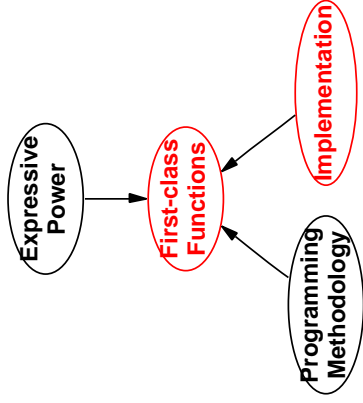


## Implementing functions



## Memory Management

In the beginning, there was FORTRAN:

- no dynamically allocated memory
- everything in static arrays with fixed limits
- a significant style as recently as 1983 (TEX)
- still important for hard real-time systems

Dynamic allocation, explicit memory management:

- Pascal (new/dispose)
- C (malloc/free)
- C++ (new/delete)

Still an area for some research:

- efficiency
- locality (important for memory hierarchy)
- avoiding fragmentation

...many schemes

(Knuth, vol. 1; also Grunwald & Zorn)

## The terror of free

When do you call free?

- must call sometime  
malloc without free leads to "memory leak"
- premature call to free ⇒ dangling reference  
hello, core dump!

The things we do....

- forget about free, just pray
- crufty interfaces — who must call free?
- two ways to call every procedure?
- return pointers to static memory? **Ouch!**
- make copies of things  
I own my copy, you own yours

We can do better...

## Automatic Memory Management

Why not have malloc without free?

let memory be freed automatically  
why not call it cons

Can be required in the language definition:

*No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation.*

Plan of study:

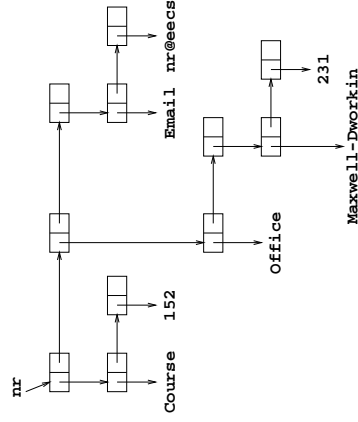
- see how garbage is created
- see how we identify it
- discuss ways of reclaiming it

## Garbage Creation

Association list for instructor

```

(set nr '((Course 152)
        (Office (Maxwell-Dworkin 231))
        (Email nr@eecs)))
  
```



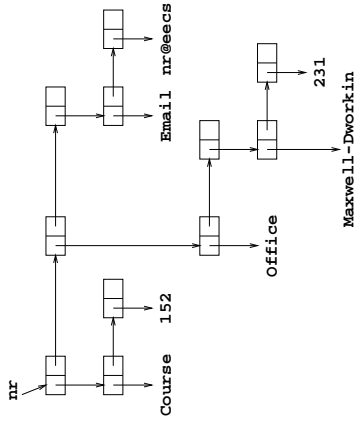
## Executions create garbage

```

(define bind (x y alist)
  (if (null? alist)
      (list1 (list2 x y))
      (if (equal? x (caar alist))
          (cons (list2 x y) (cdr alist))
          (cons (car alist)
                 (bind x y (cdr alist))))))
  
```

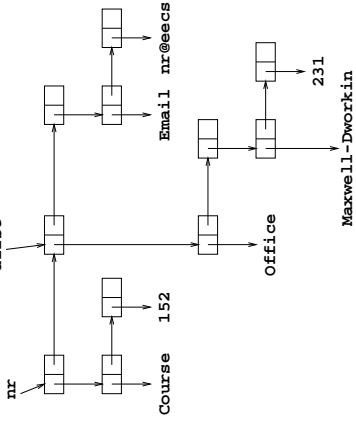
To change office from list to symbol:

```
(set nr (bind 'Office
'Maxwell-Dworkin-231
nr))
```



