Midterm Review

CS 350

Dr. Joseph Eremondi

Last updated: July 23, 2024

Overview

The Road So Far

- Functional programming

The Road So Far

- Functional programming
  - Recursion and datatypes

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST

The Road So Far

- Functional programming
  - Recursion and datatypes
  - Higher-order functions and lambdas
- Interpreters
  - Parsing basics and syntax trees
  - Arithmetic via recursion on the AST
  - Adding conditionals: `If0`

The Road So Far

- Functional programming
  - Recursion and datatypes
  - Higher-order functions and lambdas
- Interpreters
  - Parsing basics and syntax trees
  - Arithmetic via recursion on the AST
  - Adding conditionals: `If0`
  - Adding syntactic sugar

The Road So Far

- Functional programming
  - Recursion and datatypes
  - Higher-order functions and lambdas
- Interpreters
  - Parsing basics and syntax trees
  - Arithmetic via recursion on the AST
  - Adding conditionals: `If0`
  - Adding syntactic sugar
    - Now have Surface AST and Core AST

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones

The Road So Far

- Functional programming
  - Recursion and datatypes
  - Higher-order functions and lambdas
- Interpreters
  - Parsing basics and syntax trees
  - Arithmetic via recursion on the AST
  - Adding conditionals: `If0`
  - Adding syntactic sugar
    - Now have Surface AST and Core AST
    - `elab` translates from surface to core
    - Expresses some features in terms of other ones
  - Adding functions

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones
    - Adding functions
        - Substitution

The Road So Far

- Functional programming
  - Recursion and datatypes
  - Higher-order functions and lambdas
- Interpreters
  - Parsing basics and syntax trees
  - Arithmetic via recursion on the AST
  - Adding conditionals: `If0`
  - Adding syntactic sugar
    - Now have Surface AST and Core AST
    - `elab` translates from surface to core
    - Expresses some features in terms of other ones
  - Adding functions
    - Substitution
    - Environments

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones
    - Adding functions
        - Substitution
        - Environments
    - Adding variables

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones
    - Adding functions
        - Substitution
        - Environments
    - Adding variables
        - Let via substitution

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones
    - Adding functions
        - Substitution
        - Environments
    - Adding variables
        - Let via substitution
        - Let via environment extension

The Road So Far

- Functional programming
  - Recursion and datatypes
  - Higher-order functions and lambdas
- Interpreters
  - Parsing basics and syntax trees
  - Arithmetic via recursion on the AST
  - Adding conditionals: `If0`
  - Adding syntactic sugar
    - Now have Surface AST and Core AST
    - `elab` translates from surface to core
    - Expresses some features in terms of other ones
  - Adding functions
    - Substitution
    - Environments
  - Adding variables
    - Let via substitution
    - Let via environment extension
  - First-class functions

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones
    - Adding functions
        - Substitution
        - Environments
    - Adding variables
        - Let via substitution
        - Let via environment extension
    - First-class functions
        - Dynamic typing

The Road So Far

- Functional programming
    - Recursion and datatypes
    - Higher-order functions and lambdas
- Interpreters
    - Parsing basics and syntax trees
    - Arithmetic via recursion on the AST
    - Adding conditionals: `If0`
    - Adding syntactic sugar
        - Now have Surface AST and Core AST
        - `elab` translates from surface to core
        - Expresses some features in terms of other ones
    - Adding functions
        - Substitution
        - Environments
    - Adding variables
        - Let via substitution
        - Let via environment extension
    - First-class functions
        - Dynamic typing
        - Closures and environments

Part 1: Functional Programming Concepts

Functional Programming

- Functions *in the mathematical sense*

Functional Programming

- Functions *in the mathematical sense*
    - Same input always produces same output

Functional Programming

- Functions *in the mathematical sense*
    - Same input always produces same output
- Variables are *immutable*

Functional Programming

- Functions *in the mathematical sense*
    - Same input always produces same output
- Variables are *immutable*
    - Once a variable has a value, it keeps that value for its entire scope

Functional Programming

- Functions *in the mathematical sense*
    - Same input always produces same output
- Variables are *immutable*
    - Once a variable has a value, it keeps that value for its entire scope
- The main method of iteration/repetition is recursion

Functional Programming

- Functions *in the mathematical sense*
    - Same input always produces same output
- Variables are *immutable*
    - Once a variable has a value, it keeps that value for its entire scope
- The main method of iteration/repetition is recursion
    - Not loops

