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))
```

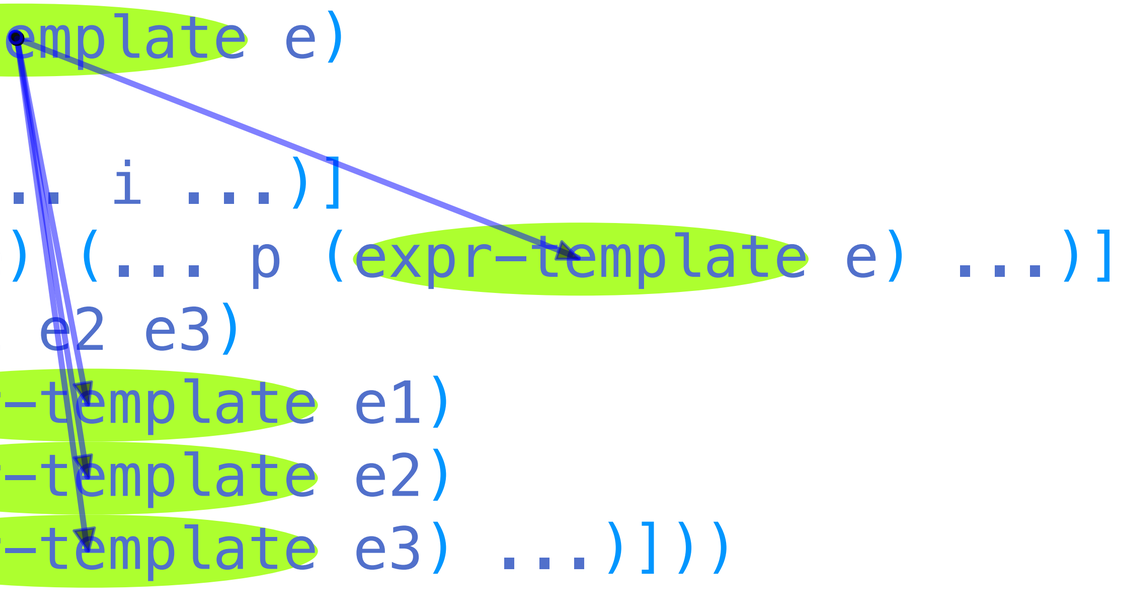
Abstract examples:

```
(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

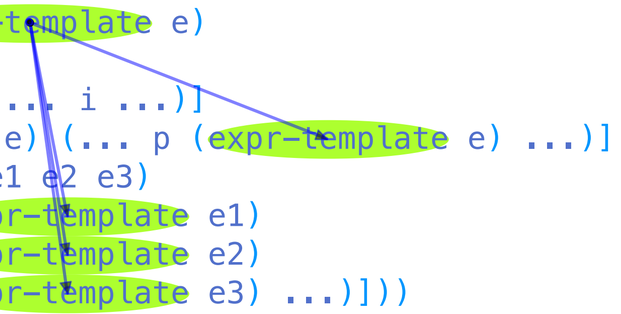
```
;; type Expr = (Lit Integer)
;;           | (Prim1 Op1 Expr)
