CSEP 505: Programming Languages

Lecture 3

January 22, 2015

op ::= **+** | ***** | …	data Op = **Add** | **Mul** | …
e ::= n	data Expr = **NumE** Integer
| **true** | **false**	| **BoolE** Bool
| (op e e)	| **OpE** Op Expr Expr
| (**if** e e e)	| **IfE** Expr Expr Expr
| *x*	| **VarE** Var
| (**fun** (*x*) e)	| **FunE** Var Expr
| (e e)	| **AppE** Expr Expr
| (**with** (*x* e) e)	| **WithE** Var Expr Expr

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


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

```
(fn e1 e2) ⇔
((fn e1) e2)

(fun (x y) <body>) ⇔
(fun (x) (fun (y) <body>))
```

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

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

```
(with [sqr (fun (y) (* y y))]
  (with [sum-of-squares
         (fun (x y)
              (+ (sqr x) (sqr y)))]
    (with [farther?
           (fun (x1 y1 x2 y2)
                (> (sum-of-squares x1 y1)
                   (sum-of-squares x2 y2)))]
      (farther? 2 5 3 4))))
```

```
(define sqr
  (fun (y) (* y y)))
(define sum-of-squares
  (fun (x y)
    (+ (sqr x) (sqr y))))
(define farther?
  (fun (x1 y1 x2 y2)
    (> (sum-of-squares x1 y1)
       (sum-of-squares x2 y2))))
```

```
e ::= n                    data Expr = NumE Integer
    | (if e e e)                     | IfE Expr Expr Expr
    | x                              | VarE Var
    | (fun (x) e)                    | FunE Var Expr
    | (e e)                          | AppE Expr Expr
```

```
interp :: Expr -> Env -> (Val, Env)
interp (DefineE var expr) env =
   case lookup var env of
      Just val -> error (var ++ “: already def’d”)
      Nothing ->
         let (val, env’) = interp expr env in
         (val, (var, val):env’)
```

```
int x = 0;
y = f(x);
x = x + 1;
z = f(x);
```

```
(with [x 0]
  (set! x (+ x 1)))
```

```
(with [x 0]
  (seq (set! x (+ x 1))
       x))
```

```
(with [cur 0]
  (with [inc!
      (fun (delta)
        (seq
          (set! cur (+ cur delta))
          cur))]
    (seq (inc! 3)
         (inc! 5))))
```

```
(with [make-counter
    (fun (init)
      (with [cur init]
        (fun (delta)
          (seq
            (set! cur (+ cur delta))
            cur))))]
  (with [count! (make-counter 0)]
    (seq (count! 3) (count! 5))))
```

```
(with [make-counter
    (fun (init)
      (with [cur init]
        (fun (delta)
          (seq
            (set! cur (+ cur delta))
            cur))))]
  (with [count! (make-counter 0)]
    (+ (count! 3) (count! 5))))
```

