

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

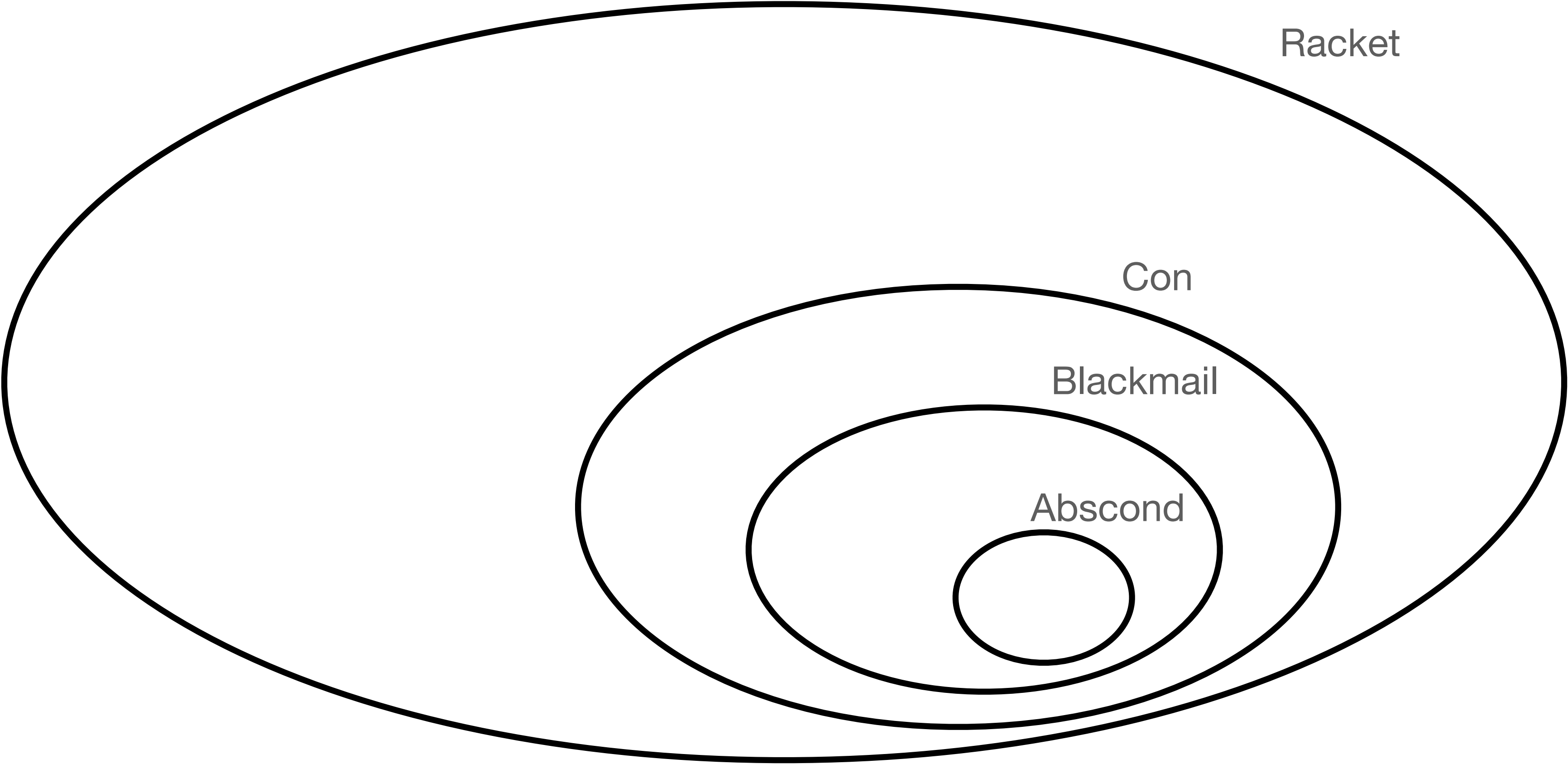
Learning objectives

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

1		#lang racket
2		42

1 #lang racket
Example Abscond program
Abstract syntax (require abscond)

3

4

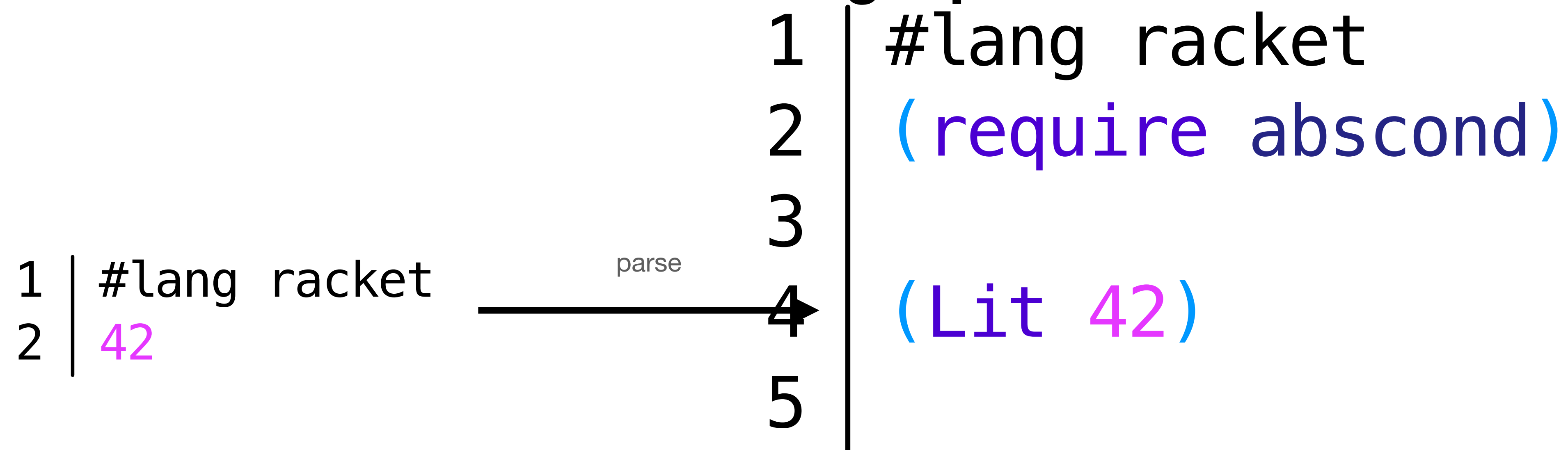
(Lit 42)

