Generative Recursion and Tail Recursion

CS 350

Dr. Joseph Eremondi

Last updated: July 29, 2024

Broad Goals

Overview

- Objectives

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion
  - Tail-calls

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion
  - Tail-calls
  - Tail-call elimination

Overview

- Objectives
  - Iteratively building solutions to problems in functional languages
  - Implementing recursive procedures efficiently
  - Capturing this pattern of recursion as a higher-order function
- Key Concepts
  - Generative recursion
  - Tail-calls
  - Tail-call elimination
  - Folds

Generative Recursion

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far

The Problem

- Some problems don’t obviously map to the style of recursion we’ve seen so far
    - Or do, but not efficiently

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about "the result so far"

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about "the result so far"
- e.g. What's the recursive version of:

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
    - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
    - No good way to talk about "the result so far"
- e.g. What's the recursive version of:

The Problem

- Some problems don't obviously map to the style of recursion we've seen so far
  - Or do, but not efficiently
- Especially problems that are about incrementally updating a value
  - No good way to talk about "the result so far"
- e.g. What's the recursive version of:

```
int x = startVal;
for (int i = 0; i < n; i++)
{
  x = f(x);
}
```

Example: Reversing a List The Naive Way

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means
    - Reversing the tail of the list

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means
  - Reversing the tail of the list
  - Putting the first element at the end of the new list

Example: Reversing a List The Naive Way

```
(define (reverse [xs : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty empty]
    [(cons h t)
       (append (reverse t) (list h))]))
(reverse '(1 2 3))
(reverse '("hello" "goodbye"))
```

```
'(3 2 1)
'("goodbye" "hello")
```

- Reversing the empty list produces the empty list
- Reversing a list with one element means
  - Reversing the tail of the list
  - Putting the first element at the end of the new list
- $\mathcal{O}(n^2)$: Each `append` has to walk through the whole list

Reverse: What we want (conceptually)

- Start with an empty list

Reverse: What we want (conceptually)

- Start with an empty list
- Pop an element off the first list, then add it to our result list so far

Reverse: What we want (conceptually)

- Start with an empty list
- Pop an element off the first list, then add it to our result list so far
- When reach the end, have a reversed list

An efficient reverse

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
    - Base case: finished, produce the list we've built so far

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list
    - Call with head appended to `listSoFar`

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
    - Base case: finished, produce the list we've built so far
    - Recursive case: keep building, but on the tail of the list
        - Call with head appended to `listSoFar`
- Main function calls the helper with the start value

An efficient reverse

```
(define (reverseHelper [xs : (Listof 'a)]
                       [listSoFar : (Listof 'a)]) : (Listof 'a)
  (type-case (Listof 'a) xs
    [empty
       listSoFar]
    [(cons h t)
       (reverseHelper t (cons h listSoFar))]))
(define (reverse xs) (reverseHelper xs '()))
```

- Helper function
  - Base case: finished, produce the list we've built so far
  - Recursive case: keep building, but on the tail of the list
    - Call with head appended to `listSoFar`
- Main function calls the helper with the start value
  - Empty list `'()`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Extra argument called the *accumulator*

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Extra argument called the *accumulator*
    - Often named `accum`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
    - Extra argument called the *accumulator*
        - Often named `accum`
    - Has same type as return type

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
    - Extra argument called the *accumulator*
        - Often named `accum`
    - Has same type as return type
        - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
    - Extra argument called the *accumulator*
        - Often named `accum`
    - Has same type as return type
        - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Extra argument called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Extra argument called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator
  - Recursive case: call the helper recursively on the sub-value

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
    - Extra argument called the *accumulator*
        - Often named `accum`
    - Has same type as return type
        - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
    - Base case: return the accumulator
    - Recursive case: call the helper recursively on the sub-value
        - Compute an updated value for the result so far

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Extra argument called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator
  - Recursive case: call the helper recursively on the sub-value
    - Compute an updated value for the result so far
    - Pass it as the accumulator for the recursive call

General Template

To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`

- Write a helper function with one extra argument
  - Extra argument called the *accumulator*
    - Often named `accum`
  - Has same type as return type
    - `fHelper : (Ty1 Ty2 ... TyN ReturnTy -> ReturnTy)`
- Helper function is recursive
  - Base case: return the accumulator
  - Recursive case: call the helper recursively on the sub-value
    - Compute an updated value for the result so far
    - Pass it as the accumulator for the recursive call
- Last step: `f` calls helper with initial accumulator value

