# CSEP 505: Programming Languages

Lecture 4
January 29, 2015

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

```
…
  errorMsg ++
   (if isSevere then "!!!" else "")
…
```

```
…
   errorMsg ++
    (if isSevere then “!!!” else “”)
…
⇔
…
   if isSevere
   then errorMsg ++ “!!!”
   else errorMsg ++ []
…
```

# Induction: Mathematical

1. Prove p(0).

2. Prove that p(n) $\Rightarrow$ p(n+1).

3. Deduce that p(n) for all n >= 0.

4. Profit.

# Induction: Structural (e.g., lists)

1. Prove p(`[]`).
2. Prove that p(`xs`) ⇒ p(`x:xs`) (for arbitrary `x`).
3. Deduce that p(`xs`) for all `xs :: [a]`.
4. Profit.

1. `[] ++ ys = ys`
2. `(x:xs) ++ ys = x:(xs ++ ys)`

To prove `xs ++ [] = xs`:

Base case: `[] ++ [] = []` (by 1., `ys = []`)

Induction step:

Assume `xs ++ [] = xs`. Then:

```
(x:xs) ++ [] = x:(xs ++ [])   (by 2., ys = [])
             = x:xs           (by the induction hypothesis)
```

```
(("hello" ++ " ") ++ "world") ++ "!"

"hello" ++ (" " ++ ("world" ++ "!"))
```

1. `[] ++ ys = ys`  2. `(x:xs) ++ ys = x:(xs ++ ys)`

To prove `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`:

Base case: `([] ++ ys) ++ zs = ys ++ zs` (by 1.)

`= [] ++ (ys ++ zs)` (by 1.)

Induction step:

Assume `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`. Then:

```
((x:xs) ++ ys) ++ zs = (x:(xs ++ ys)) ++ zs
                     = x:((xs ++ ys) ++ zs)
                     = x:(xs ++ (ys ++ zs))
                     = (x:xs) ++ (ys ++ zs)
```

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]

revappend :: [a] -> [a] -> [a]
revappend [] ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
```

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten (xs:xss) = xs ++ (flatten xss)
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x):(map f xs)
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter pred (x:xs) | pred x = x:(filter pred xs)
                   | otherwise = filter pred xs
```

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
foldl f acc [1, 2, 3, 4, 5] =
  (f (f (f (f (f acc 1) 2) 3) 4) 5)

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ init [] = init
foldr f init (x:xs) = f x (foldr f init xs)
foldr f init [1, 2, 3, 4, 5] =
  (f 1 (f 2 (f 3 (f 4 (f 5 init)))))
```

```
sum = foldl (+) 0
product = foldl (*) 1

g . f = \x -> g (f x)
flip f y x -> f x y

(++) = flip (foldr (:))
reverse = foldl (flip (:)) []
map f = foldr ((:) . f) []
```

```
interp :: Expr → Env → Val
data Val = NumV Integer
         | BoolV Bool
         | FunV Var Expr Env


type Env = [(Var, Val)]
```

```
interp (FunE var body) env =
  FunV var body env



interp (AppE fun arg) env =
  let fv = interp fun env
      av = interp arg env in
  case fv of
    FunV var body closEnv ->
      interp body ((var, av):env)
```

`interp (FunE var body) env =`<br>
`  FunV` ~~`var body env`~~<br>
`    (\ av -> interp body ((var, av):env))`

`interp (AppE fun arg) env =`<br>
`  let fv = interp fun env`<br>
`      av = interp arg env in`<br>
`  case fv of`<br>
`    FunV` ~~`var body closEnv`~~ `fn -> fn av`<br>
`      ` ~~`interp body ((var, av):env)`~~

```
type Env = [(Var, Val)]

getEnv :: Env -> Var -> Maybe Val
getEnv env var = lookup var env

emptyEnv = []

extendEnv var val env = (var, val):env
```

```
type Env = Var -> Maybe Val

getEnv :: Env -> Var -> Maybe Val
getEnv env var = env var

emptyEnv _ = Nothing

extendEnv var val env var'
   | var == var' = Just val
   | otherwise = env var'
```

```
e ::= n
    | true | false
    | (if e e e)
    | x
    | (fun (x) e)
    | (e e)
```

```
data Expr = NumE Integer
          | BoolE Bool
          | IfE Expr Expr Expr
          | VarE Var
          | FunE Var Expr
          | AppE Expr Expr
```

