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**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

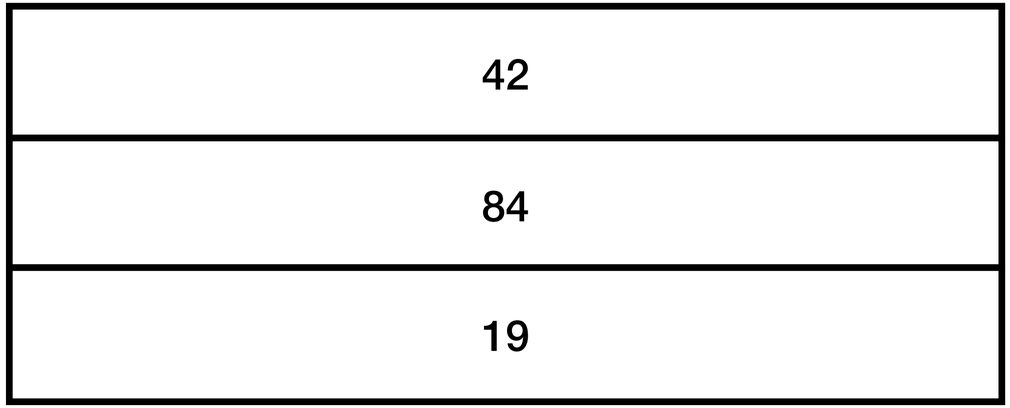
Call, Ret revisited

- 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:

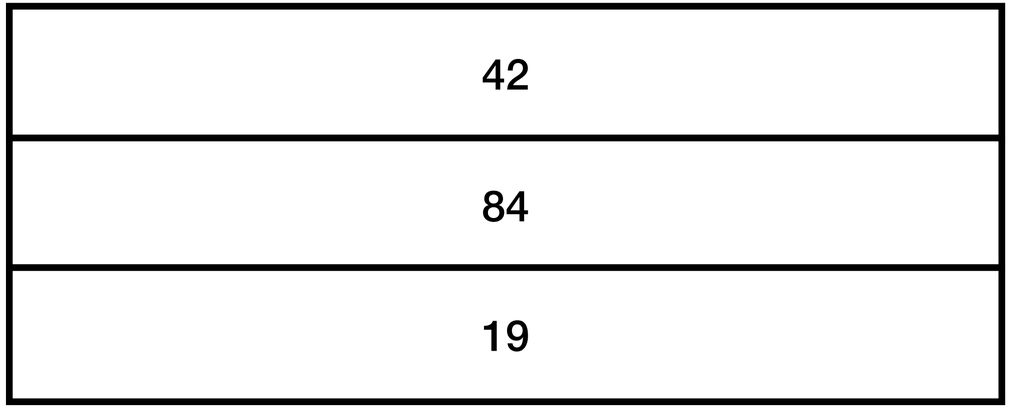


rax:

Examples with the stack

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

Stack:

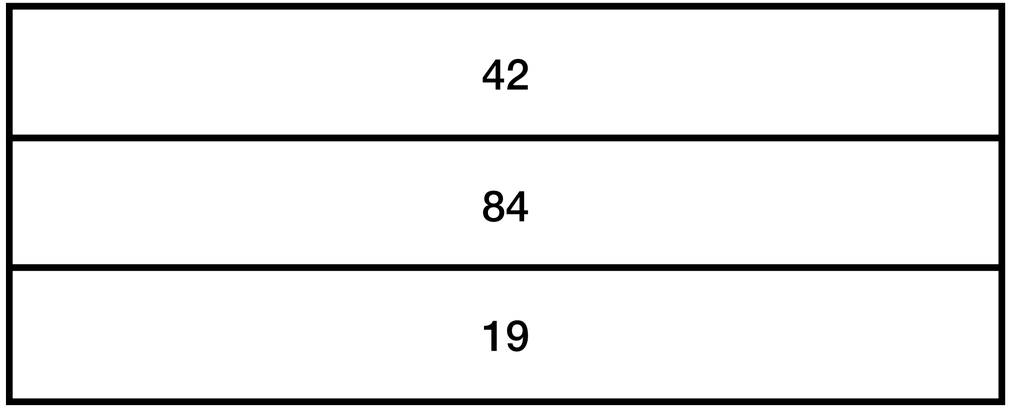


rax: **?**

Examples with the stack

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

Stack:

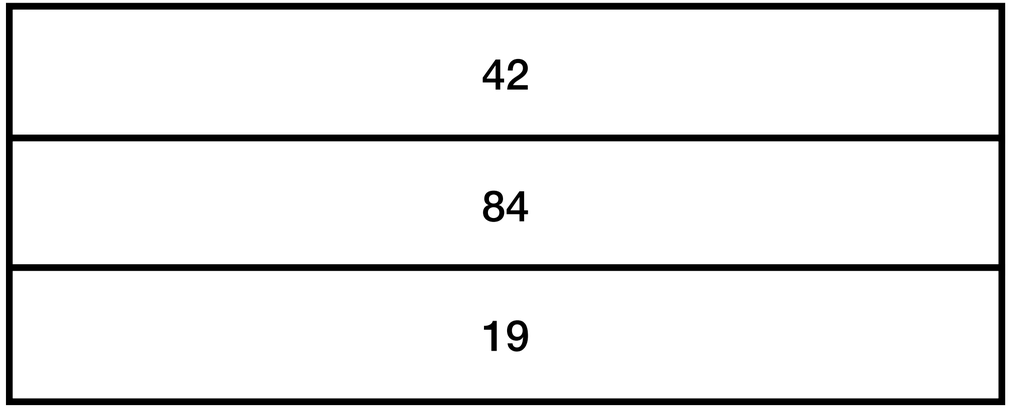


rax: **?**

Examples with the stack (memory view)

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

Stack:

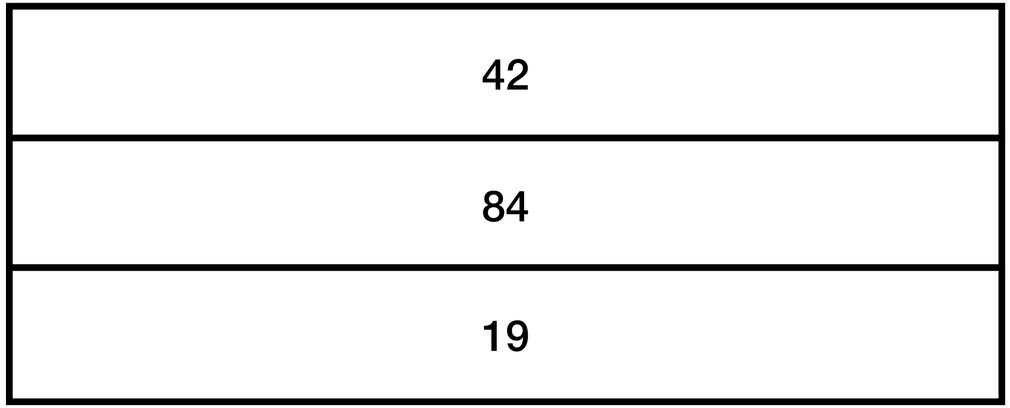


rax: **?**

Examples with the stack (memory view)

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

Stack:



rax: **?**

Abscond

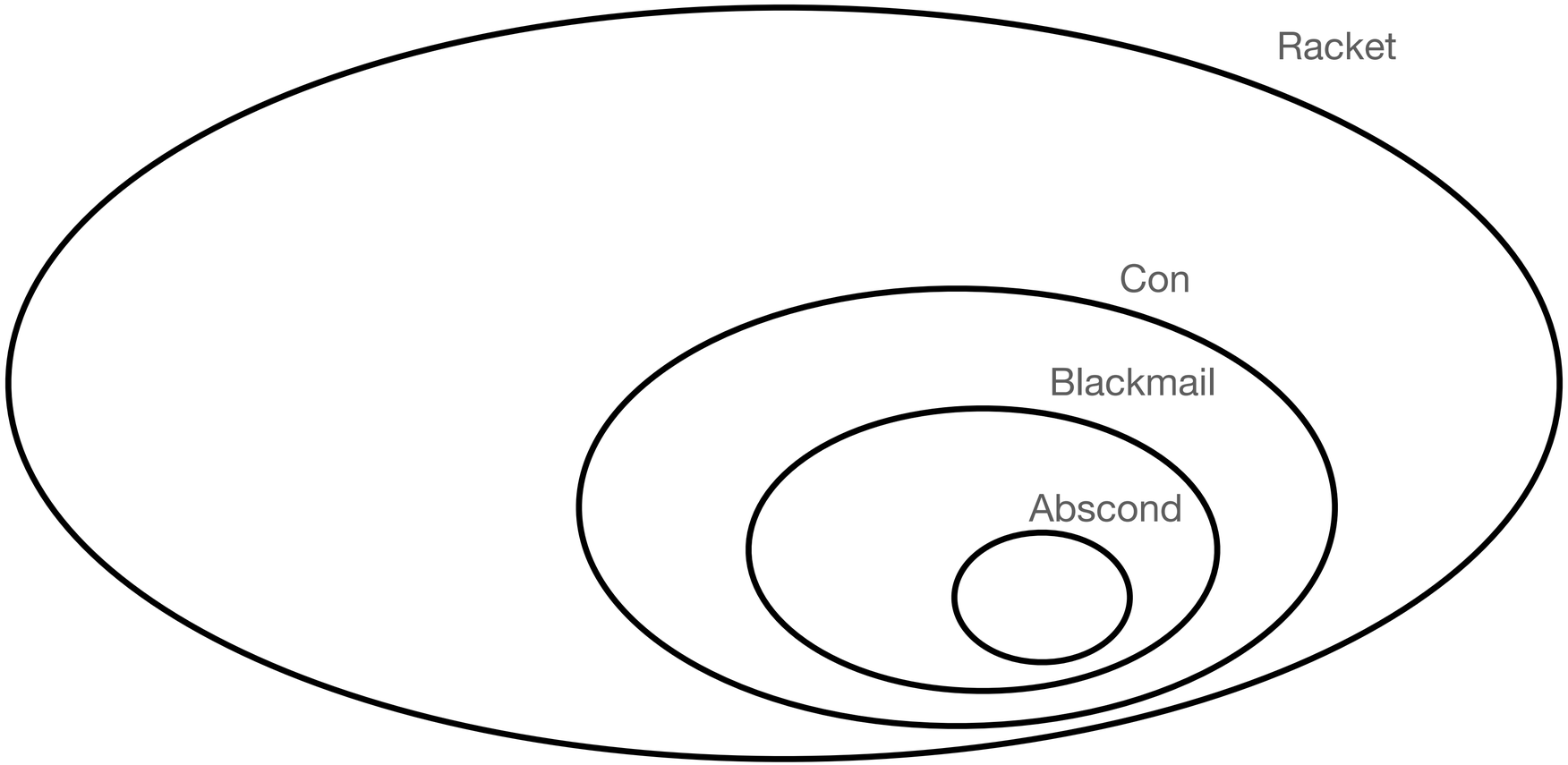
Learning objectives

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

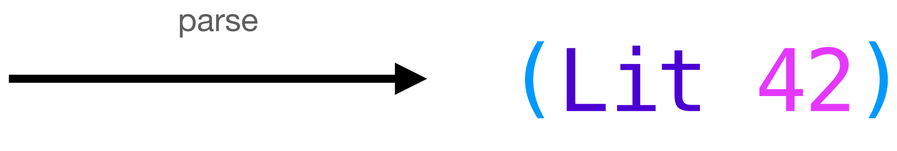
```
(Lit 42)
```

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

Example Abscond program

Parser constructs AST from string input

```
#lang racket
42
```