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
    - In C/C++: where to execute after return is computed
    - Keep track of the expression we'll plug the result into

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed
  - Keep track of the expression we'll plug the result into
    - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed
  - Keep track of the expression we'll plug the result into
    - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
    - In C/C++: where to execute after return is computed
    - Keep track of the expression we'll plug the result into
        - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns
    - You've probably seen this when you get an error message in e.g. Python

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
    - In C/C++: where to execute after return is computed
    - Keep track of the expression we'll plug the result into
        - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns
    - You've probably seen this when you get an error message in e.g. Python
- Stack contents

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed
  - Keep track of the expression we'll plug the result into
    - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns
  - You've probably seen this when you get an error message in e.g. Python
- Stack contents
  - Body of function being currently computed

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed
  - Keep track of the expression we'll plug the result into
    - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns
  - You've probably seen this when you get an error message in e.g. Python
- Stack contents
  - Body of function being currently computed
  - Environment for that body

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed
  - Keep track of the expression we'll plug the result into
    - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns
  - You've probably seen this when you get an error message in e.g. Python
- Stack contents
  - Body of function being currently computed
  - Environment for that body
  - Position in expression that's waiting for the result of this function

The Call Stack

- In Racket (or any language), when we make a function call, we need to know what to do with the result
  - In C/C++: where to execute after return is computed
  - Keep track of the expression we'll plug the result into
    - Note: We don't have a stack in Curly (yet) because our interpreter is written such that the Racket stack keeps track of where to return.
- Each function call pushes onto the call stack, pops when it returns
  - You've probably seen this when you get an error message in e.g. Python
- Stack contents
  - Body of function being currently computed
  - Environment for that body
  - Position in expression that's waiting for the result of this function
    - Called the *continuation*

Stack Overflow

- If the call stack gets too large, a *stack overflow* can be triggered

Stack Overflow

- If the call stack gets too large, a *stack overflow* can be triggered
    - When the memory allocated for the stack is exceeded

Stack Overflow

- If the call stack gets too large, a *stack overflow* can be triggered
  - When the memory allocated for the stack is exceeded
- Typically the stack has much less space than the heap (dynamically allocated memory)

Stack Overflow

- If the call stack gets too large, a *stack overflow* can be triggered
  - When the memory allocated for the stack is exceeded
- Typically the stack has much less space than the heap (dynamically allocated memory)
  - Specifics depend on the implementation of your language

Example

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:
    - Push body `(cons (g x 1) empty)`, env `x := 3`, continuation is `END_OF_PROGRAM`

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(cons (g x 1) empty)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(cons (g x 1) empty)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `(cons [] '())`

Example

```scheme
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(cons (g x 1) empty)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `(cons [] '())`
    - `[]` shows where the result of the body is used

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(cons (g x 1) empty)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `(cons [] '())`
    - `[]` shows where the result of the body is used
- Return (pop) : plug `4` in place of `[]` to get `cons 4 '()`

Example

```
(define (g y z) (+ y z))
(define (f x)
  (cons (g x 1) '()))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(cons (g x 1) empty)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `(cons [] '())`
    - `[]` shows where the result of the body is used
- Return (pop) : plug `4` in place of `[]` to get `cons 4 '()`
- Return (pop), see `END_OF_PROGRAM` marker, produce `'(4)` as result

Returning a Function Call

- Slightly different example

Returning a Function Call

- Slightly different example

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call f:

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call `f`:
    - Push body `(g x 1)`, env `x := 3`, continuation is `END_OF_PROGRAM`

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call `f`:
    - Push body `(g x 1)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(g x 1)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `[]`

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(g x 1)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `[]`
    - Result of `g` is value of `f`'s body

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(g x 1)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `[ ]`
    - Result of `g` is value of `f`'s body
- Return (pop) : plug `4` in place of `[ ]` to get 4

Returning a Function Call

- Slightly different example

```
(define (g y z) (+ y z))
(define (f x)
  (g x 1))
(f 3)
```

- Start: stack top empty
- Call `f`:
  - Push body `(g x 1)`, env `x := 3`, continuation is `END_OF_PROGRAM`
- Call `g`
  - Push body `(+ y z)`, env `y := 3, z := 1`, continuation is `[]`
    - Result of `g` is value of `f`'s body
- Return (pop) : plug `4` in place of `[]` to get 4
- Return (pop), see `END_OF_PROGRAM` marker, produce `4` as result