Functional Programming

- Functions *in the mathematical sense*
    - Same input always produces same output
- Variables are *immutable*
    - Once a variable has a value, it keeps that value for its entire scope
- The main method of iteration/repetition is recursion
    - Not loops
    - Explicit recursion: a function calling itself on different (smaller) arguments

Functional Programming

- Functions *in the mathematical sense*
  - Same input always produces same output
- Variables are *immutable*
  - Once a variable has a value, it keeps that value for its entire scope
- The main method of iteration/repetition is recursion
  - Not loops
  - Explicit recursion: a function calling itself on different (smaller) arguments
  - Higher-order functions: capture patterns of recursion that can be re-used many times

Programming With Recursion 1: Shape

- Identify the shape of the problem

Programming With Recursion 1: Shape

- Identify the shape of the problem
    - Types of inputs

Programming With Recursion 1: Shape

- Identify the shape of the problem
    - Types of inputs
    - Types of outputs

Programming With Recursion 1: Shape

- Identify the shape of the problem
    - Types of inputs
    - Types of outputs
    - Other assumptions not expressible with types

Programming With Recursion 1: Shape

- Identify the shape of the problem
    - Types of inputs
    - Types of outputs
    - Other assumptions not expressible with types
        - e.g. Can we assume numbers are integers? Positive? etc.

Programming With Recursion 1: Shape

- Identify the shape of the problem
    - Types of inputs
    - Types of outputs
    - Other assumptions not expressible with types
        - e.g. Can we assume numbers are integers? Positive? etc.
        - e.g. Can we assume a list is non-empty

Programming With Recursion 1: Shape

- Identify the shape of the problem
  - Types of inputs
  - Types of outputs
  - Other assumptions not expressible with types
    - e.g. Can we assume numbers are integers? Positive? etc.
    - e.g. Can we assume a list is non-empty
- Now you can write tests, since you know what valid inputs are

Programming With Recursion: Template

- What are the cases you have to handle?

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`

Programming With Recursion: Template

- What are the cases you have to handle?
    - Numbers: `0` vs. `(+ n 1)`
    - List: `empty` vs. `(cons h t)`

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`
  - List: `empty` vs. `(cons h t)`
  - Datatype: each constructor

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`
  - List: `empty` vs. `(cons h t)`
  - Datatype: each constructor
- What are the sub-values for each case?

Programming With Recursion: Template

- What are the cases you have to handle?
    - Numbers: `0` vs. `(+ n 1)`
    - List: `empty` vs. `(cons h t)`
    - Datatype: each constructor
- What are the sub-values for each case?
    - List: head and tail for cons case

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`
  - List: `empty` vs. `(cons h t)`
  - Datatype: each constructor
- What are the sub-values for each case?
  - List: head and tail for cons case
  - Datatype: fields for that constructor

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`
  - List: `empty` vs. `(cons h t)`
  - Datatype: each constructor
- What are the sub-values for each case?
  - List: head and tail for cons case
  - Datatype: fields for that constructor
- Write a template that looks at the input and has a placeholder for each case

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`
  - List: `empty` vs. `(cons h t)`
  - Datatype: each constructor
- What are the sub-values for each case?
  - List: head and tail for cons case
  - Datatype: fields for that constructor
- Write a template that looks at the input and has a placeholder for each case
  - Numbers: branch using `if` or `cond`

Programming With Recursion: Template

- What are the cases you have to handle?
  - Numbers: `0` vs. `(+ n 1)`
  - List: `empty` vs. `(cons h t)`
  - Datatype: each constructor
- What are the sub-values for each case?
  - List: head and tail for cons case
  - Datatype: fields for that constructor
- Write a template that looks at the input and has a placeholder for each case
  - Numbers: branch using `if` or `cond`
  - Lists or datatype: branch using `type-case`

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
    - Recursive case has a sub-value of input

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
    - Recursive case has a sub-value of input
    - Might be many base cases for datatype

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
    - Recursive case has a sub-value of input
    - Might be many base cases for datatype
        - E.g. in `Expr`, `Var x` and `NumLit n` are both base cases

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
    - Recursive case has a sub-value of input
    - Might be many base cases for datatype
        - E.g. in `Expr`, `Var x` and `NumLit n` are both base cases
- For the base case, figure out the solution for that case

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
  - Recursive case has a sub-value of input
  - Might be many base cases for datatype
    - E.g. in `Expr`, `Var x` and `NumLit n` are both base cases
- For the base case, figure out the solution for that case
  - Depends on the problem you're looking at

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
  - Recursive case has a sub-value of input
  - Might be many base cases for datatype
    - E.g. in `Expr`, `Var x` and `NumLit n` are both base cases
- For the base case, figure out the solution for that case
  - Depends on the problem you're looking at
    - E.g. "If my input is `0`, what should `f 0` be?"

