Induction

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

- Understand how proof by induction works.
- Using that, understand how recursion works.
- Go over some example recursions.

Induction

A proof by induction works by making two steps do the work of an infinite number of steps. It's really a way of being very lazy!

- Pick a property $P(n)$ which you'd like to prove for all n.
- **Base case:** Prove $P(n)$, for $n = 1$, or whatever n's smallest value should be.
- **Induction Case:** You want to prove $P(n)$, for some general n. To do that, *assume* that $P(n-1)$ is true, and use that information to prove that $P(n)$ has to be true.

The idea is that there are an infinite number of n such that $P(n)$ is true. But with this technique you only had to prove two cases.

Induction Example

To Prove: Let $P(n)$ = "The sum of the first n odd numbers is n^2."

Base Case: Let $n = 1$. Then n^2 = 1, and the sum of the list $\{1\}$ is 1; therefore the base case holds.

Induction Case: Suppose you need to show that this property is true for some n. First, pretend that somebody else already did all the work of proving that $P(n-1)$ is true. Now use that to show that $P(n)$ is true, and take all the credit.

If $\{1, 3, 5, \ldots, 2n-3\} = (n-1)^2$, then add $2n-1$ …

$$\{1, 3, 5, \ldots, 2n-3, 2n-1\} = (n-1)^2 + 2n - 1$$
$$\Rightarrow n^2 - 2n + 1 + 2n - 1 \Rightarrow n^2$$

Recursion

A recursive routine has a similar structure. You have a base case, a recursive case, and a conditional to check which case is appropriate.

- Pick a function $f(n)$ which you'd like to compute for all n.
- **Base case:** Compute $f(n)$, for $n = 1$, or whatever n's smallest value should be.
- **Recursive Case:** Assume that someone else already computed $f(n-1)$ for you. Use that information to compute $f(n)$, and then take all the credit.

Iterating Recursion Example

Suppose you want a recursive routine that computes the *n*th square.

```
1 (defn nthsq [n]
2    (cond (= n 0) 0
3          :else   (+ (* 2 n) -1 (nthsq (- n 1)))))
```

- The conditional checks which case is active.
- Line 2 is the base case — it stops the recursion.
- Line 3 is the recursive case.

Important things about recursion

```
(defn nthsq [n]
   (cond (= n 0) 0
         :else   (+ (* 2 n) -1 (nthsq (- n 1)))))
```

- Your base case has to stop the computation.
- Your recursive case has to call the function with a *smaller* argument than the original call.
- Your conditional expression has to be able to tell when the base case is reached.
- Failure to do any of the above will cause an infinite loop.

Example 2: Factorial

The Definition

$$n! = n * n - 1 * \cdots * 2 * 1$$

Example 2: The recursive part

- Find the recursive part

The Definition

$$n! = n * \underbrace{n - 1 * \cdots * 2 * 1}_{(n-1)!}$$

```
        (fact (- n 1))
```

Example 2: The recursive part

- Combine it with the “current” part. (What is your last step?)

The Definition

$$n! = \overbrace{n * \underbrace{n - 1 * \cdots * 2 * 1}_{(n-1)!}}^{\text{last step}}$$

```
(* n (fact (- n 1)))
```

Example 2: The recursive part

- What is your base case?

The Definition

$$n! = \overbrace{n * \underbrace{n - 1 * \cdots * 2 * 1}_{(n-1)!}}^{\text{last step}}$$

```
    (cond (= n 0) 1   ; base
          :else   (* n (fact (- n 1)))) ; recursive
```

Example 2: The recursive part

- Wrap it up.

The Definition

$$n! = \overbrace{n * \underbrace{n - 1 * \cdots * 2 * 1}_{(n-1)!}}^{\text{last step}}$$

```
(defn fact [n]
    (cond (= n 0) 1   ; base
          :else   (* n (fact (- n 1))))) ; recursive
```

Function Calls

- Let's look at what happens when a function is called.

Sample Function

```
(defn foo [a]
   (let [aa (* a a)]
        (+ aa a)))
```

- The above function has one paramater and one local.
- If we call it three times, what will happen in memory?

```
(+ (foo 1) (foo 2) (foo 3))
```

Function Calls

- Let's look at what happens when a function is called.

Sample Function

```
(defn foo [a]
   (let [aa (* a a)]
         (+ aa a)))
```

- The above function has one paramater and one local.
- If we call it three times, what will happen in memory?

```
(+ (foo 1) (foo 2) (foo 3))
```

First Call Second Call Third Call

a	1
aa	1

Function Calls

- Let's look at what happens when a function is called.