;;           | (IfZero Expr Expr Expr)

;; 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

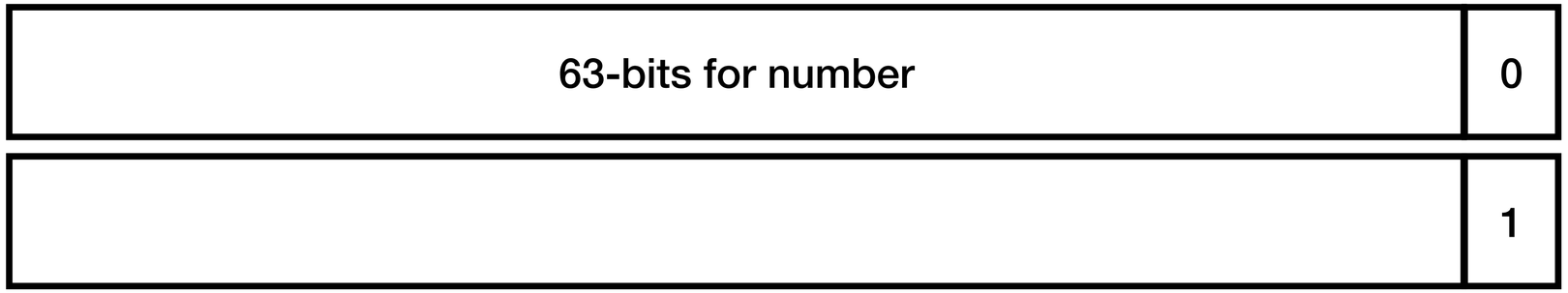
- some will represent integer values
- some will represent boolean values

Criteria: the contents of rax must uniquely determine the value

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

