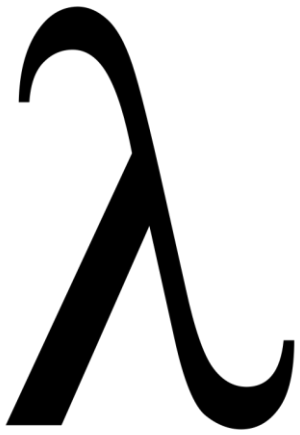


Input-output, exceptions and polymorphic functions

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

Tomorrow's tutoring

- Tomorrow's tutoring in A202

Today

- Input and output in ML
- Exceptions in ML
- Polymorphic types in ML

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
```

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
```

… and `x` can also be an expression

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

- This is the main difference between using pattern matching in functions and the `case` statement
- For instance, we can use the `case` statement for writing a function `is_lower_than5` that simulates the `if-then-else` clause, e.g., it returns 1 if a value is lower than 5, 2 otherwise

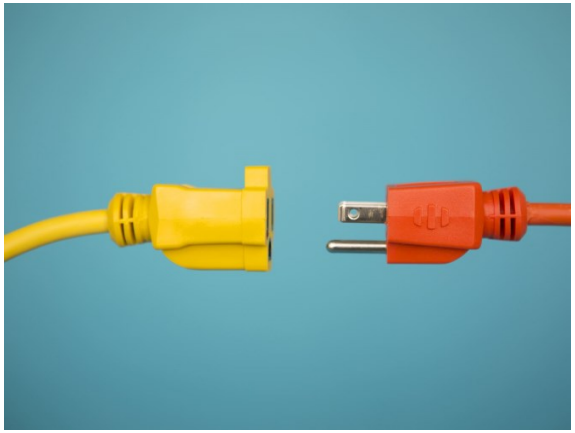
… and `x` can also be an expression

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

> 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;
```



Input and output in ML

Output in ML

Output

- `print(x)` prints a string
- What is the type of print?

Printing

```
> print;
val it = fn: string -> unit

> print ("ab");
abval it = (): unit

> print ("ab\n");
ab
val it = (): unit

> fun testZero(0) = print("zero\n")
    | testZero(_) = print("not zero\n");
val testZero = fn: int -> unit

> testZero(2);
not zero
val it = (): unit
```

`unit`: used for expressions and functions that do not return a value. It has a unique value: `()`

`print` has a side-effect: it changes the *stdout*

`print` does not return the value printed

Printing non-strings

- Characters

```
> val c = #"a";
val c = #"a": char

> str;
val it = fn: char -> string

> print (str(c));
a val it = (): unit
```

printed character

Other conversions

```
> val x = 1.0E50;
val x = 1E50: real

> print(Real.toString(x));
1E50val it = (): unit

> print(Int.toString(123));
123val it = (): unit

> print(Bool.toString(true));
trueval it = (): unit
```

Real, Int and Bool are (data) structures in ML, that are part of the standard basis in ML. The identifier `toString` can denote different functions depending on the structure it is applied to.

Compound statements

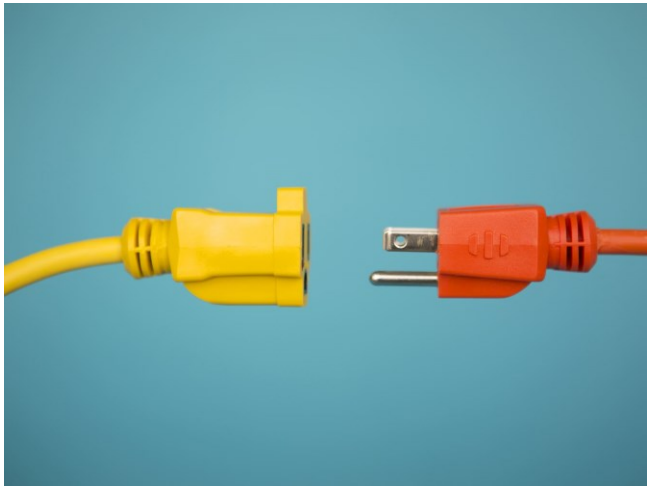
- We can also write compound statements like

