Environments, Binding, and Scope

CS 350

Dr. Joseph Eremondi

Last updated: July 15, 2024

Overview

The Road So Far

- Functional Programming (Plait Tutorial)

The Road So Far

- Functional Programming (Plait Tutorial)
- Parsing (PLAI ch2)

The Road So Far

- Functional Programming (Plait Tutorial)
- Parsing (PLAI ch2)
- Interpreters (PLAI ch3)

The Road So Far

- Functional Programming (Plait Tutorial)
- Parsing (PLAI ch2)
- Interpreters (PLAI ch3)
- Desugaring (PLAI ch4)

The Road So Far

- Functional Programming (Plait Tutorial)
- Parsing (PLAI ch2)
- Interpreters (PLAI ch3)
- Desugaring (PLAI ch4)
- Functions (PLAI ch5)

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly (PLAI ch7)

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly (PLAI ch7)
- Mon: Closures and Environments in Curly (PLAI ch7)

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly (PLAI ch7)
- Mon: Closures and Environments in Curly (PLAI ch7)
- Tues/Wed: lectures (not included on midterm)

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly (PLAI ch7)
- Mon: Closures and Environments in Curly (PLAI ch7)
- Tues/Wed: lectures (not included on midterm)
- Thurs: **MIDTERM**

The Road to Midterm

- Today: Environments in Curly (PLAI ch6)
- Tues: Lambda and First-class Functions in Plait
- Wed: Replacements for Recursion in Plait
- Thurs: Implementing Lambda in Curly (PLAI ch7)
- Mon: Closures and Environments in Curly (PLAI ch7)
- Tues/Wed: lectures (not included on midterm)
- Thurs: **MIDTERM**

Everything up to and including Closures may appear on the midterm

Environments

Functions Review

- Evaluate their argument

Functions Review

- Evaluate their argument
- Lookup the function defn (variable, body)

Functions Review

- Evaluate their argument
- Lookup the function defn (variable, body)
- Replace the parameter variable with the argument value in the body

Functions Review

- Evaluate their argument
- Lookup the function defn (variable, body)
- Replace the parameter variable with the argument value in the body
- Evaluate the result

Functions Review

- Evaluate their argument
- Lookup the function defn (variable, body)
- Replace the parameter variable with the argument value in the body
- Evaluate the result
- If we ever interpret a variable, raise an error

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
    - Not including the time to evaluate the function body

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
    - Not including the time to evaluate the function body
- Substitution is forgetful

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
  - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
  - Not including the time to evaluate the function body
- Substitution is forgetful
  - Just replaces function variable with expression

The Problem

- Each substitution is $\mathcal{O}(n)$ where n is the number of nodes in the function body AST
- This is *in addition* to the cost of actually evaluating the function
    - Very slow!
- Want a way to have $\mathcal{O}(1)$ function calls
    - Not including the time to evaluate the function body
- Substitution is forgetful
    - Just replaces function variable with expression
    - Not very useful for debugging

The Solution: Environments

- Data structure for *deferred substitution*

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs
- Intuition:

The Solution: Environments

- Data structure for *deferred substitution*
  - List of variable/value pairs
- Intuition:
  - Instead of replacing all x with value v, keep a list of replacements you need to do

The Solution: Environments

- Data structure for *deferred substitution*
  - List of variable/value pairs
- Intuition:
  - Instead of replacing all `x` with value `v`, keep a list of replacements you need to do
  - When you interpret `x`, check the environment before raising an error

The Solution: Environments

- Data structure for *deferred substitution*
  - List of variable/value pairs
- Intuition:
  - Instead of replacing all `x` with value `v`, keep a list of replacements you need to do
  - When you interpret `x`, check the environment before raising an error
  - If there's an entry for `x` in the environment, return that

The Solution: Environments

- Data structure for *deferred substitution*
    - List of variable/value pairs
- Intuition:
    - Instead of replacing all x with value v, keep a list of replacements you need to do
    - When you interpret x, check the environment before raising an error
    - If there’s an entry for x in the environment, return that
        - Error otherwise

