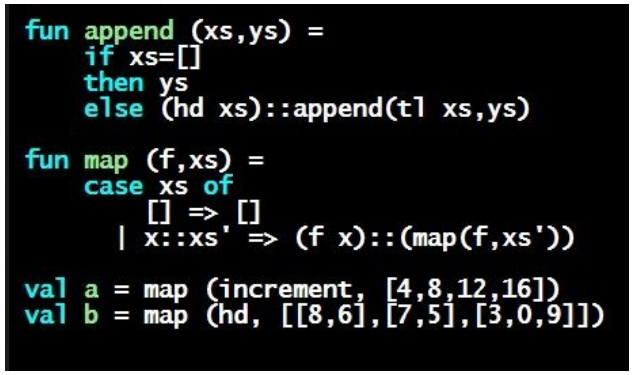


```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

*ML Expressions and Variable Bindings*

# *Mindset*

- "Let go" of all programming languages you already know
- For now, treat ML as a "totally new thing"
  - Time later to compare/contrast to what you know
  - For now, "oh that seems kind of like this thing in [Java]" will confuse you, slow you down, and you will learn less
- Start from a blank file…

# *A very simple ML program*

[The same program we just wrote in Emacs; here for conveniene if reviewing the slides]

```
(* My first ML program *)

val x = 34;

val y = 17;

val z = (x + y) + (y + 2);

val q = z + 1;

val abs_of_z = if z < 0 then 0 - z else z;

val abs_of_z_simpler = abs z
```

# A variable binding

```
val z = (x + y) + (y + 2); (* comment *)
```

*More generally:*

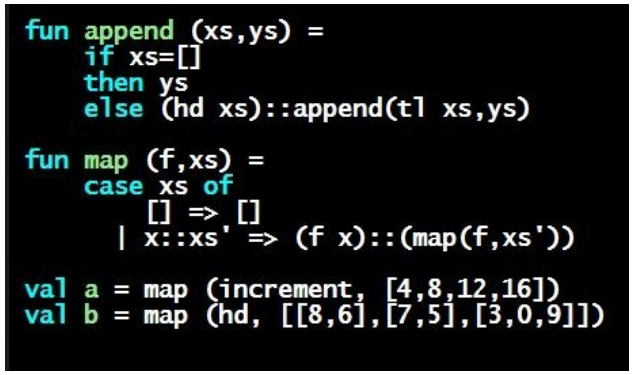
```
val x = e;
```

- *Syntax*:
  - *Keyword* **`val`** and *punctuation* **=** and **`;`**
  - *Variable* ***x***
  - *Expression* ***e***
    - Many forms of these, most containing *subexpressions*

# *The semantics*

- Syntax is just how you write something
- Semantics is what that something means
  - Type-checking (before program runs)
  - Evaluation (as program runs)
- For variable bindings:
  - Type-check expression and extend static environment
  - Evaluate expression and extend dynamic environment

So what is the precise syntax, type-checking rules, and evaluation rules for various expressions? Good question!




# Programming Languages

Dan Grossman

2013

*ML Rules for Expressions (Seen So Far)*

# *A very simple ML program*

This program has integers, variables, addition, if-expressions, less-than, subtraction, and calling a pre-defined function

```
(* My first ML program *)

val x = 34;

val y = 17;

val z = (x + y) + (y + 2);

val q = z + 1;

val abs_of_z = if z < 0 then 0 - z else z;

