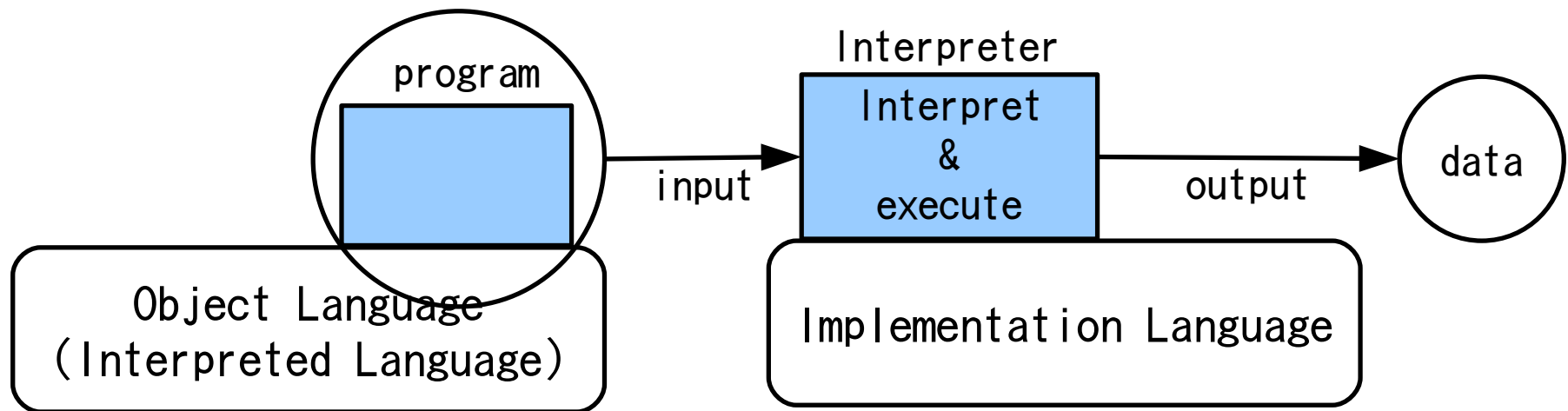


Monad Transformers and Modular Interpreters

Take 2

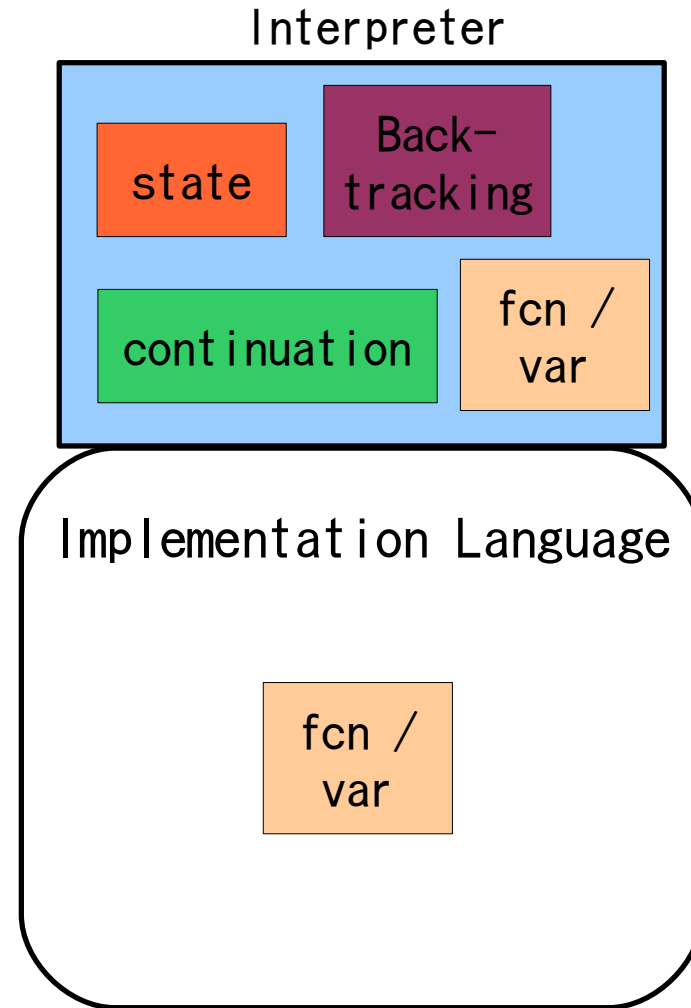
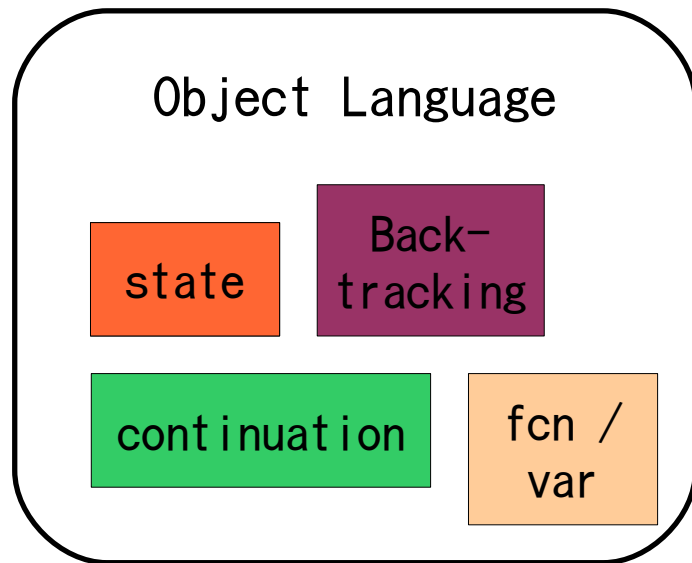
Jun Inoue

