 p e1 e2)  
     (interp-prim2 p  
                   (interp-e e1 r)  
                   (interp-e e2 r))]))
```



Fraud: 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: adding binary operations

Updated compiler

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



Fraud: adding binary operations

Updated compiler

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



Fraud: adding binary operations

Updated compiler

```
> (exec (parse '(+ 1 1)))  
2
```

Fraud: adding binary operations

Updated compiler

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



Fraud: adding binary operations

Updated compiler

```
> (exec (parse ' (+ 1 1) ))
```

```
2
```

```
> (exec (parse ' (+ 1 2) ))
```

```
3
```

Fraud: adding binary operations

Updated compiler

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



Fraud: adding binary operations

Updated compiler

> (exec (parse '(+ 1 1)))

2

> (exec (parse '(+ 1 2)))

3

> (exec (parse '(+ 1 (+ 2 3))))

6



Fraud: adding binary operations

Updated compiler

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

Breaks the ABI by calling a function
with an unaligned stack



No way to align once and for all

Calls can happen with any number of pushes

```
#lang racket
```

```
(let ((x (write-byte 97)))           ; 0 push  
  (+ (begin (write-byte 98) 1)      ; 1 push  
    (begin  
      (write-byte 99)                ; 2 push  
      2)))
```

Dynamically aligning the stack

Updated compiler

```
;; Asm
;; Dynamically pad the stack to be aligned for a call
(define pad-stack
  (seq (Mov r15 rsp)
        (And r15 #b1000)
        (Sub rsp r15)))
```

Uses callee-saved register r15 to store stack change across call.
Works so long as the stack is either 8-byte or 16-byte aligned.
Since we always push 8-bytes at a time, this works for us.

```
;; Asm
;; Undo the stack alignment after a call
(define unpad-stack
  (seq (Add rsp r15)))
```


For next time

- Read Hustle notes
- Take ELMS quiz

CMSC 430 - 10 June 2026

Allocating data in the heap

- Checking in
- Quick look at Assignment 3
- Hustle - Pairs, boxes, and lists

CMSC 430 - 10 June 2026

Announcements

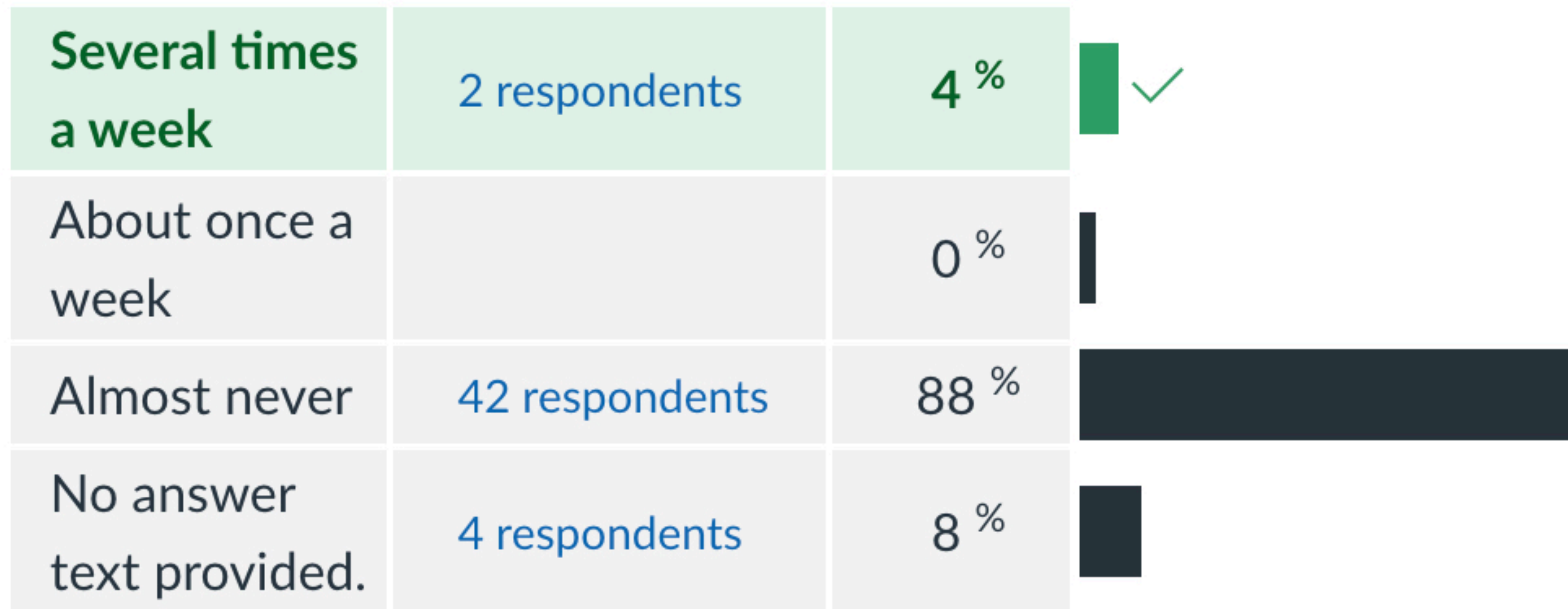
- Reminder: Assignment 3 due Thursday by midnight
- Practice exam 1 released
- Assignment 4 released later today
- New quiz released, due by start of class tomorrow

Checking in

How is the pace of this class?

Way too fast	3 respondents	6 %	
A little too fast	17 respondents	35 %	
Just right	28 respondents	58 %	
A little too slow		0 %	
Way too slow		0 %	

How often are you attending Office Hours?



Suggestions

For the slides, is it possible to have them separated based on lecture day or content rather than keeping them in a whole big file? It is kind of hard to scroll down as we get more in-depth into the course.

Can you make an announcement about what assignment we will have each week?

For longer projects maybe you can make a short video overview about it.

Hustle

Learning objectives

Hustle

You should be able to answer:

- How can inductive data be represented with heap allocated memory?
- How can type-tagging extend to pointer values?
- How is memory allocated at run-time?
- What's the difference between an immediate and a pointer value?
- What's the impact of our type tagging scheme on immediates?

Example concrete Hustle program

Lists

```
#lang racket
```

```
(car (cdr (cons 1 (cons 2 (cons 3 ' ())))))
```


Syntactic additions

```
:: type Datum = ...  
::          | '()
```

```
:: type Op1 = ...  
::          | 'car | 'cdr | 'unbox  
::          | 'empty? | 'cons? | 'box?  
::          | 'box
```

```
:: type Op2 = ...  
::          | 'cons
```

Semantic additions

Values are now inductively defined

```
:: type Value = ...  
:: | '()  
:: | (cons Value Value)  
:: | (box Value)
```

Semantic additions

Values are now inductively defined

```
;; Op1 Value -> Value { raises 'err }
(define (interp-prim1 op v)
  (match (list op v)
    ; ...
    [(list 'box v) (box v)]
    [(list 'unbox (? box?)) (unbox v)]
    [(list 'car (? pair?)) (car v)]
    [(list 'cdr (? pair?)) (cdr v)]
    [(list 'empty? v) (empty? v)]
    [(list 'cons? v) (cons? v)]
    [(list 'box? v) (box? v)]
    [_ (raise 'err)]))
```