val abs_of_z_simpler = abs z
```

# *Expressions*

- We have seen many kinds of expressions:

  **`34`** **`true`** **`false`** **`x`** ***`e1+e2`*** ***`e1<e2`***

  **`if`** ***`e1`*** **`then`** ***`e2`*** **`else`** ***`e3`***

- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.

- Every kind of expression has
  1. Syntax
  2. Type-checking rules
     - Produces a type or fails (with a bad error message ☹)
     - Types so far: **`int`** **`bool`** **`unit`**
  3. Evaluation rules (used only on things that type-check)
     - Produces a value (or exception or infinite-loop)

# *Variables*

- Syntax:
  sequence of letters, digits, _, not starting with digit

- Type-checking:
  Look up type in current static environment
  - If not there, fail

- Evaluation:
  Look up value in current dynamic environment

# *Addition*

- Syntax:

  ***e1*** **`+`** ***e2*** where ***e1*** and ***e2*** are expressions

- Type-checking:

  If ***e1*** and ***e2*** have type **`int`**,

  then ***e1*** **`+`** ***e2*** has type **`int`**

- Evaluation:

  If ***e1*** evaluates to **`v1`** and ***e2*** evaluates to **`v2`**,

  then ***e1*** **`+`** ***e2*** evaluates to sum of **`v1`** and **`v2`**

# *Values*

- All values are expressions
- Not all expressions are values
- Every value "evaluates to itself" in "zero steps"
- Examples:
  - **`34`**, **`17`**, **`42`** have type **`int`**
  - **`true`**, **`false`** have type **`bool`**
  - **`()`** has type **`unit`**

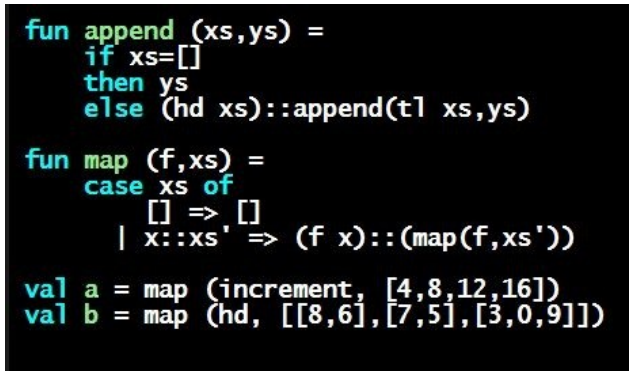
# *A slightly tougher one*

*What are the syntax, typing rules, and evaluation rules for conditional expressions?*

Let's write it out…

# *Now you try one*

Syntax, type-checking rules, and evaluation rules for less-than comparisons?




# Programming Languages

Dan Grossman
2013

*The REPL and Error Messages*

# *Pragmatics*

Last two segments have built up key conceptual foundation

But you also need some pragmatics:

- How do we run programs using the REPL?
- What happens when we make mistakes?

Work on developing resilience to mistakes

# *use*

**`use "foo.sml"`** is an unusual expression

It enters bindings from the file **`foo.sml`**

Result is **`()`** bound to variable **`it`**

- Ignorable

# *The REPL*

- Read-Eval-Print-Loop is well named

- Can just treat it as a strange/convenient way to run programs
  - But more convenient for quick try-something-out
  - Then move things over to a testing file for easy reuse

- For reasons discussed in next segment, do *not* use **`use`** without restarting the REPL session
  - (But using it for multiple files at beginning of session is okay)

# *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

Keep these straight when debugging even if sometimes one kind of mistake appears to be another

# *Play around*

Best way to learn something: Try lots of things and don't be afraid of errors

- Slow down
- Don't panic
- Read what you wrote very carefully

Maybe watching me make a few mistakes will help…

```
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

*Shadowing*

# *Multiple bindings of same variable*

Multiple variable bindings of the same variable is often poor style

- Often confusing

But it's an instructive exercise

- Helps explain how the environment "works"
- Helps explain how a variable binding "works"

(Emphasize this now to lay the foundation for first-class functions)

# *Our example*

```
val a = 10

val b = a * 2

val a = 5

val c = b

val d = a

val a = a + 1

(* val g = f - 3 *)(* does not type-check *)

val f = a * 2
```

# *Two reasons (either one sufficient)*

```
val a = 1
val b = a (* b is bound to 1 *)
val a = 2
```

1. Expressions in variable bindings are evaluated "eagerly"
   - Before the variable binding "finishes"
   - Afterwards, the expression producing the value is irrelevant

2. There is no way to "assign to" a variable in ML
   - Can only shadow it in a later environment

# *use*

This is why I am so insistent about not reusing **`use`** on a file without restarting the REPL

