# CMSC 430 - 15 Sept 2026

## Representing disjoint data; I/O effects

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

# CMSC 430 - 15 Sept 2026

## Announcements

- Assignment 2 due Thurs, midnight
- New quiz released today, due by start of class Thursday

# 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

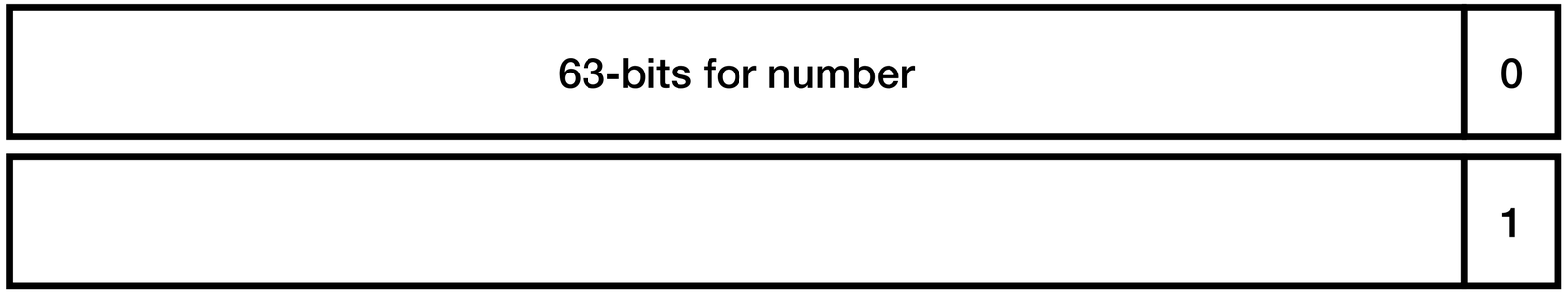
- some will represent integer values
- some will represent boolean values

