Final Exam Review: Functional Programming

CS 350

Dr. Joseph Eremondi

Last updated: August 13, 2024

Functional Programming: What We Learned

Main Topics

- Purely functional programming

Main Topics

- Purely functional programming
  - No mutable variables

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions
  - Design patterns as functions

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions
  - Design patterns as functions
    - Map, filter, foldr

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions
  - Design patterns as functions
    - Map, filter, foldr
- Tail-recursion

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions
  - Design patterns as functions
    - Map, filter, foldr
- Tail-recursion
  - Bottom-up, rather than top-down recursion

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions
  - Design patterns as functions
    - Map, filter, foldr
- Tail-recursion
  - Bottom-up, rather than top-down recursion
  - Functional version of a loop

Main Topics

- Purely functional programming
  - No mutable variables
    - Mathematical functions: same input means same output
  - Recursion instead of loops
- Algebraic data types
  - “OR” for datatypes
- Higher order functions
  - Functions that take other functions
  - Functions that produce other functions
  - Lambda for creating anonymous functions
- Higher-order polymorphic functions
  - Design patterns as functions
    - Map, filter, foldr
- Tail-recursion
  - Bottom-up, rather than top-down recursion
  - Functional version of a loop
  - foldl

Purely Functional Programming

Expressions

- Our program is made of *expressions*

Expressions

- Our program is made of *expressions*
- An expression that has no free undefined variables evaluates to a *value*

Expressions

- Our program is made of *expressions*
- An expression that has no free undefined variables evaluates to a *value*
- To compute, we write functions, which tell us what the value of an expression should be depending on its variables

Expressions

- Our program is made of *expressions*
- An expression that has no free undefined variables evaluates to a *value*
- To compute, we write functions, which tell us what the value of an expression should be depending on its variables

Expressions

- Our program is made of *expressions*
- An expression that has no free undefined variables evaluates to a *value*
- To compute, we write functions, which tell us what the value of an expression should be depending on its variables

```
(if b trueResult falseResult)
   ;; b is Boolean
   ;;t f have any type, but are the same type
(if #t trueResult falseResult) = trueResult
(if #f trueResult falseResult) = falseResult
```

  - Equations hold for whatever expressions we give in place of `trueResult` and `falseResult`

Expressions

- Our program is made of *expressions*
- An expression that has no free undefined variables evaluates to a *value*
- To compute, we write functions, which tell us what the value of an expression should be depending on its variables

```
(if b trueResult falseResult)
   ;; b is Boolean
   ;;t f have any type, but are the same type
(if #t trueResult falseResult) = trueResult
(if #f trueResult falseResult) = falseResult
```

  - Equations hold for whatever expressions we give in place of `trueResult` and `falseResult`
  - So if `(if b thenResult elseResult)` occurs in a function body, the expression will be equal to `trueResult` if the function is given arguments that make b evaluate to `#t`, and will be equal to `falseResult` otherwise

Functions

- The main way to control computation in functional languages

Functions

- The main way to control computation in functional languages

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`
  - `x2` replaced by the value of `b`

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`
  - `x2` replaced by the value of `b`
  - `x3` replaced by the value of `c`

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`
  - `x2` replaced by the value of `b`
  - `x3` replaced by the value of `c`
- Works for any number of arguments

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`
  - `x2` replaced by the value of `b`
  - `x3` replaced by the value of `c`
- Works for any number of arguments
- Same for lambdas

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`
  - `x2` replaced by the value of `b`
  - `x3` replaced by the value of `c`
- Works for any number of arguments
- Same for lambdas
  - `((lambda (x1 x2 x3) body) a b c)` equal to `body` with `x1,x2,x3` replaced by `a,b,c` values

Functions

- The main way to control computation in functional languages

```
(define (f x1 x2 x3) body)
```

