Lisp/Racket and Implementation

Garbage Collection
Later:
... Programs as Data
... Eval and Interpreters

Language Definition vs. Implementation

Most of 251 so far

Now a brief interlude

Ideally distinct, but definitely influence each other.

- Impossible/infeasible language features?

Some languages are *defined* by implementations:

- Abstraction?
- Can be complicated, difficult to reason about
- But high-level implementation can help understand definition.
- May over-fit to current system, introduce unintended corner cases
- Tends to happen early in language development or when the goal is to "just hack something up" instead of design a clean abstraction.

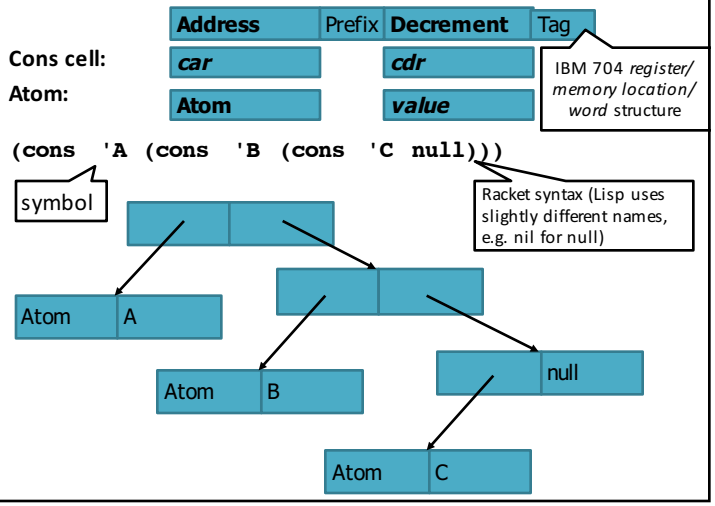
Lisp:

- Formal definitions first.
- Some practicalities of implementation crept into surface of language.
- Some "implementation details" should have been in definition.
- Definition forced new implementation features and simplified others.

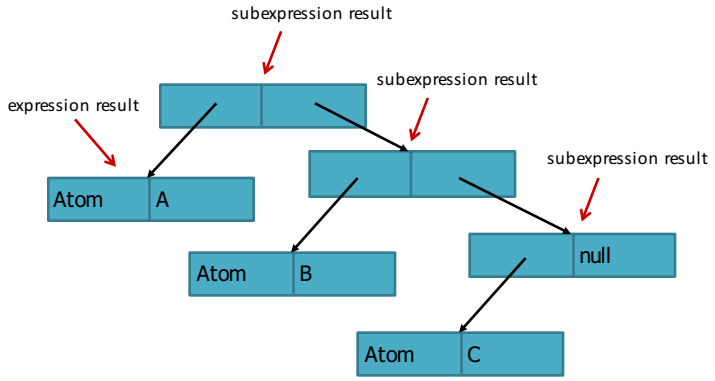
IBM 704

3

Lisp Memory Model



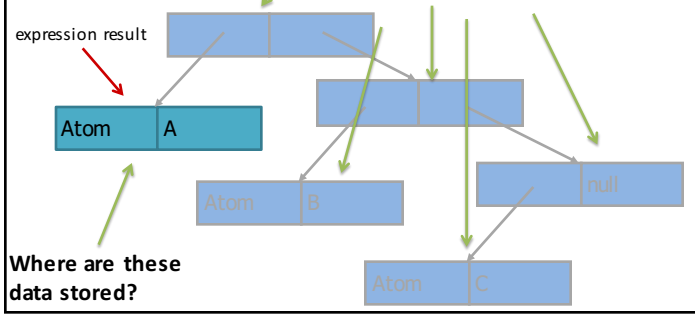
```
(car (cons 'A (cons 'B (cons 'C null))))
```



How do we remember partial result and "what to do next"?