```
;; Op2 Value Value -> Value { raises 'err }
(define (interp-prim2 op v1 v2)
  (match (list op v1 v2)
    ; ...
    [(list 'cons v1 v2) (cons v1 v2)]
    [_ (raise 'err)]))
```

We follow the familiar pattern:

- The empty list will be represented by Racket's empty list
- Pairs will be represented by Racket pairs
- Boxes will be represented by Racket boxes

Easy, but doesn't shed light on how these things are made.

How to represent pointers?

Addressing memory

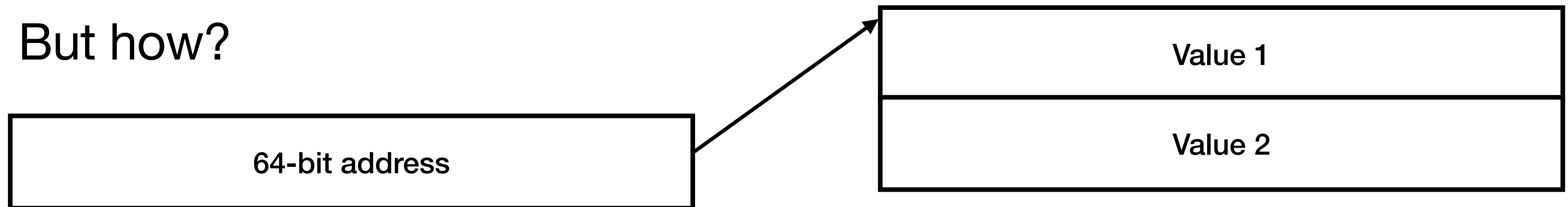
Basic idea:

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

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

Immediate tag

Encoding pointer values (Hustle)

Type tag in least significant bits

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

CMSC 430 - 11 June 2026

Allocating data in the heap

- Hustle - Pairs, boxes, and lists

CMSC 430 - 11 June 2026

Announcements

- No quiz
- Exam 1 released at midnight

Hustle

Encoding immediate values (Hustle)

Type tag in least significant bits

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

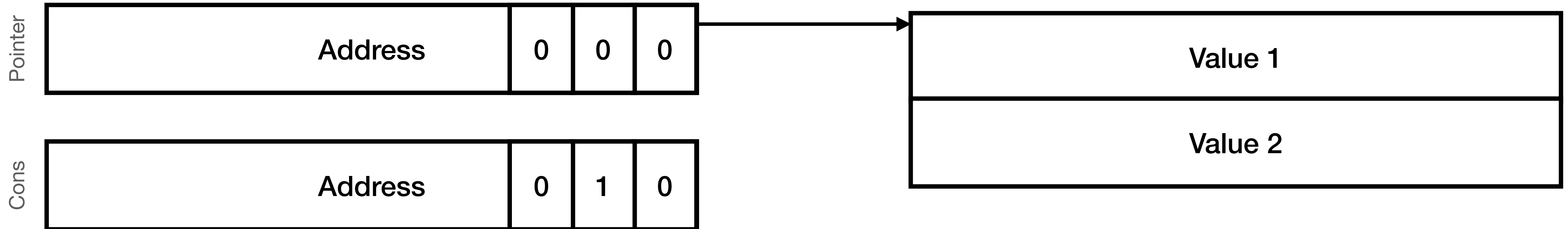
Immediate tag

Encoding pointer values (Hustle)

Type tag in least significant bits

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

Values that point to memory



Idea:

A pair is allocated as two words in memory

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

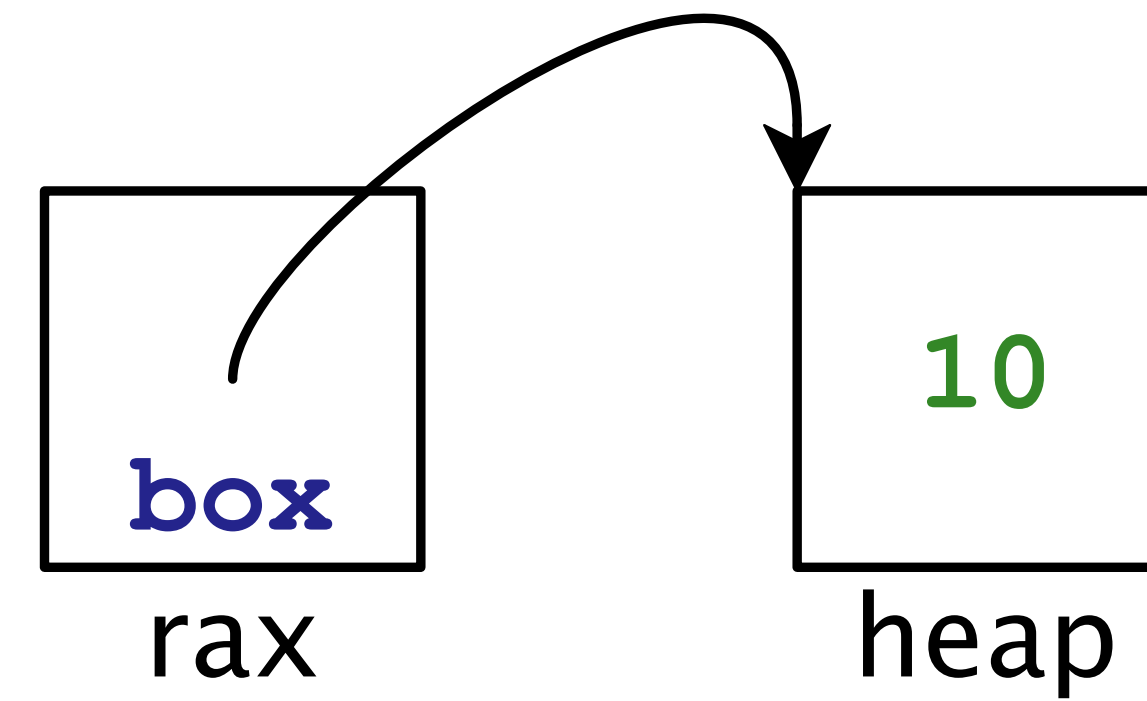
Pointers end in #b000 because we allocate in multiples of 8 bytes: advantage of this and stash type tag there (no shifting).

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

Creating a box

A box is a tagged pointer to one value in memory

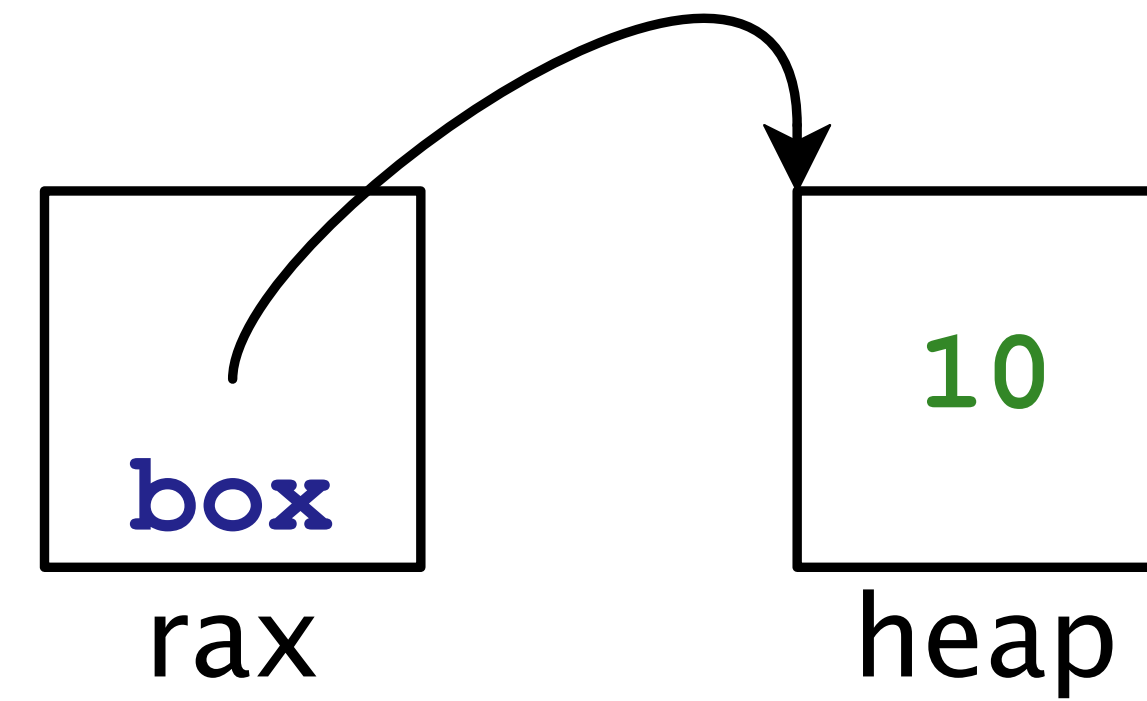
```
> (compile-op1 'box)
(list
  (Mov (Mem 'rbx) 'rax)
  (Mov 'rax 'rbx)
  (Xor 'rax 1)
  (Add 'rbx 8))
> (exec (parse '(box 10)))
'#&10
```



Accessing a box

A box is a tagged pointer to one value in memory

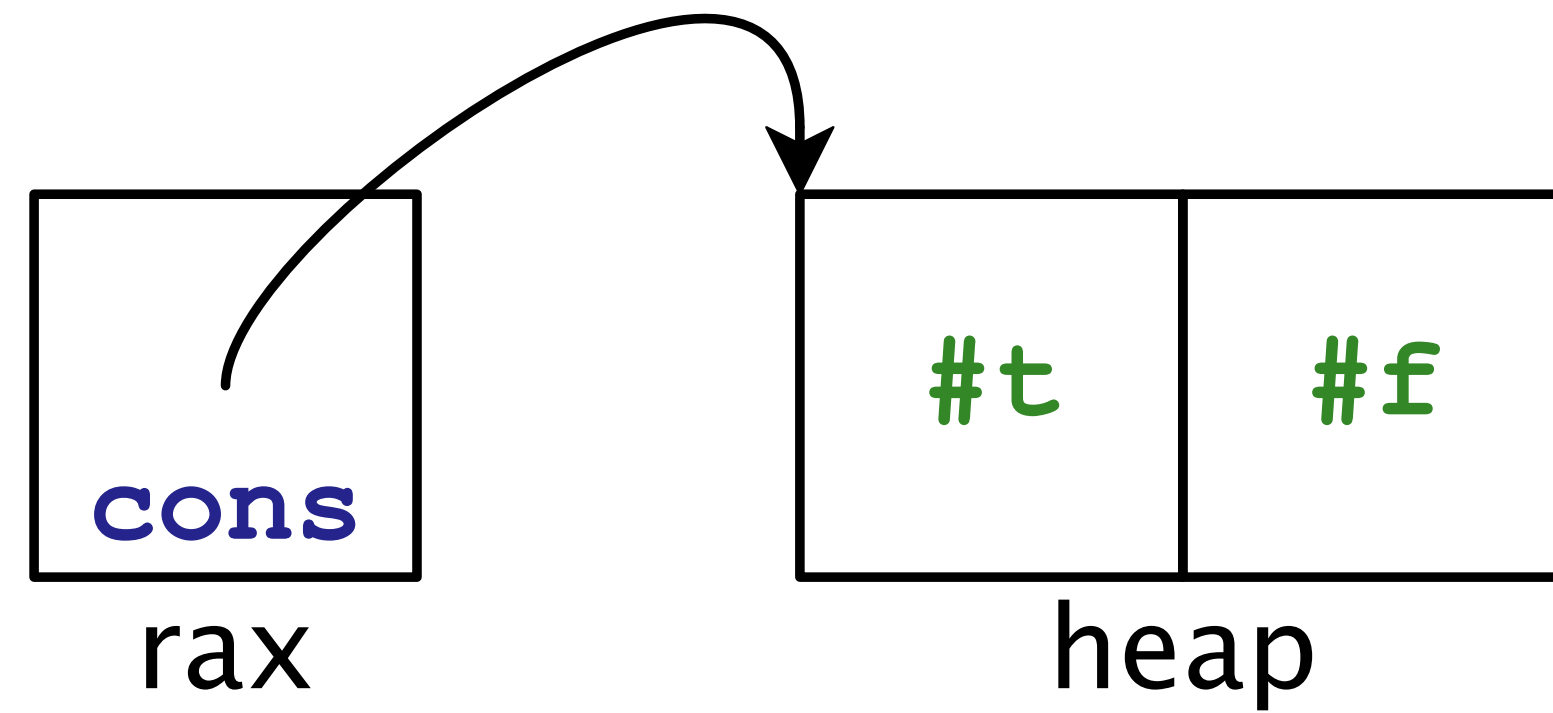
```
> (compile-op1 'unbox)
(list
  (Push 'rax)
  (And 'rax 7)
  (Cmp 'rax 1)
  (Pop 'rax)
  (Jne 'err)
  (Mov 'rax (Mem 'rax -1)))
```



Creating a pair

A pair is a tagged pointer to two values in memory

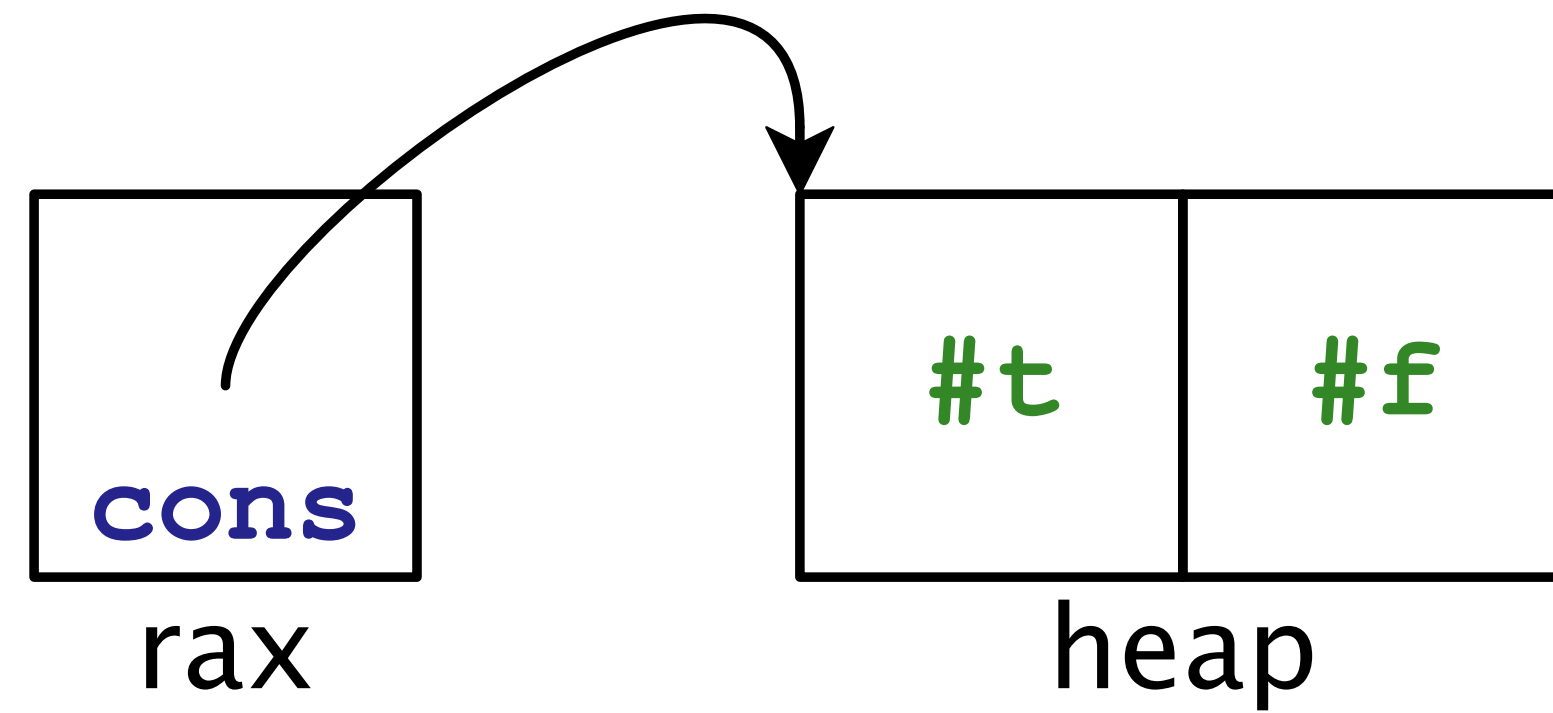
```
> (compile-op2 'cons)
(list
  (Mov (Mem 'rbx 8) 'rax)
  (Pop 'rax)
  (Mov (Mem 'rbx 0) 'rax)
  (Mov 'rax 'rbx)
  (Xor 'rax 2)
  (Add 'rbx 16))
> (exec (parse '(cons #t #f)))
'(#t . #f)
```



Accessing a pair

A pair is a tagged pointer to two values in memory

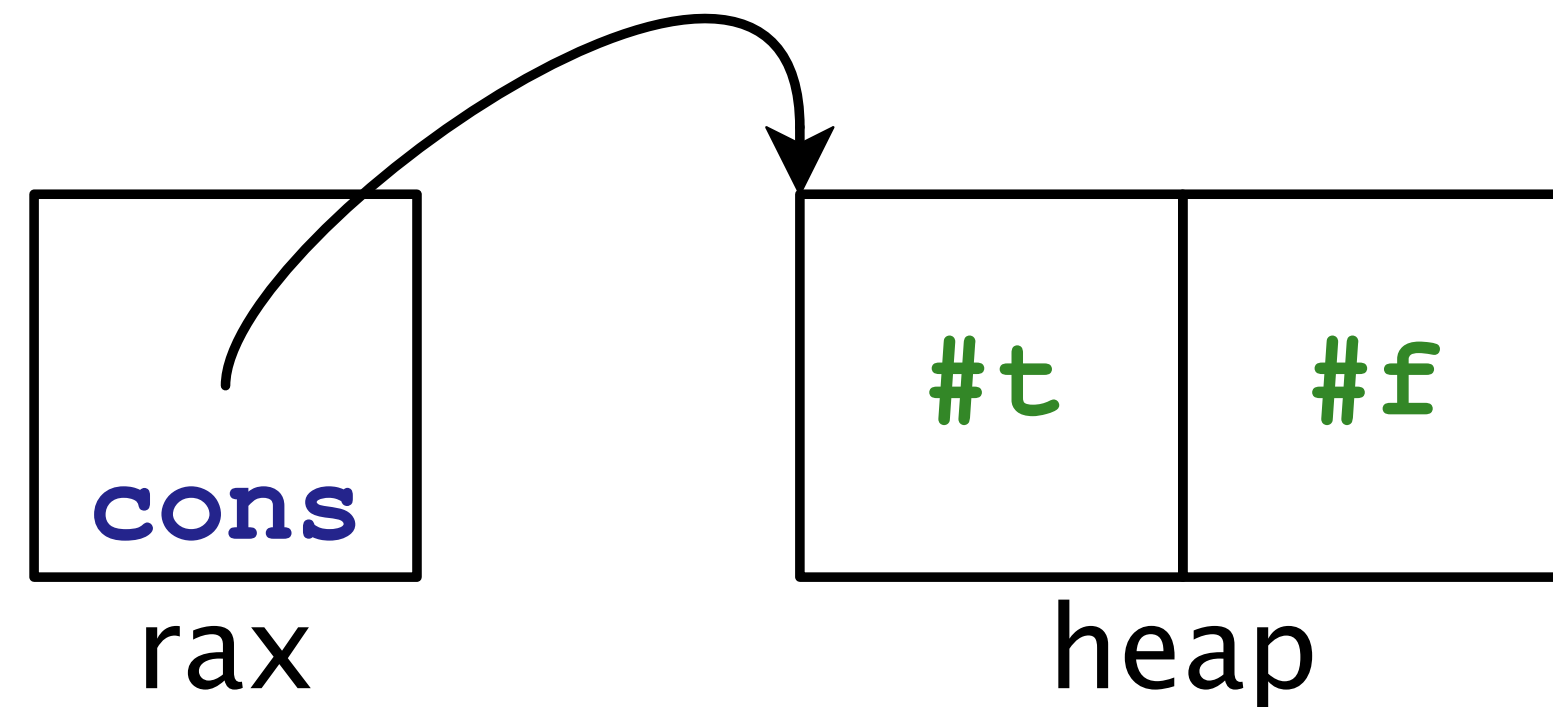
```
> (compile-op1 'car)
(list
  (Push 'rax)
  (And 'rax 7)
  (Cmp 'rax 2)
  (Pop 'rax)
  (Jne 'err)
  (Mov 'rax (Mem 'rax -2)))
```



Accessing a pair

A pair is a tagged pointer to two values in memory

