Administrivia

- Student hours (or by appointment):
  - Listed on Canvas in Syllabus
  - MTWTh 10:30 – 11:30 a.m. via Zoom
- Assignment four posted
  - Due Tuesday 4 July by **10pm**

CptS 355- Programming Language Design

Scope and Scoping

2

Instructor: Jeremy E. Thompson

WASHINGTON STATE UNIVERSITY

World Class. Face to Face.

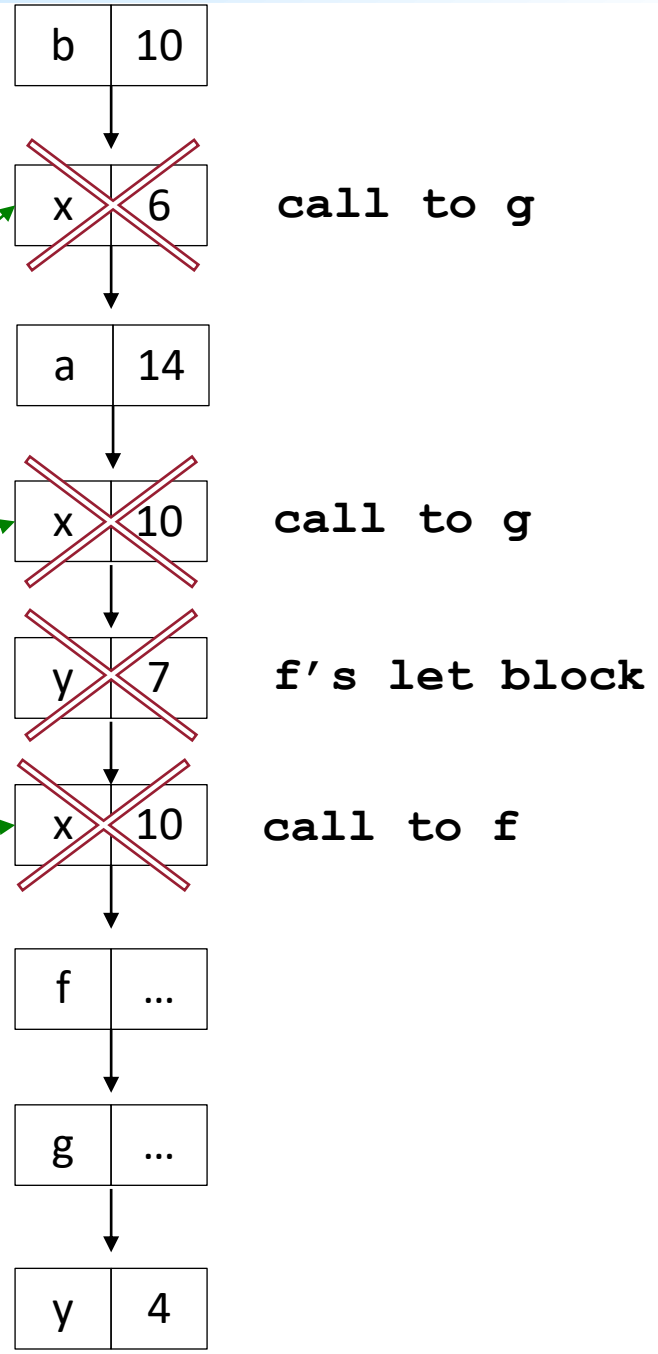
Recall: Storage Management

- How is storage managed in different programming languages and for different kinds of data?
- Storage is typically divided into 3 areas:
  - Static area
  - Stack area
  - Heap
- Stack and heap are used for dynamic allocation

