λ

ML

Programmazione Funzionale
2023/2024
Università di Trento
Chiara Di Francescomarino

Today

- Recap
- Exceptions
- Polymorphic functions
- Higher-order functions

Recap

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

`toString()` and 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

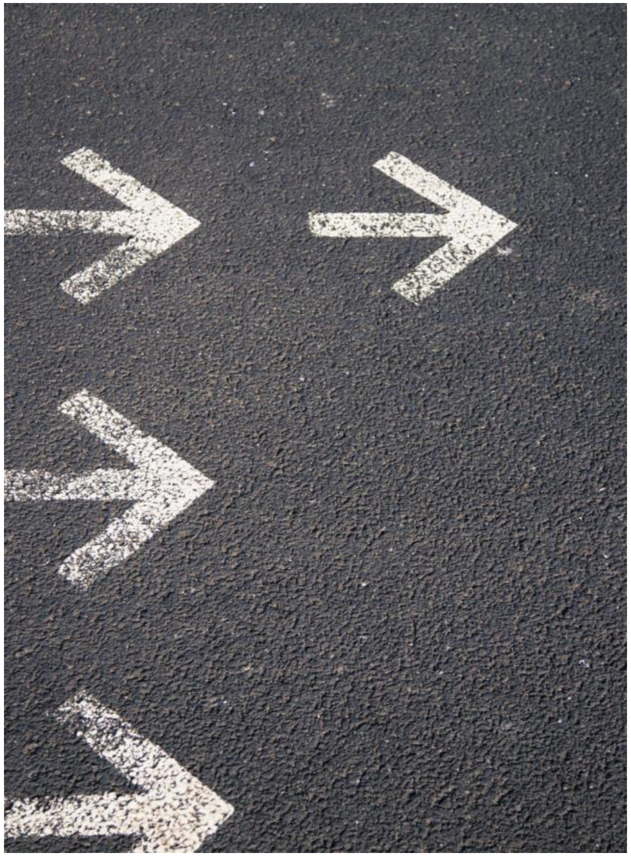
Note that the last instruction does not need the ;

The type of a compound statement is that of the last statement

Instream and its functions

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

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

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

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

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

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

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

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



Exceptions

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

> Foo;
val it = Foo: exn

> raise Foo;
Exception- Foo raised
```

exn is the type of the exception

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>
```

- 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
```

Exercise L7.1

- Write a program `returnThird(L)` that returns the third element of a list of integers. If the list is too short, it raises and handles an exception `shortList` by explicitly printing the length of the list.

Solution L7.1

```
> exception shortList of int list;
> fun returnThird1 L =
     if length(L) < 3 then raise shortList (L)
     else hd(tl(tl(L)));
val returnThird1 = fn: int list -> int
> fun returnThird L = returnThird1 L handle
     shortList L => (
     print ("List too short\n");
     0
     );
val returnThird = fn: int list -> int

> returnThird [1,2,3,4];
val it = 3: int
> returnThird [1,2];
List too short
val it = 0: int
```

Solution L7.1

- Another possible solution

```
> exception shortList of int;
> fun thirdElement1 nil = raise shortList(0)
                        |thirdElement1[x] = raise shortList(1)
                        |thirdElement1[x,y] = raise shortList(2)
                        |thirdElement1 L = hd(tl(tl(L)));

> fun thirdElement L = thirdElement1 L handle
        shortList n => (
                print("List too short\n");
                print("It only contains ");
                print(Int.toString(n));
                print(" elements\n");
                0);
```

Exercise L7.2

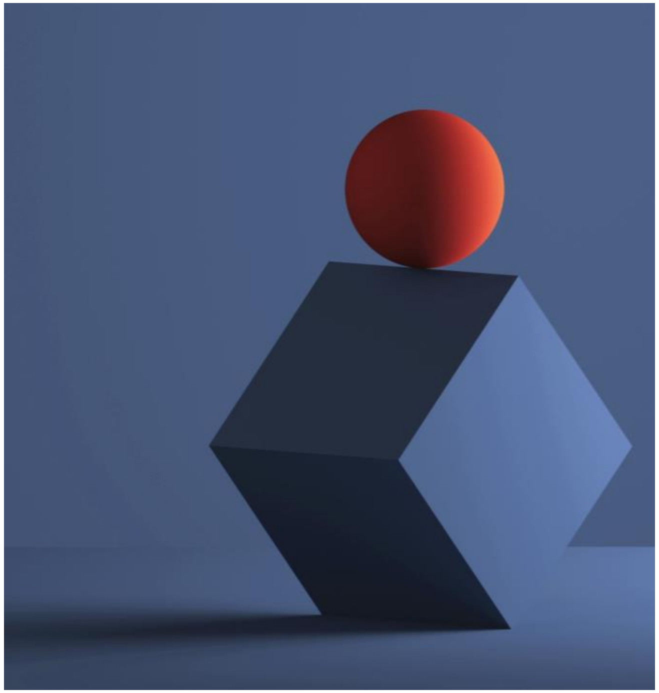
- Write a factorial function that produces 1 when its argument is 0, 0 for a negative argument, with an error message

Solution L7.2