```
> (compile-op1 'cdr)
(list
  (Push 'rax)
  (And 'rax 7)
  (Cmp 'rax 2)
  (Pop 'rax)
  (Jne 'err)
  (Mov 'rax (Mem 'rax 6)))
```



What about the empty list?

The empty list is just an immediate with a unique bit pattern

```
> (compile-datum '())  
(list (Mov 'rax 152))
```

Heap support from standalone run-time

Run-time allocates heap

Passes pointer as argument

Frees heap after return

```
#include <stdio.h>
#include <stdlib.h>
#include "values.h"
#include "print.h"
#include "runtime.h"

/* in words */
#define heap_size 10000

int main(int argc, char **argv)
{
    val_t *heap = malloc(8 * heap_size);
    if (!heap) {
        fprintf(stderr, "out of memory\n");
        return 1;
    }

    val_t result = entry(heap);

    print_result(result);
    if (val_typeof(result) != T_VOID)
        putchar('\n');

    free(heap);
    return 0;
}
```


Heap support from hosted run-time

Run-time allocates heap

Installs pointer in rdi register

```
(define heap
  (malloc 10000 'raw))

(define (asm-interp/host asm)
  (parameterize ([current-externs ...])
    (let ((l (gensym)))
      (asm-interp
        (prog (Global l)
              (Label l)
              (Mov rdi (cast heap _pointer _int64))
              (Jmp 'entry)
              asm))))))
```

Decoding bits

Decoding is now recursive

```
;; Integer -> Value
(define (bits->value b)
  (cond [(= b (value->bits #t)) #t]
        [(= b (value->bits #f)) #f]
        [(= b (value->bits eof)) eof]
        [(= b (value->bits (void))) (void)]
        [(= b (value->bits '())) '()]
        [(int-bits? b)
         (arithmetic-shift b (- int-shift))]
        [(char-bits? b)
         (integer->char (arithmetic-shift b (- char-shift)))]
        [(box-bits? b)
         (box (bits->value (mem-ref (- b type-box))))]
        [(cons-bits? b)
         (cons (bits->value (mem-ref (+ 0 (- b type-cons))))
               (bits->value (mem-ref (+ 8 (- b type-cons)))))]
        [else (error "invalid bits")]))
```

Decoding bits

Decoding is now recursive

```
;; Integer -> Value
(define (bits->value b)
  (cond [(= b (value->bits #t)) #t]
        [(= b (value->bits #f)) #f]
        [(= b (value->bits eof)) eof]
        [(= b (value->bits (void))) (void)]
        [(= b (value->bits '())) '()]
        [(int-bits? b)
         (arithmetic-shift b (- int-shift))]
        [(char-bits? b)
         (integer->char (arithmetic-shift b (- char-shift)))]
        [(box-bits? b)
         (box (bits->value (mem-ref (- b type-box))))]
        [(cons-bits? b)
         (cons (bits->value (mem-ref (+ 0 (- b type-cons))))
               (bits->value (mem-ref (+ 8 (- b type-cons)))))]
        [else (error "invalid bits")]))
```


Printing bits

Printing is now recursive

```
void print_result(val_t x)
{
    switch (val_typeof(x)) {
    case T_INT:
        printf("%" PRIu64, 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_EMPTY:
    case T_BOX:
    case T_CONS:
        printf("'");
        print_result_interior(x);
        break;
    case T_INVALID:
        printf("internal error");
    }
}
```

```
void print_result_interior(val_t x)
{
    switch (val_typeof(x)) {
    case T_EMPTY:
        printf("()");
        break;
    case T_BOX:
        printf("#&");
        print_result_interior(val_unwrap_box(x)->val);
        break;
    case T_CONS:
        printf("(");
        print_cons(val_unwrap_cons(x));
        printf(")");
        break;
    default:
        print_result(x);
    }
}
```

```
void print_cons(val_cons_t *cons)
{
    print_result_interior(cons->fst);

    switch (val_typeof(cons->snd)) {
    case T_EMPTY:
        // nothing
        break;
    case T_CONS:
        printf(" ");
        print_cons(val_unwrap_cons(cons->snd));
        break;
    default:
        printf(" . ");
        print_result_interior(cons->snd);
        break;
    }
}
```

CMSC 430 - 15 June 2026

Array data; function definitions and calls

- Exam review?
- Hoax - Vectors and strings
- Iniquity - Function definitions and calls

CMSC 430 - 15 June 2026

Announcements

- Exam 1 not yet graded, but will be soon
- Assignment 3 grades released later today
- Issue with Assignment 5 autograder, will be fixed after class
- Quiz released after class, due by start of class tomorrow
- Reminder: Assignment 4 is due tonight
- Reminder: Friday is a holiday

Hints for Assignment 5

Some thoughts on the assignment

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

Noax

Hoax: Adding vectors

Concrete examples

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

Hoax: Adding vectors

AST additions

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

Hoax: Adding vectors

Semantic additions

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


Hoax: Adding vectors

Semantic additions

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

Hoax: Adding vectors

Semantic additions

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


Encoding pointer values (Hoax)

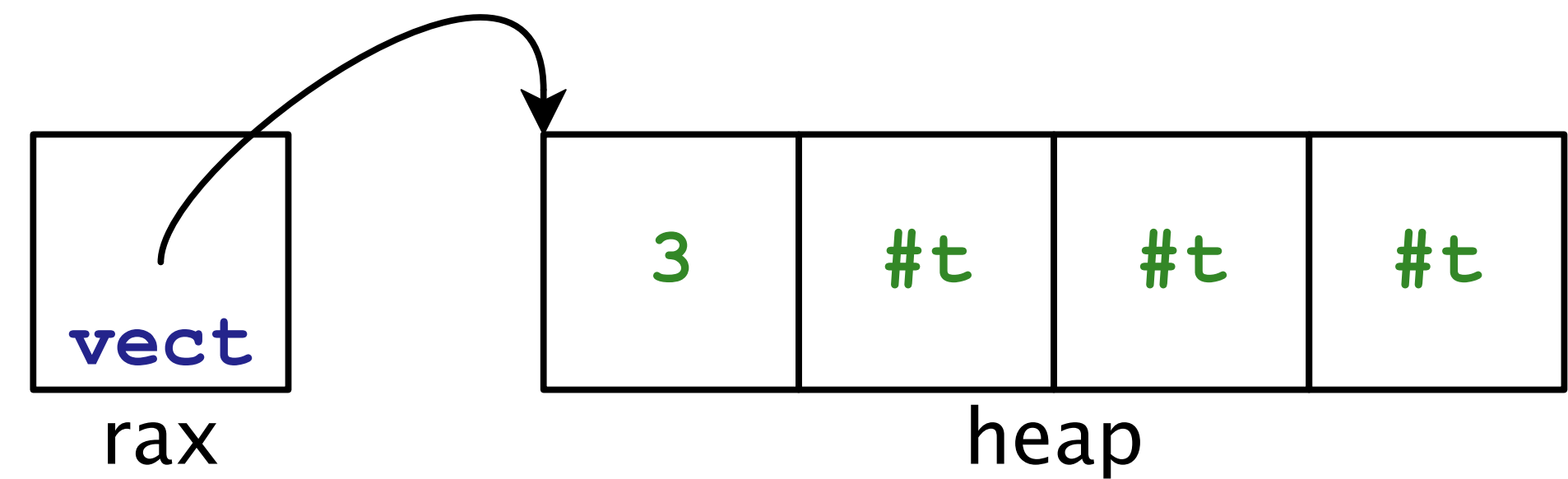
Type tag in least significant bits

61-bits for address	001	Box
61-bits for address	010	Cons
61-bits for address	011	Vector
61-bits for address	100	String

Creating a vector

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

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



For next time

- Read Hoax notes
- Take ELMS quiz
- Submit assignment 4 tonight

CMSC 430 - 16 June 2026

Array data; function definitions and calls

- Hoax - Vectors and strings
 - Review of make-vector
 - Dealing with the empty vector
 - Strings
 - Static memory
- Iniquity - Function definitions and calls

CMSC 430 - 16 June 2026

Announcements

- Exam 1 graded, released after class
- Regrade process will be posted to Piazza today
- Assignment 3 grades released
- Assignment 4 grades released later today
- `raco pkg update a86`
- Quiz released after class, due by start of class tomorrow
- Reminder: Friday is a holiday

Noax

Encoding pointer values (Hoax)

Type tag in least significant bits

61-bits for address	001	Box
61-bits for address	010	Cons
61-bits for address	011	Vector
61-bits for address	100	String

make-vector

Special case for size 0

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

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


make-vector

When size > 0

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

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

make-vector

Constructing return value and account for memory use

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

Static memory

Code lives in memory, so you can make data part of the program