Programming With Recursion: Base vs Recursive Case

- Figure out which cases are base cases and which are recursive cases
    - Recursive case has a sub-value of input
    - Might be many base cases for datatype
        - E.g. in `Expr`, `Var x` and `NumLit n` are both base cases
- For the base case, figure out the solution for that case
    - Depends on the problem you're looking at
        - E.g. "If my input is `0`, what should `f 0` be?"
        - E.g. "If my list is empty, how do I apply a function to each element in it?"

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases
    - If a sub-value has the same type, can definitely use it as argument to recursive call

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases
    - If a sub-value has the same type, can definitely use it as argument to recursive call
    - Figure out values for other arguments

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases
    - If a sub-value has the same type, can definitely use it as argument to recursive call
    - Figure out values for other arguments
        - Problem dependent

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases
    - If a sub-value has the same type, can definitely use it as argument to recursive call
    - Figure out values for other arguments
        - Problem dependent
        - E.g. What `env` do we pass in `interp`?

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases
  - If a sub-value has the same type, can definitely use it as argument to recursive call
  - Figure out values for other arguments
    - Problem dependent
    - E.g. What `env` do we pass in `interp`?
  - Might be helpful to use `let*` to store results of recursive calls in variables

Programming With Recursion: Recursive Calls

- Figure out what recursive calls you can make in the recursive cases
    - If a sub-value has the same type, can definitely use it as argument to recursive call
    - Figure out values for other arguments
        - Problem dependent
        - E.g. What `env` do we pass in `interp`?
    - Might be helpful to use `let*` to store results of recursive calls in variables
        - Can always delete if you don’t end up using them in the solution

Programming With Recursion: Putting It Together

- Look at what you have in scope

Programming With Recursion: Putting It Together

- Look at what you have in scope
    - Fields/sub-values of the input

Programming With Recursion: Putting It Together

- Look at what you have in scope
  - Fields/sub-values of the input
  - Results of recursive calls

Programming With Recursion: Putting It Together

- Look at what you have in scope
    - Fields/sub-values of the input
    - Results of recursive calls
    - Other arguments

Programming With Recursion: Putting It Together

- Look at what you have in scope
  - Fields/sub-values of the input
  - Results of recursive calls
  - Other arguments
- Look at the types

Programming With Recursion: Putting It Together

- Look at what you have in scope
  - Fields/sub-values of the input
  - Results of recursive calls
  - Other arguments
- Look at the types
  - Lets you know what you have, and what you need build things together

Programming With Recursion: Putting It Together

- Look at what you have in scope
    - Fields/sub-values of the input
    - Results of recursive calls
    - Other arguments
- Look at the types
    - Lets you know what you have, and what you need build things together
- If you don’t have what you need, ask:

Programming With Recursion: Putting It Together

- Look at what you have in scope
  - Fields/sub-values of the input
  - Results of recursive calls
  - Other arguments
- Look at the types
  - Lets you know what you have, and what you need build things together
- If you don’t have what you need, ask:
  - Could I make a helper function to get what I need?

Programming With Recursion: Putting It Together

- Look at what you have in scope
    - Fields/sub-values of the input
    - Results of recursive calls
    - Other arguments
- Look at the types
    - Lets you know what you have, and what you need build things together
- If you don’t have what you need, ask:
    - Could I make a helper function to get what I need?
        - Helper could also be recursive

Programming With Recursion: Putting It Together

- Look at what you have in scope
    - Fields/sub-values of the input
    - Results of recursive calls
    - Other arguments
- Look at the types
    - Lets you know what you have, and what you need build things together
- If you don’t have what you need, ask:
    - Could I make a helper function to get what I need?
        - Helper could also be recursive
- Compute the result you need for the recursive cases