- `x1, x2, x3` are *identifiers*, names we choose for variables when writing the function
- Then for any expressions `a b c`, the function call `(f a b c)` is equal to `body` with:
  - `x1` replaced by the value of `a`
  - `x2` replaced by the value of `b`
  - `x3` replaced by the value of `c`
- Works for any number of arguments
- Same for lambdas
  - `((lambda (x1 x2 x3) body) a b c)` equal to `body` with `x1,x2,x3` replaced by `a,b,c` values
    - e.g. calling a lambda

The Design Process

- Figure out the types of your inputs and outputs

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers
    - Recursion on lists

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers
    - Recursion on lists
    - Tail recursion on lists

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers
    - Recursion on lists
    - Tail recursion on lists
    - Type-case + recursion

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers
    - Recursion on lists
    - Tail recursion on lists
    - Type-case + recursion
    - etc.

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers
    - Recursion on lists
    - Tail recursion on lists
    - Type-case + recursion
    - etc.
- Write your implementation

The Design Process

- Figure out the types of your inputs and outputs
  - Write the signature
  - On the exam, we do this for you
- Write/think of example inputs
- Use a template for the kind of problem you're trying to solve
  - Figure out the different cases you have to handle
    - Write `if` or `type-case` with placeholders on the right hand side
  - Depends on the types/problem
    - e.g. Recursion on numbers
    - Recursion on lists
    - Tail recursion on lists
    - Type-case + recursion
    - etc.
- Write your implementation
- Try it on examples

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*
    - Contain smaller values of *the same type*

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*
    - Contain smaller values of *the same type*
- Look at other ways to build things of the type you want

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*
    - Contain smaller values of *the same type*
- Look at other ways to build things of the type you want
  - Constructors

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*
    - Contain smaller values of *the same type*
- Look at other ways to build things of the type you want
  - Constructors
  - Lambdas

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*
    - Contain smaller values of *the same type*
- Look at other ways to build things of the type you want
  - Constructors
  - Lambdas
  - Helper functions

Figuring out the implementation

- **Look at what you have in scope, and the types of things that are in scope**
  - e.g. If you are trying to make a `'b`, and you have an `'a` and an `('a -> 'b)`, then you can call the function to get a `'b`
- **Look at the type of the result you're trying to produce**
  - If you're trying to produce a `(Listof Number)`, then each case needs to have a result that is a `(Listof Number)`
- Look at what you can get with recursive calls
  - E.g. calling the function you're defining on smaller arguments
  - Specifically, on the sub-values
  - Numbers, lists, define-type are all *recursive data*
    - Contain smaller values of *the same type*
- Look at other ways to build things of the type you want
  - Constructors
  - Lambdas
  - Helper functions
  - Library functions

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem
- If you decide to call a function or a constructor, you have to give it arguments

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem
- If you decide to call a function or a constructor, you have to give it arguments
    - Can put placeholder in until you figure it out

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem
- If you decide to call a function or a constructor, you have to give it arguments
  - Can put placeholder in until you figure it out
- This creates new “goals”, where you know what type it needs to be

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem
- If you decide to call a function or a constructor, you have to give it arguments
  - Can put placeholder in until you figure it out
- This creates new “goals”, where you know what type it needs to be
- Repeat the process

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem
- If you decide to call a function or a constructor, you have to give it arguments
  - Can put placeholder in until you figure it out
- This creates new “goals”, where you know what type it needs to be
- Repeat the process
  - Try to fill in the placeholders using what’s in scope

Filling the holes

- You start trying to make an expression of a certain type, that solves your problem
- If you decide to call a function or a constructor, you have to give it arguments
  - Can put placeholder in until you figure it out
- This creates new “goals”, where you know what type it needs to be
- Repeat the process
  - Try to fill in the placeholders using what’s in scope
  - This may generate new goals, keep repeating until there’s no more left

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies
        - Look at the types of what's in scope

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies
        - Look at the types of what's in scope
        - Look at the type of the expression you're trying to produce

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies
        - Look at the types of what's in scope
        - Look at the type of the expression you're trying to produce
        - Look at what a recursive call produces