Criteria: the contents of rax must uniquely determine the value

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

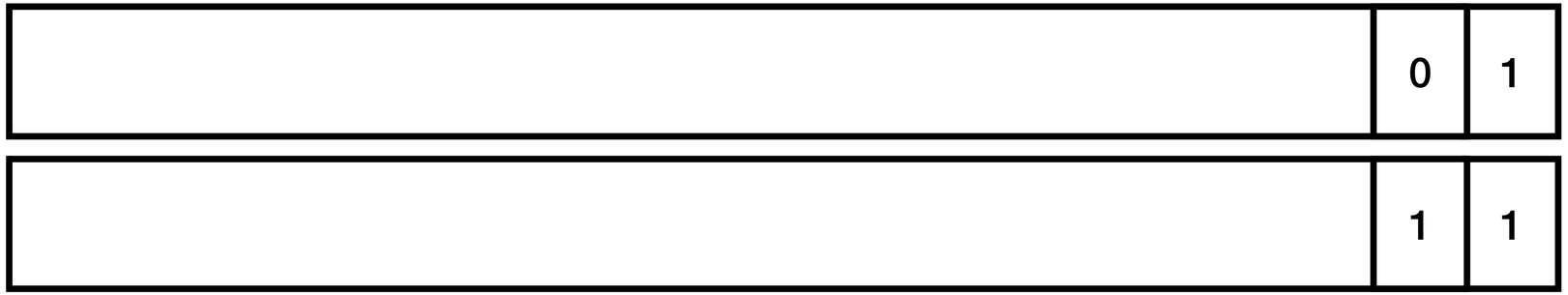
# Encoding values in Dupe

## Type tag in least significant bits




Integers

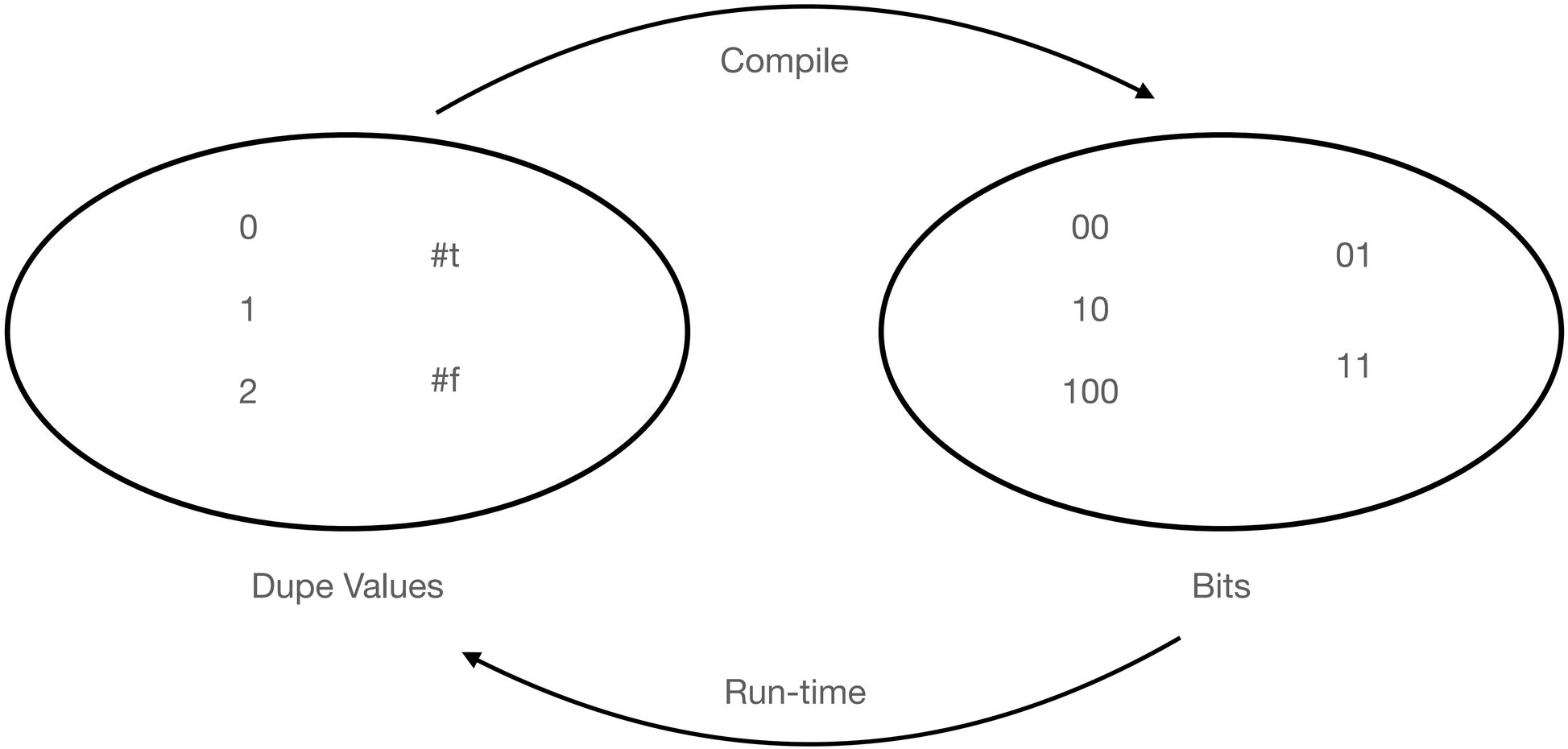
Booleans



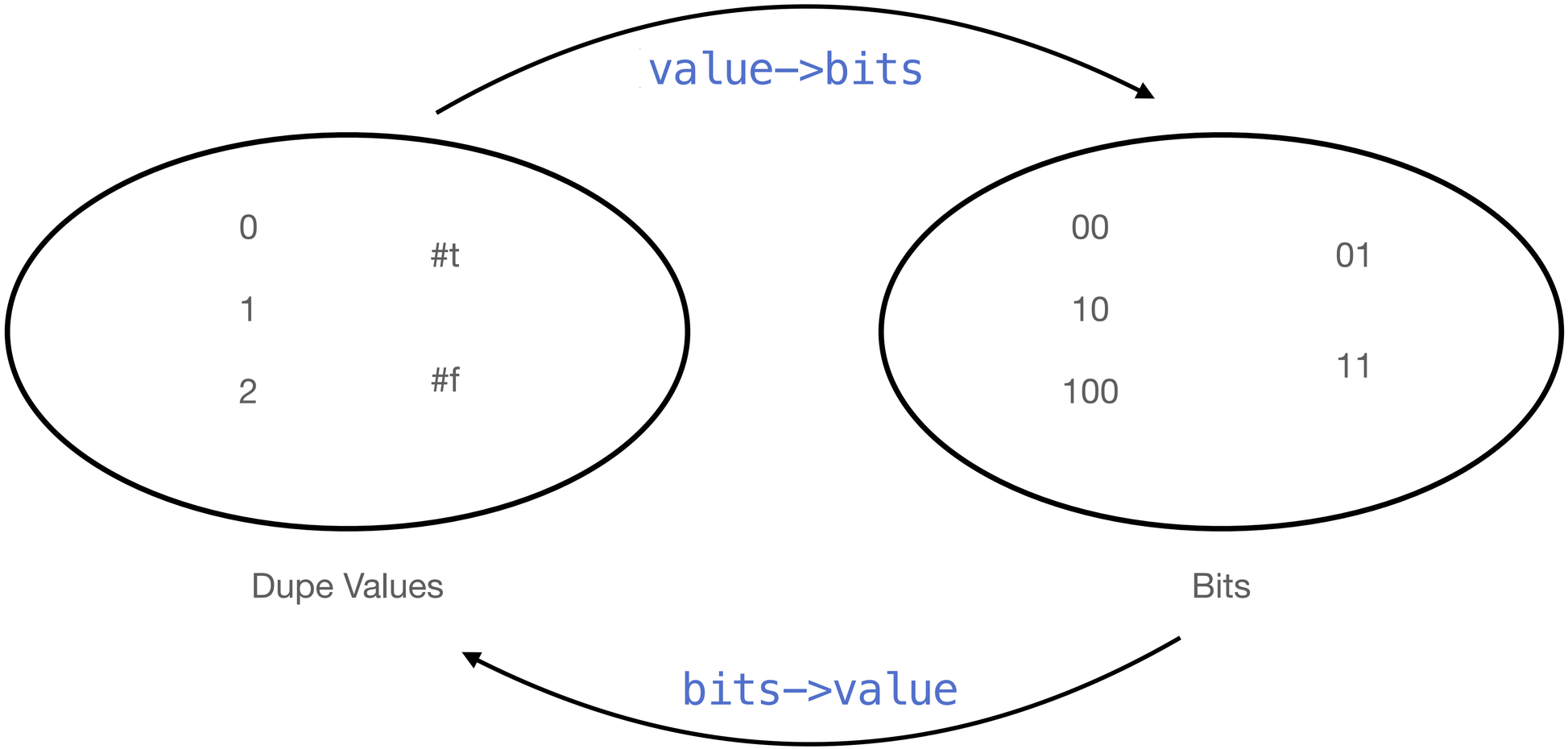

`#t`

`#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))
```

# Hosted run-time revisions

## Interpret the bits returned by compiled code

```
;; Asm -> Value
(define (run asm)
  (bits->value (asm-interp asm)))

;; Expr -> Value
(define (exec e)
  (run (compile e)))
```

```
> (exec (parse '#t))
#t
> (exec (parse '#f))
#f
> (exec (parse '0))
0
> (exec (parse '1))
1
```

# Stand-alone run-time revisions

## Interpret the bits returned by compiled code

```c
#include <stdio.h>
#include "values.h"
#include "print.h"

val_t entry();

int main(int argc, char** argv)
{
  val_t result = entry();
  print_result(result);
  putchar('\n');
  return 0;
}
```

