# CSEP 505: Programming Languages

Lecture 2
January 15, 2015

```
op ::= + | * | ...
e ::= n
    | true | false
    | (op e e)
    | (if e e e)
```

```
data Op = Add | Mul | ...
data Expr = NumE Integer
          | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
```

```
(if false
    3
    (+ 4 5))
```

```
IfE (BoolE False)
  (NumE 3)
  (OpE Add (NumE 4) (NumE 5))
```

```
"(if false\n3\n(+ 4 5))"

  (, if, false, 3, (, +, 4, 5, ), )

  [if, false, 3, [+, 4, 5]]
```

```
IfE (BoolE False)
  (NumE 3)
  (OpE Add (NumE 4) (NumE 5))
```

```
(* (+ 1 2) (+ 1 2))
```

```
(with [x (+ 1 2)]
  (* (+ 1 2) (+ 1 2)))
```

```
(with [x (+ 1 2)]
  (* x x))
```

```
op ::= + | * | ...
e ::= n
    | true | false
    | (op e e)
    | (if e e e)
```

```
data Op = Add | Mul | …
data Expr = NumE Int
          | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
```

```
op ::= + | * | ...
e ::= n
    | true | false
    | (op e e)
    | (if e e e)
    | x
    | (with [x e] e)
```

```
data Op = Add | Mul | …
data Expr = NumE Int
          | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
          | VarE Var
          | WithE Var Expr Expr
```

```
(with [x (+ 1 2)]
  (* x x))
=> 9
```

```
(with [x (+ 1 x)]
  (* x x))
=> <error>
```

```
(with [x (+ 1 2)]
  (+ x (with [y (* x x)]
         (+ x y))))
=> 15
```

```
(with [x (+ 1 2)]
  (+ x (with [x (* x x)]
         (+ x x))))
=> 21
```

```
(with [x (+ y 2)]
  (+ x 4))
=> <error>
```

```
(with [x (+ y 2)]
  (with [y 4]
    (+ x y)))
=> <error>
```

```
interp :: Expr -> Expr
interp expr = case expr of
  NumE _ -> expr
  ...
  WithE var boundExpr body ->
    interp (subst var (interp boundExpr) body)
  VarE var -> error (var ++ ": unbound")
  FunE _ _ -> expr
  AppE fun arg -> case interp fun of
    FunE var body ->
      interp (subst var (interp arg) body)
```

```
data Expr = NumE Int | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
          | VarE Var
          | WithE Var Expr Expr
```

```
subst :: Var -> Expr -> Expr -> Expr
subst v boundExpr body =
  let recur = subst v boundExpr in
  case body of
    NumE _ -> body
    ...
    VarE v' | v == v' -> boundExpr
            | otherwise -> body
    WithE v' boundExpr' body'
      | v == v'    -> WithE v' (recur boundExpr') body'
      | otherwise -> WithE v' (recur boundExpr') (recur body')
    FunE v' body' | v == v' -> body
                  | otherwise -> Fun v' (recur body)
    AppE fun arg -> AppE (recur fun) (recur arg)
```

```
data Expr = NumE Int | …
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
          | VarE Var
          | WithE Var Expr Expr
```

```
(+ (* 3 3) (* 4 4))
```

```
(with [sqr (fun (x) (* x x))]
  (+ (* 3 3) (* 4 4)))
```

```
(with [sqr (fun (x) (* x x))]
  (+ (sqr 3) (sqr 4)))
```

```
((fun (x) (* x x))
 (+ 1 2))
=> 9
```

```
((fun (f) (f (f 3)))
 (fun (x) (+ x 2)))
=> 7
```

```
((fun (x)
    (fun (f) (f (f x))))
 2)
=> (fun (f) (f (f 2)))
```

```
(+ 3 (fun (x) (+ x x)))
=> <error>
```

```
(with [f (fun (x) (+ x y))]
  (with [y 5]
    (f y)))
=> <error>
```

```
(with [y 2]
  (with [f (fun (x) (+ x y))]
    (with [y 5]
      (f y))))
=> 7
```

```
((fun (x) (* x x))
 (+ 1 2))

(with [x (+ 1 2)]
  (* x x))
```

```
((fun (x) <body>)
 <arg>)

(with [x <arg>]
  <body>)
```

```
(with [x 2]
  (with [y 3]
    (+ (* (+ 3 4) (+ 1 2))
       (+ (* 2 x) (* y 3)))))
```

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

emptyEnv = []

extendEnv var val env =

lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

```
data Expr = NumE Int | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
          | VarE Var
          | WithE Var Expr Expr
          | FunE Var Expr
          | AppE Expr Expr
```

```
data Expr = NumE Int | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
          | VarE Var
          | FunE Var Expr
          | AppE Expr Expr
```