```
> (print(Real.toString(1.0E50));
print(Int.toString(123)) );
1E50123val it = (): unit
```

Technically, we do not have statements in ML but expressions causing side - effects

Note that the last instruction does not need the ;

- The type of a compound statement is that of the last statement
- Be careful! You cannot use it as a sequence of statements in a procedural language!



Input in ML

File

```
cat test
12
ab
```

- Open the file

```
> val infile = TextIO.openIn ("test");
val infile = ? : TextIO.instream
```

Open the file "test"

Token or internal value of the structure `TextIO instream`

Instreams

```
> TextIO.endOfStream (infile);
val it = false: bool
```

Check whether it is the end of the stream

```
> TextIO.inputN (infile,4);
val it = "12\na": string
```

Read 4 characters

```
> TextIO.inputN (infile,1);
val it = "b": string
```

Read 1 character

```
> TextIO.inputN (infile,1);
val it = "\n": string
```

Read 1 character

```
> TextIO.endOfStream (infile);
val it = true: bool
```

Check whether it is the end of the stream

Reading lines of a file

```
> val infile = TextIO.openIn ("test");
val infile = ?: TextIO.instream

> TextIO.inputLine (infile);
val it = SOME "12\n": string option

> TextIO.inputLine (infile);
val it = SOME "ab\n": string option

> TextIO.inputLine (infile);
val it = NONE: string option

> TextIO.closeIn(infile);
val it = (): unit
```

Special type constructor **T option**:

- SOME, when the value is a value of type T
- NONE, otherwise

Read 1 line

No more lines to read

Close infile

Reading the complete file

```
> val infile = TextIO.openIn ("test");
val infile = ?: TextIO.instream

> val s = TextIO.input (infile);
val s = "12\nab\n": string
```

Reading a single character

- Reads a single character.

```
> TextIO.input1;
val it = fn: TextIO.instream -> char option
```

- The type `T Option` can help identify the end of a file without `endofStream`

- For instance these two functions read characters in a file and put them into a char list

```
> fun makeList1 (infile, NONE) = nil
  | makeList1 (infile, SOME c) =
        c::makeList1(infile, TextIO.input1(infile));
val makeList1 = fn: TextIO.instream * char option -> char list
> fun makeList(infile) = makeList1(infile, TextIO.input1(infile));
val makeList = fn: TextIO.instream -> char list
```

Lookahead

- Reads the next character, but leaves it in the input stream, i.e., it does not consume the character read as `input1` does.

```
> TextIO.lookahead;
  val it = fn: TextIO.instream -> char option
```

Are there n characters left?

- Are there at least n characters available on instream f? It returns int option (SOME n or SOME m<n)

```
> TextIO.canInput;
  val it = fn: TextIO.instream * int -> int option

> TextIO.canInput(f,50);
  val it  = SOME 10: int option
```

Summing up … by return type

`String`

- `inputN`
- `input`

`'a option`

- `inputLine (string option)`
- `input1 (char option)`
- `lookahead (char option)`
- `canInput (int option)`

valOf

- How to transform a type ‘a option into the corresponding type ‘a?
  - `valOf opt`: returns the value if `opt` is `SOME`, otherwise it raises the `Option` exception

```
> valOf;
val it = fn: 'a option -> 'a

> fun convert (a: 'a option) = valOf(a);
val convert = fn: 'a option -> 'a
```

Exceptions in ML

Exceptions