What’s The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`

What's The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
    - By adding it as the first field of `cons`

What's The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
  - By adding it as the first field of `cons`
- In the second example, the result of `g` *was* the result of `f`

What’s The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
  - By adding it as the first field of `cons`
- In the second example, the result of `g` *was* the result of `f`
  - We never needed `x` in the environment after making the call to `g`

What's The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
  - By adding it as the first field of `cons`
- In the second example, the result of `g` *was* the result of `f`
  - We never needed `x` in the environment after making the call to `g`
  - We never need the body of `f` after making the call to `g`

What's The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
  - By adding it as the first field of `cons`
- In the second example, the result of `g` *was* the result of `f`
  - We never needed `x` in the environment after making the call to `g`
  - We never need the body of `f` after making the call to `g`
    - All the body of `f` tells us is to compute the body of `g`

What's The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
  - By adding it as the first field of `cons`
- In the second example, the result of `g` *was* the result of `f`
  - We never needed `x` in the environment after making the call to `g`
  - We never need the body of `f` after making the call to `g`
    - All the body of `f` tells us is to compute the body of `g`
- The Racket/Plait interpreter can just change the body and env on the stack, *without pushing to it*

What's The Difference?

- In the first example, we had to take the result of the call to `g` and incorporate it into the result of `f`
  - By adding it as the first field of `cons`
- In the second example, the result of `g` *was* the result of `f`
  - We never needed `x` in the environment after making the call to `g`
  - We never need the body of `f` after making the call to `g`
    - All the body of `f` tells us is to compute the body of `g`
- The Racket/Plait interpreter can just change the body and env on the stack, *without pushing to it*
  - Mutation → fast

Tail Calls

- An expression is in **tail position** if it is

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function
  - The branch of a conditional (if, type-case, etc.) that was in tail position

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function
  - The branch of a conditional (if, type-case, etc.) that was in tail position
- A function call in tail position is a **tail call**

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function
  - The branch of a conditional (if, type-case, etc.) that was in tail position
- A function call in tail position is a **tail call**
- A function is NOT a tail call if it is

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function
  - The branch of a conditional (if, type-case, etc.) that was in tail position
- A function call in tail position is a **tail call**
- A function is NOT a tail call if it is
  - An argument to another function or constructor

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function
  - The branch of a conditional (if, type-case, etc.) that was in tail position
- A function call in tail position is a **tail call**
- A function is NOT a tail call if it is
  - An argument to another function or constructor
- Tail calls can be implemented as *jumps* and *mutation*

Tail Calls

- An expression is in **tail position** if it is
  - The body of a function
  - The branch of a conditional (if, type-case, etc.) that was in tail position
- A function call in tail position is a **tail call**
- A function is NOT a tail call if it is
  - An argument to another function or constructor
- Tail calls can be implemented as *jumps* and *mutation*
  - Don't need to add to the stack

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**
- Generative recursion (as described above) *is* tail recursion

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**
- Generative recursion (as described above) *is* tail recursion
- Tail recursive functions can be optimized into a loop

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**
- Generative recursion (as described above) *is* tail recursion
- Tail recursive functions can be optimized into a loop
    - No need to change the stack

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**
- Generative recursion (as described above) *is* tail recursion
- Tail recursive functions can be optimized into a loop
  - No need to change the stack
  - Recursive call $\longrightarrow$ update parameters then jump

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**
- Generative recursion (as described above) *is* tail recursion
- Tail recursive functions can be optimized into a loop
    - No need to change the stack
    - Recursive call $\longrightarrow$ update parameters then jump
    - Constant memory usage

Tail Recursion

- Tail recursion: when **all recursive calls in a function definition are tail calls**
- Generative recursion (as described above) *is* tail recursion
- Tail recursive functions can be optimized into a loop
  - No need to change the stack
  - Recursive call $\longrightarrow$ update parameters then jump
  - Constant memory usage
  - Less overhead

Example: List Length in Constant Space

Example: List Length in Constant Space

```
(define (fast-length-helper [xs : (Listof 'a)]
                            [accum : Number])
  (type-case (Listof 'a) xs
    [empty accum]
    [(cons h t)
       (fast-length-helper t (+ 1 accum))]))
(define (fast-length xs) (fast-length-helper xs 0))
```

- Instead of computing the result directly, we incorporate the “result so far” into the new information from the non-recursive arguments