```
> exception Negative of int;
> fun fact1(0) = 1
        | fact1(n) =
            if n>0 then n*fact1(n-1)
            else raise Negative(n);
val fact1 = fn: int -> int
> fun fact(n) = fact1(n) handle Negative(n) => (
        print("Warning: negative argument ");
        print(Int.toString(n));
        print(" found\n");
        0
    );
val fact = fn: int -> int
```

Solution L7.2

```
> fact 5;
val it = 120: int

> fact 0;
val it = 1: int

> fact ~2;
Warning: negative argument ~2 found
val it = 0: int
```



Polymorphic functions

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

This Photo by Unknown Author is licensed under CC BY-SA

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 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
```

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
```

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
```

Exercise L7.3

- Let `rev1` and `rev2` be as above. What are the results of the following calls
  - `rev1([(rev1:int list->int list), rev1])`
  - `rev2([(rev1:int list->int list), rev1])`
  - `rev1([rev1,rev1])`

Solution L7.3

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

> rev2([(rev1:int list->int list), rev1]);
val it = [fn, fn]: (int list -> int list) list

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

Exercise L7.4

- Let `rev1` and `rev2` be as above. What are the results of the following calls?
  - `rev1([chr,chr])`
  - `rev2([chr,chr])`
  - `rev1([chr,ord])`
  - `rev2([chr,ord])`

Solution exercise L7.4

```
> rev1([chr,chr]);
poly: : error: Type error in function application.
Function: rev1 : ’’a list -> ’’a list
Argument: ([chr, chr]) : (int -> char) list
Reason: Can’t unify ’’a to int -> char (Requires equality type)

> rev2([chr,chr]);
val it = [fn, fn]: (int -> char) list

> rev1([chr,ord]);
poly: : error: Elements in a list have different types.
Item 1: chr : int -> char
Item 2: ord : char -> int

> rev2([chr,ord]);
poly: : error: Elements in a list have different types.
Item 1: chr : int -> char
Item 2: ord : char -> int
```

Exercise L7.5

- Give definitions of `f(x,y,z)` where the argument has the following types
  - `’a * ’’b * (’a->’’b)`
  - `’a * ’a * int`

Solution exercise L7.5

- ’a * ’’b * (’a->“b)
  - > fun f(x,y,z)=(z(x)=y);

    val f = fn: ’a * ’’b * (’a -> ’’b) -> bool
- ’a * ’a * int
  - > fun f(x,y,z)=([x,y],z+1);

    val f = fn: ’a * ’a * int -> ’a list * int

Exercise L7.6

- Give definitions of `f(x,y,z)` where the argument has the following types
  - `’a list * ’b * ’a`
  - `(’a list * ’b list) * ’a * ’b`

Solution exercise L7.6

- ’a list * ’b * ’a

  -
    ```
    > fun f(x,y,z)=(y,z::x);
    val f = fn: ’a list * ’b * ’a -> ’b * ’a list
    ```
- (’a list * ’b list) * ’a * ‘b