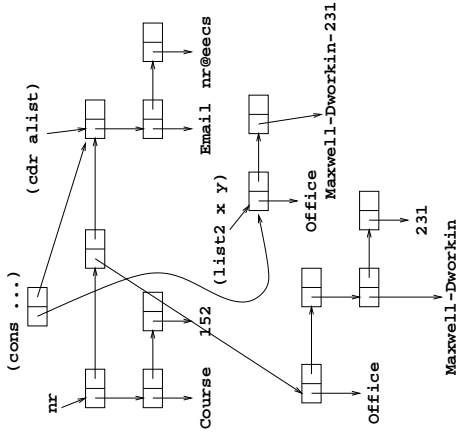
First recursive call:

```
(bind x y (cdr alist))
```



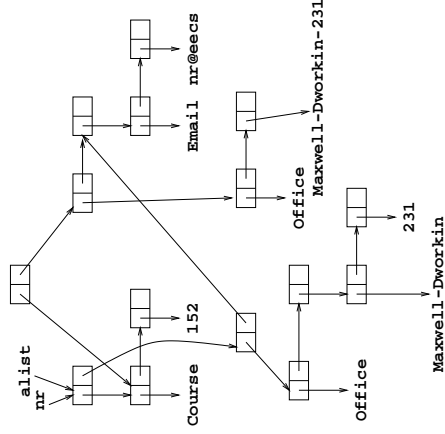
Comparison succeeds:

```
(cons (list2 x y) (cdr alist))
```

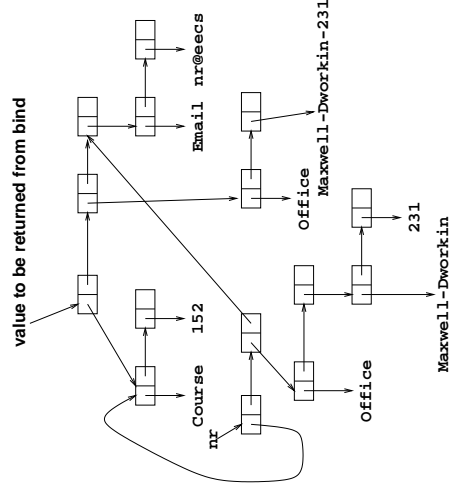


Now return, back to first call:

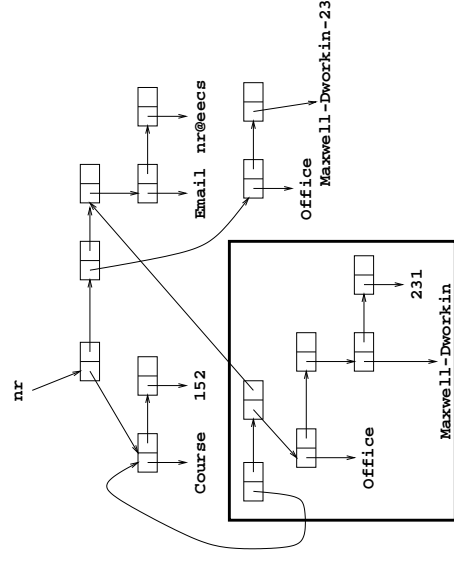
```
(cons (car alist) (bind ...))
```



Return from bind:  
Just before set, everything is live



After set, outlined cells are garbage



## What is garbage?

Objects have finite lifetime

We can allocate forever because garbage can be reused

A cell is garbage if

- its contents can't affect any future (legal) computation

As an approximation, we call a cell garbage if

- it is not *reachable* from a *root set*

Any cell that is not garbage is *live data*

- live data might affect a future computation

What values can affect a computation?

We divide these values into two parts.

Roots:

- local variables & parameters of procedures
- global variables

Other values reachable from the roots:

- what if a formal parameter is a cons cell?

## Identifying roots

Unless you planned ahead, finding roots is nontrivial

- we have done for you on homework

Roots in compiled code:

- global variables
- local variables on the call stack
- machine registers

Collector has to be told where to look

(but wait for “conservative” collection)

Roots in a bytecode interpreter

- part of the definition of the “virtual machine”

Roots in an AST-based interpreter

- any variable in any active C function that could lead to `Value` (ignore functions that don't allocate)
- global variables that could lead to `Values` (none here)

## Roots, continued

Bytecode approach simple and popular:

Icon, Java, LISP, Moscow ML, Perl, Python, Scheme, Smalltalk, ...