Else you are introducing some of the same bindings again

- May make it seem like wrong code is correct
- May make it seem like correct code is wrong
- (It's all well-defined, but we humans get confused)

```
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

*Functions (Informally)*

# *Function definitions*

Functions: the most important building block in the whole course

- Like Java methods, have arguments and result
- But no classes, **`this`**, **`return`**, etc.

Example *function binding*:

```
(* Note: correct only if y>=0 *)

fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)
```

Note: The *body* includes a (recursive) *function call*: **`pow(x,y-1)`**

# *Example, extended*

```
fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)

fun cube (x : int) =
  pow (x,3)

val sixtyfour = cube 4

val fortytwo = pow(2,2+2) + pow(4,2) + cube(2) + 2
```

# *Some gotchas*

Three common "gotchas"

- Bad error messages if you mess up function-argument syntax
- The use of `*` in type syntax is not multiplication
  - Example: `int * int -> int`
  - In expressions, `*` is multiplication: `x * pow(x,y-1)`
- Cannot refer to later function bindings
  - That's simply ML's rule
  - Helper functions must come before their uses
  - Need special construct for *mutual recursion* (later)

# *Recursion*

- If you're not yet comfortable with recursion, you will be soon ☺
  - Will use for most functions taking or returning lists

- "Makes sense" because calls to same function solve "simpler" problems

- Recursion more powerful than loops
  - We won't use a single loop in ML
  - Loops often (not always) obscure simple, elegant solutions

```
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

*Functions (Formally)*

# *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 previous example has type **int**

# *Function-calls continued*

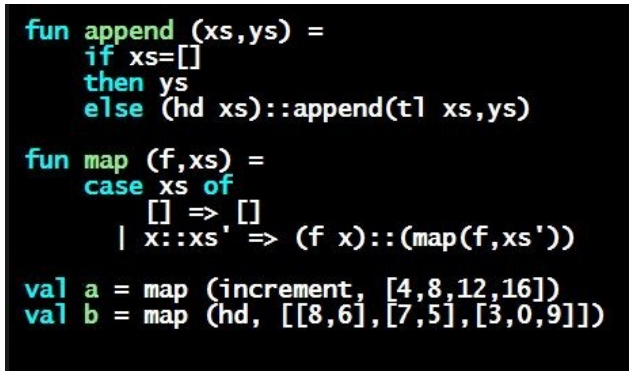
```
e0(e1,…,en)
```

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)



```
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

*Pairs and Other Tuples*

# *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

Coming soon:

- *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 **`ta`** and **`e2`** has type **`tb`**, then the pair expression has type **`ta * tb`**
  - 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 `ta * tb`, then `#1 e` has type `ta` and `#2 e` has type `tb`

# *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)`
- `ta * tb * … * tn`
- `#1 e, #2 e, #3 e, …`

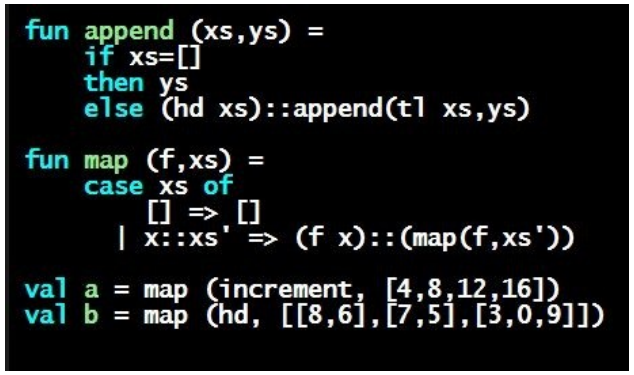
Homework 1 uses triples of type `int*int*int` a lot

# *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),(0,0)))
         (* (int*int)*((int*int)*(int*int)) *)
```



```
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

*Introducing Lists*

# *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:

```
[v1,v2,…,vn]
```

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

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

# *Accessing Lists*

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

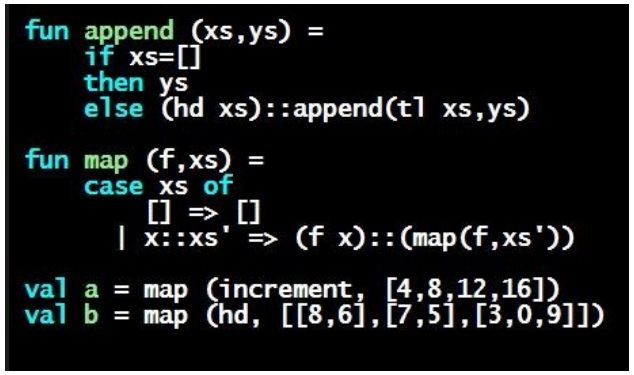
- `null e` evaluates to `true` if and only if `e` evaluates to `[]`
- If `e` evaluates to `[v1,v2,…,vn]` then `hd e` evaluates to `v1`
  - (raise exception if `e` evaluates to `[]`)
- If `e` evaluates to `[v1,v2,…,vn]` then `tl e` evaluates to `[v2,…,vn]`
  - (raise exception if `e` evaluates to `[]`)
  - 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 `[]` can have type `t list` list for *any* type
  - 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`



```
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

*List Functions*

# *Functions over lists*

Gain experience with lists and recursion by writing several functions that process and/or produce lists…

# 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!

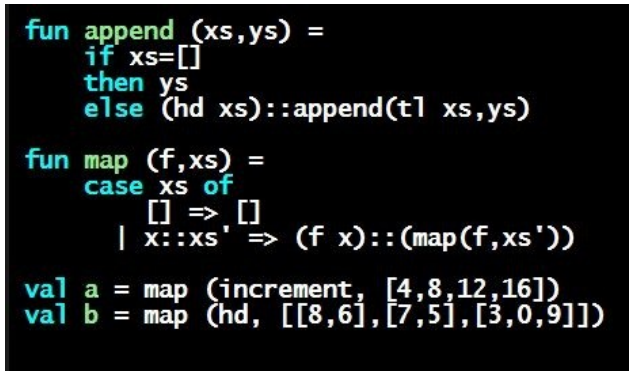
Similarly, functions that produce lists of potentially any size will be recursive

- 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) =
  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))
```




# Programming Languages

Dan Grossman
2013

*Let Expressions*

# *Review*

Huge progress already on the core pieces of ML:

- Types: **`int bool unit  t1*…*tn  t list  t1*…*tn->t`**
  - Types "nest" (each **`t`** above can be itself a compound type)
- Variables, environments, and basic expressions
- Functions
  - Build: **`fun x0 (x1:t1, …, xn:tn) = e`**
  - Use: **`e0 (e1, …, en)`**
- Tuples
  - Build: **`(e1, …, en)`**
  - Use: **`#1 e, #2 e, …`**
- Lists
  - Build: **`[]  e1::e2`**
  - Use: **`null e  hd e  tl e`**

# *Now…*

The big thing we need: local bindings

- For style and convenience

This segment:

- Basic let-expressions

Next segments:

- A big but natural idea: nested function bindings
- For efficiency (***not*** "just a little faster")

The construct to introduce local bindings is ***just an expression***, so we can use it anywhere an expression can go

# *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***.

# *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
```

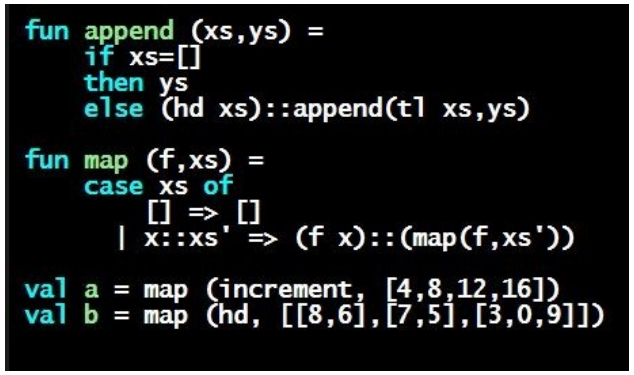
**`silly2`** is poor style but shows let-expressions are expressions

