CSE341 – Section 6
Memoization, Streams, and More

Cody Schroeder

February 13th, 2013
Outline

1 SML/Racket
   - Refresher
   - Lexical Scope
   - Mutation

2 Memoization
   - Fibonacci
   - General Memoization

3 Streams
   - Definition
   - Working with Streams
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.
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 $36 \neq 37$

1. `(define minus-fact-of-36`
2. `(let ([v (fib 36)])`
3. `(lambda (x)`
4. `(- x v))))`

Computes `(fib 36)` once

5. `(define minus-fact-of-37`
6. `(lambda (x)`
7. `(let ([v (fib 37)])`
8. `(- x v))))`

Computes `(fib 37)` every call
We care even more about scoping rules in the presence of mutation.

What do these procedures do when called?

```scheme
1 (define increment-and-return1
2  (let ([v 0])
3    (lambda (x)
4      (begin (set! v (+ x v))
6      v))))
7 (define increment-and-return2
8  (lambda (x)
9    (let ([v 0])
10   (begin (set! v (+ x v))
11     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;`

Back to Fibonacci

Why is this procedure slow?

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

Cody Schroeder  CSE341 – Section 6
There’s a lot of redundant computation in this implementation.
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.
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`.

```scheme
(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
```
A Memoized Fibonacci

```scheme
(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

1 (define function-name
2    (let ([memo '()]) ;; memo can store base cases
3      (lambda (x) ;; We could have more arguments, if we wanted.
4         (let ([prev-ans (assoc x memo)]) ;; Check for saved result
5             (if prev-ans
6                (cdr prev-ans) ;; Just return memo'd answer
7                  (let ([new-ans (compute x)]) ;; Compute a new answer
8                     (set! memo (cons (cons x new-ans) memo))) ;; Save it
9                     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

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

Cody Schroeder

CSE341 – Section 6
See code: streams.rkt.