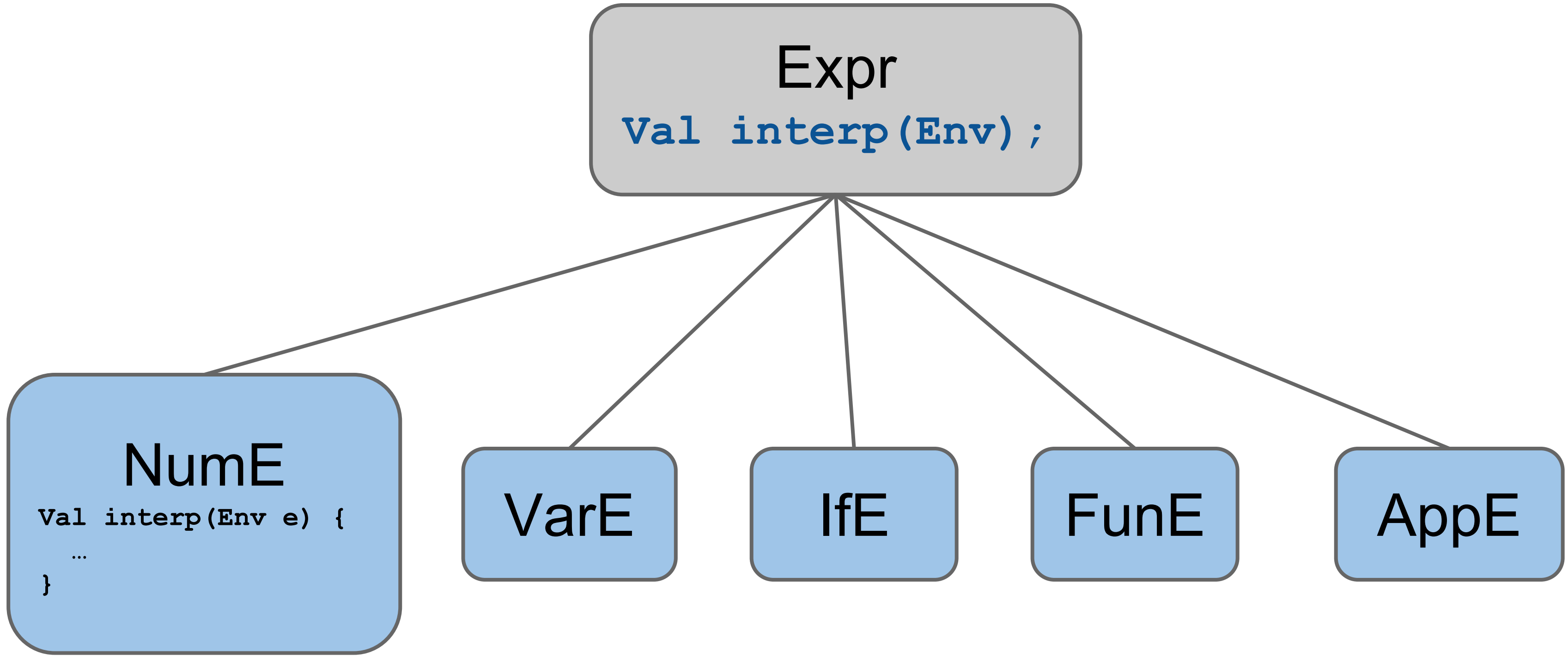
CSEP 505: Programming Languages

Lecture 5
February 5, 2015

```
data Val = NumV Int
         | BoolV Bool | …

data Expr = NumE Int
          | IfE Expr Expr Expr
          | VarE Var
          | FunE Var Expr
          | AppE Expr Expr

data Env = EmptyEnv
         | NonEmptyEnv Var Val Env
```

```
data Val = NumV Int | …
data Expr = NumE Int | …
data Env = EmptyEnv | …

lookup :: Env → Var → Maybe Val

interp :: Expr → Env → Result Val

…
```