The Solution: Environments

- Data structure for *deferred substitution*
  - List of variable/value pairs
- Intuition:
  - Instead of replacing all `x` with value `v`, keep a list of replacements you need to do
  - When you interpret `x`, check the environment before raising an error
  - If there's an entry for `x` in the environment, return that
    - Error otherwise
    - Means reference to undefined variable

The Environment Data Structure: Bindings

The Environment Data Structure: Bindings

```
;; Just a pair, but we get better names than fst and snd
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))
```

The Environment Data Structure: Bindings

```
;; Just a pair, but we get better names than fst and snd
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))
```

```
;; Get helper functions from the type-def
(bind-name (bind 'x 3))
(bind-val (bind 'x 3))
```

The Environment Data Structure: Bindings

```
;; Just a pair, but we get better names than fst and snd
(define-type Binding
  (bind [name : Symbol]
        [val : Number]))
```

```
;; Get helper functions from the type-def
(bind-name (bind 'x 3))
(bind-val (bind 'x 3))
```

```
'x
3
```

The Environment Data Structure: Environments

The Environment Data Structure: Environments

```
;; Lets us write Env instead of (Listof Binding)
;; But it's not defining a new type,
;; just a new name for the same type.
(define-type-alias Env (Listof Binding))
;; Environment is either empty or extended env
(define emptyEnv : Env
  empty)
(define (extendEnv [bnd : Binding]
                   [env : Env])
        : Env
  (cons bnd env))

emptyEnv
(extendEnv (bind 'x 3) (extendEnv (bind 'y 4) empty))
```

The Environment Data Structure: Environments

```
;; Lets us write Env instead of (Listof Binding)
;; But it's not defining a new type,
;; just a new name for the same type.
(define-type-alias Env (Listof Binding))
;; Environment is either empty or extended env
(define emptyEnv : Env
  empty)
(define (extendEnv [bnd : Binding]
                   [env : Env])
        : Env
  (cons bnd env))

emptyEnv
(extendEnv (bind 'x 3) (extendEnv (bind 'y 4) empty))
```

```
'()
(list (bind 'x 3) (bind 'y 4))
```

Looking up variables

- Find the **first** binding in the environment

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing
- Just a linear search, like we’ve seen lots already

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing
- Just a linear search, like we’ve seen lots already

Looking up variables

- Find the **first** binding in the environment
    - This is important for shadowing
- Just a linear search, like we've seen lots already

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
   ;; Can't find a variable in an empty env
   [empty (error 'lookup "undefined variable")]
   ;; Cons: check if the first binding is the var
   ;; we're looking for.
   ;; Return its value  if it is, otherwise
   ;; keep looking in the rest of the list
   [(cons b rst-env) (cond
                       [(symbol=? n (bind-name b))
                        (bind-val b)]
                       [else (lookup n rst-env)])]))
```

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment
    - Unlike fundefs, this will *change across recursive calls*

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment
    - Unlike fundefs, this will *change across recursive calls*

Intepreting CurlyFundef with Environments

- We can change the implementation *without changing the surface language*
- Programs should run the exact same in both interpreters
- Strategy: add an extra context argument for Environment
  - Unlike fundefs, this will *change across recursive calls*

```
(define (interp [env : Env]
                [defs : (Listof FunDef)]
                [e : Expr] ) : Number
  (type-case Expr e
             ....))
```

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call
- Other operations are similar

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call
- Other operations are similar

Case: Plus etc.

- Exactly like before, except we have to pass the environment in the recursive call
- Other operations are similar

