Programming Languages

Pairs, Lists, Local Bindings, Benefit of No Mutation

Jiwon Seo

Related Sections in Elements of ML Programming

Section 2.1 (Expressions), 2.3 (Variable Bindings),
2.4 (Tuples and Lists), 3.1 (Functions)

Function bindings: 3 questions

- Syntax: `fun x0 (x1 : t1, … , xn : tn) = e`
  - (Will generalize in later lecture)

- Evaluation: ***A function is a value!*** (No evaluation yet)
  - Adds **`x0`** to environment so *later* expressions can *call* it
  - (Function-call semantics will also allow recursion)

- Type-checking:
  - Adds binding **`x0 : (t1 * … * tn) -> t`** if:
  - Can type-check body **`e`** to have type **`t`** in the static environment containing:
    - "Enclosing" static environment (earlier bindings)
    - **`x1 : t1, …, xn : tn`** (arguments with their types)
    - **`x0 : (t1 * … * tn) -> t`** (for recursion)

More on type-checking

```
fun x0 (x1 : t1, … , xn : tn) = e
```

- New kind of type: `(t1 * … * tn) -> t`
  - Result type on right
  - The overall type-checking result is to give **`x0`** this type in rest of program (unlike Java, not for earlier bindings)
  - Arguments can be used only in **`e`** (unsurprising)

- Because evaluation of a call to **`x0`** will return result of evaluating **`e`**, the return type of **`x0`** is the type of **`e`**

- The type-checker "magically" figures out **`t`** if such a **`t`** exists
  - Later lecture: Requires some cleverness due to recursion
  - More magic after hw1: Later can omit argument types too

Function Calls

A new kind of expression: 3 questions

Syntax: ***`e0 (e1,…,en)`***

- (Will generalize later)
- Parentheses optional if there is exactly one argument

Type-checking:

If:

- **`e0`** has some type **`(t1 * … * tn) -> t`**
- **`e1`** has type **`t1`**, …, **`en`** has type **`tn`**

Then:

- **`e0(e1,…,en)`** has type **`t`**

Example: **`pow(x,y-1)`** in our example has type **`int`**

Function-calls continued

`e0(e1,…,en)`

Evaluation:

1. (Under current dynamic environment,) evaluate **`e0`** to a function **`fun`** ***`x0`*** **`(`*****`x1`*** **`:`** ***`t1`*****`,`** **…** **`,`** ***`xn`*** **`:`** ***`tn`*****`)`** **`=`** ***`e`***
   – Since call type-checked, result *will be* a function

2. (Under current dynamic environment,) evaluate arguments to values **`v1,`** **…`,`** **`vn`**

3. Result is evaluation of ***`e`*** in an environment extended to map **`x1`** to **`v1`**, …, **`xn`** to **`vn`**
   – ("An environment" is actually the environment where the function was defined, and includes **`x0`** for recursion)

Functions as Parameters

In ML, functions can be passed as parameters of another function or returned from another function

```
fun apply_f(f:int*int -> int, x:int, y:int) =
   …



fun ret_f():int*int -> int =
   …
```

Debugging Errors

Your mistake could be:

- Syntax: What you wrote means nothing or not the construct you intended

- Type-checking: What you wrote does not type-check

- Evaluation: It runs but produces wrong answer, or an exception, or an infinite loop

Let’s see some error examples

So far: numbers, booleans, conditionals, variables, functions

- This is essential
- Java examples: classes with fields, arrays

Now:

- *Tuples*: fixed "number of pieces" that may have different types
- *Lists*: any "number of pieces" that all have the same type

Later:

- Other more general ways to create compound data

- The big thing we need: local bindings
  - For style and convenience
  - A big but natural idea: nested function bindings
  - For efficiency (***not*** "just a little faster")

- Why not having mutation (assignment statements) is a valuable language feature
  - No need for you to keep track of sharing/aliasing, which Java programmers must obsess about

Tuples and lists

So far: numbers, booleans, conditionals, variables, functions

- Now ways to build up data with multiple parts
- This is essential
- Java examples: classes with fields, arrays