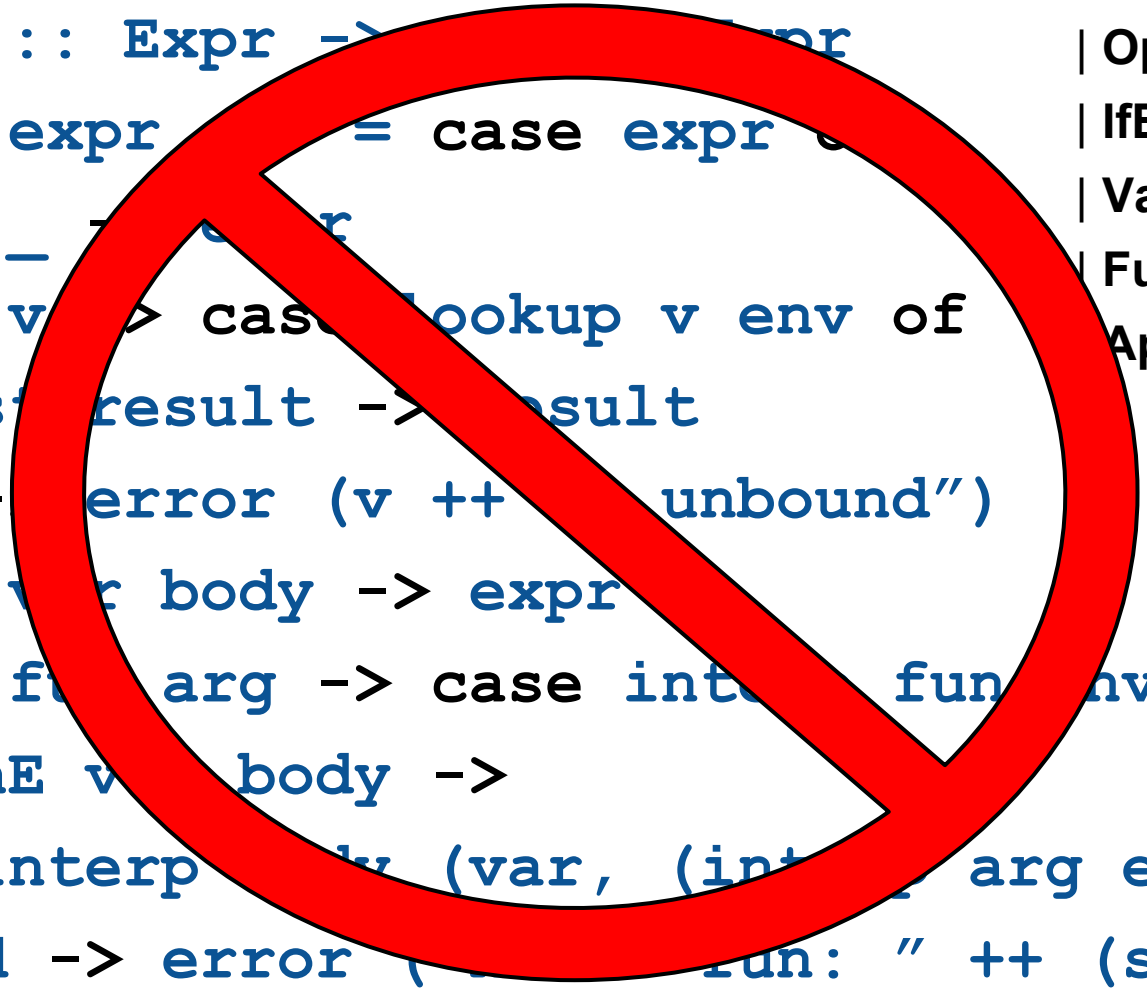
```
interp :: Expr -> Env -> Expr
interp expr env = case expr of
  NumE _ -> expr
  VarE v -> case lookup v env of
    Just result -> result
    _ -> error (v ++ ": unbound")
  FunE var body -> expr
  AppE fun arg -> case interp fun env of
    FunE var body ->
      interp body (var, (interp arg env)):env
    bad -> error ("not a fun: " ++ (show bad))
```

```
(with [y 2]
  (with [f (fun (x) (+ x y))]
    (with [y 5]
      (f y))))
```

=> 10 (!)

```
data Expr = NumE Int | BoolE Bool
          | OpE Op Expr Expr
          | IfE Expr Expr Expr
          | VarE Var
          | FunE Var Expr
          | AppE Expr Expr
```

```
interp :: Expr ->
interp expr = case expr
  NumE _ -
  VarE v -> cas lookup v env of
    Just result -> result
    _ - error (v ++ unbound")
  FunE r body -> expr
  AppE f arg -> case int fun nv of
    FunE v body ->
      interp (var, (int arg env)):env
    bad -> error ( un: " ++ (show bad))
```



```
interp :: Expr -> Env -> Val
interp expr env = case expr of
  NumE n -> NumV n
  VarE v -> case lookup v env of
    Just result -> result
    _ -> error (v ++ ": unbound")
  FunE var body -> FunV var body env
  AppE fun arg -> case interp fun env of
    FunV v body closEnv ->
      interp body ((v, interp arg env):closEnv)
    bad -> error ("not a fun: " ++ (show bad))
```

```
data Expr = NumE Int | ...
          | VarE Var
          | FunE Var Expr
          | AppE Expr Expr
data Val = NumV Int
         | BoolV Bool
         | FunV Var Expr Env
```

```
(with [y 2]
  (with [f (fun (x) (+ x y))]
    (with [y 5]
      (f y))))
=> 7
```

```
(with [f (fun (x) (* x x))]
  (f (+ 1 2)))
```

```
(with [f (fun (_) (* 3 4))]
  (f (+ 1 2)))
```

```
(with [f (fun (_) (* 3 4))]
  (f (/ 1 0)))
```

```
int i = 0;
someFunction(++i);
printf("i = %d\n", i);
```

```
λ> :t readFile
readFile :: FilePath -> IO String
```

# Concepts

- Substitution (β-reduction)
- Lexical scope
- α-renaming
- Call-by-{value, name, need} (eager vs. lazy)
- λ-abstraction (anonymous functions)
- Desugaring
- Environments
- Closures