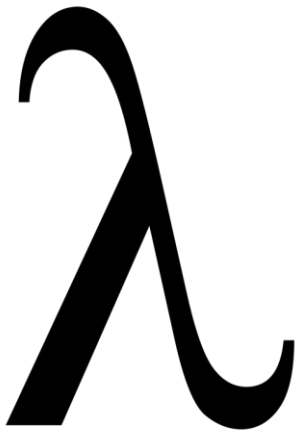


Patterns and local environment

Programmazione Funzionale

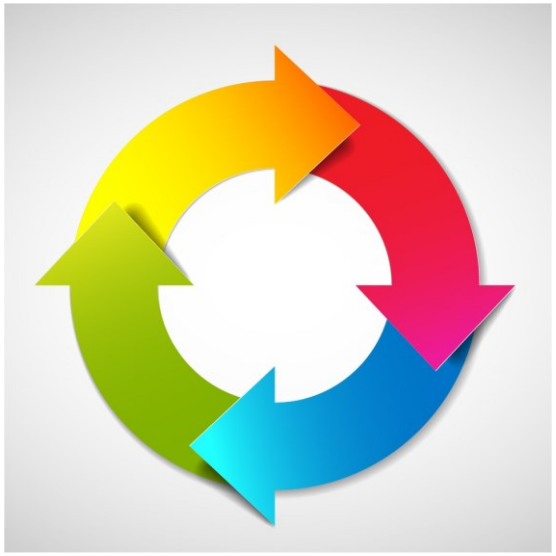
2024/2025

Università di Trento

Chiara Di Francescomarino

Today

- Recursion in ML
- Mutual recursion (in ML)
- Patterns in ML
- Cases and patterns in ML
- Local environment in ML



Recursion in ML

Reversing a list

- Example: `reverse([1,2,3])` is `[3,2,1]`
  - Base case: empty list to empty list
  - Induction: reverse the tail of the list (recursively) and then append the head

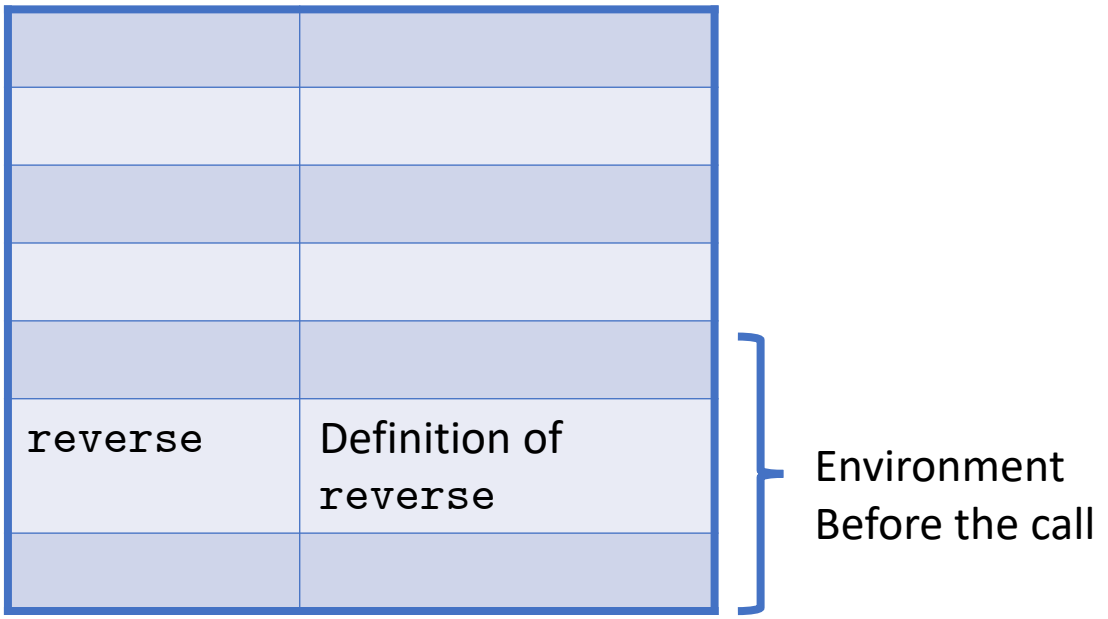
```
> fun reverse L =
    if L = nil then nil
    else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3,4];
val it = [4, 3, 2, 1]: int list
> reverse["ab","bc","cd"];
val it = ["cd", "bc", "ab"]: string list
```