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies
        - Look at the types of what's in scope
        - Look at the type of the expression you're trying to produce
        - Look at what a recursive call produces
        - If you're producing a datatype, look at what constructors you have

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies
        - Look at the types of what's in scope
        - Look at the type of the expression you're trying to produce
        - Look at what a recursive call produces
        - If you're producing a datatype, look at what constructors you have
        - Look at what a call to a helper function produces

Doing this for interpreters

- We have designed our languages so that **the different syntactic forms can be implemented independently**
- So to implement an interpreter, you just have to figure out how to implement an interpreter for each syntactic form (e.g. variant of the AST type), assuming that we can call the interpreter on all the sub-expressions of that expression
- On the exam, you're usually only writing a single case of the interpreter
    - Previous slide still applies
        - Look at the types of what's in scope
        - Look at the type of the expression you're trying to produce
        - Look at what a recursive call produces
        - If you're producing a datatype, look at what constructors you have
        - Look at what a call to a helper function produces
        - Look at what a call to a library function produces

(Algebraic) Data Types

OR for types

- Pairs gave us AND for types

OR for types

- Pairs gave us AND for types
  - Value of type `('a * 'b)` contains a value of type `'a` AND a value of type `'b`

OR for types

- Pairs gave us AND for types
  - Value of type `('a * 'b)` contains a value of type `'a` AND a value of type `'b`
- Datatypes give us OR

Datatype Declarations

Datatype Declarations

```
(define-type Building
  (House [numOccupants : Number]
         [streetAddress : String])
  (Store [businessName : String]
         [streetAddress : String]))
```

- A value of type `Building` is either a house OR a store

Datatype Declarations

```
(define-type Building
  (House [numOccupants : Number]
         [streetAddress : String])
  (Store [businessName : String]
         [streetAddress : String]))
```

- A value of type `Building` is either a house OR a store
  - If it's a house, then it has a Number AND a String

Datatype Declarations

```
(define-type Building
  (House [numOccupants : Number]
         [streetAddress : String])
  (Store [businessName : String]
         [streetAddress : String]))
```

- A value of type `Building` is either a house OR a store
  - If it's a house, then it has a Number AND a String
  - If it's a store, then it has a String AND a String

Datatype Declarations

```
(define-type Building
  (House [numOccupants : Number]
         [streetAddress : String])
  (Store [businessName : String]
         [streetAddress : String]))
```

- A value of type `Building` is either a house OR a store
  - If it's a house, then it has a Number AND a String
  - If it's a store, then it has a String AND a String
- Constructors have function types

Datatype Declarations

```
(define-type Building
  (House [numOccupants : Number]
         [streetAddress : String])
  (Store [businessName : String]
         [streetAddress : String]))
```

- A value of type `Building` is either a house OR a store
  - If it's a house, then it has a Number AND a String
  - If it's a store, then it has a String AND a String
- Constructors have function types
  - But no bodies, they just package the data together

Type-case

Type-case

```
(define (address [x : Building]) : String
  (type-case Building x
   [(House n addr)
       addr]
   [(Store nm addr)
      addr]))
```

- Left hand side is the *pattern* that we check the value against

Type-case

```
(define (address [x : Building]) : String
  (type-case Building x
   [(House n addr)
       addr]
   [(Store nm addr)
      addr]))
```

- Left hand side is the *pattern* that we check the value against
  - Choose a name to give to each field

Type-case

```
(define (address [x : Building]) : String
  (type-case Building x
   [(House n addr)
       addr]
   [(Store nm addr)
      addr]))
```

- Left hand side is the *pattern* that we check the value against
  - Choose a name to give to each field
  - Then they're in scope, can use them in the right hand side

Type-case

```
(define (address [x : Building]) : String
  (type-case Building x
   [(House n addr)
       addr]
   [(Store nm addr)
      addr]))
```

- Left hand side is the *pattern* that we check the value against
  - Choose a name to give to each field
  - Then they're in scope, can use them in the right hand side
- Right hand side is the result we produce if it matches that pattern

Recursive Data

- Can have fields of a define-type that are the type being defined

Recursive Data

- Can have fields of a define-type that are the type being defined
- E.g. Lists, Expressions, etc.

