CSC 330 / Spring 2026

OCaml
Local Bindings, Options

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

OCaml so far

- Types: `int`, `bool`, `t1 -> t2`, `t1 * t2 * ... * tN`, `t list`
- Expressions:

```
34                  x           true                  e1 + e2   if e1 then e2 else e3
f (e1, ..., eN)     f e1        (e1, e2, ..., eN)     fst       snd
[]                  e1 :: e2    l = []                List.hd   List.tl
```

- Bindings

```
let x = e                   let [rec] f ((x1 : t1), ..., (xN : tN)) = e
```

let expressions

Local bindings (a.k.a. let expressions)

- Expressions can have their own local (private) variable and function bindings
- Why? For…
  - style and convenience: let's name intermediate results
  - consistency: everything else can nest
  - efficiency: avoid unnecessary recomputation
    - not *just a little faster*!
  - safety and maintenance: hide implementation details from clients

Let expressions

- **Syntax:**

  `let x = e1 in e2`

  where `e1` and `e2` are expressions

- **Type checking:**

  - If `e1` has type `t1` in current static environment,
    and `e2` has type `t2` in static environment extended with `x` $\mapsto$ `t1`
    - then: `let x = e1 in e2` has type `t2`
    - else: report error and fail

Let expressions

- **Evaluation**:

  `let x = e1 in e2`

  - If `e1` evaluates to `v1` in current dynamic environment,
    and `e2` evaluates to `v2` in dynamic environment *extended with* `x ↦ v1`:

    - then: `let x = e1 in e2` evaluates to value `v2`

Let expressions are just expressions

A let expression can go **anywhere** an expression can go because it is an expression!

- Also: *multiple bindings* is just nested let-expressions
  - But as style, don't indent them that way

Local bindings: what's new?

- Scope:
  - More control over which parts of a program can access a binding
  - For let expressions, the scope is **just the body** of the let
- Nothing else is really new
  - let *expressions* just work pretty much like top-level let *bindings*

Let expressions: local function bindings

- **Syntax**:

  ```
  let [rec] f ((x1 : t1), ..., (xN : tN)) = e in e2
  ```

- **Type checking**: same as top-level function binding, using static environment where the let expression occurs
- **Evaluation**: same as top-level function binding, using dynamic environment where the let expression occurs
- **Local binding**: `f` used only in `e` (if `rec` present), and `e2` (always)

Let expressions: an example

- Not great style:

```
let nats_upto (x : int) =
  let rec range ((lo : int), (hi : int)) =
    if lo < hi then
      lo :: range (lo + 1, hi)
    else
      []
  in
  range (0, x)
```

```
nats_upto 5 ;;
- : int list = [0; 1; 2; 3; 4]
```

  - `range` is hidden… No one else can use it!
  - Also, does `range` really need the `hi` parameter?

Let expressions: an example

- Better style:

```
let nats_upto (x : int) =
  let rec loop (lo : int) =
    if lo < x then
      lo :: loop (lo + 1)
    else
      []
  in
  loop 0
```

```
nats_upto 5 ;;
- : int list = [0; 1; 2; 3; 4]
```

- Functions can use bindings from the environment where they are defined, including:
  - “outer” environments, parameters to outer function, etc.
- Unnecessary parameters are *bad style*

Nested functions: style

- Good style to define helper functions inside if they are:
  - Not useful elsewhere (avoid clutter)
  - Likely to be misused elsewhere (improve reliability)
  - Likely to be changed or removed (hide implementation detail)
- **Fundamental tradeoff**:
  - Reusing code saves efforts and (usually) avoids bugs, …
  - *but* makes it harder to change later (lots of people depend on its details)

☢ Repeated recursion ☢

How much work does this function do?

```
let rec bad_max (xs : int list) =
  if xs = [] then
    0 (* bad style, will fix later *)
  else if List.tl xs = [] then
    List.hd xs
  else if List.hd xs > bad_max (List.tl xs) then
    List.hd xs
  else
   bad_max (List.tl xs)
```

☢ Repeated recursion ☢

- How much work does this function do?

