CMSC 430 - 1 Sept 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 Tu/Th (except holidays, exams)

Assignments: ~10

Exams: 3 - 10/1, 11/5, 12/17

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%
Quizes & surveys	20%
Exams	50%

Key resources

Course logistics

- Class web page (syllabus, assignments, course notes, slides)
  `https://www.cs.umd.edu/class/fall2026/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

- Two in-class exams, one final
- Designed to be easily done in time allotted*
- Open notes, closed communication
- You will run out of time if you haven’t already mastered material

* If you have already mastered the material!

Day to day: office hours

How to CMSC 430

- Use online forums for daily help (Piazza)
- TA office hours scheduled will be announced soon
- To schedule office hours with me, send email

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 rules that define how a computer can be controlled.

Human readable way to control a computer so you don't have to use machine code.

A set of instructions to accomplish a certain goal.

Rules that are both syntactically and semantically defined.

- Syntax: what characters are valid
- Semantics: order or how you're able to use syntax

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

Reduces abstraction by introducing complexity

It is a program

It has to preserve the meaning of the input program

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`
- `(::) : α → α list → α list`
- `[.;.;.;...]` convient notation for lists

OCaml pairs (and tuples):

- `(,) : α → β → α * β`

Pairs and lists: **fundamentally different things**

Lists and pairs in Racket vs OCaml

Constructors in Racket

Racket lists:

- `'()  : α list`
- `cons : α → α list → α list`
- `list` convenient function for lists

Racket pairs (and tuples):

- `cons : α → β → α * β`

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

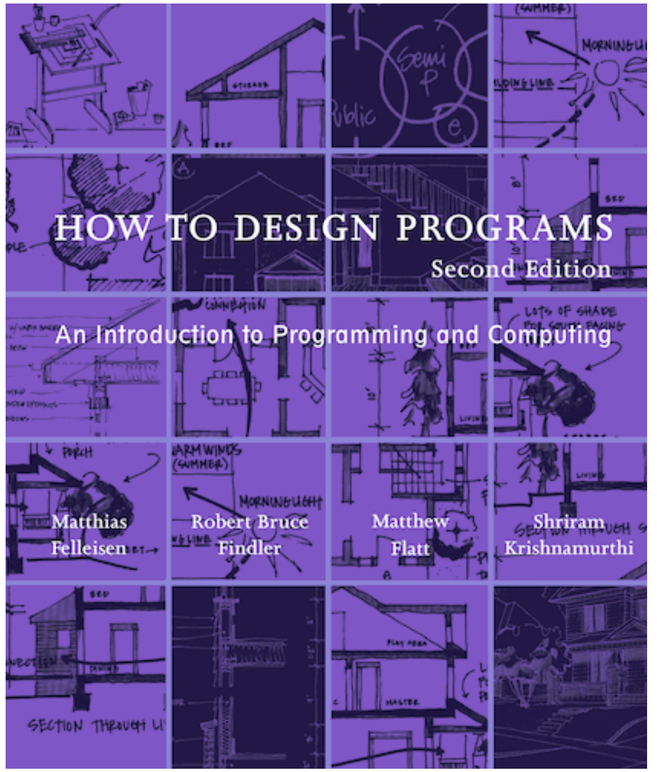
`(struct coord (x y z))` defines:

- `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 - 3 Sept 2026

Our first compiler

Quick Racket review

Intro to a86

CMSC 430 - 3 Sept 2026

Announcements

- Practice assignments 1 & 2: Racket, a86
- Assignments 1-4 released
- Video solution for practice assignment 1 & 2 (ELMS > Panopto)
- 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**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

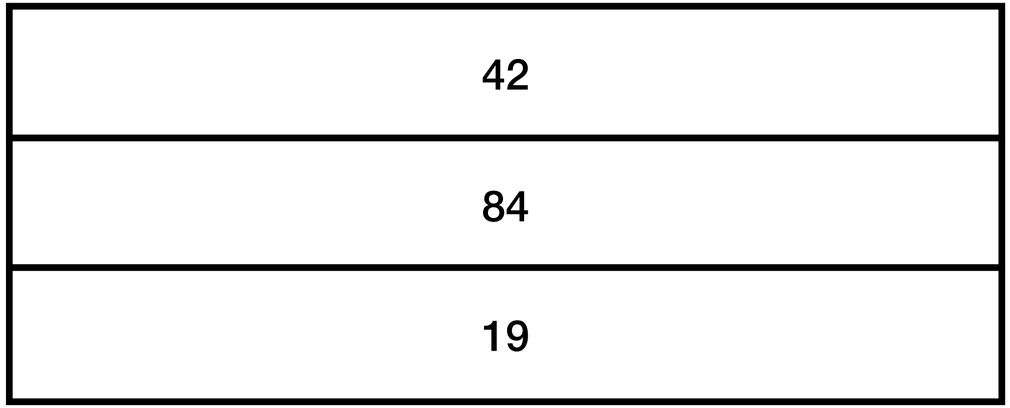
Call, Ret revisited

- 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
  Jmp name
- Ret =
  Add rsp 8
  Jmp (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:

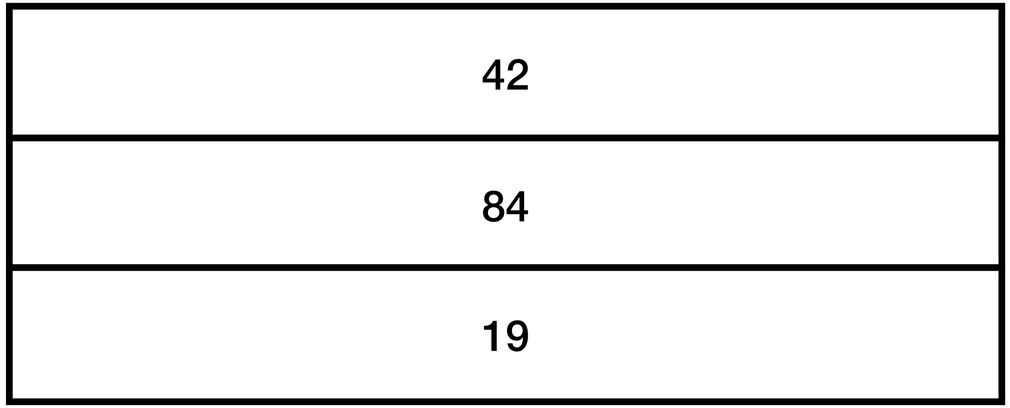


rax:

Examples with the stack

- Push 42 ⬅
- Push 84 ⬅
- Push 19 ⬅
- Add ‘rsp 8 ⬅
- Add ‘rsp 8 ⬅
- Add ‘rsp 8 ⬅

Stack:

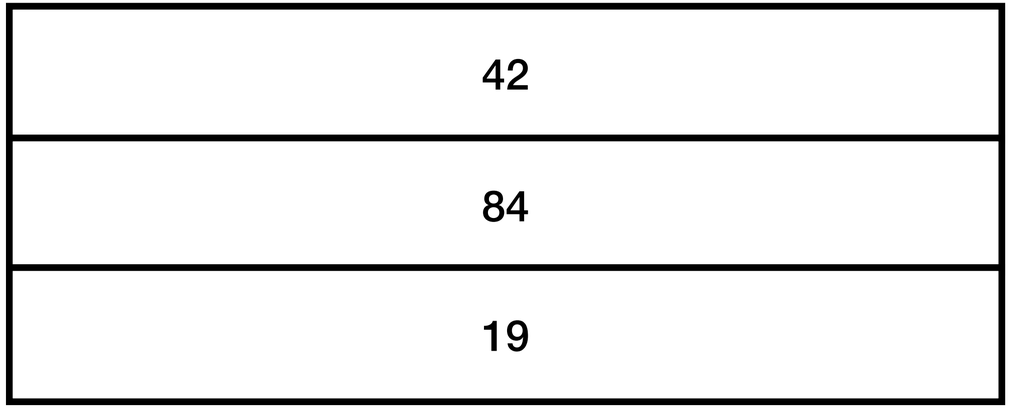


rax: **?**

Examples with the stack

- Push 42 ⬅
- Push 84 ⬅
- Push 19 ⬅
- Add ‘rsp 24 ⬅

Stack:

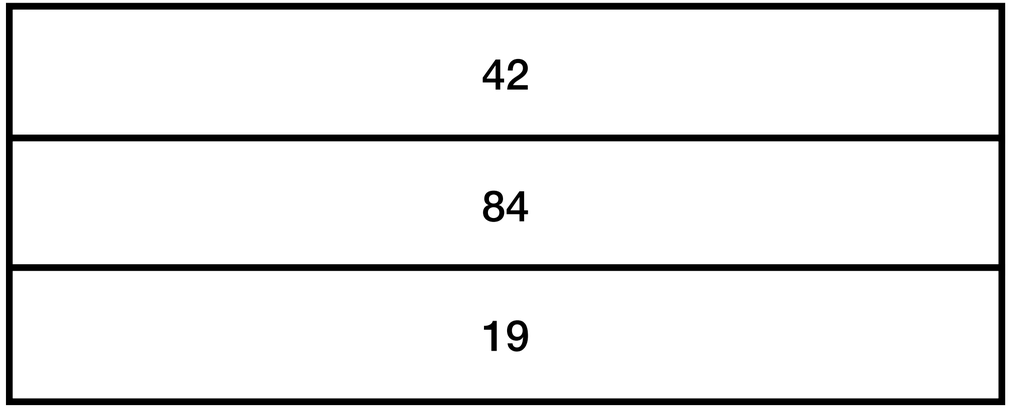


rax: **?**

Examples with the stack (memory view)

- Push 42 ⬅
- Push 84 ⬅
- Push 19 ⬅
- Add ‘rsp 8 ⬅
- Add ‘rsp 8 ⬅
- Add ‘rsp 8 ⬅

Stack:

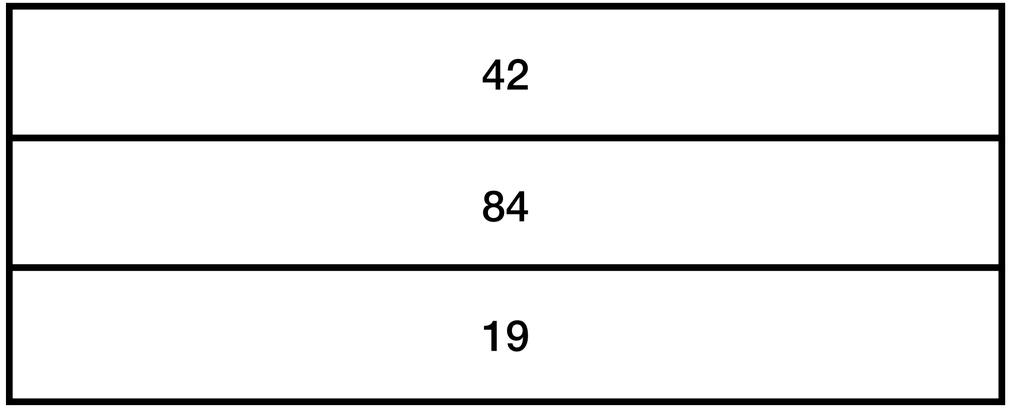


rax: **?**

Examples with the stack (memory view)

- Push 42 ⬅
- Push 84 ⬅
- Push 19 ⬅
- Add ‘rsp 24 ⬅

Stack:



rax: **?**

For next time

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

CMSC 430 - 8 Sept 2026

Our first compilers

- Quick Racket and a86 review
- Abscond - integer literals
- Blackmail - unary primitives
- Con - conditional expressions
- Dupe - booleans, revised conditional

CMSC 430 - 8 Sept 2026

Announcements

- Assignment 1 due Thurs, midnight
- New quiz released today, due by start of class Thursday
- TA Office Hours published on course web page