Example: List Length in Constant Space

```
(define (fast-length-helper [xs : (Listof 'a)]
                            [accum : Number])
  (type-case (Listof 'a) xs
    [empty accum]
    [(cons h t)
       (fast-length-helper t (+ 1 accum))]))
(define (fast-length xs) (fast-length-helper xs 0))
```

- Instead of computing the result directly, we incorporate the “result so far” into the new information from the non-recursive arguments
- We keep computing by calling `fast-length-helper` recursively

Example: List Length in Constant Space

```
(define (fast-length-helper [xs : (Listof 'a)]
                            [accum : Number])
  (type-case (Listof 'a) xs
    [empty accum]
    [(cons h t)
       (fast-length-helper t (+ 1 accum))]))
(define (fast-length xs) (fast-length-helper xs 0))
```

- Instead of computing the result directly, we incorporate the “result so far” into the new information from the non-recursive arguments
- We keep computing by calling `fast-length-helper` recursively
    - Parameter gets smaller

Example: List Length in Constant Space

```
(define (fast-length-helper [xs : (Listof 'a)]
                            [accum : Number])
  (type-case (Listof 'a) xs
    [empty accum]
    [(cons h t)
       (fast-length-helper t (+ 1 accum))]))
(define (fast-length xs) (fast-length-helper xs 0))
```

- Instead of computing the result directly, we incorporate the “result so far” into the new information from the non-recursive arguments
- We keep computing by calling `fast-length-helper` recursively
  - Parameter gets smaller
  - Accumulator gets more information

Example: List Length in Constant Space

```
(define (fast-length-helper [xs : (Listof 'a)]
                            [accum : Number])
  (type-case (Listof 'a) xs
    [empty accum]
    [(cons h t)
       (fast-length-helper t (+ 1 accum))]))
(define (fast-length xs) (fast-length-helper xs 0))
```

- Instead of computing the result directly, we incorporate the “result so far” into the new information from the non-recursive arguments
- We keep computing by calling `fast-length-helper` recursively
  - Parameter gets smaller
  - Accumulator gets more information
- Recursive call is tail call

Example: List Length in Constant Space

```
(define (fast-length-helper [xs : (Listof 'a)]
                            [accum : Number])
  (type-case (Listof 'a) xs
    [empty accum]
    [(cons h t)
      (fast-length-helper t (+ 1 accum))]))
(define (fast-length xs) (fast-length-helper xs 0))
```

- Instead of computing the result directly, we incorporate the "result so far" into the new information from the non-recursive arguments
- We keep computing by calling `fast-length-helper` recursively
  - Parameter gets smaller
  - Accumulator gets more information
- Recursive call is tail call
- Won't stack overflow, even on large arguments

Live Example: Slow and Fast Factorial

- See Racket in lecture

A Note On Terminology

- **Generative recursion** is when recursion works by building up (generating) a solution as an extra parameter, and returning that parameter in the base case

A Note On Terminology

- **Generative recursion** is when recursion works by building up (generating) a solution as an extra parameter, and returning that parameter in the base case
- **Tail recursion** is when all recursive calls in a function are tail calls

A Note On Terminology

- **Generative recursion** is when recursion works by building up (generating) a solution as an extra parameter, and returning that parameter in the base case
- **Tail recursion** is when all recursive calls in a function are tail calls
- Many examples we'll see in this course of tail recursion are also generative recursion, and vice versa

A Note On Terminology

- **Generative recursion** is when recursion works by building up (generating) a solution as an extra parameter, and returning that parameter in the base case
- **Tail recursion** is when all recursive calls in a function are tail calls
- Many examples we'll see in this course of tail recursion are also generative recursion, and vice versa
  - Some counter-examples, e.g. `foldr` that we'll see soon

Tail Recursion and While Loops

- We can relate tail recursion to while loops in imperative languages

Tail Recursion and While Loops

- We can relate tail recursion to while loops in imperative languages

Tail Recursion and While Loops

- We can relate tail recursion to while loops in imperative languages

```
x = initialValue;
while (test(x)){
 x = f(x);
}
return g(x);
```

- Is equivalent to

Tail Recursion and While Loops

- We can relate tail recursion to while loops in imperative languages

```
x = initialValue;
while (test(x)){
 x = f(x);
}
return g(x);
```