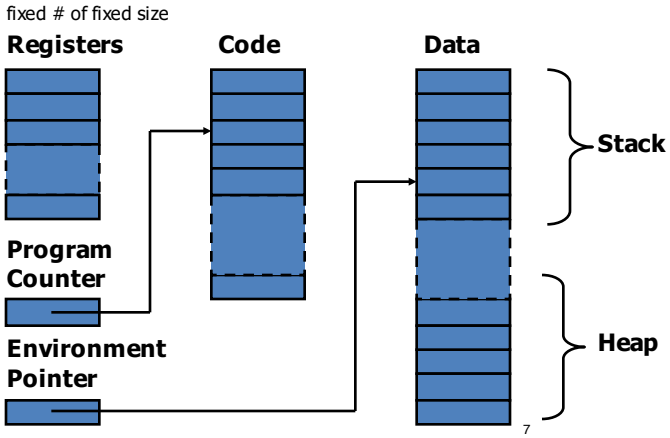
```
(car (cons 'A (cons 'B (cons 'C null))))
```

Garbage:
cells that will never be used again, but still occupy storage space.



Where are these data stored?

Simplified Machine Model



7

Storage (240 view)

NOW

Heap

- Dynamically allocated data…
- Racket: cons cells!
- …
- Racket: lots of cons cells!
- … more later for first-class functions…

revised later

Stack

- Local variables.
- Arguments, return values.
- Where to continue evaluating after current function call/ expression.
- Not the full story for first-class functions…

Garbage Collection (GC)

Every cell requires a block of the available fixed-size **heap**.

A cell is ***garbage*** once the remainder of evaluation will never access it.

Garbage collection:
Reclaim storage used for garbage cells.

- When storage full (or sooner), reuse garbage-filled space for new cells.

Required/invented to implement Lisp.

- Lisp/Racket programs tend to create new cells *very* rapidly (even vs. Java)
- No mutation => create fresh copies instead of modifying
- Cells become garbage almost as rapidly as they are created.
- Can fill up memory rapidly -- much of it is garbage.

GC: Reachability

Goal: Reclaim storage used for ***all*** garbage cells.

Reality?
```
(let ([garbage (list 1 2 3)])
  (if e (length garbage) 0)
```

Achievable goal: Reclaim storage used for all ***unreachable*** cells.

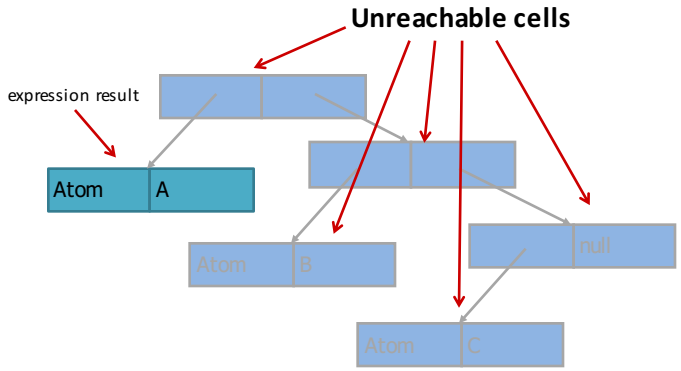
- All unreachable cells are garbage.
- Some garbage cells are reachable.

A cell is **reachable** if it is:

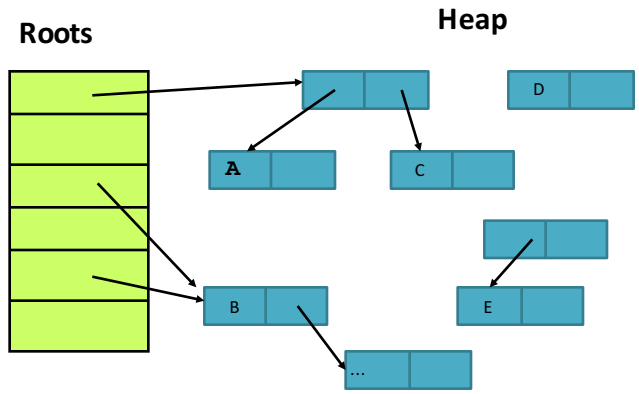
roots
- a subexpression of the expression currently being evaluated; or
- bound in the current environment; or

recursive heap cases
- bound in the environment of any reachable closure; or
- the referent of the *car* or *cdr* of any reachable cons cell.

```
(car (cons 'A (cons 'B (cons 'C null))))
```