```
> 5 div 0;
Exception- Div raised

> hd (nil: int list);
Exception- Empty raised

> tl (nil: real list);
Exception- Empty raised

> chr (500);
Exception- Chr raised
```

User-defined exceptions

```
> exception Foo;
exception Foo
```

Defining

```
> Foo;
val it = Foo: exn
```

`exn` is the type of the exception

```
> raise Foo;
Exception- Foo raised
```

Raising

An example

```
> exception BadN;
exception BadN
> exception BadM;
exception BadM
> fun comb(n,m)=
    if n<0 then raise BadN
    else if m<0 orelse m>n then raise BadM
    else if m=0 orelse m=n then 1
    else comb(n-1,m) + comb (n-1,m-1);
val comb = fn: int * int -> int

> comb(5,2);
val it = 10: int
> comb(~1,0);
Exception- BadN raised
> comb(5,6);
Exception- BadM raised
```

Exceptions with parameters

```
exception <identifier> of <type>;
```

- In this case the identifier becomes an exception constructor

```
> exception Foo of string;
exception Foo of string
> Foo;
val it = fn: string -> exn

> raise Foo ("bar");
Exception- Foo "bar" raised
> raise Foo(5);
poly: : error: Type error in function application.
> raise Foo;
poly: : error: Exception to be raised must have type exn.
```

Handling exceptions

```
<expression> handle <match>
```

Handling

- For instance

```
> exception OutOfRange of int * int;
> fun comb1(n,m)=
    if n <= 0 then raise OutOfRange (n,m)
    else if m<0 orelse m>n then raise OutOfRange (n,m)
    else if m=0 orelse m=n then 1
    else comb1 (n-1,m) + comb1 (n-1,m-1);
val comb1 = fn: int * int -> int
```

Handling exceptions

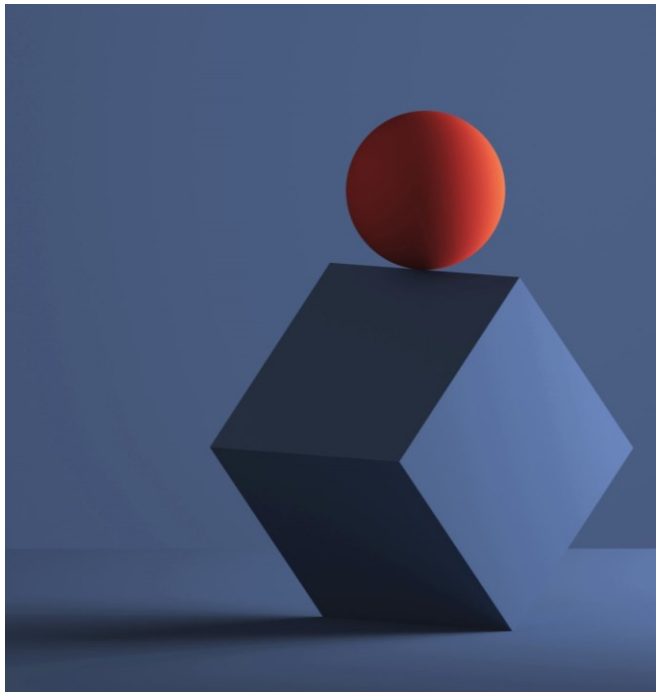
```
> fun comb (n,m) = comb1 (n,m) handle
    OutOfRange (0,0) => 1
    | OutOfRange (n,m) => (
        print ("out of range: n=");
        print (Int.toString(n));
        print (" m=");
        print (Int.toString(m));
        print ("\n");
    0
    );
val comb = fn: int * int -> int
```

Handling exceptions

```
> comb (4,2);
val it = 6: int

> comb (3,4);
out of range: n=3 m=4
val it = 0: int

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



Polymorphic functions in ML

Polymorphic functions

- Polymorphism: function capability to allow multiple types (“poly”=“many” + ”morph”=“form”)
- Remember: ML is strongly typed at compile time, so it must be possible to determine the type of any program without running it
- Although we must be able to identify the types, we can define functions whose types are partially or completely flexible
- Polymorphic functions: functions that permit multiple types
- ML uses ‘a for denoting generic polymorphic type