```
;; {+ e1 e2} evaluates e1 and e2, then adds the results together
    [(Plus l r)
     (+ (interp env defs l) (interp env defs r))]
```

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x

Case: Variable

- Can't return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there's a value *bound* to x
- If there is return it

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it
    - Otherwise, variable not found error

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it
    - Otherwise, variable not found error

Case: Variable

- Can’t return an error, because we might have added a deferred substitution to the environment
- So we look in the environment and see if there’s a value *bound* to x
- If there is return it
    - Otherwise, variable not found error

```
[(Var x)
      (lookup x env)]
```

Case: Function call

- Just like before, we get the function body + variable, and value for argument

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, put a variable-value pair in the environment

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, put a variable-value pair in the environment
    - Called *binding* the variable to its value

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, put a variable-value pair in the environment
    - Called *binding* the variable to its value

Case: Function call

- Just like before, we get the function body + variable, and value for argument
- Still interpret body
- Instead of replacing in body, put a variable-value pair in the environment
    - Called *binding* the variable to its value

```
[(Call funName argExpr)
     (let* ([argVal (interp env defs argExpr)]
             [def (get-fundef funName defs)]
             [argVar (mkFunDef-arg def)]
             [funBody (mkFunDef-body def)])
       (interp (extendEnv (bind argVar argVal) emptyEnv) ;;<------
               defs
               funBody))]
```

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
  - Didn’t extend the additional environment

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
    - Didn’t extend the additional environment
- Want functions to be *abstractions*

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
    - Didn't extend the additional environment
- Want functions to be *abstractions*
    - Should be able to predict how a function behaves from how it's called

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
    - Didn’t extend the additional environment
- Want functions to be *abstractions*
    - Should be able to predict how a function behaves from how it’s called
    - Don’t want result to depend on context, just arguments

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
    - Didn’t extend the additional environment
- Want functions to be *abstractions*
    - Should be able to predict how a function behaves from how it’s called
    - Don’t want result to depend on context, just arguments
- Gives the same results as substitution.

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
  - Didn’t extend the additional environment
- Want functions to be *abstractions*
  - Should be able to predict how a function behaves from how it’s called
  - Don’t want result to depend on context, just arguments
- Gives the same results as substitution.
- We call this *static scoping*

Static Scoping

- We evaluated the body of the function in the empty environment, plus a value for its variable
    - Didn't extend the additional environment
- Want functions to be *abstractions*
    - Should be able to predict how a function behaves from how it's called
    - Don't want result to depend on context, just arguments
- Gives the same results as substitution.
- We call this *static scoping*
- If we extend the environment from the call site, we get *dynamic scoping*

Static Scoping Definition

- A language has *static scoping* if undefined variables in a term get their values from the environment where the function is *defined*

Static Scoping Definition

- A language has *static scoping* if undefined variables in a term get their values from the environment where the function is *defined*
    - Right now, variables come from top-level functions, so undefined variables are always an error

Static Scoping Definition

- A language has *static scoping* if undefined variables in a term get their values from the environment where the function is *defined*
    - Right now, variables come from top-level functions, so undefined variables are always an error
    - We'll see more of this later

Static Scoping Definition

- A language has *static scoping* if undefined variables in a term get their values from the environment where the function is *defined*
  - Right now, variables come from top-level functions, so undefined variables are always an error
  - We'll see more of this later
- A language has *dynamic scoping* if undefined variables get their values from the point where the function is *called*

Static Scoping Exapmple

```
{define {f x} {+ x y}}
{define {g y} {f y}}
{g 3}
```

- Static scoping says this is an error

Static Scoping Exapmple

```
{define {f x} {+ x y}}
{define {g y} {f y}}
{g 3}
```

- Static scoping says this is an error
    - No value for `y` in body of `f`

Static Scoping Exapmple

```
{define {f x} {+ x y}}
{define {g y} {f y}}
{g 3}
```

- Static scoping says this is an error
  - No value for `y` in body of `f`
- Dynamic scoping produces 6

Static Scoping Exapmple

```
{define {f x} {+ x y}}
{define {g y} {f y}}
{g 3}
```

- Static scoping says this is an error
  - No value for `y` in body of `f`
- Dynamic scoping produces 6
  - Looks up `y` from `g` when evaluating `f`

Static Scoping Exapmple