5

```
1 | #lang racket
2 | (require abscond)
3 |
4 | (Lit 42)
5 |
6 | ;; type Expr =
7 | ;; | (Lit Integer)
8 | (struct Lit (n))
```

Example Abscond program

Parser constructs *AST* from string input

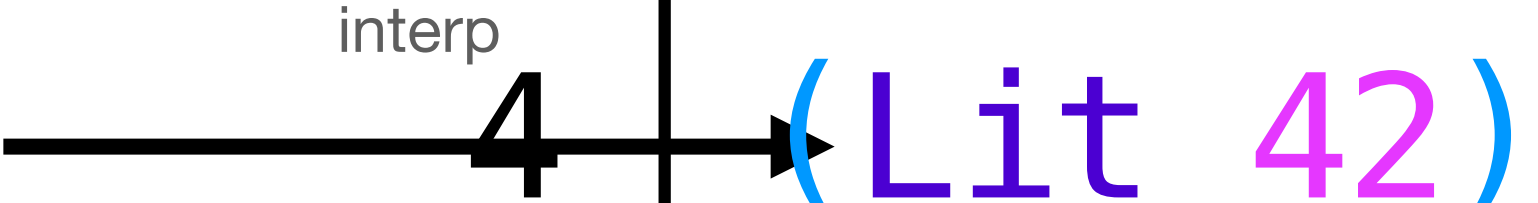


Example Abscond program

Interpreter computes meaning from AST

1	#lang racket	1	#lang racket
2	(require abscond)	2	(require abscond)
3		3	
4	(Lit 42)	4	(Lit 42)
5		5	

interp



Example Abscond program

Compiler computes a program that computes meaning

```
1 | #lang racket
2 | (require abscond)
3 |
4 | (Lit 42)
5 |
```

compile



```
1 | #lang racket
2 | (require abscond)
3 |
4 | (Lit 42)
5 |
```

interp



Example Abscond program

Compiler specification

```
1 #lang racket
2 (require abscond)
3
4 (let ([qu42e abscond])
5   (Lit 42))
```

interp

compile



```
1 #lang racket
2 (require abscond)
3
4 (Lit 42)
```

```
1 #lang racket
2 (require abscond)
3
4 (Lit 42)
```

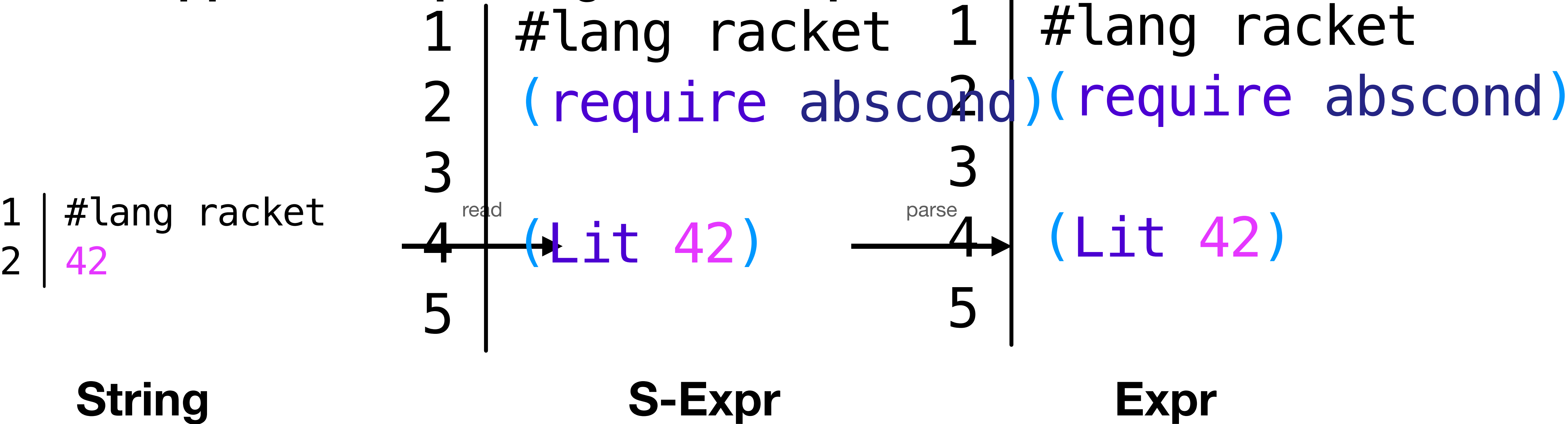
run executable

Compile time

Run time

Example Abscond program

Our approach to parsing: reader + parser



Semi-structured data;
ensures parens balanced, etc.