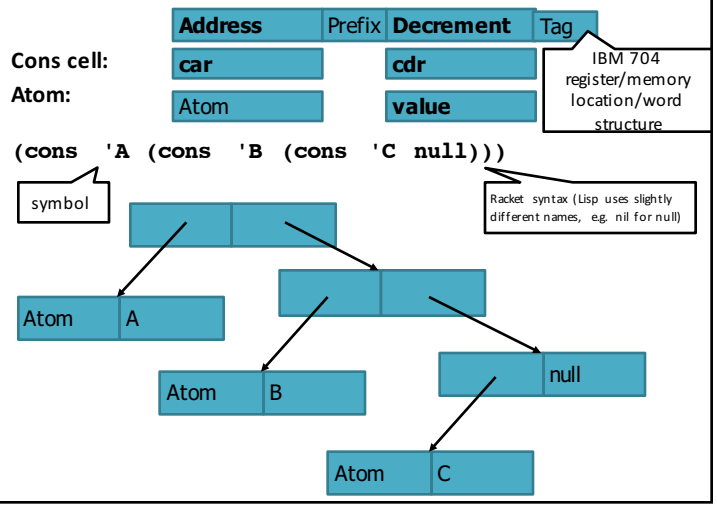


Mark-Sweep

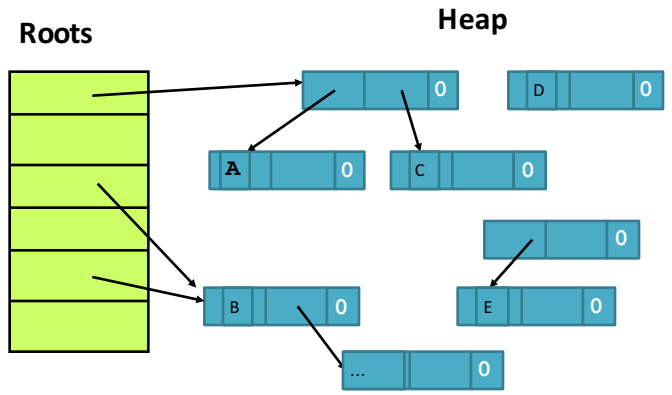


13

Lisp Memory Model

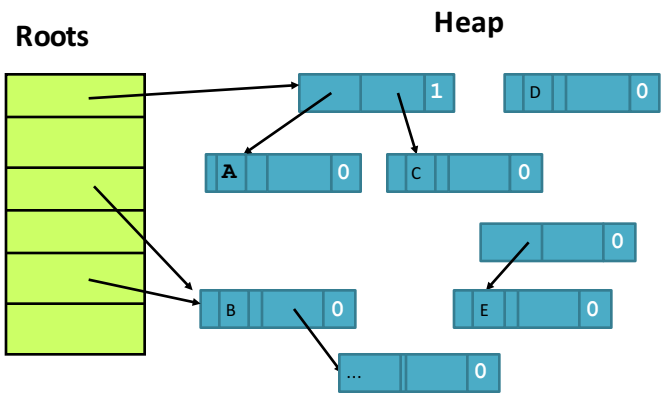


Mark-Sweep: Clear



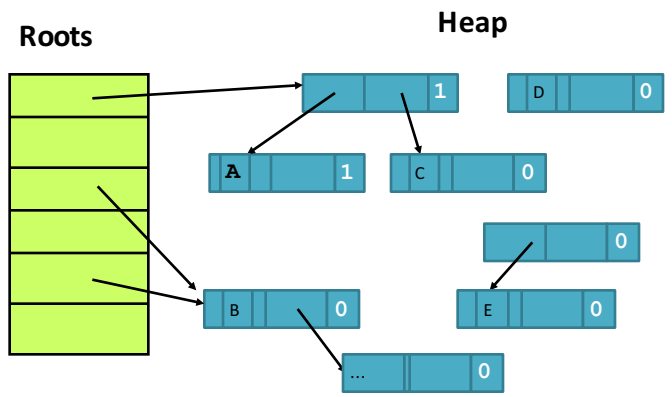
15

Mark-Sweep: Mark