```
{define {f x} {+ x y}}
{define {g y} {f y}}
{g 3}
```

- Static scoping says this is an error
  - No value for `y` in body of `f`
- Dynamic scoping produces 6
  - Looks up `y` from `g` when evaluating `f`
- Dynamic scoping is WRONG

Static Scoping Exapmple

```
{define {f x} {+ x y}}
{define {g y} {f y}}
{g 3}
```

- Static scoping says this is an error
    - No value for `y` in body of `f`
- Dynamic scoping produces 6
    - Looks up `y` from `g` when evaluating `f`
- Dynamic scoping is WRONG
    - You should understand it, but know that static scoping is what we want

Implementing Let

Curly-Let

- New language: Curly-Let

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`
        - Gives `x` the value `e1` in the expression `e2`

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
  - `{letvar x e1 e2}`
    - Gives `x` the value `e1` in the expression `e2`
    - Called letvar so we don't confuse with plait

Curly-Let

- New language: Curly-Let
- Curly-Fundef, but with one new feature
    - `{letvar x e1 e2}`
        - Gives `x` the value `e1` in the expression `e2`
        - Called letvar so we don't confuse with plait
- We'll implement with both substitution and environments

Abstract Syntax

Abstract Syntax

```
(type-def Expr
  ....
  [(Letvar [x : Symbol]
           [xval : Expr]
           [body : Expr])]
          )
```

- Parsing and Desugaring are the same as usual

Abstract Syntax

```
(type-def Expr
  ....
  [(Letvar [x : Symbol]
           [xval : Expr]
           [body : Expr])]
          )
```

- Parsing and Desugaring are the same as usual
    - See `Curly-Let.rkt`

Interpreting: Substitution

- Want variable to have the given value in the body

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once
    - Can have exponential speedup in some algorithms

Interpreting: Substitution

- Want variable to have the given value in the body
    - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once
    - Can have exponential speedup in some algorithms

Interpreting: Substitution

- Want variable to have the given value in the body
  - So just substitute the value for the variable in the body
- Key detail: expression only evaluated once
  - Can have exponential speedup in some algorithms

```
(define (interp [defs : (Listof FunDef)] ;;NEW
                [e : Expr] ) : Number
  (type-case Expr e
        ;; ....
     [(LetVar x xexp body)
        (interp defs (subst x (NumLit (interp defs xexp)) body))])
```

Substituting in a Let Expression

- `{letvar x e1 e2}` *binds* x in e2

Substituting in a Let Expression

- `{letvar x e1 e2}` *binds* x in e2
- So when substituting in `e2` we don't ever replace x

Substituting in a Let Expression

- `{letvar x e1 e2}` *binds* x in e2
- So when substituting in `e2` we don't ever replace x
- Implements shadowing

Substituting in a Let Expression

- `{letvar x e1 e2}` *binds* x in e2
- So when substituting in `e2` we don't ever replace x
- Implements shadowing

Substituting in a Let Expression

- `{letvar x e1 e2}` *binds* x in e2
- So when substituting in `e2` we don't ever replace x
- Implements shadowing

```
(define (subst [toReplace : Symbol]
               [replacedBy : Expr]
               [replaceIn : Expr]) : Expr
  (type-case Expr replaceIn
    [(LetVar x xexp body)
       (LetVar x
               (subst toReplace replacedBy xexp)
               (if (symbol=? x toReplace)
                   body
                   (subst toReplace replacedBy body)))]))
```

Interpreting: Environments

- Interpret the variable’s value in the current environment

Interpreting: Environments

- Interpret the variable’s value in the current environment
- Interpret the let body in the *extended* environment

Interpreting: Environments

- Interpret the variable's value in the current environment
- Interpret the let body in the *extended* environment
- When we hit `x` we'll look in the env

Interpreting: Environments

- Interpret the variable's value in the current environment
- Interpret the let body in the *extended* environment
- When we hit `x` we'll look in the env

Interpreting: Environments

- Interpret the variable's value in the current environment
- Interpret the let body in the *extended* environment
- When we hit x we'll look in the env

