Final Exam Review: Interpreters

CS 350

Dr. Joseph Eremondi

Last updated: August 14, 2024

Interpreters

What We’ve Learned

- Syntax trees to represent programs

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals
  - Top-level function definitions

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals
  - Top-level function definitions
  - Local variable definitions (`letvar`)

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying
- Language features
    - Conditionals
    - Top-level function definitions
    - Local variable definitions (`letvar`)
    - First-class functions via substitution

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals
  - Top-level function definitions
  - Local variable definitions (`letvar`)
  - First-class functions via substitution
- Environment-based interpreters

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals
  - Top-level function definitions
  - Local variable definitions (`letvar`)
  - First-class functions via substitution
- Environment-based interpreters
  - Variables via environments

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying
- Language features
    - Conditionals
    - Top-level function definitions
    - Local variable definitions (`letvar`)
    - First-class functions via substitution
- Environment-based interpreters
    - Variables via environments
    - Lambdas via environments: closures

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying
- Language features
    - Conditionals
    - Top-level function definitions
    - Local variable definitions (`letvar`)
    - First-class functions via substitution
- Environment-based interpreters
    - Variables via environments
    - Lambdas via environments: closures
- Store-passing interpreters

What We’ve Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying
- Language features
    - Conditionals
    - Top-level function definitions
    - Local variable definitions (`letvar`)
    - First-class functions via substitution
- Environment-based interpreters
    - Variables via environments
    - Lambdas via environments: closures
- Store-passing interpreters
    - Boxes

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals
  - Top-level function definitions
  - Local variable definitions (`letvar`)
  - First-class functions via substitution
- Environment-based interpreters
  - Variables via environments
  - Lambdas via environments: closures
- Store-passing interpreters
  - Boxes
  - Mutable Variables

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
  - Compute the value for an expression recursively
- Desugaring
  - E.g. Substitution, Currying
- Language features
  - Conditionals
  - Top-level function definitions
  - Local variable definitions (`letvar`)
  - First-class functions via substitution
- Environment-based interpreters
  - Variables via environments
  - Lambdas via environments: closures
- Store-passing interpreters
  - Boxes
  - Mutable Variables
  - Recursion via mutation

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying
- Language features
    - Conditionals
    - Top-level function definitions
    - Local variable definitions (`letvar`)
    - First-class functions via substitution
- Environment-based interpreters
    - Variables via environments
    - Lambdas via environments: closures
- Store-passing interpreters
    - Boxes
    - Mutable Variables
    - Recursion via mutation
- OOP

What We've Learned

- Syntax trees to represent programs
- Evaluation (`interp`)
    - Compute the value for an expression recursively
- Desugaring
    - E.g. Substitution, Currying
- Language features
    - Conditionals
    - Top-level function definitions
    - Local variable definitions (`letvar`)
    - First-class functions via substitution
- Environment-based interpreters
    - Variables via environments
    - Lambdas via environments: closures
- Store-passing interpreters
    - Boxes
    - Mutable Variables
    - Recursion via mutation
- OOP
- Lazy evaluation

Syntax and the Language Pipeline

Life of a program

- The Language Pipeline:

Source code	$\xrightarrow{\text{parsing}}$	Abstract syntax tree	$\xrightarrow{\textit{translation}}$	Core Syntax	$\xrightarrow{\textit{evaluation}}$	Result
text file	lexing / tokenizing first	data structure	desugaring / compilation	simpler AST / machine code	interpreter, execute on CPU	value, side effects

The Pipeline

- Start with source code

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
    - I won’t ask about parsers on the exam

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
    - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
  - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
  - Direct representation of the syntax of the program

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
  - I won’t ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
  - Direct representation of the syntax of the program
- Desugaring/Elaboration

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
  - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
  - Direct representation of the syntax of the program
- Desugaring/Elaboration
  - Translate the surface syntax tree into core syntax

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
    - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
    - Direct representation of the syntax of the program
- Desugaring/Elaboration
    - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
  - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
  - Direct representation of the syntax of the program
- Desugaring/Elaboration
  - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)
  - The type that the interpreter takes as input

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
    - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
    - Direct representation of the syntax of the program
- Desugaring/Elaboration
    - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)
    - The type that the interpreter takes as input
- Interpreter/Evaluation