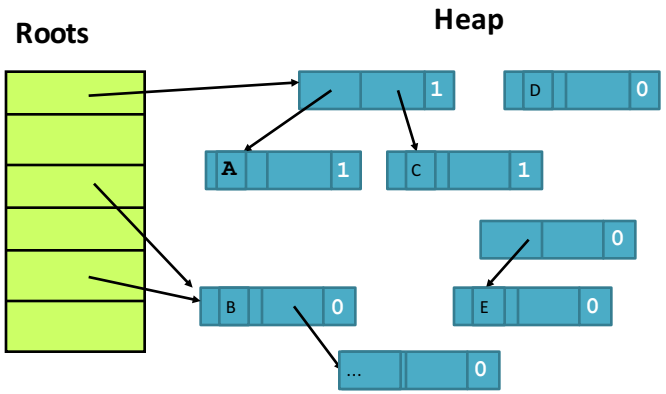
16

Mark-Sweep: Mark



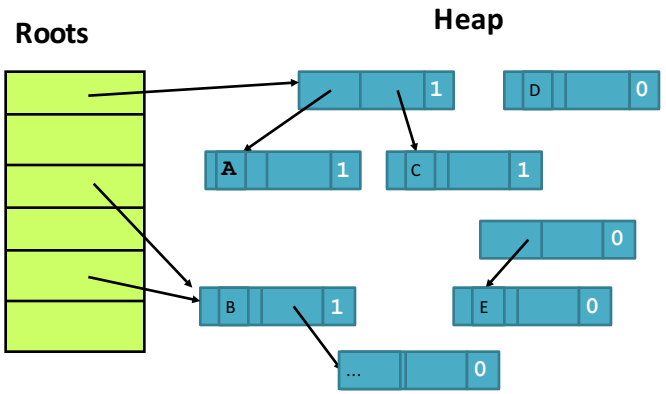
17

Mark-Sweep: Mark



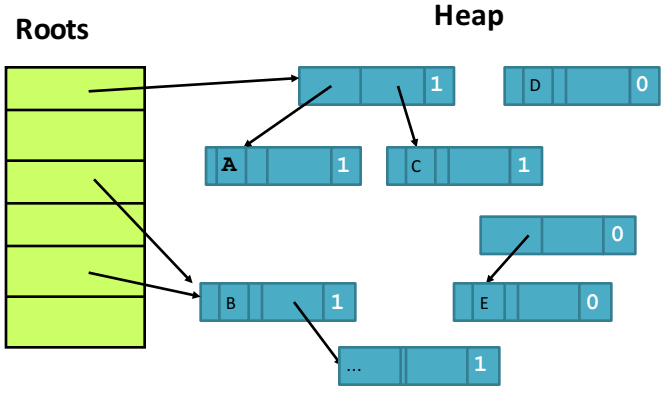
18

Mark-Sweep: Mark



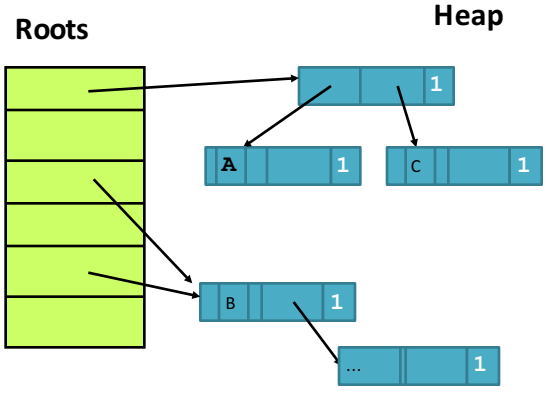
19

Mark-Sweep: Mark



20

Mark-Sweep: Sweep



21