CMSC 430 - 10 June 2026

Allocating data in the heap

- Checking in
- Quick look at Assignment 3
- Hustle - Pairs, boxes, and lists

CMSC 430 - 10 June 2026

Announcements

- Reminder: Assignment 3 due Thursday by midnight
- Practice exam 1 released
- Assignment 4 released later today
- New quiz released, due by start of class tomorrow

Checking in

How is the pace of this class?

Way too fast	3 respondents	**6 %**
A little too fast	17 respondents	35 %
Just right	28 respondents	58 %
A little too slow		0 %
Way too slow		0 %

How often are you attending Office Hours?

Several times a week	2 respondents	4 %
About once a week		0 %
Almost never	42 respondents	88 %
No answer text provided.	4 respondents	8 %

Suggestions

For the slides, is it possible to have them separated based on lecture day or content rather than keeping them in a whole big file? It is kind of hard to scroll down as we get more in-depth into the course.

Can you make an announcement about what assignment we will have each week?

For longer projects maybe you can make a short video overview about it.

Hustle

Learning objectives

Hustle

You should be able to answer:

- How can inductive data be represented with heap allocated memory?
- How can type-tagging extend to pointer values?
- How is memory allocated at run-time?
- What's the difference between an immediate and a pointer value?
- What's the impact of our type tagging scheme on immediates?

Example concrete Hustle program

Lists

```
#lang racket
(car (cdr (cons 1 (cons 2 (cons 3 '())))))
```

Syntactic additions

```
;; type Datum = ...
;;            | '()

;; type Op1 = ...
;;          | 'car | 'cdr | 'unbox
;;          | 'empty? | 'cons? | 'box?
;;          | 'box

;; type Op2 = ...
;;          | 'cons
```

Semantic additions

Values are now inductively defined

```
;; type Value = ...
;; | '()
;; | (cons Value Value)
;; | (box Value)
```

Semantic additions

Values are now inductively defined

```
;; Op1 Value -> Value { raises 'err }
(define (interp-prim1 op v)
  (match (list op v)
    ; ...
    [(list 'box v)           (box v)]
    [(list 'unbox (? box?))  (unbox v)]
    [(list 'car (? pair?))   (car v)]
    [(list 'cdr (? pair?))   (cdr v)]
    [(list 'empty? v)        (empty? v)]
    [(list 'cons? v)         (cons? v)]
    [(list 'box? v)          (box? v)]
    [_ (raise 'err)]))

;; Op2 Value Value -> Value { raises 'err }
(define (interp-prim2 op v1 v2)
  (match (list op v1 v2)
    ; ...
    [(list 'cons v1 v2) (cons v1 v2)]
    [_ (raise 'err)]))
```

We follow the familiar pattern:

- The empty list will be represented by Racket's empty list
- Pairs will be represented by Racket pairs
- Boxes will be represented by Racket boxes

Easy, but doesn't shed light on how these things are made.

How to represent pointers?

Addressing memory

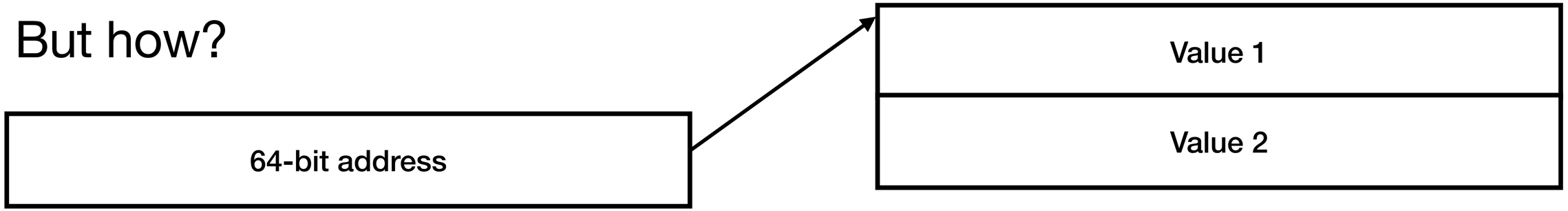
Basic idea:

A pair is allocated as two words in memory

The pair *value* will be represented by the address

+ something indicating the value is a pair

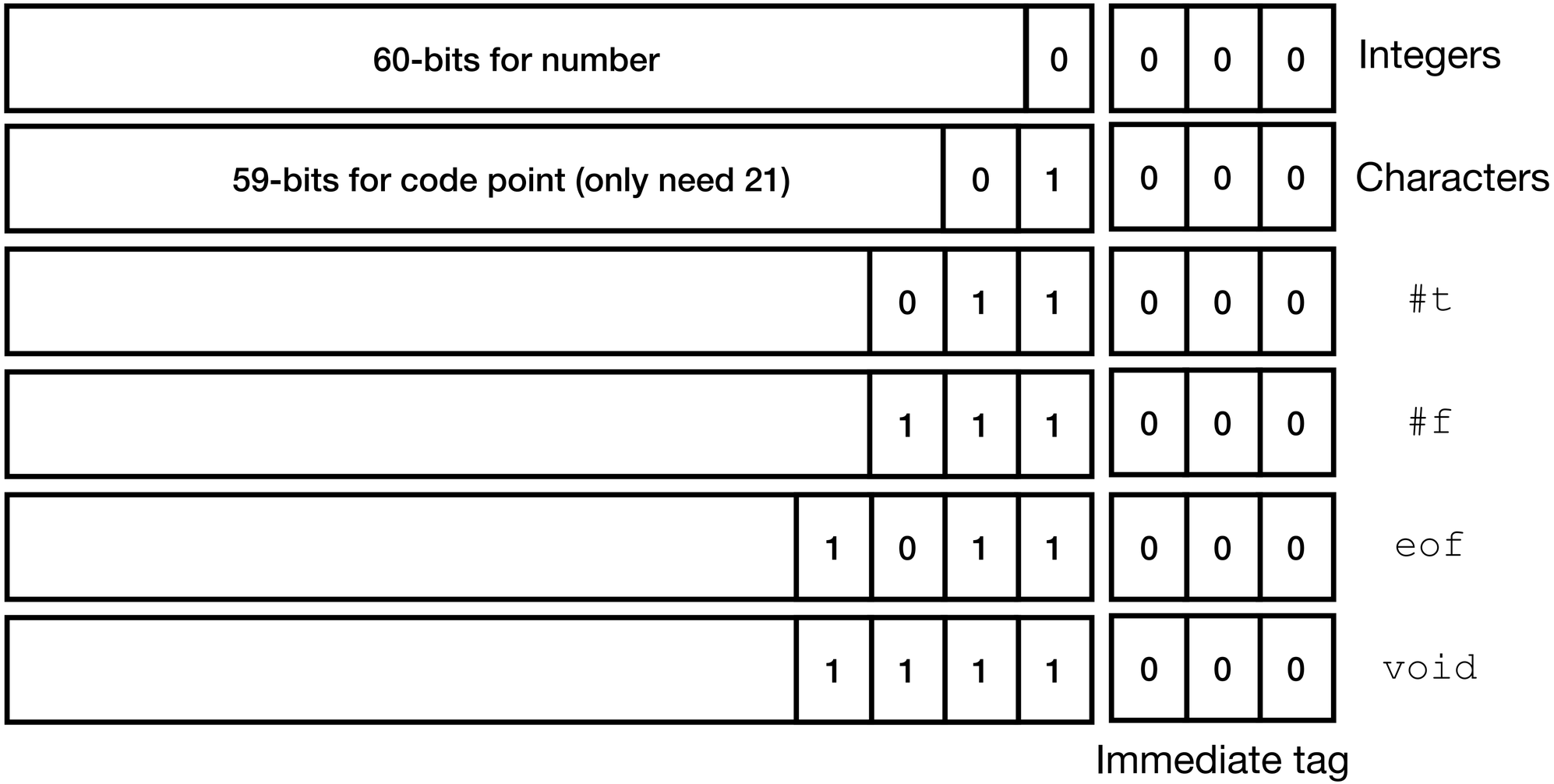
But how?