Abscond

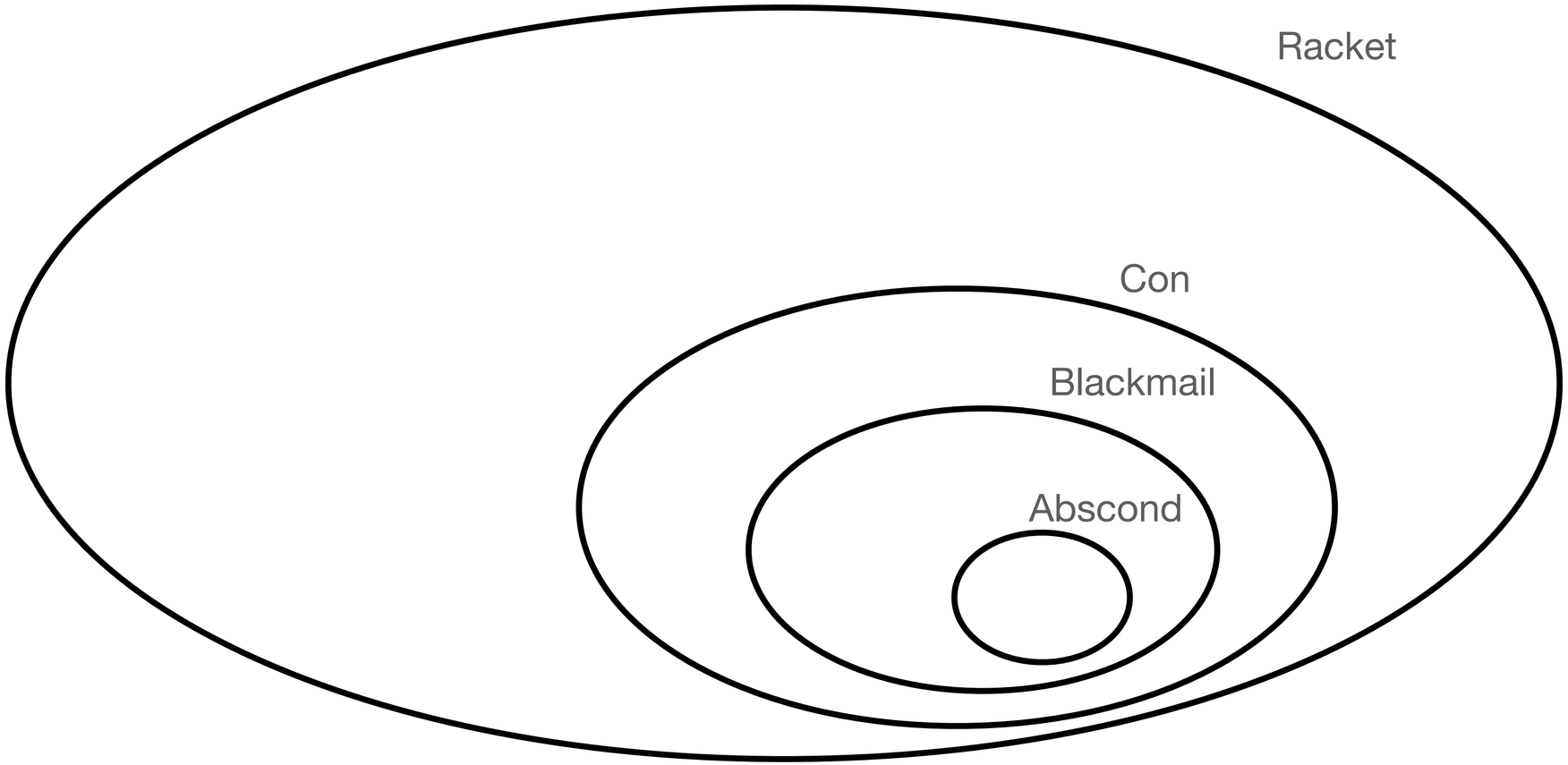
Learning objectives

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

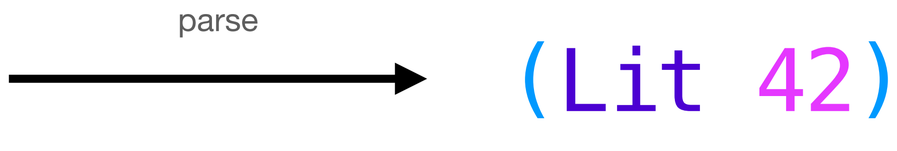
```
(Lit 42)
```

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

Example Abscond program

Parser constructs AST from string input

```
#lang racket
42
```



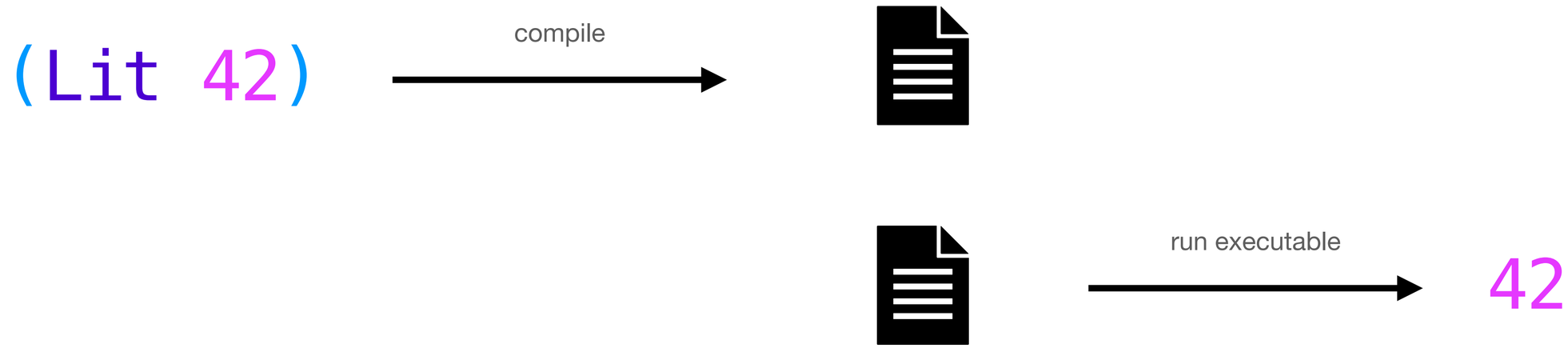

```
(Lit 42)
```

Example Abscond program

Interpreter computes meaning from AST

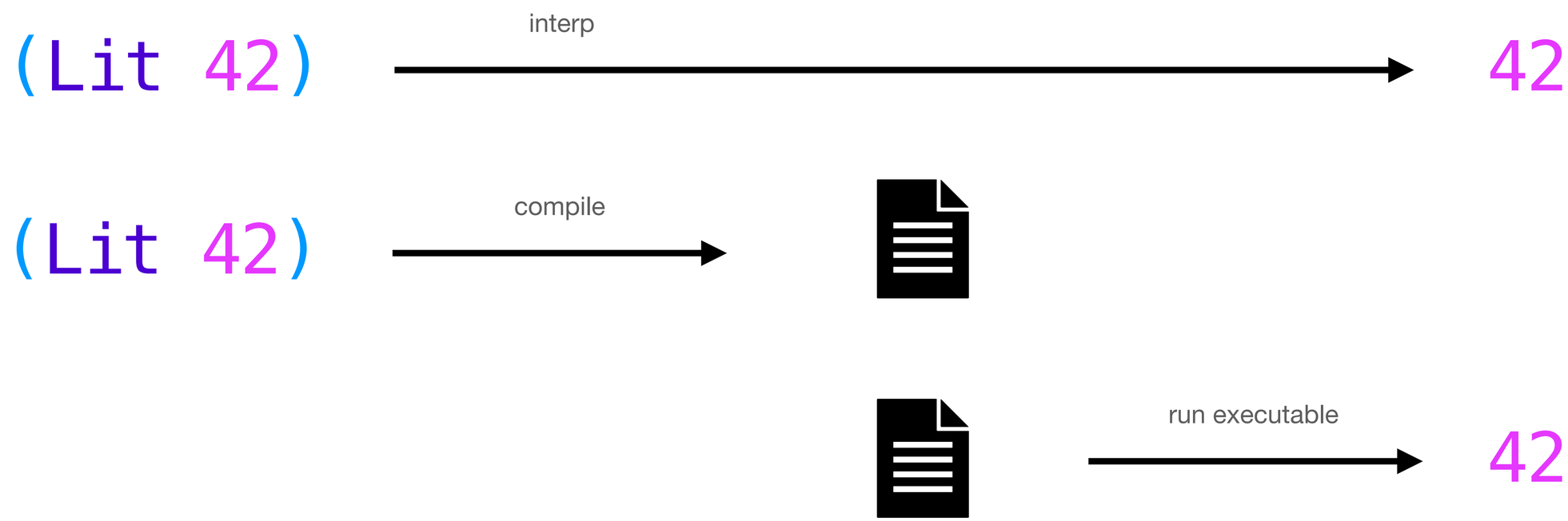
Example Abscond program

Compiler computes a program that computes meaning



Example Abscond program

Compiler specification



Example Abscond program

Our approach to parsing: reader + parser

```
#lang racket
42
```


```
42
```


```
(Lit 42)
```

String

Semi-structured data;
ensures parens balanced, etc.

S-Expr

Expr

Structured data; ensures
grammatically well-formed

Example Abscond program

Our approach to compiling: compiler + stand-alone runtime

```
(Lit 42)
```

compile →

```
(Global 'entry)
(Label 'entry)
(Mov rax 42)
(Ret)
```

asm-display →

```
        .intel_syntax noprefix
        .text
        .global "_entry"
"_entry":
        mov rax, 42
        ret
```