Programming With Recursion: Putting It Together

- Look at what you have in scope
  - Fields/sub-values of the input
  - Results of recursive calls
  - Other arguments
- Look at the types
  - Lets you know what you have, and what you need build things together
- If you don’t have what you need, ask:
  - Could I make a helper function to get what I need?
    - Helper could also be recursive
- Compute the result you need for the recursive cases
  - Problem-specific

Constructing a value of a given type

- Look at the type of the thing you’re trying to produce

Constructing a value of a given type

- Look at the type of the thing you're trying to produce
  - Some take arguments, some don't

Constructing a value of a given type

- Look at the type of the thing you're trying to produce
    - Some take arguments, some don't
- You must build it using one of:

Constructing a value of a given type

- Look at the type of the thing you're trying to produce
    - Some take arguments, some don't
- You must build it using one of:
    - A variable that is in scope

Constructing a value of a given type

- Look at the type of the thing you’re trying to produce
  - Some take arguments, some don’t
- You must build it using one of:
  - A variable that is in scope
  - A literal/constant

Constructing a value of a given type

- Look at the type of the thing you're trying to produce
  - Some take arguments, some don't
- You must build it using one of:
  - A variable that is in scope
  - A literal/constant
    - e.g. `4` or `#t` or `"hello"`

Constructing a value of a given type

- Look at the type of the thing you're trying to produce
    - Some take arguments, some don't
- You must build it using one of:
    - A variable that is in scope
    - A literal/constant
        - e.g. `4` or `#t` or `"hello"`
    - A function or constructor application

Constructing a value of a given type

- Look at the type of the thing you’re trying to produce
    - Some take arguments, some don’t
- You must build it using one of:
    - A variable that is in scope
    - A literal/constant
        - e.g. `4` or `#t` or `"hello"`
    - A function or constructor application
        - Each argument to the function must then be provided

Constructing a value of a given type

- Look at the type of the thing you're trying to produce
    - Some take arguments, some don't
- You must build it using one of:
    - A variable that is in scope
    - A literal/constant
        - e.g. `4` or `#t` or `"hello"`
    - A function or constructor application
        - Each argument to the function must then be provided
        - Look at their types and repeat this process

Constructing a value of a given type

- Look at the type of the thing you’re trying to produce
    - Some take arguments, some don’t
- You must build it using one of:
    - A variable that is in scope
    - A literal/constant
        - e.g. `4` or `#t` or `"hello"`
    - A function or constructor application
        - Each argument to the function must then be provided
        - Look at their types and repeat this process
- Which you choose depends on what you’re trying to do

Higher Order Functions

Designing Higher Order Functions

- The process is the exact same

Designing Higher Order Functions

- The process is the exact same
    - Except sometimes you have inputs or outputs that are functions

Designing Higher Order Functions

- The process is the exact same
  - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data

Designing Higher Order Functions

- The process is the exact same
    - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data
    - e.g. List or datatype

Designing Higher Order Functions

- The process is the exact same
    - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data
    - e.g. List or datatype
- Might not

Designing Higher Order Functions

- The process is the exact same
    - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data
    - e.g. List or datatype
- Might not
    - e.g. If you're writing `comp : (('b -> 'c) ('a -> 'b) -> ('a -> 'c))` there's nothing to deconstruct/recur on, so you just build the solution directly

Designing Higher Order Functions

- The process is the exact same
  - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data
  - e.g. List or datatype
- Might not
  - e.g. If you’re writing `comp : (('b -> 'c) ('a -> 'b) -> ('a -> 'c))` there’s nothing to deconstruct/recur on, so you just build the solution directly
- Same advice applies:

Designing Higher Order Functions

- The process is the exact same
  - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data
  - e.g. List or datatype
- Might not
  - e.g. If you’re writing `comp : (('b -> 'c) ('a -> 'b) -> ('a -> 'c))` there’s nothing to deconstruct/recur on, so you just build the solution directly
- Same advice applies:
  - Look at the types of what’s in scope

Designing Higher Order Functions

- The process is the exact same
  - Except sometimes you have inputs or outputs that are functions
- Might use recursion if your input is recursive data
  - e.g. List or datatype
- Might not
  - e.g. If you're writing `comp : (('b -> 'c) ('a -> 'b) -> ('a -> 'c))` there's nothing to deconstruct/recur on, so you just build the solution directly
- Same advice applies:
  - Look at the types of what's in scope
  - Look at what you can build using what's in scope