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list
```

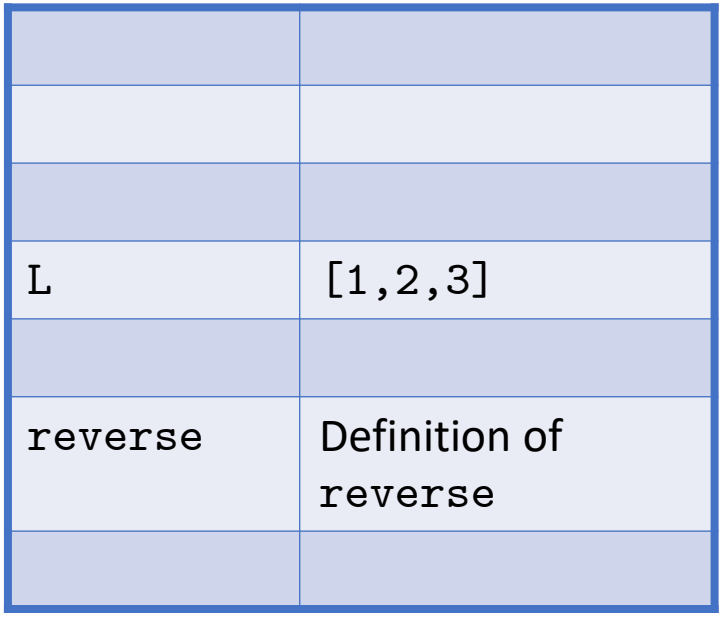


How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```

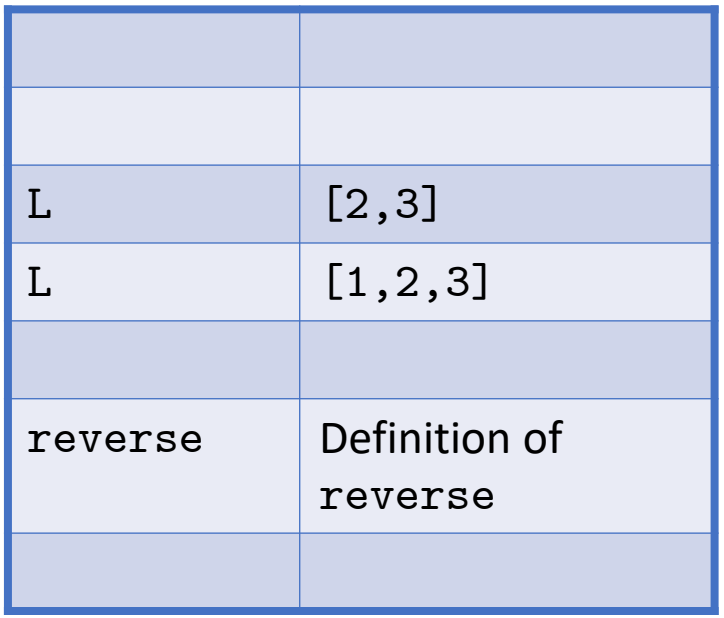


How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```

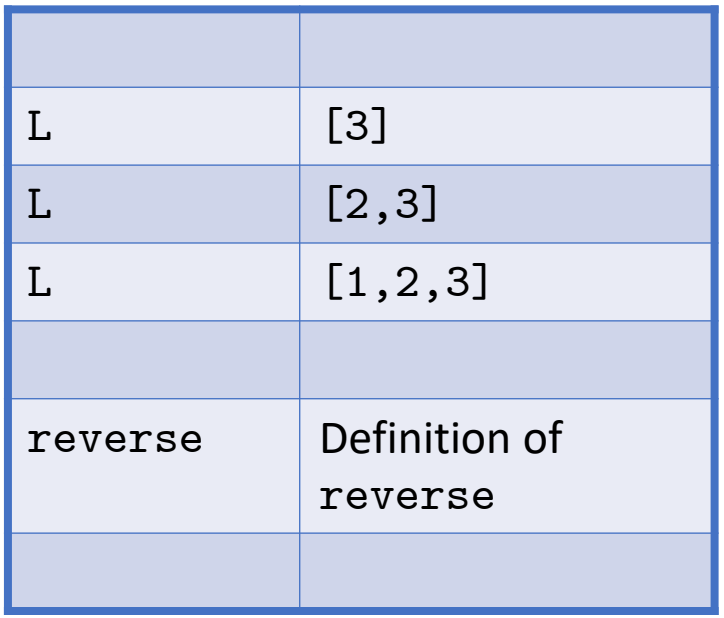


How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```

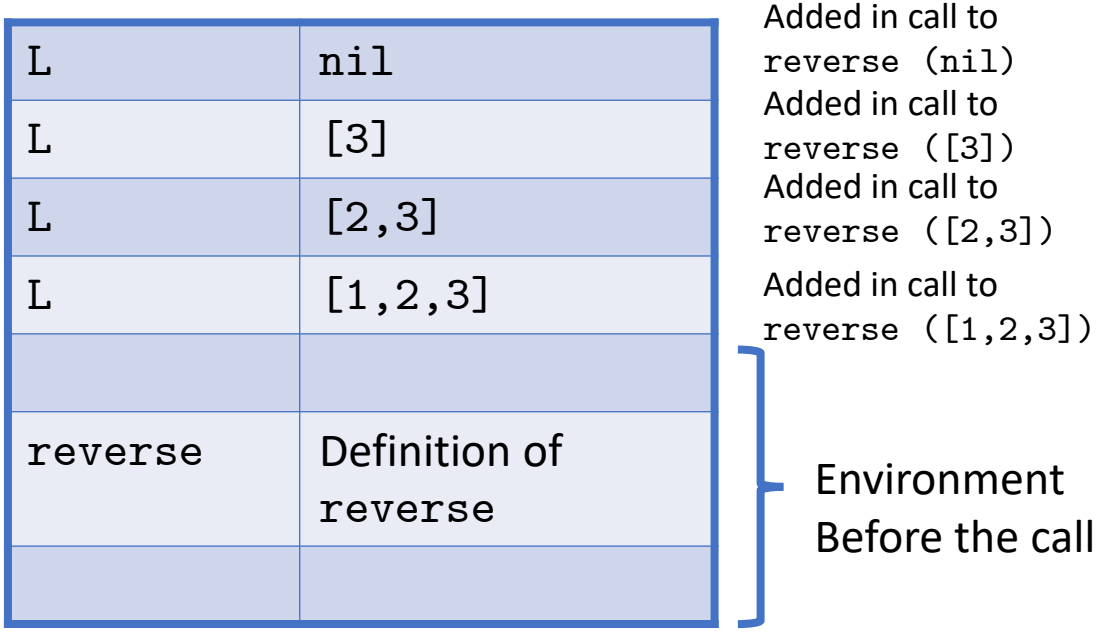


How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```

