CMSC 430 - 6 Oct 2026

Allocating data in the heap

- Hustle - Pairs, boxes, and lists

CMSC 430 - 6 Oct 2026

Announcements

- Exam 1 grades by Thursday
- Assignment 4 due Thursday, midnight
- New quiz out, due Thursday before class
- Assignment 5 out later today, due 10/15

Hustle

Learning objectives

Hustle

You should be able to answer:

- How can inductive data be represented?
- 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.

Fundamental issue

How to represent two values in a single register?

`'(1 . 2)` is a value which contains two values

How can we represent both values in `rax`?

Idea: split register 32-bits for car, 32-bits for cdr.

But… the car and cdr may themselves be pairs…

Better idea: use memory and pointers to memory.

How to represent pointers?

Addressing memory

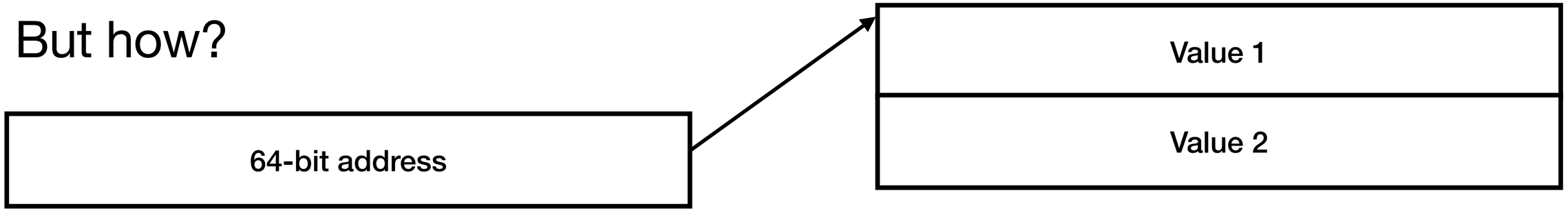
Basic idea:

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

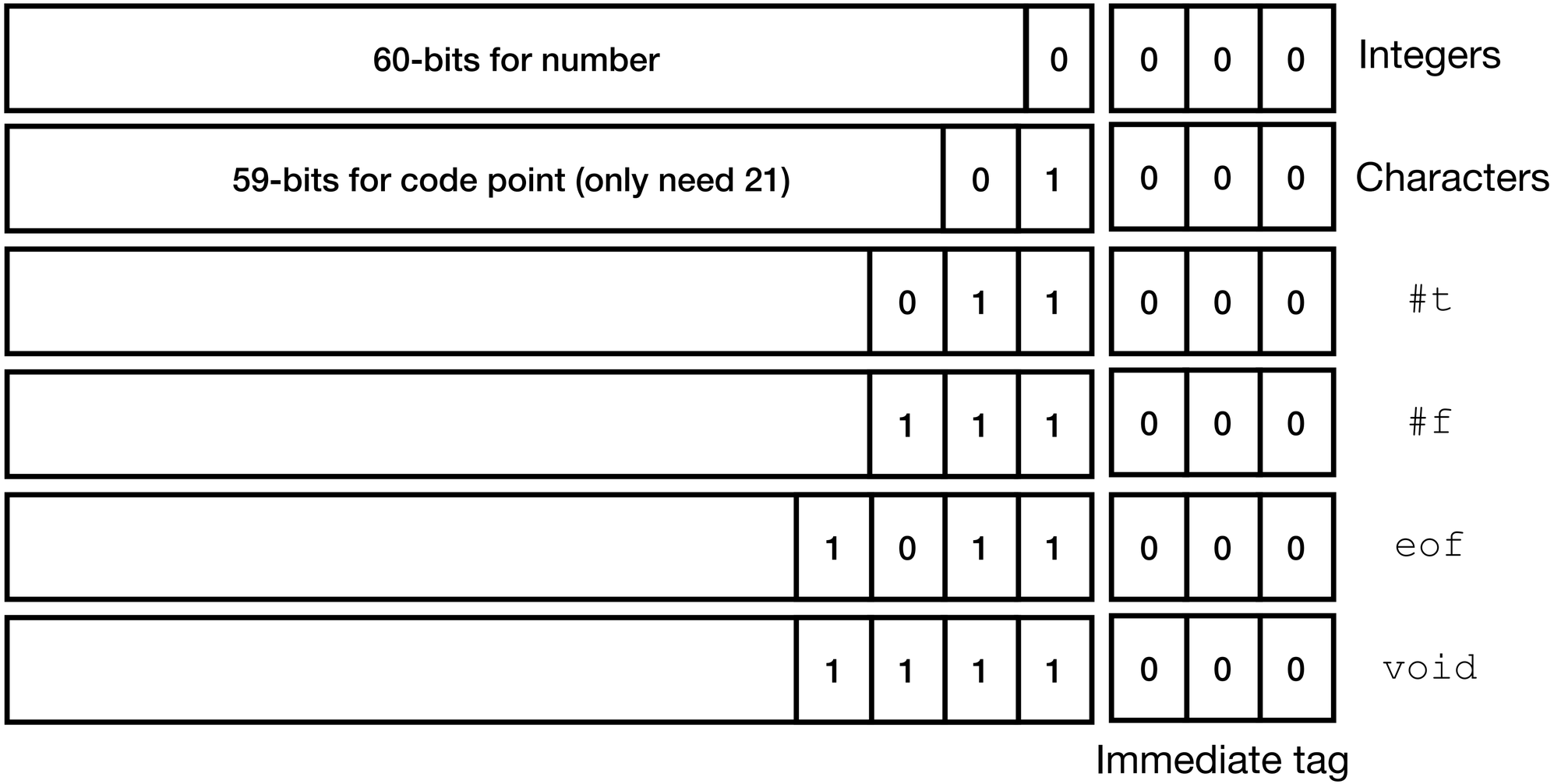
But how?