Examples

- Simple example

```
> fun identity (x) = x;
val identity = fn: 'a -> 'a
> identity (2);
val it = 2: int
> identity (2.0);
val it = 2.0: real
```

- We can even write

```
> identity (ord);
val it = fn: char -> int
```

- We can use the function twice in an expression with different types

```
> identity (2) + floor (identity (3.5));
val it = 5: int
```

Operators that restrict polymorphism

- Arithmetic operators: `+`, `-`, `*` and `~` → default type
- Division-related operators: `/`, `div` and `mod`
- Inequality comparison operators → default type
- Boolean connectives: `andalso`, `orelse` and `not`
- String concatenation operators
- Type conversion operators, ie., `ord`, `chr`, `real`, `str`, `floor`, `ceiling`, `round` and `truncate`

Operators that allow polymorphism

- Three classes in this category are:
  1. Tuple operators: `(..,..)`, `#1`, `#2`, …
  2. List operators: `::`, `@`, `hd`, `tl`, `nil`, `[]`
  3. The equality operators: `=`, `<>`

Equality types in ML

Equality types

- Types that allow the use of equality tests (= and <>)
- Integers, booleans, characters, but not reals
- Tuples or lists of equality types but not functions
- Type variables, whose values are restricted to be an equality type, are indicated with a double quote `''a`

More on equality types

- We can compare lists

```
> val L = [1,2,3];
val L = [1, 2, 3]: int list
> val M = [2,3];
val M = [2, 3]: int list
> L<>M;
val it = true: bool
> L = 1::M;
val it = true: bool
```

- But not functions

```
> identity = identity;
poly: : error: Type error in function application.
Function: = : ''a * ''a -> bool
Argument: (identity, identity) : ('a -> 'a) * ('b -> 'b)
Reason: Can't unify ''a to 'a -> 'a (Requires equality type)
```

Examples

```
> fun identity(x) = x;
val identity = fn: 'a -> 'a
> identity(2);
val it = 2: int
> identity(2.0);
val it = 2.0: real
```

```
> fun identity_eq(x) = if (x=x)
then x else x;
val identity_eq = fn: ''a ->
        ''a
> identity_eq(2);
val it = 2: int
> identity_eq(2.0);
poly: : error: Type error in
function application.
   Function: identity_eq : ''a ->
''a
   Argument: (2.0) : real
   Reason: Can't unify ''a to real
(Requires equality type)
Found near identity_eq (2.0)
Static Errors
```

Examples

```
> fun identity(x) = x;
val identity = fn: 'a -> 'a
> identity (2);
val it = 2: int
> identity (2.0);
val it = 2.0: real
```

```
> fun identity_t(x:''a) = x;
val identity_t = fn: ''a -> ''a
> identity_t(2);
val it = 2: int
> identity_t(2.0);
poly: : error: Type error in
function application.
   Function: identity_t : ''a ->
''a
   Argument: (2.0) : real
   Reason: Can't unify ''a to real
(Requires equality type)
Found near identity_t (2.0)
Static Errors
```

Examples with lists

```
> val L: 'a list=[];
val L = []: 'a list
> 2::L;
val it = [2]: int list
```

```
> val L: 'a list=[];
val L = []: 'a list
> 2.0::L;
val it = [2.0]: real list
```

```
> val M: ''a list=[];
val M = []: ''a list
> 2::M;
val it = [2]: int list
```

```
> val M: ''a list=[];
val M = []: ''a list
 2.0::M;
poly: : error: Type error in function
application.
   Function: :: : real * real list -> real
list
   Argument: (2.0, M) : real * ''a list
   Reason: Can't unify real to ''a (Requires
equality type)
Found near 2.0 :: M
Static Errors
```