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
    - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
    - Direct representation of the syntax of the program
- Desugaring/Elaboration
    - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)
    - The type that the interpreter takes as input
- Interpreter/Evaluation
    - Actually run the code

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
  - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
  - Direct representation of the syntax of the program
- Desugaring/Elaboration
  - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)
  - The type that the interpreter takes as input
- Interpreter/Evaluation
  - Actually run the code
  - Get a value and/or side-effects

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
  - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
  - Direct representation of the syntax of the program
- Desugaring/Elaboration
  - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)
  - The type that the interpreter takes as input
- Interpreter/Evaluation
  - Actually run the code
  - Get a value and/or side-effects
- Value type

The Pipeline

- Start with source code
- Parser translates it into a *syntax tree*
    - I won't ask about parsers on the exam
- Surface Syntax Tree (`SurfaceExpr`)
    - Direct representation of the syntax of the program
- Desugaring/Elaboration
    - Translate the surface syntax tree into core syntax
- Core syntax (`Expr`)
    - The type that the interpreter takes as input
- Interpreter/Evaluation
    - Actually run the code
    - Get a value and/or side-effects
- Value type
    - Whatever the result of the computation is

Interpreter Basics

- Every expression has a *value*

Interpreter Basics

- Every expression has a *value*
- Interpreter computes this value

Interpreter Basics

- Every expression has a *value*
- Interpreter computes this value
    - Can view it as *defining* the semantics of the language

Interpreter Basics

- Every expression has a *value*
- Interpreter computes this value
    - Can view it as *defining* the semantics of the language
- Recursively compute sub-expression values

Interpreter Basics

- Every expression has a *value*
- Interpreter computes this value
  - Can view it as *defining* the semantics of the language
- Recursively compute sub-expression values
  - Use to compute value for the whole expression

Interpreter Basics

- Every expression has a *value*
- Interpreter computes this value
    - Can view it as *defining* the semantics of the language
- Recursively compute sub-expression values
    - Use to compute value for the whole expression
    - Once have a value, then we can use the features of the implementation language

Interpreter Basics

- Every expression has a *value*
- Interpreter computes this value
    - Can view it as *defining* the semantics of the language
- Recursively compute sub-expression values
    - Use to compute value for the whole expression
    - Once have a value, then we can use the features of the implementation language
        - e.g. We can't use Plait `+` on an `Expr`, but once we apply `interp` and get a `Number` we can

Variables and Substitution

- To implement functions, we need a way to specify a program that's parameterized

Variables and Substitution

- To implement functions, we need a way to specify a program that's parameterized
- We use `Symbol` as a Plait type to represent variable names

Variables and Substitution

- To implement functions, we need a way to specify a program that's parameterized
- We use `Symbol` as a Plait type to represent variable names
- Add variant `(Var [x :  Symbol])` to `Expr`

Variables and Substitution

- To implement functions, we need a way to specify a program that's parameterized
- We use `Symbol` as a Plait type to represent variable names
- Add variant `(Var [x :  Symbol])` to `Expr`
- New operation: substitution

Variables and Substitution

- To implement functions, we need a way to specify a program that's parameterized
- We use `Symbol` as a Plait type to represent variable names
- Add variant `(Var [x :  Symbol])` to `Expr`
- New operation: substitution
  - Replace all *free* occurrences of the variable `x` with some expression in another expression

Variables and Substitution

- To implement functions, we need a way to specify a program that's parameterized
- We use `Symbol` as a Plait type to represent variable names
- Add variant `(Var [x :  Symbol])` to `Expr`
- New operation: substitution
    - Replace all *free* occurrences of the variable `x` with some expression in another expression
        - Ignoring bound occurrences gives us *shadowing*

Function Calls

- To implement a function call with substitution:

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value
        - Gives us a parameter symbol and a body

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value
        - Gives us a parameter symbol and a body
    - Evaluate the argument to a value

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value
        - Gives us a parameter symbol and a body
    - Evaluate the argument to a value
        - (If doing strict semantics)

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value
        - Gives us a parameter symbol and a body
    - Evaluate the argument to a value
        - (If doing strict semantics)
    - Replace the parameter symbol with the argument value in the body

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value
        - Gives us a parameter symbol and a body
    - Evaluate the argument to a value
        - (If doing strict semantics)
    - Replace the parameter symbol with the argument value in the body
    - Interpret the replaced body