For compilation, compiler tells all

New Jersey ML compiler especially simple: no globals, no stack! just some machine registers

If compiler doesn't tell, we have

“collection with ambiguous roots”  
“collection in an uncooperative environment”  
“conservative collection”

Good for C, C++

## Tracing pointers

Confusing pointer & integer could cause leaks

- suppose integer `n` is address of an object?

Two major approaches

- Type tagging: each type of heap object gets unique tag tag leads to descriptive info compiler puts “which fields are pointers” in desc

Example: Modula-3

- Pointer tagging: use special bits to distinguish pointers, integers

“LISP machines” had special “tag bits”

on stock hardware, loses a bit from integer

space

e.g., all integers have low bit set

heap object descriptors only show size

some objects are ‘non-pointer-containing’

e.g., strings

Example: Standard ML of New Jersey

## Two families of garbage collectors

Using “start at roots, follow pointers” approach:

- **mark-and-sweep** collection
  1. unmark all objects
  2. trace pointers, marking all live data
  3. “sweep away” unmarked objects unmarked objects moved to “free list”, reallocated
- **copying** collection: works with two “semi-spaces” all objects, some free space in “from-space” “to-space” is empty from-space full? **copy live data to to-space** to-space is *not* full (not all objects copied) now objects, free space are in to-space “flip”

## The managed heap as an abstraction

Program doing work is the **mutator**

Contract with mutator includes

- Agreement on how to find roots & trace pointers
- Mutator gets to call `alloc` as often as desired
- Heap gets to call OS when it needs to grow

Lurking behind `alloc` is all the machinery

- allocator (used often)
- collector, including scanning roots (used occasionally)
- heap growth (used seldom)

## The evil third family

Using heavy compiler support: **reference counting**  
 if no pointer to an object, it must be garbage  
 keep in each object a "reference count"  
 number of references (ptrs) to object  
 if reference count becomes zero, put on free  
 list  
 code must adjust ref count at every  
 assignment!  
 also, can't reclaim cycles

**BUT! No pause times**

## Tricolor marking

A conceptual tool for finding live data

- white** status unknown
- gray** definitely live, children not known yet
- black** live, and children also live

Collection:

1. Initially, all objects are white
2. Color roots gray
3. While there is a gray node
  - (a) if there is a white successor, pick one and color it gray
  - (b) otherwise, color the gray node black

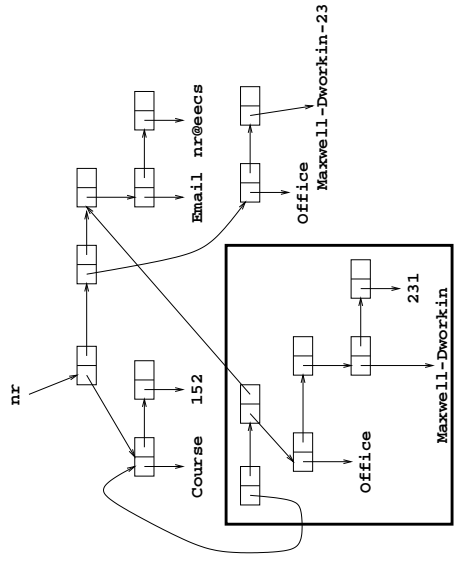
Invariant:

- black nodes point only to black and gray nodes

When no gray nodes left, white nodes are garbage

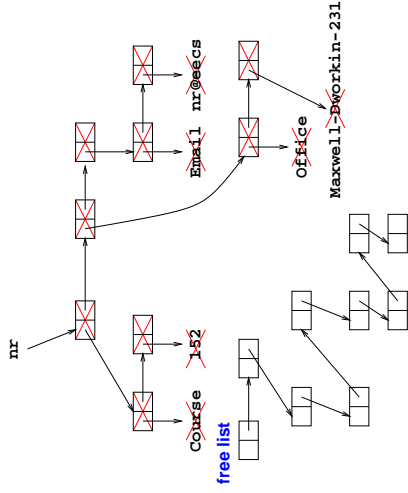
## Mark and Sweep Collection

Return to association list:



## Sweep

Sweep phase looks at every cell in the heap  
 puts unmarked cells on free list:



Store marks (X) in objects or in separate bitmap

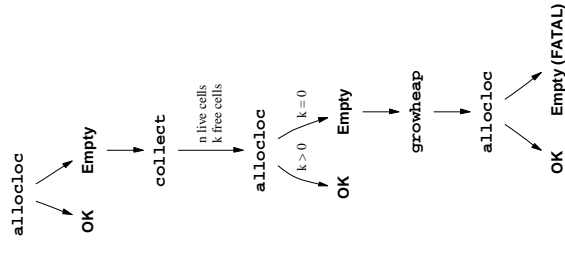
## Tracking pointers in the mark phase

While marking, must track marked nodes whose  
 children have not yet been marked (gray nodes)  
 will go back and mark those children later  
 essentially a depth-first traversal  
 depth-first walk requires a stack

Tricolor view:

- white** not yet visited
- gray** visited, children's status unknown
- black** visited, and children also visited

## Garbage collection and heap growth



## Mark-and-sweep allocator

Must always consider allocator and collector together

- can shuffle work between the two
- division of labor has strong implications for pause times

Naïve mark-and-sweep allocator takes cells from free list

```
void *alloc(int n) {
  //let fl point to a suitable free list
  if (!fl)
    collect();
  p = fl;
  fl = fl->next;
  return p;
}
```

If allocator needs to enlarge the heap?

Get more heap from the OS  
Turn it into cells  
Add all those cells to the free list

## Beating fragmentation

Good strategy called BIBOP: Big Bag of Pages

divide memory into fixed-size pages  
each page contains objects of a single type and size  
objects larger than 1 page typically get special treatment

Advantages:

can identify an object's type just by its address  
*no tag bits or descriptor word needed*  
(cons cells—33% off!)  
can put mark bits in a separate bitmap  
(less overhead, better locality)  
can bound losses to fragmentation

## Shifting work from collection to allocation

Too much work done at collection!

unmark phase proportional to total heap size  
mark phase proportional to live data  
sweep phase proportional to total heap size

large heap  $\Rightarrow$  long pause times

Work done at allocation:

constant work (take one cell off free list)

Clever ideas:

- unmark & sweep in allocator—cuts pause time
- use free-space pointer



Allocate: step free pointer to next unmarked cell  
Free list is eliminated! (saves pointer fiddling)

During one GC cycle:

collector work is proportional to live data  
total allocator work is constant per allocation, PLUS amount of live data at last collection

Most cells garbage:

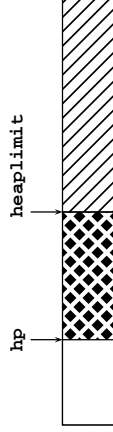
expected work per allocation is small

## Stop and copy — trading space for time

Divide memory into two contiguous semispaces

One semispace unused except for collection  
Allocate from the other

Allocation by test and increment:



```
void *alloc(int n) {
  if (hp + n > heapLimit)
    collect();
  p = hp;
  hp += n;
  return p;
}
```

Allocator almost always inlined

## Mark and sweep with objects of varying size

What if not everything in life is a cons cell?

Can't have a single free list to satisfy every request.

May have

- multiple free lists of different sizes
- free memory broken into arbitrary blocks (first fit, best fit)
- "buddy system" allocator

Just like traditional dynamic memory allocators

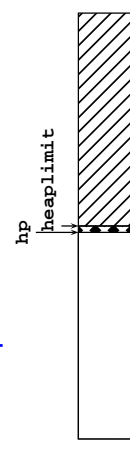
And we have the same problem: fragmentation hurts locality of reference

When we ask for an object,

we may have plenty of memory but... it may be in tiny unusable pieces between allocated bits

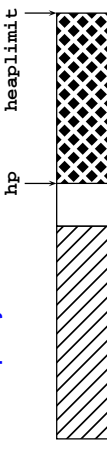
## Stop and copy — collection step

When semispace is exhausted:



Only some of the white area is live data the rest is garbage

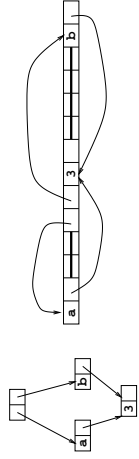
If we copy only the live data to the other semispace, we'll have plenty of room to allocate more:



Copying goes from from-space to to-space  
Change of roles (from one semi-space to the other) is a flip

## Stop and copy details

Collector must preserve sharing, cycles:



Must copy each live object exactly once.