```
(define (interp [env : Env]
                [defs : (Listof FunDef)]
                [e : Expr] ) : Number
  (type-case Expr e
        ;; ....
     [(Letvar x xexp body)
      (let ([xval (interp env defs xexp)])
        (interp (extendEnv (bind x xval) env)
                defs body))])
```

Scope

- Notice that we only add to the env to interp `body`, not `xexp`

Scope

- Notice that we only add to the env to interp `body`, not `xexp`
- This is because `x` is *in scope* for `body` but not `xexp`

Scope

- Notice that we only add to the env to interp `body`, not `xexp`
- This is because `x` is *in scope* for `body` but not `xexp`
- We say the **scope of a variable** is the part of the program in which its value is either substituted or bound

Scope

- Notice that we only add to the env to interp `body`, not `xexp`
- This is because `x` is *in scope* for `body` but not `xexp`
- We say the **scope of a variable** is the part of the program in which its value is either substituted or bound
- Let *extends* scope by adding a variable, while calls *transfer* scope to the function

Scope

- Notice that we only add to the env to interp `body`, not `xexp`
- This is because `x` is *in scope* for `body` but not `xexp`
- We say the **scope of a variable** is the part of the program in which its value is either substituted or bound
- Let *extends* scope by adding a variable, while calls *transfer* scope to the function
- Later we'll see more complex examples of scope

Scope

- Notice that we only add to the env to interp `body`, not `xexp`
- This is because `x` is *in scope* for `body` but not `xexp`
- We say the **scope of a variable** is the part of the program in which its value is either substituted or bound
- Let *extends* scope by adding a variable, while calls *transfer* scope to the function
- Later we'll see more complex examples of scope
    - e.g. in Plait, `let` and `let*` have different rules for what's in scope

The Stack

- Environments have a *stack* structure

The Stack

- Environments have a *stack* structure
    - Push on new bindings when variables are defined

The Stack

- Environments have a *stack* structure
  - Push on new bindings when variables are defined
  - Don’t directly pop, but will sometimes interpret in the unextended environment

The Stack

- Environments have a *stack* structure
  - Push on new bindings when variables are defined
  - Don't directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that's already in the environment, we say we *shadow* the old binding

The Stack

- Environments have a *stack* structure
    - Push on new bindings when variables are defined
    - Don't directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that's already in the environment, we say we *shadow* the old binding
    - `lookup` always takes the most recent binding

The Stack

- Environments have a *stack* structure
    - Push on new bindings when variables are defined
    - Don’t directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that’s already in the environment, we say we *shadow* the old binding
    - `lookup` always takes the most recent binding
- Part of the *call stack*

The Stack

- Environments have a *stack* structure
  - Push on new bindings when variables are defined
  - Don’t directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that’s already in the environment, we say we *shadow* the old binding
  - `lookup` always takes the most recent binding
- Part of the *call stack*
  - Theoretical in Curly

The Stack

- Environments have a *stack* structure
  - Push on new bindings when variables are defined
  - Don’t directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that’s already in the environment, we say we *shadow* the old binding
  - `lookup` always takes the most recent binding
- Part of the *call stack*
  - Theoretical in Curly
  - Actually implemented for most languages

The Stack

- Environments have a *stack* structure
  - Push on new bindings when variables are defined
  - Don't directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that's already in the environment, we say we *shadow* the old binding
  - `lookup` always takes the most recent binding
- Part of the *call stack*
  - Theoretical in Curly
  - Actually implemented for most languages
  - Every time we call a function or define a variable we push onto the call stack

The Stack

- Environments have a *stack* structure
    - Push on new bindings when variables are defined