```c
#include <stdio.h>
#include <inttypes.h>
#include "values.h"

void print_result(val_t x)
{
  switch (val_typeof(x)) {
  case T_INT:
    printf("%" PRId64, val_unwrap_int(x));
    break;
  case T_BOOL:
    printf(val_unwrap_bool(x) ? "#t" : "#f");
    break;
  case T_INVALID:
    printf("internal error");
  }
}
```

# 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

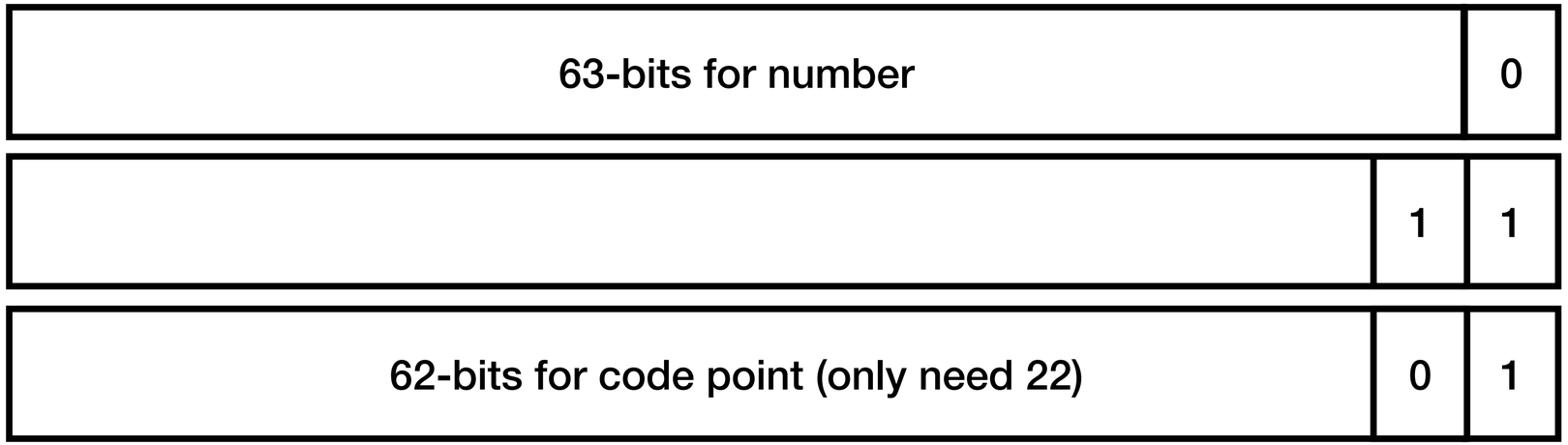
`#\a #\b #\c ... #\λ #\絳 ...`

Unicode: roughly 150K characters.

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

# Encoding values in Dodger

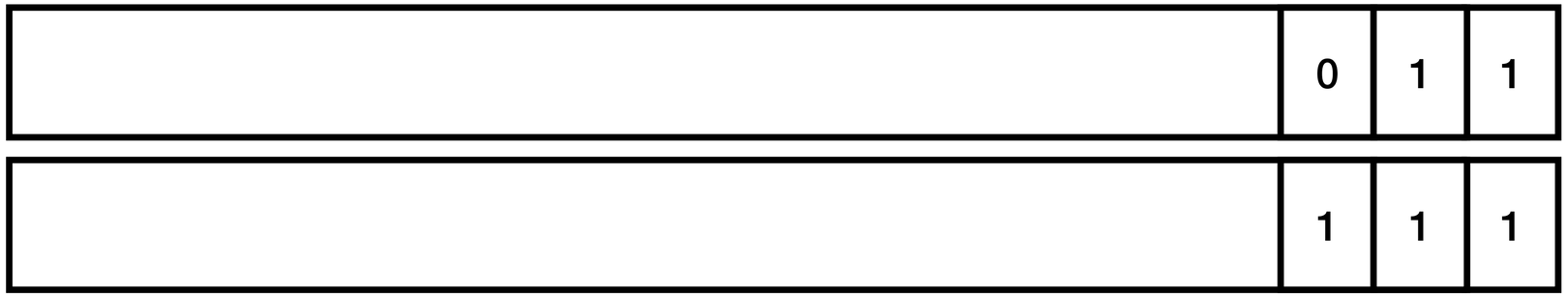
## Type tag in least significant bits




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
```