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

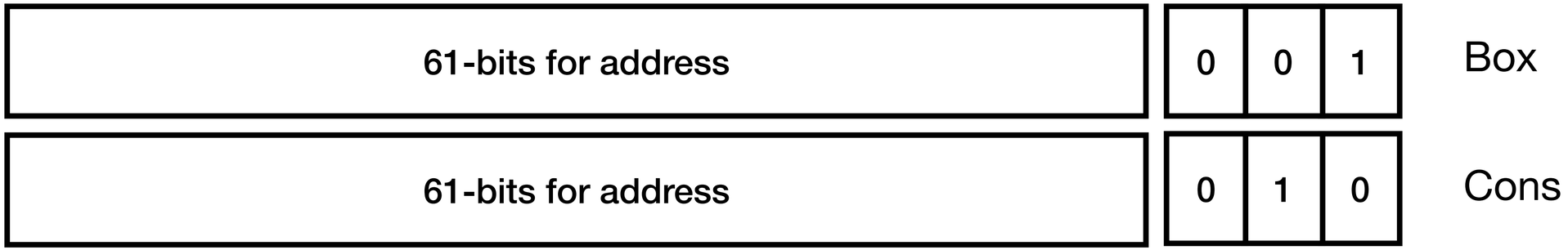
Encoding immediate values (Hustle)

Type tag in least significant bits

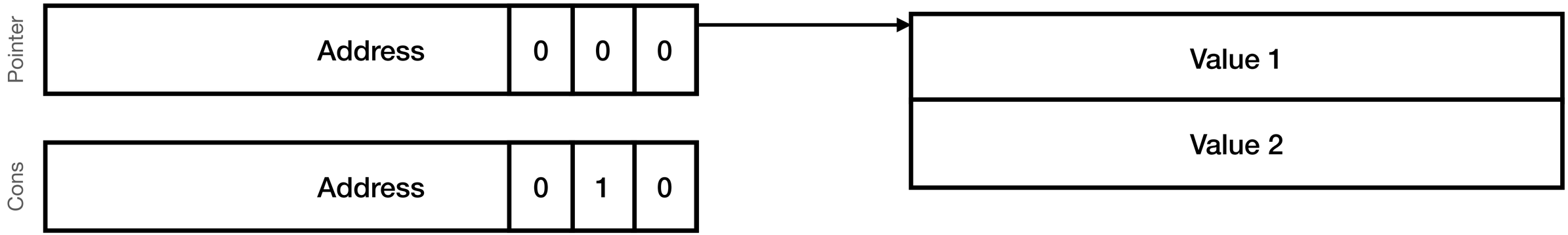


Encoding pointer values (Hustle)

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!