A High-level View: Interpreter Architecture



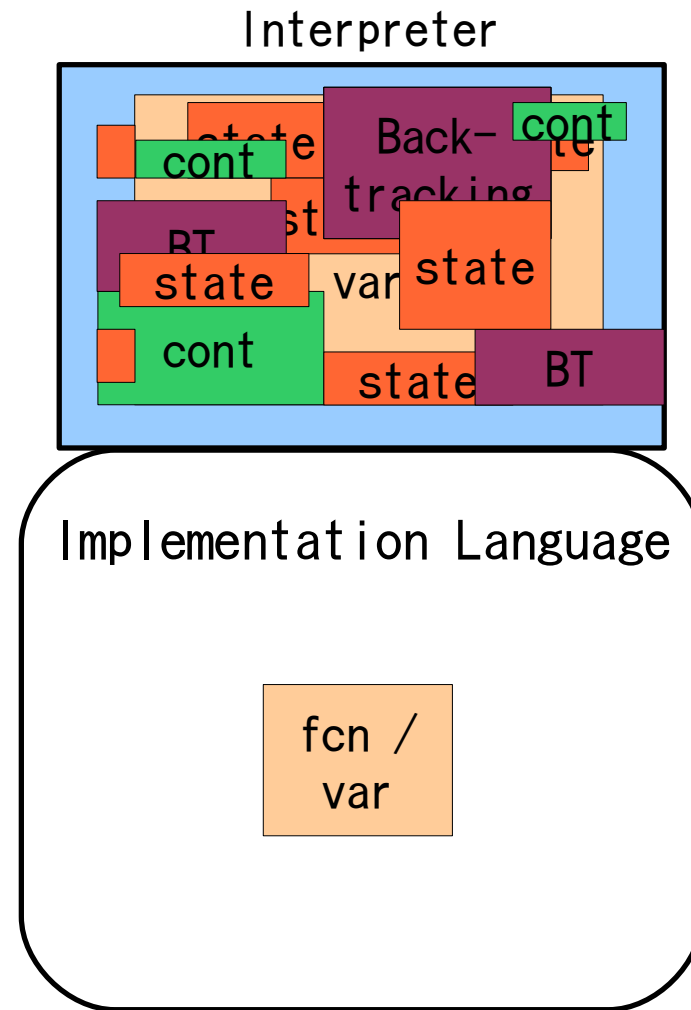
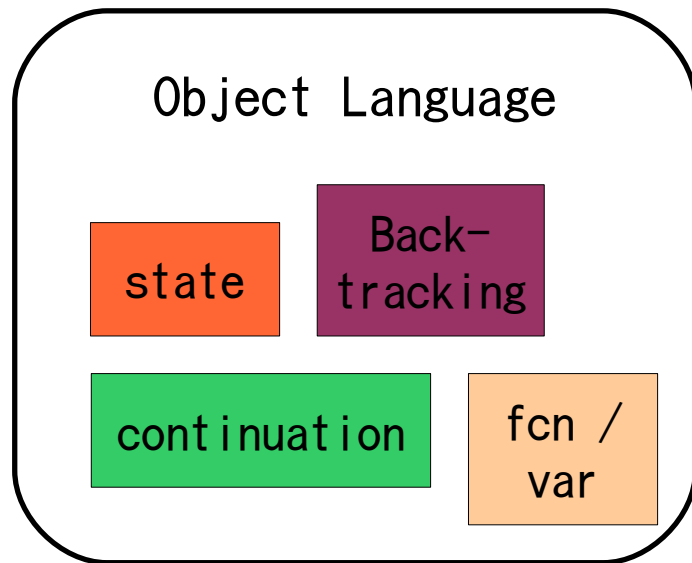
A High-level View: Interpreter Architecture

Obj. Lang. features
that are implemented in
the Interpreter.



A High-level View: Interpreter Architecture

Obj. Lang. features that are **HAPHAZARDLY** implemented in the Interpreter.



Concrete Interpreter Code

```
data Exp = Add Exp Exp | Const Int
```

```
type Val = Int
```

```
{-
```

```
e.g. an object program
```

```
  1 + (2 + 3)
```

```
becomes
```

```
  Add (Const 1) (Add (Const 2) (Const 3))
```

```
-}
```

```
interp :: Exp -> Val
```

```
interp (Add x y) = (interp x) + (interp y)
```

```
interp (Const i) = i
```

Concrete Interpreter Code

```
data Exp = Add Exp Exp | Const Int
```

```
type Val = Int
```

```
interp :: Exp -> Val
```

```
interp (Add x y) = (interp x) + (interp y)
```

```
interp (Const i) = i
```

Concrete Interpreter Code

```
data Exp = Add Exp Exp | Const Int | Fun String Exp  
        | App Exp Exp
```

```
data Val = VInt Int | VFun (Val -> Val)
```

```
interp :: Exp -> Env -> Val
```

```
interp (Add x y) env = (interp x env) + (interp y env)
```

```
interp (Const i) env = VInt i
```

```
interp (Fun fml body) = VFun (\actual -> interp body (ext (fml, actual) env))
```

```
interp (App fun arg) env = fun (interp arg env)
```

•add function

Concrete Interpreter Code

```
data Exp = Add Exp Exp | Const Int | Fun String Exp
         | App Exp Exp | Read | Write Exp
data Val = VInt Int | VFun (Val -> State -> (Val, State))
```

```
interp :: Exp -> Env -> State -> (Val, State)
```

```
interp (Add x y) env s = let (x', s') = interp x env s
                          (y', s'') = interp y env s'
                          in (x' + y', s'')
```

```
interp (Const i) env s = (VInt i, s)
```

```
interp (Fun fml body) env s =
```

```
  (VFun (\actual -> interp body (ext (fml, actual) env)), s)
```

```
interp (App fun arg) env s = let (a, s') = interp arg env s in fun a env s'
```

```
interp Read env s = (s, s)
```

```
interp (Write exp) env s = let (s', _) = interp exp env s in ((), s')
```

- add function
- add state

Concrete Interpreter Code

```
data Exp = Add Exp Exp | Const Int | Fun String Exp
         | App Exp Exp | Read | Write Exp | Or Exp Exp | Fail
data Val = VInt Int | VFun (Val -> State -> [(Val, State)])
```

