Basic Recursion

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Diagram the stack frames that result from a series of function calls.
- Use HASKELL to write a recursive function on integers.
- Use HASKELL to write a recursive function on lists.

Function Calls

- Remember the syntax of a function definition in HASKELL.

Function Syntax

```
1 foo a =
2   let aa = a * a
3    in aa + a
```

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

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

Function Calls

- Remember the syntax of a function definition in HASKELL.

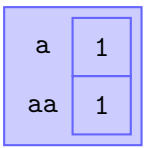
Function Syntax

```
1 foo a =
2   let aa = a * a
3    in aa + a
```

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

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

First Call Second Call Third Call



Function Calls

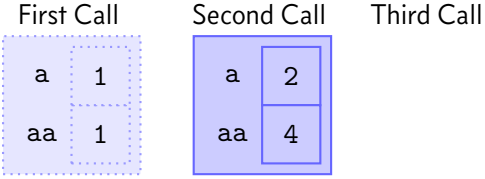
- Remember the syntax of a function definition in HASKELL.

Function Syntax

```
1 foo a =
2   let aa = a * a
3    in aa + a
```

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

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



Function Calls

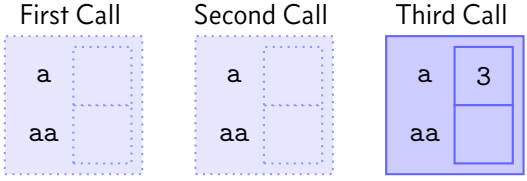
- Remember the syntax of a function definition in HASKELL.

Function Syntax

```
1 foo a =
2   let aa = a * a
3    in aa + a
```

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

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



Functions Calling Functions

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

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

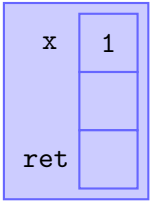
- What happens when we call `foo 1`?

Functions Calling Functions

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

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

- What happens when we call `foo 1`?

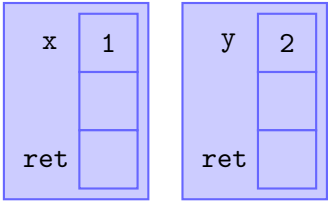


Functions Calling Functions

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

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

- What happens when we call `foo 1`?

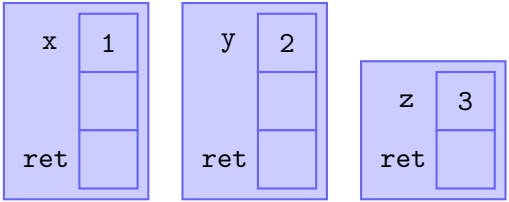


Functions Calling Functions

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

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

- What happens when we call `foo 1`?

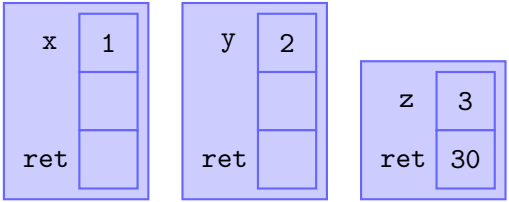


Functions Calling Functions

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

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

- What happens when we call `foo 1`?

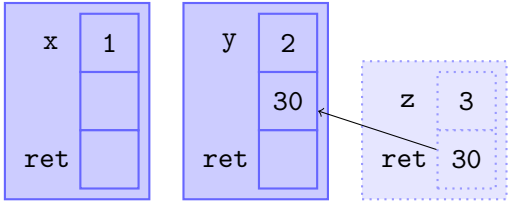


Functions Calling Functions

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

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

- What happens when we call `foo 1`?

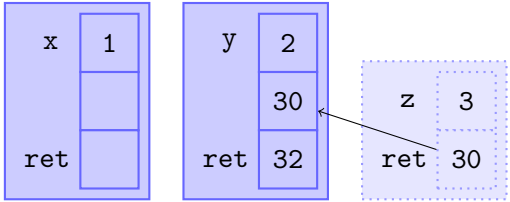


Functions Calling Functions

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

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

- What happens when we call `foo 1`?

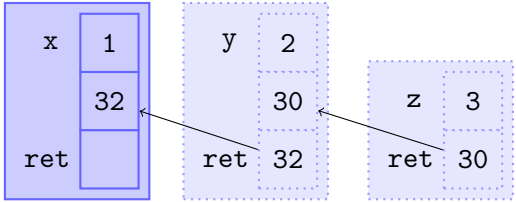


Functions Calling Functions

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

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

- What happens when we call `foo 1`?

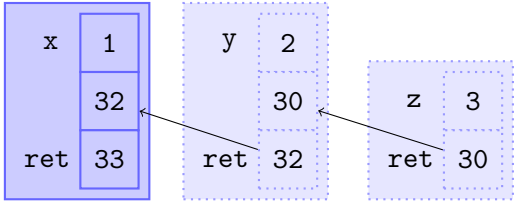


Functions Calling Functions

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

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

- What happens when we call `foo 1`?



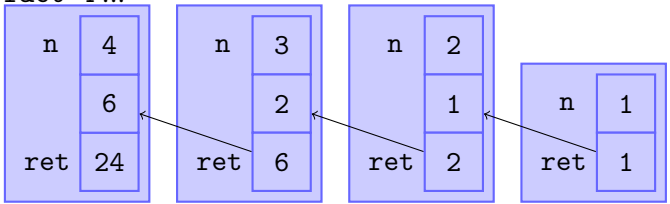
Factorial

- This works if the function calls itself.

Factorial

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

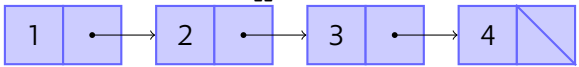
- `fact 4` ...



Lists in Haskell

- Haskell has a built-in syntax for singly linked lists.
- The empty list is `[]`.
- You can use `:` to create a new list ...

```
1 : 2 : 3 : 4 : []
```



- You can also write `[1,2,3,4]`.

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.

Activity

- Write a function `fib` that computes the nth Fibonacci number F_n. Let $F_1 = 1$ and $F_2 = 1$.
- Write a function `sumList` that takes a list and sums its elements.
- Write a function `incList` that takes a list and increments its elements.

Solutions to `fib` and `sumList`

```haskell
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)

sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Solution to incList

- Remember that you must create a new list!

```
1 incList [] = []
2 incList (x:xs) = x+1 : incList xs
```

History

- The first programming language to implement recursion was LISP in 1958. [McC79]

References

[McC79] John McCarthy. *History of Lisp*. Stanford University, 1979. URL: http://www-formal.stanford.edu/jmc/history/lisp/lisp.html.