Recall: Activation Records

- **Haskell Example-2:**

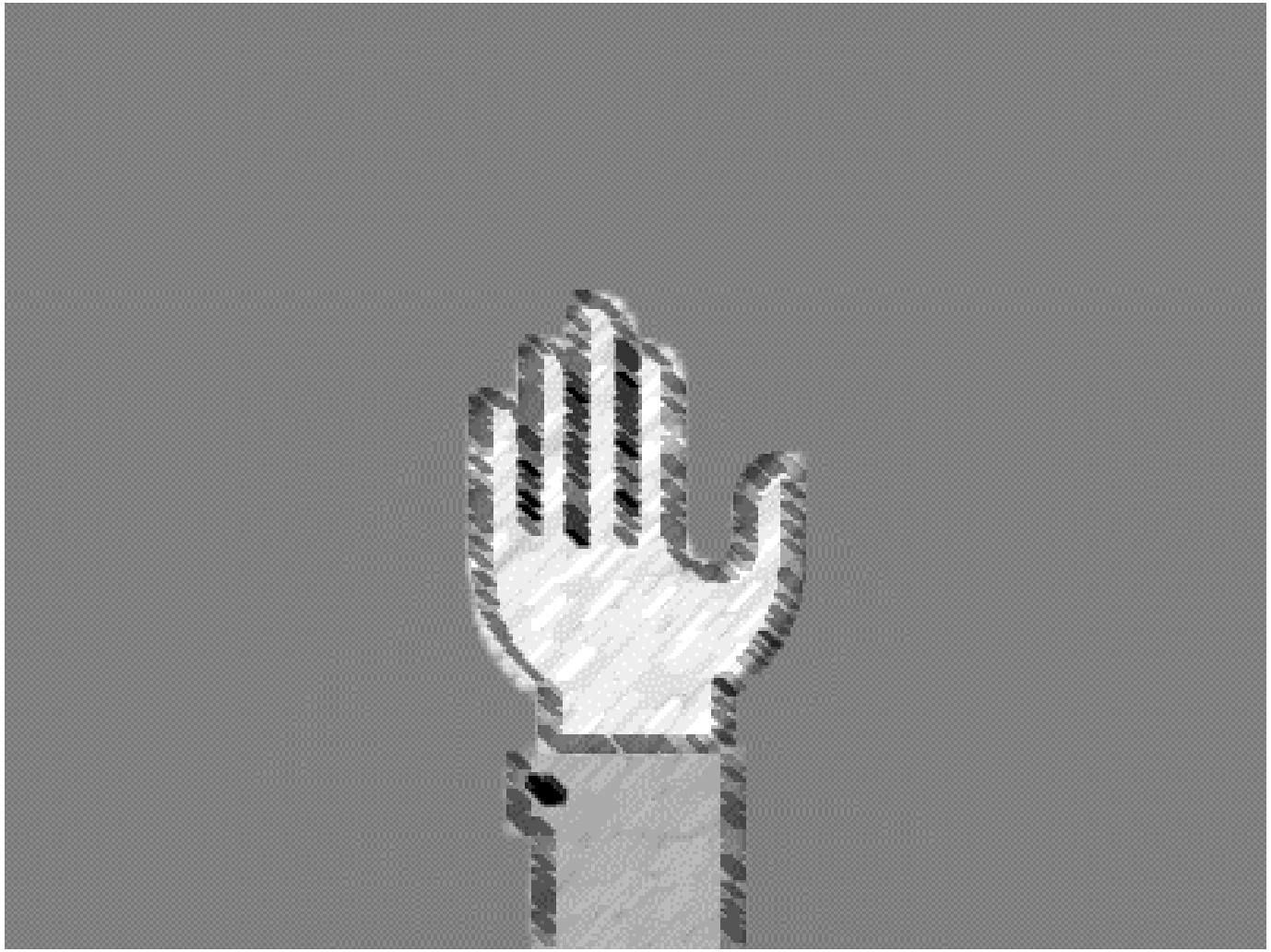
```
y=4
g x = x+y
f x  =
   let
     y = 7
   in
     g x
   end
a = f 10
b = g 6
```



Outline

- Storage Management in Programming Languages
- Scope
- Referencing environments
- Lifetime
- Static scope rules
- Dynamic scope rules

Questions?



