CS345 Lambda Calculus Project

By
Levi Wiseman
Heng Xiong

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Three types of expressions:

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Three types of expressions:

- **Variables**

Three types of expressions:

- **Variables**
- **Function applications**

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Three types of expressions:

- Variables
- Function applications
- Function abstractions

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= Var x]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Defines scope recursively

```
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Defines scope recursively

No free vars is closed

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
           where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Defines scope recursively

No free vars is closed

FV(x) = {x}

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Defines scope recursively

No free vars is closed

FV(x) = {x}

FV(M N) = FV(M) U FV(N)

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Defines scope recursively

No free vars is closed

$$FV(x) = \{x\}$$

$$FV(M\ N) = FV(M) \cup FV(N)$$

$$FV(\lambda x.M) = FV(M) - \{x\}$$

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

E[V := E'] replaces all free occurences of V by E'

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
    show (Var x) = x
    show (App [illegible]
    show (Lambda [illegible]

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

E[V := E'] replaces all free occurences of V by E'

$$x[x := N] \equiv N$$

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

E[V := E'] replaces all free occurences of V by E'

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ "." ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce e = e

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

$x[x := N] \equiv N$

$y[x := N] \equiv y$, if $x \neq y$

E[V := E'] replaces all free occurences of V by E'

```
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ [illegible] ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

$x[x := N] \equiv N$

$y[x := N] \equiv y$, if $x \neq y$

$(M1\ M2)[x := N] \equiv (M1[x := N])\ (M2[x := N])$

E[V := E'] replaces all free occurences of V by E'

$x[x := N] \equiv N$

$y[x := N] \equiv y$, if $x \neq y$

$(M1\ M2)[x := N] \equiv (M1[x := N])\ (M2[x := N])$

$(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$, if $x \neq y$ and $y \notin FV(N)$

```
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
    show (Var x) = x
    show (App x y) = ... show x ++ " " ++ show y ++ ...
    show (Lambda x e) = "\\" ++ x ++ "." ++ show e ++ ""

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce e = e

-- return list of normal order reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
    where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Captures the idea of function application

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Captures the idea of function application
Corresponds to a computational step

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)

instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Captures the idea of function application

Corresponds to a computational step

$((\lambda V.E)\ E') \equiv E[V := E']$

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
         show (Var x) = x
         show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
         show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Captures the idea of function application

Corresponds to a computational step

((λV.E) E') ≡ E[V := E']

'substitute' alpha-renames free variables if needed

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
    show (Var x) = x
    show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
    show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Captures the idea of function application

Corresponds to a computational step

$((\lambda V.E)\ E') \equiv E[V := E']$

'substitute' alpha-renames free variables if needed

Consider $(\lambda x.y)[y := x]$

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)

instance Show Expr where
  show (Var x) = x
  show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
  show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Normal order

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Normal order
Outermost redex always reduced first

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in an ...
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Normal order

Outermost redex always reduced first

Call by need (lazy evaluation) is memoized

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a term (x := r)
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

```haskell
import Data.List
import System

-- abstract syntax tree
data Expr = Var String
          | App Expr Expr
          | Lambda String Expr deriving (Eq, Read)
instance Show Expr where
        show (Var x) = x
        show (App x y) = "(" ++ show x ++ " " ++ show y ++ ")"
        show (Lambda x e) = "(\\" ++ x ++ " -> " ++ show e ++ ")"

-- return a list of free vars in an expression
freeVars :: Expr -> [Expr]
freeVars x@(Var _) = [x]
freeVars (App t s) = freeVars t ++ freeVars s
freeVars (Lambda x t) = [candidate | candidate <- freeVars t, candidate /= (Var x)]

-- substitute r for x in a, i.e. a[x := r]
substitute :: Expr -> Expr -> Expr -> Expr
substitute r (Var x) (Var y) | x == y = r
                             | otherwise = Var y
substitute r x@(Var _) (App t s) = App (substitute r x t) $ substitute r x s
substitute r (Var x) (Lambda y t) | x == y = Lambda y t
                                  | (Var y) `elem` freeVars r = Lambda (y ++ "'") $ substitute r (Var x) $ substitute (Var $ y ++ "'") (Var $ y ++ "'") (Var y) t
                                  | otherwise = Lambda y $ substitute r (Var x) t

-- does alpha reduction via substitute when necessary
betaReduce :: Expr -> Expr
betaReduce (App (Lambda x t) s) = substitute s (Var x) t
betaReduce (App t s) = App (betaReduce t) $ betaReduce s
betaReduce (Lambda x t) = Lambda x $ betaReduce t
betaReduce r = r

-- return list of equivalent reductions to an expression's normal form
normalReduce :: Expr -> [Expr]
normalReduce r = unfoldr reduce r
                 where reduce r = let reduced = betaReduce r in if r == reduced then Nothing else Just (reduced, reduced)

-- get the normal form
normalize :: Expr -> Expr
normalize = last . normalReduce
```

Church Numerals

- Def: Church Numerals are functions of two arguments, where the first argument is applied a number of times to the second, the number of times it is applied is the integer representation.
- Example:
  - zero = $\lambda fx.x$
  - one = $\lambda fx.f\ x$
  - two = $\lambda fx.f\ (f\ x)$
  - …

Library Functions

- We built the library functions based on Peano Axioms
- Started with inc(increment):
  - App inc zero = one
  - App inc one = two
  - …
- plus:
  - App(App plus two) three = five
    - Currying: App plus two is a function of one argument
    - Binded two to plus first, then applied with three

Library Function con.

- Booleans in Lambda Calculus
  - true = λxy.x
  - false = λxy.y
- Now we can do more functions with booleans
  - isZero
    - App isZero zero = λxy.x (true)
    - App isZero one = λxy.y (false)
  - ifThenElse
    - App (App (App ifThenElse true) (Var "42")) (Var "666")) = 42
    - App (App (App ifThenElse true) (Var "42")) (Var "666")) = 666

Big Cool Factorial

- Before we do factorial, we need to handle recursion
  - A fixed point combinator (Y combinator) is a higher-order function that computes a fixed point of functions.
    - (0)^2 = 0, therefore 0 is a fixed point of f(x) = x^2
- factorial = fix (λfx.ifThenElse (isZero x) 1 (mult x (f (dec x))))
  - App factorial three = six
  - App factorial four = twenty-four
    - Without normalize, it will look like …
    - Hence, thanks to normalization!

Aftermath

- We’ve shown that Lambda Calculus ≤ Haskell’s type system, therefore we have trivially proved that Haskell’s type system is Turing complete!

FIN