- add function
- add state
- add backtracking

```
interp :: Exp -> Env -> State -> [(Val, State)]
interp (Add x y) env s = concatMap (\(x', s') ->
                                   map (\(y', s'') -> (x' + y', s''))
                                   (interp y env s'))
  (interp x env s)
interp (Const i) env s = [(VInt i, s)]
interp (Fun fml body) s =
  [(VFun (\actual -> interp body (ext (fml, actual) env)), s)]
interp (App fun arg) env s = concatMap (\(a, s') -> fun a env s')
  (interp arg env s)
interp Read env s = [(s, s)]
interp (Write exp) env s = map (\(s', _) -> ((), s')) (interp exp env s)
interp Fail env s = []
interp (Or x y) env s = (interp x env s) ++ (interp y env s)
```

Goal

- Can we isolate the changes better?
- Can we just say
 `addbacktracking (addstate (addfcn
 basicinterp))`

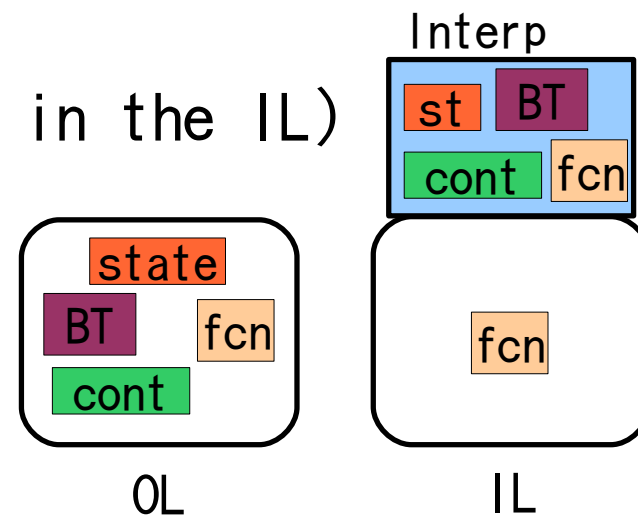
Functions / Variable Bindings

Ability to give names to intermediate results:

```
(fun x -> x + 5) (2 * 3 + 1)
```

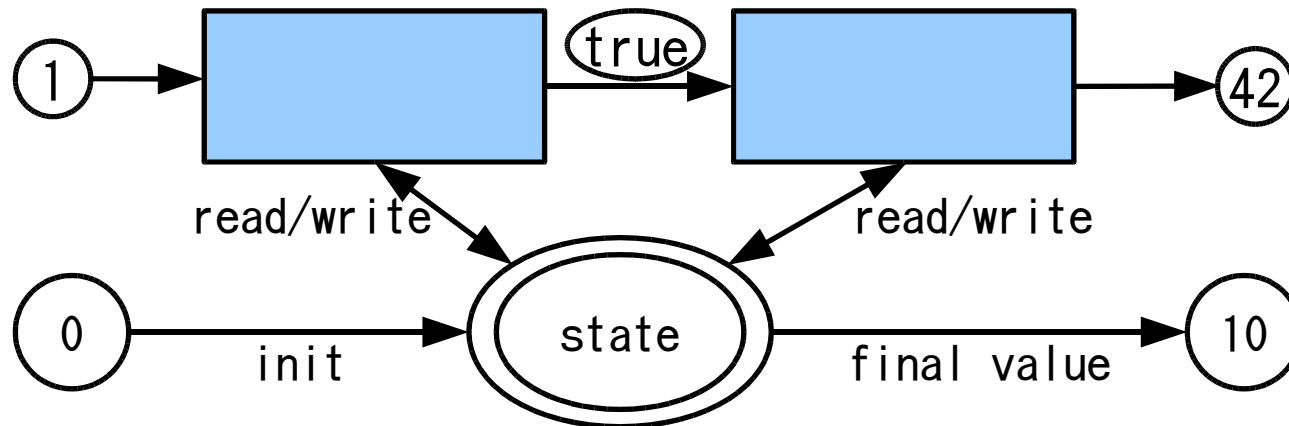
```
let x = 2 * 3 + 1  
in x + 5
```

(We'll take this one for granted in the IL)



Mutable State

Imperative computation: consult & modify state.



-- Label each leaf of tree with a unique ID

```
label (Branch l r) =
```

```
  LabeledBranch (label l) (label r)
```

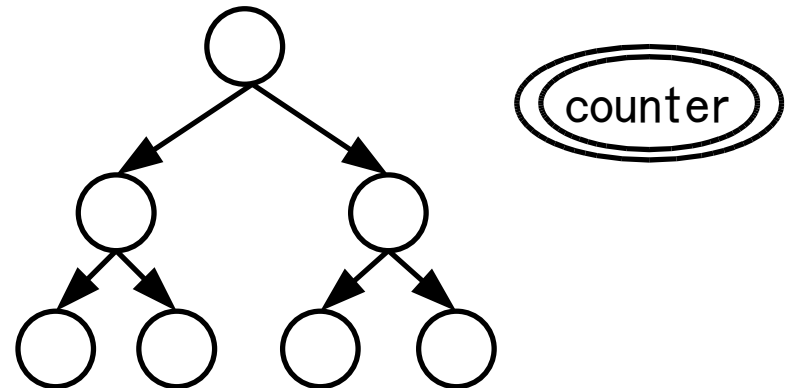
```
label (Leaf dat) =
```

```
  let id = read ()
```

```
  in
```

```
    write (id + 1);
```