Static Scoping and Dynamic Scoping

Static Scoping:

Definition:

The identifier refers to the declaration in the *closest* and enclosing block

Dynamic Scoping:

Definition:

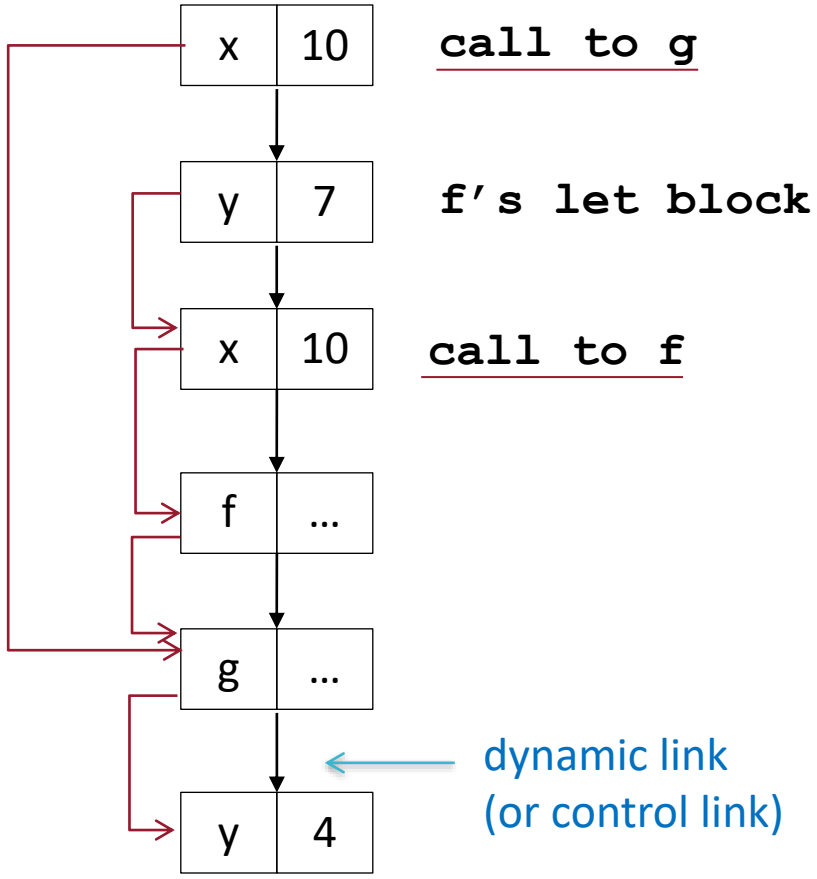
The identifier refers to the *most recent* (in time) still-live declaration

Static Scoping

- **Haskell Example:**

```
y=4
g x = x+y
f x = let
        y = 7
      in
        g x
a = f 10
b = g 6
```