```
; create '#(#t #f 99)
(Mov rax (value->bits 3))
(Mov (Mem rbx 0) rax)
(Mov rax (value->bits #t))
(Mov (Mem rbx 8) rax)
(Mov rax (value->bits #f))
(Mov (Mem rbx 16) rax)
(Mov rax (value->bits 99))
(Mov (Mem rbx 24) rax)
(Mov rax rbx)
(Xor rax type-vect)
(Add rbx 32)
```

If you know the vector you want to make at compile-time, you don't need to do this...

Static memory

Code lives in memory, so you can make data part of the program

```
(Data)
; create '#(#t #f 99)
(Label 'myvect)
(Dq (value->bits 3))
(Dq (value->bits #t))
(Dq (value->bits #f))
(Dq (value->bits 99)))
```

...you can statically allocate memory in the data section.

And create a tagged pointer to it.

```
(Lea rax (Mem 'myvect type-vect))
```

Static memory

Code lives in memory, so you can make data part of the program

```
(Data)  
; create '#()  
(Label 'empty)  
(Dq (value->bits 0))
```

Very useful for canonically
representing the empty vector.

make-vector

Special case for size 0

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

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



Hoax: Adding strings

Concrete examples

(make-string 3 #\t)	=> "ttt"
(let ((s (make-string 3 #\t))) (string-ref s 0))	=> #\t
(let ((s (make-string 3 #\t))) (string-ref s 3))	=> err
(let ((s (make-string 3 #\t))) (string-length s))	=> 3
"fred"	=> "fred"

Hoax: Adding strings

AST additions

```
:: type Lit = ... | String
:: type Op1 = ...
::          | 'string? | 'string-length
:: type Op2 = ...
::          | 'make-string | 'string-ref
```


Hoax: Adding strings

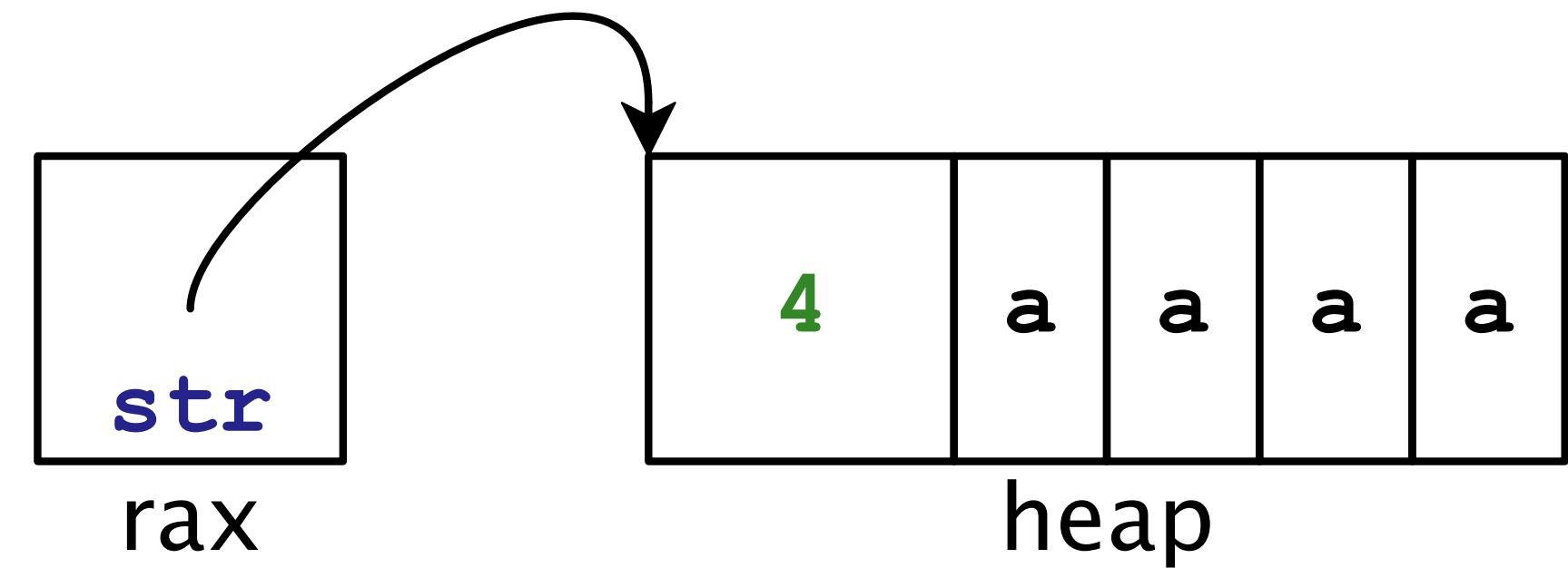
Semantic additions

```
:: type Value = ...  
:: | (string Char ...)
```

Creating a string

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

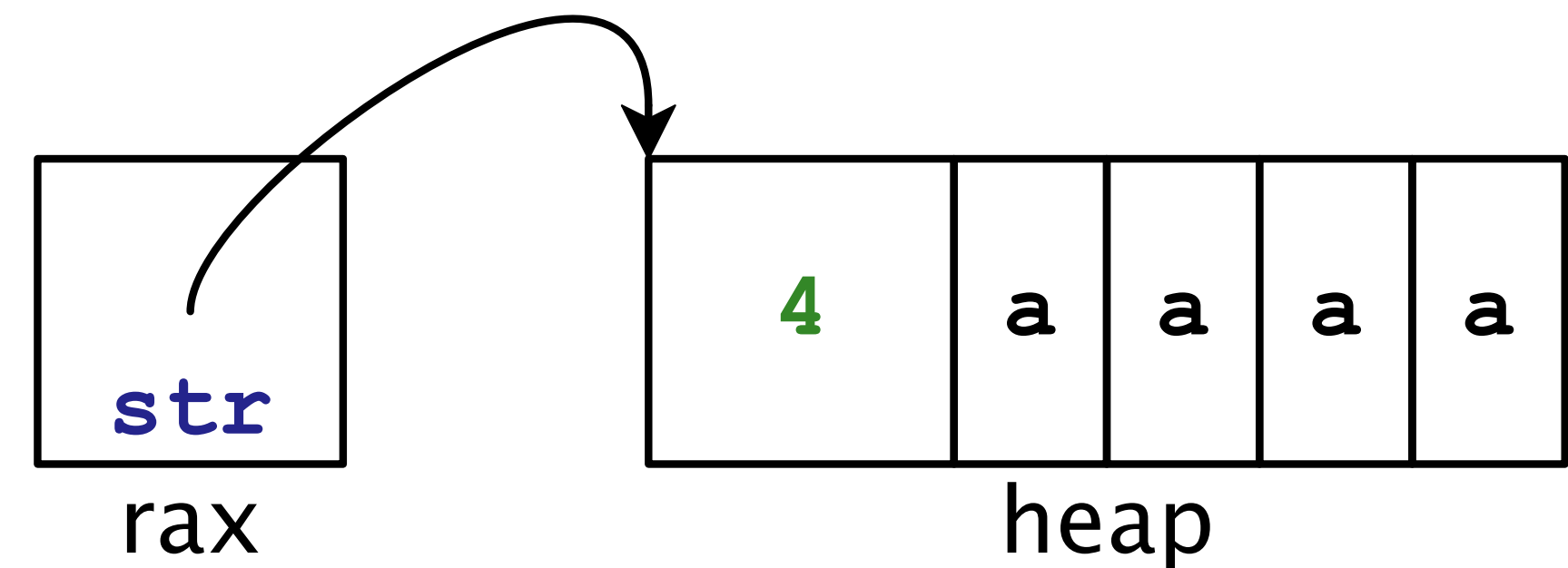
```
> (exec (parse '(make-string 4 #\a)))  
"aaaa"
```



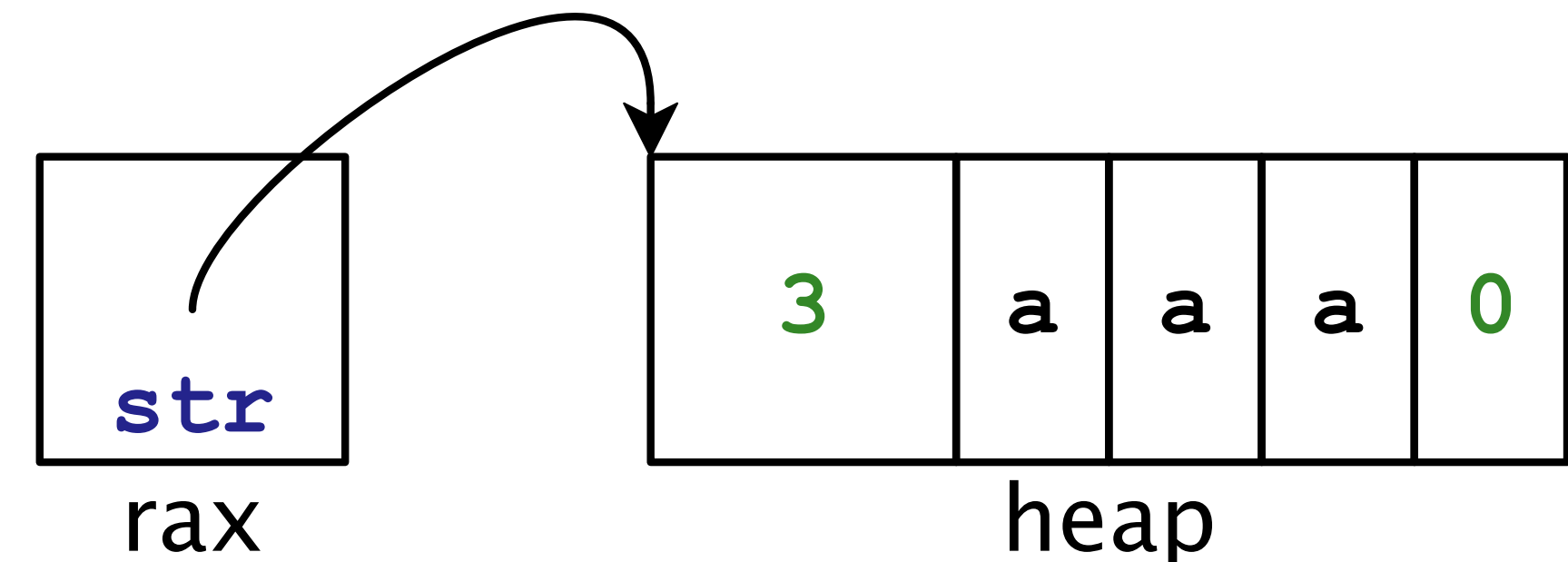
Creating a string

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

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



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



string-length

Just like vector-length

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

string-ref

Similar to vector-ref, but different

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

string-ref

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

vector-ref

string-set!

We don't include, thus strings are immutable

[This space intentionally blank.]

make-string

Similar to make-vector, but different

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

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

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

      (Sar rax char-shift) ; convert to codepoint

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

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

Let's take it in pieces

make-string (1/8)

Similar to make-vector, but different