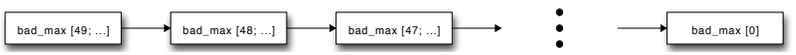
```
bad_max (List.rev (nats_upto 50)) ;;
- : int = 49 (* returns in microseconds *)

bad_max (nats_upto 50) ;;
- : int = ... (* still running *)
```

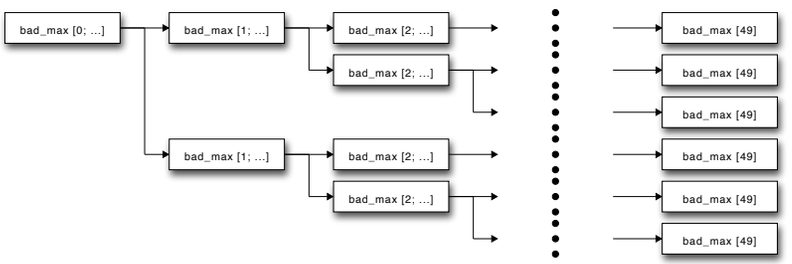
- Why?

☢ Repeated recursion ☢: fast vs. unusable

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



50 **calls**



2^{50} **calls**

Repeated recursion: math and explosion

- Suppose the logic in one `bad_max` call *not counting recursion* takes 1 ms
  - It doesn't matter if this guess is off by 1000×
- Then `bad_max [49; ... ; 0]` takes 50 ms
  - but `bad_max [0; ...; 49]` takes 2^{50} ms, or 35+ years!
    - (Not so) fun fact: `bad_max [0; ...; 51]` takes 4× longer

Using `let` to avoid repeated recursion

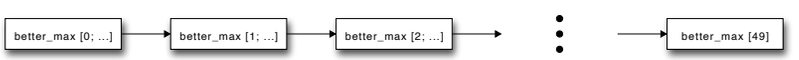
```
 1 let rec better_max (xs : int list) =
 2   if xs = [] then
 3     0 (* bad style *)
 4   else if List.tl xs = [] then
 5     List.hd xs
 6   else
 7     let tl_max = better_max (List.tl xs) in
 8     if List.hd xs > tl_max then
 9       List.hd xs
10     else
11       tl_max
```

- Line 8: Recurse over the list tail *once*
- Store result in the environment bound to `tl_max`

Fast vs. fast

```
let tl_max = better_max (List.tl xs) in
  if List.hd xs > tl_max then
    List.hd xs
  else
    tl_max
```

50 **calls**



50 **calls**

What about partial functions?

- Many computations are not defined or "misbehave" for some inputs:
  - `List.hd []`, `List.tl []`, maximum element of `[]` , `1 / 0`, etc.
- OCaml does provide *exceptions* for such cases
  - We will see and use exceptions more later
  - But exceptions let you forget-about-the-exceptional-thing, and that can lead to surprising and undesirable control flow
- OCaml also provides an option type for handling partial functions
  - also for when your data structure "may or may not have some data"

Demonstration 1: Local bindings

Demonstration 1: Local bindings

```
# let silly1 (z : int) =
  let x = if z > 0 then z else 34 in
  let y = x + 9 in
  if x > y then x * 2 else y * y

let silly2 () = (* poor style, just showing let expressions are expressions *)
  let x = 1 in
  (let x = 1 in x + 1) + (let y = x + 2 in y + 1)

let s = silly2 () ;;
```

Demonstration 1: Local bindings

```
# let silly1 (z : int) =
  let x = if z > 0 then z else 34 in
  let y = x + 9 in
  if x > y then x * 2 else y * y

let silly2 () = (* poor style, just showing let expressions are expressions *)
  let x = 1 in
  (let x = 1 in x + 1) + (let y = x + 2 in y + 1)

let s = silly2 () ;;
val silly1 : int -> int = <fun>
val silly2 : unit -> int = <fun>
val s : int = 6
```