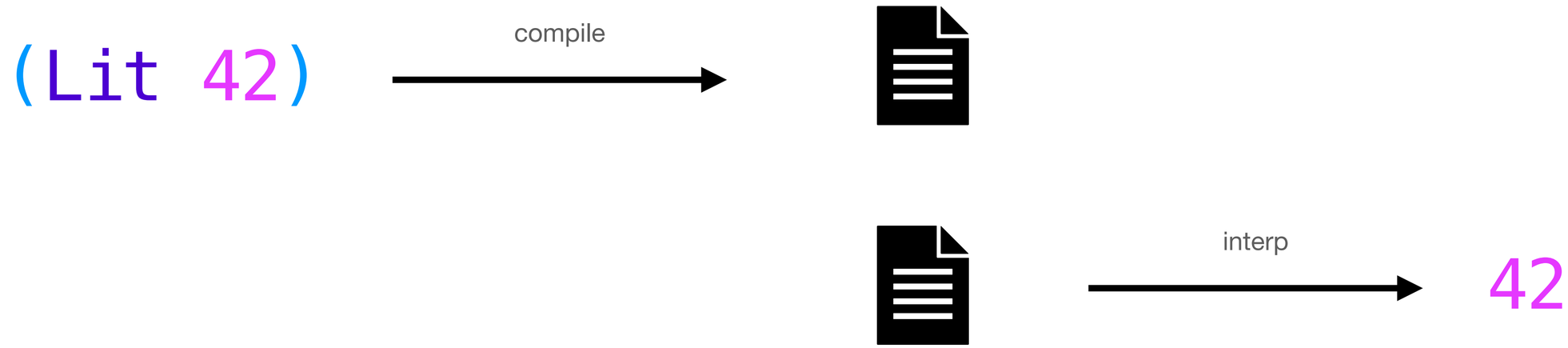

```
(Lit 42)
```

Example Abscond program

Interpreter computes meaning from AST

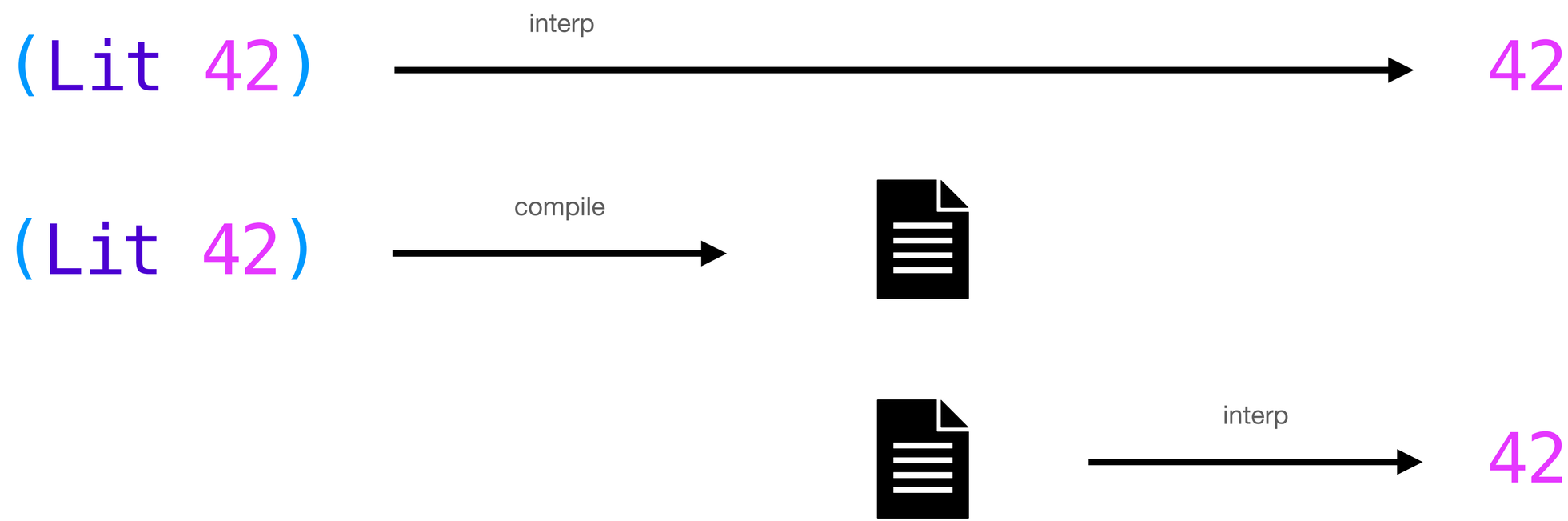
Example Abscond program

Compiler computes a program that computes meaning



Example Abscond program

Compiler specification



For next time

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