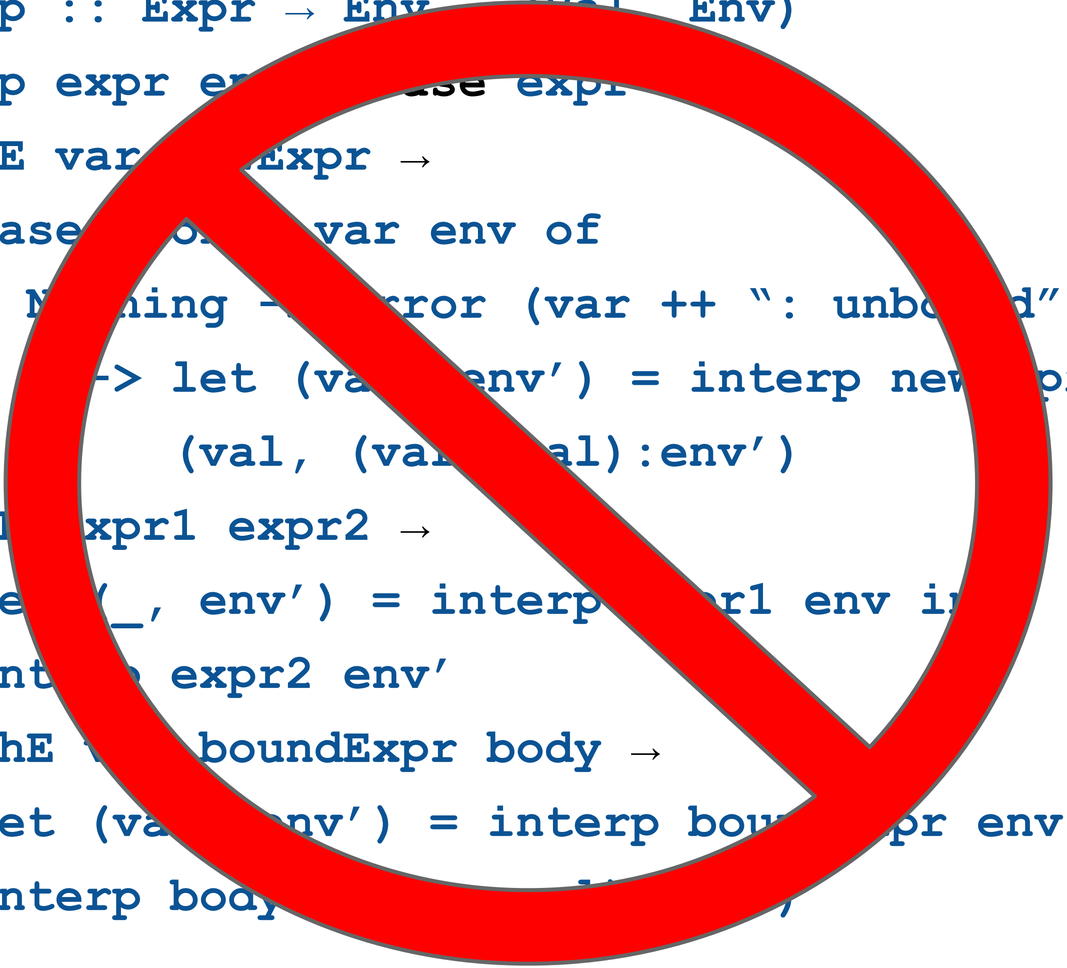
```
(with [make-counter
      (fun (cur)
        (fun (delta)
          (seq
            (set! cur (+ cur delta))
            cur)))]
  (with [init 0]
    (with [count! (make-counter init)]
      (+ (count! 3) init))))
```

```
interp :: Expr → Env → (Val, Env)
interp expr env = case expr of
  SetE var newExpr →
    case lookup var env of
      Nothing -> error (var ++ “: unbound”)
      _ -> let (val, env′) = interp newExpr env in
           (val, (var, val):env′)
  SeqE expr1 expr2 →
    let (_, env′) = interp expr1 env in
    interp expr2 env′
  WithE var boundExpr body →
    let (val, env′) = interp boundExpr env in
    interp body ((var, val):env′)
```

?

```
(with [x 3]
  (with [y 5]
    (seq
      (with [x (+ y 1)]
        (set! y (* x y)))
      (+ x y))))
```

```
interp :: Expr → Env → (Val, Env)
interp expr en    case expr
  SetE var    Expr →
    case    o    var env of
      N    ning →    rror (var ++ ": unbo   d")
         -> let (va    env') = interp new   pr env in
            (val, (va     al):env')
  SeqE  xpr1 expr2 →
    le  (_, env') = interp    pr1 env in
    int    expr2 env'
  WithE v    boundExpr body →
    let (va    nv') = interp bou     pr env in
    interp body            l    )
```