static link (access link) to referencing environment

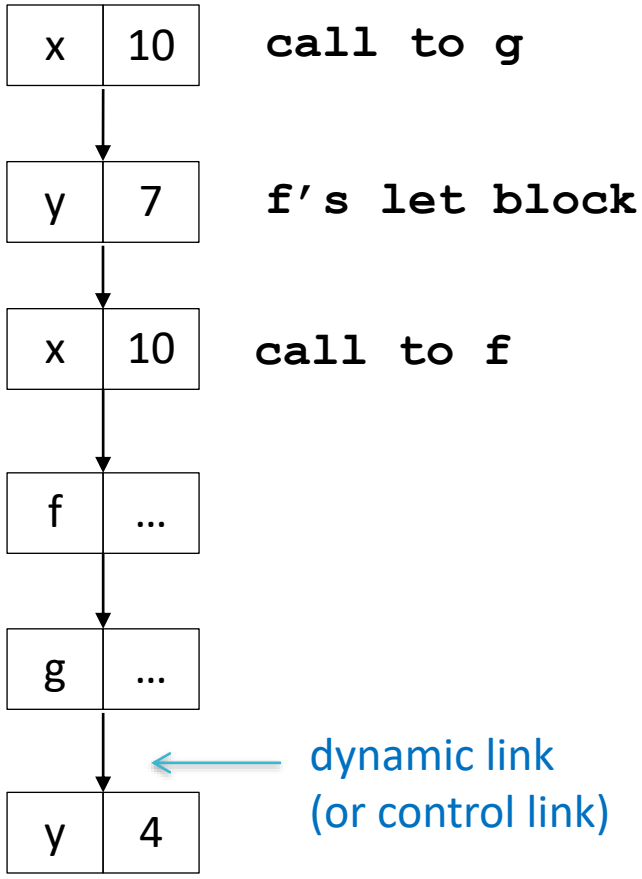


Dynamic Scoping

Simply do not create static links

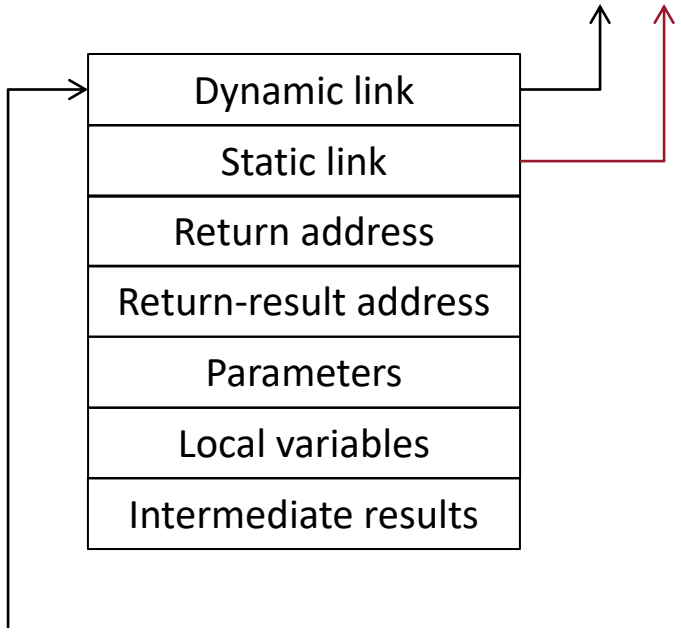
- **(non-)Haskell Example:**

```
y=4
g x = x+y
f x = let
        y = 7
      in
        g x
a = f 10
b = g 6
```

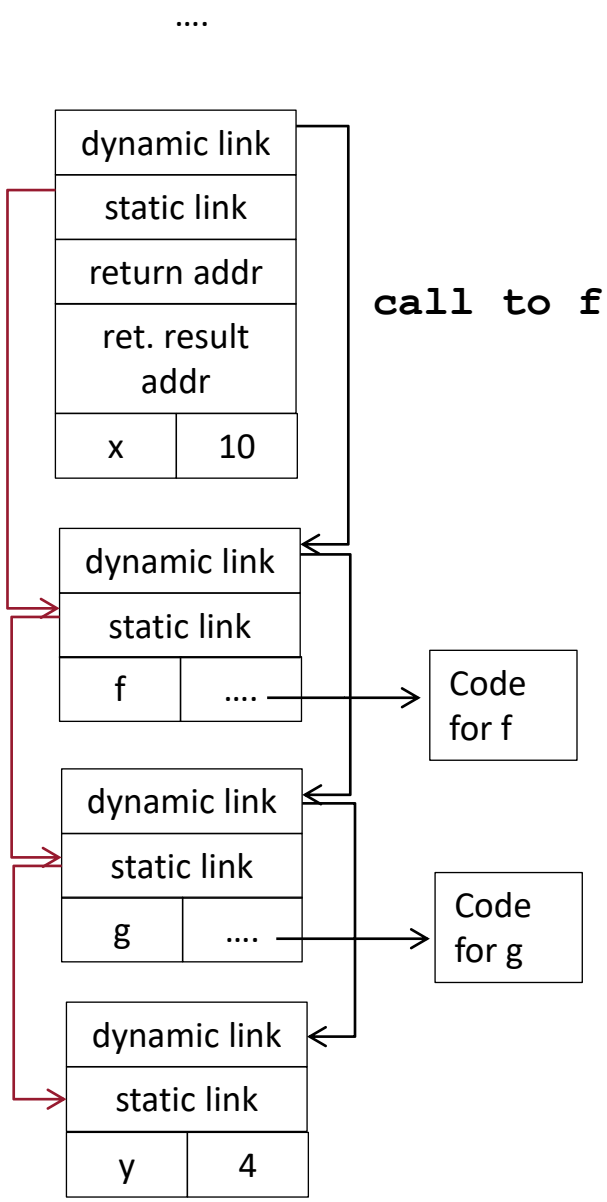


Activation Records - revisited

- Activation record with access link for function call with static scope




```
y=4
g x = x+y
f x =
    let
      y = 7
    in
      g  x
a = f 10
b = g 6
```