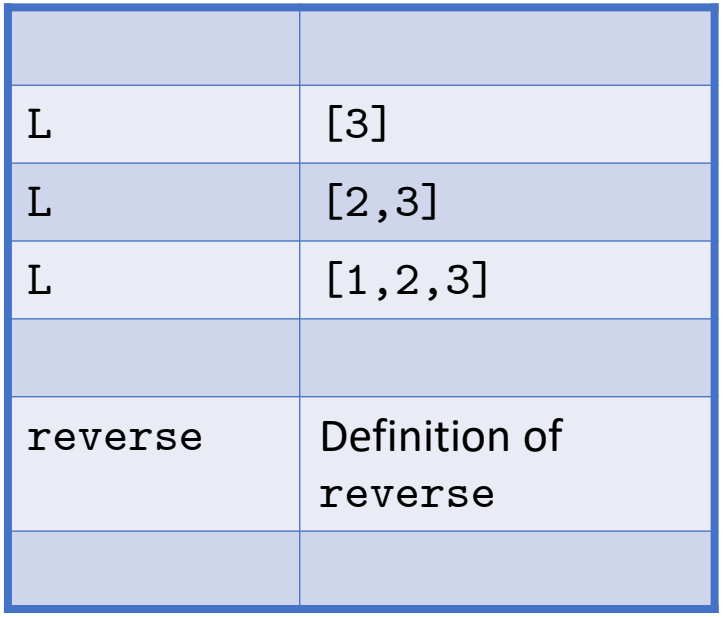


How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```



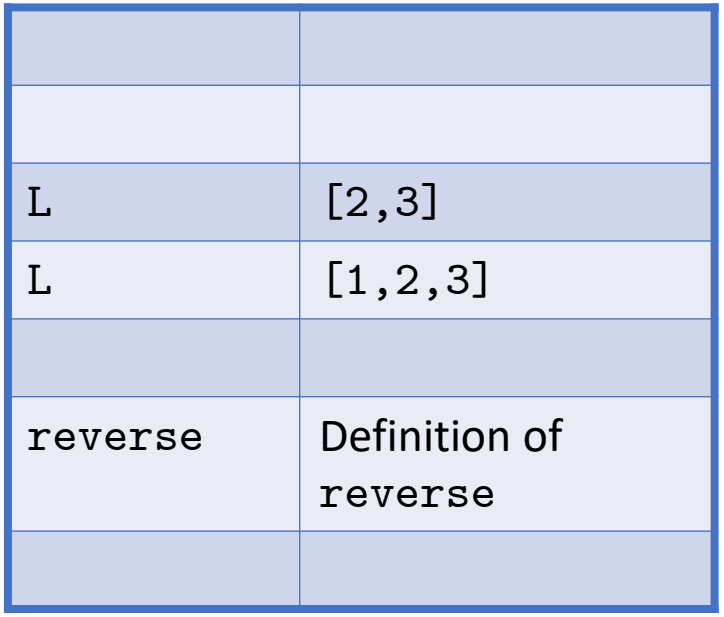
`nil@[3] =[3]`

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```



Added in call to `reverse ([2,3])`

Added in call to `reverse ([1,2,3])`

Environment Before the call

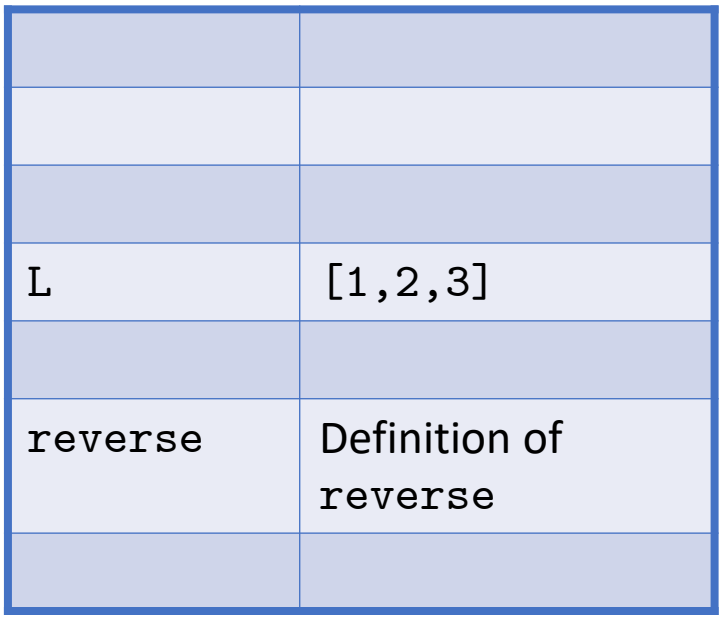
`[3]@[2] =[3,2]`

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