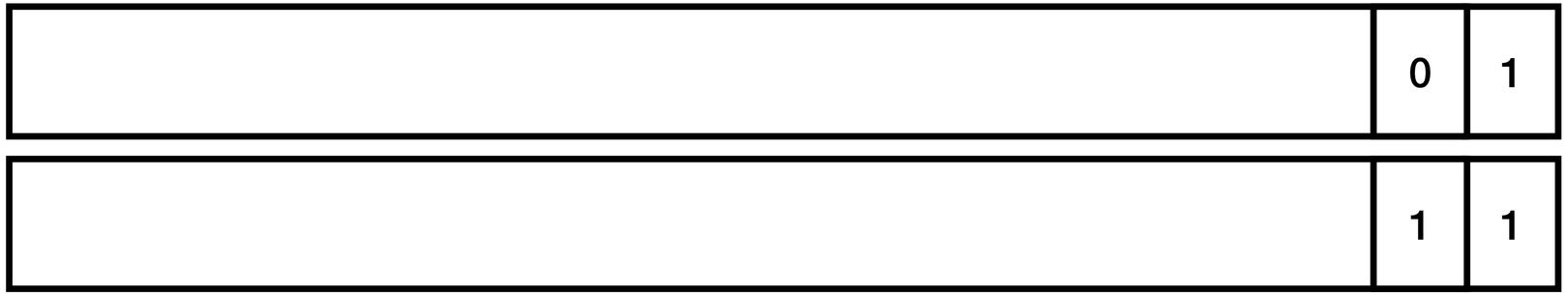
Encoding values in Dupe

Type tag in least significant bits



Integers

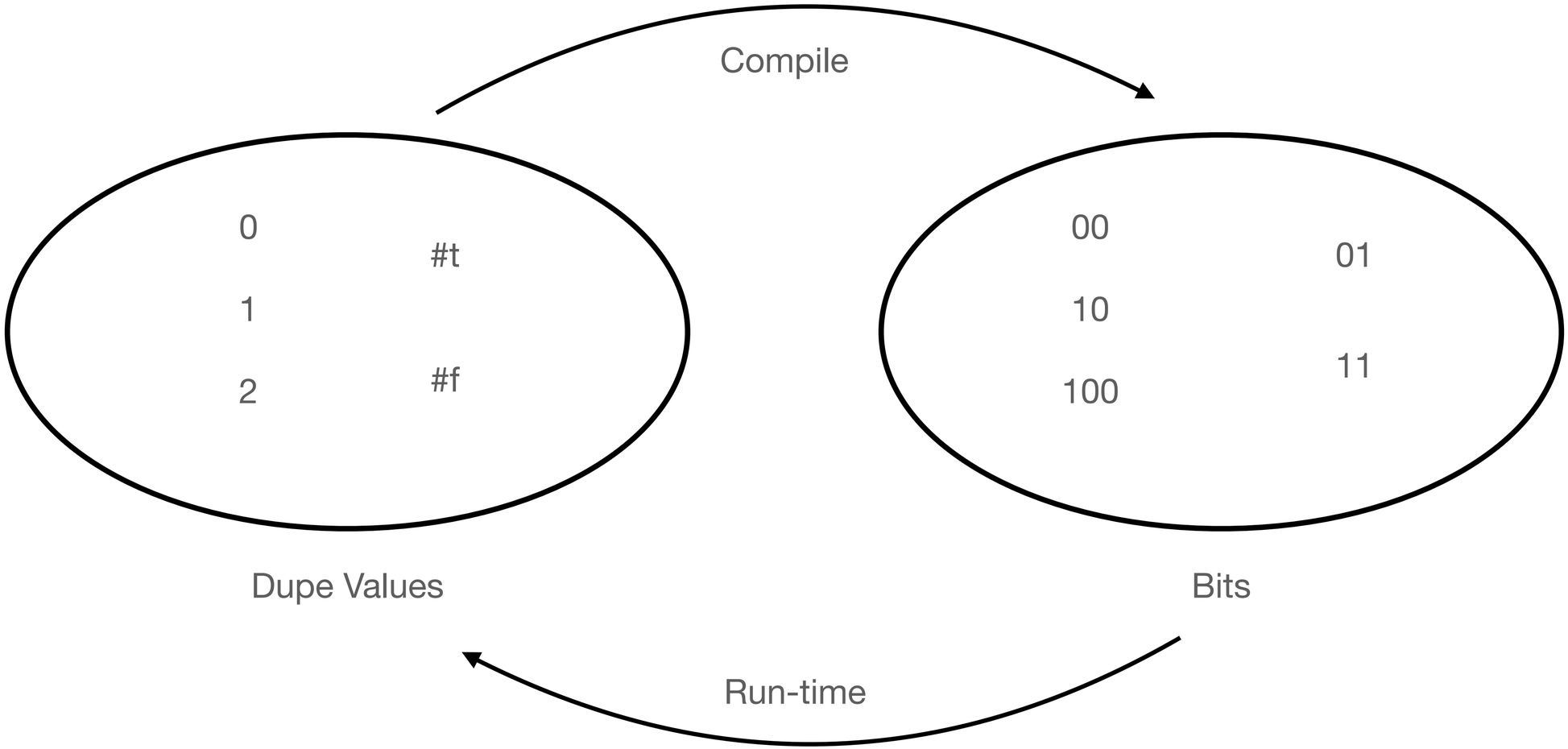
Booleans



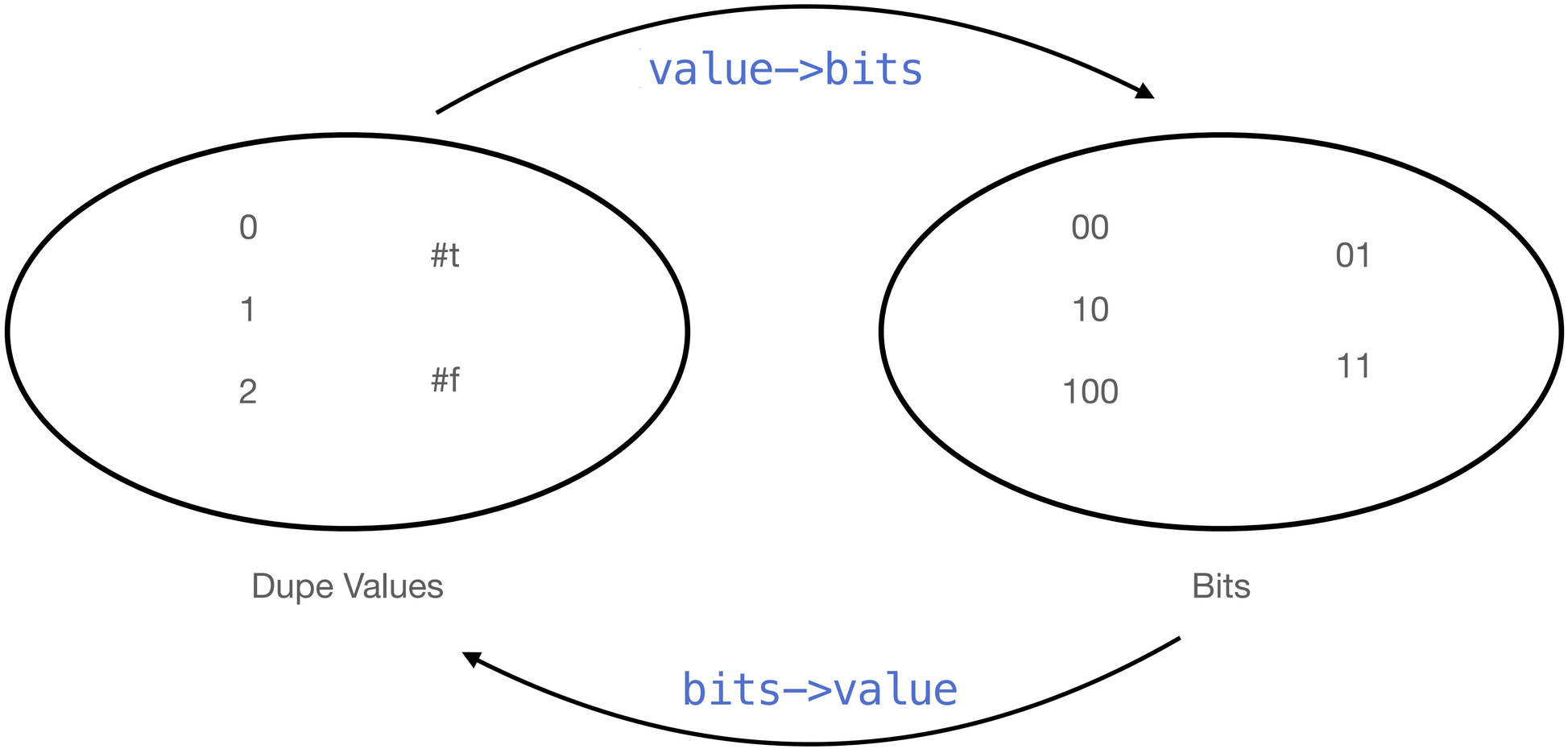
`#t`

`#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