- 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"



```
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

*Nested Functions*

# *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
```

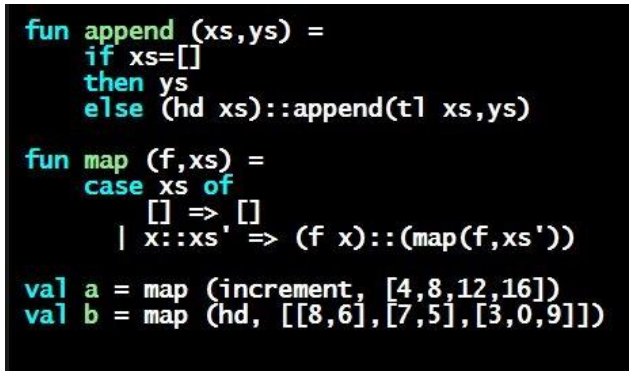
- Functions can use bindings in the environment where they are defined:
  - Bindings from “outer” environments
    - Such as parameters to the outer function
  - Earlier bindings in the let-expression

- 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




# Programming Languages

Dan Grossman

2013

## Let Expressions to Avoid Repeated Computation

# *Avoid repeated recursion*

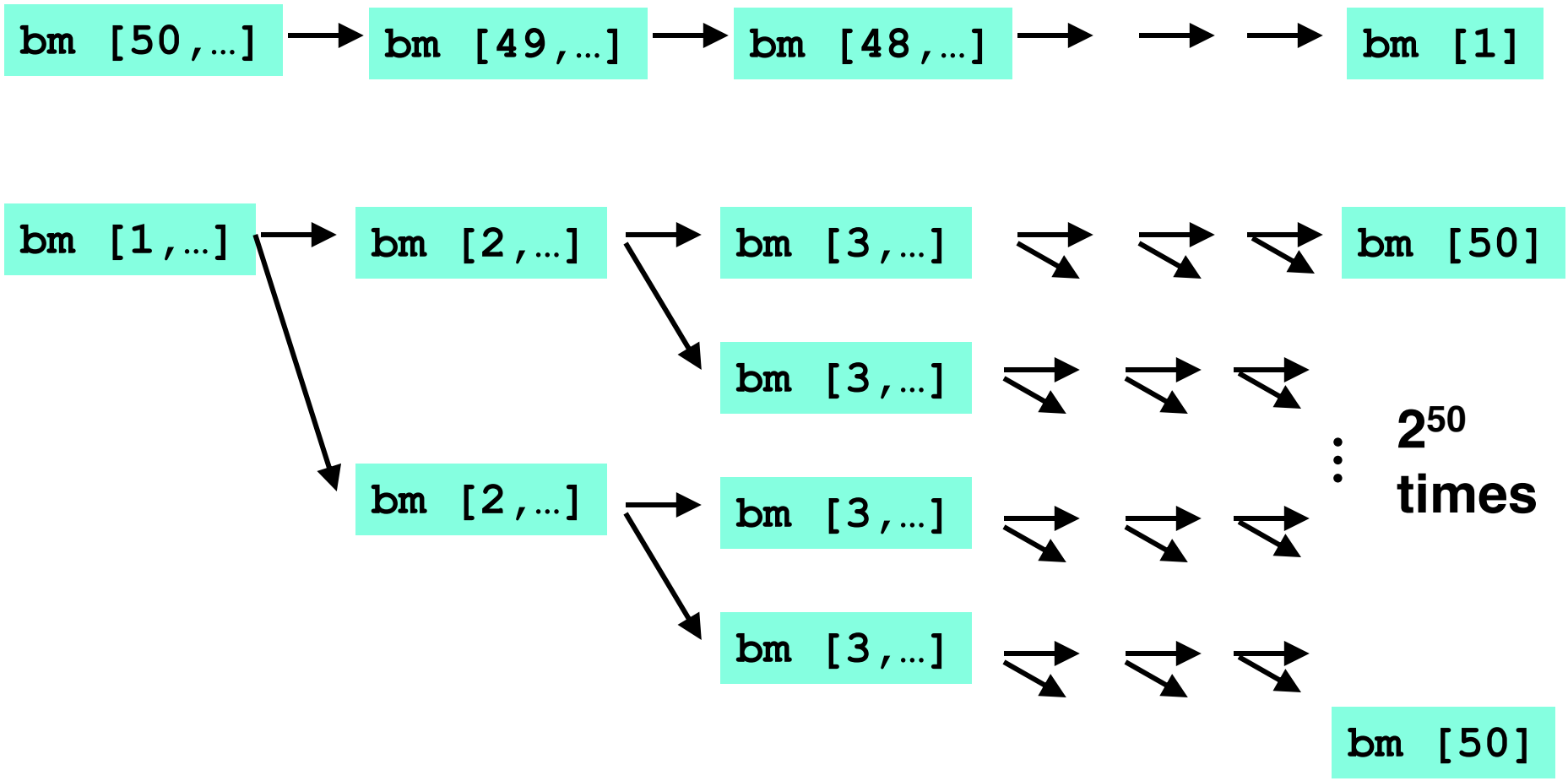
Consider this code and the recursive calls it makes