- Is equivalent to

Tail Recursion and While Loops

- We can relate tail recursion to while loops in imperative languages

```
x = initialValue;
while (test(x)){
 x = f(x);
}
return g(x);
```

- Is equivalent to

```
(define (helper x)
  (if (test x)
      (helper (f x))
      (g x)))
```

- Updating multiple variables $\longrightarrow$ multiple arguments to `helper`

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it’s *explicit state*

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it's *explicit state*
  - You can tell what state there is by looking at the type of a function

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it's *explicit state*
    - You can tell what state there is by looking at the type of a function
    - Nothing is hidden from the

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it's *explicit state*
    - You can tell what state there is by looking at the type of a function
    - Nothing is hidden from the
- Bugs in code are often due to subtle interactions between mutable states

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it's *explicit state*
    - You can tell what state there is by looking at the type of a function
    - Nothing is hidden from the
- Bugs in code are often due to subtle interactions between mutable states
    - Especially with parallelism/concurrency

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it's *explicit state*
    - You can tell what state there is by looking at the type of a function
    - Nothing is hidden from the
- Bugs in code are often due to subtle interactions between mutable states
    - Especially with parallelism/concurrency
- The functional style means that no state is hidden

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it's *explicit state*
    - You can tell what state there is by looking at the type of a function
    - Nothing is hidden from the
- Bugs in code are often due to subtle interactions between mutable states
    - Especially with parallelism/concurrency
- The functional style means that no state is hidden
    - Easier to debug state problems

Implicit vs. Explicit State

- Once again, we see that functional languages can express the same patterns as imperative language
- We can still express stateful computations in functional languages, but it’s *explicit state*
  - You can tell what state there is by looking at the type of a function
  - Nothing is hidden from the
- Bugs in code are often due to subtle interactions between mutable states
  - Especially with parallelism/concurrency
- The functional style means that no state is hidden
  - Easier to debug state problems
    - By hand, or with tools/linters

Abstracting Generative Recursion

Objective

- Goals

Objective

- Goals
  - To see how the pattern of generative recursion on lists can be represented concretely as a higher order function

Objective

- Goals
  - To see how the pattern of generative recursion on lists can be represented concretely as a higher order function
- Key Concepts

Objective

- Goals
  - To see how the pattern of generative recursion on lists can be represented concretely as a higher order function
- Key Concepts
  - `foldl` and `foldr`

Higher-Order Genrative Recursion: Folds

- Recall: higher-order polymorphic functions let us turn design patterns into functions

Higher-Order Genrative Recursion: Folds

- Recall: higher-order polymorphic functions let us turn design patterns into functions
    - e.g. `map` for “do this to each element of a list”

Higher-Order Genrative Recursion: Folds

- Recall: higher-order polymorphic functions let us turn design patterns into functions
  - e.g. `map` for “do this to each element of a list”
- Generative recursion is no exception

Foldl

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function
- Name `foldl` means “fold left”

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function
- Name `foldl` means "fold left"
  - Processes elements in order

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function
- Name `foldl` means "fold left"
    - Processes elements in order
- If list empty, gives whatever the initial `accum` was

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function
- Name `foldl` means “fold left”
  - Processes elements in order
- If list empty, gives whatever the initial `accum` was
- If not empty, uses `f` to update the `accum` based on the first element of the list

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function
- Name `foldl` means “fold left”
    - Processes elements in order
- If list empty, gives whatever the initial `accum` was
- If not empty, uses `f` to update the `accum` based on the first element of the list
    - Then processes the rest of the list with a recursive call

Foldl

```
(define (foldl [f : ('a 'b -> 'b)]
               [accum : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (foldl f (f h accum) t)]))
```

- Generative recursion on a list as a function
- Name `foldl` means “fold left”
  - Processes elements in order
- If list empty, gives whatever the initial `accum` was
- If not empty, uses `f` to update the `accum` based on the first element of the list
  - Then processes the rest of the list with a recursive call
- Tail recursive, so fast and takes constant space

Example: Reverse using folds

Example: Reverse using folds

```
(define (fold-reverse [xs : (Listof 'elType)]) : (Listof 'elType)
  (foldl (lambda (elem accum) (cons elem accum))
         '()
         xs))
(fold-reverse '(1 2 3))
```

Example: Reverse using folds

```
(define (fold-reverse [xs : (Listof 'elType)]) : (Listof 'elType)
  (foldl (lambda (elem accum) (cons elem accum))
         '()
         xs))
(fold-reverse '(1 2 3))
```

```
'(3 2 1)
```