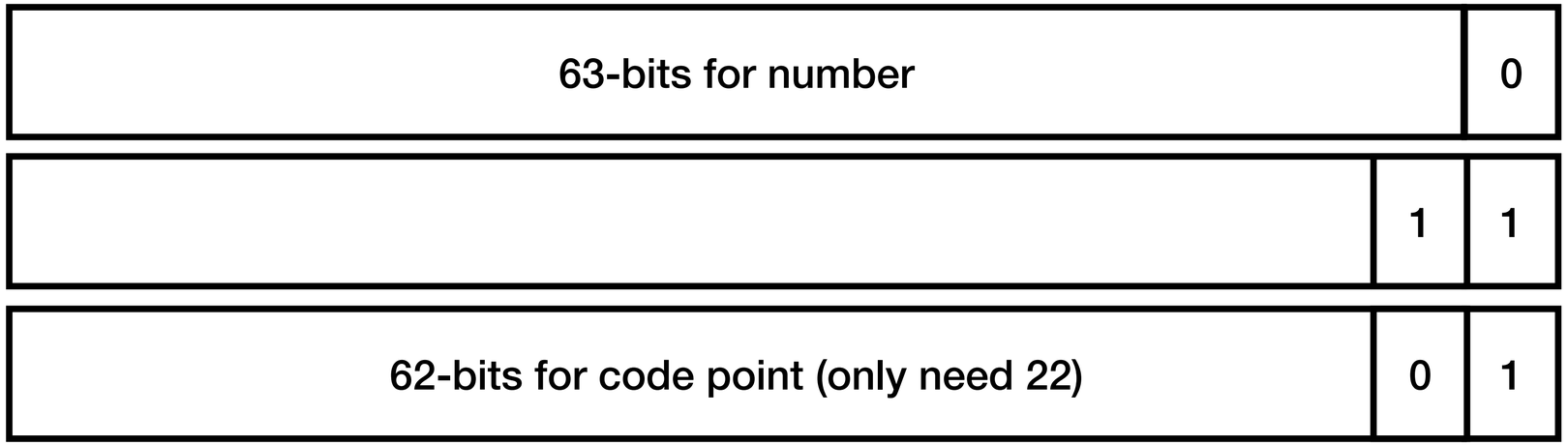
`#\a #\b #\c ... #\λ #\絳 ...`

Unicode: roughly 150K characters.

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

Encoding values in Dodger

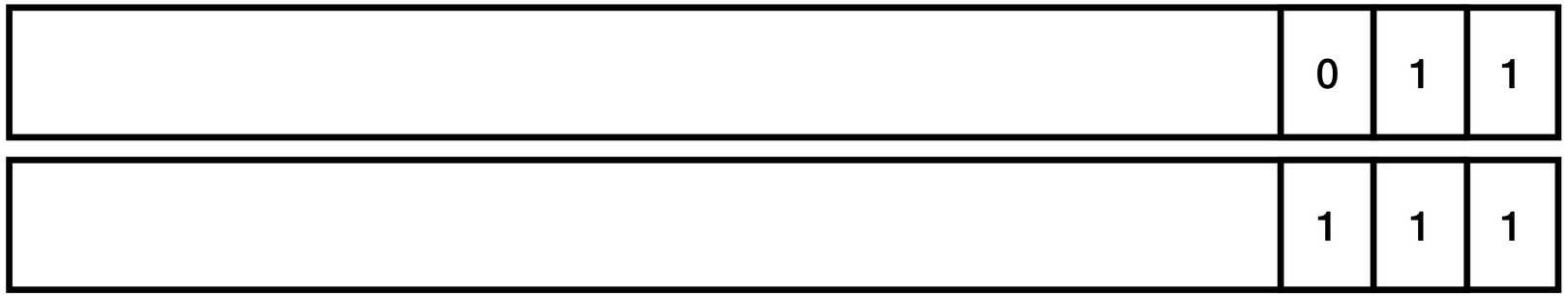
Type tag in least significant bits



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