Demonstration 1: Local bindings

```
# let silly1 (z : int) =
  let x = if z > 0 then z else 34 in
  let y = x + 9 in
  if x > y then x * 2 else y * y

let silly2 () = (* poor style, just showing let expressions are expressions *)
  let x = 1 in
  (let x = 1 in x + 1) + (let y = x + 2 in y + 1)

let s = silly2 () ;;
val silly1 : int -> int = <fun>
val silly2 : unit -> int = <fun>
val s : int = 6

# let celsius_of_fahrenheit (tempF : float) =
  (tempF -. 32.0) *. 5.0 /. 9.0

(* Example where let expressions probably make code more readable even though
   no computations are repeated (we will have better ways to do this with variant types) *)
let is_first_warmer ((temp1 : float), (is_celsius1 : bool),
                     (temp2 : float), (is_celsius2 : bool)) =
  let temp1C = if is_celsius1 then temp1 else celsius_of_fahrenheit temp1 in
  let temp2C = if is_celsius2 then temp2 else celsius_of_fahrenheit temp2 in
  temp1C > temp2C

let res = is_first_warmer (0.1, true, 5., false) ;;
```

Demonstration 1: Local bindings

```
# let silly1 (z : int) =
  let x = if z > 0 then z else 34 in
  let y = x + 9 in
  if x > y then x * 2 else y * y

let silly2 () = (* poor style, just showing let expressions are expressions *)
  let x = 1 in
  (let x = 1 in x + 1) + (let y = x + 2 in y + 1)

let s = silly2 () ;;
val silly1 : int -> int = <fun>
val silly2 : unit -> int = <fun>
val s : int = 6

# let celsius_of_fahrenheit (tempF : float) =
  (tempF -. 32.0) *. 5.0 /. 9.0

(* Example where let expressions probably make code more readable even though
   no computations are repeated (we will have better ways to do this with variant types) *)
let is_first_warmer ((temp1 : float), (is_celsius1 : bool),
                     (temp2 : float), (is_celsius2 : bool)) =
  let temp1C = if is_celsius1 then temp1 else celsius_of_fahrenheit temp1 in
  let temp2C = if is_celsius2 then temp2 else celsius_of_fahrenheit temp2 in
  temp1C > temp2C

let res = is_first_warmer (0.1, true, 5., false) ;;
val celsius_of_fahrenheit : float -> float = <fun>
val is_first_warmer : float * bool * float * bool -> bool = <fun>
val res : bool = true
```

Demonstration 1: Local bindings

Demonstration 1: Local bindings

```
# (* Example where we avoid repeating a computation, but it would not matter much *)
let rec squaring_sequence ((x : float), (n : int)) =
  if n = 0 then
    []
  else
    let sq = x *. x in
    sq :: squaring_sequence (sq, n - 1) ;;
```

Demonstration 1: Local bindings

```
# (* Example where we avoid repeating a computation, but it would not matter much *)
let rec squaring_sequence ((x : float), (n : int)) =
  if n = 0 then
    []
  else
    let sq = x *. x in
    sq :: squaring_sequence (sq, n - 1) ;;
val squaring_sequence : float * int -> float list = <fun>
```

Demonstration 1: Local bindings

```
# (* Example where we avoid repeating a computation, but it would not matter much *)
let rec squaring_sequence ((x : float), (n : int)) =
  if n = 0 then
    []
  else
    let sq = x *. x in
    sq :: squaring_sequence (sq, n - 1) ;;
val squaring_sequence : float * int -> float list = <fun>

# let rec range ((lo : int), (hi : int)) =
  if lo < hi then lo :: range (lo + 1, hi)
  else []

let nats_upto1 (x : int) =
  range (0, x)

let nats_upto2 (x : int) =
  let rec range ((lo : int), (hi : int)) =
    if lo < hi then lo :: range (lo + 1, hi)
    else []
  in
  range (0, x)

let nats_upto3 (x : int) =
  let rec loop (lo : int) = (* loop is NOT a special word, could call it fred *)
    if lo < x then lo :: loop (lo + 1) else []
  in
  loop 0

let n = nats_upto3 10 ;;
```

