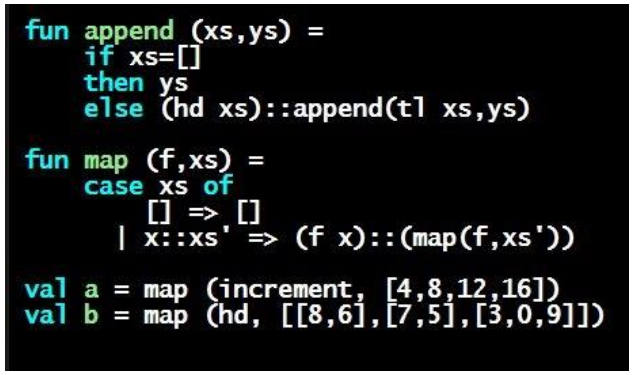

Programming Languages

Dan Grossman

Memoization

Memoization

- If a function has no side effects and does not read mutable memory, no point in computing it twice for the same arguments
  - Can keep a *cache* of previous results
  - Net win if (1) maintaining cache is cheaper than recomputing and (2) cached results are reused

- Similar to promises, but if the function takes arguments, then there are multiple "previous results"

- For recursive functions, this *memoization* can lead to *exponentially* faster programs
  - Related to algorithmic technique of dynamic programming

How to do memoization: see example

- Need a (mutable) cache that all calls using the cache share
  - So must be defined *outside* the function(s) using it

- See code for an example with Fibonacci numbers

  - Good demonstration of the idea because it is short, but, as shown in the code, there are also easier less-general ways to make **`fibonacci`** efficient

  - (An association list (list of pairs) is a simple but sub-optimal data structure for a cache; okay for our example)

assoc

- Example uses **`assoc`**, which is just a library function you could look up in the Racket reference manual:

  `(assoc v lst)` takes a list of pairs and locates the first element of **`lst`** whose car is equal to **`v`** according to **`is-equal?`**. If such an element exists, the pair (i.e., an element of **`lst`**) is returned. Otherwise, the result is **`#f`**.

- Returns **`#f`** for not found to distinguish from finding a pair with **`#f`** in cdr