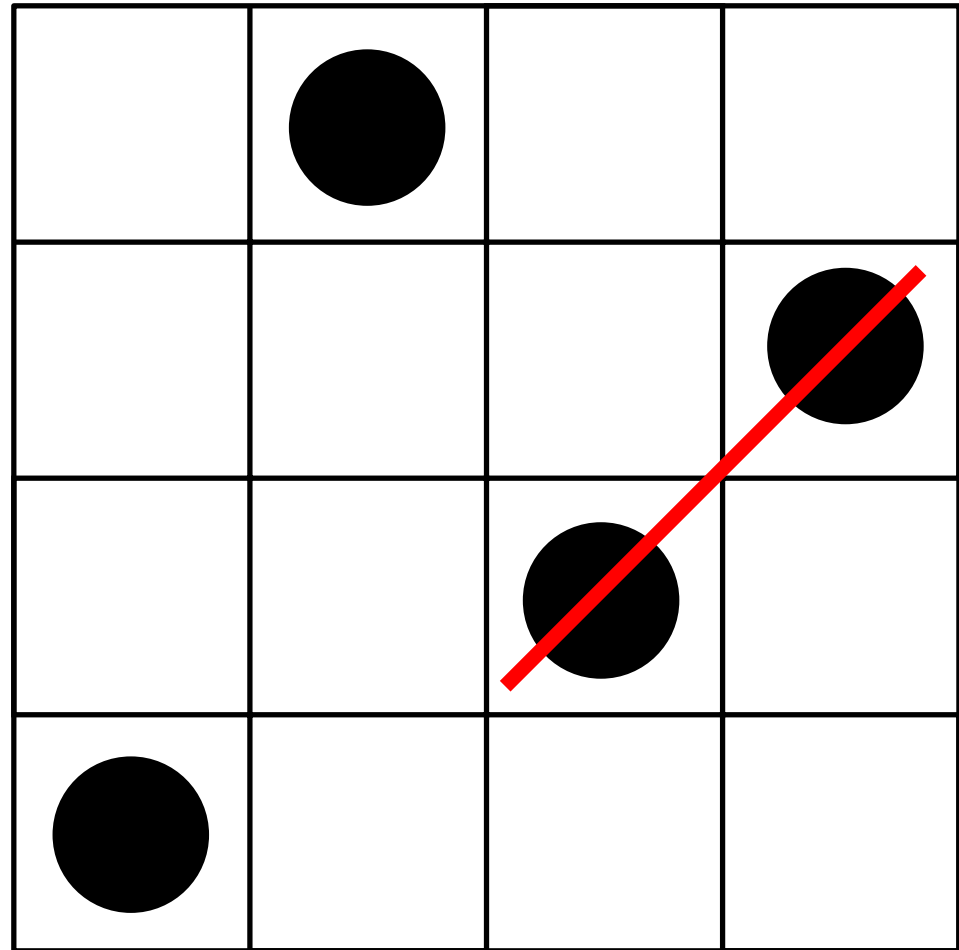
```
    LabeledLeaf dat id
```



Backtracking

4-queens problem:

Put 4 dots on a 4x4 grid, so that no two are on the same row, column, or diagonal.



Bad; two on
same diagonal

Backtracking

Natural solution is
trial & error
(backtracking)

	●		
	●		
●		●	
●			

Nice to have language
support.

Backtracking

Solution:

```
four_queens = four_q 4
```

```
four_q 0 = []
```

```
four_q n = let dots = four_q (n - 1)
           in add (next_dot n dots) dots
```

where

```
next_dot dots =
```

```
  check (1, n) dots
```

```
  <+> check (2, n) dots
```

```
  <+> check (3, n) dots
```

```
  <+> check (4, n) dots
```

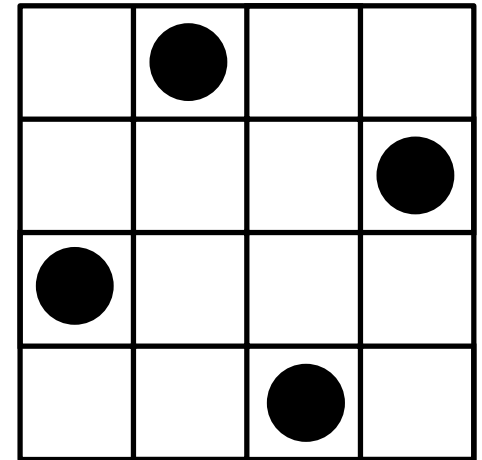
```
check coord dots =
```

```
  if conflicts coord dots then
```

```
    fail -- no alternatives
```

```
  else
```

```
    coord -- 1 alternative
```



The Observation

Every feature adds “innocence” assertions to programs’ meanings.

The program “42” means...

- Without states: “return 42”
- With states: “return 42, and don’t modify or depend on state”

```
interp (Add x y) env s = let (x', s') = interp x env s
                          (y', s'') = interp y env s
                          in (x' + y', s'')
interp (Const i) env s = (VInt i, s)
```

The Observation

Every feature adds extra information flow throughout a program.

The program “let $x = e_1$ in e_2 ” means...

- Without BT: “evaluate e_2 with (**the** result of) e_1 bound to x ”
- With BT: “evaluate e_2 with (**a** result of) e_1 bound to x , and **go back to e_1 for more alternatives if e_2 fails**”

```
interp (Add x y) env s = let (x', s') = interp x env s
                          (y', s'') = interp y env s'
                          in (x' + y', s'')
```

```
interp (Const i) env s = (VInt i, s)
```

Monads

Easy to support a feature if IL has it.