Hint: we'll always allocate memory in multiples of 8-bytes (64-bits)

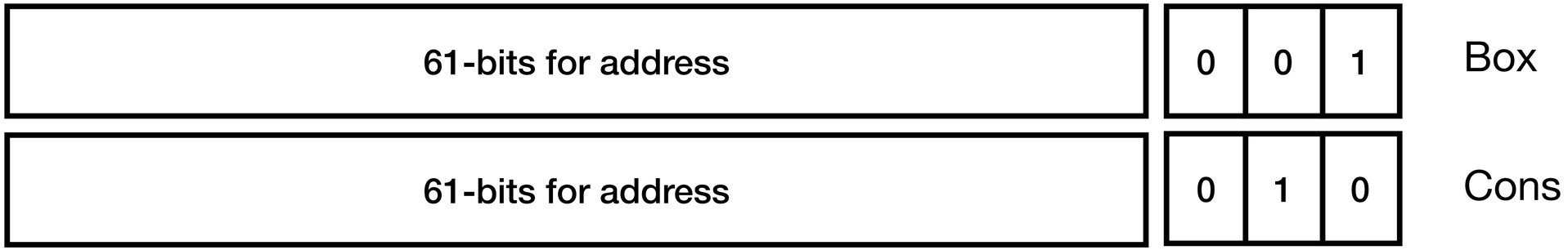
Encoding immediate values (Hustle)

Type tag in least significant bits

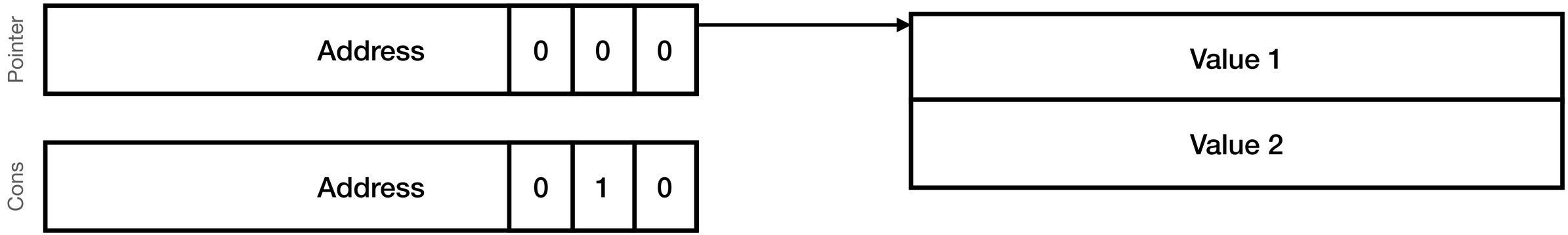


Encoding pointer values (Hustle)

Type tag in least significant bits



Values that point to memory



Idea:

A pair is allocated as two words in memory

The pair *value* will be represented by the address + type tag in 3 least significant bits

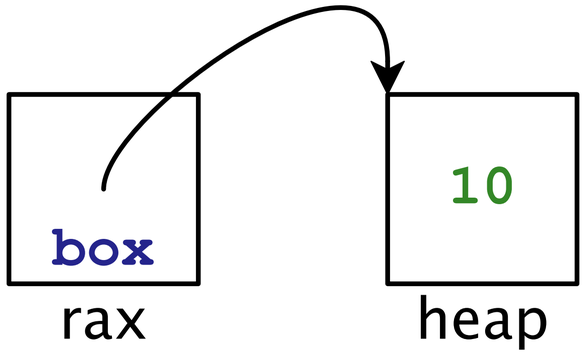
Pointers end in #b000 because we allocate in multiples of 8 bytes: advantage of this and stash type tag there (no shifting).

Observe: pointers don’t have a type, values do!

Creating a box

A pair is a tagged pointer to one value in memory

```
> (compile-op1 'box)
(list
 (Mov (Mem 'rbx) 'rax)
 (Mov 'rax 'rbx)
 (Xor 'rax 1)
 (Add 'rbx 8))
> (exec (parse '(box 10)))
'#&10
```



Creating a pair

A box is a tagged pointer to two values in memory

```
> (compile-op2 'cons)
(list
 (Mov (Mem 'rbx 8) 'rax)
 (Pop 'rax)
 (Mov (Mem 'rbx 0) 'rax)
 (Mov 'rax 'rbx)
 (Xor 'rax 2)
 (Add 'rbx 16))
> (exec (parse '(cons #t #f)))
'(#t . #f)
```