# 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 | Abstract syntax |
| --- | --- |
| `(begin e1 e2)` | `(Begin e1 e2)` |
| `(read-byte)` | `(Prim0 'read-byte)` |
| `(peek-byte)` | `(Prim0 'peek-byte)` |
| `(void)` | `(Prim0 'void)` |
| `(write-byte e)` | `(Prim1 'write-byte e)` |
| `(eof-object? 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)
```

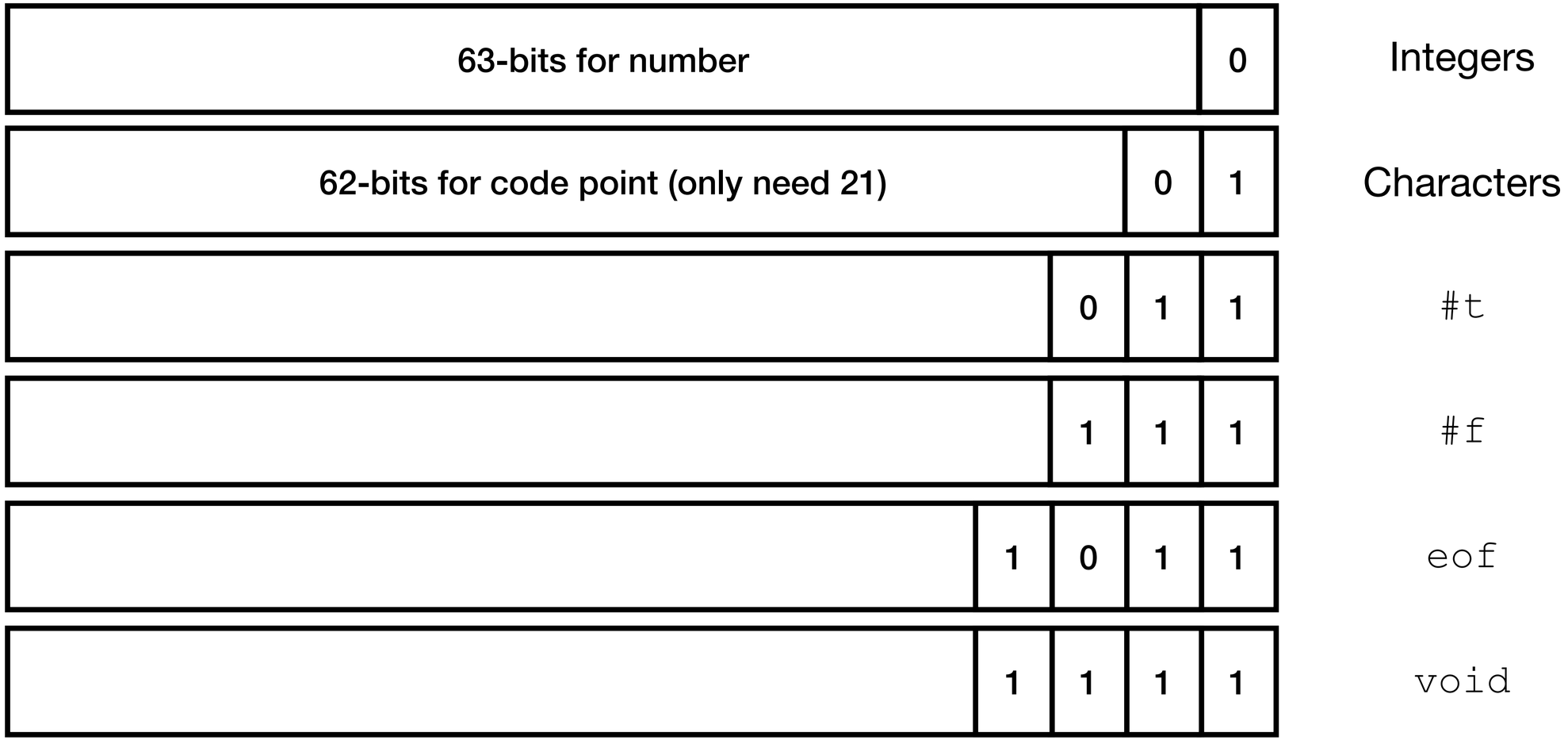
# Semantic additions in Evildoer

```
;; 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



# 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

```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();
}
```