Allocating memory

The `rbx` register will point to free memory

To allocate memory:

- write to memory starting at `rbx`
- move `rbx` to end of memory written
- always allocate in 8-byte quantities

Run-time system will allocate heap and pass in `rdi` register

Top-level set-up

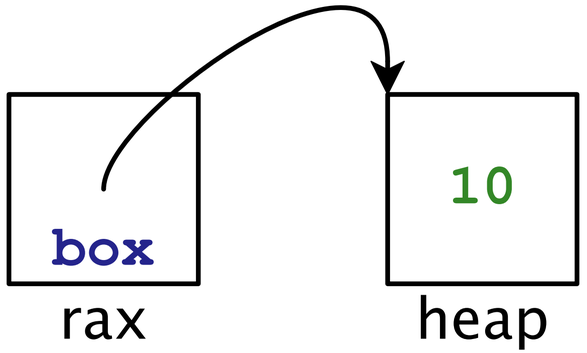
Move heap pointer into rbx

```
;; ClosedExpr -> Asm
(define (compile e)
  (prog (Global 'entry)
        (Label 'entry)
        ;; save callee-saved register
        (Push r15)
        (Push rbx)
        ;; recv heap pointer
        (Mov rbx rdi)
        (compile-e e '())
        ;; restore callee-save register
        (Pop rbx)
        (Pop r15)
        (Ret)
        #;...))
```

Creating a box

A box is a tagged pointer to one value in memory

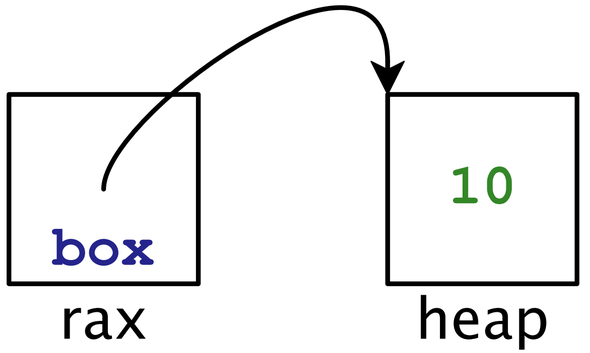
```
> (compile-op1 'box)
(list
 (Mov (Mem 'rbx) 'rax)
 (Mov 'rax 'rbx)
 (Xor 'rax 1)
 (Add 'rbx 8))
> (exec (parse '(box 10)))
'#&10
```



Accessing a box

A box is a tagged pointer to one value in memory

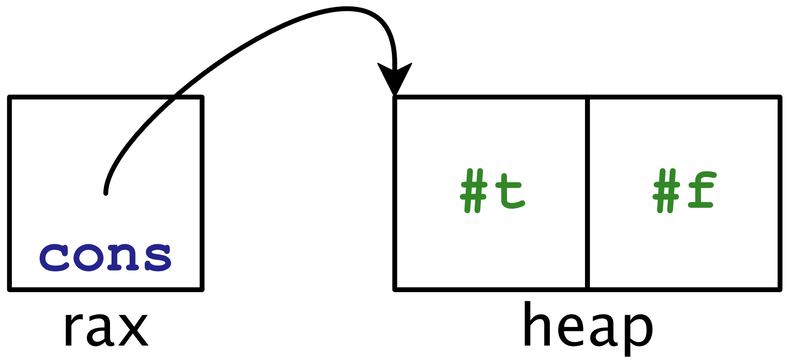
```
> (compile-op1 'unbox)
(list
 (Push 'rax)
 (And 'rax 7)
 (Cmp 'rax 1)
 (Pop 'rax)
 (Jne 'err)
 (Mov 'rax (Mem 'rax -1)))
```



Creating a pair

A pair is a tagged pointer to two values in memory

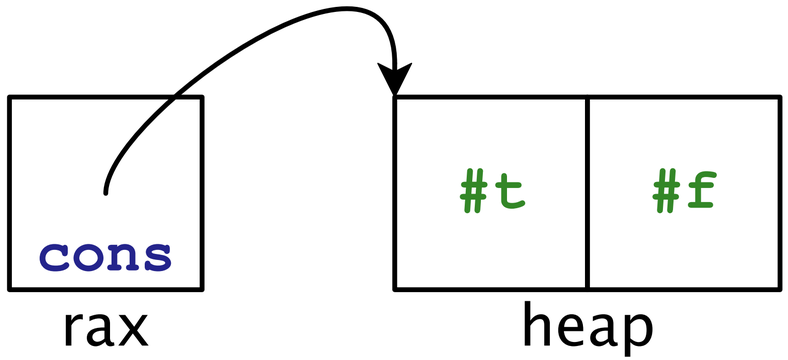
```
> (compile-op2 'cons)
(list
 (Mov (Mem 'rbx 8) 'rax)
 (Pop 'rax)
 (Mov (Mem 'rbx 0) 'rax)
 (Mov 'rax 'rbx)
 (Xor 'rax 2)
 (Add 'rbx 16))
> (exec (parse '(cons #t #f)))
'(#t . #f)
```