Structured data; ensures
grammatically well-formed

Example Abscond program

Our approach to compiling: compiler + stand-alone runtime

```
#lang racket
```

```
(require abscond)
```

```
(Lit 42)
```

Expr

Construct AST of assembly program

compile



```
1 #lang racket
2 (provide parse)
3 (require "ast.rkt")
4
5 ;; S-Expr -> Expr
6 (define (parse s)
7   (match s
8     [(? exact-integer?) (Lit s)]
9     [_ (error "parse error" s)]))
10
11 (require a86)
12
13 (Global 'entry)
14 (Label 'entry)
15 (Mov rax 42)
16 (Ret)
17
18
```

Asm

asm-display



```
Welcome to DrRacket, version 9.2 [cs].
Language: racket, with debugging; memory limit: 4096 MB.
.intel_syntax noprefix
.text
.global "_entry"
"_entry":
    mov rax, 42
    ret
>
```

.S

Print to standard assembly format; call assembler

Link w/ stand-alone runtime system

Example Abscond program

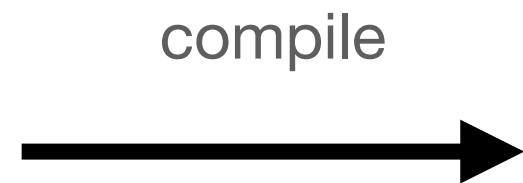
Our approach to testing: **compiler + asm-interp**

```
#lang racket
(require abscond)

(Lit 42)
```

Expr

Construct AST of assembly
program



```
1 #lang racket
2 (provide parse)
3 (require "ast.rkt")
4 ;; S-Expr -> Expr
5
6 (define (parse s)
7   (match s
8     [(? exact-integer?) (Lit s)]
9     [_ (error "parse error" s)]))
10
11 (require a86)
12
13 (Global 'entry)
14 (Label 'entry)
15 (Mov rax 42)
16 (Ret)
17
18
```

Asm

```
1 #lang racket
2 (require abscond)
3
4 (Lit 42)
5
```

Value

Run compiled program to
compute value

Abscond

At the heart of it

```
1 #lang racket
2 (require a86)
3 (define ... 1)
4
5 ;; Expr -> Asm
6 (define (compile e) ...)
7
8 ;; Asm -> Value
9 (define (run a) ...)
10
11 ;; Expr -> Value
12 (define (interp e) ...)
13
14 ;; compile ◦ run ≡ interp
```

Absconq interpreter

At the heart of it

```
1 #lang racket
2 (provide interp)
3 (require "../syntax/ast.rkt")
4
5 ;; Expr -> Integer
6 (define (interp e)
7   (match e
8     [(Lit i) i]))
9
```


Abscond compiler

At the heart of it

```
2  (provide compile)
3
4  (require "../syntax/ast.rkt")
5  (require a86/ast a86/registers a86)
6
7  ;; Expr -> Asm
8  (define (compile e)
9    (list (Global 'entry)
10         (Label 'entry)
11         (match e
12           [(Lit i) (Mov rax i)]
13           (Ret)))
14
15  ;; Asm -> Integer
16  (define (run a)
```

Abstract correctness

At the heart of it

```
1 // lang racket
2 (provide check-compiler)
3 (require rackunit)
4 (require "interpreter/interp.rkt")
5 (require "compiler/compile.rkt")
6 (require a86/interp)
7
8 ;; Expr -> Void
9 (define (check-compiler e)
10   (check-equal? (interp e)
11                 (asm-interp (compile e))))
12
```

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

```
1  #lang racket
2  (require con)
3  (require a86 rackunit)
4  Idea: write a program. interp : Expr -> Value
5  • simpler than writing compiler
6  • consider it the specification for compiler
7  ;; Expr -> Boolean
8  ;; Is the compiler correct on e?
9  (define (compiler-correct? e)
10    (= (asm-interp (compile e))
        (interp e)))
```