Recursive Data

- Can have fields of a define-type that are the type being defined
- E.g. Lists, Expressions, etc.
- Then, functions consuming that type use type-case+recursion

Recursive Data

- Can have fields of a define-type that are the type being defined
- E.g. Lists, Expressions, etc.
- Then, functions consuming that type use type-case+recursion
    - Recursive calls on fields of the same type

Recursive Data

- Can have fields of a define-type that are the type being defined
- E.g. Lists, Expressions, etc.
- Then, functions consuming that type use type-case+recursion
    - Recursive calls on fields of the same type
- e.g. Every interpreter we've written for the last 6 weeks

Higher Order Functions

Lambdas

- The main way to create a function NOT at the top level

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:
  - `(lambda ([x : 'a]) body)`

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:
  - `(lambda ([x : 'a]) body)`
  - `body` has type `'b`, can use the variable `x` of type `'a`

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:
  - `(lambda ([x : 'a]) body)`
  - `body` has type `'b`, can use the variable `x` of type `'a`
- Created dynamically

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:
  - `(lambda ([x : 'a]) body)`
  - `body` has type `'b`, can use the variable `x` of type `'a`
- Created dynamically
  - Can refer to anything that's in scope when you make the function

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:
  - `(lambda ([x : 'a]) body)`
  - `body` has type `'b`, can use the variable `x` of type `'a`
- Created dynamically
  - Can refer to anything that's in scope when you make the function
    - e.g. If you're making a function in another function, can use paramters

Lambdas

- The main way to create a function NOT at the top level
- If you're trying to make something of type `('a -> 'b)`, then you can always do so by writing:
  - `(lambda ([x : 'a]) body)`
  - `body` has type `'b`, can use the variable `x` of type `'a`
- Created dynamically
  - Can refer to anything that's in scope when you make the function
    - e.g. If you're making a function in another function, can use paramters
  - Captures the values from whenever the function was created

Functions Taking Functions

- For the most part, nothing has changed

Functions Taking Functions

- For the most part, nothing has changed
- If you have a variable `f` of type `('a -> 'b)` for any types `'a` and `'b`, and you do a call `(f x)` on `x : 'a`, then the result of the call will have type `'b`

Functions Taking Functions

- For the most part, nothing has changed
- If you have a variable `f` of type `('a -> 'b)` for any types `'a` and `'b`, and you do a call `(f x)` on `x : 'a`, then the result of the call will have type `'b`
    - Extends to multi-argument functions

Functions Taking Functions

- For the most part, nothing has changed
- If you have a variable `f` of type `('a -> 'b)` for any types `'a` and `'b`, and you do a call `(f x)` on `x : 'a`, then the result of the call will have type `'b`
  - Extends to multi-argument functions
- All the same equational rules hold

Functions Taking Functions

- For the most part, nothing has changed
- If you have a variable `f` of type `('a -> 'b)` for any types `'a` and `'b`, and you do a call `(f x)` on `x : 'a`, then the result of the call will have type `'b`
  - Extends to multi-argument functions
- All the same equational rules hold
  - You evaluate a function by replacing its parameter variables with the values of its arguments

Functions Taking Functions

- For the most part, nothing has changed
- If you have a variable `f` of type `('a -> 'b)` for any types `'a` and `'b`, and you do a call `(f x)` on `x : 'a`, then the result of the call will have type `'b`
  - Extends to multi-argument functions
- All the same equational rules hold
  - You evaluate a function by replacing its parameter variables with the values of its arguments
    - Even if that parameter is a function

