# CSE 341: Programming Languages

Dan Grossman

Spring 2008

Lecture 14— Delayed Evaluation, Thunks, Streams, Memoization

# Today

- Last time: `let` vs. `let*` vs. `letrec`
- Scheme top-level: forward references and evil mutation
- Delaying evaluation: Function bodies evaluated only at application
- Key idioms of delaying evaluation
  - Conditionals
  - Laziness
  - Streams
  - Memoization
- In general, evaluation rules defined by language semantics
  - Some languages have "lazy" function application!

# Top-level definitions

Scheme top-level allows forward references and mutation of bindings

- What should a name clash do? (In fact, it's mutation.)
- How can you program defensively?
  - General point: Make a local copy!
- How does "primitives are functions" make this harder?
- What do Schemers do in practice?
  - Don't mutate top-level bindings
  - Use a module system for namespace management

# Delayed Evaluation

For each language construct, there are rules governing when subexpressions get evaluated. In ML, Scheme, and Java:

- function arguments are "eager" (*call-by-value*)
- conditional branches are not

We could define a language in which function arguments were not evaluated before call, but instead at each use of argument in body. (*call-by-name*)

- Sometimes faster: (lambda (x) 3)
- Sometimes slower: (lambda (x) (+ x x))
- Equivalent *only* if function argument has no effects/non-termination

## Thunks

A "thunk" is just a function taking no arguments, which works great for delaying evaluation.

If thunks are lightweight enough syntactically, why not make if eager? (Example language: Smalltalk)

# Best of both worlds?

The "lazy" (*call-by-need*) rule: Evaluate the argument, the first time it's used. Save answer for subsequent uses.

- Asymptotically it's the best
- But behind-the-scenes bookkeeping can be costly
- And it's hard to reason about with effects
  - Typically used in (sub)languages without effects
- Nonetheless, a key idiom with syntactic support in Scheme
  - Which we reimplemented with `my-force` and `my-delay`
  - And related to *memoization*

# Streams

- A stream is an "infinite" list — you can ask for the rest of it as many times as you like and youll never get null.
- The universe is finite, so a stream must really be an object that acts like an infinite list.
- The idea: use a function to describe what comes next.

Note: Deep connection to sequential feedback circuits

- One new value on each clock cycle

Note: Connection to UNIX pipes

- `cmd1 | cmd2` has `cmd2` "pull" data from `cmd1`.

## Streams in Scheme

A pretty straightforward idiom:

- A stream is a thunk that when called returns a pair:

```
(next-answer . next-thunk)
```

- So "going another iteration with result `pr`" is `((cdr pr))`
- One thunk creating another thunk: use recursion
- Nice division of labor:
  - stream-creator knows how to generate values
  - stream-client knows how many are needed and what to do with each
- (No new semantics; just new idiom)

# Memoization

A "cache" of previous results is equivalent if results cannot change.

- Could be slower: cache too big or computation too cheap
- Could be faster: just a lookup
- In our fibonacci example it turns an exponential algorithm into a linear algorithm

An association list is not the fastest data structure for large memo tables, but works fine for 341.

Question: Why does `assoc` return the pair?