Blackmail

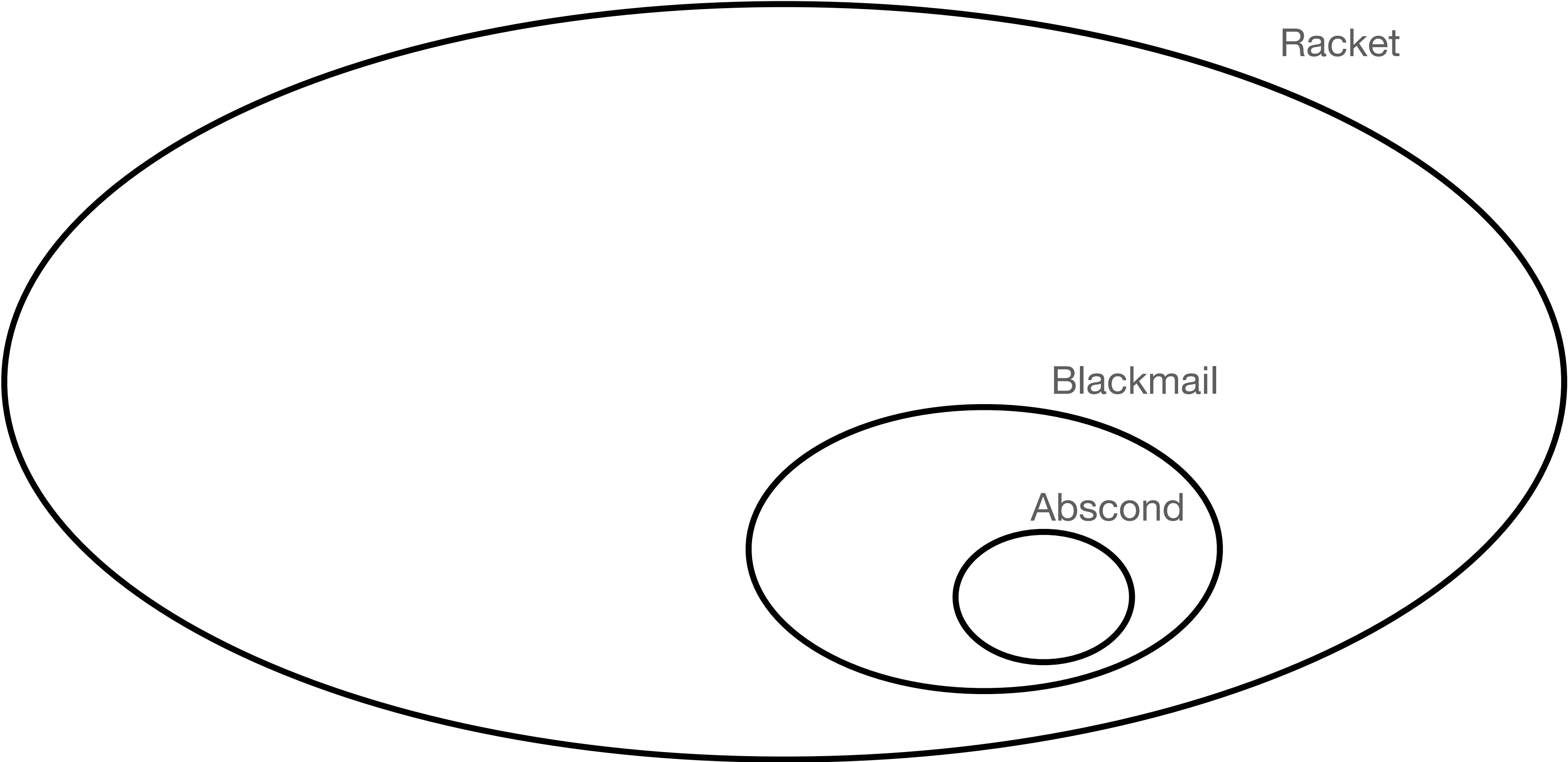
Learning objectives

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

```
1 | #lang racket
2 | (add1 (add1 40))
```

```
1 #lang racket
2 (require blackmail)
3 (abstract-syntax (add1 (add1 40)))
```

4

5

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

7

```
1 #lang racket
2 (provide Lit Prim1)
3
4 ;; type Expr = (Lit Integer)
5 ;;           | (Prim1 Op1 Expr)
6 ;; type Op1 = 'add1 | 'sub1
7 (struct Lit (i))
8 (struct Prim1 (p e))
9
```


Example Abscond program

1 ~~#lang racket~~
2 ~~(require blackmail)~~
3 ~~(parse '(add1 (add1 40)))~~
4
5
6 (Prim1 'add1 (Prim1 'add1 (Lit 40)))
7

Interpreter computes meaning from AST

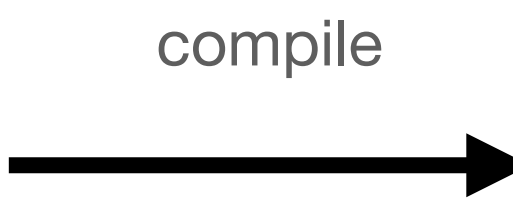
1 #lang racket
2 (require abscond)
3
4 (interp (Lit 42))
5

Example Abstract program

Interpreter computes meaning from AST

```
1  
2  
3  
4  
5  
6  
7
```

```
1 #lang racket  
2  
3 (require blackmail a86)  
4  
5 (define (expr-template e ...)  
6   (match e  
7     [(Lit i) (... i ...)]  
8     [(Prim1 o e) (... (expr-template e ...) ...)]  
9     [(Prim2 o e1 e2) (... (expr-template e1 ...) (expr-template e2 ...) ...)]  
10  
11 (Global 'entry)  
12 (Label 'entry)  
13 (Mov rax 40)  
14 (Add rax 1)  
15 (Add rax 1)  
16 (Ret)  
17
```



```
1  
2  
3  
4  
5
```

asm-interp

```
1 #lang racket  
2 (require abscond)  
3  
4 (Lit 42)  
5
```

Expr is an inductive data type

Abstract syntax

```
1 #lang racket
2 (provide Lit Prim1)
3
4 ;; type Expr = (Lit Integer)
5 ;;           | (Prim1 Op1 Expr)
6 ;; type Op1 = 'add1 | 'sub1
7 (struct Lit (i))
8 (struct Prim1 (p e))
9
```



Expr is an inductive data type

1 | #lang racket
2 Abstract syntax

3

(require blackmail)

4

5

(define (expr-template e ...)