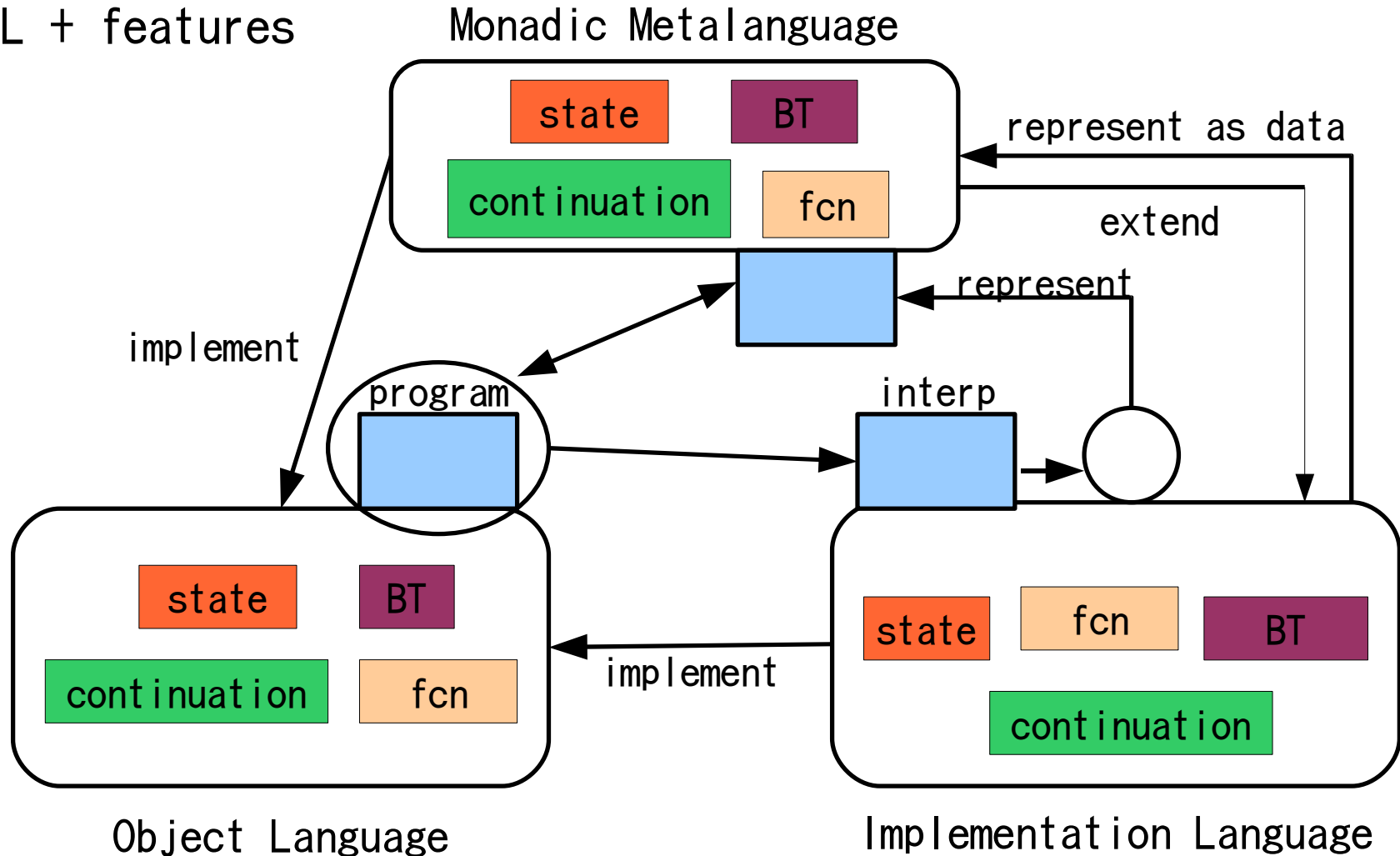
```
data Exp = Add Exp Exp | Const Int | Fun String Exp
         | App Exp Exp | Read | Write Exp
data Val = VInt Int | VFun (Val -> Val)
```

```
state = ref 0
```

```
interp :: Exp -> Env -> Val
interp (Add x y) env = (interp x env) + (interp y env)
interp (Const i) env = VInt i
interp (Fun fml body) env = VFun (\actual -> interp body (ext (fml, actual) env))
interp (App fun arg) env = fun arg
interp Read env = !state
interp (Write exp) env = state := (interp exp env)
```

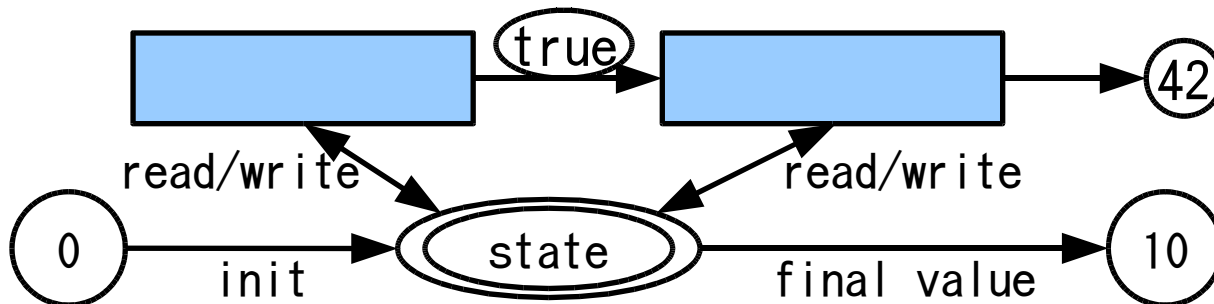
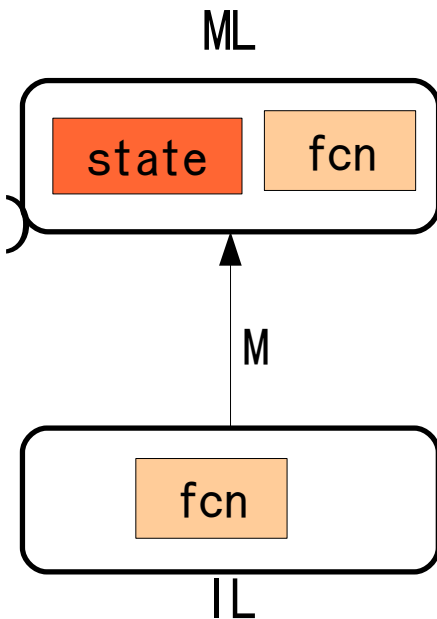
Monads

