CSC 330 / Spring 2026

OCaml
Matching, Patterns

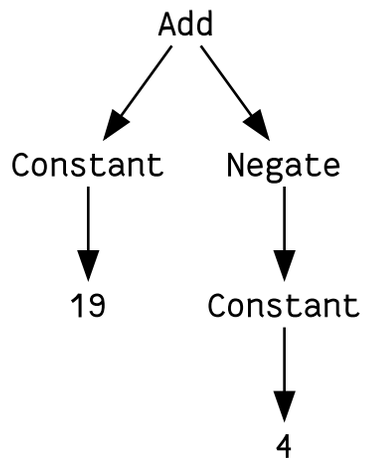
Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Recursive variant types

- Concise, elegant way to define different tree types
  - Nodes are tagged with a constructor with children for carried data

```
type expr =
| Constant of int
| Negate of expr
| Add of expr * expr
| Mul of expr * expr

Add (Constant 19, Negate (Constant 4))
```



- Functions over `expr` will typically use recursion over the tree structure

Defining our own option types

- Using what we already know:

```
type int_option = NoInt | OneInt of int
```

- Using new feature of defining our own *type constructor*:

```
type 'a myoption = MyNone | MySome of 'a
```

- *Exactly* how "built-in" options are defined (in standard library):

```
type 'a option = None | Some of 'a
```

So use pattern-matching!

- `None` and `Some` are (just) constructors for `'a option`
- *Build* with constructors and *use* with match expressions
  - Do not use `Option.get` and `Option.is_some` anymore
  - Get the advantages of variant types and pattern matching

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Variants can be recursive, describing recursive data structures like trees *)
  type expr =
    | Constant of int
    | Negate of expr
    | Add of expr * expr
    | Mul of expr * expr

  let rec eval e =
    match e with
    | Constant i -> i
    | Negate e1 -> -eval e1
    | Add (e1, e2) -> eval e1 + eval e2
    | Mul (e1, e2) -> eval e1 * eval e2

  let rec max_const (e : expr) : int =
    let max (x, y) = if x > y then x else y in
    match e with
    | Constant i -> i
    | Negate e1 -> max_const e1
    | Add (e1, e2) -> max (max_const e1, max_const e2)
    | Mul (e1, e2) -> max (max_const e1, max_const e2)

  let rec has_const_not_under_add e =
    match e with
    | Constant i -> true
    | Negate e1 -> has_const_not_under_add e1
    | Add (e1, e2) -> false
    | Mul (e1, e2) -> has_const_not_under_add e1 || has_const_not_under_add e2
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Variants can be recursive, describing recursive data structures like trees *)
  type expr =
    | Constant of int
    | Negate of expr
    | Add of expr * expr
    | Mul of expr * expr

  let rec eval e =
    match e with
    | Constant i -> i
    | Negate e1 -> -eval e1
    | Add (e1, e2) -> eval e1 + eval e2
    | Mul (e1, e2) -> eval e1 * eval e2

  let rec max_const (e : expr) : int =
    let max (x, y) = if x > y then x else y in
    match e with
    | Constant i -> i
    | Negate e1 -> max_const e1
    | Add (e1, e2) -> max (max_const e1, max_const e2)
    | Mul (e1, e2) -> max (max_const e1, max_const e2)

  let rec has_const_not_under_add e =
    match e with
    | Constant i -> true
    | Negate e1 -> has_const_not_under_add e1
    | Add (e1, e2) -> false
    | Mul (e1, e2) -> has_const_not_under_add e1 || has_const_not_under_add e2

  type expr =
      Constant of int
    | Negate of expr
    | Add of expr * expr
    | Mul of expr * expr
  val eval : expr -> int = <fun>
  val max_const : expr -> int = <fun>
  val has_const_not_under_add : expr -> bool = <fun>
  val number_of_adds : expr -> int = <fun>
  val example_exp : expr = Add (Constant 19, Negate (Constant 4))
  val example_ans : int = 15
  val example_addcount : int = 2
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Same features already used can almost define option *)
  type int_option = NoInt | OneInt of int