Expr
Val interp(Env);
NumE
Val interp(Env e) {
…
}
VarE
IfE
FunE
AppE

```
abstract class Env {
  abstract Val lookup(String var);
}

abstract class Expr {
  abstract Val interp(Env env);
}

abstract class Val {
  …
}
```

```
class EmptyEnv extends Env {
  Val lookup(String var) {
    return null;
  }
}
```

```
class NonEmptyEnv extends Env {
  final String var;
  final Val val;
  final Env restEnv;
  NonEmptyEnv(String var, Val val, Env env) { … }
  Val lookup(String var) {
    return var.equals(this.var) ? val :
        restEnv.lookup(var);
  }
}
```

```
class NumE extends Expr {
  final int n;
  NumE(int n) {
    this.n = n;
  }
  Val interp(Env env) {
    return new NumV(n);
  }
}
```

```
class VarE extends Expr {
  final String name;
  VarE(String name) {
    this.name = name;
  }
  Val interp(Env env) {
    Val val = env.lookup(name);
    if (val == null) throw new UnboundVarExn(…);
    return val;
  }
}
```

```
class FunE extends Expr {
  final String var;
  final Expr body;
  FunE(String var, Expr body) {
    this.var = var;
    this.body = body;
  }
  Val interp(Env env) {
    return new FunV(var, body, env);
  }
}
```

```
class AppE extends Expr {
  final Expr fun, arg;
  AppE(Expr fun, Expr arg) {
    this.fun = fun; …
  }
  Val interp(Env env) {
    Val fv = fun.interp(env);
    Val av = arg.interp(env);
    fv.apply(av);
  }
}
```

```
abstract class Val {
  Val apply(Val arg) {
    throw new UnsupportedOpExn(…);
  }
}
```

```
class FunV extends Val {
  final String var;
  final Expr body;
  final Env env;
  FunV(String var, Expr body, Env env) { … }
  Val apply(Val arg) {
    newEnv = new NonEmptyEnv(var, arg, env);
    return body.interp(newEnv);
  }
}
```

```
class IsZeroV extends Val {
  Val apply(Val arg) {
    if (arg instanceof NumV) {
      int n = ((NumV) arg).getIntValue();
      return (n == 0) ? BoolV.TRUE
            : new BoolV.FALSE;
    }
  }
}
```

```
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)))))
```

```
(fix (fun (fact)
       (fun (n)
         (if (zero? n)
             1
             (* n (fact (sub1 n)))))))
```

```
interp (FixE expr) env =
  case interp expr env of
    FunV var body closEnv ->
      let result = interp body ((var, result):closEnv) in
      result
    nonFun -> error ((show nonFun) ++ ": not a function")
```

```
(with [fact            …            ]
    (fact 5)))
```

```
(with [fact (fun (n) (+ n false))]









    (fact 5)))
```

```
(with [fact (fun (n) (+ n false))]

  (seq
    (set! fact (fun (n)
                 (if (zero? n)
                     1
                     (* n (fact (sub1 n))))))

    (fact 5)))
```

```
[] ++ ys = ys          (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
reverse (1:2:3:[])
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
(reverse (2:3:[])) ++ [1]
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
((reverse (3:[])) ++ [2]) ++ [1]
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
(((reverse []) ++ [3]) ++ [2]) ++ [1]
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
            (([] ++ [3]) ++ [2]) ++ [1]
```

```
[] ++ ys = ys          (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
                   ([3] ++ [2]) ++ [1]
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
                (3:([] ++ [2])) ++ [1]
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
                                (3:[2]) ++ [1]
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
                                    3:([2] ++ [1])
```

```
[] ++ ys = ys          (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
                       3:(2:([] ++ [1]))
```

```
[] ++ ys = ys        (x:xs) ++ ys = x:(xs ++ ys)
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
                          3:(2:[1])
```

```
revappend [] ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
revappend (1:2:3:[]) []
```

```
revappend [] ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
revappend (2:3:[]) (1:[])
```

```
revappend [] ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
revappend (3:[]) (2:1:[])
```

```
revappend [] ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
revappend [] (3:2:1:[])
```

```
revappend [] ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
(3:2:1:[])
```

```
append [] ys = ys
append (x:xs) ys = x:(append xs ys)

append (1:2:3:[]) [4, 5, 6]
    = 1:(append (2:3:[]) [4, 5, 6])
    = 1:2:(append (3:[]) [4, 5, 6])
    = 1:2:3:(append [] [4, 5, 6])
    = 1:2:3:[4, 5, 6]
    = [1, 2, 3, 4, 5, 6]
```

```
append [] ys = ys
append (x:xs) ys  = x:(append xs ys)
```

```
append [] ys = ys
append (x:xs) ys  = x: append xs ys (x:)
```

```
append [] ys pre = pre ys
append (x:xs) ys pre = append xs ys (pre . (x:))
```

```
appendK [] ys k   =   k   ys
appendK (x:xs) ys k   = appendK xs ys (k . (x:))
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

appendK (1:2:3:[]) [4, 5, 6]
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

appendK (1:2:3:[]) [4, 5, 6] id
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

appendK (2:3:[]) [4, 5, 6] (id.(1:))
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

appendK (3:[]) [4, 5, 6] (id.(1:).(2:))
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

appendK [] [4, 5, 6] (id.(1:).(2:).(3:))
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

(id.(1:).(2:).(3:)) [4, 5, 6]
```

```
appendK [] ys k = k ys
appendK (x:xs) ys k = appendK xs ys (k . (x:))

(1:2:3:[4, 5, 6])
```

```
append :: [a] -> [a] -> [a]

appendK :: [a] -> [a] -> ([a] -> b) -> b
appendK [1, 2, 3] [4, 5, 6] …
```

```
product [] = 1

product (n:ns) = n * product ns

product [4, 3, … , 2, 1, 7, … ]
  = 4 * 3 * … * 2 * 1 * 7 * …
```

```
product [] = 1

product (n:ns) = n * product ns

product [4, 3, … , 2, 0, 7, … ]
  = 4 * 3 * … * 2 * 0 * 7 * …
```

```
product [] = 1
product (0:_) = 0
product (n:ns) = n * product ns

product [4, 3, … , 2, 0, 7, … ]
  = 4 * 3 * … * 2 * 0 * 7 * …
```

```
product [] = 1
product (0:_) = 0
product (n:ns) = n * product ns

product [4, 3, … , 2, 0, 7, … ]
  = 4 * 3 * … * 2 * 0
```

```
productK [] k = k 1
productK (0:_) k = k 0
productK (n:ns) k = productK ns (k . (n*))

productK [4, 3, … , 2, 0, 7, … ] id
  = 4 * 3 * … * 2 * 0
```

```
productK [] k = k 1
productK (0:_) k = 0
productK (n:ns) k = productK ns (k . (n*))

productK [4, 3, … , 2, 0, 7, … ] id
  = 0
```

```
parseExprK :: SExp → (Expr → Result a) → Result a
parseExprK sexp k = case sexp of
  NumS n → k (NumE n)
  IdS name → k (VarE name)
  ListS [IdS "if", cond, cond, alt] →
    parseExprK cond (\cond′ →
      parseExprK cons (\cons′ →
        parseExprK alt (\alt′ → k (IfE cond′ cons′ alt′))
  ListS [IdS "fun", arg, body] →
    case arg of
      ListS [IdS arg′] → parseExprK body (\body′ → … )
      bad → Err ((show bad) ++ ": not a valid arg")

parseExprK sexp Ok ⇒ …
```

```
interpK :: Expr -> Env -> (Val -> Res a) -> Res a
interpK expr env k = case expr of
  NumE n -> k (NumV n)
  FunE var body -> k (FunV var body env)
  IfE cond cons alt ->
    interp cond env (\v ->
      case v of
        BoolV True -> interp cons env k
        BoolV False -> interp alt env k
        nonBool -> Err … )
```

```
interpK :: Expr -> Env -> (Val -> Res a) -> Res a
interpK expr env k = case expr of
  AppE fun arg ->
    interp fun env (\fv ->
      interp arg env (\av ->
        case fv of
          FunV var body closEnv ->
            interp body ((var, av):closEnv) k
          PrimV fn -> fn av k
          nonFun -> Err … ))
```

```
(define fact
  (fix (fun (fact n)

      (if (zero? n)
          1
          (* n (fact (sub1 n)))))))

(fact 5)
```

```
(define fact
  (fix (fun (fact n)
    (pause
      (if (zero? n)
          1
          (* n (fact (sub1 n))))))))

(fact 5)
```