Scoping Example -2a

Breakout!

```
int m, n;
m = 50;
n = 100;
procedure egg();
begin
        print("in egg-- n = ", n);
end;

procedure chicken (n: integer);
begin
        print("in chicken -- m = ", m);
        print("in chicken -- n = ", n);
        egg();
end;
print("in main program -- n = ", n);
chicken(1);
egg();
```

The output using static scope rules:

```
in main program -- n =
in chicken -- m =
in chicken -- n =
in egg -- n =
/* note that here egg is called from chicken*/
in egg -- n =
/* here egg is called from the main program */
```

The output using dynamic scope rules:

```
in main program -- n =
in chicken -- m =
in chicken -- n =
in egg -- n =
/*NOTE DIFFERENCE -- here egg is called from
chicken
in egg-- n =
/* here egg is called from the main program*/
```

Scoping Example -2b- Nested Functions

```
int m, n;
m = 50;
n = 100;
procedure chicken (n: integer);
begin
        procedure egg();
        begin
                print("in egg-- n = ", n);
        end;

        print("in chicken- m = ", m);
        print("in chicken- n = ", n);
        egg();
end;
print("in main program -- n = ", n);
chicken(1);
/* can't call egg from the main program anymore */
```

- **Static scope**: use environment where function is defined
- **Dynamic scope**: use environment where function is called

The output using static scope rules:

```
in main program -- n =
in chicken -- m =
in chicken -- n =
in egg -- n =
```

Static Scoping - Why it matters?

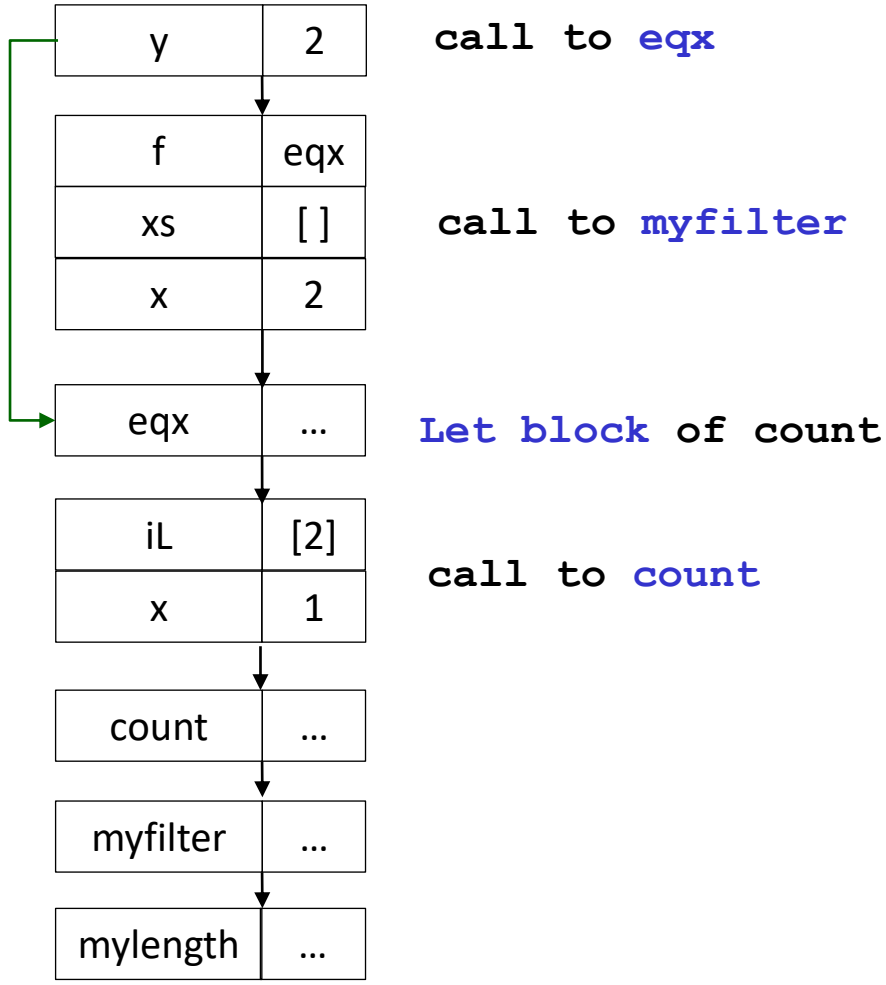
Example:

```
mylength [] = 0
mylength (x:xs) = 1 + length(xs)

myfilter f [] = []
myfilter f (x:xs) | (f x) = x:(myfilter f xs)
                  | otherwise = myfilter f xs

count x iL = let
               eqx y = (x==y)
             in
               mylength (myfilter eqx iL)
result = count 1 [2]
```