- 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…

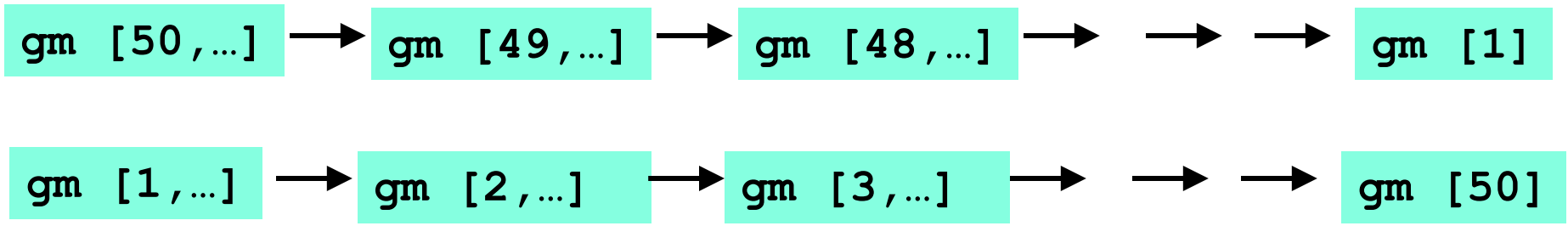
- Saving recursive results in local bindings is essential…

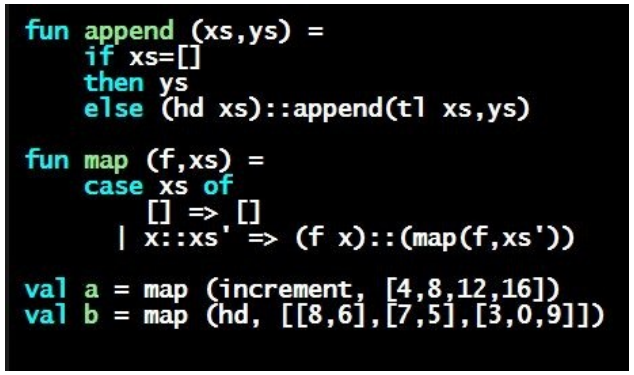
# *Efficient max*

```
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
```





```
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

*Options*

# *Motivating Options*

Having **`max`** return 0 for the empty list is really awful

- Could raise an *exception* (future topic)
- Could return a zero-element or one-element list
  - That works but is poor style because the built-in support for *options* expresses this situation directly

# *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

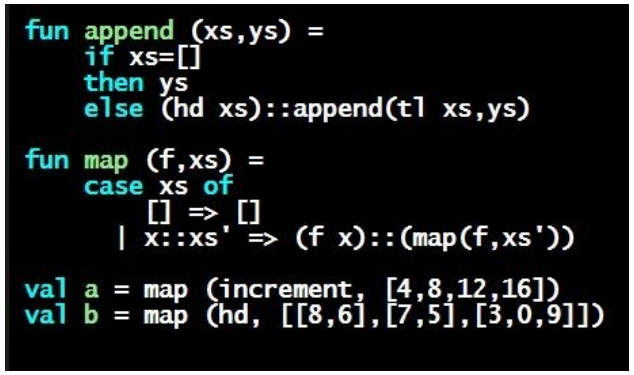
```
fun better_max (xs : int list) =
    if null xs
    then NONE
        else
          let val tl_ans = better_max(tl xs)
          in
              if isSome tl_ans
                 andalso valOf tl_ans > hd xs
              then tl_ans
              else SOME (hd xs)
          end
```

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

- 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
```