  let rec sum_int_options1 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | NoInt -> sum_int_options1 (List.tl xs)
      | OneInt i -> i + sum_int_options1 (List.tl xs)

  let test1 = sum_int_options1 [ NoInt; OneInt 7; NoInt; OneInt 2; OneInt 1 ] ;;
```

Demonstration 1: Matching

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Same features already used can almost define option *)
  type int_option = NoInt | OneInt of int

  let rec sum_int_options1 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | NoInt -> sum_int_options1 (List.tl xs)
      | OneInt i -> i + sum_int_options1 (List.tl xs)

  let test1 = sum_int_options1 [ NoInt; OneInt 7; NoInt; OneInt 2; OneInt 1 ] ;;
  type int_option = NoInt | OneInt of int
  val sum_int_options1 : int_option list -> int = <fun>
  val test1 : int = 10
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Same features already used can almost define option *)
  type int_option = NoInt | OneInt of int

  let rec sum_int_options1 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | NoInt -> sum_int_options1 (List.tl xs)
      | OneInt i -> i + sum_int_options1 (List.tl xs)

  let test1 = sum_int_options1 [ NoInt; OneInt 7; NoInt; OneInt 2; OneInt 1 ] ;;
type int_option = NoInt | OneInt of int
val sum_int_options1 : int_option list -> int = <fun>
val test1 : int = 10
# (* In fact, we /can/ define our own polymorphic variant types *)
  type 'a my_option = MyNone | MySome of 'a

  let rec sum_int_options2 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | MyNone -> sum_int_options2 (List.tl xs)
      | MySome i -> i + sum_int_options2 (List.tl xs)

  let test2 = sum_int_options2 [ MyNone; MySome 7; MyNone; MySome 2; MySome 1 ] ;;
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Same features already used can almost define option *)
  type int_option = NoInt | OneInt of int

  let rec sum_int_options1 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | NoInt -> sum_int_options1 (List.tl xs)
      | OneInt i -> i + sum_int_options1 (List.tl xs)

  let test1 = sum_int_options1 [ NoInt; OneInt 7; NoInt; OneInt 2; OneInt 1 ] ;;
type int_option = NoInt | OneInt of int
val sum_int_options1 : int_option list -> int = <fun>
val test1 : int = 10
# (* In fact, we /can/ define our own polymorphic variant types *)
  type 'a my_option = MyNone | MySome of 'a

  let rec sum_int_options2 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | MyNone -> sum_int_options2 (List.tl xs)
      | MySome i -> i + sum_int_options2 (List.tl xs)

  let test2 = sum_int_options2 [ MyNone; MySome 7; MyNone; MySome 2; MySome 1 ] ;;
type 'a my_option = MyNone | MySome of 'a
val sum_int_options2 : int my_option list -> int = <fun>
val test2 : int = 10
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Indeed, the option type constructor is not "built in" at all; just in standard library *)
  (* type 'a option = None | Some of 'a *)
  (* from now on, use pattern-matching for options *not* the previous way we showed to use them *)
  let rec sum_int_options3 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | None -> sum_int_options3 (List.tl xs)
      | Some i -> i + sum_int_options3 (List.tl xs)

  let test3 = sum_int_options3 [ None; Some 7; None; Some 2; Some 1 ] ;;
```

Demonstration 1: Matching

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Indeed, the option type constructor is not "built in" at all; just in standard library *)
  (* type 'a option = None | Some of 'a *)
  (* from now on, use pattern-matching for options *not* the previous way we showed to use them *)
  let rec sum_int_options3 xs =
    if xs = [] then 0
    else
      match List.hd xs with
      | None -> sum_int_options3 (List.tl xs)
      | Some i -> i + sum_int_options3 (List.tl xs)

  let test3 = sum_int_options3 [ None; Some 7; None; Some 2; Some 1 ] ;;

  val sum_int_options3 : int option list -> int = <fun>
  val test3 : int = 10
```

How about lists?

- We can use the same features and self-reference for our own list type constructor:

```
type 'a my_list = Empty | Cons of 'a * 'a my_list
```

- *Build* with constructors:

```
let xs = Cons (3, Cons (4, Cons (5, Empty)))
let ss = Cons ("hi", Cons ("bye", Empty))
```

- *Use* with match expressions:

```
let rec length xs =
  match xs with
  | Empty -> 0
  | Cons (x, xs') -> 1 + length xs'
```

OCaml's built-in lists

- Could have done *exactly* what we just did…
- but instead it is using special *syntax*:
  - Constructors are `[]` and `::` with `::` written **infix**
  - and/but use match expressions to use lists
    - **do not** use `List.hd`, `List.tl` and `e = []` anymore!

```
let rec length xs =
  match xs with
  | [] -> 0
  | x :: xs' -> 1 + length xs'
```

Why then `List.hd` and `List.tl`?

- Occasionally convenient
  - e.g., for passing functions to other functions (next week)
- A stepping stone for us to the more elegant and less error-prone approach (match expressions warn on missing cases!)
- Again, do not use these functions on remaining assignments
- And note how easy they are to define:

```
let hd xs =
  match xs with
  | [] -> failwith "list empty"
  | x :: _ -> x
  (* | x :: xs' -> x
     is also OK, but rather unstylish;
     compiler might even bark *)
```

```
let isEmpty xs =
  match xs with
  | [] -> true
  | _ -> false
  (* | x :: xs' -> false
     is also OK, but again not co
```

Demonstration 2: Polymorphic lists

Demonstration 2: Polymorphic lists

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type 'a my_list = Empty | Cons of 'a * 'a my_list

  let rec sum_int_options4 xs =
    match xs with
    | Empty -> 0
    | Cons (x, xs') ->
        (match x with
        | None -> sum_int_options4 xs'
        | Some i -> i + sum_int_options4 xs')
  let test4 = sum_int_options4
      (Cons (None, Cons (Some 7, Cons (None, Cons (Some 2, Cons (Some 1, Empty)))))) ;;
```

Demonstration 2: Polymorphic lists

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type 'a my_list = Empty | Cons of 'a * 'a my_list

  let rec sum_int_options4 xs =
    match xs with
    | Empty -> 0
    | Cons (x, xs') ->
        (match x with
        | None -> sum_int_options4 xs'
        | Some i -> i + sum_int_options4 xs')
  let test4 = sum_int_options4
      (Cons (None, Cons (Some 7, Cons (None, Cons (Some 2, Cons (Some 1, Empty)))))) ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10
```

Demonstration 2: Polymorphic lists

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type 'a my_list = Empty | Cons of 'a * 'a my_list

  let rec sum_int_options4 xs =
    match xs with
    | Empty -> 0
    | Cons (x, xs') ->
        (match x with
        | None -> sum_int_options4 xs'
        | Some i -> i + sum_int_options4 xs')
  let test4 = sum_int_options4
      (Cons (None, Cons (Some 7, Cons (None, Cons (Some 2, Cons (Some 1, Empty)))))) ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10

# (* This is exactly how built-in lists are defined /except/ special syntax [] and :: !
     So yes, we can pattern-match with those constructors and should no longer use = [],
     List.hd, or List.tl (!!) *)
  let rec sum_int_options5 xs =
    match xs with
    | [] -> 0
    | x :: xs' -> (  (* parentheses are optional *)
        match x with
        | None -> sum_int_options5 xs'
        | Some i -> i + sum_int_options5 xs')
  let test5 = sum_int_options5 [ None; Some 7; None; Some 2; Some 1 ] ;;
```

Demonstration 2: Polymorphic lists

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
# type 'a my_list = Empty | Cons of 'a * 'a my_list

  let rec sum_int_options4 xs =
    match xs with
    | Empty -> 0
    | Cons (x, xs') ->
        (match x with
        | None -> sum_int_options4 xs'
        | Some i -> i + sum_int_options4 xs')
  let test4 = sum_int_options4
      (Cons (None, Cons (Some 7, Cons (None, Cons (Some 2, Cons (Some 1, Empty)))))) ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10