Examples with lists and functions

```
> fun first(L) = hd(L);
val first = fn: 'a list -> 'a
> first([2]);
val it = 2: int
> first([2.0]);
val it = 2.0: real
```

```
> fun first_eq(L) = if
(hd(L)=hd(L)) then hd(L) else
hd(L);
val first_eq = fn: ''a list -> ''a
> first_eq([2]);
val it = 2: int
> first_eq([2.0]);
poly: : error: Type error in
function application.
   Function: first_eq : ''a list -
> ''a
   Argument: ([2.0]) : real list
   Reason: Can't unify ''a to real
(Requires equality type)
Found near first_eq ([2.0])
Static Errors
```

Examples with lists and functions

```
> fun first(L) = hd(L);
val first = fn: 'a list -> 'a
> first([2]);
val it = 2: int
> first([2.0]);
val it = 2.0: real
```

```
> fun first_t(L:''a list) = hd(L);
val first_t = fn: ''a list -> ''a
> first_t([2]);
val it = true: bool
> first_t([2.0]);
poly: : error: Type error in
function application.
   Function: first_t : ''a list ->
''a
   Argument: ([2.0]) : real list
   Reason: Can't unify ''a to real
(Requires equality type)
Found near first_t ([2.0])
Static Errors
```

Equality types and reverse lists

- A function computing the reverse of a list function as the one below can be applied only to equality types, e.g., we cannot apply it to real values or functions

```
> fun rev1 (L) =
    if L = nil then nil
    else rev1(tl(L)) @ [hd(L)];
val rev1 = fn: ''a list -> ''a list
```

It requires equality types

The reason is the test `L=nil`

Equality types and reverse lists

```
> rev1 [1.1,2.2,3.3];
poly: : error: Type error in function application.
   Function: rev1 : ''a list -> ''a list
   Argument: [1.1, 2.2, 3.3] : ''a list
   Reason: Can't unify ''a to ''a (Requires equality type)
Found near rev1 [1.1, 2.2, 3.3]
Static Errors

> rev1 [floor,trunc, ceil];
poly: : error: Type error in function application.
Function: rev1 : ''a list -> ''a list
Argument: [floor, trunc, ceil] : (real -> int) list
Reason: Can't unify ''a to real -> int (Requires equality type)
```

Reversing lists

- We can avoid this as follows

```
> fun rev2 (nil) = nil
    | rev2(x::xs) = rev2 (xs) @ [x];
val rev2 = fn: 'a list -> 'a list
```

- We can then reverse lists of reals

```
> rev2 [1.1,2.2,3.3];
val it = [3.3, 2.2, 1.1]: real list
```

- Or even lists of functions

```
> rev2 [floor, trunc, ceil];
val it = [fn, fn, fn]: (real -> int) list
```

Testing for empty list

- An alternative way for testing if a list is empty, without forcing it to be of equality type is

```
> fun rev3 (L) =
    if null(L) then nil
    else rev3(tl(L)) @ [hd(L)];
    val rev3 = fn: 'a list -> 'a list
> rev3 [floor,trunc, ceil];
val it = [fn, fn, fn]: (real -> int) list
```

Some questions

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
BRAFJQ

Tuple type

- Why cannot we write the following?

```
> fun f (x) = #1(x);
poly: : error: Can't find a fixed record type. Found near #1
Static Errors
```

- As the tuple could be of any arity – there is no polymorphic idea of a tuple of arbitrary arity.
- In these cases we need to use `let` so that we specify the arity of the tuple

```
> fun f(x) = let
                 val (x1,x2)=x
               in
                       x1
                end;
val f = fn: 'a * 'b -> 'a
```

- Of course, if you do not have further constraints, you can also specify that the formal parameter is a tuple of two items

```
> fun f(x,y) = x;
val f = fn: 'a * 'b -> 'a
```

Summary

- Input and output in ML
- Exceptions in ML
- Polymorphic types in ML

Next time

- Higher-order functions