Now:

- *Tuples*: fixed "number of pieces" that may have different types

Then:

- *Lists*: any "number of pieces" that all have the same type

Later:

- Other more general ways to create compound data

Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

Build:

- Syntax: `(e1,e2)`

- Evaluation: Evaluate **`e1`** to **`v1`** and **`e2`** to **`v2`**; result is **`(v1,v2)`**
  - A pair of values is a value

- Type-checking: If **`e1`** has type $\mathtt{t_a}$ and **`e2`** has type $\mathtt{t_b}$, then the pair expression has type $\mathtt{t_a * t_b}$
  - A new kind of type

Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

Access:

- Syntax: `#1 e` and `#2 e`
- Evaluation: Evaluate `e` to a pair of values and return first or second piece
  - Example: If `e` is a variable `x`, then look up `x` in environment
- Type-checking: If `e` has type t_a `*` t_b, then `#1 e` has type t_a and `#2 e` has type t_b

Examples

Functions can take and return pairs

```
fun swap (pr : int*bool) =

(* type? *)
fun sum_two_pairs


fun div_mod


fun sort_pair
```

Examples

Functions can take and return pairs

```
fun swap (pr : int*bool) =
   (#2 pr, #1 pr)

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) =
   (#1 pr1) + (#2 pr1) + (#1 pr2) + (#2 pr2)

fun div_mod (x : int, y : int) =
   (x div y, x mod y)

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

Tuples

Actually, you can have *tuples* with more than two parts

- A new feature: a generalization of pairs

- **`(e1,e2,…,en)`**
- **t_a `*` t_b `*` … `*` t_n**
- **`#1 e, #2 e, #3 e, …`**

Nesting

Pairs and tuples can be nested however you want

- Not a new feature: implied by the syntax and semantics

```
val x1 = (7,(true,9)) (* int * (bool*int) *)

val x2 = #1 (#2 x1)    (* bool *)

val x3 = (#2 x1)       (* bool*int *)

val x4 = ( (3,5),((4,8),(0,false)) )
         (* (int*int)*((int*int)*(int*bool)) *)
```

Lists

- Despite nested tuples, the type of a variable still “commits” to a particular “amount” of data

In contrast, a list:

  - Can have any number of elements
  - But all list elements have the same type

Need ways to *build* lists and *access* the pieces…

Building Lists

- The empty list is a value:

```
[]
```

- In general, a list of values is a value; elements separated by commas:

```
[e1, e2,…,en]
```

- If **`e1`** evaluates to **`v1`** and **`e2`** evaluates to a list **`[v2,…,vn]`**, then **`e1::e2`** evaluates to **`[v1,v2, …,vn]`**

```
e1::e2 (* pronounced "cons" *)
```

Accessing Lists

Until we learn pattern-matching, we will use three standard-library functions

- `null e` evaluates to `true` if and only if `e` evaluates to `[]`
- If `e` evaluates to `[v1,v2,…,vn]` then `hd e` evaluates to `v1`
  - (raise exception if `e` evaluates to `[]`)
- If `e` evaluates to `[v1,v2,…,vn]` then `tl e` evaluates to `[v2,…,vn]`
  - (raise exception if `e` evaluates to `[]`)
  - Notice result is a list

Type-checking list operations

Lots of new types: For any type **`t`**, the type **`t list`** describes lists where all elements have type **`t`**

- Examples: **`int list`** **`bool list`** **`int list list`** **`(int * int) list`** **`(int list * int) list`**

- So **`[]`** have type …

- For **`e1::e2`** to type-check, we need a **`t`** such that **`e1`** has type **`t`** and **`e2`** has type **`t list`**. Then the result type is **`t list`**
- **`null : 'a list -> bool`**
- **`hd   : 'a list -> 'a`**
- **`tl   : 'a list -> 'a list`**

Type-checking list operations

Lots of new types: For any type **`t`**, the type **`t list`** describes lists where all elements have type **`t`**

- Examples: **`int list`** **`bool list`** **`int list list`** **`(int * int) list`** **`(int list * int) list`**

- So **`[]`** can have type **`t list`** list for *any* type **`t`**
  - SML uses type **`'a list`** to indicate this ("quote a" or "alpha")
- For **`e1::e2`** to type-check, we need a **`t`** such that **`e1`** has type **`t`** and **`e2`** has type **`t list`**. Then the result type is **`t list`**
- **`null : 'a list -> bool`**
- **`hd   : 'a list -> 'a`**
- **`tl   : 'a list -> 'a list`**

Example list functions