What result do we expect here? – 3-minute breakout



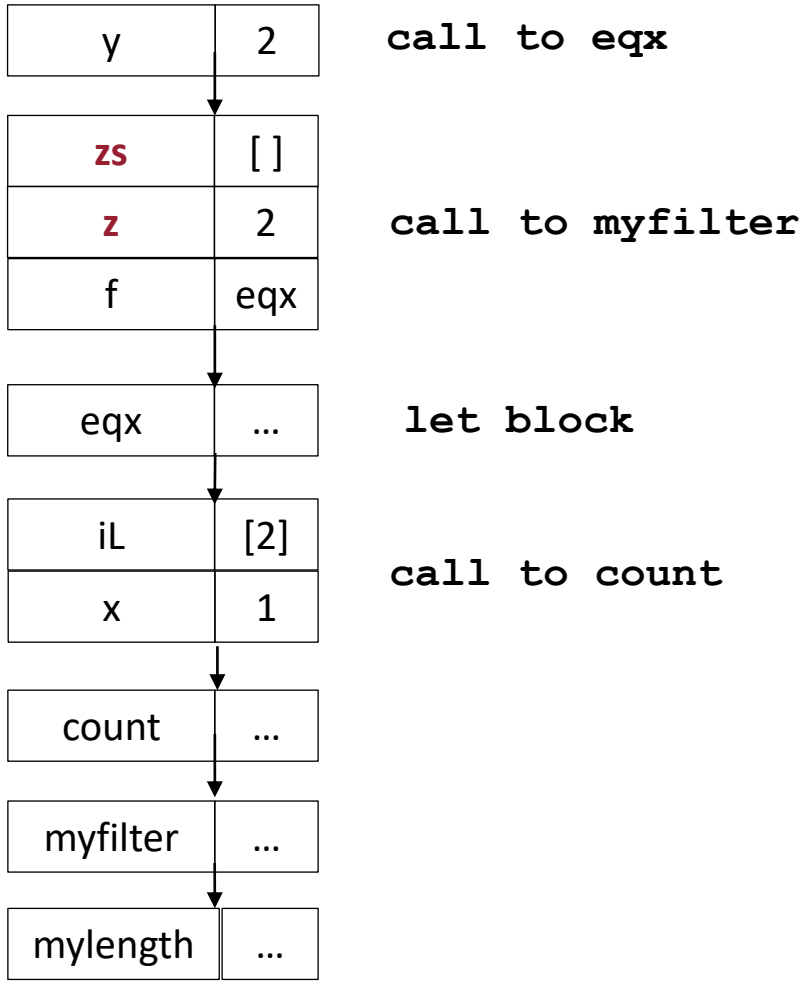
Static Scoping - Why it matters?

Example:

```
mylength [] = 0
mylength (x:xs)= 1 + length(xs)

myfilter f [] = []
myfilter f (z:zs)   | (f z) = z:(myfilter f zs)
                    | otherwise = myfilter f zs

count x iL = let
               eqx y = (x==y)
             in
               mylength (myfilter eqx iL)
result = count 1 [2]
```



Static Scoping - Why it matters?