What if there is more than one path to an object?

At first visit, replace with *forwarding pointer*

don't need extra space; overwrite object

Marks object as already copied

Shows location in to-space

All references to object must be updated to new location

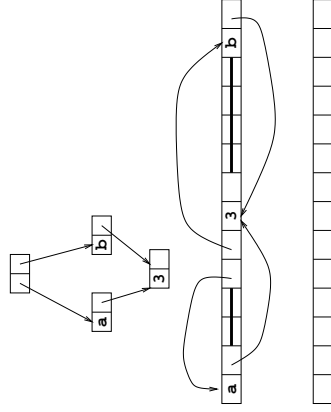
To mark "forwarded," could use 1 bit/object

but can tell just by the value of the pointer!

only forwarding pointers point into to-space

Must distinguish pointers (updated) from non-pointers (not updated)

## Stop and copy example



## Stop and copy – graph traversal

As we walk the graph of live objects, we must track objects that have been copied, but the things they point to haven't been copied.

(Compare mark-and-sweep, we had to track objects that were marked, but things they pointed to hadn't been marked.)

For mark-and-sweep, we used a stack: depth-first traversal

For copying, we use a queue: breadth-first traversal

Use to-space as the queue, at no cost!

Need only two pointers for copying collector:

hp points to next free location in to-space

scamp points to next object whose referents not copied

Invariant:

Objects before scamp point to to-space

Objects between scamp and hp point to from-space

## Stop and copy algorithm

Basic operation is to forward a pointer:

```
forward (p) =
  if *p is forwarding pointer
  then return *p
  else
  copy the object *p to location hp
  *p := hp
  hp := hp + length(*p)
  return *p
```

Copies a single object (or its forwarding pointer.)

To copy everything, start with roots,

then consume queue from scamp to hp:

```
scamp := hp := beginning of to-space
for each root r do
```

```
  r := forward(r)
```

```
  while scamp < hp do
```

```
    let L be length of *scamp
```

```
    for i := 0 to L-1 do
```

```
      if scamp[i] is a pointer then
```

```
        scamp[i] := forward(scamp[i])
```

```
    scamp := scamp + L
```

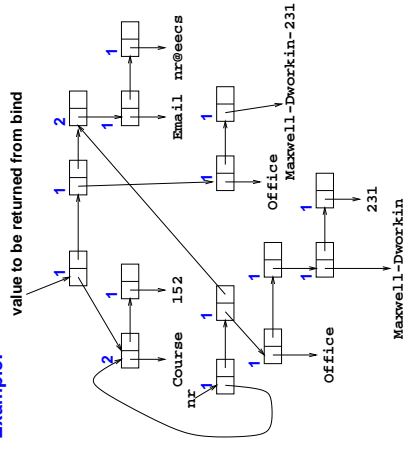
## Reference counting

Widely used technique in early days:

each object tracks number of references to it

when count goes to zero, can put object on free list

Example:



Efficiency very good in limit of large memory

(but so is mark and sweep)

Crucial advantage comes from lightning-fast allocator:

allocate by check and increment

(exactly the same cost as stack allocation)

(garbage collection can be cheaper than stack allocation)

Can reduce cost of check two ways:

VM hardware (cute, but hard to port)

check bounds register

check only at loops!

one check for many allocations

Copying step compacts all data

there is never any fragmentation of free memory!