# Programming Languages

Dan Grossman

2013

*More Boolean and Comparison Expressions*

# *Some More Expressions*

Some “odds and ends” that haven’t come up much yet:

- Combining Boolean expressions (and, or, not)
- Comparison operations

# *Boolean operations*

```
e1 andalso e2
```

- Type-checking: ***e1*** and ***e2*** must have type **`bool`**
- Evaluation: If result of ***e1*** is **`false`** then **`false`** else result of ***e2***

```
e1 orelse e2
```

```
not e1
```

- Syntax in many languages is **`e1 && e2`**, **`e1 || e2`**, **`!e`**
  - **`&&`** and **`||`** don't exist in ML and **`!`** means something different

- "Short-circuiting" evaluation means **`andalso`** and **`orelse`** are not functions, but **`not`** is just a pre-defined function

# *Style with Booleans*

Language does not *need* **`andalso`**, **`orelse`**, **`not`**

```
(* e1 andalso e2 *)
if e1
then e2
else false
```

```
(* e1 orelse e2 *)
if e1
then true
else e2
```

```
(* not e1 *)
if e1
then false
else true
```

Using more concise forms generally much better style

And definitely plea

```
(* just say e (!!!) *)
if e
then true
else false
```

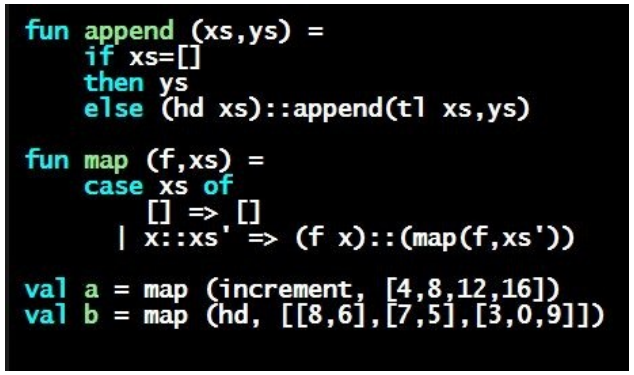
# *Comparisons*

For comparing **`int`** values:

**`=  <>  >  <  >=  <=`**

You might see weird error messages because comparators can be used with some other types too:

- **`> < >= <=`** can be used with **`real`**, but not 1 **`int`** and 1 **`real`**
- **`= <>`** can be used with any "equality type" but not with **`real`**
  - Let's not discuss equality types yet




# 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*

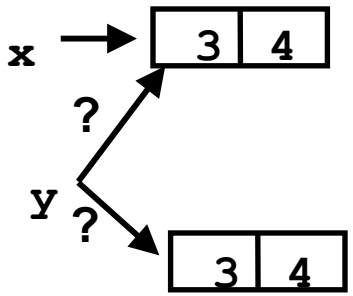
```
fun sort_pair (pr : int * int) =
  if #1 pr < #2 pr
  then pr
  else (#2 pr, #1 pr)

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…*

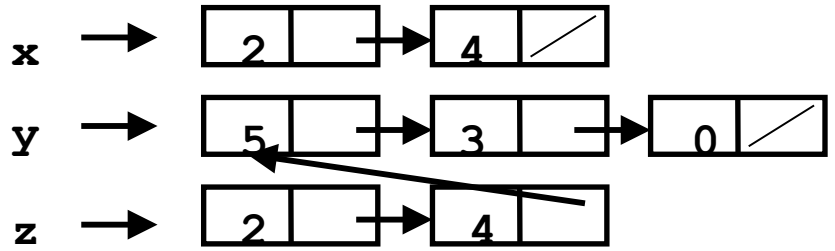
```
val x = (3,4)
val y = sort_pair x

