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 planet plai/plai:1:14/mutator` for writing progams to be collected, and
  `#lang planet plai/plai:1:14/collector` for writing collectors
- Collector interface, see section 2.2 of the PLAI docs (search for **`init-alloctor`**)

Rules of the game

- All values are allocated (we have no type information!)
- Atomic values include numbers (a lie), symbols (a less bad lie), booleans, and the empty list
- Compound values include pairs (we do ourselves) and closures (we get help)

A non-collecting collector

```
#lang planet plai/plai:1:14/collector

(define (init-allocator) (void))

(define alloc-ptr (box 0))

(define (alloc n)
  (let ([res (unbox alloc-ptr)])
    (set-box! alloc-ptr (+ (unbox alloc-ptr) n))
    (unless (< (unbox alloc-ptr) (heap-size))
      (error 'no-collection-collector
             "out of memory"))
    res))
```

A non-collecting collector, cotd

```
(define (gc:alloc-flat fv)
  (let ([ptr (alloc 2)])
    (heap-set! ptr 'flat)
    (heap-set! (+ ptr 1) fv)
    ptr))

(define (gc:cons hd tl)
  (let ([ptr (alloc 3)])
    (heap-set! ptr 'pair)
    (heap-set! (+ ptr 1) hd)
    (heap-set! (+ ptr 2) tl)
    ptr))
```

A non-collecting collector, cotd

```
(define (gc:deref loc)
  (if (equal? (heap-ref loc) 'flat)
      (heap-ref (+ loc 1))
      (error 'gc:deref "deref non flat value")))

(define (gc:first pr-ptr)
  (if (equal? (heap-ref pr-ptr) 'pair)
      (heap-ref (+ pr-ptr 1))
      (error 'first "non pair")))

(define (gc:rest pr-ptr)
  (if (equal? (heap-ref pr-ptr) 'pair)
      (heap-ref (+ pr-ptr 2))
      (error 'first "non pair")))
```

A non-collecting collector, cotd

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

(define (gc:cons? loc)
  (equal? (heap-ref loc) 'pair))

(define (gc:set-first! pr-ptr new)
  (error 'set-first! "unsupported"))

(define (gc:set-rest! pr-ptr new)
  (error 'set-rest! "unsupported"))
```

Allocator with small constants & alloc-ptr in memory

```
#lang planet plai/plai:1:14/collector

(define (init-allocator)
  (heap-set! 0 10)
  (heap-set! 1 '())
  (heap-set! 2 #f)
  (heap-set! 3 #t)
  (for ([i (in-range 0 6)])
    (heap-set! (+ i 4) i))
  (for ([i (in-range 10 (heap-size))])
    (heap-set! i 'free)))

(test (let ([v (make-vector 12 'junk)])
        (with-heap v (init-allocator))
        v)
      (vector 10 '() #f #t 0 1 2 3 4 5 'free 'free))
```

Allocator with small constants & alloc-ptr in memory

```
(define (imm-loc? loc) (<= loc 9))
(define (imm-loc->value loc)
  (case loc
    [(0) (error 'allocator "oops")]
    [(1) '()]
    [(2) #f]
    [(3) #t]
    [else (- loc 4)]))

(test (imm-loc->value 4) 0)
(test (imm-loc? 9) #t)
(test (imm-loc? 10) #f)
```

Allocator with small constants & alloc-ptr in memory

```
(define (imm-val? fv)
  (or (and (exact-integer? fv)
           (<= 0 fv 5))
      (and (member fv (list '() #t #f))
           #t)))
(test (imm-val? 10) #f)
(test (imm-val? 5) #t)
(test (imm-val? #f) #t)

(define (imm-value->loc fv)
  (case fv
    [(()) 1]
    [(#f) 2]
    [(#t) 3]
    [else (+ fv 4)]))
```

Allocator with small constants & alloc-ptr in memory

```
(test (imm-loc->value (imm-value->loc 0)) 0)
(test (imm-loc->value (imm-value->loc 1)) 1)
(test (imm-loc->value (imm-value->loc 2)) 2)
(test (imm-loc->value (imm-value->loc 3)) 3)
(test (imm-loc->value (imm-value->loc 4)) 4)
(test (imm-loc->value (imm-value->loc 5)) 5)
(test (imm-loc->value (imm-value->loc #t)) #t)
(test (imm-loc->value (imm-value->loc #f)) #f)
(test (imm-loc->value (imm-value->loc '())) '())
```

Allocator with small constants & alloc-ptr in memory