6

(match e

7

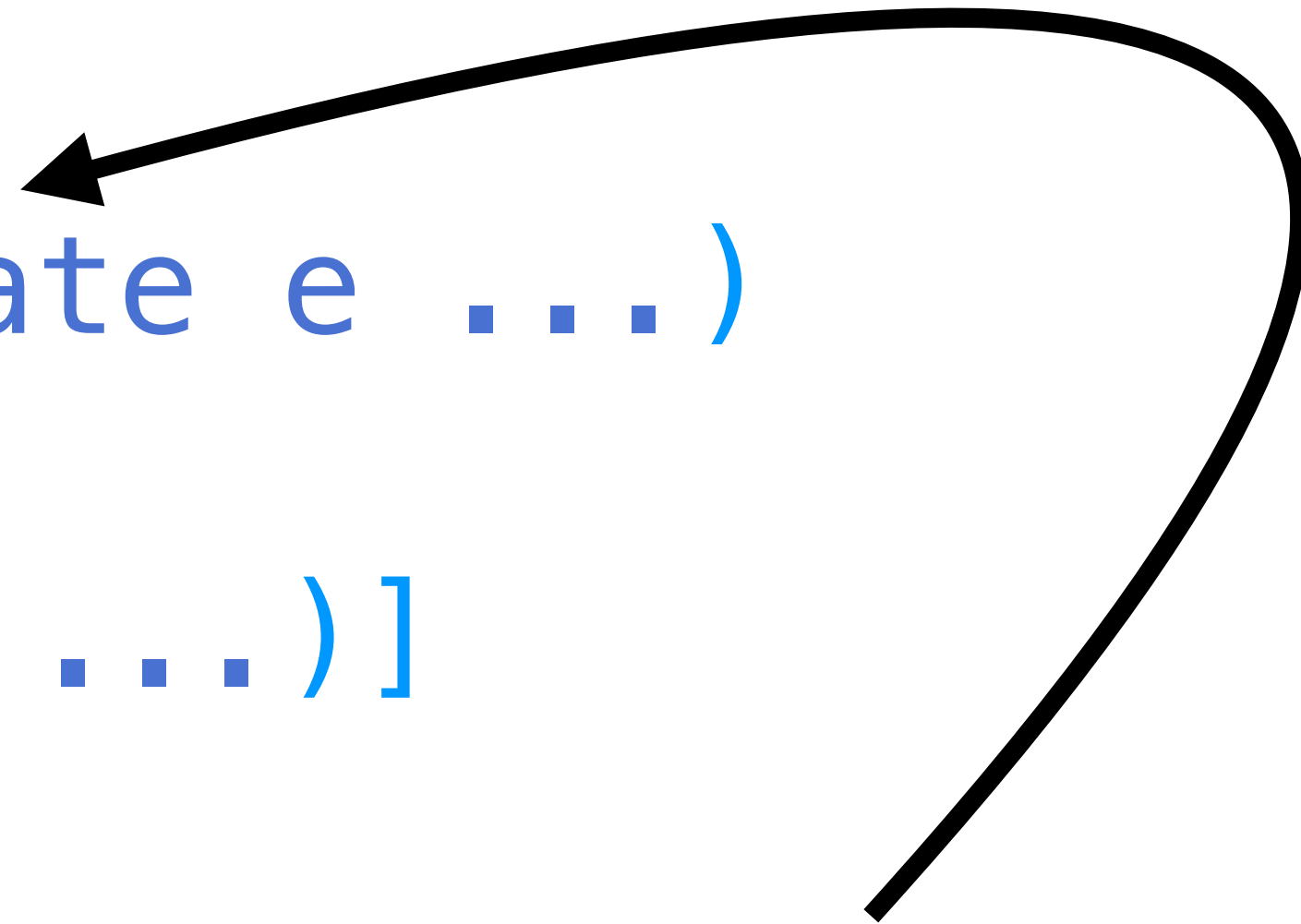
[(Lit i) (... i ...)]

8

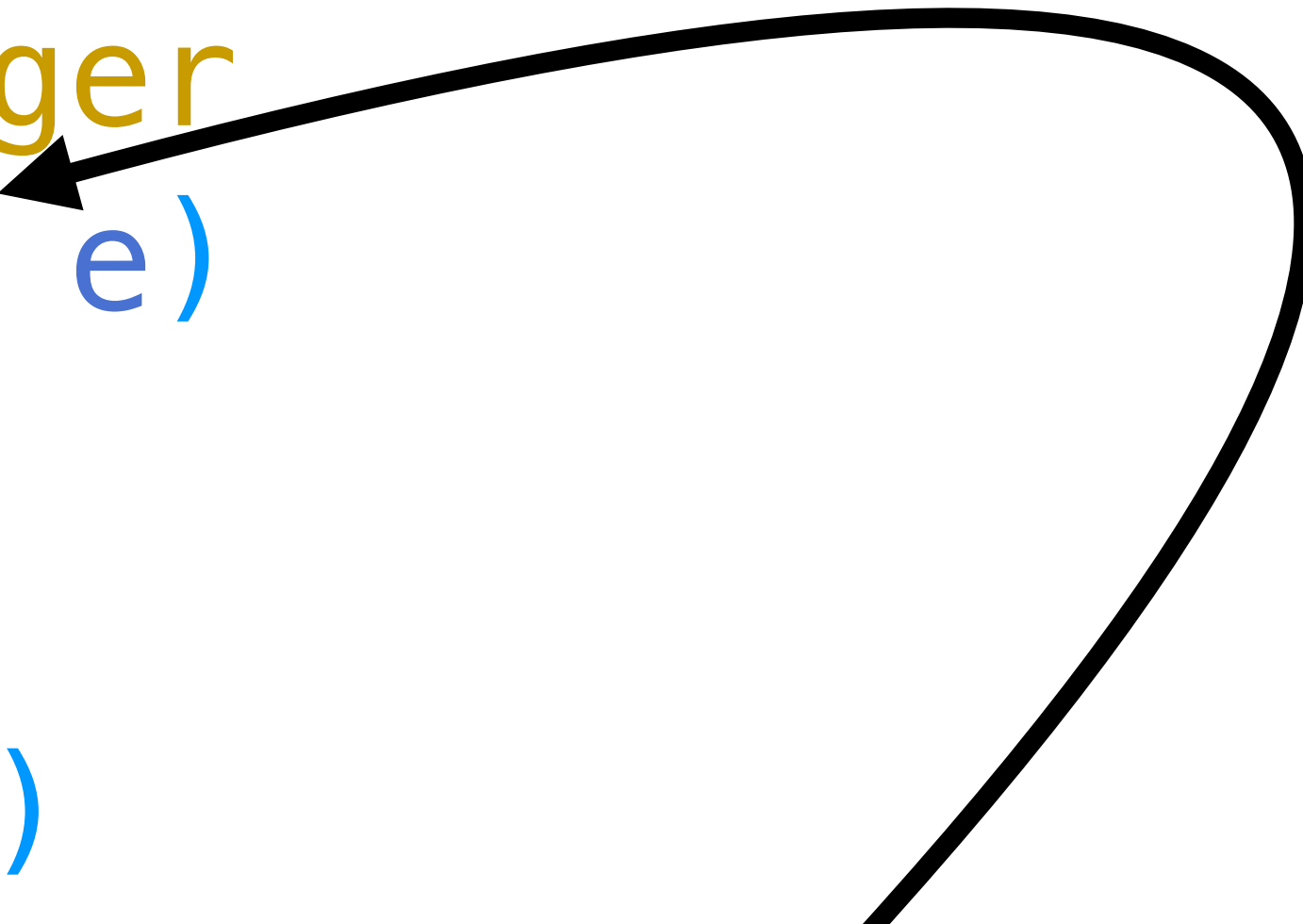
[(Prim1 o e)