Expr

Construct AST of assembly program

Asm

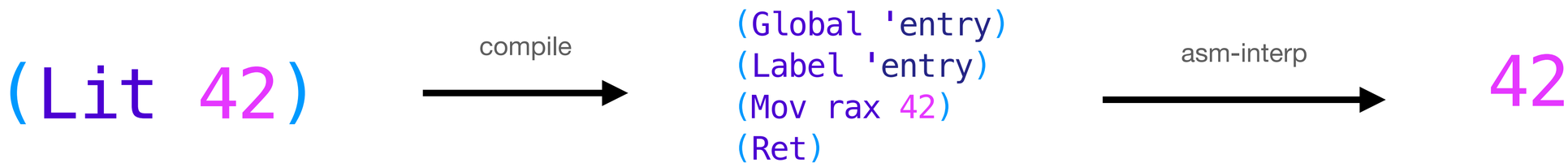
.s

Print to standard assembly format; call assembler

Link w/ stand-alone runtime system

Example Abscond program

Our approach to testing: compiler + asm-interp



Expr

Asm

Value

Construct AST of assembly program

Run compiled program to compute value

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

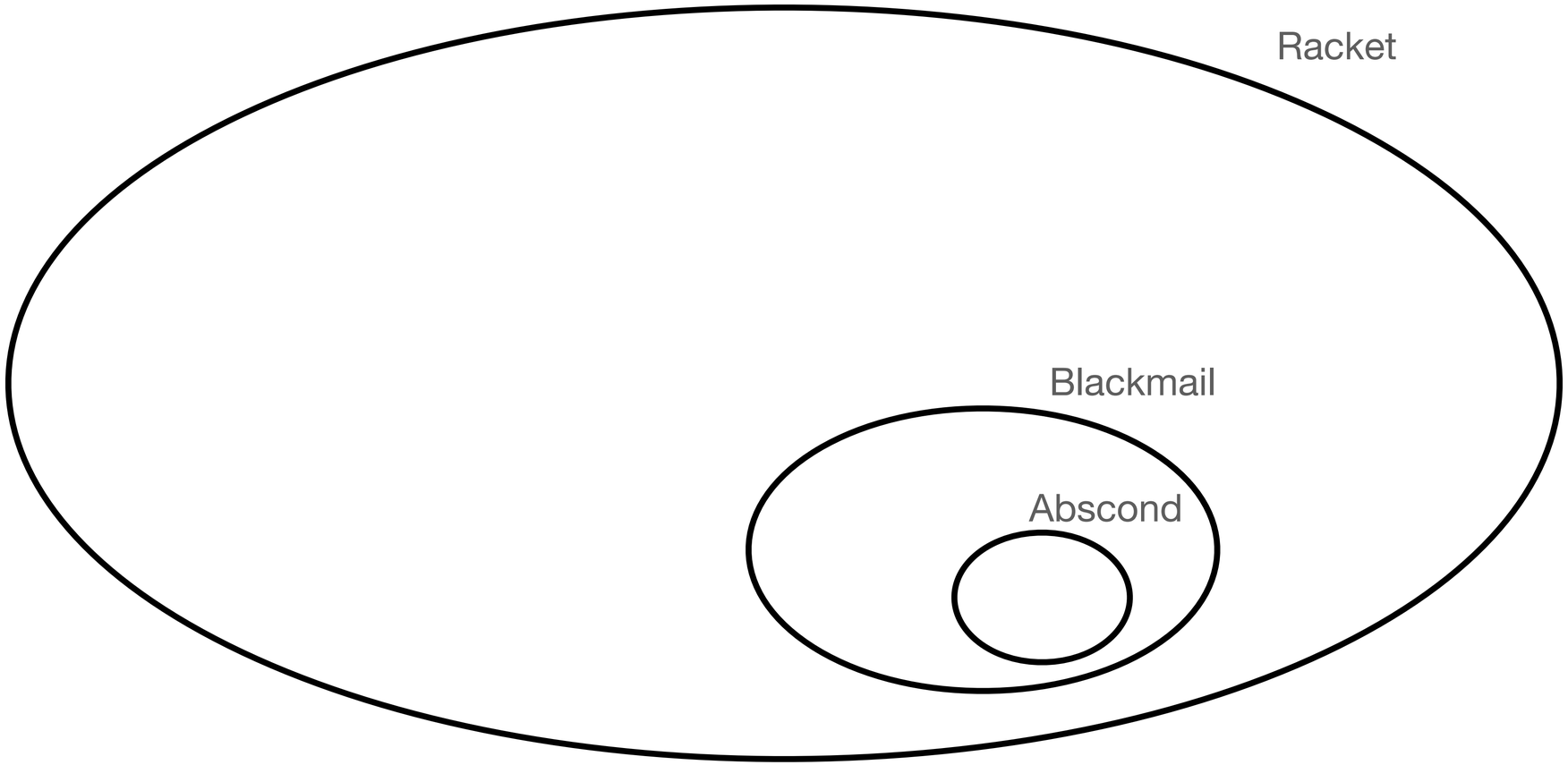
Learning objectives

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 Blackmail program