```
(Pop r8)
```

```
(assert-natural r8)
```

Just like make-vector

make-string (2/8)

Similar to make-vector, but different

```
(assert-char rax)
```

Unlike `make-vector`

make-string (3/8)

Similar to make-vector, but different

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

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

make-string (4/8)

Similar to make-vector, but different

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

Just like `make-vector`, but half
as many bytes

make-string (5/8)

Similar to make-vector, but different

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

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

make-string (6/8)

Similar to make-vector, but different

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

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

make-string (7/8)

Similar to make-vector, but different

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

Just like make-vector

make-string (8/8)

Similar to make-vector, but different

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

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

String literals

String literals are compiled to static memory

```
> (compile-e (Lit "wilma") '())  
(list  
  (Data)  
  (Label 'string435421)  
  (Dq 80)  
  (Dd 119)  
  (Dd 105)  
  (Dd 108)  
  (Dd 109)  
  (Dd 97)  
  (Dd 0)  
  (Text)  
  (Lea 'rax (Mem 'string435421 4)))
```


Iniquity

Example Iniquity program

Concrete example

```
#lang racket
(define (add1-list lon)
  (if (empty? lon)
      '()
      (cons (add1 (car lon))
            (add1-list (cdr lon)))))

(add1-list (cons 1 (cons 2 (cons 3 '()))))
```

Example Iniquity program

Concrete example

```
#lang racket
(define (read-bytes)
  (let ((b (read-byte)))
    (if (eof-object? b)
        '()
        (cons b (read-bytes)))))

(read-bytes)
```


Example Iniquity program

Concrete example

```
#lang racket
;; copy first n bytes from input to output
(define (copy-bytes n)
  (if (zero? n)
      (void)
      (let ((b (read-byte)))
        (if (eof-object? b)
            (void)
            (begin (write-byte b)
                    (copy-bytes (sub1 n)))))))

(copy-bytes 10)
```

Example Iniquity program

Concrete example

```
#lang racket
(define (even? n)
  (if (zero? n)
      #t
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      #f
      (even? (sub1 n))))

(odd? 4097)
```

Syntactic changes

Programs are now sequences of definitions and an expression

Concrete syntax:

Programs:

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

Expressions:

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

Abstract syntax:

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

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

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


Parsing changes

Programs are now sequences of definitions and an expression

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

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

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

Semantic changes

Expressions are interpreted with respect to set of definitions

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

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


Function calls in general

What should this evaluate to?

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

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

Same thing as:

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

Designing our own calling convention

Function calls are like “let at a distance”

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

is like

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

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

For next time

- Read Iniquity notes
- Take ELMS quiz

CMSC 430 - 17 June 2026

Compiling function definitions and calls

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

CMSC 430 - 17 June 2026

Announcements

- Exam 1 grades; regrade request due by Monday midnight
- Assignment 4 grades released later today
- Assignment 6 released later today
- Quiz released after class, due by start of class tomorrow
- Reminder: Friday is a holiday

Iniquity

Syntactic changes

Programs are now sequences of definitions and an expression

Concrete syntax:

Programs:

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

Expressions:

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

Abstract syntax:

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

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

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


Semantic changes

Expressions are interpreted with respect to set of definitions

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

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

New kinds of error

Applying a function to the wrong number of arguments

```
#lang racket  
(define (f x y) x)  
(f 1)
```

We will consider this a run-time error even though we can detect this at compile time.

Future languages won't be able to statically determine arity errors.

Semantic changes

Expressions are interpreted with respect to set of definitions

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


Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; Expr Env Defns -> Value { raises 'err }  
(define (interp-e e r ds) ;; where r closes e  
  (match e  
    ;...  
    [(App f es)  
     (let ((vs (interp-e* es r ds)))  
       (match (defns-lookup ds f)  
         [(Defn f xs e)  
          ; check arity matches  
          (if (= (length xs) (length vs))  
              (interp-e e (zip xs vs) ds)  
              (raise 'err))]))]))])
```

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; (Listof Expr) Env Defns  
;;    -> (Listof Value) { raises 'err }  
(define (interp-e* es r ds)  
  (match es  
    [ '() '()]  
    [(cons e es)  
     (cons (interp-e e r ds)  
            (interp-e* es r ds))]))
```

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; Defns Symbol -> Defn  
(define (defns-lookup ds f)  
  (match ds  
    [(cons (Defn g xs e) ds)  
     (if (eq? f g)  
         (Defn g xs e)  
         (defns-lookup ds f))]))
```


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)

(Jump '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*

For next time

- Read Jig notes
- Take ELMS quiz

CMSC 430 - 18 June 2026

Tail calls

- Brief look at Assignment 6 and 7
- Jig - Tail calls

CMSC 430 - 18 June 2026

Announcements

- Assignment 6 released, 7 released later today
- Quiz released after class, due by start of class **Monday**
- No lecture tomorrow

Jig

Consider this program

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

(f 100)
```

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

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

A view of the stack

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

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

A view of the stack

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

;; Morally:

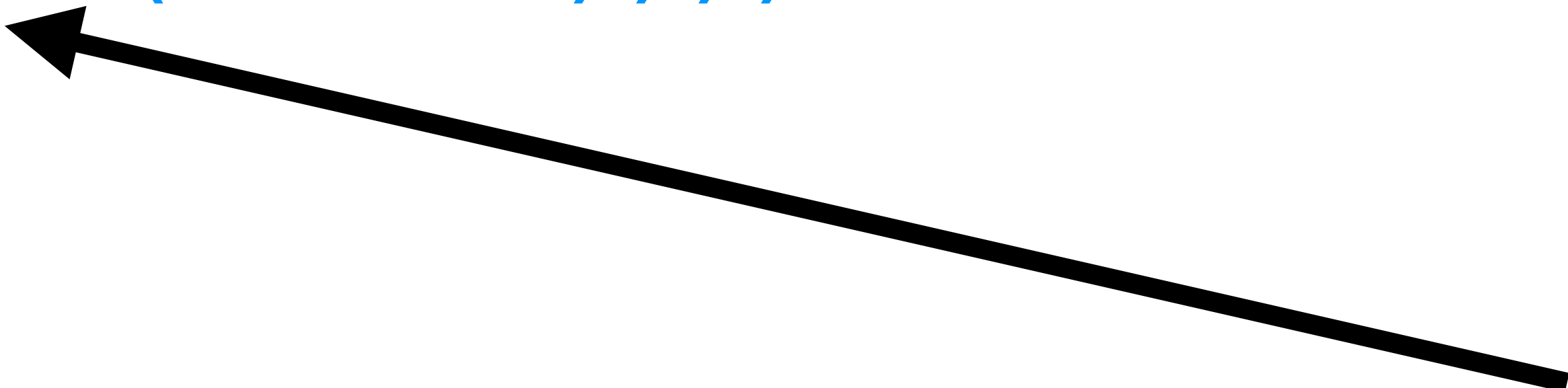
```
(seq (Mov 'rax (value->bits 100))
      (Lea 'r8 'ret1)
      (Push 'r8)
      (Push 'rax)
      (Jmp 'f)
      (Label 'ret1)
      (Ret)

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

A view of the stack

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

(f 100)
```



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

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

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

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

Pop before call

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



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

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

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

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


Pop before call

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



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

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

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

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

Pop before call

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



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

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

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

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

Revised view of the stack

```

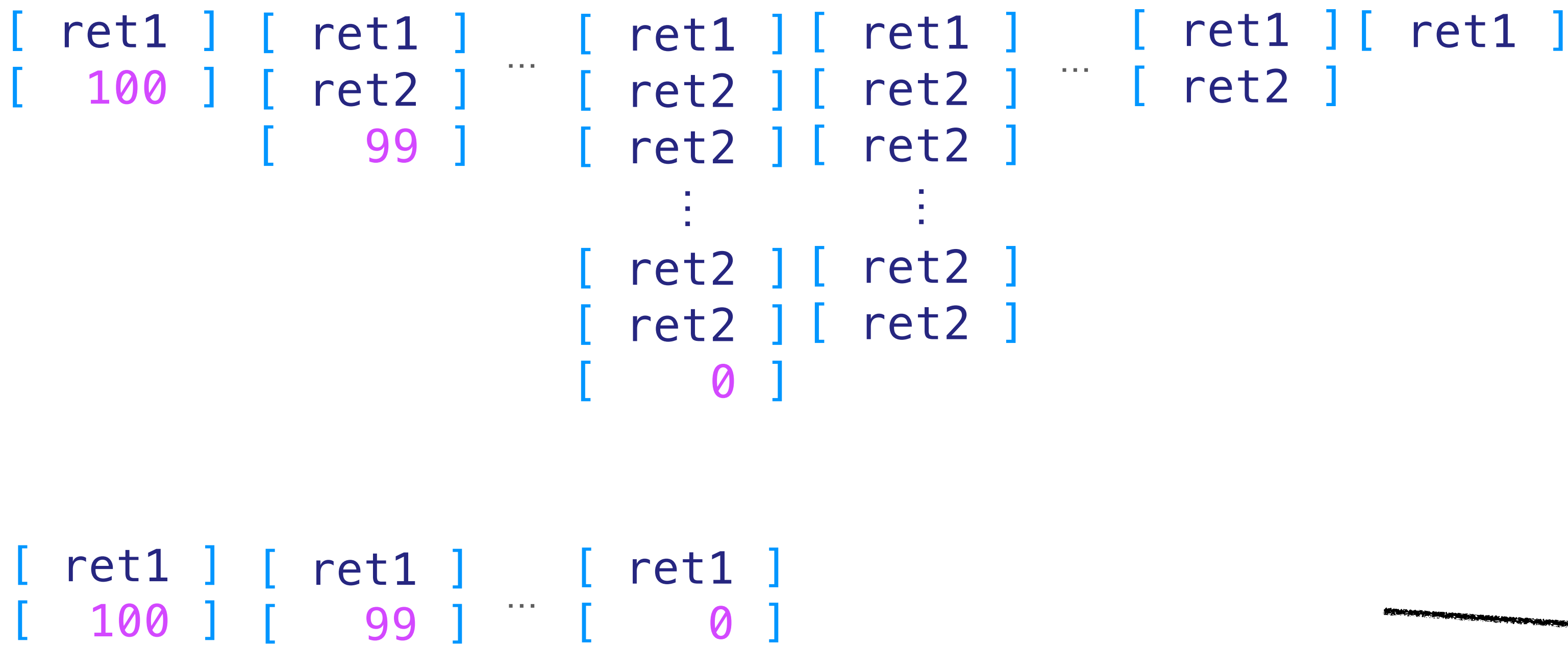
[ ret1 ] [ ret1 ] ... [ ret1 ] [ ret1 ] ... [ ret1 ] [ ret1 ]
[ 100 ] [ ret2 ]      [ ret2 ] [ ret2 ]      [ ret2 ]
      [ 99 ]      [ ret2 ] [ ret2 ]
              ⋮              ⋮
      [ ret2 ] [ ret2 ]
      [ ret2 ] [ ret2 ]
      [ 0 ]
  
```

```

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

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


Returning just to return

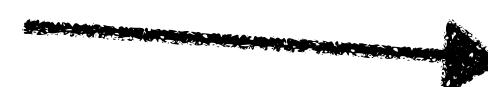
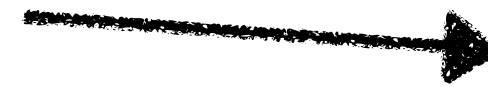


Pushes a return point so that when control returns, we can return to the caller.

