The role of environments in scoping

Nested scopes

- Whenever scopes can be nested we have the "feature" that a variable may occur more than once in the same scope.
- Resolving that ambiguity is important

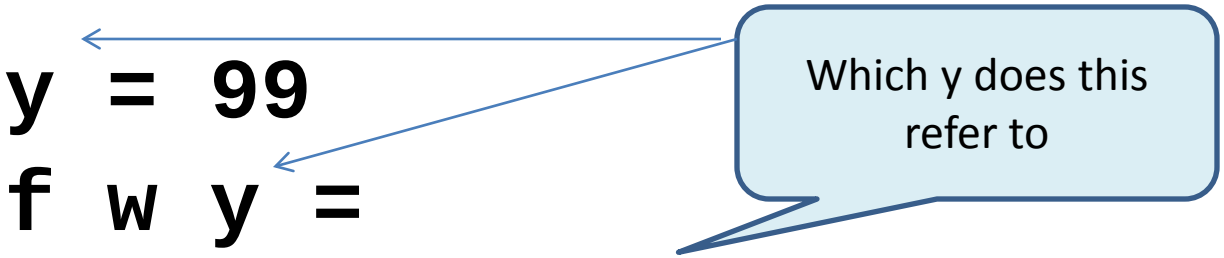
```
f x y =
   let x = 9
   in y + x
```

Which x does this refer to

What value does `(f 3 7)` return?

Local functions

- This can also happen with local functions




```
y = 99
f w y =
   let g x = y + x
   in g w

f 33 7
```

The role of the environment

- In our definitional interpreters, the environment maps names to locations.

- To determine which one of a number of possible binding sites a variable uses, we must study how the environment is changed.

The fun-arg problem

- This problem is called the fun-arg problem

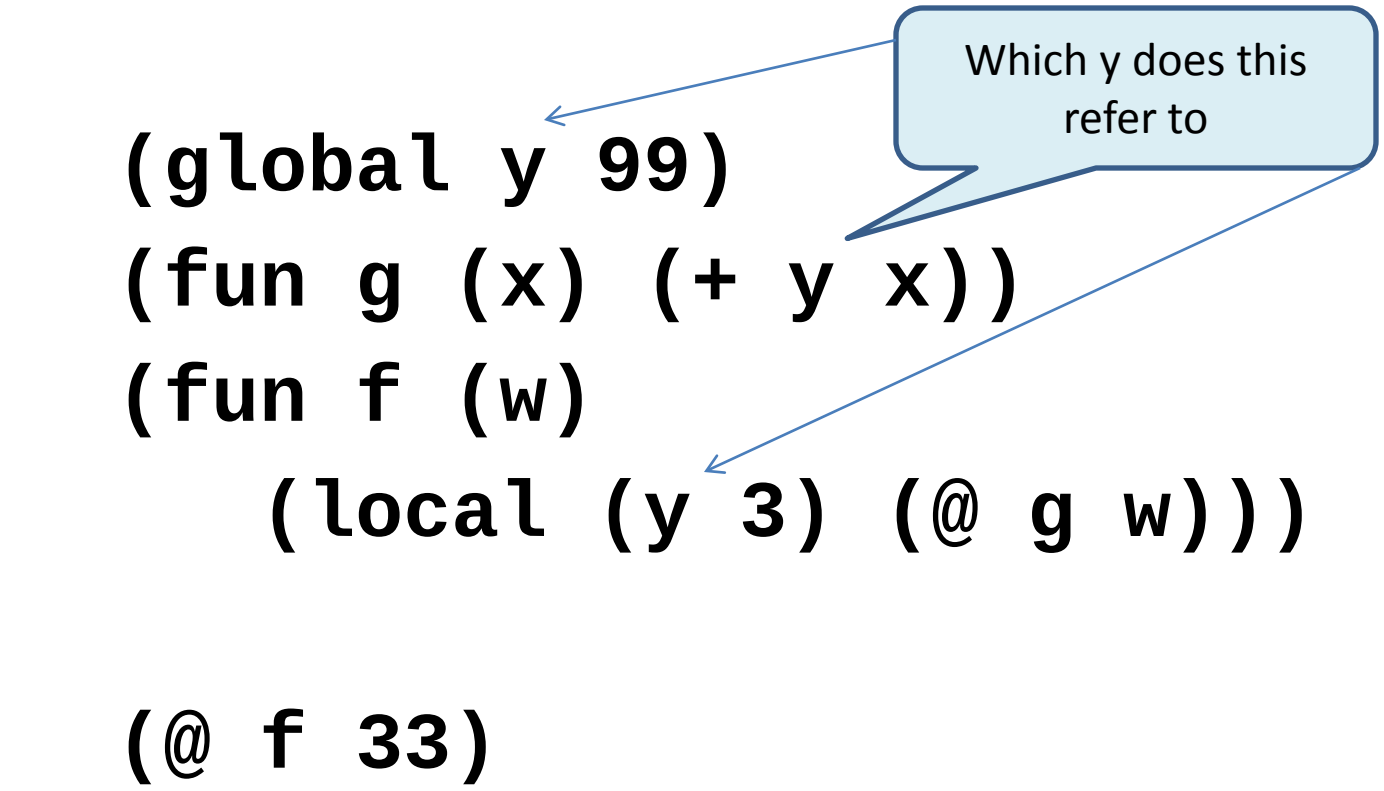
```
y = 99
f w y =
  let g x = y + x
  in g w

f 33 7
```

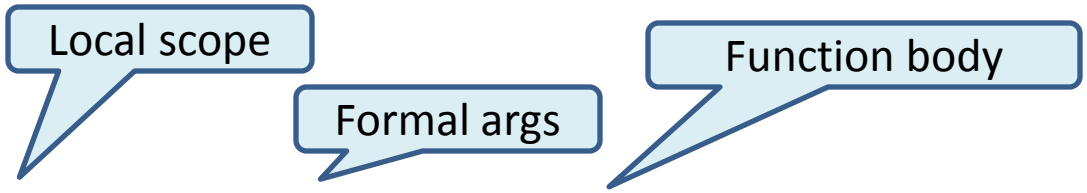
It's resolution depends upon how the body of functions (like g) are evaluated

Similar problem

We don’t need local functions to have this problem.



At definition site




```
elab :: Def
     -> (Env (Env Addr, [Vname], Exp) , Env Addr, State)
     -> IO (Env (Env Addr, [Vname], Exp), Env Addr, State)

elab (FunDef f vs e) (funs,vars,state) =
  return ( extend f (vars,vs,e) funs, vars, state )
elab (GlobalDef v e) (funs,vars,state) =
  do { (value,state2) <- interpE funs vars state e
     ; let (addr,state3) = alloc value state2
     ; return(funs, extend v addr vars,state3)}
```

At call site

```
run state (term@(At f args)) =
    case lookUp funs f of
       NotFound -> error …
       Found (vars2,formals,body) ->
         do { when (length args /= length formals)
                   (error …)
            ; (vs,state2) <- interpList funs
                                 vars state args
            ; let (pairs,state3)
                      = bind formals vs state2
            ; (v,state4) <- interpE funs
                                (push pairs vars2)
                                state3 body
            ; return(v,state4) }
```

A closure

- We call a function object that binds its free variables in the scope of definition (rather than use) a closure.

- Closures are key components in static scoping.

- It is interesting that in functional languages, where functions may return functions, variables may now outlive their scope.
- Who can give an example?
- What does this imply about implementations?