# (* This is exactly how built-in lists are defined /except/ special syntax [] and :: !
     So yes, we can pattern-match with those constructors and should no longer use = [],
     List.hd, or List.tl (!!) *)
  let rec sum_int_options5 xs =
    match xs with
    | [] -> 0
    | x :: xs' -> (  (* parentheses are optional *)
        match x with
        | None -> sum_int_options5 xs'
        | Some i -> i + sum_int_options5 xs')
  let test5 = sum_int_options5 [ None; Some 7; None; Some 2; Some 1 ] ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10
  val sum_int_options5 : int option list -> int = <fun>
  val test5 : int = 10
```

Demonstration 2: Polymorphic lists

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type 'a my_list = Empty | Cons of 'a * 'a my_list

  let rec sum_int_options4 xs =
    match xs with
    | Empty -> 0
    | Cons (x, xs') ->
        (match x with
        | None -> sum_int_options4 xs'
        | Some i -> i + sum_int_options4 xs')
  let test4 = sum_int_options4
      (Cons (None, Cons (Some 7, Cons (None, Cons (Some 2, Cons (Some 1, Empty))))));;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10

# (* This is exactly how built-in lists are defined /except/ special syntax [] and :: !
     So yes, we can pattern-match with those constructors and should no longer use = [],
     List.hd, or List.tl (!!) *)
  let rec sum_int_options5 xs =
    match xs with
    | [] -> 0
    | x :: xs' -> (  (* parentheses are optional *)
        match x with
        | None -> sum_int_options5 xs'
        | Some i -> i + sum_int_options5 xs')
  let test5 = sum_int_options5 [ None; Some 7; None; Some 2; Some 1 ] ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10
  val sum_int_options5 : int option list -> int = <fun>
  val test5 : int = 10

# (* Spoiler alert: nested patterns can make this even more concise
     we aren't /quite/ there yet, but this is the style we expect on hw2 *)
  let rec sum_int_options6 xs =
    match xs with
    | [] -> 0
    | None :: xs' -> sum_int_options6 xs'
    | Some i :: xs' -> i + sum_int_options6 xs'
  let test6 = sum_int_options6 [ None; Some 7; None; Some 2; Some 1 ] ;;
```

Demonstration 2: Polymorphic lists

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type 'a my_list = Empty | Cons of 'a * 'a my_list

  let rec sum_int_options4 xs =
    match xs with
    | Empty -> 0
    | Cons (x, xs') ->
        (match x with
        | None -> sum_int_options4 xs'
        | Some i -> i + sum_int_options4 xs')
  let test4 = sum_int_options4
      (Cons (None, Cons (Some 7, Cons (None, Cons (Some 2, Cons (Some 1, Empty)))))) ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10

# (* This is exactly how built-in lists are defined /except/ special syntax [] and :: !
     So yes, we can pattern-match with those constructors and should no longer use = [],
     List.hd, or List.tl (!!) *)
  let rec sum_int_options5 xs =
    match xs with
    | [] -> 0
    | x :: xs' -> (  (* parentheses are optional *)
        match x with
        | None -> sum_int_options5 xs'
        | Some i -> i + sum_int_options5 xs')
  let test5 = sum_int_options5 [ None; Some 7; None; Some 2; Some 1 ] ;;

  type 'a my_list = Empty | Cons of 'a * 'a my_list
  val sum_int_options4 : int option my_list -> int = <fun>
  val test4 : int = 10
  val sum_int_options5 : int option list -> int = <fun>
  val test5 : int = 10