What if we just didn't?

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

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



Tail calls (asymptotically) save *SPACE*

Non-tail call:

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

Tail call:

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

(saving space may save time too)

Key idea

Some calls don't need to be returned to

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

- Pop local environment
- Push arguments
- Jump

When can't you do this?

When there's work to do after the call

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

(f 100)

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

(g 100)

Sometimes you can rewrite to enable tail calls

So there's no work to do after the call

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

(g 100)
```

```
(define (g x)
  (g/a x 0)) ✓
```

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

(g 100)
```


Sometimes you can rewrite to enable tail calls

So there's no work to do after the call

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



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



```
;; (append-reverse xs ys)
;;   ≡ (append (reverse xs) ys)
```

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

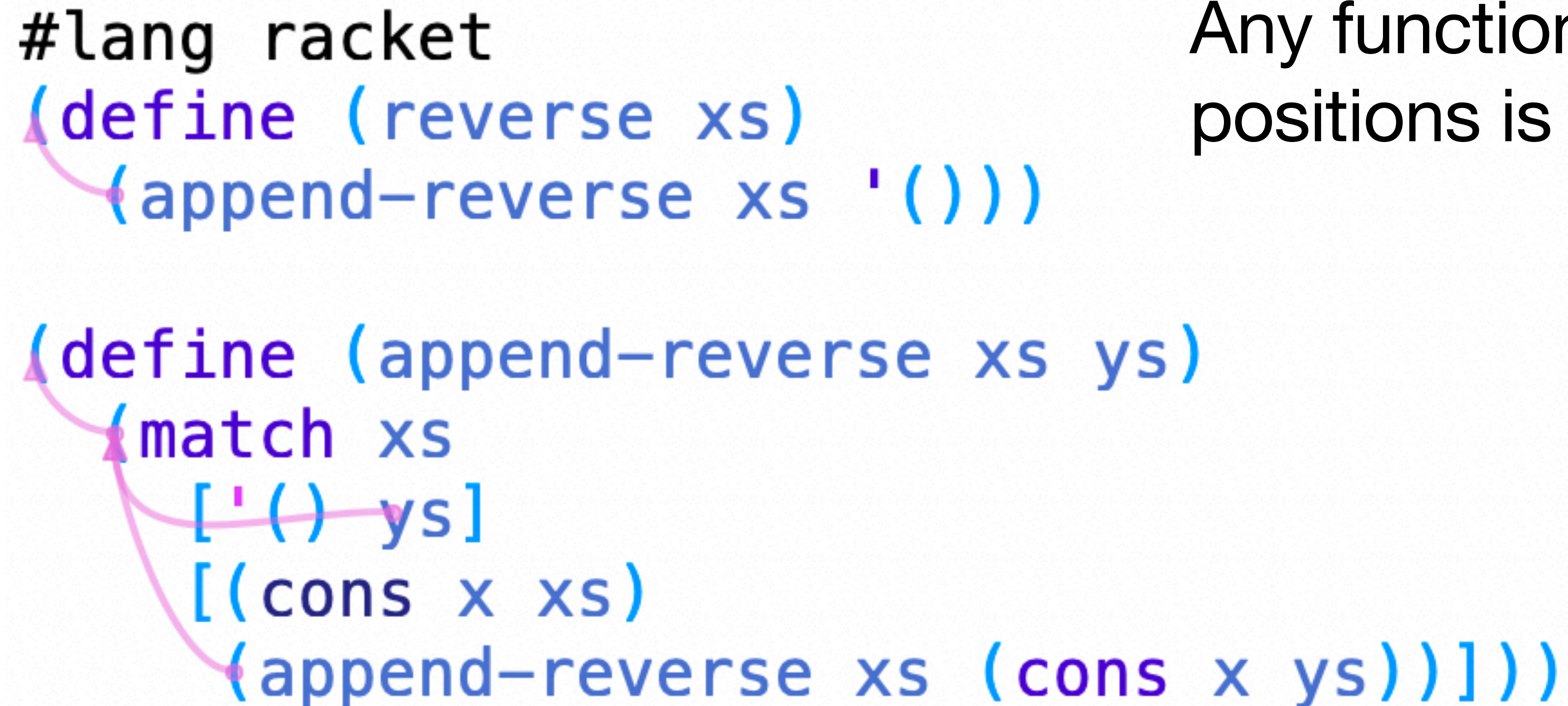


DrRacket will show you tail positions

Hover over a function definition

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

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



Any function call in these positions is a tail call.

When does a call not need to be returned to?

When it occurs in tail position

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

```
<tail> ::=
```

```
  (if e1 <tail> e3)
```

```
  (if e1 e2 <tail>)
```

```
  (let ((x e)) <tail>)
```

```
  (begin e <tail>)
```

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

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

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

When does a call not need to be returned to?

When it occurs in tail position

```
;; Expr CEnv Boolean -> Asm
(define (compile-e e c t?) ...)  t? - is this expression in tail
                                position
```

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

When does a call not need to be returned to?

When it occurs in tail position

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

t? - is this expression in tail position

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


When does a call not need to be returned to?

When it occurs in tail position

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

t? - is this expression in tail position

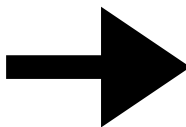
```
;; Id [Listof Expr] CEnv Boolean -> Asm  
(define (compile-app f es c t?)  
  (if t?  
      (compile-app-tail f es c)  
      (compile-app-nontail f es c)))
```

Non-tail calls

More work to do after call, so need to return

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

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



return address
c
return address
v1
v2
v3

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))
```

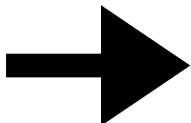


Non-tail calls

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))
```



return address
c
return address
v1
v2
v3

Non-tail calls

More work to do after call, so need to return

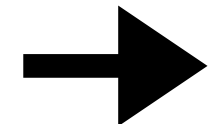
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))
```

```
(seq
```

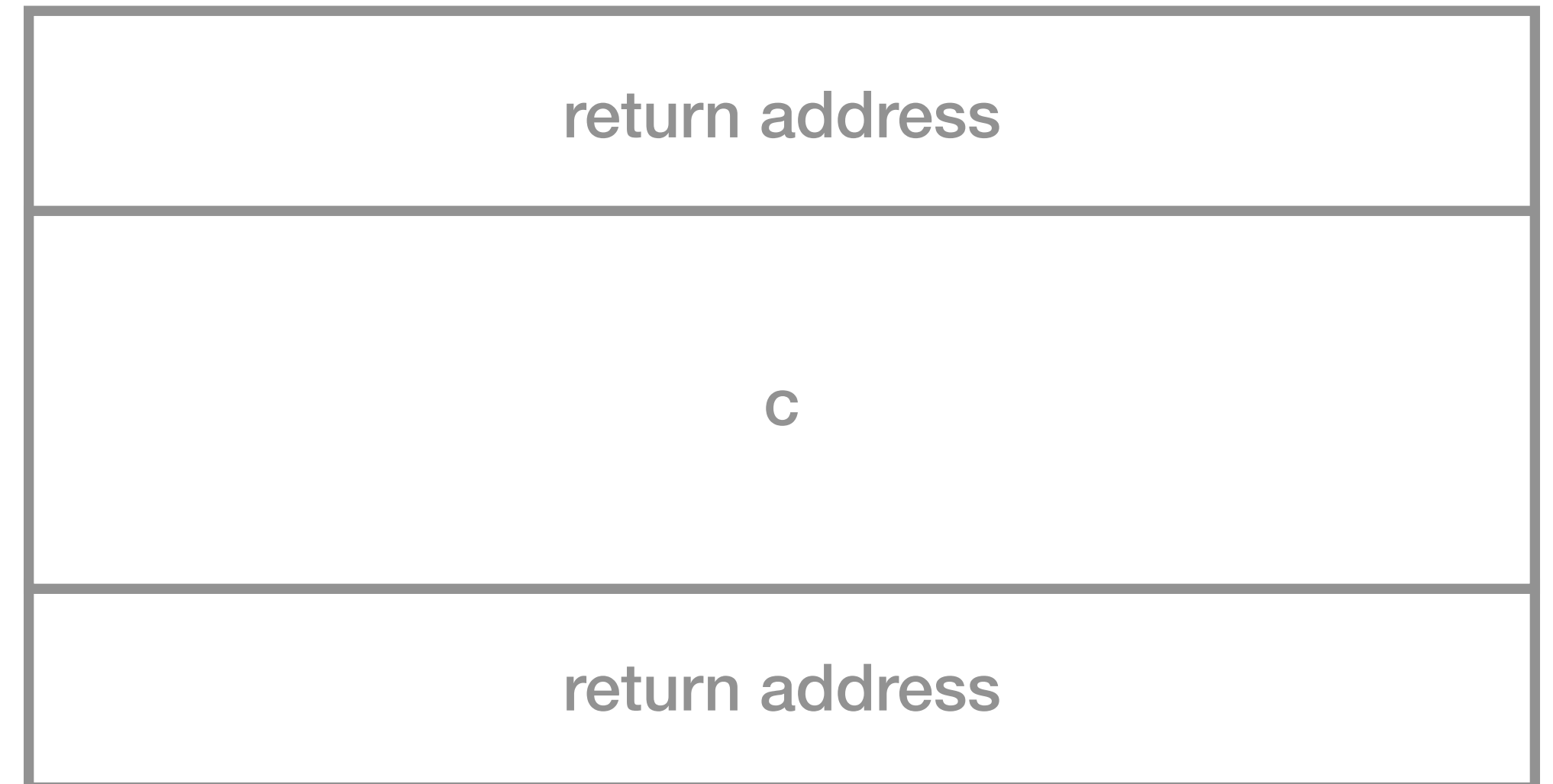
```
  (Label 'f)
```

```
  (compile-e e (list 'z 'y 'x) #t)
```

```
  (Add rsp 24)
```



```
  (Ret))
```

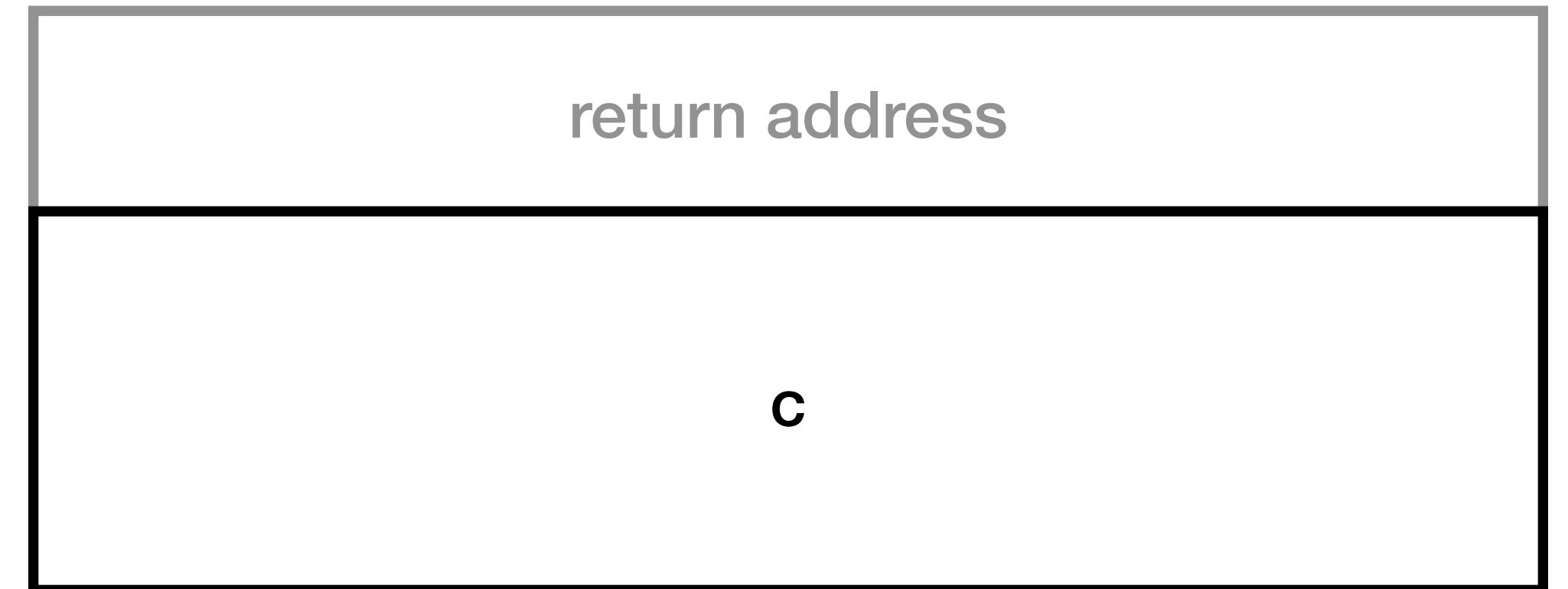
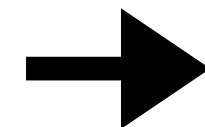


Non-tail calls

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)
  (Jump '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)
```