Interpreter computes meaning from AST

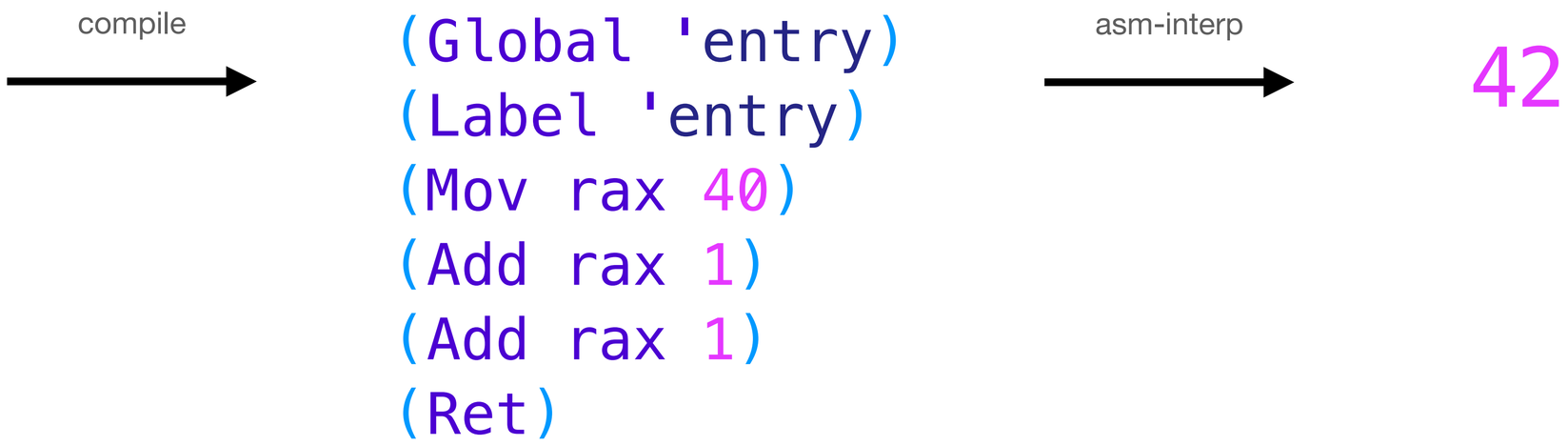
```
(Prim1 'add1 (Prim1 'add1 (Lit 40)))
```


```
42
```

Example Blackmail program

Interpreter computes meaning from AST

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



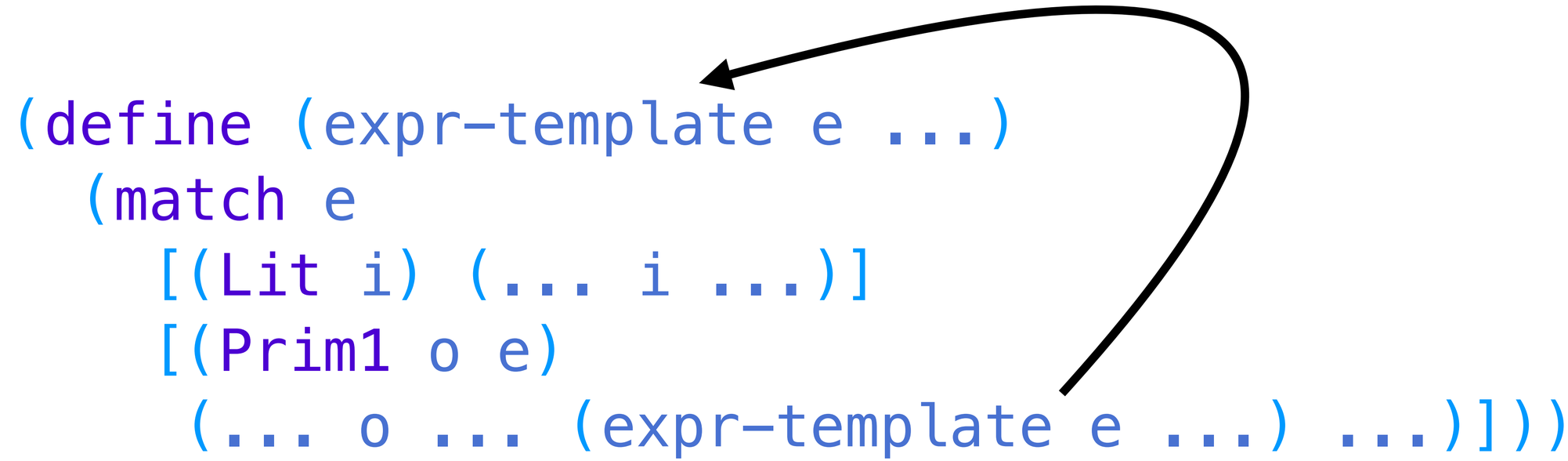
Expr is an inductive data type

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

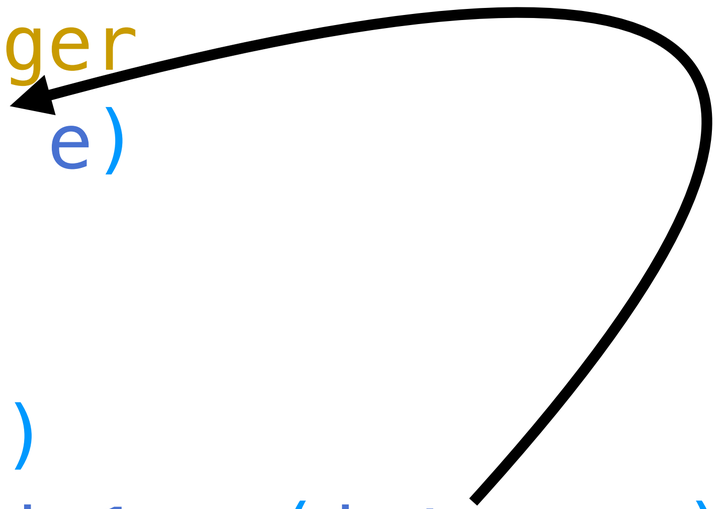
Abstract syntax



```
(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

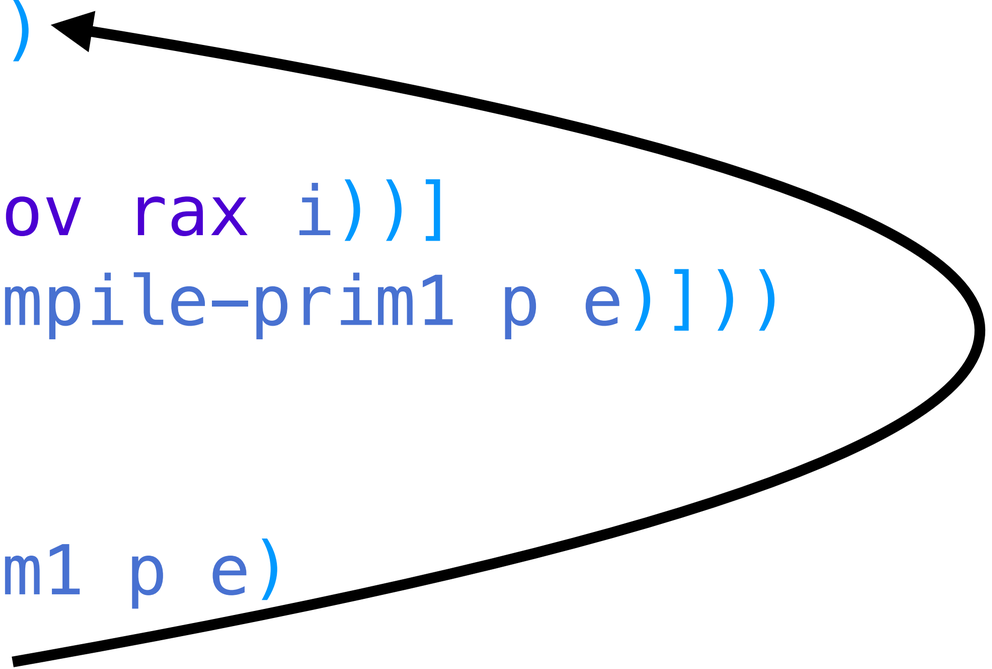
Key invariant