Example:

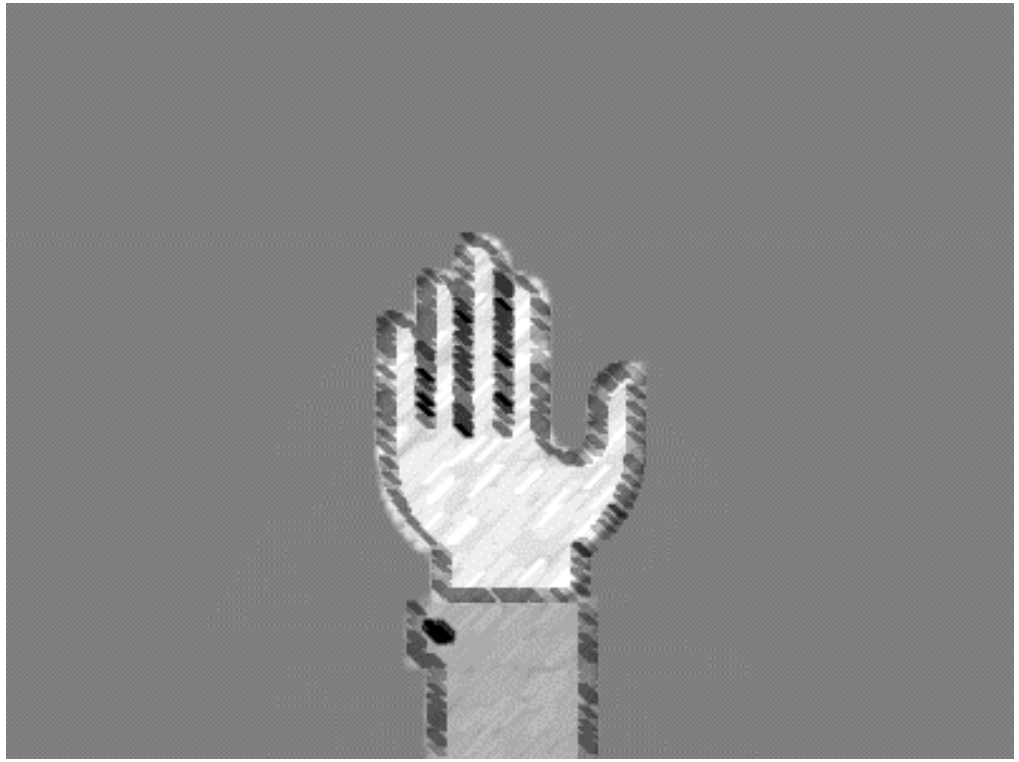
```
mylength [] = 0
mylength (x:xs)= 1 + length(xs)

myfilter f [] = []
myfilter f (x:xs) | (f x) = x:(myfilter f xs)
                  | otherwise = myfilter f xs

count x iL =let
                eqx y = (x==y)
              in
                mylength (myfilter eqx iL)
result = count 1 [2]
```

- Decades ago, both might have been considered *reasonable*, but now we know static scope makes much more sense
- Therefore, language designers have mostly concluded that the *static scope rule is preferable* to the dynamic scope rule
- Function meaning does not *depend on* variable names used
- Functions can be type-checked and reasoned about where they are defined

Questions?



Higher Order Functions – Functions as First-Class Values

- **First-class functions:** Can use them wherever we use values
  - Functions are values too
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, …

```
double x = 2*x
negate x = -1*x
f_tuple = (double, negate, double(negate 7))
```

- Most common use is as an argument / result of another function
  - Other function is called a higher-order function
  - Powerful way to factor out common functionality

Static Scope and Higher Order Functions

- The rule stays the same:
  - A function body is *evaluated* where the function body is *defined*; extended with the function argument
- Nothing changes to this rule when we take and return functions
  - Example:

```
f g =
   let
      x = 3
   in
      g 2
x = 4
h y = x + y
z = f h
```

`z` evaluates to ?

Questions?