Sample Function

```
(defn foo [a]
   (let [aa (* a a)]
        (+ aa a)))
```

- The above function has one paramater and one local.
- If we call it three times, what will happen in memory?

```
(+ (foo 1) (foo 2) (foo 3))
```

First Call

a	1
aa	1

Second Call

a	2
aa	4

Third Call

Function Calls

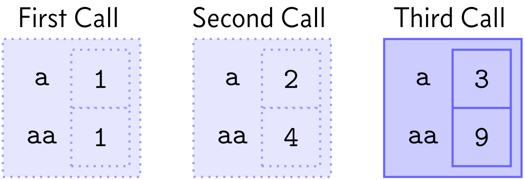
- Let's look at what happens when a function is called.

Sample Function

```
(defn foo [a]
   (let [aa (* a a)]
        (+ aa a)))
```

- The above function has one paramater and one local.
- If we call it three times, what will happen in memory?

```
(+ (foo 1) (foo 2) (foo 3))
```



Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

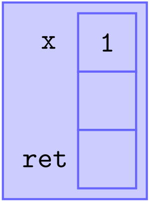
- What happens when we call `(foo 1)`?

Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```clojure
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

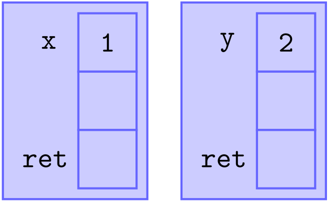


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

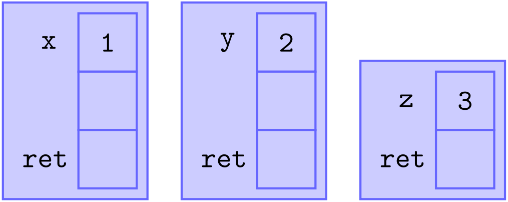


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```clojure
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

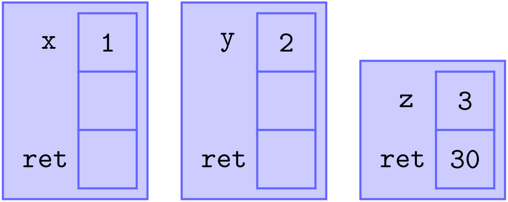


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

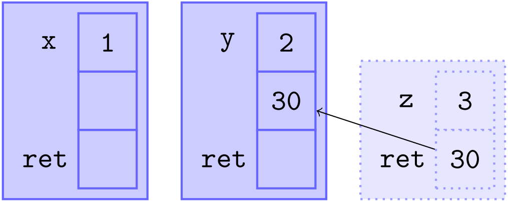


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

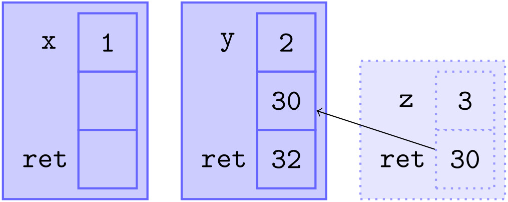


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

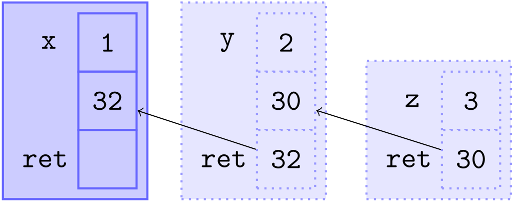


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?

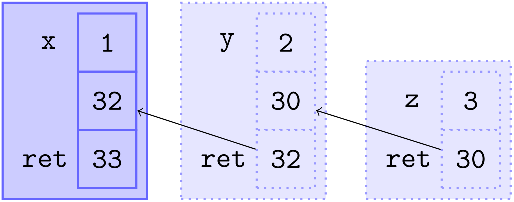


Functions Calling Functions

- If one function calls another, *both* activation records exist simultaneously.

```
(defn foo [x] (+ x (bar (+ x 1))))
(defn bar [y] (+ y (baz (+ y 1))))
(defn baz [z] (* z 10))
```

- What happens when we call `(foo 1)`?



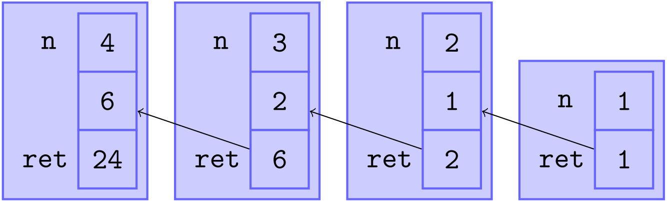
Factorial

- This works if the function calls itself.

Factorial

```
(defn fact [n]
   (cond (= n 0) 1   ; base
         :else    (* n (fact (- n 1))))) ; recursive
```

- fact 4 ...



Example 3: Fibonacci

The Definition

$$
\begin{aligned}
f_1 &= 1 \\
f_2 &= 1 \\
f_n &= f_{n-1} + f_{n-2}
\end{aligned}
$$

- Notice here you have *two* base cases and *two recursions*!

Example 3: Fibonacci

The Definition

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

```
(defn fib [n]
   (cond (= n 2) 1
         (= n 1) 1
         :else   (+ (fib (- n 1)) (fib (- n 2)))))
```