9

(... o ... (expr-template e ...) ...)]))



```
1 | #lang racket
Blackmail (provide interp) interpreter
3 | (require "../syntax/ast.rkt")
4 | (require "interp-prim.rkt")
5 |
6 | ;; Expr -> Integer
7 | (define (interp e)
8 |   (match e
9 |     [(Lit i) i]
10 |    [(Prim1 p e)
11 |     (interp-prim1 p (interp e))]
12 |    _))
13 | ;; Op1 Integer -> Integer
14 | (define (interp-prim1 op i)
```



Blackmail interpreter

```
7  (define (interp e)
8    (match e
9      [(Lit i) i]
10     [(Prim1 p e)
11       (interp-prim1 p (interp e))]))
12
13 ;; Op1 Integer -> Integer
14 (define (interp-prim1 op i)
15   (match op
16     ['add1 (add1 i)]
17     ['sub1 (sub1 i)]))
```

Blackmail compiler

Compiling programs

```
3             compile-e)
4
5 (require "../syntax/ast.rkt")
6 (require "compile-ops.rkt")
7 (require a86/ast a86/registers)
8
9 ;; Expr -> Asm
10 (define (compile e)
11   (prog (Global 'entry)
12         (Label 'entry)
13         (compile-e e)
14         (Ret)))
15
16 ;; Expr -> Asm
17 (define (compile-e e)
18   (match e
```

Blackmail compiler

Key invariant

36

37

38

39

40

41

42

43

44

45

;; Invariant:

;; Produces instructions that when run,

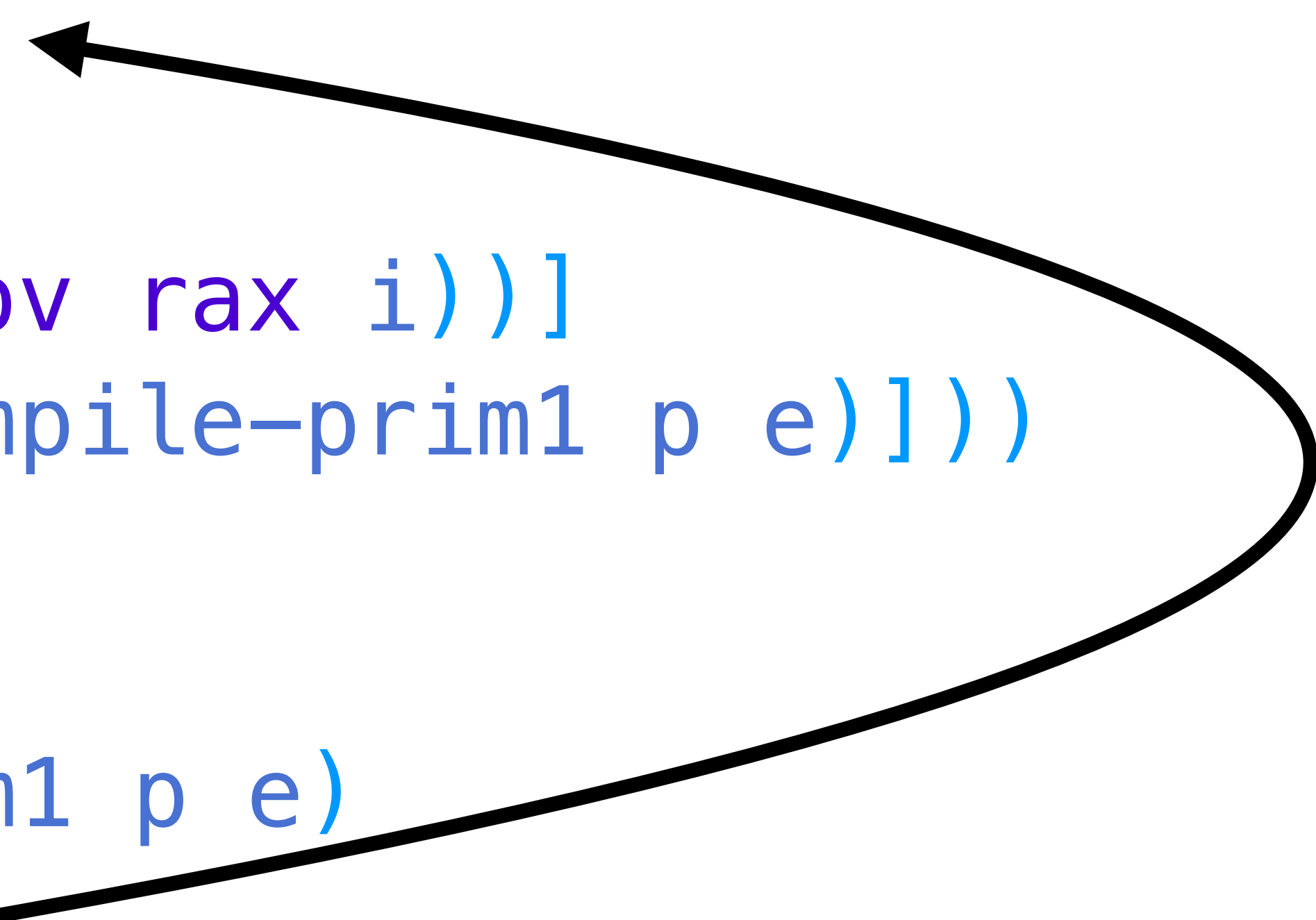
;; leave e's value in rax

(compile-e e)