```
;; 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
```

For next time

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

CMSC 430 - 10 Sept 2026

Our first compilers

- Quick Blackmail review
- Con - conditional expressions
- Dupe - booleans, revised conditional

CMSC 430 - 10 Sept 2026

Announcements

- Assignment 1 due Thurs, midnight
- Assignment 2 due next Thurs, midnight
- New quiz released today, due by start of class Tuesday
- TA Office Hours published on course web page

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))
```

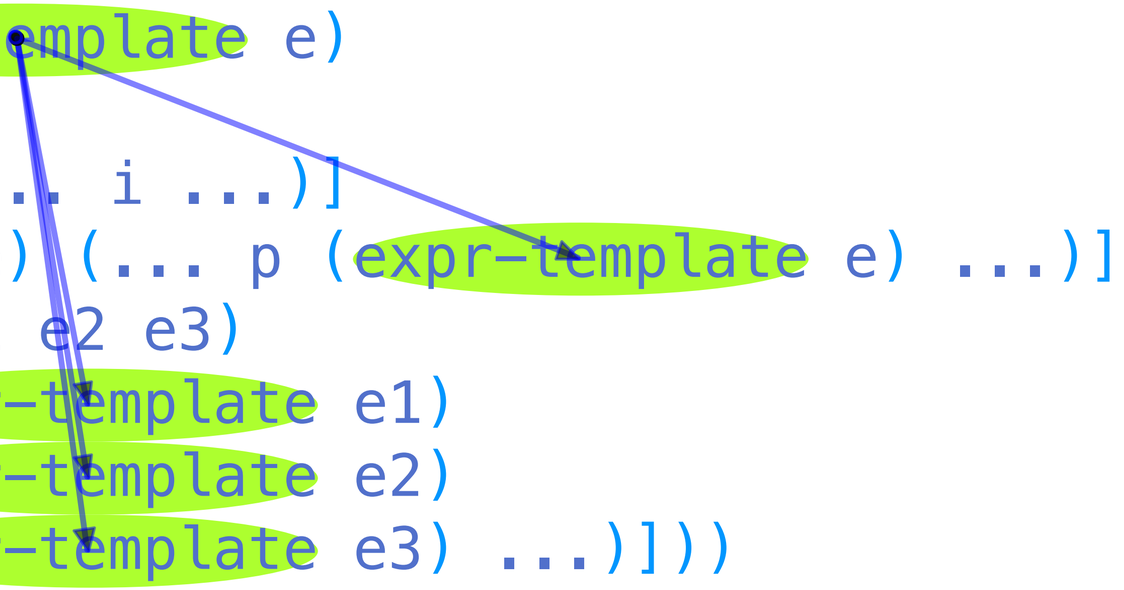
Abstract examples:

```
(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

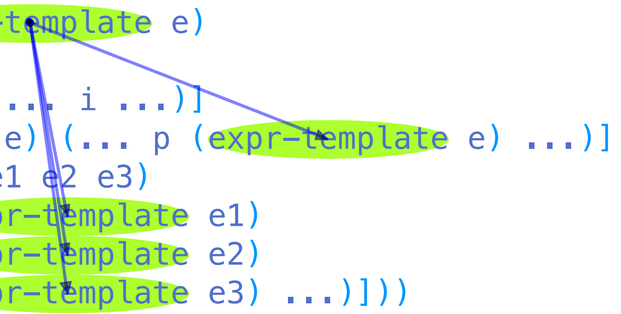
```
;; type Expr = (Lit Integer)
;;           | (Prim1 Op1 Expr)
;;           | (IfZero Expr Expr Expr)

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



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

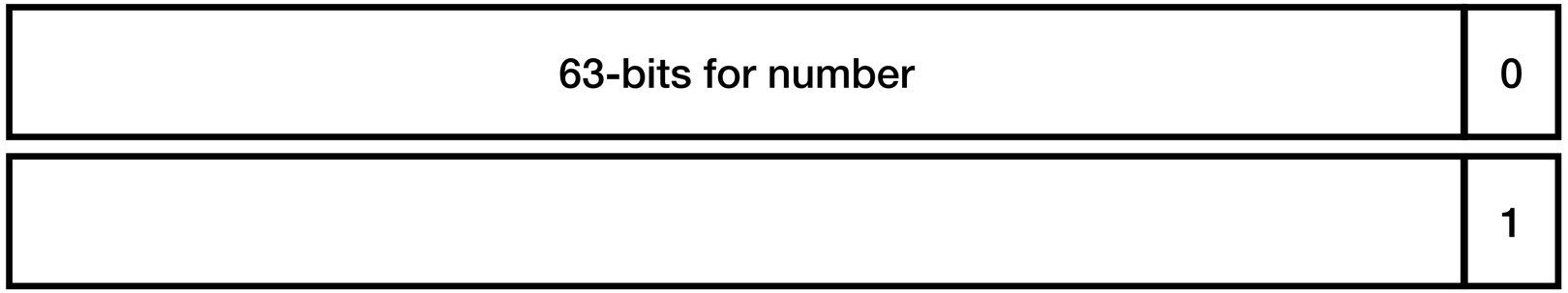
- some will represent integer values
- some will represent boolean values