ML = IL + features



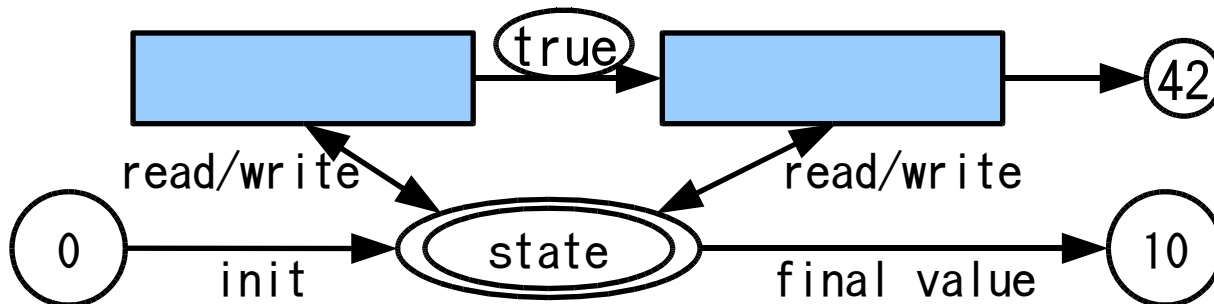
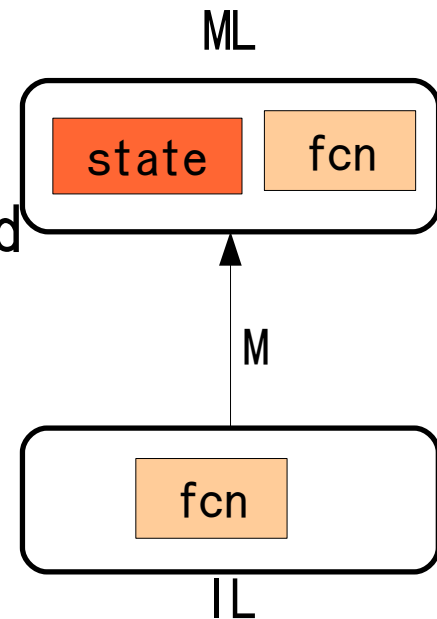
State Monad

- Data type for “a program in ML” : $M\ a$
 - no arguments, return type a
 - $M\ a = State\ M\ a = State \rightarrow (a, State)$ for state monad
- Inclusion of values
 - “42” is in IL, so it should be in ML
 - representation: `ret 42`
 - `ret x = \s -> (x, s)` for state monad



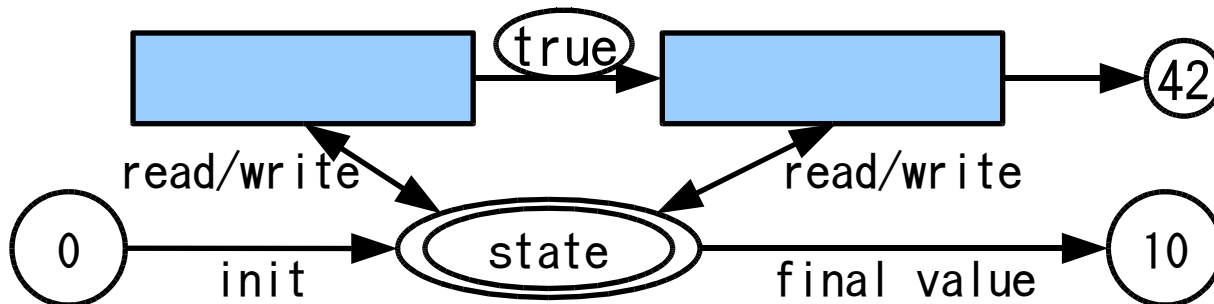
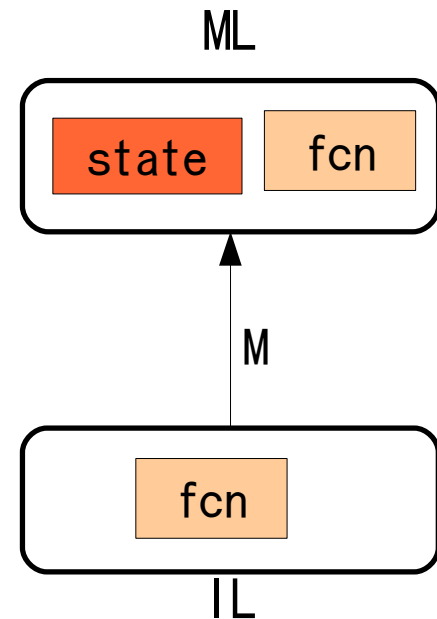
State Monad

- $M\ a = State \rightarrow (a, State)$
- Allowing bindings
 - build (data-representation of) parametrized computation (i.e. functions)
 - “let $x = 5$ in $x + x$ ” in ML
 - representation:
 $bind\ (ret\ 5)\ (\backslash x \rightarrow x + x)$
 - $bind\ m\ f = \backslash s \rightarrow let\ (v, s') = m\ s$
 $in\ f\ v\ s'$



State Monad

- Taking advantage of the feature
 - “read” in ML
 - represent with $\text{read} = \backslash s \rightarrow (s, s)$
 - $\text{read} :: \text{StateM State}$
 - “write x” in ML
 - represent with $\text{write new_s} = \backslash s \rightarrow ((), \text{new_s})$
 - $\text{write} :: \text{State} \rightarrow \text{StateM ()}$



State Monad

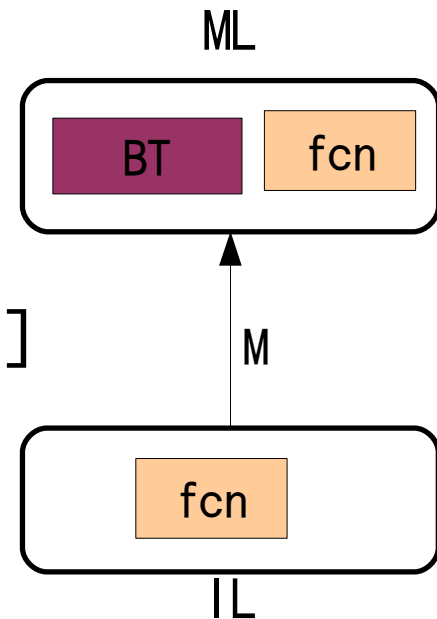
```
data Exp = Add Exp Exp | Const Int | Fun String Exp
         | App Exp Exp | Read | Write Exp
data Val = VInt Int | VFun (Val -> M Val)
type M a = State -> (a, State)

interp :: Exp -> Env -> M Val
interp (Add x y) env = bind (interp x env) (\x' ->
                             bind (interp y env) (\y' ->
                                     ret (x' + y')))
interp (Const i) env = ret (VInt i)
interp (Fun fml body) env =
    ret (VFun (\actual -> interp body (ext (fml, actual) env)))
interp (App fun arg) env = bind (interp arg env) fun
interp Read env = read
interp (Write exp) env = bind (interp exp env) (\x -> write x)

-- using the interpreter
let m = interp parse_tree
in m initial_state
```