Blackmail compiler

Compiling expressions

```
11      (prog (Global 'entry))
12      (Label 'entry)
13      (compile-e e)
14      (Ret)))
15
16 ;; Expr -> Asm
17 (define (compile-e e)
18   (match e
19     [(Lit i) (seq (Mov rax i))]
20     [(Prim1 p e) (compile-prim1 p e)]))
21
22 ;; Op1 Expr -> Asm
23 (define (compile-prim1 p e)
24   (seq (compile-e e)
25        (compile-op1 p)))
26
27 · · Op1 -> Asm
```



Blackmail compiler

Compiling primitives

```
21
22 ;; Op1 Expr -> Asm
23 (define (compile-prim1 p e)
24   (seq (compile-e e)
25        (compile-op1 p)))
26
27 ;; Op1 -> Asm
28 (define (compile-op1 p)
29   (match p
30     ['add1 (Add rax 1)]
31     ['sub1 (Sub rax 1)]))
32
33
```

Blackmail compiler

Example

Welcome to [DrRacket](#), version 9.2 [cs].

Language: racket, with debugging; memory limit: 4096 MB.

42

> (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

```
1 | #lang racket
2 | (if (zero? 5)
3 |     (add1 7)
4 |     (sub1 3))
5 |
```

```
99 ;; type Op1 = 'add1 | 'sub1
100
101 (struct Lit (i) #:prefab)
102 (struct Prim1 (p e) #:prefab)
103 (struct IfZero (e1 e2 e3) #:prefab)
104 (struct IfZero (e1 e2 e3) #:prefab)
```

Conditional expressions

Concrete examples:

```
118 (if (zero? 1) 2 3)
119 (if (zero? (sub1 1)) 2 3) ;=> 3
120 (add1 (if (zero? (sub1 1)) 2 3)) ;=> 2
121 (add1 (if (zero? (sub1 1)) (add1 1) 3)) ;=> 3
122 (add1 (if (zero? (sub1 1)) (add1 1) 3)) ;=> 3
123 (if (zero? (Lit 2)) (Lit 3))
```

Abstract examples:

```
224 (IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3))
225 (IfZero (Lit 1) (Lit 2) (Lit 3))
226 (Prim1 'add1 (IfZero (Prim1 'sub1 1) (Lit 2) (Lit 3)))
227 (IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3))
228 (Prim1 'add1 (IfZero (Prim1 'sub1 1) (Prim1 'add1 (Lit 1))
229 (Lit 3)))
230 (Prim1 'add1 (IfZero (Prim1 'sub1 (Lit 1)) (Prim1 'add1 (Lit 1))
231 (Lit 3)))
```

```
232 (define 0)
```

Conditional expressions

Revised template for Con

```
9 ;; type Op1 = 'add1 | 'sub1
10 #lang racket
11 (provide (define-syntax prim1) #:prefab)
12 (struct Prim1 (Op1 Expr) #:prefab)
13 (struct IfZero (e1 e2 e3) #:prefab)
14
15 ;; type Expr = (Lit Integer)
16 ;;           | (Prim1 Op1 Expr)
17 (define ... 0) (IfZero Expr Expr Expr)
18
19 ;; Expr Op1? = 'add1 | 'sub1
20 (define (expr-template e)
21   (match Lit (i) #:prefab)
22   (struct Prim1 (. Op1 Expr) #:prefab)
23   (struct IfZero (. e1 e2 e3) #:prefab) ...))
24   [(IfZero e1 e2 e3)
25    (... (expr-template e1)
26         (expr-template e2)
27         (expr-template e3) ...))]
28 (if (zero? 1) 2 3) ;=> 3
29 (if (zero? (sub1 1)) 2 3) ;=> 2
```


