Basic Recursion

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

Your goal for this lecture is to understand recursion — at least, to get a start on it. We will talk about

- Diagram a series of function calls.
- Show how to write a recursive function on integers.
- Show how to write a recursive function on lists.

Function Calls

- Remember the syntax of a function definition in Haskell.

Function Syntax

```haskell
foo a =
  let aa = a * a
   in aa + a
```

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

```haskell
x = (foo 1) + (foo 2) + (foo 3)
```

Function Calls

- Remember the syntax of a function definition in Haskell.

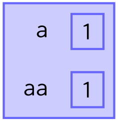
Function Syntax

```haskell
foo a =
  let aa = a * a
   in aa + a
```

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

```haskell
x = (foo 1) + (foo 2) + (foo 3)
```

First Call Second Call Third Call



Function Calls

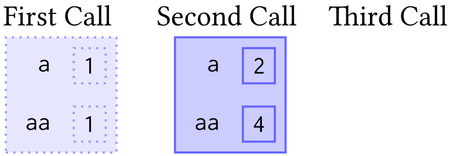
- Remember the syntax of a function definition in Haskell.

Function Syntax

```haskell
foo a =
  let aa = a * a
   in aa + a
```

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

```haskell
x = (foo 1) + (foo 2) + (foo 3)
```



Function Calls

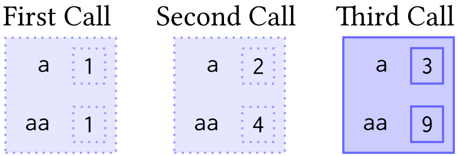
- Remember the syntax of a function definition in Haskell.

Function Syntax

```haskell
foo a =
  let aa = a * a
   in aa + a
```

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

```haskell
x = (foo 1) + (foo 2) + (foo 3)
```



Functions Calling Functions

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

```haskell
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

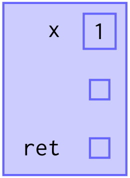
- What happens when we call `foo 1`?

Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

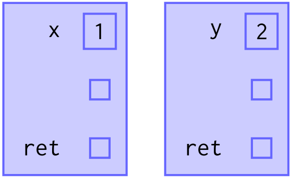


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

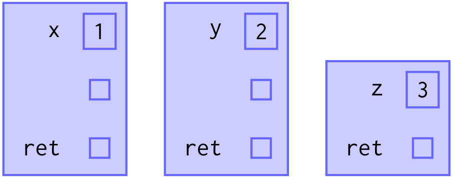


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

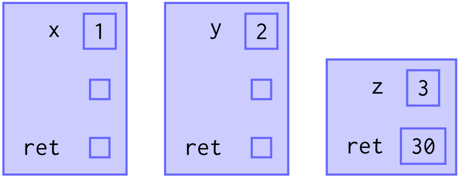


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

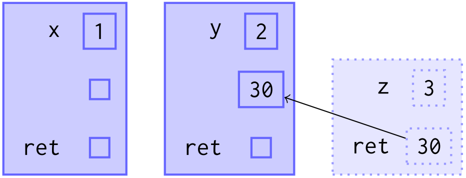


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

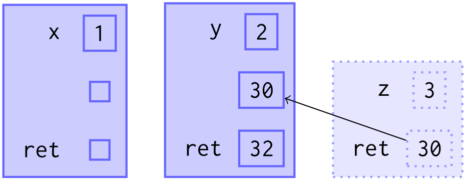


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

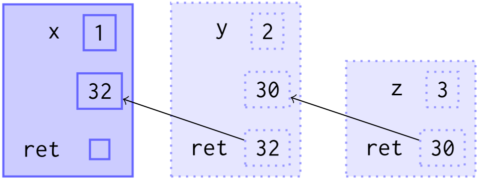


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

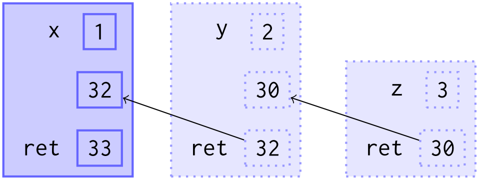


Functions Calling Functions

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

```
foo x = x + bar (x+1)
bar y = y + baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



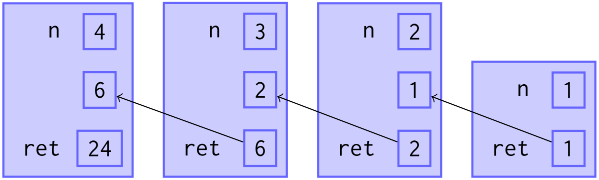
Factorial

- This works if the function calls itself.

Factorial

```
fact 0 = 1
fact 1 = 1
fact n = n * fact (n-1)
```

- `fact 4 ...`



Lists

Because lists are recursive, functions that deal with lists tend to be recursive.

Length

```haskell
mylength :: [a] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs

mylength s -- would return 3
```

- The base case stops the computation.
- Your recursive case calls itself with a *smaller* argument than the original call.