```
fun sum_list (xs : int list) =



fun countdown (x : int) =



fun append (xs : int list, ys : int list) =
```

Example list functions

```
fun sum_list (xs : int list) =
  if null xs
  then 0
  else hd(xs) + sum_list(tl(xs))

fun countdown (x : int) =
  if x=0
  then []
  else x :: countdown (x-1)

fun append (xs : int list, ys : int list) =
  if null xs
  then ys
  else hd (xs) :: append (tl(xs), ys)
```

Recursion again

Functions over lists are usually recursive

  – Only way to "get to all the elements"

- What should the answer be for the empty list?
- What should the answer be for a non-empty list?
  – Typically in terms of the answer for the tail of the list!

Similarly, functions that produce lists of potentially any size will be recursive

  – You create a list out of smaller lists

Lists of pairs

Processing lists of pairs requires no new features. Examples:

```
fun sum_pair_list (xs : (int*int) list) =



fun firsts (xs : (int*int) list) =



fun seconds (xs : (int*int) list) =



fun sum_pair_list2 (xs : (int*int) list) =
```

Lists of pairs

Processing lists of pairs requires no new features. Examples:

```
fun sum_pair_list (xs : (int*int) list) =
  if null xs
  then 0
  else #1(hd xs) + #2(hd xs) + sum_pair_list(tl xs)

fun firsts (xs : (int*int) list) =
  if null xs
  then []
  else #1(hd xs) :: firsts(tl xs)

fun seconds (xs : (int*int) list) =
  if null xs
  then []
  else #2(hd xs) :: seconds(tl xs)

fun sum_pair_list2 (xs : (int*int) list) =
 (sum_list (firsts xs)) + (sum_list (seconds xs))
```

Let-expressions

3 questions:

- Syntax: `let b1 b2 … bn in e end`
  - Each ***bi*** is any *binding* and ***e*** is any *expression*

- Type-checking: Type-check each ***bi*** and ***e*** in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of ***e***.

- Evaluation: Evaluate each ***bi*** and ***e*** in a dynamic environment that includes the previous bindings.
  Result of whole let-expression is result of evaluating ***e***.

It is an expression

A let-expression is ***just an expression***, so we can use it ***anywhere*** an expression can go

Silly examples

```
fun silly1 (z : int) =
    let val x = if z > 0 then z else 34
        val y = x+z+9
    in
        if x > y then x*2 else y*y
    end
fun silly2 () =
    let val x = 1
    in
        (let val x = 2 in x+1 end) +
        (let val y = x+2 in y+1 end)
    end
```

`silly2` is poor style but shows let-expressions are expressions

- Can also use them in function-call arguments, if branches, etc.
- Also notice shadowing

What's new

- What's new is ***scope***: where a binding is in the environment
  - *In* later bindings and body of the let-expression
    - (Unless a later or nested binding shadows it)
  - *Only in* later bindings and body of the let-expression

- *Nothing else is new:*
  - Can put any binding we want, even function bindings
  - Type-check and evaluate just like at "top-level"

Any binding

According to our rules for let-expressions, we can define functions inside any let-expression

```
let  b1 b2 … bn  in  e  end
```

This is a natural idea, and often good style

(Inferior) Example

```
fun countup_from1 (x : int) =
    let fun count (from : int, to : int) =
            if from = to
            then to :: []
            else from :: count(from+1,to)
    in
        count (1,x)
    end
```

- This shows how to use a local function binding, but:
  - Better version on next slide
  - `count` might be useful elsewhere

Better:

```
fun countup_from1_better (x : int) =
    let fun count (from : int) =
            if from = x
            then x :: []
            else from :: count(from+1)
    in
        count 1
    end
```

- Functions can use bindings in the environment where they are defined:
  - Bindings from “outer” environments
    - Such as parameters to the outer function
  - Earlier bindings in the let-expression
- Unnecessary parameters are usually bad style
  - Like `to` in previous example

Nested functions: style

- Good style to define helper functions inside the functions they help if they are:
  - Unlikely to be useful elsewhere
  - Likely to be misused if available elsewhere
  - Likely to be changed or removed later

- A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later

Avoid repeated recursion

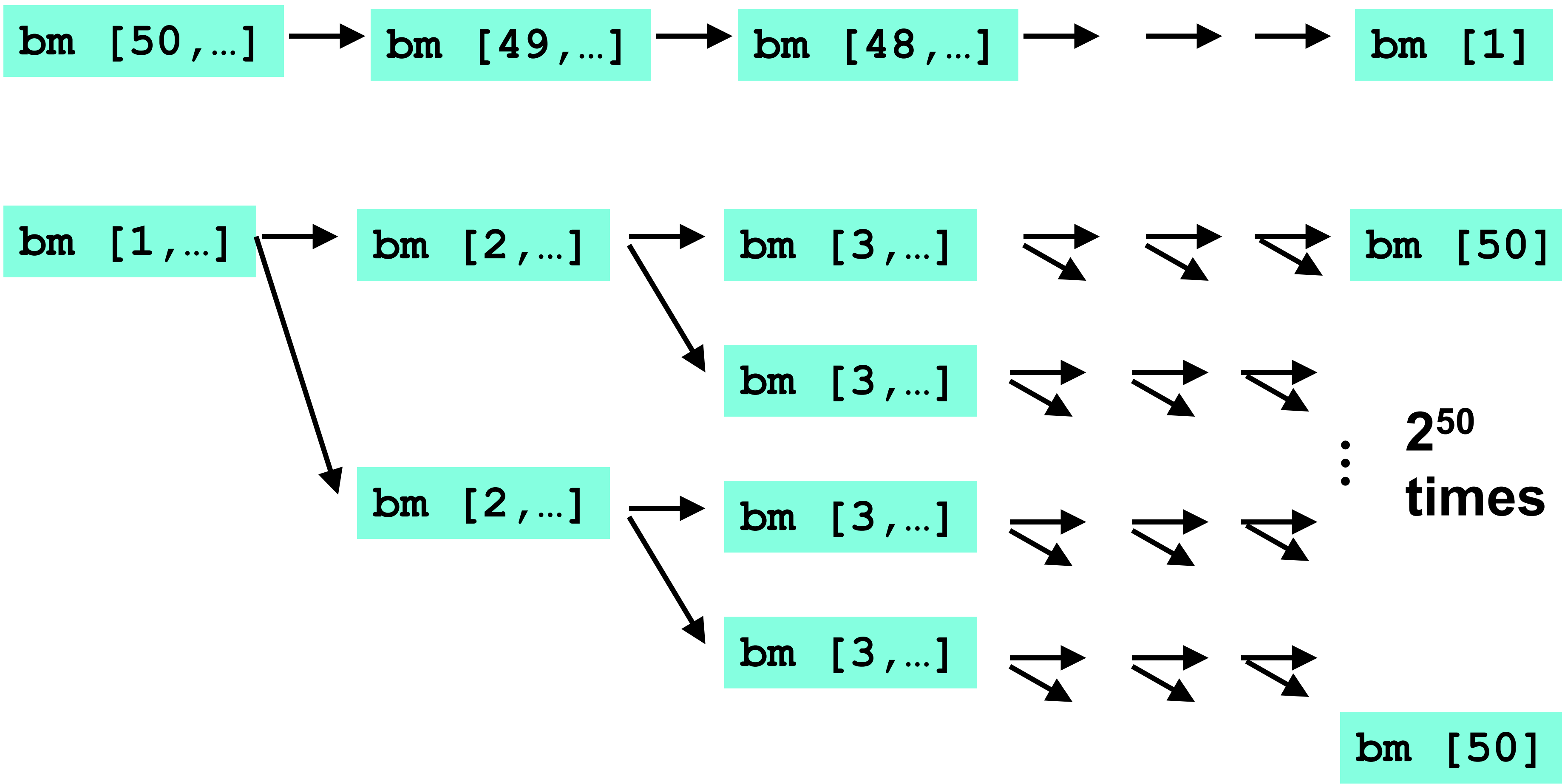
Consider this code and the recursive calls it makes

- Don't worry about calls to **`null`**, **`hd`**, and **`tl`** because they do a small constant amount of work