Function Calls

- To implement a function call with substitution:
    - Evaluate the function to a value
        - Gives us a parameter symbol and a body
    - Evaluate the argument to a value
        - (If doing strict semantics)
    - Replace the parameter symbol with the argument value in the body
    - Interpret the replaced body
        - This is how we say “run the body”

Local Definitions: Intuition

- Consider this C++ program

Local Definitions: Intuition

- Consider this C++ program

Local Definitions: Intuition

- Consider this C++ program

```
if (someCondition){
  cout >> "hello";
  int x = 10;
  while (x > 0){
    cout >> x;
    x = x - 1;
  }
 }
cout >> "goodbye";
```

- Declaring `int x = 10` creates a new variable **whose scope is not obvious from the code**

Local Definitions: Intuition

- Consider this C++ program

```
if (someCondition){
  cout >> "hello";
  int x = 10;
  while (x > 0){
    cout >> x;
    x = x - 1;
  }
 }
cout >> "goodbye";
```

- Declaring `int x = 10` creates a new variable **whose scope is not obvious from the code**
    - We can use `x` anywhere from where it's declared until the end of the `if` block

Local Definitions: Intuition

- Consider this C++ program

```
if (someCondition){
  cout >> "hello";
  int x = 10;
  while (x > 0){
    cout >> x;
    x = x - 1;
  }
 }
cout >> "goodbye";
```

- Declaring `int x = 10` creates a new variable **whose scope is not obvious from the code**
  - We can use `x` anywhere from where it's declared until the end of the `if` block
- The assignment in the `while` does **not** affect the scope

Local Definitions: Intuition

- Consider this C++ program

```
if (someCondition){
  cout >> "hello";
  int x = 10;
  while (x > 0){
    cout >> x;
    x = x - 1;
  }
 }
cout >> "goodbye";
```

- Declaring `int x = 10` creates a new variable **whose scope is not obvious from the code**
  - We can use `x` anywhere from where it's declared until the end of the `if` block
- The assignment in the `while` does **not** affect the scope
  - No equivalent in a purely functional language

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:
  - Evaluate `xExpr` to `xVal`

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:
  - Evaluate `xExpr` to `xVal`
  - Replace `x` with `xVal` in body

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:
    - Evaluate `xExpr` to `xVal`
    - Replace `x` with `xVal` in body
    - Interpret the replaced version of `body`

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:
    - Evaluate `xExpr` to `xVal`
    - Replace `x` with `xVal` in body
    - Interpret the replaced version of `body`
- Similar to:

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:
  - Evaluate `xExpr` to `xVal`
  - Replace `x` with `xVal` in body
  - Interpret the replaced version of `body`
- Similar to:

Local Definitions

- In an expression based language, the scope of a variable is *explicit*

```
{letvar x xExpr body}
```

- The scope of `x` is body
- The value of `x` in body is the value of `xExpr`
- To interpret with substitution, we:
  - Evaluate `xExpr` to `xVal`
  - Replace `x` with `xVal` in body
  - Interpret the replaced version of `body`
- Similar to:

```
if (true){ // make the scope explicit
  int x = xExpr;
  body;
}
```

Dynamic Type Checking

- When we have functions as values, we need a way to

Environments

Why Environments

- What is the *meaning* of the program `{+ x 1}`

Why Environments

- What is the *meaning* of the program `{+ x 1}`
  - With substitution, the program has no meaning until we replace `x` with a value

Why Environments

- What is the *meaning* of the program `{+ x 1}`
  - With substitution, the program has no meaning until we replace `x` with a value
- Environments *give meaning to programs with free variables*

Why Environments

- What is the *meaning* of the program `{+ x 1}`
  - With substitution, the program has no meaning until we replace `x` with a value
- Environments *give meaning to programs with free variables*
  - The program has a value *paramterized over* values for the free variables

Why Environments

- What is the *meaning* of the program `{+ x 1}`
    - With substitution, the program has no meaning until we replace `x` with a value
- Environments *give meaning to programs with free variables*
    - The program has a value *paramterized over* values for the free variables
- Performance:

Why Environments

- What is the *meaning* of the program `{+ x 1}`
    - With substitution, the program has no meaning until we replace `x` with a value
- Environments *give meaning to programs with free variables*
    - The program has a value *paramterized over* values for the free variables
- Performance:
    - Instead of traversing the entire body of the function, just push a binding onto the front of a list