Demonstration 1: Local bindings

```
# (* Example where we avoid repeating a computation, but it would not matter much *)
let rec squaring_sequence ((x : float), (n : int)) =
  if n = 0 then
    []
  else
    let sq = x *. x in
    sq :: squaring_sequence (sq, n - 1) ;;
val squaring_sequence : float * int -> float list = <fun>

# let rec range ((lo : int), (hi : int)) =
  if lo < hi then lo :: range (lo + 1, hi)
  else []

let nats_upto1 (x : int) =
  range (0, x)

let nats_upto2 (x : int) =
  let rec range ((lo : int), (hi : int)) =
    if lo < hi then lo :: range (lo + 1, hi)
    else []
  in
  range (0, x)

let nats_upto3 (x : int) =
  let rec loop (lo : int) = (* loop is NOT a special word, could call it fred *)
    if lo < x then lo :: loop (lo + 1) else []
  in
  loop 0

let n = nats_upto3 10 ;;
val range : int * int -> int list = <fun>
val nats_upto1 : int -> int list = <fun>
val nats_upto2 : int -> int list = <fun>
val nats_upto3 : int -> int list = <fun>
val n : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Options

Options

- Options are basically: “lists” of length zero or one, but with a *different* type constructor
- A value of type `t option` can either be:
  - `None`: a valid value of type `t option` for *any* type `t`
    - like `[]` for `t list`
  - `Some v`, where value `v` has type `t`
    - like `[v]` for `t list`

`option` is another type constructor, just like our other friends: `t1 * t2`, `t1 -> t2`, `t list`

Options: building

- **Syntax:**

  `None`

- **Type checking:**

  - `None` has type `'a option`

- **Evaluation:**

  - `None` is a value

> `None` has type `t option` for any `t`

Options: building

- **Syntax**:

  `Some e`

  where `e` is an expression

- **Type checking**:

  - If `e` has type `t`:
    - then: `Some e` has type `t option`
    - else: report error and fail

- **Evaluation**:

  - If `e` evaluates to value `v`:
    - then: `Some e` evaluates to value `Some v`

Options: testing for `None`

- **Syntax**:

  `e = None`

  where `e` is an expression

- **Type checking**:

  - If has `e` has type `t option`:
    - then: `e = None` has type `bool`
    - else: report error and fail

- **Evaluation**:

  - If `e` evaluates to value `None`:
    - then: `e = None` evaluates to value `true`
    - else: `e = None` evaluates to value `false`

> Alternately, use library functions `Option.is_some` or `Option.is_none`

Options: getting the element

- **Syntax**:

  `Option.get e`

  where `e` is an expression

- **Type checking**:
  - If has `e` has type `t option`:
    - then: `Option.get e` has type `t`
    - else: report error and fail
- **Evaluation**:
  - If `e` evaluates to value `Some v`:
    - then: `Option.get e` evaluates to value `v`
    - else: `e` evaluated to `None` so `Option.get e` raises an exception

Much better `max`

```
1 let rec good_max (xs : int list) =
2   if xs = [] then
3     None
4   else
5     let tl_ans = good_max1 (List.tl xs) in
6     if Option.is_some tl_ans && Option.get tl_ans > List.hd xs then
7       tl_ans
8     else
9       Some (List.hd xs)
```

- returns an `int option`, **NOT** an `int`, so can finally give a reasonable answer for *bad* inputs (e.g., `[]`)
- clients are forced to consider data and no-data cases

Demonstration 2: Options

Demonstration 2: Options

```
# let rec bad_max (xs : int list) =
  if xs = [] then
    0 (* bad semantic style *)
  else if List.tl xs = [] then
    List.hd xs
  else if List.hd xs > bad_max (List.tl xs) then
    List.hd xs
  else
    bad_max (List.tl xs) ;;
```

Demonstration 2: Options

```
# let rec bad_max (xs : int list) =
  if xs = [] then
    0 (* bad semantic style *)
  else if List.tl xs = [] then
    List.hd xs
  else if List.hd xs > bad_max (List.tl xs) then
    List.hd xs
  else
    bad_max (List.tl xs) ;;
val bad_max : int list -> int = <fun>
```

Demonstration 2: Options