Criteria: the contents of rax must uniquely determine the value

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

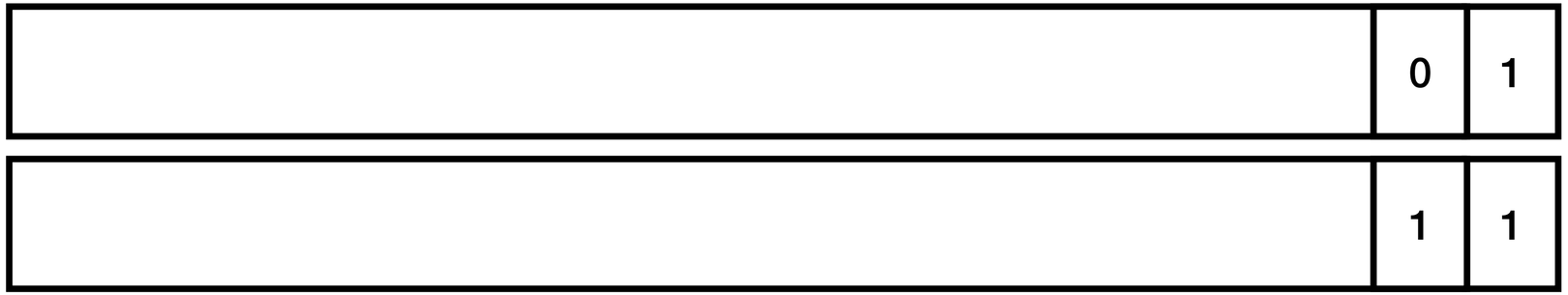
Encoding values in Dupe

Type tag in least significant bits



Integers

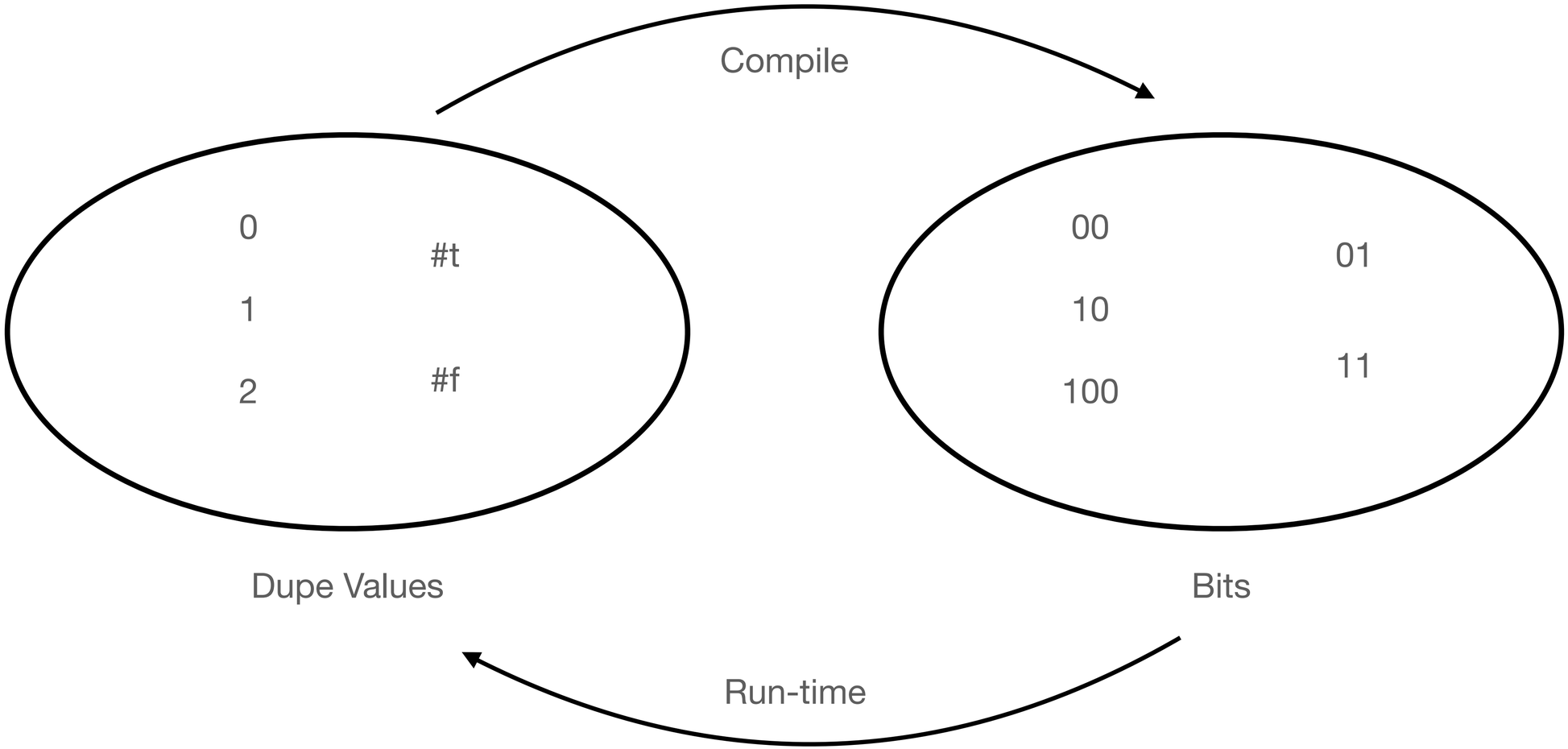
Booleans



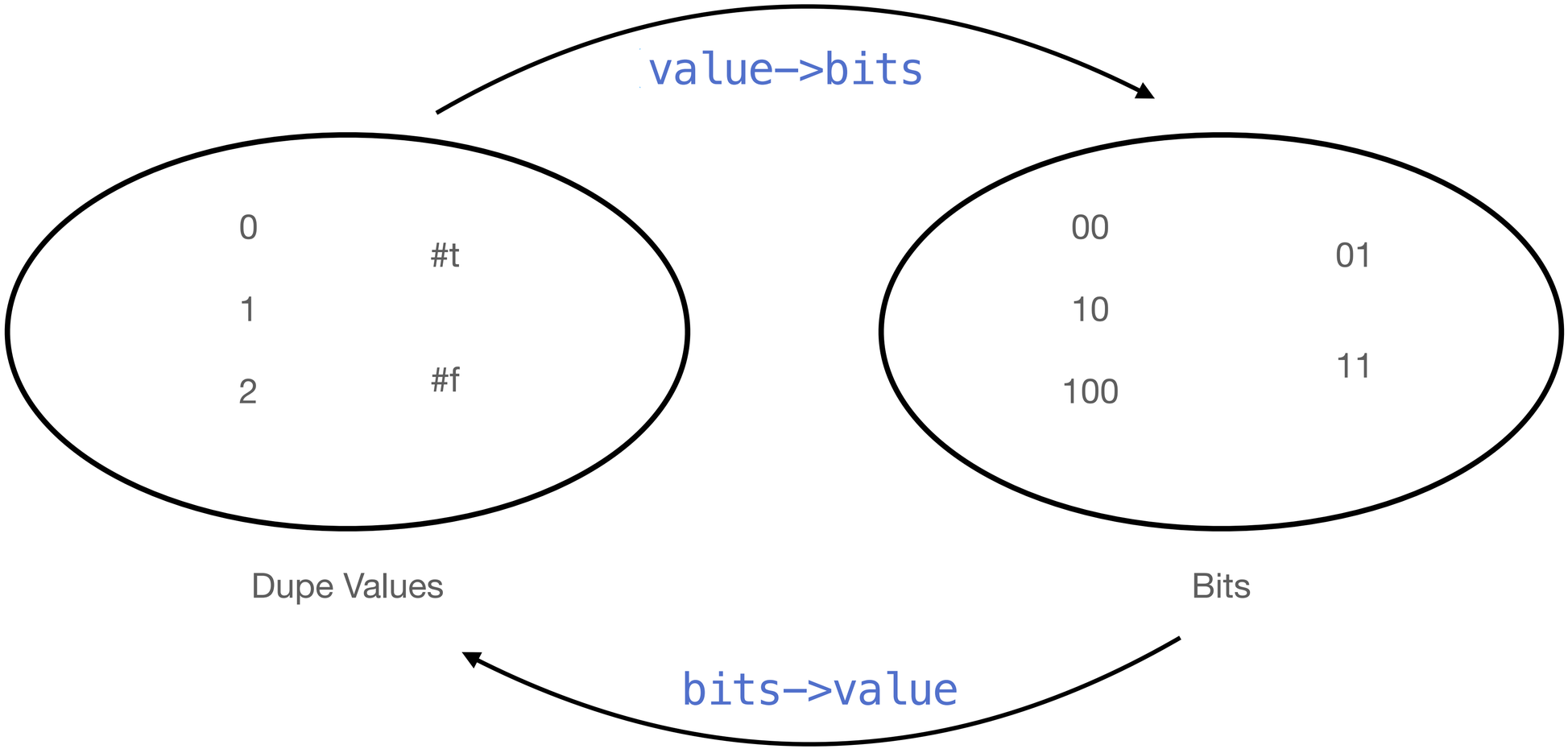
`#t`

`#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)) ;=> 1 #b01
(compile-e (Lit #f)) ;=> 3 #b11
(compile-e (Lit 0))  ;=> 0 #b00
(compile-e (Lit 1))  ;=> 2 #b10

(compile-e (If e1 e2 e3))