Backtracking Monad

- $M\ a = [a]$
- $ret\ x = [x]$
- $bind\ m\ f = concatMap\ f\ m$
 - $concatMap\ (\backslash x \rightarrow [x, x+1])\ [0, 2, 1]$
= $concat\ [[0, 1], [2, 3], [1, 2]]$
= $[0, 1, 2, 3, 1, 2]$
- $fail = []$
 - $fail :: ListM\ a$
- $x\ <+>\ y = x\ ++\ y$
 - $(<+>) :: ListM\ a \rightarrow ListM\ a \rightarrow ListM\ a$



Backtracking Monad

```
data Exp = Add Exp Exp | Const Int | Fun String Exp
         | App Exp Exp | Or Exp Exp | Fail
data Val = VInt Int | VFun (Val -> M Val)
type M a = [a]
```

```
interp :: Exp -> Env -> M Val
```

```
interp (Add x y) env = bind (interp x env) (\x' ->
                             bind (interp y env) (\y' ->
                                     ret (x' + y'))))
```

```
interp (Const i) env = ret (VInt i)
```

```
interp (Fun fml body) env =
```

```
    ret (VFun (\actual -> interp body (ext (fml, actual) env)))
```

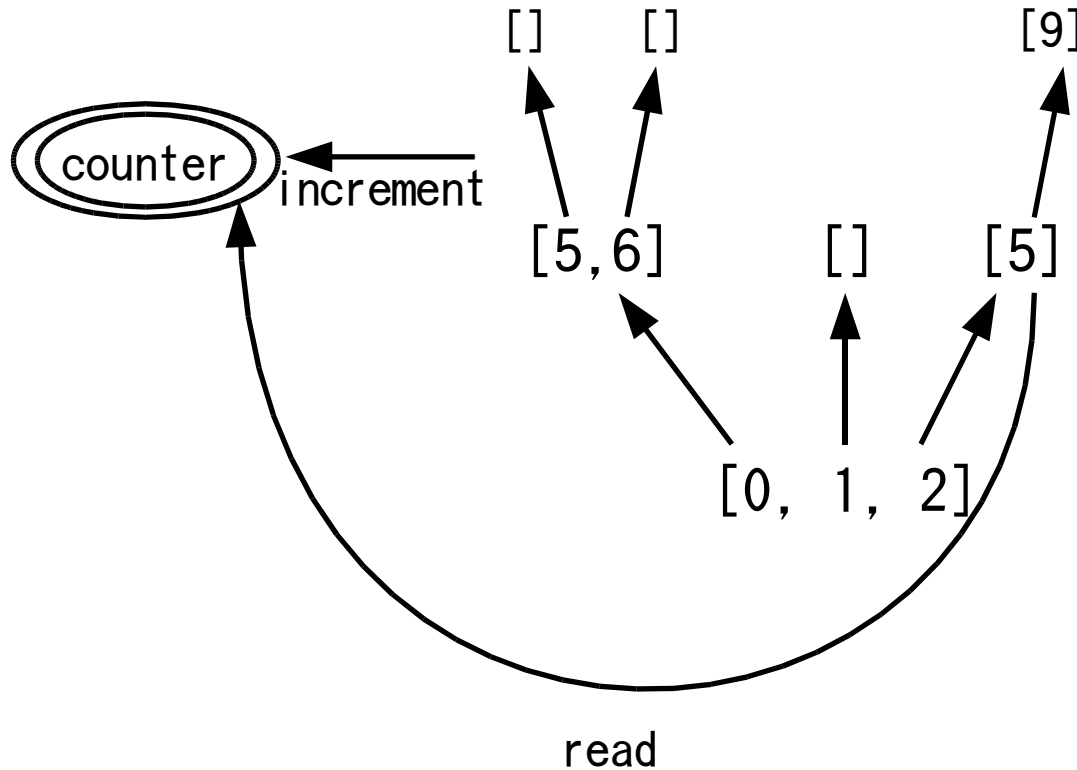
```
interp (App fun arg) env = bind (interp arg env) fun
```

```
interp Fail env = fail
```

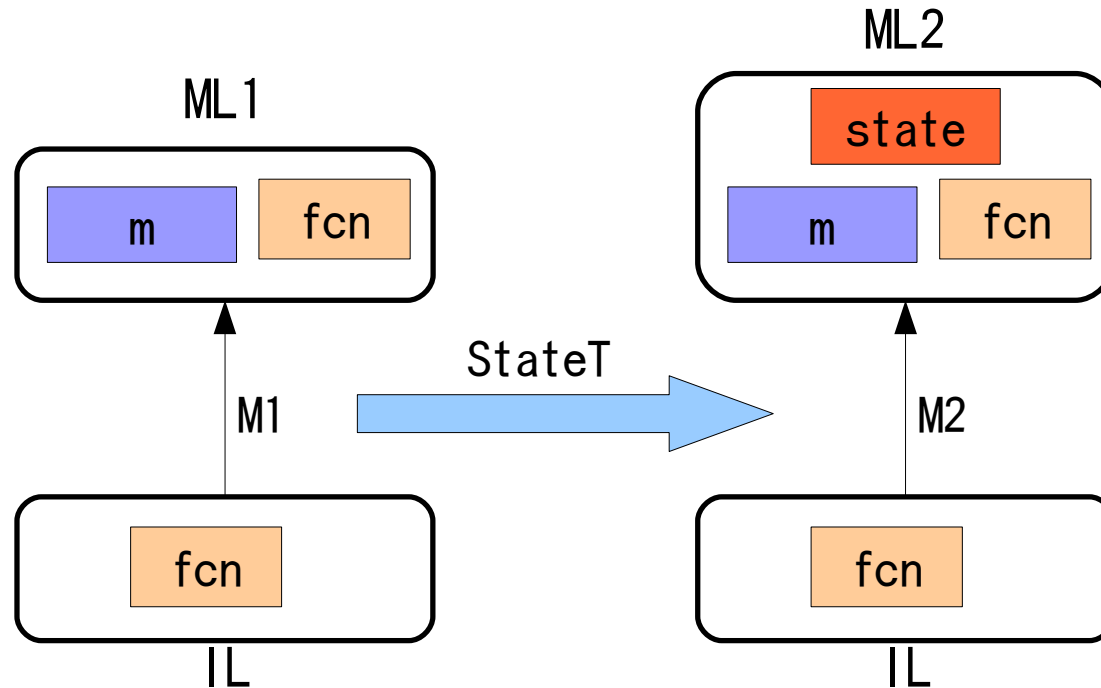
```
interp (Or x y) env = (interp x env) <+> (interp y env)
```

Language Feature Interaction

A stateful, backtracking computation:
should the increment be visible when
executing at [5]?



