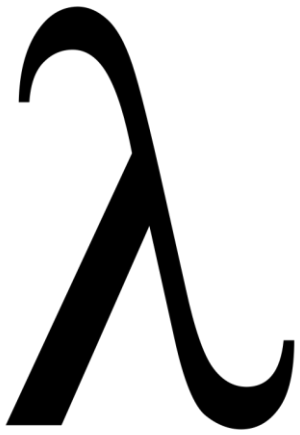


ML

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Today

- Recap
- Local environment

Recap

Function definition: patterns

- A pattern is an expression with variables
- Example

  `x::xs`

  matches any non-empty list, with `x` set to the head and `xs` to the tail
- 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>;
```

`as`: match pattern and assign variables

- At one time give the value to an identifier and match the value with a pattern

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

- Example: Merge two lists of integers `L` and `M`, assuming that they are be sorted (smallest first)
-

```
> 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
```

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
```

Patterns

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

Allowed

- Arithmetic operators
- Concatenation (@)
- Real values

Not allowed

Exercise L4.16

- Write a function `insertAll` that takes an element `a` and a list of lists `L` and inserts `a` at the front of each of these lists. For example `insertAll (1,[[2,3],[],[3]])=[[1,2,3],[1],[1,3]]`

Solution L4.16

```
> fun insertAll(a,nil) = nil
    | insertAll(a,L::Ls) =
       (a::L)::insertAll(a,Ls);
val insertAll = fn: 'a * 'a list list -> 'a list
list

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

Exercise L4.17

- Suppose that sets are represented by lists. Write a function that takes a list, and produces the power set of the list
- If S is a set, the power set of S is the set of all subsets S' such that $S' \subseteq S$

E.g., `S=[1,2,3]`,

`powerSet(S)=[[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]`

Solution L4.17

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


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

Cases and patterns

Different ways for writing functions

- Syntax `fn` (corresponds with λ in the λ-calculus, that we will see later)

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

- We can also directly apply the function to the parameter (anonymous function)

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

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

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

- In case the function is recursive use rec

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

- Syntactic sugar notation for functions with names (no need to specify rec)

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

Cases

- Syntax

```
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
- 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
- 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

- 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

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

- Assigns two variables with a single statement

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
```

Exercise L5.1

- Write a function `is_one` that returns “one” if the parameter is 1 and “anything else” otherwise, using the construct `case` and pattern matching with `fun` and `fn`.

Solution Exercise L5.1

```
val is_one = fn x => case x of
      1 => "one"
      |_ => "anything else ";

> is_one 1;
val it = "one": string
> is_one 3;
val it = "anything else": string
```

Solution Exercise L5.1

```
val is_one = fn
1 => "one"
|_ => "anything else";

> is_one 1;
val it = "one": string
> is_one 3;
val it = "anything else": string
```

- This would be wrong …why?

```
val f = fn
_ => "anything else";
| 1 => "one"
```

It would always match anything else

Exercise L5.2

- Write a case statement to simulate the if-then-else clause, e.g., write a case statement `is_lower_than5` such as if a value is lower than 5, then 1, otherwise 2

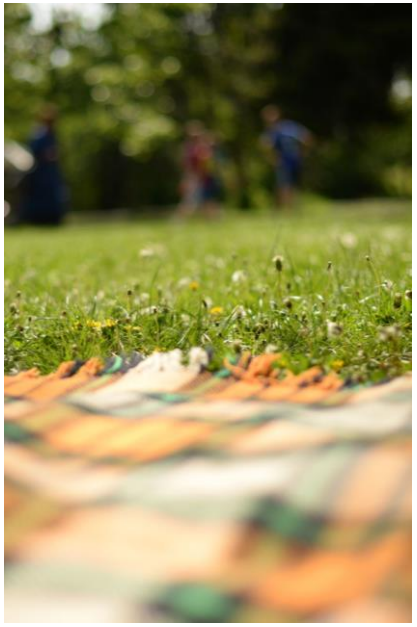
Solution Exercise L5.2

```
> val is_lower_than5 = fn n => case n<5 of
true => 1
|false => 2;
val is_lower_than5 = fn: int -> int
val it = (): unit

> is_lower_than5 3;
val it = 1: int
> is_lower_than5 7;
val it = 2: int
```

You can simulate the if-then-else clause with a case statement:

```
case booleanExpr of
true => expr1
| false => expr2;
```



Local environment

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

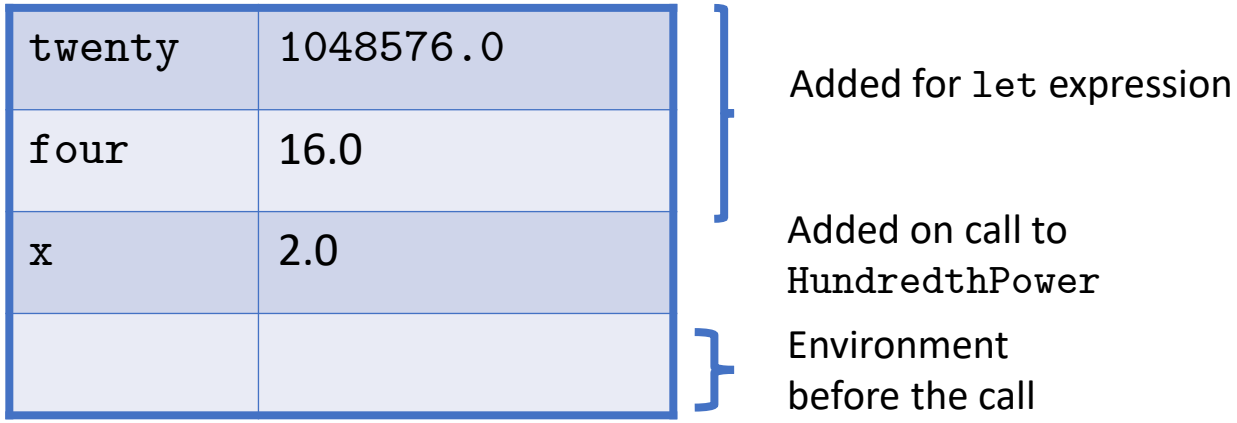
- Example: defining common subexpressions

```
> 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

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.