    - Don’t directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that’s already in the environment, we say we *shadow* the old binding
    - `lookup` always takes the most recent binding
- Part of the *call stack*
    - Theoretical in Curly
    - Actually implemented for most languages
    - Every time we call a function or define a variable we push onto the call stack
- Curly-Let and others have an *implicit* call stack

The Stack

- Environments have a *stack* structure
  - Push on new bindings when variables are defined
  - Don’t directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that’s already in the environment, we say we *shadow* the old binding
  - `lookup` always takes the most recent binding
- Part of the *call stack*
  - Theoretical in Curly
  - Actually implemented for most languages
  - Every time we call a function or define a variable we push onto the call stack
- Curly-Let and others have an *implicit* call stack
  - We don’t keep the data structure ourselves

The Stack

- Environments have a *stack* structure
    - Push on new bindings when variables are defined
    - Don’t directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that’s already in the environment, we say we *shadow* the old binding
    - `lookup` always takes the most recent binding
- Part of the *call stack*
    - Theoretical in Curly
    - Actually implemented for most languages
    - Every time we call a function or define a variable we push onto the call stack
- Curly-Let and others have an *implicit* call stack
    - We don’t keep the data structure ourselves
    - Instead, call `interp` recursively to add to the *Plait* call stack

The Stack

- Environments have a *stack* structure
    - Push on new bindings when variables are defined
    - Don't directly pop, but will sometimes interpret in the unextended environment
- If we bind a value to a variable that's already in the environment, we say we *shadow* the old binding
    - `lookup` always takes the most recent binding
- Part of the *call stack*
    - Theoretical in Curly
    - Actually implemented for most languages
    - Every time we call a function or define a variable we push onto the call stack
- Curly-Let and others have an *implicit* call stack
    - We don't keep the data structure ourselves
    - Instead, call `interp` recursively to add to the *Plait* call stack
    - When finished eval, plait returns is to part waiting for the result

Shadowing Example

- `{letvar x 3 {letvar x 4 {+ 3 x}}}`

Shadowing Example

- `{letvar x 3 {letvar x 4 {+ 3 x}}}`
    - Looks at most recent definition

Shadowing Example

- `{letvar x 3 {letvar x 4 {+ 3 x}}}`
    - Looks at most recent definition
    - So should be 7

Shadowing Example

- `{letvar x 3 {letvar x 4 {+ 3 x}}}`
    - Looks at most recent definition
    - So should be 7
- Substitution: `subst` doesn't replace x in `{+ 3 x}` because it is bound

Shadowing Example

- `{letvar x 3 {letvar x 4 {+ 3 x}}}`
    - Looks at most recent definition
    - So should be 7
- Substitution: `subst` doesn't replace `x` in `{+ 3 x}` because it is bound
- Environments: `(x,4)` is at the top of the environment, so `interp` of `x` finds `4`

Design Choices

- There are *high level design choices* for programming languages

Design Choices

- There are *high level design choices* for programming languages
    - How to deal with variable name collisions (shadowing)

Design Choices

- There are *high level design choices* for programming languages
    - How to deal with variable name collisions (shadowing)
    - How to deal with undefined variables (static vs. dynamic scope)

Design Choices

- There are *high level design choices* for programming languages
    - How to deal with variable name collisions (shadowing)
    - How to deal with undefined variables (static vs. dynamic scope)
- Decisions **are made concrete** in the implementation

Design Choices

- There are *high level design choices* for programming languages
  - How to deal with variable name collisions (shadowing)
  - How to deal with undefined variables (static vs. dynamic scope)
- Decisions **are made concrete** in the implementation
  - Behavior of `subst` on bound variables

Design Choices

- There are *high level design choices* for programming languages
    - How to deal with variable name collisions (shadowing)
    - How to deal with undefined variables (static vs. dynamic scope)
- Decisions **are made concrete** in the implementation
    - Behavior of `subst` on bound variables
    - What environment `interp` is passed for function bodies