

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

# Programming Languages

Dan Grossman  
2013

*A Key Benefit of Immutable Data*

# *A valuable non-feature: no mutation*

Have now covered all the features you need (and should use) on hw1

Now learn a very important **non-feature**

- Huh?? How could the *lack* of a feature be important?
- When it lets you know things *other* code will *not* do with your code and the results your code produces

A major aspect and contribution of functional programming:

Not being able to assign to (a.k.a. *mutate*) variables  
or parts of tuples and lists

(This is a “Big Deal”)

# *Cannot tell if you copy*

```
fun sort_pair (pr : int * int) =  
  if #1 pr < #2 pr  
  then pr  
  else (#2 pr, #1 pr)  
  
fun sort_pair (pr : int * int) =  
  if #1 pr < #2 pr  
  then (#1 pr, #2 pr)  
  else (#2 pr, #1 pr)
```

In ML, these two implementations of `sort_pair` are **indistinguishable**

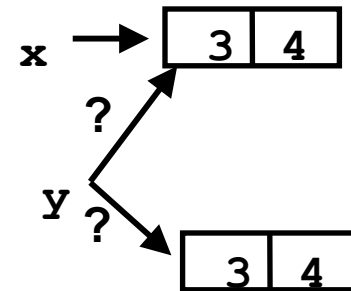
- But only because tuples are immutable
- The first is better style: simpler and avoids making a new pair in the then-branch
- In languages with mutable compound data, these are different!

# Suppose we had mutation...

```
val x = (3,4)
val y = sort_pair x
```

*somehow mutate #1 x to hold 5*

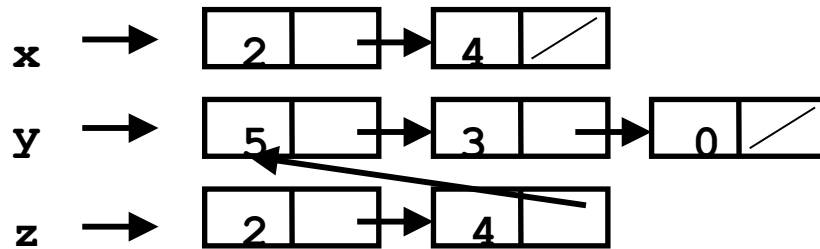
```
val z = #1 y
```



- What is `z`?
  - Would depend on how we implemented `sort_pair`
    - Would have to decide carefully and document `sort_pair`
  - But without mutation, we can implement “either way”
    - No code can ever distinguish aliasing vs. identical copies
    - No need to think about aliasing: focus on other things
    - Can use aliasing, which saves space, without danger

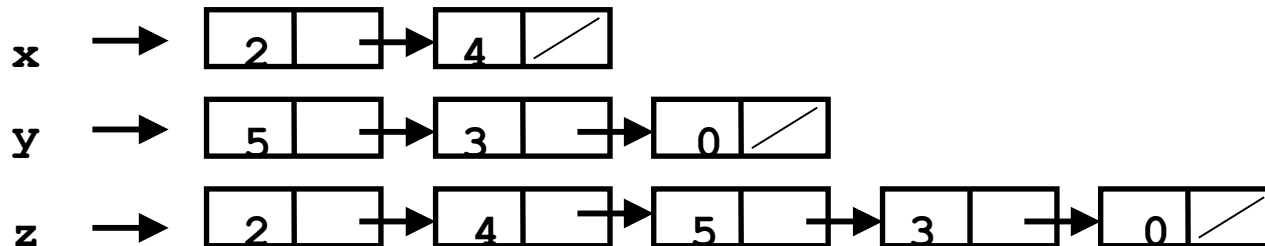
# An even better example

```
fun append (xs : int list, ys : int list) =  
  if null xs  
  then ys  
  else hd (xs) :: append (tl(xs), ys)  
val x = [2,4]  
val y = [5,3,0]  
val z = append(x,y)
```



*(can't tell,  
but it's the  
first one)*

or



# *ML vs. Imperative Languages*

- In ML, we create aliases all the time without thinking about it because it is *impossible* to tell where there is aliasing
  - Example: `tl` is constant time; does not copy rest of the list
  - So don't worry and focus on your algorithm
- In languages with mutable data (e.g., Java), programmers are *obsessed* with aliasing and object identity
  - They have to be (!) so that subsequent assignments affect the right parts of the program
  - Often crucial to make copies in just the right places
    - **Optional** Java example in next segment