➔ (Jump 'f))

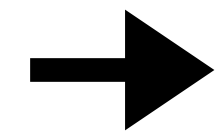
return address
c
v1
v2
v3

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))
```

return address
c
v1
v2
v3

Tail calls (first attempt)

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

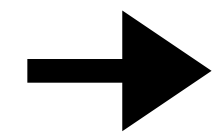
```
(compile-define (Defn 'f (list 'x 'y 'z)) e))
```

```
(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)
```

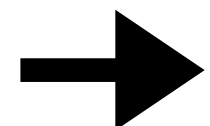
```
  (compile-e e2 (cons #f c))
```

```
  (Push rax)
```

```
  (compile-e e3 (cons #f (cons #f c)))
```

```
  (Push rax)
```

```
  (Jump '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)
```

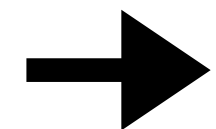
```
  (compile-e e2 (cons #f c))
```

```
  (Push rax)
```

```
  (compile-e e3 (cons #f (cons #f c)))
```

```
  (Push rax)
```

```
  (Add rsp (* 8 (length c)))
```



```
  (Jump 'f))
```

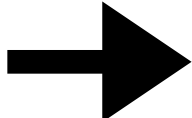
return address
?
?
?

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)))
  (Jump 'f))
```



return address
c
v1
v2
v3

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))
```

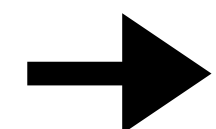
```
  (Push rax)
```

```
  (compile-e e3 (cons #f (cons #f c)))
```

```
  (Push rax)
```

```
  (move-args 3 (length c))
```

```
  (Add rsp (* 8 (length c)))
```



```
  (Jump 'f))
```

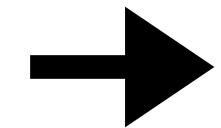
return address
v1
v2
v3

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))
```

return address
v1
v2
v3

Tail calls (final attempt)

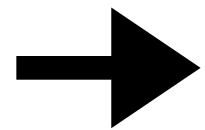
No more work to do after call, so don't return

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



return address

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



For next time

- Read Loot notes
- Take ELMS quiz

CMSC 430 - 23 June 2026

Lambdas and first-class functions

- Loot - Lambda
 - interpreter lambda expressions w/ and w/o functions
 - compiling lambda expressions

CMSC 430 - 23 June 2026

Announcements

- Assignment 7 due Thursday
- Assignment 5 grades released
- Assignment 6 will be released today
- Exam 2 Friday
- Practice exam 2 out later today
- No quiz today - instead last quiz will be extended until tomorrow

Loot

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 = ...  
:: | Function (???)
```

Semantic matters

New kind of value: function

```
:: type Value =  
:: | ((Listof Value) -> Value { raises 'err })
```

Idea: use Racket functions to
represent LooT functions

Lambda expressions

Remember the environment they were created in

```
(let ((f (let ((x 5))  
            (λ (y) (+ x y)))))  
  (f 1))
```

Lambda expressions

Remember the environment they were created in

```
(let ((f (let ((x 5))  
            (λ (y) (+ x y)))))  
  (let ((x 6))  
    (f 1)))
```

Semantic matters

New kind of value: function

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ;; where r closes e
  (match e
    ; ...
    [(Lam _ xs e)
     (λ (vs)
      ; check arity matches
      (if (= (length xs) (length vs))
          (interp-e e (append (zip xs vs) r) ds)
          (raise 'err)))]
    [(App e es)
     (let ((f (interp-e e r ds))
           (vs (interp-e* es r ds)))
       (if (procedure? f)
           (f vs)
           (raise 'err))))))
```


Lambda without Lambda

The essence of a closure

"Save the environment! Create a closure today!" — Cormac Flanagan

A closure pairs together

- a computation that can be run later
- the environment in which it was created

Semantic matters

New kind of value: function (represented as a closure)

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

Semantic matters

New kind of value: function (represented as a closure)

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ;; where r closes e
  (match e
    ; ...
    [(Lam _ xs e)
     (Closure xs e r)]
    [(App e es)
     (let ((f (interp-e e r ds))
           (vs (interp-e* es r ds)))
       (match f
         [(Closure xs e r)
          ; check arity matches
          (if (= (length xs) (length vs))
              (interp-e e (append (zip xs vs) r) ds)
              (raise 'err))]
         [_ (raise 'err)])))]))
```


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

Encoding pointer values (Loot)

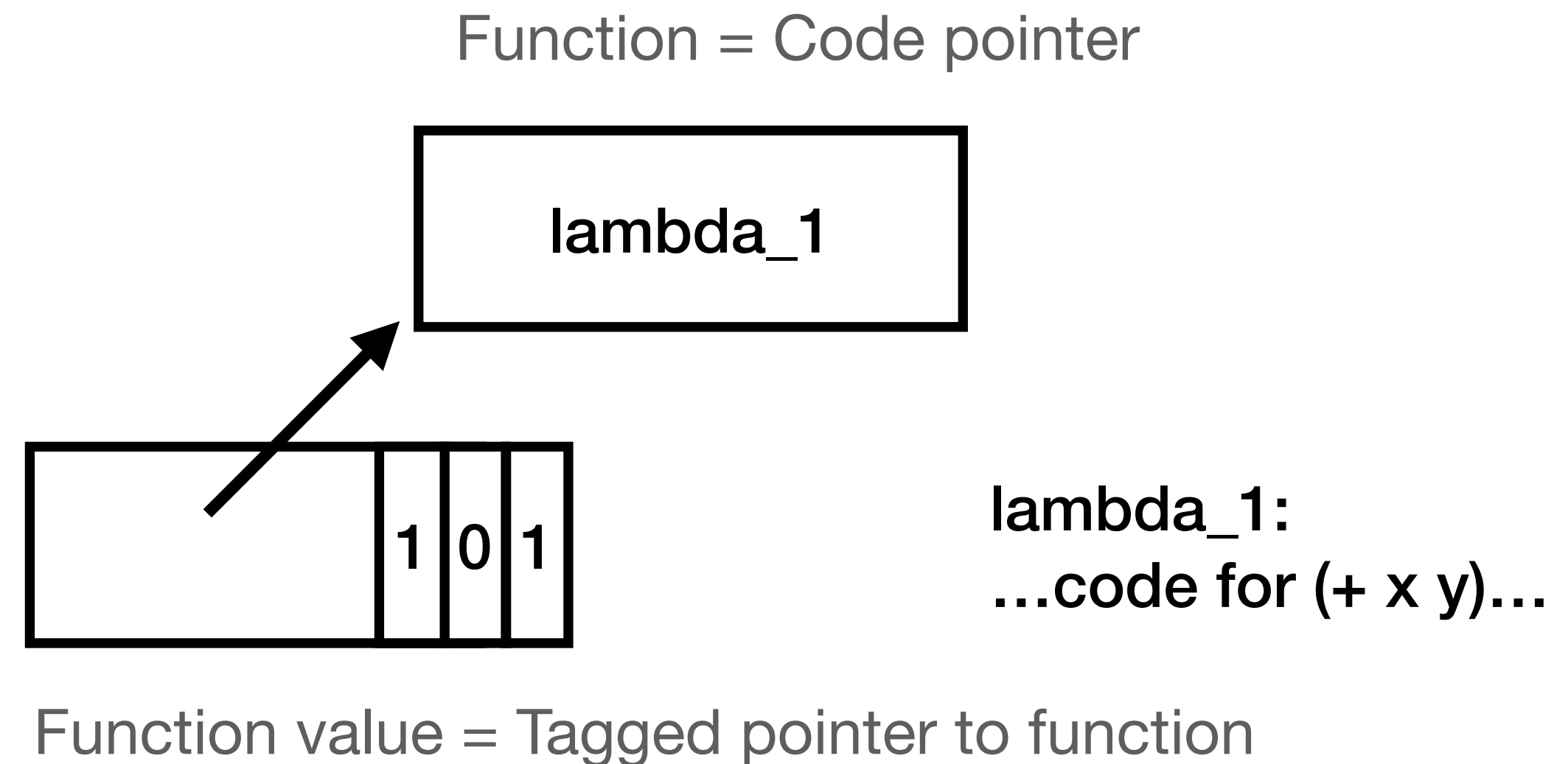
Type tag in least significant bits

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

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 (cons #f xs))))
       (seq (Label (symbol->label f))
            (Cmp r8 (length xs))
            (Jne 'err)
            (compile-e e env #t)
            (Add rsp (* 8 (length 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 (Mem rbx) 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))
          (Mov r8 (length es))
          (Jmp f)
          (Label r)))))
```

```
;; 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 r8 (length es))
```



Compiler for Jig

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 e (append (make-list (add1 (length es)) #f) c))
         (Mov r8 (length es))
         (assert-proc rax)
         (Jmp (Mem rax (- type-proc))))
    (Label r)))
```

Pros:

- saves one word 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.:

```
# 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))
            (Cmp r8 (length xs))
            (Jne 'err)
            (compile-e e env #t)
            (Add rsp (* 8 (length env)))
            (Ret))))))])
```

Representing functions

Functions are a new kind of value

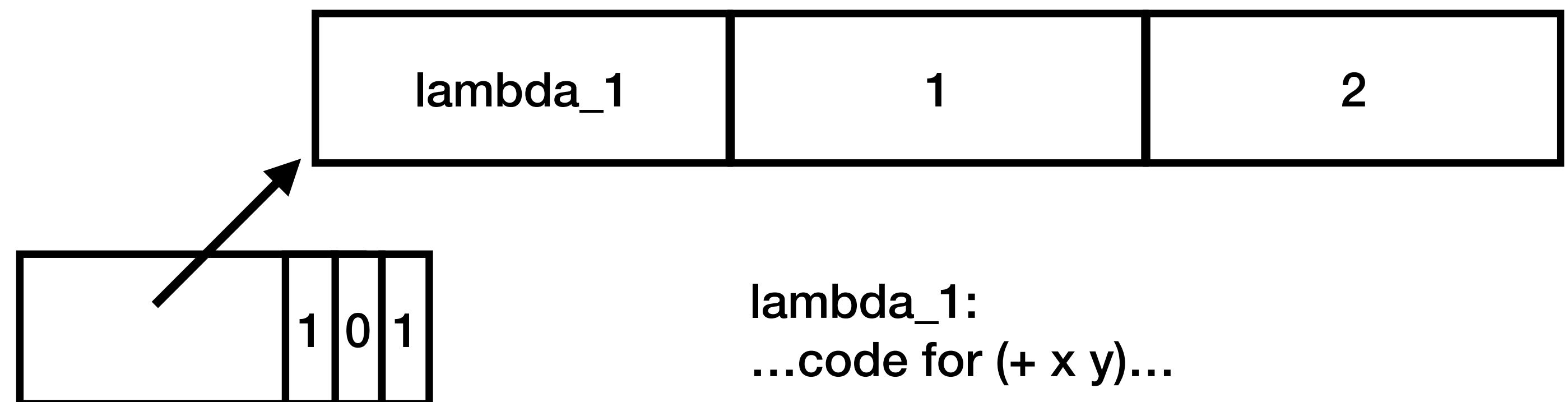
```
:: type Value =  
;; ...  
;; | (Closure [Listof Id] Expr Env)  
(struct Closure (xs e r) #:prefab)
```

Can be used to define a function in assembly

Can be represented as an array of values

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

Closure = Code pointer + run-time environment



Function value = Tagged pointer to closure

For next time

- Read Knock notes
- No quiz today!

CMSC 430 - 24 June 2026

Lambdas and first-class functions

- Loot - Lambda
 - wrapping up closures
- Knock - Pattern matching
 - interpreting patterns

CMSC 430 - 24 June 2026

Announcements

- Assignment 7 due Thursday
- Assignment 6 grades released
- Exam 2 Friday
- Practice exam 2 out later today
- Quiz released after class, due by start of class tomorrow

Loot

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

Representing functions

Functions are a new kind of value

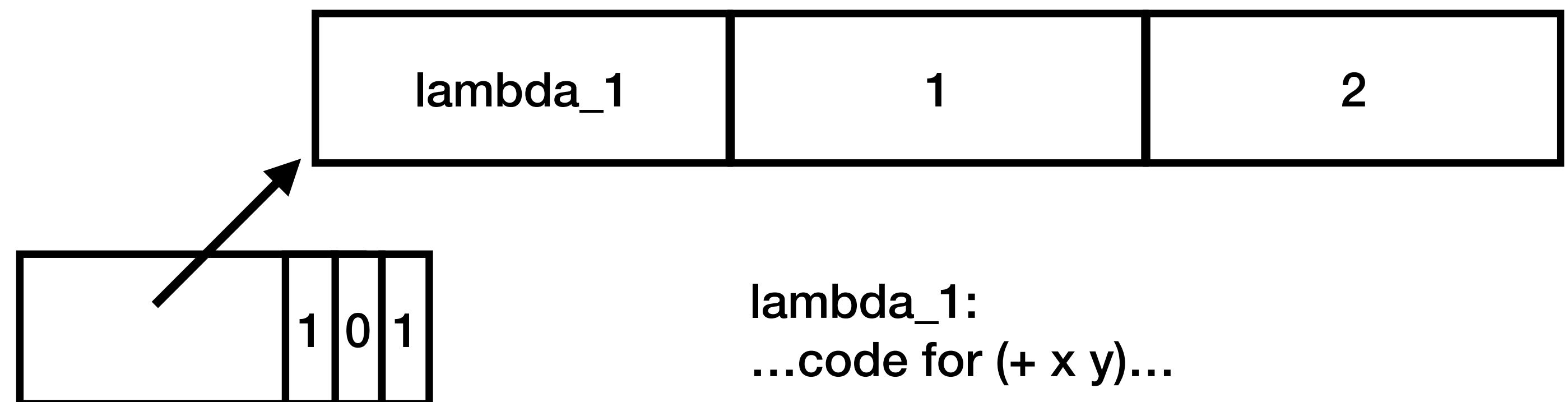
```
:: type Value =  
:: ...  
:: | (Closure [Listof Id] Expr Env)  
(struct Closure (xs e r) #:prefab)
```

Can be used to define a function in assembly

Can be represented as an array of values

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

Closure = Code pointer + run-time environment



Function value = Tagged pointer to closure

Lambda expressions evaluate to function values

Final take: lambda expressions evaluate to tagged pointers to code label AND ENVIRONMENT

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

Assumes closed!!

```
;; Id [Listof Id] Expr CEnv -> Asm
(define (compile-lam f xs e c)
  (let ((fvs (fv (Lam f xs e))))
    (seq (Lea rax (symbol->label f))
          (Mov (Mem rbx) rax)
          (free-vars-to-heap fvs c 8)
          (Mov rax rbx) ; return value
          (Xor rax type-proc)
          (Add rbx (* 8 (add1 (length fvs)))))))
```

Lambda expressions evaluate to function values

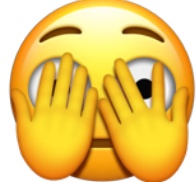
Final take: lambda expressions evaluate to tagged pointers to code label AND ENVIRONMENT

```
;; [Listof Id] CEnv Int -> Asm
(define (free-vars-to-heap fvs c off)
  (match fvs
    ['() (seq)]
    [(cons x fvs)
     (seq (Mov r8 (Mem rsp (lookup x c)))
          (Mov (Mem rbx off) r8)
          (free-vars-to-heap fvs c (+ off 8))))]))
```


Lambda expressions define functions

Final take: functions install the environment of from the closure

```
;; Lam -> Asm
(define (compile-lambda-define l)
  (let ((fvs (fv l)))
    (match l
      [(Lam f xs e)
       (let ((env (append (reverse fvs) (reverse xs) (list #f))))
         (seq (Label (symbol->label f))
              (Cmp r8 (length xs))
              (Jne 'err)

              
              (compile-e e env #t)
              (Add rsp (* 8 (length env))) ; pop env
              (Ret)))))])))
```

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

Lambda expressions define functions

Final take: functions install the environment of from the closure

```
;; [Listof Id] Int -> Asm
;; Copy the closure environment at given offset to stack
(define (copy-env-to-stack fvs off)
  (match fvs
    ['() (seq)]
    [(cons _ fvs)
     (seq (Mov r9 (Mem rax (- off type-proc)))
          (Push r9)
          (copy-env-to-stack fvs (+ 8 off))))]))
```

Knock

Example Knock program

Concrete example

```
#lang racket
(define (add1-lon xs)
  (match xs
    ['() '()]
    [(cons x xs)
     (cons (add1 x)
            (add1-lon xs))]))

(add1-lon (cons 1 (cons 2 (cons 3 '()))))
```

Syntax

Match and patterns

```
;; type Expr = ...  
;;           | (Match Expr (Listof Pat) (Listof Expr))  
  
;; type Pat  = (Var Id)  
;;           | (Lit Datum)  
;;           | (Box Pat)  
;;           | (Cons Pat Pat)  
;;           | (Conj Pat Pat)
```

Interpretation

Match expressions

```
;; Expr Env Defns -> Value { raises 'err }  
(define (interp-e e r ds)  
  (match e  
    ; ...  
    [(Match e ps es)  
     (let ((v (interp-e e r ds)))  
       (interp-match v ps es r ds))]))
```


Interpretation

Match expressions

```
;; Value [Listof Pat] [Listof Expr] Env Defns
;; -> Value { raises 'err }
(define (interp-match v ps es r ds)
  (match* (ps es)
    [( '() '()) (raise 'err)]
    [((cons p ps) (cons e es))
     (match (interp-match-pat p v r)
       [#f (interp-match v ps es r ds)]
       [r (interp-e e r ds)])])])
```

Interpretation

Patterns

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

The heart of pattern matching in the interpreter.

Interpretation

Patterns: Examples

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

```
> (interp-match-pat (Var '_) 1 '())
'()
```

```
> (interp-match-pat (Var 'x) 1 '())
'((x 1))
```

```
> (interp-match-pat (Lit 1) 1 '())
'()
```

```
> (interp-match-pat (Lit 2) 1 '())
#f
```

Interpretation

Patterns: Examples

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

```
> (interp-match-pat (Box (Var '_))
  (box 1)
  '())
```

```
'()
```

```
> (interp-match-pat (Box (Var 'x))
  (box 1)
  '())
```

```
'((x 1))
```

```
> (interp-match-pat (Box (Var 'x))
  1
  '())
```

```
#f
```

Interpretation

Patterns: Examples

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

```
> (interp-match-pat (Cons (Var '_) (Var '_))
  (cons 1 2)
  '())
```

```
'()
```

```
> (interp-match-pat (Cons (Var 'x) (Var 'y))
  (cons 1 2)
  '())
```

```
'((y 2) (x 1))
```

```
> (interp-match-pat (Cons (Var 'x) (Var 'y))
  1
  '())
```

```
#f
```


Interpretation

Patterns: Examples

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

```
> (interp-match-pat (Conj (Lit 1) (Lit 1)))
1
'()
```

```
'()
```

```
> (interp-match-pat (Conj (Lit 1) (Lit 2)))
1
'()
```

```
#f
```

```
> (interp-match-pat (Conj (Lit 1) (Var 'x)))
1
'()
```

```
'((x 1))
```

Interpretation

Patterns: Examples

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

```
> (interp-match-pat (Conj (Var 'y) (Var 'x)))
1
' (())
```

```
' ((x 1) (y 1))
> (interp-match-pat (Conj (Var 'p) (Cons (Var 'x) (Var 'y))))
(cons 1 2)
' (())
' ((y 2) (x 1) (p (1 . 2)))
```

Interpretation

Patterns

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


Interpretation

Patterns

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

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Lit l) (and (eqv? l v) r)]))
```


Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Box p)
     (match v
       [(box v)
        (interp-match-pat p v r)]
       [_ #f])]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Box p)
     (match v
       [(box v)
        (interp-match-pat p v r)]
       [_ #f])]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Cons p1 p2)
     (match v
       [(cons v1 v2)
        (match (interp-match-pat p1 v1 r)
          [#f #f]
          [r1 (interp-match-pat p2 v2 r1)])])
     [_ #f])]))
```


Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Conj p1 p2)
     (match (interp-match-pat p1 v r)
       [#f #f]
       [r1 (interp-match-pat p2 v r1)]))]))
```

Interpretation Patterns

Putting it all together...

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    [(Var '_) r]
    [(Var x) (ext r x v)]
    [(Lit l) (and (eqv? l v) r)]
    [(Box p)
     (match v
       [(box v)
        (interp-match-pat p v r)]
       [_ #f])]
    [(Cons p1 p2)
     (match v
       [(cons v1 v2)
        (match (interp-match-pat p1 v1 r)
          [#f #f]
          [r1 (interp-match-pat p2 v2 r1)])]
       [_ #f])]
    [(Conj p1 p2)
     (match (interp-match-pat p1 v r)
       [#f #f]
       [r1 (interp-match-pat p2 v r1)])]))
```

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
            (Jump 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))]]))]
    [..])
  #;...))
```

For next time

- Read Knock notes
- Take ELMS quiz

CMSC 430 - 25 June 2026

Compiling match expressions

- Knock - Pattern matching
 - Compiling patterns

CMSC 430 - 25 June 2026

Announcements

- Assignment 7 due tonight
- Exam 2 tomorrow
- No quiz today

Knock

Compiling a single pattern

The heart of pattern matching compilation

```
;; Pat CEnv Symbol -> (list Asm CEnv)  
(define (compile-pattern p cm next)
```

Two things:

- instructions that bind variables in p if it matches, or pops environment and jumps to next if it doesn't
- compile time environment that describes what p binds

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
            (Jump 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))]]))]
    [;...))
```

For next time

- Read Knock notes
- Take ELMS quiz