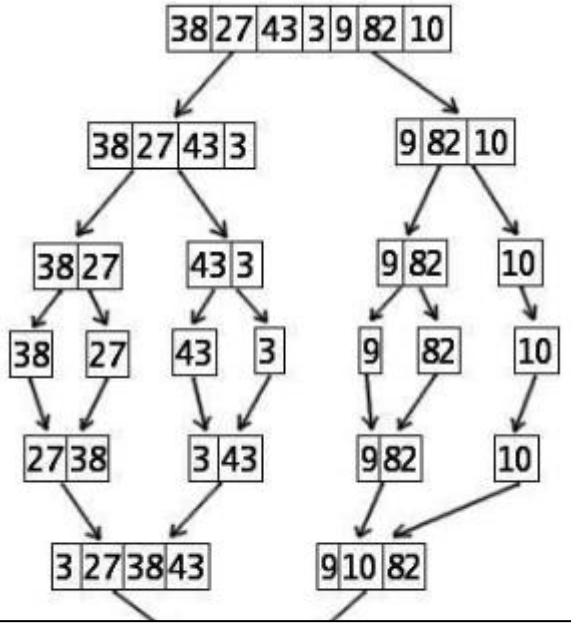
Splitting lists

```
> 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` in the last lecture: 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
```

Exercise L5.3

- Write a short program `thousandthPower` that, given a real x, uses let to compute x^{1000}

Solution L5.3

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

> thousandthPower 1.1;
val it = 2.469932918E41: real
```

Exercise L5.4

- Write the `split` program without using a pattern (the tuple) in the `val` declaration but referencing the components of the tuple

```
> 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
```

Solution L5.4

```
fun split(nil) = (nil,nil)
    | split([a]) = ([a],nil)
    | split (a::b::cs) =
    let
        val x = split (cs);
        val M = #1 x;
        val N = #2 x
    in
        (a::M,b::N)
    end;
val split = fn: 'a list -> 'a list * 'a list

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

Exercise L5.5

- Improve the powerset function by using a `let` and computing the powerset of the tail only once

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

Solution L5.5

```
> fun powerSet(nil) = [nil]
    | powerSet(x::xs) =
    let
        val L = powerSet(xs)
    in
        L @ insertAll(x,L)
    end;
val powerSet = fn: 'a list -> 'a list list

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

Exercise L5.6

- Write an improved function to find the largest of a list of reals using a `let`.
- Suggestion: you can think about the maximum of the tail

```
fun maxList([x:real]) = x
    | maxList(x::y::zs) =
       if x<y then maxList(y::zs)
       else maxList(x::zs);
poly: : warning: Matches are not exhaustive.
val maxList = fn: real list -> real
```

Solution L5.6

```
> fun maxList(nil) = 0.0
    | maxList ((x)::xs) =
    let
        val maxTail = maxList (xs)
    in
        if x<maxTail then maxTail else x
    end;
val maxList = fn: real list -> real

> maxList [1.0,4.5,3.2];
val it = 4.5: real
```

Exercise L5.7

- Write an efficient program `doubleExp` to compute x^{2^i}, for real x and nonnegative i, making only one recursive call.

- Please remember that

$$x^{(2^i)} = x^{2*2^{i-1}} = x^{2^{i-1}+2^{i-1}} = x^{2^{i-1}} * x^{2^{i-1}}$$

Solution L5.7

```
> fun doubleExp(x:real,0) = x
    | doubleExp(x,i) =
        let
            val y = doubleExp(x,i-1)
        in
            y*y
        end;
val doubleExp = fn: real * int -> real

> doubleExp(1.1,3);
val it = 2.14358881: real
```

Exercise L5.8

- Write a function `sumPairs` that takes a list of pairs of integers, and returns a pair of the sum of each component using the `let val`

Solution L5.8

```
> fun sumPairs (nil) = (0,0)
    | sumPairs ((x,y)::zs) =
    let
        val(z1,z2) = sumPairs(zs)
    in
        (x+z1,y+z2)
    end;

val sumPairs = fn: (int * int) list -> int * int
> sumPairs [(1,2),(3,4),(5,6)];
val it = (9, 12): int * int
```

Exercise L5.9

- Write a function `sumList` that takes a list of integers and returns a pair of the sum of the even positions and the sum of the odd positions
- E.g., given [1,2,3,4], `sumList`([1,2,3,4]) = (4,6)

Solution L5.9

```
> fun sumList (nil) = (0,0)
    | sumList ([x]) = (0,x)
    | sumList (x::y::zs) =
    let
        val (sumOdd, sumEven) = sumList (zs)
    in
        (x+sumOdd, y+sumEven)
    end;

val sumList = fn: int list -> int * int
> sumList [1,2,3,4,5];
val it = (9, 6): int * int
```

Summary

- Local environment

Next time

- Input/output
- Exceptions