  -
    ```
    fun f (x,y,z) = let
                        val (a,b) =x
                    in
                        (y::a,z::b)
                    end;
    val f = fn: (’a list * ’b list) * ’a * ’b -> ’a list * ’b
    list
    ```

  -
    ```
    > fun f((nil,nil),y,z)=(y,z)
      |f((xy::xys,xz::xzs),y,z)=(xy,xz)
      |f(_,y,z) =(y,z);
    val f = fn: (’a list * ’b list) * ’a * ’b -> ’a * ’b
    ```

Exercise L7.7

- Are the following equality types?
  - `int * string list`
  - `(int -> char) * string`
  - `int -> string -> unit`
  - `real * (string * string) list`

Solution exercise L7.7

- Are the following equality types?
  - `int * string list`

    Yes
  - `(int -> char) * string`

    No
  - `int -> string -> unit`

    No
  - `real * (string * string) list`

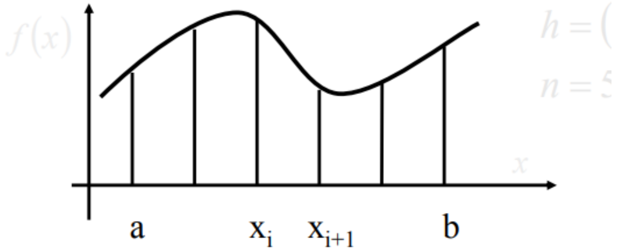
    No

Higher-order functions

Higher-order functions

- Functions that take functions as arguments
- Example: Approximate numerical integration $\int_a^b f(x)dx$
  - Divide the interval from a to b into n equal parts
  - Sum the areas of the n trapezoids

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \cdot \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2} = \sum_{i=1}^{n} \frac{b-a}{n} \cdot \frac{f(x_{i-1}) + f(x_i)}{2}$$



- We define a function `trap(a,b,n,F)` to do this, where the function `F` to be integrated is one of the parameters

Integration

$$\boxed{\sum_{i=1}^{n} \frac{b-a}{n} \cdot \frac{f(x_{i-1}) + f(x_i)}{2}}$$

```
> fun trap (a,b,n,F) =
    if n<=0 orelse b-a<=0.0 then 0.0
    else let
        val delta = (b-a)/real(n)
    in
        delta * (F(a)+F(a+delta))/2.0 + trap (a+delta,b,n-
        1,F)
    end;
val trap = fn: real * real * int * (real -> real) -> real
```

Example

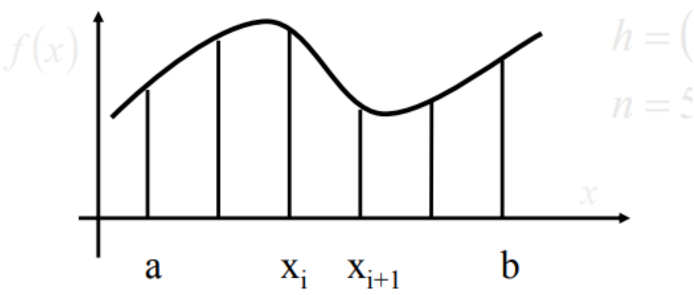
```
> fun square(x:real) = x*x;
val square = fn: real -> real

> trap (0.0,1.0,8,square);
val it = 0.3359375: real
```

An alternative implementation

- Recall the trapezoidal function for computing the integral

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \cdot \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2} = \sum_{i=1}^{n} \frac{b-a}{n} \cdot \frac{f(x_{i-1}) + f(x_i)}{2}$$

```
> fun trap (a,b,n,F) =
    if n<=0 orelse b-a<=0.0 then 0.0
    else let
           val delta = (b-a)/real(n)
    in
           delta * (F(a)+F(a+delta))/2.0 +
                  trap (a+delta,b,n-1,F)
    end;
```



- At the cost of some roundoff error it is possible to compute δ only once at the beginning (without recomputing it recursively at each recursive call). Reimplement `trap`.

An alternative implementation

```
> fun trap(a,b,n,F) =
    if n<=0 orelse b-a<=0.0 then 0.0
    else
        let
            val delta = (b-a)/real(n);
            fun trap1(x,0) = 0.0
                | trap1(x,i) = delta*(F(x)+F(x+delta))/2.0
                + trap1(x+delta,i-1)
        in
            trap1(a,n)
        end;
val trap = fn: real * real * int * (real -> real) -> real

> trap (0.0,1.0,8,square);
val it = 0.3359375: real
```

Exercise L7.8

- Write a function `tabulate` that takes an initial value a, an increment δ, a number of points n, and a function F from reals to reals and print a table with columns corresponding to x and $F(x)$, where $x = a, a + \delta, a + 2\delta, \ldots, a + (n - 1)\delta$

Solution exercise L7.8

```
> fun tabulate(x,delta,0,F) = ()
    | tabulate(x,delta,n,F) = (
        print(Real.toString(x));
        print("\t");
        print(Real.toString(F(x)));
        print("\n");
        tabulate(x+delta,delta,n-1,F)
    );
val tabulate = fn: real * real * int * (real -> real) -> unit
```

Solution exercise L7.8

```
> tabulate (1.0,0.1,9,fn x => x*x);
1.01.0
1.11.21
1.21.44
1.31.69
1.41.96
1.52.25
1.62.56
1.72.89
1.83.24
val it = (): unit
```

Summary

- Polymorphic functions
- Higher-order functions

Next time

- Logic programming – part II