Functions Taking Functions

- For the most part, nothing has changed
- If you have a variable `f` of type `('a -> 'b)` for any types `'a` and `'b`, and you do a call `(f x)` on `x : 'a`, then the result of the call will have type `'b`
  - Extends to multi-argument functions
- All the same equational rules hold
  - You evaluate a function by replacing its parameter variables with the values of its arguments
    - Even if that parameter is a function
    - Substituting turns a call to a variable in the body, into a call to a named function or a lambda

Higher Order Polymorphism

- How we capture design patterns as functions

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list
  - Filter: get a new list with only elements that function returns true on

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list
  - Filter: get a new list with only elements that function returns true on
  - Foldr: Building a result for a list in terms of the result for the tail of the list

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list
  - Filter: get a new list with only elements that function returns true on
  - Foldr: Building a result for a list in terms of the result for the tail of the list
- Again, nothing has changed, except that the things in scope might have a generic type

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list
  - Filter: get a new list with only elements that function returns true on
  - Foldr: Building a result for a list in terms of the result for the tail of the list
- Again, nothing has changed, except that the things in scope might have a generic type
  - `'a`, `(Listof 'b)`, etc.

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list
  - Filter: get a new list with only elements that function returns true on
  - Foldr: Building a result for a list in terms of the result for the tail of the list
- Again, nothing has changed, except that the things in scope might have a generic type
  - `'a`, `(Listof 'b)`, etc.
  - Again, all the same rules apply as before

Higher Order Polymorphism

- How we capture design patterns as functions
  - Map: apply function to every element of a list
  - Filter: get a new list with only elements that function returns true on
  - Foldr: Building a result for a list in terms of the result for the tail of the list
- Again, nothing has changed, except that the things in scope might have a generic type
  - `'a`, `(Listof 'b)`, etc.
  - Again, all the same rules apply as before
- When we call a function with variables in its type, the compiler figures out what concrete types fill in for the type variables

Tail Recursion

Consider the loop

Consider the loop

```
x = xInit;
for (i = 0; i < n; i = i + 1){
  x = f(x); //for some function f, or just some exp
 }
return x; // or just do something with the final va
```

- Tail recursion lets us express this functionally, and execute it as fast as a loop

Consider the loop

```
x = xInit;
for (i = 0; i < n; i = i + 1){
  x = f(x); //for some function f, or just some exp
 }
return x; // or just do something with the final va
```

- Tail recursion lets us express this functionally, and execute it as fast as a loop

Consider the loop

```
x = xInit;
for (i = 0; i < n; i = i + 1){
  x = f(x); //for some function f, or just some exp
 }
return x; // or just do something with the final va
```

- Tail recursion lets us express this functionally, and execute it as fast as a loop

```
(define (someLoop-helper i x)
  (if (< i n)
      (someFun-helper (i + 1) (f x))
      x))
;; Call the looping function with the start values for i and x
(someLoop-helper 0 xInit)
```

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.
  - `(define (f x) (g x 10))`: tail call

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.
  - `(define (f x) (g x 10))`: tail call
  - `(define (f x) (if x (g x 10) 0))`: tail call

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.
  - `(define (f x) (g x 10))`: tail call
  - `(define (f x) (if x (g x 10) 0))`: tail call
    - Value produced by the branch is the value of the whole `if`, is the return of `f`

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.
  - `(define (f x) (g x 10))`: tail call
  - `(define (f x) (if x (g x 10) 0))`: tail call
    - Value produced by the branch is the value of the whole `if`, is the return of `f`
  - `(define (f x) (+ 1 (g x 10)))`: NOT a tail call

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.
  - `(define (f x) (g x 10))`: tail call
  - `(define (f x) (if x (g x 10) 0))`: tail call
    - Value produced by the branch is the value of the whole `if`, is the return of `f`
  - `(define (f x) (+ 1 (g x 10)))`: NOT a tail call
    - We compute the return of `g` then **do something with it** instead of returning directly

