Freeing memory is a pain

- Need to decide on a protocol (who frees what?)
- Pollutes interfaces
- Errors hard to track down

Freeing memory is a pain

- Need to decide on a protocol (who frees what?)
- Pollutes interfaces
- Errors hard to track down
- ... but lets try an example anyway (fire isn't hot until it burns ***me***)

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (if (cons? (first l))
         (remove-pairs (rest l))
         (cons (first l)
               (remove-pairs (rest l))))]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (begin
       (free! (first l))
       (if (cons? (first l))
           (remove-pairs (rest l))
           (cons (first l)
                 (remove-pairs (rest l)))))]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (begin
       (free! (first l))        ; frees too much
       (if (cons? (first l))
           (remove-pairs (rest l))
           (cons (first l)
                 (remove-pairs (rest l))))))]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (begin
       (free! (first l))        ; frees too much
       (if (cons? (first l))    ;  .. and too little
           (remove-pairs (rest l))
           (cons (first l)
                 (remove-pairs (rest l))))))]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (begin
       (free! l)
       (if (cons? (first l))
           (begin (free! (first l))
                  (remove-pairs (rest l)))
           (cons (first l)
                 (remove-pairs (rest l)))))]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (begin
       (free! l)   ; frees too soon
       (if (cons? (first l))
           (begin (free! (first l))
                  (remove-pairs (rest l)))
           (cons (first l)
                 (remove-pairs (rest l)))))]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (let ([ans
            (if (cons? (first l))
                (begin (free! (first l))
                       (remove-pairs (rest l)))
                (cons (first l)
                      (remove-pairs (rest l))))])
       (free! l)
       ans)]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (cond
    [(empty? l) '()]
    [else
     (let ([ans
            (if (cons? (first l))
                (begin (free! (first l))
                       (remove-pairs (rest l)))
                (cons (first l)
                      (remove-pairs (rest l))))])
       (free! l) ; broke tail recursion!
       ans)]))
```

Freeing memory is a pain

```
; (listof any) -> (listof any[no-cons])
(define (remove-pairs l)
  (begin0 (remove-pairs/real l)
          (cleanup l)))

(define (cleanup l)
  (cond [(empty? l) (void)]
        [else
         (when (cons? (first l)) (free! (first l)))
         (let ([next (rest l)])
           (free! l)
           (cleanup next))]))

; the original function
(define (remove-pairs/real l) ...)
```

Automatic storage management

The PL has its own implementation of allocation; why not freeing too?

When can we free an object?

- When we can guarantee that it won't be used again in the compuation

Automatic storage management

The PL has its own implementation of allocation; why not freeing too?

When can we free an object?

- When we can guarantee that it won't be used again in the compuation
- ... when it isn't reachable; this is garbage collection

Garbage Collection

Garbage collection: a way to know whether a record is *live* i.e., accessible

Garbage Collection

Garbage collection: a way to know whether a record is *live* i.e., accessible

- Values on the stack & in registers are live (the roots)
- A record referenced by a live record is also live
- A program can only possibly use live records, because there is no way to get to other records

Garbage Collection

Garbage collection: a way to know whether a record is *live* i.e., accessible

- Values on the stack & in registers are live (the roots)
- A record referenced by a live record is also live
- A program can only possibly use live records, because there is no way to get to other records

- A garbage collector frees all records that are not live
- Allocate until we run out of memory, then run a garbage collector to get more space

Time to write a garbage collector

- Two new languages: **`#lang plai/gc2/mutator`** for writing progams to be collected, and **`#lang plai/gc2/collector`** for writing collectors
- Collector interface, see section 2 of the PLAI docs (search for **`init-alloctor`**)

Rules of the game

- All values are allocated (we have no type information!)
- Atomic values (fit in one cell in memory) include numbers (a lie), symbols (a less bad lie), booleans, and the empty list
- Compound values (require multiple cells in memory) include pairs and closures

A non-collecting collector

- Put the allocation pointer at address 0
- Allocate all constants in the heap, tag them with `'flat`
- Allocate all pairs in the heap, tag them with `'pair`
- Allocate all closures in the heap, tag them with `'proc`

A non-collecting collector

```
#lang plai/gc2/collector

(define (init-allocator)
  (heap-set! 0 1))

(define (alloc n)
  (define addr (heap-ref 0))
  (unless (<= (+ addr n) (heap-size))
    (error 'allocator "out of memory"))
  (heap-set! 0 (+ addr n))
  addr)
```

A non-collecting collector, cotd

```
(define (gc:flat? addr)
  (equal? (heap-ref addr) 'flat))

(define (gc:alloc-flat x)
  (define addr (alloc 2))
  (heap-set! addr 'flat)
  (heap-set! (+ addr 1) x)
  addr)

(define (gc:deref addr)
  (if (gc:flat? addr)
      (heap-ref (+ addr 1))
      (error 'gc:deref
             "expected a flat value, got addr ~s"
             addr)))
```

A non-collecting collector, cotd

```
(define (gc:cons hd tl)
  (define addr (alloc 3))
  (heap-set! addr 'cons)
  (heap-set! (+ addr 1) (read-root hd))
  (heap-set! (+ addr 2) (read-root tl))
  addr)

(define (gc:cons? pr)
  (equal? (heap-ref pr)
          'cons))
```

A non-collecting collector, cotd