# (* Spoiler alert: nested patterns can make this even more concise
     we aren't /quite/ there yet, but this is the style we expect on hw2 *)
  let rec sum_int_options6 xs =
    match xs with
    | [] -> 0
    | None :: xs' -> sum_int_options6 xs'
    | Some i :: xs' -> i + sum_int_options6 xs'
  let test6 = sum_int_options6 [ None; Some 7; None; Some 2; Some 1 ] ;;

  val sum_int_options6 : int option list -> int = <fun>
  val test6 : int = 10
```

Patterns

How languages define variant types

- OCaml built powerful pattern-matching into the language and used it as *the fundamental primitive* for one-of types
- Other languages make other choices. For example OCaml *could* have:
  - kept variant type definitions as-is
  - had them automatically introduce *functions* like `isEmpty`, `hd` etc.

More pattern-matching

- OCaml also has pattern-matching for each-of types
- Pattern `(x, y, z)` matches triples `(v1, v2, v3)`
  - Similar for any width tuple, naturally
  - Similar patterns `{ f1 = x1; ...; fN = xN }` for records
- Useful for **nested patterns** (next lecture)
- But also useful to make let expressions even *more powerful*

Bad style

- One-branch match expressions make semantic sense but are ugly!
- **Don't do this:**

```
let sum_triple tr =
  match tr with
  | (x, y, z) -> x + y + z
```

Let expressions and patterns

- Let expression syntax is actually

```
let p = e1 in e2
```

  where `p` is a *pattern*!

  - Top-level let bindings can also do this
  - Great for extracting multiple pieces of data at once
  - No more need for `fst`, `snd`, `fst3`, `snd3`, `thd3`, and so on

```
let sum_triple tr =
  let (x, y, z) = tr in
  x + y + z
```

Another new feature

- Function arguments can also use patterns!
- **Function call semantics**: pattern-match the argument, extracting data for body

```
let sum_triple tr =
  let (x, y, z) = tr in
  x + y + z
let sum_triple (x, y, z) =
  x + y + z
let ten = sum_triple (5, 2, 3)
```

Behold! 😊

Compare:

- A function that takes a triple and sums its int components:

```
let sum_triple (x, y, z) =
x + y + z
```

- A function that takes three ints and sums them:

```
let sum_triple (x, y, z) =
x + y + z
```

- See the difference? (me neither…) 😊 🎉 😊 🎉 😊

The truth about functions

- In OCaml, every function takes exactly one argument
- We can *simulate* multiple arguments with tuples and tuple-patterns
  - Elegant, compositional design
  - Enables sometimes-convenient things
    - Return a tuple from one function and pass it like multiple arguments to another
- Soon we will see a different and more common way to simulate multiple arguments that *also* has built-in syntactic support

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

Demonstration 3: Patterns

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Pattern-matching is the normal ML way to use lists; let's revisit prior functions *)
  let rec length xs =
    match xs with
    | [] -> 0
    | x :: xs' -> 1 + length xs'

  let rec append (xs, ys) =
    match xs with [] -> ys | x :: xs' -> x :: append (xs', ys)

  let rec concat ss =
    match ss with
    | [] -> ""
    | s :: ss' -> s ^ concat ss' ;;
```

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Pattern-matching is the normal ML way to use lists; let's revisit prior functions *)
  let rec length xs =
    match xs with
    | [] -> 0
    | x :: xs' -> 1 + length xs'

  let rec append (xs, ys) =
    match xs with [] -> ys | x :: xs' -> x :: append (xs', ys)

  let rec concat ss =
    match ss with
    | [] -> ""
    | s :: ss' -> s ^ concat ss' ;;

