CSE 341 AA: Section 6

Porter Jones
pbjones@cs.washington.edu
Office Hours: Thursdays 5:30 - 7:30pm

Fibonacci case study:

Here is one intuitive solution, but can we do better?

```
(define (fibonacci x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci (- x 1))
         (fibonacci (- x 2)))))
```

Memoization

- Why compute the same recursive call for a function twice when there are no major side-effects?

- Memoization is a way to “remember” previous calls

- Requires a way for the function to store both the input and the result of previous calls to that function

Lexical-scope and mutation

```
(define count-calls-correct
  (let [(count 0)] ; This remembers the count from previous calls
    (lambda ()
      (begin (set! count (+ count 1)) count))))
```

Lexical-scope and mutation

```
(define count-calls-wrong
  (lambda ()
    (let [(count 0)] ; This starts at 0 every time
      (begin (set! count (+ count 1)) count))))
```

Mutable Lists

- Similar to regular lists and pairs but not the same datatype.
    - Mutable pairs have type `mpair`. Use `mcons` for creation, `mcar` to get the first thing and `mcdr` for the second

- `set-car!` and `set-cdr!` actually change the “fields” of a `mpair`

- Use mutable types only when necessary! Prefer immutable!

Mutable Lists Example

```
(define mp (mcons 1 (mcons 2 null)))
(mpair? mp) ; #t
(mcar mp) ; get the first element in mp (car won't work!)
(mcdr mp) ; get the second element in mp (cdr won't work!)
(set-mcar! mp 5) ; change head of list in mp to 5
(set-mcdr! mp (mcons 3 null)) ; change tail list of mp
```

Associative Lists

- List of key/value pairs!
- Racket has a built in function assoc that takes a value (key), and a list, and returns the first pair with the given key it finds in the given list (false if there is no pair with the given key).

```
(define my-list (list (cons 1 2) (cons 3 4)))
(assoc 1 my-list) ; returns the pair `(1 . 2)
(assoc 4 my-list) ; returns #f
```

Putting it all together... a better fibonacci

```
(define memo-fibonacci
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans) ; return memoized answer
                     (let ([new-ans (if (or (= x 1) (= x 2))
                                        1
                                        (+ (f (- x 1))
                                           (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

Streams

- A function that when evaluated results in a pair with a value in the car and another stream in the cdr
- Create an infinitely long stream of values!

```
(define (ones) (cons 1 ones))

(define natural-numbers
  (letrec ([next-nat (lambda (x)
                       (cons x (lambda ()
                                 (next-nat (+ x 1)))))]) ; return next pair
    (lambda () (next-nat 0)))) ; "seed" the stream
```