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

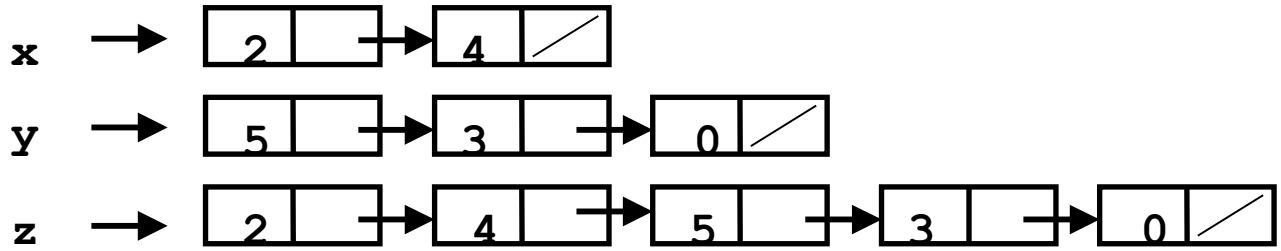
# An even better example

```
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)*

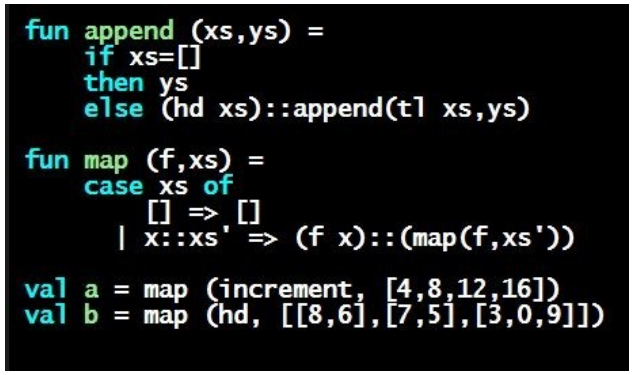
***or***



# *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



```
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

*Optional: Java Mutation Bug*

# *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…

# *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*

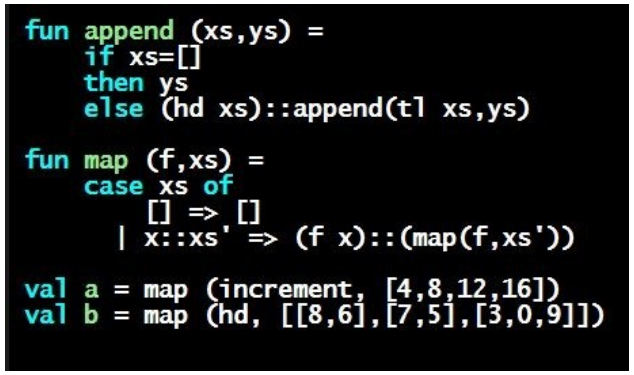
The problem:

```
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!




# Programming Languages

Dan Grossman

2013

*The Pieces of Learning a Language*

# *Five different things*

1. Syntax: How do you write language constructs?
2. Semantics: What do programs mean? (Evaluation rules)
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a well-known project) provide "standard"? (E.g., file access, data structures)
5. Tools: What do language implementations provide to make your job easier? (E.g., REPL, debugger, code formatter, …)
   - Not actually part of the language

These are 5 separate issues

- In practice, all are essential for good programmers
- Many people confuse them, but shouldn't

# *Our Focus*

This course focuses on semantics and idioms

- Syntax is usually uninteresting
  - A fact to learn, like “The American Civil War ended in 1865”
  - People obsess over subjective preferences

- Libraries and tools crucial, but often learn new ones “on the job”
  - We are learning semantics and how to use that knowledge to understand all software and employ appropriate idioms
  - By avoiding most libraries/tools, our languages may look “silly” but so would *any* language used this way