- 'a is `'elType`, 'b is `(Listof 'elType)`

Example: Reverse using folds

```
(define (fold-reverse [xs : (Listof 'elType)]) : (Listof 'elType)
  (foldl (lambda (elem accum) (cons elem accum))
         '()
         xs))
(fold-reverse '(1 2 3))
```

```
'(3 2 1)
```

- `'a` is `'elType`, `'b` is `(Listof 'elType)`
- Update function type is `('elType (Listof 'elType) -> (Listof 'elType))`

Example: Reverse using folds

```
(define (fold-reverse [xs : (Listof 'elType)]) : (Listof 'elType)
  (foldl (lambda (elem accum) (cons elem accum))
         '()
         xs))
(fold-reverse '(1 2 3))
```

```
'(3 2 1)
```

- `'a` is `'elType`, `'b` is `(Listof 'elType)`
- Update function type is `('elType (Listof 'elType) -> (Listof 'elType))`
- Initial accumulator is empty list, since we start with nothing

Example: Reverse using folds

```
(define (fold-reverse [xs : (Listof 'elType)]) : (Listof 'elType)
  (foldl (lambda (elem accum) (cons elem accum))
         '()
         xs))
(fold-reverse '(1 2 3))
```

```
'(3 2 1)
```

- `'a` is `'elType`, `'b` is `(Listof 'elType)`
- Update function type is `('elType (Listof 'elType) -> (Listof 'elType))`
- Initial accumulator is empty list, since we start with nothing
- Update function adds the current element to the front of the list we've built so far

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:
    - Assume that `accum` has the correct result so far

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:
    - Assume that `accum` has the correct result so far
    - Find the `f` that incorporates the next element of the list into the new `accum`

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:
  - Assume that `accum` has the correct result so far
  - Find the `f` that incorporates the next element of the list into the new `accum`
- For `fold-reverse`

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:
  - Assume that `accum` has the correct result so far
  - Find the `f` that incorporates the next element of the list into the new `accum`
- For `fold-reverse`
  - Assume that `accum` has the correct reversal of the list we've seen so far. How do we incorporate the next element?

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:
  - Assume that `accum` has the correct result so far
  - Find the `f` that incorporates the next element of the list into the new `accum`
- For `fold-reverse`
  - Assume that `accum` has the correct reversal of the list we've seen so far. How do we incorporate the next element?
    - We add it to the front, since it's the last of all elements we've seen so far, so the first in the reversed list

Reasoning with foldl

- `(foldl f init '(a b c d))` gives `(f d (f c (f b (f a init))))`
- Similar to normal recursion:
  - Assume that `accum` has the correct result so far
  - Find the `f` that incorporates the next element of the list into the new `accum`
- For `fold-reverse`
  - Assume that `accum` has the correct reversal of the list we've seen so far. How do we incorporate the next element?
    - We add it to the front, since it's the last of all elements we've seen so far, so the first in the reversed list
    - Recursive call says "keep processing for the rest of the list"

Non-tail Folds

Non-tail Folds

```
(define (foldr [f : ('a 'b -> 'b)]
               [init : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (f h (foldr f init t))]))
```

- Like foldr, but processes the list right-to-left

Non-tail Folds

```
(define (foldr [f : ('a 'b -> 'b)]
               [init : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (f h (foldr f init t))]))
```

- Like foldr, but processes the list right-to-left
- `(foldr f init ’(a b c d))` gives `(f a (f b (f c (f d init))))`

Non-tail Folds

```
(define (foldr [f : ('a 'b -> 'b)]
               [init : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (f h (foldr f init t))]))
```

- Like foldr, but processes the list right-to-left
- `(foldr f init '(a b c d))` gives `(f a (f b (f c (f d init))))`
- Reasoning the exact same, except we assume the argument to `f` has the accumulator for all elements *after* the current element

Non-tail Folds

```
(define (foldr [f : ('a 'b -> 'b)]
               [init : 'b]
               [xs : (Listof 'a)]) : 'b
  (type-case (Listof 'a) xs
    [empty
      accum ]
    [(cons h t)
       (f h (foldr f init t))]))
```

- Like foldr, but processes the list right-to-left
- `(foldr f init '(a b c d))` gives `(f a (f b (f c (f d init))))`
- Reasoning the exact same, except we assume the argument to `f` has the accumulator for all elements *after* the current element
- Generally `foldl` is faster