Environments Operations

- Environment is a list of symbol-value pairs

Environments Operations

- Environment is a list of symbol-value pairs
  - For purely functional language

Environments Operations

- Environment is a list of symbol-value pairs
    - For purely functional language
- Operations:

Environments Operations

- Environment is a list of symbol-value pairs
    - For purely functional language
- Operations:
    - Make empty environment

Environments Operations

- Environment is a list of symbol-value pairs
    - For purely functional language
- Operations:
    - Make empty environment
    - Look up the most recent value for a given symbol

Environments Operations

- Environment is a list of symbol-value pairs
  - For purely functional language
- Operations:
  - Make empty environment
  - Look up the most recent value for a given symbol
  - Add a new binding (e.g. add a new value for a given symbol)

Environments Operations

- Environment is a list of symbol-value pairs
    - For purely functional language
- Operations:
    - Make empty environment
    - Look up the most recent value for a given symbol
    - Add a new binding (e.g. add a new value for a given symbol)
        - Like a low-tech hashtable/dictionary/key-value store

Letvar with environments

Letvar with environments

```
{letvar x xExpr body}
```

- Interpret `xExpr` to a value

Letvar with environments

```
{letvar x xExpr body}
```

- Interpret `xExpr` to a value
- Make a new environment with `x` bound to the value of `xExpr`

Closures

- Lambdas let us create functions at any point in the program

Closures

- Lambdas let us create functions at any point in the program
  - Can refer to any in-scope variables

Closures

- Lambdas let us create functions at any point in the program
    - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values

Closures

- Lambdas let us create functions at any point in the program
  - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
  - Happens automatically with substitution

Closures

- Lambdas let us create functions at any point in the program
    - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
    - Happens automatically with substitution
- In an environment-based interpreter, we must **include the environment from when a function was created**

Closures

- Lambdas let us create functions at any point in the program
    - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
    - Happens automatically with substitution
- In an environment-based interpreter, we must **include the environment from when a function was created**
- Functions evaluate to closures

Closures

- Lambdas let us create functions at any point in the program
    - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
    - Happens automatically with substitution
- In an environment-based interpreter, we must **include the environment from when a function was created**
- Functions evaluate to closures
- Data structure with:

Closures

- Lambdas let us create functions at any point in the program
  - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
  - Happens automatically with substitution
- In an environment-based interpreter, we must **include the environment from when a function was created**
- Functions evaluate to closures
- Data structure with:
  - Parameter variable

Closures

- Lambdas let us create functions at any point in the program
    - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
    - Happens automatically with substitution
- In an environment-based interpreter, we must **include the environment from when a function was created**
- Functions evaluate to closures
- Data structure with:
    - Parameter variable
    - Body (where parameter is in scope)

Closures

- Lambdas let us create functions at any point in the program
    - Can refer to any in-scope variables
- When those variables get values, the lambda should use those values
    - Happens automatically with substitution
- In an environment-based interpreter, we must **include the environment from when a function was created**
- Functions evaluate to closures
- Data structure with:
    - Parameter variable
    - Body (where parameter is in scope)
    - Environment from when the function was **created**

Static vs. Dynamic Scope

- Closures implement static scope

Static vs. Dynamic Scope

- Closures implement static scope
    - In a lambda, free variables get their values from where the function was *created*

Static vs. Dynamic Scope

- Closures implement static scope
  - In a lambda, free variables get their values from where the function was *created*
- Dynamic scope:

Static vs. Dynamic Scope

- Closures implement static scope
  - In a lambda, free variables get their values from where the function was *created*
- Dynamic scope:
  - In a lambda, free variables get their values from where the function is *called*

Mutable State

Generative Recursion

- Recall how we used generative/tail recursion to implement loops

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
    - Take an extra argument for the *current state of memory*

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
  - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
  - Take an extra argument for the *current state of memory*
    - Called the **store**

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
    - Take an extra argument for the *current state of memory*
        - Called the **store**
    - Each recursive call is explicitly passed a store

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
    - Take an extra argument for the *current state of memory*
        - Called the **store**
    - Each recursive call is explicitly passed a store
    - `interp` now produces a value *and a store*