  val length : 'a list -> int = <fun>
  val append : 'a list * 'a list -> 'a list = <fun>
  val concat : string list -> string = <fun>
```

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Pattern-matching is the normal ML way to use lists; let's revisit prior functions *)
  let rec length xs =
    match xs with
    | [] -> 0
    | x :: xs' -> 1 + length xs'

  let rec append (xs, ys) =
    match xs with [] -> ys | x :: xs' -> x :: append (xs', ys)

  let rec concat ss =
    match ss with
    | [] -> ""
    | s :: ss' -> s ^ concat ss' ;;

  val length : 'a list -> int = <fun>
  val append : 'a list * 'a list -> 'a list = <fun>
  val concat : string list -> string = <fun>

# (* Pattern-matching for each-of types (tuples shown; records can also be pattern-matched)*)

  (* 🤔 Terrible style never used: one-arm match expressions *)
  let sum_triple1 tr = match tr with x, y, z -> x + y + z

  (* 😀 Appropriate style: let expression syntax is /actually/: let p = e1 in e2 *)
  let sum_triple2 tr =
    let x, y, z = tr in
    x + y + z

  (* 😎 Even better when useful: can put a pattern right in the function binding: let rec f p = e *)
  let sum_triple3 (x, y, z) = x + y + z
  (* in fact, thanks to a convenient fib, that's what we have done since Lecture 2! *) ;;
```

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* Pattern-matching is the normal ML way to use lists; let's revisit prior functions *)
  let rec length xs =
    match xs with
    | [] -> 0
    | x :: xs' -> 1 + length xs'

  let rec append (xs, ys) =
    match xs with [] -> ys | x :: xs' -> x :: append (xs', ys)

  let rec concat ss =
    match ss with
    | [] -> ""
    | s :: ss' -> s ^ concat ss' ;;

  val length : 'a list -> int = <fun>
  val append : 'a list * 'a list -> 'a list = <fun>
  val concat : string list -> string = <fun>

# (* Pattern-matching for each-of types (tuples shown; records can also be pattern-matched)*)

  (* 🤔 Terrible style never used: one-arm match expressions *)
  let sum_triple1 tr = match tr with x, y, z -> x + y + z

  (* 😀 Appropriate style: let expression syntax is /actually/: let p = e1 in e2 *)
  let sum_triple2 tr =
    let x, y, z = tr in
    x + y + z

  (* 😎 Even better when useful: can put a pattern right in the function binding: let rec f p = e *)
  let sum_triple3 (x, y, z) = x + y + z
  (* in fact, thanks to a convenient fib, that's what we have done since Lecture 2! *) ;;

  val length : 'a list -> int = <fun>
  val append : 'a list * 'a list -> 'a list = <fun>
  val concat : string list -> string = <fun>
  val sum_triple1 : int * int * int -> int = <fun>
  val sum_triple2 : int * int * int -> int = <fun>
  val sum_triple3 : int * int * int -> int = <fun>
```

Demonstration 3: Patterns

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* And one more nested-patterns spoiler: *)
  let rec sum_pairs xs =
    match xs with
    | [] -> 0
    | (x, y) :: xs' -> x + y + sum_pairs xs'

  (* Cute example of expressiveness of functions actually taking one tuple *)
  let rotate_left (x, y, z) = (y, z, x)
  let rotate_right tr = rotate_left (rotate_left tr)

  (* Just as never use one-branch match expressions with each-of patterns,
     it is also usually bad style to use let expressions with one-of patterns:
     you will get a warning at compile-time plus a run-time exception if match fails *)
  let get_risky1 opt =
    match opt with
    | None -> failwith "nopes"
    | Some v -> v
  let get_risky2 opt = let (Some v) = opt in v ;;
```

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* And one more nested-patterns spoiler: *)
  let rec sum_pairs xs =
    match xs with
    | [] -> 0
    | (x, y) :: xs' -> x + y + sum_pairs xs'

  (* Cute example of expressiveness of functions actually taking one tuple *)
  let rotate_left (x, y, z) = (y, z, x)
  let rotate_right tr = rotate_left (rotate_left tr)