```
# let rec bad_max (xs : int list) =
  if xs = [] then
    0 (* bad semantic style *)
  else if List.tl xs = [] then
    List.hd xs
  else if List.hd xs > bad_max (List.tl xs) then
    List.hd xs
  else
    bad_max (List.tl xs) ;;
val bad_max : int list -> int = <fun>

# let rec better_max (xs : int list) =
  if xs = [] then
    0 (* bad semantic style *)
  else if List.tl xs = [] then
    List.hd xs
  else
    let tl_max = better_max (List.tl xs) in
    if List.hd xs > tl_max  then
      List.hd xs
    else
      tl_max ;;
```

Demonstration 2: Options

```
# let rec bad_max (xs : int list) =
  if xs = [] then
    0 (* bad semantic style *)
  else if List.tl xs = [] then
    List.hd xs
  else if List.hd xs > bad_max (List.tl xs) then
    List.hd xs
  else
    bad_max (List.tl xs) ;;
val bad_max : int list -> int = <fun>

# let rec better_max (xs : int list) =
  if xs = [] then
    0 (* bad semantic style *)
  else if List.tl xs = [] then
    List.hd xs
  else
    let tl_max = better_max (List.tl xs) in
    if List.hd xs > tl_max  then
      List.hd xs
    else
      tl_max ;;
val better_max : int list -> int = <fun>
```

Demonstration 2: Options

Demonstration 2: Options

```
# let rec good_max1 (xs : int list) =
  if xs = [] then
    None
  else
    let tl_ans = good_max1 (List.tl xs) in
    if Option.is_some tl_ans && Option.get tl_ans > List.hd xs then
      tl_ans
    else
      Some (List.hd xs)

let good_max2 (xs : int list) =
  if xs = [] then
    None
  else
    let rec max_nonempty (xs : int list) =
      if List.tl xs = [] then (* xs better not be [] *)
        List.hd xs
      else
        let tl_ans = max_nonempty (List.tl xs) in
        if List.hd xs > tl_ans then
          List.hd xs
        else
          tl_ans
    in
    Some (max_nonempty xs) ;;
```

Demonstration 2: Options

```
# let rec good_max1 (xs : int list) =
  if xs = [] then
    None
  else
    let tl_ans = good_max1 (List.tl xs) in
    if Option.is_some tl_ans && Option.get tl_ans > List.hd xs then
      tl_ans
    else
      Some (List.hd xs)

let good_max2 (xs : int list) =
  if xs = [] then
    None
  else
    let rec max_nonempty (xs : int list) =
      if List.tl xs = [] then (* xs better not be [] *)
        List.hd xs
      else
        let tl_ans = max_nonempty (List.tl xs) in
        if List.hd xs > tl_ans then
          List.hd xs
        else
          tl_ans
    in
    Some (max_nonempty xs) ;;
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>

# let does_not_typecheck = (good_max1 [3; 4; 2]) + 4 ;;
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>

# let does_not_typecheck = (good_max1 [3; 4; 2]) + 4 ;;
Error: This expression (good_max1 [3; 4; 2]) has type int option
       but an expression was expected of type int
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>

# let does_not_typecheck = (good_max1 [3; 4; 2]) + 4 ;;
Error: This expression (good_max1 [3; 4; 2]) has type int option
       but an expression was expected of type int

# let risky_but_works = (Option.get (good_max1 [3; 4; 2])) + 2
let risky_and_fails = (Option.get (good_max1 [])) + 2 ;;
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>

# let does_not_typecheck = (good_max1 [3; 4; 2]) + 4 ;;
Error: This expression (good_max1 [3; 4; 2]) has type int option
       but an expression was expected of type int

# let risky_but_works = (Option.get (good_max1 [3; 4; 2])) + 2
let risky_and_fails = (Option.get (good_max1 [])) + 2 ;;
val risky_but_works : int = 6
Exception: Invalid_argument "option is None".
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>

# let does_not_typecheck = (good_max1 [3; 4; 2]) + 4 ;;
Error: This expression (good_max1 [3; 4; 2]) has type int option
       but an expression was expected of type int

# let risky_but_works = (Option.get (good_max1 [3; 4; 2])) + 2
let risky_and_fails = (Option.get (good_max1 [])) + 2 ;;
val risky_but_works : int = 6
Exception: Invalid_argument "option is None".