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
    - Take an extra argument for the *current state of memory*
        - Called the **store**
    - Each recursive call is explicitly passed a store
    - `interp` now produces a value *and a store*
        - State of memory after interpreting some expression

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
    - Take an extra argument for the *current state of memory*
        - Called the **store**
    - Each recursive call is explicitly passed a store
    - `interp` now produces a value *and a store*
        - State of memory after interpreting some expression
        - Gets passed to the next interpreter call

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
  - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
  - Take an extra argument for the *current state of memory*
    - Called the **store**
  - Each recursive call is explicitly passed a store
  - `interp` now produces a value *and a store*
    - State of memory after interpreting some expression
    - Gets passed to the next interpreter call
  - Sequence of states we pass along determines the execution order

Generative Recursion

- Recall how we used generative/tail recursion to implement loops
    - Updated a variables value by passing the new value as a parameter to the recursive call
- Implementing mutable state in a purely functional language works the same way
    - Take an extra argument for the *current state of memory*
        - Called the **store**
    - Each recursive call is explicitly passed a store
    - `interp` now produces a value *and a store*
        - State of memory after interpreting some expression
        - Gets passed to the next interpreter call
    - Sequence of states we pass along determines the execution order
- We can now view programs as functions from the current environment *and the current state of memory* to values *and states of memory*

Boxes

- Value representing a location in memory

Boxes

- Value representing a location in memory
    - Kind of like pointer

Boxes

- Value representing a location in memory
  - Kind of like pointer
- Create a box

Boxes

- Value representing a location in memory
  - Kind of like pointer
- Create a box
  - Add a **new** location to the store

Boxes

- Value representing a location in memory
  - Kind of like pointer
- Create a box
  - Add a **new** location to the store
- Unbox a value

Boxes

- Value representing a location in memory
    - Kind of like pointer
- Create a box
    - Add a **new** location to the store
- Unbox a value
    - Look up the value for the box’s location in the store

Boxes

- Value representing a location in memory
    - Kind of like pointer
- Create a box
    - Add a **new** location to the store
- Unbox a value
    - Look up the value for the box’s location in the store
- Set a box’s value

Boxes

- Value representing a location in memory
  - Kind of like pointer
- Create a box
  - Add a **new** location to the store
- Unbox a value
  - Look up the value for the box’s location in the store
- Set a box’s value
  - Update the store to have a new value *for the box’s existing location*

Mutable Variables

- One big change:

Mutable Variables

- One big change:
    - Environments store *locations*, not values

Mutable Variables

- One big change:
    - Environments store *locations*, not values
- Each variable lookup now gets a location from the env, and a value from that location in the store

Mutable Variables

- One big change:
    - Environments store *locations*, not values
- Each variable lookup now gets a location from the env, and a value from that location in the store
- Can mutate a variable's value by producing a store with a different value at the variable's location

Recursion

- Can define values self-referentially with mutable state

Recursion

- Can define values self-referentially with mutable state
- Interpret the value for a variable in an environment extended with that variable

Recursion

- Can define values self-referentially with mutable state
- Interpret the value for a variable in an environment extended with that variable
    - Dummy value at the variable’s location in the store

Recursion

- Can define values self-referentially with mutable state
- Interpret the value for a variable in an environment extended with that variable
    - Dummy value at the variable's location in the store
- After interpreting the value, put that value at the variable's location in the store

Recursion

- Can define values self-referentially with mutable state
- Interpret the value for a variable in an environment extended with that variable
    - Dummy value at the variable’s location in the store
- After interpreting the value, put that value at the variable’s location in the store
- Works as long as all recursive uses of the variable are in the body of a lambda

Recursion

- Can define values self-referentially with mutable state
- Interpret the value for a variable in an environment extended with that variable
  - Dummy value at the variable's location in the store
- After interpreting the value, put that value at the variable's location in the store
- Works as long as all recursive uses of the variable are in the body of a lambda
  - Closures don't evaluate their bodies until called

Alternate Models of Execution

OOP

- Pair data (members) with operations on that data (fields)

OOP

- Pair data (members) with operations on that data (fields)
- Self reference

OOP

- Pair data (members) with operations on that data (fields)
- Self reference
- Adding new variants of an interface is an easy (local) change

OOP

- Pair data (members) with operations on that data (fields)
- Self reference
- Adding new variants of an interface is an easy (local) change
- Adding new operations is a hard (global) change