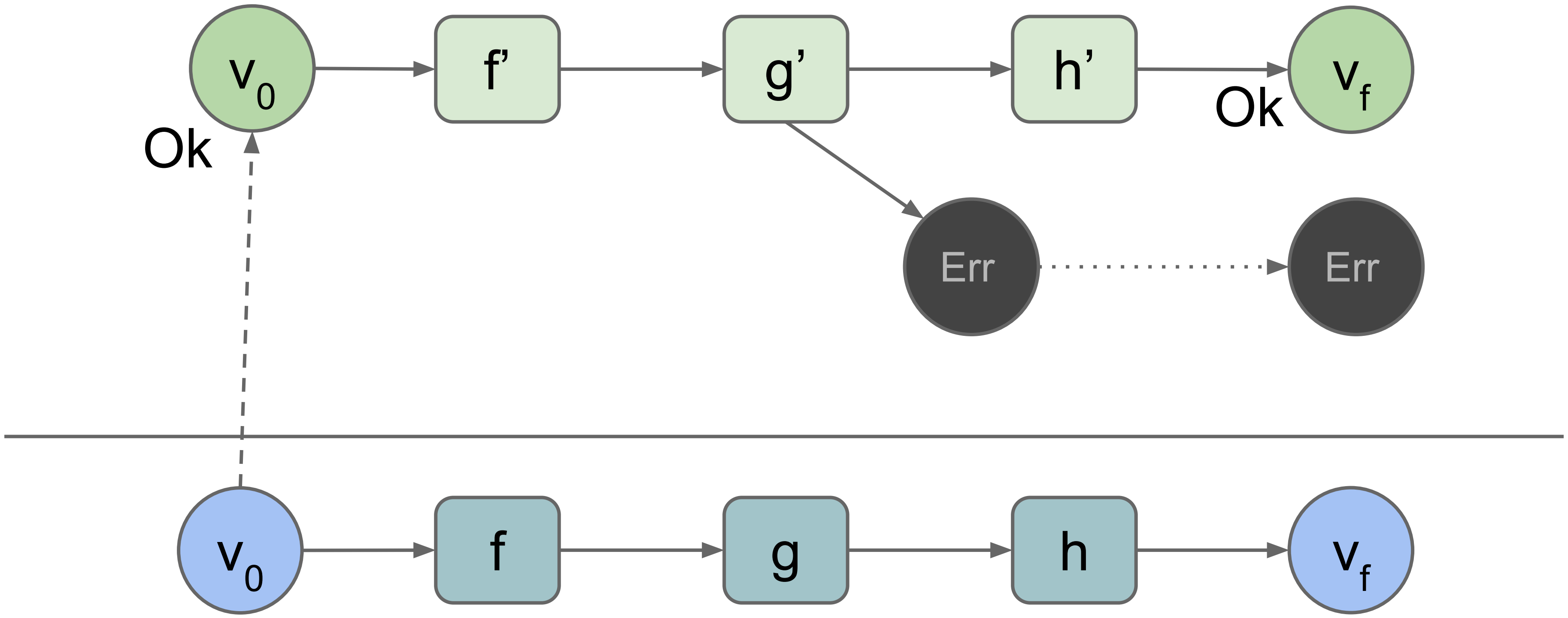
```
type Loc = Int
type Store = (Loc, [(Loc, Val)])
type Env = [(Var, Loc)]
```

```
interp :: Expr → Env → Store -> (Val, Store)
interp expr env store = case expr of
  VarE v → case lookup v env of
             Nothing -> error …
             Just loc -> let (Just val) = lookup loc store in
                         (val, store)
  SetE v newExp →
    case lookup v env of
      Nothing -> error …
      Just loc -> let (val, store′) = interp newExp env store in
                  (val, (loc, val):store′)
  SeqE expr1 expr2 →
    let (_, store′) = interp expr1 store in
    interp expr2 store′
```

```
interp :: Expr → Env → Store → (Val, Store)
interp expr env store = case expr of
  NumE n → (NumV n, store)
  FunE var body → (FunV var body env, store)
  AppE fun arg →
    let (fv, store′) = interp fun env store
        (av, store′′) = interp arg env store′
        (loc, store′′′) = alloc av store′′ in
    case fv of
      FunV var body closEnv →
        interp body ((var, loc):closEnv) store′′′
      _ → error …
```

```
data Result a = Ok a | Err String

parseExpr :: SExp → Result Expr
parseExpr (ListS [IdS “if”, test, cons, alt]) = …
```

```
data Result a = Ok a | Err String

parseExpr :: SExp → Result Expr
parseExpr (ListS [IdS "if", test, cons, alt]) =
  case parseExpr test of
    Err msg → Err msg
    Ok testExpr →
      case parseExpr cons of
        Err msg → Err msg
        Ok consExpr →
          case parseExpr alt of
            Err msg → Err msg
            Ok altExpr → Ok (IfE testExpr consExpr altExpr)
```