  (* Just as never use one-branch match expressions with each-of patterns,
     it is also usually bad style to use let expressions with one-of patterns:
     you will get a warning at compile-time plus a run-time exception if match fails *)
  let get_risky1 opt =
    match opt with
    | None -> failwith "nopes"
    | Some v -> v
  let get_risky2 opt = let (Some v) = opt in v ;;
Lines 20-21, characters 2-3:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
  Here is an example of a case that is not matched: None

val sum_pairs : (int * int) list -> int = <fun>
val rotate_left : 'a * 'b * 'c -> 'b * 'c * 'a = <fun>
val rotate_right : 'a * 'b * 'c -> 'c * 'a * 'b = <fun>
val get_risky1 : 'a option -> 'a = <fun>
val get_risky2 : 'a option -> 'a = <fun>
```

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* And one more nested-patterns spoiler: *)
  let rec sum_pairs xs =
    match xs with
    | [] -> 0
    | (x, y) :: xs' -> x + y + sum_pairs xs'

  (* Cute example of expressiveness of functions actually taking one tuple *)
  let rotate_left (x, y, z) = (y, z, x)
  let rotate_right tr = rotate_left (rotate_left tr)

  (* Just as never use one-branch match expressions with each-of patterns,
     it is also usually bad style to use let expressions with one-of patterns:
     you will get a warning at compile-time plus a run-time exception if match fails *)
  let get_risky1 opt =
    match opt with
    | None -> failwith "nopes"
    | Some v -> v
  let get_risky2 opt = let (Some v) = opt in v ;;
Lines 20-21, characters 2-3:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
  Here is an example of a case that is not matched: None

val sum_pairs : (int * int) list -> int = <fun>
val rotate_left : 'a * 'b * 'c -> 'b * 'c * 'a = <fun>
val rotate_right : 'a * 'b * 'c -> 'c * 'a * 'b = <fun>
val get_risky1 : 'a option -> 'a = <fun>
val get_risky2 : 'a option -> 'a = <fun>

# let get_risky3 (Some v) = v
  get_risky1 None ;;
```

Demonstration 3: Patterns

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# (* And one more nested-patterns spoiler: *)
  let rec sum_pairs xs =
    match xs with
    | [] -> 0
    | (x, y) :: xs' -> x + y + sum_pairs xs'

  (* Cute example of expressiveness of functions actually taking one tuple *)
  let rotate_left (x, y, z) = (y, z, x)
  let rotate_right tr = rotate_left (rotate_left tr)

  (* Just as never use one-branch match expressions with each-of patterns,
     it is also usually bad style to use let expressions with one-of patterns:
     you will get a warning at compile-time plus a run-time exception if match fails *)
  let get_risky1 opt =
    match opt with
    | None -> failwith "nopes"
    | Some v -> v
  let get_risky2 opt = let (Some v) = opt in v ;;

  Lines 20-21, characters 2-3:
  Warning 8 [partial-match]: this pattern-matching is not exhaustive.
    Here is an example of a case that is not matched: None

  val sum_pairs : (int * int) list -> int = <fun>
  val rotate_left : 'a * 'b * 'c -> 'b * 'c * 'a = <fun>
  val rotate_right : 'a * 'b * 'c -> 'c * 'a * 'b = <fun>
  val get_risky1 : 'a option -> 'a = <fun>
  val get_risky2 : 'a option -> 'a = <fun>

# let get_risky3 (Some v) = v
  get_risky1 None ;;

  Warning 8 [partial-match]: this pattern-matching is not exhaustive.
    Here is an example of a case that is not matched: None
  val get_risky3 : (('a option -> 'a) -> 'b option -> 'c) option -> 'c = <fun>
```

Demonstration 3: Patterns

Demonstration 3: Patterns

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type t = { a: int; b: string }
  let v = { a = 5; b = "6" } ;;

  let { a; b } = v in a + int_of_string b ;;
```

Demonstration 3: Patterns

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type t = { a: int; b: string }
  let v = { a = 5; b = "6" } ;;

  let { a; b } = v in a + int_of_string b ;;
  type t = { a : int; b : string; }
  val v : t = {a = 5; b = "6"}
  - : int = 11
```

Demonstration 3: Patterns

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type t = { a: int; b: string }
  let v = { a = 5; b = "6" } ;;

  let { a; b } = v in a + int_of_string b ;;
  type t = { a : int; b : string; }
  val v : t = {a = 5; b = "6"}
  - : int = 11

# match v with
  | { a = a'; b = b' } -> a' + 1
  | { a; _ } -> a - 1 ;;
```

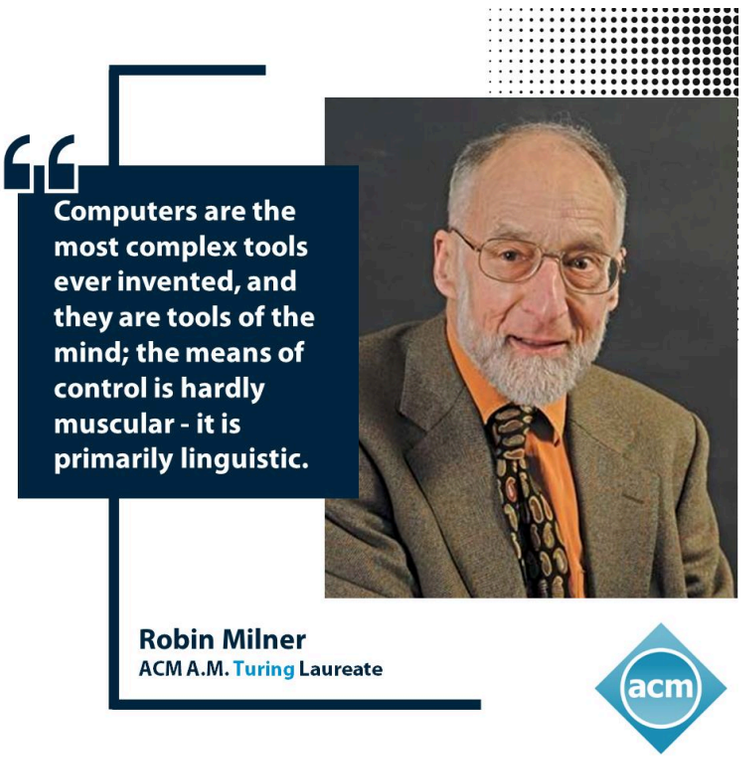
Demonstration 3: Patterns

```
                Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

# type t = { a: int; b: string }
  let v = { a = 5; b = "6" } ;;

  let { a; b } = v in a + int_of_string b ;;
  type t = { a : int; b : string; }
  val v : t = {a = 5; b = "6"}
  - : int = 11

# match v with
  | { a = a'; b = b' } -> a' + 1
  | { a; _ } -> a - 1 ;;
  Line 3, characters 2-10:
  Warning 11 [redundant-case]: this match case is unused.
```

Some history …

ML

- Typechecking: **Robin Milner** et al. at the Edinburgh Laboratory for Computer Science in Scotland
  - Based on theorem proving
  - Milner developed ML: *Meta Language* whose goal is to construct valid proofs
  - Famous typechecking (type inference) algorithm: Hindley–Milner–Damas
    - Similar to garbage collection
  - Won a Turing Award in 1991



ML: Example

```
let fact n =
    letref count=n and result=1
    in     if count=0
           then result
           loop count,result := count-1,count*result;;
# fact = - : (int -> int)

fact 4;;
# 24 : int
```

On to OCaml…

- In the early ’80s, there was a schism in the ML community:
  - 🇫🇷 vs 🇬🇧+🇺🇸
    - 🇫🇷 at Inria (Gérard Huet, Xavier Leroy, Jérôme Vouillon et al.) developed **CAML** (*Categorical Abstract Machine Language*) and later **OCaml** (*Objective CAML*; 1996)
    - 🇬🇧+🇺🇸 developed **Standard ML**
    - Both are very similar
  - Microsoft introduced a variant of OCaml suited for .NET called **F#** (2005)

Finance & economics | A large hump

Jane Street's sneaky retention tactic

It involves the use of an obscure, French programming language

ML

- Common features of ML languages:
  - Similar syntax
  - Static type system with parametric polymorphism (about that later!)
  - Functional programming
  - Modularity
  - General purpose
  - Emphasis on theorem proving and static analysis (Coq/Rocq theorem prover)