e ::= n
  | (**if** e e e)
  | *x*
  | (**fun** (*x*) e)
  | (e e)
  | (**pause** e)

data Expr = **NumE** Integer
  | **IfE** Expr Expr Expr
  | **VarE** Var
  | **FunE** Var Expr
  | **AppE** Expr Expr
  | **PauseE** Expr

```
interp :: Expr -> Env -> (Val -> Result Val) -> .
interp expr env k = case expr of
  PauseE body -> PauseV body env k
```

```
interp :: Expr -> Env -> (Val ->        ) ->
interp expr env k = case expr of
  LetCcE var body ->
    interp body ((var, ContV k):env) k
  AppE fun arg -> … -- as before
    case fv of
      FunV … -> -- as before
      PrimV … -> -- as before
      ContV k′ -> k′ av
```

```
(define foldr
  (fix (fun (foldr f init lst)
    (if (empty? lst)
        init
        (f (first lst)
           (foldr f init (rest lst)))))))

(foldr * 1 lst)
```

```
(define foldr
  (fix (fun (foldr f init lst)
    (if (empty? lst)
        init
        (f (first lst)
           (foldr f init (rest lst)))))))
(let/cc esc
  (foldr (fun (n acc)
           (if (zero? n)
               (esc 0)
               (* n acc)))) 1 lst)
```

```
(define fib
  (fix
    (fun (fib a b count)
      (if (zero? count)
          a
          (fib b (+ a b) (sub1 count))))))

(fib 1 1 10) => 89
```

```
(define fib
  (fix
    (fun (fib a b yield)
      (with* ([yield (let/cc k
                       (yield (pair a k)))])
        (fib b (+ a b) yield)))))
```

```
(define fib
  (fix
    (fun (fib a b yield)
      (with* ([yield (let/cc k
                       (yield (pair a k)))])
        (fib b (+ a b) yield)))))

(with* ([p0 (let/cc yield (fib 2 3 yield))]
        [p1 (let/cc yield ((snd p1) yield))]
        [p2 (let/cc yield ((snd p2) yield))])
  (fst p2))
```

```
(define fib
  (fix
    (fun (fib a b yield)
      (with* ([yield (let/cc k
                       (yield (pair a k)))])
        (fib b (+ a b) yield)))))
(with* ([next (fun (g) (let/cc esc (g esc)))]
        [p0 (next (fib 3 5))]
        [p1 (next (snd p1))]
        [p2 (next (snd p2))])
  (fst p2))
```

Concepts

- Object-based representations
- Recursion without Y-combinator
- Continuation-passing style
- Continuations as a language feature