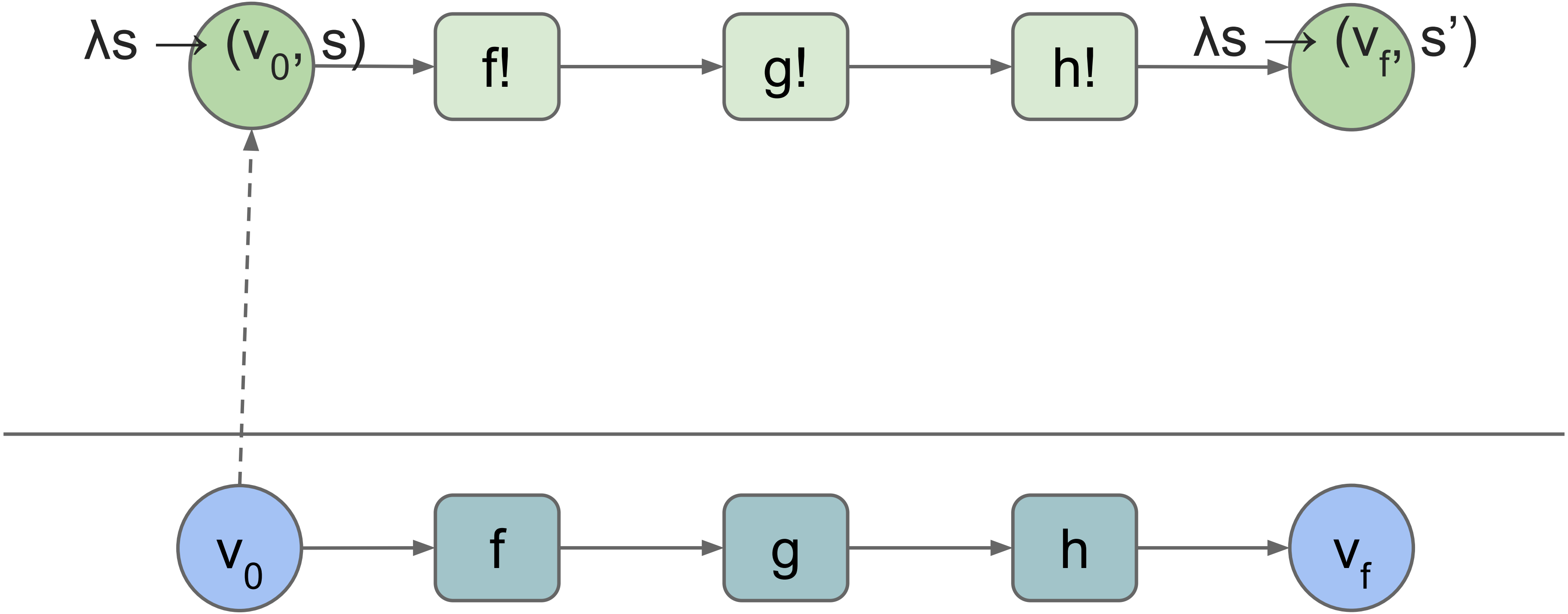
v_0
f'
g'
h'
v_f
Ok
Ok
Err
Err
v_0
f
g
h
v_f

```haskell
data Result a = Ok a | Err String

wrap :: a → Result a
wrap v =

andThen :: Result a → (a → Result b) → Result b
(Ok v)     `andThen` f =
(Err msg) `andThen` f =
```

```
data Result a = Ok a | Err String

wrap :: a → Result a
wrap v = Ok v

andThen :: Result a → (a → Result b) → Result b
(Ok v)     `andThen` f = f v
(Err msg) `andThen` f = Err Msg



(wrap v) `andThen` f = (Ok v) `andThen` f = f v

(Ok v) `andThen` wrap = wrap v = (Ok v)
(Err msg) `andThen` wrap = (Err msg)
```

```
data Result a = Ok a | Err String

parseExpr :: SExp → Result Expr
parseExpr (ListS [IdS “if”, test, cons, alt]) =
  parseExpr test `andThen` (λtestExpr →
    parseExpr cons `andThen` (λconsExpr →
      parseExpr alt `andThen` (λaltExpr →
        wrap (IfE testExpr consExpr altExpr))))
```

```
interp :: Expr → Env → Store → (Val, Store)
interp (AppE fun arg) env store =
  let (fv, store′) = interp fun env store
      (av, store′′) = interp arg env store′
      (loc, store′′′) = alloc av store′′ in
  case fv of
    FunV var body closEnv →
      interp body ((var, loc):closEnv) store′′′
```


λs → (v0, s)
f!
g!
h!
λs → (vf, s')
v0
f
g
h
vf

```
type Store = (Loc, [(Loc, Val)])
type StoreTrans a = Store → (a, Store)

wrap :: a → StoreTrans a
wrap v =

andThen :: StoreTrans a → (a → StoreTrans b) → StoreTrans b
st `andThen` f =



alloc :: Val → StoreTrans Loc
alloc v (nextLoc, cells) =
```

```
interp (AppE fun arg) env store =
  let (fv, store') = interp fun env store
      (av, store'') = interp arg env store'
      (loc, store''') = alloc av store'' in
  case fv of
    FunV var body closEnv →
      interp body ((var, loc):closEnv) store'''
```

```
interp (AppE fun arg) env =
  interp fun env `andThen` (λfv →
    interp arg env `andThen` (λav →
      alloc av `andThen` (λloc →
        case fv of
          FunV v body closEnv → interp body ((v, loc):closEnv)
```

```
interp (AppE fun arg) env =
  interp fun env >>= (λfv →
    interp arg env >>= (λav →
      alloc av >>= (λloc →
        case fv of
          FunV v body closEnv → interp body ((v, loc):closEnv)
```

```
interp (AppE fun arg) env =
  interp fun env >>= (\fv →
    interp arg env >>= (\av →
      alloc av >>= (\loc →
        case fv of
          FunV v body closEnv → interp body ((v, loc):closEnv)
```

```
interp (AppE fun arg) env =
  do fv ← interp fun env
     av ← interp arg env
     loc ← alloc av
     case fv of
       FunV var body closEnv → interp body ((var, loc):closEnv)
       …
```

Concepts

- Initial & top-level environments

- Mutable variables (and mutable values)
- Separation of scope (env) and state (store)
- Store-passing-style
- Store transformers
- Monads as technique for factoring out non-local concerns