Con: adding conditional expressions

Use examples and template to write interp

```
2 (provide Lit Prim1
3   IfZero)
4
5 ;; type Expr = (Lit Integer)
6 ;;           | (Prim1 Op1 Expr)
7 ;;           | (IfZero Expr Expr Expr)
8
9 ;; type Op1 = 'add1 | 'sub1
10
11 (struct Lit (i) #:prefab)
12 (struct Prim1 (p e) #:prefab)
13 (struct IfZero (e1 e2 e3) #:prefab)
14
15
16 (define ... 0)
17
18 ;; Expr -> ?
19 (define (expr-template e)
20   (match e
21     [(Lit i) (... i ...)]
22     [(Prim1 p e) (... p (expr-template e) ...)]
23     [(IfZero e1 e2 e3)
24      (... (expr-template e1)
25            (expr-template e2)
26            (expr-template e3) ...)]))
27
```

```
4
5 ;; type Expr = (Lit Integer)
6 ;;           | (Prim1 Op1 Expr)
7 ;;           | (IfZero Expr Expr Expr)
8
9 ;; type Op1 = 'add1 | 'sub1
10
11 (struct Lit (i) #:prefab)
12 (struct Prim1 (p e) #:prefab)
13 (struct IfZero (e1 e2 e3) #:prefab)
14
15
16
17
18 (if (zero? 1) 2 3) ;=> 3
19 (if (zero? (sub1 1)) 2 3) ;=> 2
20 (add1 (if (zero? (sub1 1)) 2 3)) ;=> 3
21 (add1 (if (zero? (sub1 1)) (add1 1) 3)) ;=> 3
22
23 (IfZero (Lit 1) (Lit 2) (Lit 3))
24 (IfZero (Prim1 'sub1 (Lit 1)) (Lit 2) (Lit 3))
25 (Prim1 'add1 (IfZero (Prim1 'sub1 1) (Lit 2) (Lit 3)))
26 (Prim1 'add1 (IfZero (Prim1 'sub1 1) (Prim1 'add1 (Lit 1)
27
```

```
25 (define (compile-prim1 p e)
```

```
26 (define (seq (compile-e e)
```

```
27 (compile-op1 p)))
```

```
28
```

```
29 ;; Expr Expr Expr -> Asm
```

```
30 (define (compile-ifzero e1 e2 e3)
```

```
31   (... (compile-e e1)
```

```
32         (compile-e e2)
```

```
33         (compile-e e3) ...))
```

```
34
```

```
35
```

```
36
```

```
37 (define ... 1)
```

```
38 #|
```

```
22      (compile-ifzero e1 e2 e3])))
```

```
23
```

```
24 ;; Op1 Expr -> Asm
```

```
25 (define (compile-prim1 p e)
```

```
26   (seq (compile-e e)
```

```
27         (compile-op1 p)))
```

```
28
```

```
29 ;; Expr Expr Expr -> Asm
```

```
30 (define (compile-ifzero e1 e2 e3)
```

```
31   ;; Goal: instrs that leave (if (zero? e1) e2 e3) value in rax
```

```
32   (... (compile-e e1) ;; instrs that leave e1's value in rax
```

```
33         (compile-e e2) ;; instrs that leave e2's value in rax
```

```
34         (compile-e e3) ;; instrs that leave e3's value in rax
```

```
35         ...))
```

```
36
```

```
37
```

```
38
```

```
39 (define ... 1)
```

```
40 #|
```


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?


```
26      (check-equal? (interp (parse '(add1 (if (zero? (sub1 1))
26      3)
27      (check-equal? (interp (parse '(add1 (if (zero? (sub1 1))
27      1) 3)) 3))
28
29
30
31      (if (zero? 1) 2 3)                ;=> 3
32      (if #t 1 2)                       ;=> 1
33      (if #f 1 2)                       ;=> 2
34      (if 0 1 2)                        ;=> ?
35      (zero? (if #t 1 2))               ;=> #f
36      (if (zero? (sub1 1)) (zero? 0) (zero? 1)) ;=> #t
```

Dupe: adding Boolean values

Abstract Syntax

```
1 | #lang racket
2 | (provide Lit Prim1 If)
3 | ;; type Expr = (Lit Datum)
4 | ;;           | (Prim1 Op1 Expr)
5 | ;;           | (If Expr Expr Expr)
6 |
7 | ;; type Datum = Integer
8 | ;;           | Boolean
9 |
10 | ;; type Op1 = 'add1 | 'sub1
11 | ;;          | 'zero?
12 |
13 | (struct Lit (d) #:prefab)
14 | (struct Prim1 (p e) #:prefab)
15 | (struct If (e1 e2 e3) #:prefab)
16 |
```

```
12 ;; | '23 zero? (require "parse.rkt")
13 24 (check-equal? (interp (parse '(if (zero? 1) 2 3))) 3)
14 (struct Lit (d) #:prefab) (check-equal? (interp (parse '(if (zero? (sub1 1)) 2 3))) 2)
15 (struct Prim1 (p e) #:prefab) (check-equal? (interp (parse '(add1 (if (zero? (sub1 1)) 2 3))))
16 (struct If (e1 e2 e3) #:prefab)
17 27 (check-equal? (interp (parse '(add1 (if (zero? (sub1 1)) (add1
18 27 1) 3)))) 3))
19 28
20 29
21 Concrete examples:
22 (if (zero? 1) 2 3) (if (zero? 1) 2 3) ;=> 3 ;=> 3
23 (if #t 1 2) 32 (if #t 1 2) ;=> 1 ;=> 1
24 (if #f 1 2) 33 (if #f 1 2) ;=> 2 ;=> 2
25 (if 0 1 2) 34 (if 0 1 2) ;=> ? ;=> ?
26 (zero? (if #t 1 2)) 35 (zero? (if #t 1 2)) ;=> #f ;=> #f
27 (if (zero? (sub1 1)) (zero? 0) (zero? 1)) ;=> #t ;=> #t
28 36 (if (zero? (sub1 1)) 1 (zero? 1))
29
30 Abstract examples:
31
32 (If (Prim1 'zero? (Lit 1)) (Lit 2) (Lit 3))
33 (If (Lit #t) (Lit 1) (Lit 2))
34 (If (Lit #f) (Lit 1) (Lit 2))
35 (If (Lit 0) (Lit 1) (Lit 2))
36 (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))]))
```

```
1 #lang racket  
2 (provide interp-prim1)  
3  
4  
5 ;; Op1 Value -> Value  
6 (define (interp-prim1 op v)  
7   (match op  
8     ['add1 (add1 v)]  
9     ['sub1 (sub1 v)]  
10    ['zero? (zero? v)]))  
11
```

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*

```
11 (define (compile e)
12 (prog (Global entry)
13 What to do with new literals? (Label entry)
14 (compile-e e)
15 (Ret)))
16
```

```
17 (compile-e (Lit 42)) ;=> (Mov rax 42)
18 (compile-e (Lit #f)) ;=> (Mov rax ??)
19
20
```

```
21 ;; Expr -> Asm
22 (define (compile-e e)
23 (match e
```


For next time

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