# Tail Recursion

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

# Objectives

- Understand what makes a function tail recursive.
- Explain how the compiler makes tail recursion efficient.

# Tail Calls

**Tail Position** A subexpression *s* of expressions *e*, if it is evaluated, will be taken as the value of *e*.

- if x > 3 then $\underline{\texttt{x + 2}}$ else $\underline{\texttt{x - 4}}$
- f (x * 3) — no (proper) tail position here.

**Tail Call** A function call that occurs in tail position.

- if h x then $\underline{\texttt{h x}}$ else x + g x

# Your Turn

Find the tail calls!

## Example Code

```
calc n i | n==2 = i
         | odd n = calc (n*3+1) (i+1)
         | otherwise = calc (n ‘div‘ 2) (i+1)

fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?

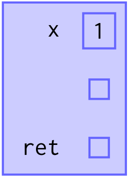
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?



x 1

ret

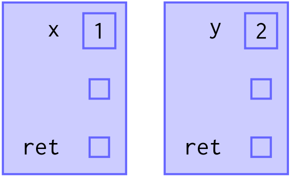
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?



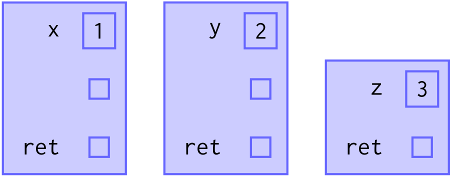
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?



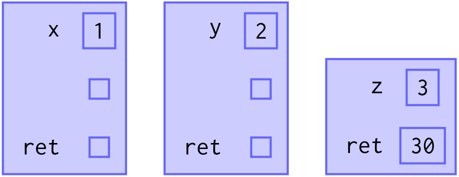
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?



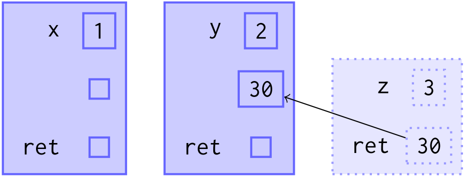
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?



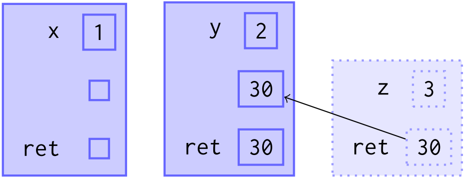
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

### Example

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

- What happens when we call `foo 1`?



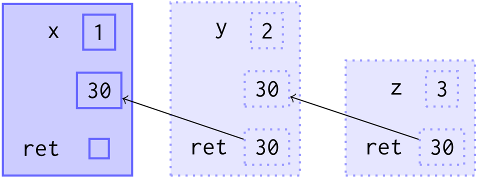
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

## Example

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

- What happens when we call `foo 1`?



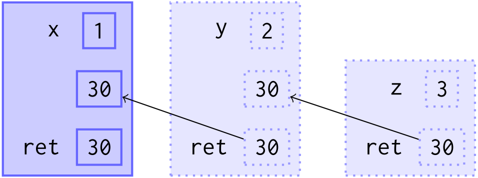
# Tail Call Example

- If one function calls another in tail position, we get a special behavior.

## Example

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

- What happens when we call `foo 1`?



# The Tail Call Optimization

### Example

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

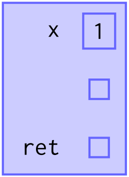
- If that's the case, we can cut out the middle man...

# The Tail Call Optimization

### Example

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

- If that's the case, we can cut out the middle man...

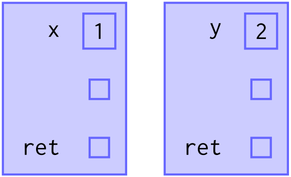


# The Tail Call Optimization

## Example

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

- If that's the case, we can cut out the middle man...

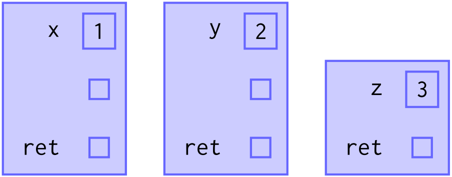


# The Tail Call Optimization

### Example

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

- If that's the case, we can cut out the middle man…

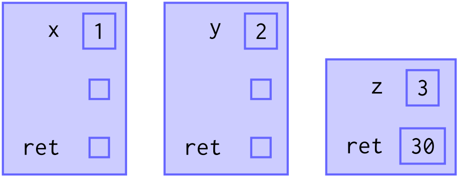


# The Tail Call Optimization

## Example

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

- If that's the case, we can cut out the middle man...

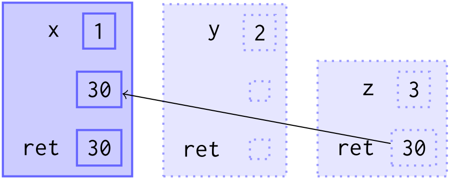


# The Tail Call Optimization

## Example

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

- If that's the case, we can cut out the middle man…



# The Tail Call Optimization

### Example

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

- If that's the case, we can cut out the middle man...
- Actually, we can do even better than that.

# The optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

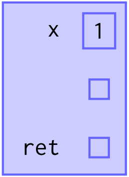
```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

# The optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

### Example

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

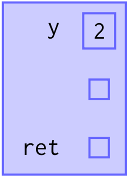


# The optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

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



y 2

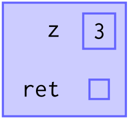
ret

# The optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

## Example

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



z 3

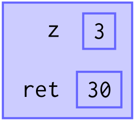
ret

# The optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

## Example

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



z 3

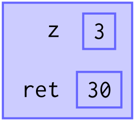
ret 30

# The optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

## Example

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



z 3

ret 30

- This allows recursive functions to be written as loops internally.

# Direct-Style Recursion

- In recursion, you split the input into the “first piece” and the “rest of the input”.
- In direct-style recursion: the recursive call computes the result for the rest of the input, and then the function combines the result with the first piece.
- In other words, you wait until the recursive call is done to generate your result.

### Direct Style Summation

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

# Accumulating Recursion

- In accumulating recursion: generate an intermediate result *now*, and give that to the recursive call.
- Usually this requires an auxiliary function.

### Tail Recursive Summation

```
sum xx = aux xx 0
   where aux [] a = a
         aux (x:xs) a = aux xs (a+x)
```

# Further Reading

- Forward recursion can be made to traverse a list at return time rather than call time, forming a pattern called “There and Back Again,” which can do some interesting things....
- Example: write a function `convolve` which takes two lists $(x_1 \quad x_2 \quad \cdots \quad x_n)$ and $(y_1 \quad y_2 \quad \cdots \quad y_n)$ and produces an output list $(x_1 y_n \quad x_2 y_{n-2} \quad \cdots \quad x_n y_1)$ where $n$ is unknown. Use only $\mathcal{O}(n)$ recursive calls, and no temporary lists.
- For the solution, see Olivier Danvy’s paper There and Back Again.