```
(define (alloc size)
  (let ([ptr (heap-ref 0)])
    (unless (<= (+ ptr size) (heap-size))
      (error 'alloc "out of space!"))
    (heap-set! 0 (+ ptr size))
    ptr))
(test (with-heap (vector 1 'x 'x 'x) (alloc 2))
      1)
(test (with-heap (vector 2 'x 'x 'x 'x) (alloc 2))
      2)
(test (let ([v (vector 2 'x 'x 'x 'x)])
        (with-heap v (alloc 2))
        v)
      (vector 4 'x 'x 'x 'x))
(test/exn (with-heap (vector 2 'x 'x 'x)
                     (alloc 5))
          "out of space!")
```

Allocator with small constants & alloc-ptr in memory

```
(define (gc:alloc-flat fv)
  (cond
    [(imm-val? fv)
     (imm-value->loc fv)]
    [else
     (let ([ptr (alloc 2)])
       (heap-set! ptr  'flat)
       (heap-set! (+ ptr 1) fv)
       ptr)]))
```

Allocator with small constants & alloc-ptr in memory

```
(test (with-heap (make-vector 12 'junk)
                 (init-allocator)
                 (gc:alloc-flat 1))
      5)
(test (with-heap (make-vector 12 'junk)
                 (init-allocator)
                 (gc:alloc-flat 11))
      10)
(test (let ([v (make-vector 12 'junk)])
        (with-heap v
                   (init-allocator)
                   (gc:alloc-flat 1))
        v)
      (vector 10 '() #f #t 0 1 2 3 4 5 'free 'free))
```

Allocator with small constants & alloc-ptr in memory

```
(define (gc:cons hd tl)
  (let ([ptr (alloc 3)])
    (heap-set! ptr 'cons)
    (heap-set! (+ ptr 1) hd)
    (heap-set! (+ ptr 2) tl)
    ptr))
```

Allocator with small constants & alloc-ptr in memory

```
(test (let ([v (make-vector 12 'junk)])
        (with-heap v
                   (init-allocator)
                   (gc:alloc-flat 11))
        v)
      (vector 12 '() #f #t 0 1 2 3 4 5 'flat 11))
(test (let ([v (make-vector 13 'junk)])
        (with-heap v
                   (init-allocator)
                   (gc:cons 2 3))
        v)
      (vector 13 '() #f #t 0 1 2 3 4 5 'cons 2 3))
(test (with-heap (make-vector 13 'junk)
                 (init-allocator)
                 (gc:cons 2 3))
      10)
```

Allocator with small constants & alloc-ptr in memory

```
(define (gc:first pr-ptr)
  (heap-ref (+ pr-ptr 1)))
(test (with-heap (vector 12 '() #f #t 0 1 2 3 4 5
                         'flat 11 'cons 1 2 'flat 12)
                 (gc:first 12))
      1)

(define (gc:rest pr-ptr)
  (heap-ref (+ pr-ptr 2)))
(test (with-heap (vector 12 '() #f #t 0 1 2 3 4 5
                         'flat 11 'cons 1 2 'flat 12)
                 (gc:rest 12))
      2)
```

Allocator with small constants & alloc-ptr in memory

```
(define (gc:flat? loc)
  (cond
    [(imm-loc? loc) #t]
    [else
     (equal? (heap-ref loc) 'flat)]))

(test (gc:flat? 1) #t)
(test (with-heap (vector 12 '() #f #t 0 1 2 3 4 5
                         'flat 11 'cons 1 2 'flat 12)
                 (gc:flat? 12))
      #f)
(test (with-heap (vector 12 '() #f #t 0 1 2 3 4 5
                         'flat 11 'cons 1 2 'flat 12)
                 (gc:flat? 10))
      #t)
```

Allocator with small constants & alloc-ptr in memory

```
(define (gc:cons? loc)
  (cond
    [(imm-loc? loc) #f]
    [else
     (equal? (heap-ref loc) 'cons)]))

(test (gc:cons? 1) #f)
(test (with-heap (vector 12 '() #f #t 0 1 2 3 4 5
                         'flat 11 'cons 1 2 'flat 12)
                 (gc:cons? 12))
      #t)
(test (with-heap (vector 12 '() #f #t 0 1 2 3 4 5
                         'flat 11 'cons 1 2 'flat 12)
                 (gc:cons? 10))
      #f)
```

Allocator with small constants & alloc-ptr in memory

```
(define (gc:set-first! pr-ptr new)
  (heap-set! (+ pr-ptr 1) new))
(test (let ([v (vector 12 '() #f #t 0 1 2 3 4 5
                       'cons 1 2)])
        (with-heap v (gc:set-first! 10 10))
        v)
      (vector 12 '() #f #t 0 1 2 3 4 5 'cons 10 2))

(define (gc:set-rest! pr-ptr new)
  (heap-set! (+ pr-ptr 2) new))
(test (let ([v (vector 12 '() #f #t 0 1 2 3 4 5
                       'cons 1 2)])
        (with-heap v (gc:set-rest! 10 10))
        v)
      (vector 12 '() #f #t 0 1 2 3 4 5 'cons 1 10))
```