λ

ML

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Suspended lectures

- Thursday April 11 (Provette)
- Tuesday April 16 (ICT days)

Today

- Recap
- Input and output
- Exceptions
- Polymorphic functions

Recap

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;
```

Semicolons at the end of the let expressions are optional.
However, usually, semicolons are used for all but the last declaration

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`: decomposing the result of a function

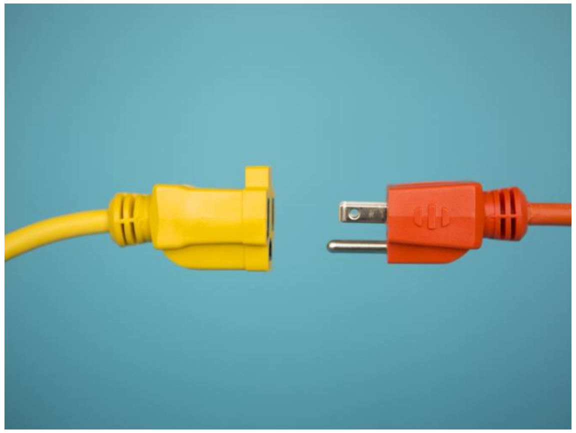
- Suppose f returns tuples of size 3
- We can decompose the result into components by writing

```
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
```



Input and output

Output

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));
aval 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 arepart 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

Exercise L6.1

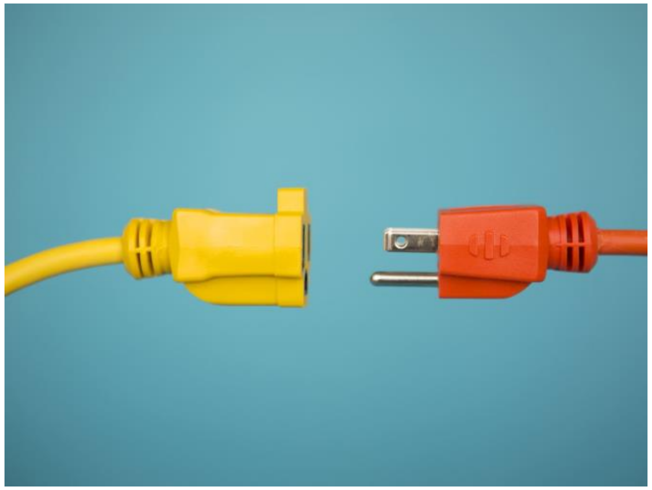
- Write a function that prints a list of integers

Exercise L6.2

- Write a function to compute $\binom{n}{m}$, while printing n, m and the result.
- Recall that $\binom{n}{m} = \frac{n!}{m!(n-m)!}$

Exercise L6.3

- Given n, print 2^n `X`'s



Input

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`

```
> 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
```

Exercise L6.4

- Write expressions to
  1. Open a file “zap” for reading
  2. Read 5 characters from the instream in1
  3. Read a line of text from the instream in1
  4. Find the first character waiting on the in1, without consuming it
  5. Read the entire file from instream in1
  6. Close the file whose instream is in1

Exercise L6.5

- Assume that we have a file with the following contents

