PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

# CSE 341: Section 6

Spring 2017

Nick Mooney

# Agenda

- Memoization
  - Motivation
  - A quick detour…
  - Better fibonacci
- Streams
  - A quick refresher on thunks
  - Infinite lists!

# Memoization

- Why is the following “natural” implementation of the Fibonacci sequence slow?

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

- Tons of repeated work!
  - In fact, execution time grows with respect to $2^x$

# Memoization

**Motivation**

Remember the results of calls the first time we evaluate them, so we don’t have to redo any work

# A quick detour…

- An “associative list” is a list of pairs that you can think of as key/value pairs

```
(define my-list (list (cons 1 2) (cons 3 4) (cons 5 6) (cons "example" #t)))

(assoc 1 my-list) ; ‘(1 . 2)
(assoc 3 my-list) ; ‘(3 . 4)
(assoc “example” my-list) ; ‘(”example” . #t)
```

- assoc is part of the standard library

# How can we improve on Fibonacci?

# Memoization Recap

- Take a problem that involves lots of repeated work
- Add the ability to “remember” results
  - Maybe using an associative list, maybe some other way
- Now we only do the repeated work once, and we can look it up after that

# Streams

- A stream is basically an infinitely long list, with the added bonus that it doesn’t take an infinite amount of time to construct
  - Good for us
  - I’m gonna show you an infinite list
  - I want to go home later
  - You probably need to eat

A *stream* is a *thunk* that, when evaluated, produces a pair whose first element is an element of the stream, and whose second element is the stream that will produce the rest of the elements.

# The Simplest Stream

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

# More complex behavior

- Instead of returning the *same* function each time, let’s return a new function, which will produce the next value/function pair, etc…

# Some slightly more complex examples