# let propagate_option =
  let x = good_max1 [3; 4; 2] in
  if Option.is_some x then Some ((Option.get x) + 2) else None ;;
```

Demonstration 2: Options

```
val good_max1 : int list -> int option = <fun>
val good_max2 : int list -> int option = <fun>

# let does_not_typecheck = (good_max1 [3; 4; 2]) + 4 ;;
Error: This expression (good_max1 [3; 4; 2]) has type int option
       but an expression was expected of type int

# let risky_but_works = (Option.get (good_max1 [3; 4; 2])) + 2
let risky_and_fails = (Option.get (good_max1 [])) + 2 ;;
val risky_but_works : int = 6
Exception: Invalid_argument "option is None".

# let propagate_option =
  let x = good_max1 [3; 4; 2] in
  if Option.is_some x then Some ((Option.get x) + 2) else None ;;
val propagate_option : int option = Some 6
```

Aliasing and Mutation

Can you spot the difference? Can a client?

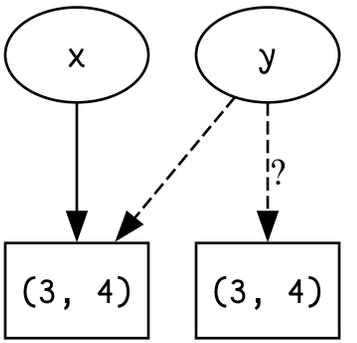
```
let sort_pair (pr : int * int) =
  if fst pr < snd pr then
    pr
  else
    (snd pr, fst pr)
```

```
let sort_pair (pr : int * int) =
  if fst pr < snd pr then
    (fst pr, snd pr)
  else
    (snd pr, fst pr)
```

- If pair already sorted, return (an alias to) the same pair, or a new pair with the same contents?
- In OCaml, *we cannot tell* which way `sort_pair` is implemented!
  - This **negative expressiveness** is a big deal
  - When data is immutable, aliasing *does not matter*

Why lack of mutation matters

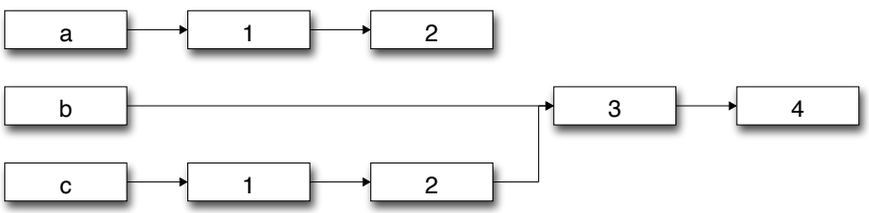
```
let x = (3, 4)
let y = sort_pair x
(* somehow mutate fst x to hold 5 *)
let z = fst y
```

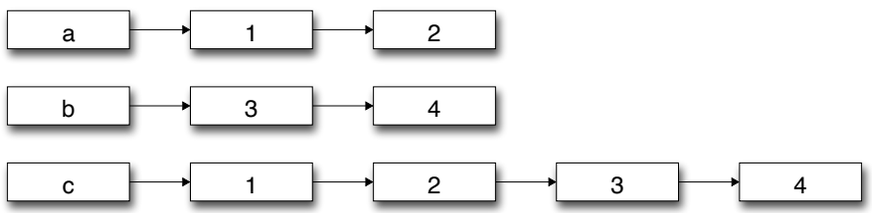


- What is `z`?
  - Would depend on how we implemented `sort_pair`
- **With mutation**: Would have to decide carefully and document `sort_pair`
- **Without mutation**: we can implement *either way*:
  - No code can ever distinguish aliasing vs. identical copies
  - No need to think about aliasing: focus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

Aliasing with `append`

```
let rec append ((xs : 'a list), (ys : 'a list)) =
  if xs = []
  then ys
  else List.hd xs :: append (List.tl xs, ys)
let a = [1; 2]
let b = [3; 4]
let c = append (a, b)
```





- Clients can never tell! (but it's actually the left one)
- Can *safely* reuse and share data. Immutability can be *more efficient*!

Immutability is great!

- In OCaml, we create aliases all the time without thinking about it because it is *impossible* to tell if there is aliasing
  - Example: `List.tl` is constant time; doesn't copy anything
  - No need to worry!
- In languages with mostly mutable data (e.g., C++, Python, Java), programmers are *obsessed* with aliasing and object identity
  - Remember `std::move` from C++? Or passing by `const T&`?
  - But… they have to be so that subsequent field assignments affect the right parts of the program
  - Often crucial to make copies in just the right places
    - Ever had to use `copy`, `copy.deepcopy` etc.?

Python security nightmare: bad code

Consider the following Python example:

```
class ProtectedResource:
  def __init__(self):
    self._allowedUsers = ...
  def allowedUsers(self):
    return self._allowedUsers
  def currentUser(self): ...
  def useTheResource(self):
    for u in self._allowedUsers:
      if self.currentUser() == u:
        # access allowed: use it ...
        return
    raise IllegalAccessError("...")
```

Python security nightmare: need to make copies

- The problem (**representation exposure**):

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

- The fix:

```
def allowedUsers(self):
  # return a copy of allowedUsers
  return copy(self._allowedUsers)
```

However, copying immutable data is **unnecessary**!

= versus ==

Equality in OCaml

- OCaml has two different equality operators: `=` and `==`
  - `e1 <> e2` is the same as `not (e1 = e2)`
  - `e1 != e2` is the same as `not (e1 == e2)`
- **In this course, always use `=` (or `<>`)**
  - ⚠ **Not** the character sequences (`==`) you are used to in Python or Java
  - It turns out not to matter for integers and booleans though
- The difference between them is interesting and relevant to aliasing…

Structural equality

- `=` is defined recursively: same data in same shape
  - `3 = 3`, `(1, 2) = (1, 2)`, `[3; 4; 5] = [3; 4; 5]`, `None = None`, `Some 7 = Some 7`, etc.
- Two subexpressions must have same type (it's hard to be equal otherwise…)
- **Note**:
  - If `x` and `y` are aliases, then `x = y` must evaluate to `true`
  - If `x` and `y` are not aliases, then `x = y` might be `true` or `false`
  - Aliasing here doesn't matter
- More like Java `equals`, but predefined by the language

Reference equality

Demonstration 3: Immutability

Demonstration 3: Immutability

```
 # (* Does not matter if returns an alias *)
let sort_pair (pr : int * int) : int * int =
  if fst pr < snd pr then
    pr (* (fst pr, snd pr) *)
  else
    (snd pr, fst pr)

(* Naturally and efficiently introduces sharing and we don't care *)
let rec append ((xs : 'a list), (ys : 'a list)) =
  if xs = [] then
    ys
  else
    List.hd xs :: append (List.tl xs, ys)

append ([1; 2; 3], [3; 4; 5]) ;;
```

Demonstration 3: Immutability

```
# (* Does not matter if returns an alias *)
let sort_pair (pr : int * int) : int * int =
  if fst pr < snd pr then
    pr (* (fst pr, snd pr) *)
  else
    (snd pr, fst pr)

(* Naturally and efficiently introduces sharing and we don't care *)
let rec append ((xs : 'a list), (ys : 'a list)) =
  if xs = [] then
    ys
  else
    List.hd xs :: append (List.tl xs, ys)

append ([1; 2; 3], [3; 4; 5]) ;;
val sort_pair : int * int -> int * int = <fun>
val append : 'a list * 'a list -> 'a list = <fun>
- : int list = [1; 2; 3; 3; 4; 5]
```