(can do compacting mark-and-sweep, too)

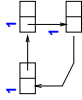
## Reference counting costs

Must change ref counts at every assignment!  
(the killer – CPU overhead can reach 20-30%)  
(various tricks used to cut down costs here)

Expensive to follow pointers when costs go to 0  
can defer to allocation, like mark-sweep

So, can get good bounds on costs  
good for real-time properties

But ref counting can't collect cyclic garbage:



Serious implementations need extra collector  
there go the real-time properties

Ref-counting can be tricky to get right

discovered by many implementors of C++ classes

Old ref-counting collectors gave GC a bad name

## Generational example: 2 generations

So, for example, 2 generations:



When free space exhausted, copy newer to reserve  
(so-called **minor collection**)

older plus survivors (x) are new older generation  
survivors are *promoted*



When older generation approaches half of memory  
copy-collect older generation (**major collection**)  
copy result to beginning of heap, start over...

Massive savings  
try to estimate (**fine exam question...**)

Copying generations easiest to understand

but, surprisingly, can do with mark and sweep

Many generations can save more

keep youngest generation entirely in cache!

## Real experience with reference counting

Flux OSKit:

Although adding COM interfaces to the OSKit  
solve many of the [linking problems], in some  
ways it worsened the usability problems... In  
practice, merely getting the reference counting  
right was a significant barrier to experimenting  
with new system configurations.

## Generational collection

Two observations about functional programs:

- newer cells tend to point to older cells  
(always true except where there is mutation)
- young cells tend to be short-lived, old cells long-lived  
(if cell kept for even 1 collection, it's probably important: will live for a while)  
(new cells often throwaway intermediate results—e.g. in simple version of rev)

Divide heap into *generations*

collect younger generations more frequently

When collecting younger generations,

roots include pointers from old objects to

young ones

such pointers can be created only by assignment!

(think about cons)

common solution: track all assignments

Works well in LISP systems

even better in ML systems!

## Conservative collection – copying

What if you don't know where roots are,  
but you do know object layouts?

“Mostly-copying” conservative collection  
(Bartlett Scheme compiler)

Anything on stack or in globals could be a pointer...  
so treat it as a pointer!

Entire stack, data, bss segments are pointers

Can't forward (change) pointers on the stack,  
so objects are “pinned” in memory

But, if you know your own objects,  
you can forward anything pointed to only by  
your own objects

BIBOP lets you identify object addresses

Conservative guess at roots may mistakenly keep  
garbage

- surprisingly little in practice
- get many advantages of regular copying collector, including little fragmentation

## Conservative collection – mark & sweep

What if you don't know object layouts?

You can still garbage-collect!

but you can't move anything,

because you can't tell pointers from integers  
anywhere

Mark-and-sweep to the rescue (Boehm/Weiser)!

Super-conservative assumptions:

Anything on stack or in globals could be a  
pointer ...

and anything on the heap could be a pointer

Treat them all as pointers

Works astonishingly well in practice

simple replacement for malloc/free yields

approximately the same CPU cost

30–150% memory overhead

a few tricks can make it even better

plus can write faster code by relying on GC

State of the art collector (Boehm et al.) widely used

Java, Cedar, libscheme, ...

dovetails nicely with C, C++ code

## Dueling memory-management schemes

Simple versions have different pros and cons

Mark-and-sweep

- fragments
  - hard to do generational collection
  - poor locality
  - easy to make conservative
- Copying
- needs big memories
  - copying large objects is expensive
  - identifying & forwarding pointers can be hard
  - easy to make generational

State-of-the-art collectors resemble one another ever more closely ...

Asymptotic behaviors are all the same.

Important practical aspects of performance are:  
constant factors  
locality of reference

## Plus Ultra

(There's more)

Memory management a hot area in research

- incremental collection
- concurrent collection
- real-time collection
- collection of persistent store
- many strategies for better performance

## Things to remember

Garbage collection makes safety possible  
huge classes of common bugs are eliminated.  
(estimated 40% of bugs in Xerox Mesa)  
(Purify works by conservative-collection techniques)

All schemes are based on  
root set  
following pointers to live data  
distinguishing pointers from non-pointers  
(always possible to some degree)

Performance even of simple schemes can be good

Sophisticated collectors outperform malloc/free  
interfaces simpler  
no copying to control ownership

Conservative collectors can support any language  
even C or C++

A highly effective use of programmer time  
use it in your next program