Monad Transformer



- $M2 \text{ a} = \text{StateT } M1 \text{ a}$ should be a monad
- $M2$ should be an extension of $M1$

StateT

- StateT takes any monad type and gives the type of the same ML except with state added
 - type StateT m a = State -> m (State, a)
 - e.g. StateT List a = State -> [(a, State)]
- Any program in M1 should be convertible to M2.
 - lift :: M1 a -> StateT M1 a (= M2 a)
 - lift m = \s -> bind m (\x -> ret (x, s))
- Exploiting language feature:
 - readT :: StateT m State
readT = \s -> ret (s, s)
 - writeT :: State -> StateT m ()
writeT new_s = \s -> ret ((), new_s)

StateT

- lift adds bookkeeping and negative assertions like bind and ret do
 - what about special functions like fail/(<+>)?
- can't use fail/(<+>) directly to build computations of type StateT List a
 - readT <+> fail -- type error!
- specialized (ad-hoc) lifting
 - type StateT List a = State -> [(a, State)]
 - failT = \s -> []
 - m1 <+T> m2 = \s -> m1 s ++ m2 s'

StateT

```
data Exp = Add Exp Exp | Const Int | Fun String Exp
         | App Exp Exp | Or Exp Exp | Fail | Read | Write Exp
data Val = VInt Int | VFun (Val -> M Val)
type M a = StateT List a
```

```
interp :: Exp -> Env -> M Val
```

```
interp (Add x y) env = bind (interp x env) (\x' ->
                             bind (interp y env) (\y' ->
                                     ret (x' + y'))))
```

```
interp (Const i) env = ret (VInt i)
```

```
interp (Fun fml body) env =
```

```
    ret (VFun (\actual -> interp body (ext (fml, actual) env)))
```

```
interp (App fun arg) env = bind (interp arg env) fun
```

```
interp Fail env = failT
```

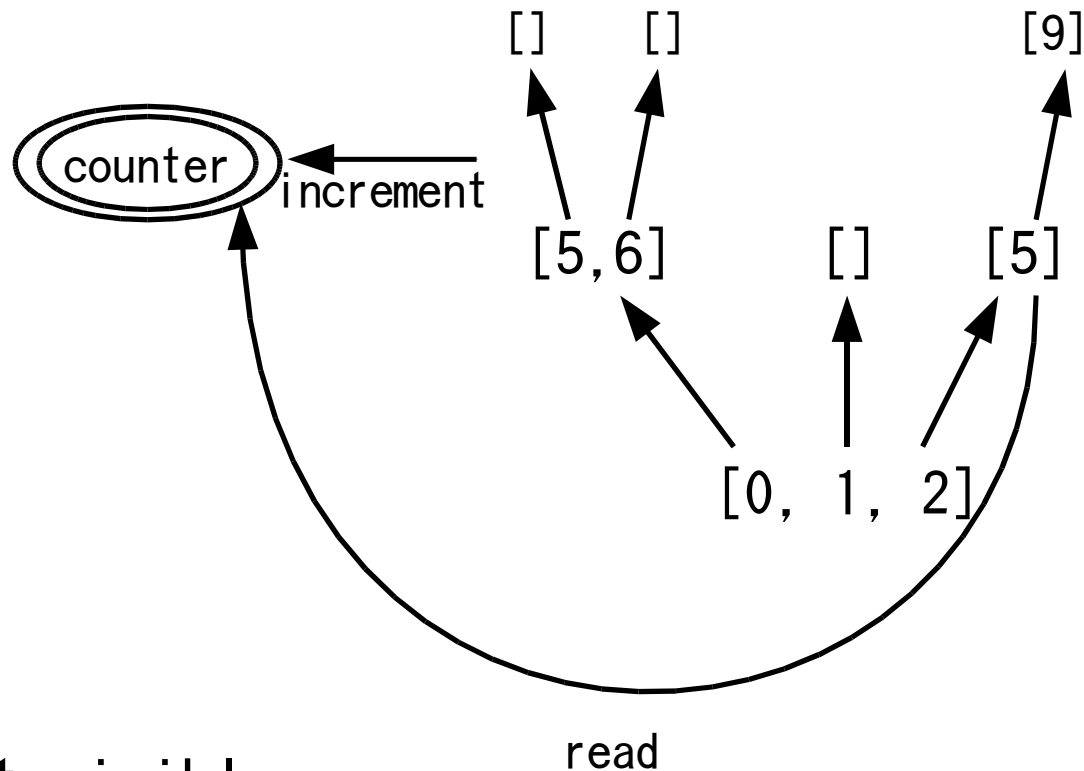
```
interp (Or x y) env = (interp x env) <+T> (interp y env)
```

```
interp Read env = readT
```

```
interp (Write exp) env = bind (interp exp env) writeT
```

ListT?

Which semantics is that?



- Not visible.
 - StateT List a = State -> [(a, State)]
 - m1 <+T> m2 = \s -> m1 s ++ m2 s'

ListT?

- A generic ListT doesn't seem to exist (can only lift a certain kind of monads)
- An implementation of backtracking based on continuations can give the other type of interaction

Conclusions

- A monad gives a systematic way to represent programs in an enriched version of the PL you're using (in this case the IL)
 - the illusion of programming in the enriched language
- With the enriched language, different parts of an interpreter interfere less with each other
- Examining ad-hoc lifting explicates interaction of language features.
 - how does staging interact with states?
 - build a StageM and/or StageT, try to do the lifting for StateT StageM or StageT StateM

Questions?
