CSE341 – Section 6

Memoization, Streams, and More

Cody Schroeder

February $13^{th}, 2013$

Outline

Refresher

Racket Fibonacci

```
(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

SML Fibonacci

```
fun fib n = if n <= 1
            then n
            else fib (n-1) + fib (n-2)
```

- SML and Racket aren't so different a lot of the time.
- A lot of what we learned in SML will transfer over.
  - For instance, dealing with lists is very similar.
  - Functional constructs are still frequently used.

Lexical Scope

- Variable lookup rules are nearly identical between SML and Racket.
  - One difference is the top-level **letrec** in a Racket module.

How do these procedures differ?

- Hint: I don't care about $36 \neq 37$

```
(define minus-fact-of-36
  (let ([v (fib 36)])
    (lambda (x)
      (- x v))))

(define minus-fact-of-37
  (lambda (x)
    (let ([v (fib 37)])
      (- x v))))
```

Computes (fib 36) once

Computes (fib 37) every call

Mutation

We care even more about scoping rules in the presence of mutation.

What do these procedures do when called?

```
(define increment-and-return1
  (let ([v 0])
    (lambda (x)
      (begin (set! v (+ x v))
             v))))

(define increment-and-return2
  (lambda (x)
    (let ([v 0])
      (begin (set! v (+ x v))
             v))))
```

Incorrect: will always return x

`increment-and-return` is meant to be a function that keeps a global counter and increments the counter with `x` during each call.

`set!` vs `set-mcar!` and friends

Mutation Functions

- In Racket there are multiple functions that have mutation as a side-effect.
  - **set!** assigns to some variable. It updates its value in the environment.
    - In Java, analogous to `x = 5;` (where 5 is just some value)
  - **set-mcar!** and **set-mcdr!** assigns to the fields of a `mpair` structure.
    - `car` and `cdr` could be considered fields in a `mpair` structure
    - In Java, analogous to `x.car = 5;` and `x.cdr = 10;`

- See http://docs.racket-lang.org/reference/mpairs.html for a reference on mpairs.

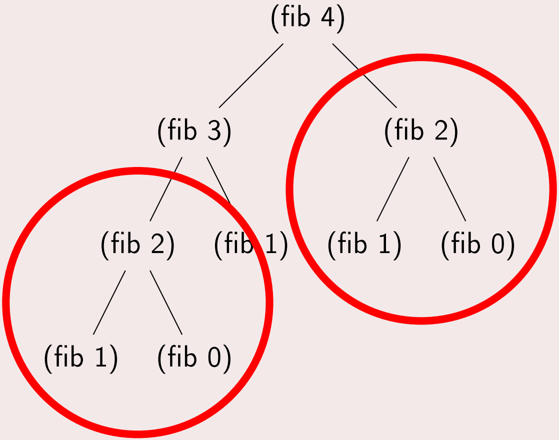
Back to Fibonacci

Why is this procedure slow?

```
(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Visualizing Fibonacci

Calling `(fib 4)`



There's a lot of redundant computation in this implementation.

Memoization

- Our fibonacci function ends up recomputing many values in the long run due to the recursive structure of the solution.
  - How can we fix this? (*Other than using an iterative solution...*)
- How about we store already computed results in some sort of *cache*?
  - The cache could be a mutable structure that will be added to as new results are computed.
  - This is the idea of memoization!
- In the previous tree example, the entire right subtree doesn't have to be recomputed. It'll be found in the cache.
- Our fibonacci function will become exponentially faster.

Associative Lists

- We will use an associative list for our cache.
- It's just a list of key-value pairs.
- There's a library function named **assoc** that will do lookups on a key in any valid associative list for us.
  - Locates the first pair in the list in which its **car** is **equal?** to the requested key value. Returns the entire pair found. If the key isn't found, it returns #f.

```
(define a-lst (list (cons 1 2)
                    (cons "Cody" "Schroeder")
                    (cons 42 #t)))

(displayln (assoc 1 a-lst))        ; (1 . 2)
(displayln (assoc "Cody" a-lst))   ; (Cody . Schroeder)
(displayln (assoc 42 a-lst))       ; (42 . #t)
(displayln (assoc "NON-EXISTANT-KEY" a-lst)) ; #f
```

One Way of Fixing Fibonacci

A Memoized Fibonacci

```
(define fib
  (let ([memo '((0 . 0) (1 . 1))])
   (lambda (n)
     (let ([prev-ans (assoc n memo)])
       (if prev-ans
           (cdr prev-ans)
           (let ([ans (+ (fib (- n 1)) (fib (- n 2)))])
             (set! memo (cons (cons n ans) memo))
             ans))))))
```

- How fast can `(fib 70000)` be computed now?

General Memoization

The Basic Pattern

```
(define function-name
  (let ([memo '()]) ;; memo can store base cases
   (lambda (x) ;; We could have more arguments, if we wanted.
     (let ([prev-ans (assoc x memo)]) ;; Check for saved result
       (if prev-ans
           (cdr prev-ans) ;; Just return memo'd answer
           (let ([new-ans (compute x)]) ;; Compute a new answer
             (set! memo (cons (cons x new-ans) memo)) ;; Save it
             new-ans)))))) ;; Return the new answer
```

Stream Definition

A Stream Is. . .

- A thunk that evaluates to a pair of an element and another stream.
- This is an infinitely recursive definition. There's **no end** to a stream.

Example

```
(define natural-numbers
  (letrec ([next-nat (lambda (n)
                       (lambda () (cons n (next-nat (+ 1 n)))))])
    (next-nat 1)))
```

Working with Streams

See code: streams.rkt.