```
fun bad_max (xs : int list) =
     if null xs
     then 0 (* horrible style; fix later *)
     else if null (tl xs)
     then hd xs
     else if hd xs > bad_max (tl xs)
     then hd xs
     else bad_max (tl xs)

let x = bad_max [50,49,…,1]
let y = bad_max [1,2,…,50]
```

Fast vs. unusable

```
if hd xs > bad_max (tl xs)
then hd xs
else bad_max (tl xs)
```



Math never lies

Suppose one **`bad_max`** call's if-then-else logic and calls to **`hd`**, **`null`**, **`tl`** take 10^{-7} seconds

- Then **`bad_max [50,49,…,1]`** takes 50 x 10^{-7} seconds
- And **`bad_max [1,2,…,50]`** takes 1.12 x 10^{8} seconds
  - (over 3.5 years)
  - **`bad_max [1,2,…,55]`** takes over 1 century
  - Buying a faster computer won't help much ☺

The key is not to do repeated work that might do repeated work that might do…

- Saving recursive results in local bindings is essential…

Efficient max

```
fun good_max (xs : int list) =
    if null xs
    then 0 (* horrible style; fix later *)
    else if null (tl xs)
    then hd xs
    else
         let val tl_ans = good_max(tl xs)
         in
             if hd xs > tl_ans
             then hd xs
             else tl_ans
         end
```

Fast vs. fast

```
let val tl_ans = good_max(tl xs)
in
    if hd xs > tl_ans
    then hd xs
    else tl_ans
end
```

gm [50,…] → gm [49,…] → gm [48,…] → → → gm [1]

gm [1,…] → gm [2,…] → gm [3,…] → → → gm [50]

Options

- **`t option`** is a type for any type **`t`**
  - (much like **`t list`**, but a different type, not a list)

Building:

- **`NONE`** has type **`'a option`** (much like **`[]`** has type **`'a list`**)
- **`SOME e`** has type **`t option`** if **`e`** has type **`t`** (much like **`e::[]`**)

Accessing:

- **`isSome`** has type **`'a option -> bool`**
- **`valOf`** has type **`'a option -> 'a`** (exception if given **`NONE`**)

Example

```
fun better_max (xs : int list) =
    if null xs
    then NONE
    else
        let val tl_ans = better_max(tl xs)
        in
            if isSome tl_ans
               andalso valOf tl_ans > hd xs
            then tl_ans
            else SOME (hd xs)
        end
```

```
val better_max = fn : int list -> int option
```

Example

```
fun better_max (xs : int list) =
    if null xs
    then NONE
    else
         let val tl_ans = better_max(tl xs)
         in
             if isSome tl_ans
                andalso valOf tl_ans > hd xs
             then tl_ans
             else SOME (hd xs)
         end
```

```
val better_max = fn : int list -> int option
```

- Nothing wrong with this, but as a matter of style might prefer not to do so much useless "**`valOf`**" in the recursion

Example variation