Accessing a pair

A pair is a tagged pointer to two values in memory

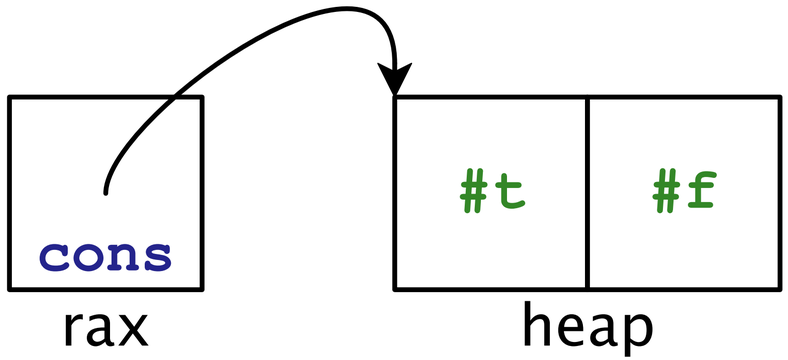
```
> (compile-op1 'car)
(list
 (Push 'rax)
 (And 'rax 7)
 (Cmp 'rax 2)
 (Pop 'rax)
 (Jne 'err)
 (Mov 'rax (Mem 'rax -2)))
```



Accessing a pair

A pair is a tagged pointer to two values in memory

```
> (compile-op1 'cdr)
(list
 (Push 'rax)
 (And 'rax 7)
 (Cmp 'rax 2)
 (Pop 'rax)
 (Jne 'err)
 (Mov 'rax (Mem 'rax 6)))
```



What about the empty list?

The empty list is just an immediate with a unique bit pattern

```
> (compile-datum '())
(list (Mov 'rax 152))
```

Heap support from standalone run-time

Run-time allocates heap

Passes pointer as argument

Frees heap after return

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

/* in words */
#define heap_size 10000

int main(int argc, char **argv)
{
  val_t *heap = malloc(8 * heap_size);
  if (!heap) {
    fprintf(stderr, "out of memory\n");
    return 1;
  }

  val_t result = entry(heap);

  print_result(result);
  if (val_typeof(result) != T_VOID)
    putchar('\n');

  free(heap);
  return 0;
}
```

Heap support from hosted run-time

Run-time allocates heap

Installs pointer in rdi register

```
(define heap
  (malloc 10000 'raw))

(define (asm-interp/host asm)
  (parameterize ([current-externs ...])
    (let ((l (gensym)))
      (asm-interp
       (prog (Global l)
             (Label l)
             (Mov rdi (cast heap _pointer _int64))
             (Jmp 'entry)
             asm)))))
```

Decoding bits

Decoding is now recursive

```
;; Integer -> Value
(define (bits->value b)
  (cond [(= b (value->bits #t)) #t]
        [(= b (value->bits #f)) #f]
        [(= b (value->bits eof))  eof]
        [(= b (value->bits (void))) (void)]
        [(= b (value->bits '())) '()]
        [(int-bits? b)
         (arithmetic-shift b (- int-shift))]
        [(char-bits? b)
         (integer->char (arithmetic-shift b (- char-shift)))]
        [(box-bits? b)
         (box (bits->value (mem-ref (- b type-box))))]
        [(cons-bits? b)
         (cons (bits->value (mem-ref (+ 0 (- b type-cons))))
               (bits->value (mem-ref (+ 8 (- b type-cons)))))]
        [else (error "invalid bits")]))
```

Printing bits

Printing is now recursive

```c
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_CHAR:
    print_char(val_unwrap_char(x));
    break;
  case T_EOF:
    printf("#<eof>");
    break;
  case T_VOID:
    break;
  case T_EMPTY:
  case T_BOX:
  case T_CONS:
    printf("'");
    print_result_interior(x);
    break;
  case T_INVALID:
    printf("internal error");
  }
}
```

```c
void print_result_interior(val_t x)
{
  switch (val_typeof(x)) {
  case T_EMPTY:
    printf("()");
    break;
  case T_BOX:
    printf("#&");
    print_result_interior(val_unwrap_box(x)->val);
    break;
  case T_CONS:
    printf("(");
    print_cons(val_unwrap_cons(x));
    printf(")");
    break;
  default:
    print_result(x);
  }
}
```

```c
void print_cons(val_cons_t *cons)
{
  print_result_interior(cons->fst);

  switch (val_typeof(cons->snd)) {
  case T_EMPTY:
    // nothing
    break;
  case T_CONS:
    printf(" ");
    print_cons(val_unwrap_cons(cons->snd));
    break;
  default:
    printf(" . ");
    print_result_interior(cons->snd);
    break;
  }
}
```