  ;=> (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

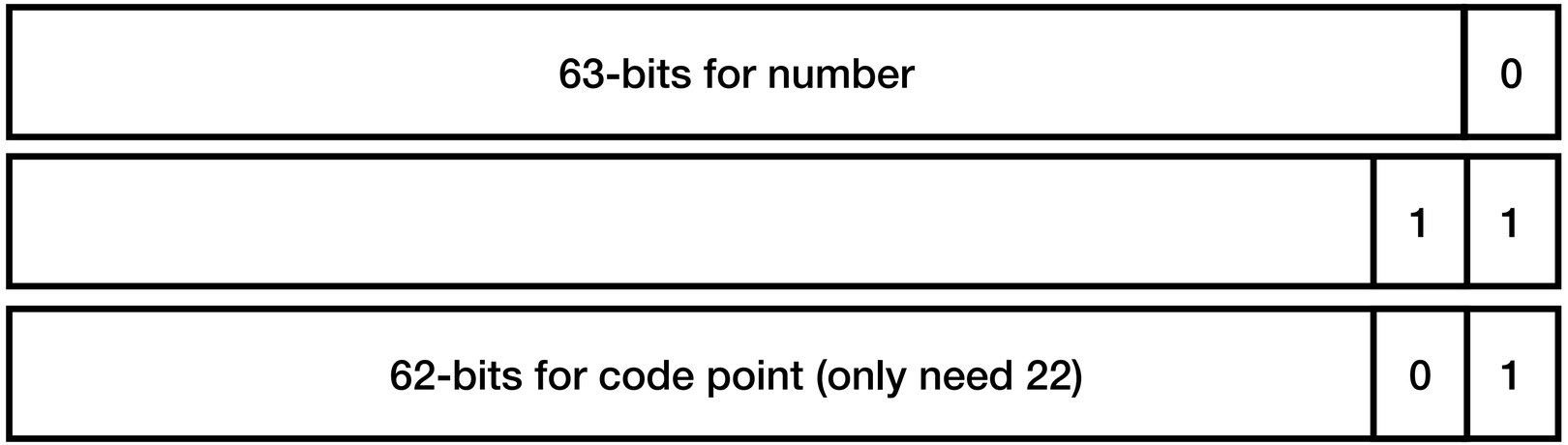
`#\a #\b #\c ... #\λ #\絳 ...`

Unicode: roughly 150K characters.

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

Encoding values in Dodger

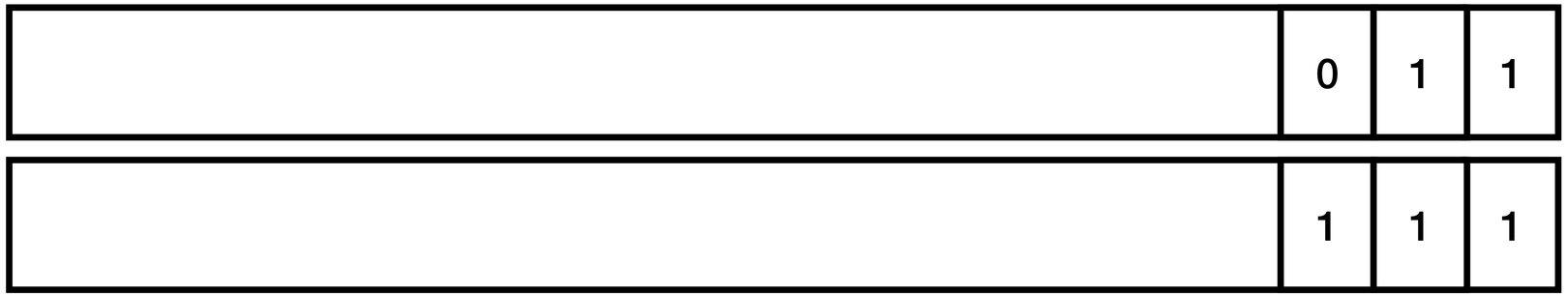
Type tag in least significant bits



Integers

Booleans

Characters



`#t`

`#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
```

For next time

- Read Evildoer notes
- Study Con, Dupe notes
- Take ELMS quiz