```
abc

de

f
```

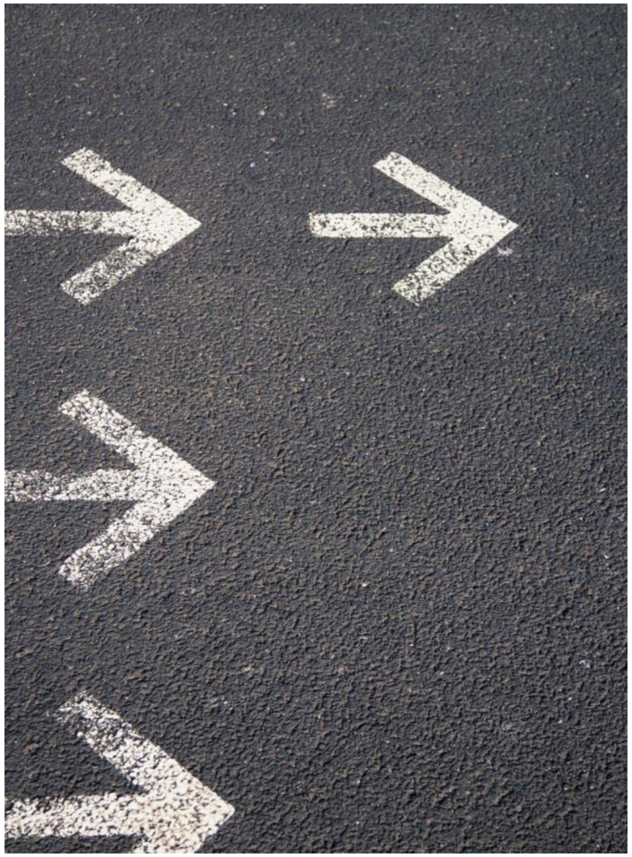
- What does each command return, if issued repeatedly

```
1. val x = TextIO.input(infile);

2. val x = TextIO.input1 (infile);

3. val x = TextIO.inputN (infile,2);

4. val x = TextIO.inputN (infile,5);

5. val x = TextIO.inputLine (infile);

6. val x = TextIO.lookahead (infile);
```

Exercise L6.6

- Write a function `getList(filename)` that reads a file, extracts the words (without space characters), and transforms the file in a list of words (without space characters).

- Hint: first write a function `getWord(in)` that extracts a word (without spaces) from a `TextIO.instream in` and then put them in a list. You can use support functions



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 L6.7

- 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`.

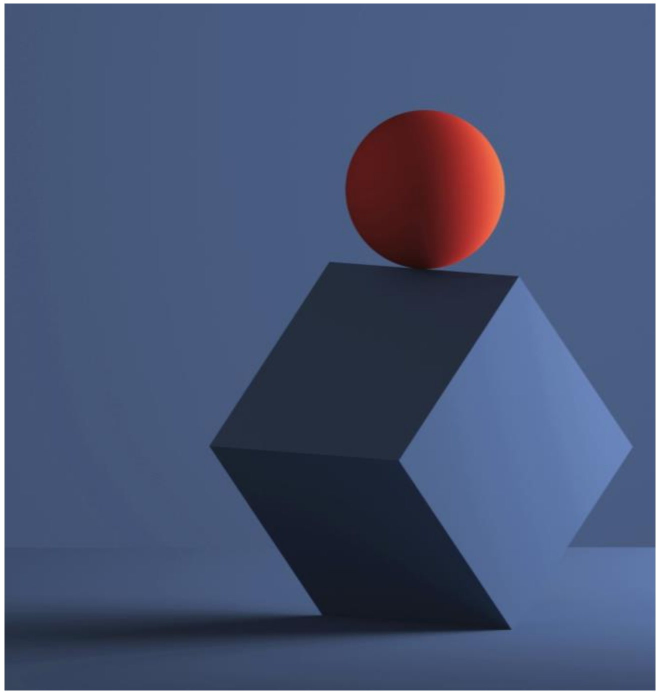
Solution L6.7

- 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);
```

Solution L6.8

```
> 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
```



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

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 `~`
- Division-related operators: `/`, `div` and `mod`
- Inequality comparison operators
- 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

Equality types

- Types that allow the use of equality tests (= and <>)
- Integers, booleans, characters, but not reals
- Tuples or lists of equality types
- Examples

```
> val x = (1,2) ;
val x = (1, 2): int * int
> val y = (2,3);
val y = (2, 3): int * int
> x=y;
val it = false: bool
> x=(1,2);
val it = true: bool
```

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)
```

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
```

Summary

- Input and output
- Exceptions
- Polymorphic functions

Next time

- Higher order functions