Tail Calls

- If we're in the body of function `f`, and we call function `g`, then the call to `g` is a tail call if the return value of `g` is used as the return value of `f`
- E.g.
  - `(define (f x) (g x 10))`: tail call
  - `(define (f x) (if x (g x 10) 0))`: tail call
    - Value produced by the branch is the value of the whole `if`, is the return of `f`
  - `(define (f x) (+ 1 (g x 10)))`: NOT a tail call
    - We compute the return of `g` then **do something with it** instead of returning directly
    - So not a tail call

Tail Recursion

- Recursion where your only calls are tail calls

Tail Recursion

- Recursion where your only calls are tail calls
- Usually have a helper function

Tail Recursion

- Recursion where your only calls are tail calls
- Usually have a helper function
  - Call the helper with the thing we're iterating over and the initial variable values

Tail Recursion

- Recursion where your only calls are tail calls
- Usually have a helper function
  - Call the helper with the thing we're iterating over and the initial variable values
- The computation happens entirely in **computing new values for the accumulator**

Accumulator

- The argument to the helper function which represents the variable whose values we're changing

Accumulator

- The argument to the helper function which represents the variable whose values we're changing
  - e.g. x in the loop example

Accumulator

- The argument to the helper function which represents the variable whose values we're changing
  - e.g. x in the loop example
- Base case returns the accumulator

Accumulator

- The argument to the helper function which represents the variable whose values we're changing
  - e.g. x in the loop example
- Base case returns the accumulator
- Recursive case computes the "next" value for the accumulator

Accumulator

- The argument to the helper function which represents the variable whose values we're changing
  - e.g. x in the loop example
- Base case returns the accumulator
- Recursive case computes the "next" value for the accumulator
  - Like updating the variable in the loop body

Accumulator

- The argument to the helper function which represents the variable whose values we're changing
  - e.g. x in the loop example
- Base case returns the accumulator
- Recursive case computes the "next" value for the accumulator
  - Like updating the variable in the loop body
  - ONLY thing recursive case does

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
    - called *accumulator*, often named `accum`

General Template

- To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`
  - Write a helper function with one extra argument
- called *accumulator*, often named `accum`
- Same type as return type

General Template

- To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`
  - Write a helper function with one extra argument
- called *accumulator*, often named `accum`
- Same type as return type
- `fHelper : (T1 T2 ... Tn ReturnTy -> ReturnTy)`

General Template

- To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`
  - Write a helper function with one extra argument
- called *accumulator*, often named `accum`
- Same type as return type
- `fHelper : (T1 T2 ... Tn ReturnTy -> ReturnTy)`
- Helper function is recursive

General Template

- To write `f : (Ty1 Ty2 ... TyN -> ReturnTy)`
  - Write a helper function with one extra argument
- called *accumulator*, often named `accum`
- Same type as return type
- `fHelper : (T1 T2 ... Tn 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
- called *accumulator*, often named `accum`
- Same type as return type
- `fHelper : (T1 T2 ... Tn 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
  - called *accumulator*, often named `accum`
  - Same type as return type
  - `fHelper : (T1 T2 ... Tn 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
  - called *accumulator*, often named `accum`
  - Same type as return type
  - `fHelper : (T1 T2 ... Tn 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
    - called *accumulator*, often named `accum`
    - Same type as return type
    - `fHelper : (T1 T2 ... Tn 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

Equational rules

- NOTHING HAS CHANGED!

Equational rules

- NOTHING HAS CHANGED!
- All the rules about functions, substitution, if, lambdas, etc. that we had before still apply

Equational rules

- NOTHING HAS CHANGED!
- All the rules about functions, substitution, if, lambdas, etc. that we had before still apply
- Tail recursion lets us express loops in a *purely functional* way, so we can reason about loops using equations

Foldl

- General form for tail recursion on lists

Foldl

- General form for tail recursion on lists
- Give a function that says what the accumulator should be updated to

Foldl

- General form for tail recursion on lists
- Give a function that says what the accumulator should be updated to
  - Takes element in the list and previous accumulator

Foldl

- General form for tail recursion on lists
- Give a function that says what the accumulator should be updated to
  - Takes element in the list and previous accumulator
- Traverses list from left to right, updating the accumulator