```
(define (gc:first addr)
  (chk-pair addr 'gc:first)
  (heap-ref (+ addr 1)))

(define (gc:rest addr)
  (chk-pair addr 'gc:rest)
  (heap-ref (+ addr 2)))

(define (chk-pair pr who)
  (unless (gc:cons? pr)
    (error who
           "expected a pair, got addr ~a"
           pr)))
```

A non-collecting collector, cotd

```
(define (gc:set-first! addr nv)
  (chk-pair addr 'gc:set-first!)
  (heap-set! (+ addr 1) nv))

(define (gc:set-rest! addr nv)
  (chk-pair addr 'gc:set-rest!)
  (heap-set! (+ addr 2) nv))
```

A non-collecting collector, cotd

```
(define (gc:closure code-ptr free-vars)
  (define fv-count (length free-vars))
  (define addr (alloc (+ fv-count 2)))
  (heap-set! addr 'proc)
  (heap-set! (+ addr 1) code-ptr)
  (for ([i (in-naturals)]
        [r (in-list free-vars)])
    (heap-set! (+ addr 2 i)
               (read-root r)))
  addr)
```

A non-collecting collector, cotd

```
(define (gc:closure? loc)
  (equal? (heap-ref loc) 'proc))
(define (ensure-closure loc)
  (unless (gc:closure? loc)
    (error 'gc
           "expected a closure @ loc ~a"
           loc)))
(define (gc:closure-code-ptr loc)
  (ensure-closure loc)
  (heap-ref (+ loc 1)))
(define (gc:closure-env-ref loc i)
  (ensure-closure loc)
  (heap-ref (+ loc 2 i)))
```

Testing a collector

We can use **`with-heap`** to test a collector. The expression

```
(with-heap h-expr body-exprs ...)
```

expects **`h-expr`** to evaluate to a vector and then it uses that vector as the memory that **`heap-ref`** and **`heap-set!`** refer to while evaluating the **`body-exprs`**.

Testing our non-collecting collector

```
(let ([h (vector 'x 'x 'x 'x 'x)])
  (test (with-heap h
                   (init-allocator)
                   (gc:alloc-flat #f)
                   h)
        (vector 3 'flat #f 'x 'x)))
```

Testing our non-collecting collector

```
(define (make-simple-root n)
  (define b (box n))
  (make-root 'simple-root
             (λ () (unbox b))
             (λ (n) (set-box! b n))))

(let ([h (vector 'x 'x 'x 'x 'x 'x 'x 'x)])
  (test (with-heap
         h
         (init-allocator)
         (gc:cons
          (make-simple-root (gc:alloc-flat #f))
          (make-simple-root (gc:alloc-flat #t)))
         h)
        (vector 8 'flat #f 'flat #t 'cons 1 3 'x)))
; (Of course, this is not enough testing.)
```

We can also use random testing to generate mutators.

A `plai` library generates code that makes interesting heap structures (randomly), and then makes up a traversal of them.

The next three slides give three example random mutators and the calls into the library that generated them.

Random mutators

```
#lang racket
(require plai/random-mutator)
(save-random-mutator "tmp.rkt" "mygc.rkt" #:gc2? #t)
```

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200)
(define (build-one)
  (let* ((x0 'x)
         (x1 (cons #f #f))
         (x2 (cons x1 #f))
         (x3
          (lambda (x)
            (if (= x 0)
              x0
              (if (= x 1)
                x2
                (if (= x 2) x2 (if (= x 3) x2 (if (= x 4) x1 x2)))))))
         (x4
          (lambda (x)
            (if (= x 0)
              x1
              (if (= x 1)
                x0
                (if (= x 2)
                  x1
                  (if (= x 3)
                    x2
                    (if (= x 4)
                      x1
                      (if (= x 5)
                        x2
                        (if (= x 6)
                          x3
                          (if (= x 7) x0 (if (= x 8) x0 x3)))))))))))
         (x5 (lambda (x) (if (= x 0) x3 x0)))
         (x6 (lambda (x) x1)))
    (set-first! x1 x4)
    (set-rest! x1 x2)
    (set-rest! x2 x5)
    x4))
(define (traverse-one x4)
  (symbol=? 'x ((rest ((first ((first ((first (x4 4)) 0)) 0)) 3)) 1)))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (cons n n) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 200)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 200)
```

Random mutators

```
#lang racket
(require plai/random-mutator)
(save-random-mutator "tmp.rkt" "mygc.rkt" #:gc2? #t)
```

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200)
(define (build-one) (let* ((x0 0)) x0))
(define (traverse-one x0) (= 0 x0))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (cons n n) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 200)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 200)
```

Random mutators

```
#lang racket
(require plai/random-mutator)
(save-random-mutator "tmp.rkt" "mygc.rkt" #:gc2? #t)
```

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200)
(define (build-one)
  (let* ((x0 empty)
         (x1
          (lambda (x)
            (if (= x 0)
              x0
              (if (= x 1)
                x0
                (if (= x 2)
                  x0
                  (if (= x 3)
                    x0
                    (if (= x 4)
                      x0
                      (if (= x 5)
                        x0
                        (if (= x 6) x0 (if (= x 7) x0 x0)))))))))))
    x1))
(define (traverse-one x1) (empty? (x1 0)))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (cons n n) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 200)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 200)
```