```
fun better_max2 (xs : int list) =
    if null xs
    then NONE
    else let (* ok to assume xs nonempty b/c local *)
             fun max_nonempty (xs : int list) =
               if null (tl xs)
               then hd xs
               else
                  let val tl_ans = max_nonempty(tl xs)
                  in
                    if hd xs > tl_ans
                    then hd xs
                    else tl_ans
                  end
         in
            SOME (max_nonempty xs)
         end
```

Cannot tell if you copy

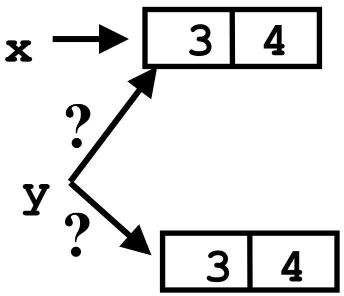
```
fun sort_pair (pr : int * int) =
  if #1 pr < #2 pr
  then pr
  else (#2 pr, #1 pr)

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…

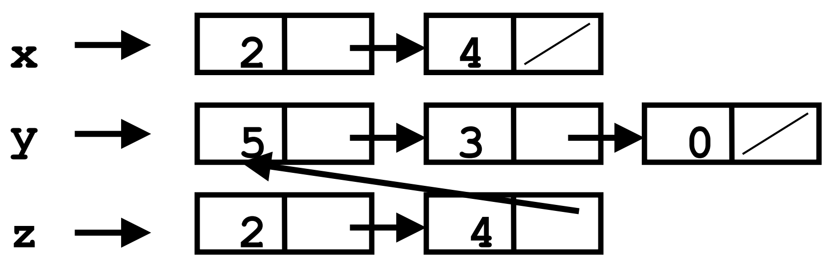
```
val x = (3,4)
val y = sort_pair x

somehow mutate #1 x to hold 5

val z = #1 y
```



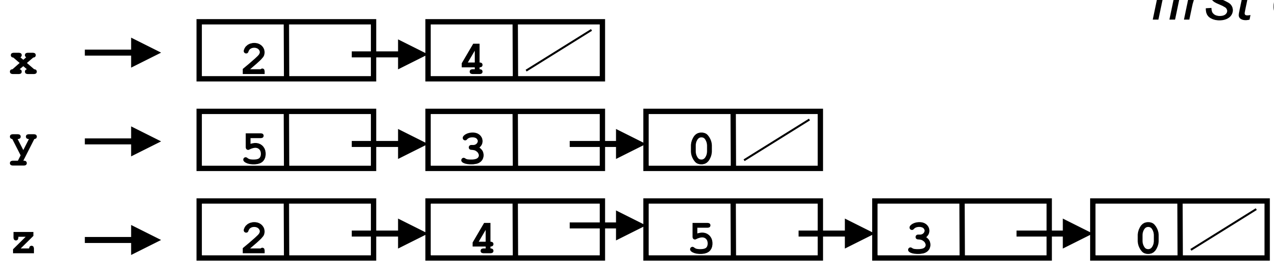
- What is **`z`**?
  - Would depend on how we implemented **`sort_pair`**
    - Would have to decide carefully and document **`sort_pair`**
  - But without mutation, we can implement “either way”
    - No code can ever distinguish aliasing vs. identical copies
    - No need to think about aliasing: focus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



or



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

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
    - Consider a Java example…

Java security nightmare (bad code)

```
class ProtectedResource {
   private Resource theResource = ...;
   private String[] allowedUsers = ...;
   public String[] getAllowedUsers() {
      return allowedUsers;
   }
   public String currentUser() { ... }
   public void useTheResource() {
      for(int i=0; i < allowedUsers.length; i++) {
         if(currentUser().equals(allowedUsers[i])) {
             ... // access allowed: use it
             return;
         }
      }
      throw new IllegalAccessException();
   }
}
```

Have to make copies

The problem:

```
p.getAllowedUsers()[0] = p.currentUser();
p.useTheResource();
```

The fix:

```
public String[] getAllowedUsers() {
    … return a copy of allowedUsers …
}
```

Reference (alias) vs. copy doesn’t matter if code is immutable!