```
if = λcte.cte
true = λte.t
false = λte.e

0 = λsz.z
1 = λsz.sz
2 = λsz.s(sz)
succ = λnsz.s(nsz)
succ 3 = λsz.s(3sz)
       = λsz.s((λxy.x(x(xy)))sz)
       = λsz.s(s(s(sz)))
```

```
add n m = λsz.(n s (m s z))
mult n m = λsz.(n (m s) z)
mult 2 3 = λsz.(2 (3 s) z)
         = λsz.(2 (λa.s(s(sa))) z)
         = λsz.((λxy.x(xy)) (λa.s(s(sa))) z)
         = λsz.((λy.(λa.s(s(sa)))
                ((λa.s(s(sa)))y)) z)
         = λsz.((λy.(λa.s(s(sa)))
                (s(s(sy))) z)
         = λsz.((λy.s(s(s(s(s(sy)))))) z)
         = λsz.s(s(s(s(s(sz)))))
```

e ::= *x*
| (**fun** (*x*) e)
| (e e)

```
data Expr = VarE Var
          | FunE Var Expr
          | AppE Expr Expr
```

```
fact =

    (λ (n)
      (if (zero? n)
          1
          (mult n (fact (sub1 n)))))
```

```
fact =
  (λ (fact)
    (λ (n)
      (if (zero? n)
          1
          (mult n (fact (sub1 n))))))
```

```
(fact fact) =

      (λ (n)
        (if (zero? n)
            1
            (mult n ((λ (fact)
                        (λ (n)
                          (if … )))
                      (sub1 n)))))
```

```
 fact =
(… (λ (fact)
     (λ (n)
       (if (zero? n)
           1
           (mult n (fact (sub1 n)))))))
```

```
 fact =
(y (λ (fact)
     (λ (n)
       (if (zero? n)
           1
           (mult n (fact (sub1 n)))))))
```

```
fact =
  (λ (fact)
    (λ (n)
      (if (zero? n)
          1
          (mult n (fact (sub1 n))))))
```

```
mkfact =
    (λ (mkfact)
      (λ (n)
        (if (zero? n)
            1
            (mult n (mkfact
                      (sub1 n))))))
```

```
mkfact =
    (λ (mkfact)
      (λ (n)
        (if (zero? n)
            1
            (mult n ((mkfact mkfact)
                     (sub1 n))))))
```

```
(mkfact mkfact) =

      (λ (n)
        (if (zero? n)
            1
            (mult n ((λ (n)
                       (if (zero? n)
                           1
                           ((λ (mkfact) (λ(n)…))
                            (λ (mkfact) …))))
                     (sub1 n)))))
```

```
fact =

          (y (λ (fact)
               (λ (n)
                 (if (zero? n)
                     1
                     (mult n (fact (sub1 n)))))))
```

```
fact =

      (… (λ (mkfact)
           ((λ (fact)
              (λ (n)
                (if (zero? n)
                    1
                    (mult n (fact (sub1 n))))))
            (mkfact mkfact))))
```

```
fact =
 (with [mkfact
        (λ (mkfact)
          ((λ (fact)
             (λ (n)
               (if (zero? n)
                   1
                   (mult n (fact (sub1 n))))))
           (mkfact mkfact)))]
   (mkfact mkfact))
```

```
y f =
 (with [mkfact
          (λ (mkfact)
            (f




              (mkfact mkfact)))]
   (mkfact mkfact))
```

```
y f =
 (with [mkfact
           (λ (mkfact)
             (f (mkfact mkfact)))]
   (mkfact mkfact))
```

```
y f =
 (with [h
          (λ (g)
            (f (g g)))]
   (h h))
```

```
y f = ((λ (g) (f (g g)))
       (λ (g) (f (g g))))
```

```
y f = ((λ (g) (f (g g)))
         (λ (g) (f (g g))))

y f = (f ((λ (g) (f (g g)))
            (λ (g) (f (g g)))))
    = f (y f)
```

```
y f = ((λ (g) (f (g g)))
         (λ (g) (f (g g))))
    = (f ((λ (g) (f (g g)))
            (λ (g) (f (g g)))))
    = f (y f)


y f = ((λ (g) (f (λ (x) ((g g) x))))
         (λ (g) (f (λ (x) ((g g) x)))))
```

# Concepts

- Structural induction
- Function-based rep’n of FunV, environment
- λ-calculus, Church encodings
- Recursion via Y-combinator