CSC 330 / Spring 2026

OCaml
Patterns, Tail Recursion, Exceptions

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Nested patterns

Nested patterns

- We can nest patterns inside other patterns
  - Just like we can nest expressions as deep as we want
  - Anywhere a variable can appear in our current patterns, we can put a pattern
  - Often avoids hard-to-read nested match expressions
- So full meaning of pattern-matching is to compare a pattern against a value for the *same shape* and bind variables to the *right parts*
  - More precise recursive definition coming after some examples

Demonstration: Nested patterns

Demonstration: Nested patterns

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

# (* 🤔 Two ways NOT to do zip3: *)
  let rec meh_zip3_v1 (xs, ys, zs) =
    if xs = [] && ys = [] && zs = [] then
      []
    else if xs = [] || ys = [] || zs = [] then
      failwith "zip3 length mismatch"
    else
      (List.hd xs, List.hd ys, List.hd zs) ::
        meh_zip3_v1 (List.tl xs, List.tl ys, List.tl zs)

  let rec meh_zip3_v2 (xs, ys, zs) =
    match xs with (* like life without && and || *)
    | [] -> (
        match ys with
        | [] -> (
          match zs with
          | [] -> []
          | _ -> failwith "zip3 length mismatch")
        | _ -> failwith "zip3 length mismatch")
    | x :: xs' -> (
        match ys with
        | [] -> failwith "zip3 length mismatch"
        | y :: ys' -> (
            match zs with
            | [] -> failwith "zip3 length mismatch"
            | z :: zs' -> (x, y, z) :: meh_zip3_v2 (xs', ys', zs'))) ;;
```

Demonstration: Nested patterns

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

```
# (* 🤔 Two ways NOT to do zip3: *)
  let rec meh_zip3_v1 (xs, ys, zs) =
    if xs = [] && ys = [] && zs = [] then
      []
    else if xs = [] || ys = [] || zs = [] then
      failwith "zip3 length mismatch"
    else
      (List.hd xs, List.hd ys, List.hd zs) ::
        meh_zip3_v1 (List.tl xs, List.tl ys, List.tl zs)

  let rec meh_zip3_v2 (xs, ys, zs) =
    match xs with (* like life without && and || *)
    | [] -> (
        match ys with
        | [] -> (
          match zs with
          | [] -> []
          | _ -> failwith "zip3 length mismatch")
        | _ -> failwith "zip3 length mismatch")
    | x :: xs' -> (
        match ys with
        | [] -> failwith "zip3 length mismatch"
        | y :: ys' -> (
            match zs with
            | [] -> failwith "zip3 length mismatch"
            | z :: zs' -> (x, y, z) :: meh_zip3_v2 (xs', ys', zs'))) ;;

val meh_zip3_v1 : 'a list * 'b list * 'c list -> ('a * 'b * 'c) list = <fun>
val meh_zip3_v2 : 'a list * 'b list * 'c list -> ('a * 'b * 'c) list = <fun>
```

Demonstration: Nested patterns

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

Demonstration: Nested patterns

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

# (* Instead... nested patterns give you "and" by matching the entire pattern,
     along with nested data extraction *)
  let rec zip3 list_triple =
    match list_triple with
    | [], [], [] -> []
    | x :: xs', y :: ys', z :: zs' -> (x, y, z) :: zip3 (xs', ys', zs')
    | _ -> failwith "zip3 length mismatch"

  (* the inverse function is also elegant *)
  let rec unzip3 xyzs =
    match xyzs with
    | [] -> [], [], []
    | (x, y, z) :: xyzs' ->
        let xs, ys, zs = unzip3 xyzs' in
        (x :: xs, y :: ys, z :: zs) ;;
```

Demonstration: Nested patterns

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

# (* Instead... nested patterns give you "and" by matching the entire pattern,
     along with nested data extraction *)
  let rec zip3 list_triple =
    match list_triple with
    | [], [], [] -> []
    | x :: xs', y :: ys', z :: zs' -> (x, y, z) :: zip3 (xs', ys', zs')
    | _ -> failwith "zip3 length mismatch"

  (* the inverse function is also elegant *)
  let rec unzip3 xyzs =
    match xyzs with
    | [] -> [], [], []
    | (x, y, z) :: xyzs' ->
        let xs, ys, zs = unzip3 xyzs' in
        (x :: xs, y :: ys, z :: zs) ;;

  val zip3 : 'a list * 'b list * 'c list -> ('a * 'b * 'c) list = <fun>
  val unzip3 : ('a * 'b * 'c) list -> 'a list * 'b list * 'c list = <fun>
```

Nested patterns

- Nested patterns are used in different contexts to express algorithms concisely and elegantly
  - Match on multiple things with same shape at same time
  - Go multiple levels into data structure at once
  - Use wildcard patterns when you do not need the data
  - Make match expressions that look like tables

Nested match expressions?

- Sometimes a match expression inside a match expression is a missed opportunity for nested pattern matching

```
match xs with
| [] -> 0
| x :: xs' -> match xs' with ...
```

- Other times it is necessary because you need to first compute with data extracted via the outer match

```
match xs with
| [] -> 0
| x :: xs' -> match f x s' with ...
```

- ⚠️: sometimes you need to wrap inner `match` statements with parentheses; otherwise you can end up with spectacularly confusing errors!

Patterns: generalization

- **Syntax:**

```
match e with p1 -> e1 | ... | pN -> eN
let p1 = e in e1
let [rec] f p1 = e1
```

  where `e, e1, ..., eN` are expressions, and `p1, ..., pN` are patterns

- **Pattern syntax:** A pattern is one of:
  - `x`: a variable
  - `_`: a wildcard
  - `(p1, ..., pN)`: a tuple of patterns
  - `C p`: a constructor `C` applied to a pattern `p`

> - Patterns are **NOT** expressions (though they kinda look like expressions)!

Match expressions: evaluation

Will now focus on match expressions, others are similar

- We need two things:
  1. Semantics of matching a pattern `p` to a value `v`
     - Does it, or does it not match?
     - If it matches, creates a set of (local) bindings
  2. Use (1) to define match expressions' semantics

Pattern-matching revealed

Yet another elegant recursive definition:

- Given pattern `p` and value `v`:
  - if `p` is a variable `x`, the match succeeds and `x` is bound to `v`;
  - if `p` is `_`, the match succeeds and no bindings are introduced;
  - if `p` is `(p1, ..., pN)` and `v` is `(v1, ..., vN)`, the match succeeds if and only if `p1` matches `v1` , and …, and `pN` matches `vN` (the bindings are the union of all bindings from the submatches);
  - if `p` is `C p1`, the match succeeds if and only if `v` is `C v1` (i.e., the same constructor) and `p1` matches `v1` (the bindings are the bindings from the submatch)

Examples

- `a :: b :: c :: d` matches all lists with $\geq$ 3 elements
- `a :: b :: c :: []` matches all lists with exactly 3 elements
- `((a, b), (c, d)) :: e` matches all non-empty lists of pairs of pairs

Match expressions: semantics

- **Evaluation:**

```
match e with p1 -> e1 | ... | pN -> eN
```

1. Evaluate `e` to `v`
2. Check in `p1`, `p2`, … order; find first `pi` that matches
3. Evaluate `ei` in environment extended with bindings from the match
4. Result of (3) is the overall result

Note: This is the *semantics*, but the implementation can *optimize* with concepts like binary search

Match expressions: type checking

- **Type checking**:

```
match e with p1 -> e1 | ... | pN -> eN
```

  - Typecheck `e` to some type `t`
  - Each `pi` must match against some value of type `t`
  - Similar pattern, type matching definition to get the types of bindings
    - Typecheck `ei` in a static environment that has those bindings
  - All `e1, ..., eN` must have the same type `t2`, which is the overall type
  - Also, some fantastic features:
    - Error if a `pi` can never match because of earlier patterns (dead code)
    - Warning if a value of type `t` could match no `pi` (incomplete match)
      - (With nested patterns, these require some fairly fancy algorithms)

Demonstration: Nested patterns

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

Demonstration: Nested patterns

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

# (* A couple more examples with lists *)
  let rec is_sorted xs =
    match xs with
    | [] -> true
    | x :: [] -> true
    | head :: neck :: rest -> head <= neck && is_sorted (neck :: rest)

  let rec cumulative_sum xs =
    match xs with
    | [] -> xs
    | x :: [] -> xs
    | head :: neck :: rest -> head :: cumulative_sum ((head + neck) :: rest)

  type sign = P | N | Z

  let multsign (x1, x2) =
    let sign_of_num x =
      if x = 0 then Z else if x > 0 then P else N
    in
    match (sign_of_num x1, sign_of_num x2) with
    | Z, _ | _, Z -> Z
    | P, P | N, N -> P
    | _ -> N (* questionable style; we are okay with it *)

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

  let rec sum_pair_list xs =
    match xs with [] -> 0 | (x, y) :: xs' -> x + y + sum pair list xs' ;;
```

Demonstration: Nested patterns

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

# (* A couple more examples with lists *)
  let rec is_sorted xs =
    match xs with
    | [] -> true
    | x :: [] -> true
    | head :: neck :: rest -> head <= neck && is_sorted (neck :: rest)

  let rec cumulative_sum xs =
    match xs with
    | [] -> xs
    | x :: [] -> xs
    | head :: neck :: rest -> head :: cumulative_sum ((head + neck) :: rest)

  type sign = P | N | Z

  let multsign (x1, x2) =
    let sign_of_num x =
      if x = 0 then Z else if x > 0 then P else N
    in
    match (sign_of_num x1, sign_of_num x2) with
    | Z, _ | _, Z -> Z
    | P, P | N, N -> P
    | _ -> N (* questionable style; we are okay with it *)

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

  let rec sum_pair_list xs =
    match xs with [] -> 0 | (x, y) :: xs' -> x + y + sum pair list xs' ;;
val is_sorted : 'a list -> bool = <fun>
val cumulative_sum : int list -> int list = <fun>
type sign = P | N | Z
val multsign : int * int -> sign = <fun>
val length : 'a list -> int = <fun>
val sum_pair_list : (int * int) list -> int = <fun>
```

Tail Recursion

Recursion

- Now have lots of practice with recursion
  - *Code follows the data*: recursive data processed recursively!
- No need for loops and assignment statements. Recursion often easier:
  - e.g., processing trees, appending lists
- But every recursive call needs to get "fresh space" for arguments and local variables, separate from the caller's "space"
- **Next**: let's refine our mental model of how calls execute (call stacks)
  - And see how recursion can be *as efficient as a loop*!

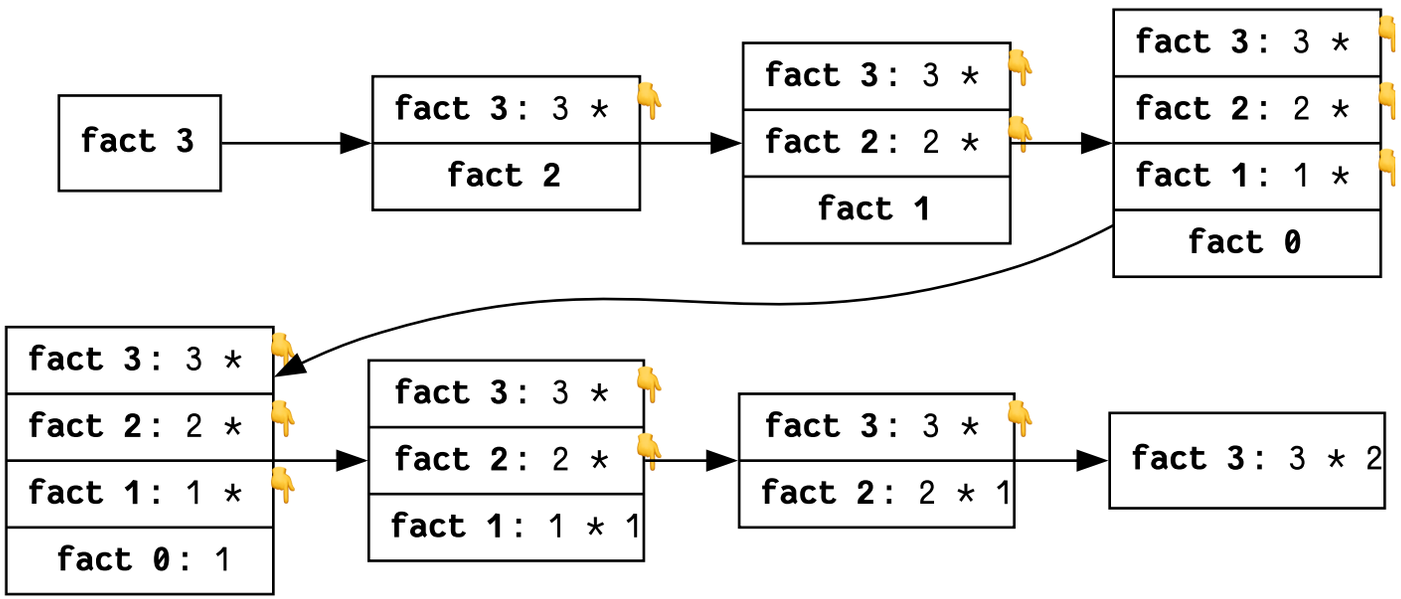
Call stacks

- While a program runs, there is a **call stack** for all function calls that have started but not yet completed
  - Calling `f` pushes a stack frame for `f` on the stack
  - When a call finishes, its stack frame is popped from the stack
  - Recursion: Can have more than one stack frames be for calls to the same function
- A stack frame stores information: values of parameters, values of local variables, and "what's left to do" for the function body

> - Function calls finish in the reverse order that they start!

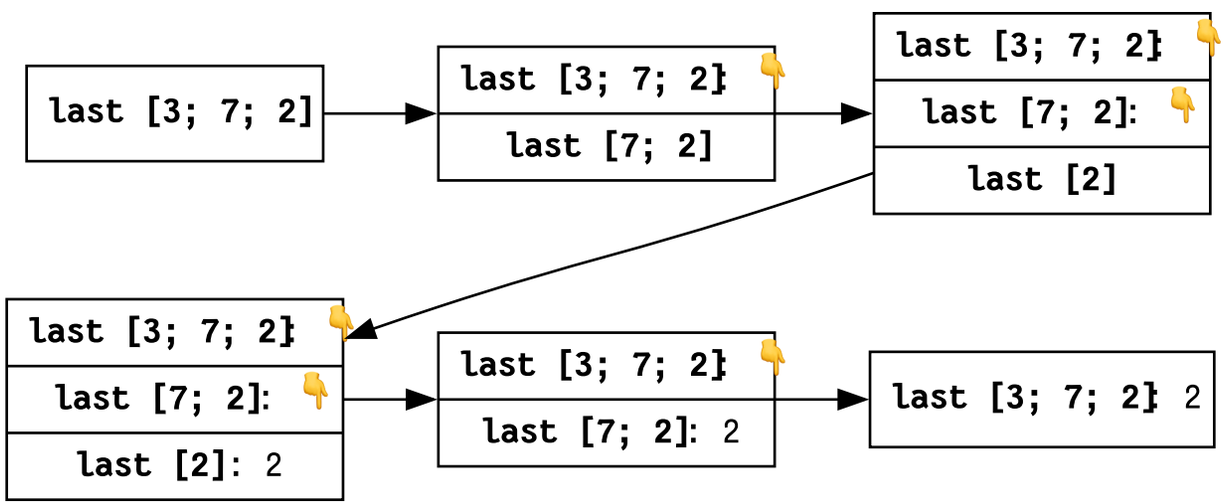
Example

```
let rec fact n =
  if n = 0 then 1 else n * fact(n - 1)
let x = fact 3
```



Example

```
let rec last xs =
  match xs with
  | x :: [] -> x
  | _ :: xs' -> last xs'
let x = last [3; 7; 2]
```



Do we *always* need a stack frame?

- **No!** It is unnecessary to keep around a stack-frame just so it can get a result from a function call and immediately return it "as is"
  - A call that will be the caller's answer *as is* is called a **tail call**
  - A recursive function where all recursive calls are tail calls is called **tail recursive**
- OCaml handles tail calls specially: **tail-call optimization**
  - Pops caller's frame **before** the call
  - This can take $O(n)$ space usage down to $O(1)$ 🤯
  - No *stack overflow* issues anymore!
  - Along with other optimizations, as efficient as a loop

`last` is tail-recursive!

```
let rec last xs =
  match xs with
  | x :: [] -> x
  | _ :: xs' -> last xs'
let x = last [3; 7; 2]
```

Making functions tail-recursive

- *Sometimes* we can rewrite our algorithm to be tail-recursive:

```
let fact n =
  let rec loop (n, acc) =
    if n = 0 then acc else loop (n - 1, acc * n)
  in
  loop(n, 1)
let x = fact 3
```

- *Sometimes* we can't easily or profitably do so:
  - processing a tree (two recursive calls can't both be tail calls)
  - if order matters (e.g., concatenating a string list)

Methodology

- There is a *methodology* to guide the transformation to a tail-recursive function:
  - Create a helper function that takes an **accumulator**
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator
- Up to you to reason about whether this works
  - Example: for factorial, it is crucial that *multiplication is associative*

Demonstration: Tail recursion

Demonstration: Tail recursion

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

# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)

  let e = fact 5

  let fact_tc n =
    let rec loop (n, acc) =
      if n = 0 then acc else loop (n - 1, acc * n)
    in
    loop (n, 1)

  let e = fact_tc 5 ;;
```

Demonstration: Tail recursion

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

# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)

  let e = fact 5

  let fact_tc n =
    let rec loop (n, acc) =
      if n = 0 then acc else loop (n - 1, acc * n)
    in
    loop (n, 1)

  let e = fact_tc 5 ;;
val fact : int -> int = <fun>
val fact_tc : int -> int = <fun>
val e : int = 120
```

Demonstration: Tail recursion

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

# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)

  let e = fact 5

  let fact_tc n =
    let rec loop (n, acc) =
      if n = 0 then acc else loop (n - 1, acc * n)
    in
    loop (n, 1)

  let e = fact_tc 5 ;;

  val fact : int -> int = <fun>
  val fact_tc : int -> int = <fun>
  val e : int = 120

# let rec last xs =
    match xs with
    | [] -> failwith "last: empty list"
    | x :: [] -> x
    | _ :: xs' -> (last [@tailcall]) xs'
    (* you can check for tailcall with @tailcall here *)

  let e = last [1; 2; 3] ;;
```

Demonstration: Tail recursion

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

# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)

  let e = fact 5

  let fact_tc n =
    let rec loop (n, acc) =
      if n = 0 then acc else loop (n - 1, acc * n)
    in
    loop (n, 1)

  let e = fact_tc 5 ;;

  val fact : int -> int = <fun>
  val fact_tc : int -> int = <fun>
  val e : int = 120

# let rec last xs =
    match xs with
    | [] -> failwith "last: empty list"
    | x :: [] -> x
    | _ :: xs' -> (last [@tailcall]) xs'
    (* you can check for tailcall with @tailcall here *)

  let e = last [1; 2; 3] ;;

  val last : 'a list -> 'a = <fun>
  val e : int = 3
```

Demonstration: Tail recursion

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

# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)

  let e = fact 5

  let fact_tc n =
    let rec loop (n, acc) =
      if n = 0 then acc else loop (n - 1, acc * n)
    in
    loop (n, 1)

  let e = fact_tc 5 ;;

  val fact : int -> int = <fun>
  val fact_tc : int -> int = <fun>
  val e : int = 120

# let rec last xs =
    match xs with
    | [] -> failwith "last: empty list"
    | x :: [] -> x
    | _ :: xs' -> (last [@tailcall]) xs'
    (* you can check for tailcall with @tailcall here *)

  let e = last [1; 2; 3] ;;

  val last : 'a list -> 'a = <fun>
  val e : int = 3

# let rec fact n = if n = 0 then 1 else n * (fact [@tailcall]) (n - 1) in
  fact 3 ;;
```

Demonstration: Tail recursion

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

# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)

  let e = fact 5

  let fact_tc n =
    let rec loop (n, acc) =
      if n = 0 then acc else loop (n - 1, acc * n)
    in
    loop (n, 1)

  let e = fact_tc 5 ;;

  val fact : int -> int = <fun>
  val fact_tc : int -> int = <fun>
  val e : int = 120

# let rec last xs =
    match xs with
    | [] -> failwith "last: empty list"
    | x :: [] -> x
    | _ :: xs' -> (last [@tailcall]) xs'
    (* you can check for tailcall with @tailcall here *)

  let e = last [1; 2; 3] ;;

  val last : 'a list -> 'a = <fun>
  val e : int = 3

# let rec fact n = if n = 0 then 1 else n * (fact [@tailcall]) (n - 1) in
  fact 3 ;;

  Line 2, characters 27-53:
  Warning 51 [wrong-tailcall-expectation]: expected tailcall

  val fact : int -> int = <fun>
  val x : int = 6
```

Demonstration: Tail recursion

Demonstration: Tail recursion

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

# let rec sum xs =
    match xs with [] -> 0 | i :: xs' -> i + sum xs'

  let sum_tc xs =
    let rec f (xs, acc) =
      match xs with [] -> acc | i :: xs' -> f (xs', i + acc)
    in
    f (xs, 0)

  let rec rev xs =
    match xs with [] -> [] | x :: xs' -> (rev xs') @ [ x ]

  let rev_tc xs =
    let rec loop (xs, acc) =
      match xs with [] -> acc | x :: xs' -> loop (xs', x :: acc)
    in
    loop (xs, []) ;;
```

Demonstration: Tail recursion

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

# let rec sum xs =
    match xs with [] -> 0 | i :: xs' -> i + sum xs'

  let sum_tc xs =
    let rec f (xs, acc) =
      match xs with [] -> acc | i :: xs' -> f (xs', i + acc)
    in
    f (xs, 0)

  let rec rev xs =
    match xs with [] -> [] | x :: xs' -> (rev xs') @ [ x ]

  let rev_tc xs =
    let rec loop (xs, acc) =
      match xs with [] -> acc | x :: xs' -> loop (xs', x :: acc)
    in
    loop (xs, []) ;;
val sum : int list -> int = <fun>
val sum_tc : int list -> int = <fun>
val rev : 'a list -> 'a list = <fun>
val rev_tc : 'a list -> 'a list = <fun>
```

Examples

- **Summing a list**: $O(1)$ stack space vs. length-of-list stack space
  - But notice both versions of factorial and list-sum take *time* linear in length of list
- **Reversing a list**: Tail-recursive version actually *much better*
  - Non-tail recursive version is *quadratic time* $O(n^2)$:
    - each recursive call uses `append`, which must traverse the list;
    - so total time for the appends is $1 + 2 + \cdots + (n + 1)$ which is roughly $\frac{n^2}{2}$
  - Moral: beware appending to lists, especially within outer recursion
  - Consing is constant time and fast (why?), so tail-recursive version is linear time

Moral: don't overdo it!

- Tail-recursion is not always feasible or elegant
- Tail-recursion is not always necessary: beware premature optimization and favor readability
- But where it is feasible and efficiency matters, it is a key tool for functional programmers to get the efficiency of loops
  - If you can write it as a loop, you can write it as a tail-recursive function: *updated values* are arguments to the tail call!
- Most functional languages, including OCaml, promise tail-call optimization

Precise definition of tail calls

- How do you know if a call is a tail call?
  - Humans: “Oh, I see the caller has no more work to do after it gets the result.”
  - Semanticists and compilers: a tail call is a function call that *is in tail position*
    - which is defined recursively over the syntax of expressions…

Tail position

- In `let f p = e`, the expression `e` is in tail position
- If `if e1 then e2 else e3` is in tail position:
  - then: `e2` and `e3` are in tail position;
    - but `e1` is **not** in tail position!
- If `let p = e1 in e2` is in tail position:
  - then: `e2` is in tail position
    - but `e1` is **not** in tail position
- If `e1 + e2`, the subexpressions are **not** in tail position
- … (more cases for every kind of expression)
- If `e` is **not** in tail position, then neither is any of its subexpressions

Exceptions

Exceptions

- Exception bindings declare new kinds of exception:

```
exception Whatever
exception BadNum of int
```

- `raise` expressions can throw an exception:

```
raise Whatever
raise (BadNum 1)
```

- `try ... with ...` expressions can catch exceptions:

```
try e with Whatever -> 0
try e with BadNum n -> n
```

Exceptions

- Exception bindings declare new kinds of exception:

```
exception Whatever
exception BadNum of int
```

- We *build* exceptions, which are just *values* (!) with these constructors:

```
Whatever
(BadNum 1)
```

- That is different than *raising* an exception with the `raise` expression:

```
raise Whatever
raise (BadNum 1)
```

A bit more precisely

- New exception bindings add new variants to "the one exception type" `exn`
- We build values of type `exn` just like we build variant types with constructors
  - Can pass them around, though that isn't that common
- We raise (throw) exceptions with `raise e`
  - **Type checking**:
    - `e` must have type `exn`
    - Result type is *any type you want (!)*
  - **Evaluation**:
    - Do not produce a result (!); but…
    - *cancel everything* and *pass the exception* produced by `e` to the nearest *try expression* on the call stack
      - and keep doing this while you can!

Try expressions

- `try ... with ...` expressions can *catch* exceptions:

```
try e1 with Whatever -> e2
try e1 with BadNum n -> e2
```

- **Evaluation**:
  - Just evaluate `e1` and that is the result
  - But if `e1` raises an exception *and* that exception *matches* the pattern, then evaluate `e2` and that is the result
- **Type checking**:
  - `e1` and `e2` must have the same type and that is the overall type

Demonstration: Exceptions

Demonstration: Exceptions

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

# exception MyUndesirableCondition
  exception MyOtherException of int * int

  let oh_no () =
    raise MyUndesirableCondition
  let oh_no_with_info () =
    raise (MyOtherException (7, 42))
  let catch_example () =
    try oh_no () with MyUndesirableCondition -> 0

  let catch_example_with_info () =
    try oh_no_with_info () with MyOtherException (x, y) -> x + y

  let boo () =
    try oh_no () with MyOtherException (x, y) -> x + y
  let hd xs =
    match xs with [] -> raise MyUndesirableCondition | x :: _ -> x

  let foo1 = hd [ 3; 4; 5 ]
  let foo2 = hd [] (* exception 💣 *) ;;
```

Demonstration: Exceptions

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

# exception MyUndesirableCondition
  exception MyOtherException of int * int

  let oh_no () =
    raise MyUndesirableCondition
  let oh_no_with_info () =
    raise (MyOtherException (7, 42))
  let catch_example () =
    try oh_no () with MyUndesirableCondition -> 0

  let catch_example_with_info () =
    try oh_no_with_info () with MyOtherException (x, y) -> x + y

  let boo () =
    try oh_no () with MyOtherException (x, y) -> x + y
  let hd xs =
    match xs with [] -> raise MyUndesirableCondition | x :: _ -> x

  let foo1 = hd [ 3; 4; 5 ]
  let foo2 = hd [] (* exception 💣 *) ;;
exception MyUndesirableCondition
exception MyOtherException of int * int
val oh_no : unit -> 'a = <fun>
val oh_no_with_info : unit -> 'a = <fun>
val catch_example : unit -> int = <fun>
val catch_example_with_info : unit -> int = <fun>
val boo : unit -> int = <fun>
val hd : 'a list -> 'a = <fun>
val foo1 : int = 3
Exception: MyUndesirableCondition.
```

Demonstration: Exceptions

Demonstration: Exceptions

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

# let bar1 =
    try Some (hd [ 3; 4; 5 ]) with MyUndesirableCondition -> None
  let bar2 =
    try Some (hd []) with MyUndesirableCondition -> None

  let rec maxlist (xs, ex) = (* int list * exn -> int *)
    match xs with
    | [] -> raise ex
    | x :: [] -> x
    | x :: xs' ->
        let m = maxlist (xs', ex) in
        if x > m then x else m

  let m1 = maxlist ([ 3; 4; 5 ], MyUndesirableCondition)

  let m2 =
    try maxlist ([ 3; 4; 5 ], MyUndesirableCondition)
    with MyUndesirableCondition -> 42 ;;
```

Demonstration: Exceptions

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

# let bar1 =
    try Some (hd [ 3; 4; 5 ]) with MyUndesirableCondition -> None
  let bar2 =
    try Some (hd []) with MyUndesirableCondition -> None

  let rec maxlist (xs, ex) = (* int list * exn -> int *)
    match xs with
    | [] -> raise ex
    | x :: [] -> x
    | x :: xs' ->
        let m = maxlist (xs', ex) in
        if x > m then x else m

  let m1 = maxlist ([ 3; 4; 5 ], MyUndesirableCondition)

  let m2 =
    try maxlist ([ 3; 4; 5 ], MyUndesirableCondition)
    with MyUndesirableCondition -> 42 ;;
  val bar1 : int option = Some 3
  val bar2 : 'a option = None
  val maxlist : 'a list * exn -> 'a = <fun>
  val m1 : int = 5
  val m2 : int = 5
```

Demonstration: Exceptions

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

# let bar1 =
    try Some (hd [ 3; 4; 5 ]) with MyUndesirableCondition -> None
  let bar2 =
    try Some (hd []) with MyUndesirableCondition -> None

  let rec maxlist (xs, ex) = (* int list * exn -> int *)
    match xs with
    | [] -> raise ex
    | x :: [] -> x
    | x :: xs' ->
        let m = maxlist (xs', ex) in
        if x > m then x else m

  let m1 = maxlist ([ 3; 4; 5 ], MyUndesirableCondition)

  let m2 =
    try maxlist ([ 3; 4; 5 ], MyUndesirableCondition)
    with MyUndesirableCondition -> 42 ;;
  val bar1 : int option = Some 3
  val bar2 : 'a option = None
  val maxlist : 'a list * exn -> 'a = <fun>
  val m1 : int = 5
  val m2 : int = 5
# let m3 = maxlist ([], MyUndesirableCondition) ;;
```

Demonstration: Exceptions

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

# let bar1 =
    try Some (hd [ 3; 4; 5 ]) with MyUndesirableCondition -> None
  let bar2 =
    try Some (hd []) with MyUndesirableCondition -> None

  let rec maxlist (xs, ex) = (* int list * exn -> int *)
    match xs with
    | [] -> raise ex
    | x :: [] -> x
    | x :: xs' ->
        let m = maxlist (xs', ex) in
        if x > m then x else m

  let m1 = maxlist ([ 3; 4; 5 ], MyUndesirableCondition)

  let m2 =
    try maxlist ([ 3; 4; 5 ], MyUndesirableCondition)
    with MyUndesirableCondition -> 42 ;;
  val bar1 : int option = Some 3
  val bar2 : 'a option = None
  val maxlist : 'a list * exn -> 'a = <fun>
  val m1 : int = 5
  val m2 : int = 5
# let m3 = maxlist ([], MyUndesirableCondition) ;;
  Exception: MyUndesirableCondition.
```

Demonstration: Exceptions

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

# let bar1 =
    try Some (hd [ 3; 4; 5 ]) with MyUndesirableCondition -> None
  let bar2 =
    try Some (hd []) with MyUndesirableCondition -> None

  let rec maxlist (xs, ex) = (* int list * exn -> int *)
    match xs with
    | [] -> raise ex
    | x :: [] -> x
    | x :: xs' ->
        let m = maxlist (xs', ex) in
        if x > m then x else m

  let m1 = maxlist ([ 3; 4; 5 ], MyUndesirableCondition)

  let m2 =
    try maxlist ([ 3; 4; 5 ], MyUndesirableCondition)
    with MyUndesirableCondition -> 42 ;;
  val bar1 : int option = Some 3
  val bar2 : 'a option = None
  val maxlist : 'a list * exn -> 'a = <fun>
  val m1 : int = 5
  val m2 : int = 5
# let m3 = maxlist ([], MyUndesirableCondition) ;;
  Exception: MyUndesirableCondition.
# let m4 =
    try maxlist ([], MyUndesirableCondition) with MyUndesirableCondition -> 42 ;;
```

Demonstration: Exceptions

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

# let bar1 =
    try Some (hd [ 3; 4; 5 ]) with MyUndesirableCondition -> None
  let bar2 =
    try Some (hd []) with MyUndesirableCondition -> None

  let rec maxlist (xs, ex) = (* int list * exn -> int *)
    match xs with
    | [] -> raise ex
    | x :: [] -> x
    | x :: xs' ->
        let m = maxlist (xs', ex) in
        if x > m then x else m

  let m1 = maxlist ([ 3; 4; 5 ], MyUndesirableCondition)

  let m2 =
    try maxlist ([ 3; 4; 5 ], MyUndesirableCondition)
    with MyUndesirableCondition -> 42 ;;
  val bar1 : int option = Some 3
  val bar2 : 'a option = None
  val maxlist : 'a list * exn -> 'a = <fun>
  val m1 : int = 5
  val m2 : int = 5
# let m3 = maxlist ([], MyUndesirableCondition) ;;
  Exception: MyUndesirableCondition.
# let m4 =
    try maxlist ([], MyUndesirableCondition) with MyUndesirableCondition -> 42 ;;
  val m4 : int = 42
```