```
> fun reverse L =
      if L = nil then nil
      else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
val it = [3, 2, 1]: int list
```



`[3,2]@[1] =[3,2,1]`

Different ways for writing functions

- Syntactic sugar notation for functions with names

```
> fun increment n = n+1;
val increment = fn: int -> int
```

- Syntax `fn` (corresponds with λ in the λ-calculus)

```
fn <param> => <expression>;
```

- We can associate the functions to a name, just like values

```
> val increment = fn n => n+1;
val increment = fn: int -> int
```

- Or we can directly apply the function to the parameter (anonymous function)

```
(fn n => n+1) 5;
```

- In case the function is recursive with fn, we need to use `rec`

```
> val rec fact n = fn 0 => 1
              |n => n*fact(n-1);
```

Nonlinear recursion

- A function can call itself recursively multiple times
- Example: Number of combinations of *k* things out of *n*
  - Written $\binom{n}{k}$
  - Can be shown to be equal to $\frac{n!}{(n-k)!k!}$
  - We can also use the following recursive definition

Combinations

- Base case. If $k = 0$ the number of ways to pick 0 out of n is 1. Similarly, if $k = n$, there is exactly one way to pick n out of n.

$$\binom{n}{0} = \binom{n}{n} = 1$$

- Induction. If $0 < k < n$ to select k out of n we can
  - reject the first thing and select k out of the remaining $n - 1$
  - pick the first thing, and pick $k - 1$ out of the remaining $n - 1$
  - formally

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- We assume that 0 <=k<=n
- We use this to write a ML program

Combinations

```
> fun comb(n,k) = (* assumes 0 <= k <= n *)
    if k=0 orelse k=n then 1
    else comb(n-1,k) + comb(n-1,k-1);
val comb = fn: int * int -> int
```

- Without using `orelse`:

```
> fun comb (n,k) =
    if k=0 then 1
    else
        if k=n then 1 else comb(n-1,k)+comb(n-1,k-1);
val comb = fn: int * int -> int

> comb (5,0);
val it = 1: int
> comb (5,5);
val it = 1: int
> comb (5,2);
val it = 10: int
```

Mutual recursion

Mutual recursion

- How can we do mutual recursion (functions, types) in languages where a name must be declared before use?
  1. Relax this rule for functions and/or types
     - Java via methods

