Lisp/Racket and Implementation

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

Language Definition vs. Implementation

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.

Lisp Memory Model

<table>
<thead>
<tr>
<th>Address</th>
<th>Prefix</th>
<th>Decrement</th>
<th>Tag</th>
</tr>
</thead>
</table>
Cons cell: car  cdr  Atom  value |
Atom: symbol |
(cons 'A (cons 'B (cons 'C null)))

IBM 704

IBM 704 Memory Model

IBM 704 register/mem location/word structure
Racket syntax (Lisp uses slightly different names, e.g. nil for null)
\[(\text{car } (\text{cons } 'A (\text{cons } 'B (\text{cons } 'C \text{ null}))))\]

**Simplified Machine Model**

- **Registers**
- **Code**
- **Data**
  - Program Counter
  - Environment Pointer

**Storage (240 view)**

- **Stack**
- **Heap**
  - Dynamically allocated data...
  - Racket: cons cells!
  - ...
  - Racket: lots of cons cells!
  - ... more later for first-class functions...

**Garbage:**

- Cells that will never be used again, but still occupy storage space.

**How do we remember partial result and “what to do next”?**

Where are these data stored?
**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?**

```scheme
(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:
- a subexpression of the expression currently being evaluated; or
- bound in the current environment; or
- bound in the environment of any reachable closure; or
- the referent of the car or cdr of any reachable cons cell.

---

**Mark-Sweep**

(expression result)

Unreachable cells

Heap

Roots

A

B

C

D

E

null

...
Lisp Memory Model

Cons cell:
Atom: Atom

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

Mark-Sweep: Clear

Mark-Sweep: Mark

Mark-Sweep: Mark