```
{void f(){
    g();
}
{void g(){
    f();
 }
}
```

     - Pascal by pointer types

```
type list = ^elem;
type elem = record
     info: integer;
     next: list;
end
```

^T denotes the type of pointers to objects of type T

Mutual recursion

2. Incomplete definitions
   - Ada

```
type elem;
type list is access elem;
type elem is record
     info: integer;
     next: list;
end
```

   - C

```
struct elem;
struct elem {
     int info;
     elem *next }
end
```

   - Pascal

```
procedure fie(A:integer); forward;
procedure foo(B: integer);
     begin ... fie(3); ... end
procedure fie;
     begin ... foo(4); ... end
```

Mutual recursion in ML

Mutual recursion in ML

- Two functions can call one another recursively
- Example. A function that takes a list `L` and produces a list with the first, third, fifth etc. elements of `L`
- Two functions:
  - `take(L)` takes the first element of `L` and then alternates
  - `skip(L)` skips the first element and then calls `take`

First attempt

```
> fun take(L) =
    if L = nil then nil
    else hd(L) :: skip(tl(L));
> fun skip(L) =
    if L = nil then nil
    else take(tl(L));

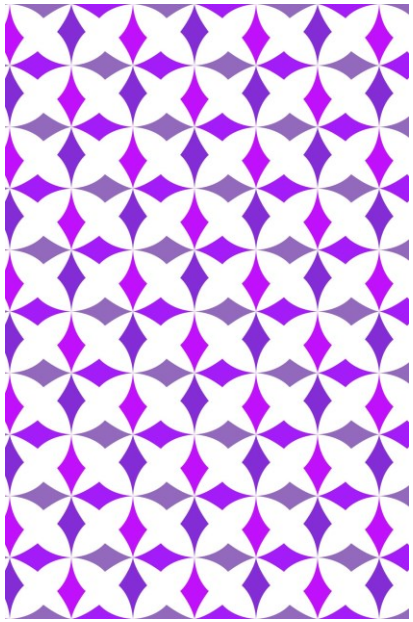
> fun take(L) =
    if L = nil then nil
    else hd(L) :: skip(tl(L));
poly: : error: Value or constructor (skip) has not been
declared
Found near if L = nil then nil else hd (L) :: skip (tl (L))
```

Solution: keyword **and**

```
fun
        <definition of first function>
and
        <definition of second function>
and ...
```

```
fun
take(L) =
        if L = nil then nil
        else hd(L) :: skip(tl(L))
and
skip(L) =
        if L = nil then nil
        else take(tl(L));
val skip = fn: ''a list -> ''a list
val take = fn: ''a list -> ''a list

> take ([1,2,3,4,5]);
val it = [1, 3, 5]: int list
```



Patterns in ML

Function definition through patterns

- Similar to “case” or “switch” statements in procedural languages
- The starting point is a pattern
- Example: `x::xs` is a pattern that matches any non-empty list, with `x` set to the head and `xs` to the tail of the list
- Function definition uses a sequence of patterns: the first that matches the argument determines the produced value

```
fun <identifier> (<first pattern>) = <first expression>
  | <identifier> (<second pattern>) = <second expression>
  ...
  | <identifier> (<last pattern>) = <last expression>;
```

Example: reverse a list

- Using patterns

```
> fun reverse (nil) = nil
    | reverse (x::xs) = reverse(xs) @ [x];
val reverse = fn: 'a list -> 'a list
```

- Without patterns

```
> fun reverse L =
    if L = nil then nil
    else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list
```

List of alternatives

- The list of alternatives must be exhaustive, as with `if-then` clauses
- If the list is not exhaustive, many implementations of ML only give a warning, with an error only if we actually use a parameter that does not match any of the possibilities

```
> fun reverse (nil) = nil;
poly: : warning: Matches are not exhaustive. Found near fun
reverse (nil) = nil
val reverse = fn: 'a list -> 'b list
> reverse([3]);
poly: : warning: The type of (it) contains a free type
variable. Setting it to a unique monotype.
Exception- Match raised
```

Reverse a list with patterns

```
> fun reverse (nil) =
nil
    | reverse (x::xs) =
    reverse(xs) @ [x];
val reverse = fn: 'a
list -> 'a list

> reverse([1,2,3])
```

We even do not try to match the second pattern

xs	nil
x	3
xs	[3]
x	2
xs	[2,3]
x	1
reverse	Definition of reverse

Added in call to `reverse (nil)`

Added in call to `reverse ([3])`

Added in call to `reverse ([2,3])`

Added in call to `reverse ([1,2,3])`

Environment Before the call

`as`: match pattern and assign variables

- We can also assign a name to the value of the whole pattern
- At one time give the value to an identifier and match the value with a pattern

```
<identifier> as <pattern>
```

- Example

```
> fun reverse (nil) = nil
    | reverse (L as x::xs) = reverse(xs) @ [x];
val reverse = fn: 'a list -> 'a list
```

Why should this be useful?

`as`: another example

- Merge two lists of integers `L` and `M`, assuming that they are sorted (smallest first)
- Base case. If `L` is empty then the merge is `M` (and viceversa)
- Inductive case. Compare the heads of `L` and `M`. If the head of `L` is smaller add it as head and recursively call on the tail of `L` and `M`, otherwise add the head of `M` as head and recursively call on `L` and on the tail of `M`.
- ```
  > fun merge (nil,M) = M
        | merge (L,nil) = L
        | merge (L as x::xs, M as y::ys) =
            if x<y then x::merge(xs,M)
            else y::merge (L,ys);
  val merge = fn: int list * int list -> int list
  ```
  ```

Without as

- Of course, we could do it also without `as`, but it would be slightly more complicated

```
fun merge (nil,M) = M
    | merge (L,nil) = L
    | merge (x::xs,y::ys) =
        if x<y then x::merge(xs,y::ys)
        else y::merge(x::xs,ys);
```

Anonymous (or wildcard) variables

- Used when we want to match a pattern, but never need to refer to the value again

```
> fun comb (_,0) = 1
    | comb (n,k) =
        if k=n then 1
        else comb(n-1,k)+comb(n-1,k-1);
val comb = fn: int * int -> int
```

Multiple uses of variables in a pattern

- A variable can be used only once in a pattern
- The following is illegal

```
> fun comb (_,0) = 1
    | comb (n,n) = 1
    | comb (n,m) =
        comb(n-1,m)+comb(n-1,m-1);
poly: : error: n has already been bound in this
clause. Found near fun
comb (n, ...) = 1
```

- This should be written using `if-then` as before

Patterns allowed

- Constants, such as `nil` and `0`
- Expressions using `::`, such as `x::xs` or `x::y::xs`
- Tuples, such as `(x,y,z)`

Example

- Sum of all integers of a list of pairs of integers, e.g., given [(1,2),(3,4),(5,6)] we want to sum all the integers 1+2+3+…

```
> fun sumPairs (nil) = 0
    | sumPairs ((x,y)::zs) = x + y + sumPairs(zs);
val sumPairs = fn: (int * int) list -> int
> sumPairs [(1,2),(3,4),(5,6)];
val it = 21: int
```

Another example

- Input: list of lists of integers, e.g., `[[1,2],[2,3],[4]]`
- Output: Sum of these integers, e.g., 12

```
> fun sumLists (nil) = 0
    | sumLists (nil::YS) = sumLists (YS)
    | sumLists ((x::xs)::YS) = x+sumLists (xs::YS);
val sumLists = fn: int list list -> int

> sumLists ([[1,2],nil,[3],[3,4,5]]);
val it = 18: int
```

Patterns not allowed

- Arithmetic operators, list concatenation, and real values
- Example

```
> fun length (nil) = 0
| length (xs@[x]) = 1 + length(xs);
poly: : error: @ is not a constructor Found near xs @ [x]
```

- Two more examples

```
> fun square (0) = 0
| square(x+1) = 1 + 2*x + square (x);
poly: : error: + is not a constructor Found near x + 1

> fun f(0.0) = 0
| f(x) = x;
poly: : error: Real constants not allowed in patterns
```

No misspell errors

- We often use identifiers with a special meaning like `nil` (we can define even more with data constructors)

```
> fun reverse (niil) = nil
    | reverse (x::xs) = reverse(xs) @ [x];
poly: : warning: Pattern 2 is redundant.
Found near fun reverse (niil) = nil | reverse (... :: ...) =
... ... @ [...]
val reverse = fn: 'a list -> 'a list
```

- We need to be careful not to misspell them – otherwise we intend a pattern that matches anything
- This is not an error, but probably not what the user wanted

Some questions

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
TYZNPD

Cases and patterns in ML

Case

- We can perform pattern matching also through the construct `case`

```
fn x => case x of
    <pattern_1> => <expression_1>
    | <pattern_2> => <expression_2>
    ...
    | <pattern_n> => <expression_n>
```

- This is an expression, so every `x` must satisfy one case

`Case`: an example

```
val day = fn n => case n of
    1 => "Monday"
  | 2 => "Tuesday"
  | _ => "Other";
```

Default value

```
> day 1;
val it = "Monday": string
> day 4;
val it = "Other": string
```

What happens if we omit the default case?

```
> val daynd = fn n => case n of
  1 => "Monday"
  | 2 => "Tuesday";
poly: : warning: Matches are not exhaustive.
Found near case n of 1 => "Monday" | 2 => "Tuesday“

val daynd = fn: int -> string
> daynd 1;
val it = "Monday": string
> daynd 4;
Exception- Match raised
```

It complains!

Patterns do not need to be constant values

- The pattern does not have to be a constant value, as in most programming languages as ML uses a mechanism of pattern matching
- Example

```
> val f = fn a => case a of
  0 => 1000.0
  | x => 1.0/real x;
val f = fn: int -> real

> f 0;
val it = 1000.0: real
> f 1;
val it = 1.0: real
> f 2;
val it = 0.5: real
> f 10;
val it = 0.1: real
```

Pattern matching with `fn`

- Case statements can be replaced by pattern matching

```
> val day = fn 1 => "Monday"
| 2 => "Tuesday"
| _ => "Other";
val day = fn: int -> string
val it = (): unit

> day 5;
val it = "Other": string
```

- Another example of pattern matching: assigns two variables with a single statement

```
> val (x,y) = (4,5);
val x = 4: int
val y = 5: int
```

Cases and pattern matching

Cases

```
> val day = fn x =>
case x of
      1 => "Monday"
    | 2 => "Tuesday"
    | _ => "other";
val day = fn: int -
> string
```

Pattern matching

```
> val day =
   fn 1 => "Monday"
     |2 => “Tuesday
     |_ => "other";
val day = fn: int -
> string
```

Fun and fn with cases

Fun

```
> fun day x = case
x of
      1 => "Monday"
    | 2 => "Tuesday"
    | _ => "other";
val day = fn: int -
> string
```

Fn

```
> val day = fn x =>
case x of
      1 => "Monday"
    | 2 => "Tuesday"
    | _ => "other";
val day = fn: int -
> string
```

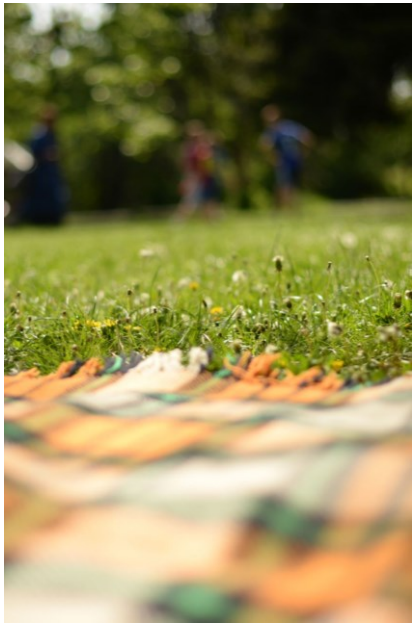
Fun and fn with pattern matching

Fun

```
> fun day 1 = "Monday"
  | day 2 = "Tuesday"
  | day _ = "other";
val day = fn: int ->
string
```

Fn

```
> val day =
    fn 1 => "Monday"
     |2 => “Tuesday
     |_ => "other";
val day = fn: int ->
string
```



Local environment in ML

Local environments using `let`

- Create local values inside a function declaration

```
> fun name(par) =
    let
        val <first variable> = <first expression>;
        val <second variable> = <second expression>;
        ...
        val <last variable> = <last expression>
    in
        <expression>
    end;
```

Block / local environment in ML

Example

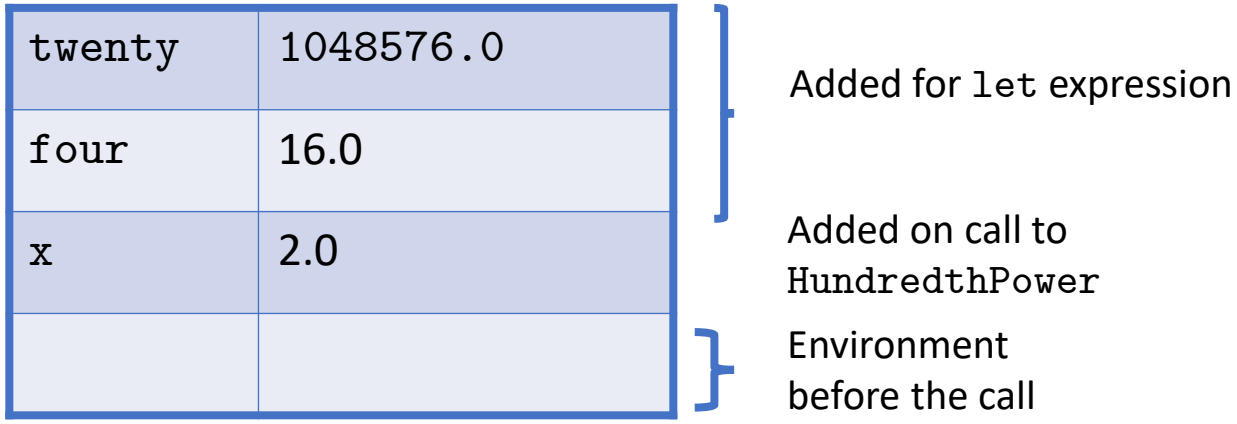
- Example: defining common subexpressions

```
> fun hundredthPower (x:real) =
    let
        val four = x*x*x*x;
        val twenty = four * four * four * four * four
    in
        twenty * twenty * twenty * twenty * twenty
    end;
val hundredthPower = fn: real -> real

> hundredthPower 1.01;
val it = 2.704813829: real
> hundredthPower 2.0;
val it = 1.2676506E30: real
```

Let environment

- When we enter a `let` expression an addition to the current environment is created



Alternative

- There is no need to introduce new names:

```
> fun hundredthPower (x:real) =
    let
        val x = x*x*x*x;
        val x = x*x*x*x*x;
    in
            x*x*x*x*x
    end;
val hundredthPower = fn: real -> real
```

x	1048576.0
x	16.0
x	2.0

Added for `let` expression

Added on call to `HundredthPower`

Environment before the call

Why do we need it?

Let: decomposing the result of a function

- Suppose f returns tuples of size 3
- We can decompose the result into components by writing

```
val (a,b,c) = f (...)
```

- Example: A function `split (L)` that splits L into 2 lists:
  - The first, third, 5th etc
  - The second, fourth etc.

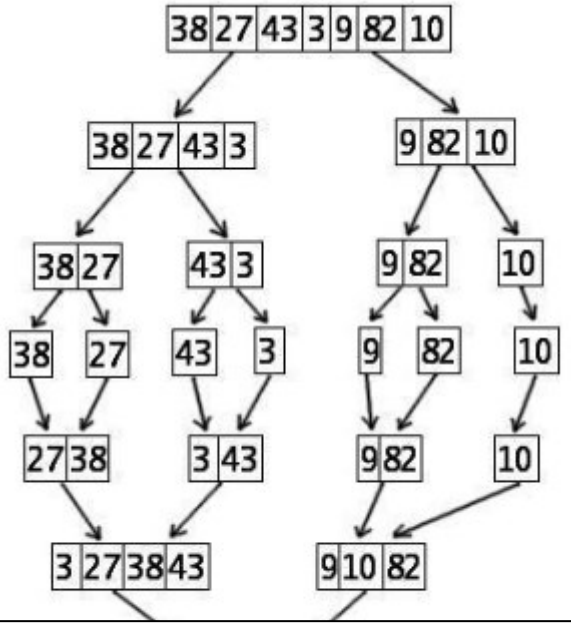
Splitting lists

```
> fun split(nil) = (nil,nil)
    | split([a]) = ([a],nil)
    | split (a::b::cs) =
        let
            val (M,N) = split (cs)
        in
            (a::M,b::N)
        end;
val split = fn: 'a list -> 'a list * 'a list

> split [1,2,3,4,5];
val it = ([1, 3, 5], [2, 4]): int list * int list
```

Another example: mergeSort

[from Wikipedia]



We have a `split` function from the previous example – that splits [1st, 3rd, 5th, …] and [2nd, 4th,6th, …]

We defined `merge` function before: it merges and orders two lists

```
fun merge (nil,M) = M
        | merge (L,nil) = L
        | merge (L as x::xs, M as y::ys) =
            if x<y then x::merge(xs,M)
            else y::merge (L,ys);
    val merge = fn: int list * int list -> int list
```

Another example: mergeSort

```
> fun mergeSort (nil) = nil
    | mergeSort([a]) = [a]
    | mergeSort (L) =
    let
        val (M,N) = split L;
        val M = mergeSort (M);
        val N = mergeSort (N)
    in
        merge (M,N)
    end;
val mergeSort = fn: int list -> int list

> mergeSort [1,4,2,3,8,7];
val it = [1, 2, 4, 3, 7, 8]: int list
> mergeSort([5,3,2,6,4,1]);
val it = [1, 2, 3, 4, 5, 6]: int list
```

Summary

- Recursion in ML
- Mutual recursion (in ML)
- Patterns in ML